GUÍA DE INICIACIÓN AL LENGUAJE JAVA

VERSIÓN 2.0 Noviembre de 1999

SUBVENCIONADO POR:

Junta de Castilla y León

REALIZADO EN:

UNIVERSIDAD DE BURGOS
BVRGENSIS VNIVERSITAS

RESUMEN

Este trabajo corresponde a una guía (que será presentada en formato HTML) que sirva de iniciación a la programación en el lenguaje Java. En él se tratan los diversos aspectos básicos que comprenden el aprendizaje de un lenguaje de programación, así como una breve noción de la Programación Orientada al Objeto en la que Java se basa.

Así mismo se incluyen comparativas con otros lenguajes de programación (especial hincapié en C++), y se tratan con un poco más de profundidad bibliotecas gráficas como AWT y Swing.

Se comentan también aspectos relacionados con Internet, como son las bibliotecas de Sockets y las famosas applets Java.

PALABRAS CLAVE

Java, Tutorial, Lenguaje de programación, Programación Orientada a Objetos, Applet, Código de Byte, Clase, Objeto, Internet, Socket, Sun, JDK, Tutorial, Código fuente, C++.

ÍNDICES

ÍNDICE DE CONTENIDOS

ÍNDICE DE IMÁGENES

ÍNDICE DE TABLAS

AUTORES

DIRECTOR DEL PROYECTO

Miguel Ángel Manzanedo del Campo

Área de Organización de Empresas

Dpto. de Ingeniería Civil

Universidad de Burgos

COORDINADOR TÉCNICO

Francisco José García Peñalvo

Área de Ciencias de la Computación e Inteligencia Artificial

Dpto. de Informática y Automática

Universidad de Salamanca

REDACTADO, EDITADO, AMPLIADO Y PUBLICADO POR:

Ignacio Cruzado Nuño

Becario del Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Civil

Universidad de Burgos

OTROS INVESTIGADORES

César Ignacio García Osorio

Jesús Manuel Maudes Raedo

Carlos Pardo Aguilar

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Civil

Universidad de Burgos

Juan José Rodríguez Díez

Área de Lenguajes y Sistemas Informáticos

Departamento de Informática

Universidad de Valladolid

ALUMNOS COLABORADORES EN LA VERSIÓN 1.0

Alumnos de la asignatura de "Programación Avanzada", asignatura optativa del 3º Curso de la Titulación de "Ingeniería Técnica en Informática de Gestión" en la Universidad de Burgos. Alumnos del curso 1997-1998 (Primera promoción).

Rubén Cobos Pomares	Ignacio Cruzado Nuño	Fernando Delgado Peña
Raquel Díez Alcalde	Alberto Díez Cremer	Ana Berta García Benito
Jorge García del Egido	Esteban José García Zamora	Alberto Gómez Revilla
Ismael González Domingue	Mª Begoña Martín Ortega	Javier Portugal Alonso
Pablo Santos Luances	David Suárez Villasante	

Dirección electrónica del proyecto: qjava@ubu.es

PRÓLOGO

PRÓLOGO DE LA PRIMERA VERSIÓN DE LA GUÍA DE JAVA

Java es actualmente uno de esos términos mágicos que revolucionan las tecnologías de la información cada cierto tiempo. Java es un lenguaje de programación orientado a objetos creado por la compañía Sun Microsystems, que desde su aparición en 1995 ha provocado una autentica conmoción en los entornos informáticos. El éxito del lenguaje Java viene de la mano de la filosofía y la forma de operación de las aplicaciones escritas en Java, todo ello estrechamente ligado a Internet y al WWW.

El hecho de que el lenguaje Java sea un lenguaje joven en evolución no le ha permitido entrar a formar parte habitualmente de los currículum universitarios, poco dados a admitir innovaciones con tanta celeridad. Sin embargo, Java comienza a entrar en las Universidades Españolas, especialmente de la mano de los proyectos de final de carrera en las titulaciones de informática.

Aprovechando la convocatoria de 1998 de la Consejería de Educación y Cultura de la Junta de Castilla y León para la concesión de ayudas para la elaboración de Recursos de Apoyo a la Enseñanza Universitaria en esta Comunidad Autonómica, se decidió realizar una actividad que tuviera como protagonista al lenguaje Java, a la vez que se exploraba una nueva forma de involucrar de una manera más activa al alumnado en las actividades docentes, así como de incentivar al profesorado en aras de conseguir una docencia de mayor calidad.

La actividad que se propuso llevar a cabo fue una Guía Hipermedia para el Aprendizaje del Lenguaje Java.

Con la realización de esta guía se perseguían una serie de objetivos tanto docentes como pragmáticos. Los objetivos docentes estaban centrados en la búsqueda de la mejora de la calidad docente, reflejada en una mayor participación de los alumnos y en una mejora de la relación profesor-alumno. Los objetivos pragmáticos se centraban en el acercamiento del lenguaje Java al curriculum de los alumnos matriculados en la asignatura “Programación Avanzada” correspondiente al tercer curso de la titulación “Ingeniería Técnica en Informática de Gestión en la Universidad de Burgos”.

A continuación se recogen tanto los objetivos docentes como los objetivos pragmáticos propuestos.

Objetivos Docentes:

- Establecer una nueva forma de participación activa de los alumnos en el desarrollo de su currículum universitario, en contraposición de la actitud pasiva que tradicionalmente asume el alumno en las clases.
- Dotar a alumnos y profesores de las responsabilidades que conlleva la realización de una actividad en grupo.
- Completar los contenidos del curriculum universitario del alumno con temas novedosos que no suelen tener cabida en el programa de las asignaturas.
- Aumentar la cantidad y la calidad de la relación profesor-alumno.

Objetivos Pragmáticos:

- Introducir el lenguaje Java dentro de la titulación Ingeniería Técnica en Informática de Gestión de la Universidad de Burgos.
- Elaborar una guía de referencia hipermedia del lenguaje Java accesible vía Internet y distribuíble en CD-ROM. Esta guía podrá ser utilizada tanto para el autoaprendizaje, como para material docente o como material de consulta en los proyectos de final de carrera.
- Aprender de la experiencia para repetir esta técnica en otras asignaturas de la titulación e incluso en otras Universidades.

Este proyecto se va a acometer en dos fases bien diferenciadas:

Una primera fase (*Enero 1998 – Diciembre 1998*) donde se planifica todo el proyecto y se obtiene una primera guía básica de referencia básica (*producto que viene representado por la presente guía*).

Una segunda fase (*Enero 1999 – Diciembre 1999*) donde se amplía la primera guía introductoria con los conceptos más avanzados del lenguaje y se enriquece de los comentarios recibidos sobre la primera guía.

Para la realización práctica de esta primera guía se ha utilizado exclusivamente HTML (Hyper Text Markup Language), debido a que sus características hipermedia lo hacen idóneo para el desarrollo de este tipo de productos.

La metodología empleada para la creación de esta guía ha consistido en formar grupos de trabajo formados por tres alumnos cada uno (*los alumnos que formaban cada uno de los grupos eran alumnos de la asignatura Programación Avanzada del tercer curso de la Ingeniería Técnica en Informática de Gestión de la Universidad de Burgos, que voluntariamente se ofrecieron para colaborar en esta iniciativa*). Cada grupo se encargaba de elaborar una serie de temas que en su conjunto dan forma a la presente guía, siendo su trabajo coordinado por alguno de los profesores colaboradores. Una vez que los temas estaban terminados eran revisados e integrados en la guía por el coordinador técnico del proyecto.

Burgos, 5 de Octubre de 1998

Miguel Ángel Manzanedo del Campo

Francisco José García Peñalvo

PRÓLOGO DE LA SEGUNDA VERSIÓN DE LA GUÍA DE JAVA

En los casi dos años en los que se ha venido desarrollando esta guía introductoria al lenguaje Java, hemos asistido al afianzamiento de Java como plataforma de desarrollo y a una constante evolución que, aunque todavía lejos de una madurez plena, ha abierto numerosos campos de aplicación para esta plataforma (acceso a bases de datos, interacción con CORBA, aplicaciones distribuidas...).

Durante este tiempo han proliferado multitud de referencias en torno al fenómeno Java (en forma de libros, artículos, tutoriales...), sin embargo, la guía que hemos realizado no pretende sustituir a ninguno de ellos, sino más bien completarlos presentando una forma sencilla y directa de introducirse al lenguaje Java y ofreciendo un acceso sencillo basado en la facilidad de navegación hipermedia que aporta HTML.

A parte de la guía realizada, como producto final resultado del proyecto financiado por la Consejería de Educación y Cultura de la Junta de Castilla y León en su convocatoria de ayudas de 1998, este proyecto ha aportado unas experiencias especialmente interesantes al verse satisfechos los objetivos docentes y pragmáticos que se buscaban al iniciar este trabajo, y que se indicaban en el prólogo a la primera versión de esta guía que aquí se presenta.

Si personalmente, tuviera que destacar una sola cosa de esta experiencia, no dudaría en destacar el valor humano logrado al potenciar la relación profesor-alumno (hoy en día ex-alumnos) y entre compañeros de diferentes Universidades de Castilla y León para lograr el fin extra académico propuesto.

También me gustaría destacar el hecho constatado de que la elaboración de esta guía a contribuido en gran manera a la introducción del lenguaje Java dentro de la titulación de Ingeniería Técnica en Informática de Gestión de la Universidad de Burgos, y está haciendo lo propio en la Ingeniería Técnica en Informática de Sistemas de la Universidad de Salamanca.

En el ámbito técnico destacar la revisión realizada de los contenidos de la primera versión de la guía, así como la ampliación en temas relacionados con la programación cliente/servidor en Internet, los entornos gráficos de usuario o la incorporación de métodos nativos entre otros interesantes temas, que dan un enorme valor a esta guía como fuente de referencia práctica.

No quisiera terminar este prólogo sin antes tener unas palabras de agradecimiento y recuerdo para todos aquéllos que participaron de una u otra manera en la elaboración de esta guía, especialmente para mis ex-compañeros del Área de Lenguajes y Sistemas Informáticos de la Universidad de Burgos, para todos mis antiguos alumnos que se dejaron “engañar” para iniciar esta aventura y para Ignacio Cruzado cuyo trabajo en estos últimos meses ha dotado de contenido y forma a la versión de la guía que hoy se hace realidad.

Salamanca, 13 de octubre de 1999

Francisco José García Peñalvo

PREFACIO

Cuando hace dos cursos el profesor Francisco José García Peñalvo nos propuso redactar el esbozo de un tutorial sobre Java, de carácter voluntario, y aunque los alumnos siempre somos reacios a una carga de trabajo sin compensación en la calificación, sorprendentemente, los alumnos aceptamos. La continua apuesta de "Fran" por su alumnado, y la confianza que en nosotros depositó, realmente nos estimuló a dar todo lo mejor de nosotros.

Aunque desgraciadamente aquel año no toda la clase pudiera superar la asignatura, sobre todo por la carga docente, pocas voces he oído en desacuerdo con las novedosas formas docentes que aquel año se experimentaron.

Aunque la redacción de la primera versión, hecha por diferentes grupos de trabajo, era muy vaga, heterogénea y estaba llena de erratas, ha sido el pilar de este tutorial, tanto como esquema de temas a tratar, como bibliografía a manejar.

En la redacción de este tutorial se ha pretendido abarcar el mayor número de aspectos posibles de la gran variedad de temas que rodean a Java. Espero no haber perdido nunca de vista que este tutorial debe de servir tanto a expertos informáticos como a programadores de cualquier tipo que, aun no teniendo idea de programación orientada a objetos, tengan interés en Java.

Así mismo espero que la bibliografía aportada sea suficiente para aquellos que hayáis quedado prendados de la potencia de este nuevo lenguaje de programación.

Debo decir que mientras desarrollaba este tutorial me ha impresionado la potencia de Java por tres motivos principales:

1. Sus amplias bibliotecas multiplataforma.

2. La posibilidad de desarrollar aplicaciones que funcionen en Internet mediante navegadores (*applets*).

3. Todo ello con un lenguaje orientado a objeto sencillo pero potente.

Esperemos que pronto los problemas de incompatibilidad entre Microsoft Explorer y Netscape Navigator se solucionen, con lo que se vivirá un futuro lleno de esplendor para Java.

CONTENIDOS

Este tutorial ha sido dividido en una serie de temas (cada uno de ellos compuesto de varios apartados) para una más fácil consulta y una mayor claridad en cuánto a qué se está intentando explicar.

A continuación se detalla de una forma general los contenidos de cada uno de los apartados.

En el primer tema *"I. Introducción"* se intenta acercar al lector el mundo Java desde una perspectiva global; su historia, sus características principales y una comparativa con otros lenguajes orientados a objeto, para que el lector pueda juzgar si le interesa aprender Java y para que vaya vislumbrando en qué puntos le va a interesar profundizar.

En un primer apartado se introducen todos los conceptos necesarios para poder entender un lenguaje de programación orientado a objeto, como es Java. Este apartado debe ser de especial interés para todos aquellos lectores que no hayan desarrollado jamás programas en un lenguaje orientado a objeto, y debe facilitar la comprensión de muchos conceptos básicos para un mejor entendimiento del resto del tutorial.

El segundo tema *"II. Lenguaje"* define todos los conceptos básicos y sintaxis del lenguaje Java; las estructuras que utiliza, su sintaxis, sus sistemas de control...

Los cuatro primeros apartados de este tema son fundamentales para comprender cualquier fragmento de código Java, y aunque sencillos son similares a los de cualquier otro lenguaje de programación.

Los temas cinco, seis y siete introducen los aspectos orientados a objeto del lenguaje Java, comenzando desde los más sencillos como son los objetos y las clases hasta los más complejos como pueden ser la herencia y el polimorfismo.

Entre los apartados ocho y once se comentan diversos aspectos propios del lenguaje Java, que dotan a los programas de una mayor potencia como son las excepciones, los *threads*, las interfaces y los paquetes.

Por último en el apartado doce se explican los fundamentos sobre la construcción de programas Java, aplicaciones que se pueden construir y diversos aspectos referentes a ellas.

El tercer tema *"III. Bibliotecas de la API de Java"* trata, de una forma global, todas las bibliotecas de la API de Java que servirán para dotar a los programas de una gran potencia con multitud de clases e interfaces estándar ya programadas y distribuidas por Sun.

Además, en un par de apartados se explican las gestiones de las cadenas y de los flujos de entrada/salida que hace Java. Estas acciones se realizan de forma sensiblemente diferente a la de otros lenguajes de programación, como por ejemplo C o C++.

En el cuarto tema *"IV. Bibliotecas gráficas"*, se explican las dos bibliotecas que Java incorpora para desarrollar interfaces gráficas de usuario: AWT y la nueva Swing.

En el quinto tema *"V. Java e Internet"*, se explica la potencia de Java para el desarrollo de aplicaciones en red, así como el paquete *java.net,* el cual da soporte a un montón de operaciones para el diálogo de diversas aplicaciones de red con aspectos como los *sockets*.

En el sexto tema *"VI. Applets"* se explica el fundamento de este tipo de aplicaciones especialmente diseñadas para su uso en Internet. Son características de Java y le aportan una potencia inusitada para el desarrollo de aplicaciones para Internet. Además en este tema se acompañan una serie de ejemplos para intentar aclarar el desarrollo y funcionamiento de este tipo de aplicaciones.

Por último al tutorial se le adjuntan una serie de apéndices que sirvan como breves reseñas sobre diferentes aspectos de Java que por una u otra cuestión se ha decidido que vayan aparte de lo que es el bloque del lenguaje.

Así en el primer apéndice *"Apéndice I: JDK"*, se explica en detalle el funcionamiento del JDK, herramienta que distribuye Sun para el desarrollo de aplicaciones Java.

En el segundo apéndice *"Apéndice II: Herramientas de desarrollo"*, se comentan las diferentes herramientas disponibles en el mercado para desarrollar aplicaciones Java, depurarlas y probarlas.

En un tercer apéndice *"Apéndice III: Métodos Nativos"* se explica mediante un ejemplo de cómo Java es capaz de utilizar código de aplicaciones escritas en otros lenguajes de programación.

En el cuarto apéndice *"Apéndice IV: Guía de referencia de Java a C++"* se explican las similitudes y diferencias entre estos dos lenguajes de programación, dado que Java es un lenguaje derivado de C++, pero con notables diferencias respecto a su predecesor. Este apéndice puede ser fundamental para aquellos programadores que proviniendo de C++ quieran conocer el lenguaje Java, ya que su similitud sintáctica es, en muchos casos, engañosa.

En el último apéndice *"Apéndice V: Guía de referencia del lenguaje"*, se explican, a modo de resumen, las características fundamentales del lenguaje Java

AGRADECIMIENTOS

Me gustaría aprovechar este momento para felicitar a Francisco José García Peñalvo por el esfuerzo que hace por una docencia más moderna y atractiva, así como la confianza que deposita en su alumnado, y por el agradable clima de trabajo que ha creado para la realización de este tutorial.

Así mismo me gustaría agradecer al Área de Lenguajes y Sistemas Informáticos de la Universidad de Burgos la confianza depositada en mi así como toda la documentación bibliográfica que me han facilitado.

Por último me gustaría agradecer muy especialmente a Amelia Pajares Rojo su colaboración espontánea en este proyecto, consiguiendo así un tutorial mucho más claro y legible.

Espero que este tutorial realmente os agrade a todos.

Ignacio Cruzado Nuño

Burgos, Septiembre de 1999

TEMA I: INTRODUCCIÓN

I.1. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

A. PROGRAMACIÓN ORIENTADA A OBJETOS

La orientación a objetos es un paradigma de programación que facilita la creación de software de calidad por sus factores que potencian el mantenimiento, la extensión y la reutilización del software generado bajo este paradigma.

La programación orientada a objetos trata de amoldarse al modo de pensar del hombre y no al de la máquina. Esto es posible gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado objeto. Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado y las operaciones que pueden modificar dicho estado, y determinan las capacidades del objeto.

Java incorpora el uso de la orientación a objetos como uno de los pilares básicos de su lenguaje.

B. LOS OBJETOS

Podemos definir objeto como el "encapsulamiento de un conjunto de operaciones (métodos) que pueden ser invocados externamente, y de un estado que recuerda el efecto de los servicios". **[Piattini et al., 1996]**.

Un objeto además de un estado interno, presenta una interfaz para poder interactuar con el exterior. Es por esto por lo que se dice que en la programación orientada a objetos "se unen datos y procesos", y no como en su predecesora, la programación estructurada, en la que estaban separados en forma de variables y funciones.

Un objeto consta de:

- Tiempo de vida: La duración de un objeto en un programa siempre está limitada en el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado *instanciación*, y cuando dejan de existir se dice que son *destruidos.*
- Estado: Todo objeto posee un estado, definido por sus *atributos*. Con él se definen las propiedades del objeto, y el estado en que se encuentra en un momento determinado de su existencia.
- Comportamiento: Todo objeto ha de presentar una interfaz, definida por sus *métodos*, para que el resto de objetos que componen los programas puedan interactuar con él.

El equivalente de un *objeto* en el paradigma estructurado sería una *variable*. Así mismo la *instanciación de objetos* equivaldría a la *declaración de variables*, y el *tiempo de vida de un objeto* al *ámbito de una variable*.

C. LAS CLASES

Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común.

Podemos definir una clase como "un conjunto de cosas (físicas o abstractas) que tienen el mismo comportamiento y características... Es la implementación de un tipo de objeto (considerando los objetos como instancias de las clases)". **[Piattini et al., 1996]**.

Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (*instanciación*) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.

Las clases presentan el estado de los objetos a los que representan mediante variables denominadas *atributos*. Cuando se instancia un objeto el compilador crea en la memoria dinámica un espacio para tantas variables como atributos tenga la clase a la que pertenece el objeto.

Los *métodos* son las funciones mediante las que las clases representan el comportamiento de los objetos. En dichos métodos se modifican los valores de los atributos del objeto, y representan las capacidades del objeto (en muchos textos se les denomina *servicios*).

Desde el punto de vista de la programación estructurada, una clase se asemejaría a un módulo, los atributos a las variables globales de dicho módulo, y los métodos a las funciones del módulo.

D. MODELO DE OBJETOS

Existen una serie de principios fundamentales para comprender cómo se modeliza la realidad al crear un programa bajo el paradigma de la orientación a objetos. Estos principios son: la abstracción, el encapsulamiento, la modularidad, la jerarquía, el paso de mensajes y el poliforfismo.

a.) Principio de Abstracción

Mediante la abstracción la mente humana modeliza la realidad en forma de objetos. Para ello busca parecidos entre la realidad y la posible implementación de *objetos del programa* que simulen el funcionamiento de los *objetos reales*.

Los seres humanos no pensamos en las cosas como un conjunto de cosas menores; por ejemplo, no vemos un cuerpo humano como un conjunto de células. Los humanos entendemos la realidad como objetos con comportamientos bien definidos. No necesitamos conocer los detalles de porqué ni cómo funcionan las cosas; simplemente solicitamos determinadas acciones en espera de una respuesta; cuando una persona desea desplazarse, su cuerpo le responde comenzando a caminar.

Pero la abstracción humana se gestiona de una manera jerárquica, dividiendo sucesivamente sistemas complejos en conjuntos de subsistemas, para así entender más fácilmente la realidad. Esta es la forma de pensar que la orientación a objeto intenta cubrir.

b.) Principio de Encapsulamiento

El encapsulamiento permite a los objetos elegir qué información es publicada y qué información es ocultada al resto de los objetos. Para ello los objetos suelen presentar sus

métodos como interfaces públicas y sus atributos como datos privados e inaccesibles desde otros objetos.

Para permitir que otros objetos consulten o modifiquen los atributos de los objetos, las clases suelen presentar métodos de acceso. De esta manera el acceso a los datos de los objetos es controlado por el programador, evitando efectos laterales no deseados.

Con el encapsulado de los datos se consigue que las personas que utilicen un objeto sólo tengan que comprender su interfaz, olvidándose de cómo está implementada, y en definitiva, reduciendo la complejidad de utilización.

c.) Principio de Modularidad

Mediante la modularidad, se propone al programador dividir su aplicación en varios módulos diferentes (ya sea en forma de clases, paquetes o bibliotecas), cada uno de ellos con un sentido propio.

Esta fragmentación disminuye el grado de dificultad del problema al que da respuesta el programa, pues se afronta el problema como un conjunto de problemas de menor dificultad, además de facilitar la comprensión del programa.

d.) Principio de Jerarquía

La mayoría de nosotros ve de manera natural nuestro mundo como objetos que se relacionan entre sí de una manera jerárquica. Por ejemplo, un perro es un mamífero, y los mamíferos son animales, y los animales seres vivos...

Del mismo modo, las distintas clases de un programa se organizan mediante la *jerarquía*. La representación de dicha organización da lugar a los denominados *árboles de herencia*:

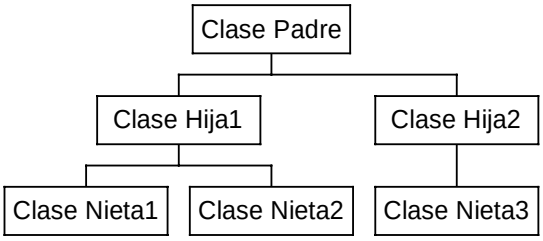

Imagen 1: Ejemplo de árbol de herencia

Mediante la *herencia* una *clase hija* puede tomar determinadas propiedades de una *clase padre*. Así se simplifican los diseños y se evita la duplicación de código al no tener que volver a codificar métodos ya implementados.

Al acto de tomar propiedades de una clase padre se denomina *heredar*.

e.) Principio del Paso de Mensajes

Mediante el denominado *paso de mensajes*, un objeto puede solicitar de otro objeto que realice una acción determinada o que modifique su estado. El paso de mensajes se suele implementar como llamadas a los métodos de otros objetos.

Desde el punto de vista de la programación estructurada, esto correspondería con la llamada a funciones.

f.) Principio de Polimorfismo

Polimorfismo quiere decir "un objeto y muchas formas". Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre. Por ejemplo un método puede presentar diferentes implementaciones en función de los argumentos que recibe, recibir diferentes números de parámetros para realizar una misma operación, y realizar diferentes acciones dependiendo del nivel de abstracción en que sea llamado.

E. RELACIONES ENTRE OBJETOS

Durante la ejecución de un programa, los diversos objetos que lo componen han de interactuar entre sí para lograr una serie de objetivos comunes.

Existen varios tipos de relaciones que pueden unir a los diferentes objetos, pero entre ellas destacan las relaciones de: asociación, todo/parte, y generalización/especialización.

a.) Relaciones de Asociación

Serían relaciones generales, en las que un objeto realiza llamadas a los servicios (métodos) de otro, interactuando de esta forma con él.

Representan las relaciones con menos riqueza semántica.

b.) Relaciones de Todo/Parte

Muchas veces una determinada entidad existe como conjunción de otras entidades, como un conglomerado de ellas. La orientación al objeto recoge este tipo de relaciones como dos conceptos; la agregación y la composición.

En este tipo de relaciones un *objeto componente* se integra en un *objeto compuesto*. La diferencia entre agregación y composición es que mientras que la composición se entiende que dura durante toda la vida del objeto componedor, en la agregación no tiene por qué ser así.

Esto se puede implementar como un objeto (*objeto compuesto*) que cuenta entre sus atributos con otro objeto distinto (*objeto componente*).

c.) Relaciones de Generalización/Especialización

A veces sucede que dos clases tiene muchas de sus partes en común, lo que normalmente se abstrae en la creación de una tercera clase (*padre* de las dos) que reúne todas sus características comunes.

El ejemplo más extendido de este tipo de relaciones es la herencia, propiedad por la que una clase (*clase hija*) recoge aquellos métodos y atributos que una segunda clase (*clase padre*) ha especificado como "heredables".

Este tipo de relaciones es característico de la programación orientada a objetos.

En realidad, la generalización y la especialización son diferentes perspectivas del mismo concepto, la generalización es una perspectiva ascendente (*bottom-up*), mientras que la especialización es una perspectiva descendente (*top-down*).

Para más información sobre el modelo de objetos en la programación avanzada, y las relaciones entre objetos véase **[García, 1998]** o para una información más detallada consulte **[Booch, 1996]**.

I.2. HISTORIA DE JAVA

A. ¿POR QUÉ SE DISEÑÓ JAVA?

Los lenguajes de programación C y Fortran se han utilizado para diseñar algunos de los sistemas más complejos en lenguajes de programación estructurada, creciendo hasta formar complicados procedimientos. De ahí provienen términos como "código de espagueti" o "canguros" referentes a programas con múltiples saltos y un control de flujo difícilmente trazable.

No sólo se necesitaba un lenguaje de programación para tratar esta complejidad, sino un nuevo estilo de programación. Este cambio de paradigma de la programación estructurada a la programación orientada a objetos, comenzó hace 30 años con un lenguaje llamado Simula67.

El lenguaje C++ fue un intento de tomar estos principios y emplearlos dentro de las restricciones de C. Todos los compiladores de C++ eran capaces de compilar programas de C sin clases, es decir, un lenguaje capaz de interpretar dos estilos diferentes de programación. Esta compatibilidad (*"hacia atrás"*) que habitualmente se vende como una característica de C++ es precisamente su punto más débil. No es necesario utilizar un diseño orientado a objetos para programar en C++, razón por la que muchas veces las aplicaciones en este lenguaje no son realmente orientadas al objeto, perdiendo así los beneficios que este paradigma aporta.

Así Java utiliza convenciones casi idénticas para declaración de variables, paso de parámetros, y demás, pero sólo considera las partes de C++ que no estaban ya en C.

Las principales características que Java no hereda de C++ son:

- Punteros: Las direcciones de memoria son la característica más poderosa de C++. El inadecuado uso de los punteros provoca la mayoría de los errores de colisión de memoria, errores muy difíciles de detectar. Además, casi todos los virus que se han escrito aprovechan la capacidad de un programa para acceder a la memoria volátil (RAM) utilizando punteros. En Java, no existen punteros, evitando el acceso directo a la memoria volátil.
- Variables globales: Con ellas cualquier función puede producir efectos laterales, e incluso se pueden producir fallos catastróficos cuando algún otro método cambia el estado de la variable global necesaria para la realización de otros procesos. En Java lo único global es el nombre de las clases.
- *goto*: Manera rápida de arreglar un programa sin estructurar el código. Java no tiene ninguna sentencia *goto*. Sin embargo Java tiene las sentencias *break* y *continue* que cubren los casos importantes de *goto*.
- Asignación de memoria: La función *malloc* de C, asigna un número especificado de bytes de memoria devolviendo la dirección de ese bloque. La función *free* devuelve un bloque asignado al sistema para que lo utilice. Si se olvida de llamar a *free* para liberar un bloque de memoria, se están limitando los recursos del sistema, ralentizando progresivamente los programas. Si por el contrario se hace un *free* sobre un puntero ya liberado, puede ocurrir cualquier cosa. Más tarde C++ añadió *new* y *delete*, que se usan de forma similar, siendo todavía el programador, el responsable de liberar el espacio de memoria. Java no tiene funciones *malloc* ni *free*.

Se utiliza el operador *new* para asignar un espacio de memoria a un objeto en el *montículo* de memoria. Con *new* no se obtiene una dirección de memoria sino un descriptor al objeto del *montículo*. La memoria real asignada a ese objeto se puede mover a la vez que el programa se ejecuta, pero sin tener que preocuparse de ello. Cuando no tenga ninguna referencia de ningún objeto, la memoria ocupada estará disponible para que la reutilice el resto del sistema sin tener que llamar a *free* o *delete.* A esto se le llama *recogida de basura.* El recolector de basura se ejecuta siempre que el sistema esté libre, o cuando una asignación solicitada no encuentre asignación suficiente.

- Conversión de tipos insegura: Los moldeados de tipo (*type casting*) son un mecanismo poderoso de C y C++ que permite cambiar el tipo de un puntero. Esto requiere extremada precaución puesto que no hay nada previsto para detectar si la conversión es correcta en tiempo de ejecución. En Java se puede hacer una comprobación en tiempo de ejecución de la compatibilidad de tipos y emitir una excepción cuando falla.

B. COMIENZOS

Java fue diseñado en 1990 por James Gosling, de Sun Microsystems, como software para dispositivos electrónicos de consumo. Curiosamente, todo este lenguaje fue diseñado antes de que diese comienzo la era World Wide Web, puesto que fue diseñado para dispositivos electrónicos como calculadoras, microondas y la televisión interactiva.

Imagen 2: Logotipo de la empresa Sun Microsystems

En los primeros años de la década de los noventa, 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.

Inicialmente Java se llamó Oak (roble en inglés), aunque tuvo que cambiar de denominación, debido a que dicho nombre ya estaba registrado por otra empresa. Se dice este nombre se le puso debido a la existencia de tal árbol en los alrededores del lugar de trabajo de los promotores del lenguaje.

Tres de las principales razones que llevaron a crear Java son:

1. Creciente necesidad de interfaces mucho más cómodas e intuitivas que los sistemas de ventanas que proliferaban hasta el momento.
2. Fiabilidad del código y facilidad de desarrollo. Gosling observó que muchas de las características que ofrecían C o C++ aumentaban de forma alarmante el gran coste de pruebas y depuración. Por ello en los sus ratos libres creó un lenguaje de programación donde intentaba solucionar los fallos que encontraba en C++.

3. Enorme diversidad de controladores electrónicos. Los dispositivos electrónicos se controlan mediante la utilización de microprocesadores de bajo precio y reducidas prestaciones, que varían cada poco tiempo y que utilizan diversos conjuntos de instrucciones. Java permite escribir un código común para todos los dispositivos.

Por todo ello, en lugar de tratar únicamente de optimizar las técnicas de desarrollo y dar por sentada la utilización de C o C++, el equipo de Gosling se planteó que tal vez los lenguajes existentes eran demasiado complicados como para conseguir reducir de forma apreciable la complejidad de desarrollo asociada a ese campo. Por este motivo, su primera propuesta fue idear un nuevo lenguaje de programación lo más sencillo posible, con el objeto de que se pudiese adaptar con facilidad a cualquier entorno de ejecución.

Basándose en el conocimiento y estudio de gran cantidad de lenguajes, este grupo decidió recoger las características esenciales que debía tener un lenguaje de programación moderno y potente, pero eliminando todas aquellas funciones que no eran absolutamente imprescindibles.

Para más información véase **[Cuenca, 1997]**.

C. PRIMEROS PROYECTOS EN QUE SE APLICÓ JAVA

El proyecto Green fue el primero en el que se aplicó Java, y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Con este fin 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 ya *Duke*, la actual mascota de Java.

Imagen 3: Icono de Duke, la mascota de Java

Más tarde Java se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva que se pensaba iba a ser el principal campo de aplicación de Java. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo.

Una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, instaron a FirstPerson a desarrollar nuevas estrategias que produjeran beneficios. Entre ellas se encontraba la aplicación de Java a Internet, la cual no se consideró productiva en ese momento.

Para más información véase **[Froufe, 1997].**

D. RESURGIMIENTO DE JAVA

Aunque muchas de las fuentes consultadas señalan que Java no llegó a caer en un olvido, lo cierto es que tuvo que ser Bill Joy (cofundador de Sun y uno de los

desarrolladores principales del sistema operativo Unix de Berckley) el que sacó a Java del letargo en que estaba sumido. Joy juzgó que Internet podría llegar a ser el campo adecuado para disputar a Microsoft su primacía en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes.

Para poder presentarlo en sociedad se tuvo que modificar el nombre de este lenguaje de programación y se tuvo que realizar una serie de modificaciones de diseño para poderlo adaptar al propósito mencionado. Así Java fue presentado en sociedad en agosto de 1995.

Algunas de las razones que llevaron a Bill Joy a pensar que Java podría llegar a ser rentable son:

- Java es un lenguaje orientado a objetos: Esto es lo que facilita abordar la resolución de cualquier tipo de problema.
- Es un lenguaje sencillo, aunque sin duda potente.
- La ejecución del código Java es segura y fiable: Los programas no acceden directamente a la memoria del ordenador, siendo imposible que un programa escrito en Java pueda acceder a los recursos del ordenador sin que esta operación le sea permitida de forma explícita. De este modo, los datos del usuario quedan a salvo de la existencia de virus escritos en Java. La ejecución segura y controlada del código Java es una característica única, que no puede encontrarse en ninguna otra tecnología.
- Es totalmente multiplataforma: Es un lenguaje sencillo, por lo que el entorno necesario para su ejecución es de pequeño tamaño y puede adaptarse incluso al interior de un navegador.

Las consecuencias de la utilización de Java junto a la expansión universal de Internet todavía están comenzando a vislumbrarse.

Para más información véase **[Froufe, 1997]**.

E. FUTURO DE JAVA

Existen muchas críticas a Java debido a su lenta velocidad de ejecución, aproximadamente unas 20 veces más lento que un programa en lenguaje C. Sun está trabajando intensamente en crear versiones de Java con una velocidad mayor.

El problema fundamental de Java es que utiliza una representación intermedia denominada *código de byte* para solventar los problemas de portabilidad. Los *códigos de byte* posteriormente se tendrán que transformar en código máquina en cada máquina en que son utilizados, lo que ralentiza considerablemente el proceso de ejecución.

La solución que se deriva de esto parece bastante obvia: fabricar ordenadores capaces de comprender directamente los códigos de byte. Éstas serían unas máquinas que utilizaran Java como sistema operativo y que no requerirían en principio de disco duro porque obtendrían sus recursos de la red.

A los ordenadores que utilizan Java como sistema operativo se les llama Network Computer, WebPC o WebTop. La primera gran empresa que ha apostado por este tipo de máquinas ha sido Oracle, que en enero de 1996 presentó en Japón su primer NC (Network Computer), basado en un procesador RISC con 8 Megabytes de RAM. Tras

Oracle, han sido compañías del tamaño de Sun, Apple e IBM las que han anunciado desarrollos similares.

La principal empresa en el mundo del software, Microsoft, que en los comienzos de Java no estaba a favor de su utilización, ha licenciado Java, lo ha incluido en Internet Explorer (versión 3.0 y posteriores), y ha lanzado un entorno de desarrollo para Java, que se denomina Visual J++.

El único problema aparente es la seguridad para que Java se pueda utilizar para transacciones críticas. Sun va a apostar por firmas digitales, que serán clave en el desarrollo no sólo de Java, sino de Internet.

Para más información véase **[Framiñán, 1997]**.

F. ESPECULACIÓN SOBRE EL FUTURO DE JAVA

En opinión de los redactores de este tutorial, Java es una plataforma que le falta madurar, pero que a buen seguro lo va a hacer. La apuesta realizada por empresas con mucho peso específico ha sido tan grande que va a dar un impulso a Java que no le permitirá caer

Además, el parque de productos (entornos de desarrollo, bibliotecas, elementos de conectividad...) ya disponible en la actualidad es tan amplio que es improbable que se quede en nada.

Por otra parte, la relación simbiótica que tiene con Internet (y por derivación con las Intranets) es un punto a favor de Java de muy difícil refutación.

I.3. CARACTERÍSTICAS DE JAVA

A. INTRODUCCIÓN

No es arriesgado afirmar que Java supone un significativo avance en el mundo de los entornos software, y esto viene avalado por tres elementos claves que diferencian a este lenguaje desde un punto de vista tecnológico:

- Es un lenguaje de programación que ofrece la potencia del diseño orientado a objetos con una sintaxis fácilmente accesible y un entorno robusto y agradable.
- Proporciona un conjunto de clases potente y flexible.
- Pone al alcance de cualquiera la utilización de aplicaciones que se pueden incluir directamente en páginas Web (aplicaciones denominadas *applets*).

Java aporta a la Web una interactividad que se había buscado durante mucho tiempo entre usuario y aplicación.

A lo largo de este apartado se estudian en detalle las principales características de Java.

B. POTENTE

a.) Orientación a objetos

En este aspecto Java fue diseñado partiendo de cero, no siendo derivado de otro lenguaje anterior y no tiene compatibilidad con ninguno de ellos.

En Java el concepto de objeto resulta sencillo y fácil de ampliar. Además se conservan elementos “no objetos”, como números, caracteres y otros tipos de datos simples.

b.) Riqueza semántica

Pese a su simpleza se ha conseguido un considerable potencial, y aunque cada tarea se puede realizar de un número reducido de formas, se ha conseguido un gran potencial de expresión e innovación desde el punto de vista del programador.

c.) Robusto

Java verifica su código al mismo tiempo que lo escribe, y una vez más antes de ejecutarse, de manera que se consigue un alto margen de codificación sin errores. Se realiza un descubrimiento de la mayor parte de los errores durante el tiempo de compilación, ya que Java es estricto en cuanto a tipos y declaraciones, y así lo que es rigidez y falta de flexibilidad se convierte en eficacia. Respecto a la gestión de memoria, Java libera al programador del compromiso de tener que controlar especialmente la asignación que de ésta hace a sus necesidades específicas. Este lenguaje posee una gestión avanzada de memoria llamada gestión de basura, y un manejo de excepciones orientado a objetos integrados. Estos elementos realizarán muchas tareas antes tediosas a la vez que obligadas para el programador.

d.) Modelo de objeto rico

Existen varias clases que contienen las abstracciones básicas para facilitar a los programas una gran capacidad de representación. Para ello se contará con un conjunto de clases comunes que pueden crecer para admitir todas las necesidades del programador.

Además la biblioteca de clases de Java proporciona un conjunto único de protocolos de Internet.

El conjunto de clases más complicado de Java son sus paquetes gráficos AWT (*Abstract Window Toolkit*) y *Swing*. Estos paquetes implementan componentes de una interfaz de usuario gráfica básica común a todos los ordenadores personales modernos.

C. SIMPLE

a.) Fácil aprendizaje

El único requerimiento para aprender Java es tener una comprensión de los conceptos básicos de la programación orientada a objetos. Así se ha creado un lenguaje simple (aunque eficaz y expresivo) pudiendo mostrarse cualquier planteamiento por parte del programador sin que las interioridades del sistema subyacente sean desveladas.

Java es más complejo que un lenguaje simple, pero más sencillo que cualquier otro entorno de programación. El único obstáculo que se puede presentar es conseguir comprender la programación orientada a objetos, aspecto que, al ser independiente del lenguaje, se presenta como insalvable.

b.) Completado con utilidades

El paquete de utilidades de Java viene con un conjunto completo de estructuras de datos complejas y sus métodos asociados, que serán de inestimable ayuda para implementar *applets* y otras aplicaciones más complejas. Se dispone también de estructuras de datos habituales, como *pilas* y *tablas hash*, como clases ya implementadas.

Existirá una interfaz *Observer/Observable* que permitirá la implementación simple de objetos dinámicos cuyo estado se visualiza en pantalla.

El JDK (*Java Development Kit)* suministrado por Sun Microsystems incluye un compilador, un intérprete de aplicaciones, un depurador en línea de comandos, y un visualizador de *applets* entre otros elementos.

D. INTERACTIVO Y ORIENTADO A RED

a.) Interactivo y animado

Uno de los requisitos de Java desde sus inicios fue la posibilidad de crear programas en red interactivos, por lo que es capaz de hacer varias cosas a la vez sin perder rastro de lo que debería suceder y cuándo. Para se da soporte a la utilización de múltiples hilos de programación (*multithread*).

Las aplicaciones de Java permiten situar figuras animadas en las páginas Web, y éstas pueden concebirse con logotipos animados o con texto que se desplace por la pantalla.

También pueden tratarse gráficos generados por algún proceso. Estas animaciones pueden ser interactivas, permitiendo al usuario un control sobre su apariencia.

b.) Arquitectura neutral

Java está diseñado para que un programa escrito en este lenguaje sea ejecutado correctamente independientemente de la plataforma en la que se esté actuando (Macintosh, PC, UNIX…). Para conseguir esto utiliza una compilación en una representación intermedia que recibe el nombre de *códigos de byte*, que pueden interpretarse en cualquier sistema operativo con un intérprete de Java. La desventaja de un sistema de este tipo es el rendimiento; sin embargo, el hecho de que Java fuese diseñado para funcionar razonablemente bien en microprocesadores de escasa potencia, unido a la sencillez de traducción a código máquina hacen que Java supere esa desventaja sin problemas.

c.) Trabajo en red

Java anima las páginas Web y hace posible la incorporación de aplicaciones interactivas y especializadas. Aporta la posibilidad de distribuir contenidos ejecutables, de manera que los suministradores de información de la Web pueden crear una página de hipertexto (*página Web*) con una interacción continuada y compleja en tiempo real; el contenido ejecutable es transferido literalmente al ordenador del usuario.

Los protocolos básicos para trabajar en Internet están encapsulados en unas cuantas clases simples. Se incluyen implementaciones ampliables de los protocolos FTP, HTTP, NNTP y SMTP junto con conectores de red de bajo nivel e interfaces de nombrado. Esto le permite interactuar con esos servicios de red poderosos sin tener que comprender realmente los detalles de bajo nivel de esos protocolos. Este lenguaje está diseñado para cumplir los requisitos de entrega de contenidos interactivos mediante el uso de applets insertados en sus páginas HTML. Además, las clases de Java admiten muy bien estos protocolos y formatos. El envío de las clases de Java a través de Internet se realiza con gran facilidad, ya que existe una interfaz unificada, resolviendo así los típicos problemas de diferencia de versiones.

Java proporciona un conjunto de clases para tratar con una abstracción de los conectores de red (*sockets)* originales de la versión UNIX de Berckley, encapsular la noción de una dirección de Internet o conectar sockets con flujos de datos de Entrada/Salida.

Con todas estas posibilidades aumenta el dinamismo y competitividad de la Web, puesto que es capaz de captar el interés del usuario durante largo tiempo y permite a los programadores convertir la Web en un sistema de entrega de software.

d.) Applets

Una *applet* (miniaplicación) es un pequeño programa en Java transferido dinámicamente a través de Internet. Presentan un comportamiento inteligente, pudiendo reaccionar a la entrada de un usuario y cambiar de forma dinámica. Sin embargo, la verdadera novedad es el gran potencial que Java proporciona en este aspecto, haciendo posible que los programadores ejerzan un control sobre los programas ejecutables de Java que no es posible encontrar en otros lenguajes.

E. Y MUCHO MÁS

a.) Seguridad

Existe una preocupación lógica en Internet por el tema de la seguridad: virus, caballos de Troya, y programas similares navegan de forma usual por la red, constituyendo una amenaza palpable. Java ha sido diseñado poniendo un énfasis especial en el tema de la seguridad, y se ha conseguido lograr cierta inmunidad en el aspecto de que un programa realizado en Java no puede realizar llamadas a funciones globales ni acceder a recursos arbitrarios del sistema, por lo que el control sobre los programas ejecutables no es equiparable a otros lenguajes.

Los niveles de seguridad que presenta son:

- Fuertes restricciones al acceso a memoria, como son la eliminación de punteros aritméticos y de operadores ilegales de transmisión.
- Rutina de verificación de los *códigos de byte* que asegura que no se viole ninguna construcción del lenguaje.
- Verificación del nombre de clase y de restricciones de acceso durante la carga.
- Sistema de seguridad de la interfaz que refuerza las medidas de seguridad en muchos niveles.

En futuras versiones se prevé contar también con encriptación y técnicas similares.

b.) Lenguaje basado en C++

Java fue desarrollado basándose en C++, pero eliminando rasgos del mismo poco empleados, optándose por una codificación comprensible. Básicamente, encontramos las siguientes diferencias con C++:

- Java no soporta los tipos *struct, union* ni punteros.
- No soporta *typedef* ni *#define.*
- Se distingue por su forma de manejar ciertos operadores y no permite una sobrecarga de operadores.
- No soporta herencia múltiple.
- Java maneja argumentos en la línea de comandos de forma diversa a como lo hacen C o C++.
- Tiene una clase *String* que es parte del paquete *java.lang* y se diferencia de la matriz de caracteres terminada con un nulo que usan C y C++.
- Java cuenta con un sistema automático para asignar y liberar memoria, con lo que no es necesario utilizar las funciones previstas con este fin en C y C++.

c.) Gestión de la Entrada/Salida

En lugar de utilizar primitivas como las de C para trabajar con ficheros, se utlizan primitivas similares a las de C++, mucho más elegantes, que permiten tratar los ficheros, sockets, teclado y monitor como flujos de datos.

De este modo se pueden utilizar dichas primitivas para cualquier operación de Entrada/Salida.

d.) Diferentes tipos de aplicaciones

En Java podemos crear los siguientes tipos de aplicaciones:

- Aplicaciones: Se ejecutan sin necesidad de un navegador.
- *Applets*: Se pueden descargar de Internet y se observan en un navegador.
- *JavaBeans*: Componentes software Java, que se puedan incorporar gráficamente a otros componentes.
- *JavaScript*: Conjunto del lenguaje Java que puede codificarse directamente sobre cualquier documento HTML
- *Servlets*: Módulos que permiten sustituir o utilizar el lenguaje Java en lugar de programas CGI (Common Gateway Interface) a la hora de dotar de interactividad a las páginas Web.

I.4. COMPARATIVA CON OTROS LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETO

A. INTRODUCCIÓN

En este apartado se va a comparar Java con otros lenguajes de programación orientados a objeto.

En principio Java fue diseñado tomando C y C++ como base para la creación de un nuevo lenguaje con la modificación de todos aquellos aspectos que no eran útiles o dificultosos para la programación de componentes electrónicos de bajo coste. Para ello el nuevo lenguaje debía incluir interfaces cómodas, debía ser fiable y fácil de desarrollar y los programas debían ser portables de un sistema a otro sin ningún tipo de problema.

B. COMPARACIÓN DE LOS TIPOS DE DATOS

a.) Tipos de datos simples (primitivos)

Java es muy parecido a C++ en el juego básico de tipos de datos con algunas pequeñas modificaciones.

En Java se distingue entre tipos de datos primitivos y clases, aunque existen unas clases especiales (*envoltorios* o *wrappers*) que permiten modificar los tipos de datos primitivos.

Los tipos de datos primitivos (o simples) pueden ser numéricos, booleanos o caracteres.

b.) Datos numéricos

Hay cuatro tipos numéricos: *byte* de 1 byte, *short* de 2 bytes, *int* de 4 bytes, y los *long* de 8 bytes. El tipo más habitual de los cuatro es el tipo *int*. El *byte* viene a sustituir el tipo *char* de C++, ya que Java introduce una interpretación diferente al tipo de datos *char.*

Las principales diferencias con C++ son:

- No existe un tipo sin signo (*unsigned)* para los números en Java.
- Los tipos numéricos reales son el *float (8* bytes) y el *double* (16 bytes).
- Los números que utilizan coma flotante (por ejemplo 18.96) son considerados *double* por defecto, y habrá que realiza un moldeado (*casting*) explícito para que sean *float*.

c.) Caracteres

Los datos carácter en Java se basan en los de C++ que a su vez son heredados de C. Los caracteres son *Unicode* de 2 bytes. Los caracteres *Unicode* son valores de 2 bytes sin signo, con lo que se define obtiene un rango de 65535 caracteres diferentes, que son suficientes para las los diferentes lenguajes y sistemas de representación del planeta.

El carácter de datos del lenguaje Java proviene del tradicional C. Hay que señalar que los caracteres en C++ eran de sólo 1 byte, con lo que en Java podremos representar muchos más caracteres que en C++.

d.) Datos booleanos

En Java se definen para las variables con valores Verdadero/Falso o Sí/No, en definitiva, valores bi-estado. Una variable booleana puede tener los valores *true (*verdadero) o *false (*falso). Son parecidos a los de C++, aunque en cualquier caso, y a diferencia de C++ estas variables no pueden ser convertidas a datos numéricos, y es un tipo de datos básico.

C. OPERADORES RELACIONALES Y ARITMÉTICOS.

Se permite en Java los mismos operadores que C++, con la variación de >>> (desplazamiento sin signo) y la utilización del operador + para la concatenación de cadenas de caracteres.

D. VECTORES

Los vectores en Java, a diferencia de C++, son una clase de objetos. Un vector es un objeto real con una representación en tiempo real. Se pueden declarar y almacenar vectores de cualquier tipo, y almacenar también vectores de vectores para obtener matrices (vectores con varias dimensiones). En este último aspecto no existe diferencia con C++.

E. CADENAS

Las cadenas en Java son objetos del lenguaje, no existen seudo-arrays de caracteres (cadenas) como era el caso de C++. Existen dos tipos de cadenas de objetos:

Las que se obtienen de la clase *String*, para cadenas de sólo lectura.

Las que se obtienen de la clase *StringBuffer* para cadenas que se pueden modificar.

Al igual que C++, el compilador de Java entiende que una cadena de caracteres rodeada de dobles comillas es una cadena, y es iniciada como un objeto de tipo *String* (en C++ sería como vector de caracteres con el carácter fin de cadena '\0' al final de la misma).

F. OTRAS CARACTERÍSTICAS

a.) Introducción

En este apartado se va a comparar Java con los lenguajes C++ y Smalltalk (primer lenguaje que presentaba un modelo de objeto).

Característica	Java	Smalltalk	C++
Sencillez	Sí	Sí	No
Robustez	Sí	Sí	No
Seguridad	Sí	Algo	No
Interpretado	Sí	Sí	No
Dinamicidad	Sí	Sí	No
Portabilidad	Sí	Algo	No

Neutralidad	Sí	Algo	No
Threads	Sí	No	No
Garbage Colection	Sí	Sí	No
Excepciones	Sí	Sí	Algunas
Representación	Alta	Media	Alta

Tabla 1: Comparación entre Java, SmallTalk y C++

b.) Sencillez

Java tiene una sencillez que no posee C++ aunque sí Smalltalk. Esto es debido a que una de las razones de la creación de Java es la de obtener un lenguaje parecido a C++ pero reduciendo los errores más comunes de la programación, algo que se logra con mucho éxito puesto que Java reduce un 50% los errores que se comenten en C++ entre los que destacan:

- Eliminación de la aritmética de punteros y de las *referencias*.
- Desaparecen los registros (*struct*), heredados del paradigma estructurado.
- No se permite ni la definición de tipos (*typedef*) ni la de macros (*#define*).
- Ya no es necesario liberar memoria (*free o delete*).

De todas formas, lo que Java hace, en realidad, es la eliminación de palabras reservadas, y la utilización de un intérprete bastante pequeño.

c.) Robustez

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución, lo que hace que se detecten errores lo antes posible, normalmente en el ciclo de desarrollo. Algunas de estas verificaciones que hacen que Java sea un lenguaje robusto son:

- Verificación del *código de byte*.
- Gestión de excepciones y errores.
- Comprobación de punteros y de límites de vectores.

Se aprecia una clara diferencia con C++ quién no realiza ninguna de estas verificaciones.

d.) Seguridad

En Java no se permite los accesos ilegales a memoria, algo que sí se permitía en C++. Esto es algo muy importante puesto que este tipo de problema puede ocasionar la propagación de virus y otras clases de programas dañinos por la red.

El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de *código de byte* 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.

Algunos de los conocimientos que podemos obtener de los códigos de byte si pasan la verificación sin generar ningún mensaje de error son:

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

Por todo esto, y por no permitirlo mediante Java la utilización de métodos de un programa sin los privilegios del núcleo (*kernel*) del sistema operativo, la obligación de autentificación por clave pública para la realización de modificaciones, se considera Java un lenguaje seguro. Todo esto no lo incorporan ni C++ ni Smalltalk, por lo que Java es el único de los tres considerable como seguro.

e.) Lenguaje interpretado

Java es un lenguaje que puede ejecutar el código directamente, es decir es un "lenguaje interpretado". Esto es una característica que sí que posee Smalltalk, aunque no C++. No obstante, y aunque en teoría se consumen menos recursos siendo los lenguajes interpretados, el actual compilador que existe es bastante lento, unas 20 veces menos rápido que C++. Esto normalmente no es vital para la aplicación ni demasiado apreciable por el usuario, y además esta diferencia se está reduciendo con los nuevos compiladores JIT (*Just In Time).*

f.) Dinamicidad

Para la obtención de un mayor provecho de la tecnología orientada a objetos, Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Esta característica ya es contemplada por Smalltalk, aunque no C++, que enlaza todos los módulos cuando se compila.

g.) Portabilidad

Un programa Java puede ser ejecutado en diferentes entornos, algo imposible para C++.

h.) Neutralidad

Se dice que Java tiene una arquitectura neutra puesto que 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 (*JRE* o *Java Runtime Enviroment*) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.

Actualmente existen sistemas de ejecución (*JRE*) para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando el portado a otras plataformas.

No es así para C++ y para Smalltalk, donde el código generado podrá ejecutarse únicamente en la plataforma en la que se generó.

i.) Threads

Java permite múltiples hilos (*multithreading*) antes de su ejecución y en tiempo de ejecución. La posibilidad de construir pequeños procesos o piezas independientes de un gran proceso permite programar de una forma más sencilla y es una herramienta muy potente que no se ofrece en C++.

j.) Recolección automática de basura (Garbage colection)

Java modifica completamente la gestión de la memoria que se hace en C/C++. En C/C++ se utilizan punteros, reservas de memoria (con las ordenes *malloc*, *new*, *free*, *delete*...) y otra serie de elementos que dan lugar a graves errores en tiempo de ejecución difícilmente depurables.

Java tiene operadores nuevos para reservar memoria para los objetos, pero no existe ninguna función explícita para liberarla.

La recolección de basura (objetos ya inservibles) es una parte integral de Java durante la ejecución de sus programas. Una vez que se ha almacenado un objeto en el tiempo de ejecución, el sistema hace un seguimiento del estado del objeto, y en el momento en que se detecta que no se va a volver a utilizar ese objeto, el sistema vacía ese espacio de memoria para un uso futuro.

Esta gestión de la memoria dinámica hace que la programación en Java sea más fácil.

k.) Representación

Uno de los objetivos perseguidos en el desarrollo de Java era la obtención de programas con interfaces cómodas e intuitivas. Esto también se permite en C++, aunque con unos métodos más costosos, y en ningún caso con interfaces portables como los que Java crea.

Tanto en Java como en C++ se logran unas interfaces con una representación mejor que la que se puede alcanzar con Smalltalk.

TEMA II: FUNDAMENTOS DEL LENGUAJE

II.1. FUNDAMENTOS

A. INTRODUCCIÓN

Java es un lenguaje orientado a objetos, que se deriva en alto grado de C++, de tal forma que puede ser considerado como un C++ nuevo y modernizado o bien como un C++ al que se le han amputado elementos heredados del lenguaje estructurado C.

B. TOKENS

Un *token* es el elemento más pequeño de un programa que es significativo para el compilador. Estos *tokens* definen la estructura de Java.

Cuando se compila un programa Java, el compilador analiza el texto, reconoce y elimina los espacios en blanco y comentarios y extrae *tokens* individuales. Los *tokens* resultantes se compilan, traduciéndolos a código de byte Java, que es independiente del sistema e interpretable dentro de un entorno Java.

Los códigos de byte se ajustan al sistema de máquina virtual Java, que abstrae las diferencias entre procesadores a un procesador virtual único.

Los *tokens* Java pueden subdividirse en cinco categorías: Identificadores, palabras clave, constantes, operadores y separadores.

a.) Identificadores

Los identificadores son *tokens* que representan nombres asignables a variables, métodos y clases para identificarlos de forma única ante el compilador y darles nombres con sentido para el programador.

Todos los identificadores de Java diferencian entre mayúsculas y minúsculas (Java es *Case Sensitive* o *Sensible a mayúsculas*) y deben comenzar con una letra, un subrayado(_) o símbolo de dólar($). Los caracteres posteriores del identificador pueden incluir las cifras del 0 al 9. Como nombres de identificadores no se pueden usar palabras claves de Java.

Además de las restricciones mencionadas existen propuestas de estilo. Es una práctica estándar de Java denominar:

- Las clases: *Clase* o *MiClase*.
- Las interfaces: *Interfaz* o *MiInterfaz*.
- Los métodos: *metodo()* o *metodoLargo().*
- Las variables: *altura* o *alturaMedia*.
- Las constantes: *CONSTATE* o *CONSTANTE_LARGA*.
- Los paquetes: *java.paquete.subpaquete*.

Sin entrar en más detalle en la siguiente línea de código se puede apreciar la declaración de una variable entera (*int*) con su correspondiente identificador:

```
int alturaMedia;
```

b.) Palabras clave

Las palabras claves son aquellos identificadores reservados por Java para un objetivo determinado y se usan sólo de la forma limitada y específica. Java tiene un conjunto de palabras clave más rico que C o que C++, por lo que sí está aprendiendo Java con conocimientos de C o C++, asegúrese de que presta atención a las palabras clave de Java.

Las siguientes palabras son palabras reservadas de Java:

abstact	*boolean*	*break*	*byte*	*byvalue*
case	*cast*	*catch*	*char*	*class*
const	*continue*	*default*	*do*	*double*
else	*extends*	*false*	*final*	*finally*
float	*for*	*future*	*generic*	*goto*
if	*implements*	*import*	*inner*	*instanceof*
int	*interface*	*long*	*native*	*new*
null	*operator*	*outer*	*package*	*private*
protected	*public*	*rest*	*return*	*short*
static	*super*	*switch*	*syncroniced*	*this*
throw	*throws*	*transient*	*true*	*try*
var	*void*	*volatile*	*while*	

Tabla 2: Palabras reservadas Java

Las palabras subrayadas son palabras reservadas pero no se utilizan. La definición de estas palabras clave no se ha revelado, ni se tiene un calendario respecto a cuándo estará alguna de ellas en la especificación o en alguna de las implementaciones de Java.

c.) Literales y constantes

Los literales son sintaxis para asignar valores a las variables. Cada variables es de un tipo de datos concreto, y dichos tipos de datos tienen sus propios literales.

Mediante determinados modificadores (*static* y *final*) podremos crear variables *constantes*, que no modifican su valor durante la ejecución de un programa. Las constantes pueden ser numéricas, booleanas, caracteres (Unicode) o cadenas (*String*).

Las cadenas, que contienen múltiples caracteres, aún se consideran constantes, aunque están implementadas en Java como objetos.

Veamos un ejemplo de constante declarada por el usuario:

```
final static int ALTURA_MAXIMA = 200;
```

Se puede observar que utilizamos *final static*, para que la variable sea total y absolutamente invariable.

d.) Operadores

Conocidos también como operandos, indican una evaluación o computación para ser realizada en objetos o datos, y en definitiva sobre identificadores o constantes. Los operadores admitidos por Java son:

+	^	<=	++	%=
>>>=	-	~	>=	-
&=	.	*	&&	<<
==	<<=	[	/	||
>>	+=	^=	]	%

!	*>>>*	*=*	*!=*	*(*
&	*<*	**=*	*)*	*|*
>	*?!!*	*/=*	*>>*	

Tabla 3: Operadores Java

Así por ejemplo el siguiente fragmento de código incrementa el valor de una variable en dos unidades, mediante la utilización del operador aritmético + que se utiliza para la suma:

```
int miNumero=0;
miNumero = miNumero + 2;
```

En el apartado *"II.3 Operadores"* de este tutorial aprenderemos que en Java hay formas más sencillas de hacer esto mismo, y estudiaremos el significado de cada uno de estos operadores.

e. Separadores

Se usan para informar al compilador de Java de cómo están agrupadas las cosas en el código.

Los separadores admitidos por Java son: *{ } , : ;*

f. Comentarios y espacios en blanco

El compilador de Java reconoce y elimina los espacios en blanco, tabuladores, retornos de carro y comentarios durante el análisis del código fuente.

Los comentarios se pueden presentar en tres formatos distintos:

Formato	Uso
/*comentario*/	Se ignoran todos los caracteres entre /* */. Proviene del C
//comentario	Se ignoran todos los caracteres detrás de // hasta el fin de línea. Proviene del C++
/**comentario*/	Lo mismo que /* */ pero se podrán utilizar para documentación automática.

Tabla 4: Formatos de comentarios Java

Por ejemplo la siguiente línea de código presenta un comentario:

```
int alturaMinima = 150; // No menos de 150 centímetros
```

C. EXPRESIONES

Los operadores, variables y las llamadas a métodos pueden ser combinadas en secuencias conocidas como expresiones. El comportamiento real de un programa Java se logra a través de expresiones, que se agrupan para crear *sentencias*.

Una expresión es una serie de variables, operadores y llamadas a métodos (construida conforme a la sintaxis del lenguaje) que se evalúa a un único valor.

Entre otras cosas, las expresiones son utilizadas para realizar cálculos, para asignar valores a variables, y para ayudar a controlar la ejecución del flujo del programa. La tarea de una expresión se compone de dos partes: realiza el cálculo indicado por los elementos de la expresión y devuelve el valor obtenido como resultado del cálculo.

Los operadores devuelven un valor, por lo que el uso de un operador es una expresión.

Por ejemplo, la siguiente sentencia es una expresión:

```
int contador=1;
```

```
contador++;
```

La expresión *contador++* en este caso particular se evalúa al valor *1*, que era el valor de la variable *contador* antes de que la operación ocurra, pero la variable *contador* adquiere un valor de *2*.

El tipo de datos del valor devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión *contador++* devuelve un entero porque *++* devuelve un valor del mismo tipo que su operando y *contador* es un entero. Otras expresiones devuelven valores booleanos, cadenas...

Una expresión de llamada a un método se evalúa al valor de retorno del método; así el tipo de dato de la expresión de llamada a un método es el mismo que el tipo de dato del valor de retorno de ese método.

Otra sentencia interesante sería:

```
in.read( ) != -1 // in es un flujo de entrada
```

Esta sentencia se compone de dos expresiones:

1. La primera expresión es una llamada al método *in.read().* El método *in.read()* ha sido declarado para devolver un entero, por lo que la expresión *in.read()* se evalúa a un entero.
2. La segunda expresión contenida en la sentencia utiliza el operador *!=*, que comprueba si dos operandos son distintos. En la sentencia en cuestión, los operandos son *in.read()* y -1. El operando *in.read()* es válido para el operador *!=* porque i*n.read()* es una expresión que se evalúa a un entero, así que la expresión *in.read()!=-1* compara dos enteros. El valor devuelto por esta expresión será verdadero o falso dependiendo del resultado de la lectura del fichero *in*.

Como se puede observar, Java permite construir sentencias (expresiones compuestas) a partir de varias expresiones más pequeñas con tal que los tipos de datos requeridos por una parte de la expresión concuerden con los tipos de datos de la otra.

D. BLOQUES Y ÁMBITO

En Java el código fuente está dividido en partes separadas por llaves, denominas *bloques*. Cada bloque existe independiente de lo que está fuera de él, agrupando en su interior sentencias (expresiones) relacionadas.

Desde un bloque externo parece que todo lo que está dentro de llaves se ejecuta como una sentencia. Pero, ¿qué es un bloque externo?. Esto tiene explicación si entendemos que existe una *jerarquía de bloques*, y que un bloque puede contener uno o más subbloques anidados.

El concepto de ámbito está estrechamente relacionado con el concepto de bloque y es muy importante cuando se trabaja con variables en Java. El ámbito se refiere a cómo las secciones de un programa (bloques) afectan el tiempo de vida de las variables.

Toda variable tiene un ámbito, en el que es usada, que viene determinado por los bloques. Una variable definida en un bloque interno no es visible por el bloque externo.

Las llaves de separación son importantes no sólo en un sentido lógico, ya que son la forma de que el compilador diferencie dónde acaba una sección de código y dónde comienza otra, sino que tienen una connotación estética que facilita la lectura de los programas al ser humano.

Así mismo, para identificar los diferentes bloques se utilizan sangrías. Las sangrías se utilizan para el programador, no para el compilador. La sangría (también denominada *indentación*) más adecuada para la estética de un programa Java son dos espacios:

```
{
  // Bloque externo
  int x = 1;
  {
    // Bloque interno invisible al exterior
    int y = 2;
  }
  x = y; // Da error: Y fuera de ambito
}
```

II.2. TIPOS DE DATOS

A. TIPOS DE DATOS SIMPLES

Es uno de los conceptos fundamentales de cualquier lenguaje de programación. Estos definen los métodos de almacenamiento disponibles para representar información, junto con la manera en que dicha información ha de ser interpretada.

Para crear una variable (de un tipo simple) en memoria debe declararse indicando su tipo de variable y su identificador que la identificará de forma única. La sintaxis de declaración de variables es la siguiente:

```
TipoSimple Identificador1, Identificador2;
```

Esta sentencia indica al compilador que reserve memoria para dos variables del tipo simple *TipoSimple* con nombres *Identificador1* e *Identificador2.*

Los tipos de datos en Java pueden dividirse en dos categorías: simples y compuestos. Los simples son tipos nucleares que no se derivan de otros tipos, como los enteros, de coma flotante, booleanos y de carácter. Los tipos compuestos se basan en los tipos simples, e incluyen las cadenas, las matrices y tanto las clases como las interfaces, en general.

Cada tipo de datos simple soporta un conjunto de literales que le pueden ser asignados, para darles valor. En este apartado se explican los tipos de datos simples (o primitivos) que presenta Java, así como los literales que soporta (sintaxis de los valores que se les puede asignar).

a.) Tipos de datos enteros

Se usan para representar números enteros con signo. Hay cuatro tipos: *byte, short, int y long.*

Tipo	Tamaño
byte	1Byte (8 bits)
short	2 Bytes (16 bits)
int	4 Bytes (32 bits)
long	8 Bytes (64 bits)

Tabla 5: Tipos de datos enteros

Literales enteros

Son básicos en la programación en Java y presentan tres formatos:

- Decimal: Los literales decimales aparecen como números ordinarios sin ninguna notación especial.
- Hexadecimal: Los hexadecimales (base 16) aparecen con un *0x* ó *0X* inicial, notación similar a la utilizada en C y C++.
- Octal: Los octales aparecen con un 0 inicial delante de los dígitos.

Por ejemplo, un literal entero para el número decimal 12 se representa en Java como 12 en decimal, como 0xC en hexadecimal, y como 014 en octal.

Los literales enteros se almacenan por defecto en el tipo *int*, (4 bytes con signo), o si se trabaja con números muy grandes, con el tipo *long*, (8 bytes con signo), añadiendo una L ó l al final del número.

La declaración de variables enteras es muy sencilla. Un ejemplo de ello sería:

```
long numeroLargo = 0xC; // Por defecto vale 12
```

b.) Tipos de datos en coma flotante

Se usan para representar números con partes fraccionarias. Hay dos tipos de coma flotante: *float* y *double*. El primero reserva almacenamiento para un número de precisión simple de 4 bytes y el segundo lo hace para un numero de precisión doble de 8 bytes.

Tipo	Tamaño
float	4 Byte (32 bits)
double	8 Bytes (64 bits)

Tabla 6: Tipos de datos numéricos en coma flotante

Literales en coma flotante

Representan números decimales con partes fraccionarias. Pueden representarse con notación estándar (563,84) o científica (5.6384e2).

De forma predeterminada son del tipo *double* (8 bytes). Existe la opción de usar un tipo más corto (el tipo *float* de 4 bytes), especificándolo con una F ó f al final del número.

La declaración de variables de coma flotante es muy similar a la de las variables enteras. Por ejemplo:

```
double miPi = 314.16e-2 ; // Aproximadamente
float temperatura = (float)36.6; // Paciente sin fiebre
```

Se realiza un moldeado a *temperatura*, porque todos los literales con decimales por defecto se consideran *double*.

c.) Tipo de datos boolean

Se usa para almacenar variables que presenten dos estados, que serán representados por los valores *true* y *false*. Representan valores bi-estado, provenientes del denominado *álgebra de bool*.

Literales Booleanos

Java utiliza dos palabras clave para los estados: *true* (para verdadero) y *false* (para falso). Este tipo de literales es nuevo respecto a C/C++, lenguajes en los que el valor de falso se representaba por un 0 numérico, y verdadero cualquier número que no fuese el 0.

Para declarar un dato del tipo booleano se utiliza la palabra reservada *boolean:*

```
boolean reciboPagado = false; // ¡¿Aun no nos han pagado?!
```

d.) Tipo de datos carácter

Se usa para almacenar caracteres *Unicode* simples. Debido a que el conjunto de caracteres *Unicode* se compone de valores de 16 bits, el tipo de datos *char* se almacena en un entero sin signo de 16 bits.

Java a diferencia de C/C++ distingue entre matrices de caracteres y cadenas.

Literales carácter

Representan un único carácter (de la tabla de caracteres Unicode 1.1) y aparecen dentro de un par de comillas simples. De forma similar que en C/C++. Los caracteres especiales (de control y no imprimibles) se representan con una barra invertida ('\') seguida del código carácter.

Descripción	Representación	Valor Unicode
Caracter Unicode	*\udddd*	
Numero octal	*\ddd*	
Barra invertida	\\	*\u005C*
Continuación	\	\
Retroceso	*\b*	*\u0008*
Retorno de carro	*\r*	*\u000D*
Alimentación de formularios	*\f*	*\u000C*
Tabulación horizontal	*\t*	*\u0009*
Línea nueva	*\n*	*\u000A*
Comillas simples	*\'*	*\u0027*
Comillas dobles	*\"*	*\u0022*
Números arábigos ASCII	*0-9*	*\u0030 a \u0039*
Alfabeto ASCII en mayúsculas	*A.-Z*	*\u0041 a \u005A*
Alfabeto ASCII en minúsculas	*a.-z*	*\u0061 a \u007A*

Tabla 7: Caracteres especiales Java

Las variables de tipo *char* se declaran de la siguiente forma:

```
char letraMayuscula = 'A'; // Observe la necesidad de las ' '
char letraV = '\u0056'; // Letra 'V'
```

e.) Conversion de tipos de datos

En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina "conversión", "moldeado" o "tipado". La conversión se lleva a cabo colocando el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir de la forma siguiente:

```
char c = (char)System.in.read();
```

La función *read* devuelve un valor *int*, que se convierte en un *char* debido a la conversión *(char)*, y el valor resultante se almacena en la variable de tipo carácter *c*.

El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un *long* en un *int*, el compilador corta los 32 bits superiores del *long* (de 64 bits), de forma que encajen en los 32 bits del *int*, con lo que si contienen información útil, esta se perderá.

Por ello se establece la norma de que "en las conversiones el tipo destino siempre debe ser igual o mayor que el tipo fuente":

Tipo Origen	Tipo Destino
byte	*double, float, long, int, char, short*
short	*double, float, long, int*
char	*double, float, long, int*
int	*double, float, long*
long	*double, float*
float	*double*

Tabla 8: Conversiones sin pérdidas de información

B. VECTORES Y MATRICES

Una matriz es una construcción que proporciona almacenaje a una lista de elementos del mismo tipo, ya sea simple o compuesto. Si la matriz tiene solo una dimensión, se la denomina *vector*.

En Java los vectores se declaran utilizando corchetes (*[y]*), tras la declaración del tipo de datos que contendrá el vector. Por ejemplo, esta sería la declaración de un vector de números enteros (*int*):

```
int vectorNumeros[ ]; // Vector de números
```

Se observa la ausencia de un número que indique cuántos elementos componen el vector, debido a que Java no deja indicar el tamaño de un vector vacío cuando le declara. La asignación de memoria al vector se realiza de forma explícita en algún momento del programa.

Para ello o se utiliza el operador *new*:

```
int vectorNumeros = new int[ 5 ]; // Vector de 5 números
```

O se asigna una lista de elementos al vector:

```
int vectorIni = { 2, 5, 8}; // == int vectorIni[3]=new int[3];
```

Se puede observar que los corchetes son opcionales en este tipo de declaración de vector, tanto después del tipo de variable como después del identificador.

Si se utiliza la forma de *new* se establecerá el valor *0* a cada uno de los elementos del vector.

C. CADENAS

En Java se tratan como una clase especial llamada *String*. Las cadenas se gestionan internamente por medio de una instancia de la clase *String*. Una instancia de la clase *String* es un objeto que ha sido creado siguiendo la descripción de la clase.

Cadenas constantes

Representan múltiples caracteres y aparecen dentro de un par de comillas dobles. Se implementan en Java con la clase *String*. Esta representación es muy diferente de la de C/C++ de cadenas como una matriz de caracteres.

Cuando Java encuentra una constante de cadena, crea un caso de la clase *String* y define su estado, con los caracteres que aparecen dentro de las comillas dobles.

Vemos un ejemplo de cadena declarada con la clase *String* de Java:

```
String capitalUSA = ”Washington D.C.”;
```

```
String nombreBonito = ”Amelia”;
```

Más tarde profundizaremos con detenimiento en las cadenas Java.

II.3. OPERADORES

A. INTRODUCCIÓN

Los operadores son un tipo de *tokens* que indican una evaluación o computación para ser realizada en objetos o datos, y en definitiva sobre identificadores o constantes.

Además de realizar la operación, un operador devuelve un valor, ya que son parte fundamental de las expresiones.

El valor y tipo que devuelve depende del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos devuelven un número como resultado de su operación.

Los operadores realizan alguna función sobre uno, dos o tres operandos.

Los operadores que requieren un operando son llamados *operadores unarios.* Por ejemplo, el operador "++" es un operador unario que incrementa el valor de su operando en una unidad.

Los operadores unarios en Java pueden utilizar tanto la notación prefija como la posfija.

La notación prefija indica que el operador aparece antes que su operando.

```
++contador // Notación prefija, se evalúa a: contador+1
```

La notación posfija indica que el operador aparece después de su operando:

```
contador++ // Notación posfija, se evalúa a: contador
```

Los operadores que requieren dos operandos se llaman *operadores binarios*. Por ejemplo el operador "=" es un operador binario que asigna el valor del operando del lado derecho al operando del lado izquierdo.

Todas los operadores binarios en Java utilizan notación infija, lo cual indica que el operador aparece entre sus operandos.

```
operando1 operador operando2
```

Por último, los operadores ternarios son aquellos que requieren tres operandos. El lenguaje Java tiene el operador ternario, "*?*":, que es una sentencia similar a la if-else.

Este operador ternario usa notación infija; y cada parte del operador aparece entre operandos:

```
expresión ? operación1 : operación2
```

Los operadores de Java se pueden dividir en las siguientes cuatro categorías:

- Aritméticos.
- De comparación y condicionales.
- A nivel de bits y lógicos.
- De asignación.

B. OPERADORES ARITMÉTICOS

El lenguaje Java soporta varios operadores aritméticos para los números enteros y en coma flotante. Se incluye + (suma), - (resta), * (multiplicación), / (división), y *%* (módulo, es decir, resto de una división entera). Por ejemplo:

```
sumaEste + aEste; //Suma los dos enteros
divideEste % entreEste; //Calcula el resto de dividir 2 enteros
```

Operador	Uso	Descripción
+	*op1 + op2*	Suma op1 y op2
-	*op1 - op2*	Resta op2 de op1
*	*op1 * op2*	Multiplica op1 por op2
/	*op1 / op2*	Divide op1 por op2
%	*op1 % op2*	Calcula el resto de dividir op1 entre op2

Tabla 9: Operadores aritméticos binarios de Java

El tipo de los datos devueltos por una operación aritmética depende del tipo de sus operandos; si se suman dos enteros, se obtiene un entero como tipo devuelto con el valor de la suma de los dos enteros.

Estos operadores se deben utilizar con operandos del mismo tipo, o si no realizar una conversión de tipos de uno de los dos operandos al tipo del otro.

El lenguaje Java sobrecarga la definición del operador + para incluir la concatenación de cadenas. El siguiente ejemplo utiliza + para concatenar la cadena "*Contados "*, con el valor de la variable *contador* y la cadena " *caracteres."*:

```
System.out.print("Contados" + contador + "caracteres.");
```

Esta operación automáticamente convierte el valor de *contador* a una cadena de caracteres.

Los operadores + y - tienen versiones unarias que realizan las siguientes operaciones:

Operador	Uso	Descripción
+	+op	Convierte op a entero si es un byte, short o char
-	-op	Niega aritméticamente op

Tabla 10: Versiones unarias de los operadores "+" y "-"

El operador - realiza una negación del número en complemento A2, es decir, cambiando de valor todos sus bits y sumando 1 al resultado final:

```
 42 -> 00101010
-42 -> 11010110
```

Existen dos operadores aritméticos que funcionan como atajo de la combinación de otros: ++ que incrementa su operando en 1, y -- que decrementa su operando en 1.

Ambos operadores tienen una versión prefija, y otra posfija. La utilización la correcta es crítica en situaciones donde el valor de la sentencia es utilizado en mitad de un cálculo más complejo, por ejemplo para control de flujos:

Operador	Uso	Descripción
++	*op++*	Incrementa op en 1; se evalúa al valor anterior al incremento
++	*++op*	Incrementa op en 1; se evalúa al valor posterior al incremento
--	*op--*	Decrementa op en 1; se evalúa al valor anterior al incremento
--	*--op*	Decrementa op en 1; se evalúa al valor posterior al incremento

Tabla 11: Operaciones con "++" y "--"

C. OPERADORES DE COMPARACIÓN Y CONDICIONALES

Un operador de comparación compara dos valores y determina la relación existente entre ambos. Por ejemplo, el operador *!=* devuelve verdadero (*true*) si los dos operandos son distintos. La siguiente tabla resume los operadores de comparación de Java:

Operador	Uso	Devuelve verdadero si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Tabla 12: Operadores de comparación

Los operadores de comparación suelen ser usados con los operadores condicionales para construir expresiones complejas que sirvan para la toma de decisiones. Un operador de este tipo es *&&*, el cual realiza la operación booleana *and*. Por ejemplo, se pueden utilizar dos operaciones diferentes de comparación con *&&* para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si la variable *index* de una matriz se encuentra entre dos límites (mayor que cero y menor que la constante *NUMERO_ENTRADAS)*:

```
( 0 < index ) && ( index < NUMERO_ENTRADAS )
```

Se debe tener en cuenta que en algunos casos, el segundo operando de un operador condicional puede no ser evaluado. En caso de que el primer operando del operador *&&* valga falso, Java no evaluará el operando de la derecha:

```
(contador < NUMERO_ENTRADAS) && ( in.read() != -1 )
```

Si *contador* es menor que *NUMERO_ENTRADAS*, el valor de retorno de *&&* puede ser determinado sin evaluar el operando de la parte derecha. En este caso *in.read* no será llamado y un carácter de la entrada estándar no será leído.

Si el programador quiere que se evalúe la parte derecha, deberá utilizar el operador *&* en lugar de *&&*.

De la misma manera se relacionan los operadores || y | para la exclusión lógica (OR).

Java soporta cinco operadores condicionales , mostrados en la siguiente tabla:

Operador	Uso	Devuelve verdadero si...
&&	*op1 && op2*	op1 y op2 son ambos verdaderos, condicionalmente evalúa op2
&	*op1 & op2*	op1 y op2 son ambos verdaderos, siempre evalúa op1 y op2
||	*op1 || op2*	op1 o op2 son verdaderos, condicionalmente evalúa op2
|	*op1 | op2*	op1 o op2 son verdaderos, siempre evalúa op1 y op2
!	*! op*	op es falso

Tabla 13: Operadores condicionales

Además Java soporta un operador ternario, el *?:*, que se comporta como una versión reducida de la sentencia *if-else*:

```
expresion ? operacion1 : operacion2
```

El operador *?*: evalúa la *expresion* y devuelve *operación1* si es cierta, o devuelve *operación2* si *expresion* es falsa.

D. OPERADORES DE BIT

Un operador de bit permite realizar operaciones de bit sobre los datos. Existen dos tipos: los que desplazan (mueven) bits, y operadores lógicos de bit.

a.) Operadores de desplazamiento de bits

Operador	Uso	Operación
>>	*op1 >> op2*	Desplaza los bits de op1 a la derecha op2 veces
<<	*op1 << op2*	Desplaza los bits de op1 a la izquierda op2 veces
>>>	*op1 >>> op2*	Desplaza los bits de op1 a la derecha op2 veces (sin signo)

Tabla 14: Operadores de desplazamiento de bits

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la parte izquierda el número de veces indicado por el operando de la parte derecha. El desplazamiento ocurre en la dirección indicada por el operador. Por ejemplo, la siguiente sentencia, desplaza los bits del entero 13 a la derecha una posición:

```
13 >> 1;
```

La representación en binario del número 13 es 1101. El resultado de la operación de desplazamiento es 1101 desplazado una posición a la derecha, 110 o 6 en decimal. Se debe tener en cuenta que el bit más a la derecha se pierde en este caso.

Un desplazamiento a la derecha una posición es equivalente a dividir el operando del lado izquierdo por 2, mientras que un desplazamiento a la izquierda de una posición equivale a multiplicar por 2, pero un desplazamiento es más eficiente, computacionalmente hablando, que una división o multiplicación.

El desplazamiento sin signo >>> funciona de la siguiente manera:

- Si se desplaza con signo el número -1 (1111), seguirá valiendo -1, dado que la extensión de signo sigue introduciendo unos en los bits más significativos.
- Con el desplazamiento sin signo se consigue introducir ceros por la izquierda, obteniendo el número 7 (0111).

Este tipo de desplazamientos es especialmente útil en la utilización de máscaras gráficas.

b.) Operadores de lógica de bits

La lógica de bits (lógica de Bool) se utiliza para modelizar condiciones biestado y trabajar con ellas (cierto/falso, *true*/*false*, 1/0).

En Java hay cuatro operadores de lógica de bits:

Operador	Uso	Operación
&	*op1 & op2*	AND
|	*op1 | op2*	OR
^	*op1 ^ op2*	OR Exclusivo
~	*~op2*	Complemento

Tabla 15: Operadores de lógica de bits

El operador *&* realiza la operación AND de bit. Aplica la *función AND* sobre cada par de bits de igual peso de cada operando. La *función AND* es evaluada a cierto si ambos operandos son ciertos.

Por ejemplo vamos a aplicar la operación *AND* a los valores 12 y 13:

```
12 & 13
```

El resultado de esta operación es 12. ¿Por qué?. La representación en binario de 12 es 1100, y de 13 es 1101. La *función AND* pone el bit de resultado a uno si los dos bits de los operandos son 1, sino, el bit de resultado es 0:

```
  1101
& 1100
------
  1100
```

El operador | realiza la operación OR de bit. Aplica la *función OR* sobre cada par de bits de igual peso de cada operando. La *función OR* es evaluada a cierto si alguno de los operandos es cierto.

El operador ^ realiza la operación OR exclusivo de bit (XOR). Aplica la *función XOR* sobre cada par de bits de igual peso de cada operando. La *función XOR* es evaluada a cierto si los operandos tienen el mismo valor.

Para finalizar, el operador de complemento invierte el valor de cada bit del operando. Convierte el falso en cierto, y el cierto en falso:

Entre otras cosas, la manipulación bit es útil para gestionar indicadores booleanos (banderas). Supongamos, por ejemplo, que se tiene varios indicadores booleanos en nuestro programa, los cuales muestran el estado de varios componentes del programa: *esVisible*, *esArrastrable,...* En lugar de definir una variable booleana para cada indicador, se puede definir una única variable para todos ellos. Cada bit de dicha variable representará el estado vigente de uno de los indicadores. Se deberán utilizar entonces manipulaciones de bit para establecer y leer cada indicador.

Primero, se deben preparar las constantes de cada indicador. Esos indicadores deben ser diferentes unos de otros (en sus bits) para asegurar que el bit de activación no se solape con otro indicador. Después se debe definir la variable de banderas, cuyos bits deben de poder ser configurados según el estado vigente en cada indicador.

El siguiente ejemplo inicia la variable de banderas *flags* a 0, lo que significa que todos los indicadores están desactivados (ninguno de los bits es 1):

```
final int VISIBLE = 1;
final int ARRASTRABLE = 2;
final int SELECCIONABLE = 4;
final int MODIFICABLE = 8;
int flags = 0;
```

Para activar el indicador *VISIBLE*, se deberá usar la sentencia:

```
flags = flags | VISIBLE;
```

Para comprobar la visibilidad se deberá usar la sentencia:

```
if ( (flags & VISIBLE) == 1 )
  //Lo que haya que hacer
```

E. OPERADORES DE ASIGNACIÓN

El operador de asignación básico es el =, que se utiliza para asignar un valor a otro. Por ejemplo:

```
int contador = 0;
```

Inicia la variable *contador* con un valor *0.*

Java además proporciona varios operadores de asignación que permiten realizar un atajo en la escritura de código. Permiten realizar operaciones aritméticas, lógicas, de bit y de asignación con un único operador.

Supongamos que necesitamos sumar un número a una variable y almacenar el resultado en la misma variable, como a continuación:

```
i = i + 2;
```

Se puede abreviar esta sentencia con el operador de atajo +=, de la siguiente manera*:*

```
i += 2;
```

La siguiente tabla muestra los operadores de atajo de asignación y sus equivalentes largos:

Operador	Uso	Equivalente a
+=	*op1 += op2*	op1 = op1 + op2
-=	*op1 -= op2*	op1 = op1 - op2
*=	*op1 *= op2*	op1 = op1 * op2
/=	*op1 /= op2*	op1 = op1 / op2
%=	*op1 %= op2*	op1 = op1 % op2
&=	*op1 &= op2*	op1 = op1 & op2

Tabla 16: Operadores de atajo de asignación

F. PRECEDENCIA DE OPERADORES

Cuando en una sentencia aparecen varios operadores el compilador deberá de elegir en qué orden aplica los operadores. A esto se le llama *precedencia.*

Los operadores con mayor precedencia son evaluados antes que los operadores con una precedencia relativa menor.

Cuando en una sentencia aparecen operadores con la misma precedencia:

- Los operadores de asignación son evaluados de derecha a izquierda.
- Los operadores binarios, (menos los de asignación) son evaluados de izquierda a derecha.

Se puede indicar explícitamente al compilador de Java cómo se desea que se evalúe la expresión con paréntesis balanceados ( ). Para hacer que el código sea más fácil de leer y mantener, es preferible ser explícito e indicar con paréntesis que operadores deben ser evaluados primero.

La siguiente tabla muestra la precedencia asignada a los operadores de Java. Los operadores de la tabla están listados en orden de precedencia: cuanto más arriba aparezca un operador, mayor es su precedencia. Los operadores en la misma línea tienen la misma precedencia:

Tipo de operadores	Operadores de este tipo
Operadores posfijos	[ ] . (parametros) expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
Creación o conversión	new (tipo) expr
Multiplicación	* / %
Suma	+ -
Desplazamiento	<<
Comparación	< <= = instanceof
Igualdad	== !=
AND a nivel de bit	&
OR a nivel de bit	^
XOR a nivel de bit	|
AND lógico	&&
OR lógico	||
Condicional	? :
Asignación	= += -= *= /= %= &= ^= |= <<= = =

Tabla 17: Prededencia de operadores

Por ejemplo, la siguiente expresión produce un resultado diferente dependiendo de si se realiza la suma o división en primer lugar:

```
x + y / 100
```

Si no se indica explícitamente al compilador el orden en que se quiere que se realicen las operaciones, entonces el compilador decide basándose en la precedencia asignada a los operadores. Como el operador de división tiene mayor precedencia que el operador de suma el compilador evaluará *y/100* primero.

Así:

```
x + y / 100
```

Es equivalente a:

```
x + (y / 100)
```

II.4. ESTRUCTURAS DE CONTROL

A. INTRODUCCIÓN

Durante un programa existen acciones que se han de repetir un número determinado de veces. Por ejemplo, leer 3 caracteres de un flujo de entrada *in* se codificaría:

```
in.read();
in.read();
in.read();
```

Este código además de poco elegante sería inviable para una repetición de 3000 lecturas. Por eso aparecen las estructuras de control, que facilitan que determinadas acciones se realicen varias veces, mientras que una condición se cumpla, y en definitiva, tomar decisiones de qué hacer en función de las condiciones que se den en el programa en un momento dado de su ejecución.

Así, nuestro ejemplo se podría indicar como:

```
int i=0;
for ( i=0 ; i <= 3 ; i++ )
  in.read();
```

Donde bastaría cambiar el *3* por cualquier otro número para que la lectura se repitiese ese número de veces.

El lenguaje Java soporta las estructuras de control:

Sentencia	Clave
Toma de decisión	if-else, switch-case
Bucle	for, while, do-while
Misceláneo	break, continue, label:, return, goto

Tabla 18: Estructuras de control

Aunque *goto* es una palabra reservada, actualmente el lenguaje Java no soporta la sentencia *goto*. Se puede utilizar las sentencias de bifurcación en su lugar.

B. LAS SENTENCIAS CONDICIONALES: IF Y SWITCH

a.) La sentencia if - else

La sentencia *if-else* de Java dota a los programas de la habilidad de ejecutar distintos conjuntos de sentencias según algún criterio.

La sintaxis de la sentencia *if-else* es:

```
if ( condición )
  Bloque de código a ejecutar si la condición es cierta
else
  Bloque de código a ejecutar si la condición es falsa
```

La parte del *else* es opcional, y un bloque de código puede ser simplemente la sentencia vacía *;* para representar que en ese caso no se ha de ejecutar nada.

Supongamos que un programa debe realizar diferentes acciones dependiendo de si el usuario oprime el botón aceptar o el botón cancelar en una ventana de dialogo. Nuestro programa puede realizar esto usando la sentencia *if - else*:

```
// La respuesta es Aceptar o Cancelar
if (respuesta == Aceptar) {
  // código para realizar la acción Aceptar
  System.out.println( "Su peticion esta siendo atendida" );
}
else {
  // código para realizar la acción Cancelar
  System.out.println( "Cancelando accion" );
}
```

Se pueden anidar expresiones *if-else*, para poder implementar aquellos casos con múltiples acciones. Esto es lo que se suele denominar como sentencias *else if*.

Por ejemplo, supongamos que se desea escribir un programa que clasifique según el contenido de una variable *valor*, asigne una letra a una variable *clasificacion*: A para un valor del 100-91, B de 90-81, C para 80-71 y F si no es ninguno de los anteriores:

```
int valor;
char clasificacion;
if (valor > 90)
  {clasificacion='A';}
else
  if (valor > 80)
    {clasificacion='B';}
  else
    if (valor > 70)
      {clasificacion='C';}
    else
     {clasificacion='F';}
```

Se pueden escribir los *if* en las mismas líneas que los *else*, pero desde este tutorial se insta a utilizar la forma indentada (como se ha podido ver en el ejemplo), pues es más clara para el lector.

Este sistema de programación (*else if)* no es demasiado recomendable, y por ello el lenguaje Java incluye la sentencia *switch,* que veremos a continuación, para dirigir el flujo de control de variables con múltiples valores.

b.) La sentencia switch

Mediante la sentencia *switch* se puede seleccionar entre varias sentencias según el valor de cierta expresión.

La forma general de *switch* es la siguiente:

```
switch ( expresionMultivalor ) {
  case valor1 : conjuntoDeSentencias; break;
  case valor2 : conjuntoDeSentencias; break;
  case valor3: conjuntoDeSentencias; break;
  default: conjuntoDeSentencias; break;
}
```

La sentencia *switch* evalúa la *expresiónMultivalor* y ejecuta el *conjuntoDeSentencias* que aparece junto a la cláusula *case* cuyo *valor* corresponda con el de la *expresiónMultivalor*.

Cada sentencia *case* debe ser única y el *valor* que evalúa debe ser del mismo tipo que el devuelto por la *expresiónMultivalor* de la sentencia *switch*.

Las sentencias *break* que aparecen tras cada *conjuntoDeSentencias* provocan que el control salga del *switch* y continúe con la siguiente instrucción al *switch*. Las sentencias *break* son necesarias porque sin ellas se ejecutarían secuencialmente las sentencias *case* siguientes. Existen ciertas situaciones en las que se desea ejecutar secuencialmente algunas o todas las sentencias *case*, para lo que habrá que eliminar algunos *break.*

Finalmente, se puede usar la sentencia *default* para manejar los valores que no son explícitamente contemplados por alguna de las sentencias *case*. Su uso es altamente recomendado.

Por ejemplo, supongamos un programa con una variable entera *meses* cuyo valor indica el mes actual, y se desea imprimir el nombre del mes en que estemos. Se puede utilizar la sentencia *switch* para realizar esta operación:

```
int meses;
switch ( meses ){
  case 1: System.out.println( "Enero" ); break;
  case 2: System.out.println( "Febrero" ); break;
  case 3: System.out.println( "Marzo" ); break;
  //Demas meses
  // . . .
  case 12: System.out.println( "Diciembre" ); break;
  default: System.out.println( "Mes no valido" ); break;
}
```

Por supuesto, se puede implementar esta estructura como una sentencia *if else if*:

```
int meses;
if ( meses == 1 ) {
  System.out.println( "Enero" );
}
else
  if ( meses == 2 ) {
    System.out.println( "Febrero" );
```

```
  }
  // Y así para los demás meses
```

El decidir si usar la sentencia *if* o *switch* depende del criterio de cada caso. Se puede decidir cuál usar basándonos en la legibilidad, aunque se recomienda utilizar *switch* para sentencias con más de tres o cuatro posibilidades.

C. SENTENCIAS DE ITERACIÓN O BUCLES: FOR, DO, WHILE

a.) Bucle while

El bucle *while* es el bucle básico de iteración. Sirve para realizar una acción sucesivamente mientras se cumpla una determinada condición.

La forma general del bucle *while* es la siguiente:

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

Las *sentencias* se ejecutan mientras la *expresiónBooleana* tenga un valor de *verdadero*.

Se utiliza, por ejemplo para estar en un bucle del que no hay que salir hasta que no se cumpla una determinada condición. Por ejemplo, multiplicar un número por 2 hasta que sea mayor que 100:

```
int i = 1;
while ( i <= 100 ) {
  i = i * 2;
}
```

Con él se podrían eliminar los bucles *do-while* y *for* por ser extensiones de éste, pero que se incluyen en el lenguaje para facilitar la programación.

b.) Bucle do-while

El bucle *do-while* es similar al bucle *while*, pero en el bucle *while* la expresión se evalúa al principio del bucle y en el bucle *do-while* la evaluación se realiza al final.

La forma general del bucle *do-while* es la siguiente:

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

La sentencia *do-while* es el constructor de bucles menos utilizado en la programación, pero tiene sus usos, cuando el bucle deba ser ejecutado por lo menos una vez.

Por ejemplo, cuando se lee información de un archivo, se sabe que siempre se debe leer por lo menos un carácter:

```
int c;
do {
  c = System.in.read( );
```

```
  // Sentencias para tratar el carácter c
} while ( c != -1 ); // No se puede leer más (Fin fichero)
```

c.) Bucle for

Mediante la sentencia *for* se resume un bucle *do-while* con una iniciación previa. Es muy común que en los bucles *while* y *do-while* se inicien las variables de control de número de pasadas por el bucle, inmediatamente antes de comenzar los bucles. Por eso el bucle *for* está tan extendido.

La forma general de la sentencia *for* es la siguiente:

```
for ( iniciación ; terminación ; incremento )
  sentencias;
```

La iniciación es una sentencia que se ejecuta una vez antes de entrar en el bucle.

La terminación es una expresión que determina cuándo se debe terminar el bucle. Esta expresión se evalúa al final de cada iteración del bucle. Cuando la expresión se evalúa a falso, el bucle termina.

El incremento es una expresión que es invocada en cada iteración del bucle. En realidad puede ser una acción cualquiera, aunque se suele utilizar para incrementar una variable contador:

```
for ( i = 0 ; i < 10 ; i++ )
```

Algunos (o todos) estos componentes pueden omitirse, pero los puntos y coma siempre deben aparecer (aunque sea sin nada entre sí).

Se debe utilizar el bucle *for* cuando se conozcan las restricciones del bucle (su instrucción de iniciación, criterio de terminación e instrucción de incremento).

Por ejemplo, los bucles *for* son utilizados comúnmente para iterar sobre los elementos de una matriz, o los caracteres de una cadena:

```
// cad es una cadena (String)
for ( int i = 0; i < cad.length() ; i++){
  // hacer algo con el elemento i-ésimo de cad
}
```

D. SENTENCIAS DE SALTO: BREAK, CONTINUE Y RETURN

a.) Sentencia break

La sentencia *break* provoca que el flujo de control salte a la sentencia inmediatamente posterior al bloque en curso. Ya se ha visto anteriormente la sentencia *break* dentro de la sentencia *switch*.

El uso de la sentencia break con sentencias etiquetadas es una alternativa al uso de la sentencia *goto*, que no es soportada por el lenguaje Java.

Se puede etiquetar una sentencia poniendo una identificador Java válido seguido por dos puntos antes de la sentencia:

```
nombreSentencia: sentenciaEtiquetada
```

La sentencia *break* se utiliza para salir de una sentencia etiquetada, llevando el flujo del programa al final de la sentencia de programa que indique:

```
break nombreSentencia2;
```

Un ejemplo de esto sería el programa:

```
void gotoBreak() {
    System.out.println("Ejemplo de break como 'goto' ");
a:  for( int i=1; i<10; i++ ){
      System.out.print(" i="+i);
      for( int j=1; j<10; j++ ){
        if ( j==5 )
          break a; //Sale de los dos bucles!!!
        System.out.print(" j="+j);
      }
      System.out.print("No llega aquí");
    }
  }
```

Al interpretar *break a*, no solo se rompe la ejecución del bucle interior (el de *j*), sino que se salta al final del bucle *i*, obteniéndose:

```
i=1 j=1 j=2 j=3
```

Nota: Se desaconseja esta forma de programación, basada en *goto*, y con saltos de flujo no controlados.

b.) Sentencia continue

Del mismo modo que en un bucle se puede desear romper la iteración, también se puede desear continuar con el bucle, pero dejando pasar una determinada iteración.

Se puede usar la sentencia *continue* dentro de los bucles para saltar a otra sentencia, aunque no puede ser llamada fuera de un bucle.

Tras la invocación a una sentencia *continue* se transfiere el control a la condición de terminación del bucle, que vuelve a ser evaluada en ese momento, y el bucle continúa o no dependiendo del resultado de la evaluación. En los bucles *for* además en ese momento se ejecuta la cláusula de incremento (antes de la evaluación). Por ejemplo el siguiente fragmento de código imprime los números del 0 al 9 no divisibles por 3:

```
for ( int i = 0 ; i < 10 ; i++ ) {
  if ( ( i % 3 ) == 0 )
    continue;
  System.out.print( " " + i );
}
```

Del mismo modo que *break*, en las sentencias *continue* se puede indicar una etiqueta de bloque al que hace referencia. Con ello podemos referirnos a un bloque superior, si

estamos en bucles anidados. Si dicha etiqueta no es indicada, se presupone que nos referimos al bucle en el que la sentencia *continue* aparece.

Por ejemplo, el siguiente fragmento de código:

```
void gotoContinue( ) {
f:  for ( int i=1; i <5; i++ ) {
    for ( int j=1; j<5; j++ ) {
      if ( j>i ) {
        System.out.println(" ");
        continue f;
      }
      System.out.print( " " + (i*j) );
    }
  }
}
```

En este código la sentencia *continue* termina el bucle de *j* y continua el flujo en la siguiente iteración de i. Ese método imprimiría:

```
1
2 4
3 6 9
4 8 12 16
```

Nota: Se desaconseja esta forma de programación, basada en *goto*, y con saltos de flujo no controlados.

c.) Sentencia return

La última de las sentencias de salto es la sentencia *return*, que puede usar para salir del método en curso y retornar a la sentencia dentro de la cual se realizó la llamada.

Para devolver un valor, simplemente se debe poner el valor (o una expresión que calcule el valor) a continuación de la palabra *return.* El valor devuelto por *return* debe coincidir con el tipo declarado como valor de retorno del método.

Cuando un método se declara como *void* se debe usar la forma de *return* sin indicarle ningún valor. Esto se hace para no ejecutar todo el código del programa:

```
int contador;
boolean condicion;
int devuelveContadorIncrementado(){
  return ++contador;
}
void metodoReturn(){
  //Sentencias
  if ( condicion == true )
```

```
    return;
  //Más sentencias a ejecutar si condición no vale true
}
```

E. EXCEPCIONES

Las excepciones son otra forma más avanzada de controlar el flujo de un programa. Con ellas se podrán realizar acciones especiales si se dan determinadas condiciones, justo en el momento en que esas condiciones se den.

Estudiaremos más este sistema de control en el capítulo *"II.8. Gestión de excepciones y errores"* de este tutorial.

II.5. CLASES Y OBJETOS

A. INTRODUCCIÓN

Durante los capítulos anteriores se han dado unas nociones básicas de la sintaxis de Java. A partir de ahora es cuando entramos la verdadera potencia de Java como lenguaje orientado a objetos: las clases y los objetos.

Aquellas personas que nunca hayan programado en un lenguaje orientado a objeto, o que no conozcan las nociones básicas de paradigma conviene que lean el capítulo *"I.1 Introducción a la programación orientada a objetos"* de este tutorial, ya que a partir de ahora los conceptos que en él se exponen se darán por entendidos.

Durante todo este capítulo se va a trabajar en la construcción de una clase *MiPunto*, que modeliza un punto en un espacio plano:

```
class MiPunto{
  int x, y;
  int metodoSuma( int paramX, int paramY ) {
    return ( paramX + paramY );
  }
  double distancia(int x, int y) {
    int dx= this.x – pX;
    int dy = this.y – pY;
    return Math.sqrt(dx*dx + dy*dy);
  }
  void metodoVacio( ) { }
  void inicia( int paramX, int paramY ) {
    x = paramX;
    y = paramY;
  }
  void inicia2( int x, int y ) {
    x = x; // Ojo, no modificamos la variable de instancia!!!
    this.y = y; // Modificamos la variable de instancia!!!
  }
  MiPunto( int paramX, int paramY ) {
    this.x = paramX; // Este this se puede omitir
    y = paramY; // No hace falta this
  }
  MiPunto() {
    inicia(-1,-1); //Por defecto ; this(-1,-1) hace lo mismo
  }
}
```

B. DEFINICIÓN DE UNA CLASE

a.) Introducción

El elemento básico de la programación orientada a objetos en Java es la clase. Una clase define la forma y comportamiento de un objeto.

Para crear una clase sólo se necesita un archivo fuente que contenga la palabra clave reservada *class* seguida de un identificador legal y un bloque delimitado por dos llaves para el cuerpo de la clase.

```
class MiPunto {
}
```

Un archivo de Java debe tener el mismo nombre que la clase que contiene, y se les suele asignar la extensión *".java".* Por ejemplo la clase *MiPunto* se guardaría en un fichero que se llamase *MiPunto.java*. Hay que tener presente que en Java se diferencia entre mayúsculas y minúsculas; el nombre de la clase y el de archivo fuente han de ser exactamente iguales.

Aunque la clase *MiPunto* es sintácticamente correcta, es lo que se viene a llamar una *clase vacía*, es decir, una clase que no hace nada. Las clases típicas de Java incluirán variables y métodos de instancia. Los programas en Java completos constarán por lo general de varias clases de Java en distintos archivos fuente.

Una clase es una plantilla para un objeto. Por lo tanto define la estructura de un objeto y su interfaz funcional, en forma de métodos. Cuando se ejecuta un programa en Java, el sistema utiliza definiciones de clase para crear instancias de las clases, que son los objetos reales. Los términos instancia y objeto se utilizan de manera indistinta. La forma general de una definición de clase es:

```
class Nombre_De_Clase {
  tipo_de_variable nombre_de_atributo1;
  tipo_de_variable nombre_de_atributo2;
  // . . .
  tipo_devuelto nombre_de_método1( lista_de_parámetros ) {
    cuerpo_del_método1;
  }
  tipo_devuelto nombre_de_método2( lista_de_parámetros ) {
    cuerpo_del_método2;
  }
  // . . .
}
```

Los tipos *tipo_de_variable* y *tipo_devuelto*, han de ser tipos simples Java o nombres de otras clases ya definidas. Tanto *Nombre_De_Clase*, como los *nombre_de_atributo* y *nombre_de_método*, han de ser identificadores Java válidos.

b.) Los atributos

Los datos se encapsulan dentro de una clase declarando variables dentro de las llaves de apertura y cierre de la declaración de la clase, variables que se conocen como atributos. Se declaran igual que las variables locales de un método en concreto.

Por ejemplo, este es un programa que declara una clase *MiPunto*, con dos atributos enteros llamados `x` e `y`.

```
class MiPunto {
  int x, y;
}
```

Los atributos se pueden declarar con dos clases de tipos: un tipo simple Java (ya descritos), o el nombre de una clase (será una *referencia a objeto*, véase el punto C.a de este mismo apartado).

Cuando se realiza una instancia de una clase (creación de un objeto) se reservará en la memoria un espacio para un conjunto de datos como el que definen los atributos de una clase. A este conjunto de variables se le denomina *variables de instancia.*

c.) Los métodos

Los métodos son subrutinas que definen la interfaz de una clase, sus capacidades y comportamiento.

Un método ha de tener por nombre cualquier identificador legal distinto de los ya utilizados por los nombres de la clase en que está definido. Los métodos se declaran al mismo nivel que las variables de instancia dentro de una definición de clase.

En la declaración de los métodos se define el tipo de valor que devuelven y a una lista formal de parámetros de entrada, de sintaxis *tipo identificador* separadas por comas. La forma general de una declaración de método es:

```
tipo_devuelto nombre_de_método( lista-formal-de-parámetros ) {
  cuerpo_del_método;
}
```

Por ejemplo el siguiente método devuelve la suma de dos enteros:

```
int metodoSuma( int paramX, int paramY ) {
  return ( paramX + paramY );
};
```

En el caso de que no se desee devolver ningún valor se deberá indicar como tipo la palabra reservada *void.* Así mismo, si no se desean parámetros, la declaración del método debería incluir un par de paréntesis vacíos (sin *void*):

```
void metodoVacio( ) { };
```

Los métodos son llamados indicando una instancia individual de la clase, que tendrá su propio conjunto único de variables de instancia, por lo que los métodos se pueden referir directamente a ellas.

El método *inicia()* para establecer valores a las dos variables de instancia sería el siguiente:

```
void inicia( int paramX, int paramY ) {
  x = paramX;
  y = paramY;
}
```

C. LA INSTANCIACIÓN DE LAS CLASES: LOS OBJETOS

a.) Referencias a Objeto e Instancias

Los tipos simples de Java describían el tamaño y los valores de las variables. Cada vez que se crea una clase se añade otro tipo de dato que se puede utilizar igual que uno de los tipos simples. Por ello al declarar una nueva variable, se puede utilizar un nombre de clase como tipo. A estas variables se las conoce como *referencias a objeto*.

Todas las referencias a objeto son compatibles también con las instancias de subclases de su tipo. Del mismo modo que es correcto asignar un *byte* a una variable declarada como *int*, se puede declarar que una variable es del tipo *MiClase* y guardar una referencia a una instancia de este tipo de clase:

```
MiPunto p;
```

Esta es una declaración de una variable *p* que es una referencia a un objeto de la clase *MiPunto*, de momento con un valor por defecto de *null*. La referencia *null* es una referencia a un objeto de la clase *Object*, y se podrá convertir a una referencia a cualquier otro objeto porque todos los objetos son *hijos* de la clase *Object*.

b.) Constructores

Las clases pueden implementar un método especial llamado *constructor*. Un constructor es un método que inicia un objeto inmediatamente después de su creación. De esta forma nos evitamos el tener que iniciar las variables explícitamente para su iniciación.

El constructor tiene exactamente el mismo nombre de la clase que lo implementa; no puede haber ningún otro método que comparta su nombre con el de su clase. Una vez definido, se llamará automáticamente al constructor al crear un objeto de esa clase (al utilizar el operador *new*).

El constructor no devuelve ningún tipo, ni siquiera *void*. Su misión es iniciar todo estado interno de un objeto (sus atributos), haciendo que el objeto sea utilizable inmediatamente; reservando memoria para sus atributos, iniciando sus valores...

Por ejemplo:

```
MiPunto( ) {
  inicia( -1, -1 );
}
```

Este constructor denominado *constructor por defecto*, por no tener parámetros, establece el valor *-1* a las variables de instancia *x* e *y* de los objetos que construya.

El compilador, por defecto ,llamará al constructor de la superclase *Object()* si no se especifican parámetros en el constructor.

Este otro constructor, sin embargo, recibe dos parámetros:

```
MiPunto( int paraX, int paraY ) {
  inicia( paramX, paramY );
}
```

La lista de parámetros especificada después del nombre de una clase en una sentencia *new* se utiliza para pasar parámetros al constructor.

Se llama al método constructor justo después de crear la instancia y antes de que *new* devuelva el control al punto de la llamada.

Así, cuando ejecutamos el siguiente programa:

```
MiPunto p1 = new MiPunto(10, 20);
System.out.println( “p1.- x = “ + p1.x + “ y = “ + p1.y );
```

Se muestra en la pantalla:

```
p1.- x = 10 y = 20
```

Para crear un programa Java que contenga ese código, se debe de crear una clase que contenga un método *main()*. El intérprete *java* se ejecutará el método *main* de la clase que se le indique como parámetro.

Para más información sobre cómo crear y ejecutar un programa, así como los tipos de programas que se pueden crear en Java, véase el capítulo *"II.12. Creación de programas Java"* de este tutorial.

c.) El operador new

El operador *new* crea una instancia de una clase (*objetos*) y devuelve una referencia a ese objeto. Por ejemplo:

```
MiPunto p2 = new MiPunto(2,3);
```

Este es un ejemplo de la creación de una instancia de *MiPunto*, que es controlador por la referencia a objeto *p2*.

Hay una distinción crítica entre la forma de manipular los tipos simples y las clases en Java: Las referencias a objetos realmente no contienen a los objetos a los que referencian. De esta forma se pueden crear múltiples referencias al mismo objeto, como por ejemplo:

```
MiPunto p3 =p2;
```

Aunque tan sólo se creó un objeto *MiPunto*, hay dos variables (*p2* y *p3*) que lo referencian. Cualquier cambio realizado en el objeto referenciado por *p2* afectará al objeto referenciado por *p3*. La asignación de *p2* a *p3* no reserva memoria ni modifica el objeto.

De hecho, las asignaciones posteriores de *p2* simplemente desengancharán *p2* del objeto, sin afectarlo:

```
p2 = null; // p3 todavía apunta al objeto creado con new
```

Aunque se haya asignado *null* a *p2*, *p3* todavía apunta al objeto creado por el operador *new*.

Cuando ya no haya ninguna variable que haga referencia a un objeto, Java reclama automáticamente la memoria utilizada por ese objeto, a lo que se denomina *recogida de basura*.

Cuando se realiza una instancia de una clase (mediante *new*) se reserva en la memoria un espacio para un conjunto de datos como el que definen los atributos de la clase que se indica en la instanciación. A este conjunto de variables se le denomina *variables de instancia*.

La potencia de las variables de instancia es que se obtiene un conjunto distinto de ellas cada vez que se crea un objeto nuevo. Es importante el comprender que cada objeto tiene su propia copia de las variables de instancia de su clase, por lo que los cambios sobre las variables de instancia de un objeto no tienen efecto sobre las variables de instancia de otro.

El siguiente programa crea dos objetos *MiPunto* y establece los valores de *x* e *y* de cada uno de ellos de manera independiente para mostrar que están realmente separados.

```
MiPunto p4 = new MiPunto( 10, 20 );
MiPunto p5 = new MiPunto( 42, 99 );
System.out.println("p4.- x = " + p4.x + " y = " + p4.y);
System.out.println("p5.- x = " + p5.x + " y = " + p5.y);
```

Este es el aspecto de salida cuando lo ejecutamos.

```
p4.- x = 10 y = 20
p5.- x = 42 y = 99
```

D. ACCESO AL OBJETO

a.) El operador punto (.)

El operador punto (.) se utiliza para acceder a las variables de instancia y los métodos contenidos en un objeto, mediante su referencia a objeto:

```
referencia_a_objeto.nombre_de_variable_de_instancia
referencia_a_objeto.nombre_de_método( lista-de-parámetros );
```

Hemos creado un ejemplo completo que combina los operadores *new* y punto para crear un objeto *MiPunto*, almacenar algunos valores en él e imprimir sus valores finales:

```
MiPunto p6 = new MiPunto( 10, 20 );
System.out.println ("p6.- 1. X=" + p6.x + " , Y=" + p6.y);
p6.inicia( 30, 40 );
System.out.println ("p6.- 2. X=" + p6.x + " , Y=" + p6.y);
```

Cuando se ejecuta este programa, se observa la siguiente salida:

```
p6.- 1. X=10 , Y=20
p6.- 2. X=30 , Y=40
```

Durante las impresiones (método *println()*) *s*e accede al valor de las variables mediante *p6.x* y *p6.y*, y entre una impresión y otra se llama al método *inicia()*, cambiando los valores de las variables de instancia.

Este es uno de los aspectos más importantes de la diferencia entre la programación orientada a objetos y la programación estructurada. Cuando se llama al método *p6.inicia()*, lo primero que se hace en el método es sustituir los nombres de los atributos

de la clase por las correspondientes variables de instancia del objeto con que se ha llamado. Así por ejemplo *x* se convertirá en *p6.x*.

Si otros objetos llaman a *inicia()*, incluso si lo hacen de una manera concurrente, no se producen *efectos laterales*, ya que las variables de instancia sobre las que trabajan son distintas.

b.) La referencia this

Java incluye un valor de referencia especial llamado *this*, que se utiliza dentro de cualquier método para referirse al objeto actual. El valor *this* se refiere al objeto sobre el que ha sido llamado el método actual. Se puede utilizar *this* siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de *this*.

Un refinamiento habitual es que un constructor llame a otro para construir la instancia correctamente. El siguiente constructor llama al constructor parametrizado *MiPunto(x,y)* para terminar de iniciar la instancia:

```
MiPunto() {
  this( -1, -1 ); // Llama al constructor parametrizado
}
```

En Java se permite declarar variables locales, incluyendo parámetros formales de métodos, que se solapen con los nombres de las variables de instancia.

No se utilizan *x* e *y* como nombres de parámetro para el método *inicia*, porque ocultarían las variables de instancia *x* e *y* reales del ámbito del método. Si lo hubiésemos hecho, entonces *x* se hubiera referido al parámetro formal, ocultando la variable de instancia *x:*

```
void inicia2( int x, int y ) {
  x = x; // Ojo, no modificamos la variable de instancia!!!
  this.y = y; // Modificamos la variable de instancia!!!
}
```

E. LA DESTRUCCIÓN DEL OBJETO

a.) La destrucción de los objetos

Cuando un objeto no va a ser utilizado, el espacio de memoria de dinámica que utiliza ha de ser liberado, así como los recursos que poseía, permitiendo al programa disponer de todos los recursos posibles. A esta acción se la da el nombre de *destrucción del objeto*.

En Java la destrucción se puede realizar de forma automática o de forma personalizada, en función de las características del objeto.

b.) La destrucción por defecto: Recogida de basura

El intérprete de Java posee un sistema de recogida de basura, que por lo general permite que no nos preocupemos de liberar la memoria asignada explícitamente.

El recolector de basura será el encargado de liberar una zona de memoria dinámica que había sido reservada mediante el operador *new*, cuando el objeto ya no va a ser utilizado más durante el programa (por ejemplo, sale del ámbito de utilización, o no es referenciado nuevamente).

El sistema de recogida de basura se ejecuta periódicamente, buscando objetos que ya no estén referenciados.

c.) La destrucción personalizada: finalize

A veces una clase mantiene un recurso que no es de Java como un descriptor de archivo o un tipo de letra del sistema de ventanas. En este caso sería acertado el utilizar la finalización explícita, para asegurar que dicho recurso se libera. Esto se hace mediante la *destrucción personalizada,* un sistema similar a los destructores de C++.

Para especificar una *destrucción personalizada* se añade un método a la clase con el nombre *finalize:*

```
class ClaseFinalizada{
  ClaseFinalizada() { // Constructor
    // Reserva del recurso no Java o recurso compartido
  }
  protected void finalize() {
    // Liberación del recurso no Java o recurso compartido
  }
}
```

El intérprete de Java llama al método *finalize()*, si existe cuando vaya a reclamar el espacio de ese objeto, mediante la recogida de basura.

Debe observarse que el método *finalize()* es de tipo *protected void* y por lo tanto deberá de sobreescribirse con este mismo tipo.

II.6. LA HERENCIA

A. INTRODUCCIÓN

La verdadera potencia de la programación orientada a objetos radica en su capacidad para reflejar la abstracción que el cerebro humano realiza automáticamente durante el proceso de aprendizaje y el proceso de análisis de información.

Las personas percibimos la realidad como un conjunto de objetos interrelacionados. Dichas interrelaciones, pueden verse como un conjunto de abstracciones y generalizaciones que se han ido asimilando desde la niñez. Así, los defensores de la programación orientada a objetos afirman que esta técnica se adecua mejor al funcionamiento del cerebro humano, al permitir descomponer un problema de cierta magnitud en un conjunto de problemas menores subordinados del primero.

La capacidad de descomponer un problema o concepto en un conjunto de objetos relacionados entre sí, y cuyo comportamiento es fácilmente identificable, puede ser muy útil para el desarrollo de programas informáticos.

B. JERARQUÍA

La herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica; una clase *padre* o *superclase* sobre otras clases *hijas* o *subclases*.

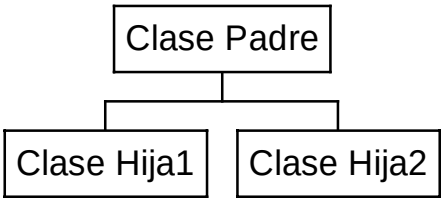

Imagen 4: Ejemplo de otro árbol de herencia

Los descendientes de una clase heredan todas las variables y métodos que sus ascendientes hayan especificado como *heredables*, además de crear los suyos propios.

La característica de herencia, nos permite definir nuevas clases derivadas de otra ya existente, que la especializan de alguna manera. Así logramos definir una jerarquía de clases, que se puede mostrar mediante un árbol de herencia.

En todo lenguaje orientado a objetos existe una jerarquía, mediante la que las clases se relacionan en términos de herencia. En Java, el punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases.

C. HERENCIA MÚLTIPLE

En la orientación a objetos, se consideran dos tipos de herencia, simple y múltiple. En el caso de la primera, una clase sólo puede derivar de una única superclase. Para el segundo tipo, una clase puede descender de varias superclases.

En Java sólo se dispone de herencia simple, para una mayor sencillez del lenguaje, si bien se compensa de cierta manera la inexistencia de herencia múltiple con un concepto denominado *interface*, que estudiaremos más adelante.

D. DECLARACIÓN

Para indicar que una clase deriva de otra, heredando sus propiedades (métodos y atributos), se usa el término *extends*, como en el siguiente ejemplo:

```
public class SubClase extends SuperClase {
  // Contenido de la clase
}
```

Por ejemplo, creamos una clase *MiPunto3D*, hija de la clase ya mostrada *MiPunto*:

```
class MiPunto3D extends MiPunto {
  int z;
  MiPunto3D( ) {
    x = 0; // Heredado de MiPunto
    y = 0; // Heredado de MiPunto
    z = 0; // Nuevo atributo
  }
}
```

La palabra clave *extends* se utiliza para decir que deseamos crear una subclase de la clase que es nombrada a continuación, en nuestro caso *MiPunto3D* es hija de *MiPunto*.

E. LIMITACIONES EN LA HERENCIA

Todos los campos y métodos de una clase son siempre accesibles para el código de la misma clase.

Para controlar el acceso desde otras clases, y para controlar la herencia por las subclase, los miembros (atributos y métodos) de las clases tienen tres modificadores posibles de control de acceso:

- *public*: Los miembros declarados *public* son accesibles en cualquier lugar en que sea accesible la clase, y son heredados por las subclases.
- *private*: Los miembros declarados *private* son accesibles sólo en la propia clase.
- *protected*: Los miembros declarados *protected* son accesibles sólo para sus subclases

Por ejemplo:

```
class Padre { // Hereda de Object
  // Atributos
  private int numeroFavorito, nacidoHace, dineroDisponible;
  // Métodos
  public int getApuesta() {
```

```
    return numeroFavorito;
  }
  protected int getEdad() {
     return nacidoHace;
  }
  private int getSaldo() {
    return dineroDisponible;
  }
}
class Hija extends Padre {
  // Definición
}
class Visita {
  // Definición
}
```

En este ejemplo, un objeto de la clase *Hija*, hereda los tres atributos (*numeroFavorito*, *nacidoHace* y *dineroDisponible*) y los tres métodos (*getApuesta()*, *getEdad()* y *getSaldo()*) de la clase *Padre*, y podrá invocarlos. Cuando se llame al método *getEdad()* de un objeto de la clase *Hija*, se devolverá el valor de la variable de instancia *nacidoHace* de ese objeto, y no de uno de la clase *Padre*.

Sin embargo, un objeto de la clase *Hija*, no podrá invocar al método *getSaldo()* de un objeto de la clase *Padre*, con lo que se evita que el *Hijo* conozca el estado de la cuenta corriente de un *Padre*.

La clase *Visita*, solo podrá acceder al método *getApuesta()*, para averiguar el número favorito de un *Padre*, pero de ninguna manera podrá conocer ni su saldo, ni su edad (sería una indiscreción, ¿no?).

F. LA CLASE OBJECT

La clase *Object* es la superclase de todas las clases da Java. Todas las clases derivan, directa o indirectamente de ella. Si al definir una nueva clase, no aparece la cláusula *extends*, Java considera que dicha clase desciende directamente de *Object*.

La clase *Object* aporta una serie de funciones básicas comunes a todas las clases:

- *public boolean equals(Object obj)*: Se utiliza para comparar, en valor, dos objetos. Devuelve *true* si el objeto que recibe por parámetro es igual, en valor, que el objeto desde el que se llama al método. Si se desean comparar dos referencias a objeto se pueden utilizar los operadores de comparación == y !=.
- *public int hashCode()*: Devuelve un código hash para ese objeto, para poder almacenarlo en una *Hashtable*.
- *protected Object clone() throws CloneNotSupportedException:* Devuelve una copia de ese objeto.

- *public final Class getClass()*: Devuelve el objeto concreto, de tipo *Class*, que representa la clase de ese objeto.
- *protected void finalize() throws Trowable*: Realiza acciones durante la recogida de basura.

Para más información véase **[Arnold y Gosling, 1997]**.

II.7. OPERACIONES AVANZADAS EN LAS CLASES

A. INTRODUCCIÓN

La programación orientada a objetos en Java va mucho más allá de las clases, los objetos y la herencia. Java presenta una serie de capacidades que enriquecen el modelo de objetos que se puede representar en un programa Java. En este capítulo entraremos en ellos.

Vamos a ver cómo programar conceptos avanzados de la herencia, polimorfismo y composición, como conceptos que se pueden programar en Java.

B. OPERACIONES AVANZADAS EN LA HERENCIA

a.) Introducción

En el capítulo anterior ya se han estudiado los fundamentos de la herencia en Java. Sin embargo, el lenguaje tiene muchas más posibilidades en este aspecto, como estudiaremos a continuación.

Conviene recordar que estamos utilizando el código de la clase *MiPunto*, cuyo código se puede encontrar en el apartado *"II.5. Clases y Objetos"* de este tutorial.

b.) Los elementos globales: static

A veces se desea crear un método o una variable que se utiliza fuera del contexto de cualquier instancia, es decir, de una manera global a un programa. Todo lo que se tiene que hacer es declarar estos elementos como *static.*

Esta es la manera que tiene Java de implementar funciones y variables globales.

Por ejemplo:

```
static int a = 3;
static void metodoGlobal() {
// implementación del método
}
```

No se puede hacer referencia a *this* o a *super* dentro de una método *static*.

Mediante atributos estáticos, todas las instancias de una clase además del espacio propio para variables de instancia, comparten un espacio común. Esto es útil para modelizar casos de la vida real.

Otro aspecto en el que es útil *static* es en la creación de métodos a los que se puede llamar directamente diciendo el nombre de la clase en la que están declarados. Se puede llamar a cualquier método *static*, o referirse a cualquier variable *static,* utilizando el operador punto con el nombre de la clase, sin necesidad de crear un objeto de ese tipo:

```
class ClaseStatic {
  int atribNoStatic = 42;
  static int atribStatic = 99;
```

```
  static void metodoStatic() {
    System.out.println("Met. static = " + atribStatic);
  }
  static void metodoNoStatic() {
    System.out.println("Met. no static = " + atribNoStatic);
  }
}
```

El siguiente código es capaz de llamar a *metodoStatic* y *atribStatic* nombrando directamente la clase (sin objeto, sin *new*), por haber sido declarados *static*.

```
System.out.println("At. static = " + ClaseStatic.atribStatic);
ClaseStatic.metodoStatic(); // Sin instancia
new ClaseStatic().metodoNoStatic(); // Hace falta instancia
```

Si ejecutamos este programa obtendríamos:

```
At. static = 99
Met. static = 99
Met. no static = 42
```

Debe tenerse en cuenta que en un método estático tan sólo puede hacerse refernecia a variables estáticas.

c.) Las clases y métodos abstractos: abstract

Hay situaciones en las que se necesita definir una clase que represente un concepto abstracto, y por lo tanto no se pueda proporcionar una implementación completa de algunos de sus métodos.

Se puede declarar que ciertos métodos han de ser sobrescritos en las subclases, utilizando el modificador de tipo *abstract.* A estos métodos también se les llama *responsabilidad de subclase*. Cualquier subclase de una clase *abstract* debe implementar todos los métodos *abstract* de la superclase o bien ser declarada también como *abstract.*

Cualquier clase que contenga métodos declarados como *abstract* también se tiene que declarar como *abstract*, y no se podrán crear instancias de dicha clase (operador *new*).

Por último se pueden declarar constructores *abstract* o métodos *abstract static*.

Veamos un ejemplo de clases abstractas:

```
abstract class claseA {
  abstract void metodoAbstracto();
  void metodoConcreto() {
    System.out.println("En el metodo concreto de claseA");
  }
}
class claseB extends claseA {
  void metodoAbstracto(){
```

```
    System.out.println("En el metodo abstracto de claseB");
  }
}
```

La clase abstracta *claseA* ha implementado el método concreto *metodoConcreto()*, pero el método *metodoAbstracto()* era abstracto y por eso ha tenido que ser redefinido en la clase hija *claseB*.

```
claseA referenciaA = new claseB();
referenciaA.metodoAbstracto();
referenciaA.metodoConcreto();
```

La salida de la ejecución del programa es:

```
En el metodo abstracto de claseB
En el metodo concreto de claseA
```

C. EL POLIMORFISMO

a.) Selección dinámica de método

Las dos clases implementadas a continuación tienen una relación subclase/superclase simple con un único método que se sobrescribe en la subclase:

```
class claseAA {
  void metodoDinamico() {
    System.out.println("En el metodo dinamico de claseAA");
  }
}
class claseBB extends claseAA {
  void metodoDinamico() {
    System.out.println("En el metodo dinamico de claseBB");
  }
}
```

Por lo tanto si ejecutamos:

```
claseAA referenciaAA = new claseBB();
referenciaAA.metodoDinamico();
```

La salida de este programa es:

```
En el metodo dinamico de claseBB
```

Se declara la variable de tipo *claseA*, y después se almacena una referencia a una instancia de la clase *claseB* en ella. Al llamar al método *metodoDinamico()* de *claseA*, el compilador de Java verifica que *claseA* tiene un método llamado *metodoDinamico()*, pero el intérprete de Java observa que la referencia es realmente una instancia de *claseB*, por lo que llama al método *metodoDinamico()* de *claseB* en vez de al de *claseA*.

Esta forma de *polimorfismo dinámico en tiempo de ejecución* es uno de los mecanismos más poderosos que ofrece el diseño orientado a objetos para soportar la reutilización del código y la robustez.

b.) Sobrescritura de un método

Durante una jerarquía de herencia puede interesar volver a escribir el cuerpo de un método, para realizar una funcionalidad de diferente manera dependiendo del nivel de abstracción en que nos encontremos. A esta modificación de funcionalidad se le llama sobrescritura de un método.

Por ejemplo, en una herencia entre una clase *SerVivo* y una clase hija *Persona*; si la clase *SerVivo* tuviese un método *alimentarse()*, debería volver a escribirse en el nivel de *Persona*, puesto que una persona no se alimenta ni como un *Animal*, ni como una *Planta*...

La mejor manera de observar la diferencia entre sobrescritura y sobrecarga es mediante un ejemplo. A continuación se puede observar la implementación de la sobrecarga de la distancia en 3D y la sobrescritura de la distancia en 2D.

```
class MiPunto3D extends MiPunto {
  int x,y,z;
  double distancia(int pX, int pY) { // Sobrescritura
    int retorno=0;
    retorno += ((x/z)-pX)*((x/z)-pX);
    retorno += ((y/z)-pY)*((y/z)-pY);
    return Math.sqrt( retorno );
  }
}
```

Se inician los objetos mediante las sentencias:

```
MiPunto p3 = new MiPunto(1,1);
MiPunto p4 = new MiPunto3D(2,2);
```

Y llamando a los métodos de la siguiente forma:

```
p3.distancia(3,3);     //Método MiPunto.distancia(pX,pY)
p4.distancia(4,4);     //Método MiPunto3D.distancia(pX,pY)
```

Los métodos se seleccionan en función del tipo de la instancia en tiempo de ejecución, no a la clase en la cual se está ejecutando el método actual. A esto se le llama *selección dinámica de método*.

c.) Sobrecarga de método

Es posible que necesitemos crear más de un método con el mismo nombre, pero con listas de parámetros distintas. A esto se le llama *sobrecarga del método*. La sobrecarga de método se utiliza para proporcionar a Java un comportamiento *polimórfico.*

Un ejemplo de uso de la sobrecarga es por ejemplo, el crear constructores alternativos en función de las coordenadas, tal y como se hacía en la clase *MiPunto*:

```
MiPunto( ) { //Constructor por defecto
```

```
  inicia( -1, -1 );
}
MiPunto( int paramX, int paramY ) { // Parametrizado
  this.x = paramX;
  y = paramY;
}
```

Se llama a los constructores basándose en el número y tipo de parámetros que se les pase. Al número de parámetros con tipo de una secuencia específica se le llama *signatura de tipo*. Java utiliza estas signaturas de tipo para decidir a qué método llamar. Para distinguir entre dos métodos, no se consideran los nombres de los parámetros formales sino sus tipos:

```
MiPunto p1 = new MiPunto(); // Constructor por defecto
MiPunto p2 = new MiPunto( 5, 6 ); // Constructor parametrizado
```

d.) Limitación de la sobreescritura: final

Todos los métodos y las variables de instancia se pueden sobrescribir por defecto. Si se desea declarar que ya no se quiere permitir que las subclases sobrescriban las variables o métodos, éstos se pueden declarar como final. Esto se utiliza a menudo para crear el equivalente de una constante de C++.

Es un convenio de codificación habitual elegir identificadores en mayúsculas para las variables que sean *final,* por ejemplo:

```
final int NUEVO_ARCHIVO = 1;
```

D. LAS REFERENCIAS POLIMÓRFICAS: THIS Y SUPER

b.) Acceso a la propia clase: this

Aunque ya se explicó en el apartado *"II.5. Clases y objetos"* de este tutorial el uso de la referencia *this* como modificador de ámbito, también se la puede nombrar como ejemplo de polimorfismo

Además de hacer continua referencia a la clase en la que se invoque, también vale para sustituir a sus constructores, utilizándola como método:

```
this(); // Constructor por defecto
this( int paramX, int paramY ); // Constructor parametrizado
```

b.) Acceso a la superclase: super

Ya hemos visto el funcionamiento de la referencia *this* como referencia de un objeto hacia sí mismo. En Java existe otra referencia llamada *super*, que se refiere directamente a la superclase.

La referencia *super* usa para acceder a métodos o atributos de la superclase.

Podíamos haber implementado el constructor de la clase *MiPunto3D* (hija de *MiPunto*) de la siguiente forma:

```
MiPunto3D( int x, int y, int z ) {
```

```
    super( x, y ); // Aquí se llama al constructor de MiPunto
    this.z = super.metodoSuma( x, y ); // Método de la superclase
  }
```

Con una sentencia *super.metodoSuma(x, y)* se llamaría al método *metodoSuma()* de la superclase de la instancia *this*. Por el contrario con *super()* llamamos al constructor de la superclase.

E. LA COMPOSICIÓN

Otro tipo de relación muy habitual en los diseños de los programas es la composición. Los objetos suelen estar compuestos de conjuntos de objetos más pequeños; un coche es un conjunto de motor y carrocería, un motor es un conjunto de piezas, y así sucesivamente. Este concepto es lo que se conoce como *composición*.

La forma de implementar una relación de composición en Java es incluyendo una referencia a objeto de la clase componedora en la clase compuesta.

Por ejemplo, una clase *AreaRectangular*, quedaría definida por dos objetos de la clase *MiPunto*, que representasen dos puntas contrarias de un rectángulo:

```
class AreaRectangular {
  MiPunto extremo1; //extremo inferior izquierdo
  MiPunto extremo2; //extremo superior derecho
  AreaRectangular() {
    extremo1=new MiPunto();
    extremo2=new MiPunto();
  }
  boolean estaEnElArea( MiPunto p ){
    if ( ( p.x>=extremo1.x && p.x<=extremo2.x ) &&
         ( p.y>=extremo1.y && p.y<=extremo2.y )  )
      return true;
    else
      return false;
  }
}
```

Puede observarse que las referencias a objeto (*extremo1* y *extremo2*) son iniciadas, instanciando un objeto para cada una en el constructor. Así esta clase mediante dos puntos, referenciados por *extremo1* y *extremo2*, establece unos límites de su área, que serán utilizados para comprobar si un punto está en su área en el método *estaEnElArea()*.

II.8. GESTIÓN DE EXCEPCIONES Y ERRORES

A. INTRODUCCIÓN

El control de flujo en un programa Java puede hacerse mediante las ya conocidas sentencias estructuradas (*if, while, return*). Pero Java va mucho más allá, mediante una técnica de programación denominada *gestión de excepciones*.

Mediante las excepciones se podrá evitar repetir continuamente código, en busca de un posible error, y avisar a otros objetos de una condición anormal de ejecución durante un programa.

Durante este capítulo estudiaremos la gestión de excepciones y errores, sin pretender profundizar demasiado, pero sí fijando la base conceptual de lo que este modo de programación supone.

Mediante la gestión de excepciones se prescindirá de sentencias de control de errores del tipo:

```
if ( error == true )
  return  ERROR;
```

B. TIPOS DE EXCEPCIONES

Existen varios tipos fundamentales de excepciones:

- Error: Excepciones que indican problemas muy graves, que suelen ser no recuperables y no deben casi nunca ser capturadas.
- *Exception*: Excepciones no definitivas, pero que se detectan fuera del tiempo de ejecución.
- *RuntimeException*: Excepciones que se dan durante la ejecución del programa.

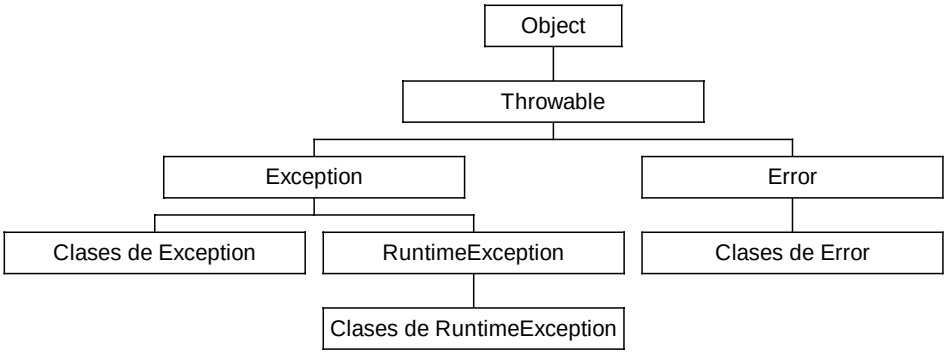

Imagen 5: Herencia de excepciones Java

Todas las excepciones tienen como clase base la clase *Throwable*, que está incluida en el paquete *java.lang*, y sus métodos son:

- *Trowable(String mensaje);* Constructor. La cadena es opcional
- *Throwable fillInStackTrace();* Llena la pila de traza de ejecución.
- *String getLocalizedMessage();* Crea una descripción local de este objeto.

- *String getMessage();* Devuelve la cadena de error del objeto.
- *void printStackTrace(PrintStream_o_PrintWriter s);* Imprime este objeto y su traza en el flujo del parámetro s, o en la salida estándar (por defecto).
- *String toString();* Devuelve una breve descripción del objeto.

C. FUNCIONAMIENTO

a.) Introducción

Para que el sistema de gestión de excepciones funcione, se ha de trabajar en dos partes de los programas:

- Definir qué partes de los programas crean una excepción y bajo qué condiciones. Para ello se utilizan las palabras reservadas throw y throws.
- Comprobar en ciertas partes de los programas si una excepción se ha producido, y actuar en consecuencia. Para ello se utilizan las palabras reservadas try, catch y finally.

b.) Manejo de excepciones: try - catch - finally

Cuando el programador va a ejecutar un trozo de código que pueda provocar una excepción (por ejemplo, una lectura en un fichero), debe incluir este fragmento de código dentro de un bloque *try*:

```
try {
  // Código posiblemente problemático
}
```

Pero lo importante es cómo controlar qué hacer con la posible excepción que se cree. Para ello se utilizan las clausulas *catch*, en las que se especifica que acción realizar:

```
try {
  // Código posiblemente problemático
} catch( tipo_de_excepcion e) {
  // Código para solucionar la excepción e
} catch( tipo_de_excepcion_mas_general e) {
  // Código para solucionar la excepción e
}
```

En el ejemplo se observa que se pueden anidar sentencias *catch*, pero conviene hacerlo indicando en último lugar las excepciones más generales (es decir, que se encuentren más arriba en el árbol de herencia de excepciones), porque el intérprete Java ejecutará aquel bloque de código *catch* cuyo parámetro sea del tipo de una excepción lanzada.

Si por ejemplo se intentase capturar primero una excepción *Throwable*, nunca llegaríamos a gestionar una excepción *Runtime*, puesto que cualquier clase hija de *Runtime* es también hija de *Throwable*, por herencia.

Si no se ha lanzado ninguna excepción el código continúa sin ejecutar ninguna sentencia *catch*.

Pero, ¿y si quiero realizar una acción común a todas las opciones?. Para insertar fragmentos de código que se ejecuten tras la gestión de las excepciones. Este código se ejecutará tanto si se ha tratado una excepción (*catch*) como sino. Este tipo de código se inserta en una sentencia *finally*, que será ejecutada tras el bloque *try* o *catch:*

```
try {
} catch( Exception e ) {
} finally {
  // Se ejecutara tras try o catch
}
```

c.) Lanzamiento de excepciones: throw - throws

Muchas veces el programador dentro de un determinado método deberá comprobar si alguna condición de excepción se cumple, y si es así lanzarla. Para ello se utilizan las palabras reservadas *throw* y *throws*.

Por una parte la excepción se lanza mediante la sentencia *throw*:

```
if ( condicion_de_excepcion == true )
  throw new miExcepcion();
```

Se puede observar que hemos creado un objeto de la clase *miExcepcion*, puesto que las excepciones son objetos y por tanto deberán ser instanciadas antes de ser lanzadas.

Aquellos métodos que pueden lanzar excepciones, deben cuáles son esas excepciones en su declaración. Para ello se utiliza la sentencia *throws*:

```
tipo_devuelto miMetodoLanzador() throws miExcep1, miExcep2 {
  // Codigo capaz de lanzar excepciones miExcep1 y miExcep2
}
```

Se puede observar que cuando se pueden lanzar en el método más de una excepción se deben indicar en su declaración separadas por comas.

d.) Ejemplo de gestión de excepciones

Ahora que ya sabemos cómo funciona este sistema, conviene ver al menos un pequeño ejemplo, que ilustre al lector en el uso de las excepciones:

```
// Creo una excepción personalizada
class MiExcepcion extends Exception {
  MiExcepcion(){
    super(); // constructor por defecto de Exception
  }
  MiExcepcion( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}
// Esta clase lanzará la excepción
```

```
class Lanzadora {
  void lanzaSiNegativo( int param ) throws MiExcepcion {
    if ( param < 0 )
      throw new MiExcepcion( "Numero negativo" );
  }
}
class Excepciones {
  public static void main( String[] args ) {
    // Para leer un fichero
    Lanzadora lanza = new Lanzadora();
    FileInputStream entrada = null;
    int leo;
    try {
      entrada = new FileInputStream( "fich.txt" );
      while ( ( leo = entrada.read() ) != -1 )
        lanza.lanzaSiNegativo( leo );
      entrada.close();
      System.out.println( "Todo fue bien" );
    } catch ( MiExcepcion e ){ // Personalizada
      System.out.println( "Excepcion: " + e.getMessage() );
    } catch ( IOException e ){ // Estándar
      System.out.println( "Excepcion: " + e.getMessage() );
    } finally {
      if ( entrada != null )
        try {
          entrada.close(); // Siempre queda cerrado
        } catch ( Exception e ) {
        System.out.println( "Excepcion: " + e.getMessage() );
        }
      System.out.println( "Fichero cerrado." );
    }
  }
}
class Excepciones {
  public static void main( String[] args ) {
    // Para leer un fichero
    FileInputStream entrada = null;
```

```
    Lanzadora lanza = new Lanzadora();
     int leo;
    try {
      entrada = new FileInputStream("fich.txt");
      while ( ( leo = entrada.read() ) != -1 )
        lanza.lanzaSiNegativo( leo );
      System.out.println( "Todo fue bien" );
    } catch ( MiExcepcion e ){ // Personalizada
      System.out.println( "Excepcion: " + e.getMessage() );
    } catch ( IOException e ){ // Estándar
      System.out.println( "Excepcion: " + e.getMessage() );
    } finally {
      entrada.close(); // Así el fichero siempre queda cerrado
      System.out.println( "Fichero cerrado" );
    }
  }
}
```

Este programa lee un fichero (*fichero.txt*), y lee su contenido en forma de números.

Si alguno de los números leídos es negativo, lanza una excepción *MiExcepcion*, Además gestiona la excepción *IOException*, que es una excepción de las que Java incluye y que se lanza si hay algún problema en una operación de entrada/salida.

Ambas excepciones son gestionadas, imprimiendo su contenido (cadena de error) por pantalla.

La salida de este programa, suponiendo un número negativo sería:

```
Excepcion: Numero negativo
Fichero cerrado
```

En el caso de que no hubiera ningún número negativo sería:

```
Todo fue bien
Fichero cerrado
```

En el caso de que se produjese un error de E/S, al leer el primer número, sería:

```
Excepcion: java.io.IOException
Fichero cerrado
```

e.) Conclusiones

En cualquier caso se recomienda al programador no abusar de este sistema como control de flujos simples, sino utilizarlo sólo en aquellos estados del programa que realmente creen un problema de ejecución que pueda ser letal para el programa.

Para más información sobre las excepciones Java, véanse **[Zolli, 1997]** y **[Naughton, 1996].**

D. EXCEPCIONES QUE INCORPORA JAVA 1.2

a.) Clases de Error

LinkageError: Una clase no satisface la dependencia que tiene respecto a otra.

- *ClassCircularityError*: Se detectó una herencia circular entre clases.
- *ClassFormatError*: Una clase cargada no ha sido incompletamente descrita.
  - *UnsupportedClassVersionError*: La versión de una clase no es correcta.
- *ExceptionInInitializerError*: Error al iniciar un miembro static.
- *IncompatibleClassChangeError*: En una clase, su interfaz no es igual al declarado
  - *AbstractMethodError*: Se ha invocado un método abstracto.
  - *IllegalAccessError*: La aplicación intentó acceder a algún miembro no visible.
  - *InstantiationError*: Se intentó instanciar una clase abstracta o interfaz.
  - *NoSuchFieldError*: No se encontró determinado atributo.
  - *NoSuchMethodError*: No se encontró determinado método.
- *NoClassDefFoundError*: No se encontró una clase cuando se necesitaba.
- *UnsatisfiedLinkError*: Se encontró un enlace insatisfecho en un método nativo.
- *VerifyError*: Se ha producido un error de verificación al cargar una clase.

ThreadDeath: Se ha lanzado en el *thread* víctima tras llamar a *stop()*.

VirtualMachineError: La máquina virtual se ha averiado o quedado sin recursos.

- *InternalError*: Error interno en tiempo de ejecución.
- *OutOfMemoryError*: El lector ha agotado la memoria.
- *StackOverflowError*: Desbordamiento de pila. ¿Recursión infinita?.
- *UnknownError*: Grave error desconocido.

b.) Clases de Exception

CloneNotSupportedException: No se pudo copiar un objeto mediante clone().

IllegalAccessException: Algún método invocado es no visible.

InstantiationException: Se ha intentado instanciar una interfaz o una clase abstracta.

InterruptedException: Cuando se invoca a *interrupt()* sobre un *thread* dormido.

NoSuchFieldException: La clase no tiene un atributo con ese nombre.

NoSuchMethodException: La clase no tiene un método con ese nombre.

c.) Clases de RuntimeException

ArithmeticException: Error de cálculo (como división por cero...).

ArrayStoreException: Intento de almacenar un objeto equivocado en un vector.

ClassCastException: Intento de conversión inválida.

IllegalArgumentException: Se ha pasado un argumento inválido a un método:

- *IllegalThreadStateException*: Un thread no estaba en el estado adecuado.
- *NumberFormatException*: Una cadena contenedora de un número, no lo contiene.

IllegalMonitorStateException: Se ha usado *wait/notify* fuera de código sincronizado.

IllegalStateException: Método invocado en un momento inapropiado.

IndexOutOfBoundsException: Acceso a un vector fuera de sus límites:

- *ArrayIndexOutOfBoundsException*: Idem, para una matriz.
- *StringIndexOutOfBoundsException*: Idem, para una cadena.

NegativeArraySizeException: Intento de creación de un vector de tamaño negativo.

NullPointerException: Se ha usado una referencia *null* para acceder a un campo.

SecurityException: Algo ha sido vedado por el sistema de seguridad.

UnsupportedOperationException: Una operación invocada no se soporta.

Para más información véase la documentación del JDK que usted vaya a utilizar.

II.9. INTERFACES

A. INTRODUCCIÓN

Las interfaces Java son expresiones puras de diseño. Se trata de auténticas conceptualizaciones no implementadas que sirven de guía para definir un determinado concepto (clase) y lo que debe hacer, pero sin desarrollar un mecanismo de solución.

Se trata de declarar métodos abstractos y constantes que posteriormente puedan ser implementados de diferentes maneras según las necesidades de un programa.

Por ejemplo una misma interfaz podría ser implementada en una versión de prueba de manera poco óptima, y ser acelerada convenientemente en la versión definitiva tras conocer más a fondo el problema.

B. DECLARACIÓN

Para declarar una interfaz se utiliza la sentencia *interface*, de la misma manera que se usa la sentencia *class*:

```
interface MiInterfaz {
  int CONSTANTE = 100;
  int metodoAbstracto( int parametro );
}
```

Se observa en la declaración que las variables adoptan la declaración en mayúsculas, pues en realidad actuarán como constantes *final*. En ningún caso estas variables actuarán como variables de instancia.

Por su parte, los métodos tras su declaración presentan un punto y coma, en lugar de su cuerpo entre llaves. Son métodos abstractos, por tanto, métodos sin implementación

C. IMPLEMENTACIÓN DE UNA INTERFAZ

Como ya se ha visto, las interfaces carecen de funcionalidad por no estar implementados sus métodos, por lo que se necesita algún mecanismo para dar cuerpo a sus métodos.

La palabra reservada *implements* utilizada en la declaración de una clase indica que la clase implementa la interfaz, es decir, que asume las constantes de la interfaz, y codifica sus métodos:

```
class ImplementaInterfaz implements MiInterfaz{
  int multiplicando=CONSTANTE;
  int metodoAbstracto( int parametro ){
     return ( parametro * multiplicando );
  }
}
```

En este ejemplo se observa que han de codificarse todos los métodos que determina la interfaz (m*etodoAbstracto()*), y la validez de las constantes (*CONSTANTE*) que define la interfaz durante toda la declaración de la clase.

Una interfaz no puede implementar otra interfaz, aunque sí extenderla (*extends*) ampliándola.

D. HERENCIA MÚLTIPLE

Java es un lenguaje que incorpora herencia simple de implementación pero que puede aportar herencia múltiple de interfaz. Esto posibilita la herencia múltiple en el diseño de los programas Java.

Una interfaz puede heredar de más de una interfaz antecesora.

```
interface InterfazMultiple extends Interfaz1,Interfaz2{ }
```

Una clase no puede tener más que una clase antecesora, pero puede implementar más de una interfaz:

```
class MiClase extends SuPadre implements Interfaz1,Interfaz2{ }
```

El ejemplo típico de herencia múltiple es el que se presenta con la herencia en diamante:

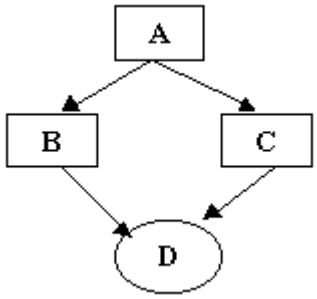

Imagen 6: Ejemplo de herencia múltiple

Para poder llevar a cabo un esquema como el anterior en Java es necesario que las clases A, B y C de la figura sean interfaces, y que la clase D sea una clase (que recibe la herencia múltiple):

```
interface A{ }
interface B extends A{ }
interface C extends A{ }
class D implements B,C{ }
```

E. COLISIONES EN LA HERENCIA MÚLTIPLE

En una herencia múltiple, los identificadores de algunos métodos o atributos pueden coincidir en la clase que hereda, si dos de las interfaces padres tienen algún método o atributo que coincida en nombre. A esto se le llama *colisión*.

Esto se dará cuando las clases padre (en el ejemplo anterior *B* y *C*) tienen un atributo o método que se llame igual. Java resuelve el problema estableciendo una serie de reglas.

Para la colisión de nombres de atributos, se obliga a especificar a qué interfaz base pertenecen al utilizarlos.

Para la colisión de nombres en métodos:

- Si tienen el mismo nombre y diferentes parámetros: se produce sobrecarga de métodos permitiendo que existan varias maneras de llamar al mismo.
- Si sólo cambia el valor devuelto: se da un error de compilación, indicando que no se pueden implementar los dos.
- Si coinciden en su declaración: se elimina uno de los dos, con lo que sólo queda uno.

F. ENVOLTURAS DE LOS TIPOS SIMPLES

Los tipos de datos de Java no forman parte de la jerarquía de objetos. Sin embargo a veces es necesario crear una representación como objeto de alguno de los tipos de datos simples de Java.

La API de Java contiene un conjunto de interfaces especiales para modificar el comportamiento de los tipos de datos simple. A estas interfaces se las conoce como *envolturas de tipo simple*.

Todas ellas son hijas de la clase abstracta *Number* y son:

- *Double*: Da soporte al tipo double.
- *Float*: Da soporte al tipo float.
- *Integer*: Da soporte a los tipos int, short y byte.
- *Long*: Da soporte al tipo long.
- *Character*: Envoltura del tipo char.
- *Boolean*: Envoltorio al tipo boolean.

Para más información sobre as envolturas de tipos simples, consúltese **[Naughton, 1996]**.

II.10. PAQUETES

A. INTRODUCCIÓN

Los paquetes son el mecanismo por el que Java permite agrupar clases, interfaces, excepciones y constantes. De esta forma, se agrupan conjuntos de estructuras de datos y de clases con algún tipo de relación en común.

Con la idea de mantener la reutilización y facilidad de uso de los paquetes desarrollados es conveniente que las clases e interfaces contenidas en los mismos tengan cierta relación funcional. De esta manera los desarrolladores ya tendrán una idea de lo que están buscando y fácilmente sabrán qué pueden encontrar dentro de un paquete.

B. CREACIÓN DE UN PAQUETE

a.) Declaración

Para declarar un paquete se utiliza la sentencia *package* seguida del nombre del paquete que estemos creando:

```
package NombrePaquete;
```

La estructura que ha de seguir un fichero fuente en Java es:

- Una única sentencia de paquete (opcional).
- Las sentencias de importación deseadas (opcional).
- La declaración de una (y sólo una) clase pública (*public*).
- Las clases privadas del paquete (opcional).

Por lo tanto la sentencia de declaración de paquete ha de ser la primera en un archivo fuente Java.

b.) Nomenclatura

Para que los nombres de paquete puedan ser fácilmente reutilizados en toda una compañía o incluso en todo el mundo es conveniente darles nombres únicos. Esto puede ser una tarea realmente tediosa dentro de una gran empresa, y absolutamente imposible dentro de la comunidad de Internet.

Por eso se propone asignar como paquetes y subpaquetes el nombre de dominio dentro de Internet. Se verá un ejemplo para un dominio que se llamase *japon.magic.com,* un nombre apropiado sería *com.magic.japon.paquete*.

c.) Subpaquetes

Cada paquete puede tener a su vez paquetes con contenidos parecidos, de forma que un programador probablemente estará interesado en organizar sus paquetes de forma jerárquica. Para eso se definen los *subpaquetes.*

Para crear un subpaquete bastará con almacenar el paquete hijo en un directorio *Paquete/Subpaquete.*

Así una clase dentro de un *subpaquete* como *Paquete.Subpaquete.clase* estará codificada en el fichero *Paquete/Subpaquete.java*.

El JDK define una variable de entorno denominada *CLASSPATH* que gestiona las rutas en las que el JDK busca los subpaquetes. El directorio actual suele estar siempre incluido en la variable de entorno *CLASSPATH*.

Para más información sobre el JDK véase el *"Apéndice I. JDK"* de este tutorial.

C. USO DE UN PAQUETE

Con el fin de importar paquetes ya desarrollados se utiliza la sentencia *import* seguida del nombre de paquete o paquetes a importar.

Se pueden importar todos los elementos de un paquete o sólo algunos.

Para importar todas las clases e interfaces de un paquete se utiliza el metacaracter *:

```
import PaquetePrueba.*;
```

También existe la posibilidad de que se deseen importar sólo algunas de las clases de un cierto paquete o subpaquete:

```
import Paquete.Subpaquete1.Subpaquete2.Clase1;
```

Para acceder a los elementos de un paquete, no es necesario importar explícitamente el paquete en que aparecen, sino que basta con referenciar el elemento tras una especificación completa de la ruta de paquetes y subpaquetes en que se encuentra.

```
Paquete.Subpaquetes1.Subpaquete2.Clase_o_Interfaz.elemento
```

En la API de Java se incluyen un conjunto de paquetes ya desarrollados que se pueden incluir en cualquier aplicación (o *applet*) Java que se desarrolle. Estos paquetes son explicados con más detalle en el capítulo "*III.1.Bibliotecas de la API de Java*" de este tutorial.

D. ÁMBITO DE LOS ELEMENTOS DE UN PAQUETE

Al introducir el concepto de paquete, surge la duda de cómo proteger los elementos de una clase, qué visibilidad presentan respecto al resto de elementos del paquete, respecto a los de otros paquetes...

Ya en la herencia se vieron los identificadores de visibilidad *public* (visible a todas las clases), *private* (no visible más que para la propia clase), y *protected* (visible a clases hijas).

Por defecto se considera los elementos (clases, variables y métodos) de un mismo paquete como visibles entre ellos (supliendo las denominadas *clases amigas* de C++).

Situación del elemento	*private*	*sin modificador*	*protected*	*public*
En la misma clase	Sí	Sí	Sí	Sí
En una clase en el mismo paquete	No	Sí	Sí	Sí
En una clase hija en otro paquete	No	No	Sí	Sí
En una clase no hija en otro paquete	No	No	No	Sí

Tabla 19: Visibilidad dentro de un paquete

Todas las reglas explicadas en este apartado son igualmente válidas para las interfaces Java.

Para más información véase **[Naughton, 1996].**

II.11. LOS THREADS O PROGRAMACIÓN MULTIHILO

A. INTRODUCCIÓN

Durante la ejecución de los programas existen muchas operaciones que precisan de una espera; en busca de una interacción con el exterior, dejando pasar el tiempo, esperando a que otro proceso acabe...

Java permite que estos tiempos desaprovechados sean utilizados por el programador para realizar determinadas tareas, y así aprovechar el microprocesador durante toda la ejecución del programa. Para ello implementa el concepto de *threads*, o hilos de control del programa.

Mediante el uso de varios *threads*, se consigue ejecutar varios procesos en paralelo, de forma que cuando uno de ellos esté esperando algún evento, permita que el microprocesador ejecute alguno de los otros *threads* en espera. Cuando el evento que el primer *thread* esperaba sucede, de nuevo se intercambian los *threads* para que el primer *thread* continúe su ejecución.

Todo esto viene a suplir a la técnica de exclusión mutua denominada *utilización de semáforos*, extendida entre los programadores de C en UNIX.

B. UTILIZACIÓN DE THREAD

Para crear un *thread*, se ha de implementar una clase, extendiendo la clase base *Runnable*, y crear un objeto de la clase *Thread*. Este objeto representará un nuevo hilo de control, que será accionado cuando invoquemos al método *start()* del *thread*. En ese momento este hilo se activará, ejecutando (si el planificador de hilos considera que es el momento), el método *run()* de la clase en que todo esto suceda.

Por ejemplo, el siguiente programa utiliza dos hilos, el hilo general *main*, y el hilo *thDemo* que creamos:

```
import java.io.*;
import java.net.*;
class ThreadDemo implements Runnable {
  ThreadDemo() {
    Thread thDemo = new Thread( this, "ThDemo" );
    thDemo.start();
  };
  public void run() {
    try {
      Thread.sleep(3000);
    } catch( InterruptedException e ) { };
    System.out.println("Saliendo del hilo hijo");
  };
public static void main( String args[] ){
```

```
    new ThreadDemo();
    try {
      for ( int i = 5 ; i >0 ; i-- ) {
        System.out.println(" Seg: " + i );
        Thread.sleep(1000);
      }
    } catch( InterruptedException e ) { };
    System.out.println("Saliendo del main");
  };
};
```

Ambos hilos esperan utilizando el método *sleep()* de la clase *Thread*; *thDemo* tres segundos, y *main* cinco segundos. Java utilizará los tres segundos de *thDemo* para ir esperando los tres primeros segundos del hilo *main*.

Por lo tanto la salida por pantalla al ejecutar el programa es:

```
prompt> java ThreadDemo
Seg: 5
Seg: 4
Seg: 3
Saliendo del hilo hijo
Seg: 2
Seg: 1
Saliendo del hilo main
```

C. SINCRONIZACIÓN DE THREADS

Durante la ejecución de un programa, muchas veces varios procesos han de realizar tareas de una forma sincronizada, actuando en un determinado orden. Para ello en Java se utilizan la palabra reservada *syncronized*, en la declaración de los procesos con este tipo de características.

Los procesos declarados como *syncronized* mediante la utilización de excepciones, y de las funciones *wait()* y *notifiy()*, respectivamente esperarán a que otro proceso acabe antes de continuar su ejecución.

A continuación se va a ir viendo cómo implementar el clásico problema de exclusión mutua conocido como *el problema del productor/consumidor*, en el que dos procesos han de acceder a una cola común, en la que el proceso *productor* inserta elementos en la pila, y el proceso *consumidor* ha de ir consumiendo los elementos en la pila, cada vez que sean insertados:

```
class ColaSincronizada {
  int n;
  boolean bandera = false;
  synchronized int obten() {
```

```
    if ( !bandera )
       try wait(); catch( InterruptedException e );
    System.out.println( "Obtenido: " + n );
    bandera = false;
    notify();
    return n;
  }
  synchronized void coloca( int paramN ) {
    if ( bandera )
      try wait(); catch( InterruptedException e );
    n = paramN;
    bandera =true;
    System.out.println( "Colocado: " + n );
    notify();
  }
}

class Productor implements Runnable {
  ColaSincronizada colaProductor;
  Productor( ColaSincronizada paramCola ) {
    colaProductor = paramCola;
    new Thread( this, "Producer" ).start();
  }
  public void run() {
    int i = 0;
    while ( true ) // Bucle infinito
      colaProductor.coloca( i++ );
  }
}

class Consumidor implements Runnable {
  ColaSincronizada colaConsumidor;
  Consumidor( ColaSincronizada paramCola ) {
    colaConsumidor = paramCola;
    new Thread( this, "Consumidor" ).start();
  }
  public void run() {
```

```
    while ( true ) // Bucle infinito
      colaConsumidor.obten( );
  }
}

public static void main( String args[] ) {
  ColaSincronizada colaLocal = new ColaSincronizada();
  new Productor( colaLocal );
  new Consumidor( colaLocal );
}
```

La salida del programa será:

```
Colocado: 1
Obtenido: 1
Colocado: 2
Obtenido: 2
Colocado: 3
Obtenido: 3
. . .
```

D. Y MUCHO MÁS

La utilización de programación concurrente y de los hilos de Java con toda su potencia va mucho más allá de los objetivos de este tutorial. Lo que aquí se ha visto es simplemente una introducción para que el lector sea consciente de cuál es la potencia de este tipo de programación.

La utilización de los *threads* se extiende con métodos para que el programador controle la alternancia de los hilos. Estos métodos son:

- *suspend();* Bloquea temporalmente la ejecución de un hilo.
- *resume();* Activa un hilo bloqueado.
- *stop();* Finaliza la ejecución de un hilo.

Para más información sobre los *threads* véase **[Zolli, 1997]**.

II.12 CREACIÓN DE PROGRAMAS JAVA

A. INTRODUCCIÓN

Una vez entendida la sintaxis y funcionalidad de Java, hay que explicar cómo combinar todos los elementos introducidos para desarrollar programas Java.

La programación Java va mucho más allá de lo que la definición del lenguaje permite. Son muchos los tipos de aplicaciones que se pueden crear con Java, así como su utilización, especialmente gracias a la versatilidad de las bibliotecas de clases que completan el Java básico.

B. TIPOS DE APLICACIONES

a.) Introducción

Con Java se pueden construir varios tipos de programas, cada uno con unas características específicas, y que se ejecutan de distintas maneras.

A continuación se explican los principales tipos: Aplicaciones, *Applets*, *JavaBeans*, *JavaScript* y *Servlets*.

Para más información véase **[Morgan, 1999]** y **[Zolli, 1997]**.

b.) Aplicaciones

Son los programas básicos de Java. Se ejecutan en una determinada máquina, por el Java Runtime Enviroment (JRE).

Para crear una aplicación hace falta incluir en alguna de las clases que compongan la aplicación un método denominado:

- *public static void main(String[] s){ }*

Hay que indicar al JRE (comando *java* del JDK) el nombre de la clase (previamente compilada a *.class*), que queremos ejecutar. Cuando se ejecute el programa lo que realmente se ejecutará será el método *main()* de la clase indicada al JRE.

Las aplicaciones soportan métodos nativos, o sea, incluir en el programa código escrito en otros lenguajes de programación, así como violar algunas de las directrices de seguridad.

En cada fichero Java (*.java*) sólo debe haber una clase pública.

c.) Applets

Las *applets* o miniaplicaciones Java, son programas que deben incluirse en páginas Web para ser observadas por otra aplicación (visualizador de applets o navegador Web), y que se ejecutan cuando el usuario intenta visualizarlas (cargando la página Web).

Las *applets* deben incluir un método de nombre *start()*, que será ejecutado cuando el navegador intente mostrar por pantalla la *applet*.

Estas aplicaciones, son seguras (cumplen la especificación Java), y al ser distribuibles por Internet no permiten incluir métodos nativos Java.

d.) JavaBeans

Los *JavaBeans* son componentes gráficos de Java, que se pueden incorporar a otros componentes gráficos. Se incluyen en la API de Java (paquete *java.beans*).

Existe una herramienta de Sun, denominada BDK (Beans Developer Kit), que es un conjunto de herramientas para desarrollar *JavaBeans*. El BDK, es el *JDK* para el desarrollo de *JavaBeans*.

Existen ya multitud de bibliotecas con *JavaBeans*, para que puedan ser utilizados.

e.) JavaScript

JavaScript es un subconjunto del lenguaje Java que puede codificarse directamente sobre cualquier documento HTML; el código fuente de *JavaScript* forma parte del propio documento HTML.

JavaScript tiene menos potencia que Java, a cambio de más control sobre el navegador Web que lo ejecute.

Se utiliza sobre todo para dar animación e interactividad a páginas Web.

JavaScript posee una habilidad limitada para interactuar con applets Java, pero Java no puede interactuar de ningún modo con *JavaScript*.

f.) Servlets

Son módulos que permiten sustituir o utilizar el lenguaje Java en lugar de programas CGI (Common Gateway Interface) a la hora de dotar de interactividad a las páginas Web.

Estas aplicaciones se ejecutan como aplicaciones servidoras en Internet, y normalmente incluyen bucles infinitos a la espera de peticiones a las que atender.

Los *Servlets* no tienen entorno gráfico, ya que se ejecutan en el servidor. Reciben datos y su respuesta más habitual suele ser código HTML (páginas Web).

C. RECOMENDACIONES DE PROGRAMACIÓN

a.) Introducción

Es un hecho innegable que cada programador tiene su forma personal de programar, sus preferencias de indentación...

En cualquier caso, desde este tutorial se proponen una serie de pautas que ayudarán a que sus programas sean más legibles, mantenibles y eficientes:

b.) Propuestas de estilo de los fuentes

Utilice nombres significativos para los identificadores, y procure respetar la siguiente notación de mayúsculas y minúsculas:

- Las clases: *Clase o MiClase.*
- Las interfaces: *Interfaz o MiInterfaz.*
- Los métodos: *metodo() o metodoLargo().*
- Los métodos de acceso: *getAtributo(), setAtributo().*

- Las variables: *altura o alturaMedia.*
- Las constantes: *CONSTATE o CONSTANTE_LARGA.*
- Los paquetes: *java.paquete.subpaquete.*

Cuide los nombres de sus clases y paquetes para que no coincidan con otros ya existentes, mediante la utilización de prefijos identificativos de usted o de su empresa.

Además se enuncian las siguientes recomendaciones:

- No utilice líneas de más de 80 caracteres, en su lugar divida la línea en varias.
- Escriba una única operación en cada línea.
- Inserte líneas en blanco en sus fuentes cuando mejoren la legibilidad.
- No deje de incluir comentarios en sus clases, describiendo su funcionalidad. Si utiliza la notación estandarizada */** */*, además podrá recolectarles automáticamente con *javadoc*.

c.) Propuestas de diseño de los programas

Para cada clase:

- Cree un constructor por defecto.
- Los atributos de las clases no deben de ser *public*.
- Declare métodos de acceso a los atributos.
- Cree métodos de acceso a los atributos; *getAtributo()* y *setAtributo(nuevoValor)*. Si no están implementados declárelos como private.
- Cree un método *main()* que valga para probar la clase.
- Sobre *finalize()*: Cree una bandera de finalización en cada clase, y en los *finalize()*, si la bandera no ha sido establecida, lance una clase derivada de *RuntimeException.* No olvide llamar a *super.finalize().*

Además se enuncian las siguientes recomendaciones:

- Cree paquetes para agrupar clases relacionadas. Utilizar la herencia para simplificar las clases con características comunes.
- Utilice interfaces antes que clases abstractas.
- Utilice composición cuando sea apropiado, no abuse de la herencia.

Para más información véase **[Johnson, 1996]** y **[Eckel, 1997].**

D. GRANDES PROGRAMAS

Aunque con todo lo que se ha visto podríamos aventurarnos a desarrollar programas Java un tanto complejos (con varias clases, paquetes, *threads*...), lo cierto es que el desarrollo de programas complejos es materia de la Ingeniería del Software.

Desde este tutorial se apuesta por la Ingeniería del Software para el desarrollo de software de calidad, en un tiempo mínimo y con unos costes mínimos. La Ingeniería del Software hace posible que un programa sea mantenible y eficiente, y resuelva un problema de una forma adecuada.

Desde aquí se recomienda el uso de una metodología orientada a objeto para el desarrollo del software. Por ejemplo OMT de **[Rambaugh et al., 1998]**, se ajusta bastante bien al desarrollo de Java, además de ser la metodología más extendida actualmente.

Todos los diagramas que se realicen siguiendo una metodología software deben de realizarse utilizando una notación unificada. El lenguaje de modelado UML **[Rational, 1997]**, se presenta como estándar de modelado.

TEMA III: BIBLIOTECAS DE LA API DE JAVA

III.1. BIBLIOTECAS DE LA API DE JAVA

A. INTRODUCCIÓN

Con cada una de las versiones que Sun lanza del JDK, se acompaña de una serie de bibliotecas con clases estándar que valen como referencia para todos los programadores en Java.

Estas clases se pueden incluir en los programas Java, sin temor a fallos de portabilidad. Además, están bien documentadas (mediante páginas Web), y organizadas en paquetes y en un gran árbol de herencia.

A este conjunto de paquetes (o bibliotecas) se le conoce como la API de Java (Application Programming Interface).

En este apartado explicaremos los paquetes básicos de la API de Java, aunque algunos de ellos tienen subpaquetes.

B. PAQUETES DE UTILIDADES

- *java.lang*: Fundamental para el lenguaje. Incluye clases como *String* o *StringBuffer*, que se tratan más en detenimiento en el capítulo *"III.2 Cadenas"* de este tutorial.
- *java.io*: Para la entrada y salida a través de flujos de datos, y ficheros del sistema. Se estudia en el capítulo *"III.3 Entrada/Salida"* de este tutorial.
- *java.util*: Contiene colecciones de datos y clases, el modelo de eventos, facilidades horarias, generación aleatoria de números, y otras clases de utilidad.
- *java.math*: Clases para realizar aritmética con la precisión que se desee.
- *java.text*: Clases e interfaces para manejo de texto, fechas, números y mensajes de una manera independiente a los lenguajes naturales.
- *java.security*: Clases e interfaces para seguridad en Java: Encriptación RSA...

C. PAQUETES PARA EL DESARROLLO GRÁFICO

- *java.applet*: Para crear applets y clases que las applets utilizan para comunicarse con su contexto. Se estudia en el capítulo *"VI. Applets"* de este tutorial.
- *java.awt*: Para crear interfaces con el usuario, y para dibujar imágenes y gráficos. Se estudia en el capítulo *"IV. Bibliotecas gráficas"* de este tutorial.
- *javax.swing*: Conjunto de componentes gráficos que funcionan igual en todas las plataformas que Java soporta. Se estudia en el capítulo *"IV. Bibliotecas gráficas"* de este tutorial.
- *javax.accesibility*: Da soporte a clases de accesibilidad para personas discapacitadas.
- *java.beans*: Para el desarrollo de *JavaBeans*.

D. PAQUETES PARA EL DESARROLLO EN RED

- *java.net*: Clases para aplicaciones de red. Se estudia en el capítulo *"V. Java e Internet"* de este tutorial.
- *java.sql*: Paquete que contiene el JDBC, para conexión de programas Java con Bases de datos.
- *java.rmi*: Paquete RMI, para localizar objetos remotos, comunicarse con ellos e incluso enviar objetos como parámetros de un objeto a otro.
- *org.omg.CORBA*: Facilita la posibilidad de utilizar OMG CORBA, para la conexión entre objetos distribuidos, aunque esté codificados en distintos lenguajes.
- *org.omb.CosNaming* : Da servicio al IDL de Java, similar al RMI pero en CORBA

E. PARA MÁS INFORMACIÓN

Para más información consulte con la documentación del JDK que vaya a utilizar o la dirección www.sun.com.

Esta información ha sido extraída de la documentación de la API de Java correspondiente al JDK 1.2 **[Sun, 1998]**.

III.2. CADENAS

A. INTRODUCCIÓN

a.) Introducción a las clases String y StringBuffer

En Java las cadenas se tratan de forma diferente a C/C++, donde las cadenas son matrices de caracteres en las que se indica el final de la cadena con un carácter de valor '\0'.

En Java las cadenas son objetos de las clases predefinida *String* o *StringBuffer*, que están incluidas en el paquete *java.lang.**.

Siempre que aparecen conjuntos de caracteres entre comillas dobles, el compilador de Java crea automáticamente un objeto *String*.

Si sólo existieran cadenas de sólo lectura (*String*), durante una serie de manipulaciones sobre un objeto *String* habría que crear un nuevo objeto para cada uno de los resultados intermedios.

El compilador es más eficiente y usa un objeto *StringBuffer* para construir cadenas a partir de las expresiones, creando el *String* final sólo cuando es necesario. Los objetos *StringBuffer* se pueden modificar, de forma que no son necesarios nuevos objetos para albergar los resultados intermedios.

Los caracteres de las cadenas tienen un índice que indica su posición. El primer carácter de una cadena tiene el índice 0, el segundo el 1, el tercero el 2 y así sucesivamente. Esto puede sonar familiar a los programadores de C/C++, pero resultar chocante para aquellos programadores que provengan de otros lenguajes.

b.) Operaciones básicas, comunes a String y StringBuffer

Existen una serie de métodos que son comunes a ambas clases.

Los siguientes métodos de acceso a las cadenas:

- *int length()*; Devuelve el número de caracteres de la cadena.
- *char charAt(int i);* Devuelve el carácter correspondiente de índice *i*.

Los siguientes métodos para crear cadenas derivadas:

- *String toString();* Devuelve una copia del objeto como una *String.*
- *String substring(int i, int fin);* Devuelve una instancia de la clase *String* que contenga una subcadena desde la posición *ini*, hasta la *fin* (si no se indica hasta el final de la cadena), del objeto cadena que invoque el método.

Y el método para transformar la cadena en un vector de caracteres:

- *void getChars(int ini, int fin, char[] destino, int destIni);* Convierte la cadena en un vector de caracteres *destino*.

B. MÉTODOS DE LA CLASE STRING

a.) Constructores

La clase *String* proporciona cadenas de sólo lectura y soporta operaciones con ellas. Se pueden crear cadenas implícitamente mediante una cadena entrecomillada o usando + ó += con dos objetos *String*.

También se pueden crear objetos *String* explícitamente con el mecanismo *new*.

La clase *String* soporta multitud de constructores.

- *String();* Constructor por defecto. El nuevo *String* toma el valor *""*.
- *String(String s);* Crea un nuevo *String*, copiando el que recibe por parámetro.
- *String(StringBuffer s);* Crea un *String* con el valor que en ese momento tenga el *StringBuffer* que recibe como parámetro.
- *String(char[] v);* El nuevo *String* toma el valor de los caracteres que tiene el vector de caracteres recibido por parámetro.
- *String(byte[] v);* El nuevo *String* toma el valor de los caracteres que corresponden a los valores del vector de bytes en el sistema de caracteres de la ordenador en que se ejecute.

b.) Búsqueda en cadenas String

Además presenta los siguientes métodos para buscar caracteres o subcadenas en la cadena, y devuelven el índice que han encontrado o el valor *–1* si la búsqueda no ha sido satisfactoria:

- *int indexOf(char ch, int start)*: Devuelve el índice correspondiente a la primera aparición del carácter *ch* en la cadena, comenzando a buscar desde el carácter *start* (si no se especifica se busca desde el principio).
- *int indexOf(String str)*: Devuelve el índice correspondiente al carácter en que empieza la primera aparición de la subcadena *str.*
- *int lastIndexOf(char ch, int start)*: Devuelve el índice correspondiente a la última aparición del carácter *ch* en la cadena, comenzando a buscar desde el carácter *start* (si no se especifica se busca desde el final).
- *int lastIndexOf(String str)*: Devuelve el índice correspondiente al carácter en que empieza la última aparición de la subcadena *str.*

c.) Comparaciones de cadenas String

Java no trabaja con el código *ASCII* habitual, sino con el código avanzado *Unicode.*

El código *Unicode* (código universal) se caracteriza, sobre todo, por el uso de dos bytes por carácter. Esto permite aumentar los caracteres hasta 65000, y así se pueden representar los caracteres que componen las lenguas, vivas o muertas, más importantes del mundo.

Hay que tener en cuenta que si nos salimos del rango 0-255 que coincide con el código *ASCII* puede que las comparaciones no sean las esperadas.

Las funciones de comparación son las siguientes:

- *boolean equals(Object o);* Devuelve *true* si se le pasa una referencia a un objeto *String* con los mismos caracteres, o *false* si no.
- *boolean equalsIgnoreCase(String s);* Compara cadenas ignorando las diferencias de ortografía mayúsculas/minúsculas.
- *boolean regionMatches(boolean b, int o, String s , int i, int n);* Compara parte de dos cadenas, carácter a carácter.
- *boolean startsWith(String s, int* i *)*; Comprueba si la cadena tiene el prefijo *s* desde *i.*
- *boolean endsWith(String s);* Comprueba si la cadena termina con el sufijo *s*.
- *int compareTo(Object o);* Devuelve un entero que es menor, igual o mayor que cero cuando la cadena sobre la que se le invoca es menor, igual o mayor que la otra. Si el parámetro es un *String*, la comparación es léxica.
- *int compareToIgnoraCase(String s);* Compara lexicográficamente, ignorando las diferencias de ortografía mayúsculas/minúsculas.

d.) Cadenas String derivadas

En Java se devuelven nuevas cadenas cada vez que se invoca a un método que crea una cadena diferente porque las cadenas *String* son de sólo lectura:

- *String replace(char oldChar, char newChar);* Devuelve una nueva cadena con todos los caracteres *oldChar* sustituidos por el carácter *newChar*.
- *String toLowerCase();* Devuelve una nueva cadena con los caracteres en minúsculas, o si se especifica parámetro, siguiendo sus reglas.
- *String toUperCase(Locale l);* Devuelve una nueva cadena con los caracteres en mayúsculas, o si se especifica parámetro, siguiendo sus reglas.
- *static String trim()*: Devuelve una nueva cadena del que se ha eliminado los espacios en blanco por el principio y por el final.
- *static String copyValueOf(char[] v, int ini, int fin);* Devuelve una cadena igual que la contenida en el vector *v*, entre los límites *ini* y *fin* (si no se especifican copia todo el vector).
- *static String concat(String s)*; Concatena la cadena que recibe al final de ésta.

e.) Conversiones entre cadenas String y tipos simples Java

Para convertir una variable de un tipo de datos simple (*char*, *boolean, int, long, float, double)* en una cadena (*String*), bastará con invocar al método *valueOf()* del objeto *String* correspondiente:

- *static String valueOf(tipo);* El parámetro *tipo* soporta un carácter (*char*) un vector de caracteres (*char[]*) o un objeto (*Object*).

Sin embargo para convertir el valor de una cadena en un tipo de datos simple deberemos utilizar los siguientes métodos:

Tipo	*De String*
boolean	*new Boolean(String).booleanValue()*
int	*Integer.ParseInt(String, int base)*
long	*Long.ParseLong(String, int base)*
float	*new Float(String).floatValue()*
double	*new Double(String).doubleValue()*

Tabla 20: Conversiones de cadenas a tipos simples

No hay ningún método que convierta los caracteres escapados Java (\b, \u*dddd*) en variables carácter (*char),* o a la inversa. Lo que se puede hacer es invocar *valueOf()* con un carácter (*char*) para obtener una cadena de la clase *String* que contenga ese carácter.

Así mismo no hay formas de crear o decodificar cadenas de números en formatos octal (*0*) o hexadecimal(*0x*).

f.) Conversiones entre cadenas String y vectores

También existen diversos constructores y métodos de la clase *String* para tratar con vectores, tanto de caracteres como de bytes.

En cuanto a los vectores de caracteres existen:

- El constructor ya citado de *String(char[] v)*. Hace copia de los datos, por lo que las modificaciones posteriores del vector no afectarán a la cadena.
- *char[] toCharArray();* Convierte la cadena en un vector de caracteres.

En cuanto a los métodos para convertir vectores de *byte* (de 8 bits) en objetos *String* con caracteres *Unicode* de 16 bits existen:

- El constructor ya citado de *String(byte[] v)*. Hace copias de los datos, por lo que las modificaciones posteriores del vector no afectarán a la cadena.
- *byte[] getBytes(String s);* Convierte la cadena en un vector de *byte*, atendiendo a la tabla de caracteres especificada en *s*, o a la de la máquina si se omite.

C. MÉTODOS DE LA CLASE STRINGBUFFER

a.) Constructores

Los constructores contenidos por la clase *StringBuffer* son:

- *StringBuffer(int lim);* Construye una cadena sin caracteres y con una capacidad inicial de *lim* caracteres (por defecto 16, si no se especifica otro valor).
- *StringBuffer(String s);* Construye una cadena con el valor *s*.

b.) Modificación de la cadena

Existen tres tipos de modificaciones que se pueden aplicar a la cadena.

Hay métodos de inserción:

- *StringBuffer insert(int i, Object o);* Desplaza los caracteres de la cadena e inserta la cadena correspondiente al segundo parámetro (de cualquier tipo).

- *StringBuffer append(Object o);* Inserta al final de la cadena, la cadena correspondiente al segundo parámetro (de cualquier tipo).

Hay métodos de substitución:

- *void setCharAt(int i, char to);* Cambia el carácter de la posición *i* por *to*. Si la posición supera la longitud de la cadena, se extiende rellenándose con caracteres nulos.
- *StringBuffer replace(int ini, int fin, String s);* Reemplaza una subcadena de esta cadena (de *ini* a *fin*) por la cadena recibida por parámetro. No se debe confundir con el método *replace()* que incluye la clase *String*.
- *StringBuffer reverse()*; Invierte la cadena (de izquierda a derecha).

Y métodos de borrado:

- *StringBuffer delete(int ini, int fin);* Borra la subcadena entre el carácter *ini* y el *fin*.
- *StringBuffer deleteCharAt(int i);* Borra el carácter en la posición *i*.

c.) Capacidad de la cadena

El buffer de un objeto *StringBuffer* tiene una *capacidad* que es la longitud de la cadena que puede almacenar sin tener que asignar más espacio. El buffer crece automáticamente a medida que se añaden caracteres, pero resulta más eficiente especificar el tamaño del buffer de una sola vez:

- *int capacity();* Devuelve la capacidad actual del buffer.
- *void ensureCapacity(int i);*. Garantiza que la capacidad del buffer es al menos *i*.
- *void setLength(int i);* Establece la longitud de esta cadena a *i*.

d.) Extracción de datos

Para obtener un objeto *String* a partir de un objeto *StringBuffer*, debe invocarse el método *toString()*, común a ambas clases.

Se debe tener en cuenta que no hay métodos *StringBuffer* para eliminar una parte de un buffer. Para resolver este problema, debe crearse un vector de caracteres a partir del buffer, y construir un nuevo buffer con el contenido restante. Para esto se usa el método *getChars()*, común con la clase *String*.

D. EJEMPLOS DE USO DE CADENAS

a.) Ejemplo de Cadena Fija (de la clase String)

En el siguiente ejemplo se muestra la utilización de los principales métodos de la clase *String*:

```
public static void main( String s[] ){
  String cad = new String("Cadena Fija");
  System.out.println("Ejemplo de String: '"+cad+"'");
  //Métodos comunes a StringBuffer
  System.out.println("Su longitud es: "+cad.length());
```

```
  System.out.println("Su tercer caracter es: "+cad.charAt(3));
  System.out.print("Su subcadena del 3 al 6 es:");
  System.out.println( cad.substring(2,6) );
  char[] vectorcad = cad.toCharArray();
  System.out.println("Creado un vector, de elemento 3: ");
  System.out.print( vectorcad[2] );
  // Búsqueda en cadenas
  String subcad=new String("ena");
  System.out.print("La subcadena '"+subcad+"'");
  System.out.print(" aparece en la posicion: ");
  System.out.println( cad.indexOf(subcad) );
  // Comparaciones
  String cadcomp=new String("CADENA Fija");
  System.out.print("La cadena '"+cadcomp+"'");
  if ( cad.compareTo(cadcomp) == 0 )
    System.out.println(" ES igual 'Sensitive'");
  else
    System.out.println(" NO es igual 'Sensitive'");
  System.out.print("La cadena '"+cadcomp+"'");
  if ( cad.equalsIgnoreCase(cadcomp) )
     System.out.println(" ES igual 'Insensitive'");
    else
     System.out.println(" NO = 'Insensitive'");
  // Derivación
  System.out.print("Cadena derivada en minusculas: ");
    System.out.println( cad.toLowerCase() );
  }
```

Lo que muestra por pantalla:

```
Ejemplo de String: 'Cadena Fija'
Su longitud es: 11
Su tercer caracter es: e
Su subcadena del 3 al 6 es: dena
Creado un vector, de elemento 3: d
La subcadena 'ena' aparece en la posicion: 3
La cadena 'CADENA Fija' NO es igual 'Sensitive'
La cadena 'CADENA Fija' ES igual 'Insensitive'
Cadena derivada en minusculas: cadena fija
```

b.) Ejemplo de Cadena Variable (de la clase StringBuffer)

En el siguiente ejemplo se muestra la utilización de los principales métodos de la clase *StringBuffer*:

```
public static void main( String s[] ){
  StringBuffer cad = new StringBuffer("Cadena Variable");
  System.out.println("Ejemplo de StringBuffer: '"+cad+"'");
  // Modificación de la cadena
  cad.delete( 0, 6 );
  System.out.println("Borrados 6 primeros caracteres: "+cad);
  cad.replace( cad.length()-3, cad.length(), "da" );
  System.out.println("Sutituidos ultimos caracteres: "+cad);
  cad.append(" Cadena");
  System.out.println("Apendizada con 'Cadena': "+cad);
  // Gestión de su capacidad
  System.out.println("Tiene capacidad de: "+cad.capacity());
  cad.ensureCapacity( 32 );
  System.out.println("Capacidad sobre 32:"+cad.capacity());
  System.out.println("");
}
```

Lo que muestra por pantalla:

```
Ejemplo de StringBuffer: 'Cadena Variable'
Borrados 6 primeros caracteres: Variable
Sutituidos ultimos caracteres: Variada
Apendizada con 'Cadena': Variada Cadena
Tiene capacidad de: 31
Capacidad sobre 32: 64
```

III.3. ENTRADA/SALIDA

A. INTRODUCCIÓN

Normalmente, cuando se codifica un programa, se hace con la intención de que ese programa pueda interactuar con los usuarios del mismo, es decir, que el usuario pueda pedirle que realice cosas y pueda suministrarle datos con los que se quiere que haga algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos para proporcionarnos una respuesta a lo solicitado.

Además, en muchas ocasiones interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como *Entrada/Salida* (*E/S)*.

Existen dos tipos de E/S; la *E/S estándar* que se realiza con el terminal del usuario y la *E/S a través de fichero*, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar de la API de Java denominado *java.io* que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros. En este tutorial sólo se van a dar algunas pinceladas de la potencia de este paquete.

B. ENTRADA/SALIDA ESTÁNDAR

Aquí sólo trataremos la entrada/salida que se comunica con el usuario a través de la pantalla o de la ventana del terminal.

Si creamos una *applet* no se utilizarán normalmente estas funciones, ya que su resultado se mostrará en la ventana del terminal y no en la ventana de la *applet*. La ventana de la *applet* es una ventana gráfica y para poder realizar una entrada o salida a través de ella será necesario utilizar el AWT.

El acceso a la entrada y salida estándar es controlado por tres objetos que se crean automáticamente al iniciar la aplicación: *System.in*, *System.out* y *System.err*

a.) System.in

Este objeto implementa la entrada estándar (normalmente el teclado). Los métodos que nos proporciona para controlar la entrada son:

- *read():* Devuelve el carácter que se ha introducido por el teclado leyéndolo del buffer de entrada y lo elimina del buffer para que en la siguiente lectura sea leído el siguiente carácter. Si no se ha introducido ningún carácter por el teclado devuelve el valor -1.
- *skip(n):* Ignora los *n* caracteres siguientes de la entrada.

b.) System.out

Este objeto implementa la salida estándar. Los métodos que nos proporciona para controlar la salida son:

- *print(a):* Imprime *a* en la salida, donde *a* puede ser cualquier tipo básico Java ya que Java hace su conversión automática a cadena.
- *println(a):* Es idéntico a *print(a)* salvo que con *println()* se imprime un salto de línea al final de la impresión de *a*.

c.) System.err

Este objeto implementa la salida en caso de error. Normalmente esta salida es la pantalla o la ventana del terminal como con *System.out*, pero puede ser interesante redirigirlo, por ejemplo hacia un fichero, para diferenciar claramente ambos tipos de salidas.

Las funciones que ofrece este objeto son idénticas a las proporcionadas por *System.out*.

d. Ejemplo

A continuación vemos un ejemplo del uso de estas funciones que acepta texto hasta que se pulsa el retorno de carro e informa del número de caracteres introducidos.

```
import java.io.*;
class CuentaCaracteres {
  public static void main(String args[]) throws IOException {
    int contador=0;
    while(System.in.read()!='\n')
      contador++;
    System.out.println(); // Retorno de carro "gratuito"
    System.out.println("Tecleados "+contador+" caracteres.");
  }
}
```

C. ENTRADA/SALIDA POR FICHERO

a.) Tipos de ficheros

En Java es posible utilizar dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio).

Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de datos (*int*, *float*, *boolean,...*).

Una lectura secuencial implica tener que acceder a un elemento antes de acceder al siguiente, es decir, de una manera lineal (sin saltos). Sin embargo los ficheros de acceso aleatorio permiten acceder a sus datos de una forma aleatoria, esto es indicando una determinada posición desde la que leer/escribir.

b.) Clases a estudiar

En el paquete *java.io* existen varias clases de las cuales podemos crear instancias de clases para tratar todo tipo de ficheros.

En este tutorial sólo vamos a trata las tres principales:

- *FileOutputStream*: Fichero de salida de texto. Representa ficheros de texto para escritura a los que se accede de forma secuencial.
- *FileInputStream*: Fichero de entrada de texto. Representa ficheros de texto de sólo lectura a los que se accede de forma secuencial.
- *RandomAccessFile*: Fichero de entrada o salida binario con acceso aleatorio. Es la base para crear los objetos de tipo fichero de acceso aleatorio. Estos ficheros permiten multitud de operaciones; saltar hacia delante y hacia atrás para leer la información que necesitemos en cada momento, e incluso leer o escribir partes del fichero sin necesidad de cerrarlo y volverlo a abrir en un modo distinto.

c.) Generalidades

Para tratar con un fichero siempre hay que actuar de la misma manera:

1. Se abre el fichero.

Para ello hay que crear un objeto de la clase correspondiente al tipo de fichero que vamos a manejar, y el tipo de acceso que vamos a utilizar:

```
TipoDeFichero obj = new TipoDeFichero( ruta );
```

Donde *ruta* es la ruta de disco en que se encuentra el fichero o un descriptor de fichero válido.

Este formato es válido, excepto para los objetos de la clase *RandomAccessFile* (acceso aleatorio), para los que se ha de instanciar de la siguiente forma:

```
RandomAccessFile obj = new RandomAccessFile( ruta, modo );
```

Donde *modo* es una cadena de texto que indica el modo en que se desea abrir el fichero; "r" para sólo lectura o "rw" para lectura y escritura.

2. Se utiliza el fichero.

Para ello cada clase presenta diferentes métodos de acceso para escribir o leer en el fichero.

3. Gestión de excepciones (opcional, pero recomendada)

Se puede observar que todos los métodos que utilicen clases de este paquete deben tener en su definición una cláusula *throws IOException*. Los métodos de estas clases pueden lanzar excepciones de esta clase (o sus hijas) en el transcurso de su ejecución, y dichas excepciones deben de ser capturadas y debidamente gestionadas para evitar problemas.

4. Se cierra el fichero y se destruye el objeto.

Para cerrar un fichero lo que hay que hacer es destruir el objeto. Esto se puede realizar de dos formas, dejando que sea el recolector de basura de Java el que lo destruya cuando no lo necesite (no se recomienda) o destruyendo el objeto explícitamente mediante el uso del procedimiento c*lose()* del objeto:

```
obj.close()
```

d.) La clase FileOutputStream

Mediante los objetos de esta clase escribimos en ficheros de texto de forma secuencial.

Presenta el método *write()* para la escritura en el fichero. Presenta varios formatos:

- *int write(int c):* Escribe el carácter en el fichero.
- *int write(byte a[]):* Escribe el contenido del vector en el fichero.
- *int write(byte a[], int off, int len):* Escribe *len* caracteres del vector *a* en el fichero, comenzando desde la posición *off*.

El siguiente ejemplo crea el fichero de texto "*/carta.txt*" a partir de un texto que se le introduce por teclado:

```
import java.io.*;
class CreaCarta {
  public static void main(String args[]) throws IOException{
    int c;
    FileOutputStream f=new FileOutputStream("/carta.txt");
    while( ( c=System.in.read() ) != -1 )
      f.write( (char)c );
    f.close();
  }
}
```

e.) La clase FileInputStream

Mediante los objetos de esta clase leemos de ficheros de texto de forma secuencial.

Presenta el método *read()* para la lectura del fichero. Este método se puede invocar de varias formas.

- *int read():* Devuelve el siguiente carácter del fichero.
- *int read(byte a[]):* Llena el vector *a* con los caracteres leídos del fichero. Devuelve la longitud del vector que se ha llenado si se realizó con éxito o –1 si no había suficientes caracteres en el fichero para llenar el vector.
- *int read(byte a[], int off, int len):* Lee *len* caracteres del fichero, insertándolos en el vector *a*.

Todos ellos devuelven -1 si se ha llegado al final del fichero (momento de cerrarle).

El siguiente ejemplo muestra el fichero de texto "*/carta.txt*" en pantalla:

```
import java.io.*;
class MuestraCarta {
  public static void main(String args[]) throws IOException {
    int c;
    FileInputStream f=new FileInputStream("/carta.txt");
    while( ( c=f.read() ) != -1 )
      System.out.print( (char)c );
```

```
      f.close();
    }
  }
```

f.) La clase RandomAccessFile

Mediante los objetos de esta clase utilizamos ficheros binarios mediante un acceso aleatorio, tanto para lectura como para escritura. En estos ficheros hay un índice que nos dice en qué posición del fichero nos encontramos, y con el que se puede trabajar para posicionarse en el fichero.

Métodos de desplazamiento

Cuenta con una serie de funciones para realizar el desplazamiento del puntero del fichero. Hay que tener en cuenta que cualquier lectura o escritura de datos se realizará a partir de la posición actual del puntero del fichero.

- *long getFilePointer();*Devuelve la posición actual del puntero del fichero.
- *void seek(long l);* Coloca el puntero del fichero en la posición indicada por *l*. Un fichero siempre empieza en la posición 0.
- *int skipBytes(int n);* Intenta saltar *n* bytes desde la posición actual.
- *long length();* Devuelve la longitud del fichero.
- *void setLength(long l);* Establece a *l* el tamaño de este fichero.
- *FileDescriptor getFD();* Devuelve el descriptor de este fichero.

Métodos de escritura

La escritura del fichero se realiza con una función que depende el tipo de datos que se desee escribir.

- *void write(byte b[], int ini, int len);* Escribe *len* caracteres del vector *b*.
- *void write(int i);* Escribe la parte baja de *i* (un byte) en el flujo.
- *void writeBoolean(boolean b);* Escribe el boolean *b* como un byte.
- *void writeByte(int i);* Escribe *i* como un byte.
- *void writeBytes(String s);* Escribe la cadena s tratada como bytes, no caracteres.
- *void writeChar(int i);* Escribe *i* como 1 byte.
- *void writeChars(String s);* Escribe la cadena *s*.
- *void writeDouble(double d)*; Convierte *d* a *long* y le escribe como 8 bytes.
- *void writeFloat(float f);* Convierte *f* a entero y le escribe como 4 bytes.
- *void writeInt(int i)*; Escribe *i* como 4 bytes.
- *void writeLong(long v);* Escribe *v* como 8 bytes.
- *void writeShort(int i);* Escribe *i* como 2 bytes.
- *void writeUTF(String s);* Escribe la cadena *s* utilizando la codificación UTF-8.

Los métodos que escriben números de más de un byte escriben el primero su parte alta.

Métodos de lectura

La lectura del fichero se realiza con una función que depende del tipo de datos que queremos leer.

- *boolean readBoolean();* Lee un byte y devuelve *false* si vale 0 o *true* sino.
- *byte readByte();* Lee y devuelve un byte.
- *char readChar();* Lee y devuelve un caracter.
- *double readDouble();* Lee 8 bytes, y devuelve un *double*.
- *float readFloat();* Lee 4 bytes, y devuelve un *float*.
- *void readFully(byte b[]);* Lee bytes del fichero y los almacena en un vector *b*.
- *void readFully(byte b[], int ini, int len);* Lee *len* bytes del fichero y los almacena en un vector *b*.
- *int readInt();* Lee 4 bytes, y devuelve un *int*.
- *long readLong();* Lee 8 bytes, y devuelve un *long*.
- *short readShort();* Lee 2 bytes, y devuelve un *short*.
- *int readUnsignedByte();* Lee 1 byte, y devuelve un valor de 0 a 255.
- *int readUnsignedShort();* Lee 2 bytes, y devuelve un valor de 0 a 65535.
- *String readUTF();* Lee una cadena codificada con el formato UTF-8.
- *int skipBytes(int n);* Salta *n* bytes del fichero.

Si no es posible la lectura devuelven *–1*.

Ejemplo

Vamos a crear un pequeño programa que cree y acceda a un fichero binario, mediante acceso aleatorio.

El siguiente ejemplo crear un fichero binario que contiene los 100 primeros números (en orden):

```
// Crea un fichero binario con los 100 primeros numeros
static void creaFichBin( String ruta ) throws IOException {
  RandomAccessFile f=new RandomAccessFile(ruta,"rw"); // E/S
  for ( int i=1; i <= 100 ; i++ )
  {
    try{
      f.writeByte( i );
    } catch( IOException e){
      // Gestion de excepcion de ejemplo
      break; // No se puede seguir escribiendo
    }
    f.close();
}
```

El siguiente método accede al elemento *cual* de un fichero binario, imprimiendo la longitud del fichero, el elemento *cual* y su 10 veces siguiente elemento:

```
static void imprimeEltoN(String ruta, long cual)
    throws IOException{

  RandomAccessFile f=new RandomAccessFile(ruta,"r"); // E/

  System.out.print( "El fichero " + ruta );
  System.out.println( " ocupa " + f.length() + " bytes." );

  f.seek( cual-1 ); // Me posiciono (-1 porque empieza en 0)
  System.out.print(" En la posicion " + f.getFilePointer() );
  System.out.println(" esta el numero " + f.readByte() );

  f.skipBytes( 9 ); // Salto 9 => Elemento 10 mas alla
  System.out.print(" 10 elementos más allá, esta el ");
  System.out.println( f.readByte() );

  f.close();
}
```

Si incluimos ambos métodos en una clase, y les llamamos con el siguiente programa principal (*main()*):

```
public static void main(String args[]) throws IOException {
  String ruta="numeros.dat"; // Fichero
  creaFichBin( ruta ); // Se crea
  imprimeEltoN( ruta, 14 ); // Accedo al elemento 14.
}
```

Obtendremos la siguiente salida:

```
El fichero numeros.dat ocupa 100 bytes.
 En la posicion 13 esta el numero 14
 10 elementos más allá, esta el 24
```

TEMA IV. BILIOTECAS GRÁFICAS

IV.1. LOS PAQUETES GRÁFICOS DE LAS JFC

A. INTRODUCCIÓN

Las JFC (Java Foundation Classes) son parte de la API de Java compuesto por clases que sirven para crear interfaces gráficas visuales para las aplicaciones y *applets* de Java.

Así como Sun presenta estas JFC, Microsoft ha desarrollado otro paquete propio con el nombre de AFD (Application Foundation Classes).

Las JFC contienen dos paquetes gráficos: AWT y Swing.

- AWT presenta componentes pesados, que en cada plataforma sólo pueden tener una representación determinada. Está disponible desde la versión 1.1 del JDK como *java.awt*.
- Swing presenta componentes ligeros, que pueden tomar diferente aspecto y comportamiento pues lo toman de una biblioteca de clases. Está disponible desde la versión 1.2 del JDK como *javax.swing* aunque antes se podían encontrar versiones previas como *com.sun.java.* o como *java.awt.swing.*

AWT se estudia con más detenimiento en el apartado *"IV.2.AWT"* de este tutorial, mientras que Swing se estudia en el apartado *"IV.3.Swing"* de este tutorial.

B. MODELO DE EVENTOS

Tanto AWT como Swing tienen en común un sistema para gestionar los eventos que se producen al interactuar con el usuario de la interfaz gráfica; su *modelo de eventos*.

El funcionamiento del modelo de eventos se basa en la gestión de excepciones.

Para cada objeto que represente una interfaz gráfica, se pueden definir objetos "oyentes" (*Listener*), que esperen a que suceda un determinado evento sobre la interfaz. Por ejemplo se puede crear un objeto oyente que esté a la espera de que el usuario pulse sobre un botón de la interfaz, y si esto sucede, él es avisado, ejecutando determinada acción.

La clase base para todos estos eventos que se pueden lanzar es la clase *AWTEvent* (perteneciente al paquete *java.awt*).

El modelo de eventos de AWT depende del paquete *java.awt.event*, que en Swing se amplía con el paquete *javax.swing.event*.

Existen dos tipos básicos de eventos:

- Físicos: Corresponden a un evento hardware claramente identificable. Ej: se ha pulsado una tecla (*KeyStrokeEvent*).
- Semánticos: Se componen de un conjunto de eventos físicos, que sucedidos en un determinado orden tienen un significado más abstracto: El usuario ha elegido un elemento de una lista desplegable (*ItemEvent*).

C. SUBPAQUETES DE AWT

A continuación se enumeran los paquetes que componen las JFC, así como su funcionalidad, ordenados por orden de antigüedad. Esto es importante, porque debe de tenerse en cuenta si se va a utilizar para crear *applets*, ya que normalmente los navegadores no soportan la última versión de la API de Java en su propia máquina virtual.

Si queremos que nuestra *applet* se vea igual en varios navegadores, deberemos de tener en cuenta qué paquetes podemos utilizar y cuáles no, comprobando la versión de Java que soportan los navegadores que nos interesen.

a.) Desde la versión 1.0

- *java.awt*: Contiene todas las clases básicas de AWT para crear interfaces e imprimir gráficos e imágenes, así como la clase base para los eventos en componentes: *AWTEvent*.
- *java.awt.image*: Para crear y modificar imágenes. Utiliza productores de imágenes, filtros y "consumidores de imágenes". Permite renderizar una imagen mientras está siendo generada.

b.) Desde la versión 1.1

- *java.awt.datatransfer*: Transferencia de datos entre aplicaciones. Permite definir clases "transferibles" entre aplicaciones, y da soporte al mecanismo del portapapeles (copiar y pegar).
- *java.awt.event*: Modelo de eventos de AWT. Contiene eventos, oyentes y adaptadores a oyentes.

c.) Desde la versión 1.2

- *java.awt.color*: Utilización de colores. Contiene la implementación de una paleta de colores basada en la especificada por el ICC (Consorcio Internacional de Color).
- *java.awt.dnd*: Operaciones de arrastrar y soltar.
- *java.awt.font*: Todo lo referente a las fuentes de texto. Soporta fuentes del tipo True Type, Type 1, Type 1 Multiple Master, y OpenType.
- *java.awt.geom:* Aporta clases de Java 2D, para crear objetos en 2 dimensiones, utilizando geometría plana (elipses, curvas, áreas...).
- *java.awt.im*: Para utilizar símbolos Japoneses, Chinos o Coreanos.
- *java.awt.image.renderable*: Para producir imágenes que sean *independientes de redering* (animación).
- *java.awt.print*: Para imprimir documentos. Incorpora capacidad para gestionar diversos tipos de documentos, formatos de página e interactuar con el usuario para controlar la impresión de trabajos.

D. SUBPAQUETES DE SWING

A continuación se enumeran los paquetes que componen Swing, así como su funcionalidad:

- *javax.swing*: Tiene los componentes básicos para crear componentes ligeros Swing.
- *javax.swing.border*: Para dibujar bordes personalizados en los componentes Swing.
- *javax.swing.colorchooser*: Para utilizar el componente *JColorChooser*.
- *javax.swing.event*: Eventos lanzados por componentes Swing, así como oyentes para dichos eventos. Extiende los que se encuentran en el paquete AWT *java.awt.event*.
- *javax.swing.filechooser*: Para utilizar el componente *JFileChooser*.
- *javax.swing.plaf*: Permite a Swing utilizar múltiples representaciones. Se utiliza por aquellos desarrolladores que no pueden crear un nuevo aspecto de interfaz, basado en los que Swing ya incorpora (como Basic o Metal).
- *javax.swing.plaf.basic*: Objetos que utilizan interfaces de aspecto "Basic". Este aspecto es el que presentan por defecto los componentes Swing. En este paquete se encuentran gestores de impresión, eventos, oyentes y adaptadores. Se puede crear un aspecto personalizado de interfaz utilizando este paquete.
- *javax.swing.plaf.metal*: Objetos que utilizan interfaces de aspecto "Metal".
- *javax.swing.plaf.multi*: Permite a los usuarios combinar aspectos de interfaz, entre auxiliares y los que existen por defecto.
- *javax.swing.text*: Para manejar componentes de texto (modificables o no). Soporta sintaxis resaltada, edición, estilos...
- *javax.swing.text.html*: Contiene la clase *HTMLEditorKit*, basada en la versión 3.2 de la especificación HTML, y clases para crear editores de texto HTML
- *javax.swing.text.html.parser*: Contiene analizadores de texto HTML.
- *javax.swing.text.rtf*: Contiene la clase *RTFEditorKit* para crear editores de documentos en formato RTF (Rich-Text-Format).
- *javax.swing.tree*: Para personalizar la forma en que son utilizados los árboles generados por la clase *java.awt.swing.JTree*.
- *javax.swing.undo*: Permite realizar operaciones de *deshacer/rehacer* en las aplicaciones que cree el usuario.

IV.2. AWT (Abstract Windowing Toolkit)

A. INTRODUCCIÓN

AWT es un conjunto de herramientas GUI (Interfaz Gráfica con el Usuario) diseñadas para trabajar con múltiples plataformas.

Este paquete viene incluido en la API de Java como *java.awt* ya desde su primera versión, con lo que las interfaces generadas con esta biblioteca funcionan en todos los entornos Java disponibles (incluyendo navegadores, lo que les hace especialmente eficientes para la creación de *applets* Java).

En el apartado "*VI.3. Ejemplo de creación de una applet*" se muestra un breve ejemplo de cómo utilizar las clases del AWT para crear una *applet* y una aplicación de Java.

La siguiente figura muestra la jerarquía de clases para las principales clases de AWT:

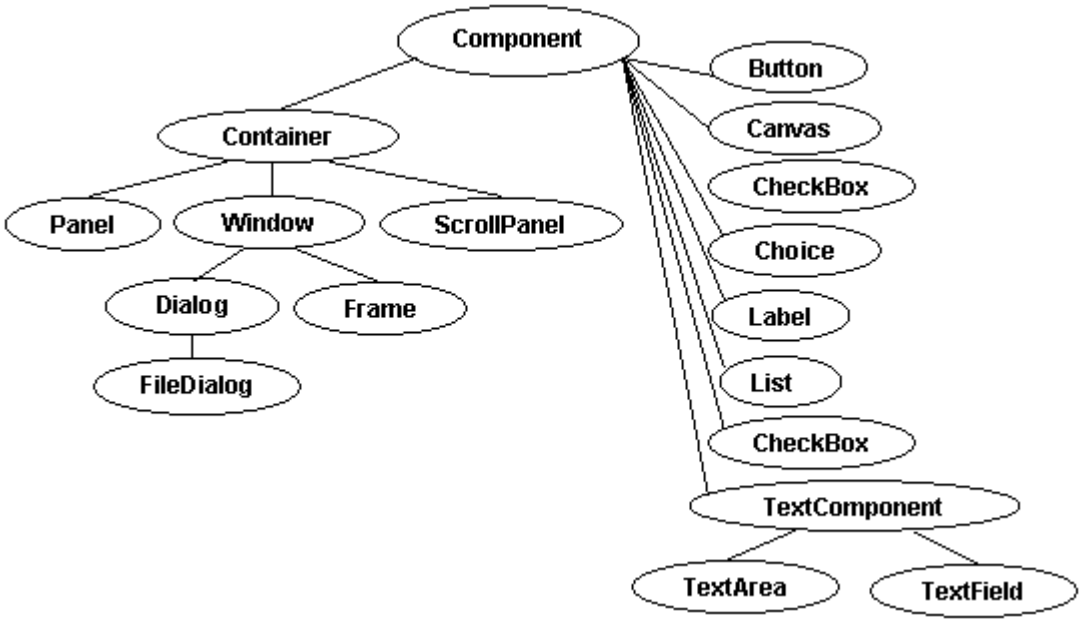

Imagen 7: Jerarquía de las clases de AWT

En este apartado vamos a estudiar algunas de las clases más importantes del AWT, así como su funcionalidad.

B. COMPONENT

Esta clase representa a cualquier objeto que puede ser parte de una interfaz gráfica de usuario. Es la clase padre de muchas de las clases del AWT.

Su propósito principal es representar algo que tiene una posición y un tamaño, que puede ser dibujado en la pantalla y que pueda recibir eventos de entrada (que responda a las interacciones con el usuario).

La clase *Component* presenta diversos métodos, organizados para cubrir varios propósitos.

A continuación se explican algunos de ellos.

a.) Tamaño y posición del componente

- *Dimension getSize();* Devuelve la anchura y altura del componente como un objeto de la clase *Dimension*, que tiene como campos: *width* (anchura) y *heigth* (altura).
- *void setSize(int ancho, int largo);* Establece la anchura y altura del componente.
- *Dimension getPreferredSize();* Devuelve el tamaño que este componente debería tener.
- *void setPreferredSize();* Establece el tamaño que este componente debería tener.
- *Dimension getMinimumSize();* Devuelve el tamaño mínimo que este componente debería tener.
- *void setMinimumSize(int ancho, int largo);* Establece el tamaño mínimo que este componente debería tener.
- *Rectangle getBounds();* Devuelve las coordenadas de este componente como un objeto de la clase *Rectangle*, que tiene como campos: *x, y, width y heigth*.
- *void setBounds(int x, int y, int ancho, int largo);* Establece las coordenadas de este componente.

b.) Acciones sobre el componente

- *boolean getEnabled();* Comprueba si el componente está o no activo.
- *void setEnabled(boolean);* Establece el componente a activo o inactivo.
- *boolean getVisible();* Comprueba si el componente está o no visible.
- *void setVisible(boolean);* Establece si el componente está visible o invisible.
- *void paint(Graphics g);* Indica al AWT que ha de dibujar el componente *g*.
- *void repaint();* Indica al AWT que ha de volver a dibujar el componente.
- *void update(Graphics g);* Es llamado por AWT cuando se invoca el método *repaint()*. Por defecto llama a *paint()*.

c.) Eventos de interacción con el usuario

A su vez hay tres tipos de métodos, para la gestión de eventos mediante el nuevo modelo de eventos de AWT (desde la versión 1.1).

Hay tres tipos de métodos:

- *void add_Tipo_Listener(_Tipo_Listener l);* Añade un oyente a la espera de algún tipo de eventos sobre este componente.
- *void remove_Tipo_Listener(_Tipo_Listener l);* Elimina algún oyente que estaba a la espera de algún tipo de eventos sobre este componente.
- *void process_Tipo_Event(_Tipo_Event e);* Procesa eventos del tipo *_Tipo_Event* enviándolos a cualquier objeto *_Tipo_Listener* que estuviera escuchando.

En estos métodos *_Tipo_* puede ser cualquiera de los siguientes:

Component, Focus, InputMethod, Key, Mouse, MouseMotion.

C. CONTAINER

La clase *Container* sabe cómo mostrar componentes embebidos (que a su vez pueden ser instancias de la clase *Container*).

Algunos de los métodos de la clase *Container* son:

- *Component add(Component c);* Añade un componente al contenedor.
- *void print(Graphics g);* Imprime el contenedor.
- *void printComponents(Graphics g);* Imprime cada uno de los componentes de este contenedor.
- *LayoutManager getLayout();* Devuelve el gestor de impresión (*LayoutManager*) asociado a este contenedor, que es el responsable de colocar los componentes dentro del contenedor.
- *void setLayout(LayoutManager 1);* Establece un gestor de impresión para este componente.

Estos objetos *Container* tienen un *LayoutManager* asociado que define la manera en que van a posicionarse los objetos componentes en su interior.

D. GESTORES DE IMPRESIÓN

LayoutManager y *LayoutManager2* son dos interfaces encargadas de la representación y posicionamiento en pantalla de componentes AWT.

De estas interfaces se proporcionan cinco implementaciones en AWT. Cada una de ellas reparte los objetos de una forma particular:

- *BorderLayout*: En cinco lugares: Norte, Sur, Este, Oeste y Centro (*North, South, East, West* y *Center*).
- *CardLayout*: Permite gestionar varios componentes de los que sólo uno se visualiza a la vez, permaneciendo los demás invisibles debajo.
- *FlowLayout*: De izquierda a derecha horizontalmente en cada línea. Cuando sobrepasan una línea se comienza a la izquierda de la siguiente.
- *GridLayout*: En una tabla en la que todas las casillas tienen el mismo tamaño.
- *GridBagLayout*: En una tabla, pero las casillas no tienen que tener el mismo tamaño.

E. OTRAS CLASES

Por supuesto AWT no se limita a estas clases. Dentro de esta biblioteca podemos encontrar multitud de clases prefabricadas para facilitar el diseño gráfico.

A continuación explicamos algunas de ellas.

a.) Clases contenedoras (hijas de Container

- *Panel*: Permite hacer una presentación más avanzada que *Container* mediante la combinación con subpaneles o subclases para crear contenedores personalizados. La clase *Applet* que sirve para crear *applets* Java, hereda de esta clase *Panel*.

- *ScrollPane*: Una barra de desplazamiento, horizontal o vertical.
- *Window*: Una ventana sin borde.
- *Frame*: Una ventana que no tiene borde. Puede tener asociado un objeto *Menubar* (una barra de herramientas o barra de menú personalizada).
- *Dialog*: Una ventana usada para crear diálogos. Tiene la capacidad de ser *modal* con lo que sólo este contenedor recibiría entradas del usuario.
- *Filedialog*: Un diálogo que usa el selector de archivos nativo del sistema operativo.

b.) Clases componentes (hijas directas de Component)

- *Button*: Un botón gráfico para el que se puede definir una acción que sucederá cuando se presione el botón.
- *Canvas*: Permite pintar o capturar eventos del usuario. Se puede usar para crear gráficos o como clase base para crear una jerarquía de componentes personalizados.
- *Checkbox*: Soporta dos estados: *on* y *off*. Se pueden asociar acciones que se ejecuten (*triggers*) cuando el estado cambie.
- *Choice*: Menú desplegable de opciones.
- *Label*: Cadena de etiqueta en una localización dada.
- *List*: Una lista desplegable de cadenas.
- *Scrollbar*: Desplegable de objetos *Canvas*.
- *TextComponent*: Cualquier componente que permita editar cadenas de texto.Tiene dos clases hijas:
  - *TextField*: Componente de texto consistente en una línea que puede ser usada para construir formularios.
  - *TextArea*: Componente para edición de texto de tamaño variable.

F. EVENTOS DE AWT

AWT tiene sus propios eventos, que se explican a continuación.

a.) Eventos físicos

Son todos hijos del evento *ComponentEvent*, que indica algún cambio en un objeto *Component*:

- *InputEvent*: Se ha producido una entrada del usuario. Tiene como eventos hijos *KeyEvent* (pulsación de una tecla) y *MouseEvent* (acción sobre el ratón).
- *FocusEvent*: Avisa al programa de que el componente ha ganado o perdido la *atención* (enfoque) del usuario. Esto se deduce de la actividad del usuario (ratón y teclado).
- *WindowEvent*: Avisa al programa de que el usuario ha utilizado uno de los controles de ventana a nivel del sistema operativo, como los controles de minimizar o cerrar.
- *ContainerEvent*: Se envía cuando se añaden o eliminan componentes a un contenedor.

- *PaintEvent*: Evento especial que señala que el sistema operativo quiere dibujar de nuevo una parte de la interfaz. Un componente debe sobreescribir el método *paint()* o el método *update()* para gestionar este evento.

b.) Eventos semánticos

Son todos hijos del evento *AWTEvent*, que es el evento base de la jerarquía de eventos:

- *ActionEvent*: Avisa al programa de acciones específicas de componentes como las pulsaciones de botones.
- *AdjustmenteEvent*: Comunica que una barra de desplazamiento ha sido ajustada.
- *ItemEvent*: Avisa al programa cuando el usuario interacciona con una elección, una lista o una casilla de verificación.
- *TextEvent*: Avisa cuando un usuario cambia texto en un componente *TextComponent, TextArea* o *TextField.*
- *InputMethodEvent*: Avisa que un texto que está siendo creado utilizando un método de entrada está cambiando (se ha escrito algo más...).
- *InvocationEvent*: Este evento ejecuta el método *run()* en una clase *Runnable* cuando es tratado por el *thread* del despachador (*dispatcher*) de AWT.

IV.3. SWING

A. INTRODUCCIÓN

El paquete Swing es el nuevo paquete gráfico que ha aparecido en la versión 1.2 de Java. Está compuesto por un amplio conjunto de componentes de interfaces de usuario que funcionen en el mayor número posible de plataformas.

Cada uno de los componentes de este paquete puede presentar diversos aspectos y comportamientos en función de una biblioteca de clases. En la versión 1.0 de Swing, que corresponde a la distribuida en la versión 1.2 de la API de Java se incluyen tres bibliotecas de aspecto y comportamiento para Swing:

- *metal.jar*: Aspecto y comportamiento independiente de la plataforma.
- *motif.jar*: Basado en la interfaz Sun Motif.
- *windows.jar*: Muy similar a las interfaces Microsoft Windows 95.

La siguiente imagen muestra una aplicación de ejemplo (adjunta al JDK 1.2) que muestra las diferentes interfaces para una misma aplicación según se utilice una u otra biblioteca:

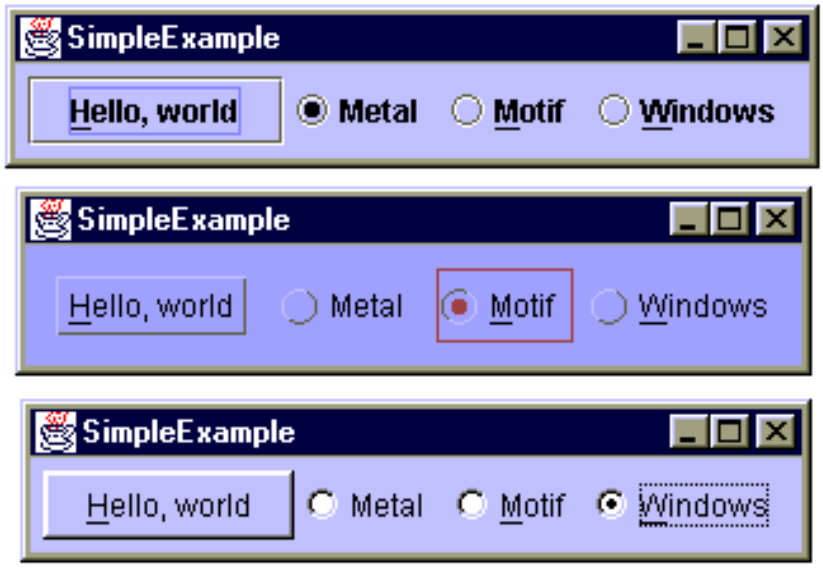

Imagen 8: Diferentes aspectos de una interfaz Swing

Es la nueva clase denominada *UiManager* la que se encarga del aspecto y comportamiento de una aplicación Swing en un entorno de ejecución.

En el apartado "*VI.3. Ejemplo de creación de una applet*" se muestra un breve ejemplo de cómo utilizar las clases de Swing para crear una aplicación utilizando Swing.

B. NUEVAS CARACTERÍSTICAS

La arquitectura Swing presenta una serie de ventajas respecto a su antecedente AWT:

- Amplia variedad de componentes: En general las clases que comiencen por "J" son componentes que se pueden añadir a la aplicación. Por ejemplo: *JButton*.

- Aspecto modificable (*look and feel*): Se puede personalizar el aspecto de las interfaces o utilizar varios aspectos que existen por defecto (Metal Max, Basic Motif, Window Win32).
- Arquitectura Modelo-Vista-Controlador: Esta arquitectura da lugar a todo un enfoque de desarrollo muy arraigado en los entornos gráficos de usuario realizados con técnicas orientadas a objetos. Cada componente tiene asociado una clase de modelo de datos y una interfaz que utiliza. Se puede crear un modelo de datos personalizado para cada componente, con sólo heredar de la clase *Model*.
- Gestión mejorada de la entrada del usuario: Se pueden gestionar combinaciones de teclas en un objeto *KeyStroke* y registrarlo como componente. El evento se activará cuando se pulse dicha combinación si está siendo utilizado el componente, la ventana en que se encuentra o algún hijo del componente.
- Objetos de acción (*action objects*): Estos objetos cuando están activados (*enabled*) controlan las acciones de varios objetos componentes de la interfaz. Son hijos de *ActionListener.*
- Contenedores anidados: Cualquier componente puede estar anidado en otro. Por ejemplo, un gráfico se puede anidar en una lista.
- Escritorios virtuales: Se pueden crear escritorios virtuales o "interfaz de múltiples documentos" mediante las clases *JDesktopPane* y *JInternalFrame*.
- Bordes complejos: Los componentes pueden presentar nuevos tipos de bordes. Además el usuario puede crear tipos de bordes personalizados.
- Diálogos personalizados: Se pueden crear multitud de formas de mensajes y opciones de diálogo con el usuario, mediante la clase *JOptionPane*.
- Clases para diálogos habituales: Se puede utilizar *JFileChooser* para elegir un fichero, y *JColorChooser* para elegir un color.
- Componentes para tablas y árboles de datos: Mediante las clases *JTable* y *JTree*.
- Potentes manipuladores de texto: Además de campos y áreas de texto, se presentan campos de sintaxis oculta *JPassword*, y texto con múltiples fuentes *JTextPane*. Además hay paquetes para utilizar ficheros en formato HTML o RTF.
- Capacidad para "deshacer": En gran variedad de situaciones se pueden deshacer las modificaciones que se realizaron.
- Soporte a la accesibilidad: Se facilita la generación de interfaces que ayuden a la accesibilidad de discapacitados, por ejemplo en *Braille*.

C. PRINCIPALES CLASES

Las clases de Swing se parecen mucho a las de AWT.

Todas las clases explicadas en el apartado "IV.2 AWT" de este tutorial tienen una nueva versión en Swing con el prefijo *J*. Así la clase *Panel* de AWT tiene una clase *JPanel* en Swing. Esto se cumple para todas las clases menos para *Choice, Canvas, FileDialgog* y *ScrollPane*.

De hecho todas las clases componentes de Swing (clases hijas de *JComponent*), son hijas de la clase *Component* de AWT.

- *ButtonGroup*: Muestra una lista de elementos (*JRadioButton*) con solo uno seleccionable. Cada elemento tiene un círculo, que en caso del elemento seleccionado contendrá un "punto".
- *JToggleButton*: Es como un botón normal, pero al ser pinchado por el usuario queda activado.
- *JProgressBar*: Representa una barra de estado de progreso, mediante la que habitualmente se muestra el desarrollo de un proceso en desarrollo (ejemplo: la instalación de una aplicación).
- *JTabbedPane*: Es una ventana con solapas (la que utiliza Windows). Este componente había sido muy solicitado.
- *JApplet*: Aunque ya existía una clase *Applet* en AWT, esta nueva versión es necesaria para crear *applets* Java que utilicen interfaces Swing.

Por supuesto Swing no se limita a estas clases, sino que posee muchas más con diversas funcionalidades. Para estudiarlas consulte la documentación del JDK 1.2 de Java.

D. NUEVOS GESTORES DE IMPRESIÓN

Swing incorpora nuevos gestores de impresión, ampliando los cinco que AWT incorporaba. Entre ellos conviene destacar los siguientes:

- *BoxLayout*: Es similar al *FlowLayout* de AWT, con la diferencia de que con él se pueden especificar los ejes (x o y). Viene incorporada en el componente *Box*, pero está disponible como una opción en otros componentes.
- *OverlayLayout*: Todos los componentes se añaden encima de cada componente previo.
- *SpringLayout*: El espacio se asigna en función de una serie de restricciones asociadas con cada componente.
- *ScrollPaneLayout*: Incorporado en el componente *ScrollPane.*
- *ViewportLayout*: Incorporado en el componente *Viewport.*

E. JROOTPANE

La clase *JRootPane* permite colocar contenido de las *applets* creadas con la clase *JApplet* en un determinado plano de impresión (capa).

Por orden de cercanía al usuario, estas capas son:

- *glassPane*: Una capa que abarca toda la parte visible (por defecto no es visible).
- *layeredPane*: Una subclase de *JComponent* diseñada para contener cuadros de diálogo, menús emergentes y otros componentes que deben aparecer flotando entre el usuario y el contenido.
- *menubar*: Una capa opcional, que si aparece estará anclada en la parte superior.
- *contenPane*: La capa en que se dibujará la mayor parte del contenido.

Así pues cada vez que se vayan a añadir componentes a una *applet* de clase *JApplet*, debe añadirse a uno de estas capas. Por ejemplo:

```
laJApplet.getContentPane().add( unComponente );
```

F. NUEVOS EVENTOS DE SWING

Swing incorpora su nuevo conjunto de eventos para sus componentes.

a.) Eventos físicos

Sólo aparecen dos nuevos eventos físicos, descendientes de *InputEvent*:

- *MenuKeyEvent*: Un menú de árbol ha recibido un evento de *KeyEvent* (acción sobre el ratón).
- *MenuDragMouseEvent*: Un menú de árbol ha recibido un evento de *MouseEvent* (pulsación de una tecla).

b.) Eventos semánticos

Son todos hijos del evento de AWT *AWTEvent*, que es el evento base de la jerarquía de eventos:

- *AncestorEvent*: Antecesor añadido desplazado o eliminado.
- *CaretEvent*: El signo de intercalación del texto ha cambiado.
- *ChangeEvent*: Un componente ha sufrido un cambio de estado.
- *DocumentEvent*: Un documento ha sufrido un cambio de estado.
- *HyperlinkEvent*: Algo relacionado con un vínculo hipermedia ha cambiado.
- *InternalFrameEvent*: Un *AWTEvent* que añade soporte para objetos *JInternalFrame.*
- *ListDataEvent*: El contenido de una lista ha cambiado o se ha añadido o eliminado un intervalo.
- *ListSelectionEvent*: La selección de una lista ha cambiado.
- *MenuEvent*: Un elemento de menú ha sido seleccionado o mostrado o bien no seleccionado o cancelado.
- *PopupMenuEvent*: Algo ha cambiado en *JPopupMenu*.
- *TableColumnModelEvent*: El modelo para una columna de tabla ha cambiando.
- *TableModelEvent*: El modelo de una tabla ha cambiado.
- *TreeExpansionEvent*: El nodo de un árbol se ha extendido o se ha colapsado.
- *TreeModelEvent*: El modelo de un árbol ha cambiado.
- *TreeSelectionEvent*: La selección de un árbol ha cambiado de estado.
- *UndoableEditEvent*: Ha ocurrido una operación que no se puede realizar.

G. EL PATRÓN DE DISEÑO MODELO-VISTA-CONTROLADOR

Muchos de los componentes Swing están basados en un patrón de diseño denominado "Modelo-Vista-Controlador".

El concepto de este patrón de diseño se basa en tres elementos:

- Modelo: Almacena el estado interno en un conjunto de clases.
- Vista: Muestra la información del modelo

- Controlador: Cambia la información del modelo (delegado).

No es menester de este tutorial explicar todo el funcionamiento de este nuevo diseño, pero si se quiere profundizar en él consulte **[Morgan, 1999]**.

TEMA V. JAVA E INTERNET

V.1 JAVA E INTERNET

A. INTRODUCCIÓN

Una de las grandes potencias del lenguaje de programación Java es la total portabilidad de sus programas gracias a su afamada "máquina virtual". Esto adquiere una importancia aún mayor en Internet donde existen tipos de computadoras muy dispares.

Las siguientes bibliotecas de la API de Java contienen una serie de clases que son interesantes de cara a la creación de aplicaciones que trabajen en red. Las más importantes son:

- *java.applet*: Da soporte a las applets.
- *java.net*: Clases para redes. Dan acceso a TCP/IP, *sockets* y URLs.

Conviene destacar la existencia de otras bibliotecas más complejas, orientadas también a la programación en red, que aunque no serán estudiadas en este tutorial, sí conviene tener presente su existencia:

- *java.sql*: Paquete que contiene el JDBC, para conexión de programas Java con Bases de datos.
- *java.rmi*: Paquete RMI, para localizar objetos remotos, comunicarse con ellos e incluso enviar objetos como parámetros de un objeto a otro.
- *org.omg.CORBA*: Facilita la posibilidad de utilizar OMG CORBA, para la conexión entre objetos distribuidos, aunque estén codificados en distintos lenguajes.
- *org.omb.CosNaming* : Da servicio al IDL de Java, similar al RMI pero en CORBA.

Una de las características de Java que lo hacen especialmente interesante para Internet es que sus programas objeto (códigos de byte) son verificables para poder detectar posibles virus en sus contenidos. Estos programas *Códigos de byte* no necesitan ser recompilados, y una vez verificados (pues Java trabaja con nombres no con direcciones), se transforman en direcciones físicas de la máquina destino.

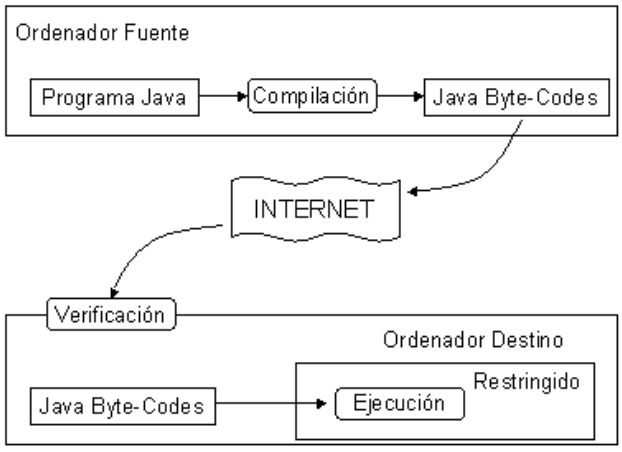

Imagen 9: Ejecución de un código de byte

Esta forma de trabajar cuida la seguridad sin un grave perjuicio de la eficiencia. Un programa en Java es sólo unas 20 veces más lento que uno programado en C, cifra

aceptable para la mayoría de las tareas, y suponiendo que no se utilice un compilador JIT.

B. EL PAQUETE JAVA.NET

Java ofrece un conjunto de clases que permiten utilizar los URLs (*Uniform Resource Locators*). Un URL es una dirección electrónica que permite encontrar una información en Internet especificando:

- El nombre del protocolo que permitirá leer la información. Por ejemplo *HTTP*.
- El nombre del servidor que proporciona la información. Por ejemplo *sunsite.unc.edu* o bien una dirección IP directamente.
- El nombre del fichero en el servidor. Por ejemplo */Javafaq/Javafaq.htm*.

C. FUTURO DEL JAVA EN INTERNET

Java es seguramente el lenguaje con más futuro en cuanto a la programación para Internet.

De hecho, podría evolucionar hasta el punto de que el navegador no interprete las applets de Java, sino que él sea un conjunto de applets que se descarguen de Internet según se vayan necesitando. Así es que en todo momento podríamos estar ejecutando la última versión del navegador.

Incluso siendo un poco más futuristas, podríamos plantearnos conectarnos a servidores que nos cobraran por el uso de sus programas (hojas de cálculo, procesadores de texto...) en función del tiempo de uso, trabajando siempre con la última versión del mismo, en lugar de invertir nuestro dinero en actualizaciones.

V.2 LOS SOCKETS EN JAVA

A. FUNDAMENTOS

Los *sockets* son un sistema de comunicación entre procesos de diferentes máquinas de una red. Más exactamente, un *socket* es un punto de comunicación por el cual un proceso puede emitir o recibir información.

Fueron popularizados por *Berckley Software Distribution*, de la universidad norteamericana de Berkley. Los *sockets* han de ser capaces de utilizar el protocolo de streams TCP (Transfer Contro Protocol) y el de datagramas UDP (User Datagram Protocol).

Utilizan una serie de primitivas para establecer el punto de comunicación, para conectarse a una máquina remota en un determinado puerto que esté disponible, para escuchar en él, para leer o escribir y publicar información en él, y finalmente para desconectarse.

Con todas primitivas se puede crear un sistema de diálogo muy completo.

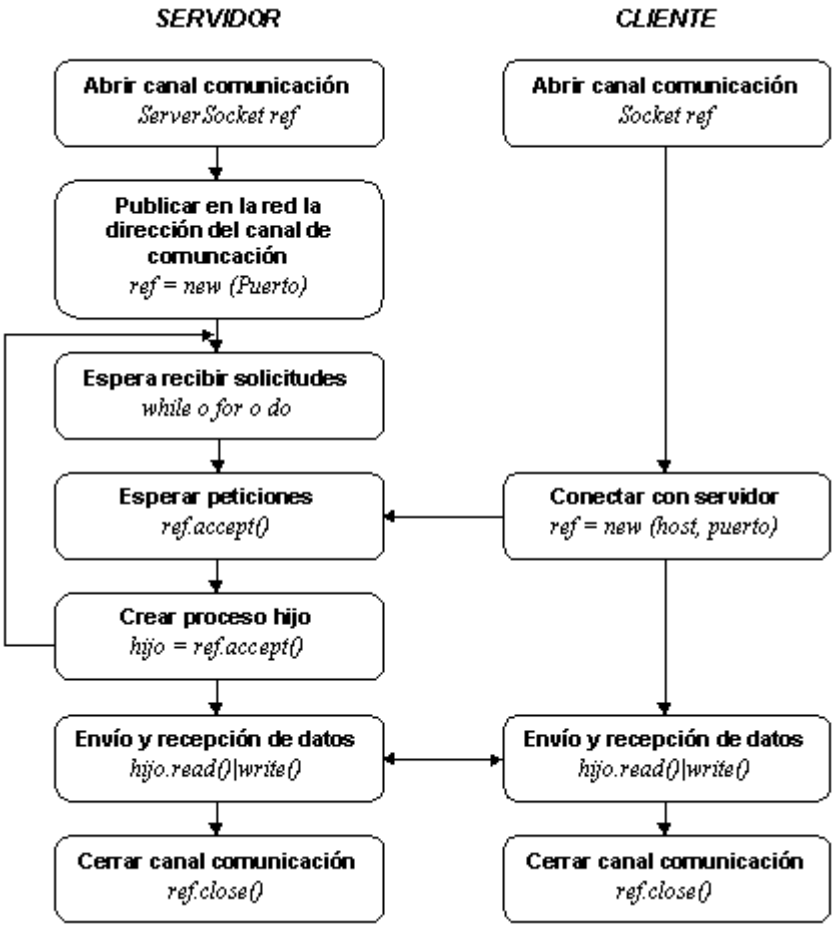

Imagen 10: Funcionamiento de una conexión socket

Para más información véase **[Rifflet, 1998]**.

B. EJEMPLO DE USO

Para comprender el funcionamiento de los *sockets* no hay nada mejor que estudiar un ejemplo. El que a continuación se presenta establece un pequeño diálogo entre un programa servidor y sus clientes, que intercambiarán cadenas de información.

a.) Programa Cliente

El programa cliente se conecta a un servidor indicando el nombre de la máquina y el número puerto (tipo de servicio que solicita) en el que el servidor está instalado.

Una vez conectado, lee una cadena del servidor y la escribe en la pantalla:

```
import java.io.*;
import java.net.*;
class Cliente {
  static final String HOST = "localhost";
  static final int PUERTO=5000;
  public Cliente( ) {
    try{
      Socket skCliente = new Socket( HOST , Puerto );
      InputStream aux = skCliente.getInputStream();
      DataInputStream flujo = new DataInputStream( aux );
      System.out.println( flujo.readUTF() );
      skCliente.close();
    } catch( Exception e ) {
      System.out.println( e.getMessage() );
    }
  }
  public static void main( String[] arg ) {
    new Cliente();
  }
}
```

En primer lugar se crea el *socket* denominado *skCliente*, al que se le especifican el nombre de *host* (*HOST*) y el número de puerto (*PORT*) en este ejemplo constantes.

Luego se asocia el flujo de datos de dicho *socket* (obtenido mediante *getInputStream)*), que es asociado a un flujo (*flujo*) *DataInputStream* de lectura secuencial. De dicho flujo capturamos una cadena (*readUTF()*), y la imprimimos por pantalla (*System.out*).

El *socket* se cierra, una vez finalizadas las operaciones, mediante el método *close()*.

Debe observarse que se realiza una gestión de excepción para capturar los posibles fallos tanto de los flujos de datos como del *socket*.

b.) Programa Servidor

El programa servidor se instala en un puerto determinado, a la espera de conexiones, a las que tratará mediante un segundo *socket*.

Cada vez que se presenta un cliente, le saluda con una frase "Hola cliente N".

Este servidor sólo atenderá hasta tres clientes, y después finalizará su ejecución, pero es habitual utilizar bucles infinitos (*while(true)*) en los servidores, para que atiendan llamadas continuamente.

Tras atender cuatro clientes, el servidor deja de ofrecer su servicio:

```
import java.io.* ;
import java.net.* ;
class Servidor {
  static final int PUERTO=5000;
  public Servidor( ) {
    try {
      ServerSocket skServidor = new ServerSocket( PUERTO );
      System.out.println("Escucho el puerto " + PUERTO );
      for ( int numCli = 0; numCli < 3; numCli++; ) {
        Socket skCliente = skServidor.accept(); // Crea objeto
        System.out.println("Sirvo al cliente " + numCli);
        OutputStream aux = skCliente.getOutputStream();
        DataOutputStream flujo= new DataOutputStream( aux );
        flujo.writeUTF( "Hola cliente " + numCli );
        skCliente.close();
      } // Cierra while
      System.out.println("Demasiados clientes por hoy");
    } catch( Exception e ) {
      System.out.println( e.getMessage() );
    }
  }
  public static void main( String[] arg ) {
    new Servidor();
  }
}
```

Utiliza un objeto de la clase *ServerSocket* (*skServidor*), que sirve para esperar las conexiones en un puerto determinado (*PUERTO*), y un objeto de la clase *Socket* (*skCliente)* que sirve para gestionar una conexión con cada cliente.

Mediante un bucle *for* y la variable *numCli* se restringe el número de clientes a tres, con lo que cada vez que en el puerto de este servidor aparezca un cliente, se atiende y se incrementa el contador.

Para atender a los clientes se utiliza la primitiva *accept()* de la clase *ServerSocket*, que es una rutina que crea un nuevo *Socket* (*skCliente*) para atender a un cliente que se ha conectado a ese servidor.

Se asocia al *socket* creado (*skCliente*) un flujo (*flujo)* de salida *DataOutputStream* de escritura secuencial, en el que se escribe el mensaje a enviar al cliente.

El tratamiento de las excepciones es muy reducido en nuestro ejemplo, tan solo se captura e imprime el mensaje que incluye la excepción mediante *getMessage()*.

c.) Ejecución

Aunque la ejecución de los *sockets* está diseñada para trabajar con ordenadores en red, en sistemas operativos multitarea (por ejemplo Windows y UNIX) se puede probar el correcto funcionamiento de un programa de *sockets* en una misma máquina.

Para ellos se ha de colocar el servidor en una ventana, obteniendo lo siguiente:

```
>java Servidor
Escucho el puerto 5000
```

En otra ventana se lanza varias veces el programa cliente, obteniendo:

```
>java Cliente
Hola cliente 1
>java Cliente
Hola cliente 2
>java Cliente
Hola cliente 3
>java Cliente
connection refused: no further information
```

Mientras tanto en la ventana del servidor se ha impreso:

```
Sirvo al cliente 1
Sirvo al cliente 2
Sirvo al cliente 3
Demasiados clientes por hoy
```

Cuando se lanza el cuarto de cliente, el servidor ya ha cortado la conexión, con lo que se lanza una excepción.

Obsérvese que tanto el cliente como el servidor pueden leer o escribir del *socket*. Los mecanismos de comunicación pueden ser refinados cambiando la implementación de los *sockets*, mediante la utilización de las clases abstractas que el paquete *java.net* provee.

TEMA VI: APPLETS JAVA

VI.1. INTRODUCCIÓN A LAS APPLETS

A. INTRODUCCIÓN

Las *applets* (miniaplicación) son programas escritos en Java que sirven para "dar vida" a las páginas Web (interacción en tiempo real, inclusión de animaciones, sonidos...), de ahí su potencia.

Las *applets* son programas que se incluyen en las páginas Web. Las *applets s*on ejecutadas en la máquina cliente, con lo que no existen ralentizaciones por la saturación del módem o del ancho de banda. Permiten cargar a través de la red una aplicación portable que se ejecuta en el navegador. Para que esto ocurra tan sólo hace falta que el navegador sea capaz de interpretar Java.

A las páginas que contienen *applets* se las denomina páginas *Java-Powered*. Las *applets* pueden ser visualizadas con la herramienta *appletviewer*, incluido en el JDK de Java.

Las *applets* no son exactamente aplicaciones Java, ya que presentan las siguientes diferencias respecto a las aplicaciones normales Java:

Se cargan mediante un navegador, no siendo lanzados por el intérprete Java.

Son cargados a través de la red por medio de páginas HTML y no residen en el disco duro de la máquina que los ejecuta.

Poseen un ciclo de vida diferente; mientras que una aplicación se lanza una vez, una *applet* se arranca (inicia) cada vez que el usuario recarga la página en la que se encuentra la *applet*.

Tienen menos derechos que una aplicación clásica, por razones de seguridad. De modo predeterminado en el puesto que los ejecuta no pueden ni leer ni escribir ficheros, ni lanzar programas, ni cargar DLLs. Sólo pueden comunicarse con el servidor Web en que se encuentra la página Web que las contiene.

B. CONSIDERACIONES SOBRE LA SEGURIDAD EN LAS APPLETS

Como ya se ha dicho las applets tienen una serie de restricciones de programación que las hacen "seguras".

Estas restricciones de seguridad son especialmente importantes, ya que evitarán que se cargue por error una *applet* que destruya datos de la máquina, que obtenga información restringida, o que produzca otros daños inesperados.

Las *applets* no dejan de ser "ejecutables" que funcionan dentro de una aplicación, como puede ser un visualizador de páginas Web (*browser*). Este ejecutable puede obtenerse de una red, lo que significa que hay código posiblemente no fiable que se ejecuta dentro de la aplicación.

Java tiene muchas salvaguardas de seguridad que minimizan el riesgo de la ejecución de applets, pero estas salvaguardas también limitan a los programadores de *applets* en su capacidad de programación.

El modelo de seguridad para las *applets* en Java trata una *applet* como código no fiable ejecutándose dentro de un entorno fiable. Por ejemplo, cuando un usuario instala una copia de un navegador Web en una máquina se está fiando de que su código será

funcional en el entorno. Normalmente los usuarios tienen cuidado de qué instalan cuando proviene de una red. Una *applet*, por el contrario, se carga desde la red sin ninguna comprobación de su fiabilidad.

El lenguaje Java y las *applets* son escritos para que eviten las *applets* no fiables.

Estas salvaguardas son implementadas para verificar que los códigos de byte de las clases de los applets, no rompen las reglas básicas del lenguaje ni las restricciones de acceso en tiempo de ejecución. Sólo cuando estas restricciones son satisfechas se le permite a la *applet* ejecutar su código. Cuando se ejecuta, se le marca para señalar que se encuentra dentro del intérprete. Esta marca permite a las clases de tiempo de ejecución determinar cuándo a una fracción del código se le permite invocar a cierto método. Por ejemplo, una *applet* está restringida en los *hosts* en los que se puede abrir una conexión de red o en un conjunto de URLs a las que puede acceder.

En su conjunto estas restricciones constituyen una política de seguridad. En el futuro, Java tendrá políticas más ricas, incluyendo algunas que usen encriptación y autentificación para permitir a las *applets* una mayor capacidad.

La actual política de seguridad afecta a los recursos que una *applet* puede usar, cuyos principales puntos son:

- Los accesos que pueden realizar las *applets* a los ficheros son restringidos. En particular escribir en ficheros y/o leerles no será una capacidad estándar que se pueda realizar en los navegadores que soporten *applets* de Java.
- Las conexiones de red serán restringidas a conectar solo con el *host* del que proviene la *applet*.
- Una *applet* no es capaz de usar ningún método que pueda resultar en una ejecución arbitraria, código no revisado o ambos. Esto incluye métodos que ejecuten programas arbitrarios (métodos nativos) así como la carga de bibliotecas dinámicas.

Se anticipa en cualquier caso que en el futuro los modelos de seguridad permitirán a las *applets* autentificadas superar estas restricciones.

VI.2. LA CLASE APPLET

A. SITUACIÓN DE LA CLASE APPLET EN LA API DE JAVA

La clase *Applet* Java, de la cual han de heredar todos los programas Java que vayan a actuar como *applets*, es la única clase que contiene el paquete *java.applet* de la API de Java.

Esta clase hereda de *Object* (como todas las clases Java), pero además hereda de *Component* y *Container*, que son dos clases del paquete gráfico AWT. Esto ya perfila las posibilidades gráficas de este tipo de aplicaciones Java.

B. MÉTODOS DEL CICLO DE VIDA

Como ya se ha indicado una *applet* no tiene un ciclo de vida tan "sencillo" como el de una aplicación, que simplemente se ejecuta hasta que finaliza su método *main()*.

La siguiente figura modeliza el ciclo de vida de una *applet*:

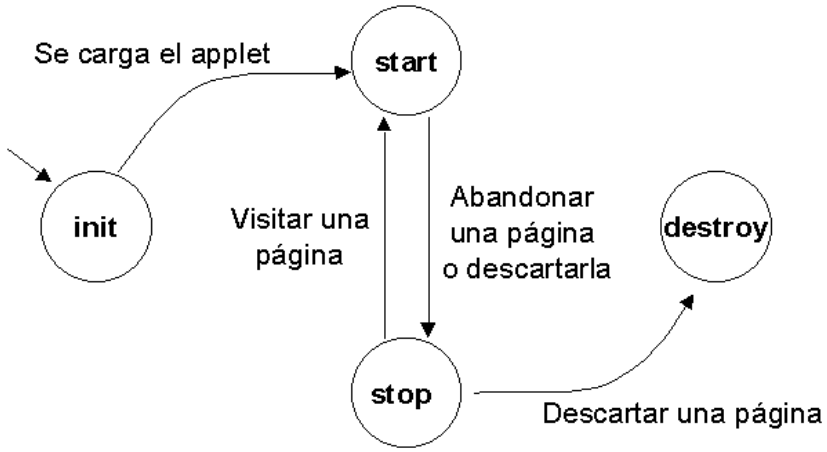

Imagen 11: Ciclo de vida de una applet

Cada círculo representa una fase en el ciclo de vida de la *applet*. Las flechas representan transiciones y el texto representa la acción que causa la transición. Cada fase está marcada con una invocación a un método de la *applet*:

- *void init();* Es invocado cuando se carga la *applet*. Aquí se suelen introducir las iniciaciones que la *applet* necesite.
- *void start();*Es invocado cuando la *applet*, después de haber sido cargada, ha sido parada (cambio de página Web, minimización del navegador,...), y de nuevo activada (vuelta a la página, restauración del navegador,...). Se informa a la *applet* de que tiene que empezar su funcionamiento.
- *void stop();* Es invocado para informar a la *applet* de que debe de parar su ejecución. Así una *applet* que utilice *threads*, debería detenerlos en el código de este método.
- *void destroy();*Es invocado para informar a la *applet* de que su espacio está siendo solicitado por el sistema, es decir el usuario abandona el navegador. La *applet* debe de aprovechar este momento para liberar o destruir los recursos que está utilizando.

- *void paint();* Es invocado cada vez que hay que el navegador redibuja la *applet*.

Al crear una *applet* no es necesario implementar todos estos métodos. De hecho habrá *applets* que no los necesiten.

Cuando un navegador carga una página Web que contiene una *applet*, suele mostrar en su parte inferior un mensaje como:

```
initializing... starting...
```

Esto indica que la *applet*, se está cargando:

1. Una instancia de la clase *applet* es creada.

2. La *applet* es iniciada, mediante su método *init()*.

3. La *applet* empieza a ejecutarse, mediante su método *start()*.

Cuando el usuario se encuentra con una página Web, que contiene una *applet* y salta a otra página, entonces la *applet* se detiene invocando a su método *stop()*. Si el usuario retorna a la página donde reside la *applet*, ésta vuelve a ejecutarse nuevamente invocando a su método *start()*.

Cuando el usuario sale del navegador la *applet* tiene un tiempo para finalizar su ejecución y hacer una limpieza final, mediante el método *destroy()*.

C. LA CLASE URL

Un URL (Uniform Resource Locator) es una dirección de Internet. Cada recurso (fichero, página Web, imagen...) tiene uno propio. En Java existe una clase denominada *URL* que modeliza esta clase de objetos.

La clase *URL* pertenece al paquete *java.net*, y tiene una cierta importancia en el desarrollo de las *applets*, puesto que muchos de los métodos de la clase *Applet* la utilizan para acceder a determinado recurso de Internet o para identificarse.

Podemos especificar un URL de manera absoluta:

```
URL URLabsoluto = new URL(“http://www.host.com/dir/fich.htm”);
```

O bien podemos especificar un *URL* de manera relativa:

```
URL URLhost = new URL(“http://www.Javasoft.com/”);
URL URLrelativo = new URL( URLhost, “dir/fich.htm”);
```

Ambos ejemplos corresponderían al URL "*http://www.host.com/dir/fich.htm*".

D. INCLUSIÓN DE LA APPLET EN UNA PÁGINA WEB

Para incluir una *applet* en una página Web, una vez compilada la *applet*, debe incluirse entre el código HTML de la página Web una etiqueta *<APPLET>*, que como mínimo ha de presentar los siguientes tres parámetros:

- *code*: Especifica el URL del fichero de clase Java (*.class) que contiene la *applet*.
- width: Especifica la anchura inicial de la *applet* (en *pixels*).
- *heigth*: Especifica la altura inicial de la *applet* (en *pixels*).

Además, de la etiqueta inicial, una *applet* puede tener parámetros que se especificarán mediante etiquetas *<PARAM>*, que como mínimo han de presentar dos parámetros:

- *name*: Indica el nombre del parámetro de la *applet* al que esta etiqueta hace referencia.
- *value*: Establece este valor al parámetro indicado en *name* de la misma etiqueta.

Así un ejemplo de esto sería:

```
<applet code="AppletDiagonal.class" width=200 height=200>
<param name=Parametro1 value=Valor1>
<param name=Parametro2 value=Valor2>
</applet>
```

En este ejemplo la *applet* puede entender los parámetro *Parametro1* y *Parametro2*, mediante los métodos que se describen en el siguiente apartado, y obtendría *Valor1* y *Valor2* respectivamente.

Se observa que además de la etiqueta *<applet>* en el código HTML también aparece una etiqueta *</applet>*. Esto sucede porque HTML es un lenguaje pareado, en el que casi todas las etiquetas de inicio de elemento (*<etiq>*) tienen una etiqueta de fin (*</etiq>*).

E. OBTENCIÓN DE LOS PARÁMETROS DE LA APPLET

Cuando se incluye una *applet* en una página Web ha de hacerse mediante la etiqueta HTML *<applet>*. Las etiquetas HTML permiten utilizar parámetros, y la etiqueta *<applet>* hace lo propio, permitiendo a la *applet* recibir parámetros de ejecución, tal y como una aplicación los recibe en el parámetro *s* (un vector de cadenas) de su método *main(String[] s).*

Los siguientes métodos se utilizan para extraer información de los parámetros que recibió la *applet* cuando fue llamada desde el código HTML:

- *URL getDocumentBase();* Devuelve el URL del documento que contiene la *applet*.
- *URL getCodeBase();* Devuelve el URL de la *applet*.
- *String getParameter(String name);* Devuelve el valor de un parámetro (etiquetas *<param>*) que aparezca en el documento HTML.

Si por ejemplo se llamase a una *applet*, con el código HTML:

```
<applet code=”AppletParam.class” width=50 height=50>
<param name=Color value=”red”>
</applet>
```

Una llamada en esta *applet* al método *getParameter(“Color”)* devolverá *“red”*.

F. OBTENCIÓN DE INFORMACIÓN SOBRE UNA APPLET

Algunos métodos de la *applet* se utilizan para comunicar información o mostrar mensajes en la pantalla referentes a la *applet*:

- *boolean isActive();* Comprueba si la *applet* está activa.
- *void showStatus(String status);* Muestra una cadena del estado en la pantalla.

- *String getAppletInfo()*; Devuelve información relativa a la *applet* como el autor, Copyright o versión.
- *String[ ][ ] getParameterInfo();* Devuelve un vector que describe algún parámetro específico de la *applet*. Cada elemento en el vector es un vector de tres cadenas que tienen la forma: {nombre, tipo, comentario}.

Un ejemplo de como definir este método para una *applet* que permita un solo parámetro, color, sería:

```
public String[][] getParameterInfo() {
   String info[][] = { {“Color”,“String”,“foreground color”} };
   return info;
}
```

G. MANIPULACIÓN DEL ENTORNO DE UNA APPLET

Algunas applets pueden afectar al entorno en que están ejecutándose. Para ello se utilizan los métodos:

- *AppletContext getAppletContext();* Devuelve un *AppletContext*, que permite a la *applet* afectar a su entorno de ejecución.
- *void resize(int ancho, int largo);* Solicita que se modifique el tamaño de la *applet*. También permite recibir un único parámetro *Dimension*.
- *Locale getLocale();* Devuelve el *Locale* de la *applet* si fue establecido.
- *void setStub(AppletStub s);* Establece el *stub* de esta *applet*.

H. SOPORTE MULTIMEDIA

La clase *Applet* también incluye métodos para trabajar con imágenes y ficheros de sonido de Internet mediante la utilización de URLs. Para ello implementa los métodos:

- *Image getImage(URL u, String s);* Obtiene una imagen de un URL *u* que será absoluto si no se especifica una ruta relativa *s*.
- *AudioClip getAudioClip(URL u, String s);* Obtiene un clip de sonido de un URL *u* que será absoluto si no se especifica una ruta relativa *s*.
- *void play(URL ur1, String name);* Ejecuta directamente un fichero de sonido de un URL *u* que será absoluto si no se especifica una ruta relativa *s*.
- *static audioClip newAudioClip(URL u);* Obtiene un nuevo fichero de sonido del URL *u*.

Mediante el uso adecuado de varios de estos métodos se pueden combinar sonidos e imágenes para conseguir efectos espectaculares.

VI.3. EJEMPLO DE CONSTRUCCIÓN DE UNA APPLET

A. CÓDIGO

Para crear una *applet* normalmente será necesario importar al menos las bibliotecas *java.awt.** y la *java.applet.**.

La clase que represente a la *applet* se ha de declarar como una subclase de la clase *Applet*, para poder sobreescribir los métodos de la clase *Applet*.

Siempre conviene sobreescribir al menos el método *paint()* que será llamado por los navegadores que soporten *applets* para mostrarles por pantalla.

Vamos a construir una *applet* denominada *AppletDiagonal* que simplemente dibuje una línea diagonal. Un posible código para esta *applet* sería:

```
import java.awt.*;
import java.applet.*;
public class AppletDiagonal extends Applet {
  public void paint(Graphics g) {
    g.setColor( Color.red );
    g.drawLine(0, 0, getWidth(), getHeight() );
  }
}
```

Pasemos a comentar el funcionamiento de este código:

- El método *paint()* recibe un objeto de la clase *Graphics*. La clase *Graphics*, incluida en el AWT, contiene métodos para mostrar varios tipos de gráficos.
- Mediante el método *setColor()* de la clase *Graphics* se establece el color de primer plano a rojo, que es uno de los colores predefinidos de la clase *Color*.
- Por último, mediante *drawLine()* se dibuja una línea dadas las coordenadas de su esquina superior izquierda y de la inferior derecha. En este caso se indican la esquina superior izquierda de la *applet* mediante las coordenadas (*0,0*), y la esquina inferior derecha se obtiene mediante dos métodos de la clase *Dimension* (*getWidth(), getHeight()*).

B. EJECUCIÓN

Para ejecutar la *applet*, una vez compilado el fichero, se introduce la llamada a la *applet* en una página Web (por ejemplo *AppletDiagonal.htm*), introduciendo entre su código HTML lo siguiente:

```
<applet code="AppletDiagonal.class" width=200 height=200>
</applet>
```

Cuando se cargue esta página Web en un navegador compatible con Java o mediante el visualizador de *applets* que viene con el JDK (*appletviewer*) se verá algo como:

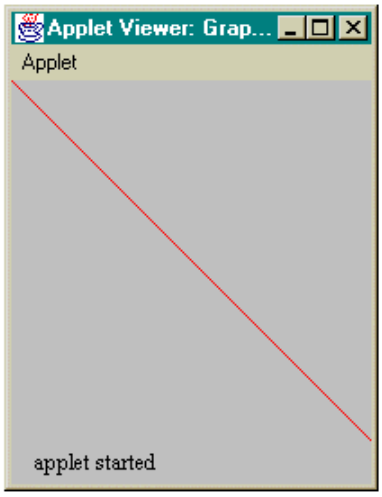

Imagen 12: Applet "Línea"

Se podría dibujar un rectángulo con cambiar la línea de código de *drawLine()* por otra que llamase al método *drawRect()*:

```
g.drawRect(10, 10, r.width –20, r.height –20);
```

C. CREACIÓN DE APPLETS MÁS AVANZADAS

La creación de *applets* complejos, escapa a las intenciones de este tutorial, con lo que no se va a presentar el código fuente de más *applets*.

El dominio de la biblioteca AWT es una condición imprescindible para la creación de *applets* de más calidad y vistosidad.

Por último recordar que con el JDK se incluyen unas cuantas *applets* que pueden servir para el estudio de las mismas, puesto que se incluye su código fuente.

Para más información consulte **[van Hoff et al., 1996]**.

D. CREACIÓN DE UNA APLICACIÓN QUE UTILICE LA APPLET (AWT)

Se va a utilizar AWT para crear una aplicación que de un resultado igual que la ejecución de la *"applet Línea"*. Será una aplicación que creará un *Frame* de AWT para incluir en su interior la *applet* que ya fue creada.

De hecho el *main()* de la aplicación lo único que hará será crear un objeto de este tipo (indicándole altura y anchura, como hacíamos en la *applet* mediante los parámetros de la etiqueta HTML).

El código fuente de la aplicación sería el siguiente:

```
import java.awt.*;
import java.awt.event.*;

class FrameLinea extends Frame {
  private AppletDiagonal unaApplet; // Se mostrará
```

```
  public static void main( String[] s ) {
    new FrameLinea( 200, 230 );
  }
  public FrameLinea( int ancho, int largo ) {
    super(); // Constructor de Component
    // Se añade un oyente que cerrara la aplicación
    addWindowListener( new OyenteLinea() );
    // Se crea una applet de diagonal
    unaApplet=new AppletDiagonal();
    unaApplet.init();
    unaApplet.start();
    // Se mete la applet en frame
    add( unaApplet );
    setSize(ancho,largo); // ajusta frame
    setVisible(true); // muestra frame
  }
  // Clase anidada
  class OyenteLinea extends WindowAdapter {
    // Sobreescribo el método de "cuando se cierra ventana"
    public void windowClosing(WindowEvent e) {
      unaApplet.stop();
      unaApplet.destroy();
      System.exit(0);
    }
  }
}
```

Vamos a crear un *Frame* en el que vamos a incluir la *applet unaApplet* que será de la clase *AppletDiagonal*, creada anteriormente.

La aplicación lo que hace es crear un oyente de la clase creada *OyenteLinea*, que será el encargado de capturar el evento de cerrar la ventana del *Frame*.

En el constructor se inicia la *applet* (*init()* y *start()*) y se añade al *Frame* mediante el método *add()* de la clase *Container* (*Frame* es hija de *Container*).

Por último se establece el tamaño del *Frame* (recibido por parámetro) mediante *setSize()* y por último se muestra el *Frame* que ya tiene en su interior la *applet* (*setVisible()*).

Cuando se cierra la ventana, el *OyenteLinea* se encarga de cerrar la *applet,* mediante *stop()* y *destroy(),* y de finalizar la aplicación mediante *System.exit()*.

E. CREACIÓN DE UNA APLICACIÓN QUE UTILICE LA APPLET (SWING)

Esta misma aplicación se puede crear utilizando Swing con solo cambiar las siguientes cosas:

1. Se ha de incluir la biblioteca de Swing:

```
import javax.swing.*;
```

2. Se han de cambiar los nombres de la clase *Frame* de AWT por la clase *JFrame* de Swing.

3. Se crea un *contentPane* mediante un objeto *JPanel*, justo antes de llamar al oyente:

```
setContentPane( new JPanel() );
```

4. Para añadir la *applet* se ha de añadir al *contentPane*:

```
getContentPane().add( unaApplet );
```

VI.4. EJEMPLOS DE APPLETS

En este apartado se comentan una serie de *applets* que pueden servir tanto para demostrar las posibilidades de estos programas, como para clasificarles por los siguientes géneros:

- Instantáneas: Muestran una secuencia de imágenes.
- Animación y Sonidos: Mezclan imágenes con sonidos.
- Gráficos Interactivos: Permiten la interacción del usuario con las imágenes, mediante respuestas a las acciones del ratón sobre la imagen.
- Trucos de Texto: Permiten animar texto dándole 'vida'.
- Financias y Negocios: Algunos nos permiten mostrar diagramas de barras, y otros elementos ilustrativos de este género.
- Demos, Juegos y Educacionales: Muy especializados, permiten al usuario interactuar consiguiendo cotas fascinantes de diversión.

A continuación veremos un ejemplo de cada grupo que sea lo más significativo posible, es decir, que resalte las características de ese grupo y las diferencias con el resto de los grupos.

En cada uno de ellos se ha incluido una descripción de lo que hace la *applet*, los parámetros que soporta, y un ejemplo del código HTML que habría que insertar en una página Web para incluir la *applet* en dicha página.

A. INSTANTÁNEAS: "TUMBLING DUKE"

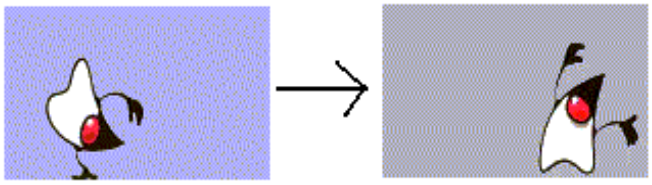

Imagen 13: Applet Instantánea "Tumbling Duke"

a.) Descripción

Se trata de una *applet* en la que *Duke,* la mascota de Java, da volteretas en la página correspondiente. La animación consta de 17 secuencias.

b.) Parámetros

- *maxwidth*: Anchura máxima de la imagen durante la animación.
- *nimgs*: Número de marcos o secuencias en la animación.
- *offset*: Desplazamiento horizontal entre la primera y la última secuencia de la animación.
- *img*: URL del directorio donde se encuentran almacenadas las diferentes secuencias de la animación: T1.gif, T2.gif...

c.) Ejemplo

```
<applet code="TumbleItem.class" width=600 height=95>
<param name=maxwidth value="120">
<param name=nimgs value="16">
<param name=offset value="-57">
<param name=img value="tumble">
</applet>
```

B. ANIMACIÓN Y SONIDO: "ANIMATOR"

Imagen 14: Applet de Animación y sonido "Animator"

a.) Descripción

Esta *applet* permite crear una animación con sonido.

Se puede especificar el orden de las secuencias, si la animación se repite, la pista de sonido, otros sonidos para determinadas secuencias, el espacio de tiempo entre secuencias, una imagen por defecto mientras se está iniciando la *applet*, la posición exacta en la que se quiere que aparezca cada secuencia...

Haciendo un *clic* con el ratón sobre la *applet* se detiene la animación. Haciendo otro continúa la ejecución.

b.) Parámetros

- *imagesource*: URL del directorio que contiene las imágenes de la animación: T1.gif...
- *startup*: URL de la imagen que aparecerá por defecto mientras se cargan el resto de las secuencias.
- *background*: URL de la imagen de fondo.
- *startimage*: Índice de la primera secuencia.
- *endimage*: Índice de la última secuencia de la animación.
- *pauses*: Lista de las pausas en milisegundos. Permite especificar una pausa específica para cada secuencia. Cada número se separa mediante el carácter |.
- *repeat*: Indicador de repetición. Se una para repetir la secuencia de animaciones. Su valor por defecto es *true*.
- *positions*: Coordenadas de la posición de cada marco o secuencia (x@y). Permite mover la animación alrededor. Cada par de coordenadas se separa por el carácter |.
- *images*: Índices de las imágenes. Permite repetir las imágenes de la animación. Cada número se encuentra separado por el carácter |.

- *soundsource*: URL del directorio que contiene los archivos de sonido.
- *soundtrack*: URL del archivo de sonido que suena "de fondo".
- *sounds*: Lista de URLs de archivos de sonido para cada secuencia de la *applet*. Se encuentran separados por el carácter |.

c.) Ejemplo

```
<applet code=Animator.class width=64 height=64>
<param name=imagesource value="tower">
<param name=endimage value=2>
<param name=soundsource value="audio">
<param name=soundtrack value=spacemusic.au>
<param name=sounds value="1.au|2.au">
<param name=pause value=200>
</applet>
```

C. GRÁFICOS INTERACTIVOS: "LINK BUTTON"

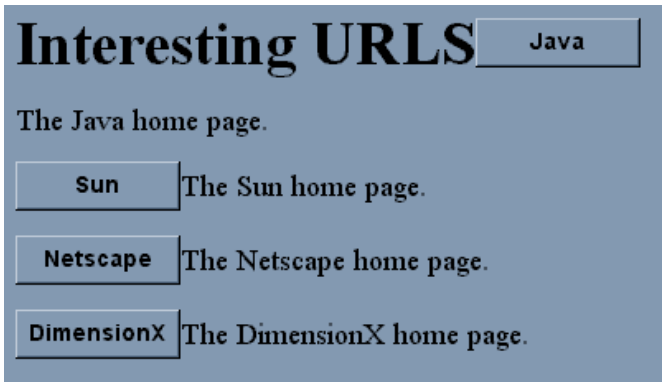

Imagen 15: Applet de gráficos interactivos "Link Button"

a.) Descripción

Esta *applet* permite colocar un botón en una página Web. Cuando se pulse el botón aparecerá una nueva página, o se reproducirá un determinado archivo de sonido,...

b.) Parámetros

- *href*: URL del documento o archivo al que hay que llamar cuando un usuario pulsa el botón. Este URL también puede hacer referencia a una posición concreta de la página actual.
- *snd*: URL del archivo de sonido que se va a reproducir cuando se pulse el botón.

c.) Ejemplo

```
<applet code=LinkButton.Java width=100 height=30>
<param name=lbl value="Java">
<param name=href value=http://www.Javasoft.com/>
```

```
<param name=snd value="computer.au">
</applet>
```

D. TRUCOS DE TEXTO: "NERVOUS TEXT"

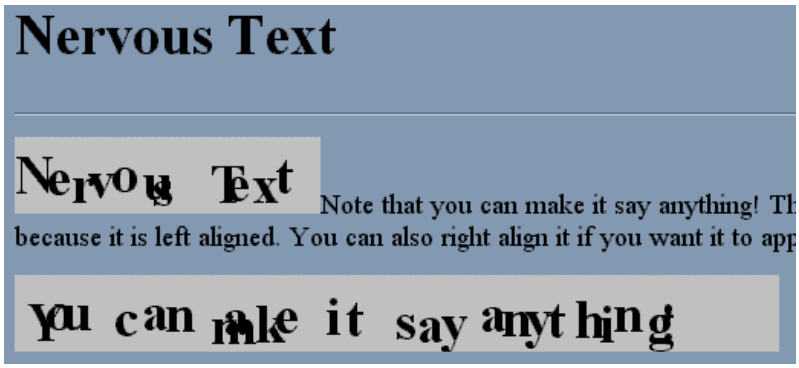

Imagen 16: Applet de texto animado "Nervous Text"

a.) Descripción

Esta *applet* muestra una línea de texto en la que las letras, aleatoriamente, se están desplazando de tal forma que se superponen con las letras contiguas.

Es algo muy sencillo pero, por otra parte, muy llamativo.

b.) Parámetros

- *text*: El texto (sólo una línea) que se mostrará en la *applet*.

c.) Ejemplo

```
<applet code="NervousText.class" width=200 height=50>
<param name=text value="hello World!">
</applet>
```

d.) Notas

Se necesitará establecer bien la anchura de la *applet* para que quepa toda la línea.

Puede servir para una firma en los mensajes de correo electrónico o de noticias, pero no podrá verse si el navegador no soporta Java.

E. FINANCIAS Y NEGOCIOS: "BAR CHART"

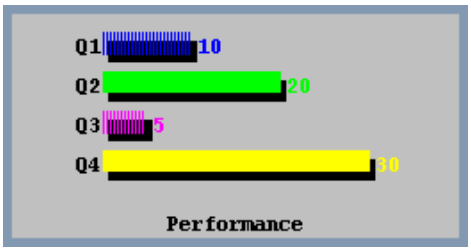

Imagen 17: Applet de financias y negocios "Bar Chart"

a.) Descripción

Esta *applet* muestra un gráfico de barras basado en los parámetros que recibe.

b.) Parámetros

- *title*: Título del gráfico. Aparecerá debajo de las gráficas.
- *colums*: Número de columnas (barras) en el gráfico.
- *orientation*: Posición de las barras: horizontales o verticales.
- *scale*: Escala de representación (en pixels por unidad de barra).
- *c<N>_style*: Textura de las barras: lisas o rayadas.
- *c<N>_value*: Unidades de medida: dólares, días...
- *c<N>_label*: Etiqueta de la barra: dinero, tiempo...
- *c<N>_color*: Color de la barra: verde, azul, rosa, naranja, magenta ,amarillo...

c.) Ejemplo

```
<applet code=”Chart.class width=251 height=125>
<param name=title value=”Performance”>
<param name=columns value=”4”>
<param name=orientation value=”horizontal”>
<param name=scale value=”5”>
<param name=c1_style value=”striped”>
<param name=c1 value=”10”>
<param name=c1_color value=”blue”>
<param name=c1_label value=”Q1”>
<param name=c2_color value=”green”>
<param name=c2_label value=”Q2”>
<param name=c2 value=”20”>
<param name=c2_style value=”solid”>
<param name=c3 value=”5”>
<param name=c3_style value=”striped”>
<param name=c3_color value=”magenta”>
<param name=c3_label value=”Q3”>
<param name=c4 value=”30”>
<param name=c4_color value=”yellow”>
<param name=c4_label value=”Q4”>
<param name=c4_style value=”solid”>
```

d.) Notas

Si se modifica la orientación (poniéndola en vertical) habrá que escoger una anchura y altura adecuadas para que todo el gráfico quepa dentro de la zona reservada para la *applet*.

El usuario no puede interactuar con el gráfico. Sólo se muestra en pantalla.

F. JUEGOS Y EDUCACIONALES: "GRAPH LAYOUT"

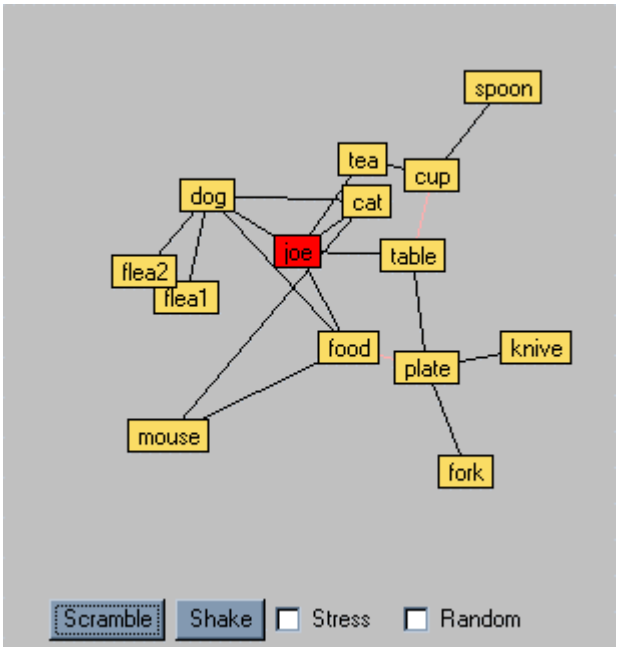

Imagen 18: Applet de juegos y educacionales "Graph Layout"

a.) Descripción

Es una *applet* que despliega un grafo, consistente en un conjunto de nodos y arcos.

Se pueden definir los nodos que se van a usar así como la longitud óptima de los arcos. El grafo está construido mediante un algoritmo heurístico.

b.) Parámetros

- *center*: Nodo central del grafo (en color rojo) que se sitúa en el centro de la pantalla.
- Los nodos se crean cuando se necesitan.
- *edges*: Arcos del grafo. Este parámetro consiste en una lista (separada por comas), de arcos. Cada arco se define mediante un par de nodos entre las etiquetas *origen-destino/longitud*, donde la longitud del arco (*longitud*) es opcional.

c.) Ejemplo

```
<applet code="Graph.class" width=400 height=400>
<param name=edges value="joe-food, joe-dog, joe-tea,
joe-cat, joe-table, table-plate/50, plate-food/30,
food-mouse/100, food-dog/100, mouse-cat/150, tab1e-cup/30,
```

```
cup-tea/30, dog-cat/80, cup-spoon/50, plate-fork,
dog-fleal, dog-f1ea2, f1ea1-f1ea2/20, p1ate-knive”>
<param name=center value="joe">
</applet>
```

d.) Notas

El usuario puede recoger nodos y distorsionar el grafo para acelerar el proceso del esquema.

APÉNDICES

APÉNDICE I. EL JDK (Java Development Kit)

A. INTRODUCCIÓN

JDK es el acrónimo de "Java Development Kit", es decir Kit de desarrollo de Java. Se puede definir como un conjunto de herramientas, utilidades, documentación y ejemplos para desarrollar aplicaciones Java.

Para la realización de este tutorial se ha trabajado con la versión 1.2.0 del JDK.

B. COMPONENTES DEL JDK

a.) Introducción

JDK consta de una serie de aplicaciones y componentes, para realizar cada una de las tareas de las que es capaz de encargarse

A continuación se explican más en profundidad cada uno de ellos, así como su sintaxis, indicando entre corchetes aquellos elementos que sean opcionales.

Se observará que todos los programas permiten la inclusión de una serie de opciones sobre su ejecución antes del primer argumento. Estas opciones se indican precedidas de un menos(-):

```
programa -opcion1 -opcion2 Parametro1
```

Todas las opciones que los ejecutables del JDK presentan se muestran llamando al programa sin parámetros o con las opciones *-?* o *-help*:

```
programa
programa -help
programa -?
```

b.) Intérprete en tiempo de ejecución (JRE)

Permite la ejecución de los programas Java (*.class) no gráficos (aplicaciones).

La sintaxis para su utilización es la siguiente:

```
java [Opciones] ClaseAEjecutar [Argumentos]
```

- *Opciones*: Especifica opciones relacionadas con la forma en que el intérprete Java ejecuta el programa.
- *ClaseAEjecutar*: Especifica el nombre de la clase cuyo método *main()* se desea ejecutar como programa. Si la clase reside en un paquete se deberá especificar su ruta mediante en forma *paquete.subpaquete.clase_a_ejecutar*.
- *Argumentos*: Especifica los argumentos que se recibirán en el parámetro *s* del método *main(String s)*, por si el programa necesita de parámetros de ejecución. Si por ejemplo el programa realiza el filtrado de un archivo, probablemente nos interese recibir como argumento la ruta del fichero a filtrar, y una ruta destino.

c). Compilador

Se utiliza para compilar archivos de código fuente Java (habitualmente *.java), en archivos de clases Java ejecutables (*.class). Se crea un archivo de clase para cada clase definida en un archivo fuente.

Este compilador es una utilidad en línea de comandos con la siguiente sintaxis:

```
javac [Opciones] ArchivoACompilar
```

- *Opciones*: Especifica opciones de cómo el compilador ha de crear las clases ejecutables.
- *ArchivoACompilar*: Especifica la ruta del archivo fuente a compilar, normalmente una fichero con extensión *".java"*.

d.) Visualizador de applets

Es una herramienta que sirve como campo de pruebas de applets, visualizando cómo se mostrarían en un navegador, en lugar de tener que esperar.

Al ser activado desde una línea de órdenes abre una ventana en la que muestra el contenido de la *applet*.

Se activa con la sintaxis:

```
appletviewer [Opciones] Applet
```

- *Opciones*: Especifica cómo ejecutar la applet Java.
- *Applet*: Indica un URL o una ruta de disco que contiene una página HTML con una applet Java empotrada.

e.) Depurador

Es una utilidad de línea de comandos que permite depurar aplicaciones Java.

No es un entorno de características visuales, pero permite encontrar y eliminar los errores de los programas Java con mucha exactitud. Es parecido en su funcionamiento al depurador *gdb* que se incluye con las distribuciones del compilador *gcc/g++* para C/C++.

Se activa con la sintaxis:

```
jdb [Opciones]
```

- *Opciones*: Se utiliza para especificar ajustes diferentes dentro de una sesión de depuración.

f.) Desensamblador de archivo de clase

Se utiliza para desensamblar un archivo de clase. Su salida por defecto, muestra los atributos y métodos públicos de la clase desensamblada, pero con la opción *-c* también desensambla los códigos de byte, mostrándolos por pantalla. Es útil cuando no se tiene el código fuente de una clase de la que se quisiera saber cómo fue codificada.

La sintaxis es la siguiente:

```
javap [Opciones] [NombresClases]
```

- *Opciones*: Especifica la forma en la que se han de desensamblar las clases.

- *NombresClase*: Especifica la ruta de las clases a desensamblar, separadas por espacios.

g.) Generador de cabecera y archivo de apéndice

Se utiliza para generar archivos fuentes y cabeceras C para implementar métodos Java en C (*código nativo*). Esto se consigue mediante la generación de una estructura C cuya distribución coincide con la de la correspondiente clase Java.

El generador de cabeceras *javah*, crea los ficheros de cabecera C/C++ para implementar en esos lenguajes los métodos nativos que presente un programa Java.

La sintaxis es la siguiente:

```
javah [Opciones] NombreClase
```

- *NombreClase*: Nombre de la clase desde la cuál se van a generar archivos fuente C.
- *Opciones*: Forma en la que se generarán los archivos fuente

h.) Generador de documentación

Es una herramienta útil para la generación de documentación API directamente desde el código fuente Java. Genera páginas HTML basadas en las declaraciones y comentarios *javadoc*, con el formato */** *comentarios* **/*:

```
/** Comentarios sobre la clase
     @autor: Ignacio Cruzado
  */
class MiClase {
};
```

La documentación que genera es del mismo estilo que la documentación que se obtiene con el JDK.

Las etiquetas, que se indican con una arroba (@), aparecerán resaltadas en la documentación generada.

Su sintaxis es:

```
javadoc Opciones NombreArchivo
```

- *Opcione*s: Opciones sobre qué documentación ha de ser generada.
- *NombreArchivo*: Paquete o archivo de código fuente Java, del que generar documentación.

i.) Applets de demostración

El JDK incluye una serie de applets de demostración, con su código fuente al completo.

j.) Código fuente de la API

El código fuente de la API se instala de forma automática, cuando se descomprime el JDK, aunque permanece en formato comprimido en un archivo llamado "*scr.zip*" localizado en el directorio Java que se creó durante la instalación.

C. USO DEL JDK

Ya se han visto las diferentes partes de que está compuesto el JDK, pero para el desarrollo de una aplicación final Java (ya sea una aplicación o una *applet*), deberemos utilizar las diferentes herramientas que nos proporciona el JDK en un orden determinado.

En el siguiente diagrama podemos ver la sucesión de pasos para generar un programa final Java:

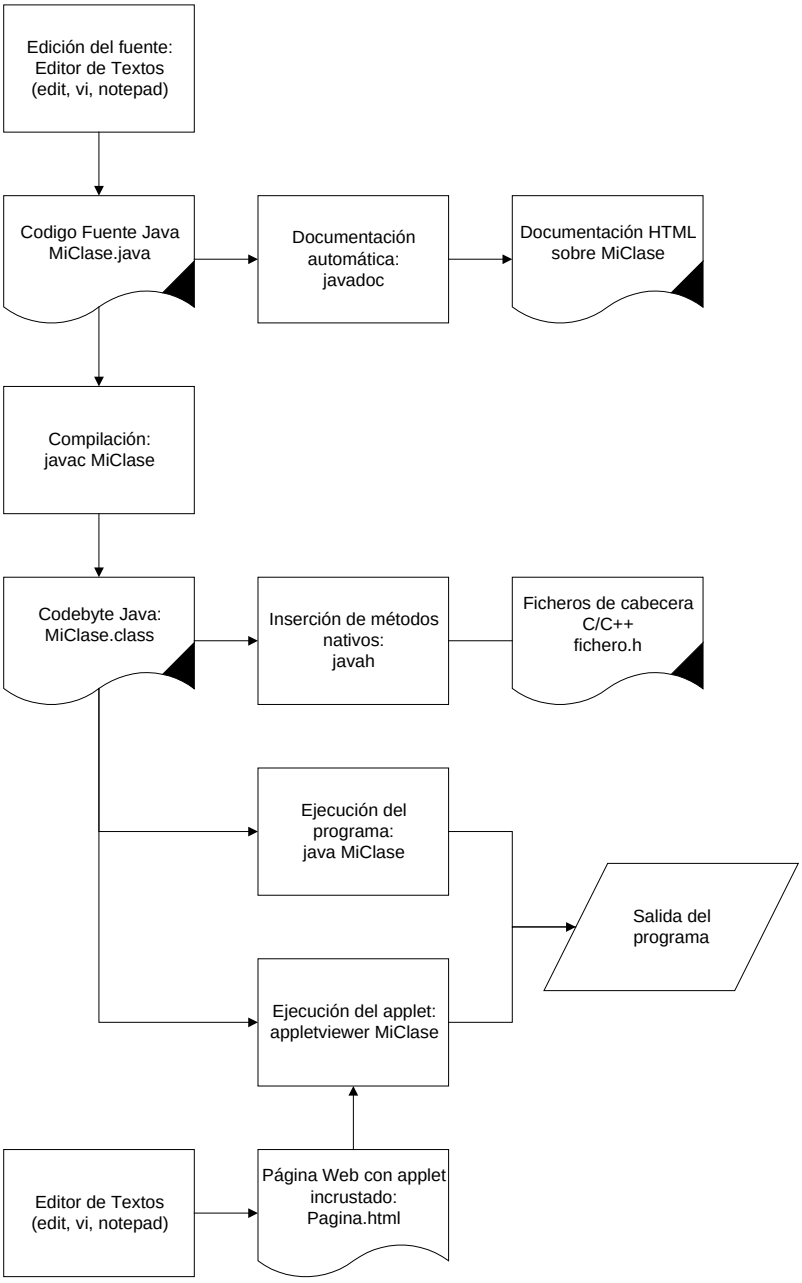

Imagen 19: Utilización del JDK

D. OBTENCIÓN E INSTALACIÓN DEL JDK

El JDK se puede obtener de las páginas de Sun (*http://java.sun.com*), y existen versiones disponibles para varias plataformas entre las que se encuentran:

- Microsoft Windows 95 y NT 4.0
- Sun Solaris 2.4 SPARC o 2.5 al 2.6 sobre x86 o SPARC.
- IBM AIX, OS/400 y OS/390
- Linux

Si su sistema operativo no ha sido enumerado, por favor consulte a su fabricante, pues Sun tiene previsto desarrollar su JDK para más plataformas.

La instalación es diferente para cada sistema operativo, pero en general muy sencilla. Se recomienda observar y establecer los valores de las variables de entorno:

- *PATH*: Variable de entorno que indica desde qué ruta (además del directorio actual) se pueden ejecutar los programas
- *CLASSPATH*: Indica al compilador Java en qué rutas se encuentran los ficheros de clase.

E. NOVEDADES EN LA VERSIÓN 1.2 DEL JDK (JAVA 2)

La aparición de la versión 1.2 del JDK (diciembre de 1997) significa un salto importante en las capacidades de Java, hasta tal punto que comercialmente se conoce a esta evolución como "Java 2".

a.) Clases

Se amplía el paquete de clases de JFC(Java Foundation Classes) pasando de 23 a 70 clases. Aparecen nuevos paquetes de la API Java:

- *Swing*: Nuevo paquete de gráficos.
- *Java 2D*: Ampliación de AWT.
- *Java Collections*: Incluye nuevas clases que representan estructuras de datos clásicas de la programación: *Vector, ArrayList, ArraySet, TreeMap*...

b.) Eficiente ejecución

- Compatible con programas realizados en otras versiones, tanto en código como en ejecución.
- Compilador más estricto y que genera un código más optimizado.
- Entorno de ejecución más rápido y estable (máquina virtual), próximo a la velocidad de ejecución de C++, especialmente utilizando los nuevos compiladores Just In Time (JIT), incorporados en las nuevas JRE, que compilan las clases para las plataformas locales, aumentando su velocidad de ejecución.
- Mejora en la gestión de la seguridad: Control de acceso basado en el plan de acción, soporte para certificados X509v3 y nuevas herramientas y certificados de seguridad.
- Mejor tratamiento de sonido.

- Más velocidad de las clases que utilizan RMI.
- Y por supuesto se solventan errores de versiones pasadas.

c.) JavaBeans

- *Java Plug-in*: Permite que las applets y los JavaBeans de una red utilicen el JRE 1.2, en vez de la máquina virtual del navegador en que se muestren.
- Se permite a los JavaBeans interactuar con applets y se les da un mejor soporte en tiempo de diseño y en tiempo de ejecución.
- Los JavaBeans se pueden incluir unos en otros.

d.) Otros

- *IDL (Interface Definition Language)*: Para interactuar con CORBA de una forma simple y práctica se utiliza Java.
- *JCE (Java Criptography Extensions)*: Se ofrecen mejores posibilidades para el tratamiento seguro de la información.
- *RMI (Remote Method Invocation)*: Permite realizar acciones sobre objetos remotos, incluso mediante SSL (Security Socket Layer) un conocido sistema de seguridad de comunicación.
- Tecnología de ayuda.
- Mejor soporte de *arrastrar y soltar*.
- Otros servicios varios.

e.) Observaciones

Aunque no existen incompatibilidades importantes, sí que se pueden observar que algunas cosas que en otras versiones no funcionaban bien o se permitían ahora se desaprueban. Entre otros conviene destacar:

- No se pueden declarar métodos abstractos como *native*, *private*, *final* ni *syncronzed.*
- No se recomienda la utilización de *finalize{}* en la gestión de excepciones.
- El paquete *Swing* en algunas versiones intermedias aparecía colgando de *com.sun.java.** o de *java.awt.swing.** y ahora pasa a ser *javax.swing.**.

APÉNDICE II: HERRAMIENTAS DE DESARROLLO

A. PROGRAMAS DE NAVEGACIÓN

a.) Introducción

Las applets Java no serían de mucha utilidad sin programas de navegación compatibles con Java. Por lo tanto, para que Java funcione necesita de estos programas de navegación, que afortunadamente se han comprometido a apoyarlo.

b.) Netscape Navigator

Es un programa de navegación con apoyo completo a Java. Además del simple apoyo al lenguaje y sistema de tiempo de ejecución, también ha ayudado en el desarrollo de *JavaScript*, que es un lenguaje de comandos basado en objetos Java. El objetivo de *JavaScript* es permitir el desarrollo rápido de aplicaciones distribuidas cliente servidor.

Para más información consultar http://home.es.netscape.com/

c.) Microsoft Internet Explorer

Microsoft tardó un poco en desarrollar una herramienta para Java: Internet Explorer. Está estrechamente ligado al sistema operativo Microsoft Windows 95, y está completamente integrado en la versión Windows 98 del mismo.

Para más información consultar http://www.microsoft.com/

d.) HotJava

Es el contendiente de Sun. Se diseñó inicialmente como un experimento en el desarrollo del programa de navegación de Java. Se ha convertido en un prometedor modelo de lo que depara el futuro para los programas de navegación de la Web. Será el programa de navegación existente más compatible con Java. Constituye un útil campo de pruebas para los programadores de Java.

Es capaz de gestionar e interactuar de forma dinámica con nuevos tipos de objeto y protocolos Internet.

Para más información consultar http://www.sun.com/

e.) Spyglass Mosaic

Fue el primer navegador de Internet, y ya está disponible con apoyo a Java.

Para más información consultar http://www.spyglass.com/

B. ENTORNOS DE DESARROLLO

a.) Introducción

Los desarrolladores se han acostumbrado a las herramientas gráficas de programación, y aunque el JDK es suficiente para desarrollar Java, se han creado muchos entornos de desarrollo (*IDEs*) para este lenguaje.

La mayor parte de los participantes en el negocio de herramientas de programación han anunciado algún tipo de entorno de desarrollo para Java. Parte de ese apoyo ha llegado en forma de módulos adicionales para productos ya existentes, mientras que otra parte consistirá en productos totalmente nuevos.

En este apartado se comentan algunos de los más valorados, y se indican otros por si el lector desea buscar algo con unas características muy específicas.

b.) Visual Café de Symantec

La empresa Symantec, dura competidora en los IDE de Java ofrece el entorno visual de desarrollo Java Visual Café, que ya goza de una gran reputación entre los desarrolladores de Java.

Integra de forma transparente las herramientas del JDK, presentándolas en formato gráfico.

Las principales funciones que soporta son las siguientes:

- Editor de programación gráfica: Posee todas las características de un moderno editor de programación: sintaxis a todo color, resaltado de palabras clave, lenguaje macro integrado para ampliar el editor...
- Editor de clases y jerarquías: Navega con rapidez a través del código Java, permitiendo trabajar directamente con clases o miembros de clases en lugar de archivos individuales; este editor localiza el código fuente correspondiente y lo carga. Además gestiona y visualiza las relaciones lógicas entre clases.
- Depurador gráfico: Gran ventaja sobre el *jdb* que es modo línea.
- Gestor de proyectos: Permite organizar proyectos Java con mayor efectividad. Soporta proyectos dentro de proyectos, por lo que puede mantenerse al día con bibliotecas anidadas y dependencias de proyectos.
- Asistentes: Para creación de eventos, bases de datos, applets...

Para más información, consultar http://cafe.symantec.com/

c.) Visual J++ de Microsoft

Es la nueva herramienta de Microsoft para desarrollar Java en sus sistemas operativos Microsoft Windows. Se encuentra incluido en el paquete de desarrollo *Microsoft Visual Studio*, y es directo heredero del tan extendido *Microsoft Visual C++*.

Presenta el serio problema de que no respeta las especificaciones de clases de Sun, lo que ha llevado a ambas compañías a juicio; en lugar de utilizar las clases del JFC, Microsoft se ha inventado un nuevo paquete WFC(Windows Foundation Classes), para el desarrollo en la plataforma Windows, rompiendo la portabilidad.

Para más información consultar http://www.microsoft.com/

d.) JBuilder de Borland

La empresa *Borland* es la desarrolladora de populares entornos de desarrollo de C++ y Delphi para Windows. Borland ha optado por desarrollar un producto totalmente nuevo para los desarrolladores de Java; *JBuilder*.

JBuilder ha sido desarrollado totalmente en Java, lo que permite a Borland salir del mercado del PC y comercializar *JBuilder* en todas las plataformas soportadas por Java.

Presenta gran conectividad con las bases de datos, soportando incluso CORBA. Tiene un programa de desarrollo de JavaBeans con más de 200 prediseñados.

Realmente es un producto muy completo de desarrollo de Java, y se distribuye en dos versiones (standard y Cliente/Servidor).

Para más información, consúltese http://www.borland.com/jbuilder/

e.) Java Studio de Sun

Entorno muy intuitivo y práctico, en el que casi la mayoría de las tareas se realizan mediante el ratón. Es muy fácil crear aplicaciones sencillas con este IDE, pero se necesita tener una versión del JDK previamente, y tiene unos requisitos hardware bastante altos.

Para más información consúltese http://www.sun.com/

f.) VisualAge for Java de IBM

Es una RAD (*Rapid Aidded Design*), que aunque tiene muchas de las características del Visual Café, con una interfaz un poco más limpia. Permite diseñar la interfaz de la aplicación o *applet*, y definiendo el sistema de eventos, la herramienta puede crear código Java.

Es muy sencillo de manejar y uno de los más potentes del mercado.

Para más información, consúltese http://www.ibm.com/ad/vajava/

g.) Y muchos más...

Los IDE están en continuo desarrollo, y seguro que tras la finalización de este tutorial ya han aparecido muchos más en el mercado.

Algunos de ellos son:

- ***CodeWarrior de Metrowerks***: Entorno de desarrollo Java para Macintosh, basado en *Codewarrior C++*. http://www.metrowerks.com/
- ***Roaster de Natural Intelligence***: Entorno de desarrollo Java para Power Macintosh. http://www.natural.com/page/products/roaster/
- ***Cosmo de Silicon Graphics***: Conjunto de herramientas de desarrollo *Cosmo* con dos bibliotecas propias. http://www.sgi.com/products/cosmo/
- ***SuperCede de Asymetrix***: Un producto a bajo precio, con algunos tutoriales. http://www.asymetrix.com/sales/
- ***Java Maker***: Sencillo entorno de desarrollo que funciona bajo Windows 95/NT, creado por Heechang Choi. http://net.info.samsung.com.kr/~hcchoi/Javamaker.html
- ***Ed de Soft As It Gets***: Editor muy potente para Windows, aunque flojo en otros aspectos. http://www.ozwmail.com.au/~saig
- ***Mojo de Penumbra Software***: Entorno visual para crear applets, fácil de usar. http://www.PenumbraSoftware.com

- ***Jamba de AimTech***: Constructor gráfico de applets, con mucha documentación. http://www.aimtech.com/prodjahome.html

La información de este apartado ha sido extraída de **[Rojo, 1998], [Morgan, 1999]** y **[Zolli, 1997]**.

Se recomienda al lector que busque en Internet en las direcciones:

- http://www.developer.com/news/userchice/n_userframe.html: Lista de los entornos de desarrollo preferidos por los desarrolladores de Java.
- http://www.yahoo.com/Business_and_Economy/Companies/Computes/Software/Programming_Tools/Languages/Java: Lista de últimas herramientas Java.

C. BIBLIOTECAS DE PROGRAMACIÓN

Java está orientado a objetos, por lo que es importante no ignorar el potencial de volver a utilizar objetos Java. De hecho ya están apareciendo algunas bibliotecas comerciales de objetos Java.

- Por ejemplo, la empresa Dimensión X cuenta con tres bibliotecas de clases Java:
- Ice: Paquete de representación de gráficos tridimensionales.
- Liquid Reality: Kit de herramientas VRML.
- JACK: Herramienta para crear applets Java a través de una interfaz sencilla.

APÉNDICE III: MÉTODOS NATIVOS JAVA (JNI)

A. INTRODUCCIÓN

Aunque la potencia de la API de Java es más que suficiente para casi todo tipo de aplicaciones, algunas de ellas necesitan utilizar la potencia específica de una plataforma en concreto, por ejemplo para conseguir un poco más de velocidad y eficiencia.

Java permite incorporar en sus programas fragmentos de *código nativo* es decir, código compilado para una determinada plataforma, generalmente escrito en C/C++. Así se puede utilizar código específico de una plataforma, bibliotecas ya escritas...

Para ello Java cuenta con el JNI (*Java Native Invocation*). Hay que tener en cuenta que una aplicación que utilice este tipo de métodos estará violando las directrices de seguridad de la máquina virtual Java, motivo por el que no se permite incluir métodos nativos en *applets* de Java.

Para agregar métodos nativos Java a una clase de Java han de seguirse los siguiente pasos:

1. Escritura del programa Java, invocando métodos nativos como *native*.
2. Compilación del programa Java.
3. Creación de un archivo de cabecera nativo (*.h*)
4. Escritura de los métodos nativos.
5. Creación de una biblioteca con esos métodos nativos.
6. Ejecución del programa Java.

B. EJEMPLO DE USO DE MÉTODOS NATIVOS

Para mostrar cómo utilizar los métodos nativos, vamos a crear un pequeño programa, escrito con métodos nativos, que lo único que hace es imprimir *"¡Hola Mundo!!!"*. Para ello vamos a utilizar el JDK y un compilador de C.

Se advierte al lector que no se deje engañar por la simpleza del ejemplo que se va a desarrollar, porque la potencia del JNI va mucho más allá de lo que estas líneas dejan entrever. Para más información consulte **[Morgan, 1999]**.

a.) Escritura del programa Java

Escribimos el programa Java en un fichero denominado *HolaNativo.java.*

El código Java que vamos a utilizar será:

```
puclic class HolaNativo{
  public native void diHola();
  static {
    System.loadLibrary("LibHola");
  }
  public static void main( String[] args ) {
```

```
    new HolaNativo().diHola();
  }
}
```

El método nativo *diHola()* no tiene cuerpo porque será añadido mediante una biblioteca nativa denominada *LibHola*. Dicha biblioteca nativa es cargada mediante la sentencia *loadLibrary()*, sentencia que ha sido incluida como *static* para que sea ejecutada cada vez que se cree una instancia (objeto) de esta clase.

El programa principal tan sólo instancia un objeto de esta clase y utiliza el método nativo que imprime la cadena de *"¡Hola Mundo!!!"*.

b.) Compilación del programa Java

Ahora ya es posible compilar la clase Java *HolaNativo* que fue creada mediante la sentencia (utilizando el JDK):

```
javac HolaNativo.java
```

c.) Creación de un fichero de cabecera nativo (.h)

Un fichero de cabecera nativo es un fichero que habitualmente tiene la extensión *".h"*. En un fichero de este tipo se definen en C/C++ las interfaces públicas (clases, métodos o funciones, variables globales, constantes...).

La herramienta *javah* incluida en el JDK es capaz de crear automáticamente un fichero de cabecera para métodos nativos Java, con sólo invocarla indicando el nombre de la clase de la que extraer las cabeceras nativas:

```
javah HolaNativo
```

Esta operación crea un fichero *HolaNativo.h* que será útil para crear los métodos nativos. Hay dos líneas importantes dentro de este fichero:

```
#include <jni.h>
JNIEXPORT void JNICALL Java_HolaNativo_diHola(JNIEnv*,jobject);
```

La primera línea importa una biblioteca JNI que valdrá a C/C++ para saber cómo crear los métodos nativos.

La segunda línea corresponde al método nativo que definimos.

Los métodos nativos suelen denominarse siempre como Java_Paquete_Clase_Metodo() aunque en este caso al no haber paquete esta parte se ha omitido.

Así mismo los métodos nativos reciben como parámetros *JNIEnv** y *jobject* que permitirán al método nativo comunicarse con su entorno.

d.) Escritura de los métodos nativos

Se ha de crear un fichero fuente nativo, en el que se defina el cuerpo de la función que actuará como método nativo. Este fichero lo denominaremos *HolaNativo.c*:

```
#include <jni.h>
#include "HolaNativo.h"
#include <stdio.h>
```

```
JNIEXPORT void JNICALL Java_HolaNativo_DiHola( JNIEnv* e,
jobject o ){
  printf("¡Hola Mundo!!!\n");
}
```

En este programa se incluyen tres bibliotecas: La de JNI (*jni.h*), el fichero de cabecera ya creado (*HolaNativo.h*) y una biblioteca de C para imprimir (*stdio.h*).

Ser puede observar que el cuerpo del método nativo lo que hace es invocar a la función de C *printf()* que imprimirá por pantalla la cadena *"¡Hola Mundo!!!"*.

e.) Creación de una biblioteca con esos métodos nativos

Cada compilador de C o C++ tiene su propia herramienta para crear bibliotecas compartidas (DLL en Windows o SO en UNIX).

Para crear la biblioteca se ha de compilar el fichero fuente nativo, y luego crear la biblioteca compartida mediante el programa correspondiente a su compilador.

Para esto cada compilador tiene su propia sintaxis, con lo que se tendrá que consultar la documentación del compilador en lo referente a la creación de bibliotecas. En cualquier caso algunos ejemplos de compilación en bibliotecas son:

Para el GCC de Solaris:

```
cc -G HolaNativo.c -o libHola.so
```

Para Microsoft Visual C++ para Windows:

```
cl -LD HolaNativo.c -Fe libHola.dll
```

En cualquier caso asegúrese que la biblioteca creada tiene el mismo nombre con que se la invoca desde el archivo de clase Java en el método *loadLibrary()*.

f.) Ejecución del programa Java

Para ejecutar este programa, debe invocarse:

```
java HolaNativo
```

Con lo que se muestra por pantalla:

```
Hola Mundo
```

Se puede observar que la forma de creación, compilación y ejecución de la clase Java es igual que la utilizada para crear una aplicación Java normal, solo que al incluir métodos nativos, han de crearse bibliotecas y ficheros de cabecera antes de ejecutar el programa.

APÉNDICE IV: GUÍA DE REFERENCIA DE C++ A JAVA

A. INTRODUCCIÓN

La sintaxis de Java resulta muy familiar a los programadores de C++, debido mayoritariamente a que Java proviene de C++. Sin embargo Java pretende mejorar a C++ en muchos aspectos (sobre todo en los aspectos orientados a objeto del lenguaje), aunque prohibe muchas de las tareas por las que C++ fue tan extendido.

Se observa que las diferencias han sido diseñadas como mejoras del lenguaje, ya que uno de los aspectos más criticado (y defendido) de C++ es su capacidad para hacer cosas "no orientadas a objetos", así como acceder a los recursos de las máquinas (lo que le permitía atacar sistemas, siendo uno de los lenguajes más difundidos entre los programadores de virus).

En este apéndice pretendemos mostrar aquellas diferencias significativas, para que los programadores familiarizados con C++ puedan programar en Java, conociendo sus posibilidades y limitaciones.

B. DIFERENCIAS SINTÁTICAS

a.) Elementos similares

Uso de bibliotecas (paquetes): Como no existen macros como *#include*, se utiliza *import*.

Operador de herencia: La herencia en Java se especifica con la palabra clave *extends*, en lugar del operador :: de C++.

Constantes: En lugar de *const* de C++ las variables constantes se declaran como *static final*.

b.) Elementos equivalentes

Miembros estáticos: No es necesaria declaración previa (fuera de las clases) de los miembros estáticos (*static*).

Métodos inline: En Java no existen (hay que incluir el cuerpo de los métodos junto a su definición), aunque los compiladores Java suelen intentar expandir en línea (a modo de los métodos inline) los métodos declarados como *final*.

Métodos virtuales: Todos los métodos no estáticos (*static*) se consideran *virtual* en Java.

Forward: No existe en Java. Simplemente se llama el método, y el compilador se encarga de buscarlo.

Clases anidadas: Aunque Java no permite anidar clases, sí que se puede modelizar este concepto mediante los paquetes Java y la composición.

No existen "amigos" (*friend*): Es su lugar se considera amigas a todas las clases y elementos que componen un paquete.

c.) Elementos añadidos

Comentarios: Java soporta los dos tipos de comentarios de C++, e incorpora un tercero, con la sintaxis */** *comentario* **/*, para la documentación automática.

Operador >>> : Java añade este operador para desplazamientos a la derecha con signo.

Iniciación: Todos las variables de tipo simple y las referencias a objeto se inician a un valor 0 (o equivalente), y *null* para las referencias a objeto.

Referencia *super*: En Java *super* hace referencia a la superclase (clase padre) de la clase actual. Dicha clase sólo puede ser una, porque Java solo soporta herencia simple.

d.) Elementos suprimidos

goto: No existe en Java, aunque con *break* y *continue*, combinados con etiquetas de bloque, se puede suplir.

Condiciones: Deben utilizarse expresiones booleanas y nunca números.

Argumentos por defecto: Java no los soporta.

C. DIFERENCIAS DE DISEÑO

a.) Tipos de datos

Tipos simples: Soporta los mismos que C++, añadiendo *boolean* (*true/false*), y ampliando el tipo *char*, para soportar caracteres Unicode de 16 bits.

Punteros: En Java no hay punteros, permitiendo así programación segura. En su lugar se crean las referencias a objeto, que pueden ser reasignadas (como si fueran un puntero a objeto de C++).

Vectores: Son objetos de sólo lectura, con un método *length()* para averiguar su longitud, y que lanzan una excepción si se intenta acceder a un elemento fuera del vector.

Clases: Todo debe de estar incluído en clases; no existen enumeraciones (*enum*) ni registros (*struct*).

Elementos globales: No existen variables o funciones globales, aunque se puede utilizar *static* para simularlas.

b.) Diseño de las clases

Cuerpo de los métodos: Todos los cuerpos de las clases han de estar codificados en las definiciones de las clases. No se pueden separa como se hace en C++ mediante ficheros de cabecera (*".h"*).

Constructores: Aunque existen constructores, como en C++, no existen constructores copia, puesto que los argumentos son pasados por referencia.

Destructores: No existen destructores en Java, ya que tiene recolección automática de basura, aunque en su lugar se pueden escribir métodos *finalize()*, que serán ejecutados cuando el recolector de basura de Java destruya el objeto.

Árbol de herencia: En Java todas las clases se relacionan en un único árbol de herencia, en cuya cúspide se encuentra la clase *Object*, con lo que todas las clases heredan de ella. En C++ sin embargo se pueden declarar clases que no tengan padre.

Herencia no restrictiva: Al heredar, no se puede reducir las ocultaciones del padre: En C++ sí se podría ampliar la visibilidad de uno de los elementos heredados. En todo caso sí se puede restringir.

Herencia simple de clases: No existe la herencia múltiple de clases. Aún así se puede implementar una herencia múltiple utilizando interfaces, dado que ellas sí la soportan.

Sobrecarga de métodos: Es exactamente igual que la de C++.

Sobrecarga de operadores: No existe. En los objetos *String* el operador + y += se permiten para la comparación de cadenas.

c.) Nuevos diseños

Plantillas (*templates*): En Java no se soportan plantillas p clases genéricas de C++, aunque existen una serie de clases en la API de Java que tratan objetos genéricos (clase *Object*) como *Vector* o *Stack*.

Interfaces (*interface*): Que son unas especies de *clases abstractas* con *métodos abstractos*, y que permiten herencia múltiple, utilizando la palabra reservada *implements*.

Diferente gestión de excepciones: Todas las excepciones de Java heredan de la clase *Throwable*, que las dota de una interfaz común.

D. DIFERENCIAS DE EJECUCIÓN

a.) Aspectos modificados:

Cadenas: Las cadenas entrecomilladas se convierten en objetos *String*, no en vectores estáticos de caracteres.

Instanciación: Los objetos se crean en el montículo (*new*) y nunca en la pila (*malloc*), y los tipos simples no permiten *new* (excepto los vectores de tipos simples, iguales que los de C++).

Intérprete: Los intérpretes Java son unas 20 veces más lentos que los de C, aunque esta diferencia se está reduciendo con lo compiladores JIT(Just In Time) para Java que están apareciendo en el mercado.

Bibliotecas estándar: En C++ existía casi una biblioteca por plataforma (si existía) para hacer cosas como: Trabajo en red, conexión a bases de datos, uso de múltiples hilos de control, uso de objetos distribuídos o compresión. Java incorpora bibliotecas estándar multiplataforma para todas estas tareas en su API.

Excepciones por fallos de descriptores: Java lanza excepciones cuando hay errores en el acceso a un descriptor, permitiendo al programador gestionar dichos fallos, y recuperar al programa de ellos.

Gestión de errores al compilar: Además comprobar el lanzamiento de excepciones en tiempo de compilación, comprueba el cumplimiento del lanzamiento de excepciones por los métodos sobreescritos.

b.) Aspectos eliminados:

Preprocesador: Java no tiene preprocesador, por lo que las macros (*#include, #define,...*) no existen.

Acceso directo al hardware: En Java está restringido, aunque para eso permite la utilización de *métodos nativos*, escritos para la plataforma (normalmente C/C++). En cualquier caso las *applets* no pueden utilizar estos *métodos nativos*, sólo las aplicaciones Java pueden hacerlo.

c.) Aspectos introducidos

Multiples hilos de control (*multithreating*): Java permite la utilización de múltiples hilos de control y la ejecución en paralelo (y sincronizada) de múltiples tareas, mediante la clase *Thread*.

Applets Java: Este tipo de aplicaciones son seguras, distribuíbles por Internet y ejecutables por los navegadores, aunque tienen restricciones (como la escritura en disco).

Extracción automática de documentación: Un nuevo tipo de comentario (*/**com_doc*/*) permite a los programadores extraer de manera automática comentarios de sus fuentes, generando automáticamente documentación estandarizada.

JavaBeans: Mediante esta biblioteca se permite crear elementos visuales multiplataforma, algo impensable en C++.

APÉNDICE V: GUÍA DE REFERENCIA DEL LENGUAJE JAVA

A. FUNDAMENTOS

Palabras reservadas (funcionales):

abstact	*boolean*	*break*	*byte*	*case*
catch	*char*	*class*	*continue*	*default*
do	*double*	*else*	*extends*	*false*
final	*finally*	*float*	*for*	*if*
implements	*import*	*instanceof*	*int*	*interface*
long	*native*	*new*	*null*	*package*
private	*protected*	*public*	*return*	*short*
static	*super*	*switch*	*syncroniced*	*this*
throw	*throws*	*transient*	*try*	*void*
volatile	*while*			

Tipos de comentarios:

`/*comentario*/`	`// Hasta fin de línea`	`//* javadoc */`

Bloques de código:

```
{  // conjunto de sentencias
}
```

Separadores Java:

```
{   }   ,   :   ;
```

Propuestas de estilo de nombres de identificadores:

Las clases: *Clase* o *MiClase*.

Los métodos: *metodo()* o *metodoLargo()*.

Las variables: *altura* o *alturaMedia*.

Las constantes: *CONSTATE* o *CONSTANTE_LARGA*.

Los paquetes: *java.paquete.subpaquete*.

B. TIPOS DE DATOS

Tipo	bytes	Tipo Datos	Rango (positivo)	Literal
byte	1	Entero	127	14
short	2	Entero	32767	14
int	4	Entero	2.147.483.647	14
long	8	Entero	9.233e15	14
float	4	Coma flotante	1.4e-45 a 3.4e38	36,6
double	8	Coma flotante	4.9e-324 a 1.7e308	3,14e2
char	2	Caracter	Unicode	'a' o \064
boolean	1	Booleano	*true* o *false*	*true* o *false*

Vectores:

```
int vectorNumeros[]=new int[numero];
```

Cadenas Constates:

```
String nombreBonito = ”Amelia”;
```

Cadenas Variables:

```
StringBuffer cadenaVariable = ”Cambiante”;
```

C. OPERADORES

Operadores Unarios:

Operador	Descripción (prefija)	Operador	Descripción (pre o posfija)
+	Convierte el operador a *int*	++	Incrementa operador
-	Niega aritméticamente oper.	--	Decrementa operador

Operadores aritméticos (binarios):

Operador	Uso	Atajo	Descripción
+	*op1 + op2*	op1 += op2	Suma op1 y op2
-	*op1 - op2*	op1 -= op2	Resta op2 de op1
*	*op1 * op2*	op1 *= op2	Multiplica op1 por op2
/	*op1 / op2*	op1 /= op2	Divide op1 por op2
%	*op1 % op2*	op1 %= op2	Resto de op1 / op2

Operadores de comparación (binarios):

Operador	Uso	Devuelve verdadero si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos
&&	*op1 && op2*	AND, condicionalmente evalúa op2
&	*op1 & op2*	AND, siempre evalúa op1 y op2
||	*op1 || op2*	OR, condicionalmente evalúa op2
|	*op1 | op2*	OR, siempre evalúa op1 y op2
!	*! op*	op es falso

Operadores de bit (binarios):

Operador	Uso	Operación
>>	*op1 >> op2*	Desplaza los bits de op1 a la derecha op2 veces
<<	*op1 << op2*	Desplaza los bits de op1 a la izquierda op2 veces
>>>	*op1 >>> op2*	Desplaza los bits de op1 a la derecha op2 veces (sin signo)
&	*op1 & op2*	AND
|	*op1 | op2*	OR
^	*op1 ^ op2*	"XOR"
~	*~op2*	Complemento

Operador terciario:

```
expresion ? sentencia_si : sentencia_si_no
```

Precedencia de operadores:

Tipo de operadores	Operadores de este tipo
Operadores posfijos	[ ] . (parametros) expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
Creación o conversión	new (tipo) expr

Multiplicación	* / %
Suma	+ -
Desplazamiento	<<
Comparación	< <= = instanceof
Igualdad	== !=
AND a nivel de bit	&
OR a nivel de bit	^
XOR a nivel de bit	|
AND lógico	&&
OR lógico	||
Condicional	? :
Asignación	= += -= *= /= %= &= ^= |= <<= = =

D. ESTRUCTURAS DE CONTROL

Toma de decisión (*if-else* y *switch*):

```
if ( condición ) {
  Bloque de código a ejecutar si la condición es cierta
}
else {
  Bloque de código a ejecutar si no
}
switch ( expresionMultivalor ) {
  case valor1 : conjuntoDeSentencias; break;
  case valor2 : conjuntoDeSentencias; break;
  case valor3 : conjuntoDeSentencias; break;
  default :     conjuntoDeSentencias; break;
}
```

Bucles iterativos (*while*, *do-while*, *for*):

```
while ( expresiónBooleana ) {
  sentencias;
};
do {
  sentencias;
} while ( expresiónBooleana );
for(inicio; condicion_continuacion; sentencia_actualizacion) {
  sentencias;
}
```

Sentencias de saltos:

```
etiquetaSentencia: sentenciaEtiquetada
break nombreEtiqueta; // Sale del último bucle
continue; // Hace otra pasada al último bucle
```

```
return valor; // Sale del método devolviendo valor
```

E. CLASES

Definición de clase:

```
acceso class NombreDeClase herencia{
  acceso tipo nombreAtributo1;
  NombreDeClase{ // Constructor
  }
  // . . .
  finalize { //Recogida de basura
  }
  acceso tipo_devuelto nombreMétodo1( parámetros ) {
    cuerpo_del_método1;
  }
}
```

Tipos de acceso de las clases:

- *final*: Sin subclases
- *abstract*: Clase abstracta, luego no se permiten instancias de esta clase.
- *public*: Accesible desde fuera de su paquete

Herencia:

- *extends*: Clase padre. La clase *Object* es superclase de todas las clases Java (raíz del árbol de herencia).
- *implements*: Interfaces padres, separadas por comas; *Interface1, Interface2*.

```
class MiClase extends SuPadre implements Interface0,Interface1;
```

F. ATRIBUTOS

Acceso (igual que para métodos):

- *public*: Los miembros declarados *public* son accesibles en cualquier lugar en que sea accesible la clase, y son heredados por las subclases.
- *private*: Los miembros declarados *private* son accesibles sólo en la propia clase.
- *protected*: Los miembros declarados *protected* son accesibles sólo para sus subclases

Composición:

```
class claseCompuesta {
  claseComponente referenciaAObjetoComponente.
}
```

G. MÉTODOS

Referencias válidas en los métodos:

- *this*: Referencia al objeto actual.
- *super*: Referencia a la clase padre en el árbol de herencia.

Modificadores permitidos:

- Acceso: *public private* o *protected.* Igual que para los atributos.
- *abstract*: Método abstractos, sin cuerpo. Sólo se permiten en clases abstractas.
- *final*: No se puede sobreescribir el método.
- *static*: Método de la clase (no de los objetos).
- *native*: Método implementado con métodos nativos (específicos de una plataforma).
- *synchronized*: Solamente permite un hilo de ejecución.

Sobrecarga del método: Varias implementaciones en función de los parámetros.

Sobreescritura del método: Un cuerpo en cada nivel de herencia.

H. OBJETOS

Instanciación:

```
NombreDeClase referenciaAObjeto = new NombreDeClase();
```

Acceso al objeto:

```
referenciaAObjeto.método( parámetros );
referenciaAObjeto.atributo;
```

Destrucción: Cuando la referencia a objeto sale de ámbito en el programa.

I. INTERFACES

Declaración de una interfaz:

```
interface MiInterfaz {
  int CONSTANTE = 100;
  int metodoAbstracto( int p ); // Por definir
}
```

Implementación de interfaces:

```
class ImplementaInterfaz implements MiInterfaz{
  int m=CONSTANTE;
  int metodoAbstracto( int p ){ return ( p*m ); }
}
```

Herencia múltiple entre interfaces:

```
interface InterfazMultiple extends Interfaz0,Interfaz1;
```

Clases de envoltura de tipos simples:

Clases de envoltura de tipos simples	
Double	*double*
Float	*float*
Integer	*int, short, byte*
Long	*long*
Character	*char*
Boolean	*boolean*

J. PAQUETES

Creación de un paquete (primera sentencia de un fichero fuente):

```
package NombrePaquete;
```

Importación de un paquete o parte de él:

```
import Paquete.Subpaquete1.Subpaquete2.Clase1;
Paquete.Subpaquetes1.Subpaquete2.Clase_o_Interfaz.elemento
```

Visibilidad en los paquetes:

Situación del elemento	*private*	*por defecto*	*protected*	*public*
En la misma clase	Sí	Sí	Sí	Sí
En una clase en el mismo paquete	No	Sí	Sí	Sí
En una clase hija en otro paquete	No	No	Sí	Sí
En una clase no hija en otro paquete	No	No	No	Sí

K. EXCEPCIONES

Tipos de excepciones:

- *Error*: Excepciones que indican problemas muy graves, que suelen ser irrecuperables y no deben casi nunca ser capturadas.
- *Exception*: Excepciones no definitivas, pero que se detectan fuera del tiempo de ejecución.
- *RuntimeException*: Excepciones que se dan durante la ejecución del programa.

Lanzamiento de una excepción:

```
metodoLanzador() throws MiException{
  throw MiException:
}
```

Tratamiento de una excepción:

```
try {
  // Código posiblemente problematico
```

```
} catch( tipo_de_excepcion e) {
  // Código para solucionar la excepcion e
} catch( tipo_de_excepcion_mas_general e)
  // Código para solucionar la excepcion e
} finally {
  // Se ejecutara tras try o catch
}
```

L. THREADS

Creación:

- Para crear un *thread*, se ha de implementar una clase, extendiendo la clase base *Runnable*, y crear un objeto de la clase *Thread*.
- Este objeto representará un nuevo hilo de control, que será accionado cuando invoquemos al método *start()* del *thread*.
- En ese momento este hilo se activará, ejecutando (si el planificador de hilos considera que es el momento), el método *run()* de la clase en que todo esto suceda.

Sincronización de procesos:

- Durante la ejecución de un programa, muchas veces varios procesos han de realizar tareas de una forma sincronizada, actuando en un determinado orden.
- Para ello se han de declarar métodos como *syncronized.*
- Mediante la utilización de excepciones, y de las funciones *wait()* y *notifiy()*, respectivamente esperarán a que otro proceso acabe antes de continuar su ejecución.

GLOSARIO

Abstract Windowing Toolkit (AWT).- Biblioteca de módulos pera representar interfaces gráficos provisto por Sun en la API de Java.

Administrador de Seguridad.- Parte de la máquina virtual Java responsable de velar por el cumplimiento de las políticas y reglas de seguridad.

Ámbito.- Parte de un programa en el que es válida una referencia a una variable.

American Standard Code for Information Interchange (ASCII).- Sistema de codificación que convierte caracteres a números en el rango de 0 a 127. Es una parte del código ANSI que se amplía hasta los 257 caracteres.

Análisis.- Proceso de conocer los requerimientos de software que tienen el cliente y el usuario final.

API.- Application Programming Interface.

Aplicación.- Programa informático, que se ejecuta sin necesidad de otro programa

Applet.- Programa informático que se ejecuta necesitando de otro programa, normalmente un navegador.

Application Programming Interface (API).- Conjunto de paquetes y clases Java, incluidos en el JDK que utilizan los programadores Java para realizar sus aplicaciones.

Árbol.- Estructura de datos, grafo no cíclico, con forma de árbol (nodos padres e hijos).

Argumentos.- Parámetros.

Array.- Vector.

ASCII.- American Standard Code for Information Interchange.

AWT.- Abstract Windowing Toolkit.

BDK.- Beans Developer Kit.

Beans Developer Kit (BDK).- Conjunto de herramientas para desarrollar *JavaBeans*.

Bit.- Unidad mínima de información digital que puede tomar los valores lógicos de *0* o de *1*.

Bloque.- Código localizado entre corchetes.

Boolean.- Tipo de datos bi-estado, que puede tomar valor de cierto (*true*) o falso (*false*).

Byte.- Secuencia de 8 bits.

Cadena.- Secuencia de caracteres.

Carácter.- Símbolo que representa información, o la codificación en una computadora. Normalmente letras de alfabeto, números o signos ASCII.

Cargador de clases.- Parte del JRE de Java responsable de encontrar archivos de clase y cargarlos en la máquina virtual Java.

Casting.- Moldeado.

CGI.- Common Gateway Interfaz.

Clase.- Unidad fundamental de programación en Java, que sirve como plantilla para la creación de objetos. Una clase define datos y métodos y es la unidad de organización básica de un programa Java.

Common Gateway Interfaz (CGI).- Es un lenguaje de programación que permite dotar a las páginas Web de interactividad, construyendo una página Web correspondiente a un enlace de hipertexto en el mismo momento en que se hace "clic" sobre el enlace. Los *script cgi* pueden estar escritos en cualquier lenguaje de programación.

Common Object Requeset Broker Architecture (CORBA).- Estándar para la conexión entre objetos distribuidos, aunque esté codificados en distintos lenguajes.

Compilador.- Programa de software que traduce código fuente en un lenguaje de programación legible por una persona a código máquina interpretable por un ordenador.

Constante.- Valor utilizado en un programa de computadoras con la garantía de no cambiar en tiempo de ejecución. La garantía es a menudo reforzada por el compilador. En Java las constantes se declaran como *static final*.

Constructor.- Método que tiene el mismo nombre que la clase que inicia. Toma cero o más parámetros y proporciona unos datos u operaciones iniciales dentro de una clase, que no se pueden expresar como una simple asignación.

Contenedor.- En diseño de interfaces de usuario, es un objeto que contiene los componentes (como botones, barras de deslizamiento y campos de texto).

Conversión de tipos de datos.- Modificación de una expresión de un tipo de datos a otro.

CORBA.- Common Object Requeset Broker Architecture.

Entero.- Un número entero, sin parte decimal, positivo o negativo.

Estructura de datos.- Una construcción de software (en memoria o en disco duro) que contiene datos y las relaciones lógicas entre ellos.

Evento.- Un mensaje que significa n incidente importante, normalmente desde fuera del entorno de software.

Excepción.- Un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones.

Flujo.- Stream.

Graphical User Inteface (GUI).- Interfaz gráfica de usuario.

Hardware.- El aspecto físico de un sistema de computadora, como el procesador, disco duro e impresora.

Herencia múltiple.- La práctica (permitida en lenguajes como C++ pero no en Java) de derivar una clase de más de una *clase base.*

Herencia.- Mecanismo encargado de relacionar clases entre sí de una manera jerárquica. En Java, sólo existe herencia simple.

Hilo.- *Thread.*

HTML (HyperText Markup Languaje).- Lenguaje que se utiliza para crear páginas Web. Los programas de navegación de la Web muestran estas páginas de acuerdo con un esquema de representación definido por el programa de navegación.

IDE.- Integral Development Environment.

IDL.- Java Interface Definition Language.

Ingeniería del software.- Rama de la ingeniería concerniente con el análisis, diseño, implementación, prueba, y mantenimiento de programas de computadoras.

Instancia.- Objeto de software construido desde una clase. Por ejemplo, puede tener una clase avión, pero una flota de quince instancias de avión.

Integral Development Enviroment (IDE).- Una herramienta de desarrollo visual en la que un programa puede ser construido, ejecutado y depurado.

Interbloqueo.- Condición en la que dos o más entidades de software se bloquean mutuamente, cada una esperando los recursos que está utilizando la otra.

Interface Definition Language (IDL).- Herramienta mediante la cual los objetos pueden invocar métodos de otros objetos que se encuentren en máquinas remotas, mediante CORBA.

Interfaz gráfica de usuario (GUI).- Una interfaz entre la máquina y el hombre como el Windows de Microsoft, el Mac OS, o el Sistema X Windows, que depende de pantallas de alta resolución, un recurso gráfico de puntero como un ratón y una colección de controles en pantalla (denominados Widgets) que el usuario puede manejar directamente.

Interfaz.- Mecanismo Java para decirle al compilador que un conjunto de métodos serán definidos en futuras clases. (Esas clases estarán definidas para implementar la interfaz).

Java 2D.- Paquete que permite a los desarrolladores incorporar texto, imágenes y gráficos en dos dimensiones de gran calidad.

Java 3D.- Conjunto de clases para crear aplicaciones y applets con elementos en tres dimensiones. Es parte del JMF.

Java DataBase Connectivity (JDBC).- Lenguaje estándar de Java para interactuar con bases de datos, similar al SQL. Es independiente no sólo de la plataforma sino también de la base de datos con que interactúe. Desde la versión 1.2 del JDK se permite interactuar con ODBC.

Java Developer Connection (JDC).- Conexión de desarrollo en la que se publican las versiones beta de las bibliotecas de Java que se están desarrollando.

Java Foundation Classes (JFC).- Conjunto de componentes y características para construir programas con interfaces gráficas.

Java Media Framework (JMF).- Protocolo de transmisión de datos para la reproducción multimedia (vídeo y sonido).

Java Native Invocation (JNI).- Capacidad de Java para ejecutar código nativo, es decir, código compilado al lenguaje máquina de un determinado ordenador. Permite a la Máquina Virtual Java (JVM) interactuar con programas o bibliotecas escritos en otros lenguajes (C/C++, ensamblador...). No se puede utilizar en applets, pues viola las directrices de seguridad.

Java Runtime Environment (JRE).- Software suministrado por Sun que permite a los programas de Java ejecutarse en una máquina de usuario. El JRE incluye la Máquina Virtual Java (JVM).

JRE.- Java Runtime Environment.

JVM.- Java Virtual Machine.

Java Virtual Machine (JVM).- El intérprete de Java que ejecuta los códigos de byte en una plataforma particular.

JavaBeans.- Paquete que permite escribir componentes software Java, que se puedan incorporar gráficamente a otros componentes.

JDBC.- Java DataBase Connectivity.

JDC.- Java Developer Connection.

JFC.- Java Foundation Classes.

JMF.- Java Media Framewok

JNI.- Java Native Invocation.

JVM.- Java Virtual Machine.

Llamada por referencia.- Una forma de transferir parámetros a una subrutina en la que se pasa un puntero o referencia a un elemento, de esta forma, la subrutina puede leer y cambiar el valor del elemento referenciado.

Llamada por valor.- Una forma de transferir parámetros a una subrutina en la que se pasa la copia del elemento; las modificaciones de la copia no afectan al elemento original.

Método.- Conjunto de sentencias que operan sobre los datos de la clase para manipular su estado.

Miniaplicación.- *Applet*.

Modelo.- En diseño orientado a objetos, una representación del mundo real en unas abstracciones de software denominadas clases y la relación entre ellas.

Moldeado.- Suplantación del tipo de un objeto o variable por otro nuevo tipo.

Multiproceso.- En sistemas operativos, la habilidad de efectuar dos o más programas independientes, comúnmente en un procesador solo (a través de *Multitarea*).

Navegador Web.- Software que permite al usuario conectarse a un servidor de Web utilizando Hypertext Transfer Protocol (HTTP). Microsoft Internet Explorer, Netscape Navigator, HotJava de Sun, son populares navegadores de Web.

Navegador.- Navegador Web.

null.- Valor de Java que significa *vacío*.

Object DataBase Conectivity (ODBC). Lenguaje estándar de Microsoft; que utiliza un driver del fabricante de una base de datos, para interactuar con ella, más orientado a C/C++ que a Java.

ODBC.- Object DataBase Conectivity.

Paquete.- Nombre de Java para una biblioteca de clases.

Parámetros formales.- Nombres utilizados dentro de una subrutina por sus parámetros.

Parámetros.- Valores u objetos pasados entre una subrutina y la rutina de llamada.

Plug-in.- Un programa de una plataforma específica diseñado para ser llamado por un navegador Web. Utilizado con frecuencia para mostrar información que el mismo navegador no puede mostrar.

Poliforfismo.- En diseño orientado a objetos, la habilidad de utilizar una clase derivada en lugar de su clase base. Por ejemplo, un programador puede escribir un método *expresarse()* para la clase Mamífero. Un Perro, una Vaca y un Gato se derivan de Mamífero, y todos pueden *expresarse()*, aunque sus voces sean bastantes diferentes.

Proceso.- Instancia de un programa ejecutable. Por ejemplo, si inicia dos copias de un intérprete de Java, tiene dos procesos de la máquina virtual de Java ejecutándose en su computadora.

Seudocódigo.- Documentación de diseño que describe el trabajo de un programa en inglés estructurado (o en otro lenguaje) en lugar de un lenguaje de computadora.

Recolector de basura.- En Java, el mecanismo por el cual se recobra y libera la memoria asociada con objetos no utilizados.

Remote Method Invocation (RMI).- Herramienta que incorpora métodos Java ara localizar objetos remotos, comunicarse con ellos e incluso enviar objetos como parámetros de un objeto a otro.

RMI.- Remote Method Invocation.

Secure Sockets Layer (SSL).- Sistema para la creación de conexiones seguras en red.

Servlets.- Módulos que permiten sustituir o utilizar el lenguaje Java en lugar de programas CGI.

Shell.- Intérprete de órdenes de un sistema operativo.

Sistema operativo.- Software responsable de asignar a los usuarios los recursos de sistemas de computadoras (incluyendo procesos). UNIX, Windows, NT y Mac OS, son ejemplos de sistemas operativos.

SQL.- Structured Query Language.

SSL.- Secure Sockets Layer.

Estático.- En diseño orientado a objetos, representa la pertenencia a la clase, en vez de a una instancia. Es un espacio compartido por todas las instancias de una clase.

Stream.- Flujo de datos. Por ejemplo las entradas y salidas de un programa.

String.- Objeto Java estandarizado en el lenguaje, que representa una cadena de caracteres.

Structured Query Language (SQL).- Lenguaje para realizar consultas a Bases de Datos relacionales.

Subclase.- Clase descendiente de otra clase de la que hereda métodos y variables.

Superclase.- Clase de la cual heredan sus métodos y variables otras clases denominadas subclases.

Swing.- Paquete que permite incorporar elementos gráficos en las aplicaciones, de una manera más potente que con el AWT. Aparece en la versión 1.2 del JDK. Es no de los componentes que están incluidos en las Java Fundation Classes, o JFC.

Thread.- Un "proceso ligero" que puede ser arrancado y utilizado más rápidamente que por un *fork* o *spawn. Véase* también: *fork, spawn y Proceso.*

Tiempo de vida.- El número de líneas sobre las que una variable es activa, esto es, el número de líneas entre la primera y la última referencia a la variable.

Tipo primitivo.- En Java, un tipo de dato que no es un objeto. Los tipos primitivos incluyen caracteres, enteros, número de coma flotante y booleanos.

UML.- Unified Modeling Language.

Unicode.- Conjunto de caracteres de 16 bits, en lugar de los 8 que soportaba ASCII. Así se pueden representar la mayor parte de los lenguajes del mundo.

Unified Modeling Language (UML).- Notación estándar de facto utilizada en el análisis y diseño orientado a objetos, basado en el trabajo de Grady Booch, James Rumbaugh, e Ivar Jacobson.

Vector.- Estructura de datos que coloca un tipo de datos en celdas continuas.

Verificador de código de byte.- Rutinas en la máquina virtual de Java, que aseguran que las instrucciones en el archivo de clase no violan ciertas restricciones de seguridad.

BIBLIOGRAFÍA

Se ha dividido este apartado en diversos puntos, intentando explicar qué referencias bibliográficas han sido utilizadas en mayor o menor medida para realizar este tutorial.

En cada reseña bibliográfica se indican el número de páginas que tiene, y un comentario personal indicando su utilidad para una mayor profundización en Java.

A. BIBLIOGRAFÍA CONSULTADA PARA LA ACTUAL VERSIÓN:

- **[Arnold y Gosling, 1997] Ken Arnold y James Gosling. Addison-Wesley/Domo.** "*El lenguaje de Programación Java"*. Wesley Iberoamericana. 1997. 334 páginas. (*Muy básico, escrito por el desarrollador del lenguaje*).
- **[Eckel, 1997] Bruce Eckel**. "*Hands -on Java Seminar*". Presindent MindView Inc. 1997. 577 páginas. (*Tutorial completo en Inglés en formato PDF*).
- **[García et al., 1999]. Javier Garciá de Jalón, José Ignacio Rodríguez, Iñigo Mingo, Aitor Imaz, Alfonso Brazález, Alberto Larzabal, Jesús Calleja y Jon García**. *"Aprenda Java como si estuviera en primero"*. Universidad de Navarra. 1999. 140 páginas. (*Tutorial muy básico, en el que se trata la potencia de Java, pero sin profundizar en ningún tema en particular*).
- **[García, 1997] Francisco José García Peñalvo.** "*Apuntes de teoría de la asignatura Programación Avanzada del tercer curso de Ingeniería Técnica en Informática de Gestión*". Universidad de Burgos. 1997. 216 páginas. (*Comentan la teoría de la programación orientación al objeto. Han sido resumidos para dar una visión de la programación orientada a objeto en el apartado I.1*).
- **[Johnson, 1996] Jeff Johnson**. "*Coding Standards for C, C++, and Java*". Vision 2000 CCS Package and Application Team. 1996. 14 páginas. (C*onsejos para el formato de los fuentes Java, C y C++*).
- **[Morgan, 1999] Mike Morgan.** "*Descubre Java 1.2*". Prentice Hall. 1999. 675 páginas. (*Sin duda muy interesante, sobre todo por su actualidad al tratar con Java 2, y por su extensión al tratar todas las bibliotecas de Java*).
- **[Naughton, 1996] Patrick Naughton**. *"Manual de Java"*. Mc. Graw Hill 1996. 395 páginas. (*Introduce todos los aspectos de la programación básica en Java*).
- **[Rojo, 1998] Ignacio Rojo Fraile**. "*Comparativa de herramientas Java*". Artículo de la revista Solo Programadores nº49. Tower. Octubre 1998. (*Compara 5 IDEs Java*).
- **[Sanz, 1998] Javier Sanz Alamillo**. "*Novedades y cambios con Java 2*". Artículo de la revista Solo Programadores nº55. Tower. Marzo 1999. (*Buen resumen de los cambios que han surgido en el JDK 1.2*).
- **[Sun, 1998] Sun Microsystems Inc.** "*JDK 1.2 Documentation"*. www.sun.com. 1997. (*Documentación de la API de Java del JDK*).
- **[van Hoff et al., 1996] Arthur van Hoff, Sami Shaioi y Orca Starbuck**. *"Hooked on Java"*. Addison-Weslet. 1996. *(Todo lo que hace falta saber para crear applets, con muchos ejemplos. En inglés)*.

- **[Zolli, 1997] Andrew Zolli**. "*La biblia de Java*". Anaya multimedia. 1997. 814 páginas. (*Completo en lo a que bibliotecas del JDK 1.1 se refiere*).

B. BIBLIOGRAFÍA ADICIONAL O CITADA EN LA ACTUAL VERSIÓN:

- **[Piattini et al., 1996] Mario G. Piattini, José A. Calvo-Manzano, Joaquín Cervera y Luis Fernández**. "*Análisis y diseño detallado de Aplicaciones informáticas de gestión*". Ra-Ma. 1996.
- **[Rambaugh et al., 1998] J. Rambaugh J., M. Blaha, W. Premerlani, F. Eddy y W. Lorensen**. "*Modelado y Diseño Orientados a Objetos. Metodología OMT*". Prentice Hall, 2º reimpresión. 1998.
- **[Rational, 1997] Rational Software Corporation**. "*The Unified Modeling Language Documentation Set 1.1*". www.rational.com. Septiembre de 1997.
- **[Rifflet, 1998] Jean-Marie Rifflet** "*Comunicaciones en UNIX*". McGraw Hill. 1998.

C. BIBLIOGRAFÍA ADICIONAL QUE SE UTILIZÓ EN LA VERSIÓN 1.0

- **[Cortés et al.,1996] José Luis Cortés, Nuria González, Virginia Pérez, Paqui Villena y Ana Rosa Freire**. "*Java, el lenguaje de programación de Internet"*. Data Becker 1996.
- **[Cuenca, 1996] Pedro Manuel Cuenca Jiménez.** "*Programación en Java para Internet*". Anaya Multimedia. 1996.
- **[Cuenca, 1997] Pedro Manuel Cuenca Jiménez,.** "*Programación en Java*". Ediciones Anaya Multimedia. 1997.
- **[Framiñán, 1997] José Manuel Framiñán Torres**. "*Manual Imprescindible de Java*". Ediciones Anaya Multimedia. 1997.
- **[Lalani, 1997] Suleiman 'Sam' Lalani**. "*Java, biblioteca del programador*". Ediciones McGraw Hill. 1997.
- **[Froufe, 1997] Agustín Froufe**. "*Tutorial de Java*". Facultad de Informática de Sevilla. 1997. http://www.fie.us.es/info/internet/JAVA/.

D. DIRECCIONES DE INTERÉS

Se recomienda suscribirse a la lista de correo de Sun sobre Java, cuya dirección es:

javanews@US.IBM.COM