UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

"DESARROLLO DE UNA APLICACIÓN CLIENTE-SERVIDOR PARA DISCUSIONES EN TIEMPO REAL UTILIZANDO JAVA. JSP Y MySQL"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENÍERO EN COMPUTACIÓN

P R E S E N T A:

ISAÍAS MARTÍNEZ ALDAMA

ASESOR:
M. EN I. ELIO VEGA MUNGUÍA

SAN JUAN DE ARAGÓN, ESTADO DE MÉXICO 2006

Dedicatoria:

A Dios.

A mis padres: Benjamín Martínez Martínez y Epifanía Aldama Esquivel, por el apoyo que todos estos años me han dado y por confiar en mí durante la etapa de preparación y estudio.

A la Universidad: Por brindarme la oportunidad de estudiar y de adquirir el conocimiento necesario para poder tener una carrera.

A la FES Aragón: Por permitirme la preparación en sus inmediaciones durante mis estudios y aun después de ellos; sus instalaciones me dieron el apoyo necesario para cumplir este objetivo.

A mis Profesores: A todos y cada uno de ellos, por que de cada uno aprendí, me brindaron su conocimiento y apoyo para lograr este sueño.

A mi novia: Galdina Tomatzin García, gracias por el apoyo y comprensión durante este tiempo de estudio y preparación.

Al M. en I. Elio Vega Munguía: Muchas gracias por apoyarme en el desarrollo de este trabajo, por la paciencia y dedicación necesarias para lograr este objetivo.

A todos mis amigos: Me han apoyado y me han brindado su amistad durante todos estos años, parte de su tiempo, comprensión y paciencia, valores muy importantes y necesarios durante la preparación académica, estoy muy agradecido.

Director de titulación:

M. en I. Elio Vega Munguía

Presidente:

Ing. Juan Gastaldi Pérez

Vocal:

M. en E. Imelda de la Luz Flores Díaz

Secretario:

Ing. Abel Verde Cruz

Suplente:

M. en I. Arcelia Bernal Díaz

Desarrollo de una aplicación cliente-servidor para discusiones en tiempo real utilizando Java. JSP y MySQL.

Índice

Introducción

Objetivos del Proyecto.

En la actualidad la convergencia de los diferentes SO es una realidad, una prueba de ello es la tecnología en Internet, la cual nos permite navegar a todos con prácticamente cualquier computadora. La información y la comunicación de manera instantánea, juegan un papel muy importante en este ámbito; una herramienta que nos ayuda a mantener esta portabilidad en las diversas plataformas es el lenguaje Java.

El objetivo es desarrollar una aplicación que interactué con una Base de Datos y la presentación sea orientada al web. Se ha elegido el lenguaje de programación Java debido a que este lenguaje fue diseñado de tal manera que la portabilidad fuera una de sus ventajas y se pueden crear pequeños programas para ejecutarse en un navegador, estos programas son los Applets.

Los beneficios de un programa con tales características y que además interactué en tiempo real con varios usuario mientras se entrevista algún personaje con bastos conocimientos en determinado tema, son aprovechados generalmente en un ambiente lúdico, como son las universidades.

Las características de los proyectos para demostrar que Java nos permite la interoperabilidad y la programación en el Web son:

Chat IRC Moderado. Ejemplo claro de las tecnologías en Internet y de la comunicación en forma inmediata, en donde varios usuarios pueden interrelacionarse.

Manejo y operabilidad de una Base de Datos (BD). La disponibilidad, control y eficiencia que nos permite tener un *Data Base Manager System* (DBMS) es una ventaja inmensurable, y con la posibilidad de consultar la información desde cualquier computadora, esta ventaja se incrementa.

Ventajas y desventajas.

El utilizar Java como lenguaje de programación brinda diversas ventajas: tiene portabilidad de las aplicaciones, utiliza Programación Orientada a Objetos (POO) al 100 %, además de ser un lenguaje fácil de implementar, se puede programar aplicaciones *Stand Alone* o para el Web y tiene un alto grado de seguridad al programar.

Los inconvenientes radican en que se tiene que compilar e interpretar el código, por lo mismo es más lento de ejecutar que los lenguajes convencionales. Para los usuarios de Internet, deben de instalar la Maquina Virtual de Java (*Java Virtual Machine* JVM), y la incompatibilidad con las primeras versiones puede generar conflictos.

Antecedentes del Proyecto.

En las empresas existen departamentos encargados de llevar el control de información en general. En muchas de ellas se hace de manera tradicional, es decir, no existe programa de cómputo alguno que permita generar una mejor administración que la que provee el manejo de papeles.

El diseño de BD y la implementación de estas mediante DBMS es algo muy común, ayuda a solucionar problemas dentro del área administrativa y brinda las siguientes ventajas:

- Rapidez y control de la información
- Información actualizada y al instante.
- Información disponible en varios niveles.
- Control estricto de las actividades que se desarrollan.
- Consultas en diferentes sitios a través de Internet.

Dichas BD se esta diseñando bajo el modelo Entidad – Relación, a partir del análisis de los requerimientos para determinar las entidades involucradas, sus relaciones, implementación dentro de los Sistemas Operativos (SO), costos comerciales, tipos de consultas, interfaces, etc. se ha llegado a la conclusión de utilizar "MySQL ver 3.23" para implementar la BD y Microsoft Access como apoyo en la demostración de los objetivos del proyecto.

Requerimientos del Sistema.

Debido a las necesidades propias del sistema, es necesario tener cierto software instalado por parte del Servidor, en la parte del cliente, solo será necesario tener un navegador o browser para poder ver el sitio Web y cargar el applet.

Para la implementación del proyecto es necesario contar con herramientas de software adicional como los son el SDK ver 1.4.0, un servidor Web Apache ver 1.3.27, Tomcat 4.0 (Contenedor de Servlet/JSP), Sistemas Operativos Linux Red Hat ver 6.0 ó Windows 2000.

Capítulo 1: Introducción a Java.

1.1 Los orígenes de Java.

En el año de 1990, Sun Microsystems desarrollaba programas para poder manipular algunos productos en la industria de la electrónica como microondas, tostadoras y, fundamentalmente, televisión interactiva basándose en una interfaz cómoda e intuitiva. Este reto presentaba diversos problemas como la proliferación de arquitecturas incompatibles, las actualizaciones de los chips por los fabricantes a modelos más potentes y baratos, obligando a reprogramar cada vez con nuevo software adaptado a los chips, también la fiabilidad del código y la facilidad de desarrollo fue un punto a considerar.

Para poder enfrentar este reto, se le asigna el proyecto a James Gosling, quien decidió, después de analizar los lenguajes disponibles en esos años, que las ventajas aportadas por dichos lenguajes, incluyendo C++ (popular lenguaje de POO desarrollado por Bjarne Stroustrup), no compensaban el gran coste de pruebas y depuración. Se hizo necesario desarrollar un nuevo lenguaje de programación que permitiera escribir programas que funcionaran en cualquier tipo de plataforma, así nació el lenguaje de programación *Oak* ("roble"), lanzado en enero de 1991, el cual estaba basado en la sintaxis de C++ y el compilador se realizo en lenguaje C.

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

Una vez que en Sun se dieron cuenta de que a corto plazo los electrodomésticos interactivos no serían un gran éxito, urgieron a FirstPerson a desarrollar con rapidez nuevas estrategias que produjeran beneficios. No lo consiguieron y FirstPerson cerró en 1994.

En esos años Internet dejaba las universidades y los centros de investigación, con la creación del primer navegador (Mosaic), permitiendo a las empresas y a particulares la integración al WWW, Bill Joy, cofundador de Sun juzgó que Internet podría llegar a ser el campo de juego adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes. *"En nuestro idioma Oak significa roble, y este apelativo obedecía exclusivamente a la presencia de un roble en las inmediaciones del lugar de trabajo del equipo"* [1]. Tras un cambio de nombre y modificaciones de diseño, obtiene un lenguaje potente, seguro y universal: el lenguaje Java, presentado en sociedad en 1995.

[1] Programación en Java, Pedro Manuel Cuenca Jiménez, Anaya Multimedia, Madrid 1997, p. 25

Este nuevo lenguaje nace junto con una versión preliminar de un navegador llamado HotJava, creado por Patrick Noughtan y Jonathan Payne, de Sun, también fue apoyado por Netscape, empresa que integró en su navegador la plataforma Java, permitiendo ejecutar programas dentro de una página Web, algo hasta entonces impensable con el HTML. La creación de un compilador de Java en Java por Arthur Von Hoff demostró la integridad de este lenguaje.

El primer kit para desarrollar en Java es introducido por Sun en 1996, conocido como *Java Developmnet Kit* (JDK), incluyendo algunas herramientas que facilitaban el acceso a base de datos (JDBC), crear interfaces gráficas (AWT) así como el estándar *JavaBeans*. En los años posteriores han salido diversas versiones del JDK, y en la actualidad se conoce como *Software Developmnet Kit* (SDK), las cuales tienen mejoras con respecto a las versiones anteriores, por ejemplo las librerías Swing.

El estándar JDBC de acceso a bases de datos, así como el estándar *JavaBeans* ("granos de café"): éste potencia el carácter de lenguaje orientado a objetos de Java, al tiempo que proporciona un entorno visual de programación que facilita y simplifica el desarrollo del software escrito en Java. De este modo, Java se convierte en un poderoso entorno de programación y de ejecución de programas.

1.2 Importancia de Java.

Java es un lenguaje de Programación Orientada a Objetos (POO) para propósitos generales, diseñado en principio para el ambiente distribuido de Internet, es el más reciente entre sus semejantes, y aun se encuentra en fase de evolución, pero una gran ventaja de este lenguaje es la máquina virtual de Java (JVM), un soporte para la ejecución de programas en distintas plataformas, es decir que las aplicaciones escritas en Java se ejecuten en cualquier computadora, independientemente del S.O. y de la configuración de hardware utilizados.

Si comparamos a Java con otros lenguajes de su mismo nivel, encontraremos varias diferencias, una de las más importantes es la manera en como crean su código ejecutable. La mayoría de los lenguajes al compilarse crean un ejecutable ligado al Hardware y al S.O., en cambio Java recopila las fuentes de sus programas en un código llamado *Bytecode*, el cual será interpretado por la JVM.

1.2.1 Applets

La integración de Java con el Internet y con los diversos S.O. mediante un navegador utilizando los applets (programas escritos en Java exclusivos para ejecutarse en un navegador) le dio a Java un gran potencial y ventaja sobre otros lenguajes, pero no solo se pueden programar applets, también es posible crear aplicaciones *Stand-Alone*.

Los applets funcionan al ser cargados en la página que se desea visualizar, una vez cargados se ejecutan en la computadora del usuario, mostrando el applet y la página en la misma pantalla. Dado que los applets deben viajar a través de Internet y el ancho de banda es limitado, estos programas deben ser pequeños (para que se carguen en poco tiempo) y

rápidos (para que se visualicen en la página web sin demoras), pero no por ello dejar de ser confiable y seguro, no se permiten errores que bloqueen la computadora o dañen parte del sistema de archivos.

1.3 Seguridad en Java.

Mencionamos en párrafos anteriores la integración de Java con el Internet mediante los applets, esta es una gran ventaja, pero dado que estos programas se pueden bajar desde cualquier parte en Internet, se pueden ejecutar en la mayor parte de los navegadores y son independientes de la plataforma; la seguridad se impuso como una necesidad de vital importancia en Java, aprovechando características del sistema de ejecución en tiempo real de la JVM y medidas de seguridad en el lenguaje, por ejemplo, la eliminación de los punteros y el manejo de excepciones.

El modelo de seguridad en Java es denominado *Sandbox*, por las semejanzas que tiene con dicho termino; concibiendo a la JVM como una caja en donde los applets pueden hacer todo lo que quieran, pero sin salirse de esta, y para interactuar con el exterior se emplean algunos mecanismos restringiendo los recursos del S.O. al applet. Al ser los applets los que más importan en la restricción al modelo de seguridad en Java, solo nos enfocaremos en las características que los involucran.

Para poder lograr la ejecución de un applet en un navegador es necesario apegarse a las siguientes restricciones:

- No se puede trabajar con el sistema de ficheros: leer, escribir, borrar, renombrar, listar, conseguir información.
- No se pueden establecer conexiones de red a máquinas distintas que la que envió el applet.
- No se permite acceso a recursos del S.O.
- No se permite manipulación de *threads*.
- No se pueden cargar métodos nativos.
- No se pueden evitar mensajes de alerta en las ventanas creadas por el applet.
- No se pueden crear subclases de *SecurityManager* en una applet.

Todas estas las restricciones anteriores se implementa mediante la construcción de cuatro barreras o líneas de defensa ejecutadas dentro de la JVM:

- Características del lenguaje/compilador
- Verificador de código de bytes
- Cargador de clases
- Gestor de Seguridad

Estas defensas no implican una sucesión, es decir una después de otra, y que sea necesario violentar la primera para después continuar con la segunda. En realidad son defensas independientes, y si la integridad de alguna de ellas esta en riesgo, el sistema también puede estarlo.

1.3.1 Características del lenguaje/compilador

Recordando que al crearse Java y también los applets, se pensó en que sería confiable, seguro y no se permitirían errores que bloquearan la computadora o dañaran parte del sistema de archivos, entre otros. Para lograrlo se incluyeron las siguientes características en el lenguaje de programación.

Ausencia de punteros

Una fuente común de errores en lenguajes que manipulan punteros, como C o C++, que ofrecen grandes ventajas, pero a costa de la integridad del sistema, razón por la cual no se incluyen en Java. De esta manera se protege frente a imitación de objetos, violación de encapsulación, errores en las referencias a posiciones de memoria específicas no reservadas.

Gestión de memoria

También fuente de errores en lenguajes que la implementan. Ayuda en gran medida a crear programas dinámicos, pero el no liberar la memoria es un error frecuente o consumirla en su totalidad. Por esta razón no se puede gestionar la memoria de forma directa, pero se instancian objetos, y la liberación de los recurso de memoria está a cargo del recolector de basura, reduciendo la interacción del programador con la memoria y con el S.O.

Recolector de Basura.

El programa no puede manejar directamente la asignación de la memoria, esto evita que se consuman recursos innecesarios, se agote la memoria del sistema, ya sea por una acción maligna o por descuido del programador. El recogedor de basura de Java se encarga de reclamar la memoria usada por un objeto una vez que éste ya no es accesible o desaparece, incluso si se desea, se puede llamar al recolector de basura de manera manual cuando se crea necesario.

Manejo de excepciones.

Java tiene muchas excepciones las cuales son lanzadas cuando se genera un error, como por ejemplo un desbordamiento, división entre cero, un índice de arreglo fuera de rango, etc. esto le da una gran estabilidad, evitando que los programas se colapsen y manejando las excepciones originadas en tiempo de ejecución. Un buen programador deberá considerar siempre esta herramienta dentro de sus aplicaciones Java, fortaleciendo la seguridad y estabilidad de las mismas.

Capítulo 2: Arquitectura Java

2.1 Diseño principal de un programa en Java.
2.2 Applets.
2.3 Interacción con los usuarios.
2.4 Flujo de datos E/S
2.5 Conexiones en el Web.

La arquitectura de Java es todo un sistema creado para desarrollar programas de una manera rápida, estables, seguros, fiables al nivel de red (como lo es Internet), multiplataformas y mucho más, logrando en Java un gran lenguaje de programación.

En este capítulo veremos las características antes mencionadas, divididas en cuatro etapas, las cuales incluyen el diseño y planeación de los programas, el proceso de codificación, la compilación y al final la ejecución del programa en la JVM.

2.1 Diseño principal de un programa en Java

Como lo mencionamos anteriormente, cualquier aplicación creada en Java podemos dividirla en cuatro bloques:

1. Diseño y planeación.
2. Codificación.
3. Proceso de compilación.
4. Ejecución de la Maquina Virtual de Java.

2.1.1 Diseño y planeación.

Al igual que ocurre con el mundo real que cada vez se vuelve más complejo, los sistemas asistidos por ordenador también aumentan cada día su complejidad. A menudo tienen implicados múltiples partes de hardware y software, conectados en red a través de grandes distancias, vinculadas a bases de datos que contienen enormes cantidades de información.

La importancia del diseño y planeación de cualquier aplicación en general es reflejada en la estabilidad y fiabilidad de los programas. Cuando un programa debe de ser modificado los frutos del diseño y planeación son cosechados en buena manera.

En este proceso sé decidirá como diseñar las aplicaciones, e incluye entender cuatro fundamentos para poder lograr una buena planeación:

- Necesidades. Es indispensable entender que hará el programa, es decir las necesidades del usuario, para lograr aplicaciones bien estructuradas orientadas a solucionar problemas en específico.

- Evaluación de lo que es posible. Hay que tener en cuenta la tecnología con la que se cuenta en la actualidad para saber lo que es posible lograr. La tecnología incluye ambientes de desarrollo, librerías, sistemas de ingeniería, etc. La tecnología es el contexto en donde el diseño es construido a partir de bloques especializados.

- Las facilidades. Las implementaciones que se realizan de manera rápida y fácil, ya sean por su naturaleza o por los medios tecnológicos. Entendiendo lo que es fácil, puede ser usado para asignar prioridades dentro del proyecto y agilizarlo tanto como sea posible.

- Las dificultades. Las características necesarias para el diseño y la codificación. Entendiendo lo que será difícil, las funciones criticas pueden ser previstas y manejadas con una prioridad adecuada dentro del proyecto.

Una vez que se han resuelto estos planteamientos, para cualquier aplicación en general, sé continua con los detalles del diseño, planeación e implementación necesarios para los programas.

Si no se conocen o entienden los requerimientos, puede llevarnos a la creación de un programa que no satisface las necesidades del usuario. El no conocer lo que es posible dentro del ámbito de la aplicación, nos llevará a tratar de diseñar algo imposible y la dificultad, por el sentido propio de la palabra, nos ayudará a conocer el tipo de problema al que nos enfrentamos.

Estos análisis no solo ayudarán a escribir programas más estables y mejores, también ayudarán a crear aplicaciones exitosas, y *"la forma más que toma una arquitectura Cliente-Servidor se llama arquitectura de dos capas"* [1], una aplicación de dos capas proporciona múltiples estaciones de trabajo con una capa de presentación uniforme.

2.1.2 Codificación.

En todos los lenguajes de programación, para poder crear cualquier aplicación, es necesario escribir el código fuente, y Java no es la excepción, obviamente con sus diferencias, las cuales hacen destacar a Java.

La forma convencional de la implementación de los lenguajes de programación es la siguiente: el código fuente es transformado por el compilador a un lenguaje *básico* conocido como código objeto, a partir de este código objeto se puede ejecutar el programa. El código fuente se rige por reglas muy estrictas, conocido como la sintaxis, la cual debe conocer el programador, si mientras el compilador trata de crear el código objeto ocurre un error, se desplegaran mensajes de errores indicando lo que ha sucedido.

Java utiliza completamente la Programación Orientada a Objetos (POO).

Los programadores y diseñadores de software siempre se han esforzado por reducir la complejidad de los programas, entre más grandes sea una aplicación, mayor será su

[1] Programación de Base de Datos con JDBC y Java, George Reese, Anaya Multimedia, Madrid 2001, p. 156

complejidad y también la depuración. Por esta razón se desarrolla la POO a principios de los noventa, como medio para organizar el código fuente y los datos, logrando un control sobre la complejidad en el proceso de desarrollo de programas.

La POO tiene sus orígenes en lenguajes tales como Simula 67 y Smalltalk, no es una idea nueva de Java, pero si la implementa en su totalidad. Los conceptos fundamentales de la POO son las clases, los objetos, la herencia y el polimorfismo.

Las clases son un tipo definido por el usuario que determina las estructuras de los datos y las operaciones asociadas con ese tipo, es decir, son plantillas o modelos que describen las características de los datos y como sé interactúa con estos datos.

Los objetos son tipos de abstracción de datos, es decir instancias de una clase, contiene datos y las funciones que operan sobre esos datos; a los elementos de un objeto se les conoce como miembros, y a las funciones que operan sobre dicho objeto se denominan métodos.

Una característica muy importante de la POO es la herencia, propiedad que permite a los objetos construirse a partir de otros objetos, de clases a subclases. El principio de este tipo de ramificación es que cada subclase comparte características comunes con la clase de la cual se deriva, pero tiene además sus propias características particulares.

EL polimorfismo, en sentido literal, significa la cualidad de tener más de una forma, en la POO, se refiere al hecho de que una misma operación puede tener diferentes comportamientos en diferentes objetos, es decir que diferentes objetos pueden reaccionar en forma distinta al mismo mensaje.

Java se diferencia de los demás lenguajes de programación desde las reglas del código fuente, en las cuales sobresalen varías características mencionadas a continuación.

Ausencia de punteros.

Los punteros son variables que contienen la dirección de otra variable, muy utilizados en lenguajes como C/C++, pero una fuente de errores muy común como acceso a áreas protegidas de memoria. Es por esta razón que no se incluyen en Java.

Tipos seguros en Java.

Los objetos no pueden intercalarse con los casting si son tipos no compatibles, y por lo tanto habrá métodos que no podrán ser llamados desde otros objetos.

Hay 4 niveles de acceso en los objetos.

EL primero de ello es el público (disponible para cualquiera), protegido(disponible para la clase y sus clases derivadas o hijas), privado (solo accesible para esa clase) y por default (accesible dentro de las mismas clases dentro del paquete o *package*).

Inicialización explícita.

Todas las variables primitivas y objetos deben de ser inicializadas, esto evitara errores en la ejecución del programa. Si no son inicializadas, marcará errores de compilación.

Arreglos con fronteras.

Desde que se conoce la longitud de cualquier arreglo, se establece una frontera para que el programa no pueda leer o realizar cambios fuera de esta frontera, si esto sucede, se lanzará una excepción.

Manejo de excepciones.

Los programadores de cualquier lenguaje, no solo en Java, se esfuerzan por escribir programas libres de errores, sin embargo, es muy difícil que los programas reales se vean libres de ellos.

Existen anomalías producidas en el momento de ejecución del programa, estas anomalías pueden provocar un fallo en el programa, y se conocen como excepción. En Java, cuando se genera una excepción, el control del programa es transferido al manejador de excepciones. Este mecanismo permite a Java manipular de una forma segura errores como división entre cero, un desbordamiento, un índice de arreglo fuera de rango, etc.

Como condiciones no usuales en los programas, las excepciones pueden detener un programa o conducir a errores. Típicamente se pueden detectar siempre si ocurre cierto tipo de errores, como la división entre cero, tales errores se conocen como excepciones síncronas, debido a que aparecen en un momento previsible. En contraste con estas excepciones existen otras excepciones conocidas como asíncronas, las cuales se producen por sucesos fuera del control del programa, por consiguiente no se pueden anticipar.

Pero no solo el programa genera excepciones, también el programador puede lanzar sus propias excepciones. Las excepciones en Java son objetos de clases derivadas de la clase base *Exception*. Existen también los errores internos que son objetos de la clase *Error,* ambas clases *Error* y *Exception* son clases derivadas de la clase base *Throwable*.

Las excepciones síncronas o en tiempo de ejecución ocurren cuando el programador no ha tenido cuidado al escribir su código. Por ejemplo, cuando se sobrepasa la dimensión de un *array* se lanza una excepción *ArrayIndexOutOfBounds*. Cuando se hace uso de una referencia a un objeto que no ha sido creado se lanza la excepción *NullPointerException*. Estas excepciones le indican al programador que tipos de fallos tiene el programa y que debe arreglarlo antes de proseguir

El segundo grupo de excepciones, las asíncronas son el más interesante, ya que indican que ha sucedido algo inesperado o fuera de control y por tal motivo, el manejo de las mismas se complica.

La importancia de esta característica debe de ser implementada por todos los programadores, aunque por su naturaleza, las excepciones asíncronas son más difíciles de manipular. Con un buen manejo de excepciones dentro de los programas, la estabilidad y fiabilidad de los mismos será muy alta.

.

2.1.3 Proceso de compilación.

Como se menciono anteriormente, el proceso de compilación convencional crea código objeto, el cual puede ser ejecutado por un hardware en específico, pero el compilador de Java crea *bytecodes*, este *bytecodes* puede ejecutarse en cualquier computadora, siendo esta una gran ventaja, pero antes de que esto sea posible es necesario tener un interprete, el cual es conocido como *Java Virtual Machine* (JVM).

Los archivo generados por el compilador de Java tienen la extensión *.class*, pues solo contienen una implementación de una clase de Java, por lo tanto habrá tantos archivos *class* como clases dentro de la aplicación. Generalmente los programas de Java se construyen con más de una clase.

El compilador debe verificar los *casting*, los índices en los arreglos, etc. en todo el código fuente, y así generar archivos *class* más fiables.

Además, la JVM ha sido implementada en un chip creado por Sun Microsystems, estos chips tienen un mayor desempeño comparado con una JVM implementada en software. Esto permite incluir aplicaciones Java en electrodomésticos como televisores, microondas, mini componentes, etc.

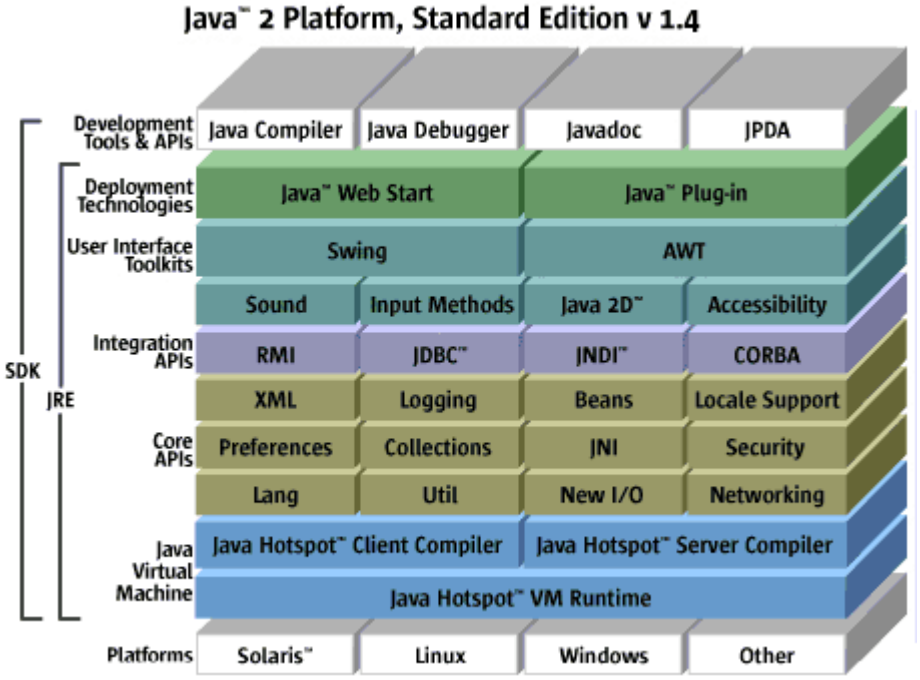
2.1.4 Ejecución de la Maquina Virtual de Java.

Todos los programa de Java son compilados en archivos clase, los cuales contiene *bytecodes*, el lenguaje máquina de la JVM. La JVM obtiene un *stream* de *bytecodes* para cada método en la clase, estos *stream* son guardados por la JVM y son llamados cuando se ejecutan en el programa.

Un método en *bytecode* es una secuencia de instrucciones para la JVM, cada instrucción consiste por un operando seguido de ceros o más operandos, el código-operando indica la acción a tomar, si más información es necesaria antes de que la JVM pueda tomar una acción, esa información es codificada en uno o más operandos, los cuales inmediatamente siguen al operando.

Para ejecutar los *bytecodes*, la JVM primero debe cargar la clase, para lo cual se auxilia del *classloader* para traer los *bytecodes* del disco duro o de la red. Cada archivo class pasa a través del verificador de *bytecodes*, el cual se asegura que tenga el formato correcto y que la clase no interfiera con la memoria. La fase de verificación del *bytecode* toma algo de tiempo para cargar la clase, pero este proceso solo se realiza una sola vez y no continuamente mientras el programa se esta ejecutando.

La JVM simplemente ejecuta las instrucciones de los *bytecodes*, es decir, es un interprete, determina lo que quiere decir y después desarrolla la función asociada. Los interpretes son más lentos comparados con los códigos nativos creados por un compilador convencional.



Capas en la plataforma Java

Afortunadamente existe una alternativa para esta situación, la cual es conocida como JIT (*Just In Time*), y convierte los *bytecodes* a instrucciones nativas de la computadora del usuario, inmediatamente antes de la ejecución.

Los compiladores JIT se ejecutan en las computadoras de los programadores, por el hecho de que son ellos quienes tienen la necesidad de compilar continuamente. La desventaja de los JIT es que producen programas no portables y por lo tanto ese programa compilado solo se ejecutará en la computadora del usuario, o en un hardware igual.

2.2 Applets

Los *applets* son pequeños programas con una interfaz gráfica que puede insertarse en una página web, para lo cual el navegador debe reconocer la etiqueta APPLET, es decir el código de un *applet* es interpretado por el propio navegador mediante la JVM, de esta manera se pueden hacer páginas Web más dinámicas.

Las características de estos programas son las siguientes:

Compatibilidad con los diferentes Sistemas Operativos. Cualquier *applet* puede ser visualizado en los navegadores compatibles con Java, si el navegador reconoce Java, ejecutará el *applet*, sin importar el S.O. en que este.

Gran escala de distribución. Una vez que el applet ha sido creado, y colocado en un servidor de Internet, instantáneamente el applet puede ser cargado y ejecutado en cualquier parte de mundo.

Applets simples y pequeños. La mayoría de los applets tienen estas características, y se mejoran con el tipo de archivos jar, los cuales pueden contener todas las clases comprimidas para una mejor transferencia desde el servidor.

Seguridad en los applets. Sun al pensar en crear este tipo de programas, también dedico tiempo a la seguridad, restringiendo las operaciones de los applets en la computadora en donde se ejecuta.

Aun con todos los métodos implementados para evitar que los applets sean nocivos, puede haber algunos que aunque no perjudiquen directamente al usuario, si puedan bloquear la computadora, o retener muchos recursos del sistema. Por tales motivos es importante realizar pruebas exhaustivas cuando se programan applets, tomar en cuenta los *bytecodes*, los cuales son una pre-compilación del código fuente de Java, el cual se interpreta con la JVM.

Un programa Java se ejecuta como una aplicación *Stand-Alone* desde la línea de comando o como un applet que corre bajo un navegador de web. Las aplicaciones y los applets comienzan la ejecución de diferentes maneras: una aplicación tiene un método principal *main* ( ), mientras que un applet tiene un método *init* ( ) iniciado por el navegador.

Es importante recalcar que todos los applets deben ser una *subclase* de la clase applet y de esta manera poder usar código que ya está disponible o bien agregarle más código.

El código fuente de un applet se almacena en un archivo con extensión *.java*. Se compila para generar bytecode y se almacena con la extensión .class, la diferencia con las aplicaciones *Stand-Alone* esta en que el archivo *.class* se incluye en el documento HTML usando la etiqueta <applet>, y es ejecutado por el navegador cuando carga un documento HTML que contiene esta etiqueta, es una manera de incluir programas complejos en el ámbito de una página web y se benefician de la potencia de este lenguaje para la Red, la principal ventaja de utilizar applets consiste en que son mucho menos dependientes del navegador.

Esta etiqueta tiene numerosos atributos para realzar su ubicación en el documento, como por ejemplo los que definen el ancho y el alto de la ventana en donde estará el applet, algunos parámetros necesarios, etc.

Atributos obligatorios:

- CODE : Nombre de la clase principal, necesaria para cargar el applet.
- WIDTH : Anchura inicial del rectángulo en el que estará el applet indicados en píxeles.
- HEIGHT : Altura inicial del rectángulo en el que estará el applet indicados en pixeles.

Atributos opcionales:

- CODEBASE : URL base del applet
- ALT : Texto alternativo en caso de que no se pueda visualizar el applet.
- NAME : Nombre de la instancia.
- ALIGN : Justificación del applet dentro de la página.
- VSPACE : Espaciado vertical que habrá entre el texto y el applet, medido en píxeles.
- HSPACE : Espaciado horizontal que habrá entre el texto y el applet, medido en píxeles.

La vida de un applet es controlada por el Navegador en donde se esta ejecutando, iniciando por el método de *init(),* después *start(),* antes de terminar *stop()* y *destroy()* para finalizar el applet.

Antes de sobrecargar estos cuatro métodos, debemos importar las clases o paquetes necesarios para el applet: *java.applet.Applet* y *java.awt.Graphics,* de esta manera le decimos al programa las librerías en las cuales buscará los métodos necesarios, ambas librerías son parte del corazón del API de Java. Obviamente se pueden importar más librerías.

En el lenguaje Java, cada clase es un paquete, si el código fuente de la clase no tiene en la parte inicial *package,* entonces la clase será el *package* por default. Como se menciono anteriormente, los applets deben ser una subclase de la clase applet, aprovechando las características de la POO.

Un applet requiere que la inicialización del código se coloque en el método init(), que es el primer método llamado por el navegador. Además, un applet puede sobrecargar los otros tres métodos para llevar a cabo funciones en determinados momentos del ciclo de vida. Controlaremos el applet mediante los siguientes métodos:

- init: Inicializamos el applet cada vez que es cargado o recargado por el navegador.

- start: Inicia la ejecución del applet, después de que es cargado o el usuario regresa a la página del applet.

- stop: Detiene la ejecución del applet, esto sucede cuando el usuario abandona la página o cierra el navegador.

- destroy: Antes de ser descargado el applet, se realiza una limpieza del sistema en donde se cargo.

El método *init* solo se ejecuta una vez, inicializando las variables necesarias para el programa, funciona como un constructor, de igual manera el método *destroy* es como destructor. No es necesario sobre escribir los cuatro métodos del applet, eso depende de la complejidad del mismo. Los Applets utilizan la concurrencia, es difícil de explicar, pero como indica Doug Lea: *"Un programa concurrente es un programa que hace más de una cosa a la vez"*[2].

2.3 Interacción con los usuarios.

La creación de una Intefaz Gráficas de Usuario (GUI) ha sido una ventaja en Java desde sus orígenes, y tiene la portabilidad al utilizar los elementos del *Abstract Window Toolkit* (AWT, incluidos desde las versiones 1.0 y 1.1 del JDK) sin importar si son aplicaciones comunes o applets. Los elementos del AWT incluyen desde APIs para imprimir, menús, barras de desplazamiento, hasta portapapeles y manejo de imágenes o gráficos.

Se puede construir una GUI con la combinación de los elementos incluidos en el AWT y escuchar los sucesos generados por los mismos; estos elementos pueden ser *Panel, Checkbox, Button, Dialog, Label*, entre otros más.

2.3.1 La POO y los eventos.

Como Java es totalmente orientado a objetos, y con las clases incluidas en el paquete AWT, entonces podemos referirnos a una programación orientada a objetos y conducida por eventos, la cual describiremos a continuación.

La aplicación es creada a partir de las clases fundamentales en Java, implementa una GUI y se ejecuta quedando a la espera de las acciones generadas por el usuario, por el S.O. o por la misma aplicación. Al generarse un evento, Java responde ejecutando el método asociado con este evento de forma predeterminada, es el programador quien redefine este método y controla las acciones. Lógicamente hay muchos eventos posibles para un

[2] Programación Concurrente en Java, Principios y Patrones de Diseño, Doug Lea, Person Educación, Madrid 2001, p. 20

programa, ya sea *Stand Alone* o applet, pero siempre se responderá a cada evento invocando su método asociado.

Para poder crear aplicaciones del tipo GUI es necesario cumplir con los siguientes requisitos:

- Importación de paquetes. Se deben importar los paquetes necesarios para crear las GUI, estos paquetes pueden ser desde *java.awt.event*, *java.graphics*, etc.

- Crear un *contenedor* principal. Este *contenedor* es necesario para poder colocar los elementos gráficos de la aplicación, se crea con la palabra reservada *Frame.*

- Declaración de elementos. Es necesario declarar e inicializar los elementos incluidos en la GUI, como los botones, las etiquetas, los menús, etc. con todas sus propiedades.

- Implementar un *manejador*. En Java conocido como *handler*, y se emplea para llevar el control de los eventos generados mientras el programa se ejecuta; estos eventos pueden ser producidos por el usuario, por la propia aplicación o por el S.O. en el cual se ejecuta el programa.

- Asignar un *listener*. Todos los elementos que necesiten implementar los registros generados por los eventos como un clic, presionar un botón, etc, necesitan un *listener* apropiado para manipular dichos eventos. El *handler* y el *listener* actúan en conjunto.

2.3.2 Manejo de eventos.

Como se menciono anteriormente, los eventos se manejan a través de un conjunto de métodos predeterminados para todos los componentes, pero siempre debe tener asociado el manejador asociado á dichos eventos, de hecho, puede haber más de un componente asociado con un escuchador de eventos.

Los manejadores de eventos más comunes en Java son los siguientes:

- *ActionListener*. Permite a un componente responder a las acciones que ocurran sobre el; por ejemplo, un clic sobre el mouse. Este manejador se puede asociar con componentes *Button, TextField, CheckBox, RadioButton, List y ComboBox*, invocando al método *addActionListener*.

- *KeyListener*. Proporciona los medios para que un componente responda a las acciones procedentes del teclado. Este manejador se puede asociar con todos los componentes, invocando al método *addKeyListener*.

- *ItemListener*. Necesario para que un componente pueda responder a los cambios producidos al actuar sobre él; por ejemplo cuando se selecciona una casilla de verificación. Este manejador se puede asociar con componentes *button, RadioButton, CheckBox y ComboBox*, invocando al método *addItemListener*.

- *MouseListener*. Permite a un componente responder a las acciones del ratón: clics, entrar y salir de un área y la posición del mismo. Este manejador se puede asociar con todos los componentes, invocando al método *addMouseListener*.

- *FocusListener*. Permite a un componente saber cuando gana o pierde el foco. Cuando un componente esta enfocado, se puede actuar directamente sobre el; por ejemplo, si quiere escribir en una caja de texto, esta tiene primero que obtener el foco. Este manejador se puede asociar con todos los componentes, invocando al método *addFocusListener*.

- *WindowListener*. Permite a una ventana responder a las acciones que ocurren sobre ella; por ejemplo minimizarla, abrirla, moverla, etc. Este manejador se puede asociar con componentes *Frame* y *Window* mediante el método *addWindowListener*.

2.4 Flujo de datos E/S

Para Java, existe un paquete de E/S el cual utiliza como base los *stream* también conocidos como flujo de datos, este paquete es conocido como *java.io* y tiene dos partes principales: *stream* de caracteres y *stream* de byte, los primeros son del tipo *Unicode* de 16 bits, y los bytes, como es bien sabido son de ocho bits. La E/S basada en texto trabaja con *streams* de caracteres legibles para los programadores y en general para las personas, es simple texto y son muy útiles en la E/S basada en texto; mientras que la E/S basada en datos funciona con *stream* de datos binarios, los cuales no son legibles fácilmente para las personas, y se emplean para la E/S basada en datos.

Los *stream* que trabajan con bytes no puede transportar adecuadamente caracteres y hay algunos aspectos con caracteres que no tienen sentido. Los *stream* de bytes se denomina *stream* de entrada y *stream* de salida, y los *stream* de caracteres se denomina lectores y escritores. Casi todo *stream* de entrada tiene un correspondiente *stream* de salida, y la mayoría de los *stream* de entrada o de salida tienen su correspondiente lector o escritor de caracteres con similar funcionalidad, y viceversa.

Las clases e interfaces de *java.io* se pueden dividir de forma general en cinco grupos:

Clases generales para construir diferentes tipos de *streams* de bytes y de caracteres como *streams* de entrada y de salida, lectores, escritores y clases para realizar conversiones entre ellos.

Un conjunto de clases que definen diversos tipos de *stream*: filtrados, *buffered*, *piped*, y algunas instancias específicas de dichos *streams*, como el lector de números de línea o el conversor de *streams* de *tokens*.

Las clases e interfaces de *streams* de datos para leer y escribir valores primitivos y cadenas de texto y también existen aquellas para interactuar con archivos independientemente del sistema.

Las clases e interfaces para formar el mecanismo de serialización de objetos, los cuales transforman objetos en *streams* de bytes, y permite reconstruirlos a partir de dichos *streams* de bytes.

2.5 Conexiones en el Web.

En Internet, las conexiones se realizan mediante sockets, un punto en donde dos programas realizan una conexión en red. Un socket es una relación a un puerto mediante el cual el protocolo TCP puede identificar quien se conecta y establecer comunicación. Para Java, las clases del tipo Socket son utilizadas para representar la conexión entre el programa cliente y el programa servidor.

Para realizar una conexión son necesarios dos elementos: un cliente, y un servidor. El cliente es quien realiza la petición para establecer la conexión, debe conocer la dirección IP de la computadora en la cual se esta ejecutando el programa servidor y el puerto por el cual se puede conectar, el servidor solo esta esperando alguna llamada para poder responder y establecer la conexión. En el lado del cliente, si la conexión es aceptada, el socket es creado exitosamente y el cliente puede utilizar este socket para comunicarse con el servidor.

En Java existen varios API's útiles para poder crear conexiones seguras en Internet, en redes locales y también para Base de Datos. Para las conexiones en red, es necesario incluir el paquete j*ava.net,* de esta manera, un cliente Java que utiliza un socket puede trabajar sin ninguna dificultad en redes LAN o WAN; también existe una clase para crear sockets para servidores.

Los sockets incluidos en este paquete trabajan en un trasfondo para cualquier sistema en particular, por lo tanto el programador no se preocupa de los detalles en la conexión, solo de los atributos necesarios para poder establecerla. Una ventaja más de Java en todas las plataformas sobre las que este la JVM.

En Internet, existe un termino conocido como URL (*Uniform Resource Locator*), el cual nos ayuda a manipular las direcciones de una manera más sencilla, es decir, son una dirección o referencia, incluyen el protocolo necesario en la conexión, y la referencia como tal. Java soporta estas conexiones, se pueden crear URL como los que se manejan en cualquier navegador, incluyendo el protocolo, el nombre de la computadora host, la ruta del archivo en especifico y el puerto de entrada, pero una vez que un objeto del tipo URL es creado, no puede ser cambiado en ninguno de sus atributos.

Veremos los pasos generales de como un programa establece una conexión a un servidor utilizando la clase *Socket* (conociendo previamente la dirección IP del Servidor y un puerto disponible para la conexión), como enviar y recibir datos mediante esta conexión, y el orden para eliminar los objetos creados.

Cabe destacar que los programas en Java pueden utilizar dos protocolos distintos en red, el TCP o el UDP, en ambos no es necesario trabajar directamente con las capas que manejan, solo utilizar las clases incluidas en los paquetes de red, los cuales tiene sistemas para realizar conexiones de una manera sencilla. Antes de decidir cual de los dos protocolos se utilizará, es conveniente realizar una evaluación de los pros y los contras de cada uno de ellos

De esta manera, las clases *URL, URLConnection, Socket y ServerSocket* son utilizadas en una conexión TCP, mientras que para el tipo UDP se utiliza *DatagramPacket*, *DatagramSocket* y *MulticastSocket*.

Una vez identificado el tipo de protocolo a utilizar, se deben incluir los paquetes necesarios (*java.net*), crear la clase principal con la cual se trabajará, después se crea el objeto socket derivándolo de la clase *Socket*, incluida en el paquete *java.net*, con dos parámetros: nombre del host o servidor, y el número de puerto por el cual se conectará. Es importante tener estas líneas de código dentro de un bloque *try – catch*, para poder atrapar las excepciones que se puedan generar en estas operaciones y evitar en lo mayor posible errores.

Cuando se establece la conexión, comenzamos a realizar las operaciones de comunicación entre el cliente y el servidor, utilizando funciones de entrada/salida incluidas en la librería *java.io*, como *PrintWriter*, *BufferedReader*, *InputStreamReader*, etc.

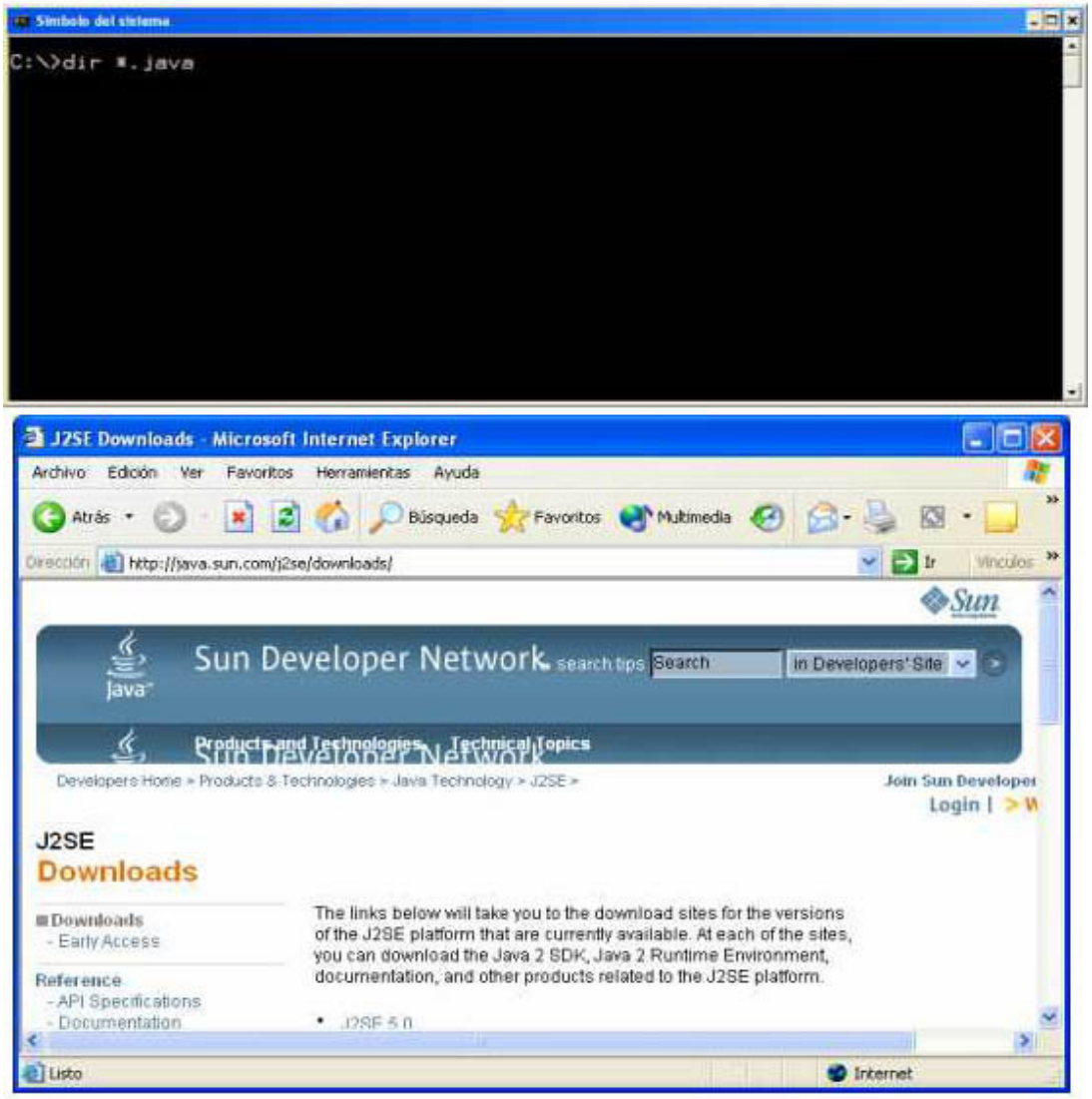
Si estamos en Internet, podemos crear un objeto del tipo URL para poder manejar las direcciones de algún servidor en especifico, se debe tomar los parámetros necesarios para crear el URL, después mediante los métodos de este objeto podemos obtener varias de las propiedades, como el protocolo, el nombre del host, etc. También podemos conectarnos directamente al servidor mediante el método *openConnection,* el cual inicializa una comunicación entre el programa Java y el servidor. En caso de que no sea posible establecer una conexión, ya sea porque el servidor este fuera de servicio o no este disponible el puerto, se lanzará una excepción.

Después de realizar todo el trabajo con la conexión, se deben cerrar las operaciones de entrada/salida y los sockets creados, el orden es importante: se deben cerrar los stream conectados al socket y al final cerrar el socket.

Capítulo 3: Interfaz Gráfica.

3.1 Importancia de GUI.

La interfaz de usuario permite a los clientes interactuar con el programa, al principio, eran desde línea de comando, como MS-DOS o UNIX, pero actualmente proliferan y son de uso común las interfaces gráficas con ventanas. La interfaz de usuario es un aspecto importante para las aplicaciones, pues con ellas se comunica el programa y el usuario, al nivel más bajo, el sistema operativo envía información desde los dispositivos de entrada, como el ratón y el teclado, dicha información se captura desde la interfaz de usuario.



Ejemplos de interfaz gráfica

3.1.1 Aspectos generales de la GUI.

En la actualidad, la iteración con los usuarios es mediante interfaces gráficas; sistemas operativos como Windows, *MacOs*, e inclusive *Linux* emplean estos medios para comunicarse con el usuario. Por sus siglas en ingles, GUI (*Graphics User Interfaz*), es en términos generales, un programa generado con ventanas, listas de selección, botones, *checkbox*, *menús*, etc.

En sus inicios, las aplicaciones basadas en GUI eran difíciles de programar, era necesario escribir mucho código, dar seguimiento a los eventos generados por el usuario era conflictivo y la comprensión de los programas no era fácil. En Java, la programación de GUI se realiza a través del AWT (*Abstract Window Toolkit*) incluido desde el JDK, el cual se acompaña con elementos necesarios para crear GUI, pero simples. Lo mejor del AWT, es como todo en Java, la portabilidad, aunque la apariencia de las aplicaciones puede variar un poco, dependiendo del S.O. en donde se ejecuten.

El AWT fue diseñado para ayudar al programador y evitar que se preocupe de los detalles como controlar el movimiento del ratón o leer el teclado, ni tampoco atender los problemas de la escritura en pantalla. El AWT constituye una librería de clases orientada a objeto para cubrir estos recursos y servicios de bajo nivel.

Con Java, hay dos maneras para tratar los eventos generados por la GUI, una de ellas es mediante la implementación del *ActionListener* en una clase, la cual interceptará el evento especifico y la otra forma es mediante el *actionPerformed*, el cual interceptará todos los eventos. Estas dos maneras de tratar los eventos se verá a detalle más adelante en la sección Manejo de Eventos.

3.2 AWT y Swing

En la versión del SDK 1.4 hay dos maneras de crear interfaces gráficas, la primera y la más antigua es mediante el AWT, la segunda es mediante los componentes Swing, los cuales son parte del JFC (*Java Foundation Classes*), estos últimos pueden utilizarse desde el JDK 1.1 o en la plataforma Java 2.

La diferencia que en dimensiones presentan los componentes gráficos varía de una plataforma a otra, por tal motivo un programa de ambiente visual que ubique los componentes de acuerdo a las coordenadas exactas no será útil, pues cuando la aplicación creada con este ambiente visual se utilice en otra plataforma distinta en la que fue creado, es seguro que los componentes tengan una ubicación y un aspecto diferente.

Cuando se desarrolla una aplicación, hay elementos Swing los cuales son contraparte de los elementos AWT, estos elementos tiene el mismo nombre que los AWT con la particularidad de que su nombre empieza con la letra J, como ejemplo tenemos el *Frame y Jframe*, como se menciono anteriormente el AWT es un conjunto de clases muy básica, útil para liberar al programador de la preocupación de los detalles, y Swing es más robusta y con más atributos.

3.2.1 AWT

Con anterioridad se ha mencionado sobre el AWT, el primer conjunto de paquetes de Java para crear GUI, por tal razón no es muy robusto, pero para aplicaciones sencillas resulta útil, además, muestra los componentes utilizando es aspecto del S.O. en el que se ejecute la aplicación, es decir, los botones tendrán un aspecto de *Macintosh* en una *Mac*, del tipo Motif en una plataforma X o botones de Windows en un sistema de Microsoft.

En el AWT se incluyen diversos administradores de interfaz gráfica, estos administradores son el *FlowLayout, GridBagLayout*, etc. La cualidad de estos administradores consiste en conocer en donde ubicar los componentes dentro de la pantalla mediante una ubicación relativa a los demás. Aunque en el AWT es posible colocar los elementos gráficos mediante una posición absoluta, pero no se recomienda en resumen, *"es un conjunto de clases que implememantan el Abstract Windows Toolkit necesarios para la interfaz gráfica"*[1].

Por estos motivos, cuando deseamos crear una GUI en Java, debemos imaginar como ubicar los componentes en relación con los demás componentes, obviamente utilizando alguno de los administradores gráficos incluidos en Java; esto nos ayudará a crear GUI capaces de ser visualizadas correctamente en las distintas plataformas.

Como la estructura básica del AWT se basa en Componentes y Contenedores, y estos últimos contienen Componentes posicionados a su respecto y son Componentes a su vez, de forma que los eventos pueden tratarse tanto en Contenedores como en Componentes, entonces, será el programador quien se encargará de llevar el control de los elementos.

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

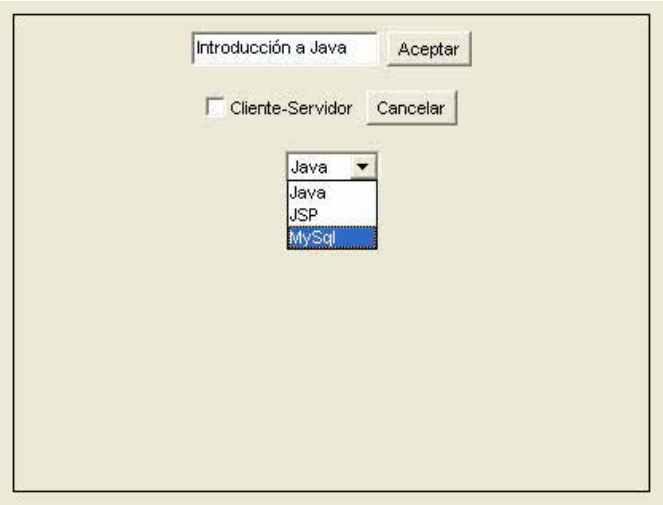
Component es una clase abstracta que representa cualquier componente con representación gráfica, Container es un componente el cual puede contener a otros componente gráficos como un Panel, y *Frame* permite representar ventanas.

Los frames son contenedores, por lo que incluyen un panel de contenido (content pane) al cual se le pueden añadir componentes gráficos como etiquetas, botones, cajas de texto, etc. y paneles. Estos *frames* tienen una clase encargada para escuchar los eventos generados en la ventana, esta clase se llama *WindowsListener*, e incluye siete métodos, los cuales corresponden a las distintas acciones realizadas sobre una ventana:

[1] Java for C/C++ Programmers, Michael C. Daconta, Wisley Computer Publishing, U.S.A. 1996 p. 151

- *WindowActivated.* Se invoca cuando la ventana se muestra de segundo al primer plano.
- *WindowDeactiveted.* Cuando la ventana pasa a un segundo plano.
- *WindowIconified* Si la ventana se minimiza, se llama este método.
- *WindowDeiconified* Al restaurar la ventana, es decir para de minizada a mostrarse en pantalla.
- *WindowOpened* Abrir la ventana.
- *WindowClosed* Cerrar la ventana.

La siguiente figura muestra una applet generado con AWT:



Applet generado con AWT

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

Como ya vimos, el AWT incluye todo un conjunto de componentes gráficos como botones, etiquetas, *checkbox*, etc., pero también es posible implementar gráficos con este conjunto de paquetes y la librería *Graphics*. Los elementos que dibujemos los incluiremos en el método *paint*, las gráficas pueden ser desde líneas, círculos, utilizar una gran variedad de colores, un manejos de fuentes y de tipo diverso, etc. e incluso podemos actualizar el área de pintado al utilizar el método *repaint*.

Para gestionar los componentes gráficos es necesario utilizar un *Listener*, el cual nos permitirá manejar los eventos generados con cualquier componente gráfico, por ejemplo un botón; en resumidas cuentas, lo primero que se hace es el aspecto gráfico, como los botones, los menús, etc., después se crean y se añaden los oyentes de los sucesos que, según el suceso, invocan uno u otro método de gestión del programa; Éste es un principio general

de la programación de Interfaces Gráficas, es decir, de la programación de los sucesos. Cada programa que se implemente, funciona de esta forma.

La estructura del AWT se puede resumir en los puntos que se exponen a continuación:

- Los Contenedores comprenden Componentes, que son los controles básicos
- Alto nivel de abstracción respecto al entorno de ventanas en que se ejecute la aplicación (no hay áreas cliente, ni llamadas a X, ni hWnds, etc.)
- Para gestionar los componentes gráficos es necesario utilizar un *Listener*
- No se usan posiciones fijas de los Componentes, sino que están situados a través de una disposición controlada (*layouts*)
- La arquitectura de la aplicación es dependiente del entorno de ventanas, en vez de tener un tamaño fijo
- Es bastante dependiente de la máquina en que se ejecuta la aplicación (no puede asumir que un diálogo tendrá el mismo tamaño en cada máquina)
- El común denominador de más bajo nivel se acerca al teclado, ratón y manejo de eventos

El tipo de programación es un poco diferente al que un programador sobre plataforma no gráficas (tipo Dos, o línea de comandos) o secuencial está acostumbrado y al principio puede resultar no muy simple, sin embargo es necesario acostumbrarse.

3.2.2 JFC y los componentes Swing

JFC es la abreviación de Java *Foundation Classes*, lo cual comprende un conjunto de características para construir GUI y agregar funcionalidades gráficas e interacción con las aplicaciones Java.

El JFC fue anunciado por primera vez en 1997, en una conferencia de *JavaOne*, tiene un conjunto de características mostradas a continuación:

Características de Java Foundation Classes	
Característica	**Descripción**
Componentes Swing GUI	Incluye cada elemento gráfico desde botones hasta paneles para crear tablas.
Soporte para Look-and-Feel	Para cualquier programa, proporciona una elección del *look and feel*. Por ejemplo, el mismo programa puede visualizarse con la apariencia proporcionada por Java o por Windows. Varios paquetes de *look-and-feel* están disponibles, en la versión 1.4.2 de Java, soporta el GTK+ look and feel.
Accesibilidad del API	Habilita diversas tecnologías de las pantallas y *display* en Braille y así obtener información del usuario.
Java 2D API	Proporciona a los programadores una forma fácil de

	incorporar graficas en 2D de alta calidad, texto e imágenes en applets. Java 2D incluye API's para generar y enviar salidas a dispositivos como impresoras.
Soporte para arrastrar y soltar (Drag-and-Drop)	Proporciona la habilidad de arrastrar y soltar entre Java y aplicaciones nativas..
Internacionalización	Permite a los desarrolladores construir aplicaciones que puedan interactuar con usuario del *worldwide* en sus propios leguajes y convenciones culturales. Con la entrada del *framework*, se pueden generar aplicaciones las cuales acepten texto en lenguajes que tienen miles de caracteres diferentes como el Japonés, Chino o Coreano.

Características del JFC

Para ejecutar los programas generados con Swing, solo es necesario tener la JVM actualizada con la versión con la cual fue compilado el programa (o una versión posterior), se pueden crear aplicaciones *Stand Alone* o applets, y pueden ser ejecutados con *Java Web Start*, la ventaja de utilizar esta última consiste en que cuando se ejecuta por primera vez una aplicación, *Java Web Start* baja automáticamente los paquetes necesarios.

En las versiones 1.4 y posteriores *Java Web Start* esta incluido como parte de la plataforma Java, por lo tanto si instala J2SE o JRE v1.4.1 o una versión posterior, entonces tiene instalado *Java Web Start*, de esta manera los navegadores pueden ejecutar las aplicaciones con *Java Web Start*.

Cabe destacar que los paneles como *Jframe*, además de contener otros componentes tienen una superficie sobre la cual se puede dibujar, llamada *canvas* o lienzo, por ejemplo para dibujar en un *Jpanel*, hay que redefinir el método *paintComponent* de la clase *Jcomponent* el cual es invocado cada vez que hay que refrescar la visualización del componente en pantalla, un ejemplo de cuando se refresca es al mover o redimensionar la ventana.

3.3 La clase Panel y los Layout.

Antes de construir una GUI es importante conocer como se distribuirán los componentes en el área de trabajo, para tal fin se crearon los Layout *Managament*, utilizados para controlar la forma en como se colocan los componentes en un contenedor, veamos esta distribución del espacio más a detalle.

El manejo de los Layout dentro de las GUI se utiliza para controlar el tamaño y la posición de los componentes dentro de la pantalla; existen cinco formas para implementar los Layout*: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, and GridLayout*, cada uno con sus propias características.

Por default, cada contenedor tiene una Layout manager, todos los objetos Panel utilizan por default *FlowLayout*, los contenedores principales como applet, *Dialog*, y *frame* utilizan *BorderLayout*, obviamente el tipo de manejador que utiliza por *default* un panel o un *content pane* puede ser modificado, solo hay que invocar el método container *setLayout*. En general, la única vez cuando se piensa en los manejadores de Layout es cuando se crea un panel o se añaden componentes para ubicarlos en pantalla.

```
Panel pane = new Panel();
pane.setLayout(new BorderLayout());
```

3.3.1 BorderLayout

Cuando se utiliza este manager, los componentes se ordenan y redimensionan en cinco regiones, las cuales son, por sus nombre en inglés: *north, south, east, west, and center* y corresponde a los puntos cardinales de acuerdo a su nombre. Cada región es identificada por una constante: NORTH, SOUTH, EAST, WEST, y CENTER; cuando se añaden componentes a un container con *BorderLayout*, se utiliza alguna de las cinco constantes, si no se especifica constante alguna, se interpreta como si fuera CENTER.

```
Panel panel = new Panel();
panel.setLayout(new BorderLayout());
panel.add(new Button("Boton Uno"), BorderLayout.SOUTH);
```

Además, *BorderLayout* soporta cuatro posicionamientos relativos, *BEFORE_FIRST_LINE, AFTER_LAST_LINE, BEFORE_LINE_BEGINS, y AFTER_LINE_ENDS*. Es decir, si en un *container* en donde el componente de orientación se especifica *ComponentOrientation.LEFT_TO_RIGHT*, entonces la ubicación será *NORTH, SOUTH, WEST, y EAST*, respectivamente.

3.3.2 BoxLayout

BoxLayout es una administrador de GUI el cual nos permite colocar componentes de manera horizontal o vertical; los componentes mantendrán su ubicación aun cuando se cambie de dimensión el *frame*, son colocados de izquierda a derecha o de arriba a abajo y son colocados en alineación con el tamaño máximo posible.

Varios programas utilizan la clase *Box* en lugar de utilizar el *BoxLayout* directamente, esto es debido a que la clase *Box* provee un componente del tipo *lightweight* y varios métodos que facilitan su implementación.

3.3.3 FlowLayout

Los componentes con *FlowLayout* son colocados en línea, de izquierda a derecha, son comúnmente usados para colocar botones en un panel hasta que no haya espacio para más componentes, los componentes en cada línea son centrados. Cuando una clase se deriva de *Applet*, *FlowLayout* es el administrador gráfico por default.

Tiene cuatro variables para indicar la alineación de los componentes. *LEFT*, indica que la alineación será con justificación a la izquierda. *CENTER*, los componentes serán colocados en el centro. *RIGHT*, la justificación de los componentes será a la derecha. *LEADING*, aquí, los componentes se colocaran en la primera línea. *TRAILING*, es muy similar a *LEADING*, pero la orientación se invierte.

```
Panel panel = new Panel();
panel.setLayout(new FlowLayout(FlowLayout.LEFT));
```

3.3.4 GridLayout

Otro administrador de Java es *GridLayout,* el cual permite colocar los componentes en una rejilla rectangular, esta rejilla esta divida en rectángulos del mismo tamaño, y cada componente es colocado en uno de ellos. Las variables principales utilizadas en el constructor de esta clase son el número de renglones y de columnas.

En casos excepcionales, el número de columnas es determinado por el número de renglones y el total de componentes. Otras variables utilizadas en esta clase son *hgap, vgap,* ambas en valores entero y especifican la distancia entre los componentes en forma horizontal y vertical, respectivamente.

```
setLayout(new GridLayout(3,2,2,2));
```

Este ejemplo especifica una rejilla de tres renglones y dos columnas, con una separación vertical y horizontal de 2 unidades.

3.3.5 GridBagLayout

GridBagLayout es el administrador más sofisticado que existe para Java, el cual nos permite alinear componentes de manera vertical y horizontal, permitiendo que algunos componentes se expandan a más de una celda, y sin más requerimientos de que los componentes sean del mismo tamaño que el área de visualización. Cada objeto *GridBagLayout* mantiene una arreglo rectangular de celdas, un componente puede ocupar una o más celdas, llamada área de despliegue o visualización.

Cada componente manejado por *grid bag layout* esta asociado con una instancia de *GridBagConstraints*, la cual especifica como serán colocados los componentes dentro del área de visualización.

La manera en como colocará *GridBagLayout* los componentes, depende del objeto *GridBagConstraints* asociado con cada uno de los componentes y el tamaño mínimo dentro del *container,* es por esta razón por la cual para optimizar el manejo de esta clase, se debe personalizar una o más de los *GridBagConstraints,* lo cual se logra mediante el uso de las siguientes variables: *gridx, gridx, gridwidth, gridheight, fill, ipadx, ipady, insets, anchor, weightx, weighty.*

3.4 Manejo de Eventos

Los eventos son uno de los aspectos más importantes en entornos gráficos como, el manejo de eventos en Java se utiliza para delegar la interacción del programa con el usuario, y como en Java no existen los punteros, la implementación de los eventos es mediante el uso de interfaces incluyendo los *Listeners*.

3.4.1 Funcionamiento de los Eventos

Cuando se trabaja con entornos gráficos como Windows la entrada por parte de usuario es a través de elementos gráficos y una misma acción puede realizarse de distintas formas. Por ejemplo, una opción de menú tiene distintas maneras de activarse:

Podemos hacer clic sobre el menú, o bien utilizar el acceso directo del menú (la combinación de teclas que aparece al lado del nombre, como Ctrl+V, F5, etc.) o incluso podemos utilizar los aceleradores del teclado asignados al menú (la letra subrayada que aparece en el nombre, como Archivo, Edición, etc.)

En el momento de recibir el mensaje, Java interpretará por nosotros las distintas posibilidades, y lanza el evento correspondiente. De este modo, en el caso de un botón, no tenemos que preocuparnos de lo que haya ocurrido (ha hecho clic, ha pulsado Enter...) sino que finalmente nuestro botón ha sido pulsado, por el método que sea, por lo que posiblemente, algo tendremos que hacer.

Como vemos, tenemos muchos caminos para llegar al mismo destino, y en ciertas ocasiones sería demasiado complicado controlar todas estas opciones. Es por esto que la interfaz gráfica nos “avisa” cada vez que ocurre algo importante, de esta manera asignaremos acciones de acuerdo a la entrada del usuario.

Cada uno de estos “avisos” recibe el nombre de “mensaje” y existen gran cantidad de ellos: cada vez que se pulsa el botón derecho del ratón, cuando se presiona una tecla del teclado, cuando se pinta la ventana, etc., estos mensajes son asignados de manera única para poder ser identificados cada uno de ellos.

Por ello, la programación basada en eventos encaja muy bien en los entornos gráficos, ya que cada vez que la aplicación reciba un mensaje del sistema operativo, podremos lanzar un evento al programador.

3.4.2 Los eventos en Java

Los eventos en Java se definen mediante clases auxiliares llamados *event listener*, los cuales pueden recibir eventos de tipo específico por ejemplo el clic del ratón. Estas clases se asocian a componentes específicos.

Cuando se utiliza el AWT, los eventos se organizan en una jerarquía de clases dentro del paquete *java.awt.event* de la siguiente manera:

- La clase *java.util.EventObject* es la clase base de todos los eventos en Java.
- La subclase *java.awt.AWTEvent* es la clase base de todos los eventos utilizados en la construcción de GUI.
- Cada tipo de Evento *XxxEvent* tiene asociada una interfaz *XxxListener*, la cual nos permite definir manejadores de Eventos.
- Para simplificar la implementación de algunos manejadores de eventos, el paquete *Java.awt.Event* incluye clases base llamadas *XxxAdapter* que implementa las interfaces *XxxListener*.

Por ejemplo, para gestionar los eventos del ratón, usaremos *MouseAdapter*, para controlar los eventos de una ventana recurriremos a *WindowAdapter* y para especificar lo que hará nuestra aplicación cuando se pulse un botón emplearemos *ActionListener*.

Los Componentes Swing no soportan el modelo de eventos de propagación, sino solamente el modelo de Delegación incluido desde el JDK 1.1; por lo tanto, si se van a utilizar Componentes Swing, se debe programar exclusivamente en el nuevo modelo. Aunque hay Componentes de Swing que se corresponden (o reemplazan) a componentes de AWT, hay otros que no tienen una contrapartida en el AWT.

Desde el punto de vista del manejo de eventos, los Componentes de Swing funcionan del mismo modo que los Componentes del AWT, a diferencia de los nuevos eventos que incorpora Swing. Desde otros puntos de vista, los Componentes Swing pueden parecerse ya más o menos a su contrapartida del AWT. Además, hay una serie de Componentes muy útiles en Swing, como son los árboles, la barra de progreso, los *tooltips*, que no tienen ningún Componente que se les pueda equiparar en el AWT.

Para algunos programas creados en AWT cuando se desea convertir a Swing, solamente se reemplaza cada instancia de *Frame* por una instancia de *JFrame*, además de incorporar la sentencia *import* que hace que las clases sean accesibles al compilador y al intérprete. Por lo demás, el control de eventos es el mismo que se ejercía en el AWT, pero esto solo es en programas sencillos.

Cuando utilizamos los eventos, podemos comprobar la utilización de fuentes de eventos, receptores de eventos y adaptadores del Modelo de Delegación de Eventos para Componentes Swing. La aplicación instancia un objeto que crea un interfaz de usuario consistente en un *JFrame*. Este objeto es una fuente de eventos que notificará a dos objetos diferentes, receptores de eventos, de eventos de tipo Window.

Uno de los objetos *Listener*, receptores de eventos, implementa el interfaz *WindowListener* y define todos los métodos que se declaran en ese interfaz. El otro objeto *Listener*, extiende la clase *Adapter*, adaptador, llamada *WindowAdapter*, que ya no tiene porqué sobre escribir todos los métodos del interfaz, sino solamente aquellos que le resultan interesantes.

3.4.3 Manejo de eventos en Java

En párrafos anteriores, mencionamos que hay dos maneras para tratar los eventos generados por la GUI, una de ellas es mediante la implementación del *ActionListener* en una clase, la cual interceptará el evento especifico y la otra forma es mediante el *actionPerformed*, el cual interceptará todos los eventos.

En la primer forma de interceptar los eventos, el AWT los recibe en alguno de los dos niveles siguientes: a nivel de los mismos elementos o también a nivel de los contenedores, es decir, que para determinar si el usuario presiono el botón derecho o izquierdo del ratón, se redefine el método action del botón y por lo tanto será necesario crear una clase derivada de la clase *Button*, la cual tendrá la función de ser un objeto de escucha intermediario.

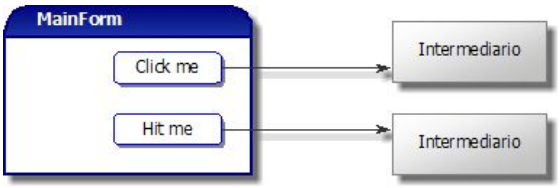
```
// Clase auxiliar
public class Intermediario implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
                ...
  }
}

public class MainForm
  extends Frame implements ActionListener {
  Button b1, b2;

  public MainForm()   {
    // ...
    b1 = new Button("Clic aqui");
    // Creamos el primer intermediario
    b1.addActionListener(new Intermediario());
    // ...
    b2 = new Button("Aceptar");
    // Creamos el segundo intermediario
    b2.addActionListener(new Intermediario());
    // ...
  }
  // ...
}
```

En el elemento se indica el objeto con el cual interactuó el usuario, cuando este evento no se implementa con una clase, no puede ser procesado por nuestra aplicación y debe ser propagado hacia el contenedor de este elemento. Esto significa que se invocará el método *action* del *contenedor*.

Al implementar la clase auxiliar, el contenedor no procesa el mismo evento, es aquí en donde nosotros determinamos que hacer con la entrada del usuario. Cuando no se redefine *action* mediante una clase, debe ser el contenedor el que procesa el evento.

La otra forma de recibir eventos es en la raíz, pero el problema de la raíz consiste en recibir los eventos de todos los elementos gráficos y por lo tanto hay que comparar lo que envía el AWT a través de la GUI con cada uno de los elementos contenidos en la raíz, para poder determinar cuál es la acción a desarrollar por la aplicación.

La imagen nos muestra como queda implementado esta manera de manejar los eventos:



Manejo de Eventos

Cuando se pulsa alguno de los botones, el aviso se envía a uno de los objetos de tipo Intermediario creados, es decir, se ejecuta un método de uno de estos objetos. Resultado: un nuevo objeto por cada evento interceptado, pero no tendremos el problema de compartir un mismo método entre montones de eventos. Ahora bien, ¿qué puede hacer un "intermediario" una vez que recibe un evento? Pues muy poca cosa, mientras no tenga acceso al objeto donde hemos situado los botones.

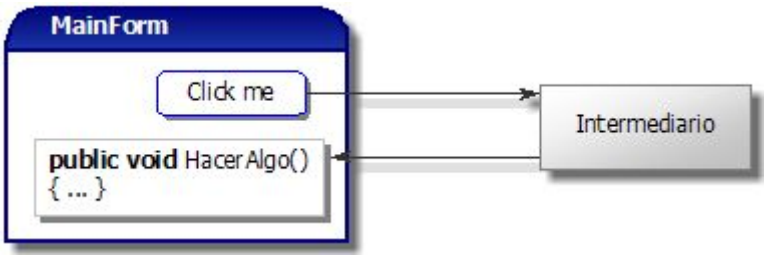
Por lo tanto, lo más frecuente será que a la clase Intermediario se le facilite un puntero a la instancia donde realmente está teniendo lugar el espectáculo:

```
public class Intermediario implements ActionListener{
  MainForm mainForm;

  public Intermediario(MainForm mf): mainForm(mf) {}

  public void actionPerformed(ActionEvent ev)  {
    // Aquí hacemos algo sobre "mainForm"
  }
}
```

Para asociar un receptor al evento del primer botón haríamos esto:

```
b1 = new Button("Clic aqui");
// Creamos el primer intermediario
b1.addActionListener(new Intermediario(this));
```

Si sólo mostramos un botón, para simplificar, la nueva situación se parecerá a la del siguiente diagrama:



Manejo de Eventos con punteros

El problema principal de esta técnica es la proliferación de clases que provoca. Para cada evento manejado, hay que crear una clase, y una instancia de esa clase; incluso es más absurdo: una sola instancia de la clase intermediaria. Una de las consecuencias de esta epidemia es que habría que inventarse nombres diferentes para cada clase intermediaria, para evitar conflictos. Además: ¡habría que escribir muchísimas líneas de código para atrapar un simple evento! Manejando clases anidadas podemos evitar los conflictos en el primer nivel de nombres, las clases anidadas son aquellas definidas dentro de otras clases.

Nos interesan en concreto las clases anónimas, o *anonymous classes*: se trata de clases sin nombres que se definen como parte de una instrucción, por ejemplo:

```
b1 = new Button("Clic aqu");
// Creamos el primer intermediario
b1.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ev) {
        // ¡¡¡Este método tiene acceso directo...
        // ... a la instancia que lo contiene!!!
        b1.setVisible(false);
        // Recuerde que b1 es un campo de la clase exterior
        }
});
```

El texto destacado en azul, y encerrado entre llaves, corresponde a toda una declaración de una clase. Antes del bloque que encierra la clase, el operador *new* crea una instancia de esta clase sin nombre. La clase implementa, sin que tengamos que hacerlo explícito, el tipo de interfaz *ActionListener*. Lo más interesante, sin embargo, es lo que destaca el comentario: la instancia creada de la clase anónima puede referirse a los campos y métodos de la clase que la encierra sin mayor dificultad. En el ejemplo anterior, se esconde el botón pulsado, algo muy sencillo, y se hace referencia a la variable b1 como si se tratase de un campo definido dentro de la clase anónima.

Está claro que el compilador de Java, a marcha forzada, ha añadido un parámetro "oculto" al constructor de la clase anónima para pasar una referencia a la clase que la contiene. Esa referencia se almacena en un campo privado añadido por Java a la clase anónima. Esta clase también podría utilizar variables locales del método donde se instancia.

En ese caso, la técnica usada por el compilador consiste en copiar el contenido de esas variables en campos locales de la clase anónima, nuevamente gracias a manipulaciones en el constructor de la misma

Para las clases anónimas, no se puede crear un constructor, los constructores en Java, al igual que en C++ y C#, se definen usando el mismo nombre de la clase, algo evidentemente imposible en el caso que estamos analizando. Se trata de una de las anomalías y peligros introducidos por este riesgoso recurso. Tenga en cuenta que, aunque las clases anónimas resuelven el problema de la proliferación de nombres de clases, no resuelven la proliferación de las propias clases. El código final es bastante engorroso de escribir, leer y mantener (cuente y vea los paréntesis y llaves necesarias) y, como demuestran las pruebas, el mecanismo resultante es bastante lento.

Existe una última forma no recomendada de captar eventos usando *handleEvent*. Lo delicado de este método es que recibe todos los eventos, por lo tanto es fácil introducir errores atrapando eventos que deberían procesarse en otro punto en la jerarquía de componentes.

Veremos un ejemplo de cómo se crea una ventana utilizando la clase *frame* del AWT, no olvidemos que los *frames* son ventanas en los cuales se pueden colocar otros elementos, el código fuente para generar este sencillo programa se muestra a continuación:

```
import java.awt.Frame;

class frameUno {
  public static void main(String args[]) {
    Frame ventana = new unaVentana();
    ventana.setVisible(true);
  }
}//Fin de la clase frame Uno

class unaVentana
    extends Frame {
  unaVentana() {
    this.setTitle("Fame de Prueba");
    this.setSize(400, 100);
  }
}
```

En el código anterior, si hacemos clic para cerrar la ventana, esta no se cerrara, pues todavía no hemos manejado el evento para cerrar la ventana. La clase encargada para escuchar los eventos generados en la ventana es *WindowsListener*.

La interfaz *java.awt.event.WindowListener* incluye siete métodos, los cuales corresponden a las distintas acciones realizadas sobre una ventana: *windowActivated*, *windowActiveted*, *windowIconified*, *windowDeiconified*, *windowOpened* y *windowClosed*

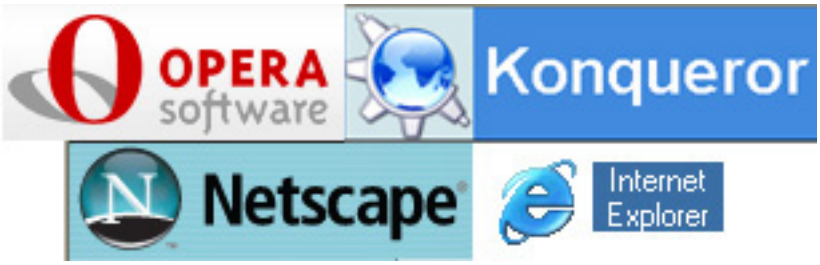
El código ya corregido queda de la siguiente manera:.

```
import java.awt.Frame;
import java.awt.event.*;

class frameUno {
  public static void main (String args []) {
    Frame ventana = new  unaVentana ();
    ventana.setVisible(true);
  }
}

class unaVentana extends Frame {
  unaVentana() {
    this.setTitle ("Frame de Prueba");
    this.setSize (400,100);
    this.addWindowListener(new ventanaListener ()) ;
  }
}

class ventanaListener extends  WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit (0);
    }
}
```

Ejemplo del código del programa

Capítulo 4: Programación en JSP

Para modificar el enfoque del Internet estático generado por páginas HTML puras, es necesario realizar una serie de programas que respondan a las peticiones del usuario, y generen páginas web de acuerdo a la entrada del usuario y crear así Webs dinámicas, para tal objetivo, es necesario utilizar una tecnología como ASP, JSP, PHP, etc o bien implementar un *Common Gateway Interface* (CGI) y así permitir a los clientes mediante un Navegador como Internet Explorer, Nestcape, Konqueror, Opera. etc. solicitar datos de un programa o ejecutar rutinas en un servidor Web.



Navegadores Web

4.1 Por que JSP

Si deseamos contestar la pregunta de ¿Por *qué* JSP?, es necesario evaluar las cualidades de los JSP y como se hace en el siguiente apartado, realizar comparaciones con las otras tecnologías.

Por principio de cuentas, los componentes Web son preferidos para crear UI utilizadas por un cliente Web, debido a que no es necesario instalar un plug-in en el lado del cliente, además se pueden crear aplicaciones más limpias y de una manera modular al presentar una forma de separar el diseño de la pagina web de la programación para la aplicación, por ejemplo, las personas involucradas en el diseño de las páginas no necesitan entender la sintaxis del lenguaje JSP o de Java para realizar su trabajo.

Dada la popularidad del lenguaje Java, existen en la actualidad varias formas de usarlo dentro de un servidor Web, de entre ellas destacamos dos: mediante servlets, pequeños programas en Java que se ejecutan de forma persistente en el servidor, y que, por lo tanto, tienen una activación muy rápida, y una forma más simple de hacerlo es con los JSP (*Java Server Pages*).

JSP, es un lenguaje de programación para agregar funcionalidad a un servidor web mediante la ejecución de programas o pequeños trozos de código en Java que se insertan dentro de páginas web, de forma análoga a los *ASPs*, generan las respuestas a las peticiones del usuario.

Este lenguaje hereda las características de Java como la estabilidad, multiplataforma, el manejo de excepciones, ausencia de punteros, e implementa el uso de etiquetas personalizadas, el manejo en conjunto con el lenguaje XML, manejo de JDBC, etc., estas características han agregado funcionalidad a JSP y que sea considerado como un potente entorno de desarrollo.

Un uso frecuente de estos programas es en procesamiento de formularios de datos, donde un navegador de Internet envía los datos a un *script* (programa) y este programa los agrega a una base de datos, y devuelve una página web de respuesta al navegador.

4.1.1 Diferencias entre JSP y Servlet

Los *Servlet* son una tecnología de Java en respuestas a la programación CGI, estos programas se ejecutan en un servidor Web, y generan páginas Web dinámicas, utilizadas comúnmente por varias razones. Se distinguen por que el código de presentación (es decir, lo que genera la interfaz de HTML) se compila dentro de la clase y como los *Servlet* se encuentran ya compilados, la ejecución es más rápida (en la primera petición).

El contenedor JSP proporciona un motor que interpreta y procesa una página JSP como un *Servlet* (en *Tomcat*, dicho motor se llama *Jasper*), las páginas JSP son transformadas a un *Servlet* y después compiladas.. Al estar basadas en los *Servlet*, las distintas peticiones a una misma página JSP son atendidas solo por una instancia del *Servlet*.

De una manera sencilla, los *Servlet* son programas en Java, los cuales de una manera dinámica procesan una petición y generan una respuesta, las páginas JSP son documentos basados en texto que ejecutan *Servlets*, pero permiten una manera más natural de generar el contenido estático en el Web.

Ambas opciones, hoy en día, son utilizadas en sitios de comercio electrónico. Frente a las demás tecnologías, tienen la ventaja de ser independientes del sistema operativo y del procesador de la máquina, para poder tener una visión más especifica de estas dos tecnologías, a continuación se exponen punto claves de las mismas:

- En una página JSP el código de presentación puede ser actualizado por un diseñador web que no conozca Java, para un *Servlet*, esta presentación se hace dentro del mismo código, y como se compila, es necesario conocer, al menos de manera básica, el lenguaje Java.

- Los *Servlet* se encuentran ya compilados, mientras que las páginas JSP se compilan bajo petición, por tal motivo, la ejecución del *Servlet* es más rápida durante la primera petición.

- Los *Servlets* no proporciona nada que en principio no se pueda genera con un JSP, pero es más fácil escribir y modificar archivos HTML que una instrucción para generar HTML, como lo hace un *Servlet*.

- Con el punto anterior, si separamos la vista del contenido (como se hace mediante el modelo MVC), se puede tener a diferentes personas en diferentes tareas: los expertos en el diseño de las páginas Web construyen el código HTML, dejando lugares específicos para el contenido generado por los programadores de dicho contenido.

4.1.2 Utilizar JSP o Servlet

Todavía podemos preguntarnos por qué dejar los *Servlets* de lado y utilizar JSP o viceversa, pero ambos son importantes y deben quedarse, de hecho, se puede ver a los JSP como una abstracción de alto nivel de *Servlets* puestos en práctica como una extensión del API *Servlet* 2.1. De todos modos no se debe usar los *Servlets* de manera indiscriminada, pues no son apropiados para todos los casos, por ejemplo, mientras los diseñadores de página, fácilmente pueden escribir una página de JSP que usa HTML convencional o instrumentos XML, los *Servlets* son mejores para el desarrollo del núcleo de la aplicación, porque a menudo son escritos usando un IDE, un proceso que generalmente requiere un nivel más alto de programación.

Cuando se desarrollan *Servlets*, aún los programadores tienen que ser cuidadosos y asegurar que no hay ninguna relación directa entre la presentación y el contenido. Por lo general puede hacer esto añadiendo a un tercero la presentación de HTML utilizando un programa como *htmlKona*. Pero aún este acercamiento que proporciona cierta flexibilidad a los cambios simples en pantalla, todavía no se protege de un cambio del formato de presentación más profundo.

Por ejemplo, si la presentación cambiara de HTML A DHTML, todavía hay que asegurar que los paquetes de programas sean compatibles con el nuevo formato. En un caso peor, si un paquete programado no está disponible, no puede continuar la presentación dentro del contenido dinámico. entonces ¿cuál es la solución? Una forma de solucionar dicho problema, es usar tanto JSP como *Servlet* para construir los sistemas.

4.2 Comparación de JSP con otras tecnologías CGI

De las tecnologías mencionadas al principio del capitulo, una de las más populares en el entorno Windows son las *ASP* (*Active Server Pages*), que consisten en una serie de etiquetas incluidas en páginas web, que usan Visual Basic como lenguaje, también esta PHP, muy vinculada con la base de datos *MySQL*.

Por su parte, PHP es un lenguaje cuyos programas se insertan también dentro de las páginas web, al igual que los *ASPs* y *JSPs*; es mucho más simple de usar, y el acceso a bases de datos desde él es muy simple. Es tremendamente popular en sitios de comercio electrónico con poco tráfico, por su facilidad de desarrollo y rapidez de implantación

Los CGI proporcionan la posibilidad de acceder a base de datos, intercambio de información a través de formularios HTML, gestión de accesos, utilidades de búsqueda, enviar por correo datos a una dirección específica de e-mail, etc., la actividad de estos programas puede resultar en documentos HTML visualizadas en el cliente. Los sckript CGI pueden elaborarse en cualquier lenguaje, aunque los más utilizados son *Perl*, C y Vb.

Los *Servlets* y los JSP tienen varias ventajas con respecto a las tecnologías comunes de CGI, pues son más eficientes, fáciles de usar, más poderosos y con mayor portabilidad.

Con los CGI tradicionales, un nuevo proceso es iniciado por cada petición http, si el programa CGI no tiene una operación rápida, esto hará que la ejecución sea más lenta, en cambio, con los *Servlets* y los JSP, la JVM se carga en el servidor, cada petición es manejada por un hilo o *thread lightweight* de Java y no un proceso *heavyweight*.

Otra diferencia es que con los CGI tradicionales, si hay N peticiones simultaneas al mismo *script*, entonces el código del programa CGI es cargado en memoria N veces, con los *Servlet*, existen N hilos, pero solo una copia de la clase generada por el *Servlet*, y además pueden mantener la conexión a la base de datos abierta.

Los *Servlets* permiten hacer varias cosas que son difíciles o imposibles con los CGI, por ejemplo, los *Servlets* pueden comunicarse directamente con el Servidor Web, y los programas CGI no pueden, esto simplifica las operaciones que necesitan buscar imágenes o información guardada en lugares ya predeterminados, también pueden compartir datos con otros, mantener información de una petición a otra, simplificando cosas como las sesiones o llevando el computo de operaciones anteriores.

La portabilidad de los *Servlets* es una característica heredada, pues son escritos en Java y siguen una estandarización API, así, un *Servlet* escrito en un *I-Planet Enterprise* Server, puede ejecutarse en Apache, Microsoft IIS o *WebStar*, además, se pueden implementar mediante programas con un bajo costo de licencia, e incluso en un servidor Apache en conjunto con *Tomcat*, utilizando una licencia del tipo *open source*.

4.2.1 Seguridad en los CGI

Cuando se desarrolla un CGI, se deben tener presentes los riesgos en la seguridad con el servidor, puesto que un *script* CGI es un programa que desde cualquier parte del mundo puede ejecutarse en el servidor, obviamente con la conexión disponible a Internet, por tal motivo, hay que buscar errores o agujeros cuando se escriben los *scripts*. Sobre todo, no se debe confiar en la entrada del usuario, en particular, verificar la entrada del usuario cuando se ejecutara un comando, pues de lo contrario se le permite a una persona mal intencionada acceder al servidor a través del agujero de seguridad.

Por ejemplo, si nuestro CGI permite al usuario realizar un *finger* (comando de Unix para obtener información sobre el usuario), el *script* en *Perl* podría tener una línea como la siguiente:

```
system ("finger $username");
```

esta línea de *perl* es similar a ejecutar en nuestra consola de *Unix*:

```
[isak@localhost isak]$ finger $username
```

En donde $username es la variable del usuario en cuestión, pero si un usuario malicioso escribe "james; rm -rf /" como nombre de usuario, tu programa correrá

```
[isak@localhost isak]$ finger james; rm -rf /
```

recuerde que el comando rm –rf eliminará un directorio y todo su contenido en un solo paso, por tal motivo hay que tener mucho cuidado al utilizarlo y siempre hay que verificar la entrada del usuario, por ejemplo:

```
$username!~ /[^\w.-]/   || die "Whoa!  Buen intento amigo." ;
```

ó utilizar una forma diferente para el comando system:

```
system("finger", $username) ;
```

Es fácil para un *hacker* enviar una variable al *script* con cualquier valor (aún caracteres no imprimibles). La seguridad no debería descansar en campos teniendo ciertos valores, ya sea que existan o no, si bien JSP contiene funciones de seguridad, es responsabilidad del programador implementarlas en el sistema y llevar el seguimiento tanto de los errores como de los posibles hoyos de seguridad.

4.2.2 Comparación con ASP

La tecnología JSP usa Java como lenguaje de *Script* mientras que ASP usa *VBScript* o *Jscript*. Java es un lenguaje mas potente y escalable que los lenguajes de *Script*. Las páginas JSP son compilados en *Servlets* por lo que actúan como una puerta a todos los servicios Java de Servidor y librerías Java para aplicaciones http. Java hace el trabajo del desarrollador más fácil y ayuda a proteger el sistema contra las "caídas" mientras que las aplicaciones ASP sobre sistemas NT son más susceptibles a sufrirlas, también ayuda en el manejo de la memoria protegiendo contra fallos de memoria y el duro trabajo de buscar los fallos de perdida de punteros de memoria que pueden hacer mas lento el funcionamiento de una aplicación.

El API JSP se beneficia de la extendida comunidad Java existente, por el contrario la tecnología ASP es específica de Microsoft que desarrolla sus procesos internamente y solo es para plataforma Windows; mientras JSP y ASP usan una combinación de *tags* y *scripts* para crear las paginas, la tecnología JSP permite a los desarrolladores crear nuevos *tags*. Así los desarrolladores pueden crear nuevos *tags* y no depender tanto de los *scripts*.

Los componentes JSP son reusables en distintas plataformas tanto sistemas operativos tipo UNIX, y Windows, además las aplicaciones que usan JSP tiene un mantenimiento más fácil que las que usan ASP.

- Los lenguajes de *Script* están bien para pequeñas aplicaciones, pero no encajan bien para aplicaciones grandes. Java es un lenguaje estructurado y es más fácil de construir y mantenimientos grandes como aplicaciones modulares.

- La tecnología JSP hace mayor énfasis en los componentes que en los Scripts, esto hace que sea más fácil revisar el contenido sin que afecte a la lógica o revisar la lógica sin cambiar el contenido.

- La arquitectura EJB encapsula la lógica de acceso a BD, seguridad, integridad transaccional y aislamiento de la aplicación.

- Debido a que la tecnología JSP es abierta y multiplataforma, los servidores web, plataformas y otros componentes pueden ser fácilmente actualizados o cambiados sin que afecte a las aplicaciones basadas en la tecnología JSP.

Las ventajas sobre utilizar la tecnología Java con respecto a la propietaria de Microsoft (ASP) son, como se ha podido ver, diversas e interesantes.

4.2.3 Conclusión

Se nota de manera muy sencilla, que los JSP tienen ventaja sobre los CGI y los lenguajes antes mencionados, todos son para desarrollar páginas Web dinámicas, sin embargo, en el fondo cada tecnología difiere una de otra. Java y JSP proveen librerías útiles para tener un nivel máximo de seguridad, las cuales pueden implementar funciones *hash*, infraestructura PKI, interacción con protocolo *https*, permisos de acceso, firma de applets, etc.

En general, podemos apuntar una ventaja de la programación en ASP y otras como PHP, pues resultan más fácil de aprender que JSP, por lo menos si no se tiene una experiencia previa en programación. Esto es debido a que Java es un lenguaje muy potente, pero un poco más complicado de usar porque es orientado a objetos y la manera de escribir los programas es más rígida, pero los beneficios son notorios y vale la pena dedicar un poco mas de esfuerzo aprendiendo Java.

4.3 Conceptos básicos de Java Server Page

Las páginas JSP son creadas para agregar funcionalidad a un servidor web mediante una interfaz de programación mezclando bloques de HTML estático y HTML dinámico generados con Java.

Nos enfocaremos en la versión 1.0 de JSP, no olvidemos que de la versión 0.92 a la versión 1.0 hubo cambios muy drásticos, los cuales mejoraron las páginas JSP, por tal razón, las páginas escritas en la versión 1.0 son incompatibles con la versión 0.92, lo importante es que de la versión 1.0 a 1.1 no hubo muchos cambios, las principales aportaciones son para definir etiquetas nuevas y utilizar la especificación 2.2 de los *Servlets*.

4.3.1 Estructura de una Página JSP

Una página JSP puede procesar formularios WEB, acceder a base de datos, y redireccionar a otras páginas, al emplear JSP en un sitio Web, es necesario un contenedor o entorno para soportar dicha tecnología, este entorno se encarga de convertir los JSP en *Servlets* y después ser compiladas, por tal motivo la primer petición de una página JSP es más lenta que las peticiones posteriores.

En el servidor, los archivos con extensión jsp identifican a las páginas creadas con JSP, en la implementación de JSP es fácil separar la parte dinámica de la estática, esto se hace encerrando el código Java de la JSP entre

<% *código Java* **%>**

```
<% String reqStr = request.getQueryString();
        out.println("Petición: " + reqStr);
%>
```

La sintaxis también se puede expresar en formato XML:

<jsp:*expresión***>**
Expresión de Java
</jsp:*expresión***>**

Existen tres tipos principales en las estructuras de JSP que se pueden incluir en la página: elementos de *Script*, directivas y acciones. Los elementos *Script* son básicamente código Java el cual permite controlar el flujo del programa y las acciones para el comportamiento de dicha página JSP. Para simplificar la codificación de JSP, existen variables predefinidas como ***request***.

Los *Script* permiten insertar código en el *Servlet*, este código será generado desde la página JSP, hay tres formas de hacerlo:

- Expresión: de la forma **<%=** *expresión* **%>**, los cual será evaluado e insertado en la salida del *Servlet*.
- *Scriptlets*: de la siguiente manera **<%** *código Java* **%>** el cual es insertado en el método ***_JspService*** *del Servlet*, llamado por ***service***.

- Declaraciones: insertadas de esta forma **<%!** *código Java* **%>** son agregadas en el cuerpo de la clase servlet, fuera de cualquier método existente.

En varios casos, las páginas JSP consisten en código HTML estático, el cuál se ve como cualquier formato HTML, e incluso, puede ser editado por algún editor HTML, como Dreamweaver, si se desea agregar un comentario en la página JSP, pero que no se vea

en el documento final, se utiliza **<%**-- *Un Comentario* --**%>**, los comentario de HTML son pasados de manera normal.

4.3.2 Expresiones en JSP

Una expresión en JSP se utiliza para insertar valores directamente en la salida, tiene la siguiente forma: **<%=** *expresión* **%>**.

La expresión es evaluada, convertida a String y después insertada en la página, la evaluación se realiza en tiempo de ejecución, es decir cuando la página es solicitada, por ejemplo, la siguiente expresión muestra la fecha y hora del momento en que la página es solicitada:

La fecha actual es: **<%=** *new java.util.Date ()* **%>**

Para simplificar la programación en los JSP, se pueden utilizar un número predefinido de variables y para utilizar con las expresiones, las más importantes son:

- HttpServletRequest
- HttpServletResponse
- HttpSession
- Out

Este último es el *PrintWriter* (una versión tipo buffer del *JspWriter*) utilizado para enviar salidas al cliente.

Un ejemplo de como se utilizan estas variables predefinidas es:

El Host es: **<%=** *request.getRemoteHost()* **%>**

La sintaxis para las expresiones en XML es la siguiente:

<jsp:*expresión***>**
Expresión de Java
</jsp:*expresión***>**

No olvidar que los elementos XML, al contrario que los del HTML, son sensibles a las mayúsculas; por eso hay que utilizar minúsculas.

<jsp:*setProperty name=* ***"autor"***
property= ***"Nombre"***
value= ***"Hugo" />***

Los atributos requieren que el valor este entre comillas o con apóstrofos, como se muestra en el siguiente ejemplo, de hecho, algunos atributos permiten utilizar expresiones, las cuales son evaluadas en tiempo de ejecución

```
<jsp:setProperty name= "user"
        property= "id"
        value='<%= "UserID" + Math.random() %>' />
```

La siguiente tabla muestra los atributos que permiten un valor en tiempo de ejecución como en el ejemplo anterior.

Nombre del elemento	Nombre(s) del atributo
jsp:setProperty	name value
jsp:include	page
jsp:forward	page
Jsp:param	value

Atributos en tiempo de ejecución

Para realizar tareas más complejas como insertar una simple expresión, los *Scriptlets* permiten agregar código de una manera más estructurada a una página JSP, tienen la siguiente sintaxis:

```
<% código Java %>
```

4.3.3 Scriptlets

Los *Scriptlets* tienen acceso a las misma variables predefinidas como *request, response, session, out, etc,* por ejemplo, para mostrar un texto en la página, se puede utilizar el siguiente código en donde se utiliza la variable *out*:

```
<%
String sDatos = request.getQueryString();
out.println("Los Datos son: " + sDatos);
%>
```

Incluso, en este ejemplo en particular, se puede lograr el mismo resultado utilizando la siguiente expresión JSP

```
Los Datos son: <%=request.getQueryString() %>
```

En general, los *Scriptlets* pueden desarrollar varias tareas que no pueden ser logradas solamente con expresiones, esas tareas incluyen asignación de cabeceras o *headers* para un *response* o códigos de estado, actualizaciones a una BD, código con ciclos o *loops*, condiciones y muchas estructuras de mayor complejidad.

Es común emplear en una misma página expresiones JSP y *Scriptlets*, los primero se pueden emplear para especificar elementos de HTML, como en el siguiente ejemplo, en

cual al pasarle el parámetro *?alinear=1* la alineación es a la derecha, *?alinear=2* es a la izquierda y *?alinear=3* es centrada:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Centrar Texto</title>
</head>

<body>
<%
int i= 0;
String sAlinear = request.getParameter("alinear");
if (sAlinear != null && sAlinear.length() > 0 )
i = Integer.parseInt(sAlinear);

switch (i) {
        case (1):
                sAlinear = "right";
                break;
        case (2):
                sAlinear = "left";
                break;
        case (3):
        default:
                sAlinear = "center";
  }
%>
<table width="200" border="1" align="center">
  <tr><!-- Aquí utilizamos una Expresión JSP -->
    <td><div align="<%= sAlinear %>">Un poco de Texto </div></td>
  </tr>
</table>
</body>
</html>
```

Otro uso de los *Scriptlets* es para utilizar condiciones dentro del código de la página JSP y así determinar que etiquetas de HTML mostrar. Incluso un *Scriptlet* no necesita tener en su cuerpo toda la estructura de una sentencia, por ejemplo un if, en el siguiente código se hace lo mismo que en el anterior, pero con if:

```
<table width="200" border="1" align="center">
  <tr><!-- Aquí utilizamos Expresiones JSP y Scriptlets -->
    <td>
    <% if (i == 1) { %>
          <div align="right">
```

```
<% } else if (i == 2){ %>
      <div align="left">
<% } else { %>
      <div align="center">
      <% } %>
Un poco de Texto
</div></td>
</tr>
</table>
```

Dentro de los *Scriptlets*, existe una manera que nos permite definir métodos para utilizarlos en el ámbito *main* de la clase del *Servlet*, y es a través de las declaraciones en JSP, estas declaraciones no generan salida alguna, por tal motivo son utilizadas con las expresiones JSP o con los *Scriptlets*, tienen la siguiente sintaxis:

<%! *código Java* **%>**

4.3.4 Variables Predefinidas

Para simplificar el código de los *Scriptlets* y las expresiones de JSP, existen ocho variables predefinidas, en algunas ocasiones llamadas como objetos implícitos, no son visibles dentro de las declaraciones mencionadas anteriormente. Estas variables son: *request, response, out, session, application, config, pageContext, y page*.

request

Esta variable esta asociada con *HttpServletRequest*, es decir, las peticiones y tiene acceso a los parámetros, a los tipos de petición como GET o POST, y a las *headers* del tipo HTTP como las cokies. Estrictamente ablando, si el protocolo que se pasa es diferente al HTTP, se puede tener una subclase de *ServletRequest*

response

Esta variable se refiere a *HttpServletResponse* asociada con la respuesta al cliente, dado que la salida de un *output stream* (ver *out*) normalmente es a través de un buffer, no se permite asignar un código de estado HTTP y un *response* en la cabeceras de la página JSP, pues no se permiten en los Servlets.

out

Esta variable se utiliza para enviar salida al cliente, y para darles utilidad a las variables *response*, esta es una versión en buffer del *PrintWriter* llamada *JspWriter*, se puede ajustar el tamaño del buffer a través de los atributos del mismo desde las directivas de página. Las expresiones JSP automáticamente se envían al stream de salida y no es necesario hacer referencia a la variable *out*, pero para los Scriptlets si es necesario y esta variable es muy común para proyectar la salida al cliente.

session

Esta variable es el objeto *HttpSession* asociado con las peticiones o *request*. Las sesiones son creadas automáticamente, se pueden deshabilitar mediante el atributo de

sesión de las directivas de la página, y si se hace referencia a la variable *session* generara un error al momento de convertir la página JSP a un Servlet.

application

Esta variable esta dentro del ámbito del *Servlet* generado por *getServletConfig()*. Las páginas JSP y los Servlets pueden guardar datos de manera persistente en el objeto *ServletContext*, el cual tiene los métodos *setAttribute* y *getAttribute* para poder almacenar los datos de manera arbitraria. La diferencia de guardar datos en variables locales y guardarlos en el *ServletContext* es que este último esta compartido para todos los Servlets.

config

Esta variable es el objeto *ServletConfig* para la página en cuestión.

pageContext

Con JSP se introduce una nueva clase llamada *pageContext* para proporcionar un punto de acceso a varios de los atributos de la página y proporcionar una manera de guardar datos compartidos.

page

Esta variable es un sinónimo para *this* y no es muy útil dentro del lenguaje de programación Java, fue creada para tener una referencia en caso de que el lenguaje de Script fuera otro a parte de Java.

4.4 Interacción de JSP con Base de Datos

JSP al igual que Java, tiene la capacidad de interactuar con base de datos, ya sea a través de un *driver* en especifico o mediante un DNS en el entorno de Windows, esto se realiza a través de un API de Java conocido como JDBC (*Java Database Connectivity*). El JDBC consiste en un conjunto de clases e interfaces escritas en Java que proporcionan un estándar para el desarrollo de aplicaciones con acceso a BD.

Mediante la versión 2.0 del JDBC se puede tener acceso a cualquier BD, desde BD relacionales hasta archivos planos, el JDBC se considera como la interface que proporciona un acceso universal para el lenguaje Java, esta versión esta disponible en dos paquetes, *java.sql* y *javax.sql*, el primero se incluye en la versión estándar del SDK (*Java 2 SDK, Standard Edition*) y el segundo paquete se incluye con la versión empresarial (*Java 2 SDK, Enterprise Edition*), la versión empresarial también se puede descargar por separado. Incluso ambas versiones tienen compatibilidad con la versión 1.0, obviamente la versión más reciente tiene mayores características.

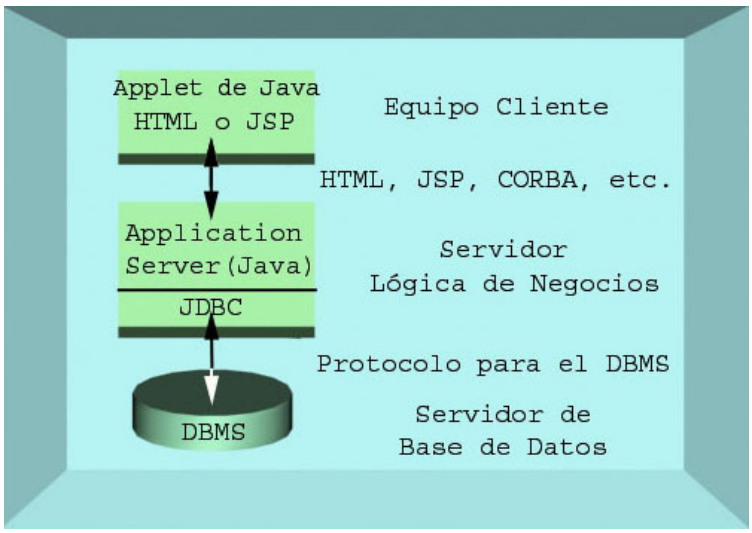
Al utilizar el JDBC, se pueden implementar sentencias SQL a BD relacionales, pero la versión 2.0 va más allá del SQL, pues implementa la interacción con otro tipo de datos, como los archivos con datos y tabuladores, además *"Una aplicación de BD Java no se preocupa por el motor de la BD, no importa las veces que este motor cambie."* [1].

[1] Programación de Base de Datos con JDBC y Java, George Reese, Anaya Multimedia, Madrid 2001, p. 31

4.4.1 El Modelo de las Tres Capas

En la actualidad es muy común utilizar el modelo de tres capas para la conexión con BD, las capas se dividen en el GUI, el procesamiento de los datos y el Servidor de BD. Al implementar este modelo se puede simplificar el desarrollo de aplicaciones y puede tener ventajas sobre el desempeño del sistema.

Al utilizar el JDBC, es fácil implementar este modelo, la interfaz gráfica se llevaría a cabo mediante applet's o con una página JSP, el procesamiento de datos se implementa con clases de Java y el acceso a la BD mediante el JDBC, el sistema que gestiona la BD puede ser diferente, desde una BD en Access hasta una muy compleja como Oracle o SQL Server, en nuestro caso será MySQL.



Capas en un programa

4.4.2 Conexión a la BD

La interacción típica con una BD consta de cuatro pasos básico:

- Crear la conexión a la BD
- Ejecutar consultas para obtener datos de la BD
- Procesar los resultados obtenidos
- Cerrar la conexión

El siguiente ejemplo muestra éstos cuatro pasos para consultar datos a una BD creada con Access, es un ejemplo sencillo, pero mostrará como utilizar el JDBC. Particularmente en Windows, el JDBC que conecta con un ODBC se conoce como puente JDBC-ODBC, la BD de este ejemplo es una agenda, con Nombres, Direcciones, Teléfonos y Correos electrónicos.

La forma tradicional para establecer una conexión es a través del método *DriverManager.getConnection*, a este método se le pasan tres parámetros del tipo String, uno con el URL referenciado a la BD, otro con el usuario y el tercero con el password.

```
String url = "jdbc:odbc:Nombre_ODBC";
Connection con = DriverManager.getConnection(url, "user", "password");
```

Dentro del ejemplo, existe una clase encargada de crear la conexión y obtener los datos, esta clase se llama conexion.java, y tiene el método creaConexion, el cual solo crea la conexión, getNombres para obtener los nombres y cierraConexion, el cual cierra la conexión.

```
public boolean creaConexion (){  //Este método solo establece la conexión, si
ocurre un error regresa false
String rutaBD= "c:/Nombres.mdb";
String strcon= "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ="
+ rutaBD;
boolean blnSet; //Bandera utilizada para determinar si se estableció bien la
conexion

blnSet = true;
try { Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
   canal=DriverManager.getConnection(strcon);
   instruccion = canal.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
   instruccion = canal.createStatement();

   } catch(java.lang.ClassNotFoundException e){
    blnSet = false;          System.err.println ("Error 1"+e);
         } catch(SQLException e) {
    blnSet = false;          System.err.println ("Error 2 "+e);
   }
return blnSet; //True conexion correcta      False = Incorrecta
}//Fin de creaConexion
```

Una vez que la conexión se estableció, se deben enviar sentencias SQL al JDBC, el cual permite una gran flexibilidad para las consultas a la BD, obviamente es responsabilidad del programador asegurarse de que el DBMS puede manipular la sentencia enviada desde el JDBC, de lo contrario se generará una excepción.

4.4.3 Sentencias con el JDBC

El JDBC tiene tres maneras de enviar sentencias SQL:

1. Sentencias (*createStatement*) creadas por el método *Connection.createStatement*, el cual recibe enunciados SQL sin parámetros, no son muy eficientes, por tal motivo no deben de ser de uso frecuente.

2. *PreparedStatement* creados por el método *Connection.prepareStatement*, un objeto utilizado para enviar sentencias precompiladas de SQL. Pueden tener uno o más parámetros como argumentos de entrada, estos valores se asignan mediante un grupo de métodos, los cuales son enviados a la BD cuando se ejecuta la sentencia. Un *PreparedStatement* tiene la ventaja de ser más eficiente que un *Statement*, pues ha sido precompilado y guardado para su uso.

3. *CallableStatement*, creados por el método *Connection.prepareCall*, utilizadas para ejecutar *stored procedures*.

Como se muestra en el siguiente fragmento de código, la implementación de la sentencia SQL es por el primer método.

```
public String getNombres (){
ResultSet tabla= null;
String datName = "";
try {
 tabla = instruccion.executeQuery("SELECT * FROM user");
 while(tabla.next()) {
        datName = datName + tabla.getString(2) + ",~, ";
  }
  tabla.close();
 } //fin try no usar ; al final de dos o mas catchs
 catch(SQLException e) {
  System.err.println ("    Clase: sqlAction|n    Metodo: getNombres: ");
  System.err.println ("Error al obtener los nombres de la BD"+e);
 };
 return datName;
}
```

4.4.4 Transacciones

Una transacción consiste en una o más sentencias que han sido ejecutadas, completadas sujetas a un *commit* o un *roll back.* Cuando el método *commit* o *roll back* es llamado, la transacción actual se termina y comienza otra. El primer método hará permanente cualquier cambio hecho por alguna sentencia SQL en la BD y libera recurso utilizado por la transacción, el método *rooll back* descartará dichos cambios.

Generalmente un objeto del tipo *Connection* esta por default en modo *auto-commit*, de esta manera cuando una sentencia SQL es completada el método *commit* es llamado automáticamente. En este caso como cada sentencia es terminada con un *coomit*, la transacción consiste en una sola sentencia. Si el modo *auto-commit* es deshabilitado, la transacción terminará hasta que se llame de manera explicita el método *commit* o *roll back*, en este caso todas las sentencias antes de la llamada al *commit* o al *roll back* se tratarán como un grupo.

En algunas ocasiones se desea que un cambio tenga efecto en la BD si y solo si otro cambio se realizo con éxito, esto puede ser logrado deshabilitando el *auto-commit* y agrupando ambas sentencias en una transacción. Si las dos sentencias se llevan a cabo de manera correcta, entonces el método *commit* es llamado, y de esta manera los dos cambios se harán de manera permanente. Si uno falla o ambos fallan, entonces el método *roll back* es llamado, restaurando los valores anteriores antes de que se ejecutaran las dos sentencias.

No olvidemos la importancia del modelo de las tres capas, *"cuyo propósito es dividir un componente o subsitema en tres partes lógicas –modelo, vista y controlador- facilitando la modificación o personalización de cada parte"*[2].

[2] Patrones de diseño aplicados a Java, Stephen Stelting, Olav Maassen, Prentice Hall, Madrid 2003, p 210

Capítulo 5: Implementación del Sistema

En este último capítulo, nos enfocaremos en el desarrollo del Sistema, tanto en el Servidor como en el Cliente y el acceso a la Base de Datos para guardar la información.

Ciclo de desarrollo de los Sistemas

El ciclo de desarrollo de los sistemas es un enfoque de análisis y diseño, que postula que el desarrollo de los sistemas mejora cuando existe un ciclo específico de las actividades necesarias para la construcción de los sistemas.

Lo primero con lo que debemos estar familiarizados es con el medio ambiente en el cual trabajaremos, esto esta centrado en el Web y la programación en Java, abordados en los primeros capítulos.

Identificación del problema.

En esta primera etapa se involucra en la identificación de los problemas, de las oportunidades y de los objetivos, parte de este trabajo se realizó en la introducción, en resumen, la problemática es generar un programa que nos ayude al esparcimiento de los conocimientos que algunas personalidades tienen y poder hacer llegar este conocimiento a varias personas a la vez.

Nuestro objetivo se centra en desarrollar una aplicación que interactué con una Base de Datos para que se lleve un registro de dicha información y ayudándonos en el Web, un medio accesible para muchas personas en tiempo real.

Requerimientos del Sistema.

Los requerimientos del Sistema nos ayudan a identificar las necesidades de quienes usarán el Sistema, en nuestro caso partimos de un supuesto, en donde los usuarios tienen la necesidad de compartir información entre sí y también de poder realizar preguntas muy especificas de algún tema. Por tales motivos, se necesita un programa para interactuar entre los usuarios y alguien que pueda contestar dichas preguntas, pero se llevará un registro de esta interacción para futuras consultas.

El tiempo destinado a contestar estas preguntas es en una hora en especifico y la duración de esta sesión también esta limitada.

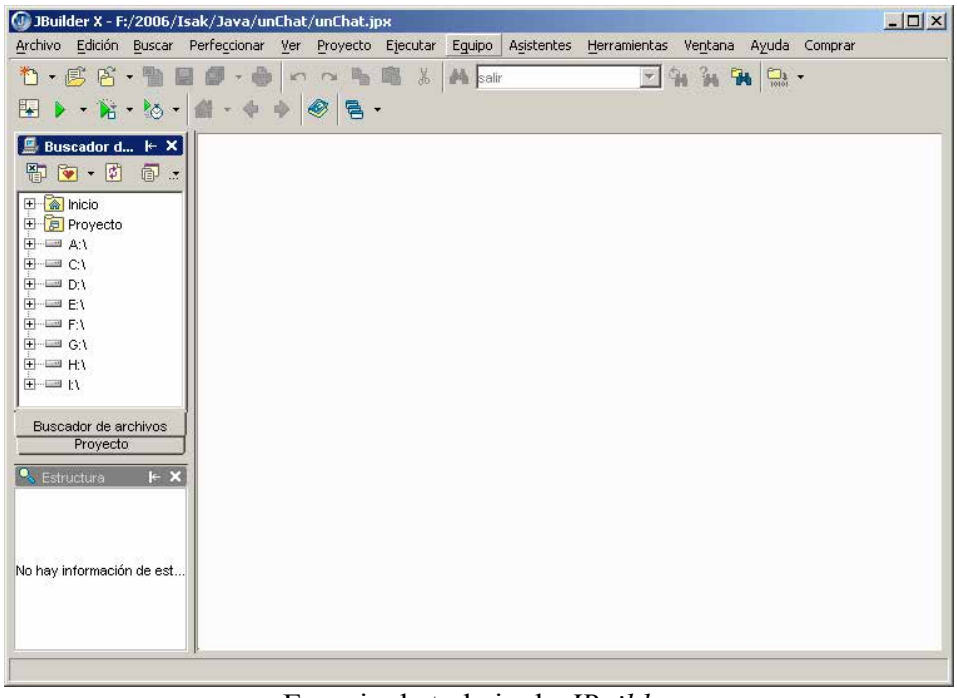
5.1 La comunicación en Internet y el IRC

La *RED* es una nueva herramienta que hace algunos años se involucro a nivel masivo, conociendo las ventajas que se generan a partir de esta tecnología, podemos obtener un beneficio y que este beneficio sea para el bien de todos, un claro ejemplo en este rubro son los contenidos de los portales, en los cuales se dan servicios muy variados, que van desde correo electrónico, agenda, horóscopos, etc.

En nuestro caso, la implementación de este Servidor, es para proporcionar información lúdica sobre temas específicos y que estas charlas se puedan consultar posteriormente.

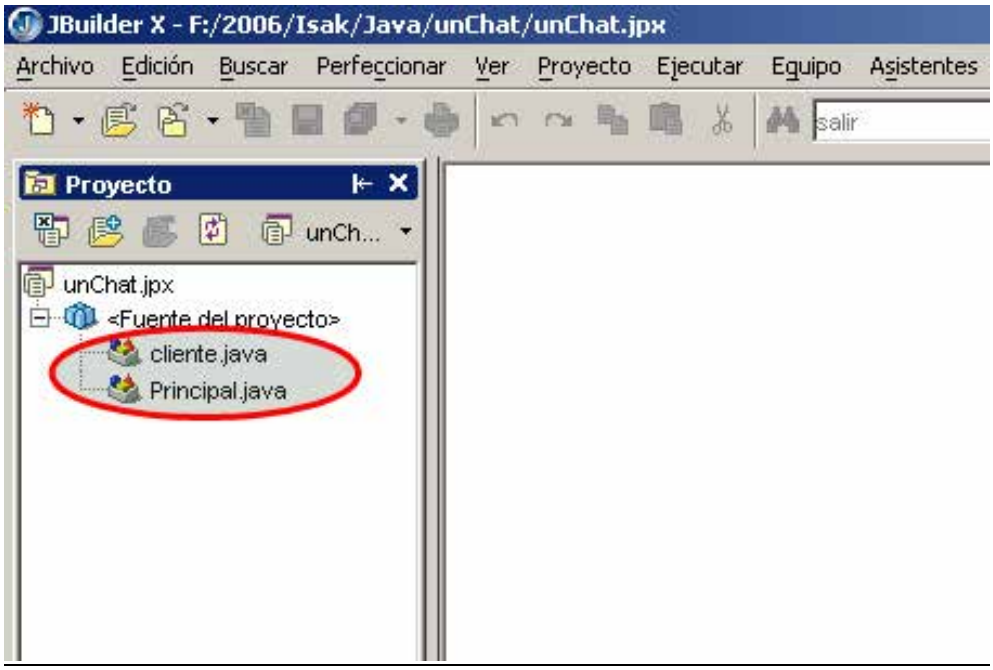
5.1.1 La clase Principal y la clase cliente

Para el desarrollo del proyecto, se utilizará la herramienta *JBuilder* de *Borland*, el espacio de trabajo se muestra a continuación:



Espacio de trabajo de *JBuilder*

Primero construiremos el servidor, para lo cual iniciaremos un nuevo proyecto y tendremos dos archivos, uno de los cuales será la clase principal y el otro implementará la uncionalidad del Servidor



La clase cliente y la clase Principal

La clase principal tendrá la función *main* y manejara los *Threads*, implementará a la clase cliente para tener el control de los usuarios que entran al servidor y tendrá la función de iniciar el Servidor.

La clase cliente tendrá un vector con todos los clientes que se conecten, un método para agregar cliente, uno para enviar mensajes y un método para el manejo del sistema, este último se encargará de llevar el control sobre los mensaje enviados por los clientes.

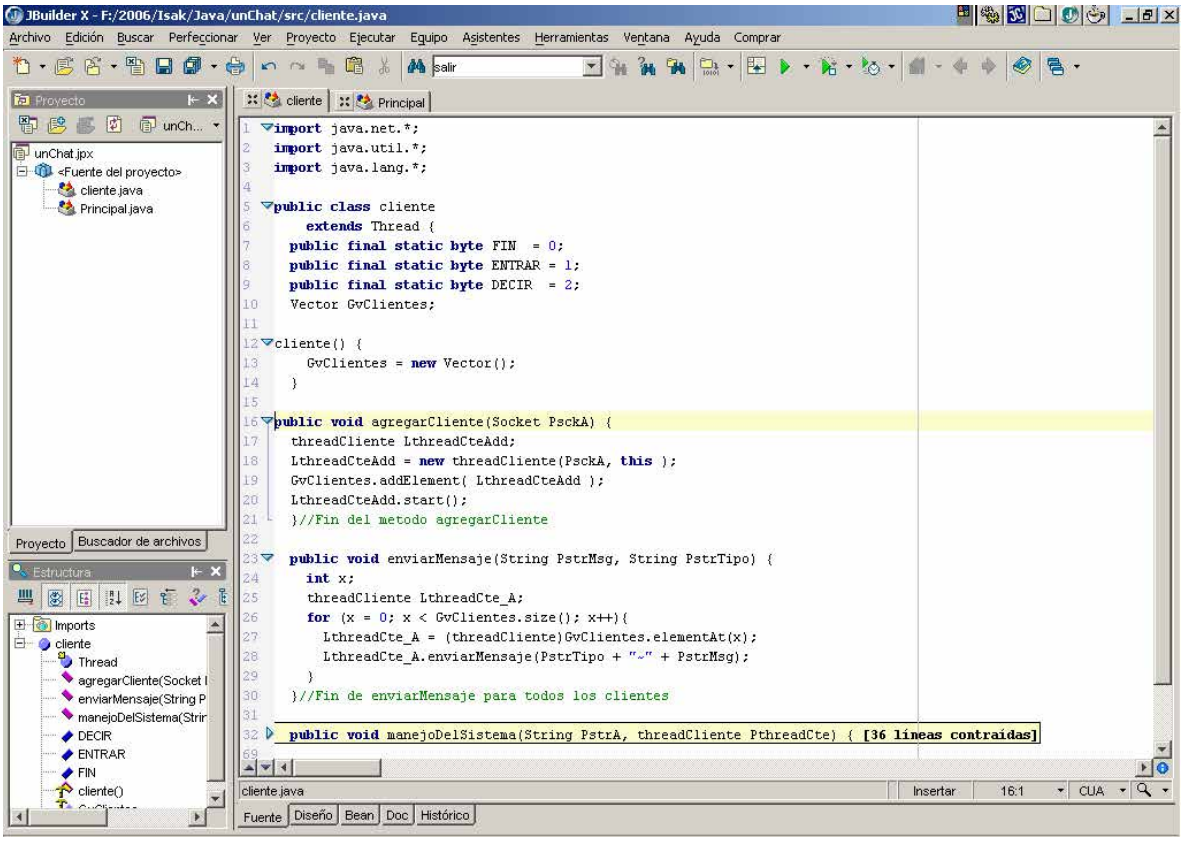
```
import java.net.*;
import java.util.*;
import java.lang.*;

public class cliente
    extends Thread {
  public final static byte FIN  = 0;
  public final static byte ENTRAR = 1;
  public final static byte DECIR  = 2;

  Vector GvClientes;

cliente() {
    GvClientes = new Vector();
  }

public void agregarCliente(Socket PsckA) {
  threadCliente LthreadCteAdd;
  LthreadCteAdd = new threadCliente(PsckA, this );
  GvClientes.addElement( LthreadCteAdd );
  LthreadCteAdd.start();
  }//Fin del metodo agregarCliente
```

```
  public void enviarMensaje(String PstrMsg, String PstrTipo) {
    int x;    threadCliente LthreadCte_A;
    for (x = 0; x < GvClientes.size(); x++){
      LthreadCte_A = (threadCliente)GvClientes.elementAt(x);
      LthreadCte_A.enviarMensaje(PstrTipo + "~" + PstrMsg);
    }
  }//Fin de enviarMensaje para todos los clientes

  public void manejoDelSistema(String PstrA, threadCliente PthreadCte) {
    String LstrComando, LstrVal;
    StringTokenizer Lst_Token;
    byte LbyteCmd = -1;
    System.out.println(PstrA + " de " + PthreadCte.getNombre());

    Lst_Token = new StringTokenizer(PstrA, "~");
    if (Lst_Token == null) {
      return;
    }
    LstrComando = Lst_Token.nextToken();
    LbyteCmd = Byte.parseByte(LstrComando);
    LstrVal = Lst_Token.nextToken();
    switch (LbyteCmd) {
      case FIN:
        System.out.print("     El usuario " + PthreadCte.getName() + "
                         abandono el Chat");
        enviarMensaje(PthreadCte.getNombre() + " se salio del chat.",
            LstrComando);
        PthreadCte.stop();
        break;
      case ENTRAR:
        PthreadCte.setNombre(LstrVal);
        enviarMensaje(PthreadCte.getNombre() + "~" +
            PthreadCte.getNombre() + " entro al chat.", LstrComando);
        break;
      case DECIR:
        enviarMensaje(PthreadCte.getNombre() + ": " + LstrVal,
                      LstrComando);
        break;
      default:
    }
  } //Fin de manejoDelSistema

}
```

La clase principal, creará una instancia de la clase cliente, de esta manera podrá agregar clientes de acuerdo a como se realicen las peticiones de los mismos, en su método run tendrá un ciclo con el cual estará a la espera de los clientes.

```
/*Esta clase tiene la funcion de iniciar el Servidor*/

import java.net.*;
import java.lang.*;

public class Principal
    extends Thread {
```

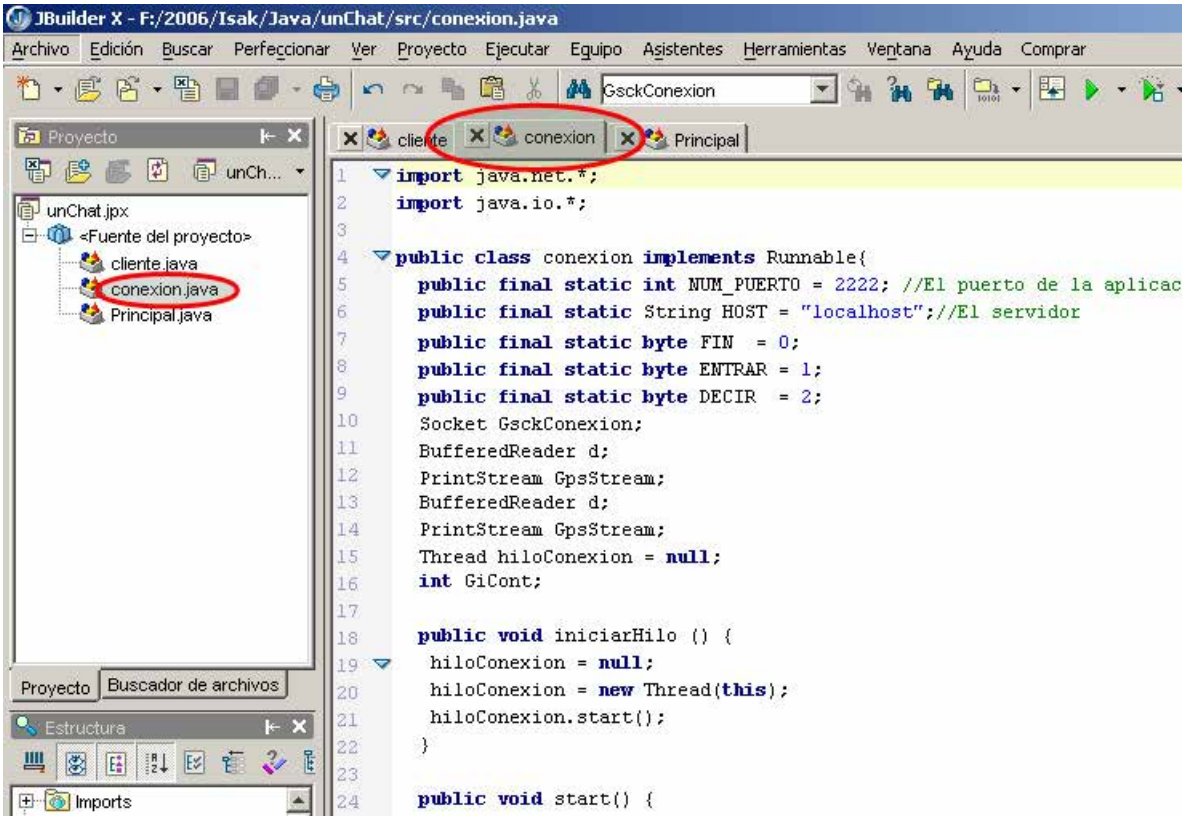
```
  ServerSocket servSock = null;
  cliente cteUsuarios;

  public static void main(String args[]) {
  new Principal().start(); //Crear la instancia e iniciar la clase
}

//Es el constructor de la clase
  Principal() {
    try {
      servSock = new ServerSocket(2222);
    }
    catch (Exception e) {
      System.out.println("No se puede establecer la conexion.");
      System.exit(1);
    }
    System.out.println("El Servidor se ha iniciado");
    cteUsuarios = new cliente();
    cteUsuarios.start();
  }//Fin del constructor

  public void run() {
    while (servSock != null) {
      Socket tempSock;
      try {
        tempSock = servSock.accept();
        cteUsuarios.agregarCliente(tempSock);
      }
      catch (Exception e) {
        System.out.println("La peticion fallo, no se pudo conectar el
                            cliente.");
        System.exit(1);
      }
    }
  }//Fin del metodo Run
}
```



Ejemplo del código en *JBuilder*

5.2 Conexiones en Java

Para el desarrollo de este proyecto, se necesita un Servidor y un Cliente, tienen comunicación entre si y están programados en Java, utilizan los *Sockets* para crear esta comunicación.

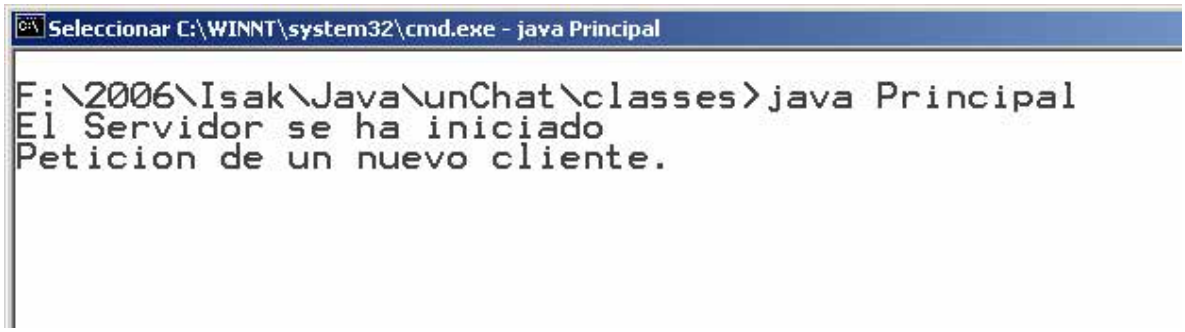


La clase conexion

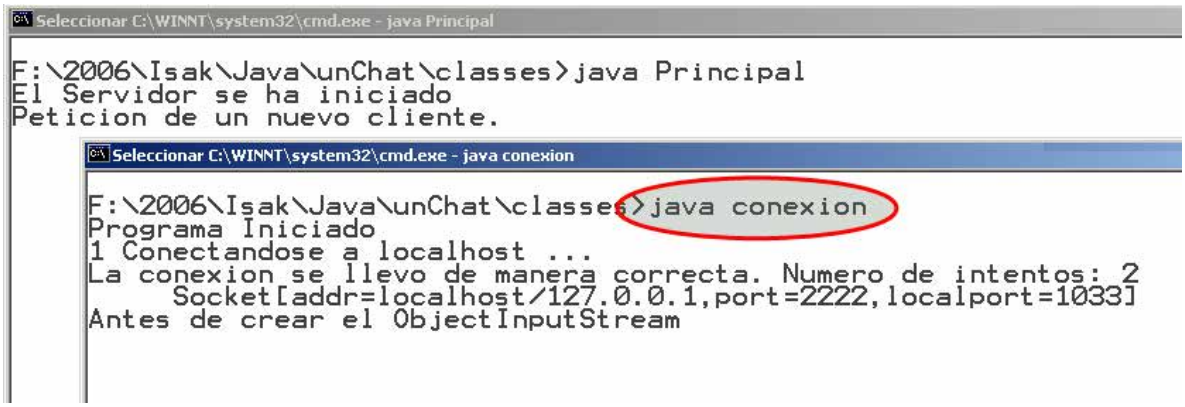
Dentro del proyecto, existe la clase *conexion*, esta clase es la encargada de establecer la comunicación del Cliente al Servidor, tiene un S*ocket* para conectarse al servidor (*GsckConexion*), un *PrintStream* utilizado para enviar la información al servidor (*GpsStream*), un *BufferedReader* necesario para leer los datos que envié el servidor (*GbrLineaEntrada*) y un Thread (*hiloConexion*) empleado para escuchar todo lo que llegue del servidor en conjunto con el BufferedReader.

En el método *start* de esta clase se crea el hilo, dentro del método *run* se abre la conexión, después se ejecuta el método *iniciarHilo* para iniciar el hilo y poder entablar la comunicación con el método *platicar*.

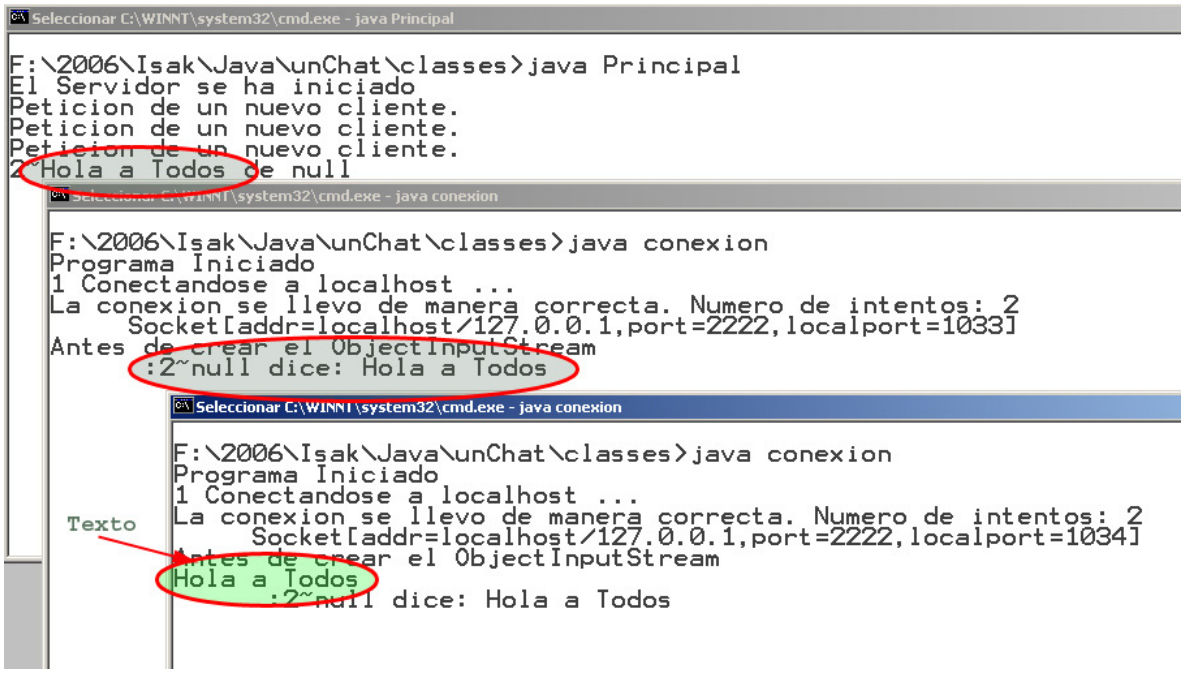
A continuación se muestran las pantallas de cómo se ejecuta el Servidor y la conexión al Servidor.



Primero iniciamos el Servidor.



Después iniciamos una de las conexiones.



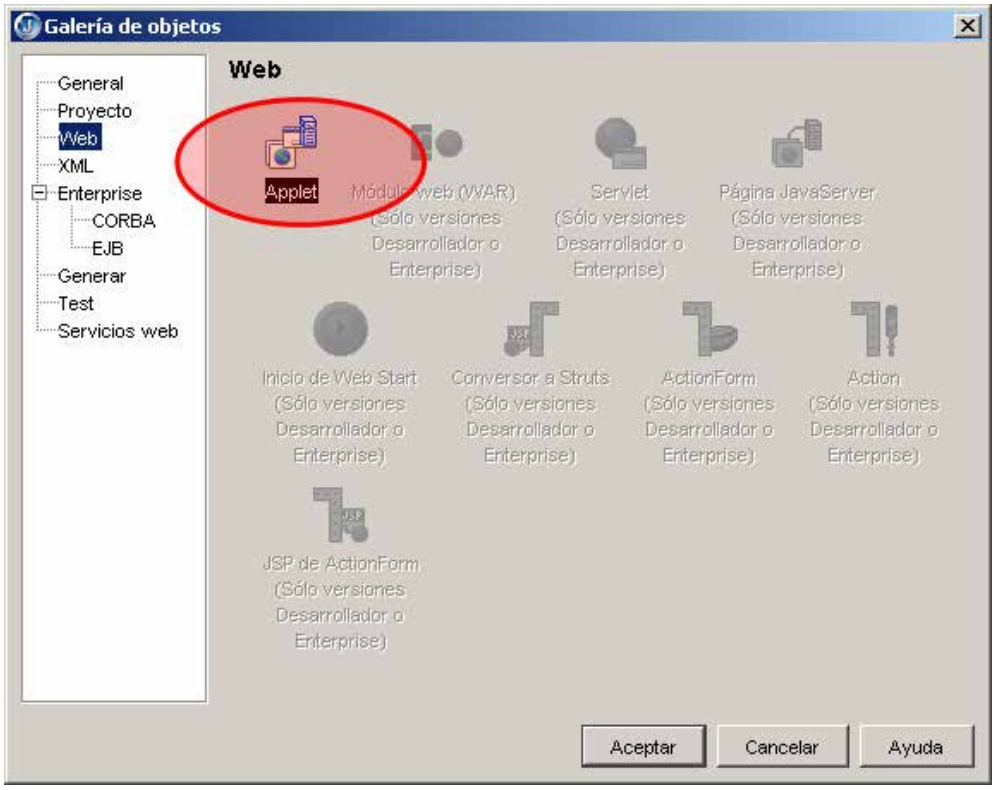
Ejemplo de cómo funciona el programa

Iniciamos otra conexión y enviamos un mensaje, nuestro mensaje fue **Hola a Todos** Se puede ver que tanto en el Servidor como en el primer cliente el mensaje se visualiza.

5.3 La Interfaz Gráfica

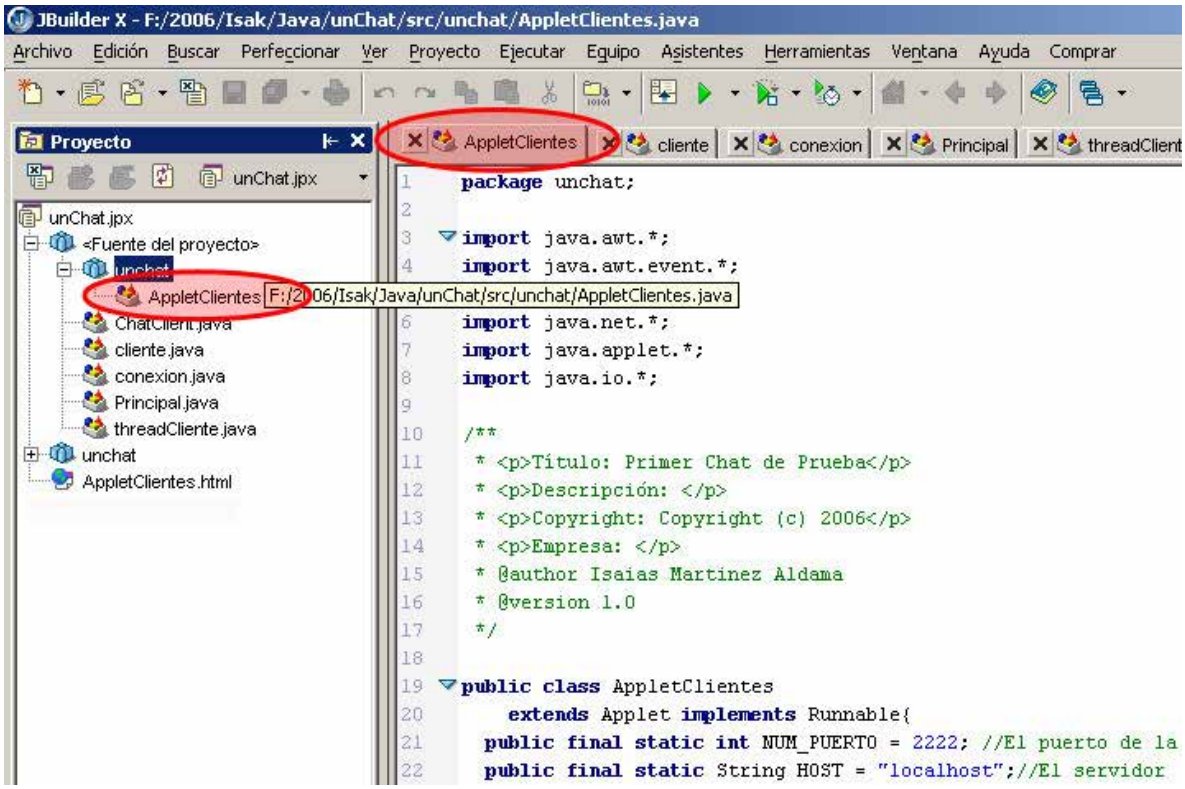
La interfaz gráfica se concentrará en el Cliente, puesto que el Servidor solo escuchará a los clientes y conducirá sus entradas a la BD. Para la creación de una conexión al Servidor en modo gráfico, emplearemos un *applet*, este *applet* será el que utilizará el cliente desde el sitio Web.

Desde el *Jbuilder*, seleccionamos el menú Archivo > Nuevo y en la pantalla que aparece, seleccionamos la opción del *applet*, como se muestra en la siguiente figura:



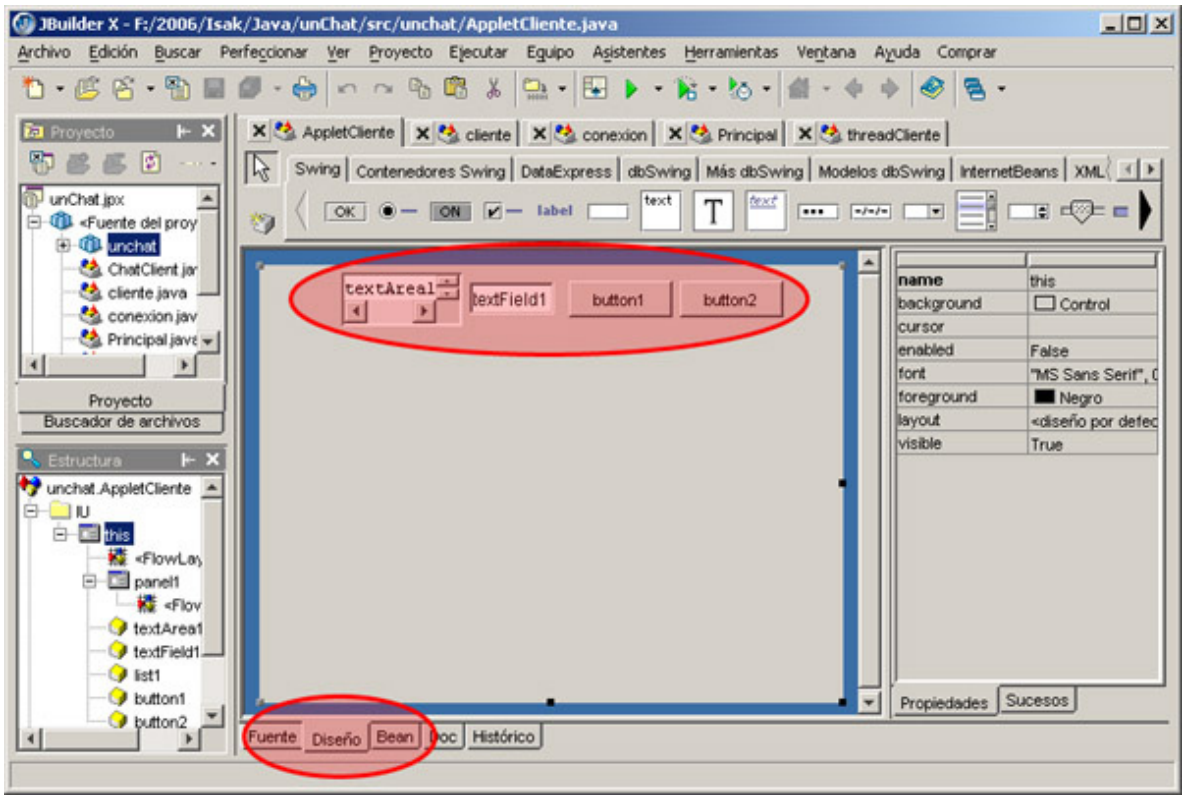
Selección de un objeto applet en *JBuilder*

El nombre de la clase será *AppletClientes*, los parámetros y la configuración del *applet* quedarán por *default*. En la siguiente imagen se muestra como queda nuestra área de trabajo con el *applet* agregado.



La clase AppletClientes

A nuestro *applet*, desde la pestaña diseño ubicada en la parte inferior del IDE, le agregaremos un área de texto (*TextArea*), una caja de texto (*TextField*), una lista (*List*) y dos botones (*Button*)

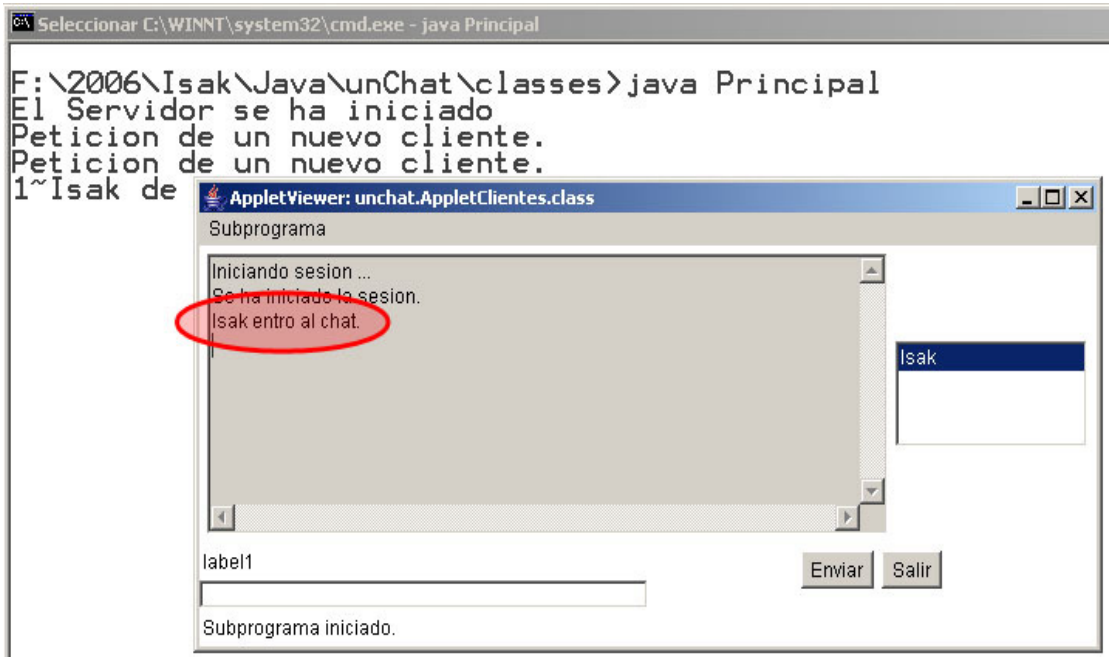


Diseño de la *GUI* en *JBuilder*

Una vez hecho lo anterior, agregaremos el código para manejar la entrada del usuario y poder entablar la conexión con el Servidor. Este *applet* tendrá un *Socket*, un *Stream* de entrada, uno de salida, un hilo para la Conexión, de hecho será muy similar a la clase *conexion* que hicimos anteriormente.

En el momento en que el usuario presione el botón para iniciar la sesión, se crea la conexión al ejecutar el método *run*, se inicia el hilo para escuchar lo que se envié desde el servidor y se inicia la plática.

El método *run* del *applet*, se encarga de llamar al método *iniciarSesion* y una vez realizada la conexión con el servidor, dentro de *run* se ejecutará un ciclo *while* para manejar la información del *GdisStream*, este ciclo no termina hasta que el *Socket*, el *Stream* de entrada o el *Thread* sean nulos.



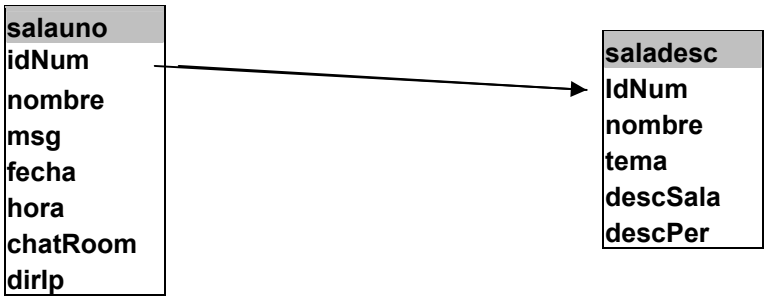
Ejemplo del programa en forma gráfica

5.4 Conexión a la Base de Datos

Para poder lograr una interconexión concurrente entre los usuarios, es necesario implementar los *Threads* o *hilos* para cada conexión que se generé. En este ejemplo crearemos el acceso a una BD mediante el *JDBC* de *MySQL*, utilizando los *Threads*. Los fundamentos utilizados en esta sección se describen en el capitulo 4, exactamente en *4.4 Interacción de JSP con Base de Datos.*

5.4.1 El modelo de datos relacional

El modelo de datos relacional organiza y presenta los datos en forma de tablas o relaciones, el término relación proviene de la matemática y representa una tabla de dos dimensiones, consiste en filas y columnas. En nuestro caso, tenemos las siguientes tablas:



Relación de las tablas en la Base de Datos

Para las dos tablas, la llave principal es idNum, la llave externa (*foreign key*) utilizada para relacionar las dos tablas, es precisamente idNum, de esta manera, en la tabla **salauno** se guardan las tuplas de cada pregunta realizada, y sé vicula con la tabla **saladesc** de acuerdo a la platica a la cual pertenece.

En la tabla **salauno**, existen siete atributos, y en la tabla **saladesc**, hay cinco atributos. Para la tabla **salauno** los atributos msg, fecha, hora, y dirIp permiten valores nulos, mientras que en la tabla **saladesc**, solo descSala y descPer permiten valores nulos.

salauno						
idNum	**nombre**	**msg**	**fecha**	**hora**	**chatRoom**	**dirIp**
1	Isak	Mensaje de Isak	2006-12-05	06:45:28	2	127.0.0.1
2	Carmen	Mensaje de Carmen	2006-12-05	06:46:31	2	127.0.0.1

saladesc				
idNum	**nombre**	**tema**	**descSala**	**descPer**
1	Ing. Martinez	Primer tema del sitio	Descripción de la sala	Descripción de la persona

5.4.2 Consulta a la Base de Datos

En principio de cuentas, tenemos una clase *sqlEje* la cual se conecta a la BD mediante el método *getTodosDatos*, ahora será invocado desde el método *run* cada determinado tiempo. Mediante una variable llamada *ESPERA*, se hará que el *Thread* se duerma durante un lapso.

La clase *sqlEje* implementará el *Thread*, y estará creada desde *main*. Lo primero que haremos en esta clase es implementarla desde *Runnable* y agregarle los métodos que deben contener todas las clases que sean *Threads*. La clase se ve así:

```
public class sqlEje implements Runnable{
  final int ESPERA = 1500;
  private Thread hiloSql;
```

Esta clase tiene ahora los métodos *start, run, stop* e *iniciarHilo* y dos variables a nivel de la clase, la variable `ESPERA` es utilizada para pausar el *thread* durante 1500 milisegundos, `hiloSql` será el *thread* y se creará una instancia de el en el método *iniciarHilo*.

```
  public void run() {
    while (hiloSql != null) {
      getTodosDatos ();
    try{ hiloSql.sleep(ESPERA);
       } catch(Exception e){};
    }//Fin del while
  }//Fin de run
```

Dentro del método *run* obtenemos los datos de la BD y hacemos una pausa, es importante implementar este método, pues es parte medular de todo clase que funcione como un *thread*.

Desde `main`, no llamaremos el método `getTodosDatos`, se hará referencia a `iniciarHilo`, el cual hará la tarea de invocar a `getTodosDatos`.

Las clases quedan de la siguiente manera:

```
class prin {

  public static void main( String args[] ) {
  sqlEje sqlA = new sqlEje ();
  sqlA.iniciarHilo();//Iniciar los elementos del Thread
  }//fin de main
}


import java.sql.*;

public class sqlEje implements Runnable{
  final int ESPERA = 1500; //Se utiliza para pausar el Thread
  private Thread hiloSql; //El hilo a ejecutar
```

```
  public void iniciarHilo () {//Este es el método que se llama desde
   hiloSql = null;            //main en la clase principal
   hiloSql = new Thread(this);
   hiloSql.start();
  }

  public void start() {
    if (hiloSql == null) //Solo si no se ha creado el hilo
      hiloSql = new Thread(this);
    hiloSql.run();//Lanza el método que estará ejecutando el Thread
  }//Fin de Start

  public void run() {
    while (hiloSql != null) {
      getTodosDatos ();//Invoca este metodo para mostrar los datos
    try{ hiloSql.sleep(ESPERA);
       } catch(Exception e){};
    }//Fin del while
  }//Fin de run

  public void stop () {
    hiloSql = null; //El hilo se termina
  }

//No cambia esta función, solo se llama de una parte diferente
public void getTodosDatos () {
Connection CONEXION = null;
String LsQuery= "";
ResultSet tabla= null;

Statement stmt;
try {
System.out.println (LsQuery);
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  CONEXION =
DriverManager.getConnection("jdbc:mysql://localhost/dbChat","usuarioUno",
"pass01");

  stmt = CONEXION.createStatement ();

  LsQuery = "select nombre, msg, fecha, chatRoom, dirIp from salaUno";
  tabla = stmt.executeQuery (LsQuery); //Ejecuta un query y devuelve
Bool
  while(tabla.next()) {
    System.out.println ("  "+ tabla.getString(1));
    System.out.println ("  "+ tabla.getString(2));
    System.out.println ("  "+ tabla.getDate(3));
    System.out.println ("  "+ tabla.getString(4));
    System.out.println ("  "+ tabla.getString(5));
  }

  } catch (java.lang.ClassNotFoundException e) {
    System.err.println ("  Error, no se encontro: "
                              +e.getMessage());
  } catch (SQLException e) {
      System.err.println ("  Error, Excepcion SQL: " +e.getMessage());
```

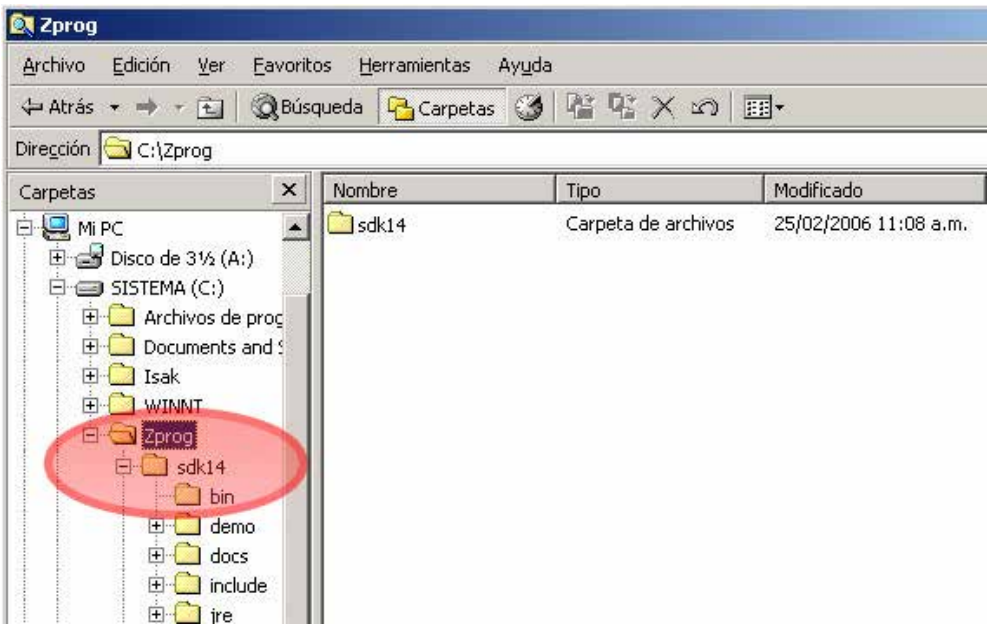
```
  } catch (Exception e) {
       System.err.println("  Error Interno getTodosDatos: "+e);
  }

try {
  CONEXION.close();
  } catch (SQLException e) {
  System.err.println ("Error. No se puede cerrar la Base: "
                              +e.getMessage());
    }
 } //Fin de getTodosDatos

}
```

5.5 Desarrollo e implementación del Sistema

En los antecedentes del proyecto, se especifico el software necesario para el desarrollo de esta aplicación, haciendo un recuento, el software utilizado será: SDK ver 1.4.0, servidor Web Apache ver 1.3.27, MySQL ver 3.23, y Tomcat 4.0.

5.5.1 Instalación del SDK de Java

La instalación del Java SDK 1.4.0 no tiene mayor complicación, pues se lleva a cabo mediante un asistente y no se requiere una configuración minuciosa; la ruta en donde lo instalaremos es: C:\Zprog\sdk14

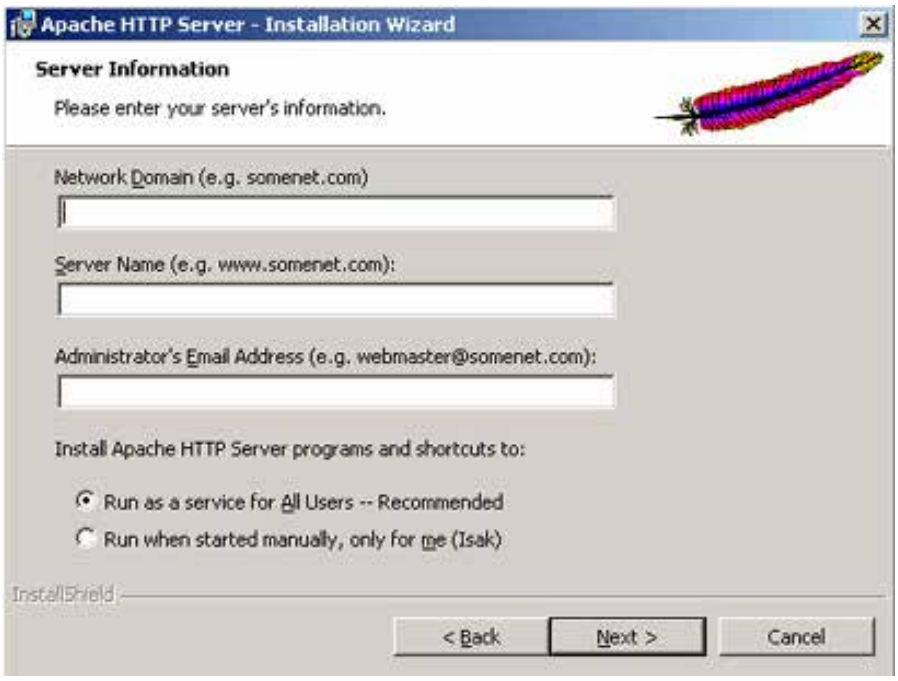


Directorio de instalación del SDK

5.5.2 Instalación del Servidor Web Apache

El servidor de páginas Web que utilizaremos es el Apache, la versión 1.3.27, el archivo para instalar de esta versión se puede encontrar en el sitio de Apache http://www.apache.org, es importante mencionar que este servidor esta desarrollado por contribuciones voluntarias de varias personas a favor de la *Apache Software Foundation*.

Nuestro archivo se llama *apache_1.3.27-win32-x86-no_src.exe*, es un setUp el cual nos guía de una manera muy sencilla a través de la instalación del Servidor. Una pantalla en donde se solicita información importante, es la siguiente:



Los parámetros en la instalación de Apache

En donde dice Network Domain, se refiere al dominio de la Red. Server Name será el nombre de nuestro servidor Web y Administrator's Email Address es la dirección de correo electrónico utilizada por el servidor para enviar algún correo en caso de que se genere un error o algo similar.

En esta misma pantalla también aparece la opción de instalar el servidor Web como un Servicio de Windows, es la opción que utilizaremos.

Network Domain: servidorWebIsak.com
Server Name: www.servidorWebIsak.com
Administrator's Email Address: isakmtz@yahoo.com.mx

La ruta en donde instalaremos el servidor es la siguiente: C:\Zprog\Apache Group\

Una vez concluida la instalación, abrimos el Navegador y en la barra de direcciones tecleamos http://127.0.0.1/, esta es la dirección local, asignada para el servidor Web, debe aparecer la siguiente pantalla, de esta manera sabremos que Apache se instalo y se esta ejecutando de manera correcta.

Página de Apache indicando una instalación correcta

La dirección http://localhost/ debe tener el mismo efecto que http://127.0.0.1/, ambas nos llevarán a la página inicial del Sitio.

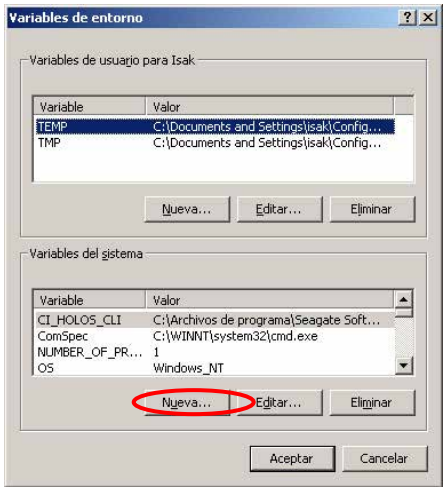
5.5.3 Instalación de Tomcat

La versión a Instalar de Tomcat, es la versión 4.0, esta versión implementa las especificaciones de los Servlets 2.3 y para JSP 1.2, es necesario tener previamente instalado el SDK de Java, la versión 1.2 o posterior.

Hay que asignar la variable de entorno JAVA_HOME a la ruta en donde ha sido instalada la versión del SDK de Java. Esto se realiza dando clic derecho al ícono de Mi Pc en el escritorio, y después en la pestaña de Avanzado, seleccionar Variables de entorno. Ahí se debe agregar una nueva variable de sistema con los siguientes datos:

Nombre de la variable: JAVA_HOME
Valor de la variable: C:\Zprog\sdk14

En nuestro caso, C:\Zprog\sdk14 es la ruta en donde esta instalado el SDK



Configuración de las variables del sistema

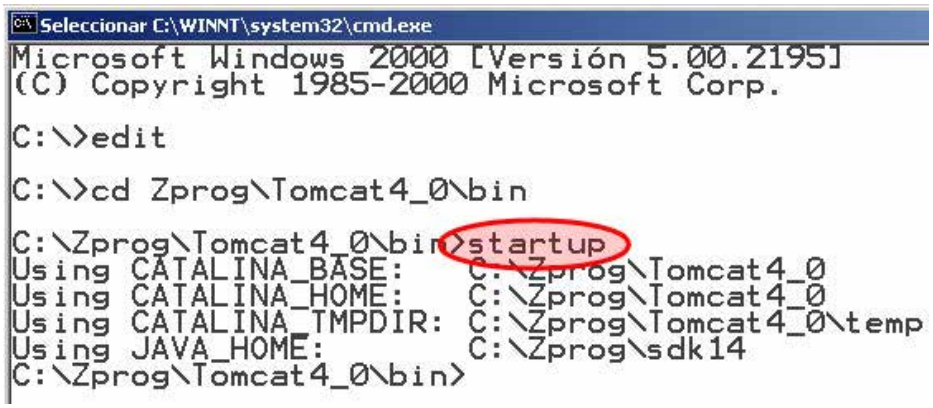
El siguiente paso es bajar la distribución de Tomcat, (jakarta-tomcat-4.0.6.zip) y desempacar los archivos en el disco duro, en nuestro caso lo hicimos en la carpeta C:\Zprog\Tomcat4_0.

Asignamos una nueva variable de entorno con los siguientes datos:

Nombre de la variable: CATALINA_HOME
Valor de la variable: C:\Zprog\Tomcat4_0

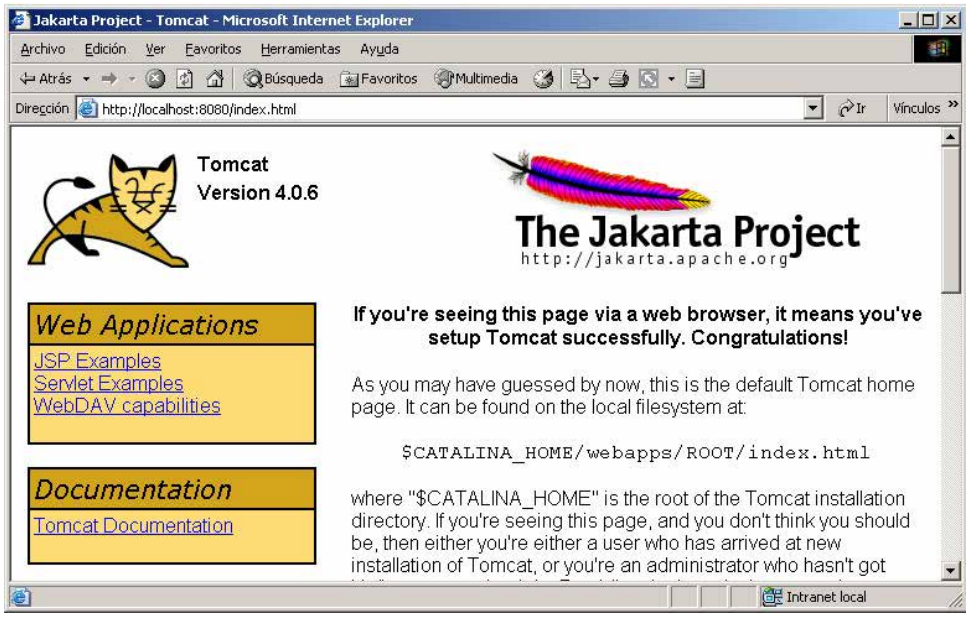
Iniciamos el servidor de páginas Web, (en nuestro caso es Apache por compatibilidad con los sistemas Linux, pero también trabaja para IIS de Microsoft) y una vez iniciado, desde la línea de comandos nos vamos al directorio en donde esta instalado Tomcat y en la carpeta bin tecleamos lo siguiente:

C:\startup



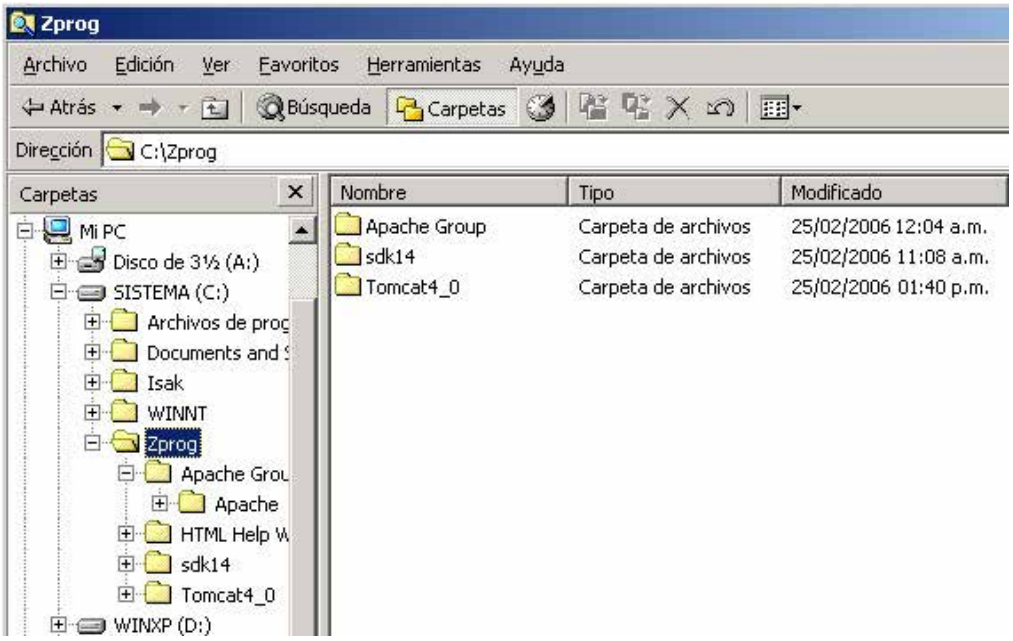
Inicio de Tomcat

De esta manera iniciamos el servidor Tomcat, para saber si esta trabajando tecleamos en la barra de direcciones del explorador: http://localhost:8080, debe aparecer la siguiente pantalla:



Página de inicio de Tomcat

La carpeta en donde tenemos instalado el SDK de Java, el servidor de páginas Web Apache y el contenedor de Servlet y JSP Tomcat es C:\Zprog y queda de la siguiente manera:

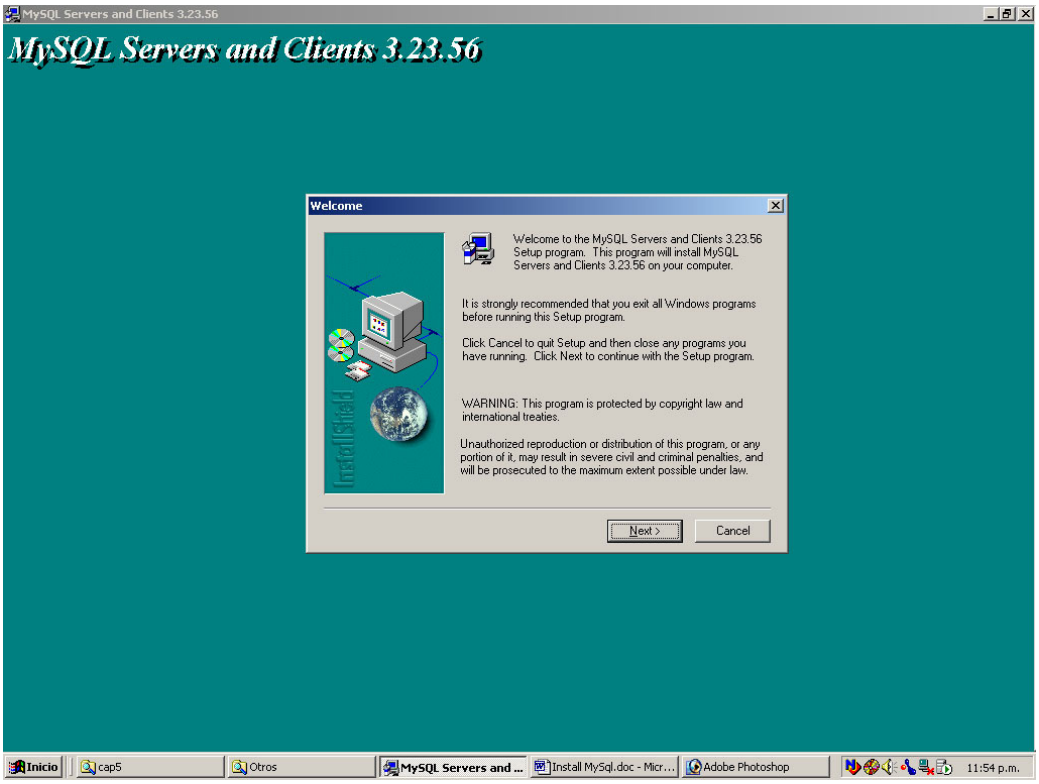


Directorio de instalación del SDK, Apache y Tomcat

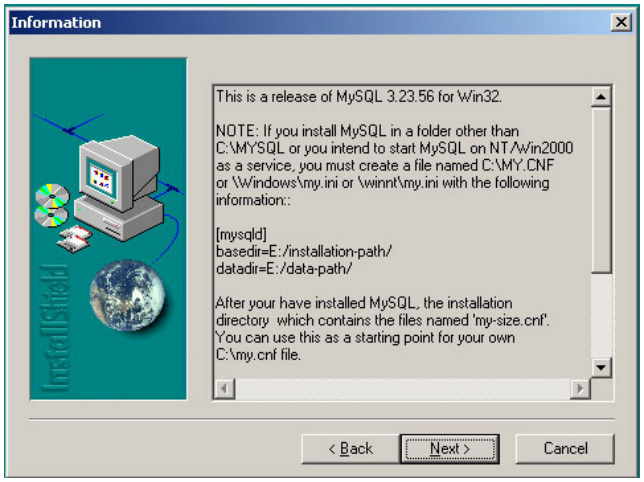
Si dentro de las páginas JSP se hace referencia a un driver de MySQL (*mysql-connector-java-2.0.14-bin.jar*), el archivo del driver hay que copiarlo a la carpeta

5.5.4 Instalación de MySQL

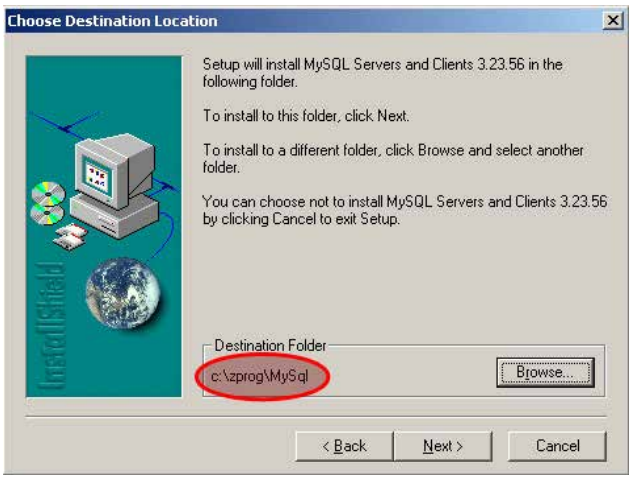
El servidor de BD utilizado es MySQL ver. 3.23.56, el SetUp de instalación se puede obtener desde la página de MySQL http://www.mysql.com/. Este SetUp tiene una interfase amigable, únicamente hay que especificar la ruta en donde se instalará la BD



Instalación de la Base de Datos MySQL

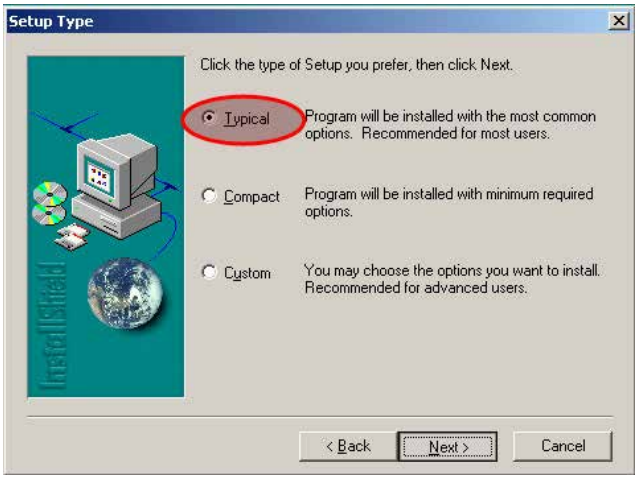


Otra pantalla de la instalación de la Base de Datos MySQL



Selección de la ruta de instalación

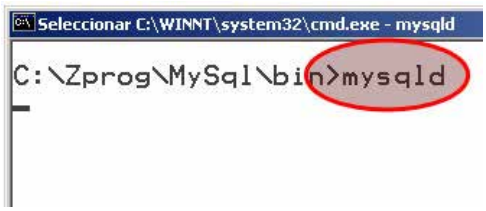
En la figura anterior se ve la pantalla para seleccionar la ruta en donde se instalará la BD, en nuestro caso el directorio es **C:\Zprog\MySql**, en la pantalla que se muestra en la Figura siguiente, seleccionamos la instalación Típica y es todo lo que se necesita para instalar la BD.



Configuración de MySQL

Una vez instalado el DBMS MySQL, es necesario levantar el servidor de la BD, esto se realiza de la siguiente manera:

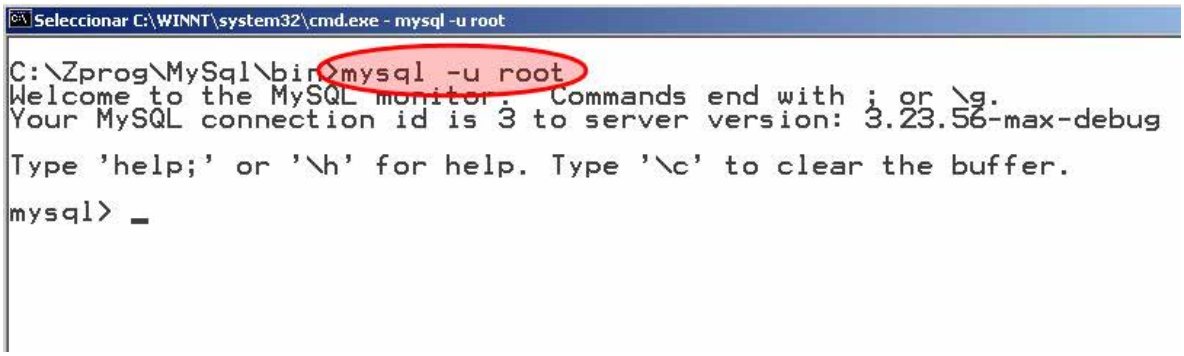
Desde la línea de comando, teclear lo siguiente:
>mysqld



Inicio de la Base de Datos

Y para conectarnos a la BD, desde la línea de comando tecleamos lo siguiente:
>mysql –u root

Si todo sale bien entonces nuestro DBMS esta instalado y listo para crear una BD.



Ingreso a la Base de Datos

5.5.4.1 Seguridad en la BD

Como se ve en la figura anterior, no pide contraseña, por razones de seguridad hay que asignar password a los usuarios y también privilegios. En este caso es un grave peligro que el usuario administrador no tiene un password para ingresar a la BD, también es importante cerrar el puerto 3306, pues se puede establecer una conexión a la BD a través de dicho puerto.

Lo primero es asignar un password al usuario root, esto es mediante las siguientes líneas:

```
UPDATE user SET Password=PASSWORD('root01')
            WHERE user='root';
FLUSH PRIVILEGES;
```

De esta manera, al intentar acceder a la BD, es necesario especificar el password, de lo contrario se negara el acceso a la BD:

```
>mysql -u root -p
Enter password:
ERROR 1045: Access denied for user: 'root@localhost' (Using
            password: NO)
```

Para agregar otro usuario, podemos hacer los siguiente:

```
GRANT ALL PRIVILEGES ON *.* TO isak@localhost IDENTIFIED
BY 'isak01' WITH GRANT OPTION;
```

5.5.4.2 Creación de la BD y la tabla

Una vez asignado el password para el super-usuario y agregado un usuario Administrador, crearemos la BD, la cual llevara el nombre dbChat

```
CREATE DATABASE dbChat
```

Dentro de esta BD estará nuestra tabla, la cual se define de la siguiente manera:

```
use dbchat;
create table salaUno (
    idNum int UNSIGNED NOT NULL AUTO_INCREMENT,
    nombre char(15) not null,
    msg char(150),
    fecha date,
    hora time,
    chatRoom char(15) not null,
    dirIp char(15),
    primary key (idNum)
);
```

- IdNum: Es la llave primaria del registro que almacena el mensaje
- nombre: Guarda el nombre de usuario que genero dicho mensaje
- msg: es el mensaje escrito por el usuario
- fecha: fecha en que fue escrito el mensaje
- hora: hora en que se escribió el mensaje
- chatRoom: Cuarto o sala en donde fue enviado el mensaje
- dirIp: Es la dirección IP de quien envió el mensaje.

salauno						
idNum	**nombre**	**msg**	**fecha**	**hora**	**chatRoom**	**dirIp**
1	Isak	Mensaje de Isak	2006-12-05	06:45:28	2	127.0.0.1
2	Carmen	Mensaje de Carmen	2006-12-05	06:46:31	2	127.0.0.1

```
use dbchat;
create table salaDesc (
 idNum int UNSIGNED NOT NULL AUTO_INCREMENT,
 nombre char(20) not null,
 tema char(30) not null,
 descSala char(150),
 descPer char(150),
 primary key (idNum)
);
```

- IdNum: Es la llave primaria del registro
- nombre: Guarda el nombre de la persona entrevistada
- tema: Tema a tratar en esa sala
- descSala: Descripción del tema a tratar en la sala
- descPer: Descripción de la persona entrevistada.

saladesc				
idNum	**nombre**	**tema**	**descSala**	**descPer**
1	Ing. Martinez	Primer tema del sitio	Descripción de la sala	Descripción de la persona

Con la BD y las tablas creadas, haremos pruebas de conexión a la BD mediante Java, para tales fines necesitamos instalar el driver de mysql: *com.mysql.jdbc.Driver*

Este driver es conocido como *MySQL Connector/J*, y se distribuye como un archivo jar, el cual contiene los archivos fuente y las clases, también existe un Jar con los archivos binarios de las clases, este archivo se llama *mysql-connector-j-2.0.13-bin.jar*.

Utilizaremos este ultimo para implementar el *mysql.jdbc,* y lo pondremos en la ruta en donde indica el *classpath* o en *$JAVA_HOME/jre/lib/ext*, recordemos que nuestra instalación de java fue en *C:\Zprog\sdk14* por tanto este archivo Jar quedara en *C:\Zprog\sdk14\jre\lib\ext.*

La siguiente clase nos muestra de manera muy sencilla como se realiza el acceso a una BD mediante el JDBC de MySQL:

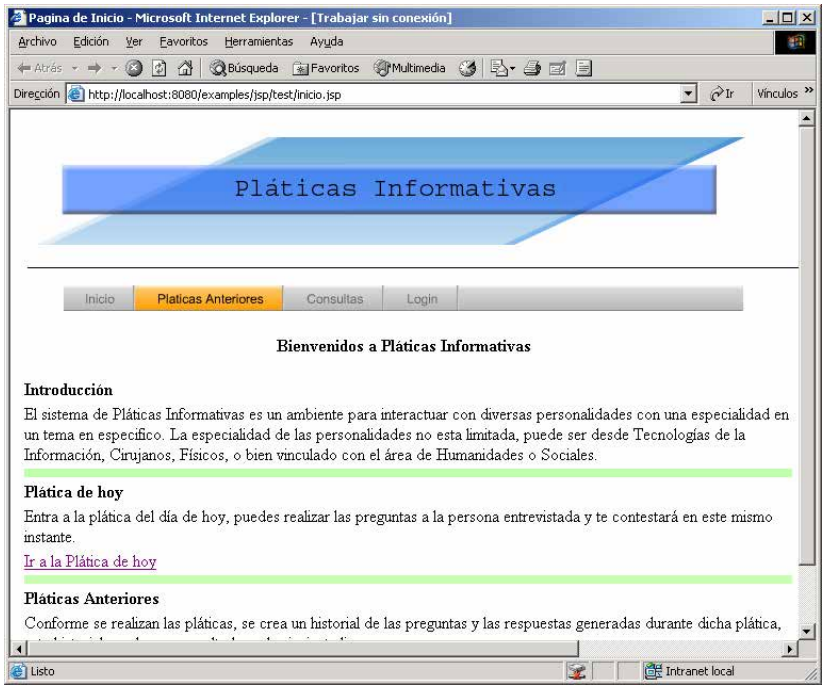
```
import java.io.*;
import java.sql.*;

public class sqlEje {

public void getTodosDatos () {
Connection CONEXION = null;
String LsQuery= "";
ResultSet tabla= null;

Statement stmt;
try {
System.out.println (LsQuery);
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  CONEXION = DriverManager.getConnection
            ("jdbc:mysql://localhost/dbChat","usuarioUno","pass01");

  stmt = CONEXION.createStatement ();

  LsQuery = "SELECT nombre, msg, fecha, chatRoom, dirIp FROM salaUno";
  tabla = stmt.executeQuery (LsQuery);  while(tabla.next()) {
    System.out.println ("  "+ tabla.getString(1));
    System.out.println ("  "+ tabla.getString(2));
    System.out.println ("  "+ tabla.getDate(3));
    System.out.println ("  "+ tabla.getString(4));
    System.out.println ("  "+ tabla.getString(5));
  }

  } catch (java.lang.ClassNotFoundException e) {
    System.err.println ("  Error, no se encontro: "
                              +e.getMessage());
  } catch (SQLException e) {
      System.err.println ("  Error, Excepcion SQL: " +e.getMessage());
  } catch (Exception e) {
      System.err.println("  Error Interno getTodosDatos: "+e);
  }


try {
  CONEXION.close();
  } catch (SQLException e) {
  System.err.println ("Error. No se puede cerrar la Base: "
                              +e.getMessage());
    }
 } /* Fin de getTodosDatos */

}
```

La implementación de la clase `sqlEje` se puede hacer en un archivo principal que incluya la clase `main`, dentro de la cual se crea una instancia de la clase `sqlEje` y se manda llamar el método `getTodosDatos`.

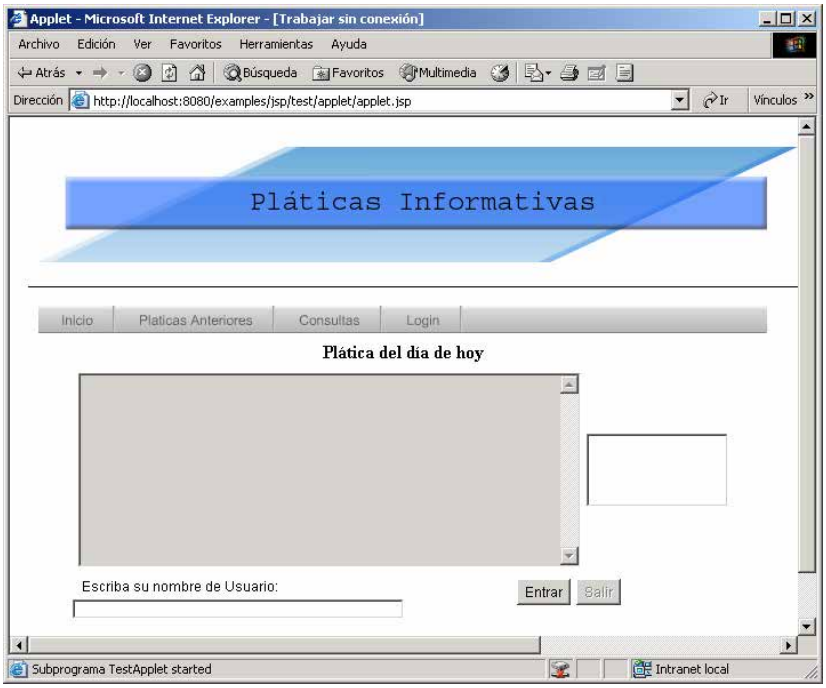
```
class principal {

  public static void main( String args[] ) {
  sqlEje sqlA = new sqlEje ();
  sqlA.getTodosDatos ();
  }/*  Fin de main  */
}
```

5.5.5 Ejemplo del Sistema.

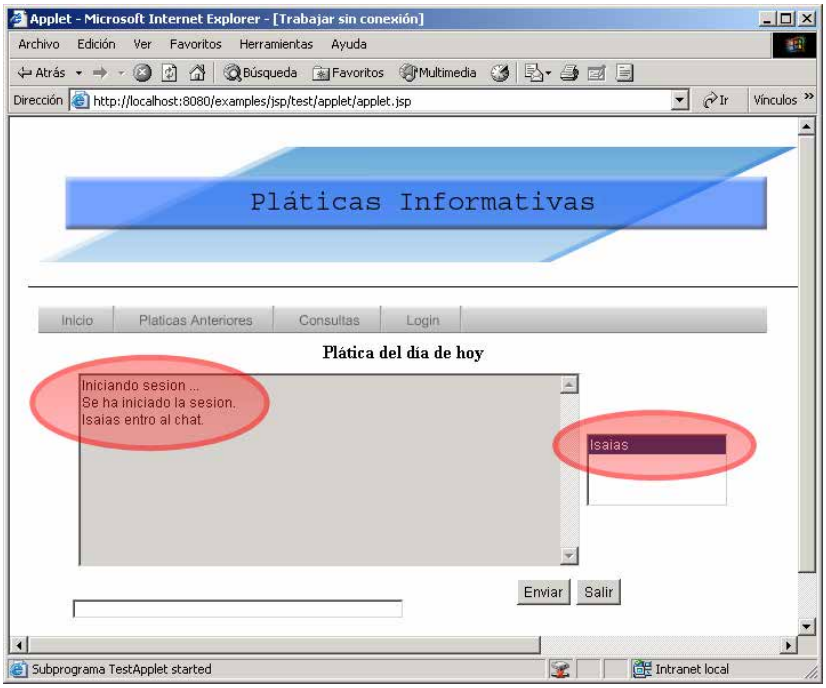
A continuación se muestran las pantallas del cliente instalado dentro del Sitio Web.



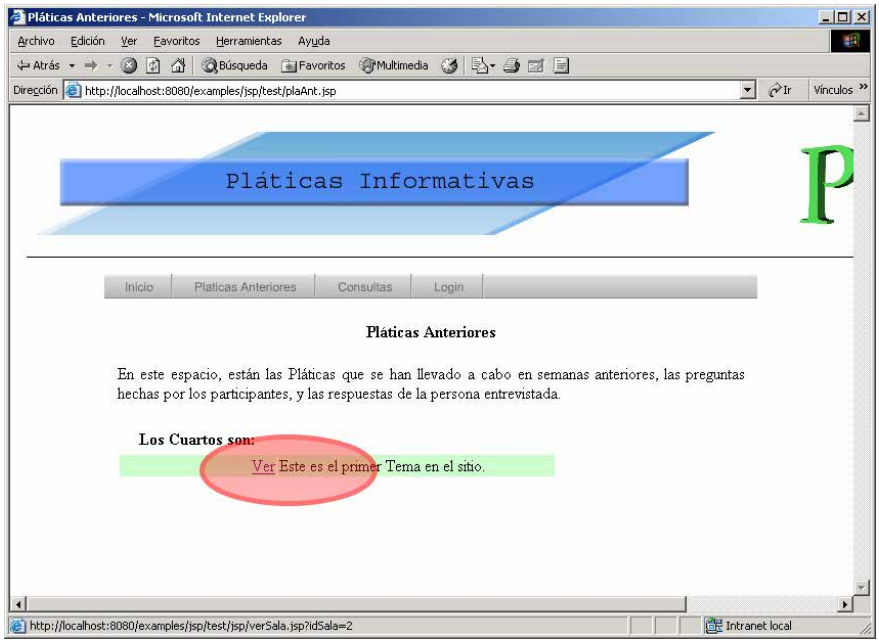
Página de Inicio



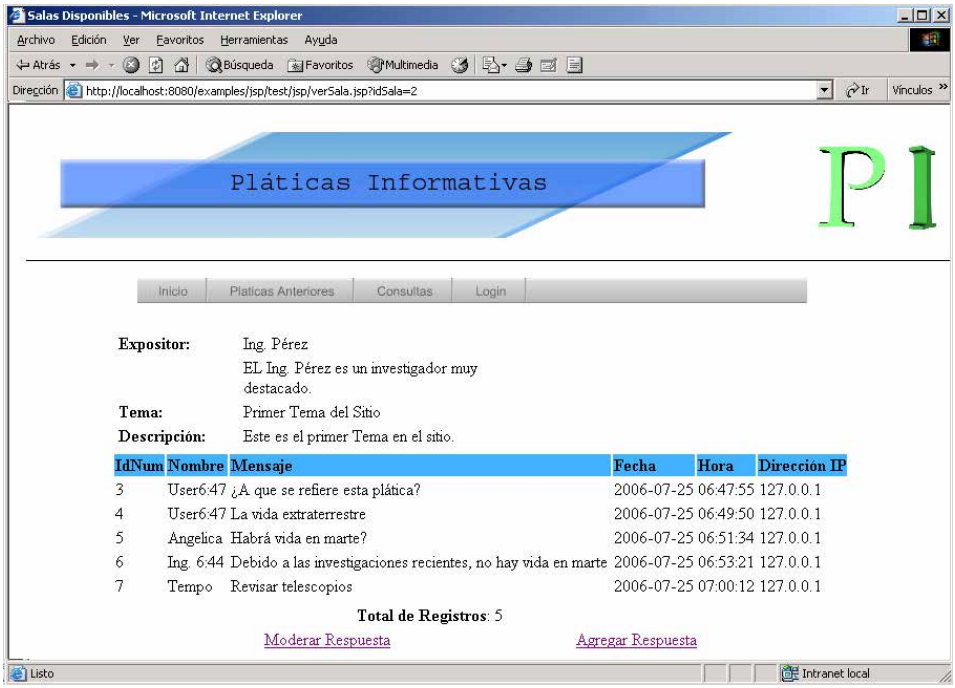
El applet dentro del Sitio Web



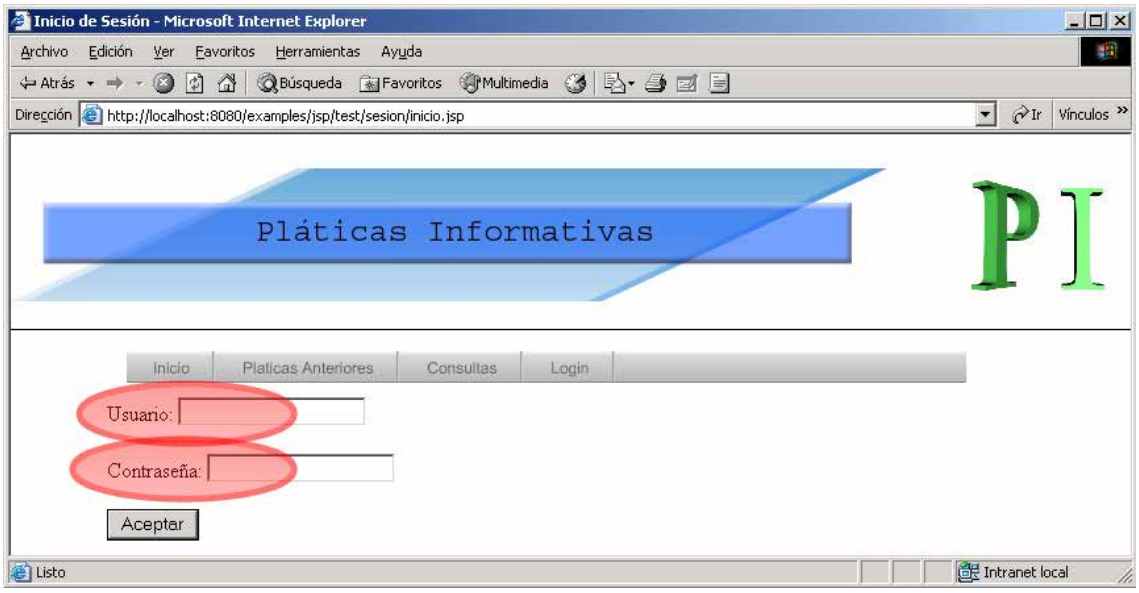
En esta pantalla ya se inicio una sesión.



En el sitio se pueden ver los cuartos que se han creado y también las preguntas hechas



Dentro de cada cuarto, están las preguntas y respuestas.



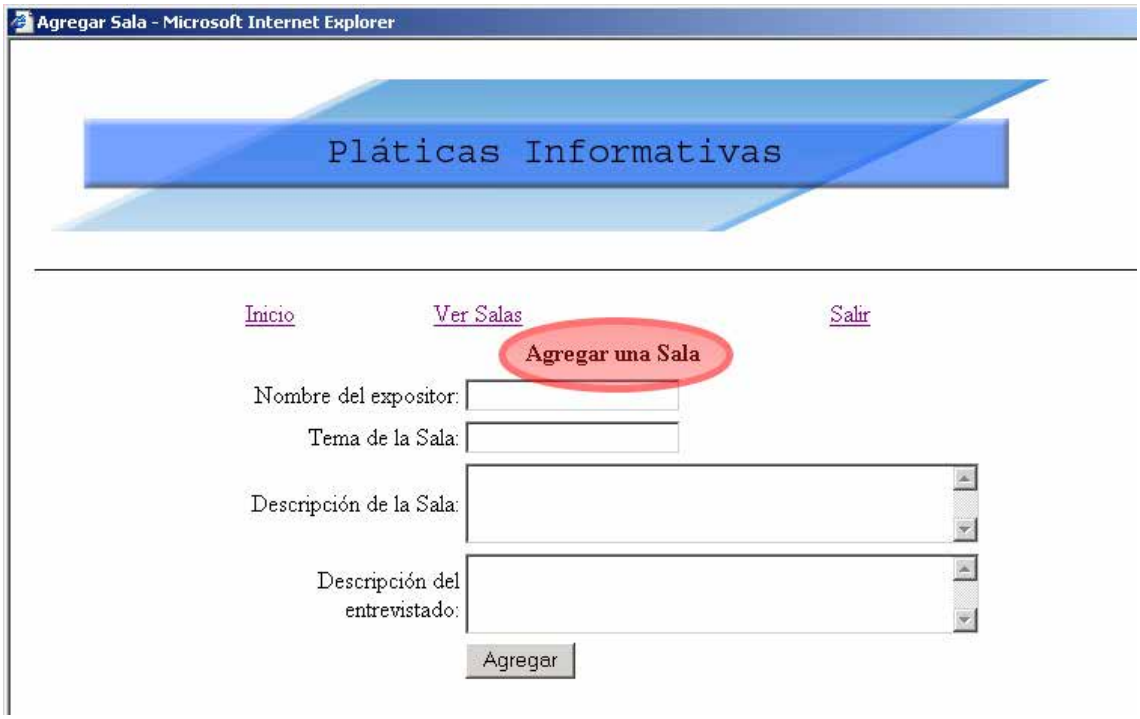
Solicitud de usuario y password

El sitio tiene la administración de las salas o cuartos, para ello se necesita un usuario y un password. En esta administración, se puede agregar, borrar y editar una sala.

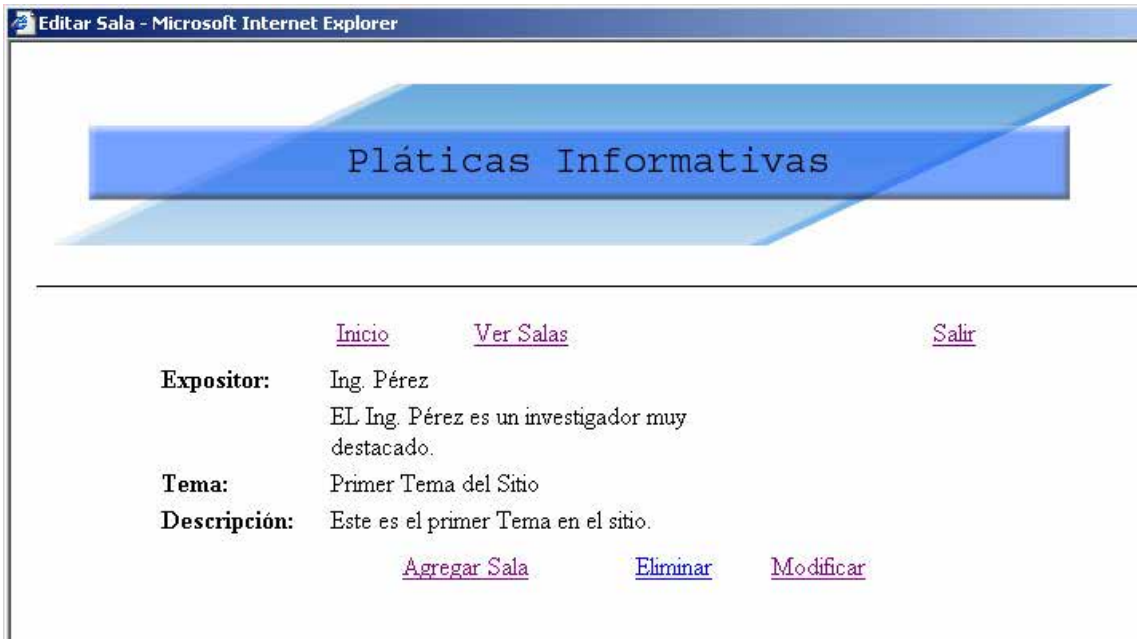
En las siguientes imágenes se ilustra algunas pantallas del inicio de sesión para la administración de las salas.

Inicio de sesión

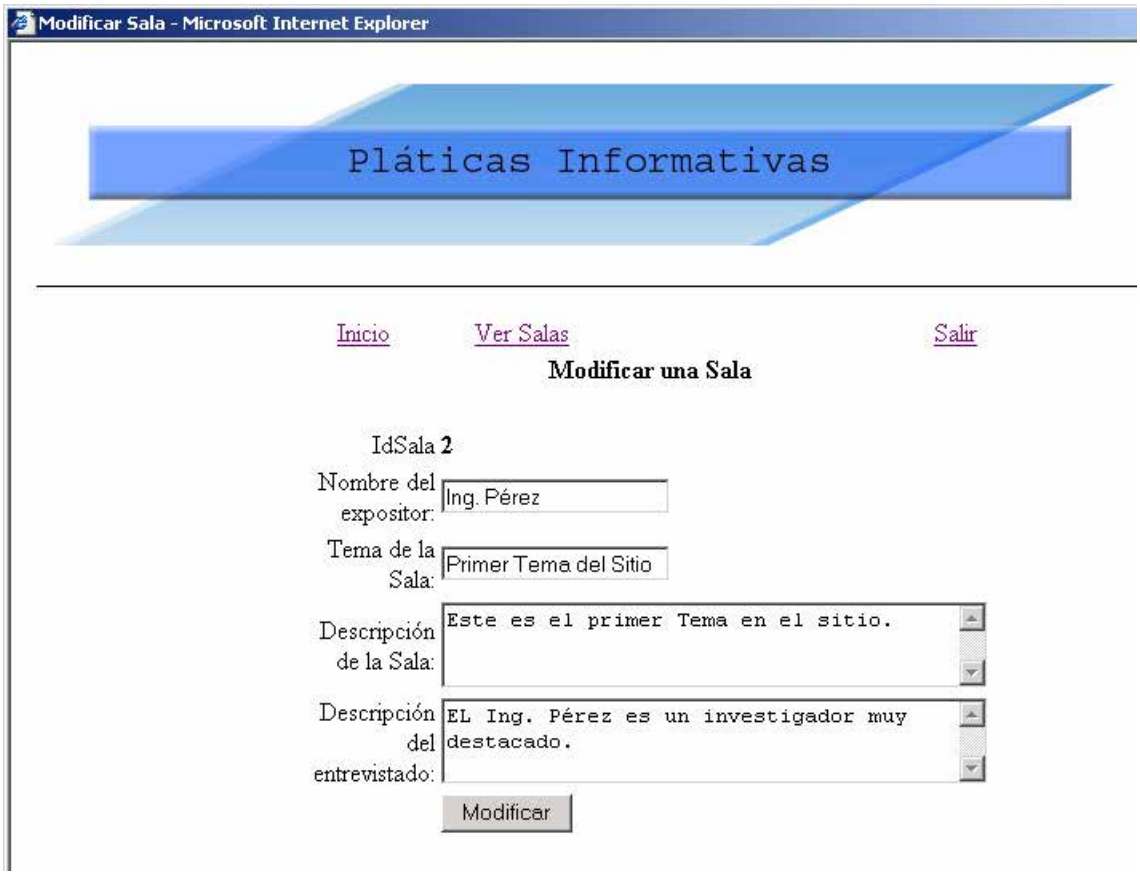
Esta es la primera pantalla mostrada en el inicio de sesión. Cabe destacar que se valida el usuario y password para esta pantalla, pero también se valida la sesión activa para cada pantalla en esta ***área*** del sitio.



Aquí se agrega una Sala



Las salas se pueden agregar, eliminar y modificar.



Es la pantalla para modificar una sala.

Conclusiones

El lenguaje de programación Java se ha utilizado de manera muy extensa tanto para desarrollo de aplicaciones *Stand Alone* como en contenido Web. Este lenguaje es muy valioso en entornos de red, sin embargo va más allá y su desarrollo esta presente en aplicaciones de propósito general y para teléfonos móviles debido a su diseño orientado a la máxima portabilidad.

En el trabajo aquí expuesto, se ilustra con una breve aplicación el uso de esta tecnología, queda demostrado la interacción de Java con otras tecnologías (la Base de Datos MySQL y el servidor de páginas Web Apache) y la manera de cómo se puede implementar un sistema desde principio a fin.

Durante el desarrollo de dicha aplicación, aparecieron diferentes retos, los cuales fueron resueltos basándose en la investigación, la ayuda de los conocimientos adquiridos en los primero capítulos y en algunas ocasiones de manera empírica. Cabe resaltar que aunque los objetivos se lograron, el desarrollo de un sistema similar o con mayores características requiere de conocimientos amplios en la programación y en el lenguaje Java.

La aplicación programada tiene fines educativos, tanto dentro de la implementación como durante el desarrollo de la misma, confiando en que algún alumno interesado en aprender de manera más profunda el uso del lenguaje de programación Java, revise este trabajo de tesis y logre entender más rápidamente las ventajas de este lenguaje y su aplicación en un ejemplo que vincula diversas etapas en el desarrollo.

Bibliografía.

- Java Server Pages, David Geary, 2002, Editorial Pearson Educación.
- Programando con Java, Tim Ritchey, 1997. Editorial Prentice-Hall.
- Descubra Java 1.2, Mike Morgan, Madrid 1999, Editorial Prentice-Hall.
- Aprendiendo Java 2 en 21 dias, Laura Lemay y Rogers Cadenhead, México 1999. Editorial Prentice-Hall.
- El lenguaje de programación Java, Fco. Javier Ceballos, España 2001, Editorial Ra-Ma.
- Programación en Java, Pedro Manuel Cuenca Jiménez, Editorial Anaya Multimedia, Madrid 1997.
- Black Art of Java Game Programming, Joel Fan, 1996 Editorial MacMillan Computer Publishing.
- Java for C/C++ Programmers, Michael C. Daconta, Wesley Computer Publishing, U.S.A. 1996
- Descubra Java 1.2, Mike Morgan, Madrid 1999, Editorial Prentice-Hall.
- Teach your self Java in 21 Days, Laura Lemay & Charles L. Perkins, EU 1996, Editorial Sams .NET.
- El lenguaje de programación Java, Ken Arnold, James Gosling y David Holmes, España 2001, Editorial Addison Wesley.
- Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Editorial Addison-Wesley, 1999.
- Programación de Base de Datos con JDBC y Java, George Reese, Editorial Anaya Multimedia, Madrid 2001.
- Patrones de diseño aplicados a Java, Stephen Stelting, Olav Maassen, Prentice Hall, Madrid 2003.
- Information Systems Concepts for Managament, Lucas H.C. Jr. Editorial McGraw-Hill, New York 1986.
- http://ww.sun.com

- http://java.sun.com/j2ee/1.4/
- http://www.dei.inf.uc3m.es/docencia/psciclo/pa4/
- http://www.mysql.com/

Conclusiones

El lenguaje de programación Java se ha utilizado de manera muy extensa tanto para desarrollo de aplicaciones *Stand Alone* como en contenido Web. Este lenguaje es muy valioso en entornos de red, sin embargo va más allá y su desarrollo esta presente en aplicaciones de propósito general y para teléfonos móviles debido a su diseño orientado a la máxima portabilidad.

En el trabajo aquí expuesto, se ilustra con una breve aplicación el uso de esta tecnología, queda demostrada la interacción de Java con otras tecnologías (la Base de Datos MySQL y el servidor de páginas Web Apache) y la manera de cómo se puede implementar un sistema desde principio a fin.

Durante el desarrollo de dicha aplicación, aparecieron diferentes retos, los cuales fueron resueltos basándose en la investigación, la ayuda de los conocimientos adquiridos en los primero capítulos y en algunas ocasiones de manera empírica. Cabe resaltar que aunque los objetivos se lograron, el desarrollo de un sistema similar o con mayores características requiere de conocimientos amplios en la programación y en el lenguaje Java.

La aplicación programada tiene fines educativos, tanto dentro de la implementación como durante el desarrollo de la misma, confiando en que algún alumno interesado en aprender de manera más profunda el uso del lenguaje de programación Java, revise este trabajo de tesis y logre entender más rápidamente las ventajas de este lenguaje y su aplicación en un ejemplo que vincula diversas etapas en el desarrollo

Con la finalidad de proporcionar ayuda para el análisis de alguna parte del contenido de esta tesis, pueden contactarme al siguiente correo electrónico: isakmtz@hotmail.com.

Bibliografía.

- Java Server Pages, David Geary, 2002, Editorial Pearson Educación.
- Programando con Java, Tim Ritchey, 1997. Editorial Prentice-Hall.
- Descubra Java 1.2, Mike Morgan, Madrid 1999, Editorial Prentice-Hall.
- Aprendiendo Java 2 en 21 dias, Laura Lemay y Rogers Cadenhead, México 1999. Editorial Prentice-Hall.
- El lenguaje de programación Java, Fco. Javier Ceballos, España 2001, Editorial Ra-Ma.
- Programación en Java, Pedro Manuel Cuenca Jiménez, Editorial Anaya Multimedia, Madrid 1997.
- Black Art of Java Game Programming, Joel Fan, 1996 Editorial MacMillan Computer Publishing.
- Java for C/C++ Programmers, Michael C. Daconta, Wesley Computer Publishing, U.S.A. 1996
- Descubra Java 1.2, Mike Morgan, Madrid 1999, Editorial Prentice-Hall.
- Teach your self Java in 21 Days, Laura Lemay & Charles L. Perkins, EU 1996, Editorial Sams .NET.
- El lenguaje de programación Java, Ken Arnold, James Gosling y David Holmes, España 2001, Editorial Addison Wesley.
- Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Editorial Addison-Wesley, 1999.
- Programación de Base de Datos con JDBC y Java, George Reese, Editorial Anaya Multimedia, Madrid 2001.
- Patrones de diseño aplicados a Java, Stephen Stelting, Olav Maassen, Prentice Hall, Madrid 2003.
- Information Systems Concepts for Managament, Lucas H.C. Jr. Editorial McGraw-Hill, New York 1986.
- http://ww.sun.com

- http://java.sun.com/j2ee/1.4/
- http://www.dei.inf.uc3m.es/docencia/psciclo/pa4/
- http://www.mysql.com/