TRABAJO DE GRADO

Aplicación a distancia para estudiantes

Antonia Maraviglia - Mariángeles Pucci

Director: Lic. Javier Díaz

Facultad de Informática

Universidad Nacional de La Plata

INDICE

LENGUAJE JAVA

AMBIENTES DE DESARROLLO

BASE DE DATOS

PRÓLOGO

En este informe se presentan los temas que se estudiaron y utilizaron para el desarrollo de la aplicación.

Se divide en tres partes: el Lenguaje Java, el Ambiente de Desarrollo y la Base de Datos.

La primer parte comprende desde conceptos básicos del lenguaje hasta conceptos avanzados del mismo, y permite, a través de su lectura, obtener los conocimientos necesarios para la programación en lenguaje Java.

Dado que el lenguaje no cuenta con un ambiente de desarrollo, se realizó una comparación entre los ambientes más renombrados para elegir uno. Se optó por el Visual Age for Java 2.0. Se dispone de esta información en la segunda parte del informe.

En la última parte se encuentran las principales características del motor de base de datos Oracle.

Lenguaje Java

UN BREVE RESUMEN DE SU CORTA HISTORIA...

Java se remonta a 1991, cuando un grupo de ingenieros de Sun, buscaron un lenguaje que pudiera ser usado para artefactos de consumo inteligentes. Estos artefactos tenían la característica de no poseer mucha memoria, mucha energía, y cada uno de los fabricantes elegían una CPU distinta. Por esto, dicho lenguaje debía ser pequeño, con una generación de código ajustada y que no dependiera de un solo tipo de CPU.

El proyecto se conoció con el nombre de Green. Las características requeridas hicieron resurgir el modelo que un lenguaje llamado UCSD Pascal probó en los tempranos años de las PC's. Este diseñaba un lenguaje portable que generaba código intermedio que se ejecutaba en cualquier máquina que tuviera un interpretador adecuado. El código intermedio generado con este modelo era siempre pequeño, así como también su interpretador; esto resolvía el problema principal.

Sin embargo, la gente de Sun, trabajaba en un ambiente Unix, por lo tanto basaron su lenguaje en C + +, en lugar de Pascal. En particular orientaron su lenguaje a la filosofía de objetos y no a la de procedimientos.

El lenguaje Java fue originalmente bautizado como "Oak" (roble), por James Gosling. Su inspiración para dicho nombre provino de un gran árbol de roble que veía desde la ventana de su oficina en Sun Microsystems. Más tarde, el equipo de desarrollo de Java descubrió que Oak era el nombre de otro lenguaje, motivo por el cual lo rebautizaron como Java (nombre de un tipo de café).

En 1992, distribuyeron el proyecto, pero no encontraron a nadie que lo produjera. Así pasaron todo el año 1993 y parte de 1994, buscando alguien que quisiera vender su tecnología. Mientras tanto la World Wide Web iba creciendo cada vez más, y así los desarrolladores del proyecto se dieron cuenta que podían manifestar toda la potencialidad de su proyecto, construyendo un browser amigable, que fuera independiente de la plataforma, seguro, confiable, características que hasta ese momento no eran importantes en el mundo de las estaciones de trabajo. Aparece entonces el browser denominado HotJava.

Transcurría el año 1995, cuando la construcción del Browser HotJava se llevó a cabo. Para su creación se utilizó el lenguaje Java, aprovechando para demostrar las capacidades del lenguaje. Los creadores además tenían en mente lo que hoy conocemos como *Applets*, por lo tanto, hicieron que el browser fuera capaz de interpretar los *bytecodes* intermedios.

El verdadero auge de Java comenzó hacia fines de 1995, cuando Netscape permitió visualizar las aplicaciones para Internet construidas con Java.

CARACTERÍSTICAS DE JAVA

♣ Java es simple;

Java es similar a C++, pero resulta mucho más simple. Todas aquellas características propias de los lenguajes de alto nivel que no son absolutamente necesarias, han sido dejadas de lado. Por ejemplo, Java no tiene sobrecarga de operadores, archivos de encabezado, preprocesadores, aritmética indicada, uniones, estructuras, arreglos multidimensionales, templates, o conversión de tipo implícita. Además, los programadores ya no deben preocuparse acerca de la administración de la memoria, pues Java incorpora un programa denominado "Garbage Collector", que hace un scanning de la memoria y libera automáticamente cualquier pieza de memoria que ya no está siendo utilizada.

♣ Es Orientado a Objetos;

Java es un lenguaje de programación orientado a objetos. La mayoría de las cosas dentro de Java son objetos, con la excepción de tipos simples parecidos a números y operadores booleanos.

Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo, y tal como sucede en todo lenguaje orientado a objetos, el código de Java está organizado en clases. Cada clase define el comportamiento de un objeto. Una clase puede heredar conductas desde otra clase diferente. En la raíz de la jerarquía de clases, siempre está la clase Objeto.

Soporta una jerarquía de clases con una sola herencia. Esto significa que cada clase solamente puede heredar desde una clase a la vez. Algunos lenguajes también admiten herencia múltiple, pero ésto puede ser confuso y hacer que el lenguaje resulte innecesariamente complicado. Es difícil de imaginar, por ejemplo, qué puede llegar a hacer un objeto que hereda conductas desde dos clases diferentes.

Java también soporta interfaces, las cuales son clases abstractas. Esto permite que los programadores puedan definir métodos para interfaces sin tener que preocuparse inmediatamente en como se implementarán dichos métodos. Una clase puede implementar múltiples interfaces. Esto representa contar con muchas de las ventajas de las verdaderas

herencias múltiples, pero sin muchos de sus problemas. Un objeto puede implementar varias interfaces.

Java incorpora funcionalidades inexistentes en C++, como por ejemplo, la resolución dinámica de métodos. En C++ se suele trabajar con librerías dinámicas (DLL's) que obligan a recompilar la aplicación cuando se retocan las funciones que se encuentran en su interior. Este inconveniente Java lo resuelve mediante una interfaz específica llamada RTTI (RunTime Type Identification), que define la interacción entre objetos excluyendo variables de instancia o implementación de métodos. Las clases en Java tienen una representación en el runtime que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

- Es Distribuido;

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como *http* y *ftp*.

Las aplicaciones Java pueden abrir y acceder a los objetos a través de Internet (vía las URL's), tan fácil como se puede acceder a archivos locales.

Java proporciona las librerías y herramientas para que los programas puedan ser distribuidos.

- Es interpretado y compilado;

Al ejecutarse un programa Java, este es en primer término compilado a código en bytes. Estos *bytecodes* son similares a instrucciones de máquina lo cual hace que los programas Java lleguen a ser muy eficientes. Sin embargo, los *bytecodes* no son específicos en ninguna máquina en particular, por lo cual pueden ser ejecutados en varias y diferentes computadoras sin tener que ser recompilados.

Los programas fuente Java son compilados en archivos de clases, los cuales contienen la representación del programa en *bytecodes*. En un archivo de clases Java, todas las referencias a métodos y variables de instancia, son realizadas por nombre, y son resueltas cuando el código es ejecutado. Esto hace que el código sea más general y menos susceptible a los cambios, y todo sin perder nada de su eficacia.

- Es Robusto;

Los programas Java no pueden hacer que una computadora se cuelgue. El sistema Java controla cuidadosamente cada acceso a memoria y se asegura que el mismo sea legal, evitando así que cause algún problema.

Pero hasta los programas Java pueden tener sus bugs. Si sucede algo imprevisto, los programas no se pinchan, pero lanzan un código de excepción. Los programas han de identificar dichas excepciones y manejarlas.

Los programas tradicionales pueden acceder a cualquier parte y a toda la memoria de la computadora. Un programa puede (en forma no intencional), cambiar cualquier valor en la memoria, lo que puede causar problemas. Los programas Java, sólo pueden acceder a aquellas partes de la memoria a las que se les está permitido hacerlo, de modo que un programa Java no puede cambiar un valor, si no está autorizado a hacerlo.

- Es Seguro;

Los poderosos mecanismos de seguridad de Java actúan sobre cuatro niveles diferentes de la arquitectura de sistemas:

En primer lugar, tenemos que el lenguaje Java en sí mismo fue diseñado para ser seguro, y es su compilador el que garantiza que el código fuente no pueda violar las normas de seguridad. Con respecto a esto, sabemos que los programas Java no cuentan con punteros, y que, además, es un lenguaje fuertemente tipado, de modo que resulta posible verificar estos programas antes de su ejecución.

En segundo término, tenemos que todos los bytecodes ejecutados por el motor runtime de Java son verificados para resguardar que también ellos obedezcan a dichas reglas. Esta capa protege contra la posibilidad de que un compilador alterado pueda producir código que viole las reglas de seguridad. Un programa Java que ha sido verificado, garantiza que no ha de romper ninguna de las restricciones del lenguaje y, además, que puede ser ejecutado con toda seguridad. La verificación bytecode de Java es utilizada por los Web browsers para asegurarse que las *Applets* no contengan virus.

El tercer tema es el relativo al Alimentador de Clases, el cual garantiza que las clases no violen el espacio de nombres o restricciones de acceso en el momento en que son cargadas dentro del sistema.

Finalmente, la seguridad especifica para APIs evita que las *Applets* realicen acciones destructivas. Esta capa depende de la seguridad e integridad provista por las otras tres primeras capas.

- Es Multihilado;

Un programa Java puede tener más de un thread o hilo de ejecución. Un Thread es un flujo de control secuencial dentro de un programa. Por ejemplo, se puede hacer un trabajo de computación extenso sobre un thread, mientras que otros interactúan con el usuario, así los usuarios no tienen que detener su trabajo para esperar que Java complete operaciones prolongadas.

La programación en un entorno multihilado es siempre dificultosa, debido a que son muchas las cosas que pueden ocurrir al mismo tiempo. Java, sin embargo, provee dispositivos destinados a la sincronización que resultan muy sencillos para su uso, lo que hace que la programación sea mucho más amigable. Para dicha sincronización cuenta con un extenso conjunto de primitivas, basadas en el paradigma de monitores.

Los hilos de Java están generalmente mapeados dentro de los hilos del sistema operativo, siempre y cuando el sistema operativo subyacente soporte esta clase de actividad. De esta forma, se puede decir que las aplicaciones escritas con Java son lo que se denomina "MP-Hot", lo cual significa que se benefician si son ejecutadas en una máquina multiprocesamiento.

La programación multihilado permite mejorar la interactividad y la performance del Sistema.

- Es Portable;

El lenguaje Java es el mismo en cualquier computadora. Por ejemplo, los tipos simples no varían: un integer es siempre de 32 bits y un longint es siempre de 64 bits.

Sin embargo, esto mismo no resulta cierto en otros lenguajes de programación, tales como C o C++, debido a que estos lenguajes están tan vagamente definidos, cada compilador y entorno de desarrollo es ligeramente diferente, lo que convierte la portabilidad en una ardua tarea. A diferencia de ésto, portar programas Java es muy fácil, ya que no necesitan ser recompilados.

- Es Independiente de la Plataforma;

El compilador genera un formato de archivo neutral a la arquitectura, el código compilado es ejecutable en cualquier procesador dado la presencia del sistema runtime de Java. El compilador Java realiza esta conversión, generando instrucciones de *bytecode*, los cuales no dependen de ninguna arquitectura de computadora en particular. Estos *bytecodes* están diseñados para ser fácilmente interpretados y traducidos a código nativo de máquina.

- Es Extensible;

Es posible hacer una interfaz del programa Java a las librerías de software existentes que estén escritas en otro lenguaje. Debido a que las estructuras de datos de Java son muy parecidas a las estructuras y tipos de C, esto resulta sumamente simple.

Un programa Java puede declarar que determinados métodos le resulten nativos. Estos métodos nativos son luego mapeados en funciones definidas en librerías de software que son dinámicamente encadenadas dentro de una máquina virtual.

La Máquina Virtual de Java

El elemento más importante de Java es lo que se denomina Virtual Machine. Está modelada en una pequeña y eficiente CPU; lleva código compilado al nivel de byte de Java y lo ejecuta como si fuese lenguaje de máquina. El nivel menor de Virtual Machine convierte a este código de pseudo máquina en llamadas reales del hardware (trabajando con cualquier sistema operativo que resida sobre la máquina en la que está funcionando).

La Virtual Machine es una Unidad Aritmético-Lógica (ALU) con variables locales y globales. Las variables locales son utilizadas para el almacenamiento provisorio y llamadas a subrutinas, mientras que las variables globales se utilizan para saber, por ejemplo, en qué lugar de la memoria se está ejecutando el código de máquina de programa a nivel de byte de Java, cuál es el objeto actual, en qué lugar de la memoria están almacenados los datos más relevantes, y otra clase de información global.

Para implementar correctamente la Máquina Virtual de Java se necesita leer el formato de archivo Java.class y ejecutar las operaciones que se especifican en él. Los detalles de implementación que no forman parte de la especificación de la Máquina Virtual de Java limitan innecesariamente la creatividad de los implementadores. Por ejemplo, el esquema de memoria de las áreas de ejecución de datos, el algoritmo de Garbage Collection usado y cualquier optimización de *bytecodes* (por ejemplo, la traducción en código de máquina) son dejados a criterio del programador.

ALGO MÁS ACERCA DE LA SEGURIDAD DE JAVA...

Como ya mencionamos anteriormente los mecanismos de seguridad de Java actúan a cuatro niveles diferentes. A continuación veremos cómo funciona cada una de las capas.

1. El lenguaje y el compilador

El lenguaje Java y su compilador forman, en cuanto a seguridad, la primera línea de defensa. Java fue diseñado para ser un lenguaje seguro.

La mayor parte de los lenguajes tipo C cuentan con facilidades para controlar el acceso a objetos, pero también tienen forma de generar acceso a los mismos. Esto introduce dos problemas que pueden resultar fatales para cualquier sistema realizado con estos lenguajes. Uno de ellos es que ningún objeto puede autoprotegerse de modificaciones realizadas desde su exterior, duplicación o falsificación. El otro radica en que un lenguaje con punteros muy poderosos tiene más posibilidades de tener serios bugs que comprometan la seguridad. Estos bugs en los punteros, en los que un pointer "prófugo" se ocupa de modificar la memoria de los objetos de alguien, fueron los responsables de muchos inconvenientes en Internet.

Java aventa todas estas amenazas al eliminar por completo de su lenguaje los punteros. De todos modos quedan en él algunas referencias a objetos, pero que son controladas para que resulten seguras.

Por lo tanto la definición del lenguaje y el compilador, crean una poderosa barrera para la seguridad en Java.

2. Verificación de los Bytecodes

¿Qué sucede si el compilador de Java se reescribe para malos propósitos? Para evitar que los bytecodes violen alguno de los requerimientos de seguridad, antes de correrlos, el runtime los somete a una serie de rigurosos tests, que permiten verificar que los bytecodes no alteren punteros, violen restricciones de acceso, accedan a objetos diferentes a ellos mismos, hagan llamadas a métodos con valores o tipos inapropiados, o desborden las pilas. Esto lo realiza utilizando información extra sobre tipos en un archivo

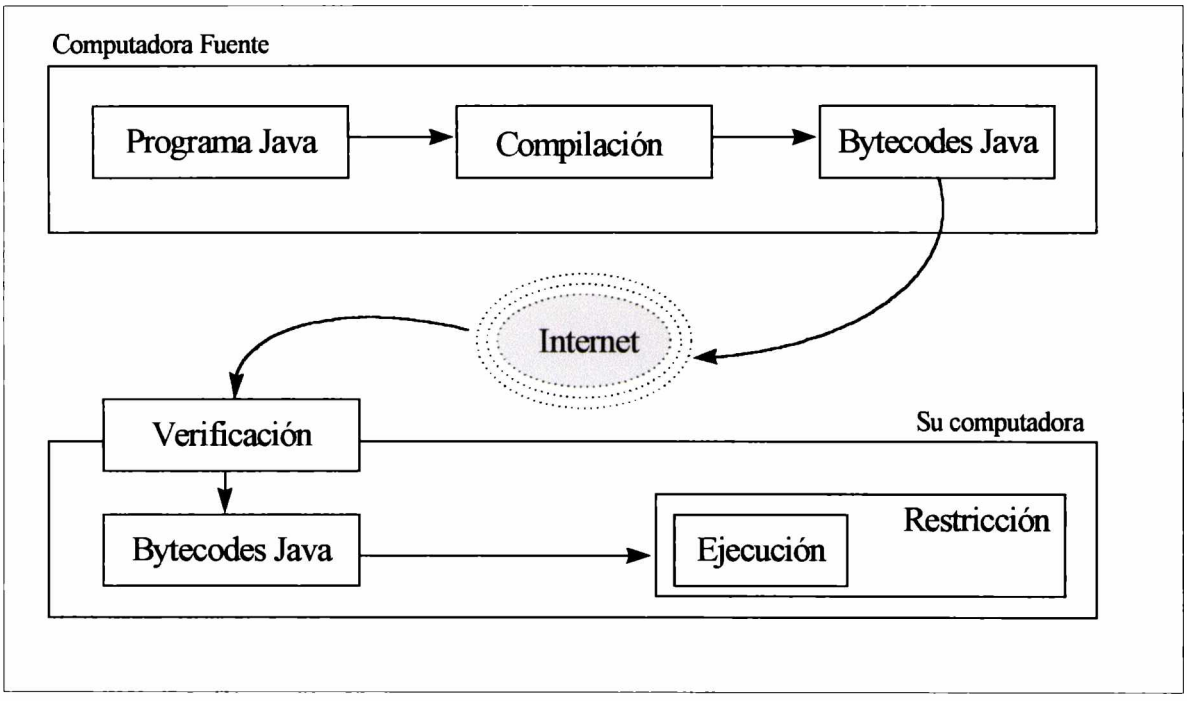
.class, y la tarea la lleva a cabo una parte del runtime, llamada *Verifier*. De esta forma actúa como si fuera una especie de portero del entorno de runtime, dejando pasar solamente a los bytecodes que cumplen los requisitos. Una vez que los bytecodes han pasado por el *Verifier*, se puede garantizar que no se violará ninguno de los requerimientos antes mencionados.

3. Cargador de Clases

El *Verifier* es lo que puede llamarse el último recurso, mientras que el *Class Loader* es un recurso de primera instancia.

Cuando se produce la carga de una nueva clase en el sistema, esta debe proceder de uno entre varios reinos. Actualmente existen tres posibles reinados: la computadora local, la red local y la red global (Internet). Cada uno de estos tres reinos es tratado en forma diferente por el cargador de clases. El cargador de clases nunca admite que una clase que provenga de un reino menos protegido reemplace a una clase que pertenece a uno más protegido. Además, las clases de un reinado no pueden invocar a los métodos de clases que están en otros ámbitos, a menos que dichas clases, hayan declarado explícitamente como públicos a dichos métodos.

En resumen, si se considera el ciclo de vida completo del método Java, éste comienza en la forma de código fuente en una determinada computadora, y luego puede viajar en la forma de un archivo de clases hasta cualquier sistema de archivos o red en cualquier parte del mundo. Cuando se ejecuta un *Applet* en un browser sensible a Java los bytecodes de los métodos son extraídos desde su archivo de clases y luego cuidadosamente escrutados por el *Verifier*. Una vez que son declarados seguros, el intérprete ya puede ejecutarlos.



Ciclo del método Java

MALENTENDIDOS ACERCA DEL LENGUAJE JAVA

1. Java es una extensión de HTML.

   Java es un lenguaje de programación, mientras que HTML es un lenguaje de descripción de páginas. No tienen nada en común, excepto que se necesitan extensiones HTML donde colocar los Applets Java, dentro de una página Web.

2. Tiene un ambiente de programación amigable.

   Java no posee un ambiente para programar. Para la construcción de programas en lenguaje Java pueden utilizarse productos como Visual Age para Java, JBuilder, entre otros.

3. Todos los programas Java corren dentro de una página Web.

   Todos los programas Java corren dentro de un browser, pero es posible y útil escribir programas stand alone, que corran independientemente de un browser. Estos programas, generalmente llamados aplicaciones, son completamente portables. Sólo toman el código y lo corren en otra máquina.

4. Java elimina la necesidad del scripting CGI

   No se elimina el uso de CGI. Con la tecnología de hoy en día CGI continua existiendo, pero no es el único camino de comunicación entre un Applet y el server,

5. Java se convertirá en un lenguaje de programación universal para todas las plataformas.

   Aún, las aplicaciones Java no se ven tan bien como las aplicaciones Windows desarrolladas por VB o MFC. En cualquier caso, las herramientas gráficas soportadas por Java son muy primitivas para hacer que el trabajo de diseño sea placentero.

6. Java es Interpretado, entonces es demasiado lento en una plataforma específica para aplicaciones serias

Varios programas gastan mucho de su tiempo en interacciones de interfaz de usuario. Es cierto que con la versión actual de Java, no haremos trabajos que requieran intensivamente la CPU a menos que tengamos que hacerlo. Sin embargo, es muy sencillo hacer un compilador que convierta los bytecodes de Java en código nativo.

7. Java revolucionará la computación Cliente-Servidor

   Esto es posible, pero a pesar de que en un entorno de desarrollo la comunicación funciona correctamente, se necesita utilizar un plug_in para el browser, pues aún éstos no soportan la comunicación vía RMI o Socket para aplicaciones Cliente-Servidor.

8. Con Java reemplazaré mi computadora por una Ampliación de Internet de $500

   Mucha gente está apostando a que ésto sucederá. Creemos que es absurdo pensar que la gente dejará de lado una computadora poderosa por una máquina limitada sin almacenamiento local. Podemos aceptar dicha Ampliación como una adición a la computadora y potenciar así a las aplicaciones Java.

LAS GARANTÍAS DE JAVA

Java es un lenguaje con un rico conjunto de garantías que, cuando se consideran como un todo, proveen un verdadero y avanzado sistema operativo para los programas. La mayoría de estas garantías están también disponibles en otros lenguajes, pero, hasta el momento ningún otro lenguaje, ni sistema operativo las ofrece a todas juntas.

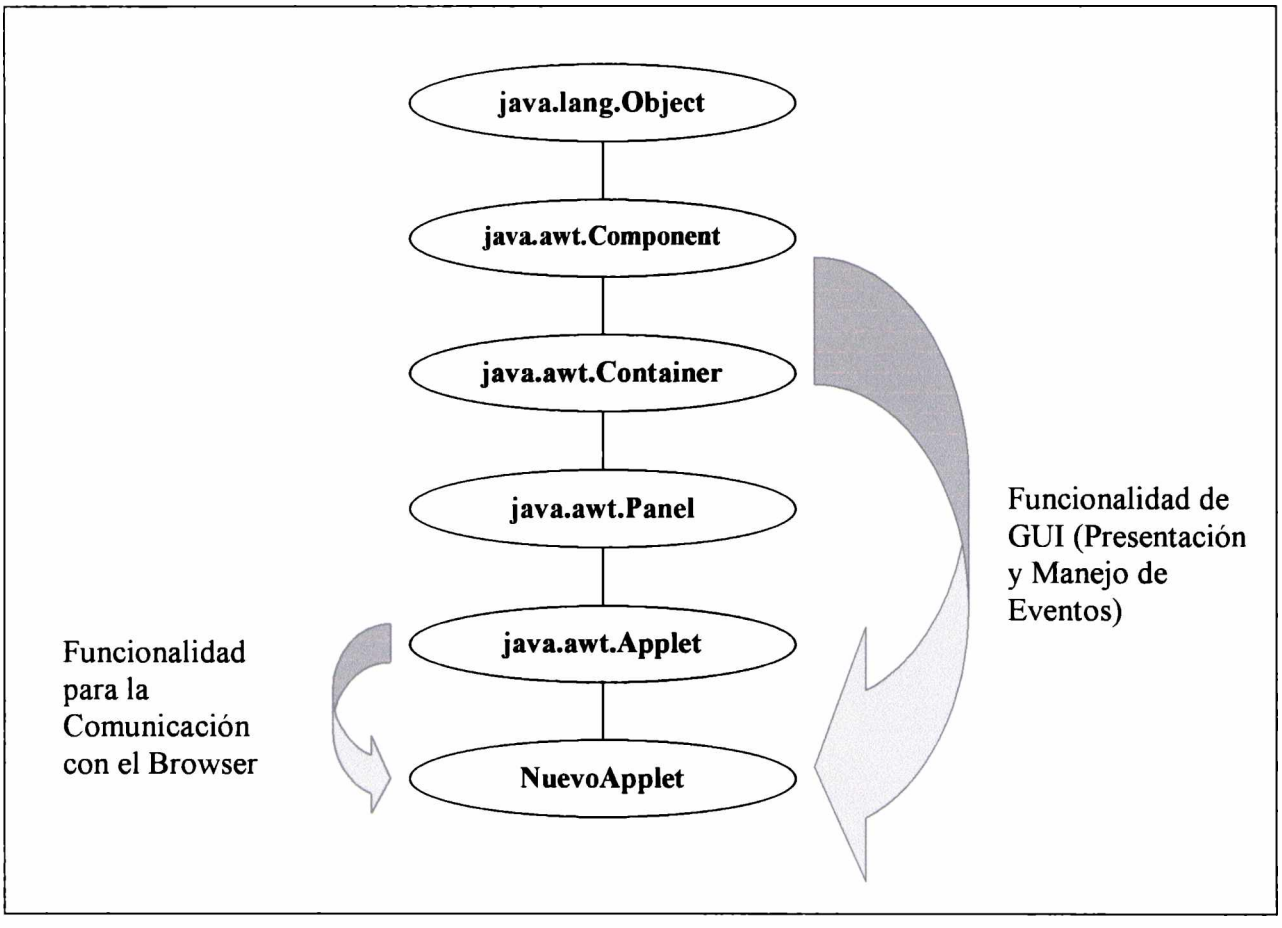
1. El código de Java es portable y correrá sin ninguna alteración en la mayoría de los sistemas operativos.
2. Java provee verdadero multihilado para los programadores, conjuntamente con métodos de sincronización.
3. Java ofrece clases de interfaces de usuarios independientes de la plataforma.
4. Java viene con capacidad de trabajo en red incorporada.
5. Todos los objetos Java saben como imprimirse.
6. El compilador Java impone el manejo de todos los errores y excepciones.
7. La memoria se limpia automáticamente.
8. Los programas escritos en forma de Applets pueden ser distribuidos a través de la red automáticamente.
9. Las Applets de Java funcionarán sin problemas en un Web Browser cliente sencillo sobre cualquier plataforma.

APPLETS JAVA

Un Applet es una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta en el lugar como parte de una página Web.

JERARQUÍA DE HERENCIA

Las applets se crean como una subclase de la Clase Applet.



COMPONENTES DE UN APPLET

El lenguaje Java implementa un modelo de programación orientado a objetos. Los objetos sirven de bloques centrales de construcción de los programas Java. También tiene, al igual que otros lenguajes de programación orientados a objetos, variables de instancia y métodos.

La estructura de un Applet es la siguiente:

```
/ * Sección de importaciones */
Public class NombreDelNuevoApplet extends Applet {
/* Declaración de las variables de estado y métodos */
/* Declaración de métodos para la interacción con los objetos */
Public void MétodoUno (parámetros) {
/* Código Java que desempeña la tarea */
                                                    }
                                                }
```

MÉTODOS DE LA CLASE APPLET

Para escribir Applets Java hay que utilizar una serie de métodos. Son los que se utilizan para iniciar y detener la ejecución del applet, para pintar y actualizar la pantalla y para capturar la información que se pasa al applet desde el archivo html a través del tag APPLET.

Invocados automáticamente por el Browser

- **init()**

Esta función se invoca al crearse el applet. Se invoca sólo una vez, cuando el applet es cargado en el sistema. Las clases derivadas deben definir este método para cambiar el tamaño durante su inicialización, y cualquier otra inicialización de los datos que solamente deba realizarse una vez.

Debería realizarse, por ejemplo, la carga de imágenes y sonido, el resize del applet para que tenga su tamaño correcto y la asignación de valores a las variables globales.

Ejemplo:

```
public void init() {
   if( width < 200  ||  height < 200 )
      resize( 200,200 );
   valor_global1 = 0;
   valor_global2 = 100;
   // carga de imágenes en memoria sin mostrarlas
   // carga de música de fondo en memoria sin reproducirla     }
```

- **start()**

Llamada para activar el applet. Esta función es invocada luego del init y cada vez que la página que contiene el applet es visitada. La Clase Applet no hace nada en este método, pero las clases derivadas deberán sobrecargarlo para comenzar una animación, sonido, etc.

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

- **stop()**

Llamada para detener el applet. Se invoca cuando el applet desaparece de la pantalla, es decir cuando se abandona la página html que lo contiene. La Clase Applet no hace nada en este método. Las clases derivadas deberán sobrecargarlo para detener la animación, el sonido, etc.

```
public void stop() {
   estaDetenido = true;
   if( /* ¿se está reproduciendo música? */ )
      musicClip.stop();
   }
```

- **destroy()**

Este método es invocado cuando el applet no se va a usar. Se liberan los recursos del sistema que tiene alocados. La Clase Applet no hace nada en este método. Las clases derivadas deberían cargarlo para hacer una limpieza final. Por ejemplo, los Applet

Multithread deberán usar destroy() para "matar" cualquier thread del applet que quedase activo.

Otros métodos

- **resize (int width, int height)**

El método init() debería llamar a esta función para establecer el tamaño del applet. Cambiar el tamaño en otro sitio que no sea init() produce un reformateo de todo el documento y no se recomienda.

En el browser Netscape, el tamaño del applet es el que se indica con el tag APPLET del HTML, prevaleciendo éste al que se indique desde el código Java del applet.

- **width**

Variable entera, su valor es el ancho definido en el parámetro WIDTH del tag APPLET del HTML.

- **height**

Variable entera, su valor es la altura definida en el parámetro HEIGHT del tag APPLET en el HTML. Por defecto, al igual que el ancho, es la altura del ícono. Tanto width como height están siempre disponibles para que se pueda chequear el tamaño del applet.

- **paint(Graphics g)**

Se llama cada vez que se necesita refrescar el área de dibujo del applet. La Clase Applet simplemente dibuja una caja con sombreado de tres dimensiones en el área.

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

- **update(Graphics g)**

Esta es la función que se llama cuando se necesita actualizar la pantalla. La clase Applet simplemente limpia el área y llama al método paint().

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

```
public void update( Graphics g ) {
  if( estaActualizado )
    {
    g.clear();  // garantiza la pantalla limpia
    repaint();  // podemos usar el método padre: super.update()
    }
  else
    // Información adicional
    g.drawString( "Otra información",25,50 );
    }
```

- **repaint()**

A esta función se la debería llamar cuando el applet necesite ser repintado.

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

- **getParameter(String attr)**

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

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

- **getDocumentBase()**

Indica la ruta http, o el directorio del disco, de donde se ha recogido la página HTML que contiene el applet.

- **getCodeBase()**

Indica la ruta http, o el directorio del disco, de donde se ha cargado el código bytecode que forma el applet.

- **print(Graphics g)**

Imprime el mapa de bits del dibujo.

ATRIBUTOS DE LOS APPLETS

Dado que los applets están destinados a ejecutarse en navegadores Web, deben estar contenidos en páginas HTML. El esquema de tags de HTML hace fácil la adición de una marca que permite la ejecución de programas Java en ellos.

La sintaxis de las etiquetas <APPLET> y <PARAM> es la siguiente:

```
<APPLET CODE= WIDTH= HEIGTH= [CODEBASE=] [ALT=]
[NAME=] [ALIGN=] [VSPACE=] [HSPACE=]>
<PARAM NAME= VALUE= >
</APPLET>
```

En cuanto a los atributos que acompañan a la etiqueta <APPLET>, algunos son obligatorios y otros son opcionales. Todos los atributos, siguiendo la sintaxis de html, se especifican de la forma: atributo=valor.

Atributos Obligatorios

- **CODE**

Es el nombre de la clase principal, la clase ejecutable, que tiene la extensión .class. No se permite un URL absoluto, aunque sí puede ser relativo al atributo opcional CODEBASE.

- **WIDTH**

Indica el ancho inicial en pixels que el navegador debe reservar para el applet.

- **HEIGHT**

Indica la altura inicial en pixels. Un applet que disponga de una geometría fija no se verá redimensionado por estos atributos. Por este motivo, si los atributos

definen una zona menor que la que el applet utiliza, únicamente se verá parte del mismo, como si se visualizará a través de una ventana, eso sí, sin ningún tipo de desplazamiento.

Atributos Opcionales

- **CODEBASE**

  Se emplea para utilizar el URL base del applet. En caso de no especificarse, se utilizará el mismo que tiene el documento.

- **ALT**

  Muestra un texto alternativo en caso de que el navegador no implemente una Máquina Virtual Java.

- **NAME**

  Otorga un nombre a esta instancia del applet en la página. Un applet puede ser cargado varias veces en la misma página tomando un alias distinto en cada momento.

- **ALIGN**

  Se emplea para alinear el applet. Puede tomar los siguientes valores: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM y ABSBOTTOM.

- **VSPACE**

  Indica el espacio vertical, en pixels, entre el applet y el texto. Sólo funciona cuando se ha indicado ALIGN = LEFT o RIGHT.

- **HSPACE**

  Funciona igual que el anterior pero indicando espaciamiento horizontal. Sólo funciona cuando se ha indicado ALIGN = LEFT o RIGHT.

PASAJE DE PARÁMETROS A APPLETS

El espacio que queda entre las marcas de apertura y cierre de la definición de un applet, se utiliza para definir los parámetros que recibe el mismo. Para indicarlos se utiliza la marca PARAM en la página HTML y el método getParameter() de la clase java.applet.Applet para leerlos en el código interno del applet.

Los atributos que acompañan a la marca PARAM son los siguientes:

- **NAME:** Nombre del parámetro que se desea pasar al applet.
- **VALUE:** Valor que se desea transmitir en el parámetro que se ha indicado antes.
- **Texto HTML:** Es el texto HTML que será interpretado por los navegadores que no entienden la marca APPLET en sustitución del applet mismo.

Los parámetros no se limitan a uno solo. Se puede pasar al applet cualquier número de parámetros y siempre hay que indicar un nombre y un valor para cada uno de ellos.

El método getParameter() es fácil de entender. El único argumento que necesita es el nombre del parámetro cuyo valor queremos recuperar. Todos los parámetros se pasan como Strings, en caso de necesitar pasarle al applet un valor entero, se ha de pasar como String, recuperarlo como tal y luego convertirlo al tipo deseado. Tanto el argumento de NAME como el de VALUE deben ir colocados entre comillas dobles.

VISUALIZACIÓN DE APPLETS

Para ver un applet en funcionamiento debemos utilizar un navegador World Wide Web como HotJava, Microsoft Explorer o Netscape.

Sun nos ofrece un visualizador de applets denominado Appletviewer. Este representa el mínimo espacio de navegación, incluyendo un área gráfica, donde se ejecutará el applet.

La llamada a appletviewer es de la forma: appletviewer [-debug] urls...

Tanto el navegador World Wide Web, como el appletviewer toman como parámetro de ejecución, o bien el nombre de un archivo html conteniendo el tag <APPLET>, o bien un URL hacia un archivo HTML que contenga esa marca.

La única opción válida que admite la llamada a appletviewer es -debug, que arranca el applet en el depurador de Java. Para poder ver el código fuente en el depurador, se tiene que compilar el archivo .java con la opción -g.

CICLO DE VIDA DE UN APPLET

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

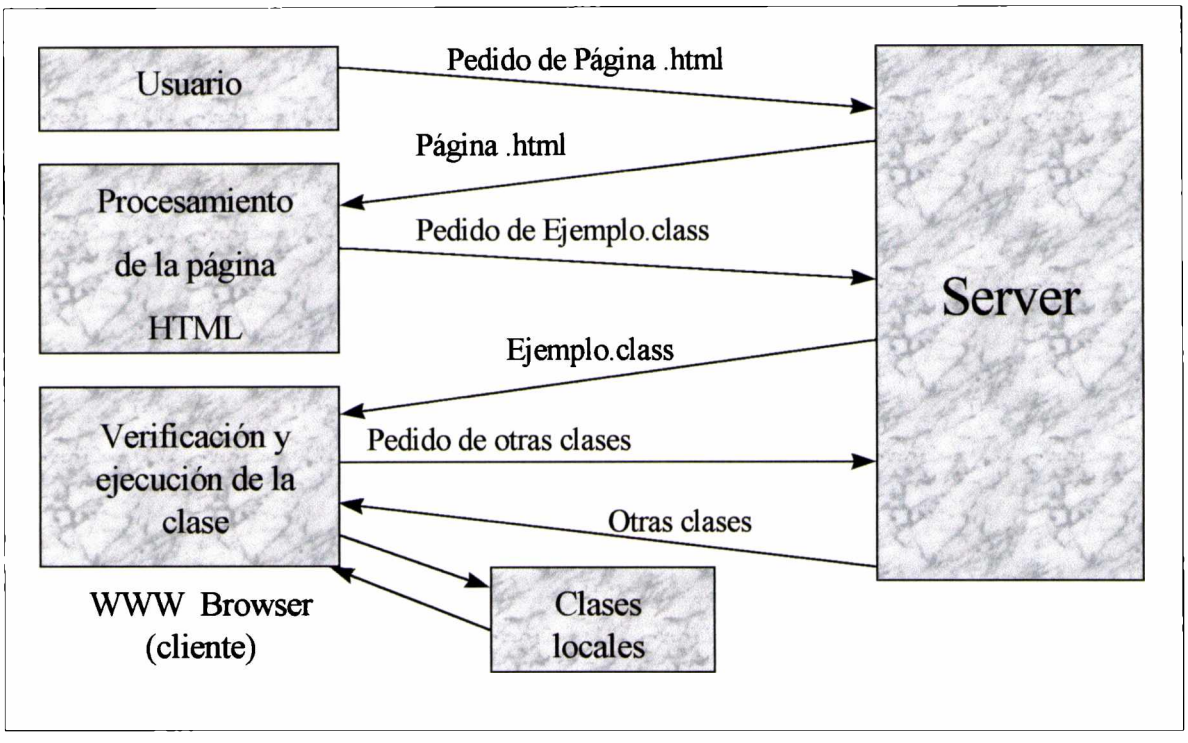
1. Se crea una instancia de la clase que controla el applet.
2. El applet se inicializa a si mismo.
3. Comienza la ejecutación del applet.
4. El applet empieza a recibir llamadas. Primero recibe una llamada init, seguida de un mensaje de start y paint. Estas llamadas pueden ser recibidas asincrónicamente.

Cuando se abandona la página, el applet detiene la ejecución. Cuando se regresa a la página que contiene el applet, se reanuda la ejecución del mismo.

Si se utiliza la opción del navegador de Reload, el applet es descargado y vuelto a cargar. El applet libera todos los recursos capturados, detiene su ejecución y ejecuta su finalizador para realizar un proceso de limpieza final de sus trazas. Después de esto, el applet se descarga de la memoria y vuelve a cargarse volviendo a comenzar su inicialización.

Finalmente, cuando se concluye la ejecución del navegador o de la aplicación que está mostrando el applet, se detiene la ejecución del mismo y se libera toda la memoria y los recursos ocupados por el applet.

UN APPLET EN FUNCIONAMIENTO



EXCEPCIONES

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, a la detección y corrección de errores. Si hay un error, la aplicación no debería morirse sino que se debería lanzar una excepción que debe ser capturada para resolver la situación de error. Utilizadas en forma adecuada, las excepciones aumentan la robustez de las aplicaciones, y por lo tanto la robustez del lenguaje.

GENERAR EXCEPCIONES EN JAVA

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

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

Se pueden definir excepciones extendiendo la clase Exception.

También pueden producirse excepciones en forma implícita cuando se realiza alguna acción ilegal o no válida.

Por lo tanto, las excepciones, pueden originarse de dos modos: el programa hace algo ilegal, o el programa explícitamente genera una excepción ejecutando la sentencia throw.

La sentencia throw tiene la siguiente forma:

```
throw ObjetoException;
```

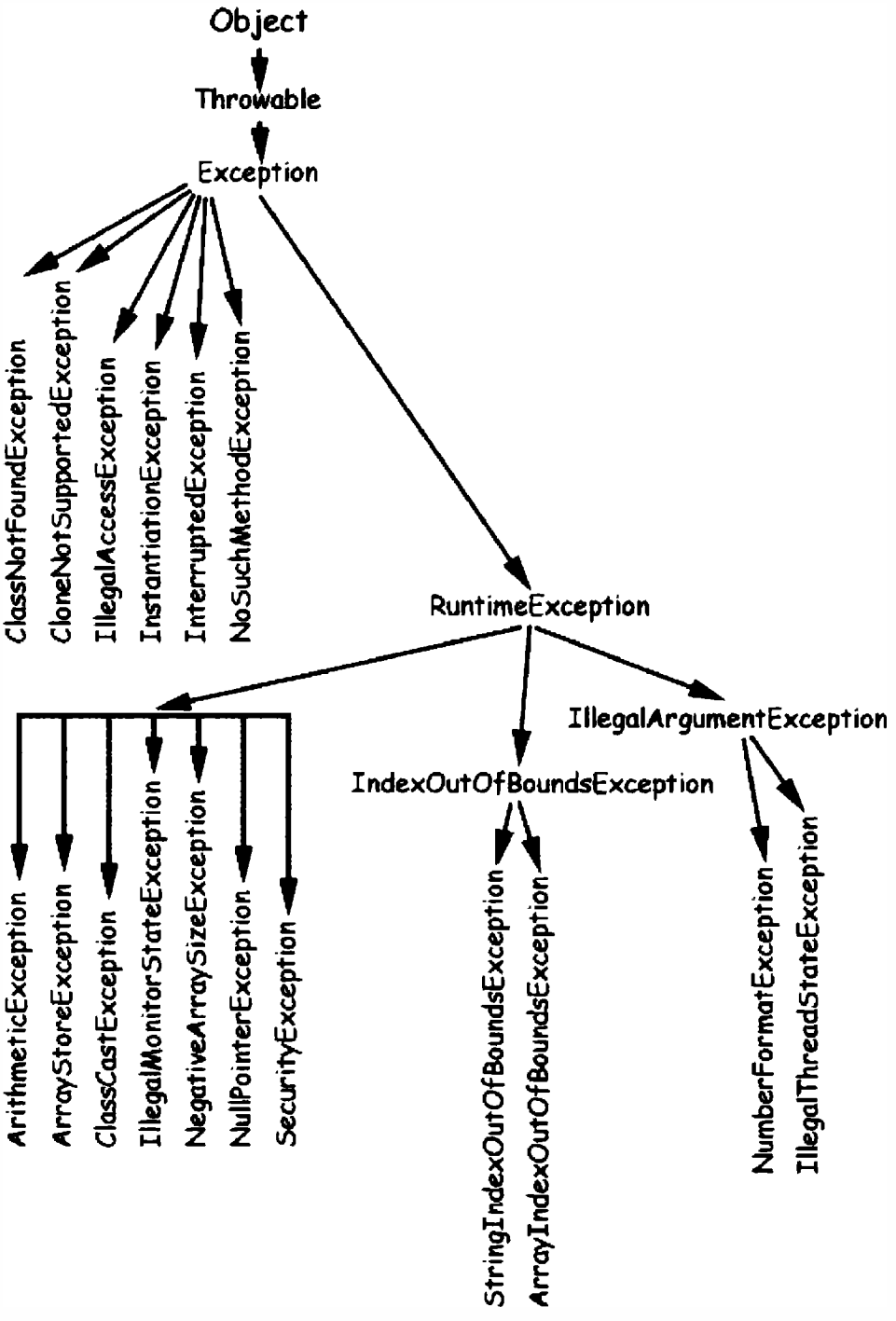
El objeto ObjetoException es un objeto de una clase que extiende la clase Exception.

El siguiente código de ejemplo origina una excepción de división por cero:

```
class melon {
  public static void main( String[] a ) {
    int i=0, j=0, k;
    k = i/j;    // Origina un error de division-por-zero
    }
  }
```

EXCEPCIONES PREDEFINIDAS

Las excepciones predefinidas se conocen como Excepciones Runtime. Su jerarquía de clases es la que se muestra el siguiente gráfico:



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

- **ArithmeticException**

  Una excepción aritmética típica es el resultado de una división por cero:

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

- **NullPointerException**

  Se produce cuando se intenta acceder a una variable o a un método antes de haberlo definido.

- **IncompatibleClassChangeException**

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

- **ClassCastException**

  Se produce cuando se intenta convertir un objeto a otra clase que no es válida.

  ```
  y = (Prueba)x;     // donde x no es de tipo Prueba
  ```

- **NegativeArraySizeException**

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

- **OutOfMemoryException**

  Se produce si se intenta crear un objeto con el operador new y no hay memoria suficiente. No debería producirse, pues el Garbage Collector se encarga de liberarla.

- **NoClassDefFoundException**

  Se produce si se realiza una referencia a una clase que el sistema no encuentra.

- **ArrayIndexOutOfBoundsException**

  Se genera al intentar acceder a un elemento de un array que excede los límites definidos inicialmente para el mismo.

- **UnsatisfiedLinkException**

  Si se hace el intento de acceder a un método nativo que no existe. Aquí no existe un método a.kk y se llama a a.kk(), cuando debería llamar a A.kk().

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

- **InternalException**

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

CREAR EXCEPCIONES PROPIAS

Se pueden lanzar excepciones propias, extendiendo la clase System.exception. Por ejemplo, si considerando un programa cliente/servidor, el código cliente se intenta conectar al servidor, y durante 5 segundos espera una respuesta. Si el servidor no responde, el servidor lanzaría la excepción de time-out:

```
class ServerTimeOutException extends Exception {}
public void conectame( String nombreServidor ) throws Exception {
   int exito;
   int puerto = 80;
   exito = open( nombreServidor,puerto );
   if( exito == -1 )
      throw ServerTimeOutException;
   }
```

Si se quieren capturar las excepciones propias, se deberá utilizar la sentencia try:

```
public void encuentraServidor() {

  try {
     conectame( servidorDefecto );
     catch( ServerTimeOutException e ) {
        g.drawString(
           "Time-out del Servidor, intentando alternativa",
           5,5 );
        conectame( servidorAlterno );
        }

  }
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interfaz del método.

CAPTURAR EXCEPCIONES

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

```
int valor;
try {
   for( x=0,valor = 100; x < 100; x ++ )
      valor /= x;
   }
catch( ArithmeticException e ) {
   System.out.println( "Matemáticas locas!" );
   }
catch( Exception e ) {
   System.out.println( "Se ha producido un error" );
   }
```

- **try**

Es el bloque de código donde se prevee que se genere una excepción. Este bloque try tiene que ir seguido por una cláusula catch o una cláusula finally

- **catch**

Es el código que se ejecuta cuando se produce la excepción. En este bloque no se debe colocar código que genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula:

```
catch( Excepción e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

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

Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
```

```
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}

  try {
    if( temp > 40 )
      throw( new demasiadoCalor() );
    if( dormir < 8 )
      throw( new demasiado Cansado() );
  } catch( Limites lim ) {
    if( lim instanceof demasiadoCalor )
      {
      System.out.println( "Capturada excesivo calor!" );
      return;
      }
    if( lim instanceof demasiadoCansado )
      {
      System.out.println( "Capturada excesivo cansancio!" );
      return;
      }
  } finally
    System.out.println( "En la clausula finally" );
```

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

- **finally**

  Es el bloque de código que se ejecuta siempre, haya o no excepción.

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

Cuando se trata una excepción, se plantea el problema de qué acciones seguir. En la mayoría de los casos, bastará con presentar un mensaje de error al usuario y que él decida si quiere o no continuar con la ejecución del programa.

PROPAGACIÓN DE EXCEPCIONES

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

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecuta el código de la cláusula finally (en caso de que la haya).

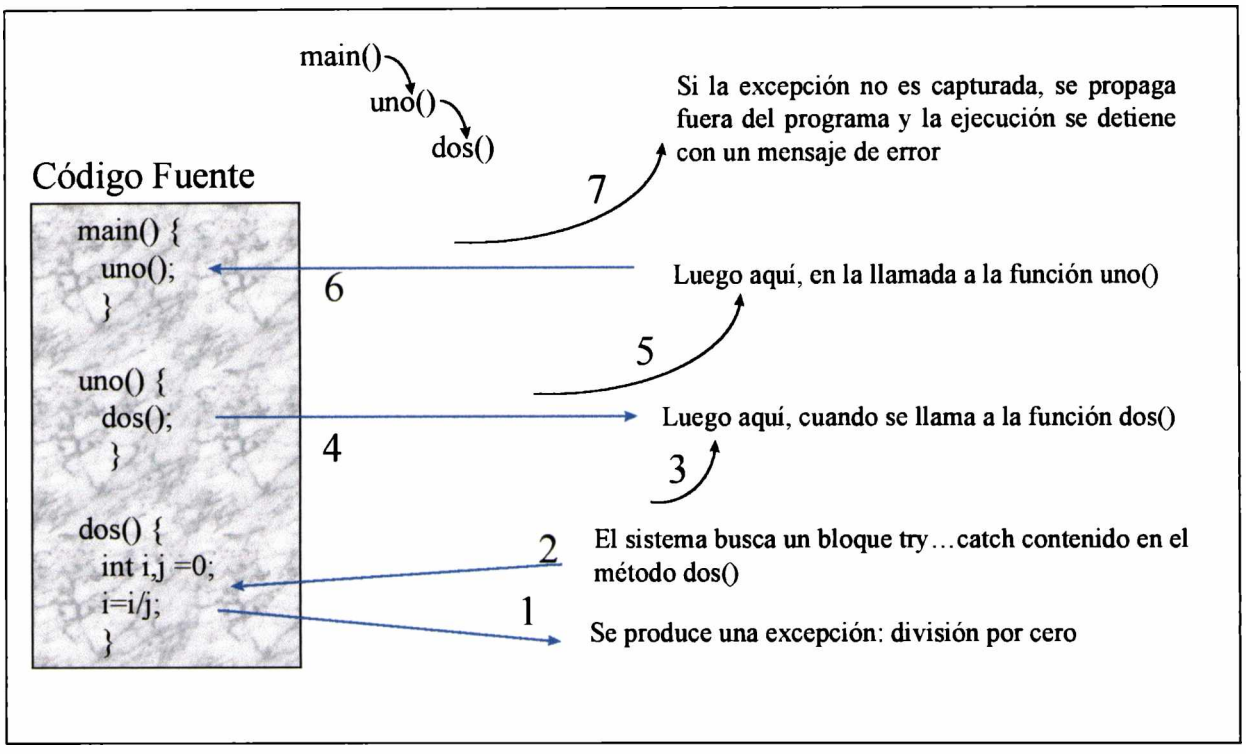
El flujo de control abandona este método y retorna al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia try, entonces se vuelve a intentar el control de la excepción.

Si una excepción no es tratada en la rutina en donde se produce, el sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo invocó. Si la excepción se propaga hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, se puede suponer que Java proporciona un bloque catch por defecto, que imprime un mensaje de error y sale.

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

Por lo tanto un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada. Esto debe ser así para que cualquiera que escriba una llamada a ese método prevea la llegada de una excepción en lugar del valor de retorno normal. Esto permite al programador que invoca a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas.

En el siguiente diagrama se muestra cómo se propaga la excepción que se genera en el código, a través de la pila de llamadas durante la ejecución del mismo.



Cuando se crea una nueva excepción, derivada de una clase Exception ya existente, se puede cambiar el mensaje que lleva asociado.

Con todo el manejo de excepciones se puede concluir que proporciona un método más seguro para el control de errores, además representa una excelente herramienta para organizar en sitios concretos todo el manejo de los mismos.

La degradación que se produce en la ejecución de programas con manejo de excepciones está compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.

THREADS

FLUJO EN PROGRAMAS

Programas de flujo único

Un programa de flujo único o mono-hilvanado utiliza un único flujo para controlar su ejecución. Muchos de los applets y aplicaciones son de flujo único sin especificarlo explícitamente.

Programas de flujo múltiple

Java posibilita la creación y control de threads explícitamente. La simplicidad para crear, configurar y ejecutar threads, permite que se puedan implementar poderosos applets.

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

CREACIÓN Y CONTROL DE THREADS

Creación de un Thread

Hay dos formas de obtener threads en Java. Una es implementando la interfaz Runnable y la otra es extender la clase Thread.

La implementación de la interfaz Runnable es la forma habitual de crear threads. La interfaz define el trabajo y las clases que implementan la interfaz realizan ese trabajo. Los diferentes grupos de clases que implementen la interfaz tendrán que seguir las mismas reglas de funcionamiento.

Extendiendo la clase Thread, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Pero esta limitación de Java puede ser superada a través de la implementación de Runnable:

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

En este caso necesitamos crear una instancia de Thread para que el sistema pueda ejecutar el proceso como un thread. Además, el método abstracto run() que está definido en la interfaz Runnable tiene que ser implementado.

Arranque de un Thread

Las aplicaciones ejecutan main() tras arrancar. Esta es la razón por la cual main() es el lugar natural para crear y arrancar otros threads.

La línea de código:

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

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

Al tener control directo sobre los threads, tenemos que arrancarlos explícitamente.

Manipulación de un Thread

Luego de la creación del thread t1, este es controlado a través del método run().

```
t1.start()
```

Una vez dentro de run(), podemos comenzar las sentencias de ejecución como en otros programas. El método run() sirve como rutina main() para los threads; cuando run() termina, también lo hace el thread. Todo lo que se espera que el thread realice, debe estar dentro de run(), por eso cuando se dice que un método es Runnable, se debe escribir un método run().

El método sleep() simplemente le dice al thread que duerma durante los milisegundos especificados. Se debería utilizar sleep() cuando se pretende retrasar la ejecución del thread. El método sleep() no consume recursos del sistema mientras el thread duerme. De esta forma otros threads pueden seguir funcionando.

```
t1.sleep( retardo );
```

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un thread sin marcar un límite de tiempo. Para este tipo de control se puede utilizar el método suspend().

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

Este método no detiene la ejecución permanentemente. El thread es suspendido y para volver a activarlo se invoca al método resume():

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

Parada de un Thread

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

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

Esta llamada no destruye el thread, sino que detiene su ejecución. La ejecución ya no se puede reanudar con t1.start(). Cuando se liberan las variables que se usan en el thread, el objeto thread (creado con new) queda marcado para ser eliminado por el Garbage Collector.

Los programas más complejos necesitan un control sobre cada uno de los threads que lanzan, el método stop() puede utilizarse para esto.

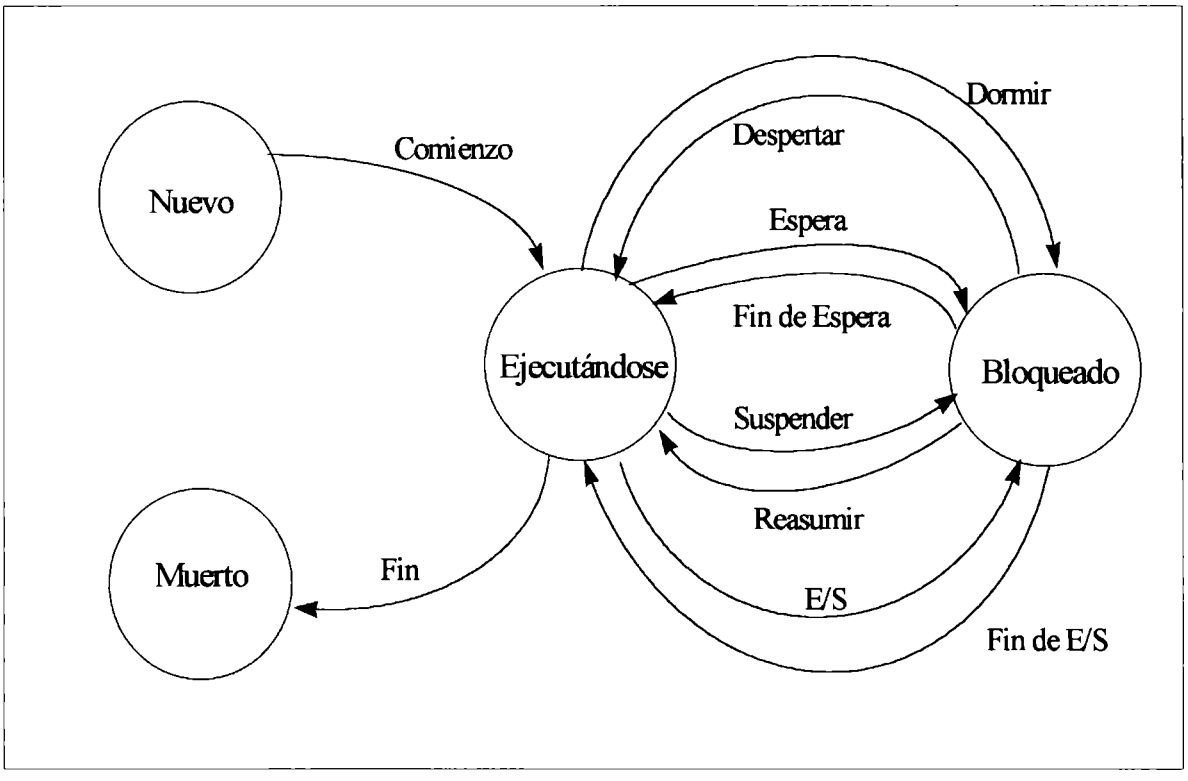
Para comprobar si un thread está vivo o no, exite el método isAlive(). Se considera vivo a un thread que ha comenzado y no ha finalizado.

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

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

ESTADOS DE UN THREAD

Durante el ciclo de vida de un thread, éste se puede encontrar en diferentes estados. En el siguiente gráfico se muestran estos estados y los métodos que provocan el paso de un estado a otro.



Nuevo Thread

La siguiente sentencia crea un nuevo thread pero no lo arranca, lo deja en el estado de "Nuevo Thread":

```
Thread MiThread = new MiClaseThread();
```

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

Ejecutable

En el siguiente código, la llamada al método start() creará los recursos del sistema necesarios para que el thread pueda ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del thread.

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

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

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

Bloqueado

El thread entra en estado "Bloqueado" cuando se invoca al método suspend(), cuando se llama al método sleep(), cuando el thread está bloqueado en un proceso de entrada/salida o cuando el thread utiliza su método wait() para esperar a que se cumpla una determinada condición.

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

Los métodos de recuperación del estado Ejecutable, en función de la forma en que han entrado en estado Bloqueado, son los siguientes:

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

Muerto

Un thread se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un thread muere normalmente cuando concluye de forma habitual su método run().

El método stop() envía un objeto ThreadDeath al thread que quiere detener. El thread morirá en el momento en que reciba la excepción ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus threads cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

El método isAlive()

La interfaz de programación de la clase Thread incluye el método isAlive(), que devuelve true si el thread ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive() devuelve false, se tiene "Nuevo Thread" o un thread "Muerto". Si devuelve true, el thread se encuentra en estado "Ejecutable" o "Bloqueado".

SCHEDULING

Java tiene un Scheduler, una lista de procesos, que monitoriza todos los threads que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los threads que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del thread; la otra, es el indicador de demonio. La regla básica del Scheduler es que si solamente hay threads demonio ejecutándose, la Máquina Virtual Java (JVM) concluirá. Los nuevos threads heredan la prioridad y el indicador de demonio de los threads que los han creado. El Scheduler determina qué threads deberán ejecutarse comprobando la prioridad de todos los threads, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El Scheduler puede seguir dos patrones, preemptivo y no-preemptivo. Los schedulers preemptivos proporcionan un segmento de tiempo a todos los threads que están corriendo en el sistema. El Scheduler decide cual será el siguiente thread a ejecutarse y llama a resume() para darle vida durante un período fijo de tiempo. Cuando el thread ha estado en ejecución ese período de tiempo, se llama a suspend() y el siguiente thread en la lista de procesos será relanzado (resume()). Los schedulers no-preemptivos deciden que thread debe correr y lo ejecutan hasta que concluye. El thread tiene control total sobre el sistema mientras esté en ejecución. El método yield() es la forma en que un thread fuerza al Scheduler a comenzar la ejecución de otro thread que esté esperando. Dependiendo del sistema en que esté corriendo Java, el Scheduler será preemptivo o no-preemptivo.

AWT - ABSTRACT WINDOW TOOLKIT

INTRODUCCIÓN AL AWT

AWT es una biblioteca de clases Java para el desarrollo de Interfaces de Usuario Gráficas (GUI). El entorno que ofrece es demasiado simple, no se han tenido en cuenta las ideas de entornos gráficos novedosos, sino que se ha ahondado en estructuras orientadas a eventos, sin soporte para la construcción gráfica. Quizá la presión de tener que lanzar algo al mercado haya tenido mucho que ver en la pobreza de AWT.

La estructura básica del AWT se basa en Componentes y Contenedores. Estos últimos pueden contener Componentes y son Componentes a su vez, de forma que los eventos pueden tratarse tanto en Contenedores como en Componentes. Es responsabilidad del programador el ensamble de todas las piezas, y la seguridad del tratamiento de los eventos.

INTERFACES DE USUARIO

Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas que proporcionan las aplicaciones más modernas.

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

En el nivel más bajo, el sistema operativo transmite información desde el mouse y el teclado como dispositivos de entrada al programa. El AWT fue diseñado de forma tal que el programador pueda abstraerse de detalles tales como controlar el movimiento del mouse o leer el teclado. El AWT constituye una librería de clases orientadas a objetos para cubrir estos recursos y servicios de bajo nivel.

Debido a que el lenguaje de programación Java es independiente de la plataforma en que se ejecutan sus aplicaciones, el AWT también es independiente de la plataforma en que se ejecute. Los elementos de interfaz proporcionados por el AWT están

implementados utilizando toolkits nativos de las plataformas, preservando una apariencia semejante a todas las aplicaciones que se ejecutan en dicha plataforma.

Peers

Son clases que manejan las implementaciones específicas de cada plataforma. Permiten que cada componente conserve el mismo look and feel del ambiente en que se ejecuta la aplicación (Windows, MAC, etc.).

Cada componente AWT tiene una componente peer nativa equivalente.Por ejemplo, un objeto Button tiene asociado un objeto ButtonPeer que se instancia en cada plataforma. Las componentes AWT delegan a los peers gran parte de la funcionalidad asociada a su apariencia.

ESTRUCTURA DEL AWT

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

- Los Contenedores contienen Componentes, que son los controles básicos.
- Las Componetes son clases que definen objetos de GUI's (botones, labels, etc).
- A través de los Layouts se posicionan los objetos dentro de los Contenedores.
- Provee un nivel alto de abstracción respecto al entorno de ventanas en que se ejecute la aplicación.
- La arquitectura de la aplicación es dependiente del entorno de ventanas. No tiene un tamaño fijo.
- La clase Graphic permite definir métodos para realizar operaciones gráficas sobre componentes.

COMPONENTES Y CONTENEDORES

Una interfaz gráfica está construida basándose en elementos gráficos básicos, los Componentes. Típicos ejemplos de estos Componentes son los botones, las barras de desplazamiento, etiquetas, listas, campos de texto, etc. Los Componentes permiten al usuario interactuar con la aplicación. En el AWT, todos los Componentes de la interfaz de usuario son instancias de la clase Component o de una de sus subclases.

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 ícono de la aplicación. En el AWT, todos los Contenedores son instancias de la clase Container o una de sus subclases.

COMPONENTES

Component es una clase abstracta que tiene asociada una posición, un tamaño, un peer, un formato de letra, color, y además responden a eventos.

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

Los Objetos derivados de la clase Component son los que aparecen a continuación:

- Button
- Canvas
- Checkbox
- Choice
- Container
  - Panel
  - Window
    - Dialog
    - Frame
- Label
- List
- Scrollbar
- TextComponent
  - TextArea
  - TextField

JFC - JAVA FOUNDATION CLASSES

LA LIBRERÍA JFC

Contiene las componentes Swing y las APIs que se mencionan a continuación:

♣ Java 2D: Conjunto de Clases de Gráficos e Imágenes 2D. Además ofrece diferentes estilos de dibujos, mecanismos para definir formas complejas, etc.

♣ Drag and Drop: Posibilita la transferencia de información entre Java y Aplicaciones nativas y dentro de los programas JAVA.

♣ Accessibility: Estas clases son compatibles con Hardware y Software diseñados para usuarios con necesidades especiales, tales como vista limitada o incapacidad para operar con el mouse.

Accessibility es una característica muy importante de los Swing, ya que es el único conjunto de componentes de lenguaje Java que puede ser utilizado para escribir páginas Web que sean accesibles para gente discapacitada.

Además, también pueden ser útiles para gente que no posea discapacidades, ya que por ejemplo permite crear programas para "touch screen".

INTRODUCCIÓN A LAS CLASES SWING

El conjunto de componentes Swing es una herramienta de Interface Gráfica de Usuario (GUI) que simplifica el desarrollo de elementos visuales tales como menúes, barras, diálogos, etc.

Con las Swing se pueden construir GUIs que posean un determinado look & feel, el cual es definido por el programador, e independientemente de la plataforma en la cual se ejecute el programa, la apariencia definida no cambia.

Además las componentes Swing, se definen como "lightweight", ya que no se apoyan en el código de interface del sistema operativo subyacente. Consecuentemente pueden ser implementadas usando menos código del que requieren las componentes "heavyweight".

LAS TRES PARTES DE LAS SWING

Las Swing están implementadas utilizando tres elementos principales que son parte de las JFC.

- Una nueva rama de componentes que desciende de JComponent. Ésta clase desciende de la clase contenedora de AWT y es la raíz de casi todos los componentes Swing. Esta jerarquía permite a los componentes contener otros componentes.
- Un conjunto de clases de soporte que no crean componentes visibles, pero proveen servicios vitales a las APIs Swing y a las Aplicaciones que se desarrollen con éstas.
- Un nuevo conjunto de Interfaces Swing relacionadas y que están implementadas por las componentes Swing y las clases de soporte.

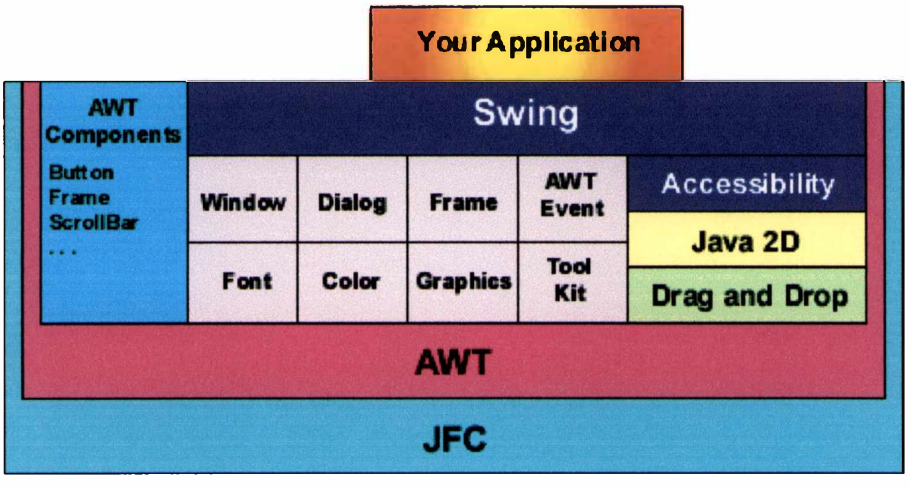
PAQUETES INCLUIDOS EN LAS SWING

- Accessibility: Permiten construir programas para gente con discapacidades fisicas.
- Swing: Implementa casi todas las clases componentes utilizadas en swing.
- Basic: Este paquete define el look & feel predeterminado de los componentes swing. Extendiendo este paquete se puede redefinir el look & feel.
- BeanInfo: Define la clase SwingBeanInfo que es la Superclase de todas las clases BeanInfo swing.
- Border: Permite dibujar distintos tipos de bordes alrededor de los componentes.
- Event: Define las clases específicas de eventos.
- Multi: Provee clases que permiten a los componentes tener Interfaces con soporte de herencia múltiple.
- Look & Feel: Contiene clases que las swing utilizan para permitir que sus componentes luzcan de diferentes maneras.

- Table: A través de este paquete se provee el manejo de tablas.
- Text: Contiene clases e interfaces para el manejo de componentes con texto.
- RTF: Implementa un editor de texto para el manejo de archivos RTF (Rich Text Format).
- Tree: A través de este paquete se provee el manejo de árboles.
- Undo: Permite deshacer acciones realizadas en componentes de edición de texto.

MODELO DE ARQUITECTURA SWING

Desde el punto vista de la arquitectura, el conjunto de componentes Swing, extiende, pero no reemplaza a las componentes AWT.



El diagrama muestra como las Swing se sitúan sobre un número de APIs que implementan las diferentes partes de las Java Foundation Classes, incluyendo Java 2D y Drag & Drop. A pesar de que éstas últimas forman parte de la JFC, no se incluyen en las Swing. Esto se debe a que varias de las tareas que utilizan requieren uso de código nativo de la máquina y se sabe que las Swing nunca lo utilizan. Por esto, en el diagrama, las Java 2D y Drag & Drop no aparecen dentro del rectángulo titulado Swing.

SWING Y AWT

Tanto AWT como Swing son componentes lightweight, pero las clases Swing tienen más objetos, más propiedades y funcionalidad que las componentes AWT. Por esto en la mayoría de los casos se utilizan componentes Swing.

Por defecto las componentes Swing toman el Look & Feel de la plataforma donde corren, cuando se crea un programa y se corre en entorno Windows, éste tiene la apariencia y el comportamiento de un programa escrito para Windows. Cuando se ejecuta el mismo programa en entorno Unix, éste se comporta como un programa escrito para Unix, y así sucede con el resto de los Sistemas Operativos. Esto puede salvarse usando el Look & Feel de las componentes Swing, ya que a diferencia de las AWT, la aplicación puede presentar siempre la misma apariencia sobre los diferentes entornos. Esto es posible gracias a que los Swing no utilizan ningún código de las plataformas sobre las cuales corren. Estas presentan un conjunto de objetos con el mismo comportamiento y la misma apariencia, sin tener en cuenta el Sistema Operativo subyacente.

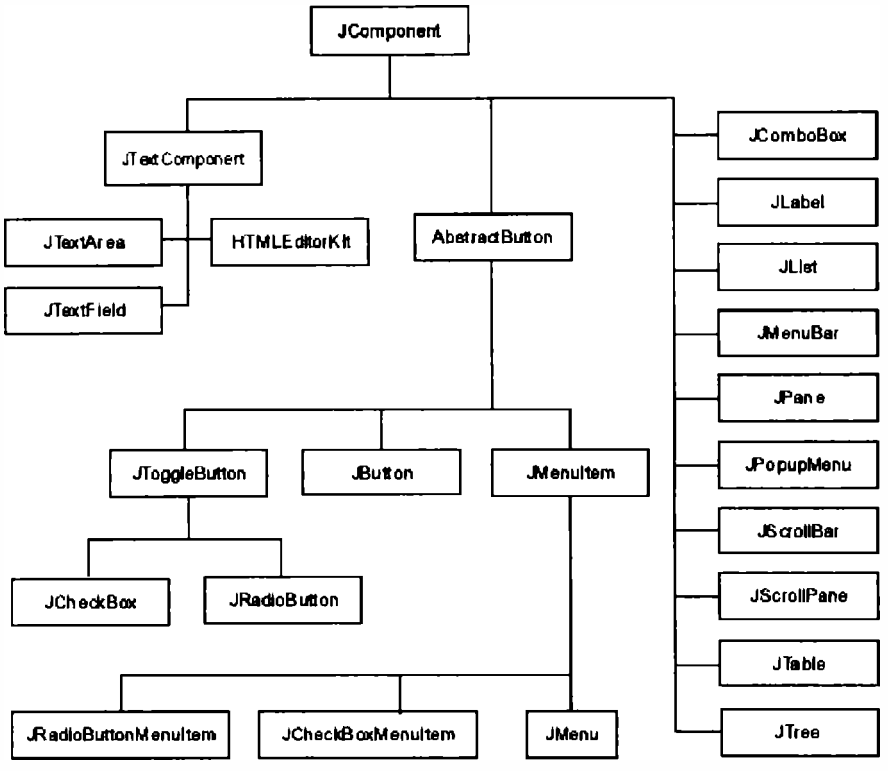
Por lo tanto, utilizando componentes Swing los desarrolladores pueden crear su propio conjunto de objetos con la apariencia que deseen.

Jerarquía de herencias

La jerarquía de las componentes Swing es similar a la de las componentes AWT, pero si se examinan con detenimiento se verán algunas diferencias.

La cantidad de componentes que posee el conjunto de herramientas de las clases Swing duplica a las de AWT. La primera es una librería extensiva de más de 250 clases y más de 75 interfaces para crear componentes lightweight 100% puro Java, cuya jerarquía de componentes ha sido simplificada.

El siguiente diagrama muestra sólo una pequeña porción de la jerarquía de las Swing, pero también muestra algunas diferencias importantes en el funcionamiento de la jerarquía de ambos paquetes.



Clases UI y clases no UI

En este diagrama se vislumbran dos características de las Swing: todas las clases de componentes Swing comienzan con "J" y todas descienden de la clase JComponent. Sin embargo no se debe concluir que todas las clases Swing son J clases, ni que todas descienden de Jcomponent.

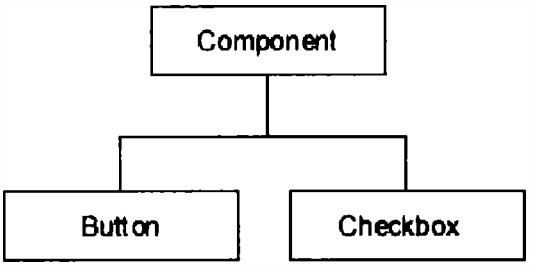
Las componentes Swing tienen dos tipos de clases: las clases UI y las clases no UI, y sólo las clases UI son J clases y descienden de Jcomponent.

La diferencia entre ambos tipos de clases es que el primero crea componentes visibles, tales como botones y menúes, y el otro, provee servicios para las del primer tipo. Las clases de eventos y de modelos corresponden al segundo grupo de clases.

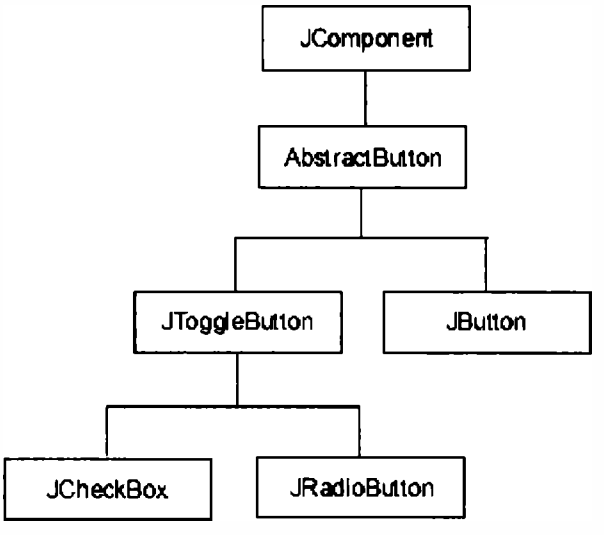
Cambios en la Jerarquía de Button

Se pueden observar diferencias en cómo funciona la jerarquía de Botones en ambos paquetes.

En AWT hay sólo dos clases de botones: Button y CheckBox, y ambas descienden de la clase Component.



En las Swing hay dos clases nuevas de botones y la jerarquía ha cambiado como muestra la figura:



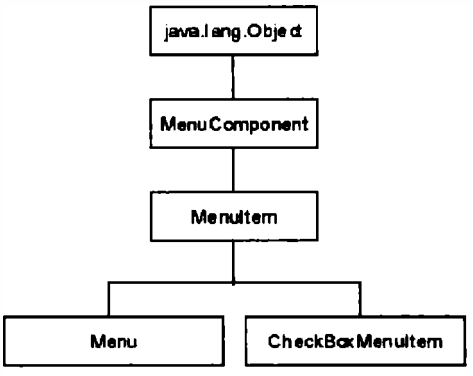
Las Swing poseen las clases radio button y check box, mientras que las AWT poseen la clase CheckBox que permite programar radio button y check box.

En las Swing, estas clases descienden de una nueva clase: JToggleButton, la cual tiene tres estados: presionado, habilitado y seleccionado. En las Swing, todas estas clases descienden de la clase abstracta AbstractButton.

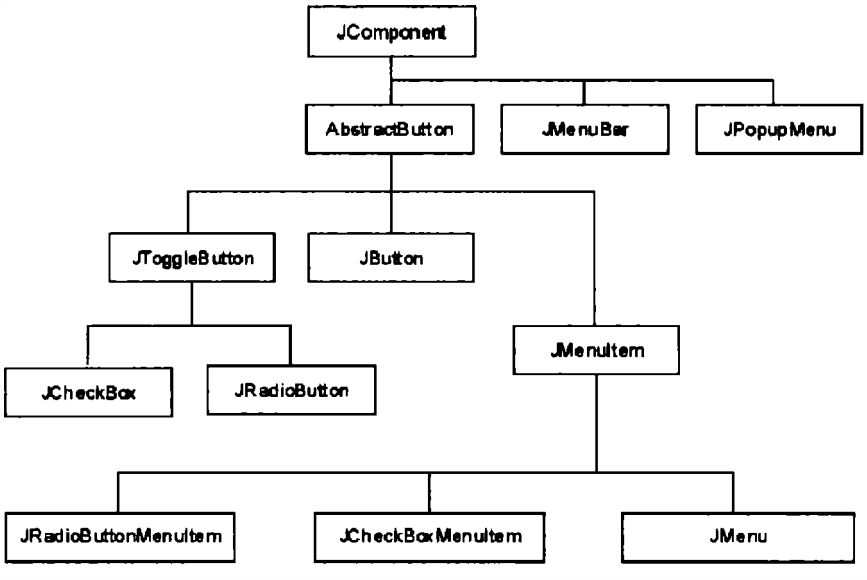
Cambios en la jerarquía Menú

Hay otras diferencias entre la jerarquía de las Swing y la de la AWT.

En las AWT, la clase Menú no desciende de la clase Component, pero sí lo hace de la clase Object. La jerarquía tiene la siguiente forma:



En las Swing, todas las clases menú descienden de la Jcomponent en un árbol más complejo que también involucra a la clase botón.



Debido a que los árboles de la clase botón y de la clase menú están estrechamente ligados, es más fácil coordinar los eventos del botón y de los menúes.

De hecho, las Swing tienen una nueva interface llamada Action Interface, que puede escuchar eventos de botón y de menú al mismo tiempo.

MEJORAR EL DISEÑO DE INTERFACES

La interfaz de usuario es el aspecto más importante de una aplicación. Un diseño pobre de interfaz es un grave problema para que el usuario pueda obtener el mejor resultado de la aplicación. Para ser efectivos, no debemos limitarnos a colocar una serie de botones, etiquetas y barras de desplazamiento sobre la pantalla.

Para crear una aplicación, se deberán tener en cuenta los puntos que se detallan a continuación:

Interface

- Crear el Marco de la aplicación (Frame, Panel).
- Inicializar Fuentes, Colores, Layouts y demás recursos.
- Crear menúes y Barras de Menúes.
- Crear los controles, diálogos, ventanas, etc.

Codificación

- Incorporar controladores de eventos
- Incorporar funcionalidad (threads, red, etc.)
- Incorporar un controlador de errores (excepciones)

CONTROL DE EVENTOS

Anteriormente, un programa que quería saber lo que estaba haciendo el usuario, debía recoger por sí mismo la información necesaria. Esto significaba que una vez inicializado, el programa entraba en un gran bucle en el que continuamente se bloqueaba para comprobar que el usuario estuviese haciendo algo interesante y tomar las acciones oportunas. Esta técnica se conoce como polling.

El polling funciona, pero se vuelve demasiado difícil de manejar con las aplicaciones modernas por dos razones fundamentales: primero, el uso de polling tiende a colocar todo el código de control de eventos en una única localización (dentro de un gran bucle); segundo, las interacciones dentro del gran bucle tienden a ser muy complejas. Además, el polling necesita que el programa esté ejecutando el bucle, consumiendo tiempo de CPU, mientras está esperando a que el usuario realice una operación, ésto supone un gran insumo de recursos.

El AWT y las JFC resuelven estos problemas abrazando un paradigma diferente, en el que están basados todos los sistemas modernos de ventanas: la orientación a eventos. Todas las acciones que pueda realizar el usuario caen dentro del conjunto de eventos. Un evento describe, con suficiente detalle, una acción particular del usuario. En lugar de que el programa activamente recoja todos los eventos generados por el usuario, el sistema Java avisa al programa cuando se produce un evento interesante.

LA CLASE EVENT

Un contenedor en un entorno gráfico se convierte en un receptor de eventos de todo tipo, relacionados con el movimiento del mouse, pulsaciones de teclas, creación/movimiento/destrucción de partes gráficas y, por último, los referidos a acciones del usuario respecto de componentes (elección de un menú, pulsación de un botón, etc.).

La clase Event es el jugador principal en el juego de los eventos. Intenta capturar las características fundamentales de todos los eventos que genera el usuario. Los datos miembro de la clase Event son los que se indican a continuación:

id - El tipo de evento que se ha producido

target - Componente sobre el que se ha producido el evento

x, y - Las coordenadas en donde se ha producido el evento relativas al Componente que actualmente está procesando ese evento. El origen se toma en la esquina superior izquierda del Componente

key - Para eventos de teclado, es la tecla que se ha pulsado. Su valor será el valor Unicode del carácter que representa la tecla. Otros valores que puede tomas son los de las teclas especiales como INICIO, FIN, F1, F2, etc.

when - Instante en que se ha producido el evento

modifiers - La combinación aritmética del estado en que se encuentran las teclas modificadoras Mays, Alt, Ctrl.

clickCount - El número de clicks de mouse consecutivos. Sólo tiene importancia en los eventos MOUSE_DOWN

arg - Es un argumento dependiente del evento. Para objetos Button, este objeto arg es un objeto String que contiene la etiqueta de texto del botón

evt - El siguiente evento en una lista encadenada de eventos

Una instancia de la clase Event será creada por el sistema Java cada vez que se genere un evento. Es posible, sin embargo, que un programa cree y envíe eventos a los Componentes a través de su método postEvent().

TIPOS DE EVENTOS

Los eventos se catalogan por su naturaleza, que se indicará en el miembro id de su estructura. Los grandes grupos de eventos son:

Eventos de Ventana

Son los que se generan en respuesta a los cambios de una ventana un frame o un diálogo.

- WINDOW_DESTROY
- WINDOW_EXPOSE
- WINDOW_ICONIFY
- WINDOW_DEICONIFY
- WINDOW_MOVED

Eventos de Teclado

Son generados en respuesta a cuando el usuario pulsa y suelta una tecla mientras un Componente tiene el foco de entrada.

- KEY_PRESS
- KEY_RELEASE
- KEY_ACTION
- KEY_ACTION_RELEASE

Eventos de Mouse

Son los eventos generados por acciones sobre el mouse dentro de los límites de un Componente.

- MOUSE_DOWN
- MOUSE_UP
- MOUSE_MOVE
- MOUSE_ENTER
- MOUSE_EXIT

- MOUSE_DRAG

Eventos de Barras

Son los eventos generados como respuesta a la manipulación de barras de desplazamiento (scrollbars).

- SCROLL_LINE_UP
- SCROLL_LINE_DOWN
- SCROLL_PAGE_UP
- SCROLL_PAGE_DOWN
- SCROLL_ABSOLUTE

Eventos de Lista

Son los eventos generados al seleccionar elementos de una lista.

- LIST_SELECT
- LIST_DESELECT

Eventos Varios

Son los eventos generados en función de diversas acciones.

- ACTION_EVENT
- LOAD_FILE
- SAVE_FILE
- GOT_FOCUS
- LOST_FOCUS

El Applet EventosPrnt.java está diseñado para observar los eventos que se producen sobre él. Cada vez que se genera un evento, el applet responde imprimiendo el evento que ha capturado en la línea de comandos desde donde se ha lanzado el applet.

MÉTODOS DE CONTROL DE EVENTOS

El método handleEvent() es un lugar para que el programador pueda insertar código para controlar los eventos. A veces, sin embargo, un Componente solamente estará interesado en eventos de un cierto tipo (por ejemplo, eventos del mouse). En estos casos, el programador puede colocar el código en un método de ayuda, en lugar de colocarlo en el método handleEvent().

La implementación del método handleEvent() proporcionada por la clase Component invoca a cada método de ayuda. Por esta razón, es importante que las implementaciones redefinidas del método handleEvent() en clases derivadas, siempre finalicen con la sentencia:

```
return( super.handleEvent( evt ) );
```

El siguiente trozo de código ilustra esta regla.

```
public boolean handleEvent( Event evt ) {
  if( evt.target instanceof MiBoton )
    {
    // Hace algo...
    return true;
    }
  return( super.handleEvent( evt ) );
  }
```

No seguir esta regla tan simple hará que no se invoquen adecuadamente los métodos de ayuda.

ACTION_EVENT

Algunos de los eventos que más frecuentemente tendremos que controlar son los siguientes:

- ACTION_EVENT
- MOUSE_DOWN
- KEY_PRESS
- WINDOW_DESTROY

Como ejemplo del manejo de eventos vamos a ver este evento que se provoca al pulsar un botón, seleccionar un menú, etc. Para su control podemos manejarlo en el método handleEvent() o en el método action().

Los dos métodos anteriores pertenecen a la clase Component por lo que todas las clases derivadas de ésta contendrán estos dos métodos y se pueden sobrecargar para que se ajuste su funcionamiento a lo que requiere nuestra aplicación.

En el siguiente ejemplo, se controla este evento a través del método handleEvent(), que es el método general de manejo de eventos:

```
public boolean handleEvent( Event evt ) {
   switch( evt.id ) {
      case Event.ACTION_EVENT:
         // evt.arg contiene la etiqueta del botón pulsado
         // o el item del menú que se ha seleccionado
         if( ( "Pulsado "+n+" veces" ).equals( evt.arg ) )
            return( true );
      default:
            return( false );
      }
   }
```

Pero en este caso, al producirse este evento, llama al método action(), que sería:

```
public boolean action( Event evt,Object arg ) {
   if( ( "Pulsado "+n+" veces" ).equals( arg ) )
      return( true );
   return( false );
   }
```

GENERACIÓN Y PROPAGACIÓN DE EVENTOS

Cuando un usuario interactúa con el applet, el sistema Java crea una instancia de la clase Event y rellena sus datos miembro con la información necesaria para describir la acción. Es en ese momento cuando el sistema Java permite al applet controlar el evento. Este control comienza por el Componente que recibe inicialmente el evento (por ejemplo, el botón que ha sido pulsado) y se desplaza hacia arriba en el árbol de Componentes, componente a componente, hasta que alcanza al Contenedor de la raíz del árbol. Durante este camino, cada Componente tiene oportunidad de ignorar el evento o reaccionar ante él en una (o más) de las forma siguientes:

- Modificar los datos miembros de la instancia de Event;
- Entrar en acción y realizar cálculos basados en la información contenida en el evento;
- Indicar al sistema Java que el evento no debería propagarse más arriba en el árbol;

El sistema Java pasa información del evento a un Componente a través del método handleEvent() del Componente. Todos los métodos handleEvent() deben ser de la forma:

```
public boolean handleEvent( Event evt )
```

Un controlador de eventos solamente necesita una información: una referencia a la instancia de la clase Event que contiene la información del evento que se ha producido.

El valor devuelto por el método handleEvent() es importante. Indica al sistema Java si el evento ha sido o no completamente manejado por el controlador. Un valor true indica que el evento ha sido controlado y que su propagación debe detenerse. Un valor false indica que el evento ha sido ignorado, o que no ha sido controlado en su totalidad y debe continuar su propagación hacia arriba en el árbol de Componentes.

INTERFACES

Una interfaz es una colección de definición de métodos sin implementar y de valores constantes.

DEFINICIÓN

La definición de una interfaz es similar a la creación de una nueva clase. Tiene dos componentes: la declaración de la interfaz y el cuerpo.

```
InterfaceDeclaration {
Interface Body
}
```

La declaración de la interfaz declara varios atributos de la misma tales como su nombre y si extiende o no otra interfaz.

El cuerpo de la interfaz contiene la declaración de constantes y métodos dentro de la interfaz.

La declaración de la Interfaz puede tener dos componentes más: el especificador de acceso público (public access specifier) y una lista de superinterfaces. Una interfaz puede extender otras interfaces, así como una clase puede extender una clase, pero a diferencia de las clases, una interfaz puede extender cualquier número de interfaces. Es decir soporta herencia múltiple.

La declaración completa sería:

```
[public] interface InterfaceName [extends list of  Super Interfaces]
```

El especificador de acceso público indica que la interfaz puede ser usada por cualquier clase en cualquier paquete.

Si no se especifica que la interfaz es pública, entonces puede ser sólo accedida por las clases que se encuentran definidas en el mismo paquete de la interfaz.

La cláusula *extends*, extiende una o varias interfaces.

Una interfaz hereda todas las constantes y métodos de su superinterface a menos que la interfaz oculte una constante con otra del mismo nombre o sobreescriba un método con una nueva declaración de métodos.

Todos los valores constantes definidos en la interfaz son públicos, estáticos y finales. Todos los métodos declarados en la interfaz son públicos y abstractos.

IMPLEMENTACIÓN DE UNA INTERFAZ

Para usar una interfaz, se debe escribir una clase que implemente la interfaz. Cuando una clase implementa una interfaz, debe proveer la implementación de cada uno de los métodos que se encuentran definidos en la interfaz.

A la implementación se le da uso cuando se escribe la clase que implementa la interfaz. Una clase declara las interfaces que implementa en la declaración de la clase. Para declarar que la clase implementa una o más interfaces se usa la palabra implements, seguido de una lista de interfaces separados por comas.

Por ejemplo:

Definamos una interfaz denominada Collection:

```
interface Collection{
int Maximum=500;
void add (Object obj);
void delete (Object obj);
int current count();
}
```

y definamos una clase que la implemente:

```
Class FifoQueue implements Collection {
    Void add (object obj) {
    . . . }
    Void delete (object obj) {
    . . . }
    Object find (object obj) {
    . . . }
    int current Count () {
    . . . }
}
```

Por convención la cláusula *implements* sigue a la cláusula *extends*, si la hubiera.

UTILIZACIÓN DE INTERFAZ COMO TIPO

Cuando se define una Interfaz, se está definiendo una nueva referencia de tipo de datos. Se puede utilizar los nombres de las Interfaces en cualquier lugar donde se utilice cualquier nombre de tipo: declaración de variables, parámetros, etc.

UTILIDAD DE LAS INTERFACES

Las Interfaces son útiles porque:

- Capturan similitudes entre clases que no están relacionadas sin forzar una relación de clases.
- Declaran métodos que una o más clases implementarán.
- Revelan una interfaz de programación de objetos, sin revelar su clase, tales objetos se denominan "objetos anónimos".
- Soportan herencia múltiple de Interfaces.
- Pueden ser implementadas por clases de cualquier paquete.

COMUNICACIONES

A - SOCKETS

Los sockets son puntos finales de enlaces de comunicación entre procesos, permitiendo el intercambio de datos entre éstos.

La información que permite transferir un socket esta definida por su tipo.

Sockets Stream (TCP, Transport Control Protocol)

Son un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados.

El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (Servidor), el otro solicita una conexión (Cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

Sockets Datagrama (UDP, User Datagram Protocol)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

DIFERENCIAS ENTRE SOCKETS STREAM Y DATAGRAMA

Ahora se nos presenta un problema, ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver algunas diferencias entre los protocolos para ayudar en la decisión. En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tenemos que establecer previamente una conexión entre los dos sockets, esto implica un cierto tiempo empleado en el establecimiento de dicha conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden. A diferencia de éste, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

USO DE SOCKETS

Podemos pensar que un Servidor Internet es un conjunto de sockets que proporciona capacidades adicionales del sistema, llamados servicios.

Puertos y Servicios

Cada servicio está asociado a un puerto. Un puerto es una dirección numérica a través de la cual se procesa el servicio. Sobre un sistema Unix, los servicios que proporciona ese sistema se indican en el fichero /etc/services, y algunos ejemplos son:

Daytime	13/udp	
ftp	21/tcp	
telnet	23/tcp	telnet
smtp	25/tcp	mail
http	80/tcp	

La primera columna indica el nombre del servicio. La segunda columna indica el puerto y el protocolo que está asociado al servicio. La tercera columna es un alias del servicio; por ejemplo, el servicio smtp, también conocido como mail, es la implementación del servicio de correo electrónico.

Las comunicaciones de información relacionada con Web tienen lugar a través del puerto 80 mediante protocolo TCP. Para emular esto en Java, usaremos la clase Socket.

CLASE URL - UNIVERSAL RESOURCE LOCATOR

La clase URL contiene contructores y métodos para la manipulación objetos o servicios en Internet. El protocolo TCP necesita dos tipos de información: la dirección IP y el número de puerto. Por ejemplo la página Web principal del buscador Yahoo, se activa al escribir http://www.yahoo.com en la línea de locación del browser.

Yahoo tiene registrado su nombre, permitiendo que se use yahoo.com como su dirección IP, o lo que es lo mismo, 205.216.146.71, su dirección IP real.

Hay un servicio, el DNS (Domain Name Service), que traslada www.yahoo.com a 205.216.146.71, el que permite teclear www.yahoo.com, en lugar de tener que recordar su dirección IP.

Para obtener la dirección IP real de la red, se deben realizar llamadas a los métodos getLocalHost() y getAddress(). El método getLocalHost() devuelve un objeto iNetAddress, que usado con getAddress() generará un array con los cuatro bytes de la dirección IP, por ejemplo:

```
InetAddress direccion = InetAddress.getLocalHost();
byte direccionIp[] = direccion.getAddress();
```

Si la dirección de la máquina en que se está ejecutando es 150.150.112.145, entonces:

```
direccionIp[0] = 150
direccionIp[1] = 150
direccionIp[2] = 112
direccionIp[3] = 145
```

Una cosa interesante en este punto es que una red puede mapear muchas direcciones IP. Esto puede ser necesario para un Servidor Web, como Yahoo, que tiene que soportar grandes cantidades de tráfico y necesita más de una dirección IP para poder atender a todo ese tráfico. El nombre interno para la dirección 205.216.146.71, por ejemplo, es www7.yahoo.com. El DNS puede trasladar una lista de direcciones IP asignadas a Yahoo en www.yahoo.com. Esto es una cualidad útil, pero por ahora abre un agujero en cuestión de seguridad.

Si no se indica un número de puerto, se utilizará el que se haya definido por defecto en el archivo de configuración de los servicios del sistema. El puerto habitual de los servicios Web es el 80. Si se escribe el siguiente URL en un navegador:

```
http://www.yahoo.com:80
```

también se carga la página principal del Yahoo.

El protocolo http (HyperText Transmission Protocol), es el utilizado para manipular documentos Web. Y si no se especifica ningún documento, muchos servidores están configurados para devolver un documento de nombre index.html.

Java permite los siguientes cuatro constructores para la clase URL:

```
public URL( String spec ) throws MalformedURLException;
public URL( String protocol,String host,int port,String file ) throws MalformedURLException;
public URL( String protocol,String host,String file ) throws MalformedURLException;
public URL( URL context,String spec ) throws MalformedURLException;
```

DOMINIOS DE COMUNICACIONES

El mecanismo de sockets está diseñado para ser todo lo genérico posible. El socket por sí mismo no contiene información suficiente para describir la comunicación entre procesos. Los sockets operan dentro de dominios de comunicación, entre ellos se define si los dos procesos que se comunican se encuentran en el mismo sistema o en sistemas diferentes y cómo pueden ser direccionados.

Dominio Unix

Bajo Unix, hay dos dominios, uno para comunicaciones internas al sistema y otro para comunicaciones entre sistemas.

Las comunicaciones intrasistema (entre dos procesos en el mismo sistema) ocurren (en una máquina Unix) en el dominio Unix. Se permiten tanto los sockets stream como los datagrama. Los sockets de dominio Unix bajo Solaris 2.x se implementan sobre TLI (Transport Level Interface).

En el dominio Unix no se permiten sockets de tipo Raw.

Dominio Internet

Las comunicaciones intersistemas proporcionan acceso a TCP, ejecutando sobre IP (Internet Protocol). De la misma forma que el dominio Unix, el dominio Internet permite tanto sockets stream como datagrama, pero además permite sockets de tipo Raw.

Los sockets stream permiten a los procesos comunicarse a través de TCP. Una vez establecidas las conexiones, los datos se pueden leer y escribir a/desde los sockets como un flujo (stream) de bytes. Algunas aplicaciones de servicios TCP son:

- File Tranfer Protocol, FTP
- Simple Mail Transfer Protocol, SMTP
- TELNET, servicio de conexión de terminal remoto

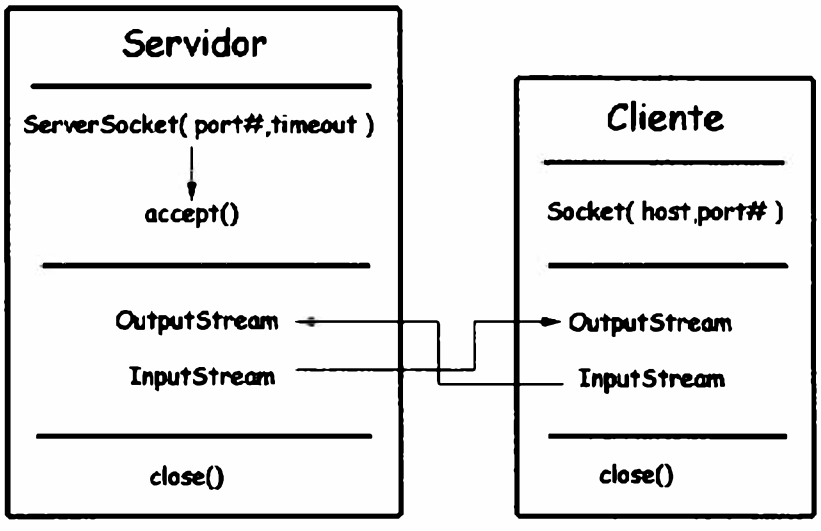
Los sockets datagrama permiten a los procesos utilizar el protocolo UDP para comunicarse a y desde esos sockets por medio de bloques. UDP es un protocolo no fiable y la entrega de los paquetes no está garantizada. Servicios UDP son:

- Simple Network Management Protocol, SNMP
- Trivial File Transfer Protocol, TFTP (versión de FTP sin conexión)
- Versatile Message Transaction Protocol, VMTP (servicio fiable de entrega punto a punto de datagramas independiente de TCP)

Los sockets raw proporcionan acceso al Internet Control Message Protocol, ICMP, y se utiliza para comunicarse entre varias entidades IP.

MODELO DE COMUNICACIONES CON JAVA

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



El modelo de sockets más simple es:

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

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

APERTURA DE SOCKETS

Al programar un cliente, el socket se abre de la forma:

```
Socket miCliente;
miCliente = new Socket( "maquina",numeroPuerto );
```

Donde máquina es el nombre de la máquina en donde se intenta abrir la conexión y numeroPuerto es el puerto (un número) del servidor que está corriendo sobre el cual se desea conectar. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las aplicaciones que se desarrollen, seleccionar un puerto superior al 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, sería conveniente la captura de excepciones cuando se está trabajando con sockets.

El ejemplo quedaría:

```
Socket miCliente;
try {
   miCliente = new Socket( "maquina",numeroPuerto );
} catch( IOException e ) {
   System.out.println( e );
   }
```

Al programar un servidor, la forma de apertura del socket es la que muestra el siguiente ejemplo:

```
Socket miServicio;
try {
   miServicio = new ServerSocket( numeroPuerto );
} catch( IOException e ) {
   System.out.println( e );
   }
```

A la hora de la implementación de un servidor también es necesario crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;
try {
   socketServicio = miServicio.accept();
} catch( IOException e ) {
   System.out.println( e );
   }
```

CREACION DE STREAMS

Creación de Streams de Entrada

En la parte cliente de la aplicación, se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream entrada;
try {
   entrada = new DataInputStream( miCliente.getInputStream() );
} catch( IOException e ) {
   System.out.println( e );
   }
```

La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readLine(). Se deberá utilizar la función necesaria, según el tipo de dato que se espera recibir del servidor.

En el lado del servidor, también se utilizará DataInputStream, pero en este caso para recibir las entradas que produzcan los clientes que se hayan conectado:

```
DataInputStream entrada;
try {
   entrada =
      new DataInputStream( socketServicio.getInputStream() );
} catch( IOException e ) {
   System.out.println( e );
   }
```

Creación de Streams de Salida

En el lado del cliente, se puede crear un stream de salida para enviar información al socket del servidor utilizando las clases PrintStream o DataOutputStream:

```
PrintStream salida;
   try {
      salida = new PrintStream( miCliente.getOutputStream() );
   } catch( IOException e ) {
      System.out.println( e )   }
```

La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write y println() tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar DataOutputStream:

```
DataOutputStream salida;
try {
   salida = new DataOutputStream( miCliente.getOutputStream() );
} catch( IOException e ) {
   System.out.println( e );
   }
```

La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, uno de los más útiles es writeBytes().

En el lado del servidor, se puede utilizar la clase PrintStream para enviar información al cliente:

```
PrintStream salida;
try {
   salida = new PrintStream( socketServicio.getOutputStream() );
} catch( IOException e ) {
   System.out.println( e );
   }
```

Además puede utilizarse la clase DataOutputStream como en el caso de envío de información desde el cliente.

CIERRE DE SOCKETS

Siempre se deben cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {
   salida.close();
   entrada.close();
   miCliente.close();
} catch( IOException e ) {
   System.out.println( e );
   }
```

Y en la parte del servidor:

```
try {
   salida.close();
   entrada.close();
   socketServicio.close();
   miServicio.close();
    }
 catch( IOException e ) {
   System.out.println( e );
   }
```

CLASES UTILES EN COMUNICACIONES

Algunas clases que resultan útiles en el desarrollo de programas de comunicaciones, además de las mencionadas anteriormente, se encuentran en el paquete sun.net y son las que se detallan a continuación:

Socket

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

ServerSocket

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

DatagramSocket

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

DatagramPacket

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

MulticastSocket

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

NetworkServer

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

NetworkClient

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

SocketImpl

Es una Interfaz que permite crear un modelo de comunicación. Al utilizarla se deberán codificar sus métodos.

B - METODOS DE INVOCACION REMOTA (RMI)

El mecanismo RMI provee los medios para invocar métodos remotos. Con esto se habilita el desarrollo de aplicaciones distribuidas. El sistema RMI de Java es mecanismo de Java para hacer programación distribuida.

Desde un punto de vista conceptual, RMI permite a las aplicaciones Java llamar a objetos remotos como si éstos estuvieran en la máquina local. Desde el punto de vista de la aplicación, RMI provee los servicios necesarios para localizar el objeto remoto, transferir la llamada incluyendo todos los parámetros usando TCP/IP al servidor, invocar el método sobre el objeto servidor y retornar el resultado a través del mismo camino.

El mecanismo RMI de Java es soportado en el lenguaje por los paquetes:

- java.rmi
- java.rmi.server
- java.rmi.registry

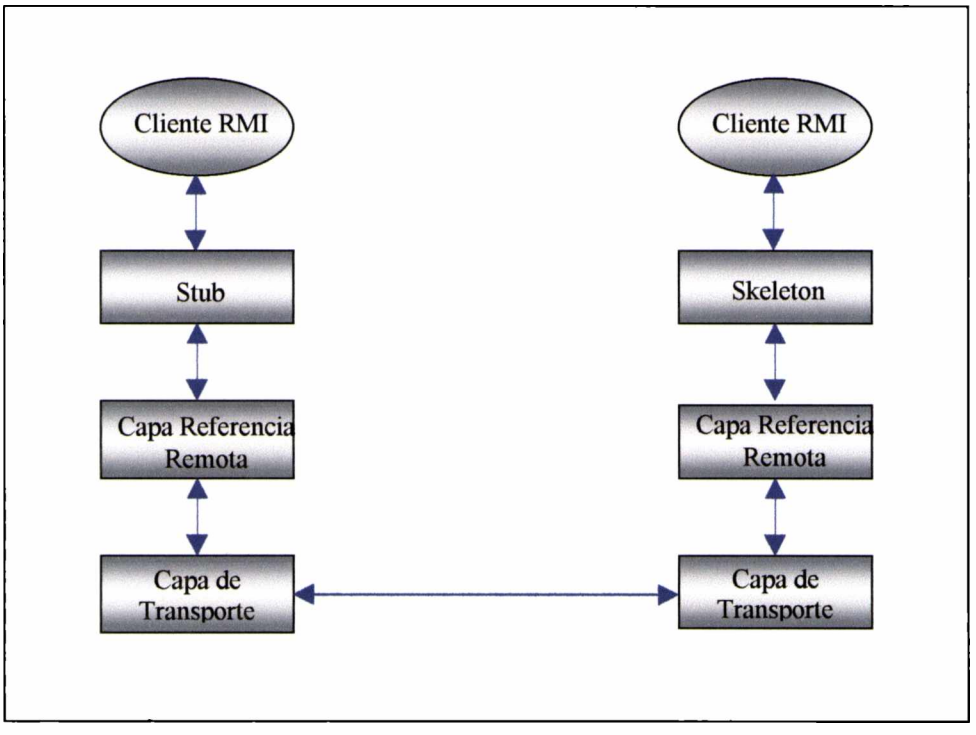
Uno de los servicios que provee RMI es enviar objetos a través de la red. Como TCP/IP no reconoce objetos y sólo maneja bits y bytes, los objetos deben ser convertidos en cadenas de bytes.

Cuando se desea enviar objetos, convertirlos en una cadena de bits y restaurarlos en el destino se conoce como serialización y deserialización respectivamente.

Para asegurar esto, la definición de la clase del objeto o alguna de sus super clases deberá incluir la interface java.io.Serializable.

ARQUITECTURA RMI

Las llamadas RMI entre el cliente y el servidor atraviesan diferentes capas hasta que logran la comunicación entre ambos.



Stub y Skeleton

Son interfaces entre la capa de aplicación y el resto del sistema RMI. El stub reside en el cliente y serializa los argumentos, realiza las llamadas a objetos remotos invocando a la capa de referencia remota y deserializa los objetos retornados y las excepciones.

El skeleton, del lado del servidor, es responsable de deserializar los argumentos, llamar al objeto servidor y serializar los objetos retornados o las excepciones.

Capa de Referencia Remota

Tiene un componente en el cliente y otro en el servidor. Es responsable de mantener un protocolo de referencia entre los componentes del cliente y del servidor, el cual es independiente del stub o skeleton. Mantiene la referencia y reconect si una conexión se pierde.

Capa de Transporte

Crea y monitorea las conexiones apoyadas en la capa de referencia. Establece conexiones de sockets y pasa la conexión a la capa de transporte. También escucha las llamadas entrantes y setea conexiones para ésto.

PROCESO DE DESARROLLO DE APLICACIONES RMI

Consiste en realizar los siguientes pasos:

- Definir la interface pública del servidor
- Escribir la implementación del servidor
- Compilar la interface y la implementación del servidor
- Crear las clases Stub y Skeleton
- Escribir y compilar la lógica del cliente

Código específico RMI

Tanto el código del servidor como el del cliente necesitan instrucciones especiales para hacer que la conexión RMI funcione:

- La definición de la interface del servidor debe ser una subclase de java.rmi. Remote:

```
public interface ServerInterface extends Remote
```

- Cada método público del servidor puede lanzar java.rmi. RemoteException:

```
aDublic returnType methodName(...parms... ) throws RemoteException
```

- La clase del servidor debe ser una subclase de java.rmi.server. RemoteServer; normalmente se usa una subclase de java.rmi.server. UnicastRemoteObject:

```
public class ServerClass extends UnicastRemoteObject
                    implements ServerInterface
```

- La clase del servidor debe setear el "security manager":

```
System. setSecurityManager (new RMISecurityManager ( ) );
```

- La clase del servidor debe registrarse con el RMI registry:

```
serverInstance = new ServerClass ( );   //constructor
Naming. rebind ("rmi:///ServerName", (Remote) serverInstance);
```

El servidor se registra bajo un nombre de usuario definido, ServerName, con java.rmi.Naming.rebind()

- La clase cliente debe setear un "security manager":

```
System. setSecurityManager (new RMISecurityManager ( ) );
```

- La clase cliente debe buscar al servidor en la máquina remota para obtener una referencia a un objeto para invocar los métodos remotos.

```
urlstring = "rmi://serverHostname/ServerName"
serverObject=(ServerClassInterface)Naming.lookup(urlstring);
```

El cliente debe conocer la dirección TCP/IP del servidor.

```
value = serverObject .methodName (.. .parms. . . )
```

Ambiente de ejecución RMI

La secuencia de operaciones para ejecutar una aplicación RMI es:

Iniciar el registro RMI en la máquina servidora (1).

Iniciar la aplicación servidor (2), la cual setea el "security manager"(3) y se registra a sí misma con el registro RMI (4).

Iniciar la aplicación cliente o applet (5), la cual setea el "security manager" (6) y busca el servidor usando el registro RMI en la máquina remota (7).

Ahora, el cliente puede usar un objeto local para invocar métodos del servidor a través del stub y el skeleton (8). Los objetos resultantes de dichos llamados son enviados al cliente (9).

EJEMPLO DE RMI

Crearemos un servidor RMI que concatena una cadena de entrada con el mismo string escrito al revés y los retorna concatenados. Usamos el paquete RMINative para implementar el ejemplo.

Interface pública Server

La interface pública define los métodos que pueden ser invocados desde el cliente. En nuestro ejemplo se define el método reverseAppend:

```
/* RMINative\SampServerIf. java */
import java.rmi.*;
 public interface SampServerIf extends Remote
  {
   public abstract String reverseAppend(String str)
                throws RemoteException;  }
```

Implementación del Server

El servidor debe extender la clase Remote e implementar la interface pública:

```
/* RMINative\SampServer.java */
package RMINative;
import java.rmi.*;
import java.rmi.server.*;
public class SampServer extends UnicastRemoteObject
            implements RMINative. SampServerIf

public static void main (String args [] )

RMINative. SampServer reverserObj ect = null;
try
{
  System.out .println ("Setting up the Security Manager . . . ");
  System. setSecurityManager (new RI~ISecurityManager ());
  System. out .println ( "Publishing the \"Reverser\" object:
            rmi:///Reverser");
  reverserObject = new RMINative. SampServer();
  Naming. rebind ( "rmi:///Reverser", (Remote) reverserObjeet);
  System.out.println("Reverser server is ready ...");
}
catch (Exception e)
{
  System.out.println("Exception "+e+" caught: \n"+e.getMessage() );
}
return;
Constructor  crea el objeto servidor:
public SampServer() throws RemoteException
{
 super();
 return }
```

El método reverseAppend es llamado por el cliente a través del stub y el skeleton.

```
public String reverseAppend(String str) throws RemoteException
{
 String result = null;
 try
 {
  System.out.println("Client "+getClientHost()+" says: "+str);
  result = performWork(str);
  return result;
 }
 catch (Exception e)
 {
  System.out.println("Exception "+e+" caught: \n"+e.getMessage ());
  return "Error";
   }
```

El método performWork method es un método privado que realiza el trabajo de construir el string resultante.

```
 private String performWork(String in)
 {
  StringBuffer buf;
  buf = (new StringBuffer(in)).reverse();
  return in+":"+(new String(buf));
 }
} //fin de la clase
```

Luego de compilar tanto la interface como la implementación del servidor se ejecutará el compilador RMI para crear las clases stub y skeleton:

Implementación del Cliente

```
/* RMINative\SampClient.java */
package RMINative;
import java.rmi.*;
import java.io.*;
public class SampClient
{
  public static void main(String args[])
   {
        RMINative. SampServerIf reverserObject = null;
        BufferedReader infile = new BufferedReader(
```

```
                    new InputStreamReader(System.in));
          String hostname = "";
          String urlstring;
          String userinput;
          String result;
          if (args.length == 1) { hostname = args[0]; };
                urlstring = "rmi: / / "+hostname+"/Reverser";
          if (hostname == "") hostname = "(local)"; //for the message
         try
          {
System. out. println ( "Registering the Security Manager . . . ");
System. setSecurityManager ( new RMISecurityManager ( ) );
System.out.println("Looking up the Reverser on "+hostname+
                  ".. . Please Wait ! ");
reverserObject= (RMINative. SampServerIf) Naming. lookup
                          (urlstring);
  do {
   System.out.println("What should I say to the server ?
                (or type: end) ");
   userinput = infile. readLine ( );
   System.out.println(" - calling server: "+userinput);
   result = reverserObject. reverseAppend (userinput);
   System.out.println(" - server replied: "+result);
             } while (!userinput.equals ("end"));
       }
catch (RemoteException el)
{System.out.println("Something is wrong with the RMI connection
    System.out.println("Exception "+el+" caught: \n"+el.getMessage
catch (java.net.MalformedURLException e2)
{System.out.println("The URL is not valid: "+urlstring);
    System.out.println("Exception "+e2+" caught: \n"+e2.getMessage
catch (Exception e3)
{System.out.println("Exception "+e3+" caught: \n"+e3.getMessage}
;}
;}
System.out .printin ("End of RMI Clientkn");
return;
private SampClient ( ) { return; }
} //fin de la clase
```

JAVA BEANS

QUÉ ES UN JAVA BEAN?

Es una componente de software reusable que es manipulada visualmente a través de una herramienta de desarrollo.

Estos beans son clases Java. En la herramienta de desarrollo, se selecciona un Bean de una paleta, se especifican sus características y se realizan conexiones entre ellos.

Existen dos tipos de Beans:

Visuales: pueden visualizarse en el programa en el momento de ejecución. Los Beans visuales, tales como ventanas, botones, campos de texto conforman la interfaz gráfica de usuario(GUI) de un programa.

No visuales: no aparecen en el programa en tiempo de ejecución. Generalmente representa a un objeto que encapsula datos e implementa un comportamiento dentro de un programa.

La interfaz pública de un Bean determina cómo interactúa con otros beans. Tiene las siguientes carácterísticas:

- Propiedades: Son los datos que pueden ser accedidos por otros beans, éstos pueden representar propiedades lógicas de un bean, tal como el balance de una cuenta, el tamaño o la etiqueta de un botón.
- Eventos: Son señalizaciones que indican que algo ocurrió. Abrir una ventana o cambiar el valor de una propiedad lanzaría un evento.
- Métodos: Son las operaciones que un bean puede realizar. Los métodos pueden lanzarse a través de las conexiones con otros beans.

CONEXIONES

Definen como interactúan los Beans. Se pueden realizar conexiones entre beans y entre conexiones. Una conexión tiene una fuente (punto donde comienza la conexión) y un destino (punto donde finaliza la conexión).

Existen seis tipos de conexiones diferentes:

♣ Propiedad a Propiedad: Une dos valores de modo que si se definen los *eventos* en la fuente y en el destino, cuando un valor cambia el otro también lo hace. En este caso cualquiera de los dos extremos pueden ser la fuente o el destino. El único inconveniente para decidir cual es la propiedad que actúa como fuente y cual como destino se presenta en el momento de la inicialización, ya que durante la misma el valor de destino se actualiza para igualarse al valor de la fuente.

♣ Evento a Método: Llama a un *método* cuando ocurre un *evento*. El *evento* es siempre la fuente y el *método* el destino. Si se intenta hacer al revés, Visual Age invierte la conexión para que ésta quede de la forma antes mencionada.

♣ Evento a Script: Ejecuta un *script* cuando ocurre un *evento*. El destino puede ser cualquier *método* de la Clase que esté siendo manipulada por el Editor de Composición Visual. Una conexión *evento a método* se realiza entre dos Beans, mientras que una conexión *evento a script* se realiza entre un bean y un método en el bean compuesto. Dicho método no necesita ser público.

♣ Parámetro desde Propiedad: Usa el valor de la propiedad como parámetro para la conexión.

♣ Parámetro desde Script: Ejecuta un Script cuando se requiere un parámetro para la conexión.

♣ Parámetro desde Método: Utiliza el resultado de un método como parámetro de la conexión.

JDBC

JDBC es un API Java que ejecuta sentencias SQL. Consta de un conjunto de clases e interfaces escritas en Lenguaje Java.

¿Qué hace JDBC?

1. Establece una conexión con la Base de Datos.
2. Envía sentencias SQL.
3. Procesa los resultados.

Un Objeto de conexión representa una conexión con la Base de Datos. Incluye las sentencias SQL ejecutadas y los resultados.

Existe un Manejador de JDBC que contiene una lista de clases de Drivers registrados, llamado Driver Manager Class.

Para abrir una conexión se utiliza el método getConnection. Cuando se llama a dicho método, el Driver Manager Class chequea la lista y busca un driver que pueda conectar la aplicación con la Base de Datos especificada en el URL.

JDBC provee tres Clases para enviar sentencias sentencias SQL y tres Métodos de la interfaz connection que crea instancias de estas tres clases.

Métodos utilizados

- ♣ createStatement: genera un objeto statement que permite manejar sentencias SQL que no contienen parámetros.
- ♣ prepareStatement: genera un objeto prepareStatement que permite el manejo de sentencias SQL con parámetros IN y sentencias de uso frecuente.
- ♣ prepareCall: genera un objeto callableStatement y permite el manejo de llamadas a procedimientos almacenados.

La interfaz Statement provee tres métodos diferentes para ejecutar las sentencias SQL. De acuerdo al resultado de la ejecución, debe utilizarse alguno de los siguientes métodos:

- *executeQuery*: Fue diseñado para sentencias que producen un conjunto simple de resultados, como las sentencias SELECT.
- *executeUpdate*: Se utiliza para ejecutar INSERT, UPDATE o DELETE. Además para sentencias SQL DDL (Lenguaje de definición de datos), tales como CREATE TABLE y DROP TABLE. El resultado es un entero que indica el número de filas afectadas por la consulta, y cero en el caso de las sentencias que no operan sobre las mismas.
- *Execute*: Utilizado para sentencias que retornan más de un conjunto de resultados, más de un contador de modificación o una combinación de ambos.

JDBC VERSUS ODBC

Hasta este momento, ODBC es probablemente la interfaz de programación más usada para acceder a Bases de Datos relacionales. Permite conectarse con casi todas las plataformas.

Por qué entonces no utilizar ODBC en lugar de JDBC?

La respuesta es que se puede usar ODBC, pero esto sería mejor si usamos la ayuda de JDBC como un puente JDBC-ODBC.

Ahora la pregunta sería por qué necesitamos JDBC?

Porque ODBC no es apropiado para usar directamente desdes Java ya que usa una interfaz C. Java no tiene punteros y ODBC los utiliza frecuentemente. Entonces podemos ver a JDBC como ODBC traducido a una interfaz orientada a objetos.

Además un APIJava como JDBC es necesario para dar una solución Java puro.

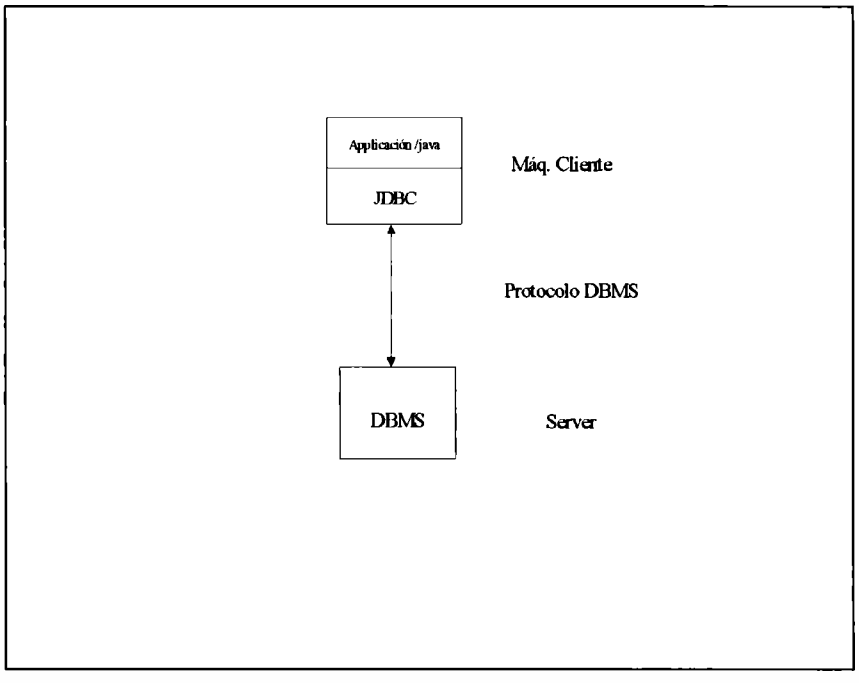
Cuando se usa el manejador de driver ODBC, los drivers deben ser instalados manualmente en cada máquina cliente, mientras que con JDBC se instalan automáticamente.

MODELOS TWO-TIER Y THREE-TIER

El API JDBC soporta ambos modelos para el acceso a las bases de datos.

En el modelo Two-tier, un Applet Java o una aplicación conversa directamente con la Base de Datos.

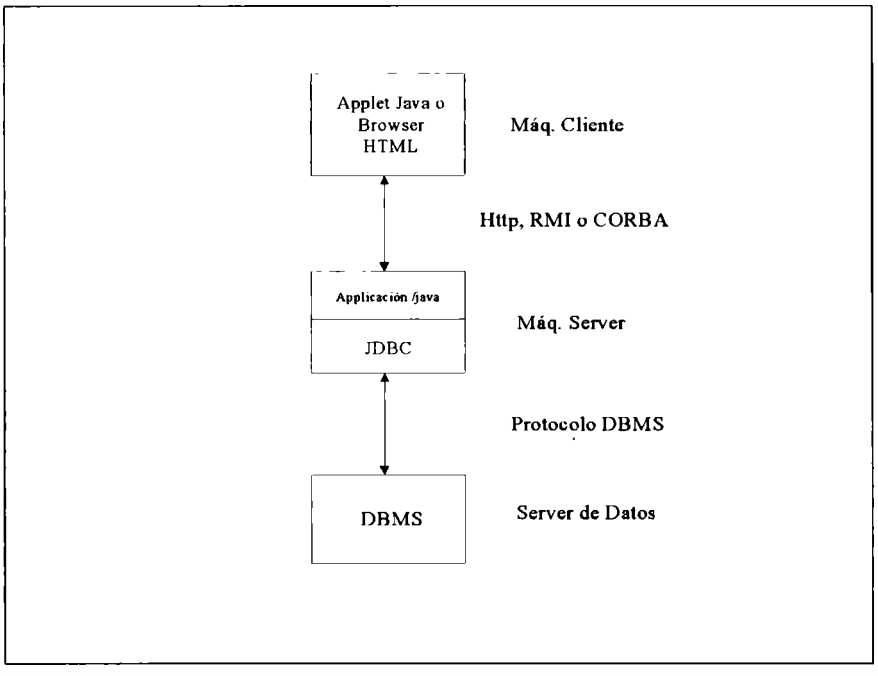
Las sentencias SQL son enviadas a la Base de Datos y los resultados de esas sentencias son devueltos al usuario. La Base de Datos debe encontrarse en otra máquina con la cual el usuario se conectará vía red. Esto hace referncia a una configuración cliente/servidor.



Modelo Two Tier

En el modelo Three-tier, los comandos son enviados a un intermediario de servicios, el cual es el encargado de enviar las sentencias SQL a la Base de Datos. La Base de Datos procesa las sentencias SQL y envía los resultados al intermediario, el cual los envía al usuario.

En este modelo se puede llevar un control de los accesos y los tipos de modificaciones que se pueden realizar sobre la Base de Datos.



Modelo Three Tier

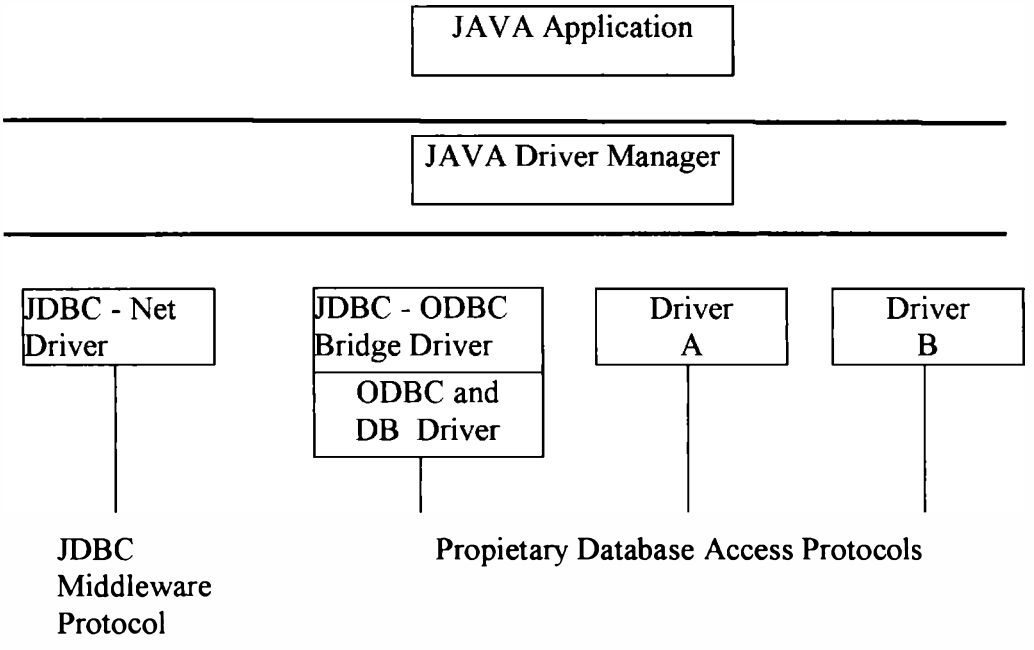
PRODUCTOS JDBC

JavaSoft provee tres componentes como parte del Java Development Kit:

♣ *driver manager JDBC:* es el esqueleto de la arquitectura JDBC, su principal función es conectar la aplicación Java al driver JDBC correcto

♣ *driver JDBC test suite:* provee la confiabilidad que el driver JDBC correrá en la aplicación

♣ *bridge JDBC-ODBC:* permite que los drivers ODBC sean usados como drivers JDBC



Tipos de Drivers JDBC

1. *Bridge JDBC-ODBC:* provee acceso JDBC a través de los drivers ODBC. En este caso se deberían instalar los drivers apropiados en cada máquina cliente.
2. *Native API partly-Java driver:* este tipo de driver convierte las llamadas JDBC en llamadas al API cliente para Oracle, Sybase, Informix u otros DBMS. Nótese que como

el driver bridge, requiere que algún código binario sea cargado en cada máquina cliente.

3. *JDBC-Net pure Java driver:* este driver traduce la llamada JDBC en un protoclo de red de DBMS independiente, el cual es traducido luego a un protocolo DBMS por el servidor. Este servidor de red intermediario puede conectar los clientes Java puro a las distintas bases de datos. En orden para que estos productos también soporten accesos a Internet deben manejar los requerimientos adicionales la Web impone para seguridad, accesos a través de firewall , y otros.

4. *Native-Protocol pure Java driver:* convierte directamente la llamada JDBC en el protocolo de red usado por el DBMS. Esto permite una llamada directa desde la máquina Cliente al Server DBMS.

La siguiente tabla muestra las cuatro categorías y sus propiedades.

Categoría de Driver	Todo	Protocolo de Red
JDBC-ODBC Bridge	No	Directo
Native API	No	Directo
JDBC-Net	Si	Requiere Conector
Native Protocol	Si	Directo

RESULT SET

Contiene todas las filas que satisfacen la condición en una sentencia SQL y provee acceso a los datos a través de métodos get que permiten obtener las distintas columnas de la fila corriente.

El Result Set mantiene un cursor que apunta a la fila corriente de datos. Este cursor se mueve una fila abajo cada vez que el método next es llamado. Inicialmente se posiciona antes de la primera fila, entonces la primer llamada al método next coloca el cursor en la primer fila, haciendo que ésta sea la fila corriente.

En SQL, el cursor para la tabla resultado es nombrado. Si la base de datos tendrá modificaciones y borrados, se necesita proveer el nombre de la tabla resultado como un parámetro para los comandos update o delete. El nombre del cursor puede obtenerse a través del método getCursorName.

Para los métodos getxxx, el driver JDBC intenta convertir el dato subyacente al tipo Java especificado y retornar un valor Java acorde.

Las sentencias SQL son ejecutadas y retornan un conjunto de resultados, o contadores de modificación.

Para ésto JDBC provee un mecanismo, para que luego de ejecutar una sentencia, la aplicación pueda procesar cualquier colección de conjuntos resultados o contadores de modificación.

Este mecanismo se basa en llamar al método general execute y luego llamar a otros tres métodos: getResultSet, getUpdateCount y getMoreResults. Estos métodos le permiten a la aplicación explorar los resultados y determinar si se obtuvo ResultSet o un contador de modificación.

Tabla de Conversión de Datos

Para obtener los datos del conjunto de resultados de la ejecución de sentencias SQL, se utiliza el método resulset.getXXX, donde XXX representa al tipo de datos a extraer.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBitDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimeStamp											x	x	x				x		X
getASCIIStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

SOPORTE JDBC PARA APPLET Y APLICACIONES

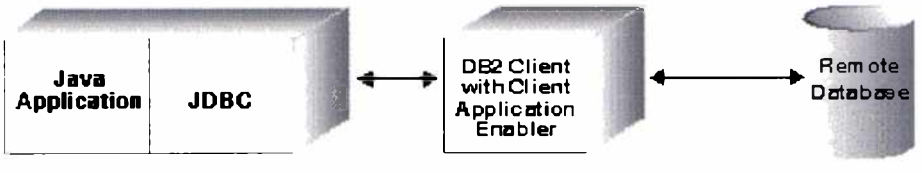
Un applet es una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta en el lugar como parte de un documento web. Estas aplicaciones están basadas en un formato gráfico sin representación independiente. Es el elemento interactivo a embeber en otras aplicaciones.

Una aplicación es un programa JAVA stand alone que corre independientemente de un browser . Las aplicaciones son completamente portables ya que solo toman el código y lo corren en otra máquina.

Implementación de una Aplicación JDBC para Base de Datos

En el Cliente debe estar instalado la componente CAE (Client Applications Enablement)

Las llamadas al JDBC son traducidas a llamadas CLI para manejo de Base de Datos, a través de métodos nativos Java, por lo tanto, el flujo normal de comunicación entre el Cliente y la Base de Datos se realiza a través del DB2 CAE.

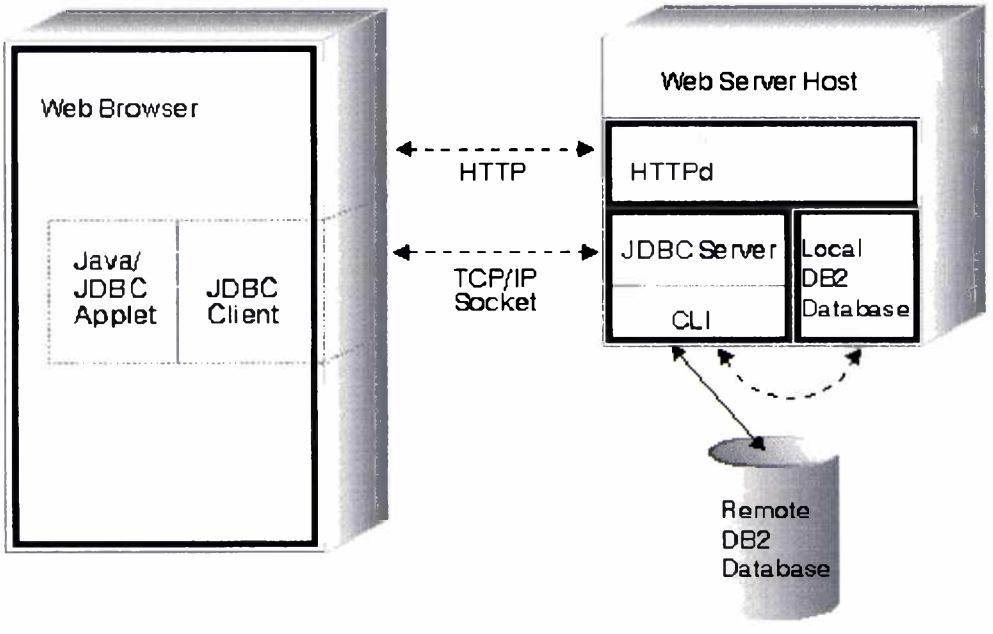


Implementación de un Applet JDBC para Base de Datos

El Driver consiste de un Cliente y un Server JDBC.

La componente Cliente JDBC es cargada en el Web browser junto con el Applet.

Cuando un applet requiere una conexión a una Base de Datos el Cliente abre el socket TCP/IP del Server JDBC a la máquina donde el Web Server está corriendo. Luego que se establece la conexión el cliente envía todos los requerimientos de acceso a las Bases de Datos desde el applet al Server JDBC mediante una conexión TCP/IP, entonces el Server JDBC realiza las llamadas a la Base para realizar las tareas (mediante una Interfaz de llamada CLI, Call Level Interface) correspondientes, y luego envía los resultados.



DIFERENCIAS DEL SOPORTE ENTRE APPLETS Y APLICACIONES

La principal diferencia entre applets y aplicaciones es que una aplicación requiere que DB2 CAE sea instalado en el Cliente y usa CAE para comunicarse con la Base de Datos, mientras que el applet depende de un Web browser que soporta Java y no requiere ningún código de Base de Datos instalado en el cliente.

Además existen otras diferencias importantes:

- Utilizan distintos drivers
- Se conectan con un formato URL distinto.
- Importan diferentes paquetes java.sql

Pasos para crear una Aplicación o un Applet JDBC

1. Importar el paquete JDBC que corresponda
   Aplicación : java.sql*
   Applet tempjava.sql*
2. Registrar el driver apropiado con el driver manager.
   Aplicación : java.sql*
   Applet tempjava.sql*
3. Pasar el URL al Driver para que se comunique en la Base de Datos. El protocolo es JDBC y el subprotocolo DB2.
4. Crear y Ejecutar las sentencias que de acuerdo al API-JDBC.
5. Recibir los resultados.
6. Cerrar las sentencias y el conjunto de resultados.
7. Cerrar la conexión con la Base de Datos.

Ambientes de Desarrollo

AMBIENTES DE DESARROLLO

COMPARACIÓN DE AMBIENTES

Java no posee un ambiente para programar. Para la construcción de programas en lenguaje Java pueden utilizarse productos tales como JBuilder, Visual Age para Java, Java Workshop, Symantec Visual Café, Cosmo Software Cosmo Code, SuperCede, entre otros.

Para elegir el ambiente para el desarrollo de la Aplicación, comparamos aquellos de los cuales disponíamos de información.

1 - JBuilder

Puntos Fuertes

Incluye las herramientas más completas para el manejo de Base de Datos.

Puede tomar código generado fuera de la herramienta y generar la composición visual en todos los casos.

Su diseñador de GUI permite tomar y arrastrar los objetos de la paleta para ubicarlos en el compositor visual, sin la necesidad de conocer las restricciones de los layouts. Permite además seleccionar varios objetos para realizar operaciones sobre ellos en forma simultánea, como copiar, cortar, cambiar propiedades.

Puntos Débiles

Es una de las herramientas que se ejecuta más lentamente. En algunos casos, cuando se presiona un botón, transcurren varios segundos antes de que suceda algo.

No posee compositor visual para crear las conexiones. Tiene sólo un asistente que realiza las interacciones. Este se ejecuta desde el menú, en vez de ejecutarse desde el objeto fuente. Posee además, un cuadro de diálogo inamovible, motivo por el cual, en el caso de olvidar el nombre de alguno de los objetos, se deberá salir del asistente y comenzar nuevamente la operación.

2 - IBM Visual Age para Java Professional 1.0

Puntos Fuertes

Provee el mejor soporte para la programación visual. Permite definir interacciones visuales con componentes visuales y no visuales. Debido a que en el editor de composición visual pueden existir varias conexiones, y esto puede dificultar la visión de la pantalla, Visual Age permite, a través de filtros, mostrar sólo algunas de las conexiones. El resto de las herramientas que se mencionan, sólo permiten mostrar todas las conexiones o ninguna de ellas.

Su diseñador de GUI, permite tomar y arrastrar los objetos de la paleta para ubicarlos en el compositor visual sin conocer las restricciones de los layouts. Permite además seleccionar varios objetos para realizar operaciones conjuntas sobre todos ellos, que incluyen copiar, cortar y cambiar propiedades.

Una característica propia de Visual Age, es que permite setear la opción de Sticky, a través de la cual se permite colocar varias veces el mismo tipo de objeto, en el editor de composición visual.

Otra característica, es que permite cambiar el tipo de objeto, una vez que está ubicado en el contenedor.

Puntos Débiles

Uno de los mayores inconvenientes que se presentan al utilizarlo, es la integración con herramientas externas. Por ejemplo se deben exportar archivos y si desde la herramienta externa se realizan cambios en los mismos, se deben importar nuevamente.

El help que posee no es sensible al contexto y si se desea acceder al mismo, se estará ejecutando un servidor de help.

Su debugger sólo funciona en su propia Virtual Machine y detecta problemas que ocurren cuando se ejecutan en su ambiente de desarrollo.

3 - Java Workshop 2.0

Puntos Fuertes

Incluye las JFC. Su diseñador de GUI, permite tomar y arrastrar los objetos de la paleta para ubicarlos en el compositor visual sin necesidad de conocer las restricciones de los layouts.

Puntos Débiles

Seleccionar un objeto en el editor de compositor visual es complicado: hay que mover el cursor sobre el objeto y en el momento en que éste cambia de forma se puede manipular el objeto.

Es una herramienta muy lenta.

4- Symantec Visual Café

Puntos Fuertes

Tiene el único debugger que trabaja con la Máquina Virtual de Symantec, Netscape y de Microsoft. Esto es muy útil, ya que varios bugs del código pueden aparecer en una determinada Máquina Virtual.

Guarda automáticamente los archivos cuando se compila, o se modifica algún error mientras se ejecuta el debugger.

Puntos Débiles

La mayoría de los problemas que surgen con el Visual Café, son a raíz de errores de configuración. Generalmente el classpath apunta a librerías erróneas, no se dispone de suficiente memoria virtual o se tienen distintas versiones de dll's.

Se debe disponer de varios megas de memoria virtual para asegurar el buen funcionamiento del ambiente. Si se dispone de menos de 64 Mb. se torna demasiado lento.

CUADRO COMPARATIVO DE AMBIENTES

	General Borland Jbuilder 1.0 Professional	**IBM Visual Age for Java 1.0**	**Sun JavaWorkShop 2.0**	**Professional Symantec Visual Café Pro Dev Edition 2.1**
Plataformas soportadas para el desarrollo	Win95/NT	Win95/NT, OS2	Solaris, Win95/NT, HP-UX	Mac, Win95/NT
Programación Visual Interacciones	No.	Excelente. Puede crear y editar las interacciones entre todas las componentes, visuales y las no visuales. Puede mostrar las relaciones creadas.	No. Tiene limitada la interacción visual, aun con las componentes no visuales.	Buena: puede crear pero no editar ni ver las relaciones entre las componentes.
Soporte para Base de Datos	Sí.	Sólo el Enterprice	Limitado, posee más soporte en el Enterprice.	Limitado, posee más soporte en el Enterprice.
Documentación en línea	Muy buena en las clases de Borland, pero no es sensible al contexto.	No es sensible al contexto. Posee un help básico.	Sí	Básica.
Generación de Páginas HTML	Sí	Sí	Sí	Sí
Incorporación de objetos propios a la paleta	Sí	Sí	Sí	Sí
AFC o JFC	Ambos	Ninguno	JFC	
Soporte JDBC	Sí	Sí	Sí	Sí

VISUAL AGE PARA JAVA

Es un ambiente visual integrado, que soporta el ciclo completo de la construcción de un programa Java.

Puede utilizarse para la construcción de Applets y Aplicaciones a través de las características de la programación visual.

Visual Age para Java genera el código Java que implementa lo que se ha especificado en el editor de composición visual. Posee un tutorial que permite realizar trabajos, tales como la creación de Applets, creación de los elementos de programas, beans, importar y exportar código del sistema de archivos, entre otros.

ELEMENTOS DE PROGRAMA

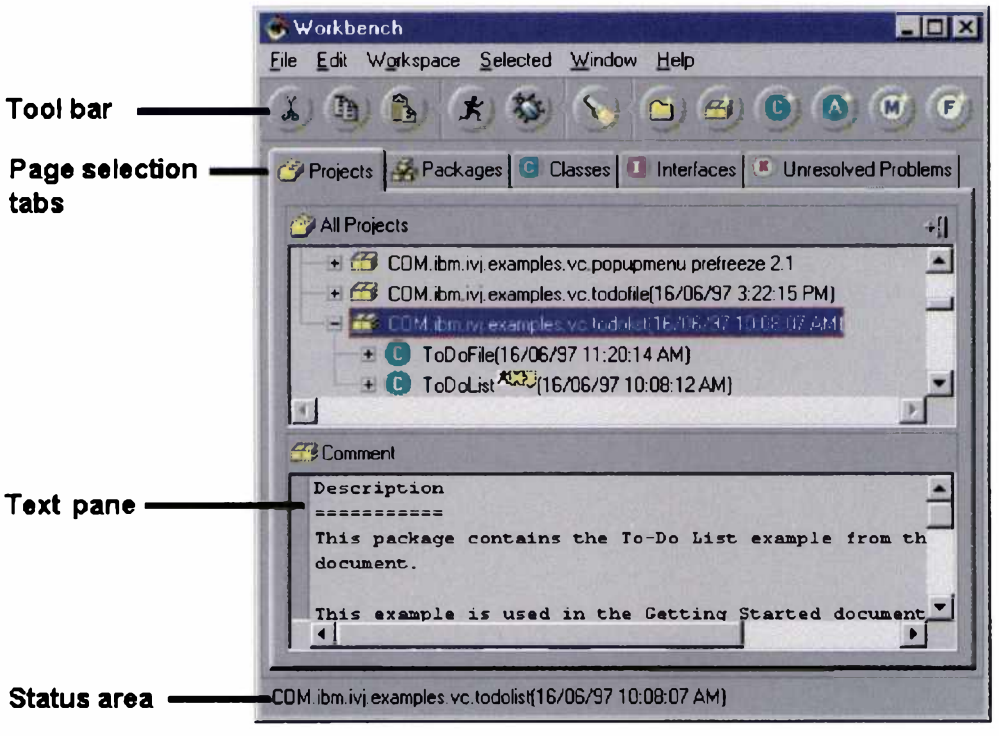
Dentro del ambiente no manipulamos archivos para el código Java, en su lugar, el código se guarda en una base de datos de objetos estructurada, y se nos muestra como una jerarquía de elementos de programas.

Visual Age para Java permite mantener varias ediciones de un mismo programa. Las ediciones se organizan dentro de dos áreas, el Workspace, que contiene el código sobre el que se está trabajando y las librerías que éste utiliza, y el Repository, que contiene todas las ediciones de todos los elementos de programa.

Debemos diferenciar entre el Workspace y Workbench. El Workspace es la ventana de interface que muestra los elementos de programa que se encuentran en el Workbench.

WORKBENCH

Es la ventana principal del ambiente, en la cual se ven todos los elementos de programa.



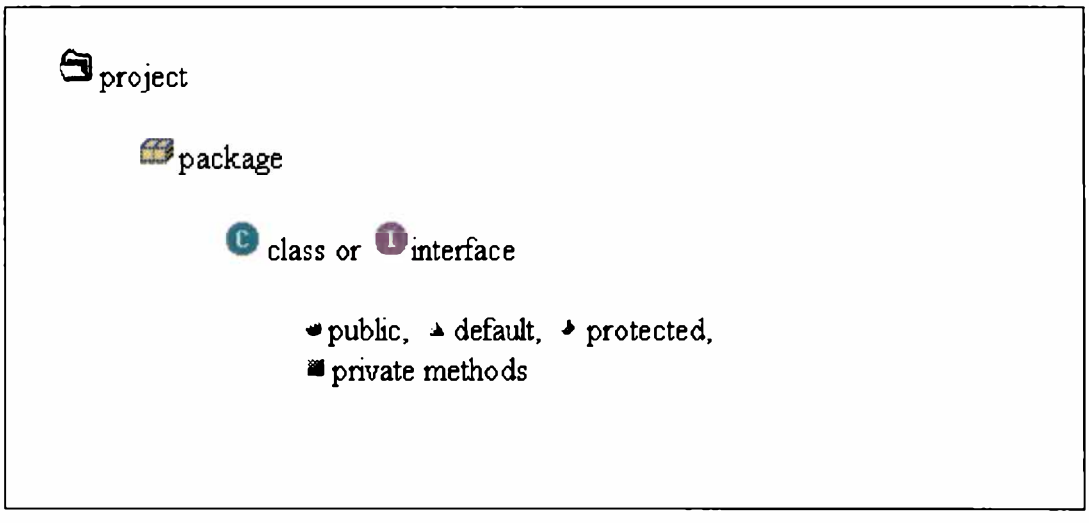
Barra de Herramientas

Se encuentra situada debajo de la barra de menú, y otorga un rápido acceso a los trabajos que se realizan más frecuentemente dentro de la ventana del Workbench. Estos trabajos incluyen edición, ejecución, debugging, búsqueda y manipulación de elementos de programas.

Páginas en la Ventana Workbench

Cada página brinda un punto de vista específico sobre el código en el Workspace.

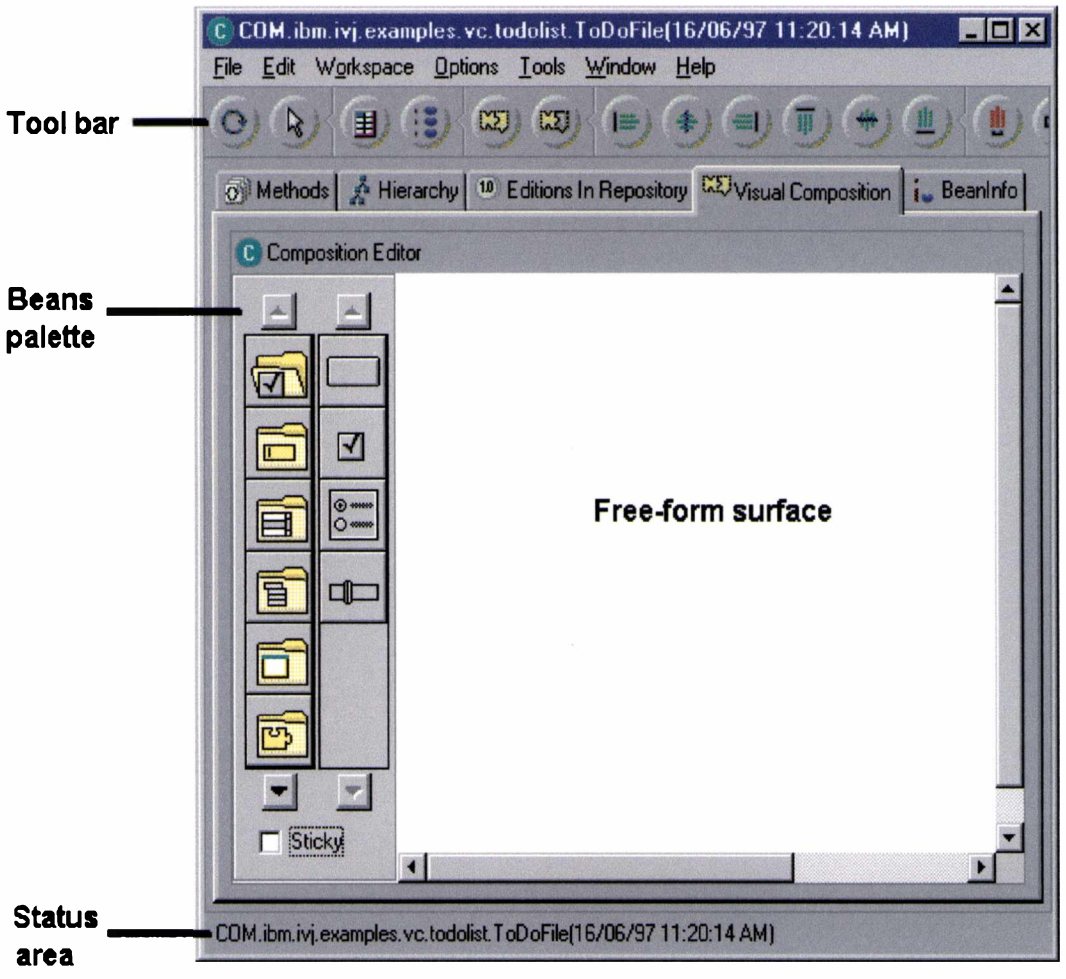
- Proyect: Es el elemento de programa que se encuentra en el mayor nivel. Muestra todos los proyectos en el Workspace. Se puede expandir y permite ver todos los elementos de programa que se encuentran dentro de él.
- Package: Muestra todos los paquetes dentro del Workspace.
- Classes: Muestra todas las clases, en una jerarquía ruteada a partir de Java.Lang.Object.
- Interfaces: Despliega todas las Interfaces dentro del Workspace.
- Unresolved Problems: Muestra todas las clases y métodos que tienen problemas que no han sido resueltos.



EDITOR DE COMPOSICIÓN VISUAL

Nos permite diseñar la interface del programa, especificar el comportamiento de los elementos de la interface y definir la relación entre la interface y los elementos de programa.

El Editor de Composición Visual nos brinda importantes herramientas para construir Beans.



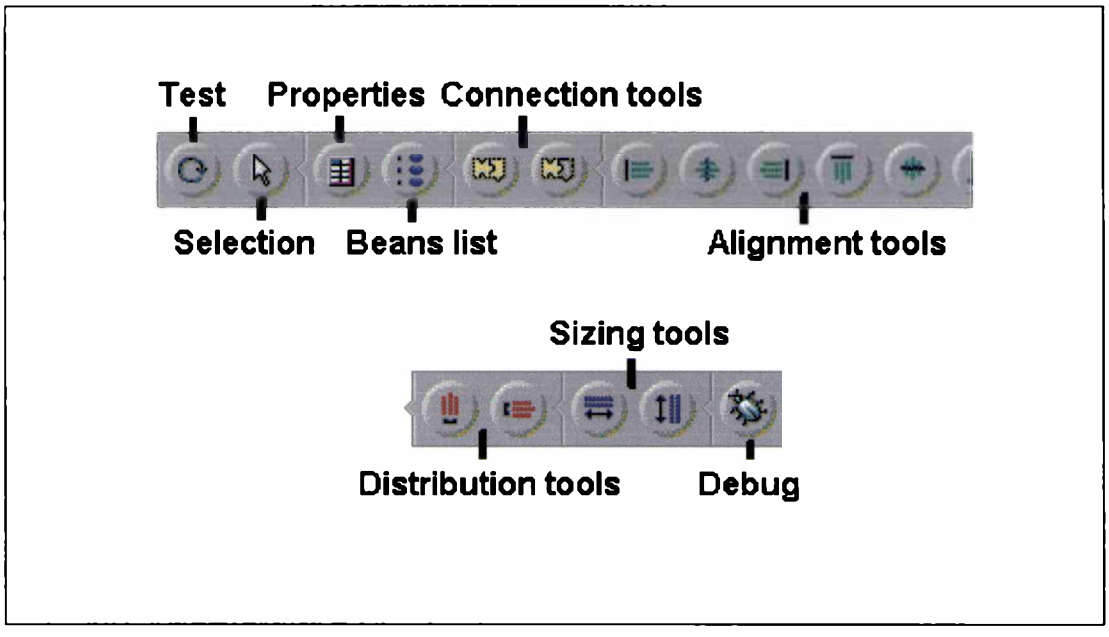
Cuenta con una *paleta de beans*, que está localizada en el área izquierda de la pantalla, ésta contiene el conjunto de Beans que se utilizan mas frecuentemente.

Se encuentran organizados dentro de Categorías de Beans.

El *área de estado*, que se encuentra en la parte inferior de la pantalla, indica la categoría y el bean seleccionado en la paleta o en el Free-form surface.

La *Barra de herramientas* provee un acceso rápido a las herramientas que más se utilizan para la manipulación de beans. Dichas herramientas permiten mostrar y ocultar conexiones entre beans, posicionar beans, modificar su tamaño y testear el programa.

Por último, *Free-form surface* es el área de programación visual donde se contruye el programa. Permite agregar, conectar y manipular beans.



Base de Datos

ORACLE 7

ORACLE es un manejador relacional de bases de datos (RDBMS).

CARACTERÍSTICAS GENERALES

- Relacional: es decir, la información se encuentra almacenada en tablas y se las puede manipular usando operaciones del álgebra relacional.
- Cliente/Servidor: una aplicación que desee manipular las tablas (un "cliente") solicita sus pedidos al manejador de la base (el "servidor"). Este tipo de arquitectura permite que los clientes y el servidor puedan encontrarse en máquinas distintas.
- SQL: los pedidos de los clientes se efectúan en SQL (Structured Query Language). SQL es un lenguaje estándar y declarativo. ORACLE cumple con el estándar internacional del SQL y agrega otras funcionalidades.
- PL/SQL: ORACLE posee el lenguaje de cuarta generación PL/SQL en el servidor y en diversas herramientas propias (Forms, Reports, etc.). PL/SQL es independiente de la máquina y/o sistema operativo, por lo que las aplicaciones que la utilicen son transportables.
- Transaccional: el servidor es transaccional, es decir, se pueden definir un conjunto de modificaciones a la base como atómicas: deben ser confirmadas (mediante un "commit") o no deben ser realizadas (en este caso se dice que la transacción es deshecha, se realiza un "rollback"). El mecanismo transaccional está siempre presente en ORACLE.
- Confiable: ORACLE es tolerante a fallas del sistema, por ejemplo, ante caídas del sistema (cortes de energía, etc.) los datos no se corrompen y las Transacciones confirmadas no se alteran. Posee también mecanismos de copia para que ante fallas de hardware (tales como roturas de discos) los datos se puedan recuperar completamente.

- Adaptable a la arquitectura: utiliza las facilidades que brinda el sistema operativo para hacer más eficiente su uso.
- Integridad de Datos: se pueden declarar restricciones que deban cumplir los datos (tales como claves primarias o foráneas) que son verificadas en todo momento por el servidor.
- Seguridad: el servidor posee usuarios con claves (independientemente de los usuarios del sistema operativo del servidor). Cada usuario posee sus propios objetos y existen sentencias SQL para otorgar o denegar permisos a otros usuarios. Los usuarios pueden agruparse en "roles" para poder manejar los permisos de una manera más sencilla.
- Concurrencia: el servidor permite realizar las transacciones de varios usuarios simultáneamente y manejando automáticamente los bloqueos necesarios para asegurar la consistencia de la información.
- Diversos tipos de objetos: además de los tipos de objetos que existen en el modelo relacional clásico, ORACLE posee un conjunto predeterminado de tipos de objetos más extenso, por ejemplo: usuarios, secuencias, clusters, snapshots.
- Distribución: varias bases de datos ORACLE pueden interconectarse de modo que una aplicación pueda acceder o modificar datos de distintas bases en una única transacción.
- Puentes a otras bases: los "puentes" (gateways) permiten manejar bases de datos no-ORACLE o manejadores de archivos como si fuesen bases ORACLE. Esto permite utilizar herramientas ORACLE sobre otros tipos de manejadores de información.
- Bases activas: ante modificaciones a las tablas se pueden especificar acciones (escritas en PL/SQL) para que se ejecuten en forma automática.
- Auditoría: diversos tipos de acciones (por ejemplo conexiones a la base, modificaciones a algunas tablas) pueden "auditarse", en el sentido de quedar registradas en una bitácora. En la bitácora figura la acción, la fecha y hora, el usuario que la ejecutó, etc.

- Diccionario de Datos: toda la información sobre los objetos creados en una base se encuentra en el "diccionario de datos", el cual es accesible a través de un conjunto predeterminado de vistas usando SQL.
- Administración: existen herramientas para facilitar la administración de una base de datos.

OBJETOS DE ORACLE

Objetos "físicos"

Los objetos físicos son áreas de discos en donde se almacenan los segmentos (tablas, índices, etc.). Los segmentos son objetos que tienen la posibilidad de crecer ilimitadamente.

Tablespaces y datafiles

Cada base de datos ORACLE está conformada por unidades de almacenamiento denominadas "tablespaces" (espacio para tablas). Cada base de datos posee al menos un tablespace llamado "SYSTEM" en donde se almacena el diccionario de datos. Los usuarios pueden crear si lo desean otros tablespaces.

Cada tablespace está conformado por un grupo de archivos llamados "datafiles" (archivos de datos). Cuando se crea un tablespace se indican los datafiles que lo componen junto al tamaño de cada uno de ellos. Si un tablespace queda sin espacio, entonces se lo puede modificar agregándole nuevos datafiles. Los datafiles no pueden ser eliminados, sólo se puede eliminar un tablespace completo. La versión 7.2 posee una cláusula en la declaración de un datafile que indica como puede expandirse luego de ser llenado.

La filosofía del manejo del espacio físico en una base ORACLE es "preasignar" espacio antes de usarlo. En esos espacios creados pueden convivir muchos segmentos.

Bloques, extents y crecimiento de los segmentos

La unidad de tamaño de los datafiles son los bloques. El tamaño de un bloque se indica al crear una base de datos, luego es inalterable. Generalmente es de 2k.

Un segmento (tabla, índice, etc.) se aloja en un único tablespace, aunque puede estar diseminado en varios datafiles. Un segmento está formado por "extents" (extensiones). Cada extents se aloja en un único datafile y su tamaño está dado también en bloques.

Cuando se crea un segmento se le asigna un primer extent. Los rollbacks por defecto toman dos a menos que se especifique la cláusula MINEXTENTS con otro valor.

Cuando el segmento crece y se queda sin espacio se le van asignando nuevos extents. El tamaño de cada uno de ellos está dado por los siguientes parámetros que se especifican al crear un segmento:

INITIAL: tamaño del primer extent

NEXT: idem para el segundo extent

PCTINCREASE: porcentaje de incremento con respecto al extent anterior (para el tercer extent en adelante).

Los valores de NEXT y PCTINCREASE pueden modificarse luego de creado un segmento (no el INITIAL). Los rollback segment asumen un PCTINCREASE de 0 (no es modificable).

Los extents asignados nunca son desasignados, salvo que:

- Se ejecute un TRUNCATE.
- Se trate de un segmento de rollback.
- Se borre el segmento.

Existe una cantidad máxima de extents que puede poseer un segmento, puede restringirse con la cláusula MAXEXTENTS, sin embargo conviene tener pocos extents para mejorar la performance.

Por ejemplo, la sentencia:

```
create table clientes (
    num         number(6),
    nombre      varchar2(30)
) storage ( initial 2m next 1 m pctincrease 30)
tablespace datos;
```

Crea una tabla en el tablespace datos con un primer extent de 2 megabytes. Cuando la tabla vaya creciendo se le irán asignando los extents:

2ndo 1 mega

3ero 1,3 mega

4rto 1,69 mega

etc.

Conviene conocer a priori la cantidad de filas que poseerá una tabla para poder predecir su tamaño. Si los storage no son adecuados, se corre el riesgo de no poder asignar extents porque el tablespace se encuentre lleno o fragmentado. El primer caso se soluciona agrandando el tablespace (se agrega un nuevo datafile). En el segundo hay que exportar los objetos, borrarlos y volver a importarlos.

Si no se especifica tablespace en la creación de un segmento, se asume el tablespace por defecto del usuario.

Si no se especifica storage asume el storage por defecto del tablespace.

SEGMENTOS

Tablas

Corresponden a las tablas del modelo relacional.

Indices

Son estructuras que permiten acelerar búsquedas. Se asocian a columnas de tablas o de clusters.

Clusters

Conjunto de tablas agrupadas en un solo segmento. Las tablas son almacenadas de forma que las filas de las tablas que coinciden en ciertas columnas (tienen el mismo valor) son puestas en los mismos bloques del segmento. Es decir, se almacenan con el "join calculado".

Las ventajas de un cluster son que mejoran la performance de los joins y permiten ahorrar espacio. Nótese que otras operaciones pueden ser menos eficientes.

Para crear un "cluster" se debe:

- Ejecutar la sentencia create cluster indicando un conjunto de columnas.
- Crear el índice del cluster. Para ello se debe indicar en la sentencia create index que éste pertenece a un cluster y si el índice es un B-tree o un "hash".
- Crear las tablas del cluster mediante un create table con la cláusula cluster nombre (columnas).

Luego las tablas del cluster se utilizan de la manera habitual (mediante inserts, updates, selects, etc.)

Rollback segments

Para poder deshacer transacciones, el manejador debe almacenar la información sobre los valores anteriores de los datos. Esta información (temporaria) es escrita en los "rollback segments".

Un rollback segment puede ser escrito por varias transacciones simultáneamente.

Al iniciarse una transacción que modifica datos, el servidor elige un rollback para la transacción, a menos que el usuario haya indicado un rollback específico mediante la sentencia:

```
set transaction use rollback segment nombre.
```

Los rollbacks pueden ser públicos o privados. Para crear rollbacks públicos se agrega la cláusula public en la sentencia DDL. Tener ambos tipos de rollbacks se justifica en las arquitecturas que permiten tener varias instancias ORACLE sobre la misma base (es decir se posee varias CPUs sobre la misma base de datos). En estos casos los rollbacks públicos son utilizados por todas las instancias y los privados son repartidos entre las distintas instancias.

En ORACLE7 los rollbacks pueden activarse o desactivarse sin cerrar la Base de Datos. Toda base tiene el rollback "SYSTEM".

Si se desea crear objetos en un tablespace que no sea el SYSTEM, se debe crear al menos otro rollback.

Segmentos temporarios

Ante ciertas operaciones de manipulación de datos, cuando no hay índices adecuados, el servidor debe crear segmentos para poder realizar la sentencia. Estos segmentos son luego borrados al finalizar la operación.

Son creados en el tablespace asociado a tal efecto al usuario que realizó la operación. El storage es el que tenga asignado por defecto el tablespace en donde se almacena el segmento.

Snapshots

Los snapshots son tablas o consultas (resultados de select) que se copian a otra base de datos. Su utilidad consiste en tener acceso localmente a tablas remotas con una mínima cantidad de transferencias entre bases. Los snapshots son solamente de consulta.

OBJETOS VARIOS

Vistas

Corresponden a las vistas del modelo relacional. Una vista es una estructura que proporciona el resultado de una consulta.

Secuencias

Son estructuras que almacenan un valor numérico y poseen como únicas operaciones: tomar el siguiente valor (con un incremento preestablecido) o tomar el valor corriente. El incremento de una secuencia se realiza como una transacción independiente, es decir, deshacer una transacción no resta el incremento a la secuencia.

Usuarios

Una base de datos contiene varios usuarios con sus claves. Inicialmente (cuando es creada) una base de datos posee los usuarios SYSTEM y SYS (usuario dueño del diccionario de datos).

Permisos y Roles

Cada usuario posee sus propios objetos, sobre los cuales puede realizar cualquier operación. Si un usuario desea utilizar objetos de otro, entonces se le debe asignar los permisos correspondientes. Cada tipo de objeto posee una lista predeterminada de permisos para otorgar o denegar.

Además de los permisos sobre objetos, existen permisos generales para realizar ciertas operaciones. Por ejemplo el permiso drop any table permite a un usuario borrar cualquier tabla de cualquier usuario. Estos permisos son llamados "privilegios de sistema".

Los roles son conjuntos de permisos (de objetos o privilegios) a los que luego se los puede otorgar (o denegar) a usuarios. Los roles pueden ser jerárquicos, es decir, se puede otorgar a un rol los permisos de otro rol.

Profiles

Es un conjunto de limitaciones a los recursos que puede utilizar un usuario.

Las acciones que pueden limitarse son, entre otras:

- Cantidad máxima de tiempo que puede permanecer conectado un usuario.
- Cantidad máxima de tiempo en que el usuario no realiza operaciones a la base.
- Cantidad de CPU que insume una sentencia SQL.
- Cantidad de E/S que insume una sentencia SQL.

Database links

Para poder utilizar datos de bases remotas, es necesario crear un objeto llamado "database link" (conexión a otra base) que identifique:

- la base remota: nombre de la máquina, nombre de la base, protocolo, etc.
- (opcional) nombre de un usuario y su password en la base remota.

Luego de creado el database link se puede referenciar tablas remotas con la sintaxis: `tabla @ database link`

Sinónimos

Los sinónimos son "alias" para objetos de ORACLE (como tablas, vistas,secuencias, etc.). El "alias" sustituye a un usuario, objeto y opcionalmeme un database link.

Por ejemplo:

```
select * from pedro.empleados@rosario
```

(datos de la tabla empleados del usuario pedro en la base rosario) se puede simplificar creando el sinónimo emp:

```
select * from emp
```

Los sinónimos no sólo facilitan la escritura sino que además permiten tener una transparencia y uniformidad referencial en los programas, es decir, no hay que estar indicando siempre quién es el dueño de los objetos ni en qué base se encuentran.

Procedimientos, funciones y paquetes

Programas en PL/SQL (versión 2) pueden ser almacenados en la base como procedimientos, funciones o paquetes (conjunto de funciones, procedimientos, constantes, variables, etc.). Estos programas pueden ser luego accedidos por cualquier operación, inclusive por bases de datos remotas.

Al crearse, estos programas son compilados y almacenados en la base.

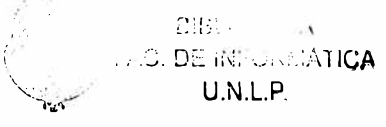
Triggers

Son programas PL/SQL que se ejecutan al realizarse alguna operación sobre una tabla. Se puede especificar bajo qué operaciones y/o columnas se debe "disparar" el trigger, y si se ejecuta una vez por cada fila afectada o una vez por sentencia.

Dependencia y estado de los objetos

ORACLE7 mantiene la dependencia entre los objetos definidos, esto es, para cada objeto determina cuales son los objetos sobre los cuales está construido.

Cuando un objeto sobre el que depende es borrado, cambiado, etc. éste queda con un estado de "inválido". Al intentar usar un objeto inválido, ORACLE intenta primero "recompilarlo" (esto vale no sólo para las funciones y procedimientos sino también para vistas, triggers, etc.), y si queda con estado válido entonces se lo puede utilizar. Existe además una sentencia para compilar objetos (alter objeto compile). Las tablas y sinónimos son objetos "base" de las dependencias, es decir, no dependen de otro objeto ni pueden recompilarse.



BIBLIOGRAFÍA Y REFERENCIAS

- *Core Java*, Gary Cornell & Cay S. Horstmann, Sunsoft Press, 1996.
- Revista *Datamation*, Vol. 5 - N° 3, 1996.
- Seminario de *Desarrollo de Aplicaciones en Java*, INFOCOM ' 96.
- Seminario de *Java*, Technology Training Corporation, Noviembre de 1996.
- Tutorial de JAVA - CACIC 1997 (U.N.L.P) - La Plata
- Archivo de Ayuda del Visual Age for JAVA.
- Revista Netmanía, N° 21, 1997.
- http:\\www.javasoft.com\products
- http:\\www.ibm.com
- http:\\www.java.sun.com
- http:\\www.redbooks.ibm.com
- http:\\www.edm2.com
- http:\\www.netscape.com
- http:\\www.oracle.com

Nota: Los Logos utilizados en los encabezados de página del presente informe pertenecen a Sun Microsystem, IBM y Oracle respectivamente.