UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES

ARAGON

"DESARROLLO DE APLICACIONES CON TECNOLOGIA JAVA"

T R A B A J O E S C R I T O

EN LA MODALIDAD DE SEMINARIOS Y CURSOS DE ACTUALIZACION Y CAPACITACION P R O F E S I O N A L

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

MIGUEL ANGEL MARTINEZ MELENDEZ

ASESOR:
M. EN C. JESUS HERNANDEZ CABRERA

MEXICO, 2007

CONTENIDO

INTRODUCCÍON

El lenguaje java no es solo un lenguaje, es una tecnología basada en clases y objetos, todo en Java (salvo algunos tipos primitivos) es un objeto. Encontramos lenguajes híbridos como C++ o Visual Basic, en Java no se permite programar fuera de un objeto o clase. Ya que no cuenta con módulos o funciones globales. Una de las características todavía más distintiva dentro de Java, es su capacidad multiplataforma. Lenguajes como C++ o incluso el visual basic se han implementado en múltiples plataformas, pero siempre han necesitado recompilaciones o adaptaciones al pasar de una a otra. Sin embargo, Java desde el principio ha sido pensado para adaptarse a varios entornos. Esto lo consigue, gracias a su maquina virtual, no sólo a nivel de código fuente, sino también a nivel de código compilado.Cualquier programa que desarrollemos se puede compilar en Windows o en Linux, inclusive en mac y funciona. Y Aún mas, hasta el programa compilado puede ejecutarse sin más preparación, en distintas máquinas. Esto se puede conseguir por que la tecnología java, se compila y ejecuta, sin ningún entorno en particular, sino en lo que los creadores de java han llamado una "Java Virtual Machine", (JVMl). Solo se necesita descargar la JVM para nuestra plataforma y podrá ejecutarse en cualquier sistema operativo. Al estar basado en clases y objetos, viene acompañado de un conjunto de éstos, que a su vez se agrupan en paquetes de clases que componen la librería Java estándar.

Como ingeniero en computación al aspirar a un trabajo o un nivel de jerarquía mejor se requiere una preparación mayor, pará sobresalir de los demás y justificar el porque debe ser este el escogido. Los ingenieros son absoluta y completamente aplicados y prácticos. Su dominio es la vida real. Estudian matemáticas, cálculo, física y un sin número de otras ciencias, pero sólo para esperar entender los problemas de la vida real y ofrecer soluciones útiles. Java es una de las tecnologías que nos sirven para poder enfrentar esos problemas de la vida real, ya que las empresas necesitan gestores en tecnología y no genios en Computación, es importante estar actualizado ya que día con día aparecen nuevas tecnologías, lo que en su momento fue un lenguaje de innovación ahora se convierte en obsoleto y da paso a nuevos lenguajes como java.

Java nos sirve a todos a mí como ingeniero, me sirve desde administración de la vida diaria, aplicaciones relacionadas a viajes, herramientas de información, aplicaciones empresariales, juegos interactivos y un sin fin de cosas mas.Tal vez la gente común por llamarla de algún modo, que se siente despegada de la computación, pero que forzosamente se tiene que comunicar de alguna u otra manera se darán cuenta que al comprar un teléfono la mayoría de estos tienen todas las aplicaciones desarrolladas en java. Ya que las posibilidades ofrecidas por la tecnología Java son prácticamente ilimitadas y muchos desarrolladores de aplicaciones están aprovechando la oportunidad para colocar su talento creativo a servicio del avance del mundo de la comunicación inalámbrica.

Para poder realizar nuestros primeros programas dentro de java podemos bajar tutoriales en español, sí no se nos facilita el inglés, se puede usar un entorno integrado de desarrollo (IDE), como Visual Age, Jdeveloper, Eclipse, NetBeans, etc. pero si hasta ahora estás empezando se recomienda que no uses ninguno de ellos, porque tu objetivo ahora es aprender a programar en Java y no luchar con todas las idiosincrasias que tienen esos entornos.

Características de Java

El lenguaje Java cuenta con una serie de características que lo hacen interesante, las cuales se pueden resumir en la siguiente tabla:

Simple

• Simple en su sintaxis, por lo que es más sencillo para programar.
• Tiene un recolector de basura que le permite al programador despreocuparse de liberar memoria.
• Elimina características que provocan errores en otros lenguajes como C y C++.
• Su intérprete es bastante pequeño.

Orientado a Objetos.

• Soporta encapsulación, herencia y polimorfismo.
• Cuenta con resolución dinámica de métodos al modo de Objetive-C.
• Proporciona soporte para identificar clases en tiempo de ejecución.

Características de interconexión.

• Cuenta con extensas capacidades de interconexión mediante TCP/IP.
• Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp.

Robusto.

• Realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución.
• Es fuertemente tipificado. La comprobación de tipos en Java ayuda a detectar errores lo antes posible en el ciclo de desarrollo.
• Obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error.
• Administra la memoria para eliminar el trabajo del programador correspondiente a la liberación o corrupción de memoria.
• Cuenta con soporte para manejo de excepciones.
• Genera byte-codes, código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

Portable.

• Implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros se representan con 32 bits en complemento a 2.
• Construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

Seguro.

• Java elimina características como los punteros o el casting implícito, mecanismo que usa para asociar datos el compilador de C y C++, con el fin de prevenir el acceso ilegal a la memoria.
• Los ambientes de ejecución de Java aplican un probador de teoremas a los programas, para detectar fragmentos de código ilegal, como lo sería el tipo de ataque Caballo de Troya. Además de confirmar que el código no produce desbordamiento de operandos en la pila, así como vigilar que no hay intento de violar los accesos de seguridad establecidos ni conversiones ilegales.
• El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de archivos local del de los recursos procedentes de la red. Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen de las clases importantes. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego

en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.
- Cuenta con firmas digitales, que se verifican antes de crear cualquier objeto.
- Imposibilita, también, abrir algún archivo de la máquina local.
  Siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el applet
- No permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra.
- Sin embargo, puede ser descompilado. Neutral a la arquitectura.
- El compilador Java genera su código a un archivo objeto de formato independiente de la arquitectura de la máquina en que se ejecutará.
- Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.
- Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows '95, Windows NT, XP, Vista Linux, Irix, Aix, Mac, Apple y probablemente existan grupos de desarrollo trabajando en portar Java hacia otras plataformas.

Interpretado.
- El intérprete Java puede ejecutarse en diversas plataformas, Java es más lento que otros lenguajes de programación por no ejecutar directamente el código objeto sino un código semiinterpretado. Por ahora, que todavía no hay compiladores de una plataforma específica, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional. Java es de 10 a 30 veces más lento que C.

Multihilo.

Al ser Multihilo, Java permite muchas actividades simultáneas en un programa. Los hilos, a veces llamados, procesos ligeros o hilos de ejecución, son pequeños procesos o piezas independientes de un gran proceso.

Al estar estos hilos construidos en el mismo lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++. El beneficio de ser multihilo consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) de la plataforma, aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento.
- Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo de un sitio interesante desde la red. En Java, las imágenes se pueden ir trayendo en un hilo de ejecución independiente, permitiendo que el usuario pueda acceder a la información de la página sin tener que esperar al navegador.

Dinámico.
- No intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales -siempre que mantengan el API anterior.
- Simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, Java es capaz de traer automáticamente cualquier pieza que el sistema necesite para funcionar.
- Para evitar que los módulos de ByteCode o los objetos o nuevas clases, haya que estar trayéndolas de la red cada vez que se necesiten, Java implementa las opciones de persistencia, para que no se eliminen cuando se limpie la caché de la máquina.

Este trabajo se desglosara en tres partes, para definir de una manera sencilla el entorno de la tecnología Java. En el capitulo 1 de este trabajo se dara una breve introduccíon de la programacíon orientada a objetos utilizando el lenguaje de java, en el capitulo dos veremos la clases mas utilizadas en java para realizar aplicaciones dentro de la tecnología J2SE y por ultimo en el capitulo 3 veremos las tecnologías para desarrollo empresarial de J2EE.

OBJETIVOS

El presente trabajo tiene un objetivo general y dos objetivos específicos. El objetivo general del trabajo es el desarrollo de conceptos y principales tecnologías del lenguaje j2EE y j2SE donde se aprenderá diferenciar los tipos de aplicaciones, el uso y sus componentes principales

Para alcanzar este objetivo general, se han perseguido dos objetivos específicos necesarios. Así, el primer objetivo especifico es la de estudiar y utilizar herramientas y tecnología J2SE para diferenciar, cuándo nos enfrentemos a un problema, cuáles son sus alcances y que herramientas nos sirven para cubrir la solución de este problema

El segundo objetivo especifico es la de estudiar y utilizar herramientas y tecnología J2EE para implementar Servlets y páginas JSP para aplicaciones empresariales y así poder diferenciar los alcances de una tecnología y otra. Tomando en cuenta que una aplicación empresarial es aquel paquete de software que nos ayude en el desempeño de nuestro negocio. Con una aplicación empresarial podemos simplificar y automatizar tareas para aumentar el rendimiento de los procesos del negocio; pero también podemos usar una aplicación de este tipo para expandir el negocio que ya tenemos o incluso transformarlo

Justificación
Entre las motivaciones que llevaron al planteamiento de este trabajo están:

- La capacidad de aprender de acuerdo a las nuevas tecnologías que se presentaron en el diplomado de java Master.

- Contribuir con las soluciones al problema de suministrar material de apoyo para las próximas generaciones en español.

- Crear un proyecto integrador utilizando los conocimientos adquiridos durante el diplomado Máster en Java

- Cubrir con el trabajo del Diplomado Máster en Java como requisito para la titulación.

1.1 TECNOLOGIA ORIENTADA A OBJETOS

La **Programación Orientada a Objetos (POO u OOP** según siglas en inglés) es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan *estado* (es decir, datos), *comportamiento* (esto es, procedimientos o *métodos*) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar. De esta forma, un objeto contiene toda la información, (los denominados atributos) que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases (e incluso entre objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos). A su vez, dispone de mecanismos de interacción (los llamados métodos) que favorecen la comunicación entre objetos (de una misma clase o de distintas), y en consecuencia, el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos).

Dada esta propiedad de conjunto de una clase de objetos, que al contar con una serie de atributos definitorios, requiere de unos métodos para poder tratarlos (lo que hace que ambos conceptos están íntimamente entrelazados), el programador debe pensar indistintamente en ambos términos, ya que no debe nunca separar o dar mayor importancia a los atributos en favor de los métodos, ni viceversa. Hacerlo puede llevar al programador a seguir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen esa información por otro (llegando a una programación estructurada camuflada en un lenguaje de programación orientado a objetos).

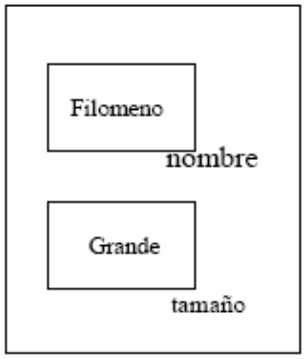
Esto difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada se escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciendo que realicen esos métodos en sí mismos.
La programación orientada a objetos introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Objeto: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). Corresponden a los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa).
- Clase: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- Método: algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- Evento: un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.
- Mensaje: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
- Propiedad o atributo: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto, y cuyo valor puede ser alterado por la ejecución de algún método.
- Estado interno: es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
- Componentes de un objeto:atributos, identidad, relaciones y métodos.
- Representación de un objeto: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una "variable", no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.
La programación orientada a objetos tiene varias características interesantes e importantes, sin embargo, el punto de partida es la encapsulación, es decir, empaquetar los datos con el código que les modifica. La encapsulación era una tentación desde antes del 80’s, porque permitiría dos cosas, por un lado mejorar la comprensión del código escrito y por otro lado reducir su costo de depuración. Sin embargo en un lenguaje era cara su implementación desde el punto de vista computacional. Pero como todos hemos experimentado, el avance en la eficiencia de los procesadores y otros dispositivos, hicieron viable este paradigma que parecía ser una moda.
En esta década, la orientación a objetos ha ido madurando y mejorando la eficiencia de sus algoritmos, proporcionando nuevos sabores y nuevos lenguajes. Java como un lenguaje “producto” de C++ y C Objetivo, apoya una buena parte de los conceptos de orientación a objetos. Los primeros conceptos que vamos a revisar son los conceptos de clase y objeto.
Un objeto o instancia es un concepto concreto del mundo real que tiene caracteristicas como:

- Unidad. Es decir que percibimos al objeto como un todo.
- Identidad. Cada objeto es único.
- Comportamiento definido. Significa que existirán funciones o rutinas bien definidas para manejar el objeto.



Ejemplo del objeto unLeón

Por medio de los lenguajes orientados a objetos, podemos representar a cada objeto como una variable que agrupa datos en otras variables llamadas atributos, y funciones que actúansobre los atributos y que determinan el comportamiento del objeto.
La clase, por otro lado, es una plantilla que unifica a una serie objetos que comparten las mismas características y comportamiento. De este modo, podemos hablar, por ejemplo, de la clase Pizzería y el objeto la pizzería de la esquina.Cuando se habla de un objeto, hablamos de manera concreta y cuando se habla de manera abstracta o general, hablamos de la clase. No es válido definir algún objeto si de antemano no hemos definido la forma de la clase.
Para representar una clase en Java, utilizamos la palabra reservada class. Por ejemplo si tenemos una clase llamada manzana, que representará a todos los objetos de su clase, la cual describe los atributos o características de la clase, como peso, color de la cascara, precio, etc. que compartirán cada uno de los objetos de la clase . Por supuesto que si dos objetos tienen los mismos atributos, no significa que los valores de esos atributos sean iguales, una manzana podría tener su atributo peso en 200 gramos, mientras que otra manzana lo podría tener en 250 gramos.Tambien tenemos las operaciones o metodos.Una operación es el medio por el cual vamos a manejar cualquier objeto, es similar al concepto de función pero asociado a los datos, también por eso a menudo se dice que es la parte dinámica o de comportamiento de la clase.

1.1.1 UML

Es un lenguaje que nos sirve para:
1. Modelar
2. Especificar
3. Construir
4. Documentar

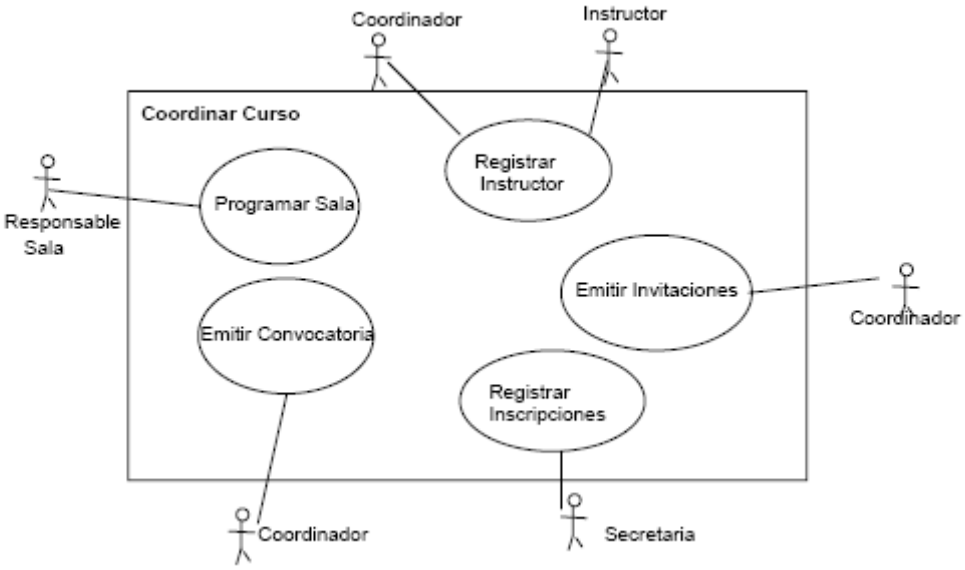
A continuacion veremos de forma breve los diagramas mas utilizados en UML y sus objetivos

1.1.2 Diagramas en UML

-Diagramas de casos de uso
-Diagrama de clases
-Diagrama de Objetos

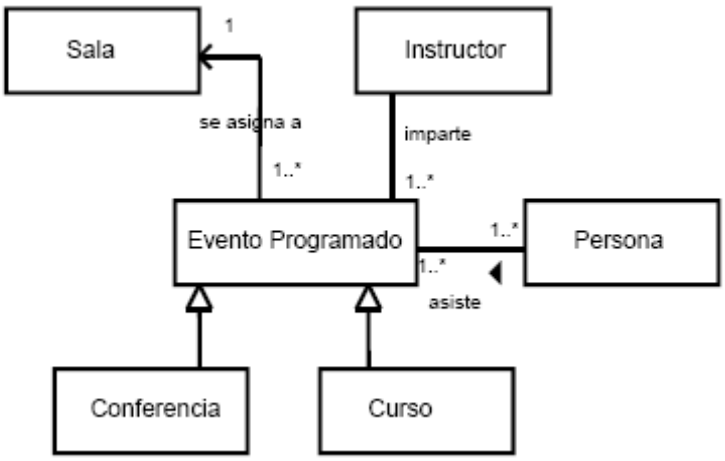
-Diagrama de Secuencia
-Diagrama de Colaboración
-Diagrama de Transición de estados
-Diagrama de Actividad
-Diagrama de Componentes
-Diagrama de Distribución
-Diagrama de Paquetes

Diagrama de casos de uso



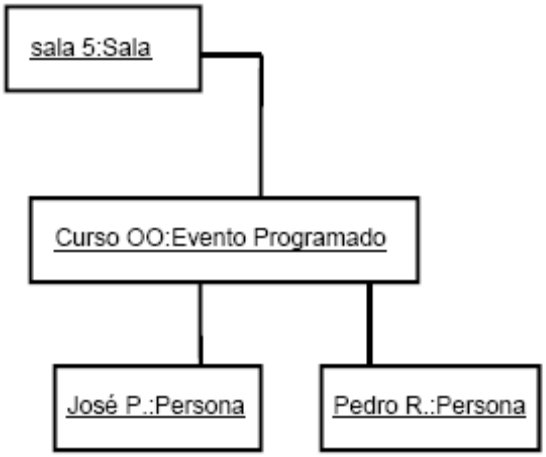
Objetivos:
• Ilustrar los roles tomados en una situación por los usuarios (actores)
• Modelar los procesos que un actor requiere de un sistema desde su propia perspectiva

Diagrama de clases



Objetivos:
• Modelar las clases participantes
• Identificar sus atributos
• Identificar sus operaciones
• Mostrar relaciones estáticas entre clases

Diagrama de objetos



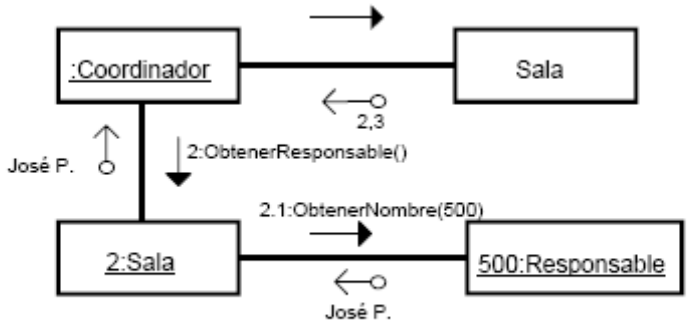
Objetivos:
• Mostrar una posible configuración del sistema
Nota:
• Corresponde con el diagrama de clases

Digramas de interaccion
Objetivos:
• Mostrar comportamiento a través de un escenario
Notas:
• Un escenario representa una situación típica de un proceso o función del sistema. En un automóvil, por ejm., quedarse sin gasolina, o acelerar en 3ª velocidad.

• Un diagrama de interacciones muestra interacciones de objetos en un escenario. Los objetos interactúan enviándose mensajes para la acción: El chofer oprime el acelerador, que se comunica con sistema de inyección de combustible, que a su vez se comunica con el motor, etc.
• UML soporta 2 formas de diagramas de interacciones:

Diagramas de Colaboración
Diagramas de Secuencia

Diagrama de Colaboración
Obtener Sala Disponible y Responsable



Objetivos:
• Mostrar un grupo de objetos y ligas describiendo un escenario.
• Mostrar eventos (mensajes) pasando entre los objetos. Puede incluir retorno de resultados.
• Mostrar el orden relativo de los eventos. Los llamados anidados son mostrados por numeración anidada.

Diagrama de secuencia

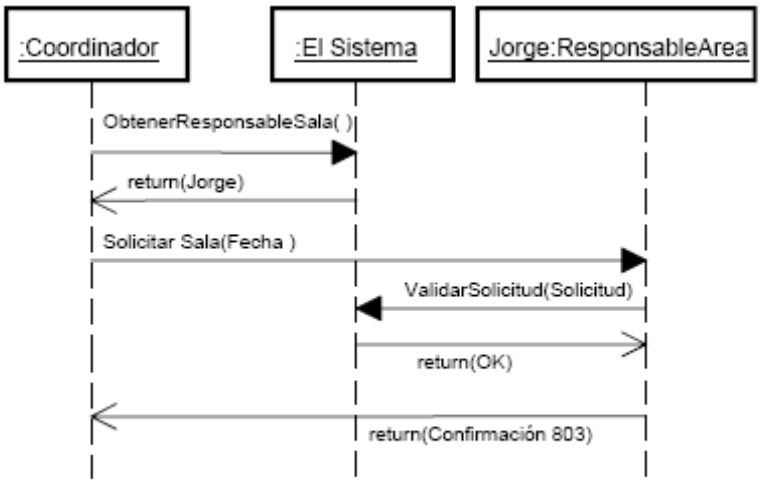
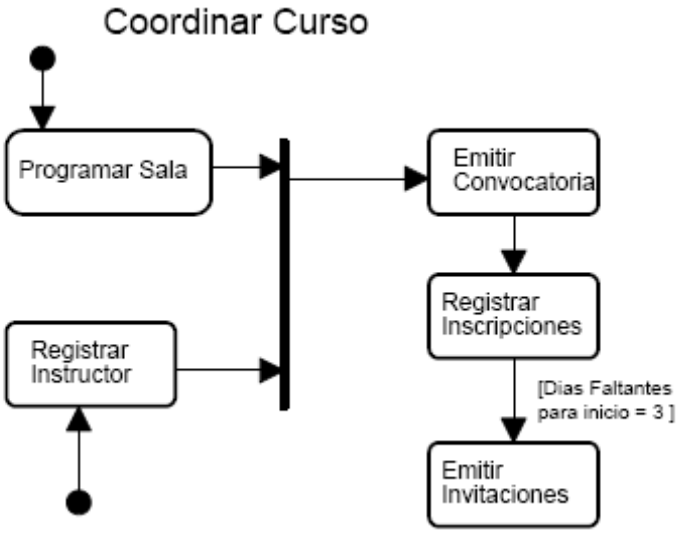


Diagrama de transición de estados

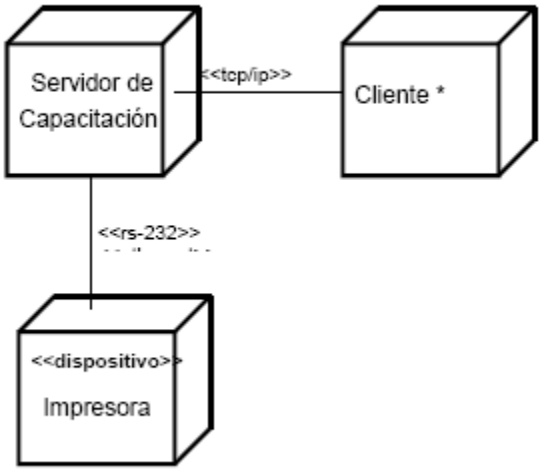
Objetivos:
• Mostrar estados. eventos y transiciones.
• Describir ciclos de vida de objetos
• Identificar operaciones relacionadas con estados y eventos

Diagrama de actividad



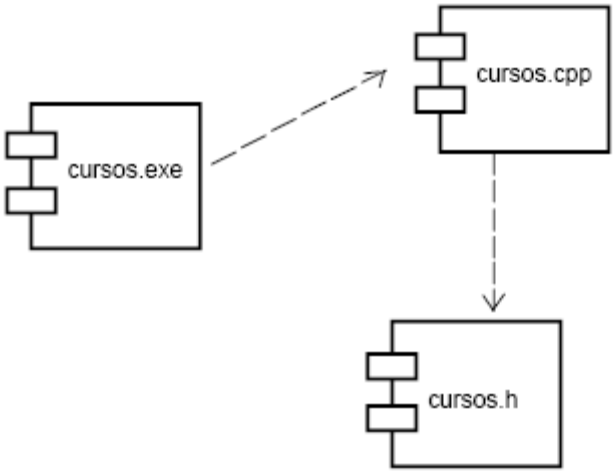
Objetivos:
• Mostrar relaciones temporales entre actividades
• Mostrar flujo de trabajo (workflow) entre casos de uso
• Identificar Paralelismo

Digrama de distribucion



Objetivos:
• Mostrar la arquitectura de distribución
•Mostrar los nodos de hardware: procesadores y/o dispositivos
•Mostrar las conexiones físicas entre los nodos

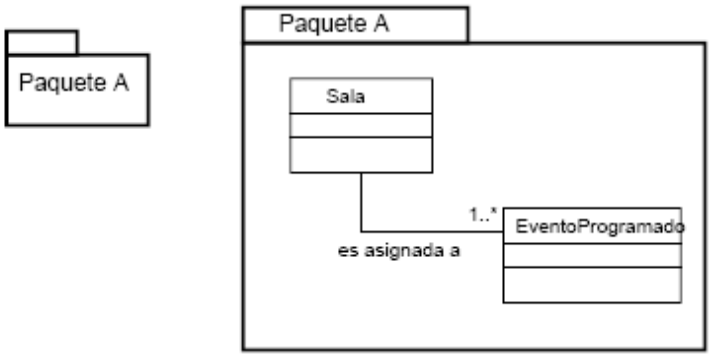
Diagrama de componentes



Objetivos:
• Representar paquetes físicos:

Componentes reusables
Archivos fuente
Archivos ejecutables
Librerías
etc.

Diagrama de paquetes



Es un grupo de elementos modelo (clases, otros paquetes, etc.)

Objetivos:
• Agrupar diversos tipos de cosas: Módulos, modelos, subsistemas, grupos físicos, etc.

1.2 CONCEPTOS DE PROGRAMACION ORIENTADA A OBJETOS CON JAVA

1.2.1 Uso y declaraciones de objetos.

Para poder efectuar las distintas operaciones con que cuenta un objeto, tenemos primeramente que crear o instanciar el objeto y posteriormente invocar cada una de ellas mediante el operador ".". Este último paso se conoce como el envío de mensajes, la forma en que se pueden crear nuevos objetos, es por medio del operador new, <Clase> <objeto>= new <Clase>(); //Aunque esto se verá posteriormente conmás detalle. Y que una operación sólo se puede invocar, mediante algún objeto y el operador punto.

1.2.2 Sobrecarga de operaciones.

La sobrecarga es la capacidad del lenguaje de soportar la coexistencia de dos o más operaciones con el mismo nombre. El medio por el cual se sirve el compilador para distinguir entre una operación y otra, es la diferencia en el número de argumentos así como los tipos respectivos de los argumentos que recibe cada operación. Al sobrecargar una operación se tiene que evitar ser ambiguo, es decir que al invocar a una operación, no quede claro a cual de las operaciones sobrecargadas se refiere el llamado, como es el erróneo caso de operaciones que sólo se distinguen por el tipo del valor de retorno.

1.2.3 Visibilidad o control de acceso.

Cuando utilizamos una grabadora, queremos reproducir una grabación, sin necesidad de conocer como se realiza el proceso. Del mismo modo, el principio de encapsulación nos dice que los datos e implementación de los objetos, deben ocultarse con el fin de mostrar lo que se

puede hacer y no cómo se hace, pero la interfaz, o dicho de otro modo las operaciones con que manipulamos los objetos, deben publicarse.
El medio por el cual vamos a efectuar el ocultamiento o publicación de las distintas partes del objeto, se conoce como visibilidad o control de acceso. Así antepondremos a un miembro de la clase, una operación o a un atributo, la palabra public, si queremos que el miembro sea accesible desde cualquier parte del programa. Por el contrario, antepondremos la palabra private, si solo queremos que sea accesible desde otro método de la clase.

1.2.4 Uso de this

Cuando en el cuerpo de una operación requerimos referirnos al objeto receptor del mensaje, se usa this. Por ejemplo, en el código:

```
class Pajaro {
void volar() {
this.aletear();
}
void aletear();
}
```

Cada objeto de la clase Pájaro, al momento de volar, se mandará un auto mensaje aletear. This además sirve para distinguir variables que se encuentran en distintos ámbitos. Por ejemplo en el siguiente caso:

```
class Hora {
private int hora;
private int minutos;
public void ajustarMinutos(int minutos) {
this.minutos=minutos;
}
}
```

Gracias a this, el compilador distingue entre la variable minutos de la clase, del parámetro de la operación. Un último uso del this se aplica cuando un objeto desea pasarse a sí mismo como parámetro en el mensaje a otro objeto. Por ejemplo:

```
class Carta {
private String direccionRemitente;
public String obtenerDireccionRemitente() {
return direccionRemitente;
}
public void enviarCorreo(Buzon miBuzon) {
miBuzon.registrarCarta(this);

miBuzon.efectuarEnvio();
}
}
```

En el ejemplo, al registrar la carta, mi Buzón tendrá la posibilidad de enviar un mensaje al correo original, en caso de fallo en el envío:

```
class Buzon {
private Carta cartaAEnviar;
public void registrarCarta(Carta unaCarta) {
cartaAEnviar=unaCarta;
}
public void efectuarEnvio() {
enviandoOficinaPostal();
if(falloEnvio())
reportarFalloRemitente(cartaAEnviar.obtenerDireccionRemitente());
}
}
```

1.2.5 Constructores y new.

A menudo uno pudiera olvidar la inicialización de un objeto y empezar a mandarle otros mensajes provocando como consecuencia, un funcionamiento equivocado. Java, tomando como ejemplo a C++, provee del concepto de constructor. Un constructor no es más que una operación de inicialización con una sintaxis determinada. Un constructor tiene el mismo nombre de la clase y no tiene valor de retorno. La invocación a un constructor se realiza por medio de la palabra new, seguida del nombre del constructor más los parámetros del constructor. De este modo, new crea el espacio, el constructor lo inicializa y finalmente, se devuelve la referencia al objeto creado e inicializado.Un ejemplo siguiente se muestra el funcionamiento de un constructor:

```
class Manzana {
private int peso;
private float tiempoAlmacenaje;
private String tipoManzana;
public Manzana(String tipoManzana) {
peso=200;
tiempoAlmacenaje=40;
this.tipoManzana=tipoManzana;
}
class ArbolManzanas {
public void hacerAlgoConManzana() {
Manzana unaManzana = new Manzana("Golden"); //Llama al constructor Manzana
unaManzana.madurar();
…
}
}
```

Sin embargo, como probablemente haya notado usted, anteriormente ya se había usado un constructor, al escribir una línea como la siguiente:

Manzana miManzana = new Manzana();

Y en este caso usted no había tenido la necesidad de declarar ningúnconstructor. Esto es, porque Java proporciona un constructor automático llamado constructor por omisión en el caso en que no se haya declarado previamente ningún constructor.

Sobrecargando constructores.

Se pueden sobrecargar los constructores, proporcionando argumentos distintos para cada constructor. Uno por cada necesidad de inicialización. Por ejemplo, en el código de la clase Manzana expuesto anteriormente, se puede agregar otro constructor:

```
class Manzana {
private int peso;
private float tiempoAlmacenaje;
private String tipoManzana;
public Manzana(String tipoManzana) {
peso=200;
tiempoAlmacenaje=40;
this.tipoManzana=tipoManzana;
}
public Manzana(int peso,int tiempoAlmacenaje) {
this.peso=peso;
this.tiempoAlmacenaje=tiempoAlmacenaje;
}
}
```

Y el llamado al último constructor queda expresado:

```
class ArbolManzanas {
public void hacerAlgoConManzana() {
Manzana unaManzana = new Manzana(130,34);
//Llama al constructor Manzana y lo inicializa con los
//valores que se mandan como paràmetros
unaManzana.madurar();
…
}
}
```

1.2.6 Inicialización en la declaración.

Algunas variables requieren inicializarse con el mismo valor para cada objeto, aunque el código de inicialización puede colocarse en un constructor, es más fácil definirlo en el momento de declarar los atributos de la clase. Por ejemplo, en la clase Manzana podemos utilizar esa propiedad:

```
class Manzana {
private int peso=0;
private float tiempoAlmacenaje=10;
…
}
```

Lo cual indicaría que de crearse cualquier objeto de esta clase, los atributos peso y tiempoAlmacenaje, a menos que se diga otra cosa, tendrán los valores de 0 y 10 respectivamente. Por supuesto cualquier miembro de la clase, incluyendo constructores, podrá cambiar ese valor.

Referencias.

Antes de introducirnos a este tema, es necesario indicar que el manejo de variables que se refieren a tipos primitivos (int , char, float) es manejado de manera distinta de las variables que se refieren a tipos de datos abstractos o clases. En este tema nos concentraremos precisamente en este último tipo demanejo. Una referencia es similar al concepto de apuntador en lenguajes como C, pero sin la sintaxis que confunde a más de uno. Cuando un programador declara:

```
Manzana miManzana;
```

Esta diciendo que existe un nombre (miManzana) que más tarde podrá referirse a una variable de un cierto tipo (Manzana), pero que, por lo pronto, no existe todavía. Para crear una variable se requerirá posteriormente la instrucción:

```
miManzana = new Manzana(23,12);
```

Lo que es más, dos variables podrían compartir el mismo nombre. Por ejemplo, si posteriormente escribimos:

```
Manzana otraManzana=miManzana;
```

Contra lo que uno podría pensar, no se está copiando la variable, se le está dando otro nombre o se dice que se está referenciando. Si realmente lo que deseamos es copiarla, entonces se debería usar otro new y copiar atributo por atributo.

1.2.7 Recolector de basura.

Como se mencionaba en la introducción, Java cuenta con un mecanismo de administración de

memoria que le permite liberar memoria de una variable cuando ya no se referencía. Cuando es referenciada una variable, es incrementado un contador interno, cuando es eliminada una referencia, es decrementado el contador. Si el contador llega a cero, la variable se elimina.
Una variable es referenciada cuando:
• Se crea una variable (un new) y se asigna a una referencia.
• Es pasada como parámetro a una operación.
• Se regresa una variable de una operación.
• Es hecha una asignación de una variable.
Y se decrementa, cuando se acaba en alcance de una referencia.

1.2.8 Miembros estáticos.

Un miembro estático de la clase, es aquel que aplica a todos los objetos de la clase, no a uno en particular. Por ejemplo, en el caso de una clase Calendario, los nombres de los meses son miembros estáticos porque aplican a todos los calendarios con los mismos valores. El motivo de existencia de los miembros estáticos es optimización. La repetición de los mismos atributos con el mismo valor además de ser redundante, es dispendiosa. Mejor es compartir estos atributos entre todos los objetos de la clase. Ésto es la esencia de lo estático Los atributos estáticos ya que no pertenecen a ningún objeto en particular, no se pueden manipular en cualquier operación. El uso de operaciones estáticas es necesario, una operación es estática cuando se le antepone la palabra static. Luego, para invocar a la operación, es necesario enviar el mensaje a la clase. Como observación final de este tema, cabe hacer mención que hay que ser cuidadoso al establecer una operación como estática, ya que en varias ocasiones, se trata del efecto de confundir a la clase dueña de la operación.

1.2.9 Colecciones de objetos.

Eventualmente, requerimos de conjuntos de objetos, para tal caso, Java provee de dos formas de realizarlo, uno implantado en el lenguaje mismo y otro por medio de ciertas clases utilitarias.

Arreglos.

Los arreglos se utilizan con una sintaxis parecida a la de C, en una declaración de la forma:

```
Mariposa mariposas[];
```

O bien

```
Mariposa []mariposas;
```

La sentencia declara un arreglo que puede guardar cualquier número de variables. Al principio, un arreglo no tiene una longitud determinada, por lo que es necesario definir el número de variables que almacenará el arreglo. El tamaño de un arreglo se especifica con la ayuda de new. Así, del ejemplo mencionado anteriormente escribimos:

```
mariposas=new Mariposas[5];
```

Esta última instrucción crea un espacio para 5 referencias a variables, sin embargo las variables no existen aún, es necesaria la creación de cada variable por separado.
Una vez que se ha creado un arreglo, podemos referirnos a su longitud por medio En Java, no existen, en el sentido estricto, los arreglos bidimensionales ni multidimensionales, pero es fácil hacer el efecto deseado. Por ejemplo, si queremos definir un arreglo bidimensional, declaramos:

```
Manzana [ ][ ]tablaManzanas;
```

Como se puede observar, no se indica el largo y ancho de la matriz, razón por la cual existen dos maneras de crear el espacio para las referencias. En la primera forma se escribe un código como el siguiente:

```
tablaManzanas=new Manzana[3][5];
```

Creando el espacio para 15 referencias. La segunda forma es similar a la manera en que se realiza el manejo de los arreglos de apuntadores en C, Pero ya sea de una o de otra forma, solo se ha apartado memoria para las referencias, no para las variables, por lo que para ambos casos, es necesario escribir un código que aparte la memoria en específico

Vectores.

Los vectores son arreglos que pueden crecer en tiempo de ejecución, no son parte del lenguaje mismo, por lo que se requiere incluir el siguiente encabezado:

```
import java.util.Vector;
```

Un vector para Java es una clase que implementa una serie de operaciones de consulta e inserción, de las cuales las principales son:

–addElement(Object objeto). Agrega un nuevo elemento al final de la lista
–insertElementAt(Object objeto,int indice). Inserta (no reemplaza) el objeto objeto en la posición indice.
–setElementAt(Object objeto,int indice). Inserta el objeto objeto en la posición índice.
–removeElementAt(Object objeto,int indice). Quita el elemento que se encuentra en índice y compacta el arreglo.
–Object elementAt(int indice). Devuelve el objeto que se encuentra en la posición índice.

Para la construcción de un objeto de la clase Vector puede están disponibles una variedad de constructores

String

La clase String es una colección de caracteres ordenados que proporciona una amplia variedad de operaciones dando una excelente funcionalidad para el manejo de cadenas. Para crear una objeto String proporciona varios constructores, de los cuales, los más relevantes son:

•String();
•String(String value);
•String(char value[]);
•String(char value[],int offset,int count);

Una vez que hemos creado un objeto String, podemos aplicar una multitud de operaciones entre ellas:

•int length(). Devuelve la extensión de una cadena
•char charAt(int indice). Devuelve el carácter que se encuentra en cierto elemento de la cadena boolean equals(String unString). Es una operación que devuelve true y false, dependiendo si dos cadenas son iguales o no. Es de notar que una letra mayúscula y su correspondiente minúscula, se consideran distintas.
•String substring(int beginIndex,int endIndex). Devuelve el string que se encuentra entre dos índices de una cadena.

Ciertamente hay una amplia variedad de operaciones sobre String pero creemos que éstas son las más fundamentales

1.2.10 Línea de comandos.

Probablemente se haya preguntado que significa aquel parámetro que recibe el main y que nunca se ha utilizado. La razón radica en que es práctico pasarle al programa parámetros desde la línea de comandos, es decir por medio del comando con que se ejecuta el programa. Describiremos a continuación un programa que hace uso de la línea de comandos: Podrá observar que para utilizar la línea de comandos, basta con utilizar el arreglo de strings que recibimos en main.

1.2.11 Composición.

La composición es una de las características más importantes de la orientación a objetos, ya que permite el escalamiento, es decir la construcción de objetos en función de objetos más pequeños. La composición es una relación entre dos clases, donde la primera es contenedora de la segunda, y la segunda contenido de la primera. La composición, además se puede implantar de distintos modos, en general requiere del manejo de colecciones, como las que hemos estudiado previamente. En el caso más simple, el de arreglos, basta con declarar como miembro de la clase contenedora al arreglo de objetos contenidos. En el caso de que se implante la composición por medio de la clase Vector, se cuenta con una ventaja relativa con respecto a la implantación con arreglos, ya que ofrece un número ilimitado de objetos contenidos

1.2.12 Interacción entre objetos.

Cuando se empieza a programar en el paradigma de objetos, es común cometer una serie de errores de concepto, como lo es intentar violar el principio de encapsulación y es que una vez que se han identificado diversas clases, no queda claro como van a colaborar entre ellas.

1.2.13 Paquetes.

Un paquete en Java permite agrupar clases comunes para facilitar su manejo. El uso de paquetes en Java requiere del uso de dos palabras clave import y package, package lo utilizamos cuando queremos formar un paquete e import cuando queremos invocarlo o referenciarlo.

Formando un paquete.

Para la construcción de un paquete, es necesario que los archivos se encuentren dentro del mismo directorio y cada archivo del paquete debe de tener al principio un encabezado package. Esto nos dice que el archivo al que le adjuntamos este encabezado pertenece al paquete.

Referenciando un paquete.

Para poder hacer referencia a un paquete es necesario utilizar la palabra reservada import que permite indicar que el programa hará uso de cierta clase (o clases) que se encuentran en cierto paquete. Por ejemplo al escribir:

```
import escritores.lewisCarroll.paisMaravillas;
```

Estamos diciendo que utilizamos el paquete paisMaravillas que se encuentra en el directorio escritores/lewisCarroll. Esto no quiere decir que paisMaravillas sea forzosamente un archivo, bien puede ser un grupo de archivos que tienen como punto común el mismo encabezado. Cabe aclarar, sobretodo a los programadores en C y C++, que import a diferencia de include no copia el contenido del paquete en el archivo, sino que le indica al compilador de Java que un archivo hará uso de clases que se encuentran en cierto paquete, la liga se realiza en tiempo de ejecución. Java también admite una declaración como la siguiente:

```
import escritores.lewisCarroll.*;
```

Esta declaración describe que incluirá todos los paquetes que se encuentran en escritores/lewisCarroll, no así los subdirectorios que se derivan de la ruta. Cuando se omite la palabra reservada public o private en un miembro de la clase,en este caso, se forma un paquete comunal entre todas las clases que tienen esas características, y todas ellas tienen acceso entre sí.

1.2.14 Herencia.

La herencia es un mecanismo para reutilizar y extender software, consiste en construir una clase en función de otra previamente definida, para agregar atributos y operaciones, o bien, redefinir las operaciones existentes sin alterar la clase primera. Sin embargo a nivel meramente conceptual se trata de la definición de un concepto como un tipo especial de otro, por ejemplo, la clase Comandante hereda de la clase Militar, mientras que la clase Reptil hereda de la clase Animal. Se dice que la clase que sirve como punto de partida es la clase base y la clase generada es una clase derivada de la primera, también se dice que la clase base es la superclase de la clase derivada, y esta a su vez subclase de la primera.La herencia es un tipo de relación transitiva, es decir, que si la clase A hereda de la B, y la B de la C, entonces A hereda de C, aunque en este caso se trata de una herencia indirecta. Cuando declaramos un objeto de una clase derivada, internamente se crea dentro del objeto dos partes, una conteniendo todos los atributos correspondientes a la clase base y otra con los atributos que corresponden a la clase derivada. Por supuesto, si construimos una jerarquía de clases A hereda de B, B de C, C de D y así sucesivamente, cada objeto de A tendrá una parte por cada clase que está en su jerarquía.
En Java la forma en que se define la herencia es por medio de la palabra reservada extends.

Por ejemplo, la clase Manzana, bien hubiera podido construirse sobre la base de una clase Fruta. En Java, a diferencia de C++, no tiene multiherencia, es decir, que una clase solo puede heredar directamente de otra clase. Este hecho ahorra a Java los problemas asociados a la multiherencia, tal como el problema conocido como del diamante, un problema específico de ambigüedad al llamar a una operación; o bien, nos forzará a hacer explícito los nombres de las clases base de cierta clase derivada, lo que hará el código más dependiente de una implementación en particular. Como veremos más tarde, si bien Java no soporta multiherencia, implementa un concepto conocido como interfaz que proporciona una flexibilidad similar.

Visibilidad en subclases.

De nuevo el problema del acceso, el acceso a un atributo u operación se puede clasificar del siguiente modo:

a) La clase a la que pertenece la operación o atributo
b) Alguna clase derivada directa o indirectamente de la clase anterior
c) Alguna clase (cliente) que hace uso de un objeto dentro del mismo paquete de la clase de la que se hace uso.
d) Alguna clase (cliente) que hace uso de un objeto fuera del mismo paquete de la clase de la que se hace uso.

Si queremos restringir el acceso solo al punto a, escogemos la palabra private, si queremos restringir el acceso solo a los puntos a y b, usamos la palabra protected.Si no queremos restringir los puntos de acceso usamos public. Si no se dice nada se asume a y c.

Redefinición de operaciones.
A menudo es útil redefinir operaciones que ya existen en una clase base, debido a que la clase derivada aunque realice la misma operación que su ancestro, probablemente requiera implementarla de manera distinta. Como quizás haya notado, la operación que se ejecuta, es aquella que pertenece a la clase que está más abajo en la jerarquía de clases.

Super y colgado de operaciones.
La palabra reservada super tiene como fin, invocar a las operaciones de la clase base desde alguna clase hija, super es parecida a la palabra this, ya que en ambos casos sirve para referirse al objeto que está recibindo el mensaje, pero en el primer caso se intenta encontrar la operación, en la parte del objeto correspondiente a la clase base, mientras que en el segundo caso, se busca en la clase derivada

Constructores.
Utilizamos herencia para observar que las operaciones se pueden redefinir, pero se mencionó que los constructores de una clase base no se heredan, este efecto se debe a que casi siempre la inicialización de la parte de datos de la clase derivada implica la inicialización de la parte de datos de la clase base más algún proceso extra. Las aplicaciones en Java pueden invocar al constructor de la clase base sin hacer explícito el nombre de la clase base, es mediante el uso de la palabra super que se logra el objetivo.

```
super(nombre,clave);
```

se invoca al constructor, la sintaxis super(parametro1,parametro2, etc), siempre llama al constructor de la clase base. La única condición adicional para usar en el llamado a un constructor base, es que sea la primera instrucción con la que empieza el constructor de la clase derivada.

Cortando herencia.
Cuando deseamos que una operación no se pueda redefinir en las clases derivadas usamos la palabra reservada final. Al intentar redefinir esta operación , el compilador mandará un error. El corte de herencia, permite proteger a nuestras clases de tener problemas colaterales al ser redefinidas. La palabra final nos prohibe sobreescribir una operación y de este modo nos protege. A pesar de esto, el uso del corte de herencia es una cuestión de balance, si cortamos la herencia a diestra y siniestra, provocaremos que la aplicación tenga muy poca flexibilidad de cambio y en consecuencia, exista una pobre reutilización.

1.2.15 Polimorfismo.
El polimorfismo es un tema interesante por diversas razones, la más importante y que incumbe a la programación orientada a objetos es la flexibilidad que se da en el desarrollo de software. En pocas palabras, el polimorfismo es la posibilidad de tratar a los objetos de diferentes clases de manera uniforme, cuando se dice uniforme, nos referimos a que no se utilizará la estructura lógica if.
Por ejemplo, suponga que usted tiene objetos de las clases DocumentoProcesadorPalabras , DocumentoHojaCalculo, DocumentoPresentaciones y desea escribir una rutina para imprimir cada uno de los objetos. Como punto de partida se debería escribir una clase que sea el vínculo común entre las distintas clases, la llamaremos simplemente clase Documento, y será una clase base para las clases descritas con anterioridad. Imprimiremos cada clase dependiendo del tipo de documento, con la rutina imprimirDocumentos La forma de codificar esta rutina tiene problemas graves, elprincipal radica en la inflexibilidad al cambio, porque

cada vez que surja una nueva clase derivada de Documento, tendremos que modificar la rutina de imprimirDocumentos y volver a compilar el programa. De este modo, ControlImpresión nunca podrá aspirar a ser una pieza independiente de código, con su consiguiente bajo reuso.
Otra forma de abordar el problema es por medio de polimorfismo. Cuando se usa polimorfismo, delegamos al ambiente de ejecución la responsabilidad de seleccionar el código correcto para el tipo de datos requerido. Las operaciones en Java son polimórficas por omisión, a menos que se indique explícitamente lo contrario. El ambiente de ejecución en una operación polimórfica siempre buscará, por decirlo de algún modo, las clases "hoja" en la jerarquía de clases. Por ejemplo, si tenemos las clases Casita, CasitaMadera y CasitaMaderaAlLadoRío, y una operación que aplica a las tres clases, CasitaMaderaAlLadoRío tendrá prioridad sobre las otras dos. Saber aplicar polimorfismo correctamente, no es evidente. El manejo correcto de polimorfismo hace la diferencia entre una aplicación verdaderamente orientada a objetos y una que solo utiliza un lenguaje orientado a objetos.

Clases abstractas.
Una clase abstracta es una forma de aplicar polimorfismo. En el ejemplo anterior, usamos la clase documento, que pese a que era necesaria, no requería generar instancias u objetos. La clase era simplemente un "puente" entre clases más complejas que debían manejarse de la misma forma. La clase Documento es un ejemplo de una clase abstracta. Las clases

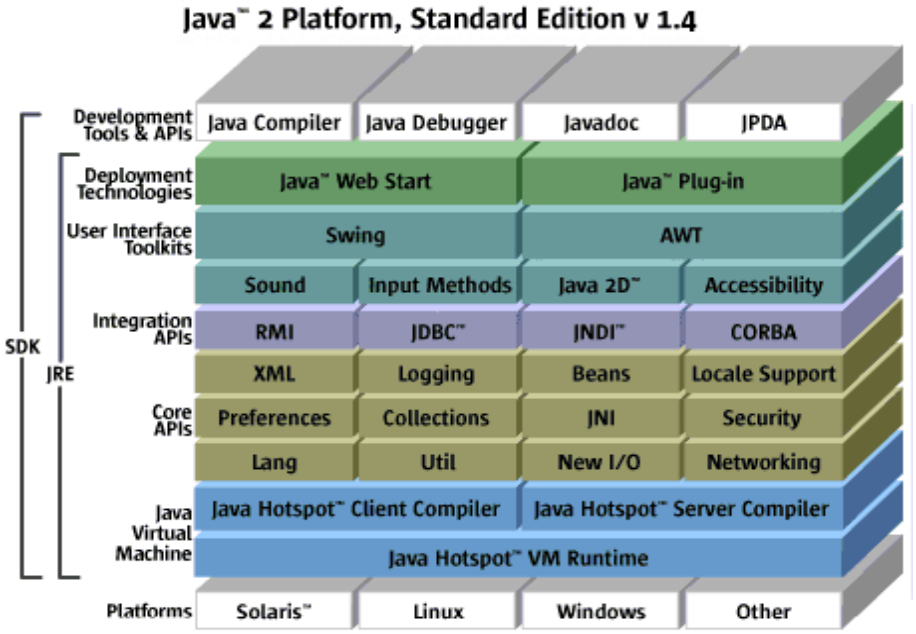
abstractas no definen el cuerpo o implementación de algunas operaciones. A estas operaciones se les conoce como operaciones abstractas o firmas.Una vez que tenemos una clase abstracta es necesario derivar otra clase, parahacer "concreta" cada operación abstracta. Su manejo no es distinto al de las herencias comunes, solo hay que tener cuidado de definir (ya no redefinir) las operaciones abstractas. Una clase derivada de una clase abstracta, también será abstracta si no se "concretan" todas sus operaciones. Las clases abstractas deben de hacer explícita su existencia, con el fin de evitar su instanciación en tiempo de ejecución y dar pie a la ejecución de una operación abstracta, ocasionando un error de ejecución en el programa. En el ejemplo, Documento siendo una clase abstracta, define tanto operaciones abstractas y no abstractas, estas últimas debido a que la clase tiene la información necesaria para definir su implementación.

Interfaces.
Las clases abstractas son un buen mecanismo para manejar polimorfismo, el inconveniente, es cuando deseamos dar uniformidad sin querer establecer una jerarquía de clases. Suponga que un profesor tiene que hacer material para su curso para lo cual tiene el rol de escritor, la clase Profesor cobra un salario y hubiésemos querido que heredara de la clase Trabajador, pero desafortunadamente ya está "ocupada" su clase base. Un mecanismo, que Java le toma prestado a C Objetivo, para solucionar este tipo de problemas, es la idea de interfaz. Una interfaz en Java es un conjunto de operaciones abstractas que deben ser implementadas por alguna clase. La sintaxis de una interfaz es similar a la definición de una clase, pero en lugar de la palabra class usa la palabra interface.
La palabra reservada implements permite describir que la clase Profesor, debe definir el cuerpo de cada una de las operaciones abstractas, descritas en la interfaz Escritor. A diferencia de las clases abstractas, en una interfaz no se debe anteponer a cada operación la palabra abstract, porque se sobreentiende abstracta.

Una interfaz como no es una clase, no puede definir detalles de implementación, como los datos; aunque es permitida la definición de constantes. Si una interfaz tuviera datos ya hablaríamos de una implementación y por lo tanto, en lugar de tener una interfaz, tendríamos una clase abstracta.
Por su naturaleza, las interfaces siempre son públicas, ni privadas ni protegidas, porque su finalidad es establecer un modo de uso para las operaciones queimplementa una clase.
Una clase puede implementar múltiples interfaces, solo es necesario agregar después de implements una lista separada por comas de todas las interfaces implementar, La propiedad de poder asociar varias interfaces a una clase, es parecida al concepto de multiherencia. En la multiherencia, la herencia de implementaciones provoca confusión en la manera de referirse a cada implementación en particular. En contraparte, al definir una clase que implemente
múltiples interfaces, si existe una firma en común entre más de una interfaz, con una sola implementación de la firma en la clase bastará.

2 PROGRAMACÍON CON JAVA J2SE

2.1 INTRODUCCÍON A LA TECNOLOGÍA J2SE



[1]fig1.1

J2SE signica java dos edicíon estandar, basicamente la diferencia entre este y J2EE(java dos edicíon empresarial) es que las aplicaciones que se hacen en J2SE son stand alone, quiere decir que pueden crearse, ejecutarse en una computadora y aplicarse en esta misma sin necesidad de tener comunicacióncon otra o otras, como J2EE que forzosamente necesita un servidor de publicacion para servlets o jsp. J2EE, es un conjunto de librerías para hacer desarrollo empresarial en Java, en la figura 1.1 se nos muestrala plataforma java 2 standard edition y sus componentes principales, que a continuacion se detallaran

Un **Software Development Kit (SDK)** o kit de desarrollo de software es generalmente un conjunto de herramientas de desarrollo que le permite a un programador crear aplicaciones para un sistema bastante concreto, por ejemplo ciertos paquetes de software, frameworks, plataformas de hardware, ordenadores, videoconsolas, sistemas operativos, etcétera

JRE o **Java Runtime Environment** (Entorno de Ejecución Java) proporciona únicamente un subconjunto del lenguaje de programación Java sólo para ejecución. El usuario final normalmente utiliza JRE en paquetes y añadidos.

[1] Digrama de java sun

Una **API** (del inglés **Application Programming Interface - Interfaz de Programación de Aplicaciones**) es el conjunto de **funciones** y **procedimientos** (o métodos si se refiere a programación orientada a objetos) que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción.

Una API representa un interfaz de comunicación entre componentes software. Se trata del conjunto de llamadas a ciertas librerías que ofrecen acceso a ciertos servicios desde los procesos y representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las APIs asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API.

El JRE, la máquina virtual de Java y las librerías básicas del J2SE son las herramientas de desarrollo. Un usuario sólo necesita el JRE para ejecutar las aplicaciones desarrolladas en lenguaje Java, mientras que para desarrollar nuevas aplicaciones en dicho lenguaje es necesario un entorno de desarrollo, denominado JSDK, que además del JRE (mínimo imprescindible) incluye, entre otros, un compilador para Java.
El desarrollo de un programa consta de tres fases:

1. *Escritura del código fuente del programa*. Mediante un editor de texto,se escribe el código del programa Java. El resultado de esta etapa es un fichero conteniendo el código fuente de un programa en Java

2. *Compilación del programa*. El compilador del lenguaje de programación Java, *javac,* toma el fichero fuente y traduce su contenido a *bytecode,* un "lenguaje" que es independiente de la máquina y sistema operativo y que la máquina virtual de Java (**Java Virtual Machine** - Java VM) entiende. En esta fase se detectan los errores léxicos, sintáticos y semánticos que pudiera contener el código. Una vez que el código fuente está libre de errores, el programa javac genera antes un fichero con extensión ".class", conteniedo el código bytecode, por cada clase contenida en el .java. Mientras existan errores, el programador deberá editar el fichero con el código fuente y corregirlos.

3. *Ejecución del programa*. **La máquina virtual de Java**, por medio del programa intérprete *java* toma el fichero en bytecode y traduce cada instrucción al lenguaje que el ordenador entiende directamente, ejecutándola seguidamente.

En este capitulo se mostrara el uso de la tecnologia J2SE por medio de los kits de interface de usuario awt y swing.

2.2 AWT.

AWT (Abstract Window Toolkit) es la parte de Java diseñada para crear interfaces de usuario y para dibujar gráficos e imágenes. Es un conjunto de clases que intentan ofrecer al desarrollador todo lo que necesita para crear una interfaz de usuario para cualquier applet o aplicación Java. La mayoría de los componentes AWT descienden de la clase java.awt.Component como podemos ver en la figura 1.1.
(Obsérvese que las barras de menú de AWT y sus ítems no encajan dentro de la jerarquía de Component.)

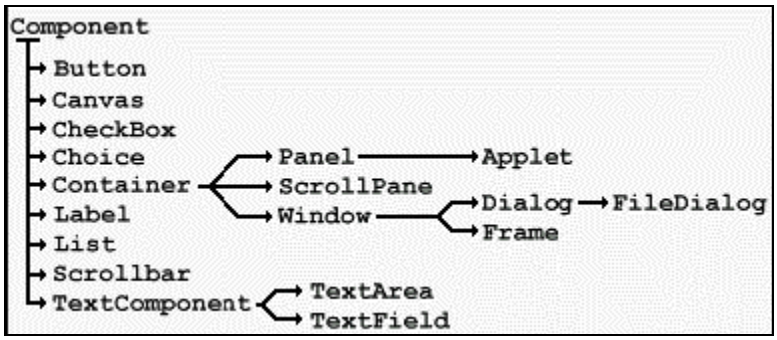


fig1,1

El AWT es un conjunto de clases que permiten construir una interfaz gráfica de usuario lo más independiente de la plataforma en que corre la aplicación. Las clases así creadas dan soporte al manejo de eventos, botones, controles y ventanas por nombrar algunos.

2.2.1 FUNDAMENTOS

Manejo de eventos.

Entendemos por evento, cada uno de los sucesos que alteran un programa desde su exterior, como lo puede ser un movimiento o click del ratón, la presión de una tecla, o el cerrado de una ventana. Inicialmente en la versión 1.02, cada uno de los eventos que se generaba en un prograama enrán manejados por el mismo programa. Por esa razón, a cada uno de los eventos le correspondía una rutina de atención. Esta forma de manejar los eventos sería óptima si en general, tuviéramos pocos eventos, pero en el caso de una amplia variedad de ellos, la agrupación de eventos comunes no es evidente y ciertamente,propensa al caos. Por esta razón, a partir de la versión 1.1. el trabajo en el manejo de eventos no se centraliza en el programa, sino que se distribuye entre distintos objetos. En este documento discutiremos solo la versión más reciente, es decir la versión 1.1, ya que consideramos la de mayor uso.

Manejo de eventos en la versión 1.1.

A partir de la versión 1.1, tenemos la posibilidad de manejar los eventos desde cualquier clase que implemente alguna de las interfaces de manejo de eventos, tales como:

- FocusListener. Eventos relativos a la recuperación/pérdida de foco del teclado.
- KeyListener. Eventos relativos al teclado.
- MouseListener. Eventos relacionados con clicks del ratón sobre un componente.
- MouseMotionListener. Eventos relacionados con el movimiento del ratón sobre un componente.

• WindowListener. Eventos relacionados con alteraciones en las ventanas (iconizar,cerrar, etc.).
• ActionListener. Recibe eventos de acción.

Una vez que se ha seleccionado cuales serán las interfaces que contienen los eventos que nos interesan, hay que poner manos a la obra. Por ejemplo si tenemos la interfaz MouseListener deberemos construir una clase que implemente cada una de las operaciones de esta interfaz (mouseClicked, mousePressed, etc.). Este tipo de clase recibe el nombre de auditor y no solo puede implementar una sola interfaz sino varias.
El siguiente paso será hacerle saber al sistema, que los eventos que ocurran durante la ejecución serán enviados a nuestro auditor. A esta operación se le conoce como registrar al auditor, y dependiendo del tipo de interfaz que implemente el auditor, se aplicarán distintas formas de registro, aunque todas muy parecidas entre sí. Por, ejemplo para registrar un auditor que implementa KeyListener, se usa addKeyListener (unAuditor), pero si se trata de registrar un auditor para la interfaz MouseListener, se usa addMouseListener(unAuditor), ya se imaginará el resto. Con el fin de clarificar el funcionamiento de eventos examinaremos una aplicación en donde se hace uso de eventos. La aplicación que presentaremos es un frame que delega en un objeto auditor el tratamiento de eventos del ratón y teclado, reportando en pantalla el evento que ocurrió. Analizaremos las distintas secciones del código:

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

En esta primer parte se incluye los paquetes de eventos, listeners y gráficos, necesarios para la correcta compilación del programa.

```
public class AplicEventos extends Frame {
private Label etiqueta = new Label("Evento Actual");
private Auditor unAuditor;
```

En el programa creado, declaramos dos variables, la primera para registrar el tipo de evento ocurrido y la segunda es auditor, responsable de atender cada uno de los eventos que ocurrirán.

```
public AplicEventos() {
unAuditor=new Auditor();
this. addMouseListener(unAuditor);
this.addKeyListener(unAuditor);
this. add(etiqueta);
unAuditor.registrarFrame(this);
}
```

En la constructor del frame, creamos al auditor y posteriormente le decimos al sistema de ejecución que queremos que el objeto unAuditor reciba tanto los eventos relativos a clicks del mouse como del teclado. Finalmente, existe la necesidad de que unAuditor tenga alguna referencia al frame ya sea para actualizar o recuperar información, lo cual es realizado mediante el método registraFrame que describiremos más adelante.

```
public void asignarMensaje(String mensaje) {
etiqueta.setText(mensaje);
}
```

El modo por el cual se va a poder modificar el tipo de evento es la operación del frame, asignarMensaje que únicamente actualiza la variable mensaje, para que en el momento en que se ejecute el método se visualice el nuevo evento.

```
class Auditor implements MouseListener,KeyListener {
private AplicEventos aplic;
```

La clase Auditor es la responsable de gestionar los distintos eventos, implementa las dos interfaces que nos interesan además de tener una variable aplic que le permite mandar mensajes hacia el frame.

```
public void registrarFrame(AplicEventos unFrame) {
aplic=uFrame;
}
```

Esta función que mencionamos previamente, permite conectar al frame con el auditor. Simplemente conserva una referencia al frame.

```
public void mouseClicked(MouseEvent e) {
aplic.asignarMensaje("Click");
}
public void mouseEntered(MouseEvent e) {
aplic.asignarMensaje("entro mouse");
}
public void mouseExited(MouseEvent e) {
aplic.asignarMensaje("mouse salió");
}
public void mousePressed(MouseEvent e) {
aplic.asignarMensaje("mouse oprimido");
}
public void mouseReleased(MouseEvent e) {
aplic.asignarMensaje("mouse liberado");
}
public void keyPressed(KeyEvent e) {
aplic.asignarMensaje("tecla oprimida");
}
public void keyReleased(KeyEvent e) {
aplic.asignarMensaje("tecla liberada");
}
public void keyTyped(KeyEvent e) {
aplic.asignarMensaje("tecla tipeada");
}
}
```

Como puede observar, cada uno de los eventos que se implementan, solamente actualiza el mensaje del frame para visualizar el nuevo evento. Cada operación recibe un objeto que describe el evento en detalle, al cual interrogaremos en caso de requerir información en más detalle. Si usted desea que la clase sea su propio auditor basta con cambiar la línea

```
public class AplicEventos extends Frame
```

con

```
public class AplicEventos Frame implements MouseListener,KeyListener
```

y en código del constructor reemplazamos las líneas que agregan auditores addMouseListener y addKeyListener por:

```
addMouseListener(this);
addKeyListener(this);
```

Finalmente la línea:

```
unAuditor.registrarFrame(this);
```

se puede eliminar, ya que no requerimos que la clase se comunique consigo misma por medio de una variable.

2.2.2 Clases principales

java.awt	Contienen todas las clases para crear interfaces de usuario y dibujar graficas e imágenes.
java.awt.color	Provee clases para dar espacios de color.
java.awt.datatransfer	Provee clases e interfaces entre transferencia de datos y dentro de aplicaciones.
java.awt.dnd	arrastra y baja directamente una manipulacion encontrando una gestion en muchas interfaces graficas de usuarios, estos sistemas proveen un mecanismo para transferir informacion entre dos entidades logicamente asociadas con presentacion de elementos dentro del gui .
java.awt.event	Provee interfaces y clases para negocios con diferentes tipos de eventos disparados por componentes AWT.
java.awt.font	Provee clases e interfaces relacionadas con las fuentes.
java.awt.geom	Provee clases Java 2D para definir y funcionar en operaciones sobre objetos relacionados en geometria de dos dimensiones.
java.awt.im	Provee interfaces y clases para el metodo base de contribucion.
java.awt.im.spi	Provee clases que son capaces de desarrollar la contribucion de metodos que pueden ser usados con cualquier Java runtime environment.
java.awt.image	Provee clases para crear y modificar imágenes.
java.awt.image.renderable	Provee clases e interfaces para producir una imagen en una version independiente.
java.awt.print	Provee clases e interfaces para impresión general del api.

En el contexto del proceso de interacción persona-ordenador, la interfaz gráfica de usuario, es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático.
La interfaz gráfica de usuario (en inglés *Graphical User Interface*, GUI) es un tipo de interfaz de usuario que utiliza un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Habitualmente las acciones se realizan mediante manipulación directa para facilitar la interacción del usuario con la computadora.

Surge como evolución de la línea de comandos de los primeros sistemas operativos y es pieza fundamental en un entorno gráfico. Como ejemplo de interfaz gráfica de usuario podemos citar el escritorio o *desktop* del sistema operativo Windows y el entorno X-Window de Linux.

2.2.3 Componentes.

Component es una clase abstracta que representa todo lo que tiene una posición, un tamaño, puede ser pintado en pantalla y puede recibir eventos. No tiene constructores públicos, ni puede ser instanciada. Sin embargo, desde el JDK 1.1 puede ser extendida para proporcionar una nueva característica incorporada a Java, conocida como componentes Lightweight.
Los Objetos derivados de la clase Component que se incluyen en el Abstract Window Toolkit son los que aparecen a continuación:

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

Sobre estos Componentes se podrían hacer más agrupaciones y quizá la más significativa fuese la que diferencie a los Componentes según el tipo de entrada. Así habría Componentes con entrada de tipo no-textual como los botones de pulsación (Button), las listas (List), botones de marcación (Checkbox), botones de selección (Choice) y botones de comprobación (CheckboxGroup); Componentes de entrada y salida textual como los campos de texto (TextField), las áreas de texto (TextArea) y las etiquetas (Label); y, otros Componentes sin acomodo fijo en ningún lado, en donde se encontrarían Componentes como las barras de desplazamiento (Scrollbar), zonas de dibujo (Canvas) e incluso los Contenedores (Panel, Window, Dialog y Frame), que también pueden considerarse como Componentes.

Etiquetas

Una etiqueta (Label) proporciona una forma de colocar texto estático en un panel, para mostrar información fija, que no varía (normalmente), al usuario.
La clase Label extiende la clase Component y dispone de varias constantes que permiten especificar la alineación del texto sobre el objeto Label. El programa que se presenta a continuación no proporciona ningún control de eventos, solo nos muestra como insertar una etiqueta dentro de nuestro Frame. Aunque no se ejemplifique, la clase Label al ser una subclase de Component puede tener cualquier receptor de eventos que se pueda registrar sobre Component.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploLabel {
```

```
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico(){
// Instancia un objeto Label con una cadena para inicializarlo y
// que aparezca como contenido en el momento de su creación
miFrame.addWindowListener( new Conclusion() );
Label miEtiqueta = new Label( "Texto inicial" );
// Coloca la eqtiqueta sobre el objeto Frame
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( miEtiqueta );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instancia y registra un objeto receptor de eventos de ventana
// para concluir la ejecucion del programa cuando el Frame se
// cierres por accion del usuario sobre el
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Concluye el programa cuando se cierra la ventana
System.exit(0);
}
}
```

Botones de Pulsación

Los botones de pulsación (Button), son los que se han utilizado fundamentalmente en los ejemplos de este Trabajo, aunque nunca se han considerado sus atributos específicamente.

La clase Button es una clase que produce un componente de tipo botón con un título.El constructor más utilizado es el que permite pasarle como parámetro una cadena, que será la que aparezca como título e identificador del botón en la interfaz de usuario. No dispone de campos o variables de instancia y pone al alcance del programador unaserie de métodos entre los que destacan por su utilidad los siguientes:

- addActionListener() Añade un receptor de eventos de tipo Action producidos por el botón
- getLabel() Devuelve la etiqueta o título del botón
- removeActionListener() Elimina el receptor de eventos para que el botón deje de realizar acción alguna
- setLabel() Fija el título o etiqueta visual del botón

Además dispone de todos los métodos heredados de las clases Component y Object.

ActionListener es uno de los eventos de tipo semántico. Un evento de tipo Action se produce cuando el usuario pulsa sobre un objeto Button. Además, un objeto Button puede generar eventos de bajo nivel de tipo FocusListener, MouseListener o KeyListener, porque hereda los métodos de la clase Component que permiten instanciar y registrar receptores de eventos de este tipo sobre objetos de tipo Button.

Botones de Comprobación

La clase CheckBox extiende la clase Component e implementa la interfaz ItemSelectable, que es la interfaz que contiene un conjunto de items entre los que puede haber o no alguno seleccionado.

Los botones de comprobación (Checkbox) se pueden agrupar para formar una interfaz de botón de radio (CheckboxGroup), que son agrupaciones de botones de comprobación de exclusión múltiple, es decir, en las que siempre hay un único botón activo.

La programación de objetos Checkbox puede ser muy simple, dependiendo de lo que se intente conseguir. La forma más sencilla para procesar objetos Checkbox es colocarlos en un CheckboxGroup, ignorar todos los eventos que se generen cuando el usuario selecciona botones individualmente y luego, procesar sólo el evento de tipoAction cuando el usuario fije su selección y pulse un botón de confirmación.La otra forma, más compleja, para procesar información de objetos Checkbox es responder a los diferentes tipos de eventos que se generan cuando el usuario selecciona objetos Checkbox distintos. Actualmente, no es demasiado complicado responder a estos eventos, porque se pueden instanciar objetos Listener y registrarlossobre los objetos Checkbox, y eso no es difícil. La parte compleja es la implementaciónde la lógica necesaria para dar sentido a las acciones del usuario, especialmentecuando ese usuario no tiene clara la selección que va a realizar y cambia continuamente de opinión.
Es muy importante que cada vez que se va a utilizar un CheckboxGroup hay que especificar un objeto de dicho tipo en nuestro programa. A continuación detallaremos algunos métodos de importancia tanto para los Checkbox como para los CheckboxGroup.

• METODOS DE Checkbox
• CONSTRUCTOR de Checkbox.

```
public Checkbox(String label, boolean state, CheckboxGroup group);
```

Este constructor es muy importante ya que en él definimos, cuestiones muy importantes. El constructor se lleva tres parámetros los cuales son, el texto del Checkbox, el valor ya sea disponible(true) o no disponible(false) y como tercero el CheckboxGroup en donde se va a ubicar dicho Checkbox.

• getLabel

```
public String getLabel();
```

Regresa la etiqueta respectiva a ese Checkbox.

• METODOS DE CheckboxGroup
• getSelectedCheckbox

```
public Checkbox addItemListener();
```

Este método devuelve un objeto de tipo Checkbox elegido en ese instante, perteneciente a dicho CheckboxGroup.El ejemplo que sigue es una demostración, de cómo se aplican los métodos que acabamos de revisar.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class EjemploCheckBox {
public static void main( String args[] ) {
// Se instancia un objeto Interfaz Hombre-maquina
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
// Clase del Interfaz gráfico
class AmbienteGrafico {
// Constructor de la clase
public AmbienteGrafico() {
// Se crea un objeto CheckboxGroup
CheckboxGroup miCheckboxGroup = new CheckboxGroup();
// Ahora se crea un objeto Button y se registra un objeto
// ActionListener sobre él
Button miBoton = new Button( "Aceptar" );
miBoton.addActionListener( new MiActionListener( miCheckboxGroup ) );
// Se crea un objeto Frame para contener los objetos Checkbox y el
// objeto Button. Se fija un FlowLayout, se incorporan a el los
// objetos, se fija el tamaño y se hace visible
```

```
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( new Checkbox( "A",true,miCheckboxGroup ) );
miFrame.add( new Checkbox( "B",false,miCheckboxGroup ) );
miFrame.add( new Checkbox( "C",false,miCheckboxGroup ) );
miFrame.add( new Checkbox( "D",false,miCheckboxGroup ) );
miFrame.add( miBoton );
miFrame.setSize( 250,100 );
miFrame.setVisible( true );
// Instanciamos y registramos un receptor para terminarla
// ejecución de la aplicación, cuando el usuario cierre la
// ventana
miFrame.addWindowListener( new Conclusion() );
}
}
// Esta clase indica la caja de selección que esta seleccionada
// cuando se pulsa el botón de Aceptar
class MiActionListener implements ActionListener {
CheckboxGroup oCheckBoxGroup;
MiActionListener( CheckboxGroup checkBGroup ) {
oCheckBoxGroup = checkBGroup;
}
public void actionPerformed( ActionEvent evt ) {
System.out.println(oCheckBoxGroup.getSelectedCheckbox().getName()+
" " + oCheckBoxGroup.getSelectedCheckbox().getLabel() );
}
}
//Concluye la aplicación cuando el usuario cierra la ventana
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
System.exit( 0 );
}
}
```

Botones de Selección

Los botones de selección (Choice) permiten el rápido acceso a una lista de elementos, presentándose como título el item que se encuentre seleccionado.
La clase Choice extiende la clase Component e implementa la interfaz ItemSelectable, que es la interfaz que mantiene un conjunto de items en los que puede haber, o no, alguno seleccionado. Además, esta clase proporciona el método addItemListener(), que añade un registro de eventos item, que es muy importante a la hora de tratar los eventos que se producen sobre los objetos de tipo Choice. Los objetos de tipo Choice poseen otros métodos, los más importantes se detallan a continuación:

• add

```
public void add(String item);
```

Permite añadir un elemento a la lista de nuestro Choice.

• select

```
public synchronized void select(String str);
```

Selecciona un elemento de la lista y lo coloca como el primero que se desplegará en pantalla.

• gtSelectedItem

```
public synchronizedString getSelectedItem();
```

Regresa la cadena del elemento de la lista actualmente seleccionado.

• addItemListener.

```
public synchronized void addItemListener(ItemListener l);
```

Añade un objeto de tipo ItemListener sobre un objeto Choice. A continuación se muestra un ejemplo en donde aplicamos estos métodos.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class EjemploChoice {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
// Clase del Interfaz Gráfico que instancia los objetos
class AmbienteGrafico {
public AmbienteGrafico() {
// Instancia un objeto Choice y coloca objetos String sobre el
// para realizar las selecciones
Choice miChoice = new Choice();
miChoice.add( "Primer Choice" );
miChoice.add( "Segundo Choice" );
miChoice.add( "Tercer Choice" );
// Seleccionamos la cadena correspondiente a la tercera selección
// por defecto, al arrancar la aplicación
miChoice.select( "Tercer Choice" );
// Instanciamos y registramos un objeto ItemListener sobre
// el objeto Choice
miChoice.addItemListener( new MiItemListener( miChoice ) );
// Colocamos el objetos Choice sobre el Frame para poder verlo
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( miChoice );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instanciamos y registramos un objeto receptor de los eventos de
// la ventana, para recoger el evento de cierre del Frame y
// concluir la ejecucion de la aplicacion al recibirlo
miFrame.addWindowListener( new Conclusion() );
}
}
// Clase para recibir los eventos ItemListener generados por el objeto
// Choice de la aplicación
class MiItemListener implements ItemListener{
Choice oChoice;
MiItemListener( Choice choice ) {
// Guardamos una referencia al objeto Choice
oChoice = choice;
}
// Sobreescribimos el metodo itemStateChanged() del interfaz del
// ItemListener
public void itemStateChanged( ItemEvent evt ) {
System.out.println( oChoice.getSelectedItem() );
}
}
// Concluye la ejecución de la aplicación cuando el usuario cierra la
// ventana, porque se genera un evento windowClosing
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
System.exit( 0 );
}
}
```

Campos de Texto

Para la entrada directa de datos se suelen utilizar los campos de texto, que aparecen en pantalla como pequeñas cajas que permiten al usuario la entrada por teclado de una línea de caracteres.

Los campos de texto (TextField) son los encargados de realizar esta entrada, aunque también se pueden utilizar, activando su indicador de no-editable, para presentar texto en una sola línea con una apariencia en pantalla más llamativa, debido al bord smulando 3D que acompaña a este tipo de elementos.
La clase TextField extiende a la clase TextComponent, que extiende a su vez, extiende a la clase Component. Por ello, hay una gran cantidad de métodos que están accesibles desde los campos de texto. La clase TextComponent también es importante en las áreas de texto, en donde se permite la entrada de múltiples líneas de texto. La clase TextComponent es un Componente que permite la edición de texto. Tiene un campo y no dispone de constructores públicos, por lo que no es posible instanciar objetos de esta clase. Sin embargo, sí dispone de un amplio repertorio de métodos que son heredados por sus subclases, que permiten la manipulación del texto. Entre esos métodos hay algunos muy interesantes, como son los que permiten la selección o recuperación del texto marcado, desde programa; la indicación de editabilidad de texto; la recuperación de los eventos producidos por ese Componente, etc.

TextField

```
public TextField(String text);
```

Este constructor coloca la cadena que recibe como parámetro como y la especifica como valor inicial de nuestro TextField. También tenemos otros dos constructores interesantes por retomar:

```
public TextField(String text, int columns);
```

El cual a parte de inicializar con el valor de la cadena text también nos determinacuantas columnas mide nuestro TextField.

```
public TextField(int columns);
```

Este constructor solo recibe el parámetro columns, con el cual crea un TextField en blanco pero con el tamaño especificado por columns

addActionListener

```
public synchronized void addActionListener(ActionListener l);
```

Especifica un ActionListener que recibe eventos para ese TextField, en particular.

getText

```
public synchronized String getText();
```

El método regresa el texto completo que se encuentra en nuestro TextField.

getSelectedText

```
public synchronized String getSelectedText();
```

Retorna el texto que se encuentra seleccionado en ese instante. El siguiente programa nos muestra un TextField el cual viene inicializado con la cadena “Texto Inicial”, dándonos posibilidad de cambiarla, así como e seleccionar una parte de nuestro texto, detectando el momento que damos un ENTER y mostrando en ese mismo instante en pantalla el texto que escribimos.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploTextField {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
// Instancia un objeto TextField y coloca una cadena como
// Texto para que aparezca en el momento de su creación
```

```
TextField miCampoTexto = new TextField( "Texto inicial" );
// Instancia y registra un receptor de eventos de tipo Action
// sobre el campo de texto
miCampoTexto.addActionListener(new MiActionListener( miCampoTexto ) );
// Coloca la eqtiqueta sobre el objeto Frame
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( miCampoTexto );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instancia y registra un objeto receptor de eventos de ventana
// para concluir la ejecucion del programa cuando el Frame se
// cierres por accion del usuario sobre el
miFrame.addWindowListener( new Conclusion() );
}
}
// Clase para recibir los eventos de tipo Action que se produzcan
// sobre el objeto TextField sobre el cual se encuentra registrado
class MiActionListener implements ActionListener {
TextField oCampoTexto;
MiActionListener( TextField iCampoTexto ) {
// Guarda una referencia al objeto TextField
oCampoTexto = iCampoTexto;
}
// Se sobrescribe el método actionPerformed() del interfaz
// ActionListener para que indique en la consola el texto que
// se introduce
public void actionPerformed( ActionEvent evt ) {
System.out.println( "Texto seleccionado: " + oCampoTexto.getSelectedText() );
System.out.println( "Texto completo: " + oCampoTexto.getText() );
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Concluye el programa cuando se cierra la ventana
System.exit( 0 );
}
}
```

Layouts.

Los layout managers o manejadores de composición, en traducción literal, ayudan a adaptar los diversos Componentes que se desean incorporar a un Panel, es decir, especifican la apariencia que tendrán los Componentes a la hora de colocarlos sobre un Contenedor, controlando tamaño y posición (layout) automáticamente. En el tratamiento de los Layouts se utiliza un método de validación, de forma que los Componentes son marcados como no válidos cuando un cambio de estado afecta a la geometría o cuando el Contenedor tiene un hijo incorporado o eliminado. La validación se realiza automáticamente cuando se llama a pack() o show(). Los Componentes visibles marcados como no válidos no se validan automáticamente.

FlowLayout

Es el más simple y el que se utiliza por defecto en todos los Paneles si no se fuerza el uso de alguno de los otros. Los Componentes añadidos a un Panel con FlowLayout se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada componente. Si el Contenedor se cambia de tamaño en tiempo de ejecución, las posiciones de los componentes se ajustarán automáticamente, para colocar el máximo número posible de componentes en la primera línea.

Los Componentes se alinean según se indique en el constructor. Si no se indica nada, se

considera que los Componentes que pueden estar en una misma línea estarán centrados, pero también se puede indicar que se alineen a izquierda o derecha en el Contenedor.

CONSTRUCTOR FlowLayout

```
public FlowLayout(int align, int hgap, int vgap);
```

El constructor inicializa un objeto de tipo FlowLayout, mediante tres parámetros, uno que nos determina la alineación, un segundo que nos determina la separación horizontal y otro que nos determina la separación vertical.

setHgap y setVgap

```
public void setHgap(int hgap);
public void setVgap(int vgap);
```

Con este método actualizamos la separación entre los componentes, para determinar la separación horizontal usamos el Hgap y para la vertical el Vgap.

getHgap y getVgap

```
public int getHgap();
public int getVgap();
```

Este método nos regresa un valor entero indicándonos la separación entre los componentes, para determinar la separación horizontal usamos el Hgap y para la vertical el Vgap. En el ejemplo que se presenta a continuación, se puede observar como se modifica un Flowlayout dinámicamente en tiempo de ejecución. Utilizándolo a la vez como manejador de posicionamiento de estos botones, fijando una separación de 3 pixeles. Para después modificarlo utilizando los atributos Vgap y Hgap.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploFlowLayout {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
Frame miFrame = new Frame( "Curso, Java" );
// Instancia un objeto FlowLayout object alieado al Centro
// y con una separacion de 3 pixels en horizonal y vertical
FlowLayout miFlowLayout = new FlowLayout( FlowLayout.CENTER,3,3 );
// Se fija este FlowLayout para que sea el controlador de
// posicionamiento de componentes para el objeto Frame
miFrame.setLayout( miFlowLayout );
// Se instancian cinco objetos Button, para indicar los
// posicionamientos del FlowLayout
Button boton1 = new Button( "Primero" );
Button boton2 = new Button( "Segundo" );
Button boton3 = new Button( "Tercero" );
Button boton4 = new Button( "Cuarto" );
Button boton5 = new Button( "Quinto" );
// Se añaden los cinco botones al Frame en las mismas posiciones
// que vienen dadas por las etiquetas que se les han asignado en
// el constructor
miFrame.add( boton1 );
miFrame.add( boton2 );
miFrame.add( boton3 );
miFrame.add( boton4 );
miFrame.add( boton5 );
miFrame.setSize( 250,150 );
```

```
miFrame.setVisible( true );
// Instancia un objeto receptor de eventos de tipo action y
// lo registra para los cinco botones que se han añadido al
// objeto Frame
MiReceptorAction miReceptorAction =
new MiReceptorAction( miFlowLayout,miFrame );
boton1.addActionListener( miReceptorAction );
boton2.addActionListener( miReceptorAction );
boton3.addActionListener( miReceptorAction );
boton4.addActionListener( miReceptorAction );
boton5.addActionListener( miReceptorAction );
// Se instancia y registra un receptor de eventos de ventana
// para terminar la ejecucion del programa cuando se cierre
// el Frame
miFrame.addWindowListener( new Conclusion() );
}
}
class MiReceptorAction implements ActionListener {
FlowLayout miObjLayout;
Frame miObjFrame;
MiReceptorAction( FlowLayout layout,Frame frame ) {
miObjLayout = layout;
miObjFrame = frame;
}
// Cuando sucede un evento Action, se incrementa el espacio que
// que hay entre los componentes en el objeto FlowLayout.
// Luego se fija el controlador de posicionamiento al nuevo
// que se construye, y luego se valida el Frame para asegurar
// que se actualiza en la pantalla
public void actionPerformed( ActionEvent evt ){
miObjLayout.setHgap( miObjLayout.getHgap()+5 );
miObjLayout.setVgap( miObjLayout.getVgap()+5 );
miObjFrame.setLayout( miObjLayout );
miObjFrame.validate();
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Termina el programa cuando se cierra la ventana
System.exit( 0 );
}
}
```

Interfaz grafica de Flowlayout

BorderLayout

La composición BorderLayout (de borde) proporciona un esquema más complejo de colocación de los Componentes en un panel. La composición utiliza cinco zonas para colocar los Componentes sobre ellas: Norte, Sur, Este, Oeste y Centro. Es el layout o composición que se utilizan por defecto Frame y Dialog. El Norte ocupa la parte superior del panel, el Este ocupa el lado derecho, Sur la zona inferior y Oeste el lado izquierdo. Centro representa el resto que queda, una vez que

se hayan rellenado las otras cuatro partes. Pero esto tiene un pequeño inconveniente, ya que el número de componentes se limita a 5, lo cual no debiera de parecernos un limite ya que esos 5 Componentes, bien pueden ser contenedores. El constructor correspondiente la clase BorderLayout, recibe dos valores enteros, los cuales determinan la separación horizontal y vertical respectivamente. El ejemplo que se verá a continuación, se crearán cinco botones, los cuales mediante un controlador de eventos se incrementan los espacios tanto horizontales como verticales, en 5 pixeles. Es importante que no olvidemos validar el Frame, ya que si no lo hacemos, los cambios nunca se harán visibles.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploBorderLayout {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
Frame miFrame = new Frame( "Curso, Java" );
// Se instancia un objeto BorderLayout con una holgura en vertical y
// horizontal de 3 pixels

BorderLayout miBorderLayout = new BorderLayout( 3,3 );
// Se fija este BorderLayout para que sea el controlador de
// posicionamiento de componentes para el objeto Frame
miFrame.setLayout( miBorderLayout );
// Se instancian cinco objetos Button, para indicar los
// posicionamientos del BorderLayout
Button boton1 = new Button( "Sur" );
Button boton2 = new Button( "Oeste" );
Button boton3 = new Button( "Norte" );
Button boton4 = new Button( "Este" );
Button boton5 = new Button( "Centro" );
// Se añaden los cinco botones al Frame en las mismas posiciones
// que vienen dadas por las etiquetas que se les han asignado en
// el constructor
miFrame.add( boton1,"South" );
miFrame.add( boton2,"West" );
miFrame.add( boton3,"North" );
miFrame.add( boton4,"East" );
miFrame.add( boton5,"Center" );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instancia un objeto receptor de eventos de tipo action y
// lo registra para los cinco botones que se han añadido al
// objeto Frame
MiReceptorAction miReceptorAction = new MiReceptorAction( miBorderLayout,miFrame );
boton1.addActionListener( miReceptorAction );
boton2.addActionListener( miReceptorAction );
boton3.addActionListener( miReceptorAction );
boton4.addActionListener( miReceptorAction );
boton5.addActionListener( miReceptorAction );
// Se instancia y registra un receptor de eventos de ventana
// para terminar la ejecucion del programa cuando se cierre
// el Frame
miFrame.addWindowListener( new Conclusion() );
}
}
class MiReceptorAction implements ActionListener{
BorderLayout miObjBorderLayout;
```

```
Frame miObjFrame;
MiReceptorAction( BorderLayout layout,Frame frame ) {
miObjBorderLayout = layout;
miObjFrame = frame;
}
// Cuando sucede un evento Action, se incrementa el espacio que
// que hay entre los componentes en el objeto BorderLayout.
// Luego se fija el controlador de posicionamiento al nuevo
// que se construye, y luego se valida el Frame para asegurar
// que se actualiza en la pantalla
public void actionPerformed( ActionEvent evt ) {
miObjBorderLayout.setHgap( miObjBorderLayout.getHgap()+5 );
miObjBorderLayout.setVgap( miObjBorderLayout.getVgap()+5 );
miObjFrame.setLayout( miObjBorderLayout );
miObjFrame.validate();
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Termina el programa cuando se cierra la ventana
System.exit( 0 );
}
}
```

CardLayout

Este es el tipo de composición que se utiliza cuando se necesita una zona de la ventana que permita colocar distintos Componentes en esa misma zona. Este layout suele ir asociado con botones de selección (Choice), de tal modo que cada selección determina el panel (grupo de componentes) que se presentarán. Quizá lo más interesante del CardLayout sean los métodos que permiten mostrar los distintos componentes, uno, después de otro. Dichos métodos son muy similares y se describen a continuación: first, next, previous, y last

```
public void first(Container parent);
public void next(Container parent);
public void previous(Container parent);
public void last(Container parent);
```

Lo que hacen estos métodos es mostrarnos un componente en pantalla, ya sea el primero(first), el siguiente(next), el anterior(previous) y el último(last). En el siguiente ejemplo veremos la aplicación de un CardLayout usando el método next, para mostrar una serie de cinco botones en pantalla.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploCardLayout {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
Frame miFrame = new Frame( "Curso, Java" );
// Se instancia un objeto BorderLayout con una holgura en vertical y
// horizontal de 3 pixels
CardLayout miCardLayout = new CardLayout();
// Se fija este CardLayout para que sea el controlador de
// posicionamiento de componentes para el objeto Frame
miFrame.setLayout( miCardLayout );
// Se instancian cinco objetos de tipo Button
Button boton1 = new Button( "Primero" );
Button boton2 = new Button( "Segundo" );
Button boton3 = new Button( "Tercero" );
```

```
Button boton4 = new Button( "Cuarto" );
Button boton5 = new Button( "Quinto" );
//Se añaden los cinco botones al Frame en las mismas posiciones
//que vienen dadas por las etiquetas que se les han asignado en
//el constructor
miFrame.add( boton1,"primero" );
miFrame.add( boton2,"segundo" );
miFrame.add( boton3,"tercero" );
miFrame.add( boton4,"cuarto" );
miFrame.add( boton5,"quinto" );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instancia un objeto receptor de eventos de tipo action y
// lo registra para los cinco botones que se han añadido al
// objeto Frame
MiReceptorAction miReceptorAction = new MiReceptorAction( miCardLayout,miFrame );
boton1.addActionListener( miReceptorAction );
boton2.addActionListener( miReceptorAction );
boton3.addActionListener( miReceptorAction );
boton4.addActionListener( miReceptorAction );
boton5.addActionListener( miReceptorAction );
// Se instancia y registra un receptor de eventos de ventana
// para terminar la ejecucion del programa cuando se cierre
// el Frame
miFrame.addWindowListener( new Conclusion() );
}
}
class MiReceptorAction implements ActionListener{
CardLayout miObjCardLayout;
Frame miObjFrame;
MiReceptorAction( CardLayout layout,Frame frame ) {
miObjCardLayout = layout;
miObjFrame = frame;

}
// Cuando sucede un evento Action, se coloca el siguiente elemento,
// que se encuentre en nuestro contenedor
public void actionPerformed( ActionEvent evt ) {
miObjCardLayout.next(miObjFrame);
miObjFrame.setLayout( miObjCardLayout );
miObjFrame.validate();
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Termina el programa cuando se cierra la ventana
System.exit( 0 );
}
}
```

Posicionamiento Absoluto

Los Componentes se pueden colocar en Contenedores utilizando cualquiera de loscontroladores de posicionamiento, o utilizando posicionamiento absoluto para realizar esta función. La primera forma de colocar los Componentes se considera más segura porque automáticamente serán compensadas las diferencias que se puedan encontrar entre resoluciones de pantalla de plataformas distintas. La clase Component proporciona métodos para especificar la posición y tamaño de un Componente en coordenadas absolutas indicadas en pixeles:

```
setBounds( int, int, int, int);
setBounds( Rectangle);
```

La posición y tamaño si se especifica en coordenadas absolutas, puede hacer más difícil la consecución de una apariencia uniforme, en diferentes plataformas, de la interfaz, pero, a pesar de ello, es interesante saber cómo se hace Las dos versiones sobrecargadas del método setBounds se describen a continuación:

setBounds

```
public void setBounds(int x, int y, int width, int height);
```

Localiza en “x” y “y” el punto inicial, de la ubicación de nuestro componente, para luego sumarle a “x” el valor de “width” y a “y” el valor de “height”.

```
public void setBounds(Rectangle r);
```

Este método localiza a nuestro componente dentro del rectángulo, que recibe. Esto es una gran ventaja, ya que usar este método resulta más flexible gracias a los siete constructores de la clase Rectangle. El ejemplo que se marca a continuación utiliza el setBounds para indicar la posición y el tamaño de un Botón y de un Objeto Label.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploPosicionamiento {
public static void main( String args[] ) {
// Instancia un objeto de tipo Interfaz Hombre-Maquina
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
// La siguiente clase se utiliza para instanciar un objeto de tipo
// Interfaz Grafica de Usuario
class AmbienteGrafico {
public AmbienteGrafico() {
// Se crea un objeto Button con el texto que se pasa como
// parametro y el tamaño y posicion indicadas dentro de
// su contenedor (en pixels)
Button miBoton = new Button( "Boton" );
// Al rectagulo se le pasan los parametros: x,y,ancho,alto
miBoton.setBounds( new Rectangle( 25,20,100,75 ) );
// Se crea un objeto Label con el texto que se indique como

// parametro en la llamada y el tamaño especificado y en la
// posicion que se indique dentro de su contenedor (en pixels)
// Se pone en amarillo para que destaque
Label miEtiqueta = new Label( "Curso Java" );
miEtiqueta.setBounds( new Rectangle( 100,75,100,75 ) );
miEtiqueta.setBackground( Color.yellow );
// Se crea un objeto Frame con el titulo que se indica en la
// llamada y sin ningun layout
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( null );
// Añade los dos componentes al Frame, fijando su tamaño en
// pixels y lo hace visible
miFrame.add( miBoton );
miFrame.add( miEtiqueta );
miFrame.setSize( 250,175 );
miFrame.setVisible( true );
}
}o ambiente = new AmbienteGrafico();
```

GridLayout

La composición GridLayout proporciona gran flexibilidad para situar Componentes. El controlador de posicionamiento se crea con un determinado número de filas y columnas y los Componentes van dentro de las celdas de la tabla así definida. Si el Contenedor es alterado

en su tamaño en tiempo de ejecución, el sistema intentará mantener el mismo número de filas y columnas dentro de los márgenes de separación que se hayan indicado. En este caso, estos márgenes tienen prioridad sobre el tamaño mínimo que se haya indicado para los Componentes, por lo que puede llegar a conseguirse que sean de un tamaño tan pequeño que sus etiquetas sean ilegibles.Algunos métodos importantes se detallan a continuación:

• CONSTRUCTORES DE GridLayout

```
public GridLayout(int rows, int cols);
```

Este constructor inicializa recibe como parámetros el número de filas (rows) y de columnas(cols), que tendrá nuestro GridLayout

```
public GridLayout(int rows, int cols, int hgap, int vgap);
```

A parte de recibir las filas y las columnas, recibe también las separaciones horizontales(hgap) y verticales(vgap).

• setRows y setColumns

```
public void setRows(int rows);
public void setColumns(int cols);
```

Estos métodos sirven para actualizar el valor de las filas(rows) y columnas(columns) del GridLayout. A continuación se presenta un ejemplo, el cual modifica mediante dos botones las filas y columnas de un GridLayout. El GridLayout sé inicializa con dos filas y tres columnas, dándole posibilidad al usuario de cambiar a 3 filas y dos columnas.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploGridLayout {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
// Se instancian los dos botones que van a proporcionar la
// funcionalidad a la aplicacion
Button boton7 = new Button( "3x2" );
Button boton8 = new Button( "2x3" );
// Se instancia un objeto layout de tipo GridLayout para ser
// utilizado con el Panel
GridLayout miGridLayout = new GridLayout( 2,3 );
// Instancia el primero de los dos objetos Panel que sera
// integrado en el objeto Frame
Panel panel1 = new Panel();
// Fijamos el layout que habiamos definido para el panel
panel1.setLayout( miGridLayout );
// Se colocan los seis botones sobre el panel con una
// etiqueta indicando su numero
for( int i=0; i < 6; i++)
panel1.add( new Button( "Boton"+i ) );
// Se instancia el segundo objeto Panel utilizando el FlowLayout
// por defecto y se colocan los dos botones funcionales sobre el.
// A estos botones se les añadira su funcionalidad a traves de
// objetos receptores de los eventos ActionListener registrados
// sobre ellos
Panel panel2 = new Panel();
panel2.add( boton7 );
panel2.add( boton8 );
```

```
// Se instancia el objeto Frame, que sera el padre de todo
// el interfaz de usuario que se esta creando
Frame miFrame = new Frame( "Curso, Java" );
// IMPORTANTE: Se añaden los dos objetos Panel que
// preparado al objeto Frame para crear el interfaz definitivo
miFrame.add( panel1,"North" );
miFrame.add( panel2,"South" );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Se instancian los objetos Receptores de eventos Action
// registran con los botones 7 y 8, que corresponden al
// segundo Panel y que van a modificar el posicionamiento
// los otrs seis botones en el Panel contiguo
boton7.addActionListener(
new A3x2ActionListener( miGridLayout,miFrame ) );
boton8.addActionListener(
new A2x3ActionListener( miGridLayout,miFrame ) );
// Se termina de dar funcionalidad al interfaz, instanciando y
// registrando un objeto receptor de eventos de la ventana para
// concluir la ejecucion de la aplicacion cuando el usuario cierre
// el Frame
miFrame.addWindowListener( new Conclusion() );
}
}
// Las siguientes dos clases son las clases de los ActionListener,
// los receptores de eventos de tipo Action. Un objeto de cada una
// de ellas se instancia y registra sobore cada uno de los dos
// botones funcionales de la aplicacion. El proposito de estos
// controladores de eventos es modificar el layout del panel
// contiguo, de forma que los botones que estan colocados sobre
// el se cambien de posicion
class A3x2ActionListener implements ActionListener {
GridLayout miObjGridLayout;
Frame miObjFrame;
A3x2ActionListener( GridLayout layout,Frame frame ) {
miObjGridLayout = layout;
miObjFrame = frame;
}
// Cuando se produce un evento Action, se fijan las filas a 3 y
// la columnas a 2 sobre el objeto GridLayout. Luego se fija el
// controlador de posicionamiento para que sea el nuevo que
// acabamos de modificar, y posteriormente se valida el Frame
// para asegurarse de que el alyout es valido y tendra efecto
// sobre la visualizacion en pantalla
public void actionPerformed( ActionEvent evt ) {
miObjGridLayout.setRows( 3 );
miObjGridLayout.setColumns( 2 );
miObjFrame.setLayout( miObjGridLayout );
miObjFrame.validate();
}
}
class A2x3ActionListener implements ActionListener {
GridLayout miObjGridLayout;
Frame miObjFrame;
A2x3ActionListener( GridLayout layout,Frame frame ) {
miObjGridLayout = layout;
miObjFrame = frame;
}
// Cuando se produce un evento Action, se fijan las filas a 2 y
// la columnas a 3 sobre el objeto GridLayout. Luego se fija el
// controlador de posicionamiento para que sea el nuevo que
// acabamos de modificar, y posteriormente se valida el Frame
// para asegurarse de que el alyout es valido y tendra efecto
// sobre la visualizacion en pantalla
```

```
public void actionPerformed( ActionEvent evt ) {
miObjGridLayout.setRows( 2 );
miObjGridLayout.setColumns( 3 );
miObjFrame.setLayout( miObjGridLayout );
miObjFrame.validate();
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Termina la ejecucion del programa cuando se cierra la
// ventana principal de la aplicacion
System.exit( 0 );
}
}
```

Interfaz grafica de GridLayout

Listas

Las listas (List) aparecen en las interfaces de usuario para facilitar a los operadores la manipulación de muchos elementos. Se crean utilizando métodos similares a los de los botones

Choice. La lista es visible todo el tiempo, utilizándose una barra de desplazamiento para visualizar los elementos que no tienen lugar en el área de la lista que aparece en la pantalla.
La clase List extiende la clase Component e implementa la interfaz ItemSelectable, que es la interfaz que contiene un conjunto de items en los que puede haber, o no, alguno seleccionado. Además, soporta el método addActionListener() que se utiliza para recoger los eventos ActionEvent que se produce cuando el usuario pica dos veces con el ratón sobre un elemento de la lista.

• add

```
public void add(String item, int index);
```

Recibe un String con el texto que deseamos mostrar así como el número por el cual queremos que se ordene en la lista.

• setMultipleMode

```
public void setMultipleMode(boolean b);
```

Nos permite determinar mediante una variable booleana si deseamos o no, elegir más de un elemento de la lista.

• select

```
public void select(int index);
```

Este método se posiciona en el elemento de la lista, que está señalado por el parámetro index.

• getSelectedItem

```
public synchronized String getSelectedItem();
```

El método regresa la cadena del elemento seleccionado. Se utiliza para selección de tipo simple o sea de un solo elemento de la lista.

• getSelectedItems

```
public synchronized String[] getSelectedItems();
```

El método regresa un arreglo de cadenas que contiene a los elementos seleccionados.
Se utiliza para selección de tipo múltiple o sea de más de un elemento de la lista.

• addActionListener

```
public synchronized void addActionListener();
```

Sirve para añadir acciones cuando el usuario da un doble click sobre la lista. El siguiente ejemplo nos muestra un programa en el cual definimos una lista, con la posibilidad de seleccionar de uno a más elementos. En él podemos ver aplicados los métodos descritos en párrafos anteriores.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploListas {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico(){

// Instancia un objeto List y coloca algunas cadenas sobre el,
// para poder realizar selecciones
List miLista = new List();
for( int i=0; i < 15; i++ )
miLista.add( "Elemento "+i );
// Activa la seleccion multiple
miLista.setMultipleMode( true );
// Presenta el elemento 1 al inicio
miLista.select( 1 );
// Instancia y registra un objeto ActionListener sobre el objeto
// List. Se produce un evento de tipo Action cuando el usuario
// pulsa dos veces sobre un elemento
miLista.addActionListener( new MiListaActionListener( miLista ) );
// Instancia un objeto Button para servicio de la seleccion
// multiple. Tambien instancia y registra un objeto ActionListener
// sobre el boton
Button miBoton = new Button( "Selecciona Multiples Items" );
miBoton.addActionListener( new miBotonActionListener( miLista ) );
// Coloca el objeto List y el objeto Button el el objeto Frame
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( miLista );
miFrame.add( miBoton );
miFrame.setSize( 250,150 );
miFrame.setVisible( true );
// Instancia y registra un objeto receptor de eventos de ventana
// para concluir la ejecucion del programa cuando el Frame se
// cierres por accion del usuario sobre el
miFrame.addWindowListener( new Conclusion() );
}
}
```

```
// Clase para recibir eventos de tipo ActionListener sobre el
// objeto List. Presenta en elemento seleccionado cuando el usuario
// pulsa dos veces sobre un item de lista cuando la seleccion es
// individual. Si el usuario pica dos veces sobre una seleccion
// multiple, se produce un evento pero el metodo getSelectedItem()
// de la clase List devuelve null y no se presenta nada en pantalla
class MiListaActionListener implements ActionListener {
List oLista;
MiListaActionListener( List lista ) {
// Salva una referencia al objeto List
oLista = lista;
}
// Sobreescribe el metodo actionPerformed() del interfaz
// ActionListener
public void actionPerformed( ActionEvent evt ) {
if( oLista.getSelectedItem() != null ) {
System.out.println( "Seleccion Simple de Elementos" );
System.out.println( " "+oLista.getSelectedItem() );
}
}
}
// Clase para recoger los eventos Action que se produzcan sobre el
// objeto Button. Presenta los elementos que haya seleccionados
// cuando el usuario lo pulsa, incluso aunque solamente haya uno
// marcado. Si no hubiese ninguno, so se presentaria nada en
// la pantalla
class miBotonActionListener implements ActionListener {
List oLista;
miBotonActionListener( List lista ) {
// Salva una referencia al objeto List
oLista = lista;
}
// Sobreescribe el metodo actionPerformed() del interfaz
// ActionListener
public void actionPerformed( ActionEvent evt ) {
String cadena[] = oLista.getSelectedItems();
if( cadena.length != 0 ) {
System.out.println( "Seleccion Multiple de Elementos" );
for( int i=0; i < cadena.length; i++ )
System.out.println( " "+cadena[i] );
}
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Concluye el programa cuando se cierra la ventana
System.exit(0);
}
}
```

Areas de Texto

Un área de texto (TextArea) es una zona multilínea que permite la presentación de texto, que puede ser editable o de sólo lectura. Al igual que la clase TextField, esta clase extiende la clase TextComponent y dispone de cuatro campos, que son constantes simbólicas que pueden ser utilizadas para especificar la información de colocación de las barras de desplazamiento en algunos de los constructores de objetos TextArea. Estas constantes simbólicas son:

- SCROLLBARS_BOTH que crea y presenta barras de desplazamiento horizontal y vertical
- SCROLLBARS_NONE que no presenta barras de desplazamiento
- SCROLLBARS_HORIZONTAL_ONLY que crea y presenta solamente barras de desplazamiento horizontal

• SCROLLBARS_VERTICAL_ONLY que crea y presenta solamente barras dedesplazamiento vertical

Esta clase TextArea contiene muchos métodos y, además, hay que tener en cuenta que hereda un sin fin de métodos definidos en las clases TextComponent, Component y Object, por lo que sería muy recomendable recurrir a la documentación del API que proporciona Sun para tener una correcta definición de cada uno de ellos. A continuación mencionamos algunos que brillan por su importancia.

• CONSTRUCTOR TextArea

```
public TextArea(String text, int rows, int columns, int scrollbars);
```

Este constructor recibe una cadena, la cual va a inicializar nuestra TextArea, así como también recibe tres valores enteros uno para las filas, otro para las columnas y un tercero para definir las scrollbars, por lo regular en este último parámetro se usan las constantes que están definidas, en los párrafos anteriores.

• append

```
public void append(String str);
```

El método append, recibe y coloca en nuestra TextArea un texto con formato; Se pueden incluir variables con la finalidad de desplegar su valor en nuestra TextArea.

• getText

```
public synchronized String getText();
```

Al estar heredando métodos de TextComponent, es lógico que si nosotros escribimos un getText, seguramente lo que nos regrese será la cadena que se encuentra contenida en el TextArea.

• addTextListener

```
public synchronized void addTextListener(Textlistener l);
```

Este método registra un objeto TextListener sobre la TextArea, el cual recogerá valores de tipo TexEvent, los cuales se producen cuando cambia algún valor en el área de texto. En el siguiente programa, se realiza un procesado de texto a muy bajo nivel, generándose un evento cada vez que cambie un solo carácter.

```
import java.awt.*;
import java.awt.event.*;
public class EjemploTextArea {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
public AmbienteGrafico() {
// Instancia un objeto TextArea, con una barra de desplazamiento
// vertical y lo inicializa con diez líneas de texto
TextArea miAreaTexto = new TextArea( "",5,20,
TextArea.SCROLLBARS_VERTICAL_ONLY );
for( int i=0; i < 10; i++ )
miAreaTexto.append( "linea "+i+"\n" );
// Instancia y registra un receptor de eventos de tipo Text
// sobre el área de texto
miAreaTexto.addTextListener(new MiTextListener( miAreaTexto ) );
// Coloca el área de texto sobre el objeto Frame
Frame miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( miAreaTexto );
miFrame.setSize( 250,150 );
```

```
miFrame.setVisible( true );
// Instancia y registra un objeto receptor de eventos de ventana
// para concluir la ejecucion del programa cuando el Frame se
// cierre por accion del usuario sobre el
miFrame.addWindowListener( new Conclusion() );
}
}
// Clase para recibir los eventos de tipo Text que se produzcan
// sobre el objeto TextArea sobre el cual se encuentra registrado
class MiTextListener implements TextListener {
TextArea oAreaTexto;
MiTextListener( TextArea iAreaTexto ) {
// Guarda una referencia al objeto TextArea
oAreaTexto = iAreaTexto;
}
// Se sobrescribe el método textValueChanged() del interfaz
// TextListener para que indique en la consola el texto que
// ocupa el área de texto cuando se cambie

public void textValueChanged( TextEvent evt ) {
System.out.println( oAreaTexto.getText() );
}
}
class Conclusion extends WindowAdapter {
public void windowClosing( WindowEvent evt ) {
// Concluye el programa cuando se cierra la ventana
System.exit( 0 );
}
}
```

Contenedores.

La clase Container es una clase abstracta derivada de Component, que representa a cualquier componente que pueda contener otros componentes. Se trata, en esencia, de añadir a la clase Component la funcionalidad de adición, sustracción, recuperación, control y organización de otros componentes.

Al igual que la clase Component, no dispone de constructores públicos y, por lo tanto, no se pueden instanciar objetos de la clase Container. Sin embargo, sí se puede extender para implementar la nueva característica incorporada a Java en el JDK 1.1, de los componentes Lightweight.

El AWT proporciona varias clases de Contenedores:

Panel
ScrollPane
Window
Dialog
FileDialog
Frame

Aunque los que se pueden considerar como verdaderos Contenedores son Window, Frame, Dialog y Panel, porque los demás son subtipos con algunas características determinadas y solamente útiles en circunstancias muy concretas.

Window

Es una superficie de pantalla de alto nivel (una ventana). Una instancia de la clase Window no puede estar enlazada o embebida en otro Contenedor. El controlador de posicionamiento de Componentes por defecto, sobre un objeto Window, es el BorderLayout.

Una instancia de esta clase no tiene ni título ni borde, así que es un poco difícil de justificar su uso para la construcción directa de una interfaz gráfica, porque es mucho más sencillo utilizar objetos de tipo Frame o Dialog. Dispone de varios métodos para alterar el tamaño y título de la ventana, o los cursores y barra de menús.

Frame

Es una superficie de pantalla de alto nivel (una ventana) con borde y título. Una instancia de la clase Frame puede tener una barra de menú. Una instancia de esta clase es mucho más aparente y más semejante a lo que todos entendemos como una ventana.

Líneas antes hemos ya utilizado la clase Frame, ya que todos los objetos los hemos contenido dentro de un objeto de este tipo. La clase Frame extiende a la clase Window, y su controlador de posicionamiento de Componentes por defecto es el BorderLayout.

Los objetos de tipo Frame son capaces de generar varios tipos de eventos, de los cuales el más interesante es el evento de tipo windowClosing, este evento se usa de forma exhaustiva y se produce cuando el usuario da un click en el botón cerrar, ubicado usualmente, en la parte superior derecha del Frame. Los métodos más usuales de esta clase son:

• add

```
public Component add(Component comp);
```

Añade un objeto a nuestro Frame.

• setSize

```
public void setSize(int width, int height);
```

Selecciona el tamaño del Frame.

• setVisible

```
public void setVisible(boolean b);
```

Hace que el Frame se visualice.

• setCursor

```
public synchronized void setCursor(Cursor cursor);
```

Elige el cursor del mouse.

• addWindowListener

```
public synchronized void addWindowListener(WindowListener l);
```

Registra un objeto de tipo Window Listener, sobre el Frame, este objeto será el que reciba eventos como el windowClosing.

El siguiente ejemplo inserta tres botones cada uno con su respectiva acción, dentro de un Frame.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class EjemploFrame {
public static void main( String args[] ) {
AmbienteGrafico ambiente = new AmbienteGrafico();
}
}
class AmbienteGrafico {
Frame miFrame;
public AmbienteGrafico() {
// Se instancian tres botones con textos indicando lo que
// hacen cuando se pulse sobre ellos
Button botonTitulo = new Button( "Imprime Titulo" );
Button botonCursorMano = new Button( "Cursor Mano" );
Button botonCursorFlecha = new Button( "Cursor Flecha" );
// Instancia un objeto Frame con su titulo indicativo de que se
// se trata, utilizando un FlowLayout
miFrame = new Frame( "Curso, Java" );
miFrame.setLayout( new FlowLayout() );
```

```
// Añade tres objetos Button al Frame
miFrame.add( botonTitulo );
miFrame.add( botonCursorMano );
miFrame.add( botonCursorFlecha );
// Fija el tamaño del Frame y lo hace visible
miFrame.setSize( 250,200 );

miFrame.setVisible( true );
// Instancia y registra objetos ActionListener sobre los
// tres botones utilizando la sintaxis abreviada de las
// clases anidadas
botonTitulo.addActionListener( new ActionListener() {
public void actionPerformed( ActionEvent evt ) {
System.out.println( miFrame.getTitle() );
}
} );
botonCursorMano.addActionListener( new ActionListener() {
public void actionPerformed(ActionEvent evt ) {
miFrame.setCursor( new Cursor( Cursor.HAND_CURSOR ) );
}
} );
botonCursorFlecha.addActionListener( new ActionListener() {
public void actionPerformed( ActionEvent evt ) {
miFrame.setCursor( new Cursor( Cursor.DEFAULT_CURSOR ) );

}
} );
// Instancia y registra un objeto WindowListener sobre el objeto
// Frame para terminar el programa cuando el usuario haga click
// con el raton sobre el boton de cerrar la ventana que se
// coloca sobre el objeto Frame
miFrame.addWindowListener( new WindowAdapter() {
public void windowClosing( WindowEvent evt ) {
// Concluye la aplicacion cuando el usuario cierra la
// ventana
System.exit( 0 );
}
} );
}
}
```

Dialog

Es una superficie de pantalla de alto nivel (una ventana) con borde y título, que permite entradas al usuario. La clase Dialog extiende la clase Window, que extiende la clase Container, que extiende a la clase Component; y el controlador de posicionamiento por defecto es el BorderLayout.

De los constructores proporcionados por esta clase, destaca el que permite que el diálogo sea o no modal. Todos los constructores requieren un parámetro Frame y, algunos de ellos, permiten la especificación de un parámetro booleano que indica si la ventana que abre el diálogo será modal o no. Si es modal, todas las entradas del usuario serán recogidas por esta ventana, bloqueando cualquier entrada que sepudiese producir sobre otros objetos presentes en la pantalla posteriormente, si no se ha especificado que el diálogo sea modal, se puede hace que adquiera esta característica invocando al método setModal().

Panel

La clase Panel es un Contenedor genérico de Componentes. Una instancia de la clase Panel, simplemente proporciona un Contenedor al que ir añadiendo Componentes.

El controlador de posicionamiento de Componentes sobre un objeto Panel, por defecto es el FlowLayout; aunque se puede especificar uno diferente en el constructor a la hora de instanciar el

objeto Panel, o aceptar el controlador de posicionamiento inicialmente, y después cambiarlo invocando al método setLayout(). Panel dispone de un método addNotify(), que se utiliza para crear un observador general (peerPerr) del Panel. Normalmente, un Panel no tiene manifestación visual alguna por sí mismo, aunque puede hacerse notar fijando su color de fondo por defecto a uno diferente del que utiliza normalmente.
La clase Panel al ser un contenedor, posee métodos muy similares a los del Frame, como el add, entre otros. A continuación describiremos el método setBackground ya que se encuentra dentro del ejemplo

• setBackground

```
public void setBackground(Color c);
```

Indica el color de fondo del panel, aunque puede ser aplicado a cualquier superclase de la clase Componente. En el programa que se verá líneas abajo se incorporan tres objetos de tipo Panel a uno de tipo Frame. El controlador de posicionamiento de los Componentes para el objeto Frame se especifica concretamente para que sea un FlowLayout, y se alteran los colores de fondo de los objetos Panel para que sean claramente visibles sobre el Frame. Sobre cada uno de los paneles se coloca un objeto, utilizando el método add(), de tal modo que se añade un objeto de tipo campo de texto sobre el Panel de fondo amarillo, un objeto de tipo etiqueta sobre el Panel de fondo rojo y un objeto de tipo botón sobre el Panel de fondo azul.

Ninguno de los Componentes es activo, ya que no se instancian ni registran objetos receptores de eventos sobre ellos. Así, por ejemplo, el único efecto que se puede observar al pulsar el botón del panel azul, se limita al efecto visual de la pulsación. Añadir Componentes a un Contenedor
Para que una interfaz sea útil, no debe estar compuesto solamente por Contenedores, éstos deben tener Componentes en su interior. Los Componentes se añaden al Contenedor invocando al método add() del Contenedor. Este método tiene tres formas de llamada que dependen del manejador de composición o layout manager que se vaya a utilizar sobre el Contenedor

Menús

Los Menús, sin duda, son siempre el centro de la aplicación, porque son el cual el usuario interactúa con la aplicación. La clave de una buena implementación de un menú, está en que su diseño sea tanto agradable deseamos que el lector genere su propio criterio en base a esas rutinas.
En Java, la jerarquía de clases que intervienen en la construcción y manipulación menús es la que se muestra en la lista siguiente:

ja va .lan g .O b je c t
Ja va .aw t.M e n uCom p o n e n t
Ja va .aw t.M e n u B a r
Ja va .aw t.M enu
Ja va.aw t.M enuItem
Java.awt.checkboxMen u Item
Ja va .aw t.P opu pM e n u

A continuación se exploran una a una las clases que se acaban de citar.

MenuComponent

Es la superclase de todos los Componentes relacionados con menús. Esta clase no contiene campos, solamente tiene un constructor y dispone de una docena de métodos que están accesibles a todas sus subclases.

Menu

Es un Componente de una barra de menú. Esta es la clase que se utiliza para construir los menús

que se manejan habitualmente, conocidos como menús de persiana (o pulldown). Dispone de varios constructores para poder, entre otras cosas, crear los menús con o sin etiqueta. No tiene campos y proporciona varios métodos que se pueden utilizar para crear y mantener los menús en tiempo de ejecución. En el programa de ejemplo java1317.java, se usarán algunos de ellos. Esta clase se sirve del método add para añadir objetos de tipo MenuItem.

MenuItem

Representa una opción en un menú. Esta clase se emplea para instanciar los objetos que constituirán los elementos seleccionables del menú. No tiene campos y dispone de varios constructores, entre los que hay que citar a:

```
MenuItem( String, menushortcut);
```

El cual crea un elemento el menú con una combinación de teclas asociada para acceder directamente a él. Esta clase proporciona una veintena de métodos, entre los que destacan los que se citan ahora:

- addActionListener(actionlistener) que añade el receptor específico que va a recibir eventos desde esa opción del menú.
- removeActionListener(actionlistener) contrario al anterior, por lo que ya no se recibirán eventos desde esa opción del menú.
- setEnabled(boolean) indica si esa opción del menú puede estar o no seleccionable.
- isEnabled() comprueba si la opción del menú esta habilitada.

El método addActionListener() ya debería resultar familiar al lector. Cuando se selecciona una opción de un menú, bien a través del ratón o por la combinación rápida de teclas, se genera un evento de tipo ActionEvent. Para que la selección de la opción en un menú ejecute una determinada acción, se ha de instanciar y registrar un objeto ActionListener que contenga el método actionPerformed() sobreescrito para producir la acción deseada. En el ejemplo java1317.java, solamente se presenta en pantalla la identificación de la opción de menú que se ha seleccionado; en un programa realmente útil, la acción seguramente deberá realizar algo más interesante que eso.

MenuShortcut

Representa el acelerador de teclado, o la combinación de teclas rápidas, para acceder a un MenuItem. Esta clase se utiliza para instanciar un objeto que representa un acelerador de teclado, o una combinación de teclas rápidas, para un determinado MenuItem. No tiene campos y dispone de dos constructores. Aparentemente, casi todas las teclas rápidas consisten en mantener pulsada la tecla Control a la vez que se pulsa cualquier otra tecla. Uno de los constructores de esta clase:

```
MenuShortcut( int, boolean);
```

Dispone de un segundo parámetro que indica si el usuario ha de mantener también pulsada la tecla de cambio a mayúsculas (Shift). El primer parámetro es el código de tecla, que es el mismo que se devuelve en el campo keyCode del evento KeyEvent, cuando se pulsa una tecla. La clase KeyEvent define varias constantes simbólicas para estos códigos de teclas, como son: VK_8, VK_9, VK_A, VK_B.

MenuBar

Encapsula el concepto de una barra de menú en un Frame. No tiene campos, sólo tiene un constructor público, y es la clase que representa el concepto que todo usuario tiene de la barra de menú que está presente en la mayoría de las aplicaciones gráficas basadas en ventanas. Posee el método add para añadir menús.

CheckboxMenuItem

Genera una caja de selección que representa una opción en un menú. Esta clase se utiliza para instanciar objetos que puedan utilizarse como opciones en un menú. Al contrario que las opciones de menú que se han visto al hablar de objetos MenuItem, estas opciones tienen mucho más parentesco con las cajas de selección.

Esta clase no tiene campos y proporciona tres constructores públicos, en donde se puede especificar el texto de la opción y el estado en que se encuentra. Si no se indica nada, la opción dejará de estar seleccionada, aunque hay un constructor que permite indicar en un parámetro de tipo booleano, que la opción se encuentra seleccionada, o marcada, indicando true en ese valor.

De los métodos que proporciona la clase, quizá el más interesante sea el método que tiene la siguiente declaración:

```
addItemListener( ItemListener )
```

Cuando se selecciona una opción del menú, se genera un evento de tipo ItemEvent. Para que se produzca la acción que se desea con esa selección, es necesario instanciar y registrar un objeto ItemListener que contenga el método itemStateChanged() sobreescrito con la acción que se quiere. Por ejemplo, en el programa que se presenta a continuación, la acción consistirá en presentar la identificación y estado de la opción de menú que se haya seleccionado.

En el siguiente programa nos encontraremos con un menú con tres opciones de tipo CheckboxMenuItem. Estas acciones hacen que el estado de la opción cambie entre “on” y “off”, y ese estado se puede conocer a través del método getState().

Al seleccionar una, se genera un evento de tipo ItemEvent, que contiene información del nuevo estado, del texto de la opción y del nombre asignado a la opción. Estos datos pueden utilizarse para identificar cuál de las opciones ha cambiado y, también, para implementar la acción requerida que, en este caso del ejemplo siguiente, consiste en presentar una línea de texto en la pantalla con la información que contiene el objeto ItemEvent, más o menos semejante a la que se reproduce a continuación:

```
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem2,label=Tercer Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=false]
java.awt.CheckboxMenuItem[chkmenuitem1,label=Segundo Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=true]
```

Como siempre, sé instancia y registra sobre el Frame un objeto receptor de eventos windowClosing() para la conclusión del programa cuando se cierre el Frame.

PopupMenu

Implementa un menú que puede ser presentado dinámicamente dentro de un Componente. Esta clase se utiliza para instanciar objetos que funcionan como menús emergentes o pop-up. Una vez que el menú aparece en pantalla, el procesado de las opciones es el mismo que en el caso de los menús de persiana.

Esta clase no tiene campos y proporciona un par de constructores y un par de métodos, de los cuales el más interesante es el método show(), que permite mostrar el menú emergente en una posición relativa al Componente origen. Este Componente origen debe estar contenido dentro de la jerarquía de padres de la clase PopupMenu. En el código que se va a mostrar a continuación se verá el funcionamiento de un menúpop-up, cabe resaltar que dicho menú no se añade al Frame con el método setMenubar, sino con el ya conocido método add. El

programa se vale de un objeto de tipo MouseListener para poder hacer que se despliegue el menú pop-up.

2.3 *SWING*

Swing es un extenso conjunto de componentes que van desde los más simples, como etiquetas, hasta los más complejos, como tablas, árboles, y documentos de texto con estilo. Casi todos los componentes Swing descienden de un mismo padre llamado JComponent que desciende de la clase de AWT Container. Es por ello que Swing es más una capa encima de AWT que una sustitución del mismo. La figura 1.2 muestra una parte de la jerarquía de JComponent. Si la compara con la jerarquía de Component notará que para cada componente AWT hay otro equivalente en Swing que empieza con "J". La única excepción es la clase de AWT Canvas, que se puede reemplazar con JComponent, JLabel, o JPanel Asimismo se percatará de que existen algunas clases Swing sin su correspondiente homólogo.
La figura 2.1 representa sólo una pequeña fracción de la librería Swing, pero esta fracción son las clases con las que se enfrentará más a menudo. El resto de Swing existe para suministrar un amplio soporte y la posibilidad de personalización a los componentes estas clases definen.
Swing es un conjunto de clases que permiten construir una interfaz gráfica de usuario independiente de la plataforma en que corre la aplicación. Swing ofrece varias ventajas sobre awt, como son: más controles, diálogos prefabricados, LookAndFeel (cambiar la aperiencia de los controles), ModelViewControler(Separar el contenido de la presentación) , entre otros.

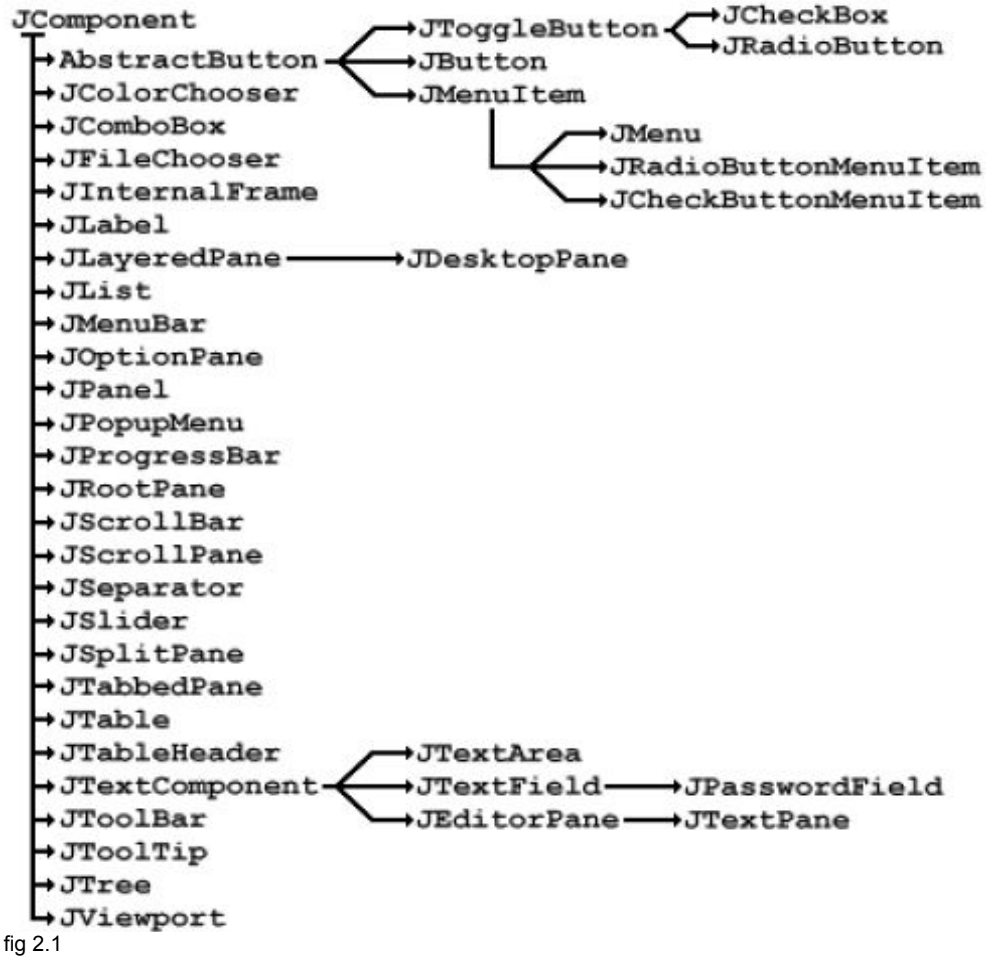


fig 2.1

2.3.1 FUNDAMENTOS

A los componentes Swing se les denomina *ligeros* mientras que a los componentes AWT se les denominados *pesados*. La diferencia entre componentes ligeros y pesados es su *orden*: la noción deprofundidad. Cada componente pesado ocupa su propia capa de orden Z. Todos los componentes ligerosse encuentran dentro de componentes pesados y mantienen su propio esquema de capas definido por Swing. Cuando colocamos un componente pesado dentro de un contenedor que también lo es, se superpondrá por definición a todos los componentes ligeros del contenedor.

Lo que esto significa es que debemos intentar evitar el uso de componentes ligeros y pesados en unmismo contenedor siempre que sea posible. Esto no significa que no podamos mezclar nunca con éxito componentes AWT y Swing, sólo que tenemos que tener cuidado y saber qué situaciones son seguras y cuáles no. Puesto que probablemente no seremos capaces de prescindir completamente del uso de componentes pesados en un breve espacio de tiempo, debemos encontrar formas de que las dos tecnologías trabajen juntas de manera aceptable.La regla más importante a seguir es que no deberíamos colocar componentes pesados dentro de contenedores ligeros, que comúnmente soportan hijos que se superponen. Algunos ejemplos de este tipo de contenedores son JInternalFrame, JScrollPane, JLayeredPane, y JDesktopPane. En segundo lugar, si usamos un menú emergente en un contenedor que posee un componente pesado, tenemos que forzar a dicho menú a ser pesado. Para controlar esto en una instancia específica de JPopupMenu podemos usar su método setLightWeightPopupEnabled().

La característica más notable de los componentes Swing es que están escritos al 100% en Java y no dependen de componentes nativos, como sucede con casi todos los componentes AWT. Esto significa que un botón Swing y un área de texto se verán y funcionarán idénticamente en las plataformas Macintosh, Solaris, Linux y Windows. Este diseño elimina la necesidad de comprobar y depurar las aplicaciones en cada plataforma destino.

2.3.2 CLASES PRINCIPALES

javax.swing	Provee un set "ligero" (de todo el lenguaje java) componentes que , al maximo grado posible, trabajan igual en todas las plataformas.
javax.swing.border	Provee clases e interfaces para dibujar bordes especializados alrededor de componentes swing.
javax.swing.colorchooser	Contiene clases e interfaces usadas por el `JColorChooser` component.
javax.swing.event	Asegura los eventos disparando componentes swing.
javax.swing.filechooser	Contiene clases e interfaces usadas en el componente `JFileChooser` .
javax.swing.plaf	Asegura una interfaz y muchas clases abstractas que Swing usa para proveer la conectividad de “observar y tocar” su capacidad.
javax.swing.plaf.basic	Asegura una interfaz de usuarioen la construccion de objetos de acuerdo con el basico”mirar y sentir”
javax.swing.plaf.metal	Provides user interface objects built according to the Java look and feel (once codenamed *Metal*), which is the default look and feel.
javax.swing.plaf.multi	Provides user interface objects that combine two or more look and feels.
javax.swing.table	Provides classes and interfaces for dealing with `javax.swing.JTable`.

javax.swing.text	Provides classes and interfaces that deal with editable and noneditable text components.
javax.swing.text.html	Provides the class `HTMLEditorKit` and supporting classes for creating HTML text editors.
javax.swing.text.html.parser	Provides the default HTML parser, along with support classes.
javax.swing.text.rtf	Provides a class (`RTFEditorKit`) for creating Rich-Text-Format text editors.
javax.swing.tree	Provee clases e interfaces para tratar con javax.swing.JTree
javax.swing.undo	Permite a los desarrolladores proveer soporte para deshacer o hacer como en textos de los editores

2.3.3 Componentes.

JComponent es una clase abstracta que hereda de Component y de Container, representa todo lo que tiene una posición, un tamaño, puede ser pintado en pantalla y puede recibir eventos. Los componentes en swing tienen nombres muy parecidos a los de awt, con la diferencia de que el nombre inicia con una J , por ejemplo en awt Button, en swing JButton. JComponent hereda de Container lo que permite a los componentes de swing, también ser contenedores lo que posibilita insertar componentes en componentes, por ejemplo insertar un JButton en una JtextArea. Para los ejemplos de comoponentes se usarán los contenedores JFrame y JPanel que funcionan muy similar a Frame y Panel de awt.

JLabel

La etiqueta tiene la misma función que en awt, la forma de instanciar una JLabel es muy similar: JLabel etiqueta = new JLabel(“Etiqueta en swing”)

JButton

La sintaxis para instanciar un boton en swing es la siguiente:

```
JButton boton = new JButton(“Botón”);
```

JButton tiene el mismo comportamiento que los botones de awt.

JCheckBox

En swing también se cuenta con Checkbox llamda JCheckBox, y tiene el mismo compotamiento que en awt.

Ejemplo

```
JCheckBox internet = new JCheckBox(“Internet”);
```

JRadioButton

Este componente de Swing se utiliza cuando se require que le usuario seleccione una solo opción de una lista de opciones. Para establecer que todos los JRadioButton pertenecen a un solo grupo se crea un objeto ButtonGroup que se les envia como parámetro, en los JRadioButton que pertenecen a un mismo ButtonGroup solo se puede seleccionar una sola opcion. De los JRadioButton solo se puede seleccionar una sola opción puesto que pertenecen a un mismo grupo, en este caso llamado grupo.

JScrollPane
Este componente adiciona barras de desplazamiento a los componentes que lo requieran. Para poder insertar barras de desplazamiento a un componente, es necesario que este componente este insertado en un contenedor con un Layout BorderLayout. El siguiente muestra como insertar barras de desplazamiento a una JTextArea.

JTextField
JTextField es muy similar a TextField de awt, una de las diferencias es la forma para encriptar texto, mientras en awt se usaba el métod setEchoChar(char c) en swing se utiliza el componente JPasswordField que hereda de JTextField.
Ejemplo:

```
JPasswordField passUsuario;
passUsuario = new JPasswordField(15);
```

Con este código todo el usuario escriba en ese tipo de JTextField será encriptado por default.

JComboBox
Es un tipo de lista desplegable, puede almacenar una serie de opciones de las cuales solo se puede seleccionar solo una.

Bordes.
Los bordes como su nombre lo dice son un marco generalmente utilizado para delimitar el área de un JPanel.
Ejemplo

```
JPanel panel = JPanel();
```

// Se crea una variable de tipo AbstractBorder y se crea un objeto de tipo TitledBorder()

```
AbstractBorder borde = new TitledBorder("Panel vacío");
```

// Por medio del método setBorder se inserta el borde al componente indicado

```
Panel.setBorder(borde);
```

Cabe destacar que no es aplicablex solo a paneles, puede ser utilizado por todo JComponent

JProgressBar
JProgressBar es un componente que muestra el avance o retroceso de un proceso. Tiene métodos que modifican sus valores para visualizar el avance.
El constructor recibe la orientación, valor inicial, el valor de avance cada vez que se actualicen sus valores, el valor minimo y máximo de la barra , respectivamente. JProgressBar(int orientacion, int valorInicial, int valorAvance, int minimo, int maximo);

JSlider.
Tiene el mismo comportamiento que JProgressBar, con la diferencia que en lugar de ser una avance continuo, utiliza un marca dentro de JSlider para indicar el avance.

JSplitPane.
Es un tipo de panel que se divide en dos , para almacenar de cada lado componentes diferentes, el contructor recibe la orientación, el componente que se ubicará del lado izquierdo y después el componente del lado derecho respectivamente.

```
JSplitPane split = new JSplitPane(int orientacion, Component ci, Componente c2);
```

TabbedPanel
Es un componente que puede agrupar varios paneles divididos por etiquetas. El siguiente código muestra la forma de usar TabbedPane

JTable.
JTable es un componente que se comporta como una Tabla. Hay dos manera de utilizar este componente. La primera es colocando los datos a la tabla por medio del constructor de Jtable

JTree.
JTree es un componente que genera una vista de árbol de los datos que se le inserten.
Al igual de JTable hay dos maneras de crear un árbol de datos. La primera es colocar los datos en el arbol uno a uno.

Layouts.
Como se vió en awt los layouts permiten organizar los elementos de una interfaz gráfica con diseños ya elaborados (FlowLayout, BorderLayout, ...) swing puede utilizar todos estos layouts, y suma un layout más llamado BoxLayout.

BoxLayout.
Este Layout permite ordenar componentes en forma vertical u horizontal, para definir que orden llevará se utilizan las variables estáticas X_AXIS y Y_AXIS. El siguiente ejemplo muestra como establecer el orden de un panel en BoxLayout en forma horizontal

```
panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXI S);
```

El panel utiliza el método setLayout y envía un nuevo objeto de BoxLayout que recibe como parámetros el mismo panel y una de las variables estáticas X_AXIS y Y_AXIS.

Contenedores.

Swing utiliza frames similares de awt como son: JFrame, JPanel y JDialog, y adiciona JDialgo prefabricados, JDeskTopPane, JInternalFrame.

JPanel
JPanel se comporta de la misma forma que Panel de awt, sirve para agrupar componentes, y utiliza todos los layouts de awt además de BoxLayout .

JFrame
JFrame utiliza el método getContentPane() que devuelve un objeto Contaniner, al cual se le puede adicionar componentes y establecer layout.
Ejemplo

```
frame.getContentPane().add(boton);
frame.getContentPane().setLayout(new FlowLayout);
```

JDialog prefabricados
En swing existen diálogos ya fabricados para las siguientes funciones:
Elegir un color de panel de colores JColorChooser
Elegir un archivo JFileChooser
Presentar un mensaje JoptionPane
Ejemplos:

```
colorChooser = new JColorChooser();
fileChooser = new JFileChooser(): // inicia el mostrado de carpetas de el directorio raiz
optionPane = new JOptionPane(); // recibe como parámetro el mensaje a visualizar
```

JDeskTopPane

Es un frame que permite que dentro de el existan de manera independiente otros frames, dando el efecto como cuando en JCreator abrimos varias ventanas de archivos java.
Los frames que se pueden insertar en JDeskTopPane son los JInternalFrame. La siguientes instrucciones ilustran la forma en que trabaja el JDeskTopPane.

```
JFrame frame = new JFrame();
JDeskTopPane desktop = new JDeskTopPane();
JInternalFrame internalF1 = new JInternalFrame();
JInternalFrame internalF2 = new JInternalFrame();
desktop.add(internalF1);
desktop.add(internalF2);
frame.add(desktop);
```

JInternalFrame

Es el unico frame que se puede insertar en un JDeskTopPane, el constructor más común para crear un objeto de tipo JInternalFrame es:

```
JInternalFrame internalF = new JinternalFrame("nombre del frame",true,true,true,true);
```

Los cuatro parametros boleanos definien el comportamiento de la ventana, es decir, si tendran el comportamiento de tamaño modificable, cerrar, maximizar y minimizar respectivamente.
Este frame a su vez puede agrupar cualquier otro componente.

LookAndFeel

Swing utiliza la clase abstracta BasicLookAndFeel para personalizar el aspecto de los componentes. La manera más simple es utilizando un LookAndFeel ya elaborado, lo único que hay que hacer es actualizar la clase UIManager (clase manejadora de la apariencia de los componentes) con el LookAndFeel deseado,

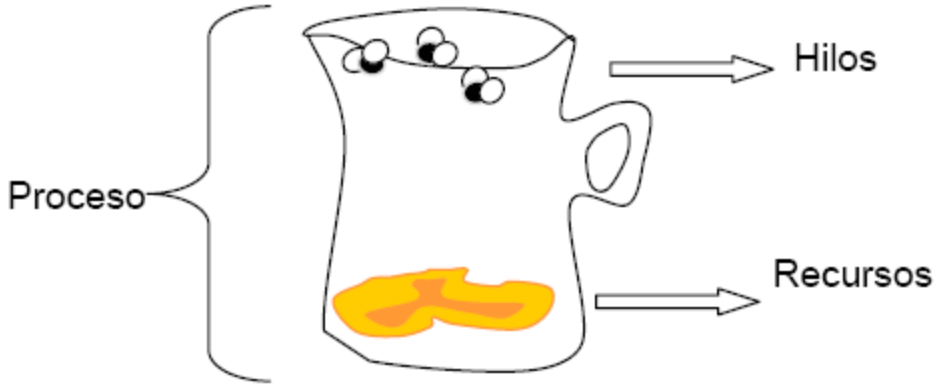
2.4 HILOS

La noción de proceso existente en los sistemas operativos como UNIX en los 80's, resultaba poco adecuada para los sistemas distribuidos. El problema principal radicaba en que esta noción asociaba solo una actividad de procesamiento con cada proceso. La solución alcanzada fue generalizar la noción de proceso para englobar varias actividades simultáneas. Es decir que un proceso consiste de un ambiente de ejecución de uno o más hilos.
Un hilo es la abstracción del sistema operativo sobre una actividad. El término deriva de la frase "un hilo de ejecución". Por otro lado, un ambiente de ejecución es una unidad de administración de recursos; una colección de recursos de los cuales, los hilos tienen acceso. Un ambiente de ejecución está formado de:

- Un espacio de direcciones
- Recursos de sincronización
- Comunicación entre hilos.

Este forma de visualizar los hilos es similar a una situación en la que tenemos una jarra donde hay alimento y aire (recursos), donde existen moscas que se alimentan del contenido de la jarra. En este símil, cada mosca es análoga a un hilo y los recursos, al ambiente de ejecución. Como las moscas, los hilos pueden:

- Trabajar simultáneamente, no pueden salirse de su espacio (jarra);
- Ponerse de acuerdo para utilizar sus recursos (alimentarse) en cierto orden.
- Generar nuevos hilos.



La aplicación del concepto de hilo es muy útil, porque como cada hilo comparte con los distintos hilos su área de datos, la comunicación es mucho más sencilla. Así por ejemplo, en una aplicación Web podríamos tener un hilo que vaya cargando imágenes y por otro lado un hilo que atienda los eventos de la interfaz, o quizás podemos tener un servidor que por cada cliente que se conecta a él, genere un hilo distinto que atienda a cada conexión.

2.4.1 FUNDAMENTOS

Hilos en java.

La ejecución de hilos por medio de Java, es implementada de distintas formas, dependiendo de la plataforma. Existen plataformas que simulan el comportamiento de hilos y otras que realmente implementan el concepto. De cualquier forma, la programación es más sencilla teniendo hilos que no teniéndolos.

Definiendo contexto del hilo.

Para poder implementar hilos en Java, requerimos asociar un contexto o ambiente de ejecución a los hilos que vayan a ejecutarse. El contexto se define por medio de una clase. Además de controlar los datos del hilo en ejecución, la clase debe implementar la operación *run*, que definirá la actividad del hilo. La manera en la cual forzamos a la clase a definir *run*, es forzándola a implementar la interfaz *Runnable*, como se puede apreciar en el ejemplo:

```
public class Hilo implements Runnable
{
private int cuenta=0;
public void run()
{
…// El código de ejecución del hilo que se mostrará posteriormente
```

Creando y arrancando hilos.

Debido a que cada hilo compartirá un contexto común, es necesario que una clase administre los hilos. La administración de hilos tiene que ver con la creación de nuevos hilos, su arranque, y su detención parcial y total.

1.- Creamos diferentes clases que se comportarán como hilos al implementar la interfaz *Runnable*
2.- Creamos la clase *AdminitradoraHilos* encargada de administrar los hilos
2.1.- En la clase *AdminitradoraHilos* utilizamos la clase Thread para arrancar los hilos

1-

```
class HiloUno implements Runnable {
public void run() {
for (int i=0; i<5; i++) {
System.out.println("Hilo Uno");
try {
// sleep(int) es un método estático de la clase Thread que permite hacer una pausa en un proceso
Thread.sleep(600);
}
catch(InterruptedException ex) {}
}
}
}
// ----------------------------------------------------------------------
class HiloDos implements Runnable {
public void run() {
for(int I=0; I<5; I++) {
System.out.println("Hilo Dos");
try {
Thread.sleep(300);
}
catch(InterruptedException ex) {}
}
}
}
```

Las clases hilo que implementan la interfaz *Runnable* son forzadas a definir el método *run( )* , las intrucciones en este métodos, seran ejecutadas cuando se arranque el hilo.

2.-

```
public class AdministradoraHilos {
Thread controlaHiloUno, controlaHiloDos;
HiloUno hiloUno = new HiloUno();
HiloDos hiloDos = new HiloDos();
controlaHiloUno = new Thread(hiloUno);
controlaHiloDos = new Thread(hiloDos);
public void arrancarHilos() {
controlaHiloUno.start();
controlaHiloDos.start();
}
public void static void main(String argv[]) {
AdministradoraHilos admin = new AdministradoraHilos( );
admin.arrancarHilos( );
}
}
```

Inicialmente en el código, declaramos referencias a las clases HiloUno e HiloDos y otras dos referencias a la clase *Thread*. Inicializamos las referencias de Thread enviando como parámetro los hilos. En el método arranzarHilos() arrancamos los hilos usando las referencias de Thread

Corriendo y deteniendo un hilo.

Analicemos el código correspondiente al *run* de la clase *Hilo*

```
public void run() {
while (vivo == true) {
while(despertar == true) {
cuenta++;
try {
Thread.sleep(1000);
} catch(InterruptedException e) {}
```

```
}
}
}
```

El hilo incrementa la cuenta cada vez que es despertado (estableciendo la varible despertar = true) . Cuando la variable despertar es igual a false, la instrucción se detiene pero la aplicación sigue viva, en este lapso de tiempo el hilo esta dormido y mientras eso sucede otros hilos pueden estar corriendo. Eventualmente el trabajo ha de quererse detener, y se querrán detener los procesos y será necesario entonces, programar la operación *matar()*:

```
public void matar() {
vivo = false;
}
hilo.matar
```

2.5 Flujos

El concepto de flujo puede resumirse como una ruta de acceso de comunicación entre el origen de cierta información y su destino final.
La ventaja del uso de los flujos, sin lugar a dudas radica, en el hecho de no tener que saber acerca del origen o destino de la información que manejamos.
El origen y destino son de hecho, productores y consumidores de Bytes. Para dominar el concepto de flujos, sería prudente introducirnos en el manejo de ficheros en Java.

2.5.1 FUNDAMENTOS

Java dispone de una serie de librerías (paquetes) estándar que ofrecen clases para las necesidades más comunes. En este artículo abordaremos el soporte de Java para operaciones de entrada y salida, así como para la gestión del sistema de archivos, que se encuentra en el paquete java.io.
Cuando pensamos en las operaciones de entrada y salida, se suele pensar en dos tipos de operaciones: operaciones de lectura/escritura sobre archivos, y operaciones de introducción de datos mediante el teclado. Si bien esto resulta muy común, a la hora de la verdad, una aplicación puede obtener datos de muchas otras formas: a través de una conexión vía Internet con un servidor remoto, a través de una conexión DDE con otro programa en la misma máquina, o incluso a través del portapapeles de Windows. Se puede ver que, a pesar de la distinta procedencia de la información en todos estos casos, su manejo será bastante similar: solicitamos al sistema que nos conecte a la fuente o destino de la información (abrimos archivo, nos conectamos al servidor de red, etc.), obtenemos la información, que será una serie de datos secuenciales, y nos desconectamos de la fuente de datos, para liberar recursos del sistema. Java utiliza el concepto de flujo (stream) para trabajar con información manejada secuencialmente.No siempre se puede o es conveniente trabajar secuencialmente: hay ocasiones en que deseamos tener acceso aleatorio, en lugar de leerla en serie hasta que llegamos a la posición donde está la información que deseamos. Java también proporciona soporte para acceso aleatorio a archivos, a través de la clase RandomAccessFile. El acceso al sistema de archivos del sistema también es fundamental: es necesario poder renombrar archivos, obtener la lista de archivos de un directorio, saber si un archivo es de escritura o lectura, etc. Java proporciona la clase File para manejar el sistema de archivos, independientemente de las clases basadas en flujos y de acceso aleatorio que utiliza para leer/escribir entre ellos.
La Tabla A muestra las clases de entrada y salida que se pueden encontrar en java.io.

InputStream
ByteArrayInputStream
FileInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
FilterInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
OutputStream
ByteArrayOutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
File
FileDescriptor
RandomAccessFile
StreamTokenizer

Tabla A: Clases en el paquete java.io

Por último, Java proporciona una clase de excepción para cada tipo de error común en las operaciones de entrada y salida: cada una de estas clases deriva de la clase base IOException.
El concepto de flujo es muy potente, dado que proporciona un modo de tratar las operaciones de entrada/salida de forma similar para distintas fuentes de datos y canales de comunicación. Podemos definir un flujo como una secuencia de bytes que viajan desde una fuente a un destino a través de un camino, de modo secuencial.

Java proporciona un conjunto de clases para leer información desde un flujo, y otro para escribir en él. Las dos clases fundamentales son InputStream y OutputStream, para lectura y escritura respectivamente, que proporcionan métodos para realizar las operaciones básicas: en función de las distintas fuentes o modos de manejar la información se tendrán distintas clases derivadas de éstas, siempre respetando el protocolo básico dictado por ellas.
El esquema básico de trabajo con los flujos es siempre el mismo: se abren, lo que se consigue con las operaciones new InputStreamFile(...), etc., se realizan las operaciones deseadas de escritura y lectura con read, write, etc., y luego se cierran, con close().

2.5.2 CLASES PRINCIPALES

La clase principal de entrada y salida de flujo de datos dada por sun es :

java.io	Provee entrada y salida atravez de flujo de datos, serializacion y archivos del sistema.

A continuacion veremos ficheros y las clases o subclases mas utilizadas de java.io.

Ficheros

Para crear un fichero o archivo, utilizamos la clase File, esta clase proporciona muchos métodos interesantes para el manejo y la obtención de información básica de nuestros ficheros. Los primeros que vamos a revisar son sus tres constructores.

```
public File(String path);
```

Este constructor recibe el path completo, donde se ubica físicamente nuestro fichero

```
public File(String path, String name);
```

El constructor recibe el path del directorio donde se ubica el archivo, más el nombre del archivo

```
public File(File dir, String name);
```

Este constructor recibe un objeto de tipo File que se comporta como un directorio y un objeto de tipo String que se comporta como el nombre del archivo, resulta ser muy útil en el caso en que ya sea el Directorio o el Fichero sean datos variables.
En las siguientes líneas describiremos algunos de los métodos más importantes de la clase File.

• getName

```
public String getName();
```

Devuelve el nombre del archivo.

• getPath

```
public String getPath();
```

Devuelve el path, donde se encuentra el archivo.

• exist

```
public boolean exist();
```

Nos regresa un valor verdadero si el archivo especificado por el objeto existe.

• canWrite

```
public boolean canWrite(); public boolean canRead()
```

El primero de estos métodos nos indica, si es que se puede escribir en ese archivo, mientras que el segundo nos indica si es que podemos leer de él.

• IsDirectory

```
public boolean isFile(); public boolean isDirectory();
```

El método isFile, nos regresa un valor verdadero si se trata de un archivo, y el método isDirectory nos devuelve un valor verdadero si se trata de un directorio.

• length

```
public long length();
```

Este método nos regresa un entero long con la longitud del archivo

• mkdir

```
public boolean mkdir();
```

Hace de nuestro objeto File un directorio.

• list

```
public String[] list();
```

En el caso en que nuestro objeto File sea un directorio este método nos regresa la lista de archivos que lo componen.

Flujos de Entrada y Salida estándar

Estos flujos no son otra cosa, más que el teclado(entrada) y el monitor(salida)

Las clases InputStream y Reader

InputStream.-Esta clase define las formas de que un consumidor o destino lee un flujo de bytes de algún origen.

Reader.-Es una clase que define las formas en las que un destino o consumidor lee un flujo de caracteres de un origen. Esta clase y sus subclases en su mayoría son análogas a las subclases de InputStream, solamente que éstas en lugar de trabajar con Bytes trabajan con caracteres.Ambas clases son abstractas y comparten entre sí, algunos métodos, los cuales se describen a continuación.

• read

```
public int read(byte b[]);
public int read(char cbuf[]);
```

Este es quizá uno de los métodos más importantes de ambas clases, y es aquel que les permite leer bytes o caracteres, de un cierto origen. Existe otra variante de este método la cual puede leer dentro del buffer especificando la posición inicial y la final, de nuestra lectura.

public int read(byte b[], int off, int len); ó public int read(char cbuf[], int off, int len);

• mark y reset

```
public void mark(int readAheadLimit) throws IOException;
public void reset() throws IOException;
public boolean markSupported();
```

El método mark, sirve para fijar una posición en nuestro flujo, para luego retomarla con el método reset, el cuál nos devuelve a esa posición. El método markSupported nos permite saber si el flujo que estamos leyendo maneja o no el concepto de marcas.

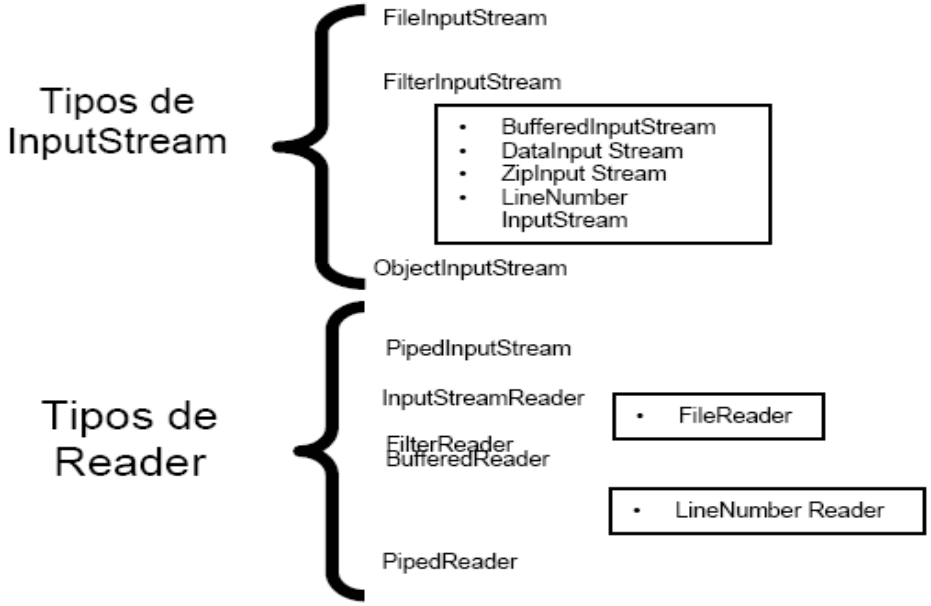
• close

```
public abstract void close(int readAheadLimit) throws IOException;
```

Como su nombre lo dice, cierra de manera explícita, nuestro flujo.

Tanto InputStream, como Reader heredan sus características, a varias clases afines, a

continuación describiremos de manera general el uso y aplicación de estas clases, pero antes sería prudente ver los siguientes esquemas:



El siguiente ejemplo nos demuestra el uso de un objeto Reader, para captura desde un flujo de entrada estándar(teclado).

```
import java.io.*;
class Lector
{
public static void main(String args[]) throws IOException
{
int caracter;
// creamos un objeto de tipo InputStreamReader,que leerá caracteres desde el teclado
InputStreamReader unReader=new InputStreamReader(System.in);
// Se usa el método read para efectuar la lectura
while((caracter=unReader.read())!=-1)
System.out.print((char)caracter);
unReader.close();
}
}
```

FileInputStream y FileReader

Una de las aplicaciones más importantes de los flujos, es la de relacionarlos con archivos. Estas clases cumplen ese cometido, mediante tres constructores y un método, comunes.

```
public FileInputStream(String name);
//Recibe el Path en el que se localiza el archivo.
public FileInputStream(File file);
```

Este recibe un objeto de tipo File.

```
public FileInputStream(FileDescriptor fdObj);
```

Recibe el descriptor del archivo, desde el cual se va a leer.

• getFD

```
public FileDescriptor getFD() throws IOException;
```

El método getFD, regresa el descriptor del archivo en el que esta basado el flujo.

A continuación se presenta el programa LeerArchivo.java, el cual tiene en su interior dos objetos uno de tipo FileInputStream y otro FileReader ambos leen el archivo

LeerArchivo.java; para después desplegar todo el contenido del archivo, o sea el código de nuestro programa.

```
import java.io.*;
class LeeArchivo {
public static void main(String args[]) throws IOException {
FileReader miArchivo;
FileInputStream unFlujo;
byte []buffer=new byte[100];
try
{
// Creamos los objetos FilerReader y FileInputStream, mandando a su respectivo constructor el nombre
físico del //archivo
miArchivo=new FileReader("LeerArchivo.java");
unFlujo=new FileInputStream("LeerArchivo.java");
}
catch(FileNotFoundException e)
{
System.err.println("No se pudo abrir");
return;
}
int caracter;
//Leemos el archivo usando el objeto de tipo FileReader
while((caracter=miArchivo.read())!=-1)
System.out.print((char)caracter);
miArchivo.close();
byte unDato;
//Leemos el archivo usando el objeto de tipo FileInputStream
while((unDato=(byte)unFlujo.read())!=-1)
System.out.print((char)unDato);
unFlujo.close();
}}
```

FilterInputStream y FilterReader

Estas clases abstractas contienen otro flujo, esto es sumamente útil, cuando deseamos “apilar Streams”. Estas clases no poseen ningún método diferente a los de InputStream y Reader, solamente tienen un constructor distinto el cual recibe un objeto InputStream o un Reader dependiendo el caso.

Al apilar, nos referimos a la capacidad que tienen las Streams de poder contener otra Stream. Esto aplica en el caso de necesitar métodos de clases diferentes. Podemos observar la siguiente línea de código, cuya función es declarar un objeto de tipo BufferedInputStream pero que también posea las características de un objeto de tipo FileInputStream.

```
BufferedInputStream newStream = new BufferedInputStream (new FileInputStream( miArchivoPrueba ));
```

¿Pero que sucede con los métodos de estas clases apiladas? Pues el método de la primera capa se encarga de llamar al método de la segunda, esto es, si nosotros decimos:

```
miStream.read();
```

Internamente llamará al método *read()* de la clase BufferedInputStream con su respectivo tratamiento, y éste a su vez llamará al método *read()* de la clase FileInputStream el cual leerá el objeto de tipo File llamado *miArchivoPrueba.* En conclusión, el objeto miStream, puede ser tratado como un FileinputStream, un BufferedInputStream o como ambos.

BufferedInputStream y BufferedReader

Estas clases facilitan mucho la lectura de flujos, pues dentro de ellas encontramos un arreglo

(char's o byte's), a manera de buffer, el cual actúa como una cache, para lecturas a futuro. Las clases BufferedInputStream y BufferedReader, son las que por tener un buffer, implementan mejor los métodos mark() y reset(). En ambos casos podemos recurrir a un apilamiento de Streams, con la finalidad de aprovechar las características de estas clases.

DataInputStream

Los métodos que usa DataInputStream están ubicados en una interfaz por separado llamada DataInput. Estos métodos son realmente útiles, puesto que no todas las veces, resulta idóneo leer directamente Bytes; así que utilizando esta interfaz y sus respectivos métodos, podemos forzar los tipos de los Bytes que leemos. Los métodos trabajan justo igual al read de InputStream, con la diferencia, de que estos no nos regresan Bytes.

```
boolean readBoolean() throws IOException;
byte readByte() throws IOException;
int readUnsignedByte() throws IOException;
short readShort() throws IOException;
int readUnsignedShort() throws IOException;
char readChar() throws IOException;
int readInt() throws IOException;
long readLong() throws IOException;
float readFloat() throws IOException;
double readDouble() throws IOException;
String readLine() throws IOException; /*Este se usa para leer cadenas en formato ASCII*/
String readUTF() throws IOException; /*Este se usa para leer cadenas en formato Unicode*/
```

Esta clase además tiene una excepción EOFException, la cual nos indica el fin del flujo.
Al ser hija de FilterInputStream, esta clase puede apilarse con otras, para así poder combinar sus ventajas con las de otras clases, como la BufferedInputStream.
El ejemplo que sigue captura caracteres mediante el uso de los métodos de la interfaz Datainput.

```
import java.io.*;
class Datos
{
public static void main(String args[]) throws IOException
{
char unDato;
DataInputStream unFlujo=new DataInputStream(System.in);
//aquí se invoca al método read para caracteres
while((unDato=unFlujo.readChar())!=-1)
{
System.out.print(unDato);
}
unFlujo.close();
}
}
```

```
/*El siguiente código realiza la misma función que el anterior, con la única diferencia, que
este lee enteros(int).*/
import java.io.*;
class DatosI
{
public static void main(String args[]) throws IOException
{
int unDato;
DataInputStream unFlujo=new DataInputStream(System.in);
//aquí se invoca al método read para enteros
while((unDato=unFlujo.readInt())!=-1)
{
```

```
System.out.println(unDato);
}
unFlujo.close();
}
}
```

Las Clases OutputStream y Writer

OutputStream.-Esta clase define las formas fundamentales en que un productor u origen escriba un flujo de bytes en algún destino.

Writer.- Es una clase que define las formas fundamentales en las que un origen o productor escribe un flujo de caracteres desde un origen. Esta clase y sus subclases en su mayoría son análogas a las Subclases de OutputStream, la diferencia radica en que estas trabajan con caracteres en lugar de Bytes.

Estas clases tienen algunos métodos comunes.

• write

```
public void write(byte b[]) throws IOException;
public void write(char cbuf[]) throws IOException;
```

Este método es de los más importantes de ambas clases, puesto que les permite escribir bytes o caracteres, hacia un cierto destino.

Existe otra variante de este método la cual puede escribir un pedazo de su buffer especificando la longitud deseada.

```
public abstract void write(byte b[],int off, int len); ó public abstract void write(char cbuf[], int off, int len);
```

• flush

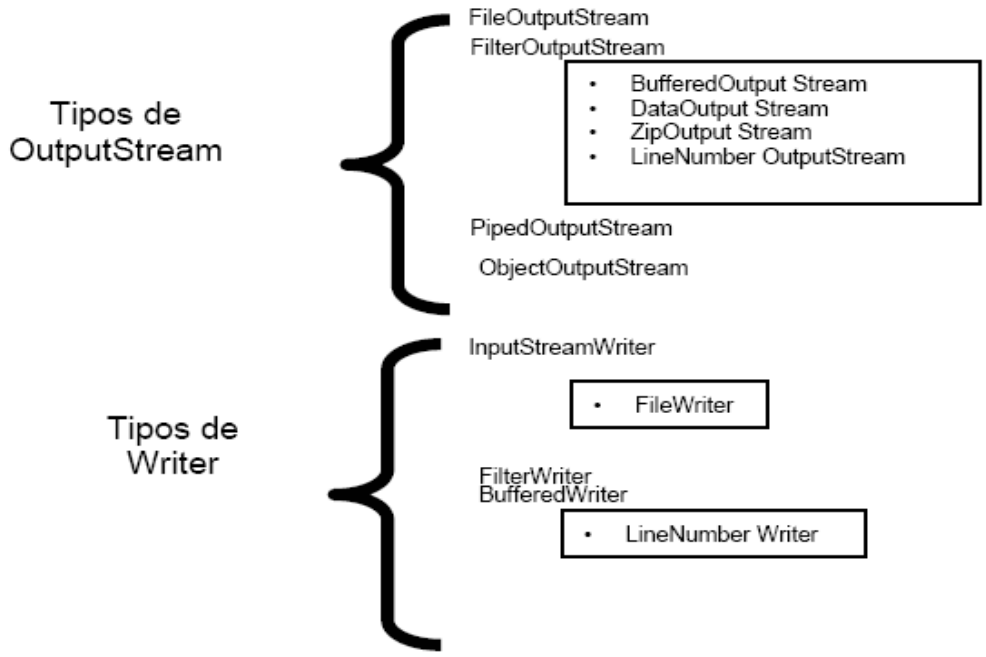
```
public abstract void flush() throws IOException;
```

El flush, sirve para desalojar la salida a través de alguna caché, con buffer.

• close

```
public abstract void close() throws IOException;
```

Como su nombre lo dice, cierra de manera explícita, el flujo.



FileOutputStream y FileWriter

La función principal de estas clases, es la de relacionar un archivo con un cierto flujo de salida, a continuación se describirán sus constructores.

```
public FileOutputStream(String name);
```

Recibe el Path en el que se localiza el archivo en el cual vamos a escribir.

```
public FileOutputStream(File file);
```

Este recibe un objeto de tipo File, el cual es el archivo en el que vamos a escribir.

```
public FileOutputStream(FileDescriptor fdObj);
```

Recibe el descriptor del archivo, en el cual se va a escribir.

Estas clases también tienen, el método getFD, ya descrito anteriormente. Otra cuestión de gran importancia, es el hecho de que tanto las clases FileOutputStream, FileWriter y las clases FileInputStream y FileReader; al tener que interactuar con los archivos del sistema, muchas veces llegan a causar violaciones a la seguridad establecida por los usuarios.

FilterOutputStream y FilterWriter

Análogamente a las clases “FilterInputStream” y “FilterReader”, estas clases nos sirven para apilar Streams, así como aprovechar las ventajas de usar diferentes tipos de Streams de salida a la vez.

BufferedOutputStream y BufferedWriter

Estas clases al igual que sus similares para lectura, son muy importantes ya que facilitan mucho la escritura de flujos, puesto que poseen un buffer, con el que podemos tener una cache, para escrituras. Las clases BufferedOutputStream y BufferedWriter, implementan de una manera particular el método flush, el cual empuja los bytes(caracteres) hacia afuera. Recuerde que puede combinar esta Clase con otras mediante el apilamiento de Streams.

DataOutputStream

Los métodos que usa DataOutputStream, están ubicados en una interfaz por separado llamada DataOutput. Estos métodos proporcionan un enfoque directo para escribir, sin tener que recurrir a los Bytes.

```
void writeBoolean(boolean b) throws IOException;
void writeByte(int i) throws IOException;
void writeShort(int i) throws IOException;
void writeChar(int i) throws IOException;
void writeInt(int i) throws IOException;
void voidLong(long l) throws IOException;
void writeFloat(float f) throws IOException;
void writeDouble(double d) throws IOException;
void writeBytes(String s) throws IOException; /*Este se usa para escribir bytes de 8 bits */
void writeChars(String s) throws IOException; /*Este se usa para escribir en Unicode de 16 bits */
void writeUTF(String s) throws IOException; /*Este se usa para escribir en flujo especial Unicode(UTF-
8)*/
```

Al ser hija de FilterOutputStream, esta clase puede apilarse con otras, para así poder combinar sus ventajas con las de otras clases, como la BufferedOutputStream.

PrintWriter

Esta clase representa a un flujo de salida y su uso es muy similar al de DataOutputStream. Los dos métodos principales de esta clase son el *print* y el *println*(imprime y da un salto de línea). Mediante estos métodos podemos dar salida con formato a los tipos primitivos y cadenas de Java. Las posibilidades que ofrece PrintWriter se multiplican, ya que nos permite utilizar el ”apilamiento de Streams”. El programa que se muestra a continuación, copia el contenido de un archivo de entrada(Frutas.txt) a uno de salida(Escrito.out); para lograr,
escribir sobre el archivo usa un objeto de tipo PrintWriter, sobre el cual se apilan un BufferedWriter y un FileWriter, en el cual le indicamos que la salida será el archivo Escrito.out.

```
import java.io.*;
public class Escritores {
public static void main(String[] args) {
try {
// Se crea el objeto in en un try diferente con la finalidad de que si el archivo no se encuentra nos mande
una //Excepción
BufferedReader in =
new BufferedReader(
new FileReader("Frutas.txt"));
try {
// Se declara un objeto de tipo String , que almacenará cada línea del archivo
String s = new String();
//Se crea el objeto PrintWriter, apilando un BufferedWriter y un FileWriter que se lleva como parámetro el
nombre //fisico del archivo de salida
PrintWriter out1 =
new PrintWriter(
new BufferedWriter(
new FileWriter("Escrito.out")));
// Se declara un entero que será el contador de líneas
int lineCount = 1;
// Lee el archivo de entrada
while((s = in.readLine()) != null )
//Escribe en el archivo de salida con el formato que le enviamos al método println
out1.println(lineCount++ + ": " + s);
out1.close();
} catch(EOFException e) {
System.out.println("End of stream");
}
} catch(FileNotFoundException e) {
System.out.println(
"File Not Found:" + args[0]);
} catch(IOException e) {
System.out.println("I/O Exception");
}
}
}
```

PipedInputStream-PipedOutputstream vs PipedReader-PipedWriter

Estas parejas de clases para funcionar necesitan forzosamente estar juntas, ya que trabajan de manera simultanea. Su función principal es la de lograr implementar hilos, en donde existan tanto hilos productores, como hilos consumidores de Bytes. La pimera pareja de clases se utilizará cuando se necesite entubar bytes(PipedInputStream y PipedOutputStream), y la segunda(Pipedreader y PipedWriter) cuando se requiera desentubar caracteres. El funcionamiento del entubamiento (pipe) de dos flujos de caracteres es similar a la escena en donde existen dos grupos de gente en torno de un tazón de fruta, en donde de un lado se encuentren cinco personas llenándolo de fruta y del otro lado diez personas comiendo fruta de ese tazón, es obvio que no llevan la misma proporción y que lo hacen todos al mismo tiempo; precisamente en eso se aplican las clases “Piped”. No olvidando que por cada PipedReader o PipedInputStream tiene que haber un PipedWriter o un PipedOutputStream respectivamente.
El siguiente ejemplo, nos muestra como aplicar el PipedReader y el PipedWriter, mediante la utilización de hilos.

```
import java.io.*;
class Consumidor implements Runnable
{
// Se declara un objeto de tipo PipedReader
private PipedReader unLector=new PipedReader();
public Consumidor(Productor unProductor) throws IOException
{
// Se da la conexión entre el Productor y el Consumidor
```

```
unProductor.escritor().connect(unLector);
}
public void run()
{
int valor;
try{
// El lector comienza a leer
while((valor=unLector.read())!=-1)
{
System.out.print((char)valor);
}
unLector.close();
}
catch(IOException e)
{
System.out.println(e.getMessage());
}
}
}
// La clase Reloj nos permitirá hacer queel programa espere 5 segundos, cuando capturemos desde el
teclado
class Reloj implements Runnable
{
boolean esFin=false;
public void terminar()
{
esFin=true;
}
public void run()
{
int cuenta=0;
while(true)
{
cuenta++;
System.out.println("T:"+cuenta);
if(esFin)
return;
try{
Thread.sleep(5000);
}
catch(InterruptedException e)
{
System.out.println(e.getMessage());
}
}
}
}
class Productor implements Runnable
{
// Se declara un objeto de tipo PipedWriter
PipedWriter unEscritor=new PipedWriter();
InputStreamReader fuente;
public Productor(InputStreamReader fuente)
{
this.fuente=fuente;
}
public PipedWriter escritor()
{
return unEscritor;
}
public void run()
```

```
{
try
{
int valor;
//
while((valor=fuente.read())!=-1)
{
unEscritor.write(valor);
Thread.yield();
}
unEscritor.close();
}
catch(IOException e)
{
System.out.println(e.getMessage());
}
}
```

```
class PruebaPiped
{
public static void main(String args[]) throws IOException
{
int valor;
int valorReader;
Reloj unReloj;
// Se crea el Productor y Se crea el consumidor.
Productor unProductor=new Productor(new InputStreamReader(System.in));
Consumidor unConsumidor=new Consumidor(unProductor);
// Se crean los hilos respectivos, para cada proceso. Una vez creados, se arrancan con el método start
Thread hiloProductor=new Thread(unProductor);
hiloProductor.start();
Thread unHilo=new Thread(unConsumidor);
unHilo.start();
Thread otroHilo=new Thread(unReloj=new Reloj());
otroHilo.start();
//
try
{
//Una vez terminado cada hilo, se avanzará ala siguiente instrucción
hiloProductor.join();
unHilo.join();
unReloj.terminar();
otroHilo.join();
}
catch(InterruptedException e)
{
System.out.println(e.getMessage());
}
System.out.println("Adios");
}
}
```

3 PROGRAMACION CON JAVA J2EE

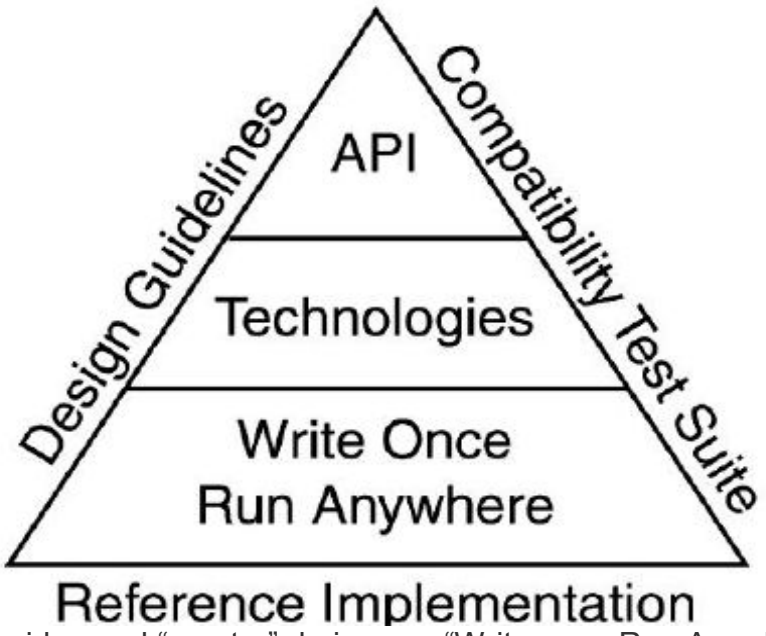
3.1 INTRODUCCION A LA TECNOLOGIA J2EE

La tecnologia J2EE define una arquitectura para desarrollo complejo, distribuido en aplicaciones java empresariales.

J2EE fue originalmente anunciado por Sun Microsystems a mediados1999 y fue oficialmente mostrado a finales del 1999. El J2EE, venia siendo relativamente nuevo, pero todavaia tenian que pasar muchos significantes cambios de lanzamiento en lanzamiento, especialmente en el area de desarrollo de los javabeans empresariales (EJB).

El J2EE consiste en lo siguiente:

- Diseño de guias para el desarrollo de aplicaciones empresariales usando J2EE
- La referencia de una implementacion que provee una operacion de vista de J2EE
- Compatibilidad para herramientasa de test para usar por terceros para verificar sus productos en acatamiento con J2EE
- Varias aplicaciones para programar interfaces (APIs) para permitir un generico acceso a recursos empresariales y de infraestructura
- Tecnologia para simplificar desarrollos empresariales en java



La plataforma construida en el “mantra” de java es “Write once Run Anywhere”(se escribe una vez y corre donde sea) via el grupo de tecnologias y el set de APIs. Estas son, por turno, sostenidas y atadas por tres llaves elementales, nombradas referencia de implementacion, el diseño de pautas, y la compatibilidad de la plataforma.

- *Java 2 Plataforma, Micro Edition (J2ME):* Plataforma para el desarrollo de software para tecnologias arraigadas en los dispositivos como telefonos, palms, etc.

- *Java 2 Plataforma, Standard Edition (J2SE):* El mas familiar en la plataforma de java *2* Esto es asi como el Java Development Kit (JDK) asi como la capacidad de desarrollar applets, JavaBeans, y aplicaciones pór el estilo.

- *Java 2 Platform, Enterprise Edition (J2EE):* Plataforma para el desarrollo empresarial en escala a grandes aplicaciones. Esto esta diseñado para ser usado en conjuncion con el J2SE.

Los componentes base del software J2EE tiene numerosas ventajas sobre el software tradicional, entre estas estan:
- *Productividad alta*
- *Rapido desarrollot*
- *Alta calidad*
- *Facil mantenimiento :*

for further reading.

Tecnologia

Para entender la tecnologia J2EE primero debemos conocer el rol del contenedor de la arquitectura J2EE. Todas las tecnologias dentro de J2EE radican dentro de obtener este poderoso concepto

Un contenedor es una entidad de software dentro del servidor y es reponsable por el manejo especifico en el tipo de componentes. Esto provee la ejecucion de los componentes de tu desarrollo dentro del ambiente J2EE

Atravez de los contendedores de la arquitectura J2ee se provee independencia entre desarrlollo y despliegue esto provee portabilidad dentro de los servidores.El contenedor tambien es responsable por el manejo del ciclo de vida de los componentes desplegados dentro y por las cosas que son el recurso para hacer cumplir la seguridad. Por ejemplo, tu puedes restringir la habilidad de acceder aun metodo especifico para un pequeño gruypo de usuarios. El contenedor deberia cumplir esta restriccion interceptando las peticiones para ese metodo y asegurarse que la entidad de respuesta esta dentro de la peticion de acceso en una lista privilegiada.

Dependiendo del tipo de contenedor, esto deberia tambien proveer acceso a algunas o todas los ApIs J2EE. Todos los componentes J2EE son desarrollados y ejecutados dentro de un tipo de contenedor por una instancia, EJBs corren dentro del contenedor EJB, y los servlets corren dentro de un contenedor web, dentro de el J2EE se tiene cuatro diferentes tipos de contenedores :

- *Application container:Hospeda* aplicaciones java stand-alone
- *Applet container:* Provee el ambiente de ejecucion para los applets
- *Web container:Hospeda los componentes de web*,como servlets and JavaServer Pages (JSP)
- *Enterprise container:* Hospeda componentes EJB

Estas son aplicaciones Java Enterprise Edition el termino empresarial(enterprise) se refiere a la organizacion de individuos o entidades, que presuntamente trabajan juntas para llegar a las mismas metas que tienen las siguientes caracterizticas

- Es una tecnología multinivel distribuidos.
- Los sistemas se contruyen mediante la inclusión de componentes modulares.
- Consta de diferentes API’s de desarrollo.

Logica de negocio + API: = Aplicación empresarial

- **para autenticación de usuarios**
- **Soporte Multiusuarios**
- **Persistencia de información**
- **Integridad de datos**
- **Comunicaciones a nivel de cliente**
- **Comunicación con otras aplicaciones E.**
- **Otras**

Las API con que cuenta j2ee son:

- Jdbc
- Servlets
- JavaServer Pages (JSP)
- Jaxp
- Enterprise JavaBeans (EJB)
- Session Beans
- Entity Beans
- Message-Driven Beans

entre otros.En el siguiente capitulo veremos el desarrollo de Servelets, JSp's, Ejb's y bean's de mensaje.

3.2 JDBC

JDBC es el acrónimo de *Java Database Connectivity*, un API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java independientemente del sistema de operación donde se ejecute o de la base de datos a la cual se accede utilizando el dialecto SQL del modelo de base de datos que se utilice.

El API JDBC se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos. Un manejador de conexiones hacia un modelo de base de datos en particular es un conjunto de clases que implementan las interfaces Java y que utilizan los métodos de registro para declarar los tipos de localizadores a base de datos (URL) que pueden manejar. Para utilizar una base de datos particular, el usuario ejecuta su programa junto con la librería de conexión apropiada al modelo de su base de datos, y accede a ella estableciendo una conexión, para ello provee en localizador a la base de datos y los parámetros de conexión específicos. A partir de allí puede realizar con cualquier tipo de tareas con la base de datos a las que tenga permiso: consultas, actualizaciones, creado modificado y borrado de tablas, ejecución de procedimientos almacenados en la base de datos, etc.

La JDBC, es una herramienta muy importante y potente para desarrollar aplicaciones en Java. Su función principal, es la de permitir que nuestras aplicaciones se conecten a bases de datos relacionales, sobre las cuales, nuestros programas sean capaces de hacer consultas, actualizar registros, borrar registros, etc. Es un traductor que convierte los mensajes propietarios de bajo nivel del SMBD a mensajes de bajo nivel comprensibles a la api JDBC y viceversa Es una especificacion basada en 100 % en Java compuesta en dos partes

- api
- driver

3.2.1 FUNDAMENTOS

Antes de adentrarnos a JDBC, es necesario mencionar que es el ODBC de Microsoft. El ODBC o Conectividad Abierta de Base de Datos no es más que una interfaz común de programación para bases de datos; la cual nos permite crear programas que pueden acceder a diversos servidores de bases de datos. La intención de JDBC, es ofrecer un API común de programación para bases de datos en Java. Sin embargo todavía ¿muchos controladores de JDBC requieren, de ODBC para comunicarse con las bases de datos por lo que fue necesario integrar como controlador puente a JDBC-ODBC. ¿Pero cual es el porque de JDBC, si ODBC se comunica con una mayor cantidad de productos de bases de datos? La respuesta está en que ODBC está creado en C por lo que no esta orientado a objetos, lo cual es necesario dentro de Java. Las clases que se necesitan para explotar una base de datos pueden encontrarse, en el paquete java.sql, en las siguientes líneas aprenderemos el uso y aplicación de estas clases. Para usar la JDBC, es necesario un servidor de base de datos, y un controlador de base de datos. Cada fabricante de SMBD distribuyen su respectiva API en una archivo JAR

3.2.2 CLASES PRINCIPALES

Java posee un paquete con las clases necesarias para realizar las interfaces requeridas entre la aplicación y un origen de datos, comúnmente una base de datos relacional. El paquete es el JAVA.SQL

Este paquete tiene la siguiente composición:

Resumen de interfaces	
Array	Hace el mapeo de un tipo de datos ARRAY en el lenguaje JAVA™
Blob	Hace el mapeo de un tipo de datos BLOB en el lenguaje JAVA™
CallableStatement	La interface para ejecutar procedimientos almacenados SQL.
Clob	Hace el mapeo de un tipo de datos CLOB en el lenguaje JAVA™.
Connection	Una conexión (sesión) con una base de datos específica.
DatabaseMetaData	Información comprensiva sobre la base de datos en su totalidad.
Driver	La interface que implementa todos los drivers necesarios.
ParameterMetaData	Un objeto que se puede utilizar para conseguir la información sobre los tipos y las características de los parámetros en un objeto PreparedStatement.
PreparedStatement	Un objeto que representa una sentencia SQL precompilada.
Ref	La representación en JAVA™ de un valor REF SQL, el cual es una referencia a un tipo de valor estructurado SQL en la base de datos.

ResultSet	Una tabla de los datos que representan un sistema del resultado de la base de datos, que es generado generalmente ejecutando una declaración que consulte la base de datos.
ResultSetMetaData	Es un objeto que puede ser usado para obtener información acerca de los tipos y propiedades de las columnas en un objeto ResultSet.
Savepoint	La representación de un savepoint, que es un punto dentro de la transacción actual que se puede referir dentro del método de Connection.rollback.
SQLData	La interface usada para el mapeo personalizado de un tipo SQL definido por el usuario (UDT) a una clase en JAVA™.
SQLInput	Un stream de entrada que contiene una secuencia de valores que representan una instancia de un tipo estructurado SQL o un tipo distintivo SQL.
SQLOutput	La corriente de la salida para escribir las cualidades de un tipo definido por el usuario de nuevo a la base de datos.
Statement	El objeto usado para ejecutar una declaración estática del SQL y volver los resultados que produce.
Struct	El mapeo estándar para un tipo estructurado SQL.

Resumen de clases	
Date	Una mascara fina a nivel de milisegundos que permite a JDBC interpretar los valores DATE del SQL.
DriverManager	El servicio básico para administrar un juego de controladores JDBC. **NOTE:** La interfaz `DataSource`, nueva en la API JDBC 2.0, provee otra forma de conectar a la base de datos.
DriverPropertyInfo	Propiedades del controlador para hacer una conexión.
SQLPermission	Los permisos por los cuales el `SecurityManager` chequeará cuando el código que está corriendo en un applet llama al método `DriverManager.setLogWriter` o al método `DriverManager.setLogStream` (desaprobado).
Time	Una máscara fina sobre la clase `java.util.Date` que permite a la API JDBC identificar esto como un valor TIME de SQL.
Timestamp	Una pequeña máscara alrededor de la clase `java.util.Date` que permite a la API JDBC identificar esto como un valor TIMESTAMP de SQL.
Types	La clase que define las constantes que se utilizan para identificar tipos genéricos del SQL, llamados tipos JDBC

Se explicaran las interfaces y clases mas utilizadas

DriveManager

Esta clase se encarga de manejar y administrar todos los controladores JDBC, que se encuentran instalados en un sistema. Estos controladores se instalan cargando la clase del controlador por medio del método forName() de la clase Class.
La clase no ofrece constructores, aunque posee dos métodos sobresalientes, el getConnection y el getDrivers.

– getDrivers

```
public static Enumeration getDrivers();
```

Regresa un objeto de tipo Enumeration, cuyos elementos son los drivers de JDBC instalados en el sistema.

– Connection

```
public static  Connection getConnection(String url) throws SQLException;
```

Establece la conexión, con una base de datos; recibiendo como paramétro un objeto de tipo String, con el Url donde se localiza la base de datos.

Driver

La interfaz Driver la implementan los controladores JDBC. Esta interfaz tiene una serie de métodos muy útiles.

Connection

La interfaz Connection, define los métodos para interactuar con la base de datos a través de la conexión establecida por el método getConnection, de la clase DriveManager. Primero describiremos los métodos encargados de crear objetos, para después hablar de aquellos, en los que no interviene la creación de objetos.

– createStatement
Crea un objeto de tipo statement, los cuales se utilizan para ejecutar instrucciones en SQL.

– PrepareStatement
Este método crea, un objeto prepareStatement. Estos objetos son instrucciones precompiladas que se ejecutan de un modo más eficiente.

– prepareCall
Este método crea un objeto de tipo CallableStatement. Estos objetos no son más que instrucciones de llamada almacenadas en sql.

– close
Cierra una conexión con una base de datos.

– getMetaData
Devuelve un objeto de la interfaz DatabaseMetaData el cual nos sirve para obtener información detallada acerca de la estructura de la base de datos.

– setAutoCommit
Este método recibe un objeto de tipo booleano, en caso de que este sea verdadero, después de cada movimiento sobre la base de datos, se guardarán los cambios; en caso contrario, esto se deshabilita, obligándonos a determinar en que momento se han de guardar los cambios hechos sobre la base de datos.

– commit

Con este método, determinamos en que momento se guardarán los cambios hechos, sobre la base de datos.

DatabaseMetaData

Esta interfaz nos muestra todas las características de una base de datos, para cumplir este fin posee más de 100 métodos, en este caso si vale la pena consultar la dcumentación acerca de esta interfaz, ya que seguramente ahí podremos encontrar lafunción de cada uno de los métodos.

Statement

Esta interfaz como ya habíamos anticipado en el apartado de Connection, nos permite ejecutar instrucciones de SQL sobre la estructura de una base de datos desde une programa en Java, el método de mayor importancia dentro de esta interfaz es el método execute.

– execute

Este método recibe como parámetro una instrucción en SQL y nos devuelve un objeto de tipo booleano, que en el caso de haber obtenido resultados tendrá un valor verdadero.

– getResultSet

La función principal de este método es la de regresarnos un objeto de tipo ResultSet, en el cual podremos consultar los resultados, de las instrucciones de SQL, que han sido llevadas a cabo sobre la base de datos.

ResultSet

La función de la interfaz ResultSet es la de mantener un señalador, dentro de la tupla, de los resultados ofrecidos por una consulta a la base de datos, esta interfaz posee importantes métodos, para moverse dentro de una tabla.

– nextHace que el señalador avance a la siguiente posición de nuestra tabla, si existe esa siguiente posición devuelve un objeto de tipo booleano con valor verdadero, si ya no existen más tuplas que leer, entonces el valor que regresa se tornará falso.

– getString

Este método recibe el número de la columna que desea leer, para regresarnos la cadena que se encuentra en dicha columna. Existen más métodos que operan de manera similar, para obtener diferentes resultados, ya sean Bytes, así como números, en la mayoría de sus formatos(int, short, long, float, double).

ResultSetMetaData

Esta interfaz nos ofrece métodos en donde podemos ver las características de los objetos ResultSet, como el getColumnName(), para saber el nombre de la columna, o bien propiedades como el ancho y formatos.

Dando de alta un driver en Windows

A continuación veremos un ejemplo de una lectura sobre una base de datos. La base de datos ha sido nombrada escuela y su nombre físico es escuela.mdb. Antes de correr el ejemplo se debe de configurar el ODBC de Windows, este se encuentra en el panel de control. Una vez localizado el ODBC debemos entrar a configurar el DSN de sistema, entonces aparecerá una lista con los manejadores de bases de datos que reconoce el sistema una vez elegido el correcto nos aparecerá una pantalla para ingresar un ODBC, aquí lo que haremos será, en el

campo de “Nombre de origen de los datos” pondremos para este caso la palabra “escuela” y posteriormente en la parte inferior donde dice ”Base de Datos”, haremos un click en el botón seleccionar, con la finalidad de buscar mediante un cuadro de diálogo la ubicación de nuestra base de datos. Una vez hecho todo esto, sólo nos resta dar un click en aceptar.

```
import java.sql.*;
class PruebaBD {
public static void main(String args[]) throws SQLException,ClassNotFoundException {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conex=DriverManager.getConnection("jdbc:odbc:Escuela");
Statement orden=conex.createStatement();
if(orden.execute("Select Materia.descripcion,Alumno.nombre from
Alumno,AsignacionAlumnoCurso,Materia,Curso where AsignacionAlumnoCurso.boletaAlumno=
Alumno.boleta and AsignacionAlumnoCurso.claveCurso = Curso.clave and
Materia.clave=Curso.claveMateria order by Materia.descripcion"))
{
ResultSet resultados=orden.getResultSet();
ResultSetMetaData metaDatos=resultados.getMetaData();
String encabezados="";
for(int i=0;i<metaDatos.getColumnCount();i++) {
encabezados+=metaDatos.getColumnName(i+1)+" | ";
}
System.out.println(encabezados);
while(resultados.next()) {
String tupla="";
for(int i=0;i<metaDatos.getColumnCount();i++) {
tupla+=resultados.getString(i+1)+" | ";
}
System.out.println(tupla);
        }
 }
conex.close();
        }
}
```

3.2 SERVLET

Es común que requiramos de aplicaciones que den servicio a otras desde un servidor y que como salida generen contenido HTML. La primera solución a esta problemática fue propuesta por los llamados CGI’s, que eran programas hechos en distintos lenguajes como C o Perl. La gran desventaja de esta solución, es que los lenguajes que se utilizan para desarrollar CGI’s dependen demasiado de la arquitectura.

Esto fue percibido por Java-Soft(SUN), por lo cual crearon un entorno similar pero con las ventajas de la Tecnología Java; para resolver las peticiones hechas al servidor se utilizarían programas en Java llamados **Servlets**. Un servlet es una aplicación que permite recibir una petición, HTTP (Hypertext transfer protocol) en la mayoría de los casos, y devolver una respuesta a una aplicación o explorador web. El conjunto de APIs que soportan los servlets, permiten establecer una serie de operaciones que permiten conocer las características de la petición, del ambiente, del usuario remoto; además de manipular los parámetros de la respuesta.

Crear aplicaciones basadas en web es muy distinto a crear aplicaciones standalone(donde la interacción con el usuario es través de una GUI), ya que al momento de desarrollar debemos de pensar en que la única entrada que tendremos serán las peticiones hechas por los

usuarios y como salida la respuesta enviada por nuestros programas. En el desarrollo de este capítulo observaremos de manera práctica estas aseveraciones. A partir de ahora también conoceremos un nuevo termino, el de **Aplicación Web**, que se refiere a un conjunto de Servlets, Recursos(HTML, GIF, JPG, etc.. .), JSP's(Que veremos más adelante) que interactúan entre sí con fin común

3.3.1 FUNDAMENTOS

Para crear un servlet es necesario crear una clase en la cual se importe el paquete javax.servlet.* (donde se ubica el API de servlets para Java); y que implemente la interfaz Servlet. Por suerte Java ya incluye implementaciones de esta interfaz y una de ellas es la clase abstracta *javax.servlet.GenericServlet*. Desgraciadamente esta clase es una clase general que permite crear un servlet para cualquier protocolo de Internet. Por lo que para probar nuestro primer Servlet necesitaremos crear una clase que soporte el protocolo HTTP; de nueva cuenta tenemos la suerte de que ya se encuentre creada una clase abstracta que permita usar dicho protocolo, esta es la clase: *javax.servlet.http.HttpServlet*. El protocolo HTTP se compone de distintos métodos de petición, como son PUT, HEAD, GET, POST, OPTIONS y TRACE. Dependiendo de cada forma de petición existirá un método encargado de manejar la petición dentro de la clase HttpServlet. Este método deberá de ser implementado al momento de crear la clase del Servlet Por ejemplo si se uso POST el método a implementar será doPost (HttpServletRequest request, HttpServletResponse response) El parámetro *request* sirve para conocer propiedades de la petición y el *response* permite modificar y conocer propiedades de la respuesta. El siguiente ejemplo muestra el código de un servlet llamado SaludadorServlet que saluda a una persona, indicando su nombre, el cual es recibido en el Servlet mediante el objeto request. En este caso el método de envió al servidor Sera POST.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class SaludadorServlet extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException,IOException {
String nombre = request.getParameter("nombre");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<TITLE>ESTAS EN EL SERVLET SALUDADOR</TITLE>");
out.println("<BODY bgcolor=\"##99FF99\"");
out.println("<P>" + nombre + ", ES UN GUSTO DARTE LA BIENVENIDA AL MÁGICO
MUNDO DE LOS SERVLETS</P>");
out.println("</BODY>");
out.println("</HTML>");
out.close();
}
}
```

Un S*ervlet* no corre por si solo necesita estar montado sobre un Servidor Web, un servidor Web es una herramienta encargada de establecer el mecanismo de conexión entre los

clientes y el *Servlet*. También necesitaremos de una página HTML (saludador.html) que mande llamar al servlet y que envié el parámetro que este recibe.

```
<HTML>
<HEAD>
<TITLE>BIENVENIDO A JAVA PARA WEB</TITLE>
</HEAD>
<BODY>
<FORM ACTION="/saludador/servlet/SaludadorServlet" METHOD="POST">
P>INGRESA TU NOMRE<INPUT TYPE="text" SIZE="40" NAME="nombre">
<INPUT TYPE="submit" VALUE="Enviar"></P>
</FORM>
</BODY>
</HTML>
```

Como servidor Web usaremos el que sugiere Java llamado TomCat, de Apache que viene en el ide net beans Una vez compilado el servlet el archivo **.class* generado se guardará en el directorio *\classes* dentro de *\saludador\WEB-INF* .
Todas las clases que se utilicen dentro de la aplicación web serán guardadas en el directorio saludador\WEB-INF\classes. Mientras que todos los recursos compartidos(HTLM, JSP, GIF, JPG, etc...) serán guardados en el directorio raiz de la aplicación en este caso \saludador. Así es que para que el ejemplo funcione correctamente debemos de colocar el archivo *saludador.html* que generamos anteriormente en el directorio *\saludador.*
El ultimo paso será escribir el archivo de configuración web.xml este archivo es un documento xml que configurá las características de una aplicación web debe de guardarse dentro del directorio \WEB-INF de una aplicación web y siempre debe de llamarse *web.xml.*
El archivo para este ejemplo tiene el siguiente aspecto:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
<servlet>
<!-- Servlet alias -->
<servlet-name>SaludadorServletAlias</servlet-name>
<!--Clase del Servlet-->
<servlet-class>SaludadorServlet</servlet-class>
</servlet>
</web-app>
```

en el ide al darle run desplegara la publicacion No se explica la intalacion ni confifuracion del net beans ya que este trabajo solo se basa en el desarrollo de las aplicaciones puras en el lenguaje

3.3.3 CLASES E INTERFAZ DE SERVLET

Los servlets viene de la clase principal dada por sun :

javax.servlet	El javax.servlet package contiene un numero de clases e interfaces que describen y definen los contratos de la clase servlet y el ambiente de tiempo de ejecucion, provee una instancia la cual se ajusta a la clase del contenedor del servlet

Dentro de las interfazces tenemos:

Filter	Un filtro es un objeto que funciona filtrando tareas o peticiones de recursos (un servlet de contenido estatico), o una respuesta del recurso, o ambos.
FilterChain	Un Objeto de tipo cadena es proveido por el servlet container para que el desarrollador nos de una vista dentro de la invocacion de una cadena de una peticion filtrada para un recurso.
FilterConfig	Es un objeto usado por un servlet container para pasar informacion al filtro duranrte la inicializacion.
RequestDispatcher	Define un objeto que recibe peticiones de un cliente y los manda hacia algun recurso(como un Servlet, un archivo HTML , o un archivo JSP) sobre elservidor.
Servlet	Defines methods that all servlets must implement.
ServletConfig	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletContextAttributeListener	Implementations of this interface receive notifications of changes to the attribute list on the servlet context of a web application.
ServletContextListener	Implementations of this interface receive notifications about changes to the servlet context of the web application they are part of.
ServletRequest	Defines an object to provide client request information to a servlet.
ServletRequestAttributeListener	A ServletRequestAttributeListener can be implemented by the developer interested in being notified of request attribute changes.
ServletRequestListener	A ServletRequestListener can be implemented by the developer interested in being notified of requests coming in and out of scope in a web component.
ServletResponse	Defines an object to assist a servlet in sending a response to the client.
SingleThreadModel	**Deprecated.** *As of Java Servlet API 2.4, with no direct replacement*

La interfaz Servlet

La interfaz Servlet es la encargada de proporcionar a otras clases las características necesarias para implementar una comunicación basada en petición y respuesta. Esta interfaz específica el contrato entre el contenedor web y un servlet. La programación de un servlet se puede hacer implementar directamente de la interfaz Servlet. Para ello se deben implementar los siguientes cinco métodos

public void init(ServletConfig configuracion)

Una vez instanciado el servlet, el contenedor web llama el método init( )

public void service(ServletRequest request, ServletReponse response)

Esta es el punto de entrada para ejecutar la aplicación lógica del servlet

public void destroy( )
El contenedor llama este método cuando esta a punto de eliminar el servlet
public ServletConfig getServletConfig( )
Devuelve el objeto de configuración que será pasado como parámetro al método init( )
public String getServletInfo( )
Devuelve un objeto String con información acerca del servlet
Para crear un servlet específico para HTTP es necesario de heredar de la clase HttpServlet que hereda de la clase abstracta GenericServlet que a su vez implementa la interfaz Servlet. La clase GenericServlet tiene la lógica de cualquier servlet, implementa los métodos de la interface Servlet excepto service( ) y adiciona nuevos métodos:
public init( )
Este método implementa el método init(ServletConfig)
public void log(String message)
Registra el nombre del servlet y el valor del parámetro message en el contenedor web
public abstract void service( )
El método service( ) es el responsable de establecer la lógica del servlet, y para obligar a las clases que heredan de GenericServlet a implementarlo se define como abstracto.
La clase HttpServlet hereda de la clase GenericServlet (La estructura lógica de un servlet) y provee de una implementación especifica de HTTP (Protocolo de transporte de hipertexto) para la interfaz Servlet. Las clases que hereden de HTTPServlet ofrecen servicios de HTTP es decir transportar información a través de HTTP. Esta clases específica los siguientes métodos:
public void service(ServletRequest request, ServletResponse response)
Este método recibe los parámetros ServletRequest y ServletResponse y hace un cast a cada uno de ellos a HttpServletRequest y HttpServletResponse y llama al método service (HttpServletRequest, HttpServletResponse)
protected void service(HttpServletRequest request, HttpServletResponse response)
Esté método toma objetos request y response específicos de HTTP, que después serán utilizados por otros métodos. Es importante indicar que todas las clases que heredan de HttpServlet ya tienen el método service( ) implementado y no será necesario programarlo, solo es necesario programar los métodos doGet( ), doPost( ), doDelete( ), doPut( ), que son llamados automáticamente por al método service( ) cuando se requieren. Los siguientes métodos programan los servios de http. Son llamados cuando se hace la petición correspondiente de http, GET, POST, DELETE, PUT.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
Este método es llamado cuando un cliente hace una petición GET, el método de petición por defecto de http para solicitar una página web. Dentro de este método, la petición del usuario está representada por un objeto HttpServletRequest y la respuesta por un objeto HttpServletResponse.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
Es método es llamado cuando el cliente hace una petición POST.
protected void doDelete(HttpServletRequest request, HttpServletResponse response)

Este método es utilizado cuando el cliente solicita eliminar una página web, utilizando la petición DELETE de http.

protected void doPut(HttpServletRequest request, HttpServletResponse response)

Este método es llamado cuando el cliente solicita almacenar una página web y utiliza la petición de http denominada PUT. Dentro del método doPut( ), la petición del usuario está representada por un objeto HttpServletRequest.

El orden de ejecución de estos métodos es el siguiente:

1.- El contenedor llama el método public service( )

2.- El método public service( ) llama al método protected service( )

3.- El método protected service( ) llama a uno de los métodos doAlgo( ), dependiendo del tipo solicitud de http

Configuración de servlets

La interfaz ServletConfig

public interface ServletConfig

Esta interfaz representa la configuración de un servlet . La información de configuración abarca: inicialización de parámetros, el nombre del servlet y un objeto de la clase ServletContext que obtiene información acerca del contenedor, esto es configurado a través del archivo xml web.xml. La clase HttpServlet ya implemente esta interfaz, así que todas sus subclases podrán hacer uso de sus métodos. Esta interfaz específica los siguientes métodos:

public String getInitParameter(String name)

Retorna el valor de inicialización del parámetro name , si el parámetro no tiene valor inicial el método retorna null

```
Ej. String nombreNegocio = this.getInitParameter("nombreNegocio");
```

public Enumeration getInitParameterNames( )

Retorna los valores de inicialización de todos los parámetros, si los valores no están inicializados retorna un Enumeration vacío

public ServletContext getServletContext( )

Retorna una referencia de ServletContext asociado con la aplicación web.

public String getServletName( )

Este método retorna el nombre asignado al servlet definido en el archivo web.xml

```
Ej. String nombreServlet = this.getServletName();
```

Excepciones de servlets

Los servlets al igual que todos los programas en java pueden lanzar excepciones. En el caso de los servlets las excepciones son manejadas por dos clases: ServletException y UnavailableException que forman parte del paquete javax.Servlet.

Clase ServletException

Los métodos init( ), service( ) , destry() y doAlgo( ) lanza este tipo de excepciones. Esta clase puede ser instanciada por los siguientes constructores:

public ServletException( )

Este constructor permite lanzar un objeto de este tipo de excepción que devuelve un mensaje por default

public ServletException(String mensaje)

Este constructor permite establecer el mensaje a mostrar cuando se lance la excepción

Clase UnavailableException
Este es un tipo especial de excepción de servlet que hereda de ServletException. Todos los métodos que pueden lanzar una ServletException también pueden lanzar la excepción UnavailableException. Su objetivo es indicar al contenedor web que el servlet esta temporalmente o definitivamente indisponible.

Los constructores de esta clase son:
public UnavailableException(String mensaje)
Recibe como parámetro el mensaje a desplegar cuando se lance la excepción
public UnavailableException(String mensaje, int segundos)
Recibe el mensaje a desplegar y el siguiente argumento indica en segundos la duración en la cual en servlet no está disponible

Configuración de un Servlet

A partir de la versión 2.2 la manera de configurar un servlet se realiza mediante el archivo web.xml el cual es encargado de configurar a todos los Servlets de una aplicación web. En el ejemplo de la sección *Desarrollando un Servlet* utilizamos algunas configuraciones del archivo web.xml para dar de alta un servlet. En aquella ocasión solo indicamos el nombre y la clase del Servlet. En cambio, ahora veremos más a fondo las principales configuraciones que se pueden realizar usando el archivo web.xml y las etiquetas definidas por la DTD web-app_2_2.dtd. A continuación se muestra una descripción de los principales tags definidos en la DTD web-app_2_2.dtd:

Etiqueta <context-param>

Este tag servirá para indicar parámetros generales para todos los Servlets de una aplicación web, estos parámetros pueden ser usados mediante los métodos getInitParameter(String Nombre_Parametro) y el método getInitParameterNames() definidos en la interfaz *ServletContext*. Para establecer el nombre del parámetro usaremos la etiqueta *<param-name>* y para indicar su valor utilizaremos el tag *<param-value>*.

Etiqueta <servlet>

Escribiremos una etiqueta *<servlet>. . .</servlet>* por cada Servlet que se desee instalar en la aplicación web. Dentro de esta etiqueta tendremos que utilizar otra serie de tags para establecer la configuración de un Servlet. Estos tags son:
<servlet-name>Nombre_de_Servlet</servlet-name>
Esta etiqueta servirá para indicar el nombre lógico del Servlet.
<servlet-class>Nombre_de_la_Clase</servlet-class>
Aquí señalaremos el nombre de la clase a la que pertenece el servlet
<jsp-file>Nombre_Archivo.jsp</jsp-file>
Este tag servirá pará que indiquemos la existencia de un archivo jsp, más adelante en la sección de JSP's veremos como y cuando usar este tag. Este Tag se usa en lugar de *<servlet-class></servlet-class>*
<load-on-startup>1</load-on-startup>
Este tag es opcional y sirve para establecer el orden en que se instanciarán e i inicializarán los Servlets(Una ves que hayamos visto la sección de Ciclo de Vida de un Servlet comprenderemos más esto). El valor que colocaremos dentro de la etiqueta será un número y dependiendo del valor de el número el contenedor ira creando la instancia del Servlet. Si no se coloca ningún valor en este Tag, el contenedor se encargará de administar el orden de instanciación e inicialización de los Servlets.

<init-param>
<param-name>Nombre_Parametro</param-name>
<param-value>Valor_Parametro</param-value>
La etiqueta *<init-param>. . .</init-param>* permite que señalemos, en caso de que el Servlet lo requiera, una serie de parámetros de inicialización. Estos parámetros se crean usando la etiqueta *<paramname>* para indicar el nombre del parámetro y la etiqueta *<paramvalue>* para indicar el valor del parámetro. Una etiqueta *<init-param>* puede contener en su interior diversas parejas de etiquetas *<paramname>* y *<param-value>*. Estos parámetros se obtienen usando el método *getInitParameter(String Nombre_Parametro)*, pero ahora utilizaremos el método definido en la interfaz *ServletConfig.*

```
<servlet>
<servlet-name>Nuevo</servlet-name>
<servlet-class> NuevoServlet </servlet-class>
<load-on-startup>1</load-on-startup>
<init-param>
<param-name>MiParametro</param-name>
<param-value>UnValor</param-value>
</ init-param >
</servlet>
```

Etiqueta <servlet-mapping>

Esta etiqueta permite añadir más de un nombre lógico a un servlet, esto es, más de una forma de acceder el servlet desde un explorador web. Para ello usará dos tags internos, el tag <servlet-name>Nombre_Servlet</servlet-name> en el cuál señalaremos el nombre del servlet a relacionar y el tag <url-pattern> en el que colocaremos el url que deseamos establecer como nombre lógico. Por ejemplo si tenemos un servlet con el nombre *Administrador* y queremos accesarlo al abrir el directorio *http://localhost:8080/aplicación/admin* tendremos que usar la etiqueta <servlet-mapping> de la siguiente manera.

```
<servlet-mapping>
<servlet-name>Adminstrador</servlet-name>
<url-pattern>/admin/*</url-pattern >
</servlet-mapping>
```

Etiqueta <session-config>

Este Tag sirve para establecer la configuración de tiempo de espera de session en una Aplicación Web. Para ello usa un tag interno llamado <sessiontimeout> tiempo en minutos</session-timeout>. Por ejemplo siqueremos que una sesión no dure más de 30 minutos usaremos el tag:

```
<session-config>
<session-timeout>30</ session- timeout >
</ session-config >
```

Etiqueta <mime-mapping>

MIME(Multipurpose Internet Mail Extensions) es un estandar que fue creado inicialmente para e-mails no basados en texto y que ahora es usado por las aplicaciones web para señalar el tipo de contenido que envián a los clientes. Un tipo MIME se especifica de la siguiente manera: tipoMIME/SubtipoMIME por ejemplo si queremos indicar que una página utiliza HTML el tipo MIME será escrito de la siguiente manera: text/html si usaramos una imagen JPG se utilizará de la siguiente forma: image/jpg. Utilizando la etiqueta <mime-mapping> se utiliza para relacionar tipos MIME con extensiones para ello usará dos etiquetas la primera <extension> indicará la extensión y la segunda <mime-type> señalará el tipo MIME correspondiente.

En el siguiente ejemplo relacionaremos la extensión *.jpg con el tipo mime Correspondiente

```
<mime-mapping>
<extension>jpg</extension>
<mime-type>image/jpg</mime-type>
</mime-mapping>
```

Etiqueta <welcome-file-list>

Usualmente el archvo de bienvenida para un sitio web siempre es el que lleva por nombre index.html. También los servidores web establecen como archivo de bienvenida a index.html. Más sin embargo podemos establecer mediante el uso de la etiqueta <welcome-file-list> </welcome-file-list> una serie de archivos de bienvenida. La siguiente sección de código xml muestra como se genera una lista de archivos de bienvenida.

```
<welcome-file-list>
<welcome-file>inicio.html</welcome-file>
<welcome-file>servicio.jsp</welcome-file>
</welcome-file-list>
```

Los archivos en la lista de archivos de bienvenida están ordenados de tal forma que el primero en mostrarse en este caso será el archivo inicio.html, en caso de no encontrarse este archivo en el servidor se mostrará el siguiente archivo de la lista en este caso el archivo servicio.jsp.

Etiqueta <error-page>

Esta etiqueta servirá para establecer páginas dedicadas al manejo de Errores http y Excepciones de la aplicación. Para realizar esto se servirá de tres etiquetas, la primera <error-code> se usará en caso de que se trate de manejar un Error de HTTP, la segunda <error-type> se utilizará cuando se trate de manejar una excepción de Java. La última etiqueta <location> indica el archivo html que va a mostrarse cada vez que ocurra el error o excepción.
A continuación se muestra un ejemplo para excepción de Java:

```
<error-page>
<exception-type>java.sql.SQLException</exception-type>
<location>/ExcepcionSQL.html</location>
</ error-page >
```

Y uno más para un error de HTTP

```
<error-page>
<error-code>404</error-code>
<location>/404.html</location>
</ error-page >
```

Existen otros tags que se utilizan para integrar aplicaciones web con el modelo de componentes EJB. Puesto que nosotros no crearemos en este momento ninguna basada en este modelo omitiremos el uso de estos tags.

Ciclo de vida de un servlet

Los métodos relevantes para el ciclo de vida de los servlets son: init( ), service( ), and destroy( ). El ciclo de vida inicia cuando el contenedor llama al método init( ), y finaliza cuando llama al método destroy( ). Básicamente, el ciclo de vida de un servlet contiene los metodos:

Instanciación: El contenedor web hace una instancia del servlet Inicialización: El contenedor llama al método init( )

Servicio: si el contenedor hace una solicitud (request) al servlet, el servlet llama al método service( )

Destrucción: Antes de eliminar el objeto, el contenedor llama al método destroy( )

El contenedor crea una instancia de un servlet como repuesta a la llegada de una solicitud http. Después de la instanciación, el contenedor inicializa el objeto invocando al método init( ).
Después de la inicialización, el objeto esta listo para las solicitudes que lleguen del servidor. El objetivo del proceso de inicialización es llamar los parámetros requeridos por el servlet, que están definidos en elarchivo de configuración web.xml.
Durante el proceso de inicialización, la instancia del servlet puede lanzar una ServletException indicando que han ocurrido errores en la inicialización, por ejemplo: No encontrar los parámetros de inicialización. Con esto el contenedor garantiza que antes del llamar al método service el método init() ha sido correctamente ejecutado.
El servlet puede lanzar una excepción de tipo ServletException o UnavailableException durante la ejecución del método service( ), en cualquier caso se podrían suspender la recepción de solicitudes del contendor temporal o permanentemente.
El contenedor web se arranca (startup) carga e inicializa los servlets o cuando el servlet es llamado por primera vez y cuida que las instancias de los servlets en memoria den servicio a todas las peticiones. El contendor puede decidir en que tiempo liberar la referencia de servlet , y así terminar con su ciclo de vida, esto podría pasar, si el servlet no ha sido llamado por algún tiempo o si el contenedor es dado de baja (shutdown). Cuando esto sucede el contenedor llama automáticamente al método destroy( ). Un ejemplo vale más que mil palabras, en el siguiente ejemplo, analizaremos el código identificando las secciones referentes al ciclo de vida de un servlet.

```
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
// ====================================================
class Libro {
private String titulo;
private String autor;
private int precio;
private int clave;
public Libro(String titulo, String autor, int precio, int clave) {
this.titulo = titulo;
this.autor = autor;
this.precio = precio;
this.clave = clave;
}
public String obtenerTitulo() {
return titulo;
}
public String obtenerAutor() {
return autor;
}
public int obtenerPrecio() {
return precio;
}
public int obtenerClave() {
return clave;
}
}
// ====================================================
```

```
class CatalogoLibros {
private Hashtable catalogo;
public CatalogoLibros() {
catalogo = new Hashtable();
}
public static CatalogoLibros catalogoPorDefault(){
CatalogoLibros unCat = new CatalogoLibros();
unCat.agregarLibro(new Libro("Distributed Systems","Coulouris",720,1));
unCat.agregarLibro(new Libro("Redes de computadoras","Tanenbaum",320,2));
unCat.agregarLibro(new Libro("Inside ODBC","Geiger",520,3));
return unCat;
}
public Enumeration libros() {
return catalogo.elements();
}
public void agregarLibro(Libro unLibro) {
catalogo.put(new Integer(unLibro.obtenerClave()),unLibro);
}
public Libro obtenerLibro(int clave) {
return (Libro)catalogo.get(new Integer(clave));
}
}
// ====================================================
public class ServletCarritoLibros extends HttpServlet {
private CatalogoLibros unCat;
private PrintWriter salida;
private String nombreLibreria;
// Para instanciar cualquier clase y entre ellas el servlet es necerio llamar al constructor, por lo
tanto cuando el contenedor web instancia un servlet lo primero que hace es llamar al
constructor
public ServletCarritoLibros() {
unCat = CatalogoLibros.catalogoPorDefault();
System.out.println("Instanciación");
}
// El método init( ) es el encargado de inicializar el servlet, accesando al archivo web.xml donde
se encuentran los parámetros a leer
public void init() {
String nombreServlet;
nombreServlet = this.getServletName();
nombreLibreria = this.getInitParameter("nombre_Libreria");
System.out.println("Inicialización del servlet: "+nombreServlet);
}
// El método service() llama al método doGet( ) , en el cual podemos identificar la activación del
servicio
public void doGet(HttpServletRequest req,HttpServletResponse resp)
throws ServletException,IOException {
System.out.println("Servicio");
salida = new PrintWriter(resp.getOutputStream());
salida.println("<HTML>");
mostrarProductos(salida,unCat);
salida.println("<P>");
salida.print("</HTML>");
salida.close();
}
public void mostrarProductos(PrintWriter salida,CatalogoLibros unCat) {
Enumeration e=unCat.libros();
salida.println("<B>Bienvenidos a la libreria " +nombreLibreria+" </B>");
salida.println("<TABLE BORDER=\"1\" WIDTH=\"50%\">");
salida.print("<TABLE BORDER=\"1\" WIDTH=\"50%\">");
salida.print("<TR>");
salida.print("<B>");
salida.print("<TD WIDTH=\"10%\">");
salida.print("Clave");
```

```
salida.print("<TD>");
salida.print("Nombre del Libro");
salida.print("</TD>");
salida.print("<TD WIDTH=\"10%\">");
salida.print("Nombre del Autor");
salida.print("</TD>");
salida.print("<TD WIDTH=\"10%\">");
salida.println("Precio");
salida.print("</TD>");
salida.print("</B>");
salida.print("</TR>");
while(e.hasMoreElements()) {
salida.print("<TR>");
Libro unLibro =(Libro)e.nextElement();
salida.print("<TD WIDTH=\"10%\">");
salida.print(unLibro.obtenerClave()+"\t");
salida.print("</TD>");
salida.print("<TD>");
salida.print(unLibro.obtenerTitulo()+"\t");
salida.print("</TD>");
salida.print("<TD WIDTH=\"10%\">");
salida.print(unLibro.obtenerAutor()+"\t");
salida.print("</TD>");
salida.print("<TD WIDTH=\"10%\">");
salida.println("$"+unLibro.obtenerPrecio());
salida.print("</TD>");
salida.print("</TR>");
}
salida.print("</TABLE>");
}
// El método destroy( ) es activado cuando el contenedor web es dado de baja y por lo tanto la
instancia de servlet es eliminada
public void destroy() {
System.out.println("Destrucción");
try {
// Retarda este método para ver el mensaje
Thread.sleep(1500);
}
catch(InterruptedException ex) {
ex.printStackTrace();
}
}
}
```

Utilizando HttpServletRequest y HttpServletResponse

Durante las anteriores secciones hemos utilizado los métodos definidos enestas interfaces solo para tomar parámetros y regresar contenido HTML a los clientes. Ahora exploraremos más afondo el uso de estas dos interfaces y evaluaremos cuando y cómo utilizar sus métodos.

HttpServletRequest.

Esta interfaz a parte de permitir conocer las características del una petición hecha al servidor, nos permite manejar elementos del protocolo HTTP como los Headers. Dividiremos los principales métodos de esta interfaz en 6 distintas categorías dependiendo de su función. A continuación se detallarán cada una de dichas categorías:

Métodos para manejar propiedades de la petición
public String getProtocol();

Nos permite conocer el protocolo que se esta usando para establecer la conexión entre el cliente y el servidor

public String getServerName();

Este método retorna una String que contiene el nombre del servidor.

public int getServerPort();

Devuelve el Puerto por el cual se encuentra escuchando peticiones el servidor.

public String getRemoteAddr();

Este método regresa la dirección IP del cliente que hace la petición

public boolean isSecure();

Determina si la connexion es segura o no.

Métodos para manejar parámetros

public String getParameter(String key)

Este método ya lo hemos utilizado con anterioridad. Su función principal es obtener un parámetro enviado en la petición. Este parámetro será aquel cuyo nombre coincida con el valor enviado como argumento al método.

public String[ ] getParameterValues(String key)

Si nuestro parámetro envía más de un valor(como una lista de selección multiple) usaremos este método para obtener los distintos valores del parámetro.

public Enumeration getParameterNames()

Regresa un objeto de tipo Enumeration el cuál contiene los nombres de todos los parámetros enviados por la petición, si la petición no ha envía ningún parámetro entonces el valor retornado será un objeto Enumeration vacio.

Métodos para manejar Atributos

Aparte de definir métodos para manejar parámetros la interfaz HttpServletRequest define métodos para manejar atributos. Conoceremos como atributos a objetos de Java que son añadidos a la petición, también pueden añadirse este tipo de atributos a objetos de Tipo HttpSession y HttpContext.

La finalidad de estos atributos es poder enviar Objetos dentro de una petición, esto es util cuando deseamos reenviar una petición(Veremos esto más a fondo en la sección de colaboración entre Servlets)

public object getAttribute(String name)

Este método se encarga de devolver el atributo indicado por el parámetro name.

public Enumeration getAttributeNames()

Este método regresa un objeto de tipo Enumeration el cual representa una lista que contiene los nombres de los atributos.

public setAttribute(String name, Object o)

Este Método agrega un atributo a una petició. El nombre del Atributo es el valor enviado en el argumento name.

public removeAttribute(String name)

Este método lo utilizaremos para eliminar un atributo de la petición.

Métodos de Entrada de Datos

Una petición también contiene un flujo del cual podemos leer tanto bytes como caracteres. Dependiendo del caso usaremos alguno de los siguientes métodos

Si queremos leer un flujo de bytes:

public ServletInputStream getInputStream()

Si queremos leer un flujo de caracteres:

public BufferedReader getReader()

Metodos para manejo de URL

La interfaz HttpRequest define métodos que nos muestran información extra sobre los URL's

public String getPathInfo()
Este método retorna la información extra del URL de la petición en caso de no existir información extra devolverá un valor nulo. La información extra del URL. Se considera información extra a todo lo que vaya escrito después del nombre del Servlet. Por ejemplo si yo tengo un Servlet llamado ConsultaServlet al cuál le asigne el alias Consulta, accederé a el mediante el siguiente URL:
http://www.patito.com:8080/MiAplicacionContext/Consulta/cuentas
En este caso el valor que nos retornaría este método sería /cuentas
public String getPathTranslated()
Este método regresa un path real para la información extra del URL. Siguiendo el caso del ejemplo anterior si nuestras clases se encuentran en el directorio C:\directorio_de_trabajo\MiAplicacion\WEB-INF\classes el método regresará el path siguiente: C:\directorio_de_trabajo\MiAplicacion\cuentas
public String getQueryString()
Regresa el Query asociado con la petición
public String getRequestURI()
Retorna el path asociado con la petición incluyendo información extra. En este caso será:

```
/MiAplicacionContext/Consulta/cuentas
```

public String getServletPath()
Retorna el path asociado con la petición sin incluir información extra. En este caso será:

```
/MiAplicacionContext/Consulta/cuentas
```

Métodos para manejo de Headers
Estos nos permitiran conocer los valores de los Headers de HTTP.
public String getHeader(String name)
Este método regresará el valor del Header especificado por el argumento name. Si no se encuentra ningún valor entonces el valor retornado será nulo.
public Enumeration getHeaders()
Devuelve un objeto de tipo Enumeration el cual contiene el valor de todos los headers de la Petición.
public Enumeration getHeaderNames()
Devuelve un objeto de tipo Enumeration el cual contiene el nombre de todos los headers de la Petición.

HttpServletResponse

La interfaz HttpServletResponse nos permite establecer y conocer propiedades de la respuesta enviada al cliente por el servidor. Dividiremos para su mejor compresión los métodos de esta interfaz en 4 categorías.
Métodos de Contenido
Estos métodos sirven para establecer el tipo de contenido MIME de la respuesta y el tamaño del contenido de la respuesta.
public void setContentType(String type)
Establece el contenido de la repuesta basado en el estandar MIME, por ejemplo si deseamos que la respuesta tenga contenido HTML el valor del argumento type será: "text/HTML".
public void setContentLength(int size)
Este método se utiliza para establecer el valor de la variable de servidor CONTENT_LEGTH

Métodos de salida de datos

Establecen la forma en que saldrán los datos hacia los clientes existen dos formas hacerlo una es utilizando Flujos Bytes y otra usando Flujos de Caracteres.

Para uso de Bytes emplearemos:
public ServletOutputStream getOutputStream() throws java.io.IOException
Para uso de caracteres emplearemos:
public PrintWriter getWriter() throws java.io.IOException
Métodos para uso de Buffers
La interfaz HttpServletResponse ofrece un conjunto de métodos encargado se manejar el buffer estos métodos son:
public void setBufferSize(int size)
Establece el tamaño del Buffer de salida
public int getBufferSize()
Devuelve un dato entero el cual representa el tamaño del buffer.
public void flushBuffer() throws java.io.IOException
Este método permite escribir el contenido del buffer en el cliente
public bolean isCommited()
Este método regresa un valor booleano dependiendo si la respuesta contenida en el buffer fue o no enviada al cliente.
public void reset()
Este método elimina el contenido existente en le bufer
Métodos para manejar Errores
Estos métodos permiten enviar errores HTTP y también permiten revisar el estado de HTTP.
public void sendError(int status)
Envia un error especificado por el número status por ejemplo si queremos enviar el error 404(Recurso no Encontrado) el método se utilizará de la siguiente manera:

```
response.sendError(404);
```

Existe una implementación más del método sendError(int status, String message) este método a parte de enviar un código de error también envía una cadena la cual contiene un mensaje.
public void setStatus(int status)
Este método sirve para establecer estados HTTP que no sean errors.
public void sendRedirect(String location)
Redirecciona la respuesta al valor definido por el argumento location.

Colaboración entre servlets

Un servlet común recibe una petición http, ejecuta alguna aplicación y envía la respuesta al navegador. Sin embargo hay escenarios en donde este modelo no es adecuado:

– Cuando un servlet_1 recibe una petición http de un cliente, atiende la petición, y otro servlet_2 es el encargado de envíar la respuesta. En este caso el servlet_1 no es el responsable de generar la respuesta, ahora el servlet_2 es el responsable de generar el contenido dinámico.

– Cuando un servlet recibe una petición http de un cliente, y procesa una parte de la aplicación y envía la petición a otro servlet para terminar el procesamiento y este asu vez a una página html.

En ambos casos, el procesamiento de la petición se hace entre varios recursoWeb. Por ejemplo, cuando el contenedor web recibe una solicitud, construye los objetos correspondientes a la solicitud (HttpRequest, HttpResponse) y con ellos invoca al servicio del servlet correspondiente. Este es un proceso donde el contenedor web envía una solicitud a un servlet. Si este servlet desea enviar la misma solicitud a otro servlet después de un proceso preliminar ¿Cómo lo haría?. Para este caso el primer servlet debería obtener una referencia del segundo servlet para poder enviar la solicitud. . Esto es lo que hace por medio de la interfaz RequestDispatcher. Esta interfaz permite obtener la referencia de otro recurso web a través de su path. Para poder enviar las referencias de los objetos HttpRequest y HttpResponse a otro recurso web, la interfaz RequestDispatcher define dos métodos:

El método forward( )

public void forward(ServletRequest request, ServletResponse response)
throws ServletEsception, IOException

Este método permite enviar las referencias de solicitud y respuesta a otro recurso web (servlet, página jsp o archivo HTML) dejando así el control al recurso ,que obtiene la información, es decir, desde el momento en que el recurso obtiene la información de la petición, es el responsable de terminar la aplicación y devolver la respuesta al navegador o de procesar la petición y enviarla a otro recurso web.

El método include( )

public void include(ServleRequest request, ServletResponse response)
throws ServletException, IOException

Este método es similar al método forward( ), el recurso obtiene la información de la petición, realiza el proceso que tenga que hacer, con la diferencia, de que cuando el proceso termina el control regresa al servlet que llamo. Con esto podemos separar una aplicación en diferentes servlets que sean llamado por uno solo.Por ejemplo, para elaborar un servlet encargado de mostrar los precios de diferentes marcas de autos, donde para cada marca tiene un servlet específico para recopilar sus precios y mostrarlos en el navegador. El servlet principal utilizaría include para poder llamar a un servlet de una marca específica y no perder el control para seguir llamando a los otros servlets. Para utilizar los método forward() e include() es necesario crear un objeto de tipo RequestDispatcher. Obtener un objeto de tipo RequestDispatcher se pudehacer de dos formas:

1. public RequestDispatcher getNamedDispatcher(String nombre_servlet)

Este método es utilizado para obtener un objeto de tipo RequestDispatcher a través del nombre del servlet definido en el archivo de configuración web.xml. Este método pertenece a la clase SerlvetContext y para obtener un objeto de este tipo se utiliza el método getServletContext() de la clases HttpServlet heredado de GenericServlet. Para obtener un objeto de tipo RequestDispatcher del 'Servlet1', almacenado en webapps/contexto/Web-inf/classes/Servlet1.class y cuyo nombre definido en el archivo web.xml es 'uno', desde el servlet 'Servlet2'. Se haría de la siguiente manera:

```
public class Servlet2 extends HttpServlet {
private RequestDispatcher rd;
....
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
rd = this.getServletContext().getNamedDispatcher("uno");
}
….
}
```

2. public RequestDispatcher getRequestDispatcher(String path)

Este método recibe la ruta del servlet del cual se obtendrá la referencia. Y también se pude utilizar para obtener referencias de otros recursos. Este método pertenece a la clases ServletRequest, por lo tanto se puede utilizar en objetos de la clase HttpServletRequest porque hereda de ServletRequest. Para obtener un objeto de tipo RequestDispatcher desde el Servlet3, de la página inscripción.html almacenada en webapps/contexto/inscripción.html. Se haría de la siguiente manera:

```
public class Servlet3 extends HttpServlet {
private RequestDispatcher rd;
….
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
rd = request.getRequestDispatcher("/inscripcion.html");
}
….
}
```

Utilizando el método forward()

El siguiente ejemplo es un servlet que recibe los datos de un usuario a través del objeto request y los válida en un registro, si la clave del usuario existe en el registro envía la petición a otro servlet para atender la petición y si no envía la petición a una página html para que se registre.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class ValidadorServlet extends HttpServlet {
private RequestDispatcher rd;
public ValidadorServlet() {
}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Aquí solo se necesita de la clave del usuario, pero el objeto request
// contiene más parámetros que serán enviados, cuando se envíe la // petición, a
otro recurso web.
String clave = request.getParameter("clave");
if(Registro.estaRegistrado(clave)) {
// Si la clave existe en el registro, se obtiene una referencia del
// servlet servicio y se manda la petición a través de forward,
// terminando así la actuación del servlet ValidadorServlet.
rd = this.getServletContext().getNamedDispatcher("servicio");
rd.forward(request, response);
}
else {
// Si la clave no se encuentra en el registro, se envía la petición a // una página html para
que el usuario se registre, pasando así el // control a la página html *
rd = request.getRequestDispatcher("/registro.htm");
rd.forward(request, response);
}
}
}
```

*Cuando se envía la petición a un recurso web, el objeto request lleva ciertos parámetros. En el caso del servlet ValidadorServlet requiere del objeto request para obtener el parámetro clave para validarla en un registro.

En el caso de la llamada a la página los parámetros enviados no sonnecesarios. Existe un método de la clase HttpServletResponse llamado sendRedirect(String ruta), que permite llamar a otro recurso sin enviarle parámetros. Para llamar a la pagina 'registro.html' se hace de la siguientemanera. response.sendRedirect("/registro.html");Utilizando el método include()

Dando seguimiento al ejemplo descrito anterior mente para explicar el funcionamiento del método include(), ahora vamos a estudiar parte delcódigo más relevante para entender como utilizar este método.

1.- El servlet principal se encarga de llamar a los servlets especializados en una sola función, en este caso especializados en mostrar los precios de una solo marca de autos

```
public class PreciosAutosServlet extends HttpServlet {
// Se establece una variable para cada objeto RequestDispatcher de cada servlet
private RequestDispatcher rdJeep;
private RequestDispatcher rdFord;
private PrintWriter salida;
….
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
salida = res.getWriter();
response.setContentType("text/html");
// Se inicia la salida del servlet que llamará a los demás servlets
salida.println("<HTML><BODY>");
salida.println("<H1>Precios de autos<H1>");
salida.println("<H3> Autos Jeep</H3>");
// Se obtiene la referencia del servlet JeepServlet
rdJeep = this.getServletContext().getNamedDispatcher("jeep");
// Se llama al método include del objeto RequestDispatcher específico para el
// servlet jeep
rdJeep.include(request,response);
salida.println("<H3> Autos Ford</H3>");
// Se obtiene la referencia del servlet FordServlet;
rdFord = this.getServletContext().getNamedDispatcher("ford");
// Se llama al método include del objeto RequestDispatcher específico para el
// servlet ford
rdFord.include(request,response);
salida.println("</BODY></HTML>");
}
}
public class JeepServlet extends HttpServlet {
priavate PrintWriter salida;
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
// Del atributo response se obtiene un flujo de salida al navegador
salida = response.getWriter();
// El salida de este servlet no es necesario incluir los tags de html de inicio, porque // ya
fueron enviados al navegador por el servlet PreciosAutosServlet,. Aquí solo //
incluimos la parte de los precios de la marca
salida.println("<TABLE>")
salida.println("<TR><TD> Nombre</TD><TD>Precio</TD></TR>");
salida.println("<TR><TD>Grand Cherokee</TD><TD>$350,000.00</TD>");
salida.println("<TR><TD>Jeep Liberty</TD><TD>$350,000.00</TD>");
salida.println("</TABLE>");
}
}
```

Seguimiento de Sesión

Dentro de la programación web definiremos el concepto de sesión como las peticiones hechas por un cliente al servidor durante un cierto periodo de tiempo. Administrar y manejar esas peticiones puede ser muy util para dar una mayor versatilidad a nuestras aplicaciones. Por

ejemplo si se tratase de una tienda electrónica, podemos conservar un historial de los artículos que ha revisado el cliente para así sugerirle algunos artículos que puedan ser de su preferencia. Otra función importante de el seguimiento de sesión es manejar los clásicos carritos de compras donde los clientes agregan los productos que desean comprar. Existen tres mecanismos para crear y administrar sesiones en aplicaciones web basadas en Java:

- *Sobreescritura de URL*
- *Cookies*
- *Uso de HttpSession*

Sobreescritura de URL

Si utilizamos esta forma de administrar la sesión el servidor asignará un identificador de la sesión el cuál deberá de ser reenviado en cada petición que el cliente realice. El establecimiento de este identificador no es transparente por lo que para utilizar este método habría que implementar alguna forma de crear el identificador de la sesión para luego enviarlo al cliente y posteriormente al servidor. Como podemos darnos cuenta este método aunque no es complicado, nos exige crear código para el manejo de sesión que con los otros métodos no es necesario. Es principalmente por esta cuestión por lo que no se considera recomendable su uso.

Cookies

Las cookies son archivos de texto que el servidor envía al cliente y este los almacena y los reenvía al servidor en cada petición. Esta tecnología fue introducida inicialmente por Netscape.

Una cookie es identificada mediante un Nombre y un Valor para el nombre. Un Nombre es simplemente el nombre del identificador de sesión y el Valor es el respectivo valor que identificara una sesión. Existen otros atributos de una cookie como son:

Max-age .- El tiempo(en segundos) que residirá la cookie en la máquina del cliente

Domain.- Si se indica este atributo las cookies instaladas en el cliente regresaran dominio indicado por este atributo.

Path .- Es el directorio en el cual se va a regresar la cookie.

Comment .- Sirve para añadir comenterarios a la cookie.

Secure.- Esta variable boleana indica si se ha habilitado seguridad en la cookie

Para crear una cookie en java utilizaremos la clase javax.servlet.http.Cookie y el proceso es el siguiente:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws IOException, ServletException
{
. .
Cookie c = new Cookie(id , “0001”) ;
. .
c.Max-age(60*60*24);
c.setDoamin(“100.100.100.202”);
c.sethPath(“/”);
. .
response.addCookie(c);
. .
}
```

La ventaja en el uso de las cookies es que son regresadas automáticamente al servidor por lo que ya no hay que pensar en ello como cuando usamos Sobreescritura de URL .

Otra ventaja es su duración ya que pueden estar guardadas por un lapso de tiempo en las máquinas clientes por lo cual se pueden recuperar en futuras sesiones.

Uso de HttpSession

La interfaz javax.servlet.http.HttpSession define métodos que se encargan de encapsular y delimitar una Sesión. Su uso es muy práctico puesto que se hace transparente el manejo de sesiones, no teniendo que preocuparnos por establecer y obtener cookies o bien sobrescribir URL's. Para crear una sesión usaremos el método HttpSession getSession() o bien el método HttpSession getSession(bolean create) ambos definidos en la interfaz HttpServletRequest. Los métodos de la interfaz javax.servlet.http.HttpSession se pueden dividir en dos grandes grupos, el primero que define métodos para manejar la sesión y el segundo que define métodos para crear atributos que sean validos durante toda la sesión. Primero revisaremos a detalle los métodos para conocer las propiedades de la sesión:

public String getId()

Devuelve el identificador único de la sesión.

public long getCreationTime()

Este método devuelve en milisegundos el tiempo en que la sesión fue creada.

public long getLastAccessedTime()

Este método en milisegundos la última vez en que fue accesada la sesión.

public int getMaxInactiveTime()

Este método regresa el máximo de tiempo en segundos, que puede estar inactiva la sesión.

public int setMaxInactiveTime(int interval)

Este método permite establecer el máximo de tiempo que puede estar inactiva la sesión.

Debemos recordar que también se puede establecer el máximo tiempo de inactividad usando el tag <session-timeout>, el cuál revisamos en la sección de configuración.

public bolean isNew()

Retorna una valor true cuando una sesión es creada y accesada por primera vez.

public void invalidate()

Este método nos permite terminar una sesión antes de que se cumpla el intervalo máximo de tiempo señalado.

Ahora revisaremos los métodos para establecer atributos:

public object getAttribute(String name)

Este método se encarga de devolver el atributo indicado por el parámetro name.

public Enumeration getAttributeNames()

Este método regresa un objeto de tipo Enumeration el cual representa una lista que contiene los nombres de los atributos

public setAttribute(String name, Object o)

Este Método agrega un atributo a la sesión. El nombre del Atributo es el valor enviado en el argumento name.

public removeAttribute(String name)

Este método lo utilizaremos para eliminar un atributo de la sesión. Las aplicaciones del uso de atributos son muchas ya que nos permiten administrar datos temporales como los carritos de compras en una tienda electrónica, productos consultados por sesión, número de articulos visitados, etcétera.

HttpSession utiliza por default cookies para rastrear sesiones, es decir mplementa un mecanismo para crear y obtener cookies. Nosotros solamente tenemos que preocuparnos por determinar en que momento será invalida la sesión.

El problema surge cuando los navegadores no aceptan cookies, ya que la aplicación no podrá

mantener el estado de la sesión y por lo tanto cada vez que el cliente realice una petición se creará una nueva sesión. Para solucionar esto el API de Servlets ofrece una implementación del método de sobreescitura de URL. Para poder utilizar este método tendremos que utilizar el método encodeURL(String url) de la interfaz HttpResponse para cada URL esto es con la finalidad de conservar la sesión y reenviar el identificador de la sesión en cada petición.
El método encodeURL(String url) se usa de la siguiente manera:

```
//Se crea una cadena para contener al url
String url = response.encodeURL("/tienda/servlet/confirmacion");
//Después insertamos la cadena generada en el código HTML que desplegará
//el browser
out.println("<p><A HREF=\"" + url + "\">Regresar</A></p>");
```

3.4 JSP

JavaServer Pages (**JSP**), en el campo de la informática, es una tecnología para crear aplicaciones web. Es un desarrollo de la compañía Sun Microsystems y su funcionamiento se basa en scripts, que utilizan una variante del lenguaje java.

*La **JSP** es una tecnología Java que permite a los programadores generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Las JSP's permiten al código Java y a algunas acciones predefinidas ser incrustadas en el contenido estático del documento web.*

En las **JSP** se escribe el texto que va a ser devuelto en la salida (normalmente, código HTML) incluyendo código java dentro de él, para poder modificar o generar contenido dinámicamente. El código java se incluye dentro de las marcas de etiqueta <% y %>; a esto se le denomina *scriptlet*.

Un JSP es un archivo que combina HTML(o XML) con nuevos tags propios. La finalidad de esta nueva herramienta es separar el contenido Web Estático del dinámico, en pocas palabras separar la presentación de los datos en una hoja web de la lógica de la Aplicación.
Los JSP's se manejan internamente como Servlets, por lo que para cada jsp el Contenedor de JSPcreará un archivo *.class el cual va a contener la clase del Servlet que fue definido por el JSP.

3.4.1 FUNDAMENTOS

En una posterior especificación, se incluyeron taglib; esto es, la posibilidad de definir etiquetas nuevas que ejecuten código de clases java. La asociación de las etiquetas con las clases java se declara en archivos de configuración en XML.

La principal ventaja de **JSP** frente a otros lenguajes es que permite integrarse con clases Java (.class) lo que permite separar en niveles las aplicaciones web, almacenando en clases java las partes que consumen más recursos (así como las que requieren más seguridad) y dejando la parte encargada de formatear el documento html en el archivo jsp. La idea fundamental detrás de este criterio es el de separar la lógica del negocio de la presentación de la información.

Independientemente de la certeza de la aseveración, Java es conocido por ser un lenguaje muy portable (su lema publicitario reza: *escríbelo una vez, córrelo donde sea*) y sumado a las capacidades de JSP se hace una combinación muy atractiva.

Sin embargo, **JSP** no se puede considerar un script al 100%, ya que, antes de ejecutarse, el servidor web compila el script y genera un servlet. Por lo tanto, se puede decir que aunque este proceso sea transparente para el programador no deja de ser una aplicación compilada.

La ventaja de ello es algo más de rapidez y disponer del API de Java en su totalidad. Por todo ello, la tecnología JSP, así como Java, está teniendo mucho peso en el desarrollo web profesional (sobre todo en intranets).

Microsoft, la más directa competencia de Sun, ha visto en esta estrategia de Sun una amenaza, lo que le ha llevado a que su plataforma .NET incluya su lenguaje de scripts ASP.NET que permite ser integrado con clases .NET (ya estén hechas en C++, VisualBasic o C#) del mismo modo que jsp se integra con clases Java.

Las siguientes líneas muestran nuestro primer JSP, el cuál se encargará de mostrar la fecha actual:

```
<%@page import="java.util.Date"%>
<html>
<body>
Hoy estamos a : <br> <%= new Date().toString() %>
</body>
</html>
```

Objetos Implícitos

Al crear un Servlet comúnmente utilizamos objetos que implementan las interfaces: HttpServletRequest, HttpServletResponse, HttpSession, etc... Estos objetos nos servían para acceder a distintos servicios de un Servlet, por ejemplo un objeto del tipo HttpServletRequest representaba los datos enviados por Http desde el cliente, este objeto posee métodos como *getParameter()* y *getHeader* .
Un JSP al estar basado en un Servlet también puede usar objetos que implementen dichas interfaces, con la gran diferencia que en JSP los objetos ya se encuentran creados mientras que en los Servlets usualmente los obteníamos mediante un método . Los objetos que se usan en JSP se muestran en la siguiente tabla:

Request	Este objeto obtiene información acerca de la petición del cliente realizada mediante GET/POST. Este objeto implementa la interfaz HttpServletRequest.
Response	Representa los datos enviados al cliente como respuesta. El contiene un flujo de salida que sirve para colocar texto e imágenes.
PageContext	Este objeto permite acceder a una serie importantes de atributos de la página. Como el objeto response, request, session, etc.

Session	Una session es creada para escuchar las peticiones de un cliente, este objeto implementa la interfa HttpservletHttp._Session.
Application	El objeto se encargará de la configuración
Out	Representa el flujo de salida del objeto HttpResponse el texto enviará la salida al monitor que se encuentre instalado.
Config	Este objeto es del tipo ServletConfig. Los métodos de este objeto solo podrán aplicarse a la página page Este objeto puede ser usado como si fuese this, ya que representa a la misma pagina

Los tags propios de los JSP se dividen en cuatro grandes grupos:

- Directivas. Son mensajes que el JSP envía al contenedor JSP. Se usan más que nada para establecer valores globales que afecten a todo el JSP.
- Elementos de Script. Estos tags permiten insertar fragmentos de código Java.
- Acciones. Estos tags se encargan de afectar el entorno de ejecución del JSP.
- Tags propios. Como su nombre lo dice, son tags elaborados por unomismo; más adelante veremos como crearlos.

Directivas

Una Directiva se inicia colocando un *"<%@"* y se termina con un *"%>"* y su sintaxis presenta el siguiente formato:

```
<%@ nombre de directiva atributo="valor" atributo="valor" . . .%>
```

Existen tres principales directivas que son:
Page, include,taglib.

La directiva page

Esta directiva sirve para configurar las características del documento JSP.
Cuenta con los siguientes atributos:

Language	Permite definir el leguaje a usar en el JSP esto es para dar posibilidad a que futuras versiones de JSP utilicen múltiples lenguajes

Extends	Con este atributo podemos indicar la clase que se usara como Servlet, esta clase debe de cumplir con ciertas reglas, por lo que hay que ser muy cuidadoso al usar este atributo puesto que si le indicamos un valor incorrecto el funcionamiento del JSP también será incorrecto.

Import	En este atributo señalaremos todos los paquetes utilice el JSP.

Session	Mediante un valor true/false indicaremos si el JSP participará en una sesión HTTP.
Buffer	Permite especificar el buffer del flujo de salida al cliente.
AutoFlush	Si a este atributo esta asignado el valor “true” el buffer del flujo de salida al cliente se limpiará automáticamente.
Info	Si indicamos algún valor en este atributo, este será retornado por el método Servlet.getServletInfo().
ErrorPage	Aquí podemos indicar un URL a otro JSP el cuál se mostrará en caso de que ocurra una excepción en el JSP.
IsErrorPage	Si este atributo tiene el valor “true” entonces dispondremos de una variable llamada *exception* la cuál extiende de java.lang.Throwable.
ContentType	Define la codificación de los caracteres que aparecen en la página JSP, el valor por default es “text/html”.

Ejemplo:
Para probar esta directiva utilizaremos el documento page.jsp, en el veremos la directiva *page* al inicio del texto, se asignaron valores a varios de sus atributos y después se insertaron tags de HTML comunes y corrientes. Utilizaremos la directiva page en conjunto con otras directivas, acciones, y tags propios.

La directiva include

La directiva include permite insertar el contenido de un archivo al JSP. El archivo se insertará justo en el lugar donde se coloco la directiva. El archivo a insertar debe ser un archivo que se encuentre en el contenedor de JSP. Los archivos a insertar pueden ser páginas HTML o bien otros JSP’s. Al insertar un archivo usando la directiva include si este cambia durante la ejecución el JSP conservará el archivo original, así que se recomienda usar esta directiva solo cuando el archivo no vaya a ser modificado constantemente.
La sintaxis para la directiva include es la siguiente:

```
<%@ include file=”/miarchivo.html” %>
```

El único atributo que posee esta directiva es *file* y en el señalaremos el URL del archivo que deseamos insertar en el JSP.

Ejemplo:
El código que se muestra a continuación pertenece al archivo info.html; el cuál no es más que un documento HTML que contiene una imagen y que será insertado mediante la directiva include en un jsp.

```
<html>
<head>
<title>INFO</title>
</head>
```

```
<body>
<img SRC="BD00372_.GIF" BORDER=0 height=42 width=796>
</body>
</html>
```

Ahora mostraremos el archivo JSP dirinclude.jsp el cual se encarga de incrustar el código de la página HTML, llamada info.html.

```
<html>
<head>
<title>Esta página probará la directiva :include</title>
</head>
<body>
<h1>A partir de la siguiente línea se insertara el archivo info.html</h1>
<%@ include file="info.html" %>
<h1>Y aqui prosigue el contenido del JSP dirinclude.jsp</h1>
</body>
</html>
```

Ahora veremos un último ejemplo, en donde insertaremos el archivo dirinclude.jsp dentro de otro archivo jsp, en este caso el archivo archivo dirincludeJSP.jsp

```
<html>
<head>
<title>Esta página probará la directiva :include</title>
</head>
<body>
<h1>A partir de la siguiente línea se insertara el archivo dirinclude.jsp</h1>
<%@ include file="dirinclude.jsp" %>
<h1>Y aqui prosigue el contenido del JSP dirincludeJSP.jsp</h1>
</body>
</html>
```

La directiva taglib

Esta directiva sirve para indicar que se usaran una librería de tags propios (tags extensions ó custom tags) como crear y usar esta herramienta se verá más adelante. La sintaxis de esta directiva es la siguiente:

```
<%@ taglib uri= ”URIde la taglib” prefix = “tagPrefix“ %>
```

Donde el atributo *uri(Identificador Uniforme de Recurso)* sirve para indicar el descriptor de la librería de tags. Mientras que el atributo *prefix* indicara el prefijo que deberán llevar todos nuestros tags. Por ejemplo si en el atributo *prefix* colocamos el valor “MiLibreria” al querer usar un tag llamado *“mi Tag”* el código se verá de la siguiente manera:
<MiLibreria:MiTag...../>

Elementos de Script

Estos elementos nos sirven para insertar código Java en un JSP. Y se dividen en tres tipos:
Declaraciones
Sriptlets
Expresiones

Scriptlets

Un elemento de este tipo es simplemente un boque de código Java. En un Scriptlets podemos modificar objetos, invocar métodos y asignar valores a variables. Para desplegar un mensaje en el navegador del cliente usaremos el objeto implícito *out* y el método *println* tal y como lo hacemos para un Servlet. La sintaxis de un Scriptlet es la siguiente:

```
<% Código Java%>
```

Ejemplo:
A continuación mostraremos un jsp llamado pregunta.jsp , el JSP recibirá el parámetro nombre del archivo pregunta.html. Primero mostraremos el archivo preguta.html

```
<html>
<head>
<title>Una pequeña pregunta</title>
</head>
<body>
<h1>¿Quién fue el primer Virrey de la Nueva España?</h1>
<form method="post" action="pregunta.jsp">
<p>Tu respuesta:
<input type="text" name="nombre">
</p>
<p><input type="submit" value="Verificar">
</form>
</body>
</html>
```

Y ahora vemos el archivo pregunta.jsp. En este archivo podremos observar como un jsp se comporta como un Servlet y como ya se encuentra implementado el objeto request de tipo HttpServletRequest.

```
<html>
<head>
<title>Página de prueba de los scrptlets</title>
</head>
<body>
<%
String nombre = request.getParameter("nombre");
if((nombre.toLowerCase()).equals("antonio de mendoza"))
out.println("<BR>Muy Bien Has Aceptado");
else
out.println("<BR>Que mal!, hay que saber algo de nuestra historia!");
%>
.</P>
</body>
</html>
```

Declaraciones

En este tipo de elemento colocaremos todas las variables y métodos que usaremos dentro del JSP. Esta expresión se ejecuta cuando se inicia el JSP así que todas la variable que fueron establecidas podrán usarse durante toda la sesión.
La sintaxis para una declaración es:

```
<%! Declaraciones de variables o métodos %>
```

Ejemplo:
En el siguiente JSP mediante una declaración se creará una instancia de la clase String, también se creará una variable llamada repeticiones la cual es de tipo entero, posteriormente se escribe un método llamado obtener nombre de curso. Posteriormente serán usados mediante un Scriptlet el cual imprimirá el objeto de tipo String creado en la declaración el número de veces que indique
la variable repeticiones.

```
<%!
String nombrecurso = new String("Java para Web");
int repeticiones = 10;
public String obtenerNombrecurso(){
return nombrecurso;
}%>
<html>
<head>
<title>Página de prueba de las declaraciones</title>
</head>
<body>
<h1>Página de prueba de las declaraciones</h1>
<P>El numero de veces de repeticiones es 10.</P>
<P>
<%for(int i=0; i<repeticiones; i++){
out.println("<BR>");
out.println("Estas en el curso de : " + obtenerNombrecurso());
}%>
.</P>
</body>
</html>
```

Expresiones.

Una expresión nos sirve para enviar una respuesta al cliente, en este caso el navegador. El valor retornado es convertido en String para poder ser desplegado. Una expresión tiene la siguiente sintaxis:

```
<%= Expresión Java a Evaluar%>
```

Ejemplo:

El archivo expresión.jsp muestra como utilizar una expresión en JSP, la expresión se encargará de imprimir el la página un contador de visitas.

```
<head>
<title>Prueba de un expresión</title>
</head>
<body>
<h1>BIENVENIDO A MI PAGINA</h1>
<%! int contador=0 ; %>
<%contador++;%>
<%= "Esta Página a sido visitada: " + contador + " veces" %>
</body>
</html>
```

Acciones Estandar

Las Acciones Estandar son tags que afectan el comportamiento de ejecución del JSP. Estos tags son remplazados por código Java durante el paso de el JSP a un Servlet.

Los tipos de acciones estandar son los siguientes:

<jsp:useBean>
<jsp:setProperty>
<jsp:getProperty>
<jsp:param>
<jsp:include>
<jsp:forward>
<jsp:plugin>

<jsp:useBean>

Mediante esta acción y las acciones <jsp:setProperty>, <jsp:getProperty> podemos instanciar y manejar en nuestros JSP's, componentes Java Beans. Para crear una instancia de un Java

Bean usaremos la acción *<jsp:useBean>* la cual se encargará de localizar al Bean, así como determinar el alcance del Bean. La sintaxis para esta acción es:

```
<jsp:useBean id=”nombre” scope=”nombredeAlcance” detalles />
```

Hay cuatro combinaciones de detalles seleccionaremos una de ellas dependiendo de:

• class = “Nombre de la clase”
• class = “Nombre de la Clase” type = “Nombre del Tipo”
• class = “Nombre del Bean” type=” Nombre del Tipo”
• type=” Nombre del Tipo”

En la siguiente tabla describiremos para que sirven cada uno de los atributos de la acción <jsp:useBean>

id	El valor que se coloque en este atributo servirá para identificar la instancia del Java Bean dentro de la página JSP.
Scope	Este atributo se refiere al alcance de la instancia del Java Bean. Esto delimitirá el uso de la rerencia al Java Bean. Los valores para este atributo son: • page.- Al indicar este valor la instancia del Java Bean solo será valida para el documento JSP desde el cuál se creo. • request.- La instancia del Java Bean podrá ser utilizada en las páginas html y documentos JSP invocados mediante las acciones <jsp:include> y <jsp:forward> • session.- El objeto estará disponible en todas las peticiones hechas desde un mismo cliente • application.- La instancia del Java Bean estará disponible en cualquier JSP de la aplicación
Class	La clase del componente Java Bean
BeanName	El nombre del Bean se colocará dentro de este atributo la combinación de este atributo y el atributo
type	Pueden suplir al atributo class type Aquí señalaremos el tipo del atributo por ejemplo si deseamos usar una superclase del Java Bean o bien una interfaz que este implemente.

En caso de que algun parámetro este incorrecto y no pueda crearse una instancia del Java Bean se arrojará una excepción de tipo: InstantiationException <jsp:setProperty>
Esta acción se encargará de establecer valores para las propiedades de los Java Beans instanciados mediante la acción <jsp:useBean >
La sintaxis para esta acción es la siguiente:
<jsp:setProperty name=”NombredelJavaBean” detallesdePropiedad />
Esta acción utiliza el soporte de introspección de los Java Beans, gracias a ello asigna valores a las propiedades del Java Bean relacionando los nombres de los parámetros que recibe el JSP con los nombres de las propiedades del Java Bean , asignándoles valores sin necesidad de indicar la propiedad. Este solo es un mecanismo de hacerlo ya que también se pueden establecer valores señalando el nombre de la propiedad o el nombre del parámetro. Gracias a esto la sección detallesdePropiedad puede tener las siguientes combinaciones:
• property=’*’
• property=”NombredePropiedad”
• property=”NombredePropiedad” param=”NombredeParámetro”
• property=”NombredePropiedad” value=”Valor de Propiedad”
La siguiente tabla muestra una breve descripción de cada uno de los atributos
de la acción <jsp:setProperty>

name	Aquí colocaremos el nombre de la instancia de un Java Bean. Este Java
Bean	debe de haber sido instanciado mediante la acción <jsp:useBean>
property	El nombre de la propieadad a la cuál deseamos establecer un nuevo valor.
Param	En este atributo colocaremos el nombre del parámetro del cuáldeseamos obtener un valor para una propiedad
value	Con este atributo indicaremos directamente el valor de una propiedad

<jsp:getProperty>

Esta acción sirve para obtener el valor de la propiedad de un Java Bean. La acción extraerá el valor de la propiedad y luego lo convertirá a cadena nviándola al flujo de salida del explorador web.Si la propiedad es una clase para convertirla en String usará el método toString(). Si la propiedad es un tipo básico utilizará el método valueOf() de la clase correspondiente a dicho tipo básico.Su sintaxis es la siguiente:

```
<jsp:getProperty name=”NombredelJavaBean” property=”NombrePropiedad” />
```

Donde *name* es el nombre del Java Bean y *property* es el nombre de la propiedad que deseamos conocer. Ejemplo (uso de *<jsp:useBean>, <jsp:setProperty>, <jsp:getProperty>*)

El siguiente ejemplo utiliza el JavaBean EmailBean para guardar mensajes en un buzón de correo. Este buzon no es más que una tabla de una base de datos.
Para Acceder al JSP utilizaremos un archivo HTML llamado email.html en las siguientes líneas se muestra el código de este documento:

```
<html>
<head>
<title>Formulario de Correo</title>
</head>
<body>
<form method="post" action="email.jsp">
nombre
<input type="text" name="nombre">
<br>remitente:
<input type="text" name="remitente">
<br>direccion:
<input type="text" name="direccion">
<br>mensaje:
<textarea name="mensaje" rows="7" cols="20">
</textarea>
<br>
<input type="submit" value="Enviar">
</form>
</body>
</html>
```

El código para el archivo jsp es:

```
<jsp:useBean id="EmailBean" scope="session" class="email.EmailBean">
</jsp:useBean>
<html>
<head>
<title>El correo ha sido enviado satisfactoriamente</title>
</head>
<body>
<jsp:setProperty name="EmailBean" property="*"/>
<%EmailBean.enviarEmail();%>
<p>El correo ha sido recibido en el buzon indicado.</p>
</body>
</html>
```

Y por último mostraremos el código para el JavaBean EmailBean:

```
package email;
import java.sql.*;
public class EmailBean {
public String nombre;
public String remitente;
public String direccion;
public String mensaje;
public EmailBean() {}
public void setNombre(String nombre) {
this.nombre = nombre;
}
public void setRemitente(String remitente) {
this.remitente = remitente;
}
public void setDireccion(String direccion) {
this.direccion = direccion;
}
```

```
public void setMensaje(String mensaje) {
this.mensaje = mensaje;
}
public String getNombre() {

return nombre;
}
public String getRemitente() {
return remitente;
}
public String getDireccion() {
return direccion;
}
public String getMensaje() {
return mensaje;
}
public void enviarEmail() {
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conex=DriverManager.getConnection("jdbc:odbc:Email");
String insert = "insert into buzon values ( ? , ? , ? , ? )";
PreparedStatement orden = conex.prepareStatement(insert);
orden.setString(1, nombre);
orden.setString(2, remitente);
orden.setString(3, direccion);
orden.setString(4, mensaje);
orden.executeUpdate();
orden.close();
conex.close();
}catch(Exception ex){
ex.printStackTrace();
}
    }
}
```

<jsp:param>

Esta acción sirve para enviar parámetros a otros documentos JSP, se utiliza en conjunto con las acciones *<jsp:include>*, *<jsp:forward>* y *<jsp:plugin>*.
La sintaxis para esta acción es la siguiente es: *<jsp:param name="NombreDeParametro" value="ValorDeParametro" />* En donde *name* representa el nombre del parámetro y *value* el valor de este. Veremos algunos ejemplos de *<jsp:param>* al usar las acciones *<jsp:include>*, *<jsp:forward>* y *<jsp:plugin>*.

<jsp:include>

La acción *include* trabaja de manera muy similar a la directiva *include* solamente con la gran diferencia de que esta acción funciona de manera dinámica, a diferencia de la directiva que lo hace solamente de manera estática.

El proceso para insertar un archivo usando la directiva include es que al momento de la compilación se inserta el archivo en el servlet generado. Mientras que usando la acción include el archivo se inserta al momento en que se hace la primera petición al JSP.
Se sugiere usar la directiva cuando se conoce que el contenido incluido no cambiará y se recomienda usar la acción cuando el recurso que se incluyo sufrirá muchas modificaciones.
La sintaxis para esta acción es:
<jsp:include page="URL" flush="true" />
Si se desea enviar parámetros al documento insertado se escribirá de la siguiente manera:
<jsp:include page="URL" flush="true" >
<jsp:param name="NombreDeParametro"

value="ValorDeParametro">
...
</jsp:include>

En la cual *page* es el atributo en el cuál indicaremos la página HTML o JSP a insertar y el atributo *flush* sirve para indicar que antes de insertar el archivo se limpie el buffer en el flujo de salida.

Ejemplo de *<jsp:include>*

Mezclaremos el ejemplo de la acción include con el de la directiva include para ver sus diferencias y semejanzas. Para ello utilizaremos los archivos includeaction.jsp, sinónimos.html y sinónimos.jsp(este último también utiliza una acción include para incluir dentro de el al archivo sinónimos.html). includeaction.jsp invocara mediante el uso de la directiva include a los archivos sinónimos.html y sinónimos.jsp y luego usando la acción include incluirá a los mismos dos archivos.

A continuación veremos el código fuente del documento includeaction.jsp:

```
<html>
<head>
<title>Acción: include</title>
</head>
<body>
<h1>En esta página se compará el uso de la directiva include con el uso de la acción</h1>
<h2>Usando la directiva</h2>
<%@ include file="sinonimos.html" %>
<hr size="2" width="100%">
<%@ include file="sinonimos.jsp" %>
<hr size="7" width="100%">
<h2>Usando la acción</h2>
<jsp:include page="sinonimos.html" flush="true" />
<hr size="2" width="100%">
<jsp:include page="sinonimos.jsp" flush="true">
<jsp:param name="miParametro" value="usando parametros" />
</jsp:include>
<hr size="7" width="100%">
<p>
Ahora modificará los archivos referenciados por la acción
y verá un sorprendente resultado
</p>
<p>
Esto es gracias a que la acción carga dinamicamente los archivos
mientras que la directiva los carga en el momento de la compilación.
</p>
</body>
</html>
```

Como podemos observar el resultado es que aparece la misma imagen cuatro veces, Dos que fueron incluidas por la directiva include y dos que fueron incluidas por la acción include.

A continuación haremos un cambio en el archivo sinonimos.jsp para demostrar como se pueden recibir parámetros. Para esto agregaremos la siguiente línea al archivo.

```
<%= request.getParameter("miParametro")%>
Archivo sinónimos.jsp
<html>
<head>
<title>Archivo Jsp que usa la accion include</title>
</head>
<body>
<p>ESTE ES EL ARCHIVO sinonimos.jsp</p>
<%= request.getParameter("miParametro")%>
```

```
<p>EL ARCHIVO QUE SE MUESTRA A CONTINUACION ES: sinomimos.html</p>
<jsp:include page="sinonimos.html" flush="true" />
</body>
</html>
```

Como se pudo observar, al ejecutar el cambio solo fue visible para el JSP invocado por la accion include, el invocado por la directiva include no cambia puesto que se compilo junto con el código del JSP

<jsp:forward>

Esta acción sirve para enviar peticiones a otro JSP o Servlet, al encontrar esta acción se suspenderá la ejecución del JSP, para poder enviar una petición al otro recurso.

Al igual que la acción *include* se pueden enviar parámetros al otro recurso. La sintaxis para esta acción cuando no se envía ningún parámetro es la siguiente:

```
<jsp:forward page="URL" />
<jsp:param name="NombreDeParametro" value="ValorDeParametro">
...
<jsp:forward page="URL">
</jsp:forward>
```

Donde el atributo *page* señalara el recurso al que nos va a enviar la acción *<jsp:forward>*

<jsp:plugin>

Esta acción sirve para poder utilizar el Java Plug-in del Explorador Cliente, con el fin de poder ejecutar Applets o Java Beans.

Aquí se usara de manera distinta el tag *<jsp:params>* a como se uso en *<jsp:include>* y *<jsp:forward>* ya que aquí los parámetros serán enviados al componente Java Bean o al Applet. Existe otro tag más llamado: *<jsp:fallback>* el cuál sirve para especificar el contenido que debe desplegarse en caso de que el plugin no pueda ser utilizado debido a que los tags generados no se encuentran soportados. La sintaxis de esta acción es la siguiente:

```
<jsp:plugin type="bean|applet" code="CódigodelObjeto" codebase="CodebasedelObjeto"
align="Alineación" archive="ListadeArchivos" heigth="Altura" hspace="Espacio H"
jreversion="Versión de JRE" name="Nombre del componente" vspace="Espacio V"
width="Ancho" nspluginurl="URL" iepluginurl="URL">
<jsp:params>
<jsp:param name="NombreDeParametro" value="ValorDeParametro">
...
</jsp:params>
<jsp:fallback>Aquí escribiremos el contenido que deseamos desplegar en caso de que no se
pueda cargar el plugin</jsp:fallback>
</jsp:plugin>
```

Extensiones de tags

Una librería de extensiones de tags permite añadir elementos de tipo acción a un documento JSP. Existen dos elementos importantes al definir una librería de tags los cuales son:

- La clase manejadora
- Y el descriptor de la librería de tags, el cual es un archivo XML con extensión tld

La clase manejadora nos servirá para ayudar a evaluar que acciones se realizará en el momento en el que el Tag sea ejecutado dentro de un documento JSP. Todas la clases

manejadoras heredan de la clase Tag Support. En caso de que nuestra acción necesite manejar el cuerpo del tag la clase manejadora heredará de la Clase BodyTagSupport

El descriptor de la librería de tags tiene como función definir todos los tags que vamos a crear para cada uno de estos tags se creará una clase manejadora. Con un custom tag se puede hacer lo mismo que con un scriptlet. La diferencia es que los scriptlets mezclan código con markups estáticos y los custom tags abstraen código mediante un markup que se asemeja a HTML. Los scriptlets frecuentente mezclan código para manipular datos y código para presentar los datos. Los custom tags mueven la lógica en clases Java que se vinculan mediante tags. La lógica puedemanipularse sin tocar la JSP,custom tags y los elementos de scripting proveen la misma funcionalidad.

URI Relativa

Es una URI sin protocolo ni host y es la manera más directa de informarle a la Aplicación Websobre la ubicación de un TLD. Consiste en especificar como valor del atributo URI de la directiva taglib, el path relativo a un archivo TLD.
Las URI's relativas, se dividen en dos:

- <%@ taglib uri="/WEB-INF/ejemplo.tld" prefix="test"%>
- <%@ taglib uri="ejemplo.tld" prefix="test"%>

Antes de comenzar a escribir nuestro descriptor debemos colocar el siguiente encabezado;

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

<tlibversion> La versión de la librería de tags esta es definida por el autor de la librería
<jspversion> Es la versión de la especificación de JSP que estamos utilizando al crear nuestra librería. Este elemento es opcional
<shortname> Esta etiqueta indicará el nombre de la librería que será usado dentro del archivo jsp. Al crear este nombre no debemos de incluir carateres subrayados y espacios.
<uri> Este valor será el identificador de recurso para la librería de tag's, este valor es opcional. **<info>** Este texto opcional describirá información relativa a la librería de tags. La siguiente etiqueta que definiremos será <tag> esta servira para definir un tag e irá dentro del cuerpo de <taglib>. Una librría de tags puede tener muchos tags. Además también contiene etiquetas internas que a continuación se describirán.
<name> Aquí indicaremos el nombre del tag. Cuando usemos el tag en un archivo JSP este nombre se escribirá después del nombre de la librería de tags.
<tagclass> Indicaremos el nombre de la clase manejadora del tag incluyendo los paquetes en los que este contenida.
<teiclass> Este tag es opcional y en el agregaremos clases que hereden de *javax.servlet.jsp.tagext.TagExtraInfo* estas clases proveerán al ambiente de ejecución de información extra acerca del tag. Especialmente cuando este usa valores variables.
<bodycontent> Este elemento opcional indicará el tipo de contenido del cuerpo del tag. Se pueden señalar tres tipos de contenidos distintos y estos son: *JSP*, *tagdependent* y *empty*. El valor *JSP* lo usaremos cuando se requiera que el contenido del cuerpo del tag sea evaluado por el motor de JSP's. El valor *tagdependent* se usará cuando no deseamos que el contenido sea evaluado por el motor de JSP's y por último el valor *empty* indicará que el tag no tendrá contenido en su cuerpo.

La etiqueta **<attribute>** ira dentro del cuerpo de la etiqueta <tag> y servirá para establecer un atributo del tag. Un tag puede tener muchos atributos. La etiqueta atributos también posee etiquetas dentro de su cuerpo las cuales señalaremos a continuación.
<name> Este elemento es requerido y en el colocaremos el nombre del atributo que nos encontramos creando.
<required> especifica si el atributo sera forzoso, si es así tendremos que colocar el valor true si no indicaremos el valor false. Si no escribimos este tag, ya que es opcional, el valor por default será true.
<rtexprvalue> Determina si el atributo será dinamico o bien se comportará de manerá estática.

El siguiente ejemplo muestra un archivo tld que define una librería de tags:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>busqueda</short-name>
<uri>busquedalib</uri>
<description>
Una librería que servirá para buscar discos en un sistema de información
</description>
<tag>
<name>busqueda</name>
<tag-class>busqueda.BusquedaBodyTag</tag-class>
<tei-class>busqueda.BusquedaBodyTagExtraInfo</tei-class>
<body-content>JSP</body-content>
<description> despliega datos generales de un artista, interprete o agrupacion</description>
<attribute>
<name>busqueda</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
</taglib>
```

3.5 BEANS EMPRESARIALES

La especificación EJB ha ido evolucionando a la par que lo hacía la propia especificación J2EE. Las diferentes versiones que han existido hasta la fecha son:

- EJB 1.0: la especificación original
- EJB 1.1: la primera incluida dentro de J2EE
- EJB 2.0: incluida en J2EE 1.3, añadía las interfaces Locales y los Message-Driven beans.
- EJB 2.1: incluida en la última revisión de J2EE, la 1.4.
- EJB 3.0: Ahora con Cluster y esta incluida en JEE 5.0

Existe una propuesta de una nueva revisión de la especificación de Enterprise JavaBeans, la EJB 3.0, cuyo objetivo es simplificar la implementación de beans sin por ello sacrificar su

potencia y flexibilidad.La nueva especificación de EJB 3.0 simplifica el proceso de creación de EJB y facilita la implementación de persistencia.Esta especificación estará disponible en la nueva versión de J2EE nombrada JEE 5.0.
Los **Enterprise JavaBeans** (también conocidos por sus siglas **EJB**) son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE de Sun Microsystems. Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor que son, precisamente, los EJBs:

- comunicación remota utilizando CORBA
- transacciones
- control de la concurrencia
- eventos utilizando JMS (Java messaging service)
- servicios de nombres y de directorio
- seguridad
- ubicación de componentes en un servidor de aplicaciones.

La especificación de Enterprise Java Bean define los papeles jugados por el contenedor de EJB y los EJBs, además de disponer los EJBs en un contenedor.

3.6 TIPOS DE BEANS EMPRESARIALES

Definición

Los EJBs proporcionan un modelo de componentes distribuido estándar del lado del servidor. El objetivo de los EJBs es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia seguridad, ...) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables.No hay que confundir los Enterprise JavaBeans con los JavaBeans. Los JavaBeans también son un modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la invocacion remota (RMI).

Tipos de Enterprise JavaBeans

Existen tres tipos de EJBs:

EJBs de Entidad (*Entity EJBs*): su objetivo es encapsular los objetos del lado del servidor que almacena los datos. Los EJBs de entidad presentan la característica fundamental de la persistencia:

-Persistencia gestionada por el contenedor (CMP): el contenedor se encarga de almacenar y recuperar los datos del objeto de entidad mediante el mapeo de una tabla de la base de datos.

-Persistencia gestionada por el bean (BMP): el propio objeto entidad se encarga, mediante una base de datos u otro mecanismo, de almacenar y recuperar los datos a los que se refiere, por lo cual, la responsabilidad de implementar los mecanismos de persistencia es del programador.

EJBs de Sesión (*Session EJBs*): gestionan el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos:

con estado (*stateful*). Los beans de sesión con estado son objetos distribuidos que poseen un estado. El estado no es persistente, pero el acceso al bean se limita a un solo cliente.
sin estado (*stateless*). Los beans de sesión sin estado son objetos distribuidos que carecen de estado asociado permitiendo por tanto que se los acceda concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método.

EJBs dirigidos por mensajes (*Message-driven EJBs*): son los únicos beans con funcionamiento asíncrono. Usando el *Java Messaging System* (JMS), se suscriben a un tema (*topic*) o a una cola (*queue*) y se activan al recibir un mensaje dirigido a dicho tema o cola. No requieren de su instanciación por parte del cliente.

Funcionamiento de un Enterprise JavaBean
Los EJBs se disponen en un contenedor EJB dentro del servidor de aplicaciones. La especificación describe cómo el EJB interactúa con su contenedor y cómo el código cliente interactúa con la combinacion del EJB y el contenedor.

Cada EJB debe facilitar una clase de implementación Java y dos interfaces Java. El contenedor EJB creará instancias de la clase de implementación Java para facilitar la implementación EJB. Las interfaces Java son utilizadas por el código cliente del EJB. Las dos interfaces, conocidas como interfaz "home" e interfaz remoto, especifican las firmas de los métodos remotos del EJB. Los métodos remotos se dividen en dos grupos:

- métodos que no están ligados a una instancia específica, por ejemplo aquellos utilizados para crear una instancia EJB o para encontrar una entidad EJB existente. Estos métodos se declaran en la interfaz "home".

- métodos ligados a una instancia específica. Se ubican en la interfaz remota.

Dado que se trata simplemente de interfaces Java y no de clases concretas, el contenedor EJB genera clases para esas interfaces que actuarán como un proxy en el cliente. el cliente invoca un método en los proxies generados que a su vez sitúa los argumentos método en un mensaje y envía dicho mensaje al servidor EJB. Los proxies usan RMI-IIOP para comunicarse con el servidor EJB.El servidor llamará a un método correspondiente a una instancia de la clase de implementación Java para manejar la llamada del método remoto.

Interfaz ”home”
Como indicamos anteriormente, la interfaz "home" permite al código cliente manipular ciertos métodos de clase del EJB, esto es, métodos que no están asociados a ninguna instancia particular.La especificación EJB 1.1 establece el tipo de métodos de clase que se pueden definir como métodos que crean un EJB o para encontrar un EJB existente si es un "bean" de entidad.La especificación EJB 2.0 permite a los desarrolladores de aplicaciones definir nuevos métodos de clase sin limitarse a su sola creación, borrado y búsqueda..

Interfaz remota
La interfaz remota especifica los métodos de instancia públicos encargados de realizar las operaciones.

Clase de implementación EJB

Las clases de implementación EJB las suministran los desarrolladores de aplicaciones, que facilitan la lógica de negocio ("business logic") o mantienen los datos ("business data") de la interfaz de objeto, esto es, implementan todos los métodos especificados por la interfaz remota y, posiblemente, algunos de los especificados por la interfaz "home".

Correspondencia entre métodos de interfaz y métodos de implementación

Las llamadas al método en la interfaz "home" se remiten al método correspondiente de la clase de implementación del bean con el prefijo 'ejb' añadido y con la primera letra de la interfaz "home" convertida en mayúscula y manteniendo exactamente el mismo tipo de argumentos. Por ejemplo:

create ---> ejbCreate

Las llamadas a métodos en la interfaz remota se remiten al método de implementación correspondiente del mismo nombre y argumentos

Ejemplo de un bean de sesion

En este caso utilizaremos Jboss como servidor de aplicaciones, y net beans como ide tendremos que escribir la clase del bean, el cual contendrá todos los métodos del negocio Y este bean cada vez que lo ejecutemos nos desplegara un pensamiento de Albert Einstein.

1.- Escribir dos interfaces para el bean: ***home y component***.

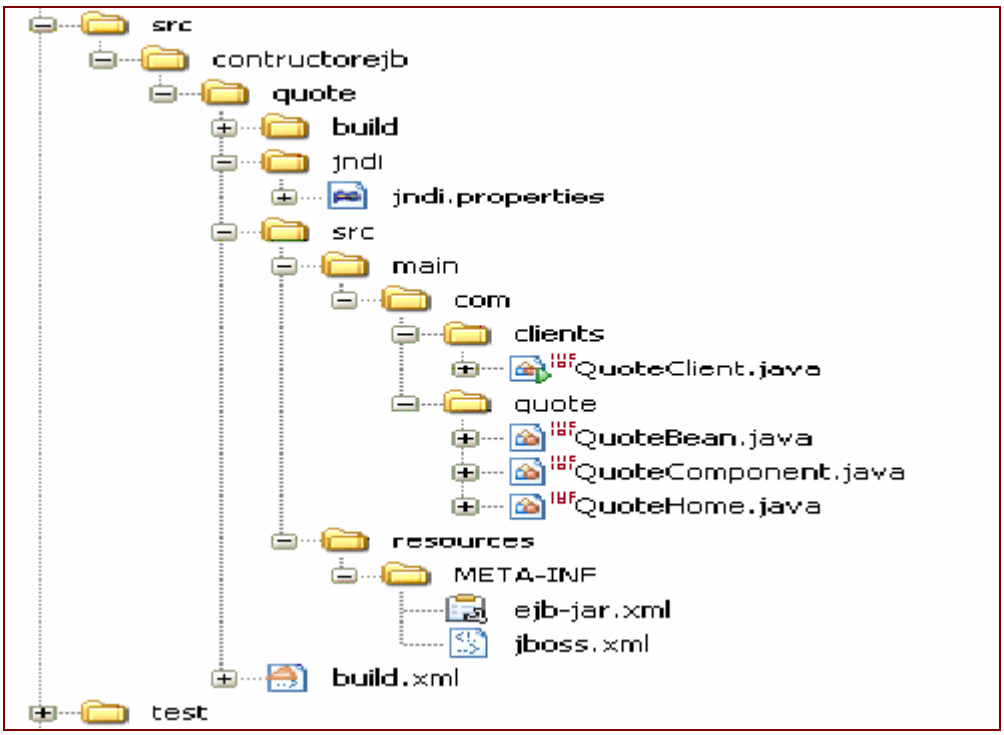
2.- Una interface home, que funciona como factoría de objetos que implementan la interface EJBObject.

3.- La interface component hereda de la interface EJBObject, y a través de los objetos que implementen la interface component es posible comunicarse con el enterprise bean.

4.- Crear un archivo XML de configuración llamado **deployment descriptor**, éste archivo le indica al servidor de que tipo es el bean y como debe ser manejado. El nombre de este archivo debe ser **ejb-jar.xml**.

5.- Empaquetar el bean, las interfaces y el descriptor de despliegue en un archivo JAR mejor conocido como **ejb-jar**, el cual podemos nombrar como sea.

6.- Desplegar el enterprise bean en el servidor. La forma de desplegar el enterprise bean depende de cada vendedor.



- La clase del bean contiene los métodos del negocio, que un cliente puede llamar, además de implementar unos métodos adicionales que controlan su ciclo de vida. De acuerdo a tipo de bean que queramos desarrollar es la interfaz que debemos implementar.

  QuoteBean.java

```
package constructorjb.quote.src.main.com.quote;
import javax.ejb.*;
public class quoteBean implements SessionBean{
        private String []quotes = {
                "Todo hay que reducirlo a su máxima simplicidad, pero no más." ,
                "La formulación de un problema, es más importante que su solución.",
                "Si no puedo dibujarlo, es que no lo entiendo."  ,
                "El sentido común se una colección de prejuicios adquiridos a los dieciocho años."  ,
                "La teoría es asesinada tarde o temprano por la experiencia"  ,
                "Los intelectuales resuelven los problemas, los genios los evitan "  ,
                "Si buscas resultados distintos, no hagas siempre lo mismo" ,
                "La imaginación es más importante que el conocimiento" ,

};
        public void ejbCreate(){
                System.out.println("ejecutando ejbCreate()");

};
```

```
        public void ejbCreate(){
                System.out.println("ejecutando ejbCreate()");

        }
          public void ejbActivate(){
                System.out.println("ejecutando ejbActivate()");

        }
          public void ejbPassivate(){
                System.out.println("ejecutando ejbPassivate()");

        }
          public void ejbRemove(){
                System.out.println("ejecutando ejbRemove()");

                }

          public void setSessionContext(SessionContext ctx){
                System.out.println("ejecutando setSessionContext()");

        }

                //metodo del negocio

        public String getQuoteEinstein(){
                System.out.println("ejecutando getQuoteEinstein") ;
                Int random = (int)(Math.random()  *   quotes.length) ;
                Return quotes[random] ;
        }

}
```

- Home:
  El cliente usa la home interface que hereda de la EJBHome para obtener una referencia a la interfaz del componente, el cual extiende de la interface EJBObject

```
import javax.ejb.*;
import java.rmi.RemoteException;

public interface QuoteHome extends EJBHome
{
        public QuoteComponent create () throws CreateException, RemoteException;

}
```

- Componet:
  La component interface es usada para invocar los métodos del negocio en el bean, porque el bean expone sus métodos de negocio en la interfaz de componente.
  Un cliente nunca accede directamente al bean, siempre invoca los métodos de negocio a través de la component interface.

```
Package com.quote;

import  javax.ejb.*;
import  java.rmi.RemoteException;

public interface QuoteComponent extends EJBObject
{

                public String getQuoteEinstein() throws RemoteException
}
```

Método remoto

- En la component interface, se declara el método **getQuoteEisten()**,
- Pertenece a la clase QuoteBean, invocado vía la interface

Descriptor de despliegue

- Contiene la información de cómo se debe desplegar el entrerprise bean, datos de configuración que necesita el bean e información acerca de seguridad.
- **ejb-jar.xml**

```
<?xml version="1.0"?>
<ejb-jar
      xmlns="http://java.sun.com/xml/ns/j2ee'
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                          http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
      version="2.1">
 <enterprise-beans>
    <session>
        <display-name>QuoteBean</display-name>
        <ejb-name>QuoteBeanSession</ejb-name>
        <home>com.quote.QuoteHome</home>
        <remote>com.quote.QuoteComponent</remote>
        <ejb-class>com.quote.QuoteBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
    </session>
 </enterprise-beans>
</ejb-jar>
```

Desplegar

- Poner el archivo en el directorio: ***server\default\deploy***
- Reiniciar el servidor

Cliente

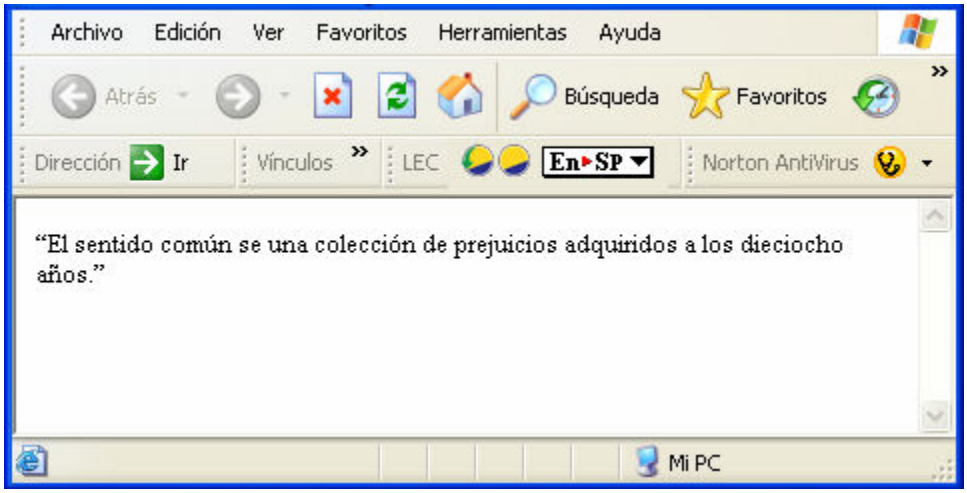
```
package com.clients;

import com.quote.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class QuoteClient {
    public static void main(String arg[]){
        try{
            Context ctx = new InitialContext();
            Object objref = ctx.lookup("quoter");
            QuoteHome home = (QuoteHome)PortableRemoteObject.narrow( objref, QuoteHome.class);
            QuoteComponent quote = home.create();
            System.out.println("La cita es " + quote.getQuoteEinstein());
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

JBoss.xml

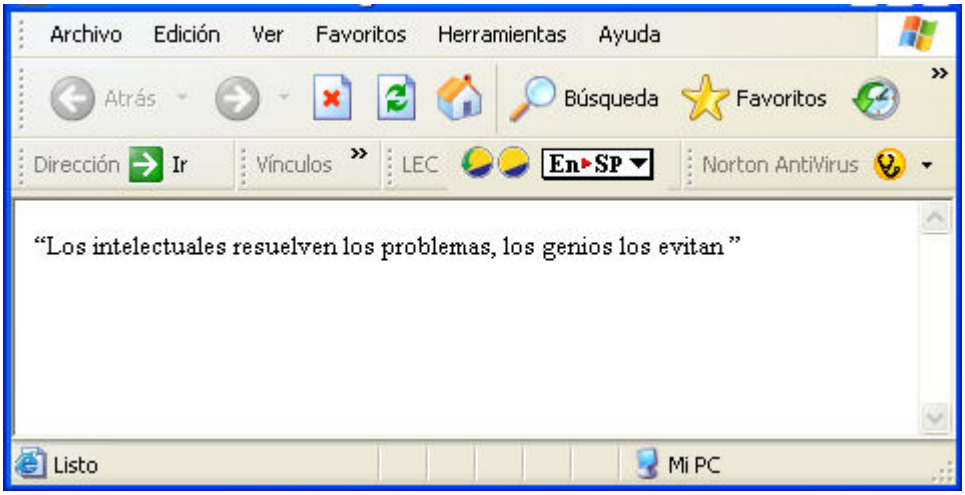
```
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
      <session>
        <ejb-name>QuoteBeanSession</ejb-name>
        <jndi-name>quoter</jndi-name>
      </session>
    </enterprise-beans>
</jboss>
```

jndi.

```
# Sample ResourceBundle properties file
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```



al ejecutarse, envia una de las citas del arreglo quotes



y al ejecutarse nuevamente envia otra.

Conclusiones

El desarrollo de software, principalmente multimedia utilizando tecnología Java es una alternativa importante para la enseñanza de las ciencias básicas, su amplia difusión deberá implicar una modificación de la estructura de la programación tradicional, así como actualización de equipos y navegadores.

La tecnología Java ha contribuido al desarrollo de software interactivo que puede funcionar eficientemente en red, si bien el despliegue de Applets, Jsp, Ejb y Servlets requiere de equipos de trabajo veloces y además de navegadores actualizados para su óptimo funcionamiento.

El resultado de este Trabajo ha sido un material escrito que servirá de apoyo teórico a la aplicación en *Java* para mostrar el desarrollo de conceptos y principales tecnologías del lenguaje j2EE y j2SE. Se pretende que tanto las páginas, como el código, sirvan de apoyo a la docencia de la carrera de Ingeniería en computación o cualquier usuario con deseos de envolverse con la tecnología de Java. Al mismo tiempo se ha intentado presentar una imagen atractiva de dicha tecnología desglosando de una manera sencilla las librerías mas usadas de Java , que ofrecen al usuario una visión más atractiva del los mismos.

Aunque los programas hechos en Java no tienden a ser muy rápidos, ya que son interpretados nunca alcanzan la velocidad de un verdadero ejecutable. Java es un lenguaje de programación. Esta es otra gran limitante, por más que digan que es orientado a objetos y que es muy fácil de aprender sigue siendo un lenguaje y por lo tanto aprenderlo no es cosa fácil. Especialmente para los no programadores. Java es nuevo. En pocas palabras todavía no se conocen bien todas sus capacidades.

Pero en general Java posee muchas ventajas y se pueden hacer cosas muy interesantes con esto. Hay que prestar especial atención a lo que está sucediendo en el mundo de la computación, a pesar de que Java es relativamente nuevo, posee mucha fuerza y es tema de moda en cualquier medio computacional. Muchas personas apuestan a futuro y piensan en Java.

Después de haber analizado las características propias del lenguaje Java, podemos concluir que hasta el momento, es un software de programación en el que podemos confiar, al diseñar nuestras aplicaciones, paginas web, esto es, por su gran versatilidad, facilidad de programación y seguridad.

Bibliografía

- **Eric Armstrong ,Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson,Ian Evans,Dale Green,Kim Haase,Eric Jendrock** The J2EE™ 1.4 Tutorial**,** June 17, 2004
- **Deitel ,Deitel** Java como programar, quinta edicion, Prentince hall, 2004
- **(Addison-Wesley)-**Developing Enterprise Java Application with J2EE and uml
- **Richard Monson-Haefel** Enterprise JavaBeans, 4th edition, (2004) [O'Reilly, ISBN 059600530X]
- **Ed Roman** Mastering Enterprise JavaBeans, 2nd edition (2001), de [Wiley&Sons, ISBN 0471417114]
- **Bruce Tate** Bitter EJB (2003) [Manning, ISBN 1930110952]
- **Kathy Sierra y Bert Bates** Head First EJB (2003), de [O'Reilly, ISBN 0596005717]

Enlaces externos

- **http:// www.javahispano.org**
  Página de java en castellano,con ejemplos y tutoriales
- **http:// www.sun.com**
  pagina de los creadores de la tecnologia java
- **http:// www.jboss.com**
  pagina de tecnologia ejb y servidor de publicacion
- **http:// www.netbeans.org**
  pagina con tutoriales y un IDE para desarrollo de aplicaciones java