FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

CURSOS INSTITUCIONALES

JAVA AVANZADO

Del 21 de Octubre al 01 de Noviembre del 2002

APUNTES GENERALES

CI-416

Instructor: MAC. Rodrigo Godo
CONSEJERÍA JURÍDICA
OCTUBRE/NOVIEMBRE DEL 2002

Palacio de Minería, Calle de Tacuba No 5, Primer piso, Delegación Cuauhtémoc, CP 06000, Centro Histórico, México D F,
APDO Postal M-2285 • Tels 5521 4021 al 24, 5623 2910 y 5623.2971 • Fax. 5510 0573

Programación Orientada a Objetos (POO)

- Objetos
    Clases Sub peso()

```
Dim tarea As Task

Dim proyecto As Tasks

Dim tot As Variant

Dim ava As Variant

Set proyecto = ActiveProject.Tasks

For Each tarea In proyecto

  'MsgBox (tarea.Duration)

  tarea.Number2 = tarea.Duration * 0.334 * tarea.Number1 / 60 / 8

  tot = tot + tarea.Number2


  ava = ava + (tarea ActualDuration * 0.334 * tarea Number1 / 60 / 8)


Next tarea

MsgBox (tot)

MsgBox (ava)

MsgBox (100 * ava / tot)

End Sub
```

    Constructores
    Métodos Públicos
    Métodos Privados
- Herencia
- Polimorfismo
- Organización de la Documentación de Java (Java API)
    Ejemplo: clase Frame

La mayoría de las veces un programa es capaz de cumplir con su objetivo recurriendo únicamente a funciones y a ciertas estructuras de control, el problema principal de crear el software de esta manera también llamado 'enfoque estructurado' es que mientras más grande sea nuestro programa será cada vez es más complejo lo cual no es bueno para nadie, ni siquiera para los programadores profesionales así que ¡imagínense como podría irnos a nosotros que somos principiantes!.

La década de los 70´s vio nacer, crecer y desarrollarse a la programación estructurada guiada principalmente por el lenguaje C, sin embargo estas mismas capacidades exigidas al máximo mediante programas cada vez más

avanzados y por lo tanto complicados dispararon la búsqueda de un nuevo paradigma (o enfoque) más rápido, correcto y económico para desarrollar software. Como en casi toda actividad humana la tendencia sigue siendo. más rápido, más fuerte y más alto, en cuanto a nuestras expectativas. Muchas investigaciones, buenas ideas y unas cuantas refinaciones terminaron en la concepción de este nuevo paradigma: La Programación Orientada a Objetos Según algunos se trata de una revolución que aún no termina y que comenzó con lenguajes como: Smalltalk ó C++ y que se encuentra en pleno apogeo justamente con Java.

Afortunadamente para los principiantes como nosotros este novedoso paradigma de programación nos queda como anillo al dedo, como estamos comenzando preferimos un código simple, claro, y bien organizado; sobre todo si sospechamos que aún existen errores por corregir (lo cual es casi seguro) y si tenemos en mente futuros cambios que se puedan implementar en un tiempo razonable sin necesidad de tener que rescribir todo el código. La Programación Orientada a Objetos o POO (por sus siglas) sirve para eso, los programas orientados a objetos son más fáciles de entender, corregir y modificar

¿En qué idea se basa la Programación Orientada a Objetos?

Al estudiar objetos de la vida real sin importar cuales sean podemos distinguir: 1) que se clasifican en tipos 2) que tienen características y 3) que podemos interactuar con ellos (sin importar de que forma se haga).

El primer punto se refiere a que tenemos una jerarquía conceptual, o traducido al español: que tenemos objetos de cocina, objetos de la oficina, objetos de baño, objetos de un auto (motor, parabrisas, sillones), objetos de una ciudad (edificios, autopistas, calles), objetos de un gobierno (secretarías, congreso, provincias) etc. etc, este orden es sumamente práctico y bastante simple

El segundo punto se refiere a que aunque dos objetos sean del mismo tipo, por ejemplo: dos autos, cada uno tiene características distintivas únicas como puede ser: el color, el número de cilindros, el número de válvulas, el número de sillones, la cantidad de kilómetros por litro, el modelo, el fabricante, velocidad máxima, potencia etc. Esto significa que aunque dos o más objetos sean del mismo tipo cada uno posee características únicas que lo definen.

El tercer y último punto se refiere a una interfaz o un método de comunicación con tales objetos, siempre podemos interactuar con ellos de alguna manera por ejemplo podemos **acelerar** el coche, **encender** el televisor, **vaciar** el bote de basura, **cerrar** la persiana, **operar** el microondas, etc, etc. El origen del nombre "métodos" (para las funciones) viene de esta idea, los métodos son nuestro medio de comunicación, interacción y manipulación con los objetos, todos las palabras anteriores que están marcadas representan acciones sobre los objetos, un método de comunicación

Resumiendo, un objeto tiene **tipo, características y métodos** Estamos acostumbrados a trabajar con objetos, los objetos de Java en teoría son iguales a los objetos de nuestra vida cotidiana como relojes, libreros, autos, calculadoras, microondas, computadoras, etc. digo que en teoría son iguales porque en Java estos objetos son código, se pretende manejar todo el código en java como si fueran objetos, pero, ¿cómo tratar como objetos algo que no lo es?, ¿cómo tratar el código de java como si fuera un reloj, un auto o una calculadora?, bueno, para tratarlos como tal primero debemos darle al código **el comportamiento** de esos objetos

Obviamente el objeto reloj de la vida real nunca será completamente igual a un objeto reloj en java, porque lo real es tangible y se puede tocar, en cambio todo objeto en java siempre será código (bytes, bytes y más bytes) y solo podré ver como intenta copiar la función de un objeto de la vida real a través de la pantalla de mi computadora. Entonces, en la vida real los objetos son tangibles, pero en Java naturalmente los objetos son simplemente archivos que contienen código **cuyo comportamiento imita a su contraparte real.**

¿Qué son los objetos en Java?

De acuerdo a lo que ya vimos, un objeto tiene un tipo, una serie de características individuales y varios métodos que nos permiten comunicarnos o interactuar con él. La definición más sencilla en base a lo anterior que se me ocurre es que un objeto en Java es un archivo de clase que contiene: un **tipo** (constructor), **características** (variables) y una serie de **métodos** (interfaz de comunicación), todo lo anterior es muy parecido a la manera 'formal' y algo aburrida en la que los programadores profesionales explican a los objetos por lo que no me quedaré con las ganas de decirles que mi explicación simple de objeto que es. **un archivo de clase lleno de funciones relacionadas.** Eso significa que **en parte** aprender POO no es del todo aprender algo nuevo propiamente dicho sino simplemente es aprender a organizar mejor las funciones, así de fácil!.

Entonces, estos objetos son archivos que contienen funciones y a estos archivos se les llama clases En la vida real tu puedes apagar un reloj, prenderlo, programar su despertador, traduciendo este comportamiento a Java tu puedes crear también tu reloj porque se supone que puedes manipularlo casi igual que a un reloj de la vida real, puedes ordenarle al reloj en Java que haga lo mismo pero al fin y al cabo es solo un archivo de clase lleno de funciones.

Nota: De aquí en adelante manejaré indistintamente a un objeto como un archivo de clase

Objetos o Archivos de Clases

Estructura Básica

Las clases (u objetos) contienen tres partes importantes:

1. **Constructores (o inicializadores)**
2. **Funciones Públicas**
3. **Funciones Privadas**

1. Constructores

Los constructores inicializan nuestra clase, le dan un estado inicial estable listo para su uso. Siempre que declarábamos una variable numérica entero (int) esta tomaba el valor cero como estado inicial estable listo para ser cambiado a nuestro gusto, los constructores hacen lo mismo pero para las clases. En un auto el constructor sería el equivalente de encender el motor, en una computadora el constructor iniciaria windows, linux, solaris o cualquier otro sistema operativo instalado preparándolo para su uso, en una calculadora el constructor cumpliría la función de encender el display y activar las teclas para su uso, si tenemos en mente crear una clase para nuestro programa en lo primero que debemos pensar es en como definir su constructor

"Constructor" es solo un nombre más elegante para referirnos a una función, el constructor en cierta forma también es una función pero es una muy especial porque no tiene tipo y no devuelve valores, solo es una sección de código que se ejecutará antes que las demás (muy parecida a la función init() de las applets). Puesto que el constructor se ejecutará antes que ninguna otra parte es un área perfecta para inicializar muchas variables a la vez por eso le llaman constructor, yo también le llamaría **función inicializadora** de variables.

2. Funciones Públicas (o Métodos Públicos)

Las funciones públicas son eso exactamente, funciones de la clase que podemos usar libremente en el momento que así lo necesitemos, pero todas las funciones que hemos manejado hasta el momento siempre han sido accesibles ¿bajo qué condiciones no lo son que debemos definirlas como públicas? cuando se quieren llamar **desde fuera de la clase**, todas las funciones que hemos manejado hasta el momento las hemos llamado desde el interior mismo de la clase (dentro de sus llaves más exteriores), más adelante veremos que crear objetos implica que llamaremos funciones desde el exterior, las funciones que son accesibles desde fuera de la clase son las públicas. Tal llamado desde afuera es el equivalente de enviar una orden al objeto, un método público sirve para modificar el estado interno de los objetos!

3. Funciones Privadas (o Métodos Privados)

Dada la explicación anterior creo que es más que obvio que las funciones privadas son las que no son accesibles desde el exterior de la clase, estas funciones solo se necesitan dentro de la misma clase, todas las que hicimos en los ejemplos han sido en cierta forma privadas porque solo las usamos en el mismo entorno de la clase, nunca desde el exterior excepto el g drawString() y otras que mencionamos como "predefinidas" en Java, nosotros llamamos estas funciones por tanto son públicas en sus respectivas clases como veremos más adelante Las funciones privadas son como las herramientas que posee internamente la clase para operar y por tal motivo no es importante que tengamos acceso a ellas desde el exterior.

Al igual que las funciones también existen variables públicas y privadas, estas tienen la misma utilidad que los métodos al ser modificadas. es decir puede ser muy práctico en algunas ocasiones declararlas como publicas o bien puede ser mejor declararlas como privadas, todo depende de la situación y del propósito de nuestra clase.

Ahora, veamos un ejemplo en pseudocódigo sobre cómo crear un objeto reloj que demuestre cuales serían sus funciones públicas y sus funciones privadas·

Constructor.
  Reloj negro, hora inicial 12:00am;
Funciones Públicas:
  Apagar
  Encender
  Poner despertador;
Funciones Privadas:
  Mecanismo interno de control
  Mecanismo interno de baterias
  Mecanismo de manecillas

Al utilizar uno de estos relojes nos importa su operación no su mecanismo interno, por eso existen funciones públicas o privadas. Las funciones públicas son la interfaz que usaremos. El constructor inicializa al objeto en un estado inicial estable para su operación.

Un ejemplo más, esta vez con una computadora:

Constructor:
  Computadora portátil compaq, sistema operativo windows98, encendida
Funciones Públicas:
  Apagado
  Teclado
  Pantalla
  Impresora
  Bocinas
Funciones Privadas.
  Caché del sistema
  Procesador
  Dispositivo de Almacenamiento
  Motherboard

Obviamente si tu abres (fisicamente) tu computadora tendrás acceso a todo, sucede lo mismo si abres el código fuente del archivo de una clase, puedes modificar todo a tu gusto, pero debe quedar bien claro que **la razón por la cual se hace todo lo anterior es para organizar mejor el código**, no para impedir el acceso a nadie a ciertas cosas, todo es para mejorar la operación general de una clase ocultando la información que no es necesario que conozcamos y exponiendo la que sí Existe un modificador más a parte de "public" y "private" que es "protected" el cual restringe el acceso para clases de otros paquetes, nosotros como principiantes aún no debemos preocuparnos por esto ya que en este curso no se tratan paquetes, mientras no se utilicen paquetes el modificador protected hace lo mismo que el public, todos regulan de alguna manera el acceso a los métodos y a las variables para organizar mejor las cosas y evitar los errores en el código final, bien aplicados evitan prácticamente que cometamos errores de modificación que de otra manera serian más difíciles de identificar con otros lenguajes, este es un resumen del alcance que nos dan los modificadores sobre los elementos de la clase.

Distintos modificadores y las restricciones que imponen a las variables y a los métodos

cceso\modificador	sin modificador	public	private	protected	private rotected
misma clase	si	si	si	si	si
subclase	si	si	no	si	si
clase ajena	si	si	no	si	no

La misma clase seria el acceso permitido a las variables globales y las funciones, para una subclase como ven tiene acceso casi a todo excepto las funciones private de sus superclases. Finalmente una clase ajena seria la que no tiene ninguna relación con la clase a la cual se pretende acceder o utilizar, observen que también es posible programar en java sin la necesidad de incluir ningún tipo de modificador para las variables o las funciones

Sintaxis en Java para crear Clases

La sintaxis de Java para los constructores, las funciones públicas y las privadas es casi intuitiva Las funciones se declararan igual excepto que ahora declararemos explicitamente antes de su nombre si se tratan de funciones públicas o privadas, en cuanto al constructor se declarará muy parecido a una función con la excepción de que no

tiene ningún tipo y por tanto no regresa ningún valor, además de que llevará el mismo nombre que la clase misma. Trataré de aclarar todo lo anterior con un ejemplo sencillo:

```
public class reloj {

  private int horas, minutos, segundos;

 //constructor
  public reloj() {
    horas = 12;
    minutos = 0;
    segundos = 0;
  }

 //funcion pública
  public void cambiarHora(int h, int m, int s) {
    horas = h;
    minutos = m;
    segundos = s;
  }

  //otra función pública
  public String obtenerHora() {
    return "Hora: " + horas + " Minutos: "
    + minutos + " Segundos. " + segundos;
  }

}
```

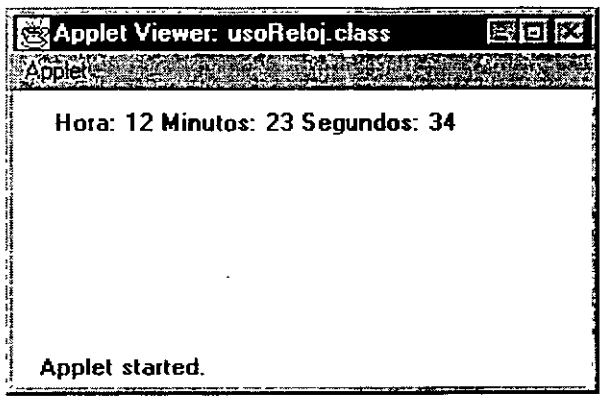
Esta clase es muy sencilla, contiene dos funciones públicas y un constructor. La palabra clave public al inicio permite que cualquier navegador tenga acceso libre para abrir nuestra clase, como ya dije su aplicación sobre las funciones permite que sean públicas, es decir, accesibles desde fuera de la clase. La contraparte de un public es un private, al declarar las primeras tres variables enteras como private estoy cerrando literalmente el acceso a cualquiera que intente manipularlas desde fuera de la clase, en cierto modo private les pone un candado para que nadie excepto la misma clase pueda modificarlos, en este momento no se ve muy claramente la utilidad del constructor porque nunca se le llamará como función dentro de la misma clase, no tiene tipo y tampoco devuelve ningún valor, solo podemos ver que inicializa las variables horas, minutos y segundos. Un reloj siempre empieza en 12 pensando en horas, 0 en minutos y 0 en segundos, en un momento veremos al constructor en acción La función pública cambiarHora es de tipo void lo que significa que no devuelve ningún valor, recibe tres argumentos enteros que sirven para reajustar los valores de horas, minutos y segundos, como esta función esta trabajando dentro de la misma clase puede acceder libremente a estas variables aunque sean private. La última función es de tipo String y devuelve una cadena que muestra la hora completa juntando el contenido de todas las variables

Bueno, la clase reloj anterior es lo que tambien se llama en Java un objeto, ¿cómo lo usamos ahora?. Es evidente que la clase sola no sirve de mucho, alguien o algo debe usarla para aprovecharla. Existen muchos términos manejados en los libros de texto que a mi juicio han respetado demasiado bien las versiones originales en Inglés de palabras que no se utilizan mucho en el Español como variables de instancia, instanciar objetos, instanciar clases, instancia de una variable, instancias de un caso, etc otra palabrita que les ha gustado mucho ha sido ejemplarizar, ejemplarizar objetos, ejemplarizar variables, ejemplarizar clases, tal vez para muchas personas sea más que obvio su significado pero también estoy segura de que ese no es el caso para alguien que esta aprendiendo desde cero cuyo primer acercamiento a la programación es Java. Antes de complicarnos la vida explicando concepto por concepto es mejor dar un ejemplo de como utilizar una clase mediante un applet sencilla que llamará y utilizará a la clase reloj que creamos anteriormente:

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

public class usoReloj extends Applet {

  reloj R;

 //inicializar applet
  public void init() {
```

```
    R = new reloj();
    R.cambiarHora(12, 23, 34);
  }

 //dibujar en la pantalla una cadena
  public void paint(Graphics g) {
    g drawString(R obtenerHora(), 20, 20);
  }

}
```

Su salida es:



Nota: Recuerda que para compilar el código anterior debes crear dos archivos y colocar en cada uno su código correspondiente, recuerda que el nombre del archivo debe ser igual al nombre de la clase.

Antes que nada, se supone que tanto la clase reloj como la applet usarReloj anterior deben encontrarse en el mismo directorio de trabajo, si estas usando RealJ ambas clases deben estar en el mismo proyecto, esto porque la clase compilada usoReloj utiliza a reloj

La applet usoReloj tiene dos métodos y una sola variable. La variable R, es de tipo reloj Esto puede sonar algo raro pero asi es. La clase reloj que creamos por el hecho de ser una clase (u objeto) se puede manejar como un tipo (igual que un int, boolean, double, char o String) eso significa que al crear una variable de tipo reloj también estaremos creando las funciones públicas y privadas contenidas dentro de él para su uso, esta es una característica muy interesante que nos permite crear muchisimas funciones de un solo "jalón". Lo primero que hacemos es nombrar nuestro reloj como R, pero esto no basta para inicializar nuestro reloj!!, para inicializarlo verdaderamente llamamos a su constructor mediante:

R = new reloj();

Esto esta dentro de init() recuerda que el método o función init() de un applet es la que siempre se llama primero, como vamos a iniciar nuestra variable reloj lo colocamos ahi. Lo que hace "new reloj()" es llamar al constructor de la clase reloj, o dicho de otra forma, esto llama a la función que inicializa de forma estable las variables de nuestra clase y no solo eso, sino que también asigna todas esas variables y funciones de la clase reloj a R (es decir, se nombra o se etiqueta la variable), el operador new indica que es un nuevo tipo (u objeto) el que estamos creando, este operador solo se utiliza al inicializar nuestra variable reloj R. Según todo lo anterior podemos notar que la clase reloj es solo como un cascarón (o modelo abstracto) de un reloj y que este no se "echa a andar" hasta nombrarlo e inicializarlo, entonces el reloj realmente se crea a partir de la clase y deja de ser algo abstracto para convertirse en un caso especifico de la clase, hacer esto es lo que se llama "instanciar objetos" o también "ejemplarizar objetos", a veces pareciera que los programadores profesionales no desean que la gente común aprenda estos términos!!, en fin Es muy importante notar que el constructor de la clase reloj no recibia ningún argumento y de igual manera al crear con el operador new nuestro objeto no fue necesario pasar ningún argumento a la función constructora, pero toma en cuenta que si el constructor recibiera argumentos entonces deberiamos pasárselos al llamarlo, esto también seria una forma de pasar datos desde el applet hasta nuestro reloj

Por lo que has visto probablemente te has dado una idea del porque se utiliza el operador punto " " Una vez creado nuestro reloj ahora necesitamos acceder a sus funciones públicas, ¿cómo se accede a ellas? eso es! con el punto!, "R cambiarHora(12, 23, 34)" accede a la función pública cambiarHora dentro de la clase reloj, ahora, la razón de poner antes R. es para aclarar que nos referimos a la función contenida especificamente en el objeto etiquetado como R, sucede lo mismo con R.obtenerHora(), esta función no recibe argumentos y devuelve un String, en cuanto al g.drawString() recuerden que recibe Strings, por eso atrapa la que devuelve R.obtenerHora() y la exhibe en las coordenadas 20,20 del applet Debido a que modificamos el valor de la hora después de haber inicializado nuestro reloj R la hora exhibida no es 12:00:00 como la habia inicializado nuestro constructor.

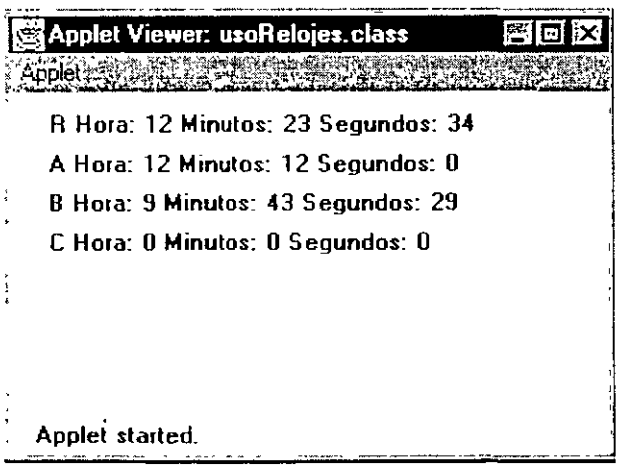
Esto es programación orientada a objetos en su forma más simple, se trata de organizar funciones con un poco mas de estilo y sobre todo orden. Lo anterior implica que si podemos crear nuestros tipos entonces podemos extender el lenguaje Java fácilmente, al restringir el acceso a ciertas funciones y ciertas variables prevenimos cualquier intento erroneo de modificarlas, esto evita muchos errores. En programas muy grandes a veces pueden

pasar semanas o meses para encontrar la fuente de un posible error, en Java, si respetas las convenciones, ni siquiera es posible que cometas los errores!.

Java no solo te permiten manejar una sola variable de una clase que tu hayas creado, puedes crear tantas como desees de ella cada una con sus propias funciones y variables públicas o privadas identificadas por la etiqueta o nombre que tu les des, Java jamás las confundirá ni las mezclará a menos que tú así lo programes:

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

public class usoRelojes extends Applet {

  reloj R;
  reloj A;
  reloj B;
  reloj C,

 //inicializar applet
  public void init() {
    R = new reloj(),
    R.cambiarHora(12, 23, 34);
    A = new reloj();
    A cambiarHora(12, 12, 0);
    B = new reloj();
    B.cambiarHora(9, 43, 29);
    C = new reloj(),
    C.cambiarHora(0, 0, 0);
  }

 //dibujar en la pantalla una cadena
  public void paint(Graphics g) {
    g drawString("R " + R.obtenerHora(), 20, 20),
    g drawString("A " + A.obtenerHora(), 20, 40),
    g drawString("B " + B.obtenerHora(), 20, 60);
    g drawString("C " + C.obtenerHora(), 20, 80);
  }

}
```

Salida:



Como puedes ver es fácil crear varios relojes con una sola clase reloj, esto se llama reutilidad del software, codifica una vez y úsalo cuantas veces quieras, otra ventaja de Java, pero claro no podia faltar el conjunto de términos formales, aburridos y a veces casi incomprensibles por parte de los programadores profesionales, del ejemplo anterior en una clase de programación típica les dirian: "instancias de objetos de la clase reloj" o bien podrian decirte, "Una clase es la generalización de un objeto, la idea de la clase es una idea común en la mayoría

de las actividades" ¿quieren más? sobran ejemplos así! "Una clase es la especificación de cualquier # de objetos que son lo mismo y que tienen un mínimo comportamiento común", pero no se preocupen porque no es tan complicado como parece.

En los anteriores ejemplos, no incluí ninguna función private porque no fue necesario, las funciones private se deberían incluir cuando así se requieran, generalmente las funciones public llaman a las private como funciones auxiliares o utilitarias, si quieren verlo así son como las subordinadas de las public porque estas las usan para lograr su objetivo que puede ser un cálculo, una conversión, etc. el caso es que no es necesario que esas funciones estén accesibles desde el exterior porque no se llaman directamente, por eso se declaran como private.

Se pueden crear varias funciones y constructores con el mismo nombre dentro de una clase, esto se llama sobrecarga de métodos o sobrecarga de constructores dependiendo el caso, la única condición es que esos constructores y funciones no reciban los mismos argumentos, siguiendo con el ejemplo de la clase reloj podríamos haberla hecho de la siguiente forma:

```
public class reloj {

  private int horas, minutos, segundos;

 //constructor UNO
  public reloj() {
    horas = 12;
    minutos = 0,
    segundos = 0;
  }

  //constructor DOS
  public reloj(int h, int m, int s) {
    horas = h,
    minutos = m;
    segundos = s;
  }

 //funcion pública UNO
  public void cambiarHora(int h, int m, int s) {
    horas = h;
    minutos = m;
    segundos = s;
  }

  //funcion pública DOS
 public void cambiarHora(int h) {
    horas = h;
  }

  //otra función pública
  public String obtenerHora() {
    return "Hora  " + horas + " Minutos  "
    + minutos + " Segundos: " + segundos,
  }

}
```

Como ven se repiten las funciones, pero no reciben los mismos argumentos, por tanto para Java son funciones distintas; El segundo constructor nos permitiría crear un reloj con una hora predeterminada en vez de tener que crearlo primero y ajustarlo después, práctico ¿no creen? La segunda función pública nos permitiría ajustar unicamente la hora sin alterar el valor de los minutos y los segundos, otra mejora práctica, también se podrían agregar otras dos funciones cambiarHora, una para ajustar únicamente los segundos y otra para los minutos.

De la misma manera que se pueden crear tipos nuevos mediante el operador new también se pueden destruir. El único propósito por el cual podríamos desear "destruirlos" es para liberar memoria, quizás la palabra "destruir" es demasiado drástica pero es que en realidad esos objetos se borran de la memoria así que de hecho si son destruidos

Una vez que los objetos son destruidos naturalmente ya no se puede volver a utilizar ninguna de sus funciones y por la misma razón si se desean utilizar nuevamente entonces debe llamarse otra vez a su respectivo constructor Retomando el ejemplo de la applet anterior podriamos destruir los relojes hasta el final de la clase poniendo.

```
R = null,
A = null;
B = null;
C = null;
```

Recuerda que solo se hace esto para ahorrar memoria o para reconstruir completamente desde cero nuestros objetos en caso de que sea conveniente. Una vez destruidos se pueden volver a crear llamando a sus constructores otra vez y así sucesivamente.

Herencia

La herencia es algo tan obvio y sencillo como lo dice su nombre. ¿No sería práctico heredar todas las variables y funciones de una clase para usarlas en otra? pues eso es lo que hace la herencia, las funciones se heredan tal cual, las variables igual, en lo único que debemos tener cuidado es en los constructores. La herencia busca aprovechar algo que ya esta implementado para reutilizarlo, la reutilidad del software facilita las cosas

La programación orientada a objetos nos permite crear tipos a partir de una clase, la herencia nos permite extender esos tipos sin necesidad de rescribir todo el código, únicamente escribiremos las "mejoras" y lo demas lo heredaremos de la clase madre. Esto se aclara con un ejemplo:

```
import java.awt.*,

public class cuadro {

  int x, y, ancho, alto,

//constructor uno
  public cuadro() {
    x = 20;
    y = 20;
    ancho = 50;
    alto = 50;
  }

//constructor dos
  public cuadro(int x, int y, int ancho, int alto) {
    this.x = x;
    this.y = y;
    this.ancho = ancho;
    this.alto = alto,
  }

  //funcion pública uno
  public void dibujar(Graphics g) {
    g.drawLine(x, y, x + ancho, y),
    g.drawLine(x + ancho, y, x + ancho, y + alto),
    g.drawLine(x + ancho, y + alto, x, y + alto),
    g.drawLine(x, y + alto, x, y);
  }

}
```

Para poder heredar, primero necesitamos una superclase. La denominación "superclase" solo indica que vamos a heredar de ella otras clases pero en realidad es una clase ordinaria, no es que tenga poderes especiales ni mucho menos!. La clase anterior llamada "cuadro" será nuestra superclase o clase madre y servirá para dibujar un cuadro ordinario en un applet, posee dos constructores sobrecargados y una función dibujar, también posee 4 variables de clase,"x" e "y" definirán la posición del cuadro a partir de la esquina izquierda en el plano coordenado del applet, el ancho será la cantidad a sumar en x a partir de la posición inicial y lo mismo para el alto, solo que en el

eje y. El primer constructor no recibe ningún argumento y es similar al primero que hicimos, crea de forma predeterminada un cuadro cuya posición esta en 20,20 de ancho y alto 50.

La palabra clave this

El segundo constructor recibe cuatro argumentos enteros e introduce la palabra this:

```
//constructor dos
  public cuadro(int x, int y, int ancho, int alto) {
    this.x = x;
    this.y = y;
    this.ancho = ancho;
    this.alto = alto;
 }
```

Existen dos funciones que cumple this aunque en realidad es la misma, esta es una de las cosas que me confundió un poco al aprender el lenguaje Primero, las variables que recibe como argumento este constructor son válidas solo en su ámbito, o sea dentro de sus llaves, eso significa que tenemos unas variables x,y,ancho,alto en el ámbito del constructor y otras variables x,y,ancho,alto fuera del ámbito del constructor, o lo que es lo mismo en el ámbito de la clase (las llaves más exteriores). Si tu recibes variables en una función que ya están declaradas fuera de su ámbito entonces las variables de tal función ocultarán a las variables fuera de ella que tengan el mismo nombre, eso significa que aunque las variables se llamen igual serán cosas distintas, en este caso this nos sirve para acceder a las variables fuera del ámbito de nuestra función (en este caso, el constructor o función inicializadora), entonces this sirve para pasar el valor de nuestras variables locales del constructor hacia las variables externas o globales de la clase, el this nos permite distinguirlas!. La otra función del this es acceder al objeto actual, o bueno, al menos esa es la explicación que dan todos los libros, pero no explica nada!. El dichoso objeto actual es nuestra clase o lo que es lo mismo, nuestro archivo. Eso significa que el constructor anterior también se puede poner así.

```
//constructor dos
  public cuadro(int x, int y, int ancho, int alto) {
    cuadro x = x,
    cuadro y = y,
    cuadro ancho = ancho;
    cuadro.alto = alto;
  }
```

¿Recuerdan que las funciones públicas de una clase se llaman con el operador punto? y claro también deben recordar que sucede lo mismo con las variables!, mientras sean públicas, se pueden acceder con un punto, naturalmente también se pueden modificar o manipular, solo mientras sean públicas, no lo olviden! Pero aquí hay algo raro, no hemos creado ningún objeto cuadro en esta clase y aún así lo estamos llamando con todos esos operadores punto, bueno, si hay un objeto cuadro, pero estamos dentro de él!. Esto significa que esta clase se esta llamando a si misma y está accediendo a sus propias variables globales x,y,ancho,alto; esta es otra forma de ver el asunto Así que el this solo te ahorra tener que poner todo el nombre de la clase pero como ya dije, es lo mismo

La última función, dibujar(Graphics g) sirve solo para pintar en el applet las líneas que conformarán nuestro cuadro, es void porque no deseamos que devuelva ningún valor, solo debe llamar a las funciones de dibujo, por cierto por esa razón importamos el paquete java.awt.*; para poder dibujar en la pantalla, los tipos Graphics se encuentran en java.awt *; El argumento Graphics recibido en la función dibujar lo nombramos g, en el tipo Graphics se almacenan muchas funciones relacionadas con la exhibición de gráficas del applet, en este caso usamos g drawLine(), esta funcion es para dibujar líneas y recibe 4 argumentos enteros, el primero es para la coordenada "x" inicial de la linea, el segundo para la "y" inicial, el tercero para la "x" final y el cuarto para la "y" final, al agregarle el ancho y el alto completamos 4 líneas que forman un cuadrado:

```
//funcion pública uno
  public void dibujar(Graphics g) {
    g drawLine(x, y, x + ancho, y);
    g drawLine(x + ancho, y, x + ancho, y + alto);
    g drawLine(x + ancho, y + alto, x, y + alto);
    g drawLine(x, y + alto, x, y);
  }
```

Bueno, lo anterior solo fue la clase de la cual vamos a heredar para crear otra clase que implementará todo lo anterior, la clase es la siguiente:

```
import java.awt.*;

public class cuadroX extends cuadro {

  public cuadroX() {
    super();
  }

  public cuadroX(int x, int y, int alto, int ancho) {
    super(x, y, alto, ancho);
  }

  public void dibujar(Graphics g) {
    super dibujar(g),
    g.drawLine(x, y, x + ancho, y + alto);
    g drawLine(x + ancho, y, x, y + alto);
  }

}
```

Como ven es más pequeña que la anterior puesto que el propósito de todo esto es ahorrarnos tiempo escribiendo menos código. Nuevamente importamos java awt *; porque dibujaremos algunas cosas extras en la pantalla mediante otro tipo Graphics. La palabra clave extends indica que heredaremos de la clase cuadro, observen bien la notación, es bastante simple:" extends "superclase". Con estas dos palabritas hemos heredado instantáneamente todas las variables y funciones de la clase anterior, pero no sus constructores, los constructores es lo único que no se hereda; Esto no es tan malo ya que quizás los constructores de esta nueva clase podrían necesitar más argumentos que la anterior. Uno de los requisitos para que una clase pueda heredar de otra es que la subclase llame a los constructores de su superclase.

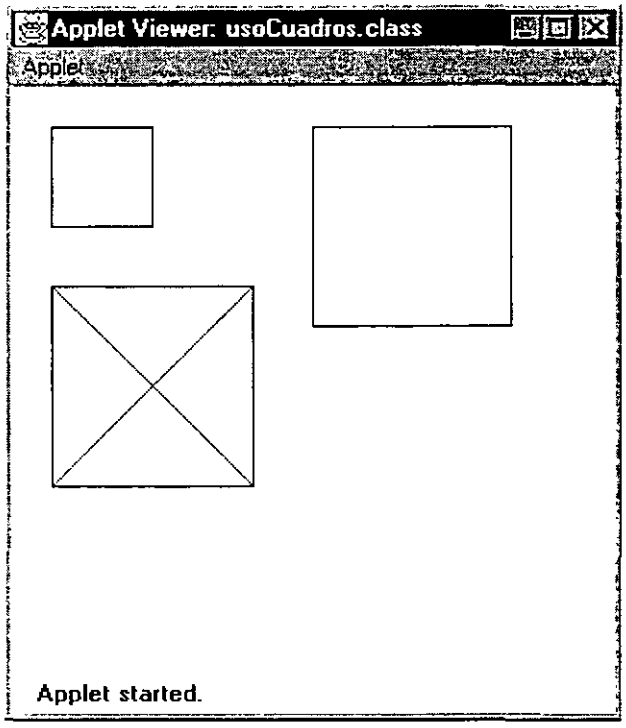
La palabra clave super

Para llamar al constructor de la superclase usamos super(), ya que no tiene ningún argumento se está refiriendo al primer constructor de la clase cuadro (el que no recibe argumentos), de la misma forma podríamos haber llamado al segundo constructor, pero en tal caso, tenemos que pasar sus debidos argumentos. El "super" no solo sirve para llamar al constructor de la superclase, también sirve para acceder a sus variables y a sus funciones, ¿pero que no ya las habíamos heredado para no tener que llamarlas? cierto, el "super" para llamar funciones y variables solo se usa cuando la subclase y la superclase tienen funciones y/o variables con el mismo nombre (de modo que las de la subclase ocultan a las de la superclase, esto también se llama supeditar los métodos de la superclase), es una forma de distinguirlas muy similar al this, por ejemplo el método dibujar existe tanto en la subclase como en la superclase pero utilicé el "super" en la función dibujar de la subclase para distinguir explícitamente que estoy haciendo referencia a la función dibujar de la superclase, no fue necesario usar "super" para acceder a las variables x,y,ancho,alto porque estas fueron heredadas, si en la subclase hubiera definido otras variables "x,y,ancho,alto" propias para acceder a las variables del mismo nombre de la superclase (ahora ocultas) usaría "super.ancho" por decir algo. Por último, vamos a usar las dos clases anteriores en un applet:

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

public class usoCuadros extends Applet {

  cuadro c, C,
  cuadroX cx;

  public void init() {
    c = new cuadro().
    C = new cuadro(150, 20, 100, 100),
    cx = new cuadroX(20, 100, 100, 100);
  }
```

```
  public void paint(Graphics g) {
    c.dibujar(g);
    C.dibujar(g);
    cx.dibujar(g);
  }

}
```

Salida del Applet:



Nota: Recuerda que para compilar el código anterior cada clase debe estar en un solo archivo y que su nombre debe ser igual al de la clase.

Lo único importante de remarcar del código anterior es la forma en la que se pasa la variable Graphics g a las funciones dibujar de cada clase, las funciones de las clases necesitan este argumento, si no, no pueden dibujar[1]. Primero se crean dos cuadros simples con la superclase, sin herencia, son los dos superiores e ilustran nuevamente el uso de los constructores El tercer cuadro cuadroX es el que heredó todas las variables y funciones de la clase cuadro, observen las nuevas adiciones como líneas que cruzan. Recuerden que el sistema coordenado de Java comienza en la esquina superior izquierda con el eje X incrementando su valor hacia la derecha y el Y hacia abajo

Polimorfismo

Existen muchas formas de explicar este concepto así que esta es una de tantas, que no te sorprenda encontrar otros ejemplos ligeramente diferentes. Apostando a mi escasa experiencia espero que les sea útil, el comportamiento polimórfico en Java indica que ciertas variables del código pueden **comportarse de distintas maneras**, es decir en un momento se comportan como si fueran de cierto tipo y si así lo deseamos pueden tomar otro comportamiento como si fueran una variable distinta

La clave del polimorfismo en Java esta en la **elección de métodos de forma dinámica** esto significa que el intérprete de java elige la función apropiada en tiempo de ejecución

Por ejemplo, supongan que tenemos una impresora por un lado y una televisión por el otro (en términos de objetos de la vida real), estos dos objetos son similares pero no son iguales, aún así se encienden casi de la misma manera· con un botoncito, si alguien te pide que apagues la TV accionas su botoncito y sucede lo mismo con la impresora pero si te piden que mandes a imprimir algo a la televisión pues evidentemente no será posible!! Bueno

pues eso es la elección dinámica de métodos, el intérprete Java realiza esta revisión en tiempo de ejecución (solo que esta vez revisa los archivos de clases), ¿en qué criterio se basa esta elección? se basa en la clase (u objeto) con el cual se esté trabajando actualmente, veamos un ejemplo que lo aclare mejor.

La siguiente clase se declara como abstracta, esto significa dos cosas: 1) que no se pueden crear tipos (u objetos) con ella, es decir que no lleva constructor y por ende **no puedes llamar a su constructor**, y 2) no hay variables abstract ni constructores abstract, pero si funciones, **una clase abstracta solo sirve para obligar a sus subclases a incluir los métodos que esta tiene** o en otras palabras: Si se va a heredar de una clase abstracta entonces las subclases deberán definir obligatoriamente (como sea, pero deberán) todos los métodos abstractos que tenga dicha superclase abstracta, bueno, primero veamos el ejemplo y a continuación extenderé más la explicación:

```
public abstract class figura {

  public int area, perimetro,

  public abstract int obtenerArea(),
  public abstract int obtenerPerimetro(),
  public abstract String mostrarFigura();

}
```

La palabra clave abstract indica que esta es una clase abstracta Las clases abstractas sirven generalmente para organizar las subclases obligando como ya mencioné a que incluyan los métodos de esta superclase porque si están heredando de la misma superclase deben compartir por lo menos algunas características y comportamientos similares. Se tienen dos variables enteras "public" y tres métodos "abstract" públicos, ahora, estos métodos o funciones no están completos precisamente porque cada subclase deberá completarlos a su modo dependiendo de su propósito para cada caso, observa muy bien la notación de estos métodos abstract (no incluyen llaves y agregan ; al final) no puedes declarar un método abstract si la clase no es también abstract, entonces la clase anterior solo sirve para obligar a las que hereden de ella a crear sus propias funciones obtenerArea(), obtenerPerimetro() y mostrarFigura(), el polimorfismo también puede ejemplificarse sin la necesidad de recurrir a las clases abstractas pero creo que se comprende mejor así como veremos más adelante ya que vamos a heredar dos subclases de la anterior "figura"

Primero vamos a crear una clase que herede de la clase anterior y definamos las funciones que nos pide la superclase abstracta:

```
public class circulo extends figura {

  private int radio;

 //Constructor
  public circulo(int radio) {
    this.radio = radio;
    area = area();
    perimetro = perimetro(),
  }

 //función pública UNO que nos exige la clase abstracta figura
  public int obtenerArea() {
    return area.
  }

  //función pública DOS que nos exige la clase abstracta figura
  public int obtenerPerimetro() {
    return perimetro;
  }

  //funcion privada UNO que utiliza nuestro constructor como
 //herramienta para calcular el área del circulo
  private int area() {
    return (int) (Math PI*Math.pow(radio, 2)),
  }
```

```
  //función privada DOS que utiliza nuestro constructor como
   //herramienta para calcular el perimetro del circulo
   private int perimetro() {
     return (int) (Math.PI*radio*2);
   }

  //función pública TRES que nos muestra (en una cadena o String)
   //las caracteristicas de nuestro objeto circulo
   public String mostrarFigura() {
     return "Círculo, radio " + radio,
   }

}
```

La clase anterior contiene un constructor, tres funciones públicas y dos funciones privadas, Tal como lo exigia la superclase "figura" tuvimos que crear nuestras versiones de los métodos abstractos. obtenerArea(), obtenerPerimetro() y mostrarFigura(). Esta vez ya no son abstract puesto que estamos definiendo las funciones, mas adelante será aún más evidente la razón por la cual la superclase debe ser abstract.

Lo único relevante de explicar de la clase anterior es el hecho de aplicar funciones privadas por primera vez, estas funciones son herramientas que utiliza el constructor para calcular el área y el perimetro de la circunferencia y por tal motivo no es necesario que estén disponibles del exterior, sin embargo los resultados de estos cálculos si deben estar disponibles, para eso estan las funciones públicas, para acceder a los resultados. Aqui no fue necesario importar el paquete java awt * puesto que aún no dibujaremos nada, solo haremos unos cuantos cálculos y los pondremos en nuestras variables enteras y en una cadena (con la función pública final). La función privada UNO·

```
//función privada UNO que utiliza nuestro constructor como
 //herramienta para calcular el área del circulo
 private int area() {
   return (int) (Math.PI*Math.pow(radio, 2));
 }
```

Utiliza la función Math pow(radio, 2) que recibe dos argumentos, esta función que se encuentra en la clase Math es para evaluar potencias, en este caso eleva el radio al cuadrado y por tanto es evidente que el primer argumento es la base de nuestra potencia y el segundo es el número de veces que se multiplicará por si misma, el valor Math PI es una constante de tipo double que contiene el valor de pi=3 14159... como este valor es double pero deseamos que nuestro resultado esté expresado como entero entonces convertimos todo a int La estructura restante de la clase es bastante simple y creo que no requiere una explicación detallada.

Ahora crearemos otra clase que también heredará de figura y por tanto, deberá definir los métodos abstractos a su modo

```
public class cuadrado extends figura {

  private int base, altura;

  public cuadrado(int base, int altura) {
    this.base = base;
    this altura = altura,
    area = area(),
    perimetro = perimetro(),
  }

  public int obtenerArea() {
    return area,
  }

  public int obtenerPerimetro() {
    return perimetro,
  }
```

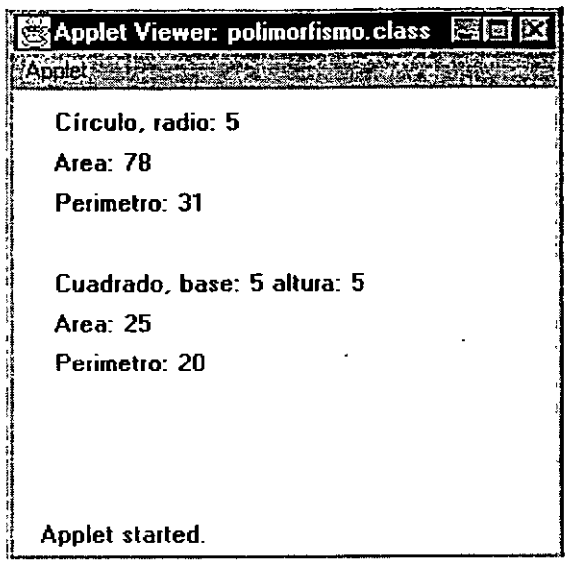
```
  private int area() {
    return base*altura;
  }

  private int perimetro() {
    return 2*base + 2*altura;
  }

  public String mostrarFigura() {
    return "Cuadrado, base: " + base + " altura: " + altura;
  }

}
```

Esta clase es aún más simple que la anterior (circulo). Noten que define a su manera las funciones abstractas obtenerArea(), obtenerPerimetro() y mostrarFigura(), siempre respetando su tipo y exactamente el mismo nombre, solo que para el caso de un cuadrado, las demás funciones también son relativamente simples.

Ya que tenemos estas tres clases ahora podemos estudiar el meollo del asunto, el polimorfismo. Para eso es la siguiente applet:

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

public class polimorfismo extends Applet {

  figura N;
  circulo circ,
  cuadrado cuadr;

  public void init() {
    circ = new circulo(5);
    cuadr = new cuadrado(5, 5),
  }

  public void paint(Graphics g) {

    N = circ,
    g.drawString(N.mostrarFigura(), 20, 20),
    g.drawString("Area: " + N obtenerArea(), 20, 40),
    g drawString("Perimetro. " + N.obtenerPerimetro(), 20, 60);

    N = cuadr,
    g.drawString(N.mostrarFigura(), 20, 100);
    g.drawString("Area: " + N.obtenerArea(), 20, 120);
    g drawString("Perimetro " + N.obtenerPerimetro(), 20, 140);

  }

}
```

Salida del Applet poliformismo:



Si han estudiado cuidadosamente el código anterior creo que ya pueden darse una idea acerca de lo que hicimos sin una explicación detallada, pero claro, aún así lo voy a explicar. Primero veamos la función init y las variables:

```
figura N;
circulo circ;
cuadrado cuadr,

public void init() {
  circ = new circulo(5);
  cuadr = new cuadrado(5, 5),
}
```

Creamos tres variables, una de tipo figura (de la clase abstracta), una circulo y una cuadrado. En la función init del applet inicializamos nuestras variables llamando a sus respectivos constructores, recuerden que no podemos inicializar a "figura" porque es abstracta y además ni siquiera tiene constructor! por tanto no podemos usar new y si lo intentáramos se produciría un error, probablemente ahora se pregunten ¿de que va a servir si no la vamos a inicializar? bueno, la respuesta es simple, nuestra figura N es la que "cambiará de forma" (de ahí viene polimorfismo) para comportarse como un círculo o como un cuadrado según sea el caso, lo interesante esta en la función paint·

```
public void paint(Graphics g) {

   N = circ;
   g drawString(N mostrarFigura(), 20, 20);
   g.drawString("Area· " + N obtenerArea(), 20, 40),
   g drawString("Perimetro  " + N obtenerPerimetro(), 20, 60),

   N = cuadr;
   g drawString(N mostrarFigura(), 20, 100),
   g drawString("Area  " + N obtenerArea(), 20, 120),
   g drawString("Perimetro: " + N.obtenerPerimetro(), 20, 140);

}
```

Primero le damos a N la forma de un objeto de tipo círculo llamada circ y como ven al llamar a sus funciones realmente se comporta como un circulo!! pues regresa un resultado de la llamada a los métodos de un circulo!, después le damos a N la forma de una clase tipo cuadrado y nuevamente se comporta como tal, observen como accedemos a las funciones de las clases con el operador punto y recuerden que si accedemos a una función deben colocarse sus paréntesis (si las funciones llevan argumentos evidentemente hay que incluirlos también) y si accedemos a una variable de una clase solo basta con poner su nombre sin paréntesis con el operador punto (siempre y cuando sean public) En muchos programas es ventajoso utilizar un valor abstracto de tal como "N" del

ejemplo anterior como referencia que pueda tomar diversos comportamientos, otra ventaja del polimorfismo es la extensibilidad o adaptabilidad que nos brinda al mejorar un programa, por ejemplo supongan que deseamos modificar alguna de las figuras: "circulo o cuadrado", solo bastaria con editar y recompilar su código sin necesidad de modificar de ninguna manera el applet o la clase abstracta pues los métodos son los mismos y su elección continuará siendo dinámica. Bueno, esto es todo lo más importante relacionado con la programación orientada a objetos, ahora solo resta que aprendan a pensar en cómo solucionar los problemas mediante clases.

Funciones y variables "static"

Finalmente, numerosas veces hemos llamado a las funciones sin la necesidad de crear objetos (o tipos a partir de una clase) con el operador new, esto es porque una clase puede declarar unas funciones especiales marcadas como static, estas funciones solo se llaman con el operador punto sin la necesidad de llamar antes al constructor de la clase, esto significa que estas clases son solo **bibliotecas de funciones**, a veces son muy útiles, la clase Math contiene muchos métodos static que podemos llamar solo con: Math.nombre_de_función(argumentos),.

También se pueden declarar variables "static", el propósito de estas variables es que estén compartidas por todas las subclases de una clase, o sea que es como una variable global de subclases, si una subclase modifica una variable declarada como "static" en la superclase de la cual hereda **todas** las otras subclases notarán el cambio aunque ellas no la hayan tocado

Como ejemplo de clases static para que vean que no es mentira aquí esta la clase Math tal como viene en el Java Developers Kit!:

Clase Math

Como pueden ver también existen los modificadores "native" y "synchronized" el native se refiere a que esa función esta en código nativo el cual me parece que generalmente es en lenguaje c y esto es porque todas las operaciones matemáticas deben ser muy veloces, aún debo investigar más al respecto, si alguien ya conoce ese tema que me ayude a extender mi inocente explicación! en cuanto al synchronized en pocas palabras indica que esa función solo se puede ejecutar una a la vez, es algo que tiene que ver con los hilos cosa que se ve un poco en la seccion de Gráficas y Animación. Bueno, recordando: todo método o función static se declarará de la siguiente forma:

```
public static nombreFunción(argumentos) {
  return bla bla bla bla.. .
}
```

El public es importante porque si no: ¿que caso tendría crear cientos de funciones dentro de una clase (como biblioteca de funciones) si no son accesibles desde afuera?.

Organización de la Documentación de Java (API)

El API (Aplications Programming Interface) de Java contiene para qué sirve cada una de las clases, subclases y las funciones predefinidas dentro del JDK, existen muchisimas funciones listas para utilizarse en vez de que tengas que "reinventar la rueda", en la introducción así como en la sección de RealJ están las ligas correspondientes para obtener el Java API desde Internet de forma gratuita.

Existe una clase que particularmente ilustra el tipo de funciones de las que se puede disponer, la clase Graphics. En esta clase puedes encontrar funciones para trazar lineas como drawLine(), para trazar circunferencias drawCircle(), para dibujar cadenas de caracteres drawString() entre otras, cada función viene con una breve descripción que incluye el tipo de datos que recibe como argumento y en caso de que devuelvan un valor de que tipo es este Cada una de las clases puede contener

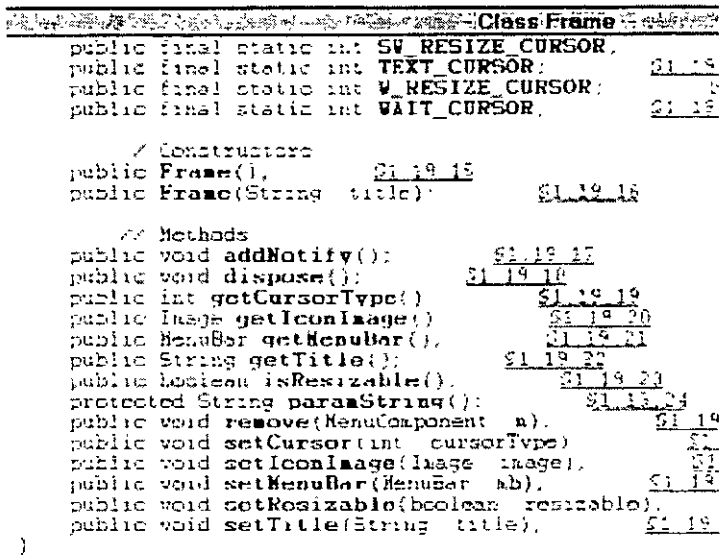
Fields **(características o variables globales)**
Constructors **(inicializadores, generalmente siempre hay más de uno)**
Methods **(comportamiento o posibles cambios que puedes realizar desde afuera)**

Los Fields no es otra cosa que una forma distinta de llamar a una variable global, puede ser de cualquier tipo conocido desde enteros, letras o bolean (generalmente están declarados como "final"). Los constructors obviamente son las funciones constructoras que no devuelven ningún tipo, cada clase puede tener muchos constructores sobrecargados para usar el que más convenga a nuestras necesidades pero recuerda que solo se

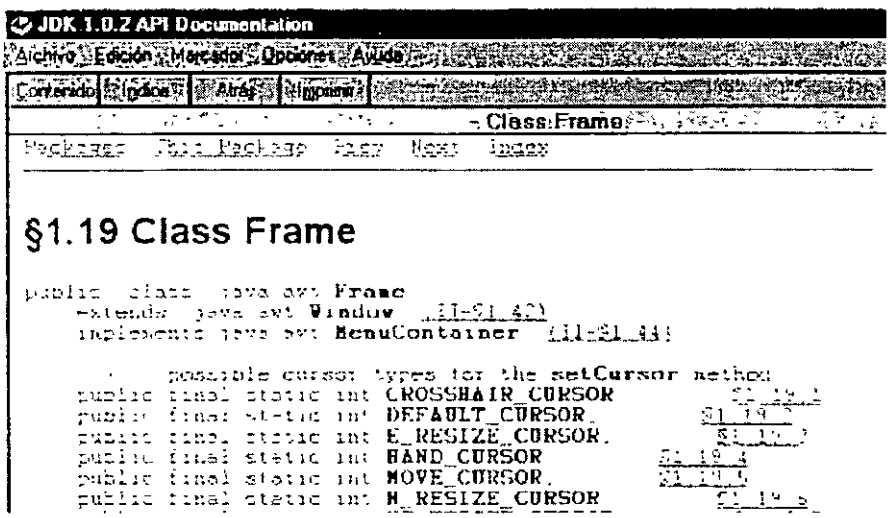
puede usar uno solo de ellos a la vez. Methods son todos los métodos o funciones dentro de la clase, recuerda que los únicos que puedes usar son los declarados como "public", también recuerda que puedes acceder a los métodos estáticos o static directamente sin la necesidad de usar el constructor de la clase (estas son bibliotecas de funciones). Esto es lo fundamental en cuanto a la documentación de Java, a continuación viene un pequeño ejemplo pero yo te aconsejo revisar algunos libros para ver como acceder y cómo usar las clases más especializadas que manipulan cadenas, números, archivos, etc

Ejemplo: clase Frame

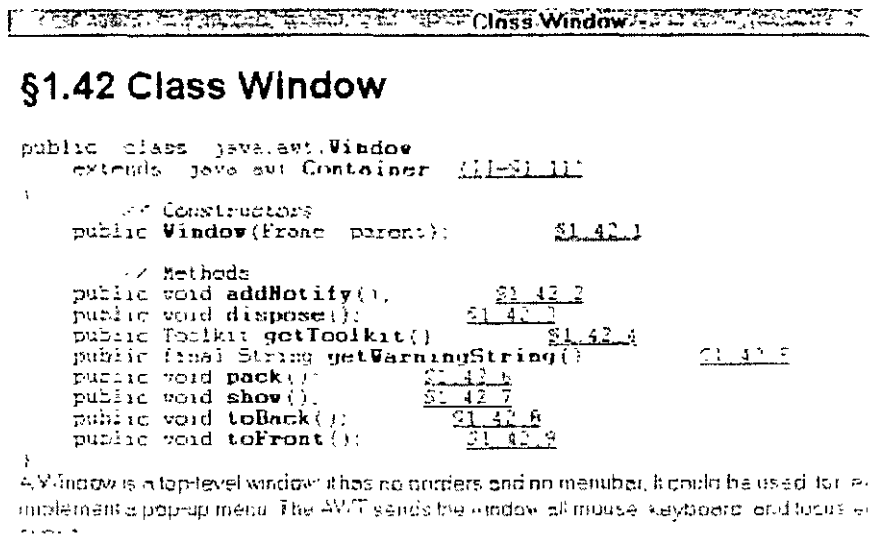
Vamos a utilizar esta clase en la sección de Aplicaciones por eso creo que es muy conveniente explicar a grandes rasgos sus caracteristicas y de que manera usarías la Java API para investigar como utilizarla. Primero localicemos la clase dentro de la API, después veamos su contenido:



Este screenshot es de la API del JDK 1.0.2 (en versión de WinHelp), así que quizás su Java API sea un poco más actualizada y quizás tenga algunas funciones además de las que aquí se muestran (también debe ser similar a la version HTML), en la parte de arriba se ven algunos "fields", pero como pueden ver solo tiene dos constructores y varios métodos (cuyo propósito esta explicado detalladamente). En la parte de abajo se ve una breve explicación de lo que hace esta clase, "A frame is a top-level windows with a title and border", un frame es una ventana de alto nivel que tiene un titulo y un borde Por poner un ejemplo sencillo, el método "setTitle(String title)" sirve para fijar el titulo de la ventana y el método "getTitle()" regresaria el String del titulo Al principio de la clase tenemos lo siguiente.



Aqui hay algo importante, podemos ver por la palabra clave "extends" que la clase Frame es una subclase de la clase Window y que también implementa la clase Menu Container, esto significa que si nos dirigimos ahora a observar el contenido de la clase Window todas las variables y funciones que se encuentren en ella también estarán disponibles (por herencia) también desde nuestra clase Frame. La clase Window tiene lo siguiente:



Class Window

§1.42 Class Window

```
public class java.awt.Window
    extends java.awt.Container
{
    // Constructors
    public Window(Frame parent);

    // Methods
    public void addNotify();
    public void dispose();
    public Toolkit getToolkit()
    public final String getWarningString()
    public void pack();
    public void show();
    public void toBack();
    public void toFront();
}
```

A Window is a top-level window that has no borders and no menubar. It could be used for a pop-up menu. The AWT sends the window all mouse, keyboard, and focus events that occur within its bounds.

Nuevamente tenemos una breve explicación de lo que exactamente es la clase Window, podemos ver un constructor que recibe como argumento un objeto Frame y varios métodos, entre ellos show() (mostrar) toBack() (enviar atrás) toFront() (enviar al frente), etc puedes ver el propósito de cada uno de ellos para aprender más, nuevamente: si esta clase es la superclase de Frame significa que estas funciones están disponibles también desde cualquier "Frame", incluso esta clase tiene a su vez otra superclase, la clase Container, si la exploras verás que en ella hay aún más métodos, es evidente que las subclases siempre serán mucho más grandes que su superclases correspondientes y no porque tengan más código sino porque cada vez heredarán más y más funciones de sus predecesores. Para utilizar estas clases solo bastará experimentar con sus constructores y con sus diversos métodos públicos.

Gráficas, Hilos y Animación

- Gráficas
- Hilos
- Controlando un Hilo Pausa/Reanudación
- Hilos con Interfaz Runnable
- Animación
  Pelota rebotando en caja virtual

Incluir algunas gráficas sencillas y animación en un programa lo hace más atractivo a la vista, afortunadamente en Java es sumamente fácil hacerlo, el propósito de esta guia es dar una introducción básica a estos divertidos conceptos sin profundizar excesivamente en ellos para hacerlo menos tedioso y menos complicado, siempre procuro ser lo más práctica que puedo por eso omitiré muchos detalles y otras aplicaciones que se les puede dar a los hilos para centrarnos exclusivamente en la animación.

La clase Graphics

La clase Graphics contiene una serie de funciones útiles que podemos utilizar desde cualquier programa Java mientras dispongamos de un contexto gráfico (un lugar para dibujar) como un frame o un panel cosas que veremos más adelante. Sin embargo esta clase es únicamente para gráficas muy sencillas tales como: lineas, círculos, elipses, rectángulos, polígonos. Aunque Java es un lenguaje adaptado extensamente para aplicaciones gráficas no todas se relacionan con dibujos directos sobre la pantalla sino más bien con el tratamiento de imágenes por eso las capacidades de la clase Graphics no son tan complejas, de hecho es una de las clases más fáciles de usar que ejemplifica de manera perfecta el uso de algunas funciones predefinidas de manera muy similar a lo que ofrecía la clase Math

En todas nuestras applets hemos estado importando el paquete java.awt.*; este paquete contiene a la clase Graphics, los paquetes son grupos de clases que ya se encuentran escritas en el lenguaje y si a ti te gusta mucho pasarte horas programando probablemente en un futuro cercano también crearás tus propios paquetes cosa que te invito a investigar más a fondo en un libro de Java más serio que este curso.

Este es un resumen de las funciones públicas más importantes de la clase Graphics, si deseas conocerlas todas y experimentar con ellas más a fondo puedes consultarlas con toda libertad en el API de Java, tan solo busca la clase Graphics

Funciones Graphics	Descripción
drawString(String string,int x,int y);	Dibuja una cadena en las coordenadas (x,y)
drawChars(char chars[], int offset, int number, int x, int y);	Dibuja los caracteres contenidos en un arreglo de tipo char y comienza a leer el arreglo desde el índice offset y a partir de él lee number cantidad de caracteres.
drawRect(int x, int y, int width, wint height);	Dibuja un rectángulo vacío a partir del punto (x,y) con un ancho width y un alto height.
fillRect(int x, int y, int width, int height);	Dibuja un rectángulo relleno a partir del punto (x,y) con un ancho width y un alto height.
clearRect(int x, int y, int width, int height);	Dibuja un rectángulo vacío a partir del punto (x,y) con un ancho width y un alto height.
draw3DRect(int x, int y, int width, int height);	Dibuja un rectángulo sombreado vacío a partir del punto (x,y) con un ancho width y un alto height.
fill3DRect(int x, int y, int width, int height, boolean b);	Dibuja un rectángulo relleno sombreado a partir del punto (x,y) con un ancho width y un alto height.
drawOval(int x, int y, int width, int height);	Dibuja un óvalo a partir del punto (x,y) con un ancho width y un alto height, si width=height se tratará de un círculo.
fillOval(int x, int y, int widht, int height);	Dibuja un óvalo relleno a partir del punto (x,y) con un ancho width y un alto height, si width=height se tratará de un círculo
drawArc(int x, int y, int width, int height, int.	Dibuja un arco vacío partiendo del punto (x,y) con un ancho

startAngle, int arcAngle);	width y un alto height comenzando en el ángulo startAngle y agregando un arco a lo largo de arcAngle
fillArc(int x, int y, int width, int height, int startAngle, int arcAngle);	Dibuja un arco relleno partiendo del punto (x,y) con un ancho width y un alto height comenzando en el ángulo startAngle y agregando un arco a lo largo de arcAngle
drawPolygon(int xPoints[], int yPoints[], int points);	Crea un polígono vacío utilizando los puntos contenidos en los arreglos xPoints[] y yPoints[]. points representa el número de puntos que se leerán de los arreglos ya mencionados
fillPolygon(int xPoints[], int yPoints[], int points);	Crea un polígono relleno utilizando los puntos contenidos en los arreglos xPoints[] y yPoints[]. points representa el número de puntos que se leerán de los arreglos ya mencionados
copyArea(int x, int y, int width, int height, int dx, int dy);	Copiar un el área contenida entre (x,y) y el ancho y alto especificados, el resultado de esta copia se coloca en (dx,dy), el cual es medido con respecto a (x,y)

Nota: Para usar cualquiera de las funciones anteriores primero debes crear un tipo Graphics y después acceder a este mediante el operador punto, recuerda que siempre nombrábamos en los ejemplos a nuestro tipo Graphics como "g" y después llamábamos a sus funciones como: g.drawString("Hola!", 20, 20); anteponiendo el nombre de nuestro tipo (en este caso "g") y el llamado a la función deseada.

Todas las funciones son bastante simples y es muy fácil experimentar con ellas, solo crea los argumentos que te pide y después pásalos en la función, por ejemplo:

```
public void paint(Graphics g) {
   g drawOval(50, 50, 35, 20);
}
```

Pero también sería válido poner lo anterior como:

```
public void paint(Graphics miDibujo) {
   miDibujo.drawOval(50, 50, 35, 20);
}
```

Ahora, la función paint siempre se llama al principio automáticamente desde cualquier applet para dibujar pero no es necesario que pongas todas tus funciones gráficas en ella, por ejemplo lo siguiente es válido:

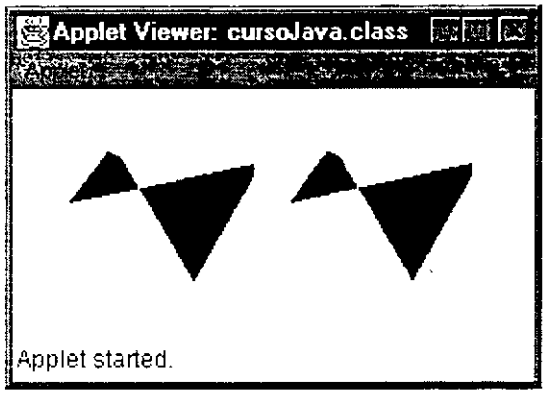
```
public void paint(Graphics g) {
   dibujar( g );
}

public void dibujar(Graphics n) {
   n drawOval(50, 50, 35, 20);
   n drawString("Este es un óvalo!", 50, 100);
}
```

Primero nuestra función paint llama a la función dibujar pasándole el argumento Graphics requerido por esta, luego la función dibujar ejecuta su contenido dibujando el óvalo y la cadena, mientras se pase el argumento de tipo Graphics requerido todo se dibujará bien, esto ayuda un poco cuando quizás son demasiadas las cosas que quieres dibujar, así las puedes separar para controlarlas y organizarlas por separado. El siguiente ejemplo demuestra el uso de las dos últimas funciones de la tabla, primero dibujamos un polígono en base a los datos de un arreglo y después copiamos dicho polígono a otra sección de la pantalla.

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

public class ejemploGraphics extends Applet {

int puntosX[] = { 27, 47, 54, 90, 124 };
int puntosY[] = { 57, 30, 34, 96, 36 };
```

```
public void paint(Graphics g) {
  dibujar( g );
}

public void dibujar( Graphics k) {
  k fillPolygon( puntosX, puntosY, 5 ),
  k.copyArea( 10, 10, 110, 110, 110, 0 ),
}

}
```

La salida del Applet es:



El dibujo es algo intrincado pero eso es porque escribí los puntos al azar, ya se que es feo pero no sean tan duros conmigo ok?. Bueno, de forma predeterminada todas las gráficas se dibujan de color negro pero podemos elegir el color para dibujar fácilmente, la clase Color contiene muchas variables public que son también de tipo Color que almacenan tres valores (en formato RGB, R-rojo, G-verde, B-azul) que indican en una escala del 0 al 255 la proporción de cada color para formar el color actual (combinando) de esta manera podríamos formar nuestros propios colores pero es mucho mejor tomar los que ya están creados como por ejemplo:

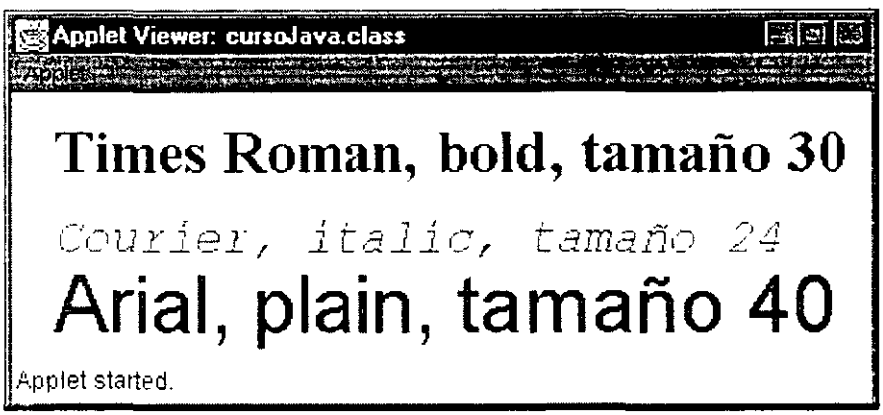
```
public final static Color black;
public final static Color blue;
public final static Color cyan,
public final static Color darkGray;
public final static Color gray,
public final static Color green;
public final static Color lightGray,
public final static Color magenta,
public final static Color orange,
public final static Color pink;
public final static Color red,
public final static Color white,
public final static Color yellow,
```

Son "public" para que podamos acceder libremente a ellos, son "final" para que nadie los modifique o sea que son colores estandarizados y finalmente son "static" para que cualquier subclase de color acceda a ellos sin necesidad de que cada una de ellas cree su propia copia en memoria, las variables "static" son la misma para todas las subclases, la comparten Veamos un ejemplo de como cambiar los colores y de paso veamos como especificar el tipo de letras de exhibición, ambas cosas son muy similares y sencillas

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

public class ejemploColorFonts extends Applet {

Font letra1 = new Font("TimesRoman", Font.BOLD, 30 ),
Font letra2 = new Font("Courier", Font ITALIC, 24 ),
Font letra3 = new Font("Arial", Font.PLAIN, 40 ),
```

```
public void paint(Graphics g) {
  g.setColor( Color.blue );
  g.setFont( letra1 );
  g.drawString( "Times Roman, bold, tamaño 12", 20, 40 );
  g.setColor( Color.magenta );
  g.setFont( letra2 );
  g.drawString( "Courier, italic, tamaño 24", 20, 80 );
  g.setColor( Color.red );
  g.setFont( letra3 );
  g.drawString( "Arial, plain, tamaño 14", 20, 120 );
}

}
```

La salida del Applet anterior es:



Al principio del applet se declaran tres variables de tipo Font (de la clase Font), este tipo sirve para seleccionar y configurar un tipo de letra en particular, como puedes ver el primer argumento del constructor es de tipo String, en esta cadena se escribe el tipo de letra tal como aparece en el sistema o sea en la carpeta de Fonts de Windows, en teoria ahi puedes ver todas las fuentes que tiene tu computadora pero ten en cuenta que estas fuentes pueden variar de sistema en sistema por lo tanto procura no utilizar las que no son comunes El segundo argumento del constructor de Font te pide acceder a una de tres constantes de la misma clase Font mediante el operador punto: BOLD, ITALIC ó PLAIN, recuerden que este tipo de valores constantes también son conocidos como "Fields" o caracteristicas del objeto, pero claro no dejan de ser solo variables constantes, las podemos acceder de esta forma tan arbitraria solo porque son public y además static. El tercer y último argumento de este constructor es el número entero que representa el tamaño que queremos utilizar·

```
Font letra1 = new Font("TimesRoman", Font.BOLD, 30 );
Font letra2 = new Font("Courier", Font.ITALIC, 24 );
Font letra3 = new Font("Arial", Font.PLAIN, 40 );
```

Después tenemos nuestra función paint

```
public void paint(Graphics g) {
  g.setColor( Color.blue );
  g setFont( letra1 ),
  g.drawString( "Times Roman, bold, tamaño 12", 20, 40 );
  g setColor( Color magenta ),
  g setFont( letra2 );
  g drawString( "Courier, italic, tamaño 24", 20, 80 );
  g setColor( Color.red ),
  g.setFont( letra3 ),
  g drawString( "Arial, plain, tamaño 14", 20, 120 ),
}
```

Aquí fijamos el color mediante g.setColor(Color blue), sucede algo muy similar al fijar la letra con. g.setFont(letra1), generalmente en todas las clases predefinidas existentes las funciones que nos permiten modificar un valor sea cual sea empiezan con un "set" (fijar) y todas las que nos permiten obtener algún valor sin importar cual

sea empiezan con un "get" (obtener). Regresando al ejemplo primero fijamos el color y después fijamos la letra por lo tanto la próxima figura que se dibuje mediante **cualquiera de las funciones de la clase Graphics** adoptará estos valores fijados, tal como se ve en la salida del applet.

Hilos

Sin Java no podríamos revisar un poco sobre hilos a estas alturas puesto que no todos los lenguajes (tales como C ó C++) disponen de esta característica de forma implícita (o predeterminada) en el lenguaje. Los hilos nos permiten crear ciertas cosas interesantes, un hilo es como una línea de ejecución independiente de otras Hasta el momento hemos creado programas cuya línea de ejecución va saltando de función en función o de clase en clase pero en todos los casos solo una sección de código se ejecutaba a la vez, en teoría con hilos puedes ejecutar varias cosas al mismo tiempo. La realidad es que sistemas operativos tal como Windows ofrece pedacitos de tiempo de ejecución, o sea que cada hilo espera su oportunidad para ocupar tiempo del procesador. Esto significa que realmente no se ejecutan dos o más tareas al mismo tiempo sino que rolan tan rápido su tiempo disponible del procesador que pareciera que sí se ejecutan todos a la vez.

En este curso utilizamos el enfoque más practico y directo para trabajar con hilos. la interfaz Runnable que literalmente transforma nuestra clase en un Hilo Existen otras formas ligeramente más complicadas en las que podemos aplicar los hilos sin embargo estas son de muy poco interés (y poco prácticas para un principiante) por lo que he decidido no extender el estudio de los hilos, este tipo de temas interesarían más a alguien que piense programar applets o aplicaciones para entornos de red o para el procesamiento de transacciones y cosas por el estilo. Entender un solo y único hilo es muy sencillo: imagina que es tal como un reloj que nos permite programar intervalos de tiempo para ejecutar ciertas secciones de nuestro código El corazón de todo hilo se encuentra en su función (o método) run, es ahí donde se colocan las instrucciones que se ejecutarán una y otra vez hasta que decidamos detenerlo permanentemente, existen dos formas para usar la función run, una es heredar la clase Thread y sobrescribir una nueva función run de acuerdo a nuestras necesidades (porque la clase Thread tiene la suya, así ocultaremos la función run() de la superclase Thread para usar el nuestro) y la otra que es mucho más sencilla es usando la Interfaz Runnable.

Hilos con Interfaz Runnable

Lo mejor para comprender esto es mostrar un ejemplo y explicarlo paso por paso:

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

public class ejemploHilo extends Applet implements Runnable {

 //variables
 Thread hilo;
 int segundos;

 public void init() {
   //inicializar hilo
   hilo = new Thread(this);
   //comenzar ejecución del hilo
   hilo.start();
 }

//función run, corazón del hilo
 public void run() {
  while (true) {
    segundos++;
     repaint();
   try { hilo.sleep(1000); }
   catch (InterruptedException e) { }
  }
 }

//exhibir la variable segundos en la pantalla
  public void paint(Graphics g) {
     g drawString( "Segundos: " + segundos, 20, 20);
```

```
  }
}
```

En la primera línea del applet implementamos la interface Runnable mediante implements, una interface es muy similar a una clase abstracta (la que solo incluye el nombre y tipo de una función, sin cuerpo), de hecho la interface que estamos implementando copiada directamente tal como viene en el JDK es la siguiente:

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

Como ven es muy simple, las interfaces a diferencia de las clases convencionales no contienen variables y tienen casi la misma función que las clases abstractas obligar a las clases que las implementan a incluir los métodos que estas tienen, si desean más información acerca de las interfaces y de las excepciones (ambos se aplican en este ejemplo) pueden ir revisar la sección de Temas Adicionales.

Suponiendo que ya comprendiste qué es lo que hace exactamente una interfaz ahora obligatoriamente debemos poner un método o función run() en nuestra applet porque la función run() es la que realiza todo el trabajo de un hilo, necesitamos implementarla de esta manera si no deseamos tener que heredar la clase Thread completa, en este caso es más práctico hacer esto último que heredar Thread. En la función init() tenemos:

```
public void init() {
  //inicializar hilo
  hilo = new Thread(this),
  //comenzar ejecución del hilo
  hilo start();
}
```

Thread tiene varios constructores como puedes comprobarlo al revisar el API del JDK, el que usamos aquí es el que recibe como argumento un objeto de tipo Runnable, ¿porque está ese **this** allí? porque al implementar Runnable nuestra clase también se convierte en un objeto de tipo Runnable y entonces lo podemos pasar como argumento para crear el Thread. De hecho el código anterior sería lo mismo que poner:

```
public void init() {
  //inicializar hilo
  hilo = new Thread(ejemploHilo),
  //comenzar ejecución del hilo
  hilo.start();
}
```

Porque "ejemploHilo" que es el nombre de nuestra clase al implementar la interfaz Runnable también se convierte en un objeto de tipo "Runnable", la función start() se encarga de arrancar el hilo (realmente lo que hace es llamar a run()) En la función run() tenemos:

```
//función run, corazón del hilo
public void run() {
 while (true) {
  segundos++,
  repaint();
  try { hilo.sleep(1000), }
  catch (InterruptedException e) { }
 }
}
```

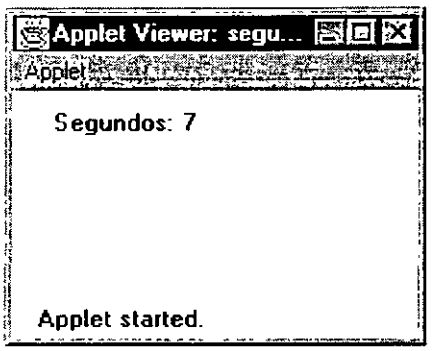
Esta función es de tipo void, lo que significa que no devuelve ningún valor, solo ejecuta lo que tiene dentro de su cuerpo (llaves) En ella tenemos un ciclo while con la condición de detenimiento "true", esto significa que este while nunca se va a detener puesto que solo lo hace si recibe un "false" y en este caso "true" no es una variable sino un valor bolean constante que nunca cambiara. En este while se encuentra el verdadero trabajo del hilo: incrementar el valor entero "segundos" y llamar a repaint(). La función repaint() redibuja todo lo que esté en la pantalla del applet ¿porqué hacer esto? porque si incrementamos el valor de "segundos" deseamos ver tal cambio **actualizado en la pantalla** Finalmente después del repaint() viene el manejo de una excepción mediante un try y un catch, si deseas más información acerca de las excepciones puedes ir a la sección de Temas Adicionales pero

igual es un tema relativamente simple que no es necesario explicar con mucho detalle, confío en que no tendrás mayor problema aunque si no deseas hacerlo simplemente ignora el try y el catch:

```
try { hilo.sleep(1000); }
  catch (InterruptedException e) { }
```

Ok, daré una breve explicación, en pocas palabras el try prueba ejecutar la función: hilo.sleep(1000) y si esta se ejecuta exitosamente todo continúa normal como si no existiera ni el try ni el catch, pero, si las cosas fallan se lanza una excepcion de tipo InterruptedException, el "catch" se encarga de atrapar la excepción aunque en este caso no se toma ninguna medida ni para manejar la excepción ni para exhibirla, esta manera de atrapar excepciones sin hacer nada al respecto no es la correcta, este curso aún mantiene un carácter introductorio por lo que este tipo de cuestiones se pasan por alto pero que quede claro que es por mantener en el centro de nuestra atención los hilos no la manera apropiada de manejar las excepciones que estos pueden lanzar.

Ahora, la función hilo.sleep(1000) Esta función se encarga de dormir al hilo 1000 milisegundos, es decir, un segundo, después de este tiempo se regresa al inicio del while y se comienza el ciclo una y otra vez. La última función paint(Graphics g) se encarga de llamar a un g.drawString() que exhibe en forma de cadena el valor del entero "segundos" en las coordenadas 20,20 del applet, una vez compilado y cargado en el documento html esta es su salida·



Pausando y reanudando la ejecución de un Hilo (opcional)

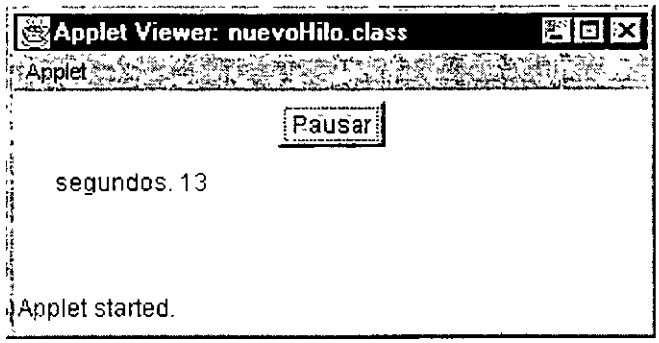
Nota: Aquí existe un problema **inevitable** de continuidad del curso, es decir que en este ejemplo **y solo en este!** se ven temas que no se han tratado anteriormente pero que no se pueden ver de otra manera. En este ejemplo se utilizan eventos y un botón así que lo ideal sería que primero estudiaras el siguiente capítulo (que incluye eventos y botones) y después regresaras a este ejemplo, ese es mi consejo aunque si sabes un poco del tema de antemano pues chécalo.

Ahora veamos como pausar y reanudar la ejecución de un hilo. Para esto recurriremos al modificador synchronized, a las funciones wait() y notify() Este programa al iniciar comienza la ejecución de un hilo y una vez iniciado un botón permite pausar o reanudar la ejecución del hilo, es muy similar al anterior solo que puedes pausar/reanudar la ejecucion, esta técnica se utiliza en el ejemplo general del Tetris, así que si deseas comprender cómo es que funciona estudia bien el siguiente codigo y experimenta con él.

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

public class nuevoHilo extends Applet implements Runnable, ActionListener {

 Button i = new Button("Pausar");
 int segundos = 0;
 Thread h = new Thread(this);
 boolean hiloSuspendido = false;

 public void init() {
  add(i);
  i.addActionListener(this);
  h.start();
 }
```

```
  public synchronized void actionPerformed(ActionEvent e) {
     hiloSuspendido = !hiloSuspendido;
      if (hiloSuspendido == false) {
         notify();
  }
     if(hiloSuspendido) {
      i.setLabel("Continuar");
     } else
      i.setLabel("Pausar");

  }

 public void paint(Graphics g) {
  g drawString("segundos: " + segundos, 20, 50),
 }

 public void run() {
     while (true) {
       try {
        ++segundos,
         Thread.sleep(1000),

          synchronized(this) {
            while (hiloSuspendido == true)
              wait();
          }

       } catch (InterruptedException e){
       }
       repaint();
     }
  }


}
```

Su salida es:



Primero importamos los paquetes pertinentes, se requiere el de evnetos ya que utilizaremos un botón para pausar o reanudar el hilo mediante la interfaz ActionListener, del mismo modo también implementamos la interfaz Runnable:

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

public class nuevoHilo extends Applet implements Runnable, ActionListener {
```

Ahora declaramos un botón, una variable entera que llevará la cuenta del tiempo, un hilo y una variable bolean que llevará el estado del hilo, si el hilo esta suspendido entonces la variable hiloSuspendido valdrá true en caso contrario será false.

```
 Button i = new Button("Pausar"),
 int segundos = 0;
 Thread h = new Thread(this);:
 boolean hiloSuspendido = false;
```

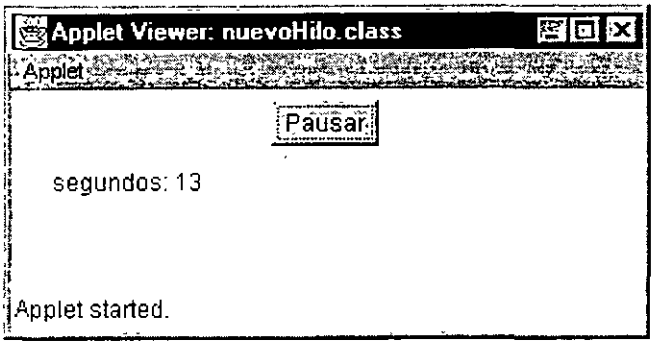
En el siguiente método init() agregamos el botón y le anexamos el Listener, también inicializamos el hilo mediante su método start()

```
 public void init() {
  add(i),
  i.addActionListener(this),
  h start(),
 }
```

El siguiente método se encarga de manejar el evento de presionar el botón, como solo es un botón no es necesario recurrir al método getSource() de la clase ActionEvent El modificador synchronized le indica al hilo que por muy rapido que se ejecute antes debe esperar y atender las modificaciones que este metodo actionPerformed pueda realizar sobre las variables globales del programa, primero se invierte el valor bolean almacenado en hiloSuspendido de modo que si es false se convierta en true con el operador boleano '!' y viceversa, asi si el hilo esta suspendido (true) se reanuda con una llamada a la función notify() que se encarga de notificar al hilo que puede continuar y si el hilo esta activo (false) únicamente se invierte su variable bolean de control sin realizar ningun cambio por lo menos en este método actionPerformed (aunque veremos más adelante que este cambio afecta al metodo run). Por último se utiliza la variable de control para marcar el botón mediante setLabel(), indicando si el hilo esta suspendido o bien si esta activo

```
   public synchronized void actionPerformed(ActionEvent e) {
      hiloSuspendido = !hiloSuspendido;
       if (hiloSuspendido == false) {
           notify(),
   }
      if(hiloSuspendido) {
       i.setLabel("Continuar");
      } else
       i.setLabel("Pausar");
```

Su salida es:



Primero importamos los paquetes pertinentes, se requiere el de evnetos ya que utilizaremos un botón para pausar o reanudar el hilo mediante la interfaz ActionListener, del mismo modo también implementamos la interfaz Runnable:

```
import java.awt.*,
import java.applet.*;
import java.awt.event.*;

public class nuevoHilo extends Applet implements Runnable, ActionListener {
```

Ahora declaramos un botón, una variable entera que llevará la cuenta del tiempo, un hilo y una variable bolean. que llevará el estado del hilo, si el hilo esta suspendido entonces la variable hiloSuspendido valdrá true en caso contrario será false.

```
 Button i = new Button("Pausar"),
 int segundos = 0;
 Thread h = new Thread(this);;
 boolean hiloSuspendido = false;
```

En el siguiente método init() agregamos el botón y le anexamos el Listener, también inicializamos el hilo mediante su método start()

```
 public void init() {
  add(i),
  i addActionListener(this);
  h.start();
 }
```

El siguiente método se encarga de manejar el evento de presionar el botón, como solo es un botón no es necesario recurrir al método getSource() de la clase ActionEvent. El modificador synchronized le indica al hilo que por muy rapido que se ejecute antes debe esperar y atender las modificaciones que este método actionPerformed pueda realizar sobre las variables globales del programa, primero se invierte el valor bolean almacenado en hiloSuspendido de modo que si es false se convierta en true con el operador boleano '!' y viceversa, asi si el hilo esta suspendido (true) se reanuda con una llamada a la función notify() que se encarga de notificar al hilo que puede continuar y si el hilo esta activo (false) únicamente se invierte su variable bolean de control sin realizar ningún cambio por lo menos en este método actionPerformed (aunque veremos más adelante que este cambio afecta al método run). Por último se utiliza la variable de control para marcar el botón mediante setLabel(), indicando si el hilo esta suspendido o bien si esta activo

```
   public synchronized void actionPerformed(ActionEvent e) {
      hiloSuspendido = !hiloSuspendido,
       if (hiloSuspendido == false) {
           notify();
   }
      if(hiloSuspendido) {
       i setLabel("Continuar");
      } else
       i.setLabel("Pausar");
```

```
}
```

La siguiente es la funció más sencilla: paint(), solo se encarga de mostrar el valor actual en pantalla para la variable "segundos" en las coordenadas (20, 50).

```
public void paint(Graphics g) {
 g.drawString("segundos: " + segundos, 20, 50);
}
```

Por último tenemos el método run, lo más importante. En su interior tenemos un ciclo while infinito (propio de todo hilo) y el incremento para la variable segundos que se ajusta perfectamente a la equivalencia de 1000 milisegundos para dormir, todo intento de dormir un hilo puede generar una excepción por lo que todo se encierra en un try-catch (para atraparla en caso de que se genere). En el interior de todo esto tenemos algo parecido a una función llamada synchronized que en realidad es una propiedad del hilo, este pseudo-método 'synchronized' es parte importante del hilo 'h' ya que continuamente (en cada ciclo) se ejecuta su interior, de hecho esta es la sección de código que se sincroniza con el método actionPerformed (declarado también como synchronized) esto significa que por muy rapido que vaya el ciclo del hilo esta sección de código esperará a que se termine de ejecutar el actionPerformed y en general cualquier otro método declarado como synchronized. ¿Y porqué debe esperar esta sección a actionPerformed? porque el método actionPerformed es capaz de modificar la variable boleana de estado del hilo nombrada 'hiloSuspendido', el hilo debe saber si esta suspendido o no y esto se logra mediante esta pequeña sección de código. Si hiloSuspendido vale true entonces se llama al método wait() y este hilo se autodetiene, pero si hiloSuspendido vale false realmente no sucede nada, el hilo continúa Noten que el while dentro de synchronized no lleva llaves pero el synchronized si, el wait() únicamente le dice al hilo que espere. Despertar el hilo pues entonces queda exclusivamente bajo la responsabilidad del actionPerformed y detenerlo queda en el propio hilo (su método run()).

```
public void run() {
    while (true) {
      try {
       ++segundos;
         h sleep(1000),

         synchronized(this) {
            while (hiloSuspendido == true)
               wait(),
         }

      } catch (InterruptedException e){
      }
      repaint();
    }
  }

}
```

Animación

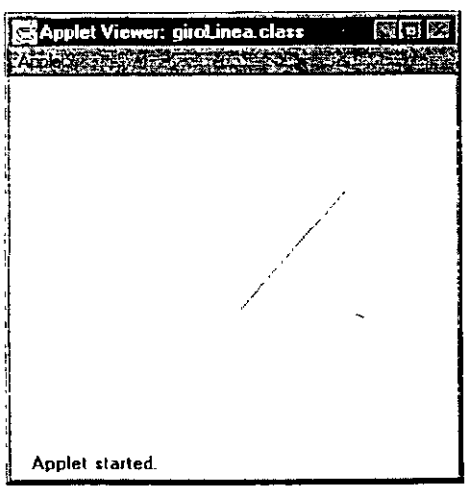
¿Qué es animación? Animar es dar la sensación de movimiento a una imagen mediante diversas técnicas. Una de tantas es cambiar varios cuadros por segundo de modo que se aparente un cambio de posición, por eso la animación está ligada tan intimamente con el tiempo. Es muy importante que comprendas muy bien el sistema cartesiano de coordenadas y las fórmulas de velocidad para lograr algo interesante, también ayuda mucho la trigonometria y las fórmulas básicas de aceleración

La siguiente applet es para ilustrar cómo se realiza una animación en Java, se puede simplificar mucho más pero considero que por claridad para aquellos que realmente son principiantes debe explicarse de esta forma. Tengo que recalcar que a menos que hayas cursado la secundaria podrás comprender bien el ejemplo porque incluye unos cuantos conceptos trigonométricos y de fisica, pero claro, no dudo que alguno de ustedes aún asi lo pueda comprender bien.

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

```
public class giroLinea extends Applet implements Runnable {

Thread hilo;
double decima_segundo;
int xPos;
int yPos;

public void init() {
  hilo = new Thread(this);
 decima_segundo = 0,
 hilo.start();
}

public void run() {
 while (true) {
  decima_segundo += 0 1;
  repaint();
  try { hilo sleep(100), }
  catch (InterruptedException e) { }
 }
}

public double calcularPosX(double tiempo) {
  return 100*Math cos(1*tiempo);
}

public double calcularPosY(double tiempo) {
  return (100*Math sin(1*tiempo));
}

public void paint(Graphics g) {
  xPos = (int) calcularPosX(decima_segundo);
  yPos =  (int) calcularPosY(decima_segundo);
 g.drawLine(150,150,150+xPos,150+yPos);
}
}
```

Salida del Applet:



Esta es un applet que dibuja una linea que gira alrededor de la coordenada (150,150) del applet. Su longitud (o radio) es de 100, recuerden que las unidades que manejan las applets son pixeles, ahora la explicación.

```
Thread hilo;
double decima_segundo;
```

```
int xPos;
int yPos;

public void init() {
  hilo = new Thread(this);
  decima_segundo = 0,
  hilo.start(),
}
```

Aquí como siempre se declaran las variables a utilizar y después se inicializan dentro del init(), recuerda que el init() es la primera función que se ejecuta dentro del applet, por eso es bueno inicializar aquí las variables (¿cuando me cansaré de repetirlo?).

```
public void run() {
 while (true) {
  decima_segundo += 0 1;
  repaint():
  try { hilo.sleep(100); }
  catch (InterruptedException e) { }
 }
}
```

Este run() incrementa en intervalos de 0 1 nuestra variable de tiempo, nota que también el hilo duerme 0.1 segundos en cada intervalo para mantenerlos ambos iguales (porque 0.1 segundos = 100 milisegundos), la variable decima_segundo será algo así como una variable de control que relaciona el hilo y el tiempo.

```
public double calcularPosX(double tiempo) {
  return 100*Math.cos(1*tiempo);
}

public double calcularPosY(double tiempo) {
  return (100*Math.sin(1*tiempo));
}
```

Estas dos funciones se encargan de recalcular en función del tiempo la posición del punto que se mueve. Me gusta usar double para los cálculos porque es mayor exactitud, la animación se logra usando la fórmula de velocidad angular

angulo = velocidad x tiempo

Usando una velocidad angular de 1 radián por segundo y tomando el tiempo de decima_segundo (más adelante) Después, la posición se calcula con las funciones trigonométricas x = R*cos(angulo), y = R*sen(angulo), donde R es el radio de nuestra línea, tomamos las funciones seno y coseno de la biblioteca de funciones estáticas (static) contenidas en la clase Math

```
public void paint(Graphics g) {
  xPos = (int) calcularPosX(decima_segundo),
  yPos =  (int) calcularPosY(decima_segundo),
 g drawLine(150,150,150+xPos,150+yPos);
}
}
```

Finalmente el paint llama a las funciones de cálculo y exhibe la línea Puedes cambiar la velocidad de la animación ajustando los valores de incremento para decima_segundo y el tiempo que duerme el hilo:

```
  decima_segundo += 0 1,  //si lo haces mas pequeño se suaviza la animación
  hilo.sleep(100),        //pero tambien tendrias que ajustar este valor
```

Recuerda que el argumento esta en milisegundos y el tiempo en decimas por ejemplo:

Si cambias 100 a 1000 entonces 0 1 debe cambiarse a 1
Si cambias 100 a 10 entonces 0 1 debe cambiarse a 0.01

También puedes cambiar la velocidad angular y su dirección cambiando su signo para experimentar, no dudes en hacerlo!.

Pelota en caja virtual

Ahora vamos a hacer una pelota que rebote dentro de una caja "virtual", para lograrlo debes utilizar las fórmulas de tiro parabólico aprendidas en física de preparatoria, la pelota la puedes dibujar mediante la función:

g.fillOval(int x, int y, int ancho, int alto);

Recuerda que esta debe ir dentro de una función paint que recibe un objeto Graphics g como argumento. Cada rebote implica un cambio de velocidad aunque sería un poco más sencillo utilizar un enfoque orientado a objetos para destruir y crear la pelota cada vez que choca con la pared virtual.

Para la coordenada x la fórmula de desplazamiento es:

x = xIni + Vix*t,

Donde xIni es coordenada x de la posición inicial, Vix es la velocidad inicial a lo largo del eje x que por ser tiro parabólico será siempre igual que la velocidad final y t evidentemente es el tiempo.

Para la coordenada y la fórmula de desplazamiento considerando la aceleración de la gravedad es

y = yIni + Viy*t + (1/ 2)*g*t*t; Vy - vel inicial a lo largo de y, g - constante gravitacional

Donde yIni es la coordenada inicial de la pelota, Viy es la velocidad inicial de la pelota a lo largo del eje y, t es el tiempo y g es la constante gravitacional.

Para velocidades finales

Vfx = Vix,
Vfy = g*t + Viy

Donde la velocidad final Vix es igual a la inicial por ser tiro parabólico y la velocidad final a lo largo del eje y esta dada por la velocidad inicial a lo largo de y mas el producto de la constante g por el tiempo t.

Este problema es bastante difícil. Primero se debe tener un alto grado de comprensión sobre como se utilizan las fórmulas de tiro parabólico así que si deseas entender bien lo siguiente te recomiendo que le des un repaso si es que ya lo olvidaste Claramente las unidades en las que se debe trabajar no serán metros por segundo sino pixeles por segundo, ya que Java trabaja con pixeles, un valor aceptable para g podría ser de 150, esta no es una conversión directa es solo un valor que da buenos resultados en el sentido grafico, puesto que son varias fórmulas muy relacionadas entre si pienso que es momento de aprovechar las capacidades de Java para trabajar con objetos entonces pongamos en una sola clase todas las funciones relacionadas con la "pelota", específicamente con el punto que se desplaza en el tiro parabólico:

```
import java.awt.*,

public class pelota {

  private int Vix, Viy,
  private int x, y;
  private double t,

  public pelota(int x, int y, int Vix, int Viy) {
    this.x = x;
    this.y = y;
    this Vix = Vix,
    this Viy = Viy,
  }

  public void tiempo(double t) {
    this t += t;
  }
```

```
  public int posX() {
    return (int)(x + Vix*t);
  }

  public int posY() {
    return (int)(y + Viy*t + (0.5)*(150)*t*t);
  }

  public int Vfx() {
    return (Vix);
  }

  public int Vfy() {
    return (int)(150*t + Viy);
  }

  public void dibujar(Graphics g) {
    g fillOval( posX(), posY(), 20, 20);
  }

}
```

Ahora la explicación de ley. El código inicial es:

```
  private int Vix, Viy;
  private int x, y;
  private double t;

  public pelota(int x, int y, int Vix, int Viy) {
    this.x = x;
    this.y = y;
    this.Vix = Vix;
    this.Viy = Viy;
  }
```

Aquí se declaran las variables

Vix - Velocidad Inicial en el eje x
Viy - Velocidad Inicial en el eje y
x - Posición actual en x
y - Posición actual en y
t - Variable de control para el tiempo

Después se declara un constructor que recibe como argumentos la posición x,y de la pelota y sus respectivas velocidades que se pasan a las variables globales con un this. La siguiente parte es

```
public void tiempo(double t) {
    this t += t;
  }

  public int posX() {
    return (int)(x + Vix*t);
  }

  public int posY() {
    return (int)(y + Viy*t + (0.5)*(150)*t*t);
  }
```

La función tiempo(double t) solo sirve para incrementar el tiempo a una tasa t, recuerda que.

this t += t;

es lo mismo que

```
this.t = this.t + t;
```

Lo cual es un simple incremento que suma t a la variable global más su valor propio inicial. La segunda función posX() sirve para obtener desde el exterior de la clase la posición actual de la pelota, observen se trunca el valor de las operaciones a un entero, esta es la fórmula de desplazamiento en x, como no hay ninguna aceleración la Vix = Vfx (Vfx = Vel. Final a lo largo del eje x).

La tercera función posY() se encarga de calcular la posición a lo largo del eje y, aquí se sustituyen los valores de g=150 y 1/2=0.5, la gravedad no es negativa porque recuerden que la coordenada "y" del plano cartesiano del applet se incrementa hacia abajo (esta al revés), por tanto una g positiva garantiza que la pelota bajará. Finalmente:

```
 public int Vfx() {
   return (Vix);
 }

 public int Vfy() {
   return (int)(150*t + Viy);
 }

 public void dibujar(Graphics g) {
   g.fillOval( posX(), posY(), 20, 20);
 }
```

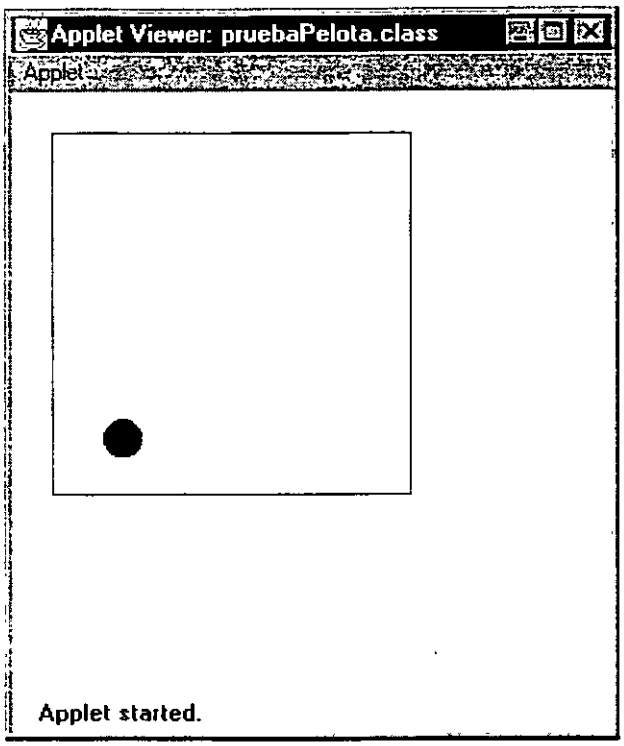
Las dos primeras funciones sirven para calcular las velocidades finales a lo largo de los respectivos ejes La última función dibujar(Graphics g) sirve para dibujar la pelota con un radio de 20 pixeles, esta función es pública para ser llamada desde el exterior de la clase.

Ahora utilicemos la clase anterior en un Applet·

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

public class pruebaPelota extends Applet implements Runnable {

  Thread hilo,
  pelota P;
  int x, y, Vx, Vy;

  public void init() {
    hilo = new Thread(this);
    P = new pelota(40, 40, 50, 10);
    hilo start();
  }

  public void run() {
    while (true) {
      P tiempo(0.04);
      repaint();
      try { hilo.sleep(40), }
      catch (InterruptedException e) { }
    }
  }

  public void paint(Graphics g) {

    setBackground(Color white);
    g.drawLine(20, 20, 20, 200);
    g.drawLine(20, 200, 200, 200);
    g drawLine(200, 200, 200, 20);
    g.drawLine(200, 20, 20, 20);
    P.dibujar(g),
```

```
    if( P.posX() > 180 && P.Vfx() > 0 ) {
      x = P.posX();
      y = P.posY();
      Vx = -P Vfx();
      Vy = P.Vfy(),
      P = null;
      P = new pelota(x, y, Vx, Vy);
    } else if( P.posX() < 20 && P.Vfx() < 0) {
      x = P.posX();
      y = P.posY();
      Vx = -P Vfx();
      Vy = P.Vfy(),
      P = null;
      P = new pelota(x, y, Vx, Vy);
    } else if( P.posY() > 180 && P.Vfy() > 0) {
      x = P.posX();
     y = P.posY(),
      Vx = P Vfx();
      Vy = -P.Vfy();
      P = null;
      P = new pelota(x, y, Vx, Vy);
    } else if( P.posY() < 20 && P.Vfy() < 0) {
      x = P.posX();
      y = P.posY(),
      Vx = P.Vfx();
      Vy = -P.Vfy();
      P = null;
      P = new pelota(x, y, Vx, Vy);
    }

  }

}
```

- Su salida es:



Nota: Recuerda que para poder correr esta Applet necesitas crear los dos archivos respectivos: "pelota.java" y "pruebaPelota.java", además deben encontrarse en la misma carpeta durante la compilación. El documento html debe cargar el applet, no pelota!.

El applet es tan divertida que también agrego el link para que veas la pelotita en acción.

Applet Pelota (Necesitas Internet Explorer 4 ó Netscape Navigator 4)

Esta applet tiene muchas cosas interesantes Comienza notando que se utiliza la interfaz Runnable para el control de la animación y el hilo Primero tenemos·

```
Thread hilo,
  pelota P,
  int x, y, Vx, Vy;

  public void init() {
    hilo = new Thread(this);
    P = new pelota(40, 40, 50, 10),
    hilo.start(),
  }
```

Aqui se declara el hilo y la pelota P, veremos la utilidad de las variables x,y,Vx,Vy más adelante con más calma. En el metodo init() se inicializan las variables como ya es costumbre y se echa a andar el hilo, observen que pasamos la coordenada inicial de la pelota a su constructor como x = 40 y = 40, y las velocidades iniciales Vix = 50, Viy = 10 (una vez que comprendas el problema experimenta cambiando estos valores, la coordenada inicial NO puede estar fuera del rectángulo a menos que cambies las condiciones de rebote como veremos mas adelante, las velocidades pueden ser negativas). Después viene el corazón del hilo, la función run():

```
  public void run() {
    while (true) {
      P.tiempo(0 04);
      repaint(),
      try { hilo.sleep(40), }
      catch (InterruptedException e) { }
```

```
    }
  }
```

Aquí tenemos el acostumbrado ciclo infinito y llamamos a la función tiempo de nuestra pelota P, el incremento es a 0.04 segundos, se llama a un repaint() para mostrar la posicion actualizada de la pelota y se checa cualquier error con el manejo de una excepción. Finalmente viene la parte más interesante de TODO el problema, en la función paint:

```
public void paint(Graphics g) {

   //dibujar cuadrado
    g.drawLine(20, 20, 20, 200);
    g.drawLine(20, 200, 200, 200);
    g.drawLine(200, 200, 200, 20);
    g drawLine(200, 20, 20, 20);
    //dibujar pelota
    P.dibujar(g),

    //condiciones de rebote
    if( P.posX() > 180 && P Vfx() > 0 ) {
      x = P.posX();
      y = P.posY();
      Vx = -P Vfx(),
      Vy = P.Vfy();
      P = null;
      P = new pelota(x, y, Vx, Vy);
    } else if( P.posX() < 20 && P.Vfx() < 0) {
      x = P.posX();
      y = P posY(),
      Vx = -P.Vfx(),
      Vy = P Vfy(),
      P = null;
      P = new pelota(x, y, Vx, Vy),
    } else if( P posY() > 180 && P.Vfy() > 0) {
      x = P.posX(),
      y = P posY(),
      Vx = P Vfx();
      Vy = -P Vfy(),
      P = null;
      P = new pelota(x, y, Vx, Vy),
    } else if( P posY() < 20 && P Vfy() < 0) {
      x = P posX();
      y = P posY(),
      Vx = P.Vfx(),
      Vy = -P Vfy(),
      P = null,
      P = new pelota(x, y, Vx, Vy),
    }

  }
```

Los primeros 4 drawLine son únicamente para dibujar los muros de nuestra cajita virtual El P dibujar(g), es para dibujar la pelota y pasa como argumento el objeto Graphics g para que la clase pelota se pueda dibujar a si misma Después vienen las 4 condiciones de rebote, solo nos centraremos en comprender una, las demas son similares

```
if( P posX() > 180 && P.Vfx() > 0 ) {
      x = P.posX(),
      y = P posY(),
      Vx = -P.Vfx(),
      Vy = P Vfy(),
      P = null,
```

```
        P = new pelota(x, y, Vx, Vy);
}
```

Nuevamente aparece un && (doble ampersand), esto significa que las dos condiciones

```
P.posX() > 180
P.Vfx() > 0
```

deben cumplirse para que el if se ejecute, la primera condición checa si se ha excedido 180 que es el límite derecho de nuestra caja virtual, ¿que no este valor debería ser 200? debería serlo si es que no contamos el radio de la pelota, este radio es 20 pixeles, por tanto se restan para mostrar claramente el efecto de rebote "al contacto con el muro" La segunda condición checa si la velocidad es positiva, esto se revisa nuevamente para evitar el problema de atascamiento, al checar esta velocidad nos aseguramos de que esta condición solo se ejecutará una sola vez. Cuando estas dos condiciones se cumplen se ejecuta el código dentro del if, que es

```
x = P posX(),
y = P posY(),
Vx = -P Vfx(),
Vy = P.Vfy();
P = null,
```

Las primeras 4 variables se toman de los datos existentes en la pelota mediante las funciones posX(), posY(), Vfx() y Vfy(), recuerda que el que tengan la P a la izquierda significa que se están llamando a las funciones de nuestra pelota, en las variables "x" e "y" se almacenan las coordenadas de la posición actual de la pelota que y en "Vx" y "Vy" se almacenan las velocidades actuales, sin embargo a Vx se le invierte el signo, esto es para crear el efecto de rebote Si observan bien todos los if-else contienen un cambio de signo dependiendo a lo largo del eje del cual se este rebotando (su condición), las condiciones que checan las posiciones en "y" cambian a la variable "Vy" de signo y las condiciones que checan las posiciones inferior y superior en "x" cambian a la variable "Vx" de signo. No se pueden poner a las dos condiciones de rebote que trabajan a lo largo del eje x en un solo if porque la dirección de las velocidades es distinta para cada caso (una positiva y otra negativa), lo mismo sucede para las condiciones de rebote a lo largo del eje y Otra forma de explicar lo que hace el fragmento de código anterior es que en pocas palabras toma el estado actual (justo antes del rebote) de la pelota para después destruirla y a partir de los datos tomados de su estado antes de ser destruida crea una nueva pelota solo que esta vez se invierte la dirección de una de sus velocidades para que se mueva en otra dirección, de ahí la presencia del P = null y después un nuevo llamado a su constructor

```
P = new pelota(x, y, Vx, Vy),
```

Este se encarga de crear la nueva pelota a partir de los datos anteriormente tomados, solo que esta vez una de las velocidades ya fue cambiada de signo y pues da la apariencia de que la pelota rebota ¿Ven que fácil es engañar a la vista? esta no es una pelota que rebota!.

Interfaz Gráfica y Eventos

- Etiquetas y Campos de Texto (Label y TextField)
- Botones (Button)
- Botones de Opción (Choice Button)
- Casillas de Verificación (Checkbox)
- Botones de Radio (CheckboxGroup)
- Listas (List)
- Paneles (Panel)
- Layouts
- Eventos
- Obteniendo Datos desde un TextField (Numéricos o de Texto)

Una interfaz gráfica nos permite utilizar un programa con más comodidad y además interactuar con él ya sea brindando datos o controlando eventos con el teclado, mouse y los botones en la pantalla Existen muchos tipos de elementos dentro de una interfaz gráfica por lo que considero muy importante introducir aquí los más básicos, una vez que comprendas la programación orientada a objetos será mucho más fácil para ti utilizarlos incluso sin conocerlos porque la Java API contiene todas las clases y muchos métodos relacionados con cada uno de ellos, así que solo será cuestión de tomarlos y utilizarlos. Este no es un tema complicado pero sí muy extenso, un buen libro de java contiene 2 o hasta 3 capítulos de Interfaces Gráficas pero su extensión se debe a que más bien son solo referencias y no tanto explicaciones concretas sobre algún tema complicado, te aconsejo que una vez que domines esta sección te acerques a algún libro para aprender más de las flexibles capacidades de Java, en este sentido es muchísimo más fácil programar una interfaz con Java que con C o C++

Todas las capacidades gráficas de Java están en el paquete java.awt (Abstract Window Toolkit) que hemos estado importando en varias ocasiones, esta vez estudiaremos algunas de las clases que contiene como Button, Label o TextField Casi todos estos elementos heredan directa o indirectamente de la clase Component así que te aconsejo revisar la Java API para conocer toda la serie de métodos existentes que puedes utilizar para manipularlos. Todos los ejemplos de esta sección son por razones prácticas en Applets pero en la sección de Aplicaciones veremos que es muy fácil utilizarlos en cualquier ventana (Frame)

Nota Importante: En estos ejemplos no doy las medidas que debe llevar cada applet en el documento html que la carga pero puedes dimensionarlos para que cada ejemplo se muestre con un tamaño razonablemente similar al mostrado aquí. En la mayoría de las ocasiones apareceran un poco diferentes.

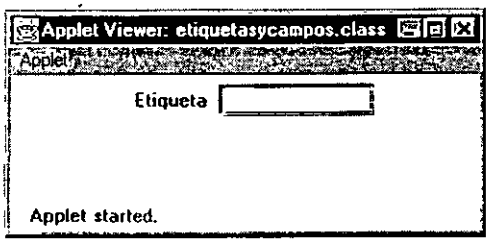
Etiquetas y Campos de Texto (Label y TextField)

Una etiqueta es una simple sección de texto que sirve para señalar algo, se acostumbra mucho colocar etiquetas junto con los campos de texto para señalar su función, un campo de texto es una sección que nos permite introducir texto o bien cadenas de caracteres. Pero es muy aburrido explicar sin imágenes y ejemplos que aclaren todo, así que aquí esta el primer ejemplo.

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

public class etiquetasycampos extends Applet {

  Label etiqueta;
  TextField campo;

  public void init() {
    etiqueta = new Label("Etiqueta");
    campo = new TextField( 10 );
    add(etiqueta);
    add(campo);
  }
```

```
  public void paint(Graphics g) {
  }

}
```

La salida del Applet anterior es:



Así de simple, la de la izquierda es la etiqueta y el campo es el de la derecha. El código comienza con.

```
Label etiqueta;
TextField campo;
```

Estos son dos objetos nuevos que no habíamos visto, el primero Label (etiqueta en Inglés) es para declarar una etiqueta y el segundo es para campos, en este caso los nombré como etiqueta y campo pero naturalmente pueden llevar otros nombres diferentes. Si buscas en la Java API se encuentran estas dos clases, "Label" y "TextField", ahí también se mencionan muchas funciones públicas para manipularlos con sus respectivos argumentos, incluidos sus constructores Recuerda que si quieres llamar a alguna de las funciones públicas de estos elementos debes utilizar el operador punto, por ejemplo

```
etiqueta.funcion();
```

Una vez declarados solo resta inicializarlos llamando a sus constructores:

```
 public void init() {
    etiqueta = new Label("Etiqueta");
    campo = new TextField( 10 ),
    add(etiqueta);
    add(campo);
}
```

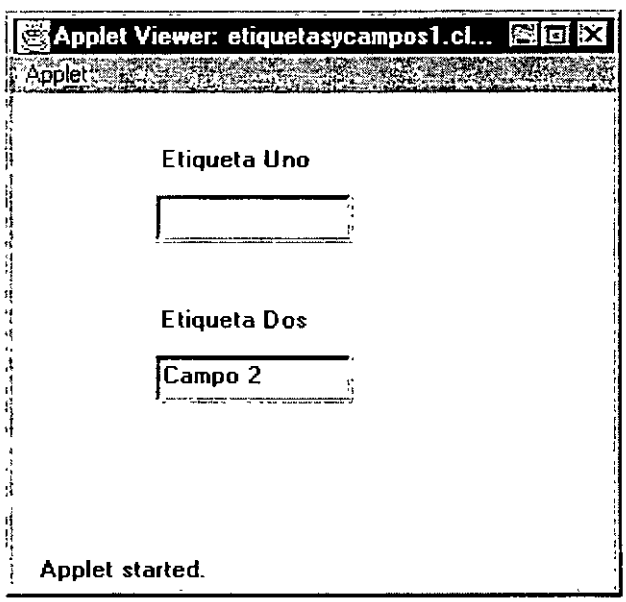
El constructor de Label recibe un argumento de tipo String que señala el texto que se exhibirá en la etiqueta, recuerda que las cadenas van entre comillas, el constructor de campo recibe como argumento el número de dígitos o caracteres que es capaz de recibir el campo o lo que es lo mismo, su longitud. Finalmente viene el llamado a dos funciones add(), estas funciones se encargan de mostrar en la pantalla nuestros elementos porque una cosa es inicializarlos y otra es exhibirlos, recuerden que la computadora nos pide que le demos la mayor cantidad de información posible para realizar bien su trabajo, eso incluye los detalles de cuando deben exhibirse estos elementos. Cada add recibe como argumento el objeto que va a exhibirse, si no los colocas ningún objeto aparecera en la pantalla El paint del final esta vacio porque en este caso no deseamos pintar nada en la pantalla excepto los elementos

El manejo anterior de un TextField y una Label en ningún momento nos pidió las coordenadas en las cuales se exhibirían cada uno de los elementos, de forma predeterminada Java centra los elementos y los coloca de acuerdo al tamaño de la ventana pero obviamente existe una forma de especificar claramente la posición exacta en la cual deseamos colocar los objetos, ahora mostrare un ejemplo de esto y además de la posibilidad de incluir más de un TextField y un Label a la vez:

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

public class etiquetasycampos1 extends Applet {

  Label etiqueta1, etiqueta2;
  TextField campo1, campo2;
```

```
  public void init() {

    etiqueta1 = new Label("Etiqueta Uno");
    campo1 = new TextField( 10 );

    etiqueta2 = new Label("Etiqueta Dos");
    campo2 = new TextField( 10 );
    campo2.setText("Campo 2");

    setLayout(null);

    etiqueta1.setBounds(60,20,100,25);
    campo1.setBounds(70,50,100,25);

    etiqueta2.setBounds(60,100,100,25);
    campo2.setBounds(70,130,100,25);

    add(etiqueta1);
    add(campo1);
    add(etiqueta2);
    add(campo2);
  }

  public void paint(Graphics g) {
  }

}
```

Su salida es:



Primero declaramos los elementos con sus respectivos nombres, esta vez vamos a crear más de uno de ahí la presencia de los números.

```
Label etiqueta1, etiqueta2;
TextField campo1, campo2;
```

Ahora, dentro de la función init llamamos a sus respectivos constructores:

```
    etiqueta1 = new Label("Etiqueta Uno");
    campo1 = new TextField( 10 );
```

```
    etiqueta2 = new Label("Etiqueta Dos");
    campo2 = new TextField( 10 );
    campo2.setText("Campo 2"),
```

Lo único distinto con respecto al ejemplo anterior es el llamado a la función setText("Campo 2") desde el elemento campo2, esta función public de TextField nos permite agregar texto al campo en este caso justo después de haberlo creado, esto es útil si se desea mostrar información cuando recién se ha ejecutado el programa o incluso podríamos tener campos cuyo fin exclusivamente sería el de mostrar información según se ejecuta un programa La siguiente sección de código en el init es:

```
    setLayout(null);

    etiqueta1.setBounds(60,20,100,25);
    campo1.setBounds(70,50,100,25),

    etiqueta2.setBounds(60,100,100,25),
    campo2.setBounds(70,130,100,25);
```

Estudiaremos más a fondo un poco más adelante la utilidad de la función setLayout() (que en inglés significa: fijar colocación) pero les adelanto que en este ejemplo nos permite eliminar (por eso es el **null**, recuerden que destruye objetos sin importar su tipo) el centrado automático que realiza Java sobre los elementos, pero naturalmente ahora nos exigirá las coordenadas específicas para colocarlos, estas coordenadas se pasan mediante las función·

```
setBounds(int x, int y, int ancho, int alto)
```

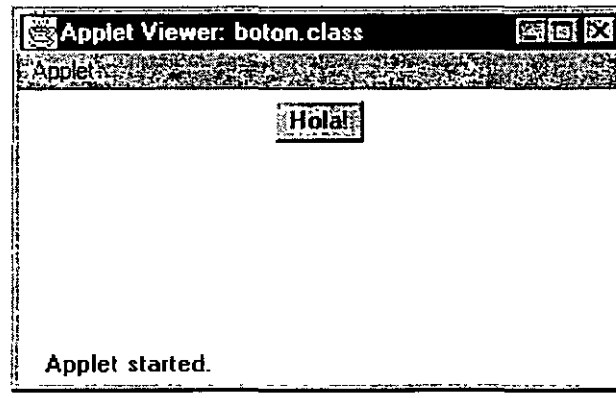
La traducción de esta función sería algo así como "fijar posición", "ceñir posición" o "restringir posición". Los argumentos x,y indican la posición del elemento a partir de su esquina superior izquierda, los otros dos especifican su ancho y alto, estos datos nos permiten colocar a nuestro gusto todos los elementos. En la parte final del init se tienen nuevamente las funciones add() que agrega los elementos a la pantalla.

Botones (Button)

Los botones se han convertido en un elemento de ley en todo el software moderno, de hecho un boton es un sinónimo de "ventas" para los desarrolladores de software ya que facilita muchisimo el uso de las aplicaciones, el único elemento que le hace la batalla de frente como elemento fundamental de una interfaz gráfica es el menú lo cual me recuerda que pronto tendré que poner algo al respecto, aunque a veces los menús me han vuelto loca, en las aplicaciones industriales abusan mucho de ellos (como en Pro-Engineer o en las estaciones de trabajo de SG). Como sea los botones siempre son muy divertidos, primero veamos como se declara un boton sencillo dentro de un applet

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

public class boton extends Applet {

  Button boton,

  public void init() {
    boton = new Button("Hola!"),
    add(boton),
  }

  public void paint(Graphics g) {
  }

}
```

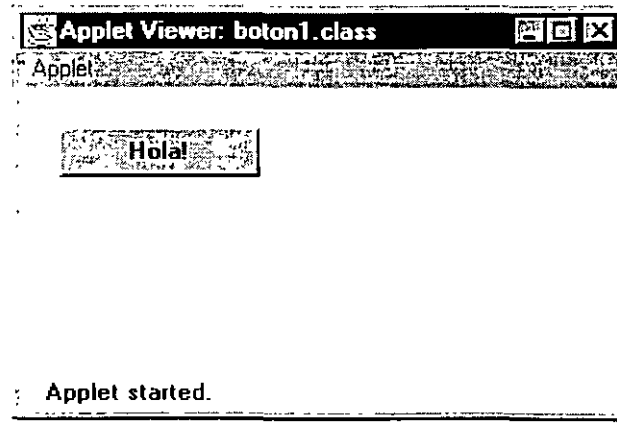
La salida es·



Como ven es facilisimo, no hay mayor ciencia. Se nombra el botón, se inicializa con su constructor pasando como argumento el mensaje que mostrará en él y por último se muestra con un add(), asi de simple. Este programa nuevamente centra los elementos de forma predeterminada, ahora usemos otra vez setLayout(null), (quitar la colocación predeterminada) para especificar la posición y el tamaño de nuestro botón:

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

public class boton1 extends Applet {

    Button boton;

  public void init() {
    boton = new Button("Hola!");
    setLayout(null);
    boton.setBounds(20,20,100,25);
    add(boton);
  }


  public void paint(Graphics g) {
  }

}
```

Salida del applet:



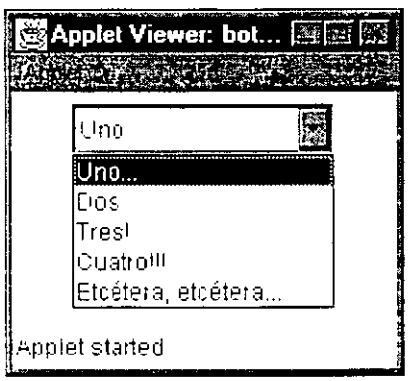
Sucede lo mismo que en el ejemplo de etiquetas y campos de texto. Si explico lo anterior estaria evidentemente atentando contra su inteligencia

Botones de Opción (Choice Button)

Los botones de opción son componentes que poseen una lista de opciones, realmente no tienen la forma de un botón convencional pero se supone que realizan una acción al elegir uno de los "items" o elementos que contiene, de ahí que también se les llame botones. Al igual que con los otros componentes una vez consultada la Java API podemos observar sus constructores y sus métodos (tanto propios como heredados). La siguiente applet crea un componente de tipo Choice y le agrega 5 ítems o elementos:

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

public class botonOpcion extends Applet {

Choice bopcion;

public void init() {

   bopcion = new Choice();
   bopcion.add("Uno...");
   bopcion.add("Dos  .");
   bopcion.add("Tres!");
   bopcion add("Cuatro!!!");
   bopcion add("Etcétera, etcétera.  ");
   add(bopcion);

}
}
```

Salida del applet:



Como pueden ver es bastante simple, primero se nombra como: bopcion, después lo inicializamos con su constructor que en este caso no recibe ningún argumento:

```
bopcion = new Choice();
Ahora agregamos algunos items:
bopcion.add("Uno.  ");
bopcion.add("Dos ..");
bopcion.add("Tres!");
bopcion.add("Cuatro!!!");
bopcion.add("Etcétera, etcétera...");
```

Es muy importante observar que aquí ya no aplicamos el add() directamente, es decir que ahora lo aplicamos sobre bopcion ¿qué significa esto realmente?. Anteriormente llamábamos a la función add() del applet que nos permitía colocar componentes precisamente en dicha applet, ésta vez llamamos a la función add() de bopcion que es en realidad un componente de tipo Choice, esto significa que la clase Choice tiene su propia versión de una

función add() (que recibe un String como argumento) que sirve para agregar elementos elegibles al botón de opción, recuerden que todos estos componentes poseen muchos más funciones que reciben y devuelven como argumentos muchos tipos (incluidos los estándar), todo esto deberá consultarse en la Java API.
Y finalmente agregamos el "bopcion" al applet para que se exhiba en él:
add(bopcion);

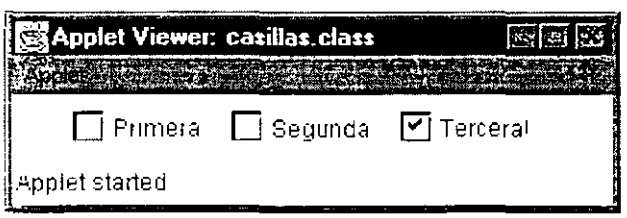
Como lo dije al principio un Choice cumple la misma función de un botón pero éste tiene la capacidad de alojar múltiples opciones, más adelante en nuestro estudio de Eventos veremos como manejar las acciones de este componente.

Casillas de Verificación (Checkbox)

Este componente también es sumamente fácil de usar, disculpen que continúe insistiendo en que revisen la Java API para mayor información pero es que definitivamente es la **referencia por excelencia** para conocer tanto los constructores como los métodos (recuerden que es la clase Checkbox), además en algunas ocasiones estos componentes heredan tantos métodos y variables que son demasiados para mí. He aprendido mucho a interpretar la Java API probando y experimentando con toda esta gama de componentes gráficos, esta práctica ayuda para interpretar otras clases que son aún más complejas (sobre todo por su gran extensión) así que experimenten!. Las casillas de verificación sirven para definir opciones de estado o si desean verlo de otra forma son casillas que se activan (con una palomita) o no se activan, solo tienen dos estados permisibles. En un programa generalmente estas casillas cambiarían variables globales de tipo bolean que controlarán otras variables o bien algunas propiedades (como tipo de letra). Veamos un ejemplo:

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

public class casillas extends Applet {

Checkbox primera, segunda, tercera;

public void init() {

primera = new Checkbox("Primera");
segunda = new Checkbox("Segunda");
tercera = new Checkbox();
tercera.setLabel("Tercera!");
tercera.setState(true);
add(primera);
add(segunda);
add(tercera);

}
}
```

La salida de esta Applet es:



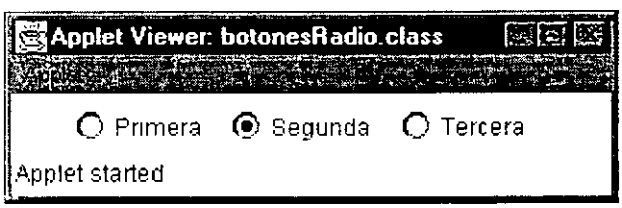
Como ven tambien son simples, primero las nombramos y después llamamos a sus constructores que en este caso reciben como argumento la cadena que exhibirá cada una (etiqueta), la tercera utiliza el constructor que no recibe argumentos, sin embargo después agregamos una etiqueta para que la muestre mediante su funcion setLabel(), casi todos los componentes tiene dicha funcion, también en la tercera fijamos el estado inicial como activado (true) por eso se exhibe palomeada de forma inicial Finalmente las agregamos a la ventana con add().

Botones de Radio (CheckboxGroup)

Los Botones de Radio son en cierta forma casillas de verificación solo que esta vez están agrupadas. La agrupación obedece a que en ciertas ocasiones podríamos desear que solo una casilla de un grupo de varias sea activable, esto sería crear una relación entre un grupo de casillas de modo que si activas una todas las demás se desactivarán, así de fácil. Para crear un grupo de casillas también llamado Botones de Radio (porque son redonditos) debes crear la cantidad de casillas que quieras agrupar y después debes construir un CheckboxGroup que las agrupará, veamos el ejemplo:

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

public class botonesRadio extends Applet {

CheckboxGroup grupoCasillas,
Checkbox primera, segunda, tercera;

public void init() {

grupoCasillas = new CheckboxGroup(),
primera = new Checkbox("Primera", grupoCasillas, false);
segunda = new Checkbox("Segunda", grupoCasillas, true);
tercera = new Checkbox("Tercera", grupoCasillas, false),
add(primera);
add(segunda);
add(tercera);

}
}
```

La salida del Applet anterior es:



Bueno, primero nombramos nuestras casillas y nuestro grupo

```
CheckboxGroup grupoCasillas;
Checkbox primera, segunda, tercera,
```

Ahora inicializamos todo con sus respectivos constructores dentro de la funcion init():

```
grupoCasillas = new CheckboxGroup();
primera = new Checkbox("Primera", grupoCasillas, false),
segunda = new Checkbox("Segunda", grupoCasillas, true);
tercera = new Checkbox("Tercera", grupoCasillas, false);
```

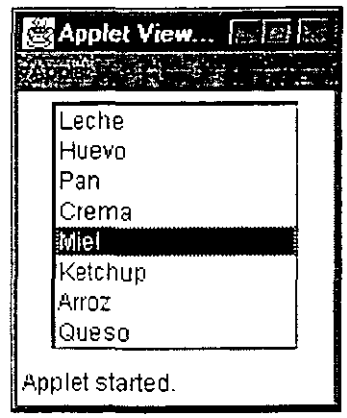
Como ya lo platiqué un CheckboxGroup únicamente sirve para agrupar casillas (Checkbox) por lo tanto es evidente que un CheckboxGroup no es un elemento visible sino más bien es solo una herramienta de agrupación así que no es necesario que su constructor reciba argumentos tal como se ve en el ejemplo. Para las casillas ahora recurrimos al constructor que recibe tres argumentos, el primero es el mensaje que exhibe cada una de las opciones como un String, el segundo es el grupo al que pertenece (en este caso todas pertenecen al mismo grupo) y el tercer y último constructor (que es un bolean) indica si la casilla estará activada de manera inicial o no, solo una de las casillas del grupo puede tener inicialmente este valor como true, si intentas poner alguna otra también como true no podras compilar este código. Finalmente y también dentro de la función init() tenemos los acostumbrados add() para agregar cada uno de nuestros componentes, te recuerdo una vez más que un CheckboxGroup no es un componente visible por esa razón no es necesario agregarlo.

Listas (List)

Las listas son componentes relativamente desconocidos, sirven para seleccionar una serie de opciones visibles y quizás en base a esa selección realizar una acción o bien copiar los elementos elegidos (o a algún elemento asociado con él) hacia otra parte. Les confieso que jamás he utilizado una lista dentro de un programa Java que haya creado, yo solo las he visto en software bastante especializado. Aún asi son fácil de configurar y de exhibir tal como lo muestra el ejemplo.

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

public class listas extends Applet {

List superMercado,

public void init() {

superMercado = new List(8, false);
superMercado.addItem("Leche");
superMercado.addItem("Huevo");
superMercado.addItem("Pan");
superMercado.addItem("Crema");
superMercado.addItem("Miel");
superMercado.addItem("Ketchup"),
superMercado.addItem("Arroz");
superMercado.addItem("Queso");
add(superMercado),

}


}
```

- La salida del Applet anterior es:



Como ven las listas son muy similares a los botones de opción (Choice Button). Primero nombramos nuestra lista:

List superMercado,

Ahora llamamos a su constructor

superMercado = new List(8, false);

El primer argumento es un número entero que represente al número de items que tenemos pensado agregar y el segundo argumento es un bolean que dice si será posible o no seleccionar varios items al mismo tiempo, en este caso es false Ahora agregamos algunos elementos a esta lista (son Strings) mediante la función addItem() (que recibe argumentos String) de la clase List:

```
superMercado.addItem("Leche");
superMercado.addItem("Huevo");
superMercado addItem("Pan"),
superMercado.addItem("Crema");
superMercado addItem("Miel");
superMercado addItem("Ketchup"),
superMercado addItem("Arroz");
superMercado addItem("Queso");
```

Y como de costumbre agregamos la Lista al applet con un add():

```
add(superMercado);
```

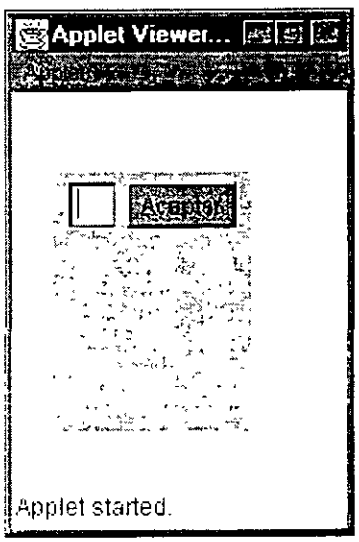
Paneles (Panel)

Los paneles son componentes realmente útiles para configurar y definir la forma en que podrán exhibirse nuestros elementos en la pantalla Un panel es un area aislada del applet en la que podemos agregar cualquier otro componente tal como Button, Label, TextField, Choice, Checkbox, etc. e incluso podemos agregar a un panel otros panel Supongan que es un área de dibujo solo que esta vez es para componentes exclusivamente. En el siguiente ejemplo creamos un Panel y le agregamos diversos componentes, incluso establecemos su color de fondo y finalmente lo colocamos en el lugar deseado (mediante coordenadas)

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

public class pruebaPanel extends Applet {

Button aceptar;
TextField entrada,
Panel mostrar;

public void init() {
```

```
mostrar = new Panel();
setLayout(null);
aceptar = new Button("Aceptar");
entrada = new TextField();

mostrar.add(entrada);
mostrar.add(aceptar);

add(mostrar),

mostrar.setBounds( 20, 40, 100, 130 );
mostrar setBackground(Color.pink);

}
}
```

La salida del Applet anterior es:



En esta applet primero declaramos tres componentes: un botón, un TextField y un Panel·

```
Button aceptar;
TextField entrada,
Panel mostrar,
```

Ahora en el método init llamamos a sus respectivos constructores:

```
mostrar = new Panel(),
setLayout(null),
aceptar = new Button("Aceptar"),
entrada = new TextField();
```

El constructor de un panel no recibe ningún argumento, sin embargo muchas de sus características son modificables Los constructores de Button y TextField ya los conocemos actualmente Después llamamos a la función add de la clase panel (en este caso desde nuestro panel llamado "mostrar") para poder agregar componentes, esta función add() recibe como argumentos datos de tipo Component, todos nuestros componentes graficos heredan de Component por lo tanto también son Component, recuerden que las funciones pueden recibir como argumentos tipos variados desde int, char, String hasta incluso tipos Component o Graphics, de hecho la función add() que tanto hemos usado que se aplica directamente sobre un applet recibe Components como argumento

```
mostrar add(entrada);
mostrar.add(aceptar),
add(mostrar);
```

Observen que al final de la sección anterior de código agregamos ahora si nuestro panel llamado "mostrar" al applet. Si observaron bien la salida del applet verán que los componentes que le agregamos al panel aparecen centrados, esto tiene que ver con el Layout (ordenamiento u acomodamiento) que posee de manera predeterminada todo panel, las applets también poseen un Layout predefinido para acomodar componentes, en este ejemplo lo eliminamos mediante. setLayout(null), sin embargo el panel continúa teniendo su Layout predefinido que centra los componentes que recibe. Recapitulando: Las applets poseen un Layout predefinido y los Panels también poseen el suyo, en este ejemplo eliminamos el Layout predefinido del applet por lo que debemos acomodar manualmente los componentes agregados:

```
mostrar.setBounds( 20, 40, 100, 130 );
```

El único componente agregado en este ejemplo al applet es un panel llamado "mostrar", si el panel tiene o no componentes en su interior eso ya es arroz de otro costal. Recuerden que el primer argumento de un setBounds() es la coordenada "x" de la esquina superior izquierda del componente, el segundo argumento seria su coordenada "y", el tercer argumento es el ancho y por último el cuarto argumento es el alto Es posible que también eliminemos el Layout (ordenamiento) de componentes que posee todo panel pero en tal caso estariamos obligados a definir manualmente cada posición Por ejemplo eliminariamos el Layout del panel "mostrar" con:

```
mostrar.setLayout( null )
```

Y ahora deberiamos especificar la posición de cada componente dentro del panel con un: mostrar.**componente**.setBounds(x, y, ancho, alto) donde **componente** seria el nombre del componente a colocar dentro del panel. Resumiendo, si deseamos colocar componentes dentro de un Panel al que se le haya eliminado su Layout predeterminado, 1) agregamos los componentes con un add() sobre el panel y 2) especificamos su posición con un setBounds() Ahora veremos más sobre Layouts.

Layouts (Acomodamientos)

Un Layout es una forma de acomodar componentes dentro de un applet, un panel o un Frame. Ya hemos visto cómo quitarlo para un applet y un panel, se quita de la misma manera de un Frame, veremos más sobre Frames en la sección de Aplicaciones sin embargo se aplica exactamente de la misma manera en un Frame que en un Applet

Existen muchas maneras de acomodar componentes (muchos Layouts), la siguiente es una tabla que los resume:

Layouts

Layout	Propiedades
BorderLayout (Acomodamiento como Borde)	Acomoda los componentes de modo que llenen una de cinco zonas: Norte, Sur, Este, Oeste y Centro
GridLayout (Acomodamiento como Reticula)	Acomoda los componentes de en una serie de lineas y columnas, el orden es importante
CardLayout (Acomodamiento como Barajas)	Trata cada componente en el contenedor como una baraja, solo una baraja es visible a la vez y el contenedor actúa como "una mano" de barajas
GridBagLayout (Acomodamiento por Bolsas de Reticulas)	Alinea los componentes en una serie de lineas y columnas dejando al programador la tarea de especificar cuantas lineas y columnas ocupará cada componente al igual que su ubicación exacta

Nota:Reticula es lo mismo que cuadricula o que cuadriculado (igual al de los cuadernos)

Cada Layout es en realidad una clase con sus variables y métodos propios que nos permiten crearlo (llamar a su constructor) y modificarlo (utilizar sus métodos) para finalmente fijarlo con setLayout() a un applet, un panel o un Frame, existen otros componentes a los cuales también es posible agregarles un Layout sin embargo eso lo verás una vez que conozcas más sobre Java en algún libro porque hay libros (bastante gruesos!) que tratan única y exclusivamente el AWT (Abstract Window Toolkit). De los anteriores digamos que el BorderLayout y el GridLayout son facilisimos, el CardLayout requiere un poquito más de atención para comprenderlo y el GridBagLayout puede ser bastante extenso.

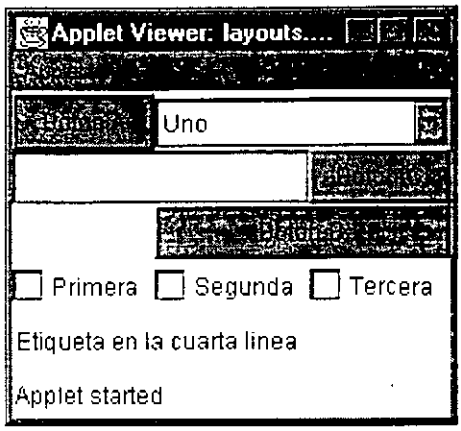
En programas serios prácticamente no se utiliza el BorderLayout. Un CardLayout puede ser interesante pero existe otro modelo similar y aún más práctico en la Java Foundation Clases (JFC cuyos elementos gráficos se conocen mejor por el nombre de su paquete: Swing) cosa que se trata en la sección de **Swing**, en fin, probablemente en el futuro ponga más información, como el GridBagLayout es el mas interesante (y el mas útil) es el único que veremos aquí por ahora.

Bueno, es muy limitante que los componentes tales como: Buttons, Labels, TextFields etc. aparezcan centrados en la pantalla, ahora, si quitamos el Layout con un: setLayout(null) es bastante tedioso tener que posicionar los elementos (casi por prueba y error) compilando y exhibiendo una y otra vez tratando que los componentes queden del tamaño y con la posición apropiada. Una vez que dejes de ser principiante entonces podrás experimentar con editores más avanzados de código que te permitan modificar gráficamente todos estos componentes sin la necesidad de escribir ni una sola linea de código pero mientras tanto **debes** aprender lo básico a mano. El siguiente ejemplo aplica un GridBagLayout en un applet, la explicación se encuentra después de la imagen de su salida:

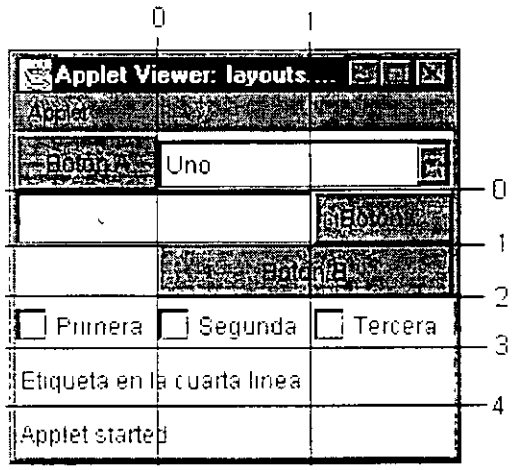
```
import java.awt.*;
import java.applet.Applet;

public class layouts extends Applet {

Button a, b, c;
Choice d;
Checkbox e, f, g;
TextField h;
Label i;

GridBagLayout reticula;
GridBagConstraints restricciones;

void addComponent( Component comp, GridBagLayout bag, GridBagConstraints bagcons,
                   int linea, int columna, int ancho, int alto ) {

bagcons.gridx = columna;
bagcons.gridy = linea;
bagcons.gridwidth = ancho;
bagcons.gridheight = alto;

bag.setConstraints( comp, bagcons );
add( comp );

}

public void init() {

reticula = new GridBagLayout();
setLayout( reticula );
restricciones = new GridBagConstraints();

a = new Button("Botón A");
b = new Button("Boton B");
c = new Button("Botón C");
d = new Choice();

d.add("Uno");
d.add("Dos");
d.add("Tres");

e = new Checkbox("Primera");
f = new Checkbox("Segunda");
g = new Checkbox("Tercera");
h = new TextField();
i = new Label("Etiqueta en la cuarta linea");
```

```
restricciones.weightx = 1;
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH;
addComponent(a, reticula, restricciones, 0, 0, 1, 1 ),

restricciones.weightx = 1;
restricciones weighty = 1;
restricciones fill = GridBagConstraints.BOTH;
addComponent(b, reticula, restricciones, 2, 1, 2, 1 );

restricciones.weightx = 1,
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH;
addComponent(c, reticula, restricciones, 1, 2, 1, 1 ),

restricciones weightx = 1;
restricciones weighty = 1;
restricciones fill = GridBagConstraints BOTH,
addComponent(d, reticula, restricciones, 0, 1, 2, 1 ),

restricciones weightx = 1,
restricciones weighty = 1,
restricciones fill = GridBagConstraints.BOTH,
addComponent(e, reticula, restricciones, 3, 0, 1, 1 );

restricciones.weightx = 1,
restricciones weighty = 1,
restricciones.fill = GridBagConstraints BOTH,
addComponent(f, reticula, restricciones, 3, 1, 1, 1 ),

restricciones weightx = 1;
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints BOTH;
addComponent(g, reticula, restricciones, 3, 2, 1, 1 ),

restricciones weightx = 1;
restricciones weighty = 1;
restricciones fill = GridBagConstraints BOTH,
addComponent(h, reticula, restricciones, 1, 0, 2, 1 ),

restricciones weightx = 10,
restricciones weighty = 1;
restricciones fill = GridBagConstraints.BOTH,
addComponent(i, reticula, restricciones, 4, 0, 3, 1 ),

}


}
```

La salida del Applet anterior es



Primero una descripción general de lo que hicimos: La idea es que la ventana de nuestra Applet se divida en 3 columnas y 6 lineas. Una vez dividida ahora queremos agregar componentes, algunos de ellos ocuparan más de una linea o más de una columna y otros podrian estar en una celda (solitos) El GridBagLayout se encarga de lo anterior, este Layout comienza con el trazado que deseamos, se idea y se dibuja sin tener que escribir codigo El siguiente screenshot muestra la división que hicimos, observen que tanto el conteo de lineas como de columnas comienza en cero, por ejemplo la primera columna es cero, la segunda es uno y la tercera es dos, lo mismo con las lineas



En el codigo primero declaramos los componentes que vamos a utilizar.

```
Button a, b, c;
Choice d,
Checkbox e, f, g,
TextField h,
Label i;

GridBagLayout reticula
GridBagConstraints restricciones,
```

Como ven al final del código anterior se encuentran los dos elementos nuevos un GridBagLayout llamado "reticula" y un GridBagConstraints llamado "restricciones", el GridBagConstraints nos servira para restringir algunas propiedades para cada componente por agregar Al principio de la funcion init() tenemos los constructores para GridBagLayout y para GridBagConstraints, ninguno de los dos recibe argumentos, también fijamos de una vez el nuevo Layout que deseamos para nuestra applet como reticula (o sea el GridBagLayout):

```
reticula = new GridBagLayout(),
setLayout( reticula ),
restricciones = new GridBagConstraints(),
```

Pareciera por lo que dijimos que el constructor del GridBagLayout deberia recibir algún argumento como por ejemplo: el número de lineas y columnas sin embargo este Layout esta diseñado para que cada componente se posicione uno a uno manualmente, así viene de manera predeterminada en el JDK. Si lo que deseamos es una retícula (con cierto número de lineas y cierto número de columnas) podemos imaginarnos de entrada el tipo de **valores necesarios** por configurar para cada componente: la línea en la que será colocado, la columna en la que será colocado, el número de líneas que ocupará (alto), el número de columnas que ocupará (ancho), entre otros, pues bien son aún más propiedades que **tenemos** que definir por cada componente que se piensa agregar. Lo anterior es la razón por la cual existe la función addComponent (creada por nosotros), que sirve para ahorrarnos tener que asignar tantos valores, pasaremos estos valores a la función y ella se encargará de asignarlos por nosotros, así por lo menos nos ahorrará un poco de código, esta fue una descripción general del applet ahora vamos a detallar aún más las cosas, primero vamos a entender bien para qué sirve exactamente un GridBagConstraints

Variables de GridBagConstraints

Variable	Define
gridx (columna)	La columna inicial donde se coloca el componente
gridy (linea)	La línea inicial donde se coloca el componente
gridwidth (ancho)	El ancho (en columnas) del componente
gridheight (alto)	El alto (en líneas) del componente
weightx (peso x)	Nivel de importancia para que el componente se extienda a lo largo de las columnas
weighty (peso y)	Nivel de importancia para que el componente se extienda a lo largo de las líneas

Nota:Constraints significa restricciones en inglés

La tabla anterior muestra las variables globales de la clase GridBagConstraints, esta clase nos permite definir ciertas propiedades para cada uno de los componentes que vamos a añadir al GridBagLayout, esto significa que por cada componente en teoria deberiamos crear un objeto GridBagConstraints (llamando a su constructor) lo cual seria muy laborioso por eso es mucho mas facil crear un solo GridBagConstraints (llamado "restricciones") y modificarlo para cada componente, para eso es como ya lo habia dicho la función addComponent() siguiente:

```
void addComponent( Component comp, GridBagLayout bag, GridBagConstraints bagcons,
                   int linea, int columna, int ancho, int alto ) {

  bagcons gridx = columna,
  bagcons gridy = linea,
  bagcons gridwidth = ancho,
  bagcons.gridheight = alto,

  bag setConstraints( comp  bagcons ),
  add( comp ),

  }
```

De la tabla de variables ahora sabemos que gridx es la columna en la que pensamos colocar nuestro componente, lo mismo con gridy solo que en lineas Las variables gridwidth y gridheight son el ancho y alto respectivamente Por último fijamos las restricciones recien definidas al GridBagLayout mediante la función setConstraints() que recibe como argumentos el GridBagLayout al cual se piensa restringir y el objeto GridBagConstraints que contiene tales restricciones La función init() se encarga de dos cosas fundamentales llamar a los constructores y modificar otras variables del GirdBagConstraints que estamos usando:

```
a = new Button("Botón A");
b = new Button("Boton B"),
```

```
c = new Button("Botón C");
d = new Choice(),

d.add("Uno"),
d.add("Dos");
d.add("Tres");

e = new Checkbox("Primera"),
f = new Checkbox("Segunda"),
g = new Checkbox("Tercera"),
h = new TextField(),
i = new Label("Etiqueta en la cuarta linea");
```

Todo lo anterior ya lo vimos al principio de esta sección, lo nuevo es lo siguiente·

```
restricciones.weightx = 1;
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH,
addComponent(a, reticula, restricciones, 0, 0, 1, 1 );
```

Esto se hace para cada uno de los componentes (desde el "a" hasta el "i"), primero accedemos a las variables public globales weightx y weighty del GridBagConstraints para modificarlos, el weightx es en realidad el nivel de importancia que se le debe dar al reacomodo del componente (a lo largo del eje x, es decir, en las columnas) cuando el usuario modifica el tamaño del applet, la mayoría de las veces las applets se cargan desde un navegador el cual rara vez permite modificar el tamaño interno del applet por lo cual los Layouts son más útiles en las aplicaciones aunque es mas simple explicarlos con un Applet. Un mayor valor para weighty y weightx hará que los componentes acaparen más del espacio disponible cuando se modifica el tamaño de la ventana. La variable fill define hacia cuál de las cuatro direcciones (arriba, abajo, izquierda, derecha) se extenderá mas el componente, esta variable es en realidad un valor entero (int) pero se acostumbra mucho colocar valores constantes con etiquetas muy descriptivas para almacenar valores constantes, no sé cuanto vale BOTH pero sí se lo que representa: que el componente se extendera con la misma importancia en dirección vertical (arriba, abajo) como en dirección horizontal (izquierda, derecha), observen muy bien que estos valores descriptivos para fill son variables public globales de la clase GridBagConstraints, si revisan la Java API verán que existen otros posibles valores que podemos utilizar en vez de BOTH como son. CENTER, EAST, HORIZONTAL, NONE, NORTH, SOUTH, entre otros, si este valor no se modifica entonces el valor predeterminado será NONE. Te aconsejo que experimentes con algunos para observan como el componente se reacomoda en las direcciones asignadas una vez que se modifica el tamaño de la ventana Por último llamamos a la funcion addComponent() que creamos

```
addComponent(a, reticula, restricciones, 0, 0, 1, 1 );
```

Recordando los argumentos, el primero es el componente por agregar en cuestion, el segundo es el GridBagLayout que estamos usando, el tercero es.el GridBagConstraints, el cuarto y el quinto indican la linea y la columna respectivamente (iniciales) para colocar el componente y los últimos dos son el ancho y el alto, bueno, eso es todo por ahora sobre Layouts

Eventos

Los eventos son las acciones que ocurren al exterior del programa, quizás yo las hubiera nombrado algo así como acciones externas al programa, aunque es mas extenso eso si Bueno, estos eventos son, clicks del mouse, teclas presionadas, teclas soltadas, tecla enter, arrastrar con el mouse, etc. entonces, todo programa que pretenda tener algo de interactividad debe recurrir a los eventos

Los pasos generales para agregar soporte para los eventos son:

1 Elegir el tipo de evento que queremos "escuchar" (del teclado o del mouse)
2. Elegir el objeto que deseamos que le ponga atención a ese evento (elegir al receptor que debe reaccionar)

Elección del tipo de Evento

Los eventos como casi todo en Java también son objetos (archivos de clase), cada uno tiene funciones que devuelven por ejemplo sobre quién se hizo el evento o la información que esta intentando pasar. La siguiente tabla resume los **tipos de eventos** disponibles en Java:

Tipos de Eventos

Tipo de Evento	Propósito
ActionEvent	Utilizado por los componentes para notificar al programador sobre un evento como un click del mouse en un botón o la acción de presionar la tecla enter en un campo de texto
ChangeEvent	Atrapa el evento generado de un componente que ha cambiado su estado tal como una Checkbox
FocusEvent	Atrapa el evento que se genera cuando un componente adquiere o pierde atención del teclado (cursor)
InputEvent	Superclase abstracta para eventos del mouse y del teclado. Posee métodos para obtener el estado actual de los modificadores del teclado (teclas TAB, ALT, SHIFT, BloqMayus, etc.)
ItemEvent	Se utiliza por los componentes que poseen múltiples selecciones. Los eventos de item son para indicar cuando una selección ha sido elegida
KeyEvent	Atrapa los eventos del teclado, ya sea al oprimir una tecla o al soltarla
ListDataEvent	Atrapa el evento generado cuando el contenido de una lista cambia
ListSelectionEvent	Atrapa el evento que se genera cuando cambia la selección de una lista
MenuEvent	Atrapa el evento que se genera cuando un menu es seleccionado o cancelado
MouseEvent	Atrapa el evento que se genera cuando el usuario manipula el mouse de diversas maneras
WindowEvent	Atrapa los eventos generados por el ciclo de vida de la ventana, como cuando se cierra o cuando se iconifica (minimiza)

Una vez elegido el evento ahora debe ser **atrapado, atendido o capturado** por alguien, por otro objeto (o archivo de clase) que puede ser desde un Applet, un botón o un TextField hasta una clase cualquiera definida y creada por nosotros. Esta **comunicación** entre un evento y el objeto que lo atrapará se realiza mediante interfaces, si no has consultado qué son las interfaces este sería un buen momento para hacerlo en la sección de Temas Adicionales. La siguiente tabla muestra un resumen de las interfaces que nos permiten atender o capturar los eventos generados desde el exterior de un programa

Algunas Interfaces de eventos importantes

Interfaz	Propósito
ActionListener	Atrapar eventos tipo ActionEvent
KeyListener	Atrapar eventos tipo KeyEvent
MouseListener	Atrapar eventos tipo MouseEvent
MouseMotionListener	"

Y por su puesto, tambien hay funciones que nos permiten agregar las interfaces anteriores, no te preocupes mucho si no comprendes tablas, y tablas, mas bien son para una futura referencia rápida porque cuando miremos los ejemplos comprenderas mejor

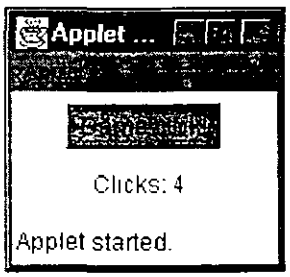
Métodos para agregar Listeners a los componentes

Método	Propósito
addComponentListener	Agrega un ComponentListener a un componente para que atrape los eventos que este genera
addFocusListener	Agrega un FocusListener a un componente que reciba las acciones de darle "atención" a dicho componente
addKeyListener	Agrega un KeyListener a un componente para que atrape los eventos del teclado
addMouseListener	Agrega un MouseListener para atrapar los eventos del mouse
addMouseMotionListener	Agrega u MouseMotionListener a un componente para recibir los eventos de movimiento del mouse generados por dicho componente.

El primer ejemplo es un Applet con un botòn que incrementa una variable a medida que hacemos click en él:

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

public class eventoBotonAlternativo extends Applet implements ActionListener {

int contarClicks;
Button botoncito;

public void init() {

botoncito = new Button("Dame click!"),
add(botoncito);
botoncito.addActionListener(this);

}

public void actionPerformed(ActionEvent event) {

++contarClicks;
repaint();

}

public void paint(Graphics g) {

g.drawString("Clicks: " + contarClicks, 40, 50),

}


}
```

Su salida es



El tamaño de la ventana del appletviewer puede variar tal como lo dije al principio, si en tu caso se ve muy diferente solo tienes que cambiar el tamañ. Revisemos el código, primero observa que importamos un paquete adicional.

import java.awt.event.*;

Esta paquete contiene todas las clases relacionadas con eventos, por lo que es indispensable. En la declaración de la clase implementamos la interfaz ActionListener (para entonces ya debiste haber checado la sección de Temas Adicionales donde se explican un poco mejor las interfaces).

public class eventoBotonAlternativo extends Applet **implements** ActionListener {

El siguiente paso es declarar una variable para contar los clicks y un botón:

int contarClicks,
Button botoncito,

El método init() se encarga de construir el botón y de agregarlo:

public void init() {

botoncito = new Button("Dame click!"),
add(botoncito),
botoncito addActionListener(**this**);

}

La función addActionListener() especifica el objeto que deseamos que "escuche" el evento que en este caso es el botoncito. No todas las clases poseen la función addActionListener, lo ideal es consultar en la Java API cuáles lo tienen y cuales no, en un curso mas avanzado verás que es relativamente sencillo colocarle a cualquier clase la posibilidad de agregar su propio ActionListener. El argumento de esta funcion es un "this" porque recuerda que al implementar una interfaz nuestra applet tambien se convierte en un objeto ActionListener así que podemos pasar el applet misma como argumento, es muy importante comprender que el **this** no significa que el Applet vaya a escuchar el evento sino que el applet es la encargada de manejar la reacción al evento (mediante la interfaz) y que el objeto que "escucha" es el botón en cuestion Despues viene la función actionPerformed que es parte de la interfaz, por lo tanto obligadamente debemos incluirla tal como el ActionListener lo exige, ésta funcion es la encargada de ejecutar la acción asociada con presionar el botón por eso incrementa la variable contarClicks y finalmente actualiza el valor exhibido en pantalla mediante un repaint()

public void actionPerformed(ActionEvent event) {

++contarClicks,
repaint(),

}

Su argumento (un ActionEvent llamado event) es parte de la interfaz, por lo tanto no podemos cambiarlo Es un objeto que contiene todas las propiedades de la acción ejecutada, si observas la clase ActionEvent en la Java API veras que tiene unas funciones que nos permiten obtener información importante acerca del evento y el lugar en el

que ocurrió, esto lo veremos después con más detalle. Finalmente el applet exhibe el valor de la variable contarClicks:

```
public void paint(Graphics g) {

g.drawString("Clicks: " + contarClicks, 40, 50);

}
```

Existe una manera distinta que produce exactamente el mismo efecto de la clase anterior sin la necesidad de implementar un ActionListener directamente sobre el Applet. Para esto creamos a parte nuestro "Listener" mediante una clase pequeñita que sí implemente el ActionListener, observa.

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

public class eventoBoton extends Applet {

int contarClicks;

Button botoncito;

public void init() {

botoncito = new Button("Dame click!");
add(botoncito);
botoncito.addActionListener( new escucharBoton() );

}

public class escucharBoton implements ActionListener {

public void actionPerformed(ActionEvent event) {

  ++contarClicks;
  repaint();

}
}
public void paint(Graphics g) {

g.drawString("Clicks: " + contarClicks, 40, 50);

}


}
```

Lo único distinto con respecto al primer ejemplo es

```
public class escucharBoton implements ActionListener {

public void actionPerformed(ActionEvent event) {

++contarClicks;
repaint();

}
}
```

Esta es una clase dentro de nuestra applet (tambien conocidas como innerclases o clases internas), estas clases son iguales a las demás, o sea que pueden tener constructor, sus propias variables y sus propios métodos con

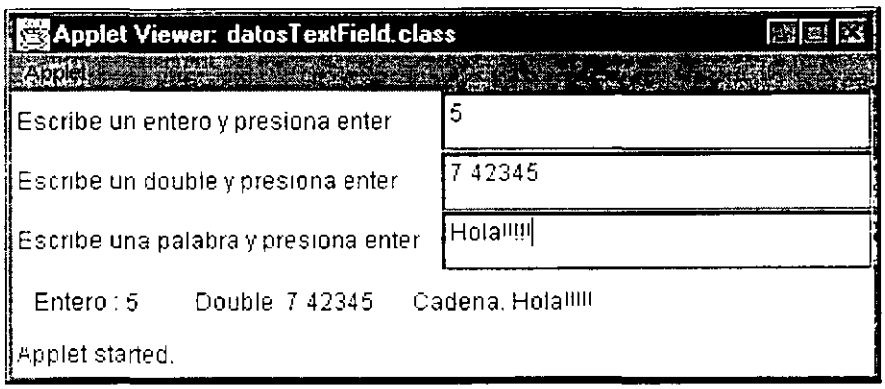
esto creamos un objeto **a parte** que implemente la interfaz ActionListener, aunque esta clase sea interna puede acceder libremente a las variables de la clase que la envuelve, por ejemplo a contarClicks, sucede lo mismo con sus métodos Como el objeto que implementará el ActionListener ya no es nuestra applet ahora pasamos como argumento al addActionListener un "nuevo" escucharBoton que si la implementa, aunque nuestra clasecita no disponga de constructor podemos crearlo mientras no pasemos ningún argumento: botoncito.addActionListener(**new escucharBoton()**); esto es válido. Como observaciones finales recuerden: 1) Elegimos el ActionListener porque ésta interfaz nos permite manejar **eventos de click** sobre un botón tal como decía en la primera tabla 2) La clase que implementa la interfaz elegida no es la que escuchará el evento, solo es la que **definirá la acción** que exige la interfaz y que se pretende realizar 3) Hay que **agregar el ActionListener** a algún objeto mediante addTipoDeListener().

Ahora veamos otro ejemplo, esta vez se pasan valores adquiridos en un TextField a variables globales de una clase

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

public class datosTextField extends Applet implements ActionListener {

Label entero, numDouble, texto;
TextField tentero, tnumDouble, ttexto;

int digitoEntero,
double digitoDouble,
String cadena,

public void init() {

entero = new Label("Escribe un entero y presiona enter:");
numDouble = new Label("Escribe un double y presiona enter ");
texto = new Label("Escribe una palabra y presiona enter");

tentero = new TextField();
tnumDouble = new TextField();
ttexto = new TextField(),

setLayout( new GridLayout( 4, 2 ) ),

add( entero ),
add( tentero );
add( numDouble );
add( tnumDouble ),
add( texto ),
add( ttexto );

tentero.addActionListener(this),
tnumDouble.addActionListener(this),
ttexto.addActionListener(this),

}

public void actionPerformed(ActionEvent event) {

if(event getSource() == tentero ) {
   digitoEntero = Integer parseInt( event getActionCommand() );
} else if (event.getSource() == tnumDouble) {
   Double d = new Double( event getActionCommand() ),
   digitoDouble = d doubleValue(),
} else if (event getSource() == ttexto) {
   cadena = ttexto getText();
}
repaint().
```

```
}

public void paint(Graphics g) {

g.drawString( "Entero : " + String.valueOf(digitoEntero), 10, 110 );
g.drawString( "Double: " + String.valueOf(digitoDouble), 90, 110 );
g.drawString( "Cadena· " + cadena, 200, 110 );

}


}
```

La salida del Applet es:



El Applet anterior es cargada con un ancho de 430 pixeles y un alto de 120 pixeles, eso significa que el TAG del applet debe quedar asi:

```
<APPLET CODE="datosTextField class" WIDTH="430" HEIGHT="120">
```

En esta ocasión si es importante el tamaño pues estamos usando un Layout (acomodamiento) llamado GridLayout, este Layout nos permite dividir la ventana del applet en líneas y columnas de manera muy similar al GridBagLayout sin embargo es mucho más sencillo de configurar y definir. En este ejemplo también implementamos un ActionListener que permite escuchar el evento que deseamos atender. oprimir enter para pasar datos a variables. Una vez importado el paquete de eventos e implementado el ActionListener, tenemos:

```
Label entero, numDouble, texto,
TextField tentero, tnumDouble, ttexto,

int digitoEntero;
double digitodouble,
String cadena,
```

Estas son tres etiquetas y tres campos de texto. las etiquetas sirven para pedir al usuario un tipo de dato especifico. los campos de texto son para recibir la informacion Las últimas tres variables: un int, un double y un String almacenaran los datos que vamos a solicitar En la funcion init() inicializamos tanto las etiquetas como los campos con sus respectivos constructores

```
entero = new Label("Escribe un entero y presiona enter ");
numDouble = new Label("Escribe un double y presiona enter:"),
texto = new Label("Escribe una palabra y presiona enter "),

tentero = new TextField();
tnumDouble = new TextField(),
ttexto = new TextField();
```

Después fijamos como Layout (acomodador) un nuevo GridLayout, el constructor del GridLayout recibe dos argumentos, el primero es el número de líneas en las que hay que dividir el Applet y el segundo es el número de columnas Es posible llamar al constructor sin la necesidad de nombrar nuestro objeto GridLayout pero esto

necesariamente implica que no lo podremos modificar (porque no establecimos su nombre y por lo tanto no lo podemos "llamar"), en estas ocasiones lo anterior no tiene mucha importancia puesto que el GridLayout únicamente lo utilizamos al principio y una vez creado no tenemos la menor intención de modificarlo. Después de fijar el Layout agregamos los componentes uno por uno, el orden es importante para un GridLayout porque el primer componente se colocará en la primera linea comenzando en la izquierda, si la primera celda ya esta ocupada entonces se continúa con la segunda y asi sucesivamente, siempre de izquierda a derecha. Una vez que una linea esta completa se pasa a la siguiente linea inferior hasta llegar al final.

```
setLayout( new GridLayout( 4, 2 ) ),

add( entero );
add( tentero );
add( numDouble );
add( tnumDouble ),
add( texto ),
add( ttexto ),

tentero.addActionListener(this);
tnumDouble.addActionListener(this);
ttexto.addActionListener(this);
```

Por último el ActionListener que define el Applet se agrega a cada uno de los campos de texto, ellos serán los que recibirán la acción de la tecla enter. Ahora solo resta definir la acción por realizar con el método actionPerformed que exige la interfaz ActionListener, este método recibe un objeto de tipo ActionEvent nombrado "event", dentro del método tenemos un if-else con tres condiciones, la clase ActionEvent hereda la función getSource() de la clase EventObject, el propósito de esta función es obtener el objeto sobre el cual se ha realizado el evento, las comparaciones de objetos son válidas en Java aunque es una idea un tanto diferente a lo que estamos acostumbrados porque seria más familiar para nosotros comparar enteros o cadenas, es importante que desde ahora tengan presente que es posible comparar objetos Una vez que se comprueba que el objeto sobre el cual se ejecutó el evento es un "tentero" (campo de texto que recibe enteros) entonces se procede a pasar el entero contenido en el TextField a la variable digitoEntero

```
public void actionPerformed(ActionEvent event) {

if(event.getSource() == tentero ) {
   digitoEntero = Integer parseInt( event getActionCommand() ),
} else if (event getSource() == tnumDouble) {
   Double d = new Double( event.getActionCommand() ),
   digitodouble = d doubleValue(),
} else if (event getSource() == ttexto) {
   cadena = ttexto getText(),
}
repaint(),

}
```

La clase Integer contiene muchas funciones públicas que nos permiten convertir diversas variables (como bytes, String o chars) en su contraparte entera una de tales funciones públicas es parseInt() (pasar entero) que recibe un String como argumento Como no utilizamos en ningun momento el constructor de Integer significa que la función parseInt() no solo es public sino tambien debe ser static Para obtener la cadena escrita en el campo de texto utilizamos la funcion getActionCommand() de la clase ActionEvent (recuerden que event es de este tipo)

```
digitoEntero = Integer parseInt( event.getActionCommand() ),
```

Si la segunda condicion del if-else resulta ser verdadera significa que enter fue oprimido en el segundo campo de texto, este campo obtiene un dato double mediante

```
Double d = new Double( event getActionCommand() ),
digitodouble = d doubleValue(),
```

Primero se pasa la cadena obtenida del evento con event.getActionCommand(), la clase Double (con D mayúscula) sirve como un puente entre diversos tipos de datos (tales como bytes, String) para convertirlos en double Es importante recalcar que un Double es distinto que un double, java distingue entre mayúsculas y

minúsculas, lo que significa que debemos convertir el valor Double "d" creado a un double ordinario, esta conversión se realiza mediante la función public doubleValue() de la clase Double. Una vez realizada la conversión ahora si podemos asignar su resultado a la variable double "digitodouble" Finalmente si la fuente del evento se identifica como el tercer campo de texto tenemos:

```
cadena = ttexto.getText(),
```

Que es la acción más simple, solo llamamos a la función getText() del componente TextField nombrado "ttexto", prácticamente los campos de texto están diseñados para trabajar con Strings por eso solo se obtiene el texto del componente en forma de String y se le asigna a nuestra variable cadena que también es un String. Finalmente se llama a un repaint() dentro del método actionPerformed() que actualizará y mostrará los datos recién tomados del teclado, la solitaria función paint del final se encarga de exhibir los datos:

```
public void paint(Graphics g) {

g.drawString( "Entero : " + String.valueOf(digitoEntero), 10, 110 ),
g.drawString( "Double: " + String valueOf(digitoDouble), 90, 110 );
g drawString( "Cadena  " + cadena, 200, 110 ),

}
```

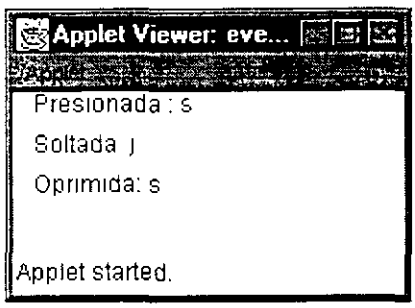
Es importante que observes que creamos 4 líneas y 2 columnas con el GridLayot, o sea un espacio para 8 componentes sin embargo solo agregamos seis componentes por lo que la última linea esta vacia en sus dos columnas, el paint() anterior exhibe los resultados de los datos capturados en ese espacio vacio

El penúltimo ejemplo recurre a la interfaz KeyListener Esta interfaz sirve para atender los eventos del teclado, existen tres posibilidades: que una tecla sea oprimida, que una tecla sea soltada o que una tecla haya sido presionada (combinación de los dos anteriores) Veamos el ejemplo:

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

public class eventosTeclado extends Applet implements KeyListener {

char presionada, soltada, oprimida;

public void init() {

addKeyListener(this);
requestFocus(),

}

public void keyPressed(KeyEvent event) {

presionada = event getKeyChar(),
repaint();

}

public void keyReleased(KeyEvent event) {

soltada = event.getKeyChar(),
repaint(),

}

public void keyTyped(KeyEvent event) {

oprimida = event getKeyChar(),
repaint(),

}
```

```
public void paint(Graphics g) {

g.drawString( "Presionada : " + String.valueOf(presionada), 10, 10 );
g.drawString( "Soltada. " + String.valueOf(soltada), 10, 30 );
g.drawString( "Oprimida: " + String.valueOf(oprimida), 10, 50 ),

}

}
```

Su salida es



La interfaz KeyListener define tres métodos: keyPressed(), keyRelased() y keyTyped() por lo tanto debemos incluirlos en nuestra applet obligatoriamente para definir las acciones a realizar cada vez que se dispare cada uno de estos eventos Una vez importado el paquete de eventos e implementado la interfaz KeyListener declaramos las variables que almacenaran la tecla oprimida en forma de char (porque las letras son caracteres), la idea entonces es mostrar la tecla que se ha oprimido

```
char presionada, soltada, oprimida,

public void init() {

addKeyListener(this);
requestFocus(),

}
```

La funcion init() agrega el KeyListener a toda el applet, esta vez no lo agregamos a ningún objeto en especifico porque deseamos que se exhiba la tecla oprimida siempre que el applet sea la ventana activa Después solicitamos "atención" del teclado para recibir estos eventos mediante la funcion requestFocus(), sin esta función el programa anterior no funciona Las siguientes funciones que exige la interfaz KeyListener (como lo puedes comprobar en la Java API) hacen lo mismo, pasar la tecla oprimida hacia las variables char, esto es reconocer la tecla oprimida y pasar su nombre en formato de un char a cada una de las variables, de la misma manera se llama a repaint() para que actualice los valores recién adquiridos

```
public void keyPressed(KeyEvent event) {

presionada = event.getKeyChar(),
repaint(),

}

public void keyReleased(KeyEvent event) {

soltada = event getKeyChar();
repaint(),

}

public void keyTyped(KeyEvent event) {
```

```
oprimida = event.getKeyChar();
repaint();

}
```

Y por último la función paint() se encarga de exhibir los datos:

```
public void paint(Graphics g) {

g.drawString( "Presionada : " + String valueOf(presionada), 10, 10 );
g.drawString( "Soltada: " + String.valueOf(soltada), 10, 30 );
g.drawString( "Oprimida: " + String.valueOf(oprimida), 10, 50 );

}
```

El programa anterior es sumamente sencillo pero existen algunas interfaces más exigentes, por ejemplo la interfaz MouseListener exige cinco funciones por lo que podría generar bastante código lo cual sería tedioso si solo deseamos usar solo una de dichas funciones, existen un tipo de clases llamadas "Adapters" o adaptadores que heredan de otras clases que ya han implementado algunas interfaces tales como MouseListener o KeyListener, así existe por ejemplo la clase MouseAdapter o KeyAdapter que ya implementaron los Listeners respectivos, estas clases ahorran un poco de código porque puedes heredarlas fácilmente y como ya incluyen las funciones que exige cada una de sus interfaces podrás llamar únicamente a la función del evento que te interesa.

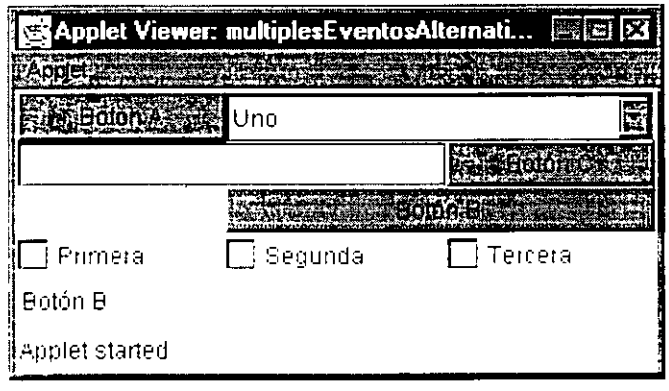
El último ejemplo recurre a otro que hicimos anteriormente (el que explica los GridBagLayouts) y muestra como identificar algunos eventos provenientes de otros componentes mostrados en esta sección tales como Checkboxes, Choice Button o como diferenciar su tipo para saber de dónde proviene el evento, de igual manera implementa dos interfaces al mismo tiempo La interfaz ItemListener es necesaria para identificar los eventos provenientes de un Choice Button, veamos el ejemplo y después su explicación correspondiente

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

public class multiplesEventosAlternativo extends Applet implements ActionListener, ItemListener {

Button a, b, c,
Choice d,
Checkbox e, f, g;
TextField h,
Label i,

GridBagLayout reticula,
GridBagConstraints restricciones,

String elegido,

void addComponent( Component comp, GridBagLayout bag, GridBagConstraints bagcons,
                   int linea, int columna, int ancho, int alto ) {

bagcons gridx = columna,
bagcons gridy = linea,
bagcons gridwidth = ancho
bagcons gridheight = alto,

bag setConstraints( comp, bagcons );
add( comp ),

}

public void init() {

reticula = new GridBagLayout(),
setLayout( reticula ),
restricciones = new GridBagConstraints(),
```

```
a = new Button("Botón A");
b = new Button("Botón B");
c = new Button("Botón C");
d = new Choice();

d.add("Uno");
d add("Dos");
d.add("Tres");

e = new Checkbox("Primera");
f = new Checkbox("Segunda");
g = new Checkbox("Tercera");
h = new TextField();
i = new Label("Etiqueta en la cuarta linea");

restricciones weightx = 1;
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH;
addComponent(a, reticula, restricciones, 0, 0, 1, 1 );

restricciones.weightx = 1,
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH;
addComponent(b, reticula, restricciones, 2, 1, 2, 1 );

restricciones weightx = 1;
restricciones.weighty = 1;
restricciones.fill = GridBagConstraints.BOTH,
addComponent(c, reticula, restricciones, 1, 2, 1, 1 );

restricciones.weightx = 1,
restricciones weighty = 1,
restricciones fill = GridBagConstraints.BOTH;
addComponent(d, reticula, restricciones, 0, 1, 2, 1 ),

restricciones weightx = 1;
restricciones weighty = 1,
restricciones fill = GridBagConstraints BOTH,
addComponent(e, reticula, restricciones, 3, 0, 1, 1 ),

restricciones weightx = 1,
restricciones weighty = 1,
restricciones fill = GridBagConstraints BOTH,
addComponent(f, reticula, restricciones, 3, 1, 1, 1 ),

restricciones weightx = 1,
restricciones weighty = 1,
restricciones fill = GridBagConstraints BOTH;
addComponent(g, reticula, restricciones, 3, 2, 1, 1 ),

restricciones.weightx = 1,
restricciones.weighty = 1,
restricciones fill = GridBagConstraints.BOTH,
addComponent(h, reticula, restricciones, 1, 0, 2, 1 );

restricciones weightx = 10,
restricciones.weighty = 1,
restricciones.fill = GridBagConstraints BOTH,
addComponent(i, reticula, restricciones, 4, 0, 3, 1 ),

a.addActionListener( this ),
b addActionListener( this ),
c addActionListener( this ),
```

```
d addItemListener( this );
e.addItemListener( this );
f.addItemListener( this );
g addItemListener( this ),
h.addActionListener( this );

}

public void actionPerformed(ActionEvent event) {

i setText(event.getActionCommand());

}

public void itemStateChanged(ItemEvent evt) {

if( evt getSource() instanceof Choice ) {
   elegido = d.getSelectedItem(),
} else if ( evt.getSource() instanceof Checkbox) {
   if(evt.getSource() == e) {
      elegido = e.getLabel(),
   } else if(evt.getSource() == f) {
      elegido = f.getLabel(),
   } else if(evt.getSource() == g) {
      elegido = g.getLabel(),
   }
}
i setText(elegido);

}

}
```

Su salida es·



La idea de este ejemplo es crear un GridBagLayout que nos permita exhibir diversos componentes, en la última linea agregamos una Label que exhibirá el componente seleccionado ya sea un botón, el item de una lista o una Checkbox Como ya explicamos en la seccion de Layouts las propiedades de un GirdBagLayout me limitaré a explicar los metodos que exigen las interfaces, el primero actionPerformed() es de la interfaz ActionListener·

```
public void actionPerformed(ActionEvent event) {

i setText(event.getActionCommand()),

}
```

Lo único que hace el método anterior es tomar la etiqueta (en el caso de los botones) o el texto escrito (en el caso del campo de texto) y fijar dicha String tomada a la etiqueta (Label) para que la exhiba. El siguiente método itemStateChanged() se encarga de definir la reacción para los componentes tipo Choice o tipo Checkbox, la

interfaz ItemListener maneja eventos de elementos seleccionados, dentro de la siguiente función tenemos primero un if-else cuya condición es: evt.getSource() instanceof Choice, el evt.getSource() regresa el objeto sobre el cual se realizó la acción mientras que el operador boleano "instanceof" regresa true si el objeto anterior es un Choice y regresa un false en caso contrario. El operador instanceof es muy útil para distinguir entre diversos tipos de objetos, como itemStateChanged() atrapa tanto eventos de Choice como de Checkboxes debemos distinguirlos antes de pensar en definir una reacción para cada uno Regresando a la primera condición, si esta es true entonces se pasa el elemento elegido del componente "d" a la variable String "elegido".

```
public void itemStateChanged(ItemEvent evt) {

if( evt getSource() instanceof Choice ) {
   elegido = d.getSelectedItem(),
} else if ( evt getSource() instanceof Checkbox) {
        if(evt.getSource() == e) {
           elegido = e getLabel();
        } else if(evt getSource() == f) {
           elegido = f getLabel(),
        } else if(evt getSource() == g) {
           elegido = g getLabel(),
        }
}

i setText(elegido),

}
```

La segunda condición del if-else más exterior identifica que el evento ha ocurrido en un objeto de tipo Checkbox, en tal caso ahora debemos definir en cual de los tres checkboxes que tenemos ha ocurrido, para eso es el if-else anidado, éste determina cuál de los tres checkboxes ha sido elegido y para los tres casos toma su etiqueta y la almacena en la variable de tipo String "elegido", todos los if-else garantizan que solo un valor se pasara a la variable "elegido", finalmente pasamos ese String determinado a la etiqueta (Label) nombrada "i" que exhibe el resultado en la parte inferior del Applet.

Existen muchos eventos, muchas interfaces que los manejan y muchos adaptadores que simplifican su manejo, este tema es extenso y laborioso pero si comprendes la manera de utilizar la Java API no necesitarás de ningún ejemplo, para entonces si necesitas recurrir a algún componente o a algún evento solo leerás unas cuantas lineas de código de la Java API y tomaras lo que mas se adapte a tus necesidades de forma simple y natural, tal como tomar la herramienta de una caja

Aplicaciones Simples

- Trabajando con Aplicaciones Independientes
- Instrucciones para trabajar con Aplicaciones y RealJ
- Aplicaciones Gráficas que trabajan desde Windows
- Frames
  - Componentes y Frames
    Menu, MenuBar y Eventos
    Abriendo múltiples Frames

Otra de las sonadas ventajas de Java se encuentra en que en teoria es independiente de la plataforma, esto significa que las aplicaciones o las applets deben comportarse aproximadamente de la misma forma sin importar que las ejecutes desde Windows, Solaris o Macintosh. Sin embargo muchas empresas enemigas de Sun Microsystems afirman que esta compatibilidad aún no es plena y que tomará aún muchos años más, sean compatibles o no las aplicaciones en Java son muy fáciles de crear.

Ahora es momento de comenzar con una introducción sencilla a las aplicaciones tal como las conocemos, es decir, programas "independientes" que pueden trabajar aislados sin la necesidad de Internet Explorer o de Netscape Navigator, estas aplicaciones se pueden crear para trabajar desde una linea de comandos como DOS o de forma gráfica para Windows, no tocaré el tema de la linea de comandos porque pienso que a la mayoria de ustedes les interesan más las aplicaciones gráficas, además, si estudian computación en cualquiera de sus formas ya les tocará en el futuro trabajar de esa manera asi que mucha suerte. Nos ocuparemos de introducir el uso de la clase "Frame" que nos permite crear ventanas sencillas, de las clases "Menu" y "MenuBar" para crear menús y de como manejar eventos simples.

Trabajando con aplicaciones independientes

Nota Importante: Antes de que bajes algo, primero lee toda esta sección hasta el primer ejemplo, si tienes RealJ y deseas ahorrarte problemas incluso puedes saltarte esta sección hasta las "Instrucciones para trabajar con Aplicaciones Java y Real J"

Como he platicado en la introduccion las aplicaciones Java realmente no corren aisladas por si solas, de hecho ni siquiera las aplicaciones convencionales que todos utilizamos diariamente como Word, Internet Explorer, Winamp, ICQ, etc lo hacen, algunas veces requieren de diversos paquetes y de bibliotecas de funciones que en algunas ocasiones ya se encuentran en Windows pero en otras deben ser obtenidos desde internet, tal es el caso de algunas aplicaciones creadas con Visual Basic Para que estas se ejecuten puedes necesitar el Visual Basic Runtime Environment (VBRE), en el caso de Java una aplicación necesita de la Java Virtual Machine (que la interpreta) y de una biblioteca de clases (a la que recurre) para poder ejecutarse, afortunadamente para nosotros al igual que en Visual Basic solo es necesario bajar una sola vez estos datos y una vez hecho todas nuestras aplicaciones Java funcionaran, este paquete se llama Java Runtime Environment o JRE, obviamente no es indispensable para poder programar porque su único fin es ejecutar aplicaciones o applets de Java (mediante un plug-in), por tal motivo **si ya tienes algún JDK no necesitas bajar el JRE.** Pero si aún asi deseas obtenerlo puedes hacerlo desde el siguiente link, también existen diversas versiones del JRE que generalmente aparecian a la par del JDK, actualmente el JRE más moderno es la versión 1.4 y está incluido dentro del paquete del J2SE 1.4, las empresas informaticas de internet con trabajos y con gran lentitud se están adaptando apenas a la versión 1.3, eso se debe a que Java esta creciendo (en terminos de bibliotecas de clases) mucho más aprisa que la industria.

Ir a la pagina de Sun para obtener el Java Runtime Environment(v1.3)

Existe una contradicción muy curiosa relacionada con las aplicaciones creadas en Java, he leido en incontables foros de programación que muchas personas desean que sus aplicaciones se ejecuten desde un archivo ejecutable (*.exe) en vez de tener que hacerlo desde un archivo de clase (*.class) o un archivo Jar (*.jar) Me imagino que se debe a que es mucho mas sencillo darle un doble click al ejecutable para correrlo mientras que el * class debe ejecutarse mediante un comando que llame al intérprete Java o bien debemos crear un archivo que llame al comando automáticamente, aunque sea poco el esfuerzo todos desean ver un ejecutable en vez de una clase También he leido que los expertos siempre les responden a estas personas con algo asi

"Se supone que una de las ventajas de Java es que es independiente de plataforma, por lo que no necesitas crear un archivo ejecutable, una clase puede correrse desde cualquier plataforma mientras que un archivo ejecutable * exe en la mayoria de los casos no."

Aún asi, la insistencia de crear programas Java que se llamen desde un ejecutable * exe es impresionante Yo concuerdo en que no es necesario crear un archivo *.exe pero también puedo notar que no se ha ofrecido una solución un poco más simple para este posible problema. Bueno, te voy a platicar lo que hice y de ti dependerá elegir pero estoy casi completamente segura de que mi solución es la más sencilla para los principiantes Primero, si deseas ejecutar aplicaciones Java que se encuentran en *.jar entonces solo necesitas bajar el JRE v1.3 o en su defecto el J2SE v1.4 porque ambos incluyen parámetros que automáticamente modifican Windows para que ejecuten los *.jar como si fueran aplicaciones (con un doble click) Segundo, si lo que deseas es ejecutar *.class existe un programita bastante pequeñito creado por un tal Kevin Kuang llamado **Java Runner**, este programa soluciona el problema de los comandos, ya que una vez instalado hará que los archivos * class que se encuentren en tu computadora se comporten como un ejecutable *.exe, ahh pero no es tan simple como bajarlo e instalarlo, primero necesitas que el intérprete del JDK (java exe) o el del JRE se encuentre en el classpath de tu archivo autoexec.bat, si deseas aventurarte primero necesitas bajar el programa (bajo tu propia responsabilidad)

http://www.download.com
(búscalo con "search" como Java Runner, en la sección de downloads)

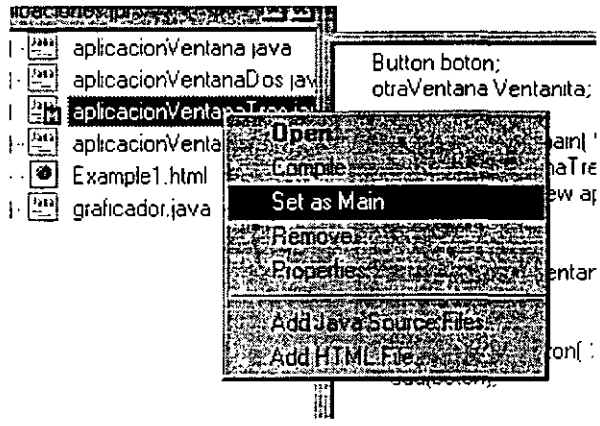
Te repito insistentemente que esto no es indispensable para trabajar con los ejemplos de esta sección, basta con el JDK 1.1 8 y el Real J, solo te lo dejo como algo opcional, en el archivo readme del Java Runner vienen unas instrucciones (en inglés) para agregar el classpath a tu archivo autoexec.bat, aqui está la traducción

1. Presiona: Inicio - Buscar - Archivos o Carpetas
2. Escribe java.exe en "Nombre", elige "Discos duros locales" en "buscar en" y presiona Buscar ahora
3 Cuando el archivo sea encontrado, anota su ubicación.
4 Presiona: Inicio - Ejecutar
5. Escribe: 'edit c \autoexec.bat'
6 Agrega una linea al final del archivo que contenga 'path=%path%;<####>
en donde #### es la ubicación completa del archivo java exe, esta ubicación no debe terminar con 'java exe'
7 Reinicia tu computadora si es necesario para que el cambio tenga efecto
8 Se incluyen dos programas de prueba, después de la instalación da click con el botón derecho del mouse en el archivo test class (ubicado en la carpeta 'test') y después da click en Run(console) para ver si funciona
9 Para una prueba de fuego da doble click en el archivo testSwing.class (también ubicado en la carpeta 'test') para ver la aplicacion Java que usa la interface Swing (solo para JDK ó JRE versiones 1 1.8 y superiores) Si no funciona entonces significa que necesitas una version más nueva del JDK o del JRE

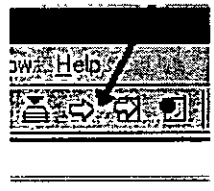
Suponiendo que no hubo ningún problema (espero!) entonces ya estas en posibilidad de ejecutar tus programas java mediante un doble click igual que los archivos ejecutables * exe convencionales, el mismo Java Runner tiene unos programas de ejemplo para probar esta técnica. Es muy importante que sepas que solo las clases que poseen un metodo main (esto se explica mas adelante) se ejecutarán con doble click, si intentas ejecutar un applet de esta manera o alguna otra clase sin método main, no funcionará.

Instrucciones para trabajar con Aplicaciones y RealJ

Tener instalado Real J facilita mucho las cosas, los archivos de código se editan exactamente igual que todos los ejemplos anteriores la única diferencia radica en que debes especificarle a Real J que archivo de código es el que contiene la funcion "main" y una vez hecho esto la podras llamar en cualquier momento. ¿Recuerdan que habia una tal funcion init() muy utilizada en las applets? esa funcion era la primera que se ejecutaba de toda el applet por default (de forma predeterminada), bueno pues sucede que hay una funcion similar para los programas independientes, esta función es la dichosa función "main", sirve para "arrancar" nuestro programa. Casi todos los siguientes ejemplos tienen funcion main, una vez que copies su código hacia un archivo en Real J deberás seleccionarlo y darle un click con el botón derecho del mouse, aparecera el siguiente menu:



A continuación deberás darle un click a "Set as Main", al hacerlo le aparecerá una pequeña "M" rosada al archivo, esto significa que la asignación fue exitosa. El proceso de compilación se mantiene igual, la única ventaja de fijar un archivo de código como "Main" radica en que Real J se encargará de correrlo automáticamente para tí (ya que este compilado), puedes hacerlo con el Menú Build, y después en Run Application, o con el siguiente botón:



Eso te ahorra al menos enredarte con el classpath del archivo autoexec.bat y con otros problemas subsecuentes.

Aplicaciones Gráficas que trabajan desde Windows

Bueno, ya basta del bla, bla, bla, ahora viene lo interesante Nuestra primera aplicación será una ventana sencilla que mostrará un texto mediante un g.drawString(); es muy importante que en este momento ya comprendas de forma aceptable la Programación Orientada a Objetos ya que en este ejemplo se aplican constructores, herencia y el manejo básico de las clases (con la Java API, cosa que puedes repasar). Naturalmente estas aplicaciones no son exclusivas para Windows ya que en teoría deben funcionar en otros sistemas operativos mientras posean su respectiva JVM

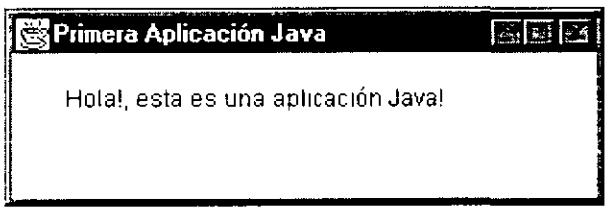
Frames

Los Frames son ventanas casi exactamente iguales a las presentes en Windows, para crear un Frame podemos tomar la clase del mismo nombre ya existente en Java o bien podemos crear una clase que herede todas las características de Frame, como lo segundo es mas ilustrativo es exactamente lo que haremos. Aunque los Frames son muy parecidos a las ventanas convencionales debemos indicarle por ejemplo el tipo de acción que debe ejecutarse cuando cerramos la ventana, por mucho que una ventana sea un componente relativamente simple aun así debemos escribir el código que inicializa y que la cierra apropiadamente, yo pense alguna vez que sería mejor que al crear una ventana esta ya contara con todas estas características que son **obvias** más que nada por la costumbre de usar windows pero pensandolo bien es mejor definir los eventos por nosotros mismos porque en muchas ocasiones desearemos comportamientos distintos a los que estamos acostumbrados Como ya es costumbre primero presento el programa y despues su explicacion.

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

public class aplicacionVentana extends Frame {

   public static void main( String args[ ] ) {
     aplicacionVentana ventana = new aplicacionVentana( "Primera Aplicación Java" );
   }

   public aplicacionVentana(String titulo) {
     super( titulo );
```

```
    setBounds(50, 50, 300, 100);
    setVisible(true);
    addWindowListener(new cerrarVentana() );
  }

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

  public void paint( Graphics g ) {
    g.drawString("Hola!, esta es una aplicación Java!", 30, 50);
  }

}
```

La salida del programa anterior es:



La estructura de una aplicación Java es muy similar a la de las clases, primero importamos el paquete java awt.*, porque de dicho paquete utilizaremos la clase Graphics (para mostrar el texto) y el de eventos java.awt.event.*; porque esta vez utilizaremos un WindowListener que sera el encargado de manejar el evento de cerrar la ventana, este es un evento porque siempre tenemos que hacer click al botón superior izquierdo marcado con una X. El nombre de nuestro primer programa es "aplicacionVentana", primero tenemos:

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

public class aplicacionVentana extends Frame {

  public static void main( String args[ ] ) {
    aplicacionVentana ventana = new aplicacionVentana( "Primera Aplicación Java" );
  }
```

Como ven el "extends" implica que heredaremos las funciones y variables de la clase "Frame", esta clase nos permitira exhibir una ventana propiamente dicha tal como las que conocemos estilo Windows. Cuando se quiere ejecutar una aplicacion todo intérprete Java busca inmediatamente el método main (principal), este método tiene un proposito muy similar al init() de las applets, es decir que **es la primera sección de código que se ejecuta** por lo tanto en esta funcion inicializaremos todo con sus respectivos constructores, es importante notar que por mucho que esta funcion main sea la principal para iniciar el programa es ignorada dentro de la clase, es decir que en realidad nunca se llama explicitamente desde otra función, únicamente sirve para "arrancar" el programa. Toda clase que pretenda ser un programa independiente debe incluir un método main para arrancarlo aunque puedes llamar a otras clases ajenas desde este programa, por lo tanto aunque un programa conste de cientos de clases con un metodo main bastara para iniciarlo

La funcion main recibe un argumento de tipo String, este argumento generalmente se brinda desde el exterior del programa mediante comandos, como esta es una aplicacion que funcionará desde un sistema operativo gráfico no será necesario aplicar este argumento sin embargo hay que ponerlo. Dentro del main se inicia el programa declarando una nueva "aplicacionVentana" que es el nombre de nuestra clase, únicamente en la funcion main se puede hacer esto (que el programa se declare a si mismo) porque si no ¿de qué otra forma iniciaria esta aplicacion?, es indispensable respetar la notación del main con los modificadores: public static y como es de tipo void no regresa ningun valor Despues tenemos.

```
  public aplicacionVentana(String titulo) {
    super( titulo ),
    setBounds(50, 50, 300, 100);
    setVisible(true);
    addWindowListener(new cerrarVentana() );
  }
```

Esta función es el constructor de nuestra clase (por eso se llama igual que la misma clase), debido a que heredamos de la clase "Frame" entonces debemos llamar a su constructor (con un "super"), pueden revisar la clase Frame en la Java API para comprobar que recibe un argumento String tal como se hace aqui, la función setBounds(int x, int y, int ancho, int alto) se hereda desde la clase Component y sirve para definir el tamaño y la posición de la ventana que vamos a mostrar, la función setVisible(true) sirve para mostrar la ventana una vez que ha sido especificada. Finalmente agregamos un WindowListener a nuestra ventana especificado por una clase interna llamada cerrarVentana (innerclass), este addWindowListener sin operador punto agrega directamente al Frame el Listener definido por la siguiente clase

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

Esta clase hereda directamente de WindowAdapter, como recordarán de la seccion de Interfaz Gráfica y Eventos los adaptadores son clases que heredan de otras clases que ya implementaron algunas interfaces, a ver.. a ver ¿cómo esta esto? Implementar una interfaz exige que coloques cierto grupo de funciones en tus clases, la clase WindowAdapter ya ha implementado la interfaz WindowListener (aunque estos no hacen nada) que maneja los eventos generados al manipular una ventana tales como cerrarla, minimizarla o maximizarla, entre otros. Entonces, al heredar WindowAdapter podemos definir el método que deseamos atender (esto se llama sobrescribir métodos, porque el que definimos en la subclase oculta a los que se encontraban en la superclase con el mismo nombre) para producir una reacción cuando este se genere sin necesidad de definir todos los métodos que exige WindowListener, los "Adapters" son para ahorrarnos un poquito de código. Entonces este WindowAdapter es un WindowListener reducido del cual tomamos solamente la función windowClosing() porque deseamos que la ventana se cierre al hacer click en su botón superior derecho X La función dispose() libera todos los recursos que posea la ventana antes de salir del programa mediante System.exit(0); La última función es ya bien conocida por nosotros:

```
  public void paint( Graphics g ) {
    g.drawString("Hola!, esta es una aplicación Java!", 30, 50),
  }
```

Se encarga de dibujar la cadena en las coordenadas (30, 50) Ahora veamos un ejemplo más interesante que involucra colocar otros elementos dentro de una aplicación con Frame como etiquetas y botones.

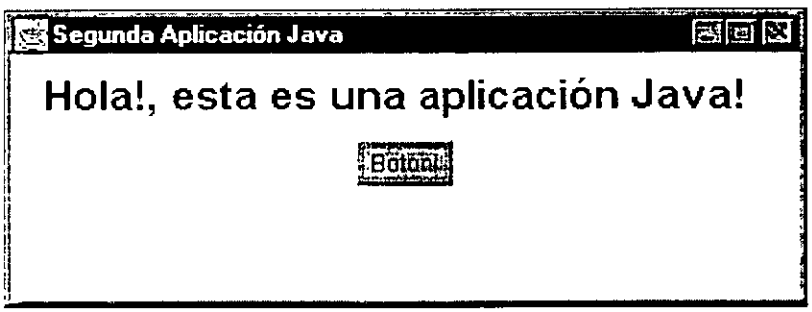
Componentes y Frames

Practicamente los componentes se manejan igual dentro de un Frame que dentro de un Applet, el siguiente ejemplo agrega un boton y un mensaje de texto con un tipo de letra elegida por nosotros.

```
import java.awt.*,
import java.awt.event.*,

public class aplicacionVentanaUno extends Frame {

    Font tipoDeLetras,
    Label etiqueta;
    Button boton,

   public static void main( String args[] ) {
     aplicacionVentanaUno ventana = new aplicacionVentanaUno( "Segunda Aplicación Java" ),
   }
```

```
    public aplicacionVentanaUno(String titulo) {

      super( titulo );

      tipoDeLetras = new Font( "Arial", Font.BOLD, 21);
      etiqueta = new Label( "Hola!, esta es una aplicación Java!" );
      etiqueta.setFont( tipoDeLetras );
      boton = new Button( "Boton!" );

      setLayout( new FlowLayout() );

      add( etiqueta );
      add( boton );

      setBounds(50, 50, 400, 150);
      setVisible(true);
      addWindowListener(new cerrarVentana() );

    }

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

    public void paint( Graphics g ) {
        //Cualquier tipo de dibujo que gustes incluir
    }

}
```

La salida del programa anterior es:



Primero tenemos

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

public class aplicacionVentanaUno extends Frame {

    Font tipoDeLetras;
    Label etiqueta;
    Button boton;

   public static void main( String args[] ) {
     aplicacionVentanaUno ventana = new aplicacionVentanaUno( "Segunda Aplicación Java" );
   }
```

Nuevamente importamos los paquetes pertinentes, también extendemos la clase Frame para poder mostrar la ventana, introducimos un nuevo tipo llamado "Font" llamado tipoDeLetras, inicializamos nuestro programa dentro del main de la misma forma en que lo hicimos en el primer programa, inicializaremos nuestras variables dentro del

constructor, no dentro del main porque el main se encarga de inicializar la ventana y la ventana debe encargarse ella misma de inicializar sus propios componentes:

```
public aplicacionVentanaUno(String titulo) {

    super( titulo );

    tipoDeLetras = new Font( "Arial", Font.BOLD, 21);
    etiqueta = new Label( "Hola!, esta es una aplicación Java!" );
    etiqueta.setFont( tipoDeLetras );
    boton = new Button( "Boton!" );

    setLayout( new FlowLayout() );

    add( etiqueta );
    add( boton );

    setBounds(50, 50, 400, 150);
    setVisible(true);
    addWindowListener(new cerrarVentana() );

}
```

Otra vez hacemos una llamada al constructor de nuestra superclase Frame mediante el "super(titulo)" que pasa el argumento de tipo String, inicializamos nuestro tipo Font mediante su constructor (cosa que puedes revisar en la Java API), el primer argumento es una cadena que indica el tipo de letra que usamos, el segundo argumento es la característica de dicha letra (puede ser negritas, cursivas o subrayadas o sea BOLD, ITALIC o PLAIN), el último argumento es el tamaño de la letra, en este caso: 21. Después inicializamos la etiqueta mediante el constructor que ya manejamos en la sección de Interfaz Gráfica sin embargo esta vez fijamos sobre esa etiqueta el tipoDeLetras recién iniciado mediante la función pública "setFont()", estas funciones para manipular las propiedades de las etiquetas también pueden encontrarse en la Java API (en este caso en la clase Component, que es superclase de Label), también se inicializa el botón. Finalmente dentro del constructor llamamos a setLayout(new FlowLayout()), esta función sirve para crear un nuevo FlowLayout que nos permita colocar la etiqueta y el botón, ya que la clase Frame no posee ninguno de manera predeterminada, el FlowLayout es como una pizarra para dibujar estos componentes que los ordena de acuerdo al orden de los "add()" de izquierda a derecha, puedes intercambiar su orden para experimentar, puedes ver una descripción más detallada de los FlowLayouts en la sección de Interfaz Gráfica, el llamado a las funciones add() tienen como propósito agregar los componentes a nuestra pantalla, el setBounds() fija el tamaño de la aplicación y el setVisible() se encarga de mostrarla en pantalla Por último

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

public void paint( Graphics g ) {
    //Cualquier tipo de dibujo que gustes incluir
}
```

Cumplen con exactamente la misma tarea que en el primer ejemplo, la innerclass cerrarVentana será el WindowListener (porque un WindowAdapter hereda de una clase que implementa un WindowListener) que nos permitirá cerrarla apropiadamente cuando se hace click en el botón superior derecho X.

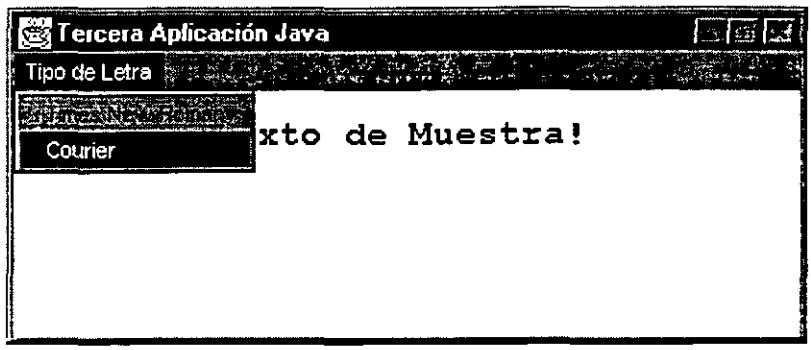
Menu, MenuBar y Eventos

Ahora, veamos un ejemplo de como incluir menus y barras de menús a una aplicación simple como las anteriores En esta ocasión al clickear los menús se cambia el tipo de letra de un texto de muestra exhibido en pantalla

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

```
public class aplicacionVentanaDos extends Frame implements ActionListener {

    Font tipoDeLetras;
    Label etiqueta;

    MenuBar barra;
    Menu primerMenu;

  public static void main( String args[] ) {
    aplicacionVentanaDos ventanaDos = new aplicacionVentanaDos( "Tercera Aplicación Java" );
  }

  public aplicacionVentanaDos(String titulo) {

    super( titulo );

    barra = new MenuBar();
    primerMenu = new Menu("Tipo de Letra");
    primerMenu.add("Times New Roman");
    primerMenu.add("Courier");

    barra.add( primerMenu );
    setMenuBar( barra );

    tipoDeLetras = new Font( "Arial Narrow", Font.BOLD, 21);
    etiqueta = new Label( "Texto de Muestra!" );
    etiqueta.setFont( tipoDeLetras );

    setLayout( new FlowLayout() );

    add( etiqueta );

    setBounds(50, 50, 400, 150);
    setVisible(true);
    addWindowListener(new cerrarVentana() );
    primerMenu addActionListener(this);

  }

  public void actionPerformed(ActionEvent e) {
      if (e getActionCommand() ==  "Times New Roman" ) {
      tipoDeLetras = new Font( "Times New Roman", Font.BOLD, 18 );
    } else if (e getActionCommand() == "Courier" ) {
      tipoDeLetras = new Font( "Courier", Font BOLD, 18 );
    }
   etiqueta setFont( tipoDeLetras );
  }

  public class cerrarVentana extends WindowAdapter {
   public void windowClosing(WindowEvent e){
  dispose();
  System exit(0);
   }
  }

  public void paint( Graphics g ) {
      //Cualquier tipo de dibujo que gustes incluir
  }

}
```

La salida del programa anterior es:



Este programa posee un menú con dos opciones, cada una cambia las propiedades del texto de la etiqueta exhibida en pantalla, uno le fija texto Times New Roman y el otro de tipo Courier, el inicio de nuestro programa es similar al de los anteriores:

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

public class aplicacionVentanaDos extends Frame implements ActionListener {
                                                          }
    Font tipoDeLetras;
    Label etiqueta;

    MenuBar barra,
    Menu primerMenu;

  public static void main( String args[] ) {
    aplicacionVentanaDos ventanaDos = new aplicacionVentanaDos( "Tercera Aplicación Java" ),
  }
```

Lo único novedoso con respecto a los ejemplos anteriores es la adición de dos tipos MenuBar y Menu, la primera es la barra de menús y la segunda es un menú que tendrá dos opciones, pueden revisar estas clases para entender como operan sus constructores, también implementamos la interfaz ActionListener que será la encargada de definir las reacciones al hecho de seleccionar un menú como veremos mas adelante, el constructor

```
  public aplicacionVentanaDos(String titulo) {

    super( titulo ),

    barra = new MenuBar();
    primerMenu = new Menu("Tipo de Letra"),
    primerMenu add("Times New Roman"),
    primerMenu.add("Courier");

    barra.add( primerMenu ),
    setMenuBar( barra ),

    tipoDeLetras = new Font( "Arial Narrow", Font BOLD, 21),
    etiqueta = new Label( "Texto de Muestra!" ),
    etiqueta setFont( tipoDeLetras ),

    setLayout( new FlowLayout() ),

    add( etiqueta ),

    setBounds(50, 50, 400, 150),
    setVisible(true);
    addWindowListener(new cerrarVentana() ),
    primerMenu.addActionListener(this),
```

```
}
```

Aquí se llama otra vez a super(), se inicializan nuestros objetos MenuBar y Menu mediante sus respectivos constructores, el primero no requiere argumentos mientras que el segundo sí los necesita, es una cadena que de hecho es el texto que aparecerá como titulo del menú, a continuación a ese primerMenu se le agregan dos opciones: "Times New Roman" y "Courier", una vez configurado nuestro Menú este debe ser agregado a la barra de menús, esto se hace con

```
barra.add( primerMenu ); //Agregar menú a barra de menús
setMenuBar( barra );  //Agregar barra de menús a la ventana
```

Parece que el código anterior es redundante pero Java exige que primero agregues el menú a la barra y que después agregues dicha barra a la ventana, son cosas distintas. Las llamadas a los constructores de tipoDeLetras y etiqueta y las funciones restantes son similares a las de ejemplos anteriores. Finalmente agregamos el respectivo Listener a la ventana para poder cerrarla apropiadamente, este se define en una innerclass posterior igual a los ejemplos anteriores, también se agrega un ActionListener al "primerMenu" con el fin de que atrape los eventos generados con el mouse, la acción que definirá éste Listener se describe más adelante Lo único nuevo dentro de lo que resta de este programa es la función actionPerformed, aquí se tiene un if-else, este método es el que exige la interfaz ActionListener y en él se definirá la reacción a realizar:

```
public void actionPerformed(ActionEvent e) {
   if (e getActionCommand() ==  "Times New Roman" ) {
     tipoDeLetras = new Font( "Times New Roman", Font BOLD, 18 );
   } else if (e.getActionCommand() == "Courier" ) {
     tipoDeLetras = new Font( "Courier", Font.BOLD, 18 ),
   }
 etiqueta.setFont( tipoDeLetras ),
}
```

Se encarga de atrapar la acción de dar click a alguna u otra de las opciones del menú, el primer if prueba la condición "e.getActionCommand()=="Times New Roman", el objeto nombrado e es un ActionEvent y la función getActionCommand regresa la etiqueta que contiene el menú, si este coincide con "Times New Roman" entonces el tipoDeLetras adquiere el valor de la nueva fuente configurada, la segunda condición del if-else es muy similar, finalmente cualquiera haya sido el caso se fija el tipo de letras a la etiqueta mediante un setFont(). Las funciones restantes ya son conocidas por las primeras explicaciones.

Abriendo múltiples Frames

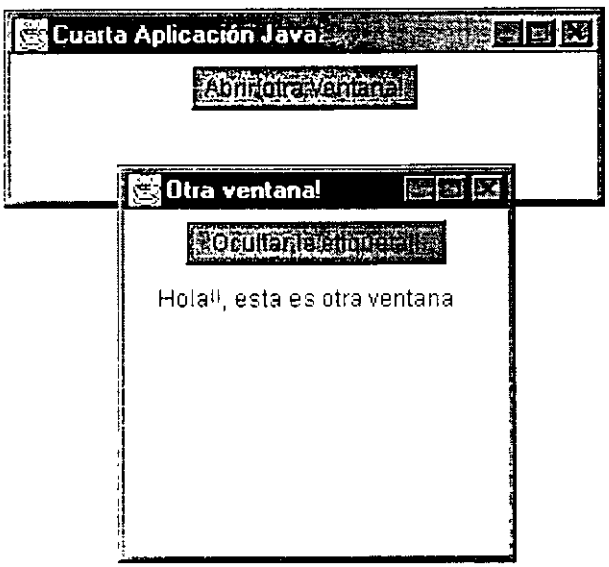
Como último ejemplo tenemos una aplicación que abre otra ventana también creada y configurada por nosotros, estas son dos clases que deben ser colocadas en archivos independientes, la principal es "aplicacionVentanaTres" y la secundaria sera nombrada "ventanitaTres ". Aquí se ve claramente que solo se requiere un solo método main aunque las clases sean muchas para poder iniciar nuestro programa.

```
import java awt *,
import java awt event *,

public class aplicacionVentanaTres extends Frame implements ActionListener {

  Button boton;
  ventanitaTres mini,

   public static void main( String args[] ) {
     aplicacionVentanaTres ventanaTres = new aplicacionVentanaTres( "Cuarta Aplicación Java" ),
   }

   public aplicacionVentanaTres(String titulo) {
     super( titulo ),

     boton = new Button("Abrir otra Ventana!"),
     add(boton),

     setLayout( new FlowLayout() );
```

```
    setBounds(50, 50, 300, 100);
   setVisible(true);
   boton.addActionListener(this);
   addWindowListener(new cerrarVentana() );
   }

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

  public void actionPerformed(ActionEvent e) {
  if(e.getSource() == boton) {
        if(mini != null) {
          mini.setVisible(false);
          mini.dispose();
          mini = null;
        } else {
          mini = new ventanitaTres("Otra ventana!");
        }
    }
  }


  public void paint( Graphics g ) {
    //Lo que gustes dibujar
  }

}

import java.awt.*;
import java.awt.event.*;

public class ventanitaTres extends Frame implements ActionListener {

  private Button b;
  private Label etiqueta;

   public ventanitaTres(String titulo) {
     super( titulo );
     b = new Button("  Ocultar la etiqueta!!  ");
     etiqueta = new Label( "Hola!!, esta es otra ventana" );

     add(b);
     add(etiqueta);

     setLayout( new FlowLayout() );

     setBounds(100, 140, 200, 200);
     setVisible(true);
     b.addActionListener(this);
     addWindowListener( new cerrarVentanaExtra() );
   }

  public void actionPerformed(ActionEvent e) {
    if(e.getSource() == b) {
        etiqueta.setVisible(false);
```

```
        }
    }

    public class cerrarVentanaExtra extends WindowAdapter {
     public void windowClosing(WindowEvent e){
     setVisible(false);
     dispose();
     }
     }

     public void paint(Graphics g) {
       //Lo que gustes dibujar en esta otra ventanita
     }

}
```

La salida de la Aplicación es:



La primera clase "aplicacionVentanaTres" es un programa que muestra una ventana y un boton, la segunda clase unicamente configura una ventana que es utilizada por el primero llamada "ventanaTres".

En la primera clase tenemos

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

public class aplicacionVentanaTres extends Frame implements ActionListener {

  Button boton;
  ventanitaTres mini;

   public static void main( String args[] ) {
     aplicacionVentanaTres ventanaTres = new aplicacionVentanaTres( "Cuarta Aplicacion Java" );
   }
```

Este fragmento de código prácticamente resume todo lo que hara nuestro programa, mostrara un botón y mediante ese botón abrira una ventana nombrada como mini, más abajo se define la clase "ventanitaTres" que es la que utilizaremos para este fin. También se declara una ventanaTres que es precisamente el tipo de nuestra clase, vamos a aclarar nuevamente: Se crearán dos ventanas, una se hará en base a la clase

"aplicacionVentanaTres" y la otra se hará en base a la clase "ventanitaTres" definida más abajo. Después tenemos el constructor de nuestro programa principal (o ventana principal) que agrega el WindowListener a la ventana para poder cerrarla apropiadamente y también agrega otro Listener al botón para atender su acción, como implementamos ActionListener se pasa un this como argumento, si no has comprendido bien por qué se hace esto regresa a la sección de Interfaz Gráfica y Eventos:

```
public aplicacionVentanaTres(String titulo) {
    super( titulo );

    boton = new Button("Abrir otra Ventana!");
    add(boton),

    setLayout( new FlowLayout() ),

    setBounds(50, 50, 300, 100);
    setVisible(true),
    boton addActionListener(this);
    addWindowListener(new cerrarVentana() ),
}
```

Es convencional, muy similar al que utilizamos en los otros ejemplos, lo mismo sucede con la innerclass que controla la salida de nuestro programa:

```
  public class cerrarVentana extends WindowAdapter {
    public void windowClosing(WindowEvent e){
     dispose();
     System exit(0),
   }
  }
```

La siguiente función "actionPerformed" es la que exige la interfaz ActionListener y se encarga de atrapar los eventos que se darán única y exclusivamente en nuestra primera ventana (ventana principal) específicamente en el botón, la otra ventana que se abrirá con dicho boton deberá implementar por si misma y a parte lo que hará con sus propios eventos, así que recuerden cada ventana maneja sus propios eventos dentro de su clase.

```
public void actionPerformed(ActionEvent e) {
  if(e getSource() == boton) {
        if(mini != null) {
          mini.setVisible(false),
          mini dispose(),
          mini = null,
        } else {
          mini = new ventanitaTres("Otra ventana!"),
        }
    }
  }
```

La función anterior revisa si el evento actúa sobre el botón, en dado caso ejecuta lo que se encuentra dentro del if, es decir otro if-else, la primera condición de este revisa mediante una comparación "!=" si la ventana "mini" realmente existe o no, recuerden que si un objeto es igual a un **null** significa que no existe o bien que esta destruido si el objeto "mini" es != (desigual) a un null significa que ya esta creado, bien entonces si la ventana existe se esconde y después se destruye y si no existe, se crea una nueva

```
        if(mini != null) {
          mini.setVisible(false);
          mini.dispose();
          mini = null;
        } else {
          mini = new ventanitaTres("Otra ventana!");
        }
```

Si no colocáramos este fragmento de código anterior cada vez que oprimiéramos el botón se crearía una ventana nueva, esto no es muy bueno para la memoria de nuestras computadoras por tanto antes de crear una nueva se

destruye la anterior si es que existe. Ahora veamos como se implementa a si misma la dichosa "ventanitaTres" en la otra clase:

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

public class ventanitaTres extends Frame implements ActionListener {

  private Button b,
  private Label etiqueta;

   public ventanitaTres(String titulo) {
     super( titulo ),
     b = new Button("  Ocultar la etiqueta!!  "),
     etiqueta = new Label( "Hola!!, esta es otra ventana" );

     add(b);
     add(etiqueta);

     setLayout( new FlowLayout() ),

     setBounds(100, 140, 200, 200),
     setVisible(true),
     b.addActionListener(this),
     addWindowListener( new cerrarVentanaExtra() ),
   }
```

Como ven aqui tambien tenemos un botón, una etiqueta y un respectivo constructor que inicializa nuestras variables, de la misma manera se implementa otro ActionListener que definirá la accion del boton (por eso le agregamos this como Actionlistener de b), sin embargo ya no hay ninguna funcion main puesto que nuestra clase anterior se encarga de iniciar el programa y crear una ventana mientras que esta clase solo se tiene que encargar de configurar y mostrar una ventana nueva, el constructor que se llamó en la función "actionPerformed" de la clase anterior es este mismo que esta aqui arriba. Ahora, esta ventana también tiene su propia innerclass que maneja la acción que la cierra mediante

```
public class cerrarVentanaExtra extends WindowAdapter {
    public void windowClosing(WindowEvent e){
   setVisible(false),
   dispose(),
    }
   }
```

Como ven aqui no es necesario incluir un System exit(0), puesto que al cerrar esta ventanita solo deseamos que se esconda y se destruya la misma, no toda la aplicacion!! Como ya lo platiqué esta ventanita también debe encargarse de sus propios eventos, esto se hace mediante su propia funcion "actionPerformed" que exige su interfaz ActionListener.

```
 public void actionPerformed(ActionEvent e) {
    if(e getSource() == b) {
        etiqueta setVisible(false),
     }
 }
```

Esta funcion que atrapa eventos realmente no hace mucho, solo esconde la etiqueta si se presiona el botón nombrado "b" Como nota final observen que cada clase también posee su propia función paint que le permitiria dibujar cualquier figura a cada una en su respectivo espacio Ya pueden crear su primera aplicación sencilla en Java

Otra forma común de implementar un WindowListener a un Frame

Esta es otra forma algo común de agregar un WindowListener a un Frame, esto evitaria por ejemplo tener que declarar la innerclass completa, realmente es lo mismo pero algunas personas lo prefieren (sobre todo profesionales):

```
addWindowListener (new WindowAdapter() {
    public void windowClosing(WindowEvent e){
        dispose();
        System.exit(0);
    }
}),
```

Este addWindowListener() iria dentro del constructor de Frame, en su interior está la creación y declaración (ambas al mismo tiempo) de un WindowAdapter mediante un constructor sin argumentos que define el método windowClosing() para cerrar la ventana. Yo creo que es más comprensible la versión que usamos en los ejemplos pero puedes usar cualquiera de las dos (aunque solo debe ser una).

Archivos de Acceso Aleatorio

- Entendiendo los Archivos de Acceso Aleatorio
- La clase File
- La clase RandomAccessFile
- Ejemplo en Pseudocódigo
- Ejemplo - Un programa simple de administración de cuentas para un Gimnasio
- La clase "cuentaRegistro"
- La clase "alta"
- La clase "baja"
- La clase "consulta"
- La clase "modificar"
- La clase "archivosAleatorios" (Programa Principal)

Nota importante Esta es la forma 'complicada' para almacenar información de nuestros programas en un archivo Existe un método mucho más sencillo e incluso más fácil de aprender y comprender. **La Serialización de Objetos**. Si deseas almacenar datos de manera mucho más simple mejor revisa antes el capitulo de serialización.

Los programas creados hasta ahora han sido interesantes, algunos han sido útiles y otros ilustrativos e incluso algunos sin utilidad especifica alguna. Las applets son divertidas en la mayoria de los casos, los programas independientes tienden a ser algo mucho más serio en cuanto a su fin A veces es necesario que los programas tengan la capacidad guardar información a largo plazo, las variables son muy útiles para almacenar valores temporales pero bajo ciertas circunstancias es mejor guardar esa información y recuperarla incluso si se cierra el programa. Es claro que los ejemplos sobran: programas administrativos, bases de datos, bibliotecas de consulta, chequeo de saldos etc. Realmente somos muy afortunados como principiantes porque aunque este tipo de aplicaciones suelen ser de un enfoque más práctico, "serio" y "profesional" son posibles para nosotros, en Java para Principiantes todo es fácil!!!

Existen dos formas generales para leer y escribir archivos. secuencialmente y aleatoriamente Los archivos de acceso secuencial escriben y recuperan la información de manera lineal, o sea que este tipo de archivos reciben los datos suponiendo que tu los estas ordenando de manera previa. En tal caso si deseas guardar información sobre clientes cada uno con su respectivo numero de cuenta entonces debes introducir la informacion en orden para que se almacene también en orden. Los archivos de acceso aleatorio te permiten almacenar o recuperar informacion en orden aleatorio, es decir que si deseas guardar cierta información en la cuenta 53 entonces puedes desplazarte libremente a dicha cuenta y escribir sobre ella. no se requiere que ordenes los datos, únicamente te ubicas en la posicion que deseas y lees o escribes información a tu gusto. Java está tan bien diseñado que es relativamente simple trabajar con archivos ya sean de acceso secuencial o aleatorio, por lo tanto a final de cuentas depende de ti elegir una opcion entre ambas En mi muy particular y humilde opinión se me facilitan más los archivos de acceso aleatorio por lo tanto seran los únicos que mostrare aqui

Entendiendo los Archivos de Acceso Aleatorio

Si eres estudiante entonces será muy facil que comprendas los archivos de acceso aleatorio. Seguramente alguna vez has utilizado un cuaderno cuadriculado. la cuadricula tiene cierta cantidad de cuadros horizontalmente y cierta cantidad de cuadros verticalmente, a mi me encantan los cuadernos de cuadro chico porque permiten escribir más información (y porque mi letra es pequeña). Un archivo aleatorio es prácticamente igual a una de estas hojas cuadriculadas solo que en vez de almacenar apuntes almacena bytes. Los tipos de datos básicos de Java (tales como: char, double, int, long, boolean) y en general toda la informacion que pasa por una computadora se puede expresar en terminos de bytes, ya sea una imagen, un sonido, un video, un documento, etc etc. todo puede convertirse a bytes! incluso nuestro código genético puede expresarse en bytes (como lo demuestra el Proyecto Genoma Humano) por lo tanto si deseamos escribir información primero hay que convertirla a bytes, en otros lenguajes de programación dicha conversion se hace mano. en lenguajes como C y C++ la conversión es casi automática y es mas facil, en Java es igual o aún mas fácil

Regresando a la explicación, antes de escribir en una hoja (en un archivo) ésta debe ser cuadriculada. En escritura a mano queda libre la elección de escribir un número o un texto, Java necesita que seamos un poquito

más claros sobre esto porque exige que antes de escribir indiquemos claramente el tipo de dato que deseamos almacenar, esta elección es sobre los tipos de datos básicos (es decir, debes elegirlos entre: char, double, int, long, boolean) Una vez realizada la elección podemos proceder a "cuadricular" nuestro archivo y así marcar cuales cuadros serán para este tipo y cuales cuadros serán para este otro tipo y esto es todo, una vez marcados los cuadros ahora podemos escribir y leer libremente, en Java existe algo llamado puntero que es algo así como el lápiz virtual que nos permite escribir la información, éste lapicito se puede desplazar a cualquier posición de nuestro archivo (de nuestra hoja cuadriculada) y escribirá el dato, ahh! pero ese dato que intentes escribir debe coincidir con el dato con el cual esta marcada la cuadricula en dicha posición, por ejemplo si intentas escribir un int en una cuadricula que esta marcada como double entonces se producirá una excepción que obviamente no produce ningún registro. A grandes rasgos eso es todo, es muy fácil ¿verdad?, ahora vamos a entrar en detalles explicando todo nuevamente con más cuidado.

Ejemplo en Pseudocódigo

Vamos a ver el siguiente ejemplo en pseudocódigo.

Problema: Realizar un programa que permita almacenar los siguientes datos de un cliente: Número, Nombre, Sexo y Saldo.

Pasos de la solución:

1. Cuadricular una hoja con 4 secciones: la primera almacenará un int (entero) para el número, la segunda un String (Cadena de caracteres) para el nombre, la tercera un boolean (true si es mujer, false si es hombre) para el sexo y la cuarta un double (para el saldo que puede ser un número decimal). Lo anterior significa que cada renglon de nuestra hojita será de cuatro cuadros y digamos que ponemos 100 renglones para almacenar los datos de 100 futuros clientes.
2. Recibir la información, supongamos que a cada cliente se le da la opción de escoger su número de cuenta así que si nuestro primer cliente nos da los siguientes datos:

Número: 73
Nombre Oscar Chávez
Sexo: Masculino
Saldo 450 45

Entonces movemos nuestro lapiz (puntero) hasta la línea 73, en dicha línea hay 4 renglones (los de los tipos de datos) y una vez allí los escribimos Una vez terminada la operación de escritura en programacion formal seria buena idea cerrar el archivo
3. Obtener la información almacenada, supongamos que ya tenemos muchos clientes y ahora deseamos realizar una consulta digamos del cliente 38 Para esto primero movemos nuestro lapicito a la línea 38 y una vez allí leemos los datos escritos:

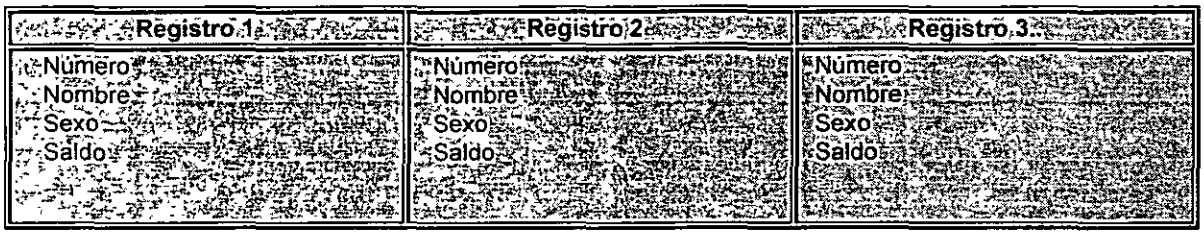
Numero. 38
Nombre. Agustin Lara
Sexo. Masculino
Saldo 984.26

Nuevamente una vez terminada esta operacion debemos cerrar el archivo. Finalmente la modificación y eliminación de datos es similar, para modificar nos colocamos en la línea correspondiente al número de nuestro cliente y sobrescribimos los datos Para eliminar nos moveríamos a la línea con el lapicito y en vez de borrar (cosa que seria lo convencional con una hoja y lapiz convencionales) esta vez tenemos que escribir datos vacios; por ejemplo

Numero 0
Nombre " "
Sexo false
Saldo 0 0

La cadena " " en Java es una cadena vacia y si mal no recuerdo un false corresponde a un bolean recién creado, si se dan cuenta realmente no tendría mucho caso borrar el número porque de cualquier manera éste será el mismo si algún cliente lo elige (siempre se encontrara en la misma línea), aún así lo borré por consistencia. Pues bien, ese fue el ejemplo Ahora veamos una clase que nos servirá para "cuadricular" nuestro archivo.

Cuando se escriben los datos sean del tipo que sean en un archivo aleatorio estos deben ser convertidos a su contraparte en bytes, lo que implica que si tu abres algún archivo aleatorio con el Bloc de Notas por decir algo, no verás los datos como los guardaste sino que los verás unos símbolos medio raritos. Cuando creamos una cuadrícula en realidad estamos marcando el archivo con bytes que representan valores vacíos (igual que al eliminar registro), este marcado se realiza de la siguiente manera (en base al ejemplo en pseudocódigo):



Y así sucesivamente hasta llegar al registro 100 Aunque por motivos didácticos lo idealizamos de la siguiente manera.

Y así sucesivamente. Recuerden que pretendemos visualizar un archivo de acceso aleatorio como un grupo de registros, cada registro tiene 4 cuadros: un cuadro entero, uno de cadena, uno bolean y uno double Por eso es mejor idealizarlo como la segunda tabla aunque el puntero (lapicito) lo percibe como la primera

La clase File

La clase File a diferencia de lo que indica su nombre (File significa Archivo) no tiene como fin servir para leer o escribir archivos, se trata únicamente de una herramienta para obtener información acerca de los archivos ¿Que tipo de informacion? si es de solo lectura, si se puede modificar, su tamaño, su nombre, su carpeta, la fecha de su última modificacion, si es que existe, entre otros datos También permite crear o borrar archivos, pero como ya dije, no sirve para escribir o leer

Se supone que una aplicacion que pretende escribir en un archivo debe crear tal archivo, si este archivo ya existe entonces no sera necesario crearlo, el ejemplo de Archivos Aleatorios mostrado en esta sección recurre a la clase File para confirmar si el archivo que vamos a crear ya existe, si aún no existe entonces se crea uno (marcando su cuadricula de datos respectiva) y si ya existe pues no se hace nada para dejarlo tal como esta, porque obviamente deseamos conservar los datos almacenados! lo que sucede es que los constructores de RandomAccessFile aunque pueden crear o abrir un archivo existente no nos permiten marcar la cuadricula de datos que deseamos. Si desean más información acerca de la informacion que pueden obtener mediante la clase File les aconsejo revisar detenidamente la Java API (la clase File, obviamente)

La clase RandomAccessFile

La clase RandomAccessFile nos da las capacidades que buscamos de lectura y escritura de archivos. El constructor que utilizaremos para este ejemplo es:

public RandomAccessFile(String name, String mode)

donde "name" representa el nombre del archivo y "mode" indica el modo en el cual abriremos el archivo, ambos son cadenas. El nombre es libre de elegir si se trata de un archivo nuevo o debe ser consultado en el caso de un archivo existente (el caso del ejemplo), el nombre se coloca con su extensión correspondiente. En cuanto al modo existen dos posibilidades: "rw" o "r", rw indica lectura y escritura (read & write) mientras que r indica solo lectura (read). Este constructor puede lanzar diversas excepciones que en general se pueden atrapar como un NullPointerException para exhibirse en pantalla o para realizar alguna otra manipulación cosa que jamás he hecho puesto que yo también sigo siendo principiante!

El siguiente listado muestra algunos métodos de la clase RandomAccessFile:

void close()
Cierra el archivo de acceso aleatorio y libera todos los recursos del sistema asociados con este archivo
long getFilePointer()
Regresa la posición actual del apuntador (lapicito)
long length()
Regresa la longitud del archivo
boolean readBoolean()
Lee un dato bolean de este archivo
char readChar()
Lee un caracter unicode de este archivo (una letra)
double readDouble()
Lee un double de ese archivo
int readInt()
Lee un entero de este archivo
String readUTF()
Lee una cadena de este archivo
void seek(long pos)
Fija el apuntador en la posición "pos" partiendo del principio del archivo para colocarlo en donde se piensa leer o escribir
void writeBoolean(boolean v)
Escribe un valor bolean
void writeChar(int v)
Escribe un caracter
void writeChars(String s)
Escribe una serie de caracteres
void writeDouble(double v)
Escribe un double
void writeInt(int v)
Escribe un int
void writeUTF(String str)
Escribe una cadena

Obviamente hay más metodos, consulten la Java API para mayor información. Aunque escribiremos varias cadenas (String) no vamos a utilizar ni writeUTF() o su contraparte readUTF(), esto es porque aún debo estudiar mejor estos metodos porque al parecer escriben cadenas de tamaño arbitrario lo que haria errático el tamaño de nuestra cuadricula de datos, la verdad aún no estoy segura del todo por eso decidi no arriesgarme, ya veremos mas adelante con una explicacion mucho más detallada la técnica que aprendi de un libro para almacenar cadenas de una manera más clara y predecible aunque si sabes bien el manejo de cadenas para archivos de acceso aleatorio y observas que mi código puede ser más simple acepto tus sugerencias

Ejemplo - Un programa simple de administración de cuentas para un gimnasio

El siguiente ejemplo ilustra una posible forma de crear un programa sencillo (aunque ya puede ser muy útil) de un administrador de cuentas para un gimnasio, el problema es el siguiente.

"Crear un programa que administre cuentas de clientes, cada registro debe contener: Número, Nombre y Peso de la persona. Se deben poder hacer altas, bajas, consultas y modificaciones a las cuentas."

El programa se organiza de la siguiente manera:

Organizacion del Programa

Clase	Descripción
clase cuentaRegistro	Crea y manipula el archivo de acceso aleatorio contiene el orden preciso en que deben escribirse y leerse los datos, no tiene interfaz gráfica
clase alta	Recurre a la clase cuentaRegistro para dar de alta una cuenta, hereda de un Frame con una interfaz gráfica para obtener los datos desde campos de texto
clase baja	Recurre a la clase cuentaRegistro para dar de baja una cuenta, hereda de un Frame con una interfaz gráfica que obtiene el número de cuenta a eliminar
clase consulta	Recurre a la clase cuentaRegistro para leer una cuenta, hereda de un Frame con una interfaz gráfica para obtener el número de cuenta a consultar
clase modificar	Recurre a la clase cuentaRegistro para modificar una cuenta, hereda de un Frame con una interfaz gráfica para obtener los datos desde campos de texto
clase archivosAleatorios	Programa principal, aquí se encuentra el método main que cargará este Frame principal, se recurre a la clase File para revisar si existe un archivo de datos, si no existe se crea uno (y su cuadrícula de datos respectiva) mediante la clase cuentaRegistro

Como ven de hecho todas las clases excepto "cuentaRegistro" heredan de frame, la clase archivosAleatorios que es el Frame principal tendrá 4 botones para crear ya sea un "alta", "baja", "consulta" o "modificar", cada una de estas subventanas tendrá sus botones, labels y campos de texto respectivos que se ordenan mediante un Layout (acomodamiento) de tipo GridLayout con 4 lineas y 2 columnas. Revisemos cada clase en detalle para comprender mejor todo

La clase "cuentaRegistro"

Esta clase es el corazón para manejar el archivo, contiene funciones para escribir o leer en él, además también permite manipular el apuntador (el lapiz) y cerrar el archivo entre otras cosas, veamos el codigo y después voy a explicar función por función

```
import java.io.*;

public class cuentaRegistro {

 public final int size = 52;
 RandomAccessFile archivo;

 public int numero;
 public String nombre;
 public double peso;

 /*
 Constructor de la clase
 de hecho únicamente carga un archivo
 */
 public cuentaRegistro(String nombre, String modo) {
  abrirArchivo(nombre, modo);
 }

 /*
 Funcion que se llama desde el constructor,
 aqui se llama al constructor de RandomAccessFile
 y se maneja la excepción que podria lanzar
 */
 public void abrirArchivo(String nombre, String modo) {
```

```
 try {
  archivo = new RandomAccessFile(nombre, modo);
 } catch (IOException e) {
  System.err println("Error al abrir el archivo " + e toString());
  System exit(0);
 }
}

/*
Función que se llama para cerrar el archivo,
también se maneja la excepción que podría lanzar
*/
public void cerrarArchivo() {
 try {
  archivo close();
 } catch (IOException e) {
  System.err.println( "Error al cerrar el archivo " + e.toString() ),
  System.exit(0);
 }
}

/*
Función que lee información desde el archivo,
obtiene los datos de las funciones respectivas
a cada tipo de dato, también maneja la excepción
si es que es lanzada
*/
public void leer() {
 try {
  numero = leerEntero(),
  nombre = leerCadena(20);
  peso = leerDouble(),
 } catch (IOException e) {
  System.err.println("Error al leer en el archivo: " + e.toString()),
  System.exit(0);
 }
}

/*
Función que escribe directamente a un archivo
llamando a las funciones respectivas para cada
tipo de dato, también maneja la excepcion si es
que ésta es lanzada
*/
public void escribir() {
 try {
  escribirEntero(numero);
  escribirCadena(nombre, 20),
  escribirDouble(peso);
 } catch (IOException e) {
  System err.println("Error al escribir en el archivo  " + e toString());
  System.exit(0),
 }
}

/*
Funcion privada que escribe un entero en
el archivo prácticamente ésta es únicamente una
traduccion al español porque realmente no
cambia la original de ninguna manera
*/
private void escribirEntero(int entero) throws IOException {
```

```
 archivo.writeInt(entero);
}

/*
Función privada que lee un entero
*/
private int leerEntero() throws IOException {
 return archivo readInt();
}

/*
Función privada que escribe un double
*/
private void escribirDouble(double precision) throws IOException {
 archivo writeDouble(precision);
}

/*
Función privada que lee un double
*/
private double leerDouble() throws IOException {
 return archivo readDouble();
}

/*
Función privada que escribe una cadena
*/
private void escribirCadena(String s, int numLetras) throws IOException {
 if(s == null)
  s = "                    ";

          if (s length() <= numLetras) {
  archivo.writeChars(s);
  for(int i = s length(); i < numLetras; ++i)
   archivo.writeChar(' ');
 } else {
  archivo.writeChars(s substring( 0, numLetras));
 }
}

/*
Función privada que lee una cadena
*/
private String leerCadena(int numLetras) throws IOException {
 String s = "";
 for(int i = 0; i < numLetras; ++i)
  s += archivo readChar();
 return s trim();
}

/*
Función que mueve el puntero, esta funcion
desplaza el puntero a la posición n (en bytes)
se maneja la excepcion si es que es lanzada
*/
public void seek(long n) {
 try {
  archivo seek(n);
 } catch (IOException e) {
  System.out println("Error al mover el puntero de escritura a n bytes " + e toString());
 }
}
```

}

Explicación de la clase cuentasRegistro

Primero importamos el paquete java.io ya que la clase RandomAccessFile pertenece a dicho paquete:

import java.io.*;

Después tenemos las siguientes variables globales:

```
public final int size = 52;
RandomAccessFile archivo;

public int numero,
public String nombre;
public double peso;
```

El RandomAccessFile nombrado "archivo" es el encargado de abrir y cargar el archivo como veremos en el constructor un poco más adelante. Las variables públicas: numero, nombre y peso servirán para almacenar los valores tomados desde los campos de texto de manera exterior ya que ésta clase no posee interfaz gráfica, alguna otra clase deberá pasar los datos introducidos por el usuario a estas variables. En cuanto a la variable size declarada como public final es un valor constante que sirve para especificar el tamaño de un registro (de una cuenta) en bytes Recuerden que todos los datos obtenidos deben convertirse a bytes, para determinar este tamaño necesitamos dos cosas· Los datos que almacenara cada cuenta y la siguiente tabla.

Tamaño de datos en bits y bytes

Tipo de dato	Tamaño en bits	Tamaño en bytes
boolean	1	1
char	16	2
int	32	4
long	64	8
float	32	4
double	64	8

Nota: Ocho bits corresponden a un solo y único byte, (8bits = 1byte).

Ahora vamos a aclarar un poco mas las cosas, según el problema tenemos que almacenar tres datos por cada registro

Un número (entero)
Un nombre (cadena)
Un peso (numero double)

Entonces, guardaremos un entero (int), una cadena (String) y un double Quede en que no iba a utilizar los metodos para guardar Strings por lo tanto voy a recurrir a la idea de que una cadena es un arreglo de caracteres, según lo anterior tenemos definido el tamaño para el numero de cuenta que seria **4 bytes**, tambien tenemos definido el tamaño para el peso (es un double) **8 bytes**, solo falta definir el tamaño de la cadena, supongan que vamos a guardar unas 20 letras que incluirán al nombre junto con su apellido o sea que la cadena contendría por ejemplo "Roxana Garza " que son 20 caracteres, es importante notar que un espacio vacio también es un caracter, es aceptable por lo tanto dejaremos 20 caracteres como tamaño de la cadena es decir 20x2 = **40 bytes**. Si hacemos la suma de todos los datos tenemos el resultado para el tamaño de un registro (una cuenta) **52 bytes totales** Pero ¿por qué debemos obtener este valor? porque nuestro puntero o lapicito se mueve en bytes, es decir nosotros le debemos indicar exactamente el número de bytes que debe moverse a partir del inicio del archivo antes de escribir Más adelante veremos que nuestro lapicito solo se moverá en múltiplos de **52 bytes** tanto para leer como para escribir El constructor es.

```
/*                                                    /
Constructor de la clase
de hecho unicamente carga un archivo
```

```
*/
public cuentaRegistro(String nombre, String modo) {
 abrirArchivo(nombre, modo);
}

/*
Función que se llama desde el constructor,
aqui se llama al constructor de RandomAccessFile
y se maneja la excepción que podria lanzar
*/
public void abrirArchivo(String nombre, String modo) {
 try {
  archivo = new RandomAccessFile(nombre, modo);
 } catch (IOException e) {
  System.err.println("Error al abrir el archivo " + e.toString());
  System.exit(0);
 }
}
```

La segunda función abrirArcivo() abre el documento con los datos que pasa el constructor, ambos: el constructor y la función abrirArchivo() están relacionadas pero bien pude haber puesto todo el contenido de la función en el constructor. También se atrapa una excepción si es que ésta se genera que indica un mensaje de error en la linea de comandos mediante System.err.println() Después tenemos la función encargada de cerrar el archivo:

```
/*
Funcion que se llama para cerrar el archivo,
también se maneja la excepción que podría lanzar
*/
public void cerrarArchivo() {
 try {
  archivo.close();
 } catch (IOException e) {
  System.err.println( "Error al cerrar el archivo " + e.toString() );
  System.exit(0);
 }
}
```

Existen dos formas de saber que una función tal como archivo.close() lanza una excepcion, la primera y más fácil es consultando la Java API, si el método incluye la clausula throws significa que puede lanzar una excepción del tipo marcado, la segunda es que el compilador te lo indique porque no te permitirá compilar hasta que atrapes la excepción que se puede generar, el mensaje general seria algo asi como: Exception java.io.IOException must be caught, or it must be declared in the throws clause of this method (La excepcion java.io.IOException debe ser atrapada o debe ser relanzada desde éste método con throws). En el caso anterior se atrapa en vez de relanzarla aunque no hago gran cosa por manejarla más que exhibir en pantalla que sucedió al cerrar el archivo. La siguiente función es la encargada de escribir en el archivo:

```
/*
Función que lee información desde el archivo,
obtiene los datos de las funciones respectivas
a cada tipo de dato. también maneja la excepcion
si es que es lanzada
*/
public void leer() {
 try {
  numero = leerEntero();
  nombre = leerCadena(20);
  peso = leerDouble();
 } catch (IOException e) {
  System.err.println("Error al leer en el archivo: " + e.toString());
  System.exit(0);
 }
}
```

Esta función llama a otras que veremos un poco más abajo, todas las funciones de escritura y lectura pueden lanzar excepciones de tipo IOException (excepción de entrada y salida, aludiendo a la entrada o salida de datos desde cualquier fuente), por lo tanto estas deben ser atrapadas en algún momento, aquí doy una solución drástica a este problema: salir del programa mediante System.exit(0); lo cual indica que algo muy malo sucedió, puede ser que el archivo haya desaparecido de alguna forma (no me imagino como, pero puede suceder). Algo muy importante respecto al método anterior y al siguiente es que ambos deben realizar ya sea la escritura o la lectura **bajo la misma secuencia**, si los datos se escriben en cierto orden los datos deben leerse en ese mismo orden para recuperarlos apropiadamente porque sería imposible leer un entero en un cuadro marcado como double por ejemplo, se produciría un error!. Estos métodos solo escriben o leen el registro de una cuenta, desplazar el puntero para leer o escribir otras cuentas.. es otra historia.

```
/*
Funcion que escribe directamente a un archivo
llamando a las funciones respectivas para cada
tipo de dato, también maneja la excepción si es
que ésta es lanzada
*/
public void escribir() {
 try {
  escribirEntero(numero);
  escribirCadena(nombre, 20);
  escribirDouble(peso),
 } catch (IOException e) {
  System err println("Error al escribir en el archivo. " + e.toString());
  System exit(0),
 }
}
```

La función anterior de escritura también recurre a funciones que veremos mas abajo, el manejo de la excepcion es similar al método de lectura, también toma la medida un poco drástica de salir del programa si se lanza una excepción, la IOException podría lanzarse si el disco se llena o si el archivo es bloqueado de alguna manera. La siguiente función se encarga de escribir un entero:

```
/*
Funcion privada que escribe un entero en
el archivo prácticamente ésta es únicamente una
traduccion al español porque realmente no
cambia la original de ninguna manera
*/
private void escribirEntero(int entero) throws IOException {
 archivo writeInt(entero),
}
```

Creo que su descripcion es clara, en vez de manejar la excepcion que puede lanzar el método writeInt() se ocupa mejor de relanzarla a cualquier metodo que la llame de modo que él la maneje a su modo, en otras palabras el throwsIOException es como si este metodo advirtiera a cualquier otra método que intente usarlo: bien, me puedes usar pero yo soy capaz de lanzar una excepcion de tipo IOException, te advierto para que tú la manejes o bien para que la relances, tu decides Los metodos leer() y escribir() se encargan de manejar esa excepción Las siguientes funciones son muy similares a la anterior

```
/*
Funcion privada que lee un entero
*/
private int leerEntero() throws IOException {
 return archivo readInt(),
}

/*
Funcion privada que escribe un double
*/
private void escribirDouble(double precision) throws IOException {
 archivo writeDouble(precision),
}
```

```
/*
Función privada que lee un double
*/
private double leerDouble() throws IOException {
 return archivo.readDouble();
}
```

Lo anterior es simple, aquí comienza lo interesante. ¿Recuerdan que no iba a utilizar ninguno de los métodos de RandomAccessFile para leer y escribir cadenas? pues eso significa que debo crear algo similar que lea y escriba las dichosas cadenas, como una cadena es un arreglo de caracteres lo que hago para escribir la cadena es tomar los caracteres que contiene y después utilizar la función writeChars() para escribirlos de manera individual·

```
/*
Función privada que escribe una cadena
*/
private void escribirCadena(String s, int numLetras) throws IOException {
 if(s == null)
  s = "                    ";

            if (s.length() <= numLetras) {
  archivo.writeChars(s),
  for(int i = s length(); i < numLetras; ++i)
   archivo.writeChar(' ');
 } else {
  archivo.writeChars(s.substring( 0, numLetras));
 }
}
```

El primer "if" se encarga de checar que la cadena no esté vacia y en caso de que lo esté la llena con 20 caracteres vacios (tal como habiamos acordado este es el tamaño de una cadena por almacenar), el segundo if revisa que el tamaño de la cadena no exceda a la variable entera "numLetras", esta variable sirve para fijar precisamente el limite que acordamos de 20 caracteres, siempre se pasará tanto en este método como el que lee una cadena (el siguiente) Si esta condicion se cumple entonces se escriben todos los caracteres que contenga la cadena mediante writeChars(), aunque también si esta condición es verdadera tambien implica que la cadena posiblemente es incluso menor que el requisito de 20 caracteres entonces se ejecuta un ciclo for que llena los espacios restantes (que no ocupa la cadena) con espacios vacios o caracteres vacios. Finalmente el "else" implica que la cadena no cumplio la primera condicion, es decir que la cadena contiene más de 20 caracteres, lo que se hace aquí es inevitablemente recortar la cadena y solo incluir los 20 caracteres acordados, la funcion substring() de la clase String recorta la cadena, su primer argumento es el índice en el cual se inicia y el último indica el final del recorte dentro del arreglo, en otras palabras este método recorta todo lo que esté fuera del indice 0 de la cadena hasta el indice 20. La siguiente funcion se encarga de leer una cadena escrita

```
/*
Función privada que lee una cadena
*/
private String leerCadena(int numLetras) throws IOException {
 String s = ""
 for(int i = 0; i < numLetras, ++i)
  s += archivo.readChar()·
 return s trim(),
}
```

Se supone que en este ejemplo siempre pasaremos un argumento de 20 como "numLetras", primero creamos una cadena nombrada "s" para almacenar todos los caracteres (del 1 al 20), después le asignamos un valor vacio· "", el ciclo for se encarga de recorrer 40 bytes, convertirlos en caracteres e ir anexando cada lectura a la cadena. Finalmente se regresa la cadena leida mediante return s.trim(), la funcion trim() se encarga de "acortar" la cadena eliminando los caracteres en blanco (los espacios propiamente dichos) que pudiera tener al final, los intermedios no se eliminan Como esta es una lectura no importa si se eliminan los espacios en blanco del final con tal de obtener los caracteres que nos importan, en la escritura los espacios son más importantes como ya vimos Finalmente tenemos otra función muy importante, la que desplaza nuestro puntero o lápiz:

```
/*
Funcion que mueve el puntero, esta función
```

```
desplaza el puntero a la posición n (en bytes)
se maneja la excepción si es que es lanzada
*/
public void seek(long n) {
 try {
  archivo.seek(n);
 } catch (IOException e) {
  System.out.println("Error al mover el puntero de escritura a n bytes " + e.toString());
 }
}
```

El argumento "n" recibe la cantidad de bytes que nos desplazaremos desde el principio del archivo. Esta función solo se encarga de atrapar la excepción para evitarnos tener que atraparla cada vez que la llamemos desde el exterior, es bastante útil mientras se coloque un mensaje correspondiente indicando exactamente que fue lo que sucedió si es que es lanzada!.

Todos los System.out.println() reciben una cadena como argumento, esta cadena generalmente exhibe información. Este mensaje solo se muestra en la consola del sistema tal como la ventana de DOS, asi que nunca se verá desde una interfaz gráfica.

La clase "alta"

Esta clase es mucho más sencilla que la anterior, su propósito es mostrar un Frame con diversos componentes gráficos los cuales incluyen campos de texto que obtienen los datos para dar de alta una cuenta, es decir un registro de un número, un nombre y un double A estas alturas doy por hecho que ya conocen la creación básica de una interfaz gráfica sencilla tal como la que usaremos, lo mismo para las interfaces porque utilizaremos muchos ActionListeners para manejar los eventos y WindowAdapters para cerrar los Frames. En si la única parte de código relacionada con el alta de una cuenta es el que se encuentra dentro de la función actionPerformed():

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

public class alta extends Frame implements ActionListener {

 Label Lnumero = new Label("Número:");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso:");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button guardarCuenta = new Button("Guardar!");

 cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "rw");

 public alta() {
  super("Alta de Cuenta - GymMaster!");
  setLayout( new GridLayout( 4, 2 ) );
  setBounds(200, 170, 240, 130);

  add(Lnumero);
  add(Tnumero);
  add(Lnombre);
  add(Tnombre);
  add(Lpeso);
  add(Tpeso);
  add(guardarCuenta);

  guardarCuenta.addActionListener(this);
  this.addWindowListener ( new cerrarVentana() );

  setVisible(true);
 }
```

```
public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo.cerrarArchivo();
  dispose();
 }
}

public void actionPerformed(ActionEvent e) {
 if(e.getSource() == guardarCuenta && Tnumero.getText() != "") {
  if( Integer parseInt( Tnumero.getText() ) > 0 && Integer parseInt( Tnumero.getText() ) < 101 ) {
   archivo numero = Integer.parseInt( Tnumero.getText() );
   archivo.nombre = Tnombre.getText();
   Double p = new Double(Tpeso.getText()),
   archivo.peso = p doubleValue();
   archivo.seek((long) (archivo.numero - 1)*archivo size );
   archivo.escribir();
   Tnumero.setText("");
   Tnombre.setText("");
   Tpeso.setText("");
   System out.println("Registro " + (archivo.numero - 1) + " guardado exitosamente"),
  } else {
   Tnumero setText("Escribe entre 1 y 100");
   Tnumero selectAll();
  }
 } else {
  Tnumero.setText("Escribe algo!");
  Tnumero.selectAll();
 }
   }

}
```

Explicación de la clase "alta"

Primero importamos los paquetes que vamos a utilizar, heredamos de la clase Frame porque el siguiente codigo creara una ventana con diversos componentes. implementamos la interfaz ActionListener para atrapar los eventos generados:

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

public class alta extends Frame implements ActionListener {
```

Ahora declaramos e inicializamos al mismo tiempo diversos componentes tales como etiquetas y campos de texto, el boton nos permite guardar la informacion una vez capturada:

```
 Label Lnumero = new Label("Número ");
 Label Lnombre = new Label("Nombre  "),
 Label Lpeso = new Label("Peso:");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField(),
 TextField Tpeso = new TextField(),
 Button guardarCuenta = new Button("Guardar!");
```

Como siguiente paso creamos un nuevo archivo basandonos en la clase cuentaRegistro creada anteriormente, lo llamamos cuentas.dat y lo abrimos en modo lectura-escritura para poder escribir en él

```
 cuentaRegistro archivo = new cuentaRegistro("cuentas dat", "rw"),
```

Después creamos el constructor, el super() llama al constructor de la superclase, es decir de la clase Frame y pasa una cadena como argumento que representa el titulo de la ventana Establecemos un nuevo acomodamiento de componentes mediante setLayout() que recibe como argumento un nuevo GridLayout que divide nuestra

ventana en 4 lineas y 2 columnas, fijamos la ventana en la coordenada 200, 170 y le fijamos un ancho de 240 y un alto de 130 (recuerda que estas coordenadas son en pixeles)

```
public alta() {
 super("Alta de Cuenta - GymMaster!");
 setLayout( new GridLayout( 4, 2 ) );
 setBounds(200, 170, 240, 130);

 add(Lnumero);
 add(Tnumero);
 add(Lnombre);
 add(Tnombre);
 add(Lpeso);
 add(Tpeso);
 add(guardarCuenta);

 guardarCuenta.addActionListener(this);
 this addWindowListener ( new cerrarVentana() );

 setVisible(true);
}
```

Como siguiente paso agregamos uno a uno los componentes con add() en el orden correcto. Finalmente agregamos el Listener (definido en esta misma clase, de ahi que se pase un this) que implementa nuestra clase al botón nombrado "guardarCuenta" y agregamos un WindowListener a la ventana que esta definido como una clase interna o innerclass de esta misma clase tal como sigue:

```
public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo cerrarArchivo();
  dispose();
 }
}
```

Recuerden WindowAdapter es una clase abstracta que hereda de WindowListener que sirve para ahorrarnos el tener que definir todos los métodos que esta interfaz (WindowListener) exige, así solo definimos el método que necesitamos en este momento para cerrar la ventana: windowClosing() Por último viene la parte más importante de toda la clase, el metodo actionPerformed() que define y establece las acciones a realizar cuando se presiona el boton "guardarCuenta".

```
public void actionPerformed(ActionEvent e) {
 if(e.getSource() == guardarCuenta && Tnumero.getText() != "") {
  if( Integer.parseInt( Tnumero getText() ) > 0 && Integer.parseInt( Tnumero.getText() ) < 101 ) {
   archivo numero = Integer.parseInt( Tnumero getText() );
   archivo nombre = Tnombre getText();
   Double p = new Double(Tpeso getText());
   archivo.peso = p doubleValue();
   archivo seek((long) (archivo numero - 1)*archivo size );
   archivo escribir();
   Tnumero setText("");
   Tnombre setText("");
   Tpeso.setText("");
   System.out println("Registro " + (archivo numero - 1) + " guardado exitosamente");
  } else {
   Tnumero.setText("Escribe entre 1 y 100");
   Tnumero selectAll();
  }
 } else {
  Tnumero.setText("Escribe algo!");
  Tnumero selectAll();
 }
}
```

Para comprender éste método lo vamos a ir analizando cada vez más profundamente, primero tenemos un if-else que vamos a "vaciar" para visualizarlo mejor:

```
if(e.getSource() == guardarCuenta && Tnumero.getText() != "") {

} else {
 Tnumero setText("Escribe algo!"),
 Tnumero selectAll();
}
```

El operador AND && (doble ampersand) indica que deben cumplirse las dos condiciones para que el primer if se ejecute, la primera condición verifica el origen del evento, la función getSource() regresa el nombre del objeto sobre el cual se hizo el evento y si este coincide con el nombre de nuestro botón llamado "guardarCuenta" significa que el evento se originó en él. La segunda condición verifica que el contenido del campo de texto nombrado "Tnumero" sea desigual a una cadena vacia, la función getText() obtiene el contenido del campo, es decir, regresa el String que contiene, creo que es más que obvio que necesitamos antes que nada el número de cuenta que deseamos guardar por lo tanto no se puede si quiera entrar a este if sin este dato. En el caso de que ninguna de las dos condiciones anteriores se cumpla se ejecuta el contenido del else: primero se fija al campo de texto Tnumero el texto "Escribe algo!" y la función selectAll() remarca este contenido para resaltarlo y llamar la atención del usuario para que lo modifique, siempre que este else se ejecute significa que el usuario olvidó escribir un número de cuenta. Ahora veamos el contenido interno del if-else que se ejecuta en caso de que la condición del anterior se cumpla, justamente se trata de otro bendito if-else:

```
if( Integer.parseInt( Tnumero.getText() ) > 0 && Integer.parseInt( Tnumero.getText() ) < 101 ) {

} else {
 Tnumero.setText("Escribe entre 1 y 100"),
 Tnumero.selectAll();
}
```

Para este if-else tenemos otro operador AND && que devuelve un true solo si las dos condiciones son verdaderas, en pocas palabras se revisa si el entero en el campo del número de cuenta esta entre 1 y 100 ya que la idea es que nuestro archivo contenga solo 100 clientes como máximo, si se dan cuenta ya podemos calcular el tamaño del archivo final, cada registro es de 52 bytes y si tenemos 100 registros el archivo final será de 100x52 = 5200 bytes que también es igual a **5.2kb**, es muy pequeñito, perfecto!!. La función parseInt de la clase Integer convierte cadenas a enteros, en esta ocasión la cadena que se recibe es la contenida en el campo de texto "Tnumero". Si resulta que el entero no es mayor a cero y ademas tampoco es menor a 101 entonces se ejecuta el "else", este pide al usuario que escriba un número de cuenta entre 1 y 100, también resalta este texto para recalcar el mensaje mediante la función selectAll() Por último examinemos el contenido más interno de este if-else que obviamente se ejecutará solo si la condicion del if-else anterior se cumple:

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.nombre = Tnombre.getText();
Double p = new Double(Tpeso.getText());
archivo.peso = p.doubleValue();
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.escribir();
Tnumero.setText("");
Tnombre.setText("");
Tpeso.setText("");
System.out.println("Registro " + (archivo.numero - 1) + " guardado exitosamente");
```

Aqui conviene recordar lo que hacia el metodo escribir() de la clase cuentaRegistro, éste método tomaba los valores almacenados en sus variables globales "numero", "nombre" y "peso" para almacenarlos en forma de bytes de manera ordenada lo cual significa que antes de pretender escribir debemos pasarle estos valores Las primeras líneas (en rojo) se encargan únicamente de eso. **de pasar los valores contenidos en los campos de texto hacia las variables globales de una cuentaRegistro nombrada como archivo**, noten que como estas variables son public significa que podemos acceder a ellas libremente mediante el operador punto, primero se pasa el número, después la cadena que contiene el nombre del cliente y por último su saldo, aqui recurrimos (al igual que en la sección de aplicaciones, cosa que puedes consultar) a la clase Double para convertir un valor String obtenido del campo a un double, un Double es similar aunque no es lo mismo que un double (con d minúscula), un Double solo nos sirve de punte para convertir un String a un double. Recurrimos al constructor de Double-que

recibe una cadena como argumento para tomar el valor double de dicha cadena mediante la función doubleValue()

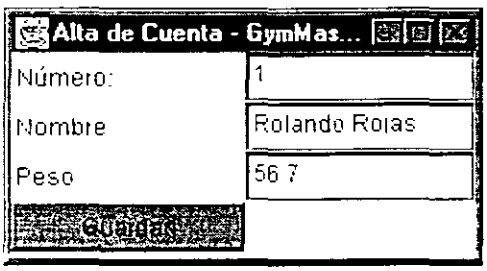
En el código intermedio (marcado en azul marino) se tiene el desplazamiento de nuestro lapicito mediante la función seek hacia el renglón adecuado, esta función recibe como argumento un número de tipo long, como nuestras variables de número de cuenta y de tamaño son enteros estos deben ser convertidos a long (de ahi la palabra long entre paréntesis) se supone que cada renglón tiene una cuenta de un solo cliente completa, si tenemos 100 renglones cada uno con 52 bytes de longitud significa que cada desplazamiento debe ser únicamente de múltiplos de 52, para eso es la variable constante size!!. Por ejemplo si deseamos escribir en la cuenta número 33 tenemos que desplazarnos a la linea 33 pero cada linea mide 52 bytes es decir que en realidad nos desplazamos 33x52 = 1716 bytes totales Si deseamos escribir en la última cuenta entonces nos desplazamos: 99x52 = 5148 bytes, si deseamos escribir en la primera cuenta nos desplazamos 0x52 = 0 bytes, sin embargo las cuentas comienzan desde 1 y terminan en 100 pero los registros comienzan en 0 y terminan en 99, de ahi que se le tenga que restar siempre un uno a la variable que almacena el número de cuenta

```
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.escribir();
```

Si ya obtuvimos los datos y ya desplazamos nuestro lapicito hacia la posición correcta entonces es momento de escribir llamando a la función escribir() de la case cuentaRegistro. Y si ya guardamos los datos ahora tenemos que limpiar los campos de texto para obtener nuevos datos, esto se logra con

```
Tnumero.setText("");
Tnombre.setText("");
Tpeso.setText("");
System.out.println("Registro " + (archivo.numero - 1) + " guardado exitosamente");
```

El System.out.println() recibe una cadena como argumento, recuerden que este mensaje es puramente informativo para indicar en la consola del sistema que por lo menos hasta aquí todo se ha ejecutado bien Esta clase solo es un Frame, no es un programa independiente. Más adelante veremos como cargarla desde el programa principal y se supone que se verá algo asi



La clase "baja"

La siguiente clase es casi igual que la anterior, incluye un poquitito mas de código, aún asi su explicación sera más breve Esta clase hereda de Frame para mostrar una ventana con una interfaz grafica sencilla que nos permita dar de baja una cuenta Como de costumbre les muestro el código y al final su explicacion

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

public class baja extends Frame implements ActionListener{

 Label Lnumero = new Label("Número ");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso ");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button buscarCuenta = new Button("Buscar Cuenta");
 Button eliminarCuenta = new Button("EliminarCuenta");
```

```
cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "rw");

public baja() {
 super("Baja de Cuenta - GymMaster!");
 setLayout( new GridLayout( 4, 2 ) ),
 setBounds(200, 170, 240, 130);

 add(Lnumero);
 add(Tnumero);
 add(Lnombre),
 add(Tnombre);
 add(Lpeso);
 add(Tpeso);
 add(buscarCuenta);
 add(eliminarCuenta);
 eliminarCuenta.setEnabled(false);

 buscarCuenta.addActionListener(this),
 eliminarCuenta.addActionListener(this);
 this addWindowListener ( new cerrarVentana() ),

 setVisible(true),
}

public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo cerrarArchivo();
  dispose();
 }
}

public void actionPerformed(ActionEvent e) {
 if(e getSource() == buscarCuenta && Tnumero.getText() != "") {
  if( Integer parseInt( Tnumero getText() ) > 0 && Integer.parseInt( Tnumero.getText() ) < 101 ) {
   archivo.numero = Integer parseInt( Tnumero.getText() ),
   archivo seek((long) (archivo numero - 1)*archivo size ),
   archivo.leer(),
   Tnumero setText(String.valueOf(archivo numero));
   Tnombre setText(archivo nombre),
   Tpeso setText(String valueOf(archivo peso));
   eliminarCuenta setEnabled(true),
   System out println("Registro " + (archivo numero -1) + " leido exitosamente"),
  } else {
   Tnumero setText("Escribe entre 1 y 100");
   Tnumero selectAll(),
  }
 } else if( e getSource() == eliminarCuenta ) {
  archivo.numero = Integer parseInt( Tnumero.getText() );
  archivo.seek((long) (archivo numero - 1)*archivo size ),
  archivo.numero = 0;
  archivo.nombre = "                ",
  archivo peso = 0.0,
  archivo.escribir(),
  Tnumero.setText(""),
  Tnombre.setText(""),
  Tpeso setText(""),
  eliminarCuenta setEnabled(false),
  System.out println("Registro " + (archivo numero - 1) + " borrado exitosamente"),
 } else {
  Tnumero setText("Escribe algo!"),
  Tnumero selectAll(),
 }
}
```

}

Explicación de la clase "baja"

Primero importamos los paquetes que vamos a utilizar, heredamos de la clase Frame e implementamos la interfaz ActionListener para manejar el evento que generen los botones

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

public class baja extends Frame implements ActionListener {
```

Declaramos nuestros componentes para la interfaz gráfica con el usuario, son casi los mismos que en la clase anterior, lo único que cambia es que ahora son dos botones, el primero nos permitirá buscar la cuenta a eliminar y se supone que cuando se presiona se mostrará la información de dicha cuenta a modo de confirmacion visual y una vez confirmado se activará el botón eliminar.

```
 Label Lnumero = new Label("Número:");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso ");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button buscarCuenta = new Button("Buscar Cuenta");
 Button eliminarCuenta = new Button("EliminarCuenta");
```

Creamos una nueva cuentaRegistro en modo lectura-escritura

```
 cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "rw");
```

Definimos el constructor llamando al constructor de nuestra superclase Frame mediante "super" Especificamos un nuevo Layout (acomodamiento), fijamos la posición y el tamaño de la ventana con setBounds(), agregamos los componentes en orden y agregamos los Listeners a los componentes correspondientes (a la ventana y a los botones), es muy importante que hasta el final ya que todo este configurado y definido plenamente se exhiba el Frame mediante setVisible(true);

```
 public baja() {
  super("Baja de Cuenta - GymMaster!");
  setLayout( new GridLayout( 4, 2 ) );
  setBounds(200, 170, 240, 130);

  add(Lnumero);
  add(Tnumero)
  add(Lnombre);
  add(Tnombre);
  add(Lpeso);
  add(Tpeso)
  add(buscarCuenta)
  add(eliminarCuenta);
  eliminarCuenta.setEnabled(false);

  buscarCuenta.addActionListener(this);
  eliminarCuenta.addActionListener(this);
  this.addWindowListener ( new cerrarVentana() );

  setVisible(true);
 }
```

Definimos la clase interna que se encarga de cerrar el evento heredando de la clase abstracta WindowAdapter que a su vez hereda de WindowListener

```
 public class cerrarVentana extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
```

```
  archivo.cerrarArchivo();
  dispose();
 }
}
```

Uff, por fin llegamos a lo bueno, el método actionPerformed. Al igual que con la explicación anterior vamos a recorrer estructura por estructura cada vez más internamente estudiándola paso a paso.

```
public void actionPerformed(ActionEvent e) {
 if(e.getSource() == buscarCuenta && Tnumero.getText() != "") {
  if( Integer.parseInt( Tnumero.getText() ) > 0 || Integer.parseInt( Tnumero.getText() ) < 101 ) {
   archivo.numero = Integer.parseInt( Tnumero.getText() ),
   archivo.seek((long) (archivo.numero - 1)*archivo.size ),
   archivo.leer();
   Tnumero.setText(String.valueOf(archivo.numero));
   Tnombre.setText(archivo.nombre);
   Tpeso.setText(String.valueOf(archivo peso)),
   eliminarCuenta.setEnabled(true);
   System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente"),
  } else {
   Tnumero.setText("Escribe entre 1 y 100"),
   Tnumero.selectAll();
  }
 } else if( e.getSource() == eliminarCuenta ) {
  archivo.numero = Integer.parseInt( Tnumero.getText() ),
  archivo.seek((long) (archivo.numero - 1)*archivo.size );
  archivo.numero = 0;
  archivo.nombre = "                ",
  archivo.peso = 0.0,
  archivo.escribir(),
  Tnumero setText("");
  Tnombre setText("");
  Tpeso setText("");
  eliminarCuenta.setEnabled(false),
  System out println("Registro " + (archivo.numero - 1) + " borrado exitosamente");
 } else {
  Tnumero.setText("Escribe algo!");
  Tnumero.selectAll(),
 }
}
```

Dentro de esta función tenemos un if-else anidado (uno dentro de otro), el más exterior contiene dos condiciones y un else mientras que el más interno posee una condicion y un else, primero vamos a ver el exterior:

```
 if(e getSource() == buscarCuenta && Tnumero getText() != "") {

 } else if( e getSource() == eliminarCuenta ) {

 } else {
  Tnumero setText("Escribe algo!"),
  Tnumero.selectAll();
 }
```

La primera condición esta en función de dos condiciones con el operador && (doble ampersand) e.getSource() obtiene el nombre del componente u objeto sobre el cual se hizo la acción y si este coincide con el botón "nombrarCuenta" significa que dicho botón se ha oprimido, Tnumero getText() != "" comprueba que el contenido del número de cuenta no este vacio, la segunda condicion de este if comprueba si se ha oprimido el segundo botón mediante otro getSource() solo que esta vez lo compara con el nombre "eliminarCuenta". El else del final maneja el caso en que no se haya escrito ningún numero y se haya oprimido buscarCuenta, obviamente es imposible determinar la cuenta sin el numero! Ahora veamos el if-else que se encuentra dentro de la primera condición del otro bendito if-else más exterior.

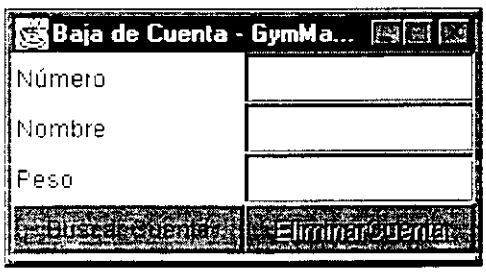
```
  if( Integer.parseInt( Tnumero getText() ) > 0 && Integer.parseInt( Tnumero getText() ) < 101 ) {
```

```
} else {
 Tnumero.setText("Escribe entre 1 y 100");
 Tnumero.selectAll();
}
```

Esto hace exactamente lo mismo que la condición vista para la clase "alta". Se revisa el número de cuenta para proceder solo si esta entre 1 y 100. El "else" muestra un mensaje en caso de que no hayas introducido el número correctamente. Ahora veamos el contenido del if anterior que con mucho es la parte más interesante de toda esta clase.

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.leer();
Tnumero.setText(String.valueOf(archivo.numero));
Tnombre.setText(archivo.nombre);
Tpeso.setText(String.valueOf(archivo.peso));
eliminarCuenta.setEnabled(true);
System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
```

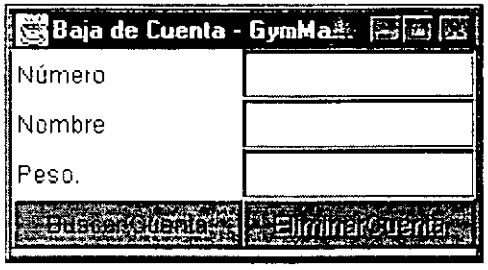
Observa cuidadosamente que para llegar a la sección de código anterior se tuvo que haber cumplido (por los if-else) que el botón presionado sea buscarCuenta y que el número de cuenta haya estado entre 1 y 100. El código en rojo se encarga de obtener el número de cuenta que se pretende buscar directamente desde el campo de texto correspondiente. El código en azul se encarga de colocar nuestro lapicito de lectura en la cuenta apropiada multiplicando los 52 bytes de la variable constante size de la clase cuentaRegistro nombrada "archivo" por el número de cuenta menos uno, ya dijimos que esta resta se debe a que la primera cuenta comienza en cero, no en uno! Una vez colocado el puntero de lectura en la posición correcta del archivo ahora podemos proceder a leer la información mediante la función leer() de la clase cuentaRegistro (sección de código en verde), si esta operación se realiza apropiadamente significa que los valores tomados del archivo se almacenaron en las variables globales de la clase cuentaRegistro y por lo tanto podemos tomarlos libremente con el operador punto tal como se observa, estos valores inmediatamente se fijan a los campos de texto mediante setText(), para esto el número de cuenta debe ser convertido a cadena mediante la función String.valueOf(), lo mismo para el valor double del peso, el nombre ya es una cadena! por lo tanto no es necesario convertirlo. Por último en púrpura activamos el botón "eliminarCuenta", este boton inicialmente aparece desactivado como se ve en la siguiente figura.



¿Porque inicialmente debe estar desactivado este boton? porque antes de borrar cualquier cuenta el programa debe saber su numero, además es muy recomendable exhibir los datos que se pretenden borrar a modo de confirmacion visual, solo hasta ese momento sera posible borrar la cuenta por eso el botón de borrado solo se activa hasta los datos se han leido exitosamente El System out println() tiene como propósito exhibir un mensaje en la consola (desde DOS) que indique el número de registro que se ha leido Ahora revisemos el contenido de la segunda condición del if-else mas exterior que se encarga de borrar la cuenta

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.numero = 0;
archivo.nombre = "                ";
archivo.peso = 0.0;
archivo.escribir();
Tnumero.setText("");
Tnombre.setText("");
Tpeso.setText("");
eliminarCuenta.setEnabled(false);
System.out.println("Registro " + (archivo.numero - 1) + " borrado exitosamente");
```

Observen que para llegar aquí solo se requiere que el botón borrarCuenta sea oprimido, como este botón solo se activa mediante la búsqueda de una cuenta obligamos al usuario a realizar dicha búsqueda antes de intentar borrar cualquier cuenta. El código en rojo obtiene el número de cuenta que pretendemos borrar y despues llama a la función seek de la clase cuentaRegistro para colocar en la posición apropiada nuestra lapicito de escritura. Una vez en la posición correcta de nuestro archivo asignamos a las variables globales de cuentaRegistro valores "vacios" para sobrescribir los datos existentes de modo que desaparezcan los anteriores, como número de cuenta se asigna un cero, como nombre una cadena vacia de 20 caracteres y como peso cero punto cero, acepto que no es del todo necesario borrar el número de cuenta porque de hecho cuando se ingrese otro dato se colocará exactamente el mismo número de cuenta que habia antes (solo cambiará el nombre y el peso) pero es mejor hacerlo por consistencia, por último se llama a la función escribir() de la clase cuentaRegistro. El código en azul cielo limpia los campos de texto de los datos y desactiva el botón de borrado mediante la función setEnabled() que recibe un bolean como argumento, con esto obligamos otra vez al usuario a realizar la búsqueda antes de intentar borrar otra cuenta.



Esta fue toda la clase "baja", por si acaso les recuerdo que esta clase solo es un Frame, no una aplicación. Más adelante veremos que esta clase es llamada desde el programa principal.

La clase "consulta"

La siguiente clase tiene como propósito realizar consultas rápidas de solo lectura, afortunadamente para nosotros es sumamente simple, se parece mucho a las anteriores, veamos el código y al final su explicación

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

public class consulta extends Frame implements ActionListener {

 Label Lnumero = new Label("Número:");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso ");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button buscarCuenta = new Button("Buscar Cuenta");

 cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "r");

 public consulta() {
  super("Consulta de Cuenta - GymMaster!");
  setLayout( new GridLayout( 4, 2 ) );
  setBounds(200, 170, 240, 130);

  add(Lnumero);
  add(Tnumero);
  add(Lnombre);
  add(Tnombre);
  add(Lpeso);
  add(Tpeso);
  add(buscarCuenta);

  buscarCuenta.addActionListener(this);
  this.addWindowListener ( new cerrarVentana() );
```

```
 setVisible(true);
}

public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo.cerrarArchivo(),
  dispose();
 }
}

public void actionPerformed(ActionEvent e) {
 if(e getSource() == buscarCuenta && Tnumero.getText() != "") {
  if( Integer.parseInt( Tnumero.getText() ) > 0 && Integer parseInt( Tnumero getText() ) < 101 ) {
   archivo.numero = Integer parseInt( Tnumero getText() );
   archivo seek((long) (archivo numero - 1)*archivo.size ),
   archivo leer(),
   Tnumero.setText(String valueOf(archivo.numero));
   Tnombre setText(archivo nombre),
   Tpeso.setText(String valueOf(archivo.peso));
   System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
  } else {
   Tnumero setText("Escribe entre 1 y 100"),
   Tnumero.selectAll();
  }
 } else {
  Tnumero setText("Escribe algo!"),
  Tnumero.selectAll(),
 }
}

}
```

Explicación de la clase "consulta"

Primero heredamos los paquetes para interfaz gráfica y para eventos

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

Despues heredamos de la clase Frame e implementamos la interfaz ActionListener que definirá las acciones a realizar una vez atrapado el evento

```
public class consulta extends Frame implements ActionListener {
```

Ahora declaramos e inicializamos los componentes gráficos

```
 Label Lnumero = new Label("Numero:"),
 Label Lnombre = new Label("Nombre: "),
 Label Lpeso = new Label("Peso:"),
 TextField Tnumero = new TextField()
 TextField Tnombre = new TextField(),
 TextField Tpeso = new TextField(),
 Button buscarCuenta = new Button("Buscar Cuenta"),
```

Creamos una nueva cuentaRegistro para abrir el archivo cuentas dat en modo de solo lectura porque esta clase solo hara consultas

```
 cuentaRegistro archivo = new cuentaRegistro("cuentas dat", "r"),
```

Definimos el constructor de la clase que configura tanto la interfaz gráfica como el acomodamiento de los componentes, usamos setBounds() para fijar la posición y el tamaño de la ventana, agregamos los componentes en orden para exhibirlos apropiadamente

```
public consulta() {
 super("Consulta de Cuenta - GymMaster!");
 setLayout( new GridLayout( 4, 2 ) );
 setBounds(200, 170, 240, 130);

 add(Lnumero);
 add(Tnumero);
 add(Lnombre);
 add(Tnombre);
 add(Lpeso);
 add(Tpeso);
 add(buscarCuenta);

 buscarCuenta.addActionListener(this);
 this.addWindowListener ( new cerrarVentana() );

 setVisible(true);
}
```

Al final del constructor anterior registramos los Listeners al botón y a la ventana, la siguiente innerclass define la acción a realizar.

```
public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo.cerrarArchivo();
  dispose();
 }
}
```

El método windowClosing() especifica que el archivo debe cerrarse y después que la ventana debe destruirse mediante dispose() Y ahora llegamos a la parte interesante, la consulta de la informacion del archivo mediante el botón·

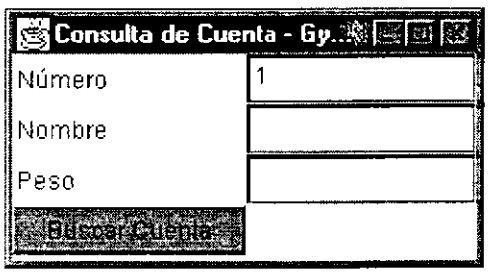
```
public void actionPerformed(ActionEvent e) {
 if(e.getSource() == buscarCuenta && Tnumero.getText() != "") {
  if( Integer.parseInt( Tnumero.getText() ) > 0 && Integer.parseInt( Tnumero.getText() ) < 101 ) {
   archivo.numero = Integer.parseInt( Tnumero.getText() );
   archivo.seek((long) (archivo.numero - 1)*archivo.size );
   archivo.leer();
   Tnumero.setText(String.valueOf(archivo.numero));
   Tnombre.setText(archivo.nombre);
   Tpeso.setText(String.valueOf(archivo.peso));
   System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
  } else {
   Tnumero.setText("Escribe entre 1 y 100");
   Tnumero.selectAll();
  }
 } else {
  Tnumero.setText("Escribe algo!");
  Tnumero.selectAll();
 }
}
```

Esta pelicula ya la vimos Se trata del ya familiar if-else anidado. otra vez se recurre a getSource() para identificar la fuente del evento y a la comparacion del texto contenido en el número de cuenta para comprobar que no este vacio. el if-else interno revisa que el numero de cuenta este entre 1-100 y en tal caso ejecuta

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.leer();
Tnumero.setText(String.valueOf(archivo.numero));
Tnombre.setText(archivo.nombre);
```

```
Tpeso.setText(String.valueOf(archivo.peso));
System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
```

Aquí primero se toma el número de cuenta desde el campo de texto, después se posiciona el lapicito de lectura en la cuenta correcta (con un múltiplo de 52) y se toman los datos del archivo con la función leer(). Finalmente el código en rojo asigna los datos obtenidos a los campos de texto y el mensaje System out.println() exhibe un mensaje de éxito!, juego de niños!.



La clase "modificar"

Este es otro Frame que sirve para modificar los datos, esto implica que abriremos el archivo en modo lectura-escritura "rw" Veamos el código y su explicación correspondiente

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

public class modificar extends Frame implements ActionListener {

 Label Lnumero = new Label("Número ");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso ");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button buscarCuenta = new Button("Buscar Cuenta");
 Button guardarCambios = new Button("Guardar Cambios");

 cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "rw");

 public modificar() {
  super("Modificar Cuenta - GymMaster!");
  setLayout( new GridLayout( 4, 2 ) );
  setBounds(200, 170, 240, 130);

  add(Lnumero);
  add(Tnumero);
  add(Lnombre);
  add(Tnombre);
  add(Lpeso);
  add(Tpeso);
  add(buscarCuenta);
  add(guardarCambios);
  guardarCambios.setEnabled(false);

  buscarCuenta.addActionListener(this);
  guardarCambios.addActionListener(this);
  this.addWindowListener ( new cerrarVentana() );

  setVisible(true);
 }

 public class cerrarVentana extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
```

```
  archivo.cerrarArchivo();
  dispose();
 }
}

public void actionPerformed(ActionEvent e) {
 if(e.getSource() == buscarCuenta && Tnumero.getText() != "") {
  if( Integer.parseInt( Tnumero.getText() ) > 0 && Integer.parseInt( Tnumero.getText() ) < 101 ) {
   archivo.numero = Integer.parseInt( Tnumero.getText() );
   archivo.seek((long) (archivo.numero - 1)*archivo.size );
   archivo.leer();
   Tnumero.setText(String.valueOf(archivo.numero));
   Tnombre.setText(archivo.nombre);
   Tpeso.setText(String.valueOf(archivo.peso));
   guardarCambios.setEnabled(true);
   System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
  } else {
   Tnumero.setText("Escribe entre 1 y 100");
   Tnumero.selectAll();
  }
 } else if( e.getSource() == guardarCambios ) {
  archivo.numero = Integer.parseInt( Tnumero.getText() );
  archivo.nombre = Tnombre.getText();
  Double p = new Double(Tpeso.getText());
  archivo.peso = p.doubleValue();
  archivo.seek((long) (archivo.numero - 1)*archivo.size );
  archivo.escribir();
  Tnumero.setText("");
  Tnombre.setText("");
  Tpeso.setText("");
  guardarCambios.setEnabled(false);
  System.out.println("Registro " + (archivo.numero - 1) + " cambiado exitosamente");
 } else {
  Tnumero.setText("Escribe algo!");
  Tnumero.selectAll();
 }
}

}
```

Explicación de la clase "modificar"

Primero importamos el paquete awt (abstract window toolkit) y el de eventos, heredamos de la clase Frame e implementamos un ActionListener que definira los eventos atrapados

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

public class modificar extends Frame implements ActionListener {
```

Ahora declaramos e inicializmos los componentes que vamos a utilizar

```
 Label Lnumero = new Label("Número:");
 Label Lnombre = new Label("Nombre: ");
 Label Lpeso = new Label("Peso:");
 TextField Tnumero = new TextField();
 TextField Tnombre = new TextField();
 TextField Tpeso = new TextField();
 Button buscarCuenta = new Button("Buscar Cuenta");
 Button guardarCambios = new Button("Guardar Cambios");
```

Y claro también declaramos una nueva cuentaRegistro que en esencia es el archivo que nos permite leer y escribir

```
cuentaRegistro archivo = new cuentaRegistro("cuentas.dat", "rw");
```

Definimos el constructor y en él configuramos la interfaz gráfica, registramos los Listeners y por último hacemos visible el Frame con setVisible(true):

```
public modificar() {
 super("Modificar Cuenta - GymMaster!");
 setLayout( new GridLayout( 4, 2 ) );
 setBounds(200, 170, 240, 130);

 add(Lnumero);
 add(Tnumero);
 add(Lnombre);
 add(Tnombre);
 add(Lpeso);
 add(Tpeso);
 add(buscarCuenta);
 add(guardarCambios);
 guardarCambios.setEnabled(false);

 buscarCuenta.addActionListener(this);
 guardarCambios.addActionListener(this);
 this.addWindowListener ( new cerrarVentana() );

 setVisible(true);
}
```

Ahora declaramos una innerclass que define la acción a realizar al cerrar la ventana cerrar el archivo y destruir el Frame

```
public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  archivo cerrarArchivo();
  dispose();
 }
}
```

Por ultimo tenemos el corazon de los eventos para los botones: el método actionPerformed que exige la interfaz ActionListener

```
public void actionPerformed(ActionEvent e) {
 if(e getSource() == buscarCuenta && Tnumero getText() != "") {
  if( Integer parseInt( Tnumero getText() ) > 0 && Integer parseInt( Tnumero getText() ) < 101 ) {
   archivo numero = Integer parseInt( Tnumero getText() );
   archivo seek((long) (archivo numero - 1)*archivo size );
   archivo leer();
   Tnumero setText(String valueOf(archivo numero));
   Tnombre.setText(archivo nombre);
   Tpeso setText(String.valueOf(archivo peso));
   guardarCambios.setEnabled(true);
   System out println("Registro " + (archivo.numero -1) + " leido exitosamente");
  } else {
   Tnumero.setText("Escribe entre 1 y 100");
   Tnumero.selectAll();
  }
 } else if( e.getSource() == guardarCambios ) {
  archivo numero = Integer parseInt( Tnumero getText() );
  archivo nombre = Tnombre.getText();
  Double p = new Double(Tpeso getText());
  archivo peso = p.doubleValue();
  archivo.seek((long) (archivo numero - 1)*archivo size );
  archivo.escribir();
  Tnumero.setText("");
```

```
Tnombre.setText("");
Tpeso.setText("");
guardarCambios.setEnabled(false);
System.out.println("Registro " + (archivo.numero - 1) + " cambiado exitosamente");
} else {
Tnumero.setText("Escribe algo!");
Tnumero selectAll();
}
  }
```

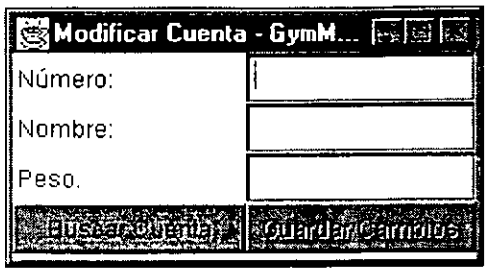
Este método es casi igual al que escribimos para la clase "consulta", tenemos un if-else anidado (uno dentro de otro), el más exterior tiene dos condiciones, la primera es que el event provenga del botón "buscarCuenta" y además que el número de cuenta no este vacio, en su interior tenemos otro if-else que tiene como condición que el número de cuenta este entre 1 y 100. Si lo anterior se cumple (lo cual significa que estamos obteniendo la cuenta que pretendemos modificar) entonces se ejecuta.

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.leer();
Tnumero.setText(String.valueOf(archivo.numero));
Tnombre.setText(archivo.nombre);
Tpeso.setText(String.valueOf(archivo.peso));
guardarCambios.setEnabled(true);
System.out.println("Registro " + (archivo.numero -1) + " leido exitosamente");
```

La sección en verde del código anterior pasa el entero del campo de texto del número de cuenta hacia la variable global "numero" de la clase cuentaRegistro Después llama al método seek que se encarga de colocar el lapicito (que en este caso nos servirá para leer) en la posición correcta dentro del archivo. En rojo iniciamos leyendo el contenido del archivo y después pasamos los valores leidos y contenidos en las variables globales de cuentaRegistro hacia todos y cada uno de los campos de texto. Por último activamos el botón "guardarCambios", nuevamente al igual que en la clase "baja" obligamos al usuario a obtener la cuenta antes de modificarla activando y desactivando la posibilidad de usar los botones solo cuando se tenga una confirmación visual, en este caso es aun mas importante hacer esto Una vez obtenida la cuenta anterior ahora se activa el botón guardarCambios, si oprimimos este botón se ejecuta el contenido de la segunda condición del if-else superior:

```
archivo.numero = Integer.parseInt( Tnumero.getText() );
archivo.nombre = Tnombre.getText();
Double p = new Double(Tpeso.getText());
archivo.peso = p.doubleValue();
archivo.seek((long) (archivo.numero - 1)*archivo.size );
archivo.escribir();
Tnumero.setText("");
Tnombre.setText("");
Tpeso.setText("");
guardarCambios.setEnabled(false);
System.out.println("Registro " + (archivo.numero - 1) + " cambiado exitosamente");
```

La seccion en rojo del codigo anterior se encarga a grandes rasgos de tomar los valores de los campos de texto y pasarlos hacia las variables globales de la clase cuentaRegistro cuyo nombre es archivo, nuevamente recurrimos al puente Double que nos permite convertir una cadena en un valor double (recuerden que aunque son similares no son iguales). El código en azul oscuro desplaza el puntero o lapicito para escribir en la linea correcta de nuestro archivo, este desplazamiento siempre es en multiplos de 52 bytes por eso se multiplica por la variable size de la clase cuentaRegistro El codigo verde limpia los campos de texto justo después de que se ha escrito y finalmente desactiva el boton guardarCambios para obligar al usuario a localizar otra cuenta antes de intentar modificarla



La clase "archivosAleatorios" (Programa Principal)

Quizás podríamos sospechar que el programa principal que cargará las cinco clases anteriores podría ser bastante extenso y complejo pero no es así, es muy simple!, observen la clase y noten que posee el método main() que lo define como una verdadera aplicación independiente, al final explico su código.

```
import java.awt.*;
import java.awt.event.*;
import java.io.File;

public class archivosAleatorios extends Frame implements ActionListener {

 Button alta = new Button("Alta de Cuenta");
 Button baja = new Button("Baja de Cuenta");
 Button modif = new Button("Modificar Cuenta");
 Button cons = new Button("Consulta Rapida");

 alta a;
 baja b;
 modificar m;
 consulta c;

 public archivosAleatorios() {
  super("GymMaster!");
  setLayout(new GridLayout( 2, 2  10, 10 ));
  setBounds(200, 150, 250, 130);
  this.addWindowListener ( new cerrarVentana() );

  add(alta);
  add(modif);
  add(baja);
  add(cons);
  alta.addActionListener(this);
  modif.addActionListener(this);
  baja.addActionListener(this);
  cons.addActionListener(this);

  setVisible(true);
 }

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

 public void actionPerformed(ActionEvent event) {
  System.gc();
  if(event.getSource() == alta) {
   a = new alta();
  } else if (event.getSource() == baja) {
   b = new baja();
```

```
 } else if (event.getSource() == modif) {
  m = new modificar();
 } else if (event.getSource() == cons) {
  c = new consulta();
 }

}

public static void main(String args[ ])  {

 try {
  File cuentas = new File("cuentas.dat");
  if(cuentas.exists() == false) {
   cuentaRegistro R = new cuentaRegistro("cuentas.dat", "rw");
   for(int i = 0; i < 100; ++i)
    R.escribir(),
   R.cerrarArchivo();
   System.out.println("El archivo cuentas.dat no existia, fue creado exitosamente");
  } else {
   System.out.println("Usando el archivo de cuentas existente");
  }
 } catch (NullPointerException e) {
  System.out.println("No diste el nombre del archivo a verficar si existe!" + e.toString()),
 }

 archivosAleatorios f = new archivosAleatorios(),
}
}
```

Explicación de la aplicación "archivosAleatorios"

Antes de continuar hay que aclarar ciertas cuestiones importantes **Primero** esta clase también heredará de un Frame asi que sera una ventana que exhiba los botones principales asociados con cada una de las cuatro principales acciones: Alta de cuenta, Baja de Cuenta, Modificación de cuenta y Consulta Rápida de una cuenta. **Segundo**, esta clase hará uso de la clase File para checar si existe el archivo "cuentas dat", si el archivo ya existe entonces no se tocará para dejarlo tal cual y asi las otras clases lo puedan utilizar sin ningún tipo de modificación Si el archivo no existe entonces debe ser "cuadriculado" (para definir su tamaño) con los datos que pretendemos almacenar (es decir un int, una String y un double), este cuadriculado se logra recurriendo nuevamente a la clase "cuentaRegistro" como veremos mas adelante Y **Tercero** cada Frame creado anteriormente: "alta", "baja", "consulta" y "modificar" posee la capacidad de cerrarse solo asi que este programa no necesita preocuparse de eso, solo debe ocuparse de crearlos cuando se oprima el botón correspondiente, lo mismo sucede con la lectura y/o escritura en el archivo, eso es asunto personal de cada Frame que no tiene nada que ver con el siguiente programa principal!

Primero importamos el paquete del Abstract Window Toolkit, el de eventos y la clase File.

```
import java.awt.*,
import java.awt.event.*,
import java.io.File,
```

Ahora heredamos de un Frame e implementamos la interfaz ActionListener que definira la acción asociada con los eventos en los botones:

```
public class archivosAleatorios extends Frame implements ActionListener {
```

Declaramos e inicializamos cuatro botones, la idea es que cada uno llame a las otras clases creadas anteriormente:

```
 Button alta = new Button("Alta de Cuenta"),
 Button baja = new Button("Baja de Cuenta"),
 Button modif = new Button("Modificar Cuenta"),
 Button cons = new Button("Consulta Rapida"),
```

Declaramos solamente (sin inicializar aún) cuatro tipos de las clases ya creadas con nombres bastante simples

```
alta a;
baja b;
modificar m;
consulta c,
```

Definimos el constructor de nuestro programa principal que configura toda la interfaz gráfica.

```
public archivosAleatorios() {
 super("GymMaster!");
 setLayout(new GridLayout( 2, 2, 10, 10 )),
 setBounds(200, 150, 250, 130);
 this.addWindowListener ( new cerrarVentana() ),

 add(alta);
 add(modif),
 add(baja);
 add(cons);
 alta addActionListener(this),
 modif.addActionListener(this),
 baja.addActionListener(this),
 cons addActionListener(this),

 setVisible(true),
}
```

Primero se llama al constructor de Frame mediante super() y se le pasa el titulo como argumento, despues fijamos un nuevo acomodamiento con setLayout() que será un nuevo GridLayout con dos lineas y dos columnas, el tercer y cuarto argumento de este constructor son las separaciones horizontal y vertical entre componentes respectivamente. setBounds() nos permite fijar la colocacion y el tamaño de la ventana y se agrega un nuevo WindowListener a la ventana cuya acción esta definida en la siguiente innerclass:

```
public class cerrarVentana extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  dispose()
  System exit(0),
 }
}
```

En esta ocasión no solo se cerrara el Frame, tambien se cerrara completamente el programa mediante la llamada a System exit(0). El siguiente es el método actionPerformed que es impresionantemente simple (gracias a que definimos cada Frame de "alta", "baja", "modificar" y "consulta" de manera independiente)

```
public void actionPerformed(ActionEvent event) {
 if(event getSource() == alta) {
  a = new alta(),
 } else if (event.getSource() == baja) {
  b = new baja(),
 } else if (event.getSource() == modif) {
  m = new modificar(),
 } else if (event getSource() == cons) {
  c = new consulta():
 }
}
```

Nota: No deben confundir el nombre de los botones con el nombre de las clases "alta", "baja", "modificar" y "consulta", son cosas totalmente diferentes!

Como para exhibirlos solo basta llamar a su constructor puesto que cada uno se configura a si mismo, asi de fácil!! Solo tenemos cuatro botones, el if-else anterior verifica de donde proviene el evento generado y si el objeto cuyo nombre es el boton "alta" entonces se crea un nuevo objeto de tipo alta mediante su constructor, si el boton

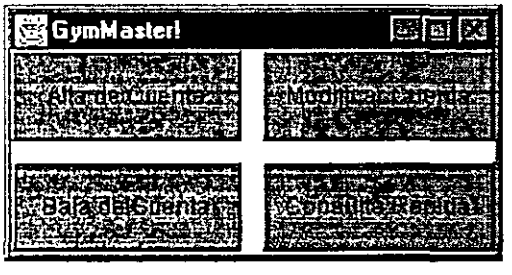
de donde proviene el evento es el nombrado como "baja" entonces se crea un nuevo objeto baja y así sucesivamente.

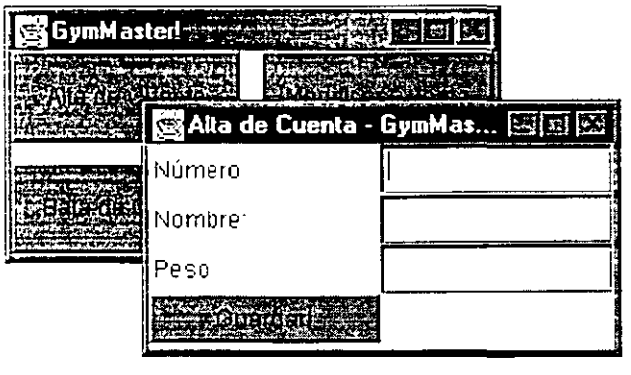
Ahora el método main que es el encargado de cargar este Frame principal y el que revisa si es que el archivo "cuentas.dat" existe o no:

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

 try {
  File cuentas = new File("cuentas.dat");
  if(cuentas.exists() == false) {
   cuentaRegistro R = new cuentaRegistro("cuentas.dat", "rw");
   for(int i = 0; i < 100; ++i)
    R.escribir(),
   R cerrarArchivo();
   System.out.println("El archivo cuentas.dat no existia, fue creado exitosamente");
  } else {
   System out.println("Usando el archivo de cuentas existente");
  }
 } catch (NullPointerException e) {
  System.out.println("No diste el nombre del archivo a verficar si existe!" + e.toString());
 }

 archivosAleatorios f = new archivosAleatorios();
}
```

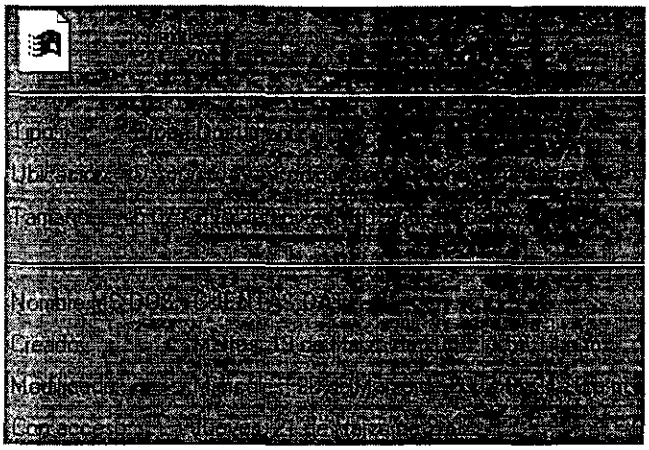
Primero creamos un nuevo File tomando como argumento para su constructor el nombre del archivo "cuentas.dat", este constructor puede lanzar diversos tipos de excepciones que pueden ser atrapadas como un NullPointerException tal como se ve. Después tenemos el if-else que revisa si el archivo "cuentas.dat" ya esta presente o no mediante la función exists() de la clase File, si no existe entonces creamos un nuevo cuentasRegistro nombrado R y escribimos en él cien registros vacios, terminada la escritura cerramos el archivo y mostramos un anuncio de que el archivo se creo correctamente. Por último llamamos al constructor de esta clase "archivosAleatorios", recuerden que el método main es como si fuera una sección independiente de la clase que es llamada por el interprete Java por eso es posible llamar el constructor dentro de la misma clase El método main es lo primero que se busca al ejecutar un programa independiente escrito en Java

Si todo sale bien entonces ya pueden correr la aplicación que se verá algo así





Originalmente no hay archivo "cuentas.dat", así que se supone que debes introducir algunas cuentas (darlas de alta) para después probar con ellas ya sea una baja, una modificación o una consulta. A modo de confirmación observen que al consultar las propiedades del archivo "cuentas.dat" podemos notar que su tamaño coincide con todas las suposiciones sobre conversión a bytes para los tipos de datos, tenemos 100 clientes, cada cliente tiene: cuenta, nombre y peso (52bytes) 100x52 = 5200 bytes.



El código de este programa no es mucho sin importar lo grande que parezca esta sección, un programa de este tipo con práctica sale cuando mucho en 2 horas y eso exageradamente! aún así les pongo el código en zip si desean ahorrarse codificar todo (no debería promover la flojera he!)

Código del Programa Ejemplo (GymMaster!)

Gracias a Java tenemos más posibilidades, cada uno de ustedes fijará su propio límite. Experimenta!! pero sobre todo· nunca olvides que aprender siempre puede ser divertido!!

Temas Avanzados de Java

- Introducción a Swing
- JFrame y JButton
- JTextField
- JLabels
- JTabbedPane

Introducción a Swing

El paquete llamado Swing corresponde a la Java Foundation Classes (JFC) que originalmente era parte de un proyecto alterno (de JavaSoft) para crear bibliotecas Java un tanto más simples, flexibles y prácticas, a partir del JDK versión 1.2 se incluyó de manera oficial en el lenguaje La JFC se divide en la biblioteca de Accesibilidad (para personas discapacitadas) y en Swing que es un paquete para interfaces gráficas, Swing tiene la característica principal de que esta escrito totalmente en Java al 100%, estos componentes se les clasifican como "lightweight" porque en teoría no se relacionan de ninguna manera con la maquinaria gráfica nativa del sistema cosa que el Abstract Window Toolkit si hace.

El código para AWT en realidad muestra botones, ventanas y etiquetas (entre otros componentes) basándose en el sistema operativo de la máquina local, o sea que en realidad solo se "toman prestados". Se supone que con Swing estos componentes se renderizan (dibujan) a partir del mismo Java y por lo tanto se verán exactamente iguales en cualquier plataforma capaz de interpretar java, entonces el término "lightweight" solo hace alusión a que son componentes portátiles El hecho de que los componentes Swing se dibujen implica que necesariamente son un poco más lentos de interpretar que los componentes del AWT sin embargo existen muchisimos más componentes en Swing que en el Abstract Window Toolkit, además todos los elementos de Swing se diseñaron teniendo en mente entornos de desarrollo gráficos tales como JBuilder, Forté, Visual Age, entre otros, sin embargo cualquier componente Swing se mantiene relativamente simple (aunque con muchas más propiedades que los de AWT) para poder programarlo a mano sin la necesidad de un entorno gráfico

Esta seccion pretende ser una introducción super básica a Swing, es más ni siquiera eso, solo se tratan de algunos ejemplo sencillos. El paquete Swing es tan extenso que existen libros muy grandes que lo tratan aunque un buen programador experimentado solo requeriria de la Java API para utilizarlo

Nota Importante: Para compilar y ejecutar estos ejemplos necesitas del JDK 1 2 ó superior, si intentas compilarlos con el JDK 1 1 8 no podrás hacerlo, esta seccion es breve y con un caracter puramente introductorio por eso el curso sigue basandose en el JDK 1 1 8 excepto esta sección, en teoría un principiante primero debería dominar bien el AWT y una vez comprendido pasar a Swing y sus extraordinarias capacidades (Se supone que ya revisaste bien la seccion de Interfaz Grafica y Eventos)

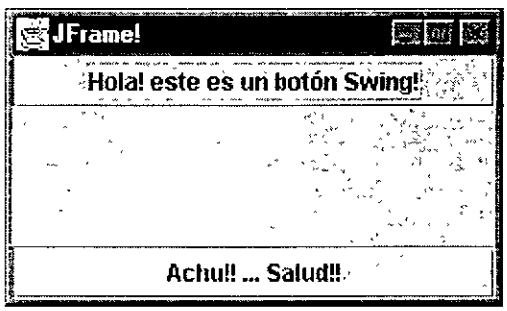
Ejemplo de JFrame y JButton

Aunque los componentes gráficos Swing son mas flexibles se procuró mantenerlos simples y relativamente parecidos (en su uso) a los del AWT, los siguientes ejemplos muestra el uso de un JFrame y JButton que son casi iguales que los Frame y Button convencionales del AWT, incluso sus constructores y sus métodos son muy parecidos como podran ver

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class pruebaJFrame extends JFrame {

 JButton a, b;

 public pruebaJFrame() {
   setTitle("JFrame!");
   setBounds( 50, 50, 250, 150);
```

```
    a = new JButton("Achu!! ... Salud!!");
    b = new JButton("Hola! este es un botón Swing!");
    getContentPane() add( "North", b );
    getContentPane() add( "South", a );

    addWindowListener(new cerrarVentana() );
    setVisible(true);
  }

  public class cerrarVentana extends WindowAdapter {
    public void windowClosing(WindowEvent e){
    dispose();
    System exit(0);
   }
  }

  public static void main(String args[]) {
   pruebaJFrame f = new pruebaJFrame();
  }
 }
```

La salida de esta aplicación es



Comenzamos importando los paquetes que vamos a utilizar, la razón por la cual se continúa importando el java awt *, es porque swing hereda directamente de este aunque solo recurre a sus capacidades de dibujo (no de componentes), esta vez tambien importamos el paquete javax swing *

```
import java awt *;
import java awt event *;
import javax swing *;
```

Al igual que cuando utilizabamos Frame esta vez extendemos JFrame, lo anterior implica que ésta será una aplicacion Java

```
public class pruebaJFrame extends JFrame {
```

Ahora declaramos dos botones

```
 JButton a, b;
```

Como puedes ver los metodos son muy similares a los del AWT, setTitle() fija el título que se exhibira en la ventana, setBounds() fija la colocacion y el tamaño en pixeles, los constructores de JButton son prácticamente iguales a los de Button que recibian un argumento String que representaba la etiqueta que exhiben

```
 public pruebaJFrame()  {
    setTitle("JFrame!");
    setBounds( 50  50, 250, 150);

    a = new JButton("Achu!!    Salud!!");
    b = new JButton("Hola! este es un boton Swing!");
    getContentPane() add( "North", b );
    getContentPane().add( "South", a );
```

```
    addWindowListener(new cerrarVentana() );
    setVisible(true);
}
```

La única sutil diferencia es que para agregar componentes se recurre al método getContentPane() add() que recibe cualquier tipo de componente Swing como argumento, el Layout predefinido para un JFrame es BorderLayout, los Layouts se manejan exactamente igual que con el AWT aunque Swing soporta Layouts adicionales, si se desea cambiar la posición predeterminada (como en este ejemplo) se especifica la nueva colocación con una cadena "North", "South", "East", "West" o "Center" seguido del nombre del componente Finalmente en el constructor anterior se agrega un WindowListener (basado en la siguiente innerclass) que permitirá cerrar la ventana correctamente, la función setVisible(true) garantiza que el JFrame se exhibirá en pantalla.

```
public class cerrarVentana extends WindowAdapter {
   public void windowClosing(WindowEvent e){
  dispose();
  System exit(0),
 }
}
```

Esta innerclass al igual que con los ejemplos del AWT maneja el evento para liberar los recursos que ocupa el JFrame (con dispose()) y provoca el final del programa con System exit(0) únicamente cuando se cirra la ventana, nuevamente esta clase se extiende de la clase WindowAdapter para ahorrarnos tener que definir todos los metodos que exige la interfaz WindowListener (asi solo definimos el que queramos, en este caso windowClosing()) Por último como esta es una aplicación requiere de un método main que declare e inicialice el JFrame

```
public static void main(String args[]) {
 pruebaJFrame f = new pruebaJFrame(),
}
```

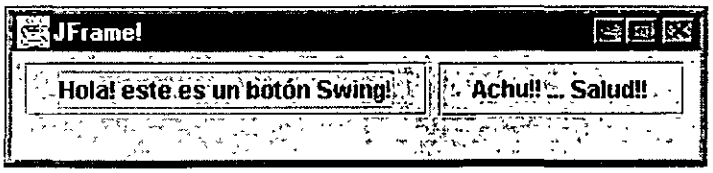
Otro Ejemplo de JFrame, ahora con FlowLayout

Nuevamente se verán cosas muy familiares, el siguiente ejemplo muestra un JFrame Swing con FlowLayout

```
import java awt *,
import java awt event *,
import javax swing *,

public class pruebaFlowLayout extends JFrame {

JButton a, b

public pruebaFlowLayout()  {
   setTitle("FlowLayout!"),
   setBounds( 50, 50  350, 80),
   getContentPane() setLayout(new FlowLayout() ),

   a = new JButton("Achu!!    Salud!!"),
   b = new JButton("Hola! este es un boton Swing!"),
   getContentPane() add( b ),
   getContentPane() add( a ),

   addWindowListener(new cerrarVentana() ),
   setVisible(true),
}

public class cerrarVentana extends WindowAdapter {
   public void windowClosing(WindowEvent e){
  dispose(),
  System exit(0),
 }
}
```

```
public static void main(String args[]) {
 pruebaFlowLayout f = new pruebaFlowLayout();
}
}
```

La salida de esta aplicación es:



Se comienza importando paquetes y heredando de la clase JFrame, también se declaran dos botones

```
import java.awt.*;
import java.awt event *;
import javax.swing *;

public class pruebaFlowLayout extends JFrame {

 JButton a, b;
```

Ahora definimos el constructor de nuestra clase, este configura el JFrame fijándole un titulo y un tamaño, también agrega un nuevo Layout de tipo FlowLayout, este Layout tal como lo indica la tabla de tipos de Layouts mostrada en la sección de Interfaz Gráfica y Eventos sirve para acomodar los componentes de manera lineal, es decir que conforme se agregan se ordenan de izquierda a derecha (sin la necesidad de columnas, simplementa van uno seguido de otro)

```
 public pruebaFlowLayout()  {
   setTitle("FlowLayout!");
   setBounds( 50, 50, 350, 80);
   getContentPane().setLayout(new FlowLayout() );

   a = new JButton("Achu!!   . Salud!!");
   b = new JButton("Hola! este es un botón Swing!");
   getContentPane() add( b );
   getContentPane() add( a );

   addWindowListener(new cerrarVentana() );
   setVisible(true);
 }
```

Nuevamente recurrimos a getContentPane() add() observen que primero coloqué el botón "b" y despues el "a" cosa que se nota en la salida del programa Finalmente agregamos un WindowListener definido por otra innerclass y hacemos visible el JFrame mediante setVisible(true)

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

 public static void main(String args[]) {
  pruebaFlowLayout f = new pruebaFlowLayout();
 }
}
```

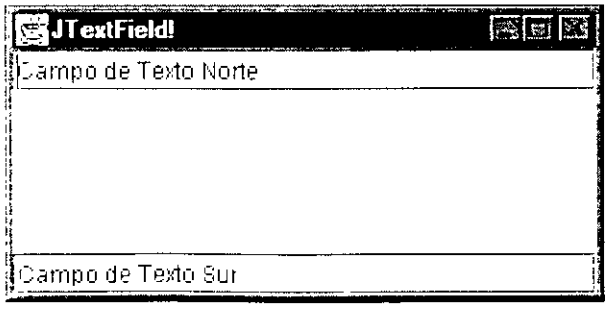
Esta pelicula ya la vimos, nada nuevo por aqui.

Ejemplo de JTextField

Un JTextField es la versión del TextField del AWT, es decir un campo de texto capaz de recibir datos tales como números o cadenas, sería muy fácil por ejemplo adaptar la versión que hicimos de la obtención de datos de la sección de Interfaz Gráfica a Swing Este código no hace realmente nada más que exhibir los JTextField, veamos el código y después, su explicación.

```
import java.awt *;
import java awt event.*;
import javax.swing *;

public class pruebaJTextField extends JFrame {

 JTextField a, b,

 public pruebaJTextField()  {
   setTitle("JTextField!");
   setBounds( 50, 50, 300, 150);

   a = new JTextField();
   b = new JTextField();
   getContentPane() add( "North", a );
   getContentPane().add( "South", b );
   a setText("Campo de Texto Norte");
   b setText("Campo de Texto Sur"),

   addWindowListener(new cerrarVentana() );
   setVisible(true),
 }

 public class cerrarVentana extends WindowAdapter {
   public void windowClosing(WindowEvent e){
  dispose(),
  System exit(0),
  }
 }

 public static void main(String args[]) {
  pruebaJTextField f = new pruebaJTextField(),
 }
}
```

La salida de la aplicación es



Lo mismo de siempre, importar paquetes, heredar de JFrame, declarar componentes (JTextFields "a" y "b").

```
import java.awt *,
import java awt event *,
import javax swing *,
```

```
public class pruebaJTextField extends JFrame {

 JTextField a, b;
```

Después definir configurar el JFrame mediante el siguiente constructor:

```
 public pruebaJTextField()  {
   setTitle("JTextField!");
   setBounds( 50, 50, 300, 150);

   a = new JTextField(),
   b = new JTextField(),
   getContentPane().add( "North", a );
   getContentPane() add( "South", b );
   a setText("Campo de Texto Norte");
   b.setText("Campo de Texto Sur");

   addWindowListener(new cerrarVentana() );
   setVisible(true);
 }
```

Primero fijamos el titulo con setTitle(), después el tamaño y la posición con setBounds(), inicializamos los JTextField llamado a sus constructores que no reciben argumentos, agregamos los componentes al JFrame mediante getContentPane().add() especificando la colocación exacta en el BorderLayout (que es el Layout activado por default) Finalmente registramos el WindowListener sobre este JFrame que realizará la acción especificada en la siguiente innerclass:

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

 public static void main(String args[]) {
  pruebaJTextField f = new pruebaJTextField();
 }
}
```

Y claro colocamos el método main que arranca nuestro programa llamando a su respectivo constructor.

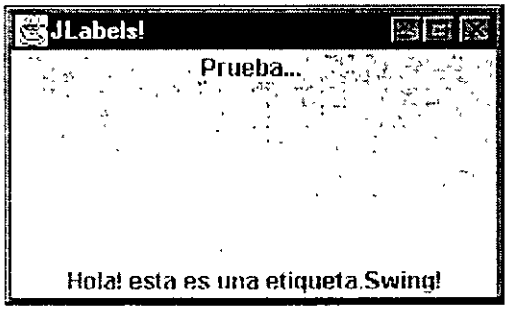
Ejemplo de JLabel

Un JLabel es la contraparte Swing de una Label, es decir una simple etiqueta que exhibe texto Este ejemplo utiliza el BorderLayout predeterminado para colocar dos JLabel una al Norte y otra al sur

```
import java.awt *,
import java.awt event *,
import javax swing *;

public class pruebaLabels extends JFrame {

 JLabel a, b;

 public pruebaLabels()  {
   setTitle("JLabels!");
   setBounds( 50, 50, 250, 150);

   a = new JLabel("Prueba .");
   b = new JLabel("Hola! esta es una etiqueta Swing!"),
   getContentPane().add( "North", a );
   getContentPane().add( "South", b ),
   a setHorizontalAlignment( JLabel CENTER ),
   b setHorizontalAlignment( JLabel.CENTER ),
```

```
    addWindowListener(new cerrarVentana() ),
    setVisible(true),
  }

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

  public static void main(String args[]) {
   pruebaLabels f = new pruebaLabels(),
  }
 }
```

Es muy similar a los ejemplos anteriores por lo que considero que no requiere mayor explicación en detalle, su salida es.



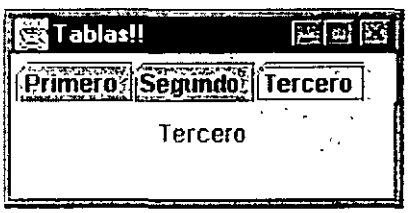
Ejemplo de JTabbedPane

Esta clase de componente es la mas interesante de esta introduccion super básica a los componentes Swing Los TabbedPane (Paneles de Tabla) sirven para intercambiar diversas vistas, cada una de estas vistas puede contener otros JPanels y dentro de tales JPanels más componentes tales como JButtons, JLabels o JTextFields, es uno de tantos componentes interesantes que simplemente no estan disponibles en el AWT

```
import java.util.*,
import java.awt *,
import javax.swing *,
import javax.swing.border *,

public class pruebaTablas extends JPanel {

 public pruebaTablas() {
  JTabbedPane tablas
  JPanel panel,

  setLayout(new BorderLayout()),

  tablas = new JTabbedPane(),

  panel = new JPanel();
  panel.add( new JLabel("Primero") ),
  tablas.addTab("Primero", panel)

  panel = new JPanel(),
  panel.add( new JLabel("Segundo") ),
  tablas.addTab("Segundo", panel),
```

```
 panel = new JPanel();
 panel.add( new JLabel("Tercero") );
 tablas addTab("Tercero", panel);

 tablas.setSelectedIndex(0);
 add(tablas, "Center");
}

public static void main(String args[]) {
 JFrame f = new JFrame("Tablas!!");
 pruebaTablas p = new pruebaTablas();
 f.getContentPane().add(p, "Center");
 f.addWindowListener(new cerrarVentana());
 f.setBounds(25, 25, 200, 100);
 f setVisible(true);
}
}
```

La salida de esta aplicación es:



Primero importamos diversos paquetes necesarios para utilizar tablas, esta vez son más que en los anteriores ejemplos

```
import java.util *;
import java awt *;
import javax.swing.*;
import javax.swing.border.*;
```

Soy sincera, aún estoy aprendiendo a usar Swing al igual que ustedes por lo que debo investigar más a fondo la razon por la cual se importan tantos paquetes Recuerden que este es un curso para principiantes al menos por ahora! Observen que en este programa no heredamos la clase principal directamente de JFrame sino de JPanel, sin embargo si utilizaremos un JFrame que exhibira nuestra JTabbedPane lo que pasa es que esta vez el programa no se centrara en definir el JFrame sino unicamente utilizar uno sencillito para exhibir un muy elaborado JPanel.

```
public class pruebaTablas extends JPanel {
```

El constructor de nuestra clase queda

```
public pruebaTablas() {
 JTabbedPane tablas;
 JPanel panel;

 setLayout(new BorderLayout());

 tablas = new JTabbedPane();

 panel = new JPanel();
 panel add( new JLabel("Primero") );
 tablas addTab("Primero", panel);

 panel = new JPanel();
 panel add( new JLabel("Segundo") );
 tablas addTab("Segundo", panel);
```

```
 panel = new JPanel();
 panel add( new JLabel("Tercero") ),
 tablas.addTab("Tercero", panel);

 tablas setSelectedIndex(0);
 add(tablas, "Center");
}
```

Primero declaramos un JTabbedPane nombrado "tablas", después un JPanel nombrado "panel" ¿se nota que tengo mucha imaginación verdad?, en fin. Después fijamos como Layout de nuestro Panel como BorderLayout, recuerden que tanto Frames, JFrames como Panes y JPanels soportan Layouts Ahora sigue algo curioso, utilizamos el mismo objeto panel tres veces (llamando en tres ocasiones a su constructor) esto es para poder agregar tres Panels distintos al JTabbedPane nombrado "tablas", claro que es posible declarar tres distintos JPanel cada uno con su propio nombre y sus propias caracteristicas pero aqui con el único proposito de mantener el ejemplo relativamente simple se utiliza el mismo JPanel (modificandolo ligeramente en cada caso) para agregarlo al JTabbedPane. La función addTab() que actúa sobre el objeto "tablas" de la clase JTabbedPane() sirve para agregar una "tabla" (que en este caso son panels), el objeto "tablas" actúa como administrador de cada una de los JPanels agregados, el primer argumento es el nombre con el cual vamos a identificar cada tabla y el segundo la tabla propiamente que queremos agregar, recuerden: creamos tres JPanel basándonos en el mismo nombre pero no en el mismo contenido pues cada uno posee una JLabel diferente, lo ideal seria declarar cada JPanel de manera independiente

Finalmente desde el método main creamos un nuevo JFrame que será el que contendrá todos los componentes preparados incluido obviamente el JTabbedPane.

```
 public static void main(String args[]) {
  JFrame f = new JFrame("Tablas!!"),              //Crea el JFrame que contendrá todo
  pruebaTablas p = new pruebaTablas(),            //Crea un JPanel que contendrá el JTabbedPane y los
subJPanels
  f getContentPane().add(p, "Center"),            //agrega al JFrame el JPanel principal (creado a partir de nuestra
clase)
  f addWindowListener(new cerrarVentana()),       //agrega un WindowListener definido en una clase adicional que
veremos mas abajo
  f setBounds(25, 25, 200, 100),                  //Fija la posición y el tamaño del JFrame
  f setVisible(true),                             //Hace visible el JFrame
 }
}
```

Estas funciones que anteriormente habíamos definido en el constructor también se pueden invocar desde el exterior de la clase pues todas son public, unicamente recurrimos al operador punto y listo! En esta ocasión nuestra clase heredaba de JPanel, sin embargo sigue siendo un programa basado en un JFrame que debe ser cerrado asi que debemos definir en otra clase a parte nombrada "cerrarVentana" el objeto WindowListener que se encargara de cerrar la aplicacion

```
import java awt *,
import java awt event *,

public class cerrarVentana extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    Window win = e getWindow(),
  win dispose(),
  System exit(0)
 }
}
```

Como ahora esta no es una innerclass debe obtener de alguna manera la ventana que se pretende cerrar, afortunadamente para nosotros el objeto WindowEvent puede obtener tal ventana mediante el método getWindow() El metodo dispose() unicamente esta disponible para las subclases de "Window", como Frame y JFrame son subclases de esta funciona a la perfección Si estan interesados en aprender muchisimo mas sobre Swing consigan el JavaTutorial que es el curso oficial de Sun Microsystems, este se consigue en

http://www.java.sun.com

Buscando· Java Tutorial, el único defecto de este Tutorial es que es de unos 10Mb y además esta en Inglés, suerte

Serialización

- ¿Qué es Serialización?
- Ejemplo de Serialización - Programa Administrador de Clientes
- Pseudocódigo del Programa
- Código del Programa
- Explicación del Código
- Bajar el Código Fuente del Programa

Nota: Para comprender bien esta sección debes consultar el tema de Vectores de la sección de Temas Adicionales

La serialización de objetos es un recurso simple y poderoso para guardar y recuperar grandes cantidades de información de diversa naturaleza de manera muy fácil. Aunque es una técnica relativamente lenta comparada con el uso de por ejemplo Archivos de Acceso Aleatorio o Secuencial es con mucho la más simple, en éste capítulo vamos a estudiar un ejemplo de serialización mediante el uso de Vectores, no es el más óptimo ni el más eficiente pero ilustra su funcionamiento

¿Qué es la serialización objetos?

La idea es muy simple, Java es un lenguaje Orientado a Objetos de modo que podemos crear clases que representen tales objetos de manera muy rápida, en el ejemplo con Archivos de Acceso Aleatorio los datos guardados eran principalmente primitivos y debíamos implementar diversas clases tanto como para acceder a ellos como para guardarlos, como en Java el elemento principal son Objetos pues que mejor que guardar objetos completos, la serialización permite guardar o transmitir objetos completos con todas sus variables (caracteristicas) y métodos (comportamiento) Así por ejemplo, si tenemos la clase 'cliente' y tal clase tiene por variables: nombre, dirección, teléfono, etc podemos guardar tantos clientes como queramos cada uno con sus datos específicos sin necesidad de guardar dato por dato, todo se guarda de una sola vez. Precisamente el ejemplo de ésta sección es una pequeña aplicación que guarda objetos de tipo cliente en un Vector, otra ventaja de la serialización además de ahorrarnos tiempo y facilitarnos el trabajo es que practicamente cualquier objeto puede ser serializado, sea un arreglo, un vector, una lista, etc Por si todo lo anterior fuera poco la interfaz serializable no contiene métodos, asi que solo bastara colocar al principio de nuestra clase implements Serializable y listo, nuestra clase puede ser serializada.

Ejemplo de Serialización - Pequeño Administrador de Clientes

Ahora veamos un ejemplo, aunque es pequeño es ya sumamente util Primero veamos el problema a grandes rasgos:

Crear un programa que administre clientes, cada cliente tiene como datos: nombre, dirección, número Telefónico y número de fax. El programa debe permitir altas, modificaciones y bajas de clientes. Los clientes deben ser visibles en todo momento

Este programa es sencillo, usaremos Vectores porque es lo más simple para administrar clientes aunque como lo he dicho quizás no sea lo más eficiente en cuestión de tiempo del procesador, pero bueno... que mas da, en estos tiempos nuestras computadoras son más poderosas de lo que necesitamos asi que vamos a explotarlas, aún asi muchos programadores profesionales aún usan Vectores (Por cierto, una técnica mas eficiente seria usar un arreglo de clientes o un ArrayList) Bueno, el siguiente paso es crear una clase 'cliente', en tal clase implementaremos la interfaz serializable y definiremos las variables pertinentes para almacenar los datos, observen lo simple que es hacer nuestro cliente 'serializable', también es importante incluir java.io.* para poder serializar

```
import java.io.*;

public class cliente implements Serializable {

/**
*Los datos generales del cliente
*/
```

```
private String nombre = null, domicilio = null;
private int telefono, fax;

/**
*Clase cliente
*@author JPP
*@version 1.0 Septiembre 2002
*/

/**
*Constructor de la clase cliente
*@param nombre El nombre del Cliente
*@param numero El número de cliente
*/
public cliente(String nombre, String domicilio,
               int telefono, int fax) {
  this.nombre = nombre;
  this.domicilio = domicilio;
  this.telefono = telefono;
  this.fax = fax;
}

/**
*Obtiene el nombre del cliente
*@return Una cadena con el nombre
*/
public String getNombre() {
 return nombre;
}

/**
*Obtiene la direccion del cliente
*@return Una cadena con la dirección
*/
public String getDirec() {
 return domicilio;
}

/**
*Obtiene el número de Teléfono del cliente
*@return Un entero con el numero
*/
public int getTel() {
 return telefono;
}

/**
*Obtiene el numero de Fax del cliente
*@return Un entero con el numero
*/
public int getFax() {
 return fax
}

/**
*Obtiene la cadena que representa éste cliente
*@return Una cadena con la descripción completa del cliente
*/
public String toString() {
 return "Nombre  " + nombre + ", " + domicilio + ", Tel  "
 + telefono + ", Fax  " + fax;
}
```

}

Pseudocódigo del Programa

Ya que tenemos la clase por serializar, ya podemos pensar en cómo crear una interfaz gráfica y un administrador simple para manejar múltiples clientes. Ahora, si la idea es almacenar los datos en un archivo debemos pensar en ciertas posibilidades, al iniciar el programa puede que los datos ya estén presentes en un archivo o bien puede que no existan los datos, pero los datos van a estar en un Vector y ¿cómo se obtienen vectores desde un archivo? ¿cómo se guardan?. Antes de responder lo anterior con detalle vamos a pensar en términos de pseudocódigo, los pasos que deberá realizar el programa son

1 Checar si existen datos guardados
   - Si están guardados, tomarlos y usarlos
   - Si no están guardados, crear una lista de clientes nueva
2 Exhibir los datos en pantalla
3 Teniendo los datos poder realizar.
   - Alta de un nuevo cliente
   - Baja de un cliente existente
   - Modificación de un cliente existente
4. Al terminar el programa guardar todos los datos, tal como están

A grandes rasgos lo anterior es el programa. Ahora vamos a refinar mas el pseudocódigo anterior, para guardar los clientes usaremos Vectores, si consultas la Java API notarás que la clase Vector es serializable de modo que podemos guardarlos fácilmente, tambien como ya declaramos nuestra clase 'cliente' serializable entonces significa que podemos guardar clientes y serializarlos en un Vector sin problemas Tomando en cuenta lo anterior tenemos:

1. Revisar si el archivo de datos existe
   - Si existe, tomar el vector de clientes almacenado en él
   - Si no existe, crear un nuevo vector para administrar clientes
2. Exhibir el contenido del Vector en pantalla, es decir, exhibir cada cliente
3. Teniendo los clientes del Vector poder realizar
   - Alta de cliente, o lo que es lo mismo agregar un nuevo elemento
   - Baja de cliente, en un Vector es tan sencillo como borrar el elemento del indice deseado
   - Modificación de cliente, en un Vector bastara con sobrescribir el elemento del índice deseado
4. Al terminar el programa guardar el Vector de clientes a un archivo "clientes.dat"

Bien, ahora solo resta hacer el programa en Java, el programa principal heredará de Frame (porque es una aplicación, no un applet) también implementará la interfaz ActionListener para manejar los eventos de los botones, en general la Interfaz Gráfica será muy similar a la presente en el ejemplo de Vectores de la sección de Temas Adicionales. Antes de ver el código les diré como se guarda un objeto cualquiera hacia un archivo y cómo se recupera

Guardando un objeto en un archivo

En éste ejemplo guardamos un objeto de tipo Vector nombrado miVector, primero creamos un Flujo de Salida de Objetos (ObjectOutputStream) nombrado salidaDatos, el constructor de tal objeto recibe como argumento un Flujo de Salida de Archivo (FileOutputStream) que define al mismo tiempo en su constructor el nombre del archivo para guardar el vector clientes dat, escribimos el vector en el archivo mediante el metodo writeObject y por ultimo cerramos el flujo de salida de objetos Todo debe ir en una sentencia try-catch que atrapa la excepción IOException (Excepcion de Entrada/Salida) ya que al escribir el objeto puede que el disco duro se encuentre lleno o que no tengamos permiso de escritura, dicho sea de paso no es posible escribir objetos serializados a un archivo desde un applet La sentencia try-catch muestra la excepción en el caso de que se lance.

```
try {
   ObjectOutputStream salidaDatos = new ObjectOutputStream(
   new FileOutputStream("clientes.dat"));
   salidaDatos.writeObject(miVector);
   salidaDatos close();
 } catch(IOException ex) {
   System out println("Error al guardar datos " + ex.toString());
 }
```

- **Recuperando un objeto de un archivo**

Recuperar objetos de un Archivo es similar a guardarlos. Primero creamos un Flujo de Entrada de Objetos (ObjectInputStream) nombrado entradaDatos, como argumento de su constructor pasamos precisamente el archivo de origen de datos mediante un Flujo de Entrada de Archivo (FileInputStream) con el nombre clientes.dat, para obtener el objeto Vector almacenado en ese archivo usamos el método readObject() de la clase ObjectInputStream y aplicamos un cast (Vector) para que el Vector tome su forma original. Tambien se incluyen dos sentencias try-catch, la primera se encarga de atrapar una excepción del tipo IOException (Excepción de Entrada/Salida) que podría lanzar la lectura del objeto y la segunda se encarga de atrapar una excepción del tipo ClassNotFoundException (Excepción por clase no encontrada) que podría lanzar el cast que aplicamos sobre el contenido del archivo, por si nos confundimos y deseamos obtener un objeto que originalmente no guardamos allí.

```
try {
  try {
     ObjectInputStream entradaDatos = new ObjectInputStream(
     new FileInputStream("clientes.dat")),
     miVector = (Vector)(entradaDatos.readObject()),
     entradaDatos close();
     System.out.println("Datos existentes recuperados ");
   } catch (ClassNotFoundException ex) {
     System out.println("Error al recuperar datos " + ex toString()),
  }
 } catch(IOException ex) {
   System out println("Error de entrada de datos " + ex.toString()),
 }
```

Código del Programa Principal

Ahora veamos el programa El código es relativamente extenso, al final mostraremos su explicación.

```
import java io *
import java awt *,
import java awt event *,
import java util Vector;

public class clienteSerializacion extends Frame implements ActionListener {

 Button agregar, eliminar, modificar;
 List listaClientes
 static Vector miVector,
 cliente c

 /**
 *Constructor del Frame principal
 */
 public clienteSerializacion() {
  agregar = new Button("Agregar Cliente")
  eliminar = new Button("Eliminar Cliente"),
  modificar = new Button("Modificar Cliente"),
  listaClientes = new List(),
  listaClientes setMultipleMode(false)
  setLayout(null),

  agregar setBounds(25, 35, 150, 25),
  eliminar setBounds(225  35, 150, 25),
  modificar setBounds(425, 35, 150, 25),
  listaClientes setBounds(25, 75, 550, 200),

  add(agregar),
  add(listaClientes),
  add(eliminar),
  add(modificar),
```

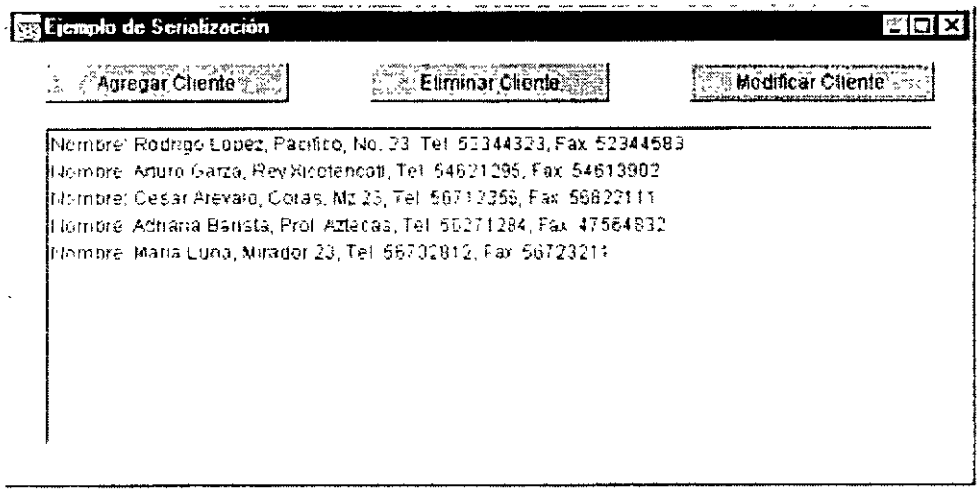
```
 agregar.addActionListener(this);
 eliminar.addActionListener(this);
 modificar addActionListener(this),

 addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
  setVisible(false);
  dispose();
  guardarDatos();
  System exit(0),
 }
 });

 actualizarListaClientes();
}

/**
*Guarda los datos en un archivo
* nombrado clientes.dat
*/
public void guardarDatos() {
 try {
  ObjectOutputStream salidaDatos = new ObjectOutputStream(
  new FileOutputStream("clientes.dat"));
  salidaDatos writeObject(miVector),
  salidaDatos.close();
  System.out.println("Datos guardados."),
 } catch(IOException ex) {
  System.out.println("Error al guardar datos " + ex toString())
 }
}

/*
Método privado que actualiza
la lista de clientes, primero limpia
la lista y después toma los datos nuevos
del contenido del Vector
*/
private void actualizarListaClientes() {
 listaClientes.removeAll():
 if(miVector != null)
   for(int i = 0, i < miVector.size(), ++i)
     listaClientes.add(miVector.elementAt(i) toString()),
}

/*
Método actionPerformed que controla
las acciones al oprimir los botones
*/
public void actionPerformed(ActionEvent e) {
 if(e getSource() == agregar) {
  ventanaCliente vcliente = new ventanaCliente(),
  vcliente.addWindowListener(new WindowAdapter() {
   public void windowClosed(WindowEvent wevt) {
     if(c != null)
       miVector.addElement(c),
      c = null,
     actualizarListaClientes(),
   }
  });
 } else if(e getSource() == eliminar) {
  if(listaClientes getSelectedIndex() != -1)
     miVector removeElementAt(listaClientes.getSelectedIndex()),
```

```
    actualizarListaClientes();
} else if(e.getSource() == modificar) {
 int i = listaClientes.getSelectedIndex();
 if(i != -1) {  //Verificar que se selecciono un elemento
    c = (cliente)(miVector.elementAt(i));
    ventanaCliente vcliente = new ventanaCliente();
    vcliente modifCliente(c);
    vcliente.addWindowListener(new WindowAdapter() {
       public void windowClosed(WindowEvent wevt) {
          if(c != null)
             miVector setElementAt(c, listaClientes.getSelectedIndex());
          c = null;
          actualizarListaClientes();
       }
    });
 }
}
}

/*
Método Main, ésta es una aplicacion,
aquí se inicializa un nuevo clienteSerializacion
*/
public static void main(String args[]) {

 File f = new File("clientes.dat");

 if(f exists()) {
  try {
   try {
     ObjectInputStream entradaDatos = new ObjectInputStream(
     new FileInputStream("clientes dat"));
     miVector = (Vector)(entradaDatos readObject());
     entradaDatos close();
     System out println("Datos existentes recuperados ");
   } catch (ClassNotFoundException ex) {
     System out println("Error al recuperar datos " + ex toString());
   }
  } catch(IOException ex) {
   System out println("Error de entrada de datos " + ex toString());
  }
 } else {
  miVector = new Vector();
  System out println("No existian datos, creando nuevos ");
 }

 clienteSerializacion e = new clienteSerializacion()
 e setTitle("Ejemplo de Serializacion");
 e setBounds(100  150, 600, 300);
 e setVisible(true)

}

/*
Aqui se declara la clase interna o innerclass
ventanaCliente  sirve para agregar y modificar clientes
*/
public class ventanaCliente extends Frame implements ActionListener {

 Button aceptar;
 Label nombre, direc, tel, fax;
 TextField tnombre, tdirec  ttel, tfax;
```

```
//Constructor de ventanaCliente
public ventanaCliente() {
 super("Nuevo Cliente");
 setLayout(new GridLayout(5, 2));

 nombre = new Label("Nombre:");
 direc = new Label("Dirección:");
 tel = new Label("Teléfono:");
 fax = new Label("Fax:");
 tnombre = new TextField();
 tdirec = new TextField();
 ttel = new TextField();
 tfax = new TextField();
 aceptar = new Button("Aceptar");

 add(nombre);
 add(tnombre);
 add(direc);
 add(tdirec);
 add(tel);
 add(ttel);
 add(fax);
 add(tfax);
 add(aceptar);

 aceptar.addActionListener(this);

 addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    setVisible(false);
    dispose();
 }
});

 setBounds(200, 200, 350, 150);
 setVisible(true);
}

/**
*Fija los datos de un cliente para
* que pueda ser modificado
*/
public void modifCliente(cliente clie) {
 tnombre setText(clie getNombre());
 tdirec.setText(clie.getDirec());
 ttel.setText(String valueOf(clie.getTel()));
 tfax setText(String valueOf(clie.getFax()));
}

/*
Control de las acciones del botón,
al agregar un nuevo cliente
*/
public void actionPerformed(ActionEvent e) {
 if(e getSource() == aceptar && tnombre getText() != null
 && tdirec getText() != null && ttel getText() != null
 && tfax.getText() != null) {
     c = new cliente(tnombre getText(), tdirec getText(),
     Integer.parseInt(ttel getText()),
     Integer parseInt(tfax getText()));
     setVisible(false);
     dispose();
```

```
  }
 }

 }
}
```

Salida del programa (después de haber agregado 5 clientes), el link para bajar el código completo de las dos clases se encuentra hasta abajo:



Explicación del Código del Programa Principal

Primero importamos los paquetes necesarios, la interfaz gráfica y el manejo de eventos requieren del java awt *, java awt event *, mientras que el paquete que contiene la clase Vector se encuentra en java.util y el que tiene las clases de Flujos de entrada/salida es el java io.*,

```
import java io *,
import java awt *,
import java awt.event *,
import java util Vector,
```

Ahora declaramos nuestra clase heredando de Frame e implementando un ActionListener, tambien declaramos las variables globales, en éste caso se trata de algunos elementos graficos como los tres botones y la lista de clientes, tambien declaramos un Vector para almacenar los clientes y un cliente temporal nombrado 'c' para los movimientos y modificaciones, el Vector debe ser declarado como static para que incluso las innerclases o clases internas que usaremos puedan tener acceso para manipularlo

```
public class clienteSerializacion extends Frame implements ActionListener {

 Button agregar, eliminar, modificar,
 List listaClientes,
 static Vector miVector,
 cliente c
```

El constructor inicializa los elementos de la Interfaz Grafica y los posiciona de acuerdo a las coordenadas mediante setBounds(), por esta razon quitamos el Layout predeterminado, también fijamos la Lista en modo simple mediante setMultipleMode(false), con esto garantizamos que solo un cliente sera elegible a la vez. Al final registramos los listeners de los botones y el encargado de cerrar la ventana, en tal sección de codigo llamamos al metodo guardarDatos() para almacenar todos los objetos precisamente al cerrar la aplicación Por último actualizamos la lista de clientes mediante actualizarListaClientes(), para exhibir los clientes existentes, si es que los hay

```
 /**
 *Constructor del Frame principal
 */
 public clienteSerializacion() {
```

```
  agregar = new Button("Agregar Cliente"),
  eliminar = new Button("Eliminar Cliente"),
  modificar = new Button("Modificar Cliente");
  listaClientes = new List();
  listaClientes setMultipleMode(false);
  setLayout(null),

  agregar.setBounds(25, 35, 150, 25),
  eliminar setBounds(225, 35, 150, 25);
  modificar.setBounds(425, 35, 150, 25),
  listaClientes setBounds(25, 75, 550, 200);

  add(agregar),
  add(listaClientes);
  add(eliminar),
  add(modificar),

  agregar.addActionListener(this),
  eliminar.addActionListener(this);
  modificar.addActionListener(this);

  addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
   setVisible(false);
   dispose();
   guardarDatos(),
   System exit(0),
  }
  }),

  actualizarListaClientes(),
}

/**
*Guarda los datos en un archivo
* nombrado clientes dat
*/
public void guardarDatos() {
 try {
   ObjectOutputStream salidaDatos = new ObjectOutputStream(
   new FileOutputStream("clientes dat"))
   salidaDatos writeObject(miVector)
   salidaDatos close(),
   System out println("Datos guardados "),
 } catch(IOException ex) {
   System out println("Error al guardar datos " + ex toString()),
 }
}
```

El metodo anterior nombrado guardarDatos() se encarga de crear el flujo de salida que almacenara el vector en el archivo 'clientes dat', esta pelicula ya la vimos El metodo siguiente (de abajo) es el encargado de actualizar el contenido de la lista de acuerdo al contenido del Vector Como el metodo add() de la clase List solo agrega elementos String (pero no los quita) debemos limpiar la lista en cada actualizacion mediante removeAll(). Checar la condicion miVector != null previene una consulta en un Vector inexistente, sin ésta comprobación provocariamos una excepcion sobre el Vector (que estaria vacio) tal error no seria grave pero si molesto, en pocas palabras garantiza que no se exhiban datos si estos no existen Accedemos al elemento mediante su indice, el ciclo for garantiza que se exhibiran los datos presentes al checar el tamaño del Vector, se muestra la representación de cadena para cada cliente mediante toString(), recuerden que toString() es el único método accesible de manera directa (sin necesidad de hacer una promocion) porque el de la clase cliente sobrescribe el de la clase Object

```
/*
Metodo privado que actualiza
la lista de clientes, primero limpia
```

```
la lista y después toma los datos nuevos
del contenido del Vector
*/
private void actualizarListaClientes() {
 listaClientes.removeAll();
  if(miVector != null)
    for(int i = 0, i < miVector.size(); ++i)
      listaClientes.add(miVector.elementAt(i).toString());
}
```

Ahora viene lo relativamente complicado Dentro del siguiente método vienen las acciones a realizar para cada uno de los tres botones agregar, eliminar y modificar cliente Para agregar un cliente se abre una ventana nueva definida en la innerclass (mostrada más abajo) nombrada como vcliente (tipo ventanaCliente), se trata simplemente de una ventana que obtiene los datos del cliente, en tal ventanita registramos un windowListener que 'pondrá atención' al momento de que la ventanita se cierre, cuando suceda agregará un cliente (si es que es válida o lo que es lo mismo, desigual de null) y actualizará la lista de clientes para mostrar los cambios realizados en el contenido del Vector Para eliminar un cliente simplemente se verifica si la lista tiene al menos un elemento seleccionado (desigual de -1) y si es así se elimina ese elemento mediante el método removeElementAt() de la clase Vector, recuerden que el índice de la Lista y el índice del Vector inicia en cero, son iguales, por eso es posible hacer esto

```
/*
Método actionPerformed que controla
las acciones al oprimir los botones
*/
public void actionPerformed(ActionEvent e) {
 if(e.getSource() == agregar) {
  ventanaCliente vcliente = new ventanaCliente(),
  vcliente.addWindowListener(new WindowAdapter() {
   public void windowClosed(WindowEvent wevt) {
     if(c != null)
       miVector addElement(c);
     c = null,
     actualizarListaClientes(),
   }
  }),
 } else if(e getSource() == eliminar) {
  if(listaClientes getSelectedIndex() != -1)
      miVector removeElementAt(listaClientes getSelectedIndex()),
      actualizarListaClientes();
 } else if(e getSource() == modificar) {
  int i = listaClientes getSelectedIndex(),
  if(i != -1) { //Verificar que se selecciono un elemento
      c = (cliente)(miVector.elementAt(i)),
      ventanaCliente vcliente = new ventanaCliente()
      vcliente modifCliente(c)
      vcliente addWindowListener(new WindowAdapter() {
         public void windowClosed(WindowEvent wevt) {
            if(c != null)
               miVector setElementAt(c, listaClientes getSelectedIndex()),
            c = null,
            actualizarListaClientes(),
         }
       });
  }
 }
}
```

En la parte final del metodo anterior tenemos la accion de modificar un cliente existente La comprobación se hace nuevamente si por lo menos un elemento de la lista se encuentra seleccionado (distinto de -1, mediante la variable entera 'i') despues tomamos el cliente elegido del Vector usando el índice (tomado de la lista) mediante un cast y lo pasamos a la variable global cliente nombrada 'c' Posteriormente abrimos una ventanita para editar al cliente recien tomado del Vector y fijamos sus datos mediante el llamado a la función modifCliente(c); Finalmente

registramos un WindowListener que será el encargado de actualizar el cambio realizado al cliente y actualizar la lista, nótese que todas las modificaciones del cliente quedan a cargo de la ventanita 'vcliente', ésta sección del programa le deja todo el trabajo y no hace nada más que pasarle el cliente, más adelante veremos cómo la ventanita se encarga de modificarlo.

El método siguiente es el que inicia la aplicación. El método main contiene primero una variable File que se encargará de comprobar si el archivo 'clientes.dat' existe o no realmente. Es importante tener presente que un File existe para obtener datos de un archivo los cuales incluyen su existencia, no se trata de un archivo sino de un medio para obtener información de un archivo. Así, si el archivo nombrado 'f' existe se toma el Vector que tiene almacenado tal como se explicó anteriormente, si el archivo 'f' no existe pues mucho menos el Vector asi que se crea uno nuevo.

```
/*
Método Main, ésta es una aplicación,
aqui se inicializa un nuevo clienteSerializacion
*/
public static void main(String args[]) {

 File f = new File("clientes.dat"),

 if(f.exists()) {
  try {
   try {
      ObjectInputStream entradaDatos = new ObjectInputStream(
      new FileInputStream("clientes dat")),
      miVector = (Vector)(entradaDatos readObject()),
      entradaDatos close(),
      System out.println("Datos existentes recuperados "),
    } catch (ClassNotFoundException ex) {
     System out println("Error al recuperar datos " + ex toString()),
   }
  } catch(IOException ex) {
   System out println("Error de entrada de datos " + ex.toString()),
  }
 } else {
  miVector = new Vector(),
  System out println("No existian datos, creando nuevos "),
 }

 clienteSerializacion e = new clienteSerializacion(),
 e setTitle("Ejemplo de Serializacion"),
 e setBounds(100, 150, 600, 300)
 e setVisible(true),

}
```

En la parte final del metodo main se inicializa la ventana del programa principal, también se especifican algunas propiedades como titulo y posicionamiento A continuación y dentro de la misma clase anterior se declara una clase interna que definirá las propiedades de la ventanita encargada tanto de agregar como de modificar a los clientes Sus propiedades son similares a la ventana principal, tambien hereda de Frame e implementa un ActionListener encargado de manejar los eventos de sus propios botones.

```
/*
Aqui se declara la clase interna o innerclass
ventanaCliente, sirve para agregar y modificar clientes
*/
public class ventanaCliente extends Frame implements ActionListener {
```

Declaramos algunas variables graficas tales como Botones, Etiquetas y Campos de Texto, el constructor también especifica ciertas caracteristicas de la ventana, por ejemplo un GirdLayout() (acomodamiento de componentes en cuadricula o reticula) de 5 lineas y 2 columnas, después de inicializar los componentes los agregamos en orden de acomodamiento preciso para que se muestren apropiadamente, si desean más información revisen la sección de Interfaz Gráfica

```
Button aceptar;
Label nombre, direc, tel, fax;
TextField tnombre, tdirec, ttel, tfax;

//Constructor de ventanaCliente
public ventanaCliente() {
 super("Nuevo Cliente");
 setLayout(new GridLayout(5, 2));

 nombre = new Label("Nombre:");
 direc = new Label("Dirección.");
 tel = new Label("Teléfono ");
 fax = new Label("Fax:");
 tnombre = new TextField();
 tdirec = new TextField();
 ttel = new TextField();
 tfax = new TextField();
 aceptar = new Button("Aceptar");

 add(nombre);
 add(tnombre);
 add(direc);
 add(tdirec);
 add(tel);
 add(ttel);
 add(fax);
 add(tfax);
 add(aceptar);

 aceptar addActionListener(this);

 addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    setVisible(false)
    dispose();
 }
 });

 setBounds(200, 200, 350, 150);
 setVisible(true);
}
```

En la parte final del constructor anterior se registra un listener sobre el único botón existente, tambien se registra un WindowListener encargado de cerrar la ventana y se especifica el posicionamiento de este Frame El siguiente método toma los datos de un cliente y los fija en todos los campos de texto, éste método se usara para modificar el cliente.

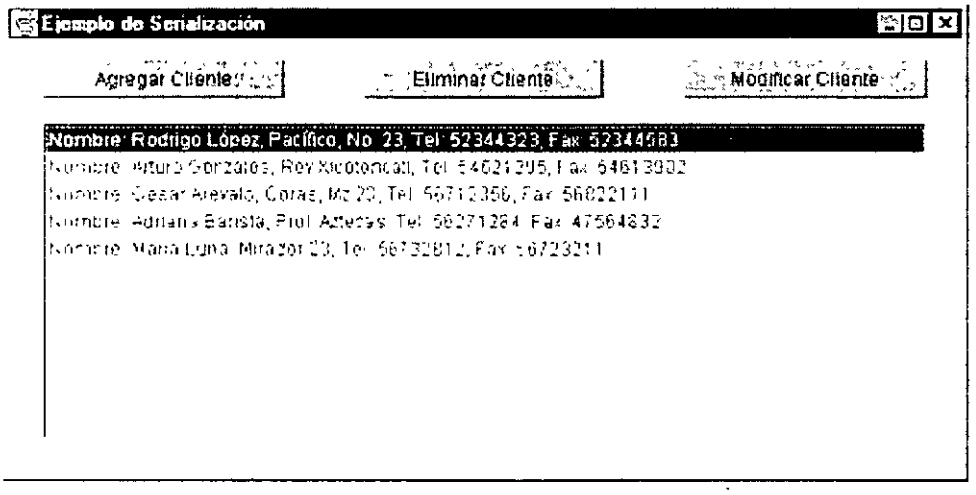
```
/**
*Fija los datos de un cliente para
* que pueda ser modificado
*/
public void modifCliente(cliente clie) {
 tnombre setText(clie getNombre())
 tdirec setText(clie getDirec())
 ttel.setText(String valueOf(clie getTel()));
 tfax setText(String valueOf(clie getFax()));
}
```

Por último pero no menos importante tenemos la acción del botón de la ventanita para agregar y modificar clientes, esta es la seccion de código que se encarga de crear los nuevos clientes para agregarlos como nuevos o bien para reemplazar a uno existente mediante modificación. Simplemente usa la variable global estatica 'cliente c' llamando a su constructor y tomando los datos de los campos de texto dependiendo de su tipo, ya el programa principal se encargará como ya hemos visto de guardarlo como nuevo cliente o bien de actualizar un cliente

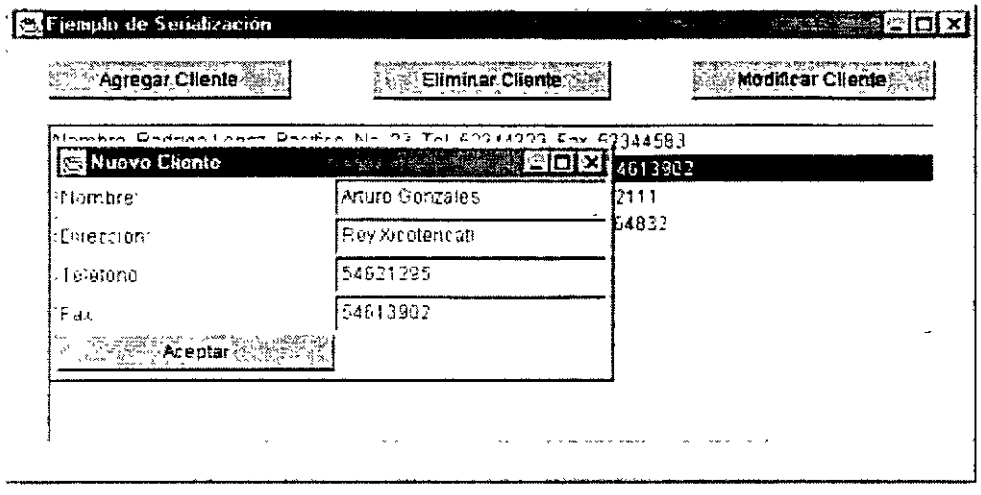
existente (ésta división del trabajo simplifica mucho las cosas). Como ven la variable cliente 'c' debe ser estática para que ésta innerclass pueda modificarla a su gusto, después de oprimir aceptar se oculta la ventanita

```
 /*
 Control de las acciones del botón,
 al agregar un nuevo cliente
 */
 public void actionPerformed(ActionEvent e) {
  if(e.getSource() == aceptar && tnombre.getText() != null
  && tdirec.getText() != null && ttel.getText() != null
  && tfax.getText() != null) {
      c = new cliente(tnombre.getText(), tdirec.getText(),
      Integer.parseInt(ttel.getText()),
      Integer.parseInt(tfax.getText())),
      setVisible(false);
      dispose();
  }
 }

 }
}
```

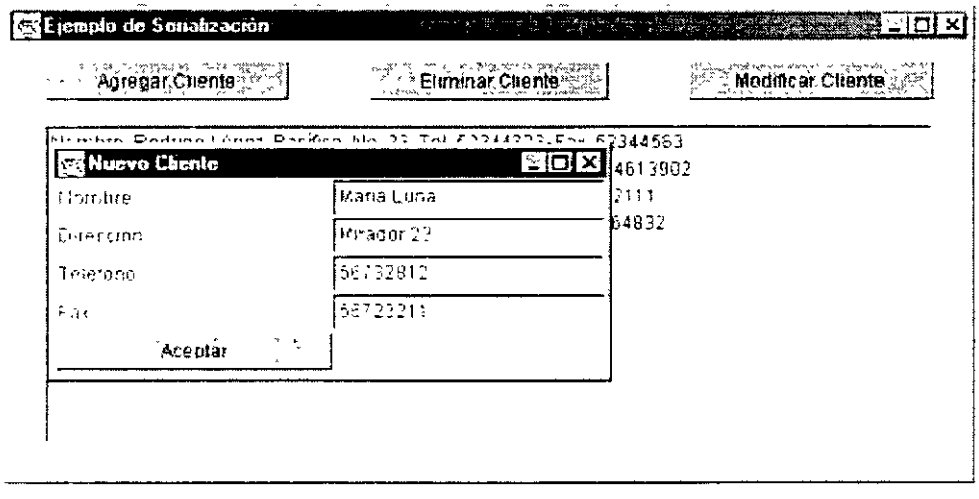
Mediante un programa de éste tipo podemos agregar y administrar diversos clientes con sus respectivos datos, incluso podemos modificar posteriormente las propiedades de un cliente para agregarle más características pero evidentemente tendríamos que modificar la ventanita de alta/modificación. (el link para bajar el código se encuentra en la parte de abajo)



Alta de un Cliente



Modificación de un Cliente Existente



Y por supuesto, si cerramos la aplicacion con clientes agregados, al abrirla estarán presentes El código de las dos clases se encuentra en el siguiente archivo

Jserializacion zip

Temas Adicionales

- Interfaces
- Excepciones
- Vectores
- JavaDoc y RealJ

Esta sección esta reservada para temas que por ser relativamente sencillos no requieren de una explicación extensa Las interfaces son un medio de comunicación muy eficaz que nos permite obtener ciertas propiedades sin la necesidad de heredar, en este curso las utilizo para el manejo de eventos y para la animación mediante la interfaz Runnable. Las excepciones sirven para manejar casos fuera de lo común que pueden llegar a suceder durante la ejecución de un programa que generalmente serán errores, si no manejamos estos casos a veces puede ser que nuestro programa inevitablemente se detenga.

Interfaces

Una interfaz tal como su nombre lo dice es una especie de puente entre dos o más clases, un puente de métodos Por ejemplo, supongan que a ustedes que son chicos les agrada mucho una chica X y claro se mueren de ganas por un beso. ¿Qué condición podría poner esta chica X para el beso? posiblemente, pon la mejilla. Entonces una interfaz le exige a cualquiera que la implemente algo a cambio: que defina los metodos que dicha interfaz tiene, por eso las interfaces solo tienen **cuerpos de métodos sin definir** o sea que realmente no tienen métodos propiamente dichos, si consideramos el ejemplo del beso podríamos hacer la interfaz

```
interface beso {
   void ponLaMejilla();
}
```

Entonces, todos las clases "chico" que deseen un beso deberán primero, implementar la clase y segundo: definir los métodos que ella tiene (las condiciones), vamos a implementar la case anterior, observen:

```
public class chicoEnamorado implements beso {

   ponLaMejilla() {
      //Aqui cada chico pone su reacción personal
   }

}
```

Simple, ¿verdad? Las interfaces pueden tener muchos metodos de varios tipos, en algunas ocasiones como en la interfaz Runnable el metodo a implementar será solo uno run() y en otras ocasiones serán varios como por ejemplo en la interfaz MouseListener

```
void mouseClicked(MouseEvent e);
void mouseEntered(MouseEvent e);
void mouseExited(MouseEvent e);
void mousePressed(MouseEvent e);
void mouseReleased(MouseEvent e);
```

Entonces esto es algo así como un prestamo, las interfaces no son las que prestan, son solo el medio de comunicacion La interfaz beso que hicimos es practicamente inútil, o sea realmente no hace nada Para que fuera útil debería heredar de alguna otra clase (que es la que realmente hace el prestamo) que contenga lo que las clases que quieren implementarla desean, por ejemplo el caso de la interfaz Runnable, esta interfaz hereda de Thread por lo tanto hereda el comportamiento de hilo y muy especialmente el método run(), toda applet que desee utilizar un hilo recurre entonces a esta interfaz y **como requisito** debe implementar su propia versión del método run(), un hilo sin metodo run() es inútil por eso si lo quiere esta obligada a implementarlo

Es posible que una clase implemente varias interfaces al mismo tiempo, pero deberá definir los métodos de cada una, la sintaxis para implementar más de una interfaz es:

```
public class chicoEnamorado implements beso, cariñitos, boda {

  //Y aqui define todos los métodos que exige cada interfaz, por ejemplo:

  void ponLaMejilla() {
  }

  void comprameUnHelado() {
  }

  void comparmeUnCoche() {
  }

  void ponmeCasa() {
  }

}
```

Este seria el caso claro de una chica bastante interesada y exigente

Excepciones

Las excepciones son casos anormales que surgen en ciertas secuencias de código durante la ejecución, el compilador Java puede revisar la coherencia de los tipos, las funciones, los operadores y la sintaxis general pero definitivamente no puede checar todas las posibilidades que casi siempre también incluyen algún error. Tanto el compilador de Java como su intérprete pueden generar excepciones. Cuando se da una excepción en tiempo de ejecución el intérprete Java se ocupa de "lanzarla" y si el programador no se encargó de "atraparla" entonces el programa se detiene. Una excepción es lanzada cuando: 1) El intérprete Java la genera en base a su lista predeterminada de excepciones (objetos tipo Exception) ó 2) cuando creamos nuestras propias excepciones que pueden llegar a ocurrir y nosotros las lanzamos explícitamente bajo ciertas condiciones Atrapar una excepción sería el equivalente de tratar el problema para su corrección Este tema es muy sencillo porque siempre que se genere una excepción podemos hacer una de dos cosas:

1 No atraparla
2 Atraparla y Manejarla

1. Excepciones no atrapadas

Este es un ejemplo de una aplicación sencilla que con toda intención genera una excepción aritmética y una vez generada ni se preocupa por ella, es decir: no la atrapada

```
import java.io.*;

public class divisionCero {

  public static void main( String args[] ) {
     int a = 0;
     int b = 5;
     int resultado = b / a;
     System.out.println( "Resultado = " + resultado );
  }

}
```

La salida de este programa (desde la linea de comandos) es

```
D:\cursoJava\codigoFuente>java divisionCero
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at divisionCero.main(divisionCero.java:8)
```

El programa anterior se compila correctamente, sin embargo al ejecutarlo ocurre un problema. De matemáticas sabemos que un intento de division entre cero siempre esta prohibido porque es una indeterminación, en este caso el interprete java se encargó automaticamente de generar y lanzar la excepción en tiempo de ejecución (con

su lista de Excepciones predeterminadas), el programa se detiene a partir del punto en el que se generó el problema, por eso no aparece el resultado de la división, después el intérprete muestra algo llamado "trayecto de pila" que es algo así como la trayectoria que siguió la excepción desde que se lanzó, es decir los métodos o las clases que fue recorriendo desde que se originó, primero se muestra el tipo de excepción: ArithmeticException y después muestra el problema / by zero ó intento de división entre cero, finalmente señala el lugar exacto en donde se generó la excepción en la clase divisionCero en su método main y en la línea 8 específicamente En este caso la excepción Aritmética ya esta creada de manera predeterminada por eso al menos se lanza apropiadamente, sin embargo este ejemplo no la captura (o sea, que solo se exhibe pero que realmente no se corrige el problema, solo provoca que se muestre), el intérprete Java tiene muchísimas excepciones que puede lanzar si se encuentra algún problema, sus nombres generalmente estarán muy de acuerdo con la dificultad hallada como en este caso· una excepción aritmética, puedes buscar y consultar muchas excepciones y su descripción respectiva en la Java API.

2. Excepciones atrapadas y manejadas

Como ya dije una cosa es generar una excepción y otra es capturarla, esto último implica que corregiremos el problema, la sentencia try-catch sirve para esto, la sintaxis de un try-catch es la siguiente:

```
try {
    //Sección de código a intentar ejecutar y que posiblemente podría generar una excepción
} catch ( TipoDeExcepcion problemita ) {
  //Atrapar la excepción y tomar las medidas necesarias para corregir el problema
}
```

Nota: try viene de "intentar" o "probar" y **catch** de "atrapar" o "capturar"

Entonces dentro del try se engloba la sección de código que podría generar la excepción y el catch se encarga de atraparla (la identifica y si coincide con el nombre de excepción, la atrapa) y además para intentar corregirla o en el peor de los casos dejar las variables y objetos involucrados en un estado estable para continuar la ejecucion como si nada hubiera sucedido En el siguiente ejemplo atrapamos y manejamos la excepción del problema anterior

```
import java io *;

public class divisionCeroA {

   public static void main( String args[] ) {

   int a = 0;
   int b = 5;

    try {
     int resultado = b / a;  //Aqui podria lanzarse una excepción
     System out println( "Resultado = " + resultado );
    } catch (ArithmeticException e) {        //Aqui la atrapamos y corregimos en problema en caso de que se lance
     System out println("Intento de dividir entre cero, el denominador se resetea a 1 por seguridad");
     a = 1;
    }

   System out println("Programa Terminado");
   }

}
```

La salida de este programa es·

```
D:\cursoJava\codigoFuente>java divisionCeroA
Intento de dividir entre cero, el denominador se resetea a 1 por seguridad
Programa Terminado
```

Como pueden ver el hecho de que se haya exhibido el mensaje de "Programa Terminado" significa que el programa no se detuvo prematuramente como antes gracias a que atrapamos y manejamos la excepción. En este curso los únicos temas que recurren al try-catch son la animación (con hilos) y el manejo de archivos (de acceso aleatorio), en ambos casos únicamente mis manejos de excepciones son para exhibir el error ocurrido aunque

bien podría tomar medidas para corregirlos pero sería meterme a cuestiones del sistema aún desconocidas para mí. Crear nuestras propias definiciones de excepciones es bastante sencillo pero incluso he leído a bastantes autores que recomiendan manejar las excepciones con estructuras de control (sin necesidad de crearlas), ambos caminos son fáciles de seguir depende de ti elegir. Si estas interesado en aprender a crear tus propias excepciones es porque tienes un proyecto bastante grande en mente lo cual generalmente no es el caso de nosotros los principiantes.

Sentencia throws

La sentencia throws tiene un propósito bastante simple, ya quedamos en que es posible hacer dos cosas con una excepción: 1) no atraparla ó 2) atraparla y manejarla La sentencia throws es una especie de modificador para los métodos que indica explícitamente que no atraparemos la excepción sino que la relanzaremos nuevamente, en pocas palabras cualquier método con la sentencia throws estaría conciente de que puede lanzar una excepción pero no se hace responsable de manejarla. Así que, cualquiera que utilice un metodo que posee la sentencia throws deberá obligatoriamente: atrapar la excepción y manejarla con un try-catch o bien relanzar nuevamente la excepción. Esto significa que siempre en algún momento acabaremos por atrapar las excepciones, ya sea en funciones internas o en funciones externas al programa, recuerdo una comparación muy ilustrativa que encontré en un libro:

'Una excepción es como una papa caliente que se pasa entre métodos y clases, pero que finalmente alguien sin importar quién sea deberá tomar la responsabilidad de hacerse cargo de ella'

Y vaya que es verdad, todo método con la cláusula throws te dice 'yo soy capaz de lanzar una excepción, más vale que o bien la manejes con un try-catch o bien la re-lances con otro throws'. Si no haces ninguna de las dos cosas el compilador Java no te permitirá compilar tu programa Generalmente un error de este tipo aparece algo así como

Exception 'cualquier tipo' must be caught (la excepción debe ser atrapada)

Así, por ejemplo, en la sección de Archivos Aleatorios de este curso algunos métodos poseen:

```
private void escribirEntero(int entero) throws IOException {
    archivo writeInt(entero);
}
```

Esto indica que la posible excepción generada al escribir un entero **no** se maneja en este metodo pero cualquier método externo que llame a este metodo deberá: o bien manejarla o relanzarla nuevamente Si estudian el codigo verán que todas las posibles excepciones se manejan en algún momento dentro del programa ejemplo para las cuentas todas esas excepciones tienen su origen en la clase RandomAccessFile tal como se puede ver en la Java API muchos de sus metodos poseen **throws IOException**, lo que indica que pueden lanzar una excepcion de tipo Entrada/Salida (In/Out) de datos Bueno, solo restaría exponer el tema para tu puedas crear tus propios tipos de excepciones pero ese tema es muy sencillo y tal como dije anteriormente no es indispensable para un principiante ya que es para crear programas mucho más robustos y estables, sin embargo es mucho mas fácil y aconsejable manejar los errores con estructuras de control que 'chequen' la validez de nuestros datos en vez de crear docenas de Excepciones que bien pueden complicarnos mucho la vida Un programa que verdaderamente requiera de la creacion de nuestras propias excepciones contaría con no menos de 100 clases distintas, considerando claro algunas pocas **excepciones.**

Vectores

Los vectores son una herramienta útil para diversas aplicaciones, sirven para almacenar objetos de cualquier tipo (o sea que cualquier clase que hayamos creado podrá guardarse a un vector) tal como si fuera un arreglo pero solo de objetos. Entonces se utilizan mucho como verdaderos administradores de objetos porque fácilmente se pueden agregar elementos, quitar elementos, consultar elementos, sobrescribirlos (para agregar uno actualizado donde ya existía anteriormente otro) entre otras funciones. El constructor de un vector tiene la siguiente forma:

```
Vector cualquierVector = new Vector();
```

Bastante simple ¿verdad?, tal constructor no recibe ningún argumento porque otra de las ventajas de los Vectores es que su tamaño es dinámico, es decir que crecerá o disminuirá de acuerdo a los elementos que agreguemos o quitemos de su contenido, así un vector recién creado tendrá un tamaño igual a cero y si agregamos un elemento (u objeto) su tamaño crecerá a uno, si agregamos otro elemento su tamaño seria igual a dos y así sucesivamente. Los vectores tienen muchos métodos útiles para manipularlos, todos se pueden ver con detalle en la Java API específicamente en la clase 'Vector', sin embargo los más útiles son.

Método	Acción
size();	Regresa el tamaño del Vector (entero)
add(Object o);	Agrega un objeto cualquiera al final del Vector
add(int index, Object o);	Agrega un objeto cualquiera en el índice especificado
elementAt(int index);	Obtiene el objeto en el índice especificado, si el índice es más grande que el tamaño del vector se lanza una excepción. Este método es para extraer objetos almacenados
insertElementAt(Object o, int index);	Hace exactamente lo mismo que el otro add()
removeElementAt(int index);	Quita el elemento en el índice 'index' y recorre todos los elementos restantes un lugar hacia abajo (disminuye el tamaño del vector automáticamente)

Los elementos se agregan directamente sin embargo para recuperar un objeto guardado de un vector debemos aplicar algo llamado 'casting' o 'promocion', todo vector guarda objetos de cualquier tipo así que no te pone ningún 'pero' para guardarlos pero al momento de recuperarlos te los regresa como objetos generales, es decir, que no tienen la misma forma que cuando los guardaste, no porque se hayan alterado sino porque se guardaron como objetos genéricos, así si quieres recuperar un objeto de tipo 'pelota' de un Vector debes promocionarlo a su estado de pelota original, por ejemplo

```
Vector librero = new Vector(),   //creamos un vector para almacenar objetos de tipo libro
libro b = new libro(),           //creamos un libro llamado 'b'
librero.add( b ),                //guardamos el libro en el Vector
libro a = (libro)(librero elementAt( 0 )),  //obtenemos el libro guardado en el indice cero.. y.
                                            // .aplicamos un cast para convertirlo a su forma original
```

Del mismo modo si el objeto por recuperar se llama cliente aplicamos (cliente)miVector.elementAt(i); donde i es el índice o posición del Vector Muchos programadores consideran que un vector puede ser una herramienta de bajo rendimiento y en parte es verdad pues el costo de tener un arreglo dinámico se refleja en el tiempo necesario para manipularlo, otra opcion mas rapida seria un arreglo de objetos con la única desventaja de que no podrá crecer ni disminuir dinamicamente Para nosotros los principiantes el rendimiento de entrada no tiene mucha importancia, comprender y saber utilizar un Vector servira para entender en el futuro técnicas mucho más poderosas así que por ahora no nos meteremos con las cuestiones de rendimiento Ahora veamos un ejemplo de como utilizar un Vector, éste ejemplo requiere de algo de conocimientos previos tales como· Aplicaciones, Eventos e Interfaz Gráfica, es un ejemplo un poco extenso pero espero que logren entenderlo sin problemas

Explicación del Programa Ejemplo de Vectores

A grandes rasgos haremos lo siguiente. Primero, vamos a definir una clase 'cliente', ésta clase servirá para disponer de un objeto cualquiera creado por nosotros para almacenarlo en nuestro Vector; Segundo, vamos a crear una pequeña aplicación con una sencilla interfaz gráfica que tenga un vector y una lista, en tal vector

almacenaremos objetos 'clientes' y los mostraremos en pantalla mediante la lista, asi si agregamos un cliente nueva al vector se exhibirán sus datos y si eliminamos un cliente se borrará de la lista. Tercero, para agregar un nuevo cliente necesitamos una pequeña ventanita para introducir sus datos, esa ventanita la incluiremos en el programa principal como una innerclass o clase interna; así unificaremos un poco nuestro código Según todo lo anterior vamos a crear dos clases: La clase 'cliente' y la clase 'ejemploVectores' que será el programa principal.

La siguiente es la clase cliente, como verán es sumamente simple y no requiere de una explicación detallada

```
public class cliente {

private String nombre = null;
private int numero = 0;

/**
*Clase cliente
*@author JPP
*@version 1 0 Agosto 2002
*/

/**
*Constructor de la clase cliente
*@param nombre El nombre del Cliente
*@param numero El número de cliente
*/
public cliente(String nombre, int numero) {
this nombre = nombre;
this numero = numero;
}

/**
*Obtiene el nombre del cliente
*@return Una cadena con el nombre
*/
public String getNombre() {
return nombre.
}

/**
*Obtiene el numero del cliente
*@return Un entero con el número
*/
public int getNumero() {
return numero
}

/**
*Obtiene la cadena que representa éste cliente
*@return Una cadena con la descripcion completa del cliente
*/
public String toString() {
return "Numero  " + numero + ", Nombre  " + nombre.
}

}
```

Los metodos que contiene son simples y agregue algunos comentarios de documentación que serviran al final para demostrar como utilizar JavaDoc, pero ese es un tema aparte de los Vectores Realmente no vamos a usar todos los metodos de la clase 'cliente' mas que toString() Ahora veamos el programa principal, este si requerirá una explicacion con mas detalle que daré al final del codigo

Código del Programa Principal

```
import java awt *,
```

```
import java.awt.event.*;
import java.util.Vector;

public class ejemploVectores extends Frame implements ActionListener {


    Button agregar, eliminar;
    List listaClientes,
    Vector miVector;
    cliente c;

    /*
    Constructor del Frame principal
    */
    public ejemploVectores() {

      agregar = new Button("Agregar Cliente");
      eliminar = new Button("Eliminar Cliente"),
      listaClientes = new List();
      listaClientes.setMultipleMode(false),
      miVector = new Vector();
      setLayout(null);

      agregar.setBounds(25, 35, 150, 25);
      eliminar.setBounds(225, 35, 150, 25);
      listaClientes.setBounds(25, 75, 350, 200),

      add(agregar);
      add(listaClientes);
      add(eliminar);

      agregar.addActionListener(this);
      eliminar addActionListener(this);

      addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            setVisible(false),
            dispose(),
            System exit(0),
          }
      }),
      actualizarListaClientes(),
    }

    /*
    Metodo privado que actualiza
    la lista de clientes, primero limpia
    la lista y despues toma los datos nuevos
    del contenido del Vector
    */
    private void actualizarListaClientes() {
      listaClientes removeAll(),
      for(int i = 0, i < miVector.size(); ++i)
          listaClientes add(miVector.elementAt(i).toString()),
    }

    /*
    Método actionPerformed que controla
    las acciones al oprimir los botones
     */
    public void actionPerformed(ActionEvent e) {
      if(e.getSource() == agregar) {
```

```
        ventanaCliente vcliente = new ventanaCliente();
        vcliente.addWindowListener(new WindowAdapter() {
          public void windowClosed(WindowEvent wevt) {
            if(c != null)
              miVector.addElement(c);
            c = null;
            actualizarListaClientes();
          }
        });
        } else if(e.getSource() == eliminar) {
          if(listaClientes.getSelectedIndex() != -1)
            miVector.removeElementAt(listaClientes.getSelectedIndex());
          actualizarListaClientes();
        }
      }

      /*
      Método Main, ésta es una aplicación,
      aquí se inicializa un nuevo ejemploVectores
      */
      public static void main(String args[]) {
        ejemploVectores e = new ejemploVectores();
        e.setTitle("Ejemplo del uso de Vectores");
        e.setBounds(200, 150, 400, 300);
        e.setVisible(true);
      }

      /*
      Aquí se declara la clase interna o innerclass
      ventanaCliente, sirve para agregar clientes
      */

public class ventanaCliente extends Frame implements ActionListener {

        Button aceptar;
        Label nombre, numero;
        TextField tnombre, tnumero;

      //Constructor de ventanaCliente
    public ventanaCliente() {
      super("Nuevo Cliente");
      setLayout(new GridLayout(3, 2));
      aceptar = new Button("Aceptar");
      nombre = new Label("Nombre:");
      numero = new Label("Número");
      tnombre = new TextField();
      tnumero = new TextField();
      add(nombre);
      add(tnombre);
      add(numero);
      add(tnumero);
      add(aceptar);

      aceptar.addActionListener(this);

      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          setVisible(false);
          dispose();
        }
      });
```

```
    setBounds(280, 250, 250, 100);
    setVisible(true);
  }

  /*
  Control de las acciones del botón,
  al agregar un nuevo cliente
  */
  public void actionPerformed(ActionEvent e) {
      if(e.getSource() == aceptar && tnombre.getText() != null
      && tnumero.getText() != null) {
        c = new cliente(tnombre.getText(),
        Integer.parseInt(tnumero getText()));
        setVisible(false);
        dispose();
      }
  }

  }
}
```

Explicación del código del Programa Principal

Bueno, primero importamos los paquetes de clases necesarios tanto para una aplicación gráfica como para los eventos, también importamos el paquete java.util Vector; que es el que evidentemente contiene la clase Vector, desde ahora siempre que deseemos usar un Vector debemos importar éste paquete:

```
import java.awt.*;
import java.awt.event *,
import java.util Vector;
```

Ahora nombramos nuestra clase como será una ventana heredamos de Frame e implementamos la interfaz ActionListener para manipular los botones

```
public class ejemploVectores extends Frame implements ActionListener {
```

Como siguiente paso declaramos las variables globales del programa, en éste caso dos botones, uno para agregar clientes y el otro para eliminarlos del Vector, una lista que mostrará en todo momento el contenido del Vector, un Vector que almacenará los clientes y un cliente temporal nombrado 'c' que servirá para definir cada cliente nuevo por almacenar o en otras palabras: el cliente 'c' después de creado se almacenará en el Vector y posteriormente sera borrado para agregar otro cliente 'c', pero cada cliente tendrá sus caracteristicas propias unicas como veremos más adelante

```
  Button agregar, eliminar,
  List listaClientes,
  Vector miVector,
  cliente c,
```

Ahora tenemos el constructor del programa principal En éste constructor se inicializan los botones y la lista, el metodo setMultipleMode(false) indica que solo podra elegirse un elemento de la lista a la vez, esto porque al borrar un cliente como veremos más adelante el indice del elemento por borrar se tomara de la Lista, seria un poco mas complicado borrar mas de un elemento u objeto a la vez

```
  /*
  Constructor del Frame principal
  */
  public ejemploVectores() {

    agregar = new Button("Agregar Cliente")
    eliminar = new Button("Eliminar Cliente"),
    listaClientes = new List()
    listaClientes.setMultipleMode(false),
```

```
miVector = new Vector();
setLayout(null);
```

Fijar el Layout como null indica que colocaremos o acomodaremos los componentes que hemos creado de acuerdo a sus coordenadas, para ésto recurrimos a setBounds() que recibe cuatro argumentos, este tema se trata cuidadosamente en la sección de Interfaz Gráfica.

```
agregar.setBounds(25, 35, 150, 25);
eliminar.setBounds(225, 35, 150, 25);
listaClientes.setBounds(25, 75, 350, 200);

add(agregar);
add(listaClientes);
add(eliminar);
```

Por último fijamos los Listeners a los botones que nos permiten asociarles una acción propia al oprimirlos También agregamos un WindowListener a la ventana del programa principal, éste se activa para ocultar la ventana y salir del sistema al oprimir su respectivo botón X.

```
agregar.addActionListener(this);
eliminar.addActionListener(this);

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      setVisible(false);
      dispose();
      System.exit(0);
    }
});
```

Finalmente pero no menos importante tenemos un llamado a la función actualizarListaClientes(), este metodo definido un poco más abajo tiene como objetivo mostrar en la Lista todos los elementos contenidos en el Vector, al principio de la aplicación no debe existir ningún elemento por tanto no tiene mucho sentido llamar éste metodo pero es una buena costumbre, en otros programas en los que se almacenen clientes en archivos mediante serialización la llamada a éste método es fundamental al principio del programa.

```
  actualizarListaClientes();
}

/*
Metodo privado que actualiza
la lista de clientes, primero limpia
la lista y después toma los datos nuevos
del contenido del Vector
*/
private void actualizarListaClientes() {
  listaClientes.removeAll();
  for(int i = 0; i < miVector.size(); ++i)
      listaClientes.add(miVector.elementAt(i).toString());
}
```

Me parece que no hay nada confuso o complicado detras del método actualizarListaClientes(), la razon por la cual se limpia la Lista mediante removeAll() es para que no se acumulen clientes ya presentes en el Vector en cada pasada ya que add() agrega elementos pero no verifica si ya estan presentes en la Lista, otra observacion curiosa es el hecho de que el llamado a miVector.elementAt(i).toString() regresa la cadena del método definido en la clase cliente, esto significa que al definir un metodo toString() en la clase cliente ocultamos o sobrescribimos el metodo toString() de la clase Object (porque cualquier objeto o clase creada hereda de object incluso si no lo colocas explícitamente) asi, si intentamos acceder directamente al método getNombre() de la clase cliente de la siguiente manera miVector.elementAt(i).getNombre(), no vamos a poder porque la clase Object no tiene el método getNombre(), para acceder a getNombre() primero debemos hacer un cast para convertir el objeto a cliente y despues de eso podremos acceder a él. El siguiente método es con mucho el más interesante del pequeño programa, ya que define las acciones de los botones. Si se oprime el botón agregar entonces se abre una nueva

ventanita 'ventanaCliente' que sirve para agregar un nuevo cliente y que está definida más abajo en una innerclass, al mismo tiempo a tal ventanita se le registra un WindowListener que 'escuchará' cuando la ventana sea cerrada, cuando suceda esto verificaremos si el cliente 'c' fue creado apropiadamente (o sea que es distinto que null, de allí la presencia del if) si es así agregamos el nuevo cliente recién creado al Vector y después reseteamos el cliente asignándole null en espera de un nuevo cliente futuro, por último actualizamos la lista de clientes, noten que todo sucede al cerrar la ventanita. Ahora, si lo que hacemos es oprimir el botón para eliminar un cliente primero comprobamos que el elemento seleccionado de la lista sea distinto de -1 mediante el método getSelectedIndex(), éste método regresa -1 si no hay ningún elemento de la lista seleccionado (de allí su comprobación), así como el primer elemento de la lista inicia en cero al igual que los elementos del vector podemos tomar directamente el índice del objeto que deseamos eliminar, una vez tomado lo quitamos del vector mediante su método removeElementAt(), por último hacemos otra actualización a la Lista de clientes que mostrará los resultados de tales cambios.

```
/*
Método actionPerformed que controla
las acciones al oprimir los botones
*/
public void actionPerformed(ActionEvent e) {
  if(e.getSource() == agregar) {
    ventanaCliente vcliente = new ventanaCliente();
   vcliente.addWindowListener(new WindowAdapter() {
    public void windowClosed(WindowEvent wevt) {
       if(c != null)
         miVector addElement(c);
       c = null;
       actualizarListaClientes();
   }
});
  } else if(e.getSource() == eliminar) {
      if(listaClientes getSelectedIndex() != -1)
        miVector.removeElementAt(listaClientes getSelectedIndex());
      actualizarListaClientes();
  }
}
```

Ahora, como todo este programa es una aplicación obligadamente debemos incluir un metodo main que defina la existencia de la misma clase, así que aquí llamamos al constructor de nuestro programa principal, le ponemos un título, especificamos mediante setBounds() sus coordenadas y hacemos visible la ventana. Recuerden que el intérprete Java siempre ejecuta este método al inicio de la ejecución, sin él, el programa no arrancaría

```
/*
Método Main, esta es una aplicación,
aquí se inicializa un nuevo ejemploVectores
*/
public static void main(String args[]) {
  ejemploVectores e = new ejemploVectores();
  e.setTitle("Ejemplo del uso de Vectores");
  e setBounds(200, 150, 400, 300);
  e setVisible(true);
}
```

Ahora vamos a declarar una clase interna o innerclass que definirá la ventana para agregar un nuevo cliente Se trata de una ventanita simple que recibe un nombre y un número para nuestro cliente. También muestra un botón de aceptar.

```
/*
Aquí se declara la clase interna o innerclass
ventanaCliente  sirve para agregar clientes
*/

public class ventanaCliente extends Frame implements ActionListener {
```

```
Button aceptar;
Label nombre, numero;
TextField tnombre, tnumero;
```

El constructor y la inicialización de las variables de la interfaz gráfica es simple, también se registra un listener sobre el botón uno sobre la ventana que la cerrará al oprimir su respectivo X. Aquí no se llama una salida al sistema porque al cerrar ésta ventana no se desea cerrar el programa por completo sino solo agregar un cliente.

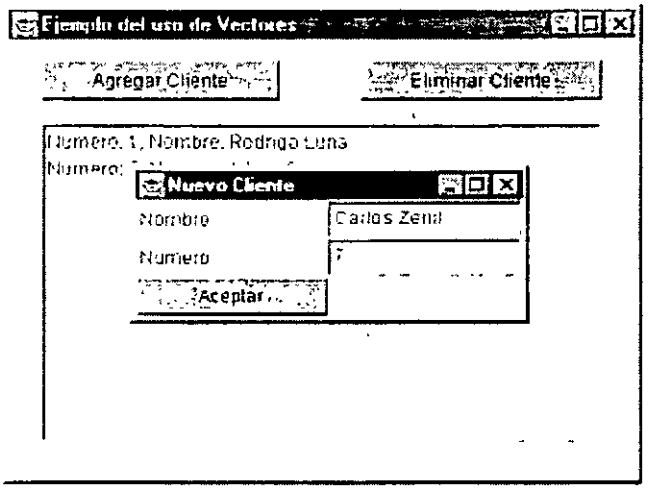
```
//Constructor de ventanaCliente
public ventanaCliente() {
  super("Nuevo Cliente");
  setLayout(new GridLayout(3, 2));
  aceptar = new Button("Aceptar");
  nombre = new Label("Nombre ");
  numero = new Label("Número");
  tnombre = new TextField();
  tnumero = new TextField();
  add(nombre);
  add(tnombre);
  add(numero);
  add(tnumero);
  add(aceptar);

  aceptar.addActionListener(this);

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      setVisible(false);
      dispose();
    }
  });

  setBounds(280, 250, 250, 100);
  setVisible(true);
 }
```

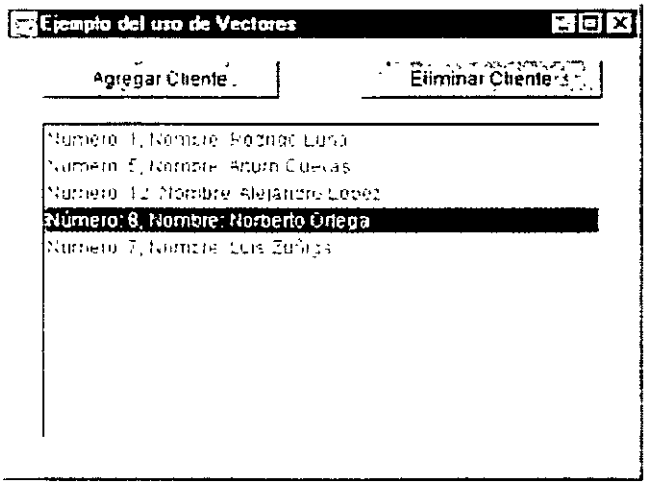
Finalmente pero no menos importante, tenemos la acción sobre el botón de la ventanita para agregar clientes Al oprimir aceptar también verificamos que el contenido de los dos campos de texto sea válido (o lo que es lo mismo que sea distinto de null) así comprobamos que por lo menos se ha introducido cierta información aunque no comprobamos del todo su naturaleza, es decir, no comprobamos si la información introducida en el campo de texto del nombre es realmente un nombre y no un numero, para el campo de texto del número sucede lo mismo pero viceversa Bueno, total, si se introdujo informacion entonces se crea un nuevo cliente tomando los datos de los campos, observen que los datos del segundo campo se convierten en entero mediante Integer parseInt(tnumero getText())), también observen que con esto se cierra la ventana mediante un llamado a dispose() Aquí hay algo importante, precisamente el WindowListener que le registramos a la ventanita de clientes anteriormente (en el método actionPerformed del programa principal) escucha el llamado del dispose(), un windowClosing() no es lo mismo que un windowClosed(), cuando le registramos un windowClosed() ponemos una seccion de codigo en espera de un dispose(), ¡Ese dispose() es el que se encuentra presente aquí! así al dar aceptar se agrega el cliente (si es que se creo bien) al Vector.

```
/*
Control de las acciones del boton,
al agregar un nuevo cliente
*/
public void actionPerformed(ActionEvent e) {
 if(e getSource() == aceptar && tnombre getText() != null
 && tnumero.getText() != null) {
    c = new cliente(tnombre getText(),
    Integer.parseInt(tnumero.getText())),
 setVisible(false);
 dispose(),
 }
```

```
  }

  }
}
```

A continuación hay unas imágenes del funcionamiento general del programa, la primera muestra un cliente siendo agregado y la segunda muestra la Lista con una serie de clientes existentes. Evidentemente éste programa no almacena información en archivos, por eso cada vez que se inicie deberás agregar algunos clientes, realmente no tiene mucha utilidad pero es una introducción perfecta para el tema de serialización y de Vectores.





JavaDoc y RealJ

Como bien dije anteriormente la clase cliente se encuentra relativamente bien documentada, lo hice asi para demostrar la manera en que se genera la documentacion de Java automáticamente mediante RealJ A continuacion se muestra nuevamente la clase 'cliente'

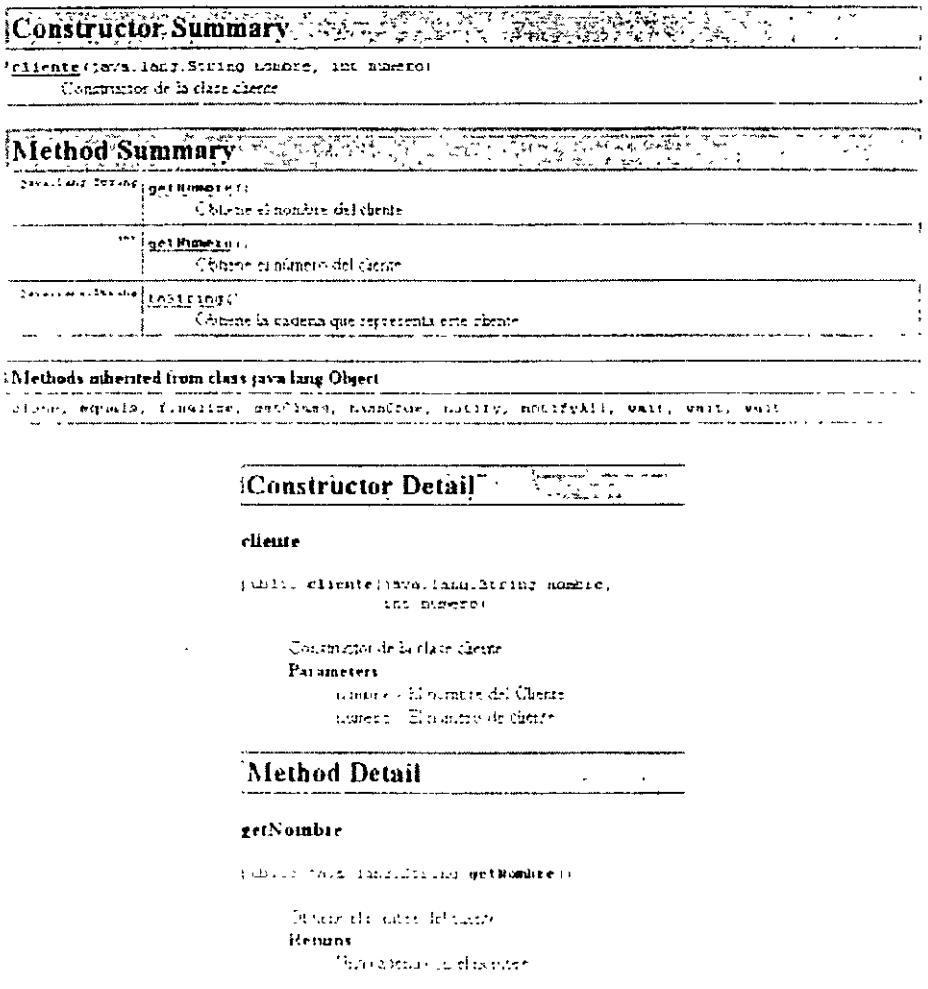
```
public class cliente {

private String nombre = null,
private int numero = 0,

/**
*Clase cliente
*@author JPP
*@version 1.0 Agosto 2002
*/
```

```
/**
*Constructor de la clase cliente
*@param nombre El nombre del Cliente
*@param numero El número de cliente
*/
public cliente(String nombre, int numero) {
this.nombre = nombre;
this.numero = numero;
}

/**
*Obtiene el nombre del cliente
*@return Una cadena con el nombre
*/
public String getNombre() {
return nombre;
}

/**
*Obtiene el número del cliente
*@return Un entero con el número
*/
public int getNumero() {
return numero;
}

/**
*Obtiene la cadena que representa este cliente
*@return Una cadena con la descripción completa del cliente
*/
public String toString() {
return "Número: " + numero + ", Nombre: " + nombre;
}

}
```

Realmente no es muy extensa, los comentarios escritos con cuidado pueden convertirse en la documentación completa del programa, así ahorraríamos un poco de tiempo para documentar todo programa, la documentación tiene como fin auxiliarnos a nosotros o a otros programadores en un futuro para mejorar, corregir o aumentar un programa existente. La documentacion generada es muy similar a la que acompaña la Java API, muestra todos los métodos, sus argumentos y los tipos asociados de todas las clases. Para un principiante éste viene siendo un tema secundario de poca importancia ya que solo programadores profesionales recurren extensamente a la documentación sin embargo es bueno familiarizarnos poco a poco con ella, no todos los programas del curso contienen comentarios de documentacion bien hechos sin embargo la clase anterior es un breve ejemplo de cómo realizarlos. Todo comentario de documentacion tiene la siguiente forma.

```
/**
*Aquí van los comentarios que describen el metodo
*Y aquí van otros parametros
*/
```

Existen muchos parametros tal como se ve en el ejemplo tales como @author, @param, @return, etc. Todos se encuentran perfectamente descritos en la documentacion de la Java API, por ejemplo param sirve para describir los argumentos de un metodo o de un constructor mientras que returnn describe el valor y el tipo que regresa una funcion. Una vez que una clase ha sido documentada podemos usar JavaDoc para generar una documentacion simple, usando RealJ (estando en un proyecto activo) usamos el menú Tools->JavaDoc Proyect Files; después especificamos una carpeta para colocar nuestra documentacion (que sera en formato HTML) y se generan todos los archivos, por ejemplo al generar la documentación de la clase 'cliente' anterior se genera:



Constructor Summary

cliente(java.lang.String nombre, int numero)
Constructor de la clase cliente

Method Summary

java.lang.String	getNombre() Obtiene el nombre del cliente
int	getNumero() Obtiene el número del cliente
java.lang.String	toString() Obtiene la cadena que representa este cliente

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

cliente

public cliente(java.lang.String nombre,
int numero)

Constructor de la clase cliente
Parameters
nombre - El nombre del Cliente
numero - El número de cliente

Method Detail

getNombre

public java.lang.String getNombre()

Obtiene el nombre del cliente
Returns
El nombre del cliente

Como lo he dicho realmente no es de gran importancia dominar JavaDoc para un principiante, ahora si tu objetivo es certificarte o bien vivir de la programacion entonces es un tema fundamental que deberas aprender en el futuro