Java desde Cero

Con ésta comienzo una serie de notas sobre Java, especialmente para aquellos que quieren comenzar a conocerlo y usarlo. Esto se originó en un interés que surgió en algunos de los suscriptores del mailing list de desarrolladores de web, y que pongo a disposición también del de webmasters.
Seguramente muchos de ustedes sabrán mucho más sobre Java que yo, y les agradeceré todo tipo de comentarios o correcciones.
La idea es dar una guía ordenada para el estudio de este lenguaje, muy poderoso y de gran coherencia, aunque todavía adolece de algunas limitaciones que seguramente se irán superando con el tiempo.

Qué es Java

Java es un lenguaje originalmente desarrollado por un grupo de ingenieros de Sun, utilizado por Netscape posteriormente como base para Javascript. Si bien su uso se destaca en el Web, sirve para crear todo tipo de aplicaciones (locales, intranet o internet).
Java es un lenguaje:

- de objetos
- independiente de la plataforma

Algunas características notables:

- robusto
- gestiona la memoria automáticamente
- no permite el uso de técnicas de programación inadecuadas
- multithreading
- cliente-servidor
- mecanismos de seguridad incorporados
- herramientas de documentación incorporadas

Lenguaje de Objetos

Por qué puse "de" objetos y no "orientado a" objetos? Para destacar que, al contrario de otros lenguajes como C++, no es un lenguaje modificado para poder trabajar con objetos sino que es un lenguaje creado para trabajar con objetos desde cero. De hecho, TODO lo que hay en Java son objetos.

Qué es un objeto?

Bueno, se puede decir que todo puede verse como un objeto. Pero seamos más claros. Un objeto, desde nuestro punto de vista, puede verse como una pieza de software que cumple con ciertas características:

- encapsulamiento
- herencia

Encapsulamiento significa que el objeto es auto-contenido, o sea que la misma definición del objeto incluye tanto los datos que éste usa (*atributos*) como los procedimientos (*métodos*) que actúan sobre los mismos.
Cuando se utiliza programación orientada a objetos, se definen *clases* (que definen objetos genéricos) y la forma en que los objetos interactúan entre ellos, a través de *mensajes*. Al crear un objeto de una clase dada, se dice que se crea una *instancia* de la clase, o un objeto propiamente dicho. Por ejemplo, una clase podría ser "autos", y un auto dado es una *instancia* de la clase.
La ventaja de esto es que como no hay programas que actúen modificando al objeto, éste se mantiene en cierto modo independiente del resto de la aplicación. Si es necesario modificar el objeto (por ejemplo, para darle más capacidades), esto se puede hacer sin tocar el resto de la aplicación… lo que ahorra mucho tiempo

de desarrollo y debugging! En Java, inclusive, *ni siquiera existen las variables globales*! (Aunque parezca difícil de aceptar, esto es una gran ventaja desde el punto de vista del desarrollo).
En cuanto a la herencia, simplemente significa que se pueden crear nuevas clases que hereden de otras preexistentes; esto simplifica la programación, porque las clases hijas incorporan automáticamente los métodos de las madres. Por ejemplo, nuestra clase "auto" podría heredar de otra más general, "vehículo", y simplemente redefinir los métodos para el caso particular de los automóviles… lo que significa que, con una buena biblioteca de clases, se puede reutilizar mucho código inclusive sin saber lo que tiene adentro.

Un ejemplo simple

Para ir teniendo una idea, vamos a poner un ejemplo de una clase Java:

```
public class Muestra extends Frame {
// atributos de la clase
Button si;
Button no;
// métodos de la clase:
public Muestra () {
Label comentario = new Label("Presione un botón", Label.CENTER);
si = new Button("Sí");
no = new Button("No");
add("North", comentario);
add("East", si);
add("West", no);
}
}
```

Esta clase no está muy completa así, pero da una idea… Es una clase heredera de la clase *Frame* (un tipo de ventana) que tiene un par de botones y un texto. Contiene dos atributos ("si" y "no"), que son dos objetos del tipo *Button*, y un único método llamado Muestra (igual que la clase, por lo que es lo que se llama un *constructor*).

Independiente de la plataforma

Esto es casi del todo cierto…
En realidad, Java podría hacerse correr hasta sobre una Commodore 64! La realidad es que para utilizarlo en todo su potencial, requiere un sistema operativo multithreading (como Unix, Windows95, OS/2…).
Cómo es esto? Porque en realidad Java es un lenguaje interpretado… al menos en principio.
Al compilar un programa Java, lo que se genera es un seudocódigo definido por Sun, para una máquina genérica. Luego, al correr sobre una máquina dada, el software de ejecución Java simplemente interpreta las instrucciones, emulando a dicha máquina genérica. Por supuesto esto no es muy eficiente, por lo que tanto Netscape como Hotjava o Explorer, al ejecutar el código por primera vez, lo van compilando (mediante un *JIT*: *J*ust *I*n *T*ime compiler), de modo que al crear por ejemplo la segunda instancia de un objeto el código ya esté compilado específicamente para la máquina huésped.
Además, Sun e Intel se han puesto de acuerdo para desarrollar procesadores que trabajen directamente en Java, con lo que planean hacer máquinas muy baratas que puedan conectarse a la red y ejecutar aplicaciones Java cliente-servidor a muy bajo costo.
El lenguaje de dicha máquina genérica es público, y si uno quisiera hacer un intérprete Java para una Commodore sólo tendría que implementarlo y pedirle a Sun la aprobación (para que verifique que cumple con los requisitos de Java en cuanto a cómo interpreta cada instrucción, la seguridad, etc.)

Algunas características…

Entre las características que nombramos nos referimos a la robustez. Justamente por la forma en que está diseñado, Java no permite el manejo directo del hardware ni de la memoria (inclusive no permite modificar

valores de punteros, por ejemplo); de modo que se puede decir que es virtualmente imposible colgar un programa Java. El intérprete siempre tiene el control.
Inclusive el compilador es suficientemente inteligente como para no permitir un montón de cosas que podrían traer problemas, como usar variables sin inicializarlas, modificar valores de punteros directamente, acceder a métodos o variables en forma incorrecta, utilizar herencia múltiple, etc.
Además, Java implementa mecanismos de seguridad que limitan el acceso a recursos de las máquinas donde se ejecuta, especialmente en el caso de los Applets (que son aplicaciones que se cargan desde un servidor y se ejecutan en el cliente).
También está diseñado específicamente para trabajar sobre una red, de modo que incorpora objetos que permiten acceder a archivos en forma remota (via URL por ejemplo).
Además, con el JDK (Java Development Kit) vienen incorporadas muchas herramientas, entre ellas un generador automático de documentación que, con un poco de atención al poner los comentarios en las clases, crea inclusive toda la documentación de las mismas en formato HTML!

El Java Development Kit

Todo lo que puedan pedir para desarrollar aplicaciones en Java está en:

- http://java.sun.com/aboutJava/index.html

En particular, deberían bajarse el JDK y el API Documentation de:

- http://java.sun.com/java.sun.com/products/JDK/1.0.2/index.html

(También les puede interesar en particular el Tool Documentation y alguno de los otros paquetes de la página)
Nota: en este site también hay un tutorial de Java, aunque es un poco difícil de seguir para el principiante.
El JDK (versión 1.0.2) está disponible para SPARC/Solaris, x86/Solaris, MS-Windows 95/NT, y MacOS.
También está disponible el fuente para el que quiera adaptarlo para otro sistema operativo, y he leído por ahí que hay una versión dando vueltas para Linux y HP-UX.
Básicamente, el JDK consiste de:

- el compilador Java, javac
- el intérprete Java, java
- un visualizador de applets, appletviewer
- el debugger Java, jdb (que para trabajar necesita conectarse al server de Sun)
- el generador de documentación, javadoc

También se puede bajar del mismo site un browser que soporta Java (y de hecho está escrito *totalmente* en Java), el Hotjava.
Para instalarlo simplemente hay que descompactar el archivo (sugiero que creen un directorio java para eso), pero tengan en cuenta NO DESCOMPRIMIR el archivo classes.zip!
Importante para los usuarios de Windows95: todas estas aplicaciones deben ejecutarse desde una ventana DOS. En particular, utilizan nombres largos y distinguen mayúsculas de minúsculas, así que tengan en cuenta esto que es fuente de muchos errores.
Una cosa muy importante: para que todo ande bien aceitado, agreguen:

- el directorio de los programas en el path (ej: c:\java\bin)
- las variables de entorno:
    - CLASSPATH=.;C:\java\lib\classes.zip
    - HOMEDRIVE=C:
    - HOMEPATH=\
    - HOME=C:\

con los valores adecuados a su entorno.
Noten que en CLASSPATH agregué el directorio actual (.), para poder compilar y ejecutar desde cualquier directorio.

Empecemos de una vez!

Bueno, suponiendo que hayan instalado todo, y antes de comenzar a programar en Java, una pequeña aclaración :

En realidad se puede decir que hay tres Javas por ahí:

- Javascript: es una versión de Java directamente interpretada, que se incluye como parte de una página HTML, lo que lo hace muy fácil y cómodo para aplicaciones muy pequeñas, pero que en realidad tiene muchas limitaciones:
  - no soporta clases ni herencia
  - no se precompila
  - no es obligatorio declarar las variables
  - verifica las referencias en tiempo de ejecución
  - no tiene protección del código, ya que se baja en ascii
  - no todos los browsers lo soportan completamente; Explorer, por ejemplo, no soporta las últimas adiciones de Netscape, como las imágenes animadas.
- Java standalone: programas Java que se ejecutan directamente mediante el intérprete java.
- Applets: programas Java que corren bajo el entorno de un browser (o del appletviewer)

En sí los dos últimos son el mismo lenguaje, pero cambia un poco la forma en que se implementa el objeto principal (la aplicación). Vamos a ver cómo crear las aplicaciones para que, sin cambios, se puedan ejecutar casi igual en forma standalone o como applet (en realidad hay cosas que los applets no pueden hacer, como acceder a archivos sin autorización).

Javascript

No vamos a detenernos mucho en Javascript, por las limitaciones antedichas; si les interesa podemos dedicarnos un poco a este lenguaje en el futuro. Por ahora, sólo un ejemplo sencillo:
Calculadora en Javascript:

```
<HTML>
<HEAD>
<SCRIPT LANGUAJE="Javascript">
 function calcula(form) {
  if (confirm("¿Está seguro?"))
    form.resultado.value = eval(form.expr.value)
  else
    alert("Vuelva a intentarlo...")
 }
</SCRIPT>
</HEAD>
<BODY>
<FORM>
 Introduzca una expresión:
 <INPUT TYPE="text" NAME="expr" SIZE=15>
 <INPUT TYPE="button" NAME="Boton" VALUE="Calcular" ONCLICK="calcula(this.form)">
 <BR>
 Resultado:
 <INPUT TYPE="text" NAME="resultado" SIZE=15>
 <BR>
</FORM>
</BODY>
</HTML>
```

Básicamente, el código se encuadra entre los tags <SCRIPT>…</SCRIPT>, y los parámetros se pasan al mismo mediante un form (<FORM>…</FORM>). El lenguaje utilizado es muy parecido al C++, y básicamente el código se ejecuta mediante una acción de un botón (…ONCLICK="calcula(this.form)").
Al presionar el botón, se llama a la función **calcula** con el parámetro **this.form**, que se refiere al form al que pertenece el botón.

La función asigna al **valor** del campo **resultado** del **form** que se le pasa como parámetro (*form.resultado.value*) el resultado de **eval**uar el **valor** de la expresión del campo **expr** de dicho **form** (*eval(form.expr.value)*).
Hay MUCHOS ejemploes de Javascript en:

- http://www.c2.net/~andreww/javascript/

incluyendo decenas de calculadoras, juegos y otras yerbas!
Allí también encontrarán la documentación y un tutorial sobre Javascript.

Las clases en Java

Bueno, antes que nada conviene saber que en Java hay un montón de clases ya definidas y utilizables.
Éstas vienen en las bibliotecas estándar:

- java.lang - clases esenciales, números, strings, objetos, compilador, runtime, seguridad y threads (es el único paquete que se incluye automáticamente en todo programa Java)
- java.io - clases que manejan entradas y salidas
- java.util - clases útiles, como estructuras genéricas, manejo de fecha, hora y strings, número aleatorios, etc.
- java.net - clases para soportar redes: URL, TCP, UDP, IP, etc.
- java.awt - clases para manejo de interface gráfica, ventanas, etc.
- java.awt.image - clases para manejo de imágenes
- java.awt.peer - clases que conectan la interface gráfica a implementaciones dependientes de la plataforma (motif, windows)
- java.applet - clases para la creación de applets y recursos para reproducción de audio.

Para que se den una idea, los números enteros, por ejemplo, son "instancias" de una clase no redefinible, *Integer*, que desciende de la clase *Number* e implementa los siguientes atributos y métodos:

```
public final class java.lang.Integer extends java.lang.Number {
    // Atributos
  public final static int MAX_VALUE;
  public final static int MIN_VALUE;
    // Métodos Constructores
  public Integer(int value);
  public Integer(String s);
    // Más Métodos
  public double doubleValue();
  public boolean equals(Object obj);
  public float floatValue();
  public static Integer getInteger(String nm);
  public static Integer getInteger(String nm, int val);
  public static Integer getInteger(String nm, Integer val);
  public int hashCode();
  public int intValue();
  public long longValue();
  public static int parseInt(String s);
  public static int parseInt(String s, int radix);
  public static String toBinaryString(int i);
  public static String toHexString(int i);
  public static String toOctalString(int i);
  public String toString();
  public static String toString(int i);
```

```
   public static String  toString(int  i, int  radix);
   public static Integer valueOf(String  s);
   public static Integer valueOf(String  s, int  radix);
}
```

Mucho, no?
Esto también nos da algunas ideas:

- la estructura de una clase
- caramba, hay métodos repetidos!

De la estructura enseguida hablaremos; en cuanto a los métodos repetidos (como *parseInt* por ejemplo), al llamarse al método el compilador decide cuál de las implementaciones del mismo usar basándose en la cantidad y tipo de parámetros que le pasamos. Por ejemplo, parseInt("134") y parseInt("134",16), al compilarse, generarán llamados a dos métodos distintos.

Estructura de una clase

Una clase consiste en:

```
algunas_palabras class nombre_de_la_clase [algo_más] {
            [lista_de_atributos]
            [lista_de_métodos]
}
```

Lo que está entre [y] es opcional…
Ya veremos qué poner en "algunas_palabras" y "algo_más", por ahora sigamos un poco más.
La lista de atributos (nuestras viejas variables locales) sigue el mismo formato de C: se define primero el tipo y luego el nombre del atributo, y finalmente el ";".

```
public final static int MAX_VALUE
;
```

También tenemos "algunas_palabras" adelante, pero en seguida las analizaremos.
En cuanto a los métodos, también siguen la sintaxis del C; un ejemplo:

```
public int incContador() {          // declaración y apertura de {
   cnt++;                           // instrucciones, separadas por ";"
   return(cnt);
}                                                   // cierre de }
```

Finalmente, se aceptan comentarios entre /* y */, como en C, o bien usando // al principio del comentario (el comentario termina al final de la línea).
Veamos un ejemplo:

```
// Implementación de un contador sencillo
// GRABAR EN UN ARCHIVO     "Contador.java"        (OJO CON LAS MAYUSCULAS!)
// COMPILAR CON:            "javac Contador.java"  (NO OLVIDAR EL .java!)
// ESTA CLASE NO ES UNA APLICACION, pero nos va a servir enseguida

public class Contador {                     // Se define la clase Contador

 // Atributos
 int cnt;                                           // Un entero para guardar el valor actual

 // Constructor                                     // Un método constructor…
 public Contador() {                        // …lleva el mismo nombre que la clase
  cnt = 0;                                  // Simplemente, inicializa (1)
 }

 // Métodos
```

```
 public int incCuenta() {                    // Un método para incrementar el contador
  cnt++;                                     // incrementa cnt
  return cnt;                                // y de paso devuelve el nuevo valor
 }
 public int getCuenta() {                    // Este sólo devuelve el valor actual
  return cnt;                                // del contador
 }
}
```

Cuando, desde una aplicación u otro objeto, se crea una **instancia** de la clase **Contador**, mediante la instrucción:

```
new Contador()
```

el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto o instancia, ya que una vez que fue creado no puede recrearse sobre sí mismo).

En tiempo de ejecución, al encontrar dicha instrucción, el intérprete reserva espacio para el objeto/instancia, crea su estructura y llama al constructor.

O sea que el efecto de `new Contador()` es, precisamente, reservar espacio para el contador e inicializarlo en cero.

En cuanto a los otros métodos, se pueden llamar desde otros objetos (lo que incluye a las aplicaciones) del mismo modo que se llama una función desde C.

Por ejemplo, usemos nuestro contador en un programa bien sencillo que nos muestre cómo evoluciona:

```
// Usemos nuestro contador en una mini-aplicación
// GRABAR EN UN ARCHIVO    "Ejemplo1.java"      (OJO CON LAS MAYUSCULAS!)
// COMPILAR CON:           "javac Ejemplo.java" (NO OLVIDAR EL .java!)
// EJECUTAR CON:           "java Ejemplo1"                (SIN el .java)

import java.io.*;                              // Uso la biblioteca de entradas/salidas

public class Ejemplo1 {                        // IMPORTANTE: Nombre de la clase
                                               // igual al nombre del archivo!
 // entero para asignarle el valor del contador e imprimirlo
 // aunque en realidad no me hace falta.
 static int n;
 // y una variable tipo Contador para instanciar el objeto…
 static Contador laCuenta;

 // ESTE METODO, MAIN, ES EL QUE HACE QUE ESTO SE COMPORTE
 // COMO APLICACION. Es donde arranca el programa cuando ejecuto "java Ejemplo1"
 // NOTA: main debe ser public & static.
 public static void main ( String args[] ) {
  System.out.println ("Cuenta… ");             // Imprimo el título
  laCuenta = new Contador();                   // Creo una instancia del Contador
  System.out.println (laCuenta.getCuenta());   // 0 - Imprimo el valor actual (cero!)
  n = laCuenta.incCuenta();                    // 1 - Asignación e incremento
  System.out.println (n);                      // Ahora imprimo n
  laCuenta.incCuenta();                        // 2 - Lo incremento (no uso el valor…
  System.out.println (laCuenta.getCuenta());   // …de retorno) y lo imprimo
  System.out.println (laCuenta.incCuenta());   // 3 - Ahora todo en un paso!
 }
}
```

En el capítulo III vamos a analizar este programa en detalle. Por ahora veamos la diferencia con un applet que haga lo mismo:

```
            // Applet de acción similar a la aplicación Ejemplo1
            // GRABAR EN ARCHIVO:        "Ejemplo2.java"
            // COMPILAR CON:             "javac Ejemplo2.java"
            // PARA EJECUTAR:  Crear una página HTML como se indica luego
import java.applet.*;
import java.awt.*;

public class Ejemplo2 extends Applet {
 static int n;
 static Contador laCuenta;

 // Constructor…
 public Ejemplo2 () {
   laCuenta = new Contador();
 }

 // El método paint se ejecuta cada vez que hay que redibujar
 // NOTAR EL EFECTO DE ESTO CUANDO SE CAMBIA DE TAMAÑO LA
 // VENTANA DEL NAVEGADOR!
 public void paint (Graphics g) {
   g.drawString ("Cuenta...", 20, 20);
   g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 35 );
   n = laCuenta.incCuenta();
   g.drawString (String.valueOf(n), 20, 50 );
   laCuenta.incCuenta();
   g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 65 );
   g.drawString (String.valueOf(laCuenta.incCuenta()), 20, 80 );
 }
}
```

Ahora es necesario crear una página HTML para poder visualizarlo. Para esto, crear y luego cargar el archivo ejemplo2.htm con un browser que soporte Java (o bien ejecutar en la ventana DOS: "appletviewer ejemplo2.htm"):

```
<HTML>
<HEAD>
<TITLE>Ejemplo 2 - Applet Contador</TITLE>
</HEAD>
<BODY>
<applet code="Ejemplo2.class" width=170 height=150>
</applet>
</BODY>
</HTML>
```

Para terminar este capítulo, observemos las diferencias entre la aplicación standalone y el applet:

- La aplicación usa un método main, desde donde arranca
- El applet, en cambio, se arranca desde un constructor (método con el mismo nombre que la clase)

Además:

- En la aplicación utilizamos System.out.println para imprimir en la salida estándar
- En el applet necesitamos "dibujar" el texto sobre un fondo gráfico, por lo que usamos el método g.drawString dentro del método paint (que es llamado cada vez que es necesario redibujar el applet)

Con poco trabajo se pueden combinar ambos casos en un solo objeto, de modo que **la misma** clase sirva para utilizarla de las dos maneras:

```
// Archivo:  Ejemplo3.java
// Compilar con: javac Ejemplo3.java
import java.applet.*;
import java.awt.*;
import java.io.*;

public class Ejemplo3 extends Applet {
  static int n;
  static Contador laCuenta;

  public Ejemplo3 () {
    laCuenta = new Contador();
  }

  public static void main(String args[]) {
    laCuenta = new Contador();
    paint();
  }

  public static void paint () {
    System.out.println ("Cuenta...");
    System.out.println (laCuenta.getCuenta());
    n = laCuenta.incCuenta();
    System.out.println (n);
    laCuenta.incCuenta();
    System.out.println (laCuenta.getCuenta());
    System.out.println (laCuenta.incCuenta());
  }
  public void paint (Graphics g) {
    g.drawString ("Cuenta...", 20, 20);
    g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 35 );
    n = laCuenta.incCuenta();
    g.drawString (String.valueOf(n), 20, 50 );
    laCuenta.incCuenta();
    g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 65 );
    g.drawString (String.valueOf(laCuenta.incCuenta()), 20, 80 );
  }
}
```

Esta clase puede ejecutarse tanto con "java Ejemplo3" en una ventana DOS, como cargarse desde una página HTML con:

```
<applet code="Ejemplo3.class" width=170 height=150>
</applet>
```

Notar que conviene probar el applet con el appletviewer ("appletviewer ejemplo3.htm"), ya que éste indica en la ventana DOS si hay algún error durante la ejecución. Los browsers dejan pasar muchos errores, simplemente suprimiendo la salida a pantalla del código erróneo.

Notar que en todo este desarrollo de las clases Ejemplo1, Ejemplo2 y Ejemplo3, en ningún momento volvimos a tocar la clase Contador!

Estructura de clases

Vamos a comenzar analizando la clase Contador, para ir viendo las partes que forman una clase una por una y en detalle. Este capítulo va a ser un poco aburrido por lo exhaustivo (aunque algunos puntos más complicados como las excepciones y los threads los dejaremos para después), pero me parece bueno tener un resumen completo de la sintaxis desde ahora.
Luego iremos armando pequeñas aplicaciones para probar cada cosa.
Recordemos la definición de la clase Contador:

```
// Implementación de un contador sencillo
public class Contador {
// Atributos
int cnt;
// Constructor
public Contador() {
cnt = 0;
}
// Métodos
public int incCuenta() {
cnt++;
return cnt;
}
public int getCuenta() {
return cnt;
}
}
```

Declaración de la clase

La clase se declara mediante la línea public class Contador. En el caso más general, la declaración de una clase puede contener los siguientes elementos:

[**public**] [**final** | **abstract**] **class** Clase [**extends** ClaseMadre] [**implements** Interfase1 [, Interfase2]…]
o bien, para interfaces:
[**public**] **interface** Interfase [**extends** InterfaseMadre1 [, InterfaseMadre2]…]
Como se ve, lo único obligatorio es **class** y el nombre de la clase. Las interfases son un caso de clase particular que veremos más adelante.

Public, final o abstract

Definir una clase como pública (**public**) significa que puede ser usada por cualquier clase en cualquier paquete. Si no lo es, solamente puede ser utilizada por clases del mismo paquete (más sobre paquetes luego; básicamente, se trata de un grupo de clases e interfaces relacionadas, como los paquetes de biblioteca incluídos con Java).
Una clase final (**final**) es aquella que no puede tener clases que la hereden. Esto se utiliza básicamente por razones de seguridad (para que una clase no pueda ser reemplazada por otra que la herede), o por diseño de la aplicación.
Una clase abstracta (**abstract**) es una clase que puede tener herederas, pero no puede ser instanciada. Es, literalmente, abstracta (como la clase *Number* definida en java.lang). ¿Para qué sirve? Para modelar conceptos. Por ejemplo, la clase *Number* es una clase abstracta que representa cualquier tipo de números (y sus métodos no están implementados: son abstractos); las clases descendientes de ésta, como *Integer* o *Float*, sí implementan los métodos de la madre *Number*, y se pueden instanciar.
Por lo dicho, una clase no puede ser **final** y **abstract** a la vez (ya que la clase abstract requiere descendientes…)
¿Un poco complejo? Se va a entender mejor cuando veamos casos particulares, como las interfases (que por definición son abstractas ya que no implementan sus métodos).

Extends

La instrucción **extends** indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **Object**.
Cuando una clase desciende de otra, esto significa que hereda sus atributos y sus métodos (es decir que, a menos que los redefinamos, sus métodos son los mismos que los de la clase madre y pueden utilizarse en forma transparente, a menos que sean *privados* en la clase madre o, para subclases de otros paquetes, protegidos o propios del paquete). Veremos la calificación de métodos muy pronto, a no desesperar!

Implements

Una interfase (**interface**) es una clase que declara sus métodos pero no los implementa; cuando una clase implementa (**implements**) una o más interfases, debe contener la implementación de todos los métodos (con las mismas listas de parámetros) de dichas interfases.
Esto sirve para dar un ascendiente común a varias clases, obligándolas a implementar los mismos métodos y, por lo tanto, a comportarse de forma similar en cuanto a su interfase con otras clases y subclases.

Interface

Una interfase (**interface**), como se dijo, es una clase que no implementa sus métodos sino que deja a cargo la implementación a otras clases. Las interfases pueden, asimismo, descender de otras interfases pero no de otras clases.
Todos sus métodos son por definición abstractos y sus atributos son finales (aunque esto no se indica en el cuerpo de la interfase).
Son útiles para generar relaciones entre clases que de otro modo no están relacionadas (haciendo que implementen los mismos métodos), o para distribuir paquetes de clases indicando la estructura de la interfase pero no las clases individuales (objetos anónimos).
Si bien diferentes clases pueden implementar las mismas interfases, y a la vez descender de otras clases, esto no es en realidad herencia múltiple ya que una clase no puede heredar atributos ni métodos de una interface; y las clases que implementan una interfase pueden no estar ni siquiera relacionadas entre sí.

El cuerpo de la clase

El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que constituyen la clase.
No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

Declaración de atributos

En Java no hay variables globales; todas las variables se declaran dentro del cuerpo de la clase o dentro de un método. Las variables declaradas dentro de un método son **locales** al método; las variables declaradas en el cuerpo de la clase se dice que son *miembros* de la clase y son accesibles por todos los métodos de la clase.
Por otra parte, además de los atributos de la propia clase se puede acceder a todos los atributos de la clase de la que desciende; por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos *npoints*, *xpoints* e *ypoints*.
Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia*; se dice que son atributos *de clase* si se usa la palabra clave **static**: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). Si no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).
La declaración sigue siempre el mismo esquema:
[**private**|**protected**|**public**] [**static**] [**final**] [**transient**] [**volatile**] Tipo NombreVariable [= Valor]**;**

Private, protected o public

Java tiene 4 tipos de acceso diferente a las variables o métodos de una clase: privado, protegido, público o por paquete (si no se especifica nada).
De acuerdo a la forma en que se especifica un atributo, objetos de otras clases tienen distintas posibilidades de accederlos:

Acceso desde:	**private**	**protected**	**public**	(package)
la propia clase	S	S	S	S
subclase en el mismo paquete	N	S	S	S
otras clases en el mismo paquete	N	S	S	S
subclases en otros paquetes	N	X	S	N
otras clases en otros paquetes	N	N	S	N

S: puede acceder
N: no puede acceder
X: puede acceder al atributo en objetos que pertenezcan a la subclase, pero no en los que pertenecen a la clase madre. Es un caso especial ; más adelante veremos ejemplos de todo esto.

Static y final

Como ya se vio, **static** sirve para definir un atributo como *de clase*, o sea único para todos los objetos de la clase.
En cuanto a **final**, como en las clases, determina que un atributo no pueda ser sobreescrito o redefinido. O sea: no se trata de una variable, sino de una *constante*.

Transient y volatile

Son casos bastante particulares y que no habían sido implementados en Java 1.0.
Transient denomina atributos que no se graban cuando se archiva un objeto, o sea que no forman parte del estado permanente del mismo.

Volatile se utiliza con variables modificadas asincrónicamente por objetos en diferentes *threads* (literalmente "hilos", tareas que se ejecutan en paralelo); básicamente esto implica que distintas tareas pueden intentar modificar la variable simultáneamente, y *volatile* asegura que se vuelva a leer la variable (por si fue modificada) cada vez que se la va a usar (esto es, en lugar de usar registros de almacenamiento como buffer).

Los tipos de Java

Los tipos de variables disponibles son básicamente 3:

- tipos básicos (no son objetos)
- arreglos (arrays)
- clases e interfases

Con lo que vemos que cada vez que creamos una clase o interfase estamos definiendo un nuevo tipo.
Los **tipos básicos** son:

Tipo	**Tamaño/Formato**	**Descripción**
byte	8-bit complemento a 2	Entero de un byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
float	32-bit IEEE 754	Punto flotante, precisión simple
double	64-bit IEEE 754	Punto flotante, precisión doble
char	16-bit caracter Unicode	Un caracter
boolean	true, false	Valor booleano (verdadero o falso)

Los **arrays** son arreglos de cualquier tipo (básico o no). Por ejemplo, existe una clase *Integer*; un arreglo de objetos de dicha clase se notaría:
Integer vector[ ];
Los arreglos siempre son dinámicos, por lo que **no es válido** poner algo como:
Integer cadena[5];
Aunque sí es válido inicializar un arreglo, como en:
int días[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
char letras[ ] = { 'E', 'F', 'M', 'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D' };
String nombres[ ] = new String[12];
Nota al margen: no confundir un String (*cadena* de caracteres) con un *arreglo* de caracteres! Son cosas bien distintas!
Ya hablaremos más adelante de las clases String y StringBuffer.
En Java, para todas las variables de tipo básico se accede al valor asignado a la misma directamente (no se conoce la dirección de memoria que ocupa). Para las demás (arrays, clases o interfases), se accede a través de un puntero a la variable. El valor del puntero no es accesible ni se puede modificar (como en C); Java no necesita esto y además eso atentaría contra la robustez del lenguaje.
De hecho, en Java no existen los tipos **pointer**, **struct** o **union**. Un objeto es más que una estructura, y las uniones no se hacen necesarias con un método de programación adecuado (y además se evita la posibilidad de acceder a los datos incorrectamente).
Algo más respecto a los arreglos: ya que Java gestiona el manejo de memoria para los mismos, y lanza excepciones si se intenta violar el espacio asignado a una variable, se evitan problemas típicos de C como acceder a lugares de memoria prohibidos o fuera del lugar definido para la variable (como cuando se usa un subíndice más grande que lo previsto para un arreglo…).

Y los métodos…

Los métodos, como las clases, tienen una declaración y un cuerpo.
La declaración es del tipo:
[**private**|**protected**|**public**] [**static**] [**abstract**] [**final**] [**native**] [**synchronized**] TipoDevuelto NombreMétodo **(** [tipo1 nombre1[, tipo2 nombre2]…] **)** [**throws** excepción1 [,excepción2]…]
A no preocuparse: poco a poco aclararemos todo con ejemplos.

Básicamente, los métodos son como las funciones de C: implementan, a través de funciones, operaciones y estructuras de control, el cálculo de algún parámetro que es el que devuelven al objeto que los llama. Sólo pueden devolver **un** valor (del tipo *TipoDevuelto*), aunque pueden no devolver ninguno (en ese caso *TipoDevuelto* es **void**). Como ya veremos, el valor de retorno se especifica con la instrucción **return**, dentro del método.
Los métodos pueden utilizar valores que les pasa el objeto que los llama (*parámetros*), indicados con *tipo1 nombre1, tipo2 nombre2…* en el esquema de la declaración.
Estos parámetros pueden ser de cualquiera de los tipos ya vistos. Si son tipos básicos, el método recibe el *valor* del parámetro; si son arrays, clases o interfases, recibe un puntero a los datos (*referencia*). Veamos un pequeño ejemplo:

```
public int AumentarCuenta(int cantidad) {
cnt = cnt + cantidad;
return cnt;
}
```

Este método, si lo agregamos a la clase **Contador**, le suma *cantidad* al acumulador **cnt**. En detalle:

- el método recibe un valor entero (*cantidad*)
- lo suma a la variable de instancia *cnt*
- devuelve la suma (*return cnt*)

¿Cómo hago si quiero devolver más de un valor? Por ejemplo, supongamos que queremos hacer un método dentro de una clase que devuelva la posición del mouse.
Lo siguiente no sirve:

```
void GetMousePos(int x, int y) {
        x = ….;          // esto no sirve!
        y = ….;          // esto tampoco!
}
```

porque el método no puede modificar los parámetros *x* e *y* (que han sido pasados por valor, o sea que el método recibe el valor numérico pero no sabe adónde están las variables en memoria).
La solución es utilizar, en lugar de tipos básicos, una clase:

```
class MousePos { public int x, y; }
```

y luego utilizar esa clase en nuestro método:

```
void GetMousePos( MousePos m ) {
        m.x = ……;
        m.y = ……;
}
```

El resto de la declaración

Public, **private** y **protected** actúan exactamente igual para los métodos que para los atributos, así que veamos el resto.
Los métodos estáticos (**static**), son, como los atributos, métodos *de clase*; si el método no es **static** es un método *de instancia*. El significado es el mismo que para los atributos: un método *static* es compartido por todas las instancias de la clase.
Ya hemos hablado de las clases abstractas; los métodos abstractos (**abstract**) son aquellos de los que se da la declaración pero no la implementación (o sea que consiste sólo del encabezamiento). Cualquier clase que contenga al menos un método abstracto (o cuya clase madre contenga al menos un método abstracto que no esté implementado en la hija) es una clase abstracta.
Es **final** un método que no puede ser redefinido por ningún descendiente de la clase.

Las clases **native** son aquellas que se implementan en otro lenguaje (por ejemplo C o C++) propio de la máquina. Sun aconseja utilizarlas bajo riesgo propio, ya que en realidad son ajenas al lenguaje. Pero la posibilidad de usar viejas bibliotecas que uno armó y no tiene ganas de reescribir existe!.
Las clases **synchronized** permiten sincronizar varios *threads* para el caso en que dos o más accedan concurrentemente a los mismos datos. De nuevo, más detalles habrá en el futuro, cuando hablemos de threads.
Finalmente, la cláusula **throws** sirve para indicar que la clase genera determinadas excepciones.

El cuerpo de los métodos

Otra vez recordaremos nuestra vieja clase Contador:

```
// Implementación de un contador sencillo
public class Contador {
……..
public int incCuenta() {
cnt++;
return cnt;
}
……
}
```

Dentro de los métodos pueden incluirse:

- Declaración de variables locales
- Asignaciones a variables

- Operaciones matemáticas
- Llamados a otros métodos:
    - dentro de la clase
    - *de instancia*, de otras clases
    - *de clase*, de cualquier clase
- Estructuras de control
- Excepciones (try, catch, que veremos más adelante)

Declaración de variables locales

Las variables locales se declaran igual que los atributos de la clase:
Tipo NombreVariable [= Valor];
Ej: int suma;
float precio;
Contador laCuenta;
Sólo que aquí no se declaran private, public, etc., sino que las variables definidas dentro del método sólo son accesibles por él.
Las variables pueden inicializarse al crearse:
Ej: int suma = 0;
float precio = 12.3;
Contador laCuenta = new Contador ( );

Asignaciones a variables

Se asigna un valor a una variable mediante el signo **=**:
Variable = Constante | Expresión ;
Ej: suma = suma + 1;
precio = 1.05 * precio;
laCuenta.cnt = 0;
El último caso es válido si cnt es una variable pública de la clase **Contador**. Personalmente no creo conveniente acceder directamente a variables de otro objeto, ya que futuras modificaciones del objeto llamado o del que llama puede propender la difusión de errores… Es mejor usar métodos como *getCuenta* o un hipotético *inicializarContador* para ello. De hecho, algunos sugieren que todas las variables de una clase se declaren como **private**.
En el primer caso, o sea en general:
Variable = Variable Operador Expresión;
se puede escribir en forma más sencilla:
Variable Operador= Expresión;
Por ejemplo, suma = suma + 9 - cantidad;
puede escribirse: suma += 9-cantidad;
y precio = precio * 0.97;
como: precio *= 0.97;

Operaciones matemáticas

Hay varios tipos de operadores:
Unarios: + - ++ -- ~ ! (tipo) …..etc.
Se colocan antes (o en algunos casos después) de la constante o expresión.
Por ejemplo: -cnt; // cambia de signo; por ejemplo si cnt es 12 el resultado es -12; cnt no cambia.
++cnt; // equivale a cnt += 1;
cnt++; // equivale a cnt +=1; veremos la diferencia al hablar de estructuras de control
--cnt; // equivale a cnt -= 1;
cnt--; // equivale a cnt -= 1;
Binarios: + - * / % …..etc.
Van entre dos constantes o expresiones o combinación de ambas.

Por ejemplo: cnt + 2; // debuelve la suma de ambos.
promedio + (valor / 2); // como se ve, se pueden usar paréntesis.
horas / hombres; // división.
acumulado % 3; // resto de la división entera entre ambos.
Nota: + sirve también para concatenar cadenas de caracteres; hablaremos de String y StringBuffer pronto.
Cuando se mezclan Strings y valores numéricos, éstos se convierten automáticamente a cadenas:
"La frase tiene " + cant + " letras"
se convierte en: "La frase tiene 17 letras" // suponiendo que cant = 17

Precedencia de operadores en Java

La siguiente es la precedencia de los operadores en expresiones compuestas. De todos modos, como en todos los lenguajes, se recomienda usar paréntesis en caso de duda.
Posfijos [] . (params) expr++ expr--
Operadores unarios ++expr --expr +expr -expr ~ !
Creación y "cast" new (type)
Multiplicativos * / %
Aditivos + -
Desplazamiento << >> >>>
Relacionales < > <= >= instanceof
Igualdad == !=
AND bit a bit &
OR exclusivo bit a bit ^
OR inclusivo bit a bit |
AND lógico &&
OR lógico ||
Condicional ? :
Asignación = += -= *= /= %= ^= &= |= <<= >>= >>>=
Algunos ejemplos:
[] define arreglos: int lista[];
(params) es la lista de parámetros cuando se llama a un método: convertir(valor, base);
new permite crear una instancia de un objeto: new Contador();
(type) cambia el tipo de una expresión a otro: (float)(total % 10);
>> desplaza bit a bit un valor binario: base >> 3;
<= devuelve "true" si un valor es menor o igual que otro: total <= maximo;
instanceof devuelve "true" si el objeto es una instancia de la clase: papa instanceof Comida;
|| devuelve "true" si cualquiera de las expresiones es verdad: (a<5) || (a>20)

Llamadas a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis, como se ve, entre otros, en el ejemplo en negrita:

```
		// Archivo:  Complejo.java
		// Compilar con:		javac Complejo.java
		public final class Complejo extends Number {
		// atributos:
				private float x;
				private float y;
		// constructor:
public Complejo(float  rx, float iy) {
		x = rx;
		y = iy;
}
		// métodos:
```

```
                        public float Norma() {
                                return (float)Math.sqrt(x*x+y*y);
                        }
                        // obligatorios (son abstractos en Number):
public double doubleValue() {
        return (double)Norma( );
}
public float floatValue() {
        return Norma();
}
public int intValue() {
        return (int)Norma();
}
public long longValue() {
        return (long)Norma();
}
public String toString() {
        return "("+x+")+i("+y+")";
}
        }
```

Pueden probar la clase (mínima) con el siguiente ejemplo de aplicación; la línea en negrita es un ejemplo de un llamado a un método de un objeto de otra clase. Notar que es este caso, es necesario llamar al método sobre un objeto (instancia) existente, por lo que se indica:

Nombre_del_Objeto<punto>Nombre_del_Método(parámetros)

```
// Archivo:  Ejemplo4.java
// Compilar con: javac Ejemplo4.java
// Ejecutar con: java Ejemplo4
import java.io.*;

public class Ejemplo4 {
public static void main(String args[]) {
Complejo numComp = new Complejo(4,-3);
System.out.println(numComp.toString());
System.out.println(numComp.Norma());
}
}
```

En la clase Complejo tenemos también un ejemplo de un llamado a un método *de clase*, o sea static:
return (float)**Math.sqrt(x*x+y*y)**;
Como el método es de clase, no hace falta llamarlo para un objeto en particular. En ese caso, en lugar del nombre de un objeto existente se puede utilizar directamente el nombre de la clase:
Nombre_de_la_Clase<punto>Nombre_del_Método(parámetros)

Las estructuras de control

Las estructuras de control en Java son básicamente las misma que en C, con excepción del **goto**, que no existe (al fin un lenguaje serio!)

if…[else]

La más común de todas, permite ejecutar una instrucción (o secuencia de instrucciones) si se da una condición dada (o, mediante la cláusula *else*, ejecutar otra secuencia en caso contrario).

if (expresión_booleana) instrucción_si_true;
[else instrucción_si_false;]

o bien:

```
if (expresión_booleana) {
        instrucciones_si_true;
}
else {
        instrucciones_si_false;
}
```

Por ejemplo:

```
public final String toString() {
        if (y<0)
                return x+"-i"+(-y);
        else
                return +x+"+i"+y;
}
```

Switch…case…brake…default

Permite ejecutar una serie de operaciones para el caso de que una variable tenga un valor entero dado. La ejecución saltea todos los *case* hasta que encuentra uno con el valor de la variable, y ejecuta desde allí hasta el final del *case* o hasta que encuentre un *break*, en cuyo caso salta al final del *case*. El *default* permite poner una serie de instrucciones que se ejecutan en caso de que la igualdad no se de para ninguno de los *case*.

```
switch (expresión_entera) {
        case (valor1): instrucciones_1;
                                [break;]
        case (valor2): instrucciones_2;
                                [break;]
        …..
        case (valorN): instrucciones_N;
                                [break;]
        default: instrucciones_por_defecto;
}
```

Por ejemplo:

```
switch (mes) {
        case (2):   if (bisiesto()) dias=29;
                    else dias=31;
                    break;
case (4):
case (6):
case (9):
case (11):  dias = 30;
                    break;
default:    dias = 31;
}
```

While

Permite ejecutar un grupo de instrucciones mientras se cumpla una condición dada:

```
while (expresión_booleana) {
        instrucciones…
```

```
}
```

Por ejemplo:

```
while ( linea != null)  {
linea = archivo.LeerLinea();
System.out.println(linea);
}
```

Do…while

Similar al anterior, sólo que la condición se evalúa al final del ciclo y no al principio:

```
do {
        instrucciones…
} while (expresión_booleana);
```

Por ejemplo:

```
do {
linea = archivo.LeerLinea();
if (linea != null) System.out.println(linea);
} while (linea != null);
```

For

También para ejecutar en forma repetida una serie de instrucciones; es un poco más complejo:

```
for ( instrucciones_iniciales; condición_booleana; instruccion_repetitiva_x ) {
        instrucciones…
}
```

Si bien las instrucciones pueden ser cualquiera (el bucle se repite mientras la condición sea verdadera), lo usual es utilizarlo para "contar" la cantidad de veces que se repiten las instrucciones; se podría indicar así:

```
for ( contador = valor_inicial; contador < valor_final; contador++ ) {
        instrucciones…
}
```

Por ejemplo:

```
for ( i=0; i<10; i++ ) {
        System.out.println( i );
}
```

o, para contar hacia atrás:

```
for ( i=10; I>0; I-- ) {
        System.out.println( i );
}
```

Break y continue

Estas instrucciones permiten saltar al final de una ejecución repetitiva (break) o al principio de la misma (continue).
Por ejemplo, en:

```
import java.io.*;
class Bucles {
```

```
   public static void main (String argv[ ]) {
            int i=0;
for (i=1; i<5; i++) {
            System.out.println("antes "+i);
            if (i==2) continue;
            if (i==3) break;
            System.out.println("después "+i);
}
   }
}
```

La salida es:

```
antes 1
después 1
antes 2
antes 3
```

Por qué? "i" comienza en 1 (imprime "antes" y "después"); cuando pasa a 2, el *continue* salta al principio del bucle (no imprime el "después"). Finalmente, cuando "i" vale 3, el *break* da por terminado el bucle *for*.

Otras...

Hay otras instrucciones que controlan el flujo del programa:

- synchronized (para ver junto con los threads)
- catch,
- throw,
- try,
- finally (para ver con las excepciones)

Ahora sí, podemos usar todo nuestro conocimiento sobre Java para ir creando algunas aplicaciones y de paso ir viendo las bibliotecas estándar...

Hagamos algo...

Bueno, vamos a hacer una pequeña aplicación para practicar un poco.
Para empezar, vamos a desarrollar un poquito una clase para trabajar con números complejos.

La clase Complejo

```
// grabar como Complejo.java
// compilar con "javac Complejo.java"
public final class Complejo extends Number {

        // atributos:
                private float x;
                private float y;

        // constructores:
        public Complejo() {
                x = 0;
                y = 0;
        }
        public Complejo(float  rx, float iy) {
                x = rx;
                y = iy;
        }

        // métodos:
        // Norma
        public final float Norma() {
                return (float)Math.sqrt(x*x+y*y);
        }
        public final float Norma(Complejo c) {
                return (float)Math.sqrt(c.x*c.x+c.y*c.y);
        }
        // Conjugado
        public final Complejo Conjugado() {
                Complejo r = new Complejo(x,-y);
                return r;
        }
        public final Complejo Conjugado(Complejo c) {
                Complejo r = new Complejo(c.x,-c.y);
                return r;
        }
        // obligatorios (son abstractos en Number):
        public final double doubleValue() {
                return (double)Norma();
        }
        public final float floatValue() {
                return Norma();
        }
        public final int intValue() {
                return (int)Norma();
        }
        public final long longValue() {
                return (long)Norma();
        }
        public final String toString() {
                if (y<0)
                        return x+"-i"+(-y);
```

```
                else
                        return x+"+i"+y;
        }
        // Operaciones matemáticas
        public static final Complejo Suma(Complejo c1, Complejo c2) {
                return new Complejo(c1.x+c2.x,c1.y+c2.y);
        }
        public static final Complejo Resta(Complejo c1, Complejo c2) {
                return new Complejo(c1.x-c2.x,c1.y-c2.y);
        }
        public static final Complejo Producto(Complejo c1, Complejo c2) {
                return new Complejo(c1.x*c2.x-c1.y*c2.y,c1.x*c2.y+c1.y*c2.x);
        }
        // Nos va a venir bien para aprender excepciones...
        // como división por cero!
        public static final Complejo DivEscalar(Complejo c, float f) {
                return new Complejo(c.x/ f,c.y/f);
        }
        public static final Complejo Cociente(Complejo c1, Complejo c2) {
                float x = c1.x*c2.x+c1.y*c2.y;
                float y = -c1.x*c2.y+c1.y*c2.x;
                float n = c2.x*c2.x+c2.y*c2.y;
                Complejo r = new Complejo(x,y);
                return DivEscalar(r,n);
        }
}
```

Podemos hacer algunos comentarios...
Primero: no hay include aquí, ya que la única biblioteca que usamos es **java.lang** y se incluye automáticamente.
Segundo: la clase es public final, lo que implica que cualquier clase en éste u otros paquetes puede utilizarla, pero ninguna clase puede heredarla (o sea que es una clase estéril...).
Hagamos un resumen de los atributos y métodos de la clase:

```
        // atributos:
                private float x;
                private float y;
```

Siendo privados, no podemos acceder a ellos desde el exterior. Como además la clase es final, no hay forma de acceder a x e y. Además, al no ser static, cada instancia de la clase tendrá su propio x e y.

```
        // constructores:
        public Complejo()
        public Complejo(float  rx, float iy)
```

La clase tiene dos constructores, que se diferencian por su "firma" (*signature*), o sea por la cantidad y tipo de parámetros. El primero nos sirve para crear un objeto de tipo **Complejo** y valor indefinido (aunque en realidad el método lo inicializa en cero); con el segundo, podemos definir el valor al crearlo.

```
        // métodos:
        public final float Norma()
        public final float Norma(Complejo c)
        public final Complejo Conjugado()
        public final Complejo Conjugado(Complejo c)
```

Estos métodos también son duales; cuando los usamos sin parámetros devuelven la norma o el conjugado del objeto individual (instancia):

```
v = miComplejo.Norma();                          // por ejemplo
                    otroComplejo = miComplejo.Conjugado();
```
Con parámetros, en cambio, devuelven la norma o el conjugado del parámetro:

```
                    v = unComplejo.Norma(miComplejo);
                    otroComplejo = unComplejo.Conjugado(miComplejo);
```
Notar que lo siguiente es inválido:
```
otroComplejo = Complejo.Norma(miComplejo); // NO SE PUEDE!
```
...porque el método no es *static*, por lo tanto **debe** llamarse para una instancia en particular (en este caso, *unComplejo*).

```
        // obligatorios (son abstractos en Number):
        public final double doubleValue()
        public final float floatValue()
        public final int intValue()
        public final long longValue()
```
Estos métodos es obligatorio definirlos, ya que en la clase madre *Number* son métodos abstractos, o sea que debemos implementarlos aquí.
Como todos los métodos de esta clase son final, o sea que no puede ser redefinido. No es importante en realidad puesto que la clase no puede tener descendientes...

```
        public final String toString()
```
Este método nos sirve para representar el complejo como una cadena de caracteres, de la forma x+iy.

```
        // Operaciones matemáticas
        public static final Complejo Suma(Complejo c1, Complejo c2)
        public static final Complejo Resta(Complejo c1, Complejo c2)
        public static final Complejo Producto(Complejo c1, Complejo c2)
        public static final Complejo DivEscalar(Complejo c, float f)
        public static final Complejo Cociente(Complejo c1, Complejo c2)
```
Aquí definimos varias operaciones matemáticas. Notar que se han definido como static, o sea que los métodos son únicos independientemente de las instancias. Esto permite que los podamos ejecutar sobre una instancia o directamente sobre la clase:

```
miComplejo = unComplejo.Suma(comp1,comp2);       // vale
miComplejo = Complejo.Suma(comp1,comp2);         // TAMBIEN VALE!
```
Por ejemplo, la siguiente aplicación nos muestra cómo podemos usar algunos de estos métodos:

```
// Archivo:  Ejemplo5.java
// Compilar con: javac Ejemplo5.java
// Ejecutar con: java Ejemplo5
import java.io.*;

public class Ejemplo5 {

  public static void main(String args[]) {
    Complejo c1 = new Complejo(4,-3);
    System.out.println(c1+"\tNorma="+c1.Norma());
    Complejo c2 = new Complejo(-2,5);
    System.out.println(c2+"\tNorma="+c2.Norma()+"\n");

    System.out.println("("+c1+")/4 :"+Complejo.DivEscalar(c1,4));
    System.out.println("Suma  : "+Complejo.Suma(c1,c2));
```

```
    System.out.println("Resta : "+Complejo.Resta(c1,c2).toString());
    System.out.println("Multip: "+Complejo.Producto(c1,c2).toString());
    System.out.println("Divis : "+Complejo.Cociente(c1,c2).toString());
  }
}
```

Hay varias cosas para notar: por ejemplo, que podemos declarar las variables a la vez que las creamos:

```
Complejo c1 = new Complejo(4,-3);
```

c1 y c2 son dos objetos (instancias) de la clase Complejo.
Notar también que no hace falta poner para imprimir:

```
System.out.println(c1.toString().....);
```

ya que println automáticamente usa el método toString() de la clase para imprimir. Basta con poner c1, como en el programa, aunque c1.toString() también es válido.
También se ve el uso de los métodos static, accediéndolos directamente por la clase, en:

```
System.out.println("Suma  : "+Complejo.Suma(c1,c2));
```

Y tampoco aquí usamos toString(), aunque no está mal si se usa Complejo.Suma(c1,c2).toString().

Algo sobre los métodos

Analicemos un poco ahora cómo implementamos los métodos de la clase *Complejo*.

```
public final int intValue() {
        return (int)Norma();
}
```

Ya que no podemos convertir así nomás un complejo en un entero, para implementar estos métodos hemos elegido usar como valor de retorno la norma del complejo. En este caso, y dado que el método Norma() devuelve un float, usamos *typecasting*, es decir, lo convertimos en entero precediéndolo con (int).

```
public final String toString() {
        if (y<0)
                return x+"-i"+(-y);
        else
                return x+"+i"+y;
}
```

Aquí representamos el complejo en forma de cadena de caracteres. Hemos usado el *if* para representar adecuadamente el signo de la parte imaginaria. Noten también la asombrosa ayuda que nos brinda Java, al convertir automáticamente las variables x e y a String para la concatenación (mediante el signo "+")!

```
public static final Complejo Cociente(Complejo c1, Complejo c2) {
        float x = c1.x*c2.x+c1.y*c2.y;
        float y = -c1.x*c2.y+c1.y*c2.x;
        float n = c2.x*c2.x+c2.y*c2.y;
        Complejo r = new Complejo(x,y);
        return DivEscalar(r,n);
}
```

Aquí tengan en cuenta que las variables x e y, definidas como *float*, no tienen nada que ver con las variables (atributos) de la clase que están definidas al principio de la misma, sino que son variables locales al método.
Podemos usar return DivEscalar(r,n), ya que DivEscalar es un método propio de la clase; no hace falta poner Complejo.DivEscalar.
Qué pasa con r, el new Complejo(x,y) que creamos? Nada; cuando un objeto no se usa más, el "recogedor de basura" de Java lo elimina automáticamente (tarde o temprano) de la memoria.

```
public final float Norma(Complejo c) {
```

```
            return (float)Math.sqrt(c.x*c.x+c.y*c.y);
        }
```

Aquí estamos usando otra clase, **Math**, que nos permite realizar varias operaciones matemáticas. Esta clase dispone de las constantes E y PI, y los métodos:

abs(x)	valor absoluto
acos(x)	arco coseno
asin(x)	arco seno
atan(x)	arco tangente
atan2(x,y)	componente angular de la representación polar de x,y
ceil(x)	menor entero mayor que x
cos(x)	coseno
exp(x)	e^x
floor(x)	mayor entero menor que x
IEEEremainder(x,y)	resto de la división x/y según el estándar IEEE 754
log(x)	logaritmo natural
max(x,y)	el mayor de x e y
min(x,y)	el menor de x e y
pow(x,y)	x^y
random()	número aleatorio entre 0 y 1
rint(x)	entero más cercano a x (devuelve un doble)
round(x)	entero más cercano a x (devuelve un entero o un long)
sin(x)	seno
sqrt(x)	raíz cuadrada
tan(x)	tangente

Algunos de estos métodos disparan *excepciones*, como sqrt o log de números negativos. Más adelante veremos cómo se usan las excepciones.

Otra clase que hemos estado usando mucho es la PrintStream, a la que pertenece el método println. En

```
            System.out.println(...)
```

out es un *atributo* de la clase System, del tipo (clase) PrintStream:

```
public final class System extends Object
{
    // Fields
  public static PrintStream err;
  public static InputStream in;
  public static PrintStream out;

    // Methods
.............
}
```

Veremos otras bibliotecas (para entrada/salida, gráficos, etc) muy pronto.

Java a través de la ventana

Para hacer algo un poco más divertido, vamos a empezar a trabajar con la biblioteca java.awt, que es la que contiene todo un grupo de objetos para trabajar con ventanas y sus contenidos: botones, listas, etc.

Nuestra primera ventana

En Java, la clase **Window** (descendiente de **Container**), en la biblioteca *java.awt*, permite implementar ventanas "peladas", es decir, sin bordes ni menús. Son la base para cualquier tipo de ventanas (normales, pop-up, diálogos, etc.). El otro descendiente de **Container**, **Panel**, es más sencillo aún y sirve como espacio para que una aplicación incorpore dentro suyo otros elementos (incluyendo otros paneles).
La interface Java dirige tanto a uno como a otro todos los eventos de teclado, mouse y foco que los afecten (en seguida veremos cómo usar estos eventos).
De la clase **Window** descienden **Dialog** (para implementar diálogos) y **Frame**, que es una ventana algo más completa: ya tiene borde y menú, así como los botones de cerrar, maximizar, etc.
El siguiente ejemplo crea una ventana que no hace nada pero contiene varios elementos; se puede usar directamente (desde la ventana DOS o Unix con *java Ejemplo7*) o como applet dentro de una página HTML. Si bien los elementos no disparan ninguna acción, se pueden utilizar con toda su funcionalidad (por ejemplo, editar el texto dentro de los cuadros de texto o presionar el botón).

```
// grabar como "Ejemplo7.java"
// compilar con "javac Ejemplo7.java"
import java.awt.*;

public class Ejemplo7 extends Frame {
  boolean inAnApplet = true;

  public static void main(String args[]) {
    Ejemplo7 window = new Ejemplo7();
    window.inAnApplet = false;
    window.setTitle("Ejemplo");
    window.pack();
    window.show();
  }

  public Ejemplo7() {
            Panel panelAlto = new Panel();
            panelAlto.add("West", new Label("Cartel", Label.CENTER));
            panelAlto.add("East", new TextArea("Area de texto", 5, 20));
            add("North", panelAlto);

            Panel panelBajo = new Panel();
            panelBajo.add(new TextField("Campo de Texto"));
            panelBajo.add(new Button("Botón"));
            add("South",panelBajo);
  }

  public boolean handleEvent(Event ev) {
    if (ev.id == Event.WINDOW_DESTROY) {
      if (inAnApplet) {
```

```
            dispose();
          } else {
            System.exit(0);
          }
        }
        return super.handleEvent(ev);
    }
}
```

Un poco de detalle

La clase desciende de **Frame** (o sea que será una ventana con borde, aunque no le vamos a poner menú). Vamos a usar el flag **inAnApplet** para saber si se arrancó como applet o como aplicación standalone (hay que cerrarla en manera diferente en cada caso)

```
public class Ejemplo7 extends Frame {
   boolean inAnApplet = true;
```

Si se llama como aplicación standalone, lo primero que se ejecuta es main(...); en este caso la aplicación crea una instancia de Ejemplo7 (ejecutando el constructor Ejemplo7() a través de *new*), define que **no** es un applet, y llama a tres métodos de la "abuela" window:

- setTitle que define cuál va a ser el título que aparece en la ventana
- pack que dimensiona los elementos que componen la ventana a su tamaño preferido
- show que muestra la ventana

```
public static void main(String args[]) {
      Ejemplo7 window = new Ejemplo7();
      window.inAnApplet = false;
      window.setTitle("Ejemplo");
      window.pack();
      window.show();
   }
```

Ojo! No confundir el objeto (instancia) window con la clase **Window**!
Si se carga como applet, entonces se ejecuta el constructor Ejemplo7() como en el caso anterior:

```
public Ejemplo7() {
                Panel panelAlto = new Panel();
                panelAlto.add("West", new Label("Cartel", Label.CENTER));
                panelAlto.add("East", new TextArea("Area de texto", 5, 20));
                add("North", panelAlto);

                Panel panelBajo = new Panel();
                panelBajo.add(new TextField("Campo de Texto"));
                panelBajo.add(new Button("Botón"));
                add("South",panelBajo);
   }
```

Este constructor define dos paneles que forman el contenido de la ventana (panelAlto y panelBajo), los llena con un par de componentes y los pone dentro de la ventana (recordar que Ejemplo7 es una ventana!).
Para verlo más claro, se crea el panel (o espacio para contener objetos) con:

```
                Panel panelAlto = new Panel();
```

Se agregan componentes al panel con el método add:

```
panelAlto.add("West", new Label("Cartel", Label.CENTER));
panelAlto.add("East", new TextArea("Area de texto", 5, 20));
```

Se agregan el panel dentro de nuestro objeto con:

```
add("North", panelAlto);
```

que equivale a:

```
this.add("North", panelAlto);
```

lo que se puede ver (aunque es inválido porque la clase no es static) como:

```
Ejemplo7.add("North", panelAlto);
```

Como nuestra clase Ejemplo7 desciende de Frame, ésta de Window, y ésta de Container, el método *add* lo está heredando de... su bisabuela! Por otra parte, Panel es hija de Container, y usa el mismo método para agregar sus componentes. Interesante, no? Veamos la estructura:

```
Object --- Component --- Container --+-- Panel
                                     |
                                     +-- Window --- Frame --- Ejemplo7
```

Noten que hemos usado dos métodos add con diferente *signature*:

```
panelAlto.add("West", new Label("Cartel", Label.CENTER));
.........
panelBajo.add(new Button("Botón"));
```

El método add(Component) agrega un componente al final; el método add(String,Component) lo agrega en un lugar especificado por una palabra que depende del *LayoutManager*, el objeto que se encarga de ordenar los componentes dentro del contenedor.
LayoutManager es una *interface*, y como tal debe implementarse a través de objetos no abstractos de los que hay varios predefinidos en la librería java.awt: BorderLayout, CardLayout, FlowLayout, GridBagLayout y GridLayout.
El Layout por defecto es BorderLayout, que define en el contenedor las áreas "North", "South", "West", "East" y "Center" y es que usamos aquí. CardLayout permite "apilar" los componentes como cartas y ver uno por vez, FlowLayout los ordena de izquierda a derecha como un texto, GridLayout los ordena en una cuadrícula donde cada componente tiene un tamaño fijo y GridBagLayout los pone en una cuadrícula pero cada uno puede tener el tamaño deseado.
Noten que no hace falta llamar, en el caso del applet, a Pack() y Show().

Y los eventos...

Ahora vamos a ver un método que viene de la clase tatarabuela! Hace falta decir que me gusta esto de los objetos?
Vamos a redefinir handleEvent(Event), que es el método que analiza los eventos dirigidos al componente y toma las acciones adecuadas.
La clase Event define básicamente una serie de métodos que permiten saber si hay alguna tecla de control presionada y muchas constantes que indican si se presionó o movió el mouse, si se presionó alguna tecla en particular, si se cambió el tamaño de la ventana, etc. En particular hay algunos atributos interesantes:

- id que indica el tipo de evento
- target que indica sobre qué componente se produjo el evento
- key qué tecla se presionó si fue un evento de teclado

etc.

En los descendientes de Component, el método handleEvent se llama automáticamente cada vez que se produce un evento sobre el componente. En este caso, simplemente vamos a mirar si el evento (sobre nuestro objeto de clase Ejemplo7) fue "cerrar la ventana", que se identifica mediante event.id = WINDOW_DESTROY (una constante estática de la clase Event, y como tal la podemos usar con el nombre de la clase como Event.WINDOW_DESTROY):

```
public boolean handleEvent(Event ev) {
      if (ev.id == Event.WINDOW_DESTROY) {
         if (inAnApplet) {
            dispose();
         } else {
            System.exit(0);
         }
      }
      return super.handleEvent(event);
   }
```

En ese caso, si nuestro ejemplo se disparó como aplicación llamamos al método System.exit(0), que cierra la aplicación; y si era un applet llamamos a dispose(), implementación de un método de la interface ComponentPeer que se encarga de remover todos los componentes y la propia ventana.
Noten que cualquier otro tipo de evento deja seguir hasta return super.handleEvent(event), que llama al método handleEvent de la clase madre: así como el prefijo this. se refiere a un método de la propia clase, el prefijo super. llama al método de la clase madre (aunque esté redefinido). En este caso, la llamada se remonta hasta Component.handleEvent, que determina el tipo de evento y llama a uno de los métodos action, gotFocus, lostFocus, keyDown, keyUp, mouseEnter, mouseExit, mouseMove, mouseDrag, mouseDown o mouseUp según sea apropiado (y devuelve true). Si ningún método es aplicable, devuelve false.
Es muy común, al redefinir un método, tener en cuenta llamar antes o después al método de la clase antecesora para inicializar o terminar alguna tarea.

Una ventana con vida

Antes que nada, vamos a crear una página HTML para cargar nuestra clase Ejemplo8, que será un applet (aunque también la podremos ejecutar en forma standalone con "java Ejemplo8"), por ejemplo:

```
<!-- Archivo Ejemplo8.htm - HTML de ejemplo -->
<HTML>
<HEAD>
<TITLE>Ejemplo 8 - Ventana de datos</TITLE>
</HEAD>
<BODY>
Aqu&iacute; se tiene que abrir una ventana de entrada de datos
<applet code="Ejemplo8.class" width=170 height=150>
</applet>
</BODY>
</HTML>
```

Nuestro applet será muy sencillo, ya que utilizará clases que iremos definiendo en este capítulo; por empezar sólo creará una ventana que definiremos en la clase Ventana8:

```
// Archivo: Ejemplo8.java
// Compilar con "javac Ejemplo8.java"

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

public class Ejemplo8 extends Applet {

public static void main (String arg[]) {          // para poder llamarla con "java Ejemplo8"
new Ventana8("Ejemplo Standalone", true);
}

public void init() {                              // se ejecuta al abrirse un applet
new Ventana8("Ejemplo Applet", false);
}

}
```

Con los parámetros que le pasamos a la clase Ventana8 le indicamos el título de la ventana y si se carga como applet o no (ya que el método de cierre varía).

Viajando con Java

Ahora vamos a trabajar con nuestra clase Ventana8, una ventana que nos permita seleccionar una fecha y dos ciudades (desde y hasta) que simula una ventana de compra de pasajes de, por ejemplo, una terminal de ómnibus.
El ejemplo está basado en uno del libro "Programación Java" de Macary y Nicolas, aunque algo mejorado y ampliado.
En nuestra ventana podremos entrar una fecha a mano o directamente mediante los botones Hoy y Mañana, elegiremos la ciudad de salida y la de llegada de dos listas, y presionaremos luego un botón que nos mostrará los servicios disponibles, nos permitirá comprar los pasajes, etc.
A medida que entramos los datos, en el botón se irá mostrando el detalle de lo que se fue seleccionando.

Nuestra ventana quedará más o menos así:

Empecemos por armar la estructura de la clase Ventana8:

```
import java.awt.*;

class Ventana8 extends Frame {      // hija de Frame

// aquí agregaremos luego
// algunas variables para guardar datos
// (ciudades de salida y llegada, fecha)
button                  ok;                     // también el botón de compra de pasajes
boolean     enApplet;   // y otra para indicar si es un applet o no

Ventana8 (String titulo, boolean enApplet) {     // un constructor
super(titulo);                                                  // llama al de Frame
this.enApplet = enApplet;                       // guardamos esto
// aquí crearemos los botones, listas, etc
// con sus valores iniciales
// y los pondremos en la ventana.
// por ejemplo:
ok = new Button("Viaje: de ? a ? el ?/?/?");
add("South",ok);
pack();                                                         // dimensionamos la ventana
show();                                                         // y la mostramos!
}

public boolean handleEvent(Event e) {                   // para manejar los eventos
if (e.id == Event.WINDOW_DESTROY) {     // cerrar la ventana
if (enApplet) dispose();
else System.exit(0);
}
// aquí miraremos si se presionó un botón
// o se eligió algo de una lista
// y actuaremos en consecuencia
return super.handleEvent(e);            // los demás eventos los maneja Frame
}

void ActualizaBoton() {
                // aquí pondremos un método que servirá
                // para actualizar el botón de compra de pasajes,
                // ya que el texto del mismo se actualiza cada
                // vez que se selecciona una ciudad o se cambia la fecha
}

void Activar() {
                // y aquí un método para cuando se presione
                // dicho botón, que se supone que va a consultar
                // una base de datos y abrir una ventana
```

```
// para vendernos el pasaje
}
}
```

Nuestro programa ya funciona! Aunque un poquito incompleto, claro...
Igual vamos a analizarlo un poco el constructor, que es lo más interesante aquí.
Primero llamamos al constructor de la clase madre, que se encarga de crear la ventana:

```
Ventana8 (String titulo, boolean enApplet) {      // un constructor
super(titulo);                                    // llama al de Frame
```

Esto sería como llamar a super.Frame(titulo), o bien Frame(titulo), ya que el método constructor tiene el mismo nombre de la clase. Luego, con:

```
this.enApplet = enApplet;                         // guardamos esto
```

asignamos a nuestra variable enApplet de la clase el valor del parámetro que se pasó al constructor, que se llama igual. El prefijo this, que se refiere a la instancia particular de la clase, permite diferenciar uno de otro (esto es válido tanto para variables como para métodos).

```
ok = new Button("Viaje: de ? a ? el ?/?/?");
add("South",ok);
```

Aquí hemos creado un botón ubicado al pie de la ventana (por ahora lo único que pusimos), y luego dimensionamos la ventana y la mostramos:

```
pack();                                   // dimensionamos la ventana
show();                                   // y la mostramos!
```

Preparando listas

Ahora vamos a empezar a crear otros objetos para ir completando nuestra aplicación. Comencemos con las listas de ciudades.
Para eso, vamos a crear un objeto descendiente de Panel que simplemente contenga una lista de ciudades predefinidas y un título que diga "Seleccione ciudad de", y a continuación "salida" o "llegada".
También agregaremos un método

```
import java.awt.*;

class SelecPueblo extends Panel {
private List listaPueblos;

SelecPueblo (String salidaOllegada) {
setLayout (new BorderLayout (20,20));

// armamos el título, que va a ser un Label:
StringBuffer titulo = new StringBuffer();
titulo.append("Seleccione ciudad de ");
titulo.append(salidaOllegada);
titulo.append(": ");
add("North", new Label(titulo.toString()));

// armamos la lista de ciudades, que va a ser un List:
listaPueblos = new List (4, false);
```

```
listaPueblos.addItem("Buenos Aires");
listaPueblos.addItem("La Plata");
listaPueblos.addItem("Azul");
listaPueblos.addItem("Rosario");
listaPueblos.addItem("Cordoba");
listaPueblos.addItem("Bahía Blanca");
add("South", listaPueblos);
}

public String getDescription() {
return listaPueblos.getSelectedItem();
}
}
```

No hay mucho para analizar aquí, creo. La variable listaPueblos es privada, pero puede consultarse cuál es la ciudad seleccionada mediante getDescription (que es public). Este método llama al método getSelectedItem de la lista, que devuelve el texto seleccionado.
En el constructor, armamos el texto del título como un StringBuffer. Los objetos StringBuffer son similares a los de clase String pero pueden ser modificados. En cambio los objetos String, una vez creados, no pueden ser modificados directamente: sus métodos (concat, toLowerCase, etc.) simplemente crean un nuevo String con el nuevo valor.
Esto lo hicimos para introducir esta nueva clase; por supuesto hubiera sido más fácil poner, como pueden comprobar, con el mismo resultado:

```
String tit = "Seleccione ciudad de "+salidaOllegada+": ";
add("North", new Label(tit));
```

Por otra parte, creamos el objeto listaPueblos como new List(4, false), que indica que la lista va a tener 4 renglones y sólo se puede seleccionar un ítem por vez. Agregamos luego 6 ítems mediante addItem y la agregamos al panel.
Ahora ya podemos agregar las listas a nuestra ventana y poner un par de variables para guardarlas:

```
class Ventana8 extends Frame {      // hija de Frame

SelecPueblo          cs;             // ciudad de salida
SelecPueblo          cl;             // ciudad de llegada
button               ok;             // también el botón de compra de pasajes
boolean     enApplet; // y otra para indicar si es un applet o no

Ventana8 (String titulo, boolean enApplet) {     // un constructor
super(titulo);                                   // llama al de Frame
this.enApplet = enApplet;                        // guardamos esto
cs = new SelecPueblo("SALIDA");                  // CIUDAD DE SALIDA
add ("Center", cs);
cl = new SelecPueblo("LLEGADA");                 // CIUDAD DE LLEGADA
add ("East", cl);
ok = new Button("Viaje: de ? a ? el ?/?/?");
add("South",ok);
pack();                                          // dimensionamos la ventana
show();                                          // y la mostramos!
}
```

.....
Ya pueden ir probando cómo queda, aunque por ahora mucha funcionalidad no tenemos...

Agregando fechas

Otro panel más nos servirá para seleccionar o entrar la fecha:

```
import java.util.*;
import java.awt.*;

class DiaPartida extends Panel {
private TextField       elDia;
private Button                  hoy;
private Button                  diasiguiente;

DiaPartida() {
setLayout (new GridLayout (4,1));
elDia = new TextField();
elDia.setText(GetHoy());
hoy = new Button ("Hoy");
diasiguiente = new Button ("Mañana");
add (new Label ("Día salida: "));
add (elDia);
add (hoy);
add (diasiguiente);
}

private String GetHoy() {
Date d = new Date();
int dia = d.getDate();
int mes = d.getMonth();
int ano = d.getYear();
return dia+"/"+mes+"/"+ano;
}

private String GetManana() {
Date d = new Date();
int dia = d.getDate();
int mes = d.getMonth();
int ano = d.getYear();
dia = dia++;
switch (mes) {
case (1):
        case (3):
        case (5):
        case (7):
        case (8):
        case (10): if (dia>31) {
                        dia = 1;
                                mes++;
}
                break;
        case (12): if (dia>31) {
                                dia = 1;
                        mes = 1;
                                ano++;
                                }
```

```
                                                break;
case (4):
                case (6):
                case (9):
                case (11): if (dia>30) {
                                                                dia = 1;
                                        mes++;
                                                }
                                                break;
                default: if (dia>28) {     // ojo, hay que corregir para bisiestos!
                                                                dia = 1;
                                        mes++;
                                                }
}
return dia+"/"+mes+"/"+ano;
}

public String getDescription() {
return elDia.getText();
}

public boolean handleEvent (Event e) {
if (e.target == hoy)
elDia.setText(GetHoy());
if (e.target == diasiguiente)
elDia.setText(GetManana());
return super.handleEvent(e);
}

}
```

Este es un poco más largo pero no más complejo. Vamos por parte:

```
DiaPartida() {
setLayout (new GridLayout (4,1));
elDia = new TextField();
elDia.setText(GetHoy());
hoy = new Button ("Hoy");
diasiguiente = new Button ("Mañana");
add (new Label ("Día salida: "));
add (elDia);
add (hoy);
add (diasiguiente);
}
```

El constructor crea un panel con cuatro campos en forma de grilla vertical, donde mostrará el texto "Día salida: ", el campo de entrada de texto elDia y los botones hoy y diasiguiente.
El método privado getHoy usa los métodos getDate, getMonth y getYear de la clase date para armar un String con la fecha actual. El método privado getManana hace lo mismo para leer la fecha actual, y le suma 1 al día para tener el día siguiente. El switch siguiente verifica que si pasó de fin de mes tome el primer día y el mes siguiente (o el primer día del año siguiente si es en diciembre). Notar que no se consideraron los años bisiestos en febrero para no complicar el método, pero no es difícil de corregir.

Otra manera sería armar un *array* con los días de cada mes, corregir los días de febrero para los años bisiestos, y comparar contra este *array* en lugar de usar un *switch*. La idea siempre es la misma: devolver un String con la fecha del día siguiente.
Notar algo interesante: como estas clases se cargan y ejecutan en la máquina *cliente*, la fecha que aparece es la del *cliente* y no la del *servidor* (que puede ser diferente depende la hora y el lugar del mundo en que estén ambas máquinas).
El método getDescription es público y se usa para acceder a la fecha que se ha ingresado desde las demás clases; simplemente devuelve el contenido del campo elDia, de clase *TextField*.
Aquí hemos desarrollado también el método handleEvent:

```
public boolean handleEvent (Event e) {
if (e.target == hoy)
elDia.setText(GetHoy());
if (e.target == diasiguiente)
elDia.setText(GetManana());
return super.handleEvent(e);
}
```

En caso de alguna acción sobre uno de los botones, el método setText (de la clase *TextField*) pone en el campo de texto elDia el valor del día actual o el siguiente.
Notar que sólo hemos considerado que haya algún evento y no un tipo de evento en particular; en realidad el método va a actuar por ejemplo tanto al presionar el mouse sobre el botón como al soltarlo. Pero esto no nos molesta.
super.handleEvent se encarga de otros eventos dirigidos al panel, como la entrada de datos por teclado al campo de texto por ejemplo.

Juntando todo hasta aquí

Bueno, ahora vamos a reunir las piezas que tenemos hasta ahora agregando estos métodos a nuestra clase Ventana8 para ver cómo queda la ventana completa:

```
class Ventana8 extends Frame {      // hija de Frame

SelecPueblo          cs;                    // ciudad de salida
SelecPueblo          cl;                    // ciudad de llegada
DiaPartida dp;                    // día de salida
button               ok;                    // botón de compra de pasajes
boolean      enApplet;  // para indicar si es un applet o no

Ventana8 (String titulo, boolean enApplet) {     // un constructor
super(titulo);                                          // llama al de Frame
this.enApplet = enApplet;                    // guardamos esto
dp = new DiaPartida();                                  // DIA DE SALIDA
add ("West", dp);
cs = new SelecPueblo("SALIDA"); // CIUDAD DE SALIDA
add ("Center", cs);
cl = new SelecPueblo("LLEGADA");             // CIUDAD DE LLEGADA
add ("East", cl);
ok = new Button("Viaje: de ? a ? el ?/?/?");
add("South",ok);
pack();                                                 // dimensionamos la ventana
show();                                                 // y la mostramos!
}
```

Completando la ventana

Vamos a empezar por completar nuestro método ActualizaBoton, que modificará el texto del botón ok a medida que seleccionemos las ciudades y la fecha:

```
void ActualizaBoton() {
   StringBuffer b = new StringBuffer("Viaje: de ");
   if (cs.getDescription() != null) b.append(cs.getDescription());
   else b.append("?");
   b.append(" a ");
   if (cl.getDescription() != null) b.append(cl.getDescription());
   else b.append("?");
   b.append(" el ");
   if (dp.getDescription() != null) b.append(dp.getDescription());
   else b.append("?/?/?");
   ok.setLabel(b.toString());
 }
```

Nuestro método comienza por crear un StringBuffer con las palabras "Viaje: de ", y va agregando el resto:

- la ciudad de partida, llamando al método getDescription de cs (ciudad de salida)
- el texto constante " a "
- la ciudad de llegada, llamando al método getDescription de cl (ciudad de llegada)
- el texto constante " el "
- la fecha seleccionada, llamando al método getDescription de dp (día de partida)

Si en cualquier caso recibe un string nulo, pone un signo de pregunta (o ?/?/? para la fecha).
El método setLabel, sobre el objeto ok de tipo *Label*, modifica la "etiqueta" del botón.
Realmente nos devuelven null los métodos que llamamos si no hay selección hecha?
Veamos:

```
class SelecPueblo extends Panel {
private List listaPueblos;
.....
public String getDescription() {
return listaPueblos.getSelectedItem();
}
}
```

El método getSelectedItem de la clase *List* devuelve null si no hay ítems seleccionados, así que acá andamos bien. En cuanto a la clase DiaPartida, de entrada inicializa el valor del texto en la fecha actual, así que aquí no se daría nunca este caso... Aunque al crear el objeto Ventana8 estamos poniendo un texto fijo en el botón, y no el que devuelve el objeto dp.
Sería mejor, para ser más consistente, modificar el constructor de Ventana8 para que arme el texto mediante el método ActualizaBotón:

```
Ventana8 (String titulo, boolean enApplet) {
.....
            ok = new Button("cualquiera");
            ActualizaBoton();
            add("South",ok);
            pack();
            show();
 }
```

Esto ya se ve mejor! Y de paso probamos el método...

Un poquito de actividad

Ahora sí, pasemos a completar nuestro manejador de eventos:

```
public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
                if (enApplet) dispose();
                else System.exit(0);
        }
if ( (e.target==dp)||(e.target==cs)||(e.target==cl) )
ActualizaBoton();
if (e.target==ok)
Activar();
}
return super.handleEvent(e);
}
```

Simplemente, si detectamos un evento sobre alguno de nuestros paneles actualizamos el texto del botón; y si se presiona dicho botón llamamos al método Activar que se supone que va a tomar los datos de la base de datos, indicarnos servicios disponibles, etc.
Algo importante a notar es que el simple hecho de mover el mouse sobre uno de los paneles ya llama a ActualizaBoton (se nota porque titila el texto, sobre todo en una máquina lenta). Además, si hacen click sobre el botón **Hoy** o **Mañana** *sin* mover el mouse, el texto del botón ok no se actualiza ya que el evento va dirigido al botón presionado y no al panel.
Una forma de filtrar sólo los eventos que nos interesan sería usar, por ejemplo:

```
if ((e.target=cs.listaPueblos) && (e.id==Event.LIST_SELECT)) ActualizaBoton();
```

que está dirigida a la *lista* y no al *panel* en general, y tiene en cuenta el tipo de evento.
Lamentablemente, listaPueblos es privada dentro de la clase SelecPueblo y por lo tanto dentro de cs. Pero es mejor así, porque declararla pública y leerla desde afuera sería bastante sucio (así como la leemos podríamos escribirla).
Hay varias formas de mejorar esto sin cometer la torpeza de declarar pública a listaPueblos. Una posibilidad es verificar, usando cs.getDescription(), si el texto cambió (y sólo en ese caso modificar el texto del botón). Otra, es hacer que los objetos de la clase SelecPueblo pasen a sus padres cualquier evento sobre ellos, o mejor solamente la selección de un elemento de la lista; para eso basta agregar a la clase SelecPueblo:

```
public boolean handleEvent(Event e) {
if ((e.target==listaPueblos) && (e.id==Event.LIST_SELECT)) {
e.target=this;
}
return super.handleEvent(e);
}
```

En resumen: si el evento en el panel es una selección de la lista (tanto con mouse como moviendo la selección con las flechas), cambio el target del evento para que indique el *panel* (y no la *lista*); si no, lo paso a la clase antecesora.
Lo mismo podemos hacer con handleEvent para la clase DiaPartida:

```
public boolean handleEvent (Event e) {
if (e.target == hoy) {
elDia.setText(GetHoy());
```

```
e.target=this;
}
if (e.target == diasiguiente) {
elDia.setText(GetManana());
e.target=this;
}
if (e.target == elDia) {
e.target=this;
}
return super.handleEvent(e);
}
```

Esto no anda como esperaríamos! El campo de texto no se comporta muy bien...
Esto es porque el código dependiente de la plataforma procesa los eventos de mouse *antes* de llamar a handleEvent, pero procesa los de teclado *después* de llamar a handleEvent.
Lo que significa que, en el caso del campo de texto, handleEvent (y por lo tanto ActualizaBotón) se llama *antes* de modificar el texto!
Para corregir esto, deberíamos procesar nosotros las teclas presionadas (lo que podríamos aprovechar para verificar que se presiona una tecla válida).
Cuidado! En futuras versiones de Java podría implementarse el mismo comportamiento para el mouse, y por lo tanto tendríamos que repensar la estrategia.
Para colmo, sólo los eventos que la plataforma envía llegan a Java; por ejemplo, Motif no envía eventos de movimiento de mouse dentro de un campo de texto... lo que significa que nunca podríamos capturar ese tipo de eventos. Sólo el componente **Canvas** pasa todos los eventos.
Para simplificar, sólo actualizaremos el texto del botón cuando se presiona **Enter** (Event.key=10):

```
if ((e.target == elDia)&&(e.id==Event.KEY_PRESS)) {
        if (e.key==10) e.target=this;
}
```

Ahora debemos modificar el método handleEvent en nuestra clase Ventana8 para que soporte todos estos eventos:

```
public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
                if (enApplet) dispose();
                else System.exit(0);
        }
        if ( ((e.target==dp)&&((e.id==Event.ACTION_EVENT)||(e.id==Event.KEY_PRESS)))
||((e.target==cs)&&(e.id==Event.LIST_SELECT))
||((e.target==cl)&&(e.id==Event.LIST_SELECT)) )
                        ActualizaBoton();
        if (e.target==ok)
Activar();
        return super.handleEvent(e);
}
```

Obviamente, procesar todas las teclas nosotros sería bastante más complicado... de todos modos, el método en DiaPartida sería más o menos así:

```
if ((e.target == elDia)&&(e.id==Event.KEY_PRESS)) {
        // 1- leer el contenido del campo con: elDia.getText()
        // 2- modificarlo de acuerdo a la tecla presionada: e.key
        // 3- poner el resultado en el campo con: elDia.setText(texto)
```

```
            // 4- modificar el objeto del evento al panel con: e.target=this;
            // 5- enviar el evento al objeto padre (no la clase padre),
            //     en este caso Ventana8, mediante: getParent().deliverEvent(e)
            // 6- evitar proceso posterior del evento mediante: result(true)
}
```

Me ahorro explicar estos dos últimos pasos; se complica bastante todo porque hay que manejar la posición del cursor dentro del campo de texto, etcétera. Con lo que hicimos es bastante... creo!

Y para terminar...

Bueno, sólo nos queda por definir el método Activar(). Primero vamos a llamar a ActualizaBoton() por si alguien lo último que hizo fue entrar un texto sin presionar **Enter**, y dejo para otro día más tranquilo consultar un archivo o base de datos con lo que vamos a mostrar al usuario de nuestro programa.
Por ahora simplemente vamos a mostrar una ventana con la selección y un lindo botón de OK.
Primero vamos a hacer una muy pequeña modificación a ActualizaBoton() para que nos devuelva el valor del texto del botón (para no calcularlo de nuevo):

```
String ActualizaBoton() {
   StringBuffer b = new StringBuffer("Viaje: de ");
.....................................................
   ok.setLabel(b.toString());
  }
```

Y ahora vamos a definir nuestro método, teniendo en cuenta que nuestro botón sólo actuará si se han entrado todos los datos:

```
void Activar() {
if ( (cs.getDescription() != null) && (cl.getDescription() != null) )
// también podríamos verificar que la fecha sea válida aquí
Result8 resultado = new Result8("Resultado",ActualizaBoton());
else ok.setLabel("Especificación incompleta!");
  }
```

Sólo nos falta definir una sencilla clase Result8 para nuestra ventanita resultado:

```
// archivo Result8.java, compilar con javac Result8.java
import java.awt.*;

class Result8 extends Frame {

Button       r_ok;

Result8 (String titulo, String texto) {                // constructor
super(titulo);
Label r_lbl = new Label(texto);
r_ok = new Button("Ok");
            add("Center", r_lbl);
                        add("South", r_ok);
            pack();
                        show();
}

public boolean handleEvent(Event e) {
if ((e.id == Event.WINDOW_DESTROY)||(e.target==r_ok))
```

```
dispose();    // cierra esta ventana pero no la aplicación
              return super.handleEvent(e);
}
}
```

Noten que usé dispose y no System.exit! Esto permite cerrar sólo la ventana de resultado, y seguir usando la aplicación hasta que se nos ocurra cerrarla mediante meta-F4, alt-F4, el menú de sistema de la ventana, la cruz de Windows 95 o lo que le resulte a su sistema operativo.

Finale con tutto

Espero que se haya entendido! Esta aplicación costó bastante pero en el camino hemos tenido oportunidad de aprender unas cuantas cosas... Si logran juntar todo el código y generar las varias clases que definimos, todo tiene que andar sobre rieles e independientemente de la plataforma.
Si no... avísenme, y subo también los fuentes o las clases.
Por las dudas, pueden probar esta aplicación como applet cargando:

http://www.amarillas.com/rock/java/Ejemplo8.htm

Un paréntesis de Entrada/Salida

En Java hay muchas clases para leer y escribir archivos (u otros dispositivos de E/S). Están reunidos en la biblioteca **java.io**.
Vamos a empezar como siempre con un pequeño ejemplo funcional y en seguida nos meteremos en el necesario camino de las excepciones...

Primera Lectura

```
// archivo: Ejemplo9.java - compilar con "javac Ejemplo9.java", etc. etc.
import java.io.*;

public class Ejemplo9 {
public static void main(String args[]) throws FileNotFoundException,IOException {
FileInputStream          fptr;
DataInputStream     f;
String                   linea = null;

fptr = new FileInputStream("Ejemplo9.java");
f = new DataInputStream(fptr);
do {
linea = f.readLine();
if (linea!=null) System.out.println(linea);
} while (linea != null);
fptr.close();
}
}
```

(Caramba! ¿Qué hace ese *throws* ahí?)
El programa de ejemplo simplemente lee un archivo de texto y lo muestra en pantalla, algo así como el type del DOS o el cat de Unix.
Dejemos por ahora el throws FileNotFoundException,IOException y vamos al código.

```
fptr = new FileInputStream("Ejemplo9.java");
```

La clase FileInputStream (descendiente de InputStream) nos sirve para referirnos a archivos o conexiones (sockets) de una máquina. Podemos accederlos pasando un String como aquí, un objeto de tipo File o uno de tipo FileDescriptor, pero en esencia es lo mismo. Al crear un objeto de este tipo estamos "abriendo" un archivo, clásicamente hablando.
Si el archivo no existe (por ejemplo reemplacen "Ejemplo9.java" por alguna otra cosa, como "noexiste.txt"), al ejecutarlo nos aparece un error:

```
C:\java\curso>java Ejemplo9
java.io.FileNotFoundException: noexiste.txt
        at java.io.FileInputStream.<init>(FileInputStream.java:51)
        at Ejemplo9.main(Ejemplo9.java:9)
```

(Caramba! ¿Dónde vi ese FileNotFoudException antes?)
Justamente, cuando el archivo al que quiero acceder no existe, Java "lanza" una *excepción*. Esto es, un aviso de que algo falló y, si no se toma ninguna acción, detiene el programa.
La clase FileInputStream puede "lanzar" (throws) la excepción FileNotFoundException.

¿Cómo capturar y tratar las excepciones? En seguida; primero terminemos con nuestro programa.

```
f = new DataInputStream(fptr);
```

La clase DataInputStream nos permite leer, en forma independiente del hardware, tipos de datos de una "corriente" (stream) que, en este caso, es un archivo. Es descendiente de FilterInputStream e *implementa* DataInput, una *interface*.
Al crear un objeto de tipo DataInputStream lo referimos al archivo, que le pasamos como parámetro (fptr); esta clase tiene toda una serie de métodos para leer datos en distintos formatos.
En nuestro programa usamos uno para leer líneas, que devuelve *null* cuando se llega al final del archivo o un String con el contenido de la línea:

```
do {
    linea = f.readLine();
    System.out.println(linea);
} while (linea != null);
```

En seguida de leer la línea la imprimimos, y repetimos esto mientras no nos devuelva null.
Al final, cerramos el archivo:

```
fptr.close();
```

Tanto readLine como close pueden lanzar la excepción IOException, en caso de error de lectura o cierre de archivo.
En realidad, podríamos no haber usado un DataInputStream y trabajar en forma más directa:

```
import java.io.*;

public class Ejemplo10 {
public static void main(String args[]) throws FileNotFoundException,IOException {
FileInputStream                 fptr;
int                             n;

fptr = new FileInputStream("Ejemplo9.java");
do {
n = fptr.read();
if (n!=-1) System.out.print((char)n);
} while (n!=-1);
fptr.close();
}
}
```

Ya que la clase FileInputStream también dispone de métodos para leer el archivo. Sólo que son unos pocos métodos que nos permiten leer un entero por vez o un arreglo de bytes. DataInputStream tiene métodos para leer los datos de muchas formas distintas, y en general resulta más cómodo.

Capturando excepciones

Ahora sí, vamos a ver cómo nos las arreglamos con las excepciones para que no se nos pare el programa con un mensaje tan poco estético...
En lugar de lanzar las excepciones al intérprete, vamos a procesarlas nosotros mediante la cláusula catch:

```
// Archivo:  Ejemplo11.java
// Compilar con:        javac Ejemplo11.java
```

```
            // Ejecutar con:        java Ejemplo11 <nombre_archivo>
import java.io.*;

public class Ejemplo11 {
public static void main(String args[]) {
FileInputStream                 fptr;
DataInputStream     f;
String                          linea = null;

try {
fptr = new FileInputStream(args[0]);
f = new DataInputStream(fptr);
do {
linea = f.readLine();
if (linea!=null) System.out.println(linea);
} while (linea != null);
fptr.close();
}
catch (FileNotFoundException e) {
System.out.println("Hey, ese archivo no existe!\n");
}
catch (IOException e) {
System.out.println("Error de E/S!\n");
}
}
}
```

También hicimos un cambio para elegir el archivo a imprimir desde la línea de comandos, en lugar de entrarlo fijo, utilizando para eso el argumento del método main(arg[]), que consiste en una lista de *Strings* con los parámetros que se pasan en la línea a continuación de java nombre_programa. Por ejemplo, si llamamos a este programa con:

```
java Ejemplo11 archi.txt otro.xxx
```

arg[0] contendrá "archi.txt", arg[1] contendrá "otro.xxx", y así sucesivamente.
Por supuesto, si llamamos a Ejemplo11 sin parámetros se lanzará otra excepción al intentar accederlo:

```
C:\java\curso>java Ejemplo11
java.lang.ArrayIndexOutOfBoundsException: 0
      at Ejemplo11.main(Ejemplo11.java:10)
```

Pero también podríamos capturarla!
Veamos un poquito cómo es esto de capturar excepciones.
La cláusula **try** engloba una parte del programa donde se pueden lanzar excepciones. Si una excepción se produce, Java busca una instrucción **catch (nombre_de_la_excepción variable)**, y, si la encuentra, ejecuta lo que ésta engloba. Si no encuentra un catch para esa excepción, para el programa y muestra el error que se produjo.
Por ejemplo, para evitar este último error bastaría con agregar:

```
catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Debe ingresar un nombre de archivo!");
System.out.println("Ej.: java Ejemplo11 pepe.txt");
}
```

Hay que notar que cuando se lanza una excepción el programa igual se detiene, porque el código que sigue al lanzamiento de la excepción no se ejecuta. Veremos luego cómo se comporta esto en un objeto que fue creado por otro, y cómo usar la instrucción finally para poner una parte de código que se ejecute pase lo que pase.

Los applets y los archivos

Veamos cómo se comporta esta aplicación si la modificamos para usarla como applet.

```
/*
// ----- Archivo:          Ejemplo12.java
*/

import java.io.*;
import java.awt.*;
import java.applet.*;

public class Ejemplo12 extends Applet {

public void init() {
                          new Ventana12();
}
}

/*
// -------- Esta clase es la que en realidad hace el trabajo
*/

class Ventana12 extends Frame {

TextArea    contenido;
Button                    cerrar;

Ventana12() {
super("Ejemplo de E/S");
contenido = new TextArea();
cerrar = new Button("Cerrar");
CargarArchivo();
add("North",contenido);
add("South",cerrar);
pack();
show();
}

public boolean handleEvent(Event e) {
if ((e.id==Event.WINDOW_DESTROY)||(e.target==cerrar))
dispose();
return super.handleEvent(e);
}

void CargarArchivo() {
FileInputStream                fptr;
DataInputStream        f;
String                         linea = null;
try {
```

```
fptr = new FileInputStream("Ejemplo12.java");
f = new DataInputStream(fptr);
do {
linea = f.readLine();
if (linea!=null)
contenido.appendText(linea+"\n");
} while (linea != null);
fptr.close();
}
catch (FileNotFoundException e) {
contenido.appendText("Hey, ese archivo no existe!\n");
}
catch (IOException e) {
contenido.appendText("Error de E/S!\n");
}
}
}
```

Lo cargamos desde la página Ejemplo12.html:

```
<HTML>
<HEAD>
<TITLE>Ejemplo 12 - Ejemplo con archivo</TITLE>
</HEAD>
<BODY>
<applet code="Ejemplo12.class" width=170 height=150>
</applet>
</BODY>
</HTML>
```

Mientras corramos esto en la misma máquina, no hay problema (anda muy bien!). Pero qué pasa si intentamos cargarlo desde la red? Para los que no tengan server html puse una copia en:
http://www.amarillas.com/rock/java/Ejemplo12.htm
El archivo no aparece! En su lugar se produce una excepción; en la línea de estado del Microsoft Internet Explorer, por ejemplo, se lee:

exception: com.ms.applet.AppletSecurityException: security.file.read: Ejemplo12.java

Esto es debido a una restricción de seguridad de Java: NO SE PUEDEN CARGAR ARCHIVOS QUE ESTEN EN UNA MAQUINA DISTINTA A AQUELLA DESDE LA CUAL SE CARGO EL APPLET. El applet se corre en el cliente, e intenta acceder a un archivo local. Eso es lo que provoca la excepción (que, por supuesto, puede detectarse con un `catch` y tratarse...)
Por cuestiones de seguridad, los applets son más limitados que las aplicaciones Java locales. Las políticas de seguridad las manejan los browsers (no Java), y generalmente los límites que se imponen a los applets son:

- Un applet no puede cargar bibliotecas (libraries) ni definir métodos nativos
- No puede leer o escribir normalmente archivos en el cliente que lo carga desde otro server
- No puede establecer conexiones de red, salvo al servidor del que proviene
- No puede arrancar programas en la máquina donde se está ejecutando
- No puede leer ciertas propiedades del sistema
- En las ventanas de los applets se indica que se trata de un applet

Sin embargo, pueden:

- Reproducir sonidos
- Pueden establecer conexiones con el servidor del que provienen

- Pueden llamar fácilmente páginas HTML desde el browser
- Pueden invocar métodos públicos de otros applets de la misma página
- Si se cargan desde la propia máquina (localmente) no tienen ninguna de las restricciones anteriores
- Pueden seguir corriendo aunque se cambie de página en el browser

En realidad, la especificación de Java permite que los applets lean archivos en otras máquinas dando la URL completa; sin embargo, los browsers no lo permiten. Veremos más adelante cómo intercambiar datos entre máquinas para poder ver un archivo del server, por ejemplo.

Nuestro modesto "Editor"

Para terminar este capítulo, el siguiente applet nos permite cargar, editar y grabar archivos ascii a elección. Podemos usar inclusive las acciones "cut & paste" del windows manager (Ctrl-C y Ctrl-V en Windows)! Cargarlo con "appletviewer Ejemplo13" luego de haberlo compilado (o usar una página html desde un browser):

```
/*
// ----- Archivo:          Ejemplo13.java
*/

import java.io.*;
import java.awt.*;
import java.applet.*;

public class Ejemplo13 extends Applet {

  public void init() {
              new Ventana13();
  }
}

/*
// -------- Esta clase es la que en realidad hace el trabajo
*/

class Ventana13 extends Frame {

 TextArea  contenido;
 Botones13 pieVentana;

 Ventana13() {
   super("Ejemplo de E/S");
   contenido = new TextArea();
   pieVentana = new Botones13();
   add("North",contenido);
   add("South",pieVentana);
   pack();
   show();
 }

 public boolean handleEvent(Event e) {
   if ((e.id==Event.WINDOW_DESTROY)||(e.id==2003))
     dispose();
   if (e.id==2001) CargarArchivo(pieVentana.toString());
   if (e.id==2002) GrabarArchivo(pieVentana.toString());
```

```
    return super.handleEvent(e);
  }

  void CargarArchivo(String nombre) {
    FileInputStream       fptr;
    DataInputStream       f;
    String                linea = null;
    contenido.setText("");
    try {
      fptr = new FileInputStream(nombre);
      f = new DataInputStream(fptr);
      do {
        linea = f.readLine();
        if (linea!=null) contenido.appendText(linea+"\n");
      } while (linea != null);
      fptr.close();
    }
    catch (FileNotFoundException e) {
                new Error13("El archivo no existe!");
    }
    catch (IOException e) {
                new Error13("Error leyendo archivo!");
    }
  }

  void GrabarArchivo(String nombre) {
    FileOutputStream      fptr;
    DataOutputStream      f;
    try {
      fptr = new FileOutputStream(nombre);
      f = new DataOutputStream(fptr);
      f.writeBytes(contenido.getText());
      fptr.close();
    }
    catch (IOException e) {
                new Error13("Error grabando archivo!");
    }
  }
}

/*
// -------- Esta es para los botones y el nombre del archivo
*/

class Botones13 extends Panel {

  TextField  fname;
  Button     cargar;
  Button     grabar;
  Button     cerrar;

  Botones13() {
    setLayout(new GridLayout(1,4));
    fname = new TextField();
```

```
    cargar = new Button("Cargar");
    grabar = new Button("Grabar");
    cerrar = new Button("Cerrar");
    add(new Label("Archivo:"));
    add(fname);
    add(cargar);
    add(grabar);
    add(cerrar);
   }

   public boolean handleEvent(Event e) {
    if ((e.id==Event.ACTION_EVENT)&&(e.target==cargar))
                e.id=2001;
    if ((e.id==Event.ACTION_EVENT)&&(e.target==grabar))
                e.id=2002;
    if ((e.id==Event.ACTION_EVENT)&&(e.target==cerrar))
                e.id=2003;
    return super.handleEvent(e);
   }

   public String toString() {
    return fname.getText();
   }
}

/*
// ------- Para mostrar los errores...
*/

class Error13 extends Frame {

   Error13(String error) {
                add("Center",new Label(error));
                add("South", new Button("Ok"));
                pack();
                show();
   }

   public boolean handleEvent(Event e) {
    dispose();
    return super.handleEvent(e);
   }
}
```

Volviendo al AWT

Para aprender un poquito más sobre la biblioteca gráfica (AWT), vamos a modificar nuestro último programa para usar menús.
Vamos a volver a poner todo el código (que ampliamos para usar como applet o aplicación local) marcando las diferencias más notables:

```
/*
// ----- Archivo:          Ejemplo14.java
*/

import java.io.*;
import java.awt.*;
import java.applet.*;

public class Ejemplo14 extends Applet {

  public void init() {
              new Ventana14(true);                          // con "true" avisamos que es applet
  }

  public static void main(String args[]) {      // para usarlo como aplicación
    Ventana14 v14 = new Ventana14(false);       // con "false" avisamos que no es applet
  }
}

/*
// -------- Esta clase es la que en realidad hace el trabajo
*/

class Ventana14 extends Frame {

  TextArea  contenido;
  boolean   enApplet;               // para indicar si lo llamamos como applet
  String                nombreArchivo;      // para guardar el nombre del archivo abierto
  MenuItem              mArchivoAbrir;      // ACA ESTAN LOS ITEMS DE LOS MENUS
  MenuItem              mArchivoGrabar;     //           .
  MenuItem              mArchivoSalir;                  //          .
  MenuItem              mEditCortar;                    //          .
  MenuItem              mEditCopiar;                    //          .
  MenuItem              mEditPegar;                     //          .
  MenuItem              mEditTodo;                      //          v
  String                clipboard;          // buffer para cortar y pegar
  boolean   editado = false;        // acá indicamos si modificamos el archivo

  Ventana14(boolean enApp) {

    super("Ejemplo de E/S");
    enApplet = enApp;                                           // recordamos si es applet o no

    Menu menuArchivo = new Menu("&Archivo");        // CREAMOS LOS MENUS!!!
    mArchivoAbrir = new MenuItem("&Abrir...");
    mArchivoGrabar = new MenuItem("&Grabar...");
```

```
    mArchivoSalir = new MenuItem("&Salir");
    menuArchivo.add(mArchivoAbrir);
    menuArchivo.add(mArchivoGrabar);
    menuArchivo.add(new MenuItem("-"));
    menuArchivo.add(mArchivoSalir);

    Menu menuEdit = new Menu("&Edit");
    mEditCortar = new MenuItem("Cor&tar");
    mEditCopiar = new MenuItem("&Copiar");
    mEditPegar = new MenuItem("&Pegar");
    mEditTodo = new MenuItem("&Seleccionar todo");
    menuEdit.add(mEditCortar);
    menuEdit.add(mEditCopiar);
    menuEdit.add(mEditPegar);
    menuEdit.add(new MenuItem("-"));
    menuEdit.add(mEditTodo);

    MenuBar barraMenu = new MenuBar();
    barraMenu.add(menuArchivo);
    barraMenu.add(menuEdit);
    setMenuBar(barraMenu);

    contenido = new TextArea();                    // solo pongo una ventana de texto
    add("Center",contenido);
    pack();
    show();

    clipboard = new String("");                    // clipboard vacío,
    mEditPegar.disable();                          // nada para pegar,
    mArchivoGrabar.disable();                      // nada para grabar
  }

  public boolean handleEvent(Event e) {
    if ((e.id==Event.WINDOW_DESTROY)||(e.target==mArchivoSalir)) {
            if (editado) System.out.println("Pedir confirmación!\n");          // debería confirmar
                                                                                // si se quiere
ir sin grabar!
            if (enApplet) dispose();
            else System.exit(0);
    }
    if (e.target==mArchivoAbrir) CargarArchivo();          // acá proceso selecciones
    if (e.target==mArchivoGrabar) GrabarArchivo();         // de menú
    if (e.target==mEditCortar) {
            clipboard = contenido.getSelectedText();
            mEditPegar.enable();
            contenido.replaceText("",contenido.getSelectionStart(),contenido.getSelectionEnd());
            editado=true;
    }
    if (e.target==mEditCopiar) {
clipboard = contenido.getSelectedText();
mEditPegar.enable();
    }
    if (e.target==mEditPegar) {
            contenido.replaceText("",contenido.getSelectionStart(),contenido.getSelectionEnd());
```

```
            contenido.insertText(clipboard,contenido.getSelectionStart());
            editado=true;
   }
   if (e.target==mEditTodo) contenido.selectAll();
   if ((e.id==Event.KEY_PRESS)&&(e.target==contenido)) editado=true;
   mArchivoGrabar.enable(editado);
   return super.handleEvent(e);
  }

  void CargarArchivo() {
   FileInputStream       fptr;
   DataInputStream       f;
   String                linea = null;
   if (editado) System.out.println("Pedir confirmación!\n");
   FileDialog fd = new FileDialog(this,"Abrir...",FileDialog.LOAD);          // elijo archivo
   fd.show();                                                                // usando el diálogo estándar del
sistema!
   nombreArchivo = fd.getFile();
   try {
    fptr = new FileInputStream(nombreArchivo);
    f = new DataInputStream(fptr);
    contenido.setText("");                    // vacío la ventana antes de cargar nuevo archivo
    do {
     linea = f.readLine();
     if (linea!=null) contenido.appendText(linea+"\n");
    } while (linea != null);
    fptr.close();
    editado=false;                            // archivo nuevo -> no editado
   }
   catch (FileNotFoundException e) {
            new Error14("El archivo no existe!");
   }
   catch (IOException e) {
            new Error14("Error leyendo archivo!");
   }
   catch (NullPointerException e) {
            ;
   }
  }

  void GrabarArchivo() {
   FileOutputStream      fptr;
   DataOutputStream      f;
   FileDialog fd = new FileDialog(this,"Grabar...",FileDialog.SAVE);         // grabo archivo
   fd.setFile(nombreArchivo);                                 // usando el diálogo estándar del sistema!
   fd.show();
   nombreArchivo = fd.getFile();
   try {
    fptr = new FileOutputStream(nombreArchivo);
    f = new DataOutputStream(fptr);
    f.writeBytes(contenido.getText());
    fptr.close();
    editado=false;                                            // recién grabado -> no editado
   }
```

```
   catch (IOException e) {
                new Error14("Error grabando archivo!");
   }
   catch (NullPointerException e) {
                ;
   }
  }
}

/*
// ------- Para mostrar los errores...
*/

class Error14 extends Frame {

  Error14(String error) {
                add("Center",new Label(error));
                add("South", new Button("Ok"));
                pack();
                show();
  }

  public boolean handleEvent(Event e) {
   dispose();
   return super.handleEvent(e);
  }
}
```

Menú a la Java

Bueno, lo primero que vamos a ver son los menús.
La barra de menú está compuesta por menúes, que a su vez están compuestos de ítems (que pueden también ser menúes). Por ejemplo la barra de menú la declaramos con:

```
MenuBar barraMenu = new MenuBar();
```

y le agregamos los menúes Archivo y Edit (que habremos creado previamente) con:

```
barraMenu.add(menuArchivo);
barraMenu.add(menuEdit);
```

Finalmente la declaramos como EL menú de la ventana (Frame):

```
setMenuBar(barraMenu);
```

Cada uno de los menús los declaramos previamente:

```
Menu menuArchivo = new Menu("&Archivo");
...
Menu menuEdit = new Menu("&Edit");
```

Noten que el "&" no se visualiza, sino que la letra que le sigue aparece subrayada: Archivo, Edit. Esto permite que se pueda seleccionar el menú tanto con el mouse como con la tecla alt- o meta-, seguida de la tecla subrayada.

A su vez, el método **add** está presente también en la clase **Menú** y nos permite agregar los ítems:

```
mArchivoAbrir = new MenuItem("&Abrir...");
mArchivoGrabar = new MenuItem("&Grabar...");
mArchivoSalir = new MenuItem("&Salir");
menuArchivo.add(mArchivoAbrir);
menuArchivo.add(mArchivoGrabar);
menuArchivo.add(new MenuItem("-"));
menuArchivo.add(mArchivoSalir);
```

A estos ítems los hemos declarado como globales en la clase para usarlos luego en los eventos. Noten además que

```
menuArchivo.add(new MenuItem("-"));
```

no agrega un ítem al menú sino una línea de separación, y no necesitamos crearlo como objeto permanente.
Si miramos la arquitectura de las clases, tanto MenuBar como MenuItem descienden de MenuComponent. A su vez, Menu desciende de MenuItem, por lo que implementa los mismos métodos y vamos a lo que decíamos antes: un menú puede ser un ítem de otro menú, y así sucesivamente tantos subniveles de menús como queramos.
Finalmente, en nuestro manejador de eventos simplemente necesitamos verificar si se eligió un ítem probando si el evento ocurrió sobre el ítem determinado:

```
if ((e.id==Event.WINDOW_DESTROY)||(e.target==mArchivoSalir)) {
            if (editado) System.out.println("Pedir confirmación!\n");
            if (enApplet) dispose();
            else System.exit(0);
   }
if (e.target==mArchivoAbrir) CargarArchivo();
.................
if (e.target==mEditTodo) contenido.selectAll();
```

En resumen lo que hago es:

- Si eligió Archivo/Salir (o alt-F4 o lo que sea) salgo del programa
- Si eligió Archivo/Abrir, llamo al método CargarArchivo
- Si eligió Archivo/Grabar, llamo al método GrabarArchivo
- Si eligió Edit/Cortar copio el texto seleccionado a mi *clipboard* y borro la selección
- Si eligió Edit/Copiar sólo copio el texto seleccionado a mi *clipboard*
- Si eligió Edit/Pegar borro el texto seleccionado e inserto el de mi *clipboard*
- Si eligió Edit/Seleccionar_todo marco todo el texto

En todos los casos, si se modifica el texto del contenido lo indico poniendo editado en true; lo mismo si presiono una tecla sobre el área de edición:

```
if ((e.id==Event.KEY_PRESS)&&(e.target==contenido)) editado=true;
```

Un par de aclaraciones:

- getSelectionStart() y getSelectionEnd() marcan los límites del texto seleccionado (si no lo hay, son iguales).
- getSelectedText() devuelve el texto seleccionado en el TextArea.
- replaceText() reemplaza una parte (o todo) del TextArea por un String.
- insertText() inserta un String en un lugar determinado del TextArea.
- selectAll() selecciona todo el texto del TextArea.

- MenuItem.enable() habilita un ítem de menú. Lo utilizo para habilitar Edit/Pegar sólo luego de cortar o copiar algo a mi *clipboard*.
- En el caso del ítem Archivo/Grabar, lo habilito o no dependiendo de la variable editado, utilizando la otra forma de *enable*: MenuItem.enable(boolean).

Diálogos

En Java disponemos de la clase Dialog para crear diálogos, es decir, ventanitas temporarias para entradas de usuario, que dependen de otra (de hecho la clase Dialog es heredera de la clase Window).
Si bien podemos crear diálogos a medida usando la clase Frame, se supone que usar diálogos debe ser más fácil. La realidad es que por ahora no se puede usar mucho más que los diálogos estándar (y el único que vale la pena es FileDialog), ya que las implementaciones actuales de Java tienen un problema: en algunas plataformas el programa que abre el diálogo sigue, en lugar de esperar que se cierre el diálogo y devuelva la respuesta.
Por eso hemos puesto solamente una indicación adonde debería haber un diálogo de confirmación:

```
if (editado) System.out.println("Pedir confirmación!\n");
```

En ese lugar deberíamos llamar por ejemplo a un diálogo que nos permita decidir por sí o por no:

```
if (editado) {
sino = new ConfirmarDlg(this,"Archivo modificado!");
if (sino.getResponse()==true) ....;
else ....;
}
```

o algo así. Esto mismo lo podemos hacer de otras maneras, por ejemplo usando threads y comunicaciones entre procesos, pero se complica mucho para esta altura del curso. Esperemos un poco más adelante, aunque Sun me prometió que en la versión 1.1 ya va a estar corregido (sale para fines del '96).
Por lo pronto, veamos un caso simple con la clase FileDialog:

```
FileDialog fd = new FileDialog(this,"Abrir...",FileDialog.LOAD);
fd.show();
nombreArchivo = fd.getFile();
```

Primero declaramos una variable de tipo FileDialog, y creamos la instancia con new. Como parámetros se pasa el padre (this, o sea "esta ventana"), el título de la ventanita de diálogo, y una constante LOAD o SAVE (son *static*, por lo que se denominan directamente con el nombre de la clase y no necesariamente de una instancia) que indica si el diálogo es para cargar o grabar un archivo (Obviamente la tarea en sí de cargar o grabar el archivo la tenenmos que hacer nosotros, el diálogo sólo espera que elijamos un nombre).
El método show() muestra el diálogo y **espera** que seleccionemos y presionemos Ok o Cancel. Aquí es donde fallan los demás diálogos ya que es programa sigue sin esperar.
Finalmente, el diálogo se cierra pero no se elimina el objeto (posiblemente está implementado usando el método hide(), que lo oculta de la vista pero no se pierde hasta no salir del método que lo creó, donde actuaría el *recogedor de basura* de la memoria). Esto hace que aunque no lo veamos podamos llamar al método getFile() sobre este objeto, que nos devuelve el nombre del archivo seleccionado (o null si se presionó Cancel).

DibuJava

Además de los componentes estándar (botones, listas, etc.), hay un componente para dibujo "libre" que nos permite implementar cualquier otro tipo de control: la clase **Canvas**. Típicamente se usa para dibujar, y corresponde a una zona rectangular dentro de una ventana.
La clase en sí no hace prácticamente nada; el programador debe definir una subclase de **Canvas** a la que el AWT le envía todos los eventos de mouse y teclado. Redefiniendo los métodos **gotFocus, lostFocus, keyDown, keyUp, mouseEnter, mouseExit, mouseMove, mouseDrag, mouseDown** y **mouseUp**, el programador puede hacer lo que se le ocurra dentro de ese rectángulo. Vamos a hacer uso de un **Canvas** para generar un applet donde habrá una zona rectangular dentro de la que, haciendo click con el mouse y moviéndolo sin soltar el botón, dibujaremos un rectángulo dinámicamente.
Esto nos permitirá ver cómo usar un **Canvas** para dibujar, capturar eventos, etc. El borde tiembla un poco al redibujar, pero ya veremos cómo evitar eso.

Canvas en acción

Primero vamos a poner, como ya se está haciendo costumbre, el código del applet (Recordar que debe cargarse desde una página html para verlo! Aquí no creamos ninguna ventana y no podremos verlo como aplicación standalone) y luego intentaremos explicar cómo funciona.

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

public class Ejemplo15 extends Applet {
   public void init() {
                Label label = new Label("Pique y arrastre con el mouse!");
                miCanvas zonaDib = new miCanvas();
                zonaDib.resize(new Dimension (200,200));
                add("North", label);
                add("Center", zonaDib);
                resize(300,250);
   }
}

class miCanvas extends Canvas {
   Rectangle rectActual;

   public boolean mouseDown(Event e, int x, int y) {
                rectActual = new Rectangle(x, y, 0, 0);
                repaint();
                return false;
   }

   public boolean mouseDrag(Event e, int x, int y) {
                rectActual.resize(x-rectActual.x, y-rectActual.y);
                repaint();
                return false;
   }

   public boolean mouseUp(Event e, int x, int y) {
                rectActual.resize(x-rectActual.x, y-rectActual.y);
                repaint();
                return false;
```

```
    }

    public void paint(Graphics g) {
                Dimension d = size();
                g.setColor(Color.red);
                g.drawRect(0, 0, d.width-1, d.height-1);
                g.setColor(Color.blue);
                if (rectActual != null) {
                        Rectangle box = cortarRect(rectActual, d);
                        g.drawRect(box.x, box.y, box.width-1, box.height-1);
                }
    }

    Rectangle cortarRect(Rectangle miRect, Dimension areaDib) {
        int x = miRect.x;
        int y = miRect.y;
        int ancho = miRect.width;
        int alto = miRect.height;

        if (ancho < 0) {
            ancho = -ancho;
            x = x - ancho + 1;
            if (x < 0) {
                ancho += x;
                x = 0;
            }
        }
        if (alto < 0) {
            alto = -alto;
            y = y - alto + 1;
            if (y < 0) {
                alto += y;
                y = 0;
            }
        }

        if ((x + ancho) > areaDib.width) {
            ancho = areaDib.width - x;
        }
        if ((y + alto) > areaDib.height) {
            alto = areaDib.height - y;
        }

        return new Rectangle(x, y, ancho, alto);
    }
}
```

El applet-container

En primer lugar hemos tenido en cuenta que un **Applet** es un **Panel**, y por lo tanto también un **Container**, así que en lugar de crear una ventana aparte simplemente le agregamos dos componentes: un **Label** y un **Canvas**.

```
        zonaDib.resize(new Dimension (200,200));
```

```
        add("North", label);
        add("Center", zonaDib);
        resize(300,250);
```

El método rezise, sobre la clase miCanvas, nos permite redimensionar el mismo al tamaño deseado. Igualmente, usamos resize sobre el applet para darle un tamaño adecuado. Si se modifica el tamaño de la ventana en el appletviewer se observará un comportamiento algo extraño en cuanto al posicionamiento relativo del rectángulo y el cartel, pero para simplificar esto bastará.

Nuestro Canvas a medida

Como no vamos a tomar ninguna acción especial al crear el canvas, no hemos definido el constructor (se utiliza el constructor por defecto de la clase Canvas).
Simplemente hemos redefinido algunos métodos para actuar al presionar, arrastrar y soltar el mouse, para redibujar el área de dibujo (canvas) y para recortar el rectángulo dibujado si nos vamos con el mouse fuera del espacio que ocupa el canvas.
La variable global rectActual, de la clase Rectangle, contendrá las coordenadas del rectángulo que estamos dibujando. El método Paint se llama automáticamente cada vez que es necesario redibujar el componente, o si llamamos explícitamente al método repaint():

```
public void paint(Graphics g) {
        Dimension d = size();
        g.setColor(Color.red);
        g.drawRect(0, 0, d.width-1, d.height-1);
        g.setColor(Color.blue);
        if (rectActual != null) {
                Rectangle box = cortarRect(rectActual, d);
                g.drawRect(box.x, box.y, box.width-1, box.height-1);
        }
  }
```

En primer lugar le asignamos a una variable d el tamaño del canvas usando el método size(), luego elegimos un color (rojo) para dibujar un borde y dibujamos un rectángulo del tamaño del componente:

```
        Dimension d = size();
        g.setColor(Color.red);
        g.drawRect(0, 0, d.width-1, d.height-1);
```

Dos atributos de la clase Dimension, width y height, se han cargado con el tamaño del canvas y son los que usamos para dar el tamaño del rectángulo.
Luego, si se está dibujando un rectángulo (rectActual != null) simplemente lo recortamos (en caso de que hayamos arrastrado el mouse fuera del canvas) y lo dibujamos.
El método que lo recorta a los límites del canvas, cortarRect, asigna a cuatro variables las coordenadas del rectángulo (que se le pasaron como parámetro miRect al llamarlo):

```
    int x = miRect.x;
    int y = miRect.y;
    int ancho = miRect.width;
    int alto = miRect.height;
```

Si el ancho (o el alto) es negativo, simplemente lo cambia de signo y toma como coordenada x (y) de origen el otro vértice del rectángulo, que corresponderá al x que se pasó menos el ancho y más uno (recordar que el origen de coordenadas empieza en cero y no en uno). Si este vértice está fuera del canvas (x<0), lo pone en cero y le resta al ancho la parte recortada (notar que ancho+=x, como x es negativo, es en realidad una resta).

```
if (ancho < 0) {
        ancho = -ancho;
        x = x - ancho + 1;
        if (x < 0) {
           ancho += x;
           x = 0;
        }
     }
```

Si nos vamos del área de dibujo por la derecha (o por abajo), simplemente le recortamos al ancho (alto) el exceso de modo que llegue hasta el borde del área de dibujo (que también hemos pasado al método como parámetro):

```
if ((x + ancho) > areaDib.width) {
        ancho = areaDib.width - x;
}
```

Sólo nos quedan por ver los métodos que responden al mouse.
Cuando presionamos el mouse dentro del canvas, comenzamos la creación de un nuevo rectángulo de ancho y alto cero que comienza en el punto en que hemos presionado el mouse, y redibujamos el canvas:

```
public boolean mouseDown(Event e, int x, int y) {
            rectActual = new Rectangle(x, y, 0, 0);
            repaint();
            return false;
  }
```

Al mover el mouse, redimensionamos el rectángulo con ancho x menos el origen de dibujo (y alto y menos el origen de dibujo), y repintamos:

```
  public boolean mouseDrag(Event e, int x, int y) {
            rectActual.resize(x-rectActual.x, y-rectActual.y);
            repaint();
            return false;
  }
```

Finalmente, al soltar el mouse, redimensionamos como antes y redibujamos:

```
  public boolean mouseUp(Event e, int x, int y) {
            rectActual.resize(x-rectActual.x, y-rectActual.y);
            repaint();
            return false;
  }
```

Como no se toma ninguna medida para guardar el rectángulo dibujado, al crear uno nuevo (reasignando rectActual a un nuevo rectángulo), el anterior se pierde.

DibuJava II

Vamos a retocar un poquito nuestro ejemplo15 para que no se borren los rectángulos cuando queremos dibujar uno nuevo. Aprenderemos algo sobre la clase **Vector**, perteneciente al paquete java.util.

Vectores en acción

Los vectores nos permiten hacer arreglos de cualquier tipo de objeto, y referirnos individualmente a cualquier elemento del vector, aunque para utilizarlos (debido a que para java el vector contiene objetos genéricos) tendremos que decirle qué clase de objeto es mediante un "cast". Vamos a ver cómo quedan nuestras clases Ejemplo16 (ex Ejemplo15) y miCanvas:

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

public class Ejemplo16 extends Applet {
   public void init() {
..... (esta parte no cambia).....
   }
}

class miCanvas extends Canvas {
   Vector v = new Vector();          // inicializamos con tamaño indeterminado
                                                  // Java se encarga de manejar la memoria necesaria!

   public boolean mouseDown(Event e, int x, int y) {
            v.addElement( new Rectangle(x, y, 0, 0) );     // nuevo elemento!
            repaint();
            return false;
   }

   public boolean mouseDrag(Event e, int x, int y) {
            Rectangle r = (Rectangle)v.lastElement();     // cast: v son rectángulos
            r.resize( x - r.x, y - r.y );                          // (creé r sólo por claridad)
            repaint();
            return false;
   }

   public boolean mouseUp(Event e, int x, int y) {
            Rectangle r = (Rectangle)v.lastElement();     // cast: v son rectángulos
            r.resize( x - r.x, y - r.y );                          // (creé r sólo por claridad)
            repaint();
            return false;
   }

   public void paint(Graphics g) {
            int i;                                                  // contador de rectángulos
            Dimension d = size();
            g.setColor(Color.red);
            g.drawRect(0, 0, d.width-1, d.height-1);
            g.setColor(Color.blue);
            if (v.size() > 0)
```

```
    for (i=0; i<v.size(); i++) {
                        Rectangle box = cortarRect( (Rectangle)v.elementAt( i ), d);
                        g.drawRect(box.x, box.y, box.width-1, box.height-1);
                }
   }
..... (el resto no cambia) .....
}
```

Les sugiero utilizar un HTML que reserve espacio suficiente para ver todo el applet, como:

```
<HTML>
<HEAD>
<TITLE>Ejemplo 16 - Ejemplo con canvas</TITLE>
</HEAD>
<BODY>
<applet code="Ejemplo16.class" width=300 height=250>
</applet>
</BODY>
</HTML>
```

Veamos los pasos ahora. En primer lugar creamos una variable (global a la clase) llamada v, de clase Vector, y sin asignarle un tamaño definido:

```
Vector v = new Vector();
```

Al crear un nuevo rectángulo agregamos un elemento (objeto) al vector mediante el método add:

```
v.addElement( new Rectangle(x, y, 0, 0) );
```

Para acceder a un atributo de un objeto del vector no basta utilizar directamente el vector, como:

```
v.lastElement().x
```

(lastElement() nos permite acceder al último elemento agregado al vector). Es necesario aclarar explícitamente que el elemento en cuestión es un rectángulo, ya que el vector puede contener objetos de cualquier tipo. Para eso usamos el *casting*:

```
(Rectangle)v.lastElement().x
```

En nuestro código original reemplazaríamos por:

```
(Rectangle)v.lastElement().resize( x - (Rectangle)v.lastElement().x, .....
```

Pero es más claro si usamos una variable local de clase Rectangle, le asignamos el mismo objeto que acabamos de agregar al vector, y lo usamos en su lugar:

```
Rectangle r = (Rectangle)v.lastElement();
r.resize( x - r.x, y - r.y );
```

Finalmente, en el método paint() no podemos asignar el elemento hasta no saber que existe (originalmente el vector estaba vacío!). Así que un if nos permite verificar que el tamaño del vector es mayor que cero (tiene elementos), y un for nos permite dibujarlos uno por uno.

Se puede acceder a todos los elementos, uno por uno, mediante el método elementAt(x), que nos da el x-ésimo elemento del vector. El método size() nos da la cantidad de elementos (el primero es el número 0, y así):

```
            if (v.size() > 0)
    for (i=0; i<v.size(); i++) {
                        Rectangle box = cortarRect( (Rectangle)v.elementAt( i ), d);
                        g.drawRect(box.x, box.y, box.width-1, box.height-1);
                }
```

Aquí no hemos creado variables intermedias ya que igualmente es claro (eso creo...).

Flicker molesto!

Bueno, el problema que nos queda es el molesto "flicker", o sea la manera en que titila el dibujo cuando movemos el mouse. Esto es porque cada vez que se llama a paint(), el fondo se borra y se redibuja todo el canvas.
Básicamente, la manera de evitarlo es reescribiendo el método update(), que es el que borra el fondo antes de llamar a paint() para que no lo borre; otro método (que es el que vamos a usar) es dibujar *no sobre la pantalla* sino sobre un "buffer" gráfico, y luego copiar ese buffer sobre la pantalla (lo que es mucho más eficiente que dibujar sobre la misma).
Para eso vamos a crear un par de objetos:

```
class miCanvas extends Canvas {
   Vector v = new Vector();
   Image    imgBuff;
   Graphics          grafBuff;
.............................
```

Image es una clase abstracta, madre de todas las clases que representan imágenes gráficas. Graphics es también abstracta y nos permite obtener un contexto en el cual dibujar.
Lo que vamos a hacer es modificar nuestro método paint() para que simplemente llame a update(), y redefinir el método update():

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

El método update() es el que hará todo el trabajo y básicamente es como nuestro viejo paint() con algunos agregados:

```
public void update(Graphics g) {
            int i;
            Dimension d = size();

if (grafBuff == null) {
                        imgBuff = createImage(d.width, d.height);
                        grafBuff = imgBuff.getGraphics();
}
grafBuff.setColor(getBackground());
grafBuff.fillRect(0, 0, d.width, d.height);
            grafBuff.setColor(Color.red);
            grafBuff.drawRect(0, 0, d.width-1, d.height-1);
            grafBuff.setColor(Color.blue);
```

```
            if (v.size() > 0) for (i=0; i<v.size(); i++) {
                        Rectangle box = cortarRect((Rectangle)v.elementAt(i), d);
                        grafBuff.drawRect(box.x, box.y, box.width-1, box.height-1);
            }
            g.drawImage(imgBuff, 0, 0, this);
}
```

En **negrita** hemos indicado los agregados.
Si no está creado todavía (grafBuff==null), creamos nuestro buffer de dibujo. Para crear dicho buffer gráfico (de clase Graphics), primero creamos una imagen que en este caso tiene las mismas dimensiones que el canvas (d.width x d.height), y luego asignamos a grafBuff el contexto de dicha imagen mediante el método getGraphics(). Imagínense que con createImage(...) crean una "pantalla virtual", y getGraphics() nos da una forma de acceder a esa pantalla como si fuera real.
Utilizando dicho contexto, elegimos como color el mismo color de fondo del applet (getBackground()) y dibujamos un rectángulo lleno (fillRect(...)), borrando así cualquier cosa que hubiera estado dibujada.
En *itálica* hemos indicado las modificaciones a nuestro método anterior. Simplemente, en lugar de usar el contexto de la pantalla (el parámetro g del método), dibujamos sobre nuestro contexto-pantalla virtual.
Finalmente, y para poder visualizar nuestro dibujo, usamos el método drawImage sobre el contexto de la pantalla real (g), que copia nuestro contexto imgBuff en las coordenadas (0,0) sobre la pantalla. Se hace también referencia al canvas (...this): el cuarto parámetro de drawImage es un objeto de clase ImageObserver, una interface que sirve para que el objeto dentro del cual se dibuja reciba mensajes asincrónicos que le indican cómo está siendo construida la imagen, y cuándo está lista.

Animate!

Si bien puede ser un poco más complejo de entender que un dibujo directo sobre la pantalla, notarán que la implementación es directa y no trae ningún problema. Esta misma aproximación puede utilizarse para crear animaciones.
En este ejemplo, para manejar la ejecución cuadro a cuadro de la animación, usamos *Threads*. No se preocupen por eso, lo veremos pronto. Únicamente tengan en cuenta que nuestro applet debe implementar la clase *runnable*, y el thread se encarga de ejecutar el método run() que simplemente llama a repaint() y espera 100 milisegundos entre cuadro y cuadro.
El trabajo de cálculo y dibujo lo hace update(). Se los dejo para que lo estudien; no es nada complicado y también usa doble buffering (como el ejemplo anterior).

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

public class Ejemplo18 extends Applet implements Runnable {

   Thread    animador;
   Image     imgBuff;
   Graphics grafBuff;
   double ang = 0.0;

   public void init() {
            resize(new Dimension (200,200));
   }

   public void start() {
            if (animador == null) animador = new Thread(this);
            animador.start();
   }
```

```
   public void run() {
            while (Thread.currentThread() == animador) {
                     repaint();
                     try {
                              Thread.sleep(100);
                     }
                     catch (InterruptedException e) {
                              break;
                     }
            }
   }

   public void update(Graphics g) {
            int i;
            int dx, dy;

            Dimension d = size();

if (grafBuff == null) {
                     imgBuff = createImage(d.width, d.height);
                     grafBuff = imgBuff.getGraphics();
}
grafBuff.setColor(getBackground());
grafBuff.fillRect(0, 0, d.width, d.height);
            grafBuff.setColor(Color.red);
            grafBuff.drawRect(0, 0, d.width-1, d.height-1);

            grafBuff.setColor(Color.blue);
            dx = (int)(50 * Math.abs(Math.cos(ang)));
            dy = (int)(50 * Math.abs(Math.sin(ang)));
            ang = ang + 0.1;
            if (ang>2*Math.PI) ang = 0.0;
            grafBuff.drawRect(100-dx, 100-dy, 2*dx, 2*dy);

            g.drawImage(imgBuff, 0, 0, this);
   }
}
```

Java en hebras

La clase anterior usamos, en el último ejemplo, un concepto al que vamos a dedicar ahora nuestra atención: los *threads.*
La traducción literal de *thread* es hilo o hebra, y se utiliza también para referirse al hilo de un discurso. El concepto de threads en los ambientes y sistemas operativos es un poco complejo de explicar pero sencillo de entender: independientemente del sistema elegido, puede pensarse que un thread es algo así como el lugar de ejecución de un programa.
En la mayoría de los programas que hemos visto, hemos usado un solo thread; es decir que un programa comienza y su ejecución sigue un camino único: como un monólogo.
Java es multithreading. Esto significa algo así como que tiene capacidad de diálogo, y más aún: puede ejecutar muchos threads en paralelo, como si tratáramos de una conversación múltiple y simultánea.
No confundir aquí multithreading con la capacidad de ejecutar varios programas a la vez. Esta es una posibilidad, pero también un mismo programa puede utilizar varios threads ("caminos de ejecución"?) simultáneamente.
Esto, por supuesto, depende fundamentalmente de la capacidad del sistema operativo para soportar multithreading, y por esto Java no puede ejecutarse (al menos en forma completa) en sistemas que no lo soporten.
El uso de threads nos permite, por ejemplo, ejecutar simultáneamente varios programas que interactúen entre ellos; o, también, que un programa, mientras por ejemplo actualiza la pantalla, simultáneamente realice una serie de cálculos sin tener que hacer esperar el usuario.
Una forma sencilla de verlo es imaginar que tenemos un grupo de microprocesadores que pueden ejecutar, cada uno, un solo thread; y nosotros asignamos programas (o partes de programas) a cada uno de ellos. Además, podemos imaginar que esos microprocesadores comparten una memoria común y recursos comunes, de lo que surgirá una serie de problemas importantes a tener en cuenta cuando se usan *threads.*

Los pasos básicos

Hay tres cosas a tener en cuenta para usar threads en un programa:

- La clase que queremos asignar a un thread debe implementar la interface *Runnable*.
- Debemos crear una variable (instancia) del tipo *Thread*, que nos permitirán acceder y manejar el thread. En los applets, en el método *start()* simplemente crearemos el thread (y, posiblemente, lo pondremos a ejecutar)
- Y por último tenemos que crear un método run() que es el que ejecuta el código del programa propiamente dicho.

La interface Runnable, simplemente definida como:

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

le asegura al compilador que nuestra clase (la que utilizará el thread para ejecutarse) dispone de método *run()*.
Vamos a ver un par de ejemplos, primero una aplicación standalone y luego un applet.

Reunión de amigos

El siguiente ejemplo (Ejemplo19.java) usa threads para activar simultáneamente tres objetos de la misma clase, que comparten los recursos del procesador peleándose para escribir a la pantalla.

```
class Ejemplo19 {

            public static void main(String argv[])
```

```
        throws InterruptedException {
                Thread Juan = new Thread (new Amigo("Juan"));
                Thread Luis = new Thread (new Amigo("Luis"));
                Thread Nora = new Thread (new Amigo("Nora"));
                Juan.start();
                Luis.start();
                Nora.start();
                Juan.join();
                Luis.join();
                Nora.join();
        }

}

class Amigo implements Runnable {

        String mensaje;

        public Amigo(String nombre) {
                mensaje = "Hola, soy "+nombre+" y este es mi mensaje ";
        }

        public void run() {
                for (int i=1; i<6; i++) {
                        String msg = mensaje+i;
                        System.out.println(msg);
                }
        }

}
```

Como siempre, compilarlo con *javac Ejemplo19.java* y ejecutarlo con *java Ejemplo19*.
En un sistema operativo *preemptivo*, la salida será más o menos así:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Luis y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 1
Hola, soy Nora y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 3
Hola, soy Juan y este es mi mensaje 3
.....etc.
```

Qué significa que un sistema operativo es preemptivo? Casos típicos son Unix o Windows 95: cada tarea utiliza una parte del tiempo del procesador, y luego lo libera para que puedan ejecutarse otras tareas (otros threads). Por eso se mezclan los mensajes de salida. Si el sistema operativo es *no preemptivo*, el procesador no se libera hasta que no termina con el thread actual, y por lo tanto la salida sería así:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Juan y este es mi mensaje 3
Hola, soy Juan y este es mi mensaje 4
Hola, soy Juan y este es mi mensaje 5
```

Hola, soy Luis y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 2
.....etc.

Si ustedes están utilizando un sistema operativo no preemptivo, deben explícitamente indicarle al procesador cúando puede ejecutar (*dar paso*) a otra tarea; para eso simplemente modifiquen el método *run()*:

```
public void run() {
        for (int i=1; i<6; i++) {
                String msg = mensaje+i;
                System.out.println(msg);
                Thread.yield();
        }
}
```

En este ejemplo, tanto en sistemas preemptivos como no preemptivos la salida será:

Hola, soy Juan y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 1
Hola, soy Nora y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Luis y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 2
Hola, soy Juan y este es mi mensaje 3
Hola, soy Luis y este es mi mensaje 3
.....etc.

Esto es porque en seguida de imprimir estamos liberando al procesador para que pase a otro thread (si hay alguno esperando). Noten la diferencia con el primer caso, sin usar *yield()*, para sistemas preemptivos: el procesador reparte su trabajo en forma (aparentemente) impredecible, por eso el orden de los mensajes no será el mismo en cualquier máquina o sistema operativo.
Ya lo vimos funcionar, pero sería bueno que lo entendamos! Por eso, vamos paso a paso.

Creando Threads

Thread es una clase básica en Java, que implementa la interface *Runnable* y dispone de unos cuantos métodos por defecto. Lo importante a tener en cuenta que, para usar Threads, debemos crearlas como instancias y ponerlas a "andar":

```
Thread Juan = new Thread (new Amigo("Juan"));
.............
Juan.start();
.............
Juan.join();
```

Un thread tiene cuatro estados posibles:
creado: ha sido creado mediante *new()*, pero no se ha puesto en marcha todavía.
activo: está en ejecución, ya sea porque arrancó con *start()* o fue "despertado" con *resume()*.
dormido: ha sido suspendida su ejecución momentáneamente mediante *wait()*, *sleep()* o *suspend()*.
muerto: se ha detenido definitivamente, ya sea porque se terminó el programa o mediante el llamado a *stop()*.
En este ejemplo hemos creado un thread asignándole simultáneamente un objeto que lo utiliza (`new Amigo("Juan")`), y seguidamente lo hemos activado, llamando al método `start()`. Este método se encarga de inicializar el thread y, finalmente, llamar al método `run()` que hemos implementado.

De este modo, todo ocurre como si los métodos run() de cada objeto se ejecutaran en paralelo, concurrentemente. La forma de manejar esto depende del sistema operativo.
El método join() que llamamos al final hace que el programa principal espere hasta que este thread esté "muerto" (finalizada su ejecución). Este método puede disparar la excepción InterruptedException, por lo que lo hemos tenido en cuenta en el encabezamiento de la clase.
En nuestro ejemplo, simplemente a cada instancia de Amigo(...) que creamos la hemos ligado a un thread y puesto a andar. Corren todas en paralelo hasta que mueren de muerte natural, y también el programa principal acaba.
Cuando usamos Thread.yield() (que en rigor debería ser *Thread.currentThread().yield()*, pero siendo algo de uso muy común los desarrolladores de Java lo han simplificado), simplemente el thread actual le permite al procesador dedicarse a otro (si es que hay alguno deseando utilizar sus servicios).
La clase Amigo() es muy simple y con lo que hemos visto hasta ahora no creo que tengamos que explicar nada más.

Y los applets...?

También podemos usar estos conceptos en los applets. Veamos un ejemplo para terminar la clase de hoy, muy similar al anterior, donde tres contadores cuentan (en un sistema preemptivo) en forma simultánea. Recuerden crear una página HTML con el tag

<applet code="Ejemplo20.class" width=300 height=100></applet>
para poder verlo en acción con el appletviewer o su browser favorito (que desde ya supongo que soporta Java! ;-)
El programa es extremandamente sencillo, y pueden verlo en acción si lo desean cargando via Internet la página:
http://www.amarillas.com/rock/java/Ejemplo20.htm

```
//              Ejemplo de applet que usa multithreading
import java.awt.*;
import java.applet.*;

public class Ejemplo20 extends Applet {

        TextField tfa,tfb,tfc;

        public void init() {
                setLayout(new GridLayout(3,2));
                tfa = new TextField("0");
                tfb = new TextField("0");
                tfc = new TextField("0");
                add(new Label("Contador A"));
                add(tfa);
                add(new Label("Contador B"));
                add(tfb);
                add(new Label("Contador B"));
                add(tfc);
        }

        public void start() {
                Thread A = new Thread (new Counter(tfa));
                Thread B = new Thread (new Counter(tfb));
                Thread C = new Thread (new Counter(tfc));
                A.start();
                B.start();
```

```
                C.start();
        }

}

class Counter implements Runnable {

        TextField texto;
        String s;

        public Counter(TextField txtf) {
                texto = txtf;
        }

        public void run() {
                for (int i=0; i<1000; i++) {
                        texto.setText(s.valueOf(i));
                }
        }

}
```

La liebre y la tortuga (y el guepardo)

Java dispone de un mecanismo de prioridades para los *threads*, de modo de poder asignar más tiempo de CPU a un thread que a otro. Típicamente se asigna una prioridad de 1 a 10 (10 es la mayor prioridad) mediante *setPriority*, como en el ejemplo que sigue:

```
public class Ejemplo21 {

   static Animal          tortuga;
   static Animal          liebre;
   static Animal          guepardo;

   public static void main(String argv[])
     throws InterruptedException {

                tortuga  = new Animal(2, "T");
                liebre   = new Animal(3, "L");
                guepardo = new Animal(4, "G");
                tortuga.start();
                liebre.start();
                guepardo.start();
                tortuga.join();
                liebre.join();
                guepardo.join();
   }
}

class Animal extends Thread {
                String nombre;

                public Animal(int prioridad, String nombre) {
                                this.nombre = nombre;
                                setPriority(prioridad);
                }

                public void run() {
                                for (int x = 0; x < 30; x++) {
                                                System.out.print( nombre );
                                                yield();
                                }
                                System.out.println("\nLlega "+nombre );
                }
}
```

La salida de este programa, ejecutado con *java Ejemplo21*, es por ejemplo:

```
C:\java\curso>java Ejemplo21
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
Llega G
LTLLLLLLLLLLLLLLLLLLLLLLLLLLL
Llega L
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
Llega T
```

Como se ve, a pesar de haber arrancado antes la tortuga, casi todo el tiempo de CPU lo usa primero el Guepardo, luego la Liebre (aunque algo queda para la pobre tortuga, como se ve en la T marcada), y finalmente para la Tortuga. No todas las corridas ni todos los sistemas dan igual salida, ya que ésta depende de la carga del procesador y de la implementación de Java particular.
Este programa simplemente crea tres animales (clase Animal), asigna un thread a cada uno y los ejecuta. Este ejemplo está hecho en base a uno del libro "Programación Java" de Macary y Nicolas.

Sincronicemos los relojes

Un problema básico del multithreading es cuando varios programas (o, para el caso, varios threads) acceden a los mismos datos: ¿cómo sabemos si uno de ellos no los modifica mientras los está usando otro?.
Veamos un ejemplo, donde suponemos que varios threads usan la variable valorImportante:

```
if  (valorImportante > 0 ) {
        ..... algo se procesa acá .....
        valorImportante = valorImportante - 1;
        ..... sigue.....................
}
```

¿Cómo nos aseguramos que valorImportante no cambió entre el if y la línea resaltada? Otros threads pueden haberlo modificado mientras tanto. Asimismo, puede suceder que dos threads estén ejecutando la misma porción de código, y se pierda uno de los decrementos. Imaginen algo así:

```
(antes)          valorImportante = 10
(thread 1)   lee valorImportante = 10
(thread 2)   lee valorImportante = 10
(thread 1)   10 -1 = 9
(thread 2)   10 -1 = 9
(thread 2)   asigna 9 a valorImportante
(thread 1)   asigna 9 a valorImportante
(después)    valorImportante = 9
```

Como vemos, a pesar de haber restado dos veces, hemos perdido una de las restas. Aunque usemos -= en vez de la resta es lo mismo, porque el código igualmente se resuelve en varios pasos (varias *operaciones atómicas*).
Para evitar esto, Java nos brinda la palabra clave Synchronized, que bloquea el acceso a una variable a todos los threads menos el que lo está usando.
Vamos a ver un caso específico; se trata de dos contadores que usan el mismo sumador para sumar de a uno una cantidad a. Supuestamente entre los dos deben llevar el sumador (a) hasta 20000.

```
// Archivo Ejemplo22.java, compilar con javac Ejemplo22.java, ejecutar con java Ejemplo22

public class Ejemplo22 {
        public static void main(String argv[]) {
                Sumador A = new Sumador();                      // un único sumador
                Contador C1 = new Contador(A);   // dos threads que lo usan...
                Contador C2 = new Contador(A);   // ...para sumar
                C1.start();
                C2.start();
                try {
                        C1.join();
                        C2.join();
                }
                catch (Exception e) {
```

```
                    System.out.println(e);
                }
        }
}


class Contador extends Thread {
        Sumador s;

        Contador (Sumador sumador) {
                s = sumador;                    // le asigno un sumador a usar
        }

        public void run() {
                s.sumar();              // ejecuto la suma
        }
}


class Sumador {
        int a = 0;
        public void sumar() {
                for (int i=0; i<10000; i++ ) {
                        if ( (i % 5000) == 0 ) {        // "%" da el resto de la división:
                                System.out.println(a);  // imprimo cada 5000
                        }
                        a += 1;
                }
                System.out.println(a);                  // imprimo el final
        }
}
```

Ejecutando esto nos da más o menos así (cada corrida es diferente, dependiendo de cómo se "chocan" los threads y la carga de la CPU):

```
C:\java\curso>java Ejemplo22
0
87
8926
10434
14159
17855
```

Esto se debe justamente a lo que explicábamos al principio: a veces los dos threads intentan ejecutar `a += 1` simultáneamente, con lo que algunos incrementos se pierden.
Podemos solucionar esto modificando el método `run()`:

```
        public void run() {
                synchronized (s) {
                        s.sumar();
                }
        }
```

Con esto, sólo a uno de los dos threads se les permite ejecutar s.sumar() por vez, y se evita el problema. Por supuesto, el otro thread queda esperando, por lo que más vale no utilizar esto con métodos muy largos ya que el programa se puede poner lento o aún bloquearse.
La salida ahora será:

```
C:\java\curso>java Ejemplo22
0                                       <
5000                                    < primer thread
10000                                   <
10000                                   (
15000                                   ( segundo thread
20000                                   (
```

Lo mismo logramos (y en forma más correcta) declarando como synchronized al método sumar():

```
        public synchronized void sumar() { .....
```

Esto es mejor porque la clase que llama a sumar() no necesita saber que tiene que sincronizar el objeto antes de llamar al método, y si otros objetos (en otros threads) lo llaman, no necesitamos preocuparnos.

Más sincronización

Otra manera de sincronizar el acceso de los threads a los métodos, es lograr que éstos se pongan de acuerdo entre sí, esperando uno hasta que otro realizó alguna tarea dada. Para esto se usan los métodos wait() y notify(). Cuando un thread llama a wait() en un método de un objeto dado, queda detenido hasta que otro thread llame a notify() en algún método del mismo objeto.
Por ejemplo, vamos a suponer cuatro empleados que se encuentran con su jefe y lo saludan, pero sólo luego de que éste los salude primero.

```
public class Ejemplo23 {

        public static void main(String argv[]) {
                Saludo hola = new Saludo();
                Personal pablo = new Personal(hola, "Pablo", false);
                Personal luis = new Personal(hola, "Luis", false);
                Personal andrea = new Personal(hola, "Andrea", false);
                Personal pedro = new Personal(hola, "Pedro", false);
                Personal jefe = new Personal(hola, "JEFE", true);
                pablo.start();
                luis.start();
                andrea.start();
                pedro.start();
                jefe.start();
                try {
                        pablo.join();
                        luis.join();
                        andrea.join();
                        pedro.join();
                        jefe.join();
                }
                catch (Exception e) {
                        System.out.println(e);
                }
        }
}
```

```
class Saludo {
	synchronized void esperarJefe(String empleado) {
	 try {
		wait();
		System.out.println(empleado+"> Buenos dias jefe!");
	 }
	 catch (InterruptedException e) {
		System.out.println(e.toString());

	 }
	}

	synchronized void saludoJefe() {
		System.out.println("JEFE> Buenos dias!");
		notifyAll();
	}
}


class Personal extends Thread {
	String nombre;
	Saludo saludo;
	boolean esJefe;

	Personal (Saludo s, String n, boolean j) {
		nombre = n;
		saludo = s;
		esJefe = j;
	}

	public void run() {
		System.out.println("("+nombre+" llega)");
		if (esJefe)
			saludo.saludoJefe();
		else
			saludo.esperarJefe(nombre);
	}
}
```

Usé notifyAll() en lugar de notify(), porque en el segundo caso sólo se notificaría al primer thread (el primer empleado en llegar) y no a los demás, que se quedarían en el wait().
Como se ve en la salida, a pesar de que los empleados están en condiciones de saludar, no lo hacen hasta que no llega el jefe:

```
C:\java\curso>java Ejemplo23
(Pablo llega)
(Luis llega)
(Andrea llega)
(Pedro llega)
(JEFE llega)
JEFE> Buenos dias!
Luis> Buenos dias jefe!
```

Pedro> Buenos dias jefe!
Andrea> Buenos dias jefe!
Pablo> Buenos dias jefe!

Aquí hice trampa: a veces, el jefe llega y saluda antes que alguno de los empleados, por lo que ese empleado se queda esperando indefinidamente. Prueben de modificar las clases para que el jefe no salude hasta que no estén todos los empleados presentes...

Capítulo XV - Solución al problema propuesto

En forma muy sencilla, modificando sólo la clase Personal, podemos solucionar el problema de que el jefe llegue antes que un empleado:

```
class Personal extends Thread {
        .............
        static int llegaron = 0;
        .............

        public void run() {
                System.out.println("("+nombre+" llega)");
                if (esJefe) {
                        while (llegaron < 4) {
                                System.out.println("(Esperando...)");
                        }
                        saludo.saludoJefe();
                }
                else {
                        synchronized(this) {
                                llegaron++;
                        }
                        saludo.esperarJefe(nombre);
                }
        }
}
```

Preparamos una variable static (de clase) para contar todos los empleados que pasaron por aquí; la incrementamos justo antes de ejecutar saludo.esperarJefe (sincronizando el thread en el incremento para que no pasen los problemas que vimos en el capítulo).
En el caso del jefe, simplemente espera que el contador llegue a 4. Podríamos modificar esto un poco, pasando la cantidad de empleados como parámetro para que sea más flexible.
Inclusive, podemos usar dos constructores distintos (uno para los empleados y otro para el jefe, y en este último caso pasamos la cantidad de empleados a esperar). Les dejo el ejemplo para que lo estudien.

```
public class Ejemplo23 {

        public static void main(String argv[]) {
                Saludo hola = new Saludo();
                Personal jefe = new Personal(hola, "JEFE", 3);
                Personal pablo = new Personal(hola, "Pablo");
                Personal luis = new Personal(hola, "Luis");
                Personal andrea = new Personal(hola, "Andrea");
                jefe.start();
                pablo.start();
                luis.start();
                andrea.start();
                try {
                        pablo.join();
                        luis.join();
```

```
                    andrea.join();
                    jefe.join();
                }
                catch (Exception e) {
                    System.out.println(e);
                }
        }
}


class Saludo {
        synchronized void esperarJefe(String empleado) {
         try {
                wait();
                System.out.println(empleado+"> Buenos dias jefe!");
         }
         catch (InterruptedException e) {
                System.out.println(e.toString());

         }
        }

        synchronized void saludoJefe() {
                System.out.println("JEFE> Buenos dias!");
                notifyAll();
        }
}


class Personal extends Thread {
        String nombre;
        Saludo saludo;
        boolean esJefe;
        static int llegaron = 0;
        int numEmp;

        Personal (Saludo s, String n) {
                esJefe = false;
                nombre = n;
                saludo = s;
        }

        Personal (Saludo s, String n, int x) {
                esJefe = true;
                nombre = n;
                saludo = s;
                numEmp = x;
        }

        public void run() {
                System.out.println("("+nombre+" llega)");
                if (esJefe) {
                    while (llegaron < numEmp) { System.out.println("(Esperando...)"); }
                    saludo.saludoJefe();
```

```
                        }
                        else {
                                synchronized(this) { llegaron++; }
                                saludo.esperarJefe(nombre);
                        }
                }
}
```

Multimedia!

Java permite cargar y visualizar archivos GIF o JPEG de imagen y AU de audio (solamente en mono, 8 bits, 8000Hz de muestreo).
Para el caso del sonido, un archivo de audio se carga mediante un objeto de la clase AudioClip, mediante el método getAudioClip(URL, archivo), se ejecuta con los métodos play() o loop() y se detiene con stop().
Noten esto! Si bien dijimos que un applet no puede acceder al disco de la máquina cliente, SI puede leer archivos del server desde donde se cargó. Por lo tanto, pasándole el URL de la máquina desde donde se cargó el applet, podemos leer cualquier tipo de archivo a través de la red.
La forma más segura de indicar dicho URL es mediante el método getDocumentBase(), que nos da el URL adecuado.
Por ejemplo, puedo cargar y reproducir audio con sólo dos líneas:

```
.....
AudioClip sonido = getAudioClip( getDocumentBase(), "sonido.au" );
sonido.play();
.....
```

Por otra parte, una foto puede cargarse mediante un objeto de clase Image mediante el método getImage(URL, archivo). Luego la mostramos en un objeto Graphics correspondiente al applet (o al área de dibujo) mediante drawImage(imagen, x, y, observador). Observador es un objeto que implementa la interface ImageObserver; los applets, por descender de Component (que implementa dicha interface) también la implementan. Típicamente, la imagen se visualiza en el método paint(...) del applet:

```
.............
algunMetodo(...) {
.........
Image imagen = getImage(getDocumentBase(), "imagen.gif");
.........
}
...........
        public void paint(Graphics g) {
                g.drawImage(imagen, xOffset, yOffset, this);     // "this" representa al applet
        }
.............
```

El problema con las imágenes es asegurarse que fue cargada antes de mostrarla. Para eso se utiliza un MediaTracker (también debería servir para los archivos de audio, pero en esta versión aún no está implementado).
Mediante addImage(imagen, grupo) se agrega una imagen a la lista del MediaTracker, y hay métodos para esperar que sea cargada (como waitForAll() o waitForID(grupo)), para verificar que se haya cargado correctamente (como checkAll(), checkID(grupo), isErrorAny()...), etcétera.
El siguiente applet utiliza estos conceptos para cargar una imagen y un archivo de audio y mostrarlos:

```
//          Ejemplo24.java
import java.awt.*;
import java.applet.*;

public class Ejemplo24 extends Applet {

        MediaTracker        supervisor;
        String              archImagen, archAudio;
        Image               imagen;
```

```
	AudioClip	audio;
	Label		titulo;
	Panel		cuadro;

	public void init() {
		supervisor = new MediaTracker(this);
		archImagen = "javacero.gif";
		archAudio = "tada.au";

		// carga imagen
		imagen = getImage(getDocumentBase(), archImagen);
		supervisor.addImage(imagen,0);
		try {
			supervisor.waitForID(0);		// espero que
se cargue
		}
		catch (InterruptedException e) {
			System.out.println("Error cargando imagen!");
		}
		showStatus("Imagen cargada");

		// carga sonido
		audio = getAudioClip(getDocumentBase(), archAudio);

		// arma layout
		setLayout(new BorderLayout());
		titulo = new Label(archImagen);
		setFont(new Font("helvetica", Font.BOLD, 18));
		add("South", titulo);
	}

	public void start() {
		repaint();
		audio.play();
	}

	public void paint(Graphics g) {
		if (supervisor.isErrorAny()) {
			g.setColor(Color.black);
			g.fillRect(0, 0, size().width, size().height);
			return;
		}
		g.drawImage(imagen, 0, 0, this);
	}
}
```

Para visualizarlo, como siempre, creamos un HTML:

```
<HTML>
<HEAD>
<TITLE>Ejemplo 24 - Ejemplo Multimedia</TITLE>
</HEAD>
<BODY>
```

```
<applet code="Ejemplo24.class" width=150 height=200>
</applet>
</BODY>
</HTML>
```

Parametrizando un applet

Vamos a aprovechar este ejemplo, modificándolo un poco para indicarle desde el HTML qué archivos debe cargar, mediante parámetros. Nuestro HTML modificado será:

```
<HTML>
<HEAD>
<TITLE>Ejemplo 24 - Multimedia</TITLE>
</HEAD>
<BODY>

<applet code="Ejemplo24.class" width=150 height=200>
<param name="imagen" value="javacero.gif">
<param name="sonido" value="tada.au">
</applet>
</BODY>
</HTML>
```

Para leer estos parámetros desde el applet, usamos el método getParameter(nombreParámetro), así que podemos modificar nuestro applet simplemente modificando un par de líneas:

```
archImagen = getParameter("imagen");
archAudio = getParameter("sonido");
```

Voilá! Pueden probar de cargar este applet en http://www.amarillas.com/rock/java/Ejemplo24.htm.
De esta manera podemos pasar cualquier valor como parámetro para un applet, haciéndolo más flexible.
Y esto es todo por hoy!
Con esto hemos visto una gran parte de lo que es Java. No hemos profundizado demasiado en cada punto, pero hemos hecho ejemplos que funcionan para ilustrar cada cosa.
Sin embargo, hemos dejado un punto importante y muy fuerte de Java, que es el de las comunicaciones entre aplicaciones y, especialmente, el uso de sockets y la programación de aplicaciones cliente-servidor.

Paseando por la Red

Es muy sencillo acceder a archivos en la red utilizando Java. El paquete java.net dispone de varias clases e interfases a tal efecto.
En primer lugar, la clase URL nos permite definir un recurso en la red de varias maneras, por ejemplo:

```
URL url1 = new URL ("http://www.rockar.com.ar/index.html");
URL url2 = new URL ("http", "www.rockar.com.ar", "sbits.htm");
```

Por otra parte, podemos establecer una conexión a un URL dado mediante openConnection:

```
URLConnection conexion = url.openConnection();
```

Una vez lograda la conexión, podemos leer y escribir datos utilizando *streams* (*corrientes* de datos), como en el caso de manejo de archivos comunes (ver capítulo X). Un DataInputStream nos permite leer datos que llegan a través de la red, y un DataOutputStream nos permite enviar datos al host.
Por ejemplo:

```
DataInputStream datos = new DataInputStream( corrienteEntrada  );
```

En nuestro caso, la corriente de entrada de datos proviene de la conexión al URL. El método getInputStream() del objeto URLConnection nos provee tal corriente:

```
DataInputStream datos = new DataInputStream(conex.getInputStream())
```

De este modo podemos escribir un pequeño programa para, por ejemplo, leer una página HTML desde una dirección arbitraria de internet. El programa, luego de compilarse mediante *javac Ejemplo25.java*, se ejecuta con *java Ejemplo25 <url>;* por ejemplo: java Ejemplo25 http://www.rockar.com.ar/index.html.

```
import java.io.*;
import java.net.*;

public class Ejemplo25 {

        public static void main(String argv[]) {
          String s;
          try {
                URL url = new URL (argv[0]);
                URLConnection conex = url.openConnection();
                System.out.println("Cargando "+argv[0]);
                DataInputStream datos = new DataInputStream(conex.getInputStream());
                do {
                        s = datos.readLine();
                        if (s != null) System.out.println(s);
                } while (s != null);
          }
          catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Sintaxis: java Ejemplo25 <url>");
          }
          catch (UnknownHostException e) {
```

```
                    System.out.println("El host no existe o no responde");
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
            }
}
```

Este programa muestra el HTML como texto en la pantalla, pero podríamos grabarlo a un archivo para guardarlo. Inclusive, podríamos procesarlo a medida que lo recibimos, identificar los tags <A HREF=url>, guardarlos en un vector, y seguidamente conectarnos y bajar los links que figuran en la página original hasta bajar un site completo.
Noten que esto no sólo sirve para establecer conexiones a páginas HTML. En realidad, un URL puede referirse también a otros protocolos, como gopher, ftp, etcétera; si bien según la implementación de Java puede haber problemas para conectarse a algunos tipos de URL.
Para ver los tipos de URL posibles les recomiendo leer la página:

http://www.ncsa.uiuc.edu/demoweb/url-primer.html

Los Sockets

Los *sockets* (zócalos, referido a los enchufes de conexión de cables) son mecanismos de comunicación entre programas a través de una red TCP/IP. De hecho, al establecer una conexión via Internet estamos utilizando sockets: los sockets realizan la interfase entre la aplicación y el protocolo TCP/IP.
Dichos mecanismos pueden tener lugar dentro de la misma máquina o a través de una red. Se usan en forma cliente-servidor: cuando un cliente y un servidor establecen una conexión, lo hacen a través de un socket. Java proporciona para esto las clases ServerSocket y Socket.
Los sockets tienen asociado un *port* (puerto). En general, las conexiones via internet pueden establecer un puerto particular (por ejemplo, en http://www.rockar.com.ar**:80**/index.html el puerto es el 80). Esto casi nunca se especifica porque ya hay definidos puertos por defecto para distintos protocolos: 20 para ftp-data, 21 para ftp, 79 para finger, etc. Algunos servers pueden definir otros puertos, e inclusive pueden utilizarse puertos disponibles para establecer conexiones especiales.
Justamente, una de las formas de crear un objeto de la clase URL permite especificar también el puerto:

```
URL url3 = new URL ("http", "www.rockar.com.ar", 80,"sbits.htm");
```

Para establecer una conexión a través de un socket, tenemos que programar por un lado el servidor y por otro los clientes.
En el servidor, creamos un objeto de la clase ServerSocket y luego esperamos algún cliente (de clase Socket) mediante el método accept():

```
ServerSocket conexion = new ServerSocket(5000);          // 5000 es el puerto en este caso
Socket cliente = conexion.accept();                      // espero al cliente
```

Desde el punto de vista del cliente, necesitamos un Socket al que le indiquemos la dirección del servidor y el número de puerto a usar:

```
Socket conexion = new Socket ( direccion, 5000 );
```

Una vez establecida la conexión, podemos intercambiar datos usando streams como en el ejemplo anterior. Como la clase URLConnection, la clase Socket dispone de métodos getInputStream y getOutputStream que nos dan respectivamente un InputStream y un OutputStream a través de los cuales transferir los datos.

Un servidor atento

Vamos a crear un servidor Ejemplo26a.java (que podemos correr en una ventana) que atenderá a un cliente de la misma máquina (lo vamos a correr en otra ventana). Para hacerlo simple, el servidor sólo le enviará un mensaje al cliente y éste terminará la conexión. El servidor quedará entonces disponible para otro cliente.
Es importante notar que, para que el socket funcione, los servicios TCP/IP deben estar activos (aunque ambos programas corran en la misma máquina). Los usuarios de Windows asegúrense que haya una conexión TCP/IP activa, ya sea a una red local o a Internet.
El servidor correrá "para siempre", así que para detenerlo presionen control-C.

```
// servidor
import java.io.*;
import java.net.*;

public class Ejemplo26a {

		public static void main(String argv[]) {
		  ServerSocket		servidor;
		  Socket	cliente;
		  int			numCliente = 0;
		  try {
				servidor = new ServerSocket(5000);
				do {
				  numCliente++;
				  cliente = servidor.accept();
				  System.out.println("Llega el cliente "+numCliente);
				  PrintStream ps = new PrintStream(cliente.getOutputStream());
				  ps.println("Usted es mi cliente "+numCliente);
				  cliente.close();
				} while (true);
		  }
		  catch (Exception e) {
				e.printStackTrace();
		  }
		}
}
```

Utilizamos un PrintStream para enviar los datos al cliente, ya que es sencillo de utilizar para mandar Strings. El método PrintStream.println maneja los datos como System.out.println, simplemente hay que indicarle el stream a través del cual mandarlos al crearlo (en este caso, el OutputStream del cliente, que obtenemos con cliente.getOutputStream()).

El cliente satisfecho

Ahora vamos a crear la clase cliente, Ejemplo26b.java. El cliente simplemente establece la conexión, lee a través de un DataInputStream (mediante el método readLine()) lo que el servidor le manda, lo muestra y corta.

```
// cliente:
import java.io.*;
import java.net.*;
```

```
public class Ejemplo26b {

        public static void main(String argv[]) {
          InetAddress       direccion;
          Socket    servidor;
          int               numCliente = 0;
          try {
                    direccion = InetAddress.getLocalHost();         // direccion local
                    servidor = new Socket(direccion, 5000);
                    DataInputStream datos =
                              new DataInputStream(servidor.getInputStream());
                    System.out.println( datos.readLine() );
                    servidor.close();
          }
          catch (Exception e) {
                    e.printStackTrace();
          }
        }
}
```

Para probar esto, asegúrense que los servicios TCP/IP estén activos, corran java Ejemplo26a en una ventana y corran varias veces java Ejemplo26b en otra. Las salidas serán más o menos así:

Ventana servidor:

```
C:\java\curso>java Ejemplo26a
Llega el cliente 1
Llega el cliente 2
Llega el cliente 3
        (----- cortar con control-C -----)
```

Ventana cliente:

```
C:\java\curso>java Ejemplo26b
Usted es mi cliente 1

C:\java\curso>java Ejemplo26b
Usted es mi cliente 2

C:\java\curso>java Ejemplo26b
Usted es mi cliente 3

        (----- aquí cerramos el servidor -----)

C:\java\curso>java Ejemplo26b
java.net.SocketException: connect
    at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:223)
    at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:128)
    at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:115)
    at java.net.Socket.<init>(Socket.java:125)
    at java.net.Socket.<init>(Socket.java:101)
     at Ejemplo26b.main(Ejemplo26b.java:12)
```

Esto es todo por ahora. El ejemplo fue lo más sencillo posible, pero mediante el mismo método el servidor y los clientes pueden intercambiar datos escribiendo en ambas direcciones. Inclusive, el servidor puede correr en una máquina y los clientes en otras; además, si bien en este caso utilizamos aplicaciones standalone, se pueden utilizar applets.
Por ejemplo, una aplicación servidora puede correr constantemente en un server de Internet (por ejemplo, para buscar datos en una base de datos) y diferentes clientes en distintas máquinas, posioblemente applets Java, pueden conectarse a ella y consultarla.