FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

Programación Básica en Java

Instructores :
Ing. José Luis Santiago Rodríguez
Ing. Arturo Velásquez Mayoral

Facultad de Ingeniería, UNAM
División de Educación Continua
Abril 1997

Programación Básica en Java

José Luis Santiago Rod
jlsantiago@iserve.net.mx
Arturo Velásquez Mayoral
avelasqu@mpsnet.com.mx

Facultad de Ingeniería, UNAM
División de Educación Continua
Abril 1997

El origen de Java

Java, cuya denominación original fue Oak (1991), fue diseñado para programar dispositivos electrónicos de consumo y crear una red heterogénea de productos electrónicos domésticos. Para funcionar en este entorno, Java fue definido como un intérprete de tiempo real fiable y pequeño que, sobre todo, debía ser portable y funcionar a través de canales de comunicación. La primera aplicación práctica proyectada por el equipo de Java fue utilizar Java en decodificadores de televisión. Después, a mediados de 1994, los ingenieros de Sun se dieron cuenta de que podían utilizar Java para crear un visualizador del Web. Así apareció HotJava, un navegador escrito totalmente con lenguaje Java, el cual, permite acceder a webs que contienen applets Java.

Características de Java

Independencia de la plataforma hardware.

El entorno de desarrollo de Java utiliza un compilador (javac) para generar los bytecodes, como se ha citado antes, independientes de la plataforma. Para ejecutar el programa se llama a un intérprete de bytecodes (denominado simplemente java), el cual sí depende de cada arquitectura. Este interprete está escrito en ANSI C con los tipos de datos definidos por el IEEE, para que resulte portable a cualquier plataforma. El intérprete ocupa tan solo 40 KBytes de memoria activa (aunque con librerías adicionales puede llegar hasta 175 KBytes).

Características de Java

Lenguaje de programación orientado a objetos.

Java es un lenguaje de programación orientado a objetos (por tanto soporta las tres características de este tipo de programación: Encapsulación, herencia y polimorfismo).
Java se basa en C++, con una sintaxis similar, pero está simplificado para evitar las características más problemáticas de C++ (no existe aritmética de punteros en Java, los strings y los arrays se consideran objetos y la gestión de la memoria es automática), lo que hace más fácil la programación en Java. Java incluye un conjunto de librerías de clases para obtener los tipos de datos básicos, procedimientos de entrada/salida, comunicaciones a través de red, lleva integrados los protocolos de Internet (TCP/IP, HTTP y FTP) y funciones para desarrollar interfaces de usuario.

Qué es un objeto?

Se puede decir que todo puede verse como un objeto. Pero siendo 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, que la misma definición del objeto incluye tanto los datos que éste usa (atributos) como los procedimientos (métodos) que actúan sobre los mismos.

Qué es un objeto?

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.

La ventaja de esto es que como no hay programas que actúan modificando al objeto, éste se mantiene en cierto modo independiente del resto de la aplicación. Si es necesario modificar objeto (por ejemplo, para darle nuevas características), esto se puede hacer sin tocar el resto de la aplicación... lo que ahorra mucho tiempo de desarrollo.

Qué es un objeto?

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, la 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 jerarquía de clases, se puede reutilizar mucho código inclusive sin saber lo que tiene adentro.

Un ejemplo simple

```
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);
}
}
```

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).

Características de Java

Robustez.

Java se puede considerar un lenguaje robusto. A diferencia de C++, con el que resulta sumamente fácil tener que reiniciar la computadora por culpa de algún error de programación, obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Java soporta apuntadores, pero no la aritmética propia que éstos tienen en C++. Se implementan auténticos, en vez de listas enlazadas de apuntadores.
De esta manera, se consigue evitar el problema de corrupción de memoria resultante de apuntadores que señalan a zonas equivocadas.

Características de Java

Multienhebramiento (multithreading).

Java puede trabajar con sistemas operativos de alto nivel que soporten multienhebramiento. De esta forma, un programa Java puede tener más de una hebra en ejecución. Por ejemplo, podría realizar un cálculo largo en una hebra, mientras otras interactúan con el usuario. Así los usuarios no tienen que trabajar mientras los programas Java completan las operaciones más largas.

Perspectivas de Java

Sun ha adoptado la misma estrategia de mercado que adoptó en su momento Netscape, la distribución gratuita de sus productos: el entorno de desarrollo Java (el JDK) y el visualizador HotJava.. Creando así una masa de usuarios que permita el establecimiento de un estándar de facto. Otras empresas, como Borland y Symantec, han incorporado módulos para la implementación de Java en sus paquetes de desarrollo C/C++. Sega ha anunciado tecnología Java (hardware y software) para sus próximos video-juegos.

La difusión de Java ha llegado a límites insospechados. Cada vez que un programador Java se encuentra ante la incertidumbre de optimizar o no su tiempo de producción. Debe decidir si implementar el algoritmo que necesita en cada momento o dedicarse a buscarlo enInternet.

Perspectivas de Java

Entre sus competidores, AT&T está trabajando en las especificaciones de un nuevo lenguaje, llamado Inferno, con el que pretende robar cuotas dé mercado a Sun. Microsoft parece seguir apostando por VisualBasic, pero la red de la "gestión de la complejidad" no ha sido la especialidad del gigante de las ventanas.

A lo largo del segundo semestre de 1996, de la mano de la especificación JDBC, Sun comenzó a licenciar unos drivers similares a los ODBC´s de Microsoft, para el acceso a bases de datos comerciales (Informix, Oracle, Sybase,...) desde Java, cubriendo con esta estrategia uno de los terrenos que hasta ahora tenía más abandonados, la informática dirigida a los sistemas de información.

Lo más reciente de Java

Aparece la primera revisión del, entorno de desarrollo J.D.K.-1.1, compatible con la versión anterior. Aunque el código es compatible, algunos desperfectos de diseño han sido corregidos.

El A.W.T. (las clases para el manejo de interfaces gráficos) para Win32 ha sido reescrito totalmente, mejorando su rendimiento.

Verdadero soporte internacional, aunque ya en la versión 1.0 se sentaron las bases de la internacionalización del lenguaje, es posible la visualización de caracteres UNICODE (de 16 bits), fechas y horas sensibles a la zona, ordenaciones alfabéticamente correctas, conversión de caracteres, creación de identificadores UNICODE, etc.

Lo más reciente de Java

Herramientas para la incorporación de elementos de seguridad en aplicaciones Java: criptografía, firmas digitales, manejo de "llaves" y control de acceso.

Han sido corregidas las deficiencias encontradas en el A.W.T. de la versión anterior, incluyendo API,s para impresoras, menús pop-up, métodos para el manejo del portapapeles y un nuevo modelo para el manejo de eventos.

La serialización de objetos: un mecanismo para la creación de objetos persistentes.

Aparecen los Java Archives (JARS): un formato de archivos que nos permitirá encerrar dentro de un único archivo todos los componentes de nuestro proyecto: clases, imágenes, ficheros audio, etc. El archivo guardará una versión comprimida (formato ZIP) de los elementos del proyecto, viajando por la red de forma que consuma el menor ancho de banda posible.

Lo más reciente de Java

Capacidad para crear aplicaciones distribuidas, invocando métodos remotos (RMI) desde diferentes máquinas virtuales Java, situadas en diferentes servidores.
Se añaden nuevas funcionalidades a los paquetes de red (java.net) y entrada/salida (java.io).
Incluye un interface uniforme de acceso a bases de da relacionales, el JDBC.
El número de clases aumenta hasta llegar a las 1336 (ocu 7MB), frente a las 580 de la especificación anterior (ocupa 1,5MB).
"Lo malo" es que todavía hay que esperar a que aparezcan los primeros navegadores compatibles con esta especificación, pa aprovechar el 100% de sus nuevas posibilidades.

Los JavaBeans

Junto con el nuevo kit de desarrollo Java, aparece la especificación de los JavaBeans (componentes Java). Los JavaBeans harán uso de los nuevos mecanismos encontrados en el J.D.K.-1.1: serialización de objetos y su mecanismo de persistencia, las nuevas funcionalidades del A.W.T., la invocación de métodos remotos, etc.

Un JavaBeans es, en palabras de JavaSoft: "un componente software reutilizable que puede ser manipulado visualmente herramienta de construcción" (es algo semejante a los contro de MicroSoft, o a sus recientes ActiveX). Como "componente podemos entender: elementos GUI tipo botones o barras de desplazamiento, visualizadores de bases de datos, e incluso aplicaciones más sofisticadas tipo procesadores de texto u hojas cálculo.

Los JavaBeans

La finalidad es conseguir un modelo software para la creación y manipulación de componentes Java, así terceras partes podrán desarrollar componentes que podremos incorporar en nuestras aplicaciones. Los componentes nos llegarán incl desde el servidor de nuestros proveedores de componentes. ta forma, ya sentados delante de nuestro Network Computer, no te que preocuparnos por la aparición de nuevas versiones, del p componentes no utilizados, etc.
Ya es posible una programación a la carta, seleccionando di componentes que vienen de diferentes máquinas y con un cos utilización del componente.
Empiezan a sentarse las bases para la creación de mercados de objetos que circularán libremente por la red.

¿Que se requiere para trabajar con Java?

Todo lo que se necesita para desarrollar aplicaciones en Java está en:

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

En particular, debe bajarse el JDK y el API Documentat

http://java.sun.com/java.sun.com/products/JDK/1.0.2/in

Algunas direcciones útiles y bilbiografía

- *http://www.gamelan.com*
- *http://www.javaworld.com*
- Vanhelsuwé L., Phillips I., et al, "Mastering Java", 1996
- Lemay, L. - Perkins, C. "Teach yourself Java in 21 day
  Indianapolis IN, Sams Net 1996.

El Java Developers Kit (JDK)

javac	compilador
java	intérprete
appletviewer	intérprete para ejecutar applets enidos en archivos HTML
javadoc	Generador de documentación basada ML
jdb	Debugger
javah	generador de header files para C
javap	Desensamblador para mostrar las funcione accesible y los datos en una clase compilad

En realidad se puede decir que hay tres Javas

Javascript: es una versión de Java directamente interpretada, que se incluye como parte de una página HTML

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)

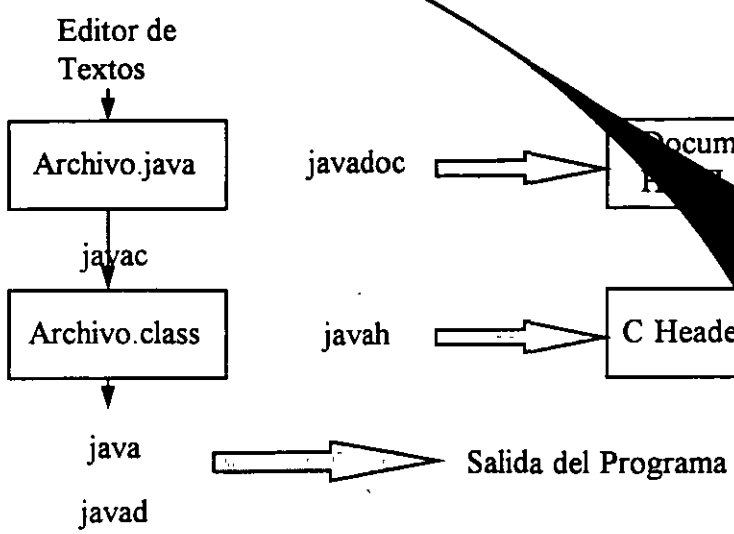
Aplicaciones en Java

Las aplicaciones son programas independientes y más generales escritos en lenguaje Java. Estas aplicaciones no necesitan de un navegador para ejecutarse, de hecho, se puede utilizar Java para escribir un programa como lo haríamos con C o Pascal.

Para ejecutar estos programas, se debe utilizar el intérprete

Por ejemplo, el navegador de Web creado por Sun, HotJava, es una aplicación escrita íntegramente en lenguaje Java.

Construcción de aplicaciones Java



Hello World

```
class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello World!");
  }
}
```

javac HelloWorld.java

Applets

Como los Applets se ejecutan dentro de un navegador, por lo tanto tienen las mismas capacidades que el navegador : Gráficos sofisticados, elementos de interfaz de usuario, funciones de red y funciones para tratar eventos generados tanto por el usuario como por el sistema.

Las ventajas de los Applets sobre las aplicaciones respect[illegible]os e interfazs de usuario, se ven mermadas por fuertes restric[illegible]

Dado que los Applets residentes en un servidor se ejecutan en [illegible] sistema cliente, son necesarias ciertas restricciones para preven[illegible] un Applet pueda producir efectos no deseados (a nivel de seguri[illegible]

Restricciones de los Applets

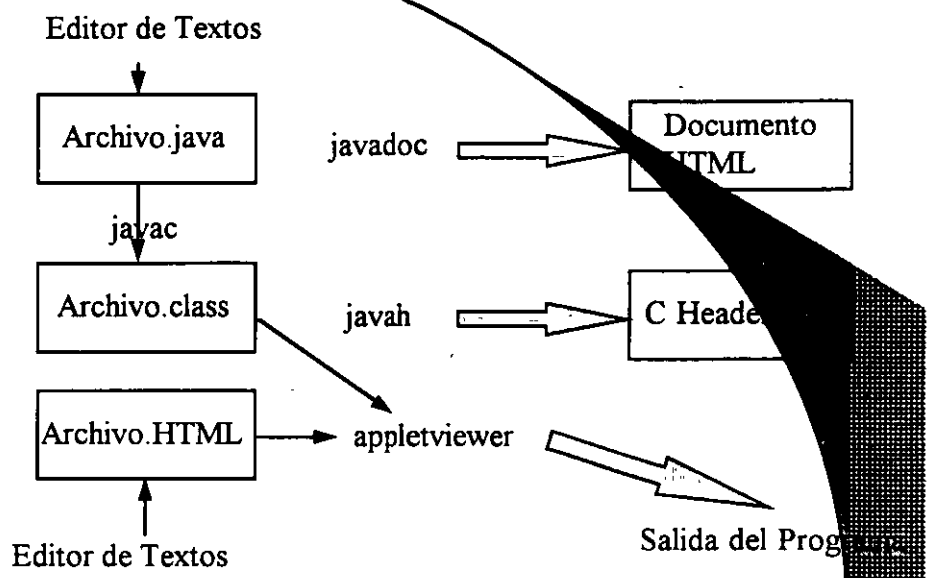
A.Los Applets no pueden leer o escribir en el sistema de archivos del cliente, excepto en directorios específicos (los cuales son especificados por el usuario a través de una lista de control, la cual, por defecto, se encuentra vacía). Algunos visualizadores no permiten a un Applet realizar una operación de lectura o escritura en ningún caso.

B.Los Applets no pueden establecer una comunicación con otro servidor que no sea el que contiene el Applet.

C.No se permite a un Applet ejecutar un programa de los que se encuentran en el sistema cliente.

D.Tampoco se permite a un Applet cargar programas nativos en la estación cliente, incluyendo librerías de acceso dinámico (DLLs).

Construcción de Applets usando el JDK



Ejemplo de un applet

```
import java.applet.*;
import java.awt.* ;
public class Corre extends Applet
{       Image n;
        int a; int b;
        public void init()
      {     n=getImage(getCodeBase(),"corre.gif");
            resize(300,500);
        }
        public void paint(Graphics g)
        {       g.drawImage(n,a,b,this);
        }
```

continuación

```
        public boolean mouseUp(Event e,int x,int y)
        {       a=x;
                b=y;
                repaint();
                return true;
        }
}
```


HTML para Applets

Los archivos HTML son archivos de texto con secuencias de caracteres especiales, denminadas "tags" que especifican las características de formato de un documento.
Un archivo HTML contiene "tags" de formato y de estructura

```
<HTML>
        <HEAD>
        <TITLE> Ejemplo de Documento HTML </TITL
        </HEAD>
        <BODY>
                <H1> DEMOSTRACION</H1>
        </BODY>
</HTML>
```

HTML para Applets

Para desplegar en un visualizador que soporte Java, el applet del ejemplo anterior, puede usarse el siguiente archivo HTML:

```
<HTML>
 <HEAD>
  <TITLE> Ejemplo de un applet </TITLE>
 </HEAD>
 <BODY>
  <H1> Presione con el mouse en diferentes lugares del área en que cor
  </H1>
  <P>
  <APPLET CODE = "Corre.class" WIDTH=600 HEIGHT = 220 >
  </APPLET>
 </BODY>
</HTML>
```


Javascript

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 visualizadores lo soportan completamente;

Ejemplo de un Javascript

```
<!-
- Documento: calendario.html
- Autor   : Arturo Velásquez Mayoral-
->

<html>
<head>
<title>Calendario digital</title>
<script language="javascript">
<!-
 var nHora;
 var nMinuto;
 var nSegundo;
 var nDia;
 var nMes;
 var nAnnio;
```


Ejemplo de un Javascript

continuación

```
function getData () {
 var cHTMLOut;
  var cImagen='<img src="/imagenes/';
  var cImagenExtension='.gif" width=32 height=32>';
 var dAhora=new Date();
 nHora=dAhora.getHours();
  nMinuto=dAhora.getMinutes();
  nSegundo=dAhora.getSeconds();
 nDiaSemana=dAhora.getDay(),
 nDia=dAhora.getDate();
  nMes=dAhora.getMonth();
  nAnnio=dAhora.getYear();
 dAhora=null;
 if (nDiaSemana==1) nDiaSemana="Lunes",
   else if (nDiaSemana==2) nDiaSemana="Martes";
   else if (nDiaSemana==3) nDiaSemana="Mi&eacute;rcoles";
   else if (nDiaSemana==4) nDiaSemana="Jueves";
   else if (nDiaSemana==5) nDiaSemana="Viernes";
   else if (nDiaSemana==6) nDiaSemana="S&aacute;bado";
   else if (nDiaSemana==0) nDiaSemana="Domingo";
```

Ejemplo de un Javascript

continuación

```
CheckData();

cHTMLOut="<center><table><tr><td>Seg&uacute;n tu PC es ";
cHTMLOut+=nDiaSemana+",</td><td><table border><tr><td>";

for (nID=0;nID<nDia.length;nID++)
  cHTMLOut+=cImagen+nDia.substring(nID,nID+1)+cImagenExtension;

 cHTMLOut+=cImagen+"sepfecha"+cImagenExtension,

for (nID=0;nID<nMes.length;nID++)
  cHTMLOut+=cImagen+nMes.substring(nID,nID+1)+cImagenExtension;

 cHTMLOut+=cImagen+"sepfecha"+cImagenExtension;

for (nID=0;nID<nAnnio.length;nID++)
  cHTMLOut+=cImagen+nAnnio.substring(nID,nID+1)+cImagenExtension;

cHTMLOut+="</td></tr></table></td>";
cHTMLOut+="<td>a las</td><td><table border><tr><td>";
```


Ejemplo de un Javascript

continuación

```
for (nID=0;nID<nHora.length;nID++)
  cHTMLOut+=cImagen+nHora.substring(nID,nID+1)+cImagenExtension;

 cHTMLOut+=cImagen+"sephora"+cImagenExtension;

for (nID=0;nID<nMinuto.length;nID++)
  cHTMLOut+=cImagen+nMinuto substring(nID,nID+1)+cImagenExtens

 cHTMLOut+=cImagen+"sephora"+cImagenExtension;

for (nID=0;nID<nSegundo.length;nID++)
  cHTMLOut+=cImagen+nSegundo substring(nID,nID+1)+cImagenExtension;

cHTMLOut+="</td></tr></table></td><td>horas.</td></tr></table></center>";

return(cHTMLOut);
}
```

Ejemplo de un Javascript

continuación

```
function CheckData () {
 nHora=""+nHora;
 if (nMinuto<10) nMinuto="0"+nMinuto; else nMinuto=""+nMinuto;
 if (nSegundo<10) nSegundo="0"+nSegundo; else nSegundo=""+nSegundo;
 nDia=""+nDia;
 nMes=++nMes;
 nMes=""+nMes;
 nAnnio=""+nAnnio;
}
function reloj() {
 document.write(getData());
}
//-- Final Funciones Fecha -->
</script>
</head>
<script language="javascript">
<!--
 reloj();
//-- Final JavaScript -->
</script>
<br>
</body>
</html>
```


La Gramática de Java

La gramática de un lenguaje define como y cuando pueden usarse las palabras, así como la puntuación requerida.

COMENTARIOS
Marcas agregadas por el programador con propósitos de documentación.

/* */	comentarios de varias líneas
//	comentario hasta el final de la línea
/** */	comentarios con un significado especial para el javadoc

La Gramática de Java

BLOQUES DE CODIGO

Los enunciados pueden agruparse en bloques, de modo que un solo enunciado pueda controlar la ejecución de varios de ellos.

En Java se utilizan llaves
{
}
para delimitar un bloque

La Gramática de Java

ESTRUCTURA DE LOS ARCHIVOS FUENTES

package statement
: Define el paquete al que pertenecerán las clases en el archivo

import statement
: Establece un camino corto para referirse a clases existentes, solamente por el nombre de la clase sin especificar el nombre completo del paquete

class statement
: Define las clases del usuario

La Gramática de Java

PALABRAS RESERVADAS

Palabras con un significado especial para el compilador

abstract	boolean	break	byte	cas	catch
char	class	const*	continue	defaul	do
double	else	extends	final	finally	
for	goto*	if	implements	import	
int	interface	long	native	new	nu
package	private	protected	public	return	shor
static	super	switch	synchronized	this	throw
throws	transient*	try	void	volatile	while

* Palabras reservadas que actualmente no se usan

La Gramática de Java

IDENTIFICADORES

Un identificador es un nombre dado a una variable, clase o función.
Deben comenzar con una letra y no pueden ser palabras reservadas.
Es recomendable seguir las convenciones del API.

Nombres de clases	Mayúsculas al principio de ca dentro del identificador
Nombres de funciones	Mayúsculas al principio de cada p dentro del identificador, excepto la
Nombres de variables	Mayúsculas al principio de cada pala dentro del identificador, excepto la pri
Constantes	Todo en mayúsculas, subrayado entre palabras

La Gramática de Java

LITERALES

Una literal es un valor actual, su formato depende de cada tipo de dato:

byte, short, int	Dígitos decimales (sin iniciar con 0) 0x seguido de dígitos hexadecimales 0 seguido de dígitos octales
long	igual al anterior, pero seguido del caracte l o L
float	digitos con punto decimal y/o exponente, seguidos del carácter f o F
double	igual al anterior, pero sin la f o F y con una d o D opcional

La Gramática de Java

LITERALES

boolean	true o false
char	Un carácter ASCII entre comillas sencillas o una secuencia de escape predefinida entre comillas sencillas
string	secuencia de caracteres o secuencias de escape entre comillas dobles

La Gramática de Java

Secuencias de Escape

\b	backspace
\t	tabulador
\n	linefeed
\f	formfeed
\r	carriage return
\"	comillas dobles
\'	comillas sencillas
\\	backslash

La Gramática de Java

EXPRESIONES Y OPERADORES

Las expresiones son combinaciones de variables, palabras reservadas o símbolos que son evaluadas a algún tipo de valor.

OPERADORES ARITMETICOS

++, --	autoincremento, autodecremento
+, -	más uno, menos uno
*	multiplicación
/	división
%	residuo
+, -	adición, substracción

La Gramática de Java

OPERADORES RELACIONALES

>, <, >=, <=	Prueban la magnitud relativa
==	Prueban igualdad
!=	Prueban desiguladad
?:	condición, devuelve uno de dos operandos dependiendo de un tercero

La Gramática de Java

Operadores Lógicos

!	Not
&	And
^	XOR
|	Or
&&	And condicional
||	Or condicional

La Gramática de Java

Operadores de desplazamiento de bits

~	Not (complemento)
<<, >>	Desplazamiento a la izquierda o a la derecha
>>>	Desplazamiento a la derecha, sin el bit de signo
&	AND
^	XOR
|	OR

La Gramática de Java

Declaración de variables:

tipo identificador [=default] {, identificador [=default] } ;

Tipos numéricos

byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807
float	32 bits	positivo 1.40239846E45 negativo -3.40282347E38	positivo 3.40282347E38 negativo -1.40239846E-45
double	64 bits	4.94065645841246544E-324 1.79769313486231570E308	

Las estructuras de control

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;
}
```


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 c

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

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

```
while (expresión_booleana) {
    instrucciones...
}
```

Do...while
Similar al anterior, sólo que la condición se evalúa al final y no al principio:

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


For

Sirve para ejecutar en forma repetida una serie de instrucciones:

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

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 Ciclos {
   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

Otras

Hay otras instrucciones que controlan el flujo del programa:

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

Un ejemplo para crear un archivo HTML con Java

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

public static void main (String[] args) throws IOException {

FileOutputStream file;
PrintStream html;

        if (args.length == 0) {
                System.out.println("Falta el nombre del applet.");
                System.exit(10),
        }
        if (args[0].indexOf("?") != -1 || args[0].equals("-h")) {
                System.out.println("Uso: GenAppletHTML <Nombre del applet
                System exit(0);
        }
```


Un ejemplo para crear un archivo HTML con Java

```
        if (!(
                Character.isLowerCase(args[0].charAt(0)) ||
                Character.isUpperCase(args[0].charAt(0))
          ) {
                System.out.println(""+args[0]+" No es un nomb  clase correcto");
                System exit(10);
        }
        if (Character.isLowerCase(args[0].charAt(0) )) {
                System.out.println("LOS NOMBRES DE CLASES, DEB
CONVENCION INICIAR CON MAYUSCULAS");
                System.out.println(args[0]),
                System.out.println("^");
                System.out.println("ES MINUSCULA.");
                System.exit(10),
        }

        file = new FileOutputStream("Pagina html");
        html = new PrintStream(file);
```

Un ejemplo para crear un archivo HTML con Java

```
        html.print("<HTML><HEAD></HEAD><BODY><HR><APPLET CODE=");
        html.print(args[0] + ".class");
        html.println(" WIDTH=400, HEIGHT=300>");
        html.println("</APPLET><HR></BODY></HTML>");
        html.close();
}}
```

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

Programación Básica en Java

Instructores :
Ing. José Luis Santiago Rodríguez
Ing. Arturo Velásquez Mayoral

Facultad de Ingeniería, UNAM
División de Educación Continua
Abril 1997

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

Las clases en Java

En Java hay gran cantidad 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.applet - clases para la creación de applets y recursos para reproducción de audio.

Las clases en Java

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);
```

Las clases en Java

```
    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);
}
```


Ejemplo de una clase

```
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;                   // inicializa
 }
 // Métodos
 public int incCuenta() {    // Un método para incrementar
  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
 }
}
```

Como funciona una clase

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; una clase puede tener más de un constructor (no así un objeto o instancia, 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.

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.

Ejemplo para utilizar la clase Contador

```
import java.io.*;              // Uso la biblioteca de entradas/salidas
public class EjemploContador {
 static int n;
 // una variable tipo Contador para instanciar el objeto
 static Contador laCuenta;

 public static void main ( String args[] ) {
  System.out.println ("Cuenta ");          // Imprimo el título
  laCuenta = new Contador();               // Crea una instancia del Co
  System.out.println (laCuenta.getCuenta()); //Imprime el valor actua
  n = laCuenta.incCuenta();                // Asignación e incremento
  System.out.println (n);
  laCuenta.incCuenta();                    // Incremento sin usar el valor
  System.out.println (laCuenta.getCuenta()); // imprime
  System.out.println (laCuenta.incCuenta()); // Todo en un paso
 }
}
```

Veamos la diferencia con un applet que haga lo mismo

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

public class EjemploContador2 extends Applet {
 static int n;
 static Contador laCuenta;

 // Constructor
 public EjemploContador2 () {
  laCuenta = new Contador();
 }
```


continuación

```
// El método paint se ejecuta cada vez que hay que redibujar
 // Notar el efecto al cambiar de tamaño la ventana del visualizador
 public void paint (Graphics g) {
  g.drawString ("Cuenta...", 20, 20);
  g.drawString (String.valueOf(laCuenta.getCuenta()), 20,
  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 );
 }
}
```

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 Interfaz1 [, Interfaz2]...]

o bien, para interfaces:

[public] interface Interfaz [extends InterfazMadre1 [, InterfazMadre2]...]

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 (un paquete, 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.
¿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.

De modo que una clase no puede ser final y abstract a la vez.

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).

Implements
Una interfaz (interface) es una clase que declara sus métodos pero no los implementa. Esto sirve para dar un ascendiente común a varias clases, obligando a implementar los mismos métodos y, por lo tanto, a comportarse de forma similar en cuanto a su interfaz con otras clases y subclases.

Interface
Las interfases pueden, asimismo, descender de otras interfaces 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 interfaz).

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 dice que son miembros de la clase y son accesibles por todos los métodos de la

Por otra parte, además de los atributos de la propia clase se puede acceder a los atributos de la clase de la que desciende.

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 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	(paquete)
la propia clase	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

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 ;

Static y final
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.

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 el objeto, o sea que no forman parte del estado permanente del mismo.
Volatile se utiliza con variables modificadas asincrónicamente por distintos 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 métodos

[private|protected|public] [static] [abstract] [final] [native] [synchronized]
TipoDevuelto NombreMétodo ([tipo1
nombre1[, tipo2 nombre2]...]) [throws excepción1 [,excepción2]...]

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). El valor de retorno se especifica con la instrucción return, 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 declaración.

Los métodos

El resto de la declaración
Public, private y protected actúan exactamente igual para los métodos que para los atributos.
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.
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 [illegible] contenga al menos un método abstracto que no esté implementado en [illegible] una clase abstracta.

Es final un método que no puede ser redefinido por ningún descendiente de la clase.

Los métodos

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 exist

Las clases synchronized permiten sincronizar varios threads para [illegible]n que dos o más accedan concurrentemente a los mismos datos.

La cláusula throws sirve para indicar que la clase genera determinadas excepciones.

Ejemplo de integración de conceptos

```
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);
    }
```

continuación

```
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(),
}
```


continuación

```
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;
}
```

continuación

```
// 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);
}

public static final Complejo DivEscalar(Complejo c, float f) {
      return new Complejo(c.x/f,c.y/f);
}
```


continuación

```
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);
}
}
```

Algunas observaciones

La única biblioteca que se usa es java.lang y se incluye automáticamente.

La clase es public final, lo que implica que cualquier clase en éste u otros paquetes puede utilizarla, pero ninguna clase puede heredarla.

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.

Algunas observaciones

```
// 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 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).

Algunas observaciones

Con parámetros, en cambio, devuelven la norma o el conjugado del parámetro:

```
// 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, por lo que debemos implementarlos aquí.

```
public final String toString()
```

Este método sirve para representar el complejo como una cadena de caracteres de la forma x+iy.

Algunas observaciones

```
// 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, [illegible])
    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.

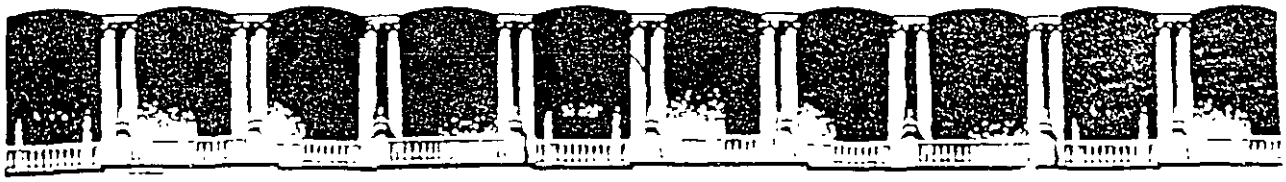
Ejemplo de uso de la clase

```
import java.io.*;

public class EjemploComplejo {

 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());
 }
}
```



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

Programación Básica en Java

Instructores :
Ing. José Luis Santiago Rodríguez
Ing. Arturo Velásquez Mayoral

Facultad de Ingeniería, UNAM
División de Educación Continua
Abril 1997

AWT y GUI

En Java, la clase Window (descendiente de Container), en la biblioteca java.awt, permite implementar ventanas 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 de sí otros elementos (incluyendo otros paneles).

El siguiente ejemplo crea una ventana que no hace nada pero contiene varios elementos.

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).

ejemplo de una ventana

```
import java.awt.*;

public class PrimeraVentana extends Frame {
   boolean inAnApplet = true;

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

continuación

```
public PrimeraVentana() {
   Panel panelAlto = new Panel();
   panelAlto.add("West", new Label("Etiqueta", Label.CEN
   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);
 }
```


continuación

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

Algunas observaciones

La clase desciende de Frame o sea que será una ventana con borde.
Vamos a usar la variable inAnApplet para saber si se arrancó como applet o como aplicación standalone (hay que cerrarla en manera diferente en cada caso)

Si se llama como aplicación standalone, lo primero que se ejecuta es main(...);
en este caso la aplicación crea una instancia de PrimeraVentana (invocando el constructor PrimeraVentana() 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
show que muestra la ventana

Es importante NO confundir el objeto (instancia) window con la clase Window

Algunas observaciones

Si se carga como applet, entonces se ejecuta el constructor PrimeraVentana() como en el caso anterior:
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
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);
```

Algunas observaciones

Como la clase PrimeraVentana 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.

Veamos la estructura:

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


Algunas observaciones

Vale la pena notar que se han 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 cadena que depende del LayoutManager, el objeto que se encarga de ordenar los componentes dentro del contenedor.

LayoutManager es una interfaz, y como tal debe implementarse a través de objetos no abstractos de los que hay varios predefinidos en la librería java: BorderLayout, CardLayout, FlowLayout, GridBagLayout y GridLayout.

Algunas observaciones

El Layout por defecto es BorderLayout, que define en el contenedor las áreas "North", "South", "West", "East" y "Center" y es el que usa

CardLayout permite "apilar" los componentes como cartas y ver una vez, FlowLayout los ordena de izquierda a derecha como un texto, GridLayout los ordena en una cuadrícula donde cada componente tiene un tamaño fijo, GridBagLayout los pone en una cuadrícula pero cada uno puede tener el tamaño deseado.

No hace falta llamar, en el caso del applet, a Pack() y Show().

95

Algunas observaciones

Y los eventos...

Se redefinió 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.

96

Algunas observaciones

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 PrimeraVentana) 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):

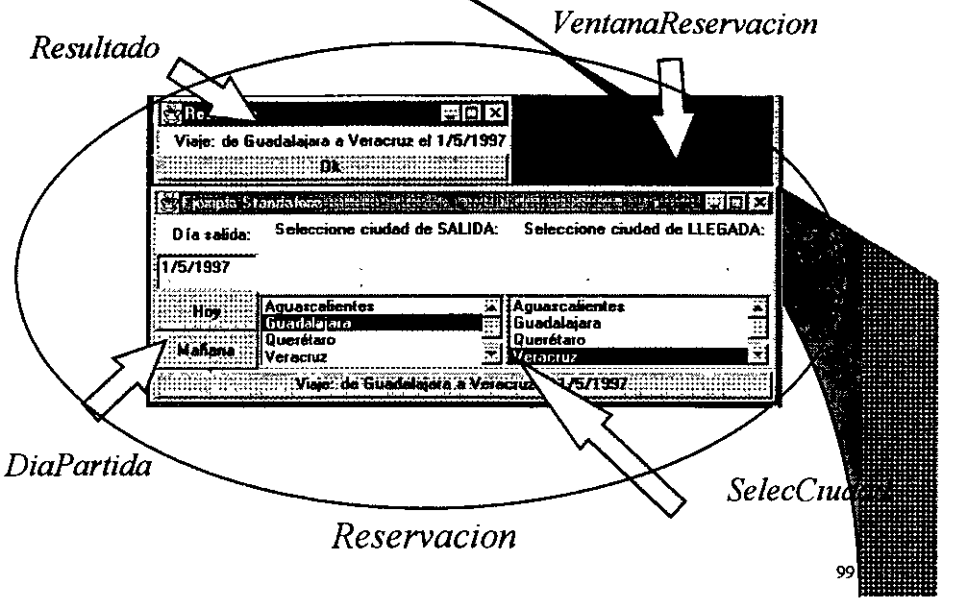
En ese caso, si nuestro ejemplo se ejecutó como aplicación llamamos al método System.exit(0), que cierra la aplicación; y si era un applet llamamos a dispose(), implementación de un método de la interface ComponentPeer que se encarga de remover todos los componentes y la propia ventana.

Algunas observaciones

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 asciende monta 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 (si alguno 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.

Ejemplo de un Sistema de Reservaciones



VentanaReservacion

```
import java.awt.*;

class VentanaReservacion extends Frame {
SelecCiudad    cs;          // ciudad de salida
SelecCiudad    cl;          // ciudad de llegada
DiaPartida     dp;          // día de salida
Button         ok;          // botón de reservación
boolean enApplet;       // para indicar si es un applet o no

VentanaReservacion (String titulo, boolean enApplet) {    // constructo
super(titulo);
this.enApplet = enApplet;
dp = new DiaPartida();              // dia de salida
add ("West", dp),
cs = new SelecCiudad("SALIDA"); // ciudad de salida
add ("Center", cs);
cl = new SelecCiudad("LLEGADA");       // ciudad de llegada
```

VentanaReservacion

```
add ("East", cl);
ok = new Button("cualquiera");
ActualizaBoton();
add("South",ok);
pack();                    // dimensionamos la ventan
show();                    // y la mostramos!
}

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();
```


VentanaReservacion

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

String ActualizaBoton() {
   StringBuffer b = new StringBuffer("Viaje: de ");
   if (cs.getDescription() != null) b.append(cs.getDescriptio
     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());
   return b.toString();
}
```

VentanaReservacion

```
void Activar() {
 if ((cs.getDescription() != null) && (cl.getDescription() != null))
   Resultado result = new Resultado("Resultado", ActualizaBoton());
 else ok.setLabel("Especificación incompleta!");
}

}
```

103

DiaPartida

```
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);
}
```

104

DiaPartida

```
private String GetHoy() {
  Date d = new Date();
  int dia = d.getDate();
  int mes = d.getMonth()+1;
  int anio = d.getYear()+1900;
  return dia+"/"+mes+"/"+anio;
}
private String GetManana() {
  Date d = new Date();
  int dia = d.getDate()+1;
  int mes = d.getMonth()+1;
  int anio = d.getYear()+1900;

 switch (mes) {
  case (1):
  case (3):
  case (5):
  case (7):
  case (8):
```


DiaPartida

```
case (10): if (dia>31) {
        dia = 1;
        mes++;
 }
 break;
 case (12): if (dia>31) {
        dia = 1;
        mes = 1;
        anio++;
 }
 break;
 case (4):
 case (6):
 case (9):
 case (11): if (dia>30) {
        dia = 1,
        mes++;
 }
 break;
```

DiaPartida

```
 default: if (dia>28) {
          dia = 1;
          mes++;
  }
}
return dia+"/"+mes+"/"+anio;
}

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

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


DiaPartida

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

}
```

SelecCiudad

```
import java.awt.*;
class SelecCiudad extends Panel {
private List   listaCiudades;

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

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:
listaCiudades = new List (4, false);
listaCiudades.addItem("Aguascalientes");
listaCiudades.addItem("Guadalajara");
listaCiudades.addItem("Querétaro");
```

109

SelecCiudad

```
listaCiudades.addItem("Veracruz");
listaCiudades.addItem("Zacatecas"),
add("South", listaCiudades);
}

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

110

Resultado

```
import java.awt.*;
class Resultado extends Frame {
Button  r_ok;
Resultado (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);
}
```

111

Reservación

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

public class Reservacion extends Applet {

public static void main (String arg[]) {
new VentanaReservacion("Ejemplo Standalone", true);
}

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

}
```

112

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

Programación Básica en Java

Instructores :
Ing. José Luis Santiago Rodríguez
Ing. Arturo Velásquez Mayoral

Facultad de Ingeniería, UNAM
División de Educación Continua
Abril 1997

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987. Fax 510-0573 521-4020 AL 26

```
            append(" Convertible.acelerar()");
            super.acelerar() ;
        }
        public void descapotar() { append(" Descapotar() ") ; }
        public static void main(String args[ ]) {
            Convertible x = new Convertible() ;
            x.encender() ; x.acelerar() ;
            x.frenar() ; x.apagar() ;
            x.descapotar() ; x.print() ;
            System.out.println("Probando la clase base :") ;
            Auto.main(args) ;
        }
}
```

Cuando se hereda de una clase, se obtiene un subobjeto de la clase base dentro de la derivada. Java automáticamente inserta llamadas al constructor de la clase derivada. Para llamar explícitamente a algún constructor de la clase base se utiliza la palabra reservada **super**, con la lista de argumentos que sea necesario utilizar.

Ejercicio

1) Defina un constructor en la clase Auto.
   Modifíquelo para que reciba como argumento el nombre del auto.
   Cree el constructor apropiado en la clase Convertible.
2) Implemente la jerarquía de clases :
   Arte<---- Dibujo<---- Caricatura
   Cada una con su respectivo constructor por default (sin argumentos).
   Cree una instancia de la clase Caricatura y explique la salida del programa.
   Modifique el constructor de la clase Arte para que reciba un argumento.

La composición es generalmente utilizada cuando se requieren características de una clase existente, mas no su interfaz. Es común combinar composición y herencia para crear clases. Por ejemplo se pueden crear objetos miembros públicos en la definición de una clase. Esto es seguro, siempre que los objetos utilizados empleen apropiadamente los principios de encapsulamiento.

Ejercicio

Implemente la clase Sedan de acuerdo a los siguientes requerimientos :
El sedan debe estar compuesto por un Motor que se puede encender, acelerar ó parar.
Debe tener cuatro Llantas, las cuales podrán Inflarse con una presión dada.
Tendrá dos Puertas, que se podrán abrir o cerrar. A su vez cada puerta debe tener una Ventana, que podrá subirse o bajarse.
Cree una subclase llamada Compacto que se derive de Sedan.
Pruebe los siguientes mensajes con un objeto tipo Compacto :

```
vocho.puertaIzquierda.ventana.subir() ;
vocho.motor.encender() ;
vocho.llanta[0].inflar(28) ;
```

4.5.7.3. *La palabra reservada **final**.*

Esta palabra reservada tiene diferentes significados dependiendo del contexto en el que es usada, pero en términos generales significa que algo no puede ser modificado.

Cuando se utiliza con primitivas, la palabra reservada **final** indica que el valor es constante, pero cuando se trata de objetos entonces la referencia al objeto es la que no cambia, pero el estado del objeto si puede modificarse. La referencia al objeto debe inicializarse en el punto de declaración.

Considere el siguiente ejemplo :

```
// : FinalData.java
import java.util.* ;

class Valor {
        public int i =1 ;
}

public class FinalData {
        public static final int const1 = 11 ;
        final int i1= (int)(Math.random()*20) ;
        static final int i2=(int)(Math.random()*20) ;

        Valor v1 = new Valor() ;
        final Valor v2 = new Valor() ;

        public static void main(String args[ ]) {
                FinalData f1 = new FinalData() ;
                f1.const1 ++ ;
                f1.v2.i ++ ;
                f1.v1 = new Valor() ;
                f1.v2 = new Valor() ;
                FinalData f2 = new FinalData() ;
                System.out.println("f1.i1 = "+f1.i1+", f1.i2 ="+f1.i2) ;
                System.out.println("f2.i1 = "+f2.i1+", f2.i2 ="+f2.i2) ;
        }
}
```

Los métodos tipo **final** se utilizan, por un lado, para prevenir que una clase derivada cambie la implementación del método, pero mas predominantemente por eficiencia. Al declarar un método como final, el compilador convierte las llamadas en línea. Esto significa que el compilador, al ver esta palabra reservada, no sigue el mecanismo normal de llamada a un método, y en cambio lo substituye con una copia del código del método.

Cuando se especifica una clase entera como **final** entonces no puede heredarse desde esta clase. Esto implica que todos los métodos de una clase final son implícitamente tipo fina, ya que no hay opción de sobreescribirlos.

4.5.7.4. *Inicialización con herencia.*

Consideremos ahora el proceso completo de inicialización tomando en cuenta la herencia.

```
//: Hormiga.java

import java.util.*;

class Insecto {
    int i = 9;
    int j ;
    public Insecto() {
        prt("i = "+i+", j = "+j);
        j = 77;
    }
    static int x = prt("static Insecto.x inicializado");
    static int prt(String s) {
        System.out.println(s);
        return 45;
    }
}

public class Hormiga extends Insecto {
    int k = prt("Hormiga.k inicializado");
    public Hormiga() {
            prt("k = "+k);
            prt("j = "+j);
            prt("x = "+x);
            prt("super.x = "+super.x);
    }
    static int x = prt("static Hormiga.x inicializado");
    static int prt(String s) {
        System.out.println(s);
        return 63;
    }
    public static void main(String args[]) {
        prt("Constructor de Hormiga:");
        new Hormiga();
    }
}
```

La primera cosa que sucede cuando se corre Hormiga es que se localiza la clase. En el proceso de carga, el interprete identifica que tiene una clase base, la cual carga. Esto ocurre se cree o no un objeto de la clase Hormiga. Si la clase base estuviera basada en otra, entonces la segunda clase se cargaría, y así sucesivamente. Posteriormente, se realiza la inicialización de los miembros tipo **static** en la clase raíz (en este caso Insecto), luego en la siguiente clase derivada, y así sucesivamente.

En este punto todas las clases necesarias han sido cargadas de forma que se pueden comenzar a crear objetos. Al hacerlo, todas las primitivas del objeto son inicializadas a su

valor por default y los objetos se hacen apuntar a **null**. Entonces el constructor de clase base se llama. La construcción de la clase base sigue el mismo proceso Después de que se completa el constructor de la clase base, los campos son inicializados, en el orden en que aparecen. Finalmente el resto del cuerpo del constructor se ejecuta.

4.5.8. Polimorfismo.

Al usar herencia uno puede tratar objetos de una clase derivada como si fueran de la clase base. Esto permite que muchos tipos (derivados de la clase base) se traten como si fueran de un solo tipo, y exista código que trabaje para todos ellos igualmente. Veamos el siguiente ejemplo.

```
//: Musica.java

import java.util.*;

class Instrumento {
    public void play() {
        System.out.println("Instrumento.play()");
    }
    static void afinar(Instrumento i) {
        System.out.println("Inicia afinacion:");
        i.play();
    }
}

class Cuerdas extends Instrumento {
    public void play() {
        System.out.println("Cuerdas.play()");
    }
}

public class Musica {
    public static void main(String args[]) {
        Cuerdas c = new Cuerdas();
        Instrumento.afinar(c);
    }
}
```

Es interestante ver en este ejemplo como se utiliza el método afinar(), que acepta como argumento un objeto tipo Instrumento. Desde main() se le llama con una referencia a un objeto tipo Cuerdas. A primera vista pudiera parecer extraño que un método acepte un tipo diferente al que tiene definido en su declaración de argumentos, sin embargo, debemos damos cuentas que no puede haber ninguna llamada a un método de Instrumento en afinar() que no este en Cuerdas también. Dentro de afinar() el código funciona para cualquier objeto tipo Instrumento, así como para objetos derivados de el. Al acto de utilizar el objeto tipo Cuerdas como si fuera un objeto Instrumento se le llama *upcasting*.

Uno podría preguntarse si no sería mas simple crear un nuevo método afinar() que acepte un objeto tipo Cuerdas. Si se tomara esa decisión, entonces habría que crear un nuevo método afinar() cada vez que se creara una nueva definición de clase tipo Instrumento.

Ejercicio

Defina una clase Percusiones como un tipo de Instrumento.
Afínelo.

Para lograr el comportamiento polimorfico Java utiliza lo que se conoce como "dynamic binding", que implica que las llamadas a los métodos se conectan con el cuerpo del método en tiempo de ejecución. Todos los métodos en Java utilizan dynamic binding a menos que se hayan declarado como **final**.

Ejercicio

Implemente la jerarquía de clases Figura <---Circulo
<---Triangulo
<---Cuadrado

con los métodos dibujar() y borrar().
Cree un una clase Polimorfismo, y desde el main() cree un objeto utilizando :

```
Figura c = new Circulo() ;
```

¿Que pasa si llama al método c.dibujar() ?

Veamos ahora como podríamos sacar provecho del polimorfismo. En el ejercicio anterior insertemos el siguiente código:

```
class Figuras {
    public static Figura randFigura() {
        switch((int)(Math.random()*3)) {
            default:
            case 0: return new Circulo();
            case 1: return new Triangulo();
            case 2: return new Cuadrado();
        }
    }
}

public class Polimorfismo {
    public static void main(String args[]) {
        Figura f[ ] = new Figura[10];
        for(int i=0; i< f.length; i++) {
            f[i] = Figuras.randFigura();
            f[i].dibujar();
        }
    }
}
```

La clase Figuras contiene un método estático que regresa un apuntador a un objeto elegido de manera aleatoria. Note que en cada sentencia **return** de este método esta

ocurriendo upcasting. Dentro de main() se sabe que se tiene un array de objetos tipo Figura, pero no se sabe nada mas específico. Este ejemplo ilustra claramente el concepto de dynamic binding.

Ejercicio

En el ejemplo de Musica defina dentro de la clase Musica un método afinaTodos() que reciba como argumento un array de Instrumentos y los afine. Desde el main() cree un array con dos elementos, uno de Cuerdas y otro de Percusiones y haga una llamada al método afinaTodos().

4.5.8.1. Clases y métodos abstractos

Se crea un clase abstracta cuando se desea manipular un conjunto de clases a través de una interfaz común. Todos los métodos de las clases derivadas serán llamados usando el mecanismo de dynamic binding. Es importante notar que si los argumentos de un método en una clase derivada no coinciden con los de la clase base, entonces no se sobreescribe el método sino se hace overloading.

El el caso de Musica.java, puede ser que la clase Instrumento únicamente se utilice para definir la interfaz, y por lo tanto no tenga sentido que el método play() haga nada. El mecanismo en Java para lograr esto es utilizando la palabra reservada **abstract**. Por ejemplo en el caso de la definición del método play() utilizaríamos :

```
abstract void play() ;  //No hay implementación !
```

Una clase que contiene un método abstracto se llama una clase abstracta. Si una clase contiene uno o más métodos abstractos, la clase misma debe ser definida como **abstract**.

Una característica de las clases abstractas es que no pueden crearse instancias de dicha clase. Si en un clase derivada de una clase abstracta no se implementan todos los métodos abstractos de la clase base, entonces la clase derivada debe indicarse como abstracta.

Ejercicio

Modifique la clase Instrumento para hacerla abstracta.

4.5.8.2. *Interfaces.*

La palabra reservada **interface** lleva el concepto de abstracción un paso mas adelante. Puede pensarse en una interfaz como una clase abstracta pura. Permite al programador establecer la forma de una clase : nombre de los métodos, lista de argumentos y tipos de retorno, pero nada mas. No hay cuerpos de métodos ni variables. Una interfaz dice : "esto es lo que todas las clases que implementen esta interfaz deben hacer". De forma que cualquier código que use una interfaz sabe que métodos pueden ser llamados.

Para crear un interfaz, se usa la palabra reservada interface en lugar de class. Para hacer que una clase se comporte de acuerdo a una interfaz se utiliza la palabra reservada **implements**. Lo que dice es "la interfaz en como se ve y aquí esta como funciona". Una vez que se implementa una interfaz, esta implementación se convierte en una clase ordinaria que puede extenderse de la manera vista.

Una interfaz tambien puede tener datos de tipo primitivo, pero estos son implícitamente de tipo **static final**, así que tienen el efecto de ser constantes definidas en tiempo de compilación.

Ejercicio

Modifique la clase Instrumento para hacerla una interfaz.
Modifique el resto del programa Musica.java para reflejar este cambio.

Como puede notar no hay diferencia entre hacer upcasting a una clase regular, un clase abstracta o una interfaz. El comportamiento es el mismo.

4.5.8.3. *Herencia múltiple en Java*

Una interfaz no es únicamente una clase abstracta pura. Puesto que una interfaz no tiene implementación, es decir, no hay ningún almacenamiento asociado a ella, no existe impedimento para combinar múltiples interfaces. Esto es muy útil, ya que hay ocasiones cuando se necesita decir "un x es del tipo a y del tipo b y del tipo c". Aunque no es estrictamente necesario a tener una clase concreta, no puede tenerse mas que una. Se pueden tener tantas interfaces como sea necesario y cada una se convierte una clase independiente con la que podemos hacer upcasting.

Considere el siguiente ejemplo :

```
//: Aventura.java

import java.util.*;

interface Peleador {
    public void pelea();
}

interface Nadador {
    public void nada();
}

interface Volador {
    public void vuela();
}

class PersonajeDeAccion {
    public void pelea() {
        System.out.print("Personaje.pelea()");
```

```
    }
}

class Heroe extends PersonajeDeAccion
    implements Peleador, Nadador, Volador {
    public void nada() {
        System.out.println("Heroe.nada()");
    }
    public void vuela() {
        System.out.println("Heroe.vuela()");
    }
}

public class Aventura {
    static void p(Peleador x) {x.pelea();}
    static void n(Nadador x) {x.nada();}
    static void v(Volador x) {x.vuela();}
    static void w(PersonajeDeAccion x) {x.pelea();}

    public static void main(String args[]) {
        Heroe z = new Heroe();
        p(z);
        n(z);
        v(z);
        w(z);
    }
}
```

Note que a pesar de que la clase Heroe no da una implementación para pelea(), la definición viene con la clase PersonajeDeAccion, y por lo tanto es posible instanciar objetos tipo Heroe. Como puede verse con el objeto tipo Heroe puede hacerse upcasting con cualquiera de sus interfaces.

Todas la declaraciones de métodos en una interfaz son por definición públicas, aún cuando no se use explícitamente la palabra reservada **public**. Por lo tanto la implementación de dichos métodos debe ser indicada como pública.

Como una regla de dedo para decidir cuando utilizar clases abstractas y cuando interfaces podemos seguir el siguiente procedimiento : cuando se sepa que algo va a ser una clase, la primera opción es definirlo como una interfaz, y sólo si se necesita implementar algún método o definir campos entonces cambiar a una clase abstracta.

Se pueden agregar nuevos métodos a una interfaz usando herencia, y además se pueden combinar distintas interfaces (usando la palabra reservada **extends**) para crear una nueva. En ambos casos se obtiene una nueva interfaz. Normalmente sólo se puede usar **extends** con una clase, pero dado que una interfaz puede crearse a partir de muchas otras, **extends** se puede referir en este caso a varias interfaces para construir una nueva. Simplemente se separan los nombres de las interfaces utilizando comas.