Capítulo 1 - Java

¿Qué es Java?

Hacia 1990, James Gosling, quien trabaja para Sun Microsystems, fue el encargado de crear programas para controlar aparatos electrónicos domésticos. Originalmente Gosling y su equipo empezaron el diseño de su software usando C++, debido a su orientación a objetos. Sin embargo, se dieron cuenta que C++ no satisfacía los proyectos que se tenían en mente; encontraron dificultades con aspectos complicados de C++ como la herencia múltiple de clases, errores de programación (bugs) relacionados con huecos de memoria. De esta manera Gosling decidió que tenía que empezar por escribir un lenguaje simplificado que le evitara todos los problemas que se encontró con C++.

Aunque a Gosling no le importó la complejidad de lenguajes como C++, tomo la sintaxis básica y la orientación a objetos del lenguaje. Cuando terminó el nuevo lenguaje lo llamó Oak (se cuenta que el nombre le vino a Gosling mientras veía un roble por la ventana de su oficina.)

Oak se uso por primera vez en un proyecto llamado Proyecto Green, donde el equipo de desarrollo intentó diseñar un sistema de control para usar dentro del hogar. Este sistema de control permitiría al usuario manipular distintos dispositivos, como televisiones, video caseteras, luces caseras y teléfonos, todo desde una computadora de mano llamada *7 (Star Seven). El sistema *7 incluía una pantalla sensible para que el dueño seleccionara y controlara estos dispositivos.

La pantalla del *7 tenía diversas figuras, entre las que se encontraba Duke (la actual mascota de Java). Duke ha sido incluido en muchos ejemplos de applets en la página de Sun Microsystems.

El siguiente paso para Oak fue el proyecto Video En Demanda (VOD), en el que el lenguaje era usado como la base para el software que controlaría un sistema de televisión interactivo. Aunque ni *7 ni el proyecto VOD concluyeron en productos actuales, le dieron a Oak una oportunidad de crecer y madurar. Con el tiempo Sun Microsystems descubrió que el nombre Oak ya había sido usado y le cambió el nombre por Java, y vio nacer un lenguaje poderoso y sencillo.

Java es un lenguaje independiente de la plataforma, lo que significa que los programas desarrollados en Java correrán en cualquier sistema sin cambios. Esta independencia de plataforma se logró usando un formato especial para los programas compilados en Java. Este formato de archivo, llamado "byte-code" puede ser leído y ejecutado por cualquier computadora que tenga un intérprete de Java. Este intérprete de Java, por supuesto, debe ser escrito especialmente para el sistema en el que correrá.

En 1993, después de que Internet se transformó de un ambiente basado en texto a un ambiente gráfico, el equipo de Java se dio cuenta de que el lenguaje sería perfecto para la programación en el Web. Así nació la idea de los applets, que son pequeños programas que pueden ser incluidos en páginas de Web, y también surgió la idea de escribir un navegador de Web que demostraría el poder del lenguaje, este navegador es el HotJava.

Finalmente, hacia mayo de 1995, Sun Microsystems anunció oficialmente a Java. El nuevo lenguaje fue aceptado como una poderosa herramienta para el desarrollo de aplicaciones para Internet. Netscape Communications, el creador del navegador Netscape Navigator, dio soporte a Java desde su versión 2.0. Otros desarrolladores de software también incluyeron soporte para Java, incluyendo al Internet Explorer 3 de Microsoft. Actualmente, Java puede correr en máquinas con procesadores SPARC, Intel, Digital.

De acuerdo con Sun Microsystems, Java es "simple, orientado a objetos, tipificado estáticamente, compilado, independiente de arquitectura, multi-procesos, con recolector de basura, robusto, seguro y ampliable."

Es simple porque los desarrolladores en Java deliberadamente dejan muchas de las características innecesarias de otros lenguajes de programación de alto nivel. Por ejemplo, Java no soporta aritmética de apuntadores, cast de tipos implícito, estructuras o uniones, sobrecarga de operadores, plantillas, archivos de cabecera o múltiple herencia.

Es orientado a objetos, porque como C++, Java usa clases para organizar el código en módulos. En tiempo de ejecución, un programa crea objetos a partir de las clases. Las clases en Java pueden heredar de otras clases, pero la múltiple herencia, donde una clase hereda métodos y datos de varias clases, no está permitida.

Es tipificado estáticamente porque todos los objetos usados en un programa deben ser declarados antes de que puedan ser usados. Esto permite al compilador de Java localizar y reportar conflictos con los tipos de datos.

Es compilado porque antes de que se pueda correr un programa, primero tiene que ser compilado por el compilador de Java. El resultado de la compilación es el archivo "byte-code", que, similar a un archivo con código máquina, puede ser ejecutado bajo cualquier sistema operativo que tenga un intérprete de Java. Este intérprete lee el archivo byte-code y traduce los comandos en comandos de lenguaje máquina que pueden ser ejecutados directamente por la computadora.

Es multiprocesos porque los programas de Java pueden contener múltiples procesos en ejecución, lo que permite a los programas manejar varias tareas simultáneamente. Por ejemplo, un programa multiprocesos puede definir una imagen (render) en un proceso mientras continua aceptando entrada del teclado en el proceso principal. Todas las aplicaciones tienen al menos un proceso (llamado thread) que representa la ejecución del programa.

Tiene recolector de basura, ya que los programas de Java no se encargan de liberar de memoria los objetos, esto es una tarea del administrador de memoria y el recolector de basura.

Es robusto porque el intérprete de Java revisa todos los accesos al sistema dentro de un programa, por esto, los programas desarrollados en Java no pueden tirar el sistema. Esto es, cuando un error serio es encontrado, los programas en Java crean una excepción. Esta excepción puede ser capturada y manejada por el programa sin el riesgo de bloquear el sistema.

Es seguro porque el compilador no sólo verifica todos los accesos a memoria, sino que también se asegura que no entren virus en un applet en ejecución. Ya que los apuntadores no son soportados por el lenguaje, los programas no pueden acceder a áreas del sistema a las que no tienen autorización.

Es ampliable porque los programas en Java soportan métodos nativos, que son funciones escritas en otros lenguajes, generalmente C++. Este soporte a métodos nativos permite a los programadores escribir funciones que pueden ser ejecutadas más rápido que las funciones equivalentes escritas en Java. Los métodos nativos son ligados a los programas en forma dinámica, es decir, son asociados con los programas en tiempo de ejecución.

Ventajas de Java.
Es seguro.
Se aprende con facilidad.
Es orientado a objetos.
No bloquea el sistema.
Aplicaciones para comunicación en red.
No tiene aritmética de apuntadores.
Es independiente de la plataforma.
Soportado por Microsoft.

Desventajas de Java.
Es 10 a 20 veces más lento en ejecución que C++.
Soportado por Microsoft.

Introducción a Java

Java tiene las siguientes características:
- La Máquina Virtual de Java (JVM)
- Recolección de basura
- Seguridad en el código

La especificación de la Máquina Virtual de Java define a ésta como:
"Una máquina imaginaria que es implantada por la emulación de software en una máquina real. El código para la JVM es almacenado en archivos .class, cada uno contiene código para al menos una clase pública". Esta especificación permite a los programas Java ser independientes de la plataforma porque la compilación es hecha por una máquina genérica. Al intérprete de Java de cada plataforma de hardware le corresponde asegurar la ejecución del código compilado para la JVM.

Muchos lenguajes de programación permiten el alojamiento dinámico de memoria en tiempo de ejecución. Este proceso varía en la sintaxis de los lenguajes, pero siempre hay un valor de retorno de un apuntador a la dirección de inicio del bloque de memoria. Una vez que la memoria ocupada ya no se necesita, el programa o el ambiente de ejecución debería liberar la memoria para prevenir que el programa corra sin memoria disponible.

En C y C++ (y otros lenguajes), el programador es responsable de liberar la memoria. Esto puede ser tedioso, porque no se sabe con anticipación cuando se va a liberar memoria. Los programas que no liberan memoria pueden bloquear el sistema cuando no queda memoria disponible. Java quita esa responsabilidad de liberar memoria explícitamente integrando un proceso a nivel de sistema que sigue cada alojamiento de memoria y mantiene una cuenta del número de referencias a cada apuntador a memoria. Durante los intervalos de tiempo de ocio en la JVM, el proceso de recolección de basura revisa si hay apuntadores a memoria donde el número de referencias es igual a cero. Si hay algunos, el bloque de memoria marcado por el recolector es liberado.

Una vez compilados los programas en Java, en el momento de ejecución se lleva a cabo una tarea de carga, revisión y ejecución. La carga consta de cargar en memoria el archivo byte-code, puede cargarse desde la máquina local o remotamente a través de la red. La revisión consta de verificar que no haya: violaciones de acceso, operaciones que conduzcan a "overflow" o "underflow", tipos de parámetros incorrectos, conversiones de datos incorrectas, acceso a objetos sin inicializar, entre otras funciones. En el proceso de ejecución ya se corren las instrucciones del programa.

Comentarios

Los comentarios en Java se pueden escribir en tres formas:

```
// comentario de una línea
/* comentario de una o más líneas */
/** comentario para documentación */
```

Los comentarios de documentación se colocan justo antes de la variable o función. Estos sirven para el programa javadoc, el cual genera archivos html, y sirven como una descripción del tópico declarado.

Identificadores

En Java, un identificador empieza con una letra, el carácter de subraya o el signo $. Los demás caracteres pueden contener dígitos. Todos los identificadores son sensibles a mayúsculas / minúsculas.
Ejemplos de identificadores validos:

variable
nombreUsuario
Nombre_Usuario
_numero
$cadena

Los últimos tres ejemplos son muy poco usados en la generalidad de los programas. Los identificadores pueden contener palabras reservadas, pero no pueden ser palabras reservadas; por ejemplo, es valido integer, pero no int.

Palabras reservadas

abstract	do	implements	private	throw
boolean	double	import	protected	throws
break	else	instanceof	public	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	super	volatile
class	float	new	switch	while
continue	for	null	synchronized	
default	if	package	this	

Nota.- En Java, true, false y null se escriben en minúsculas, al contrario que en C++.

No existe un operador sizeof; el tamaño y representación de todos los tipos es fija y no es dependiente de la implantación.

Las palabras goto y const no se usan en Java.

Tipos de datos

Java define ocho tipos de datos primitivos y uno especial. Se pueden agrupar en: lógicos, textuales, integrales y de punto flotante.

El tipo de dato boolean sólo tiene dos valores: false y true. En C/C++ se permite que valores numéricos sean interpretados como valores lógicos, pero no es el caso de Java; sólo se permiten valores lógicos.

Los tipos de datos textuales son: char y String. Los caracteres se representan por el tipo de dato char, usando un número de 16 bits sin signo con un rango de cero a 2^{16-1}. Los caracteres se encierran entre apóstrofes.

'a'	Letra a
'\t'	Un tabulador
'\u????'	Un carácter especifico, ???? es reemplazado con cuatro dígitos hexadecimales.

El tipo String, que no es primitivo, es usado para representar secuencias de caracteres. Una cadena de caracteres se encierra entre comillas.

"Esto es un mensaje"

Los tipos integrales son: byte, short, int y long. Todos son números con signo. Los números se pueden representar en forma decimal, octal o hexadecimal.

2	Valor decimal es dos
077	El cero que inicia indica un valor octal
0xBC27	0x indica un valor hexadecimal

Para especificar un valor long se debe poner L al final del número. Se puede usar l o L, pero en minúscula se puede confundir con el número 1 en algunos casos.

2L	Valor decimal dos, como un valor largo
077L	El cero que inicia indica un valor octal
0xBC27L	0x indica un valor hexadecimal

A continuación se presenta una tabla con los cuatro tipos de datos. La representación del rango es definida por la especificación del lenguaje como un complemento a dos y es independiente de la plataforma.

Tamaño	Tipo	Rango
8 bits	byte	-2^{7} ... 2^{7-1}
16 bits	short	-2^{15} ... 2^{15-1}
32 bits	int	-2^{31} ... 2^{31-1}
64 bits	long	-2^{63} ... 2^{63-1}

Los tipos de dato para número de punto flotante son: float y double. Un valor en punto flotante puede incluir el punto decimal, una parte exponente (letra E), o es seguido por la letra F (float) o la letra D (double).

3.14	Valor de punto flotante
6.02E23	Valor de punto flotante muy grande
2.718F	Valor sencillo de tipo float
123.4E+306D	Valor de punto flotante muy grande de tipo double

Tamaño	Tipo
32 bits	float
64 bits	double

En Java, todos los valores de punto flotante son double, a menos que se indique explícitamente que sean float. Por lo tanto, en los ejemplos anteriores:

2.718F	debe llevar la letra F para mantenerlo como float
123.4E+306D	la D es redundante

Modificadores

Dentro de las palabras reservadas, Java utiliza las siguientes para modificar el acceso a una variable, clase o función y se colocan al inicio de la declaración: public, protected, default, private.

El modificador public da acceso a cualquier objeto externo.

```
public int numero; // cualquier objeto puede accesar a esta variable
```

El modificador protected da acceso a objetos que son parte del mismo paquete, y las subclases. (Más adelante se explica el concepto de paquete)

El modificador default da acceso a objetos que son parte del mismo paquete. Sin embargo, en los programas no se especifica el modificador porque no hay una palabra para ello.

```
int numero; // acceso default
```

El modificador private da acceso únicamente a la clase que lo contiene.

```
private int numero; // únicamente lo puede accesar la clase
```

Convenciones en la programación

Clases.- Los nombres de las clases deberían ser sustantivos, utilizando mayúsculas para la primera letra y minúsculas para las restantes, y se pueden mezclar varios sustantivos.

```
class CuentaBancaria
```

Interfaces.- Los nombres de las interfaces deberían tener la primera letra mayúscula, como en los nombres de clase.

```
interface Cuenta
```

Métodos.- Los nombres de los métodos deberían ser verbos, todo el verbo en minúscula. Se pueden agregar sustantivos con la primera letra en mayúscula. Evitar el uso de subrayas.

```
void revisarCuenta()
```

Constantes.- Las constantes de tipos de datos primitivos deberían escribirse completamente en mayúsculas y separadas las palabras por subrayas. Las constantes de objeto pueden combinar mayúsculas y minúsculas

```
final int MAX_CREDITO
```

Variables.- Todas las variables deberían ser en minúsculas, y si se agregan palabras se separarán con una letra mayúscula. Evitar el uso del signo $.

```
primerUsuario
```

Las variables deben tener significado e indicar su uso. Las variables de una letra deberían evitarse, excepto las que suelen usarse en ciclos (x, y, i, j) para controlarlo.

Otras convenciones de la programación incluyen el uso de llaves ({}) alrededor de un bloque de instrucciones, incluso cuando se trate de una sola instrucción, ya que esto ayuda en el mantenimiento del programa.

```
if(condición)
{
 bloque
}
```

El espaciado ayuda en la comprensión del programa. Se sugiere escribir una instrucción por línea y usar indentación de uno o dos espacios.

Los comentarios también ayudan en la comprensión y mantenimiento del programa al dar una descripción clara de lo que hace cada función y el uso de las variables.

Ejemplo:

```
// primer programa en Java
public class HelloWorld
{
 public static void main(String argv[])
 {
  System.out.println("Hello world!");
 }
}
```

En detalle:

```
// primer programa en Java
```

La primera línea es un comentario.

```
public class HelloWorld
{
```

Las siguientes dos líneas son la declaración de la clase, que al momento de ser compilado el programa, generará un archivo .class. Es importante que el nombre de la clase sea el mismo que el nombre del archivo: si la clase se va a llamar HelloWorld, el archivo se debe llamar HelloWorld.java.

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

En las siguientes dos líneas se declara el inicio del programa. Para que el intérprete de Java pueda ejecutar el programa debe tener la misma sintaxis (excepto para el nombre del parámetro de main). Se declara public para que lo pueda accesar el intérprete de Java. Se declara static porque no se ha creado algún objeto y no se crea una instancia. Se declara void porque no se regresa valor alguno. En este ejemplo no se va a esperar parámetros de la línea de comandos. En argv[] se guardan los parámetros y la primera posición contiene el primer parámetro, no el nombre del programa:
argv[0] parametro1
argv[1] parametro2

```
System.out.println("Hello world!");
```

La siguiente línea muestra el uso de una clase y un método que imprime en la salida estándar (la pantalla) un mensaje.

```
 }
}
```

Finalmente se termina el bloque del método main y la declaración de la clase. Una vez que se tiene el código fuente en el archivo HelloWorld.java se usa el compilador de Java de la siguiente manera:

```
javac HelloWorld.java
```

Si el compilador no regresa mensajes de error, se habrá creado un nuevo archivo HelloWorld.class en el mismo directorio que el código fuente.

Después de la compilación, se puede ejecutar el programa y ver el resultado usando el intérprete de Java:

```
java HelloWorld
```

Modifica el programa anterior para dejar errores y ver lo que presenta el compilador.

Laboratorio 1

Modificar el programa de "HelloWorld.java" para que reciba un nombre por medio de un parámetro en la línea de comandos y que lo imprima en forma de saludo. Por ejemplo:

C:\>java Hello JoseLuis

Esto deberá desplegar una salida:

Gusto en conocerte JoseLuis

*Nota: Recuerde que los parámetros que son recibidos en la línea de comandos se guardan en el arreglo de tipo String (comúnmente llamada argv[ ]) del método main. Trate de utilizar la propiedad "length" del ya mencionado arreglo para controlar el número de parámetros recibidos.

Ver Cap1\Hello.java

Capítulo 2 - El lenguaje

Inicialización de variables

Java no permite que una variable tenga un valor indefinido. Cuando un objeto es creado, sus variables son inicializadas con los siguientes valores:

byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000' (NULO)
boolean	false
todas las referencias	null

Si algún objeto hace referencia a algo con valor de null, creará una excepción (un error que es manejable). Para evitar que las variables tengan valores indeseables, se debe asignárseles algún valor útil. El compilador estudia el código para determinar que cada variable ha sido inicializada antes de su primer uso. Si el compilador no puede determinar esto, entonces ocurre un error en tiempo de compilación.

```
public void calcula()
{
 int x = (int)(Math.random() * 100);
 int y;
 int z;
 if(x > 50)
 {
  y = 9;
 }
 z = y + x;
 // el posible uso antes de la inicialización de y creara un error de compilación
}
```

Expresiones lógicas

Los operadores relacionales y lógicos regresan un valor boolean. En Java no existe conversión automática de int a boolean, como en C++.

```
int i = 1;
if(i) // error en tiempo de compilación
if(i != 0) // correcto
```

Operadores y su Precedencia

Los operadores en Java son muy similares en estilo y función a aquellos en C y C++. La siguiente tabla enlista los operadores por orden de precedencia:

Separador	. [] () ; ,
D a I	++ -- + - ~ !
I a D	* / %
I a D	+ -
I a D	<< >>
I a D	< > <= >= instanceof
I a D	== !=
I a D	&
I a D	^
I a D	|
I a D	&&
I a D	||
D a I	?:
D a I	= *= /= %= += -= <<= >>= &= ^= |=

El operador + se puede utilizar para concatenar cadenas de caracteres, produciendo una nueva:

```
String saludo = "Sr. ";
String nombre = "Luis " + "Torres";
String persona = saludo + nombre;
```

Los operadores && (and) y || (or) realizan una evaluación corta en expresiones lógicas. Por ejemplo:

```
String unset = null;
if((unset != null) && (unset.length() > 5))
{
 // hacer algo con unset
}
```

La expresión que forma a if() es legal y completamente segura. Esto es porque la primera subexpresión es falsa, y es suficiente para probar que toda la expresión es falsa. El operador && omite la evaluación de la segunda subexpresión y una excepción de null pointer es evitada. De forma similar, si se usa el operador || y la primera subexpresión es verdadera, la segunda subexpresión no es evaluada porque toda la expresión es verdadera.

Cast

Cuando la asignación de valores no es compatible por los tipos de datos, se usa un cast para persuadir al compilador de reconocer tal asignación. Esto se puede hacer para asignar un long a un int, por ejemplo.

```
long bigValue = 99L;
int smallValue = (int)(bigValue);
```

No es necesario el segundo grupo de paréntesis, los que encierran a bigValue, pero es muy recomendable dejarlos.

Aunque short y char ocupan 16 bits, se debe hacer un cast explícito, debido al rango que tienen asignado.

Flujo de programa

Sentencia if/else.

Permite elegir una de dos opciones. La sintaxis básica de la sentencia if/else es:

```
if(condición)
{
 instrucción_o_bloque
}
else
{
 instrucción_o_bloque
}
```

Ejemplo:

```
int aleatorio = (int)(Math.random() * 100);
if(aleatorio < 50)
{
 System.out.println("menor a 50");
}
else
{
 System.out.println("mayor o igual a 50");
}
```

Sentencia switch.

Permite seleccionar una de varias opciones. La sintaxis para switch es la siguiente:

```
switch(expresión_a_evaluar)
{
 case valor1:
  instrucciones;
  break;
 case valor2:
  instrucciones;
  break;
 case valor3:
  instrucciones;
  break;
 default:
  instrucciones;
  break;
}
```

El valor de expresion_a_evaluar debe ser compatible con el tipo int, como short, byte y char. No se permite evaluar long o valores de punto flotante.

Ejemplo:

```
switch(colorNum)
{
```

```
 case 0:
  setBackground(Color.red);
  break;
 case 1:
  setBackground(Color.green);
  break;
 case 2:
  setBackground(Color.blue);
  break;
 default:
  setBackground(Color.black);
  break;
}
```

La sentencia for.

Permite realizar una serie de instrucciones mientras se cumple una condición. La sintaxis básica para for es:

for(inicialización;condición;alteración)
{
instrucciones;
}

Ejemplo:

```
int x;
for(x = 0;x < 10;x++)
{
 System.out.println("dentro de for");
}
System.out.println("fin de for");
```

El tercer parámetro puede ser tanto de incremento como de decremento, y no únicamente de uno en uno. Java permite el uso de comas dentro de la declaración de for, como en C, por lo que lo siguiente es legal:

```
for(i = 0, j = 0;j < 10;i++,j++)
```

En el ejemplo anterior, la variable x es "visible" en el método en el que es declarada. Se puede usar una variable que sea visible únicamente para el ciclo for:

```
for(int x=0;x<10;x++)
{
 ...
}
// una vez terminado el ciclo, x ya no puede ser accesada
```

La sentencia while.

Permite realizar una serie de instrucciones mientras se cumple una condición. La sintaxis básica de while es:

while(condición)
{
instrucciones;
}

Ejemplo:

```
int i = 0;
while(i<15)
{
 System.out.println("dentro de while");
 i+=2;
}
```

La sentencia do/while.

Permite realizar una serie de instrucciones hasta que deje de cumplirse una condición. La sintaxis básica de la sentencia es:

```
do
{
 instrucciones;
}while(condición);
```

Ejemplo:

```
int i = 0;
do
{
 System.out.println("dentro de while");
 i++;
}while(i<10);
```

Paquetes

Java provee el mecanismo de paquetes (package) como una forma de organizar las clases. Se puede indicar que las clases en el código fuente van a pertenecer a un paquete empleando la palabra package.

```
package empresa.sistemas;

public class Empleado
{
 ...
}

$ javac -d <ruta> Archivo.java
```

La declaración de paquete, si la hay, debe estar al inicio del código fuente, puede estar precedida únicamente de comentarios. Solo se permite una declaración package por archivo fuente. Los nombres de los paquetes son jerárquicos, separados por puntos. Por lo general, los elementos de los paquetes son escritos enteramente en minúsculas. Una vez compilado el archivo, puede ser usado por otro mediante la sentencia import, que indica donde se encuentran los paquetes. Import debe preceder a todas las declaraciones de clases.

```
import empresa.sistemas.*;

public class JefeArea extends Empleado
{
 String departamento;
 Empleado subordinados[];
 ...
}
```

Laboratorio 2

1. Realizar un programa que demuestre la diferencia entre declarar la variable de control de un ciclo "for" dentro o fuera de este.
   Ver Cap2\Ciclo1.java
2. Realizar un programa que imprima un contador en un ciclo (de preferencia while) que sea controlado por medio de una bandera (tipo de dato boolean).
   Ver Cap2\Ciclo2.java
3. Realizar un programa que calcule el factorial de un numero de manera recursiva.
   *Nota: Recuerde que la recursividad se lleva a cabo cuando una función se llama a sí misma "n" veces.
   Ver Cap2\Factorial.java

Capítulo 3 - Arreglos

Arreglos

Se pueden declarar arreglos de cualquier tipo de dato:

```
char letras[];
Point punto[];
```

En Java, un arreglo es un objeto, aun cuando el arreglo es de tipos de datos primitivos, y como con los demás objetos, la declaración no crea el objeto en sí. De modo que estas declaraciones no crean los arreglos, solo hacen referencia a variables que pueden ser usadas para accesar al arreglo. También se pueden crear arreglos con la siguiente sintaxis:

```
char [] letras;
```

Ambas declaraciones son válidas, pero la segunda tiene la ventaja de que, si se declaran más arreglos, sólo basta con poner el nombre de cada arreglo:

```
char letras[], numeros[];
```

por:

```
char [] letras, numeros;
```

Creación e inicialización

Para crear los arreglos se usa la palabra new:

```
letras = new char[20];
punto = new Point[100];
```

La primera línea crea un arreglo de 20 valores char. La segunda línea crea un arreglo de 100 variables de tipo Point, que es un objeto. Sin embargo, no crea los 100 objetos Point. Se tiene que crear separadamente cada objeto:

```
punto[0] = new Point();
punto[1] = new Point();
punto[2] = new Point();
...
```

Cuando se crea un arreglo, cada elemento es inicializado. En el caso del arreglo letras de tipo char, cada valor es inicializado al carácter nulo (\u0000). En el caso del arreglo punto, cada valor fue inicializado a null, indicando que no hay referencia al objeto Point. Después de la asignación punto[0] = new Point(), el primer elemento del arreglo se refiere a un objeto Point.

Java permite una manera fácil de crear arreglos con valores iniciales:

```
String nombres[] = {"Juan","Pedro","Luis"};
```

La línea anterior es similar a:

```
String nombres[];
nombres = new String[3];
nombres[0] = "Juan";
nombres[1] = "Pedro";
nombres[2] = "Luis";
```

Esta forma se puede aplicar tanto a tipos de datos primitivos como a objetos, por ejemplo:

```
Color paleta[] = {Color.red,Color.green,Color.blue};
```

Java no provee arreglos multidimensionales, en cambio, como se pueden crear arreglos de cualquier tipo, se crean arreglos de arreglos de arreglos...

```
int dosDim[][] = new int[4][];
dosDim[0] = new int[5];
dosDim[1] = new int[5];
```

Primero se declara un arreglo de arreglos de int. Luego se crea el arreglo con cuatro arreglos de int. Finalmente, se crea cada arreglo con cinco valores int.

Debido a que no hay arreglos multidimensionales, se pueden crear arreglos de arreglos no-rectangulares, por ejemplo:

```
dosDim[0] = new int[2];
dosDim[1] = new int[5];
dosDim[2] = new int[8];
```

Pero esta forma de arreglos no es muy común y es tediosa de programar. La forma rectangular de los arreglos es la más común, por lo que Java provee una forma fácil de crear arreglos bidimensionales:

```
int dosDim[][] = new int[4][5];
```

esto crea un arreglo de cuatro arreglos de cinco enteros cada uno.

Control del tamaño del arreglo

En Java todos los índices de los arreglos empiezan en cero. El número de elementos en un arreglo es almacenado como parte del objeto arreglo. El valor es usado para realizar evaluaciones de límite en todos los accesos en tiempo de ejecución. Si hay un acceso fuera del límite del arreglo, se crea una excepción.

El tamaño de un arreglo puede ser determinado en tiempo de ejecución usando la variable miembro length. Por ejemplo:

```
int lista[] = new int[10];
for(int x = 0;x < lista.length; i++)
{
 lista[x] = x;
}
```

El límite del ciclo se determina por la comparación con lista.length, en vez de comparar directamente contra el valor 10. Esto es más robusto cuando se trata de dar mantenimiento al programa.

Copiado de arreglos

Una vez creado, un arreglo no puede cambiar de tamaño. Sin embargo, se puede usar la misma variable de referencia para un arreglo nuevo:

```
int elementos[] = new int[6];
elementos = new int[10];
```

De esta manera, se pierden los seis valores del arreglo elementos, a menos que se hayan almacenado en otro lugar. Java provee un método en la clase System para copiar arreglos. Este método es arraycopy(). Por ejemplo:

```
// arreglo original
int elementos[] = {1,2,3,4,5,6};

// arreglo destino
int masnumeros[] = {10,9,8,7,6,5,4,3,2,1};

// copiar todos los valores de elementos en masnumeros
System.arraycopy(elements,0,masnumeros,0,elements.length);
```

En este punto, el arreglo más números tiene los siguientes valores: 1,2,3,4,5,6,4,3,2,1.

Ejemplo de arreglos:

```
import java.io.*;
import java.lang.*;

public class DemoArreglos
{
 static int facto(int num)
 {
  if (num == 0 || num == 1) return 1;
  return (num * facto(num-1));
 }

 public static void main(String args[])
 {
  int thisArray[], thatArray[]; // Variables arreglo de enteros
  int thirdArray[] = new int[10]; // Variable arreglo de 10 enteros

  for(int x = 0; x < thirdArray.length; x++)
  { // asignar valores a thirdArray
   thirdArray[x] = x + 1;
  }
  thisArray = thirdArray;
  // thisArray apunta a la misma dirección que thirdArray
  // aunque thisArray no había sido inicializado

  System.out.print("thisArray ");
  for(int x = 0; x < thisArray.length; x++)
  { // presenta lo que tiene thisArray
   System.out.print(thisArray[x]+" ");
  }
  System.out.println("");

  for(int x = 0; x < thisArray.length; x++)
  { // ahora thisArray tiene factoriales
   thisArray[x] = facto(thisArray[x]);
  }
  System.out.print("thisArray ");
  for(int x = 0; x < thisArray.length; x++)

  { // presenta el contenido de thisArray
   System.out.print(thisArray[x]+" ");
  }
  System.out.println("");

  thatArray = thisArray;
  // thatArray apunta a la misma dirección que thisArray
  // thatArray no tenía valores

  System.out.println("thatArray ");
  for(int x = 0; x < thatArray.length; x++)
  { // presenta el contenido de thatArray
   System.out.print(thatArray[x]+" ");
  }
  System.out.println("");

  for(int x = 0; x<5; x++)
  { // cambia algunos valores de thisArray
   thisArray[x] = 19;
  }
  System.out.println("thatArray ");
  for(int x = 0; x < thatArray.length; x++)
  { // presenta el contenido de thatArray
   System.out.print(thatArray[x]+" ");
  }
  System.out.println("");
```

```
  int fourthArray[] = new int[20];
  /* un cuarto arreglo de 20 enteros
     y thatArray que apuntaba a un arreglo de 10 enteros
     ahora apunta a un arreglo de 20 enteros */
  thatArray = fourthArray;
  System.out.println("thatArray ");
  for(int x = 0; x < thatArray.length; x++)
  { // presenta el contenido de thatArray
   System.out.print(thatArray[x]+" ");
  }
  System.out.println("");
  System.out.println("fourthArray ");
  for(int x = 0; x < fourthArray.length; x++)
  { // presenta el contenido de fourthArray
   System.out.print(fourthArray[x]+" ");
  }
  System.out.println("");

  System.arraycopy(thisArray,0,thatArray,0,thisArray.length);
  // se copian los elementos de thisArray en thatArray
  System.out.println("thatArray ");
  for(int x = 0; x < thatArray.length; x++)
  { // presenta el contenido de thatArray
   System.out.print(thatArray[x]+" ");
  }
  System.out.println("");

  for(int x = 0; x<5; x++)
  { // cambiar valores de thatArray
   thatArray[x] = 29;
  }
  System.out.println("thisArray ");
  for(int x = 0; x < thisArray.length; x++)
  { // y presentar los valores de thisArray
   System.out.print(thisArray[x]+" ");
  }
  System.out.println("");
  System.out.println("thatArray ");


  for(int x = 0; x < thatArray.length; x++)
  { // y de thatArray
   System.out.print(thatArray[x]+" ");
  }
  System.out.println("");
 }
}
```

Laboratorio 3

1. Realizar un programa que imprima los valores de dos arreglos inicializados de la siguiente forma:
   Uno contendrá 10 valores acomodados de manera ascendente.
   El otro deberá contener los mismos valores que el arreglo anterior pero de manera descendente.
   *Nota: El segundo arreglo deberá contener una referencia al primer arreglo y NO los valores directamente.
   Ver Cap3\Arreglos1.java
2. Realizar un programa que demuestre las capacidades de copiar arreglos usando igualación.
   Ver Cap3\Arreglos2.java
3. Realizar un programa que demuestre las capacidades de copiar arreglos usando el método "arraycopy" de la clase System.
   Ver Cap3\Arreglos3.java

Capítulo 4 - Objetos y Clases

Conceptos básicos

Una clase es la definición de las tareas que se van a realizar. Incluye las variables necesarias y los métodos, tanto públicos como privados. En lenguajes estructurados, los métodos equivalen a las funciones y procedimientos, como en C o Pascal.

Un *objeto* es la instancia de una clase. Con un objeto se pueden ejecutar las tareas definidas en la *clase*.

Una *subclase* es una clase que se deriva de otra, que hereda de otra sus variables y métodos.

Constructor es el método que da valores iniciales al objeto al momento de hacerse la instancia.

Herencia es la capacidad de recibir todos los métodos y variables de una o más clases para realizar ciertas tareas. Por lo general, las subclases agregan métodos y modifican algunos métodos para realizar tareas diferentes. Por ejemplo, la clase Object (de la que heredan todas las demás clases en Java) define un método llamado toString() que regresa la representación textual del objeto. Cada clase modifica en cierta manera el comportamiento de toString() para regresar valores de acuerdo a la tarea para la que fue creada la clase.

Encapsulamiento es la característica de organizar datos y funciones en una estructura. Además esto permite ocultar código para facilitar la programación y mantenimiento. Las variables casi siempre son privadas, ya que dejar que un objeto ajeno modifique el valor de una variable puede conducir a un mal funcionamiento del objeto que contiene esa variable o se pueden presentar resultados indeseables.

Polimorfismo es la capacidad de un objeto de asumir diferentes formas pero aun siendo compatible en tipo con el código existente.

Creación de una clase

Una clase debe tener la siguiente sintaxis:

[<modificador>] class <nombre_de_clase>

En Java, generalmente se crean clases public o default. Y en un código fuente sólo puede haber una clase public. El siguiente es un ejemplo sencillo de creación de una clase:

```
public class Fecha
{
 private int dia,mes,anio;
 public void manana()
 {
  // calcular el día siguiente
 }
}
```

Creación de un objeto

Para que se puedan realizar las tareas que define la clase Fecha, se tiene que hacer una instancia de la clase:

```
Fecha fecha;
fecha = new Fecha();
// tal vez el constructor ya definió la fecha actual
fecha.manana();
// dentro del código de tomorrow se calcula el día siguiente
```

Si definimos otra variable de tipo Fecha y le asignamos el valor de fecha, Java no crea un nuevo objeto, sino que ambas variables apuntan a la misma localidad de memoria:

```
Fecha fecha1,fecha2;
fecha1 = new Fecha();
fecha2 = fecha1;
```

En memoria esto se vería así:

```
fecha1              Fecha
fecha2
```

La referencia de variables y métodos con this

En Java la palabra reservada this se usa para accesar variables y métodos del mismo objeto (ya creado) para facilitar la programación. this no se puede usar con llamadas static de métodos.

```
public class Fecha
{
 private int dia,mes,anio;
 public void manana()
 {
  this.day = this.day + 1;
  // más código
 }
}
```

En este ejemplo resulta redundante el uso de this, pero puede entenderse que this.day es la variable de la clase. En el siguiente ejemplo se fija un número de día pasando un parámetro al método fijaDia():

```
public class Fecha
{
 private int dia,mes,anio;
 public void fijaDia(int dia)
 {
  this.dia = dia;
 }
}
```

Este ejemplo es completamente válido y legal, ya que el primer elemento de la asignación se refiere a la variable de la clase, y el segundo elemento de la asignación se refiere al parámetro del método fijaDia.

Sobrecarga de métodos

En ciertas circunstancias, uno desea escribir varios métodos en la misma clase que hagan básicamente lo mismo pero con diferentes parámetros. Considérese el ejemplo en el que se quiere imprimir la representación textual de un parámetro, por ejemplo el método print().

Este método podría recibir como parámetros un int, long, double, String, etc. Es posible que cada tipo de dato requiera de alguna conversión para poder imprimir su valor como cadena de caracteres. En lenguajes estructurados, la solución sería crear funciones printint(), printfloat(), printString(), pero sería una tarea tediosa.
Reusando el nombre del método, se tienen las siguientes definiciones:

```
public void print(int i)
public void print(double d)
public void print(String s)
```

En tiempo de ejecución, el programa sabe a que método llamar dependiendo del tipo de dato del argumento:

```
clase.print(5);
clase.print(3.14159264);
clase.print("mensaje");
```

Constructores

Un método constructor, como ya se dijo, es el que se encarga de dar ciertos valores iniciales a las variables de la clase.

Existen dos reglas importantes para los constructores:
1.El nombre del método debe ser exactamente el mismo que el de la clase.
2.No se debe declarar tipo de dato de regreso del método.

```
public class Clase
{
 // variables de la clase
 public Clase()
 {
  // inicialización de algunas variables
 }
 public Clase(int x)
 {
  // inicialización del objeto con un parámetro
 }
}

Clase c,d;
c = new Clase();
d = new Clase(1);
```

Como se ve en el ejemplo anterior, los constructores también se pueden sobrecargar para poder recibir cualquier cantidad de parámetros y los parámetros pueden ser de diferentes tipos. En tiempo de ejecución, el programa sabe a que constructor llamar.

Cada clase tiene al menos un constructor. Si no se define uno, Java lo crea automáticamente sin parámetros ni código. De no ser así, se tendría que definir al menos un constructor para poder crear la instancia del objeto.

Si en una clase se definen constructores con parámetros, se pierde el constructor que crea Java, por lo que una llamada a new Clase() generaría un error de compilación.

Subclases

En Java todas las clases son subclases de Object. Se crean subclases para realizar tareas específicas a partir de una clase padre que contiene información básica (variables y métodos). Por ejemplo, se puede crear la clase Empleado y luego crear subclases con características específicas.

```
public class Empleado
{
 private String nombre;
 private Date fechaNacimiento;
 private String puesto;
 ...
}

public class Gerente
{
 private String nombre;
 private Date fechaNacimiento;
 private String puesto;
 private String departamento;
 private Empleado subordinados[];
 ...
}
```

En este ejemplo se muestra la duplicidad de información entre las clases Empleado y Gerente. Y no solo en las variables, también habrá métodos que se dupliquen, por ejemplo el del cálculo de antigüedad.

Por lo tanto, se puede ver que la clase Gerente tiene las mismas y más características y acciones que la clase Empleado, por lo que es posible crear una subclase Gerente que herede de Empleado.

```
public class Empleado
{
 private String nombre;
 private Date fechaNacimiento;
 private String puesto;
 ...
}

public class Gerente extends Empleado
{
 private String departamento;
 private Empleado subordinados[];
 ...
}
```

Se usa la palabra reservada extends para indicar la herencia de una clase a otra, en este ejemplo la clase Gerente "extiende" a la clase Empleado en variables y métodos extra. Ahora la clase Gerente tiene las mismas características que Empleado y además otras que son propias para esa clase. La clase Gerente heredó de la clase Empleado sus variables y métodos.

Las subclases permiten claridad y mantenimiento en la programación. Si se hace una corrección en la clase Empleado, la clase Gerente es corregida sin que el programador haya trabajado doble.

Polimorfismo

Siguiendo con el ejemplo de las clases Empleado y Gerente, se puede tener un método calculaSalario() en la clase Empleado, por lo que también se hereda a Gerente. Esto lleva a la idea de que los objetos son polimórficos. Un objeto en particular puede tener la forma de Gerente, pero también tiene la forma de Empleado.

Para ver un efecto del polimorfismo con estas clases, se puede crear un arreglo que contenga tanto empleados como gerentes:

```
Empleado staff[] = new Empleado[100];
staff[0] = new Gerente();
staff[1] = new Empleado();
```

Esto crea un arreglo heterogéneo, es decir, una colección de objetos diferentes. También en el paso de parámetros el polimorfismo toma importancia. Considérese el cálculo de antigüedad:

```
public int calcularAntiguedad(Empleado e)
{
 // calcular
}

Gerente g = new Gerente();
int antiguedad = calcularAntiguedad(g);
```

Al principio puede parecer ilógico crear un gerente y calcular su antigüedad como empleado. En este caso se uso la clase Empleado como objeto genérico en el parámetro de calcularAntiguedad(), porque un Gerente es un Empleado, pero un Empleado no es un Gerente.

Cast de objetos y la palabra reservada instanceof

Se usa la palabra reservada instanceof para determinar a que clase pertenece el objeto que está siendo utilizado en cierto momento. Por ejemplo:

```
public class Empleado
public class Gerente extends Empleado
public class Director extends Empleado

public void quien(Empleado e)
{
 if(e instanceof Gerente)
 {
  // jefe inmediato superior
 }
 else
 if(e instanceof Director)
 {
  // toma decisiones importantes
 }
 else
 {
  // le puedo hablar de tu
 }
}
```

Dentro de este bloque de código se pueden llamar métodos específicos de cada objeto. Para hacer esto se crea una instancia de la clase apropiada usando un cast, y se llaman a los métodos que se requieran:

```
public void quien(Empleado e)
{
 if(e instanceof Gerente)
 {
  Gerente g = (Gerente)e;
  System.out.println("Gerente del departamento " + g.departamento);
 }
 ...
}
```

Esto permite que el programa compile y corra sin errores. Se tiene que hacer la instancia porque la variable departamento no pertenece a la clase Empleado:

```
System.out.println(e.departamento); // error!
```

Redefinición de métodos

Además de heredar los métodos de una clase padre, se puede modificar la acción de cualquier método, con el fin de mejorar o ampliar esa acción.

La regla que se debe seguir para redefinir un método es que el tipo de dato de retorno y el número y tipo de datos de los parámetros tiene que ser idéntico al definido en la clase padre. Sigamos con una variación del ejemplo de Empleado y Gerente:

```
public class Empleado
{
 private String nombre;
 private int salario;

 public String obtenDetalles()
 {
  return "Nombre: " + nombre + "\n" + "Salario: " + salario;
 }
}

public class Gerente extends Empleado
{
 private String departamento;

 public String obtenDetalles()
 {
  return "Nombre: " + detalles + "\n" + "Departamento: " + departamento;
 }
}
```

La clase Gerente tiene un método llamado obtenDetalles() porque lo hereda de Empleado. El método original ha sido redefinido o reemplazado para regresar un mensaje diferente.

Por ultimo, una comparación entre Java y C++ con respecto a lo visto en este Capítulo. En C++ se puede redefinir un método marcándolo como virtual en el código fuente. En lenguajes puros orientados a objetos esto no es normal. C++ hace esto para incrementar la velocidad de ejecución.

Ejemplo:

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

```
public class Point
{
 int x = 0;
 int y = 0;

 void setX(int iniX)
 {
  x = iniX;
 }

 void setY(int iniY)
 {
  y = iniY;
 }
 int getX()
 {
  return x;
 }

 int getY()
 {
  return y;
 }

 public static void main(String args[])
 {
  Point start = new Point();
  Point end = new Point();
  Point stray = end;

  start.setX(10);
  start.setY(10);
  end.setX(20);
  end.setY(30);
  System.out.println("Initial values:");
  System.out.println("Start point: "+start.getX()+","+start.getY());
  System.out.println("End point: "+end.getX()+","+end.getY());
  System.out.println("stray point: "+stray.getX()+","+stray.getY());
  System.out.println("\nAssigning new values to stray: 19,91");
  stray.setX(19);
  stray.setY(91);
  System.out.println("End point: "+end.getX()+","+end.getY());
  System.out.println("stray point: "+stray.getX()+","+stray.getY());
  System.out.println("\nAssigning new values to start: 25,52");
  start.setX(25);
  start.setY(52);
  System.out.println("Final values:");
  System.out.println("Start point: "+start.getX()+","+start.getY());
  System.out.println("End point: "+end.getX()+","+end.getY());
  System.out.println("stray point: "+stray.getX()+","+stray.getY());
 }
}
```

Laboratorio 4

1. Realizar un programa que demuestre la sobrecarga de métodos.
   Ver Cap4\SobreCarga.java
2. Realizar un programa que defina una clase genérica llamada Empleado. Esta clase debe heredar a otra llamada Gerente y ésta a su vez a una llamada Director. Todas deben imprimir las características especificas para cada clase. Además, este programa deberá de tener la capacidad (por medio de otra clase que va a funcionar como clasificador) de identificar a cada uno de estos cuando reciba como parámetro un objeto de este tipo.
   Ver Cap4\Prueba.java (Empleado.java, Gerente.java, Director.java)

Capítulo 5 - Excepciones

Excepciones

En Java, la clase Exception define condiciones de error leves que los programas pueden encontrar. En vez de dejar que el programa termine, el programador puede escribir código para manejar esas excepciones y continuar con la ejecución del programa.

Excepciones y errores

Y qué es un error? En Java, la clase Error define lo que es considerado como una condición de error grave que el programador no debería de recuperar. En la mayoría de los casos, es aconsejable dejar que el programa termine cuando se encuentra un error.

Java implanta el estilo de C++ de las excepciones para construir código flexible. Cuando ocurre un error en el programa, el código que encuentra el error "lanza" una excepción. Este proceso de lanzar una excepción indica al proceso actual en ejecución que ha ocurrido un error. El programador puede atrapar la excepción y, cuando sea posible, recuperar el control del programa. Considérese el siguiente programa que es una extensión del programa HelloWorld.java:

```
public class HelloWorld2
{
 public static void main(String argv[])
 {
  int i = 0;
  String saludos[] = {"Hola mundo!",
   "No, creo que mejor digo",
   "HOLA MUNDO!"};

  while(i < 4)
  {
   System.out.println(saludos[i]);
   i++;
  }
 }
}
```

Después de que el programa compiló sin errores y se ejecuta, se obtiene el siguiente resultado:

```
$java HelloWorld2
Hola mundo!
No, creo que mejor digo
HOLA MUNDO!
java.lang.ArrayIndexOutOfBoundsException: 3
     at HelloWorld.main(HelloWorld.java):12)
```

Como se ve, la importancia de manejar las excepciones es poder escribir código para atrapar las excepciones, manejarlas, y continuar con la ejecución del programa.

Uso de try y catch

Para manejar una excepción en particular, se usa la palabra try con el código que puede lanzar una excepción, este código también es llamado código protegido. Para atrapar y actuar cuando es lanzada una excepción, se usa catch para especificar la excepción a atrapar y el código a ejecutar si se lanza la excepción.

```
try
{
 // código que puede lanzar una excepción en particular
}
catch(ExceptionType e)
{
 // código a ejecutar si ExceptionType es lanzada
}
```

La palabra finally sirve para definir el bloque de código que se ejecutara siempre, sin importar si la excepción fue atrapada o no.

```
try
{
 ...
}
catch(Exception e)
{
 ...
}
finally
{
 // esto se ejecuta aunque no se atrape la excepción
}
```

En el siguiente ejemplo se rescribe el anterior, pero ahora se atrapa la excepción y el programa no generará mensajes de error:

```
public class HelloWorld2
{
 public static void main(String argv[])
 {
  int i = 0;
  String saludos[] = {"Hola mundo!",
   "No, creo que mejor digo",
   "HOLA MUNDO!"};

  while(i < 4)
  {
   try
   {
    System.out.println(saludos[i]);
   }
   catch(ArrayOutOfBoundsException e)
   {
    System.out.println("Ha ocurrido la excepción: "+e.toString());
   }
   finally
   {
    System.out.println("Esto siempre se ejecutara");
   }
   i++;
  }
 }
}
```

Excepciones más comunes

A continuación se presentan algunas de las excepciones más comunes:

ArithmeticException.- Típicamente el resultado de dividir entre cero.
```
int = 12 / 0;
```
NullPointerException.- Un intento de accesar un objeto o método antes de que sea instanciado.
```
String mensaje = null;
System.out.println(mensaje);
```

NegativeArraySizeException.- Un intento de crear un arreglo con el valor de tamaño negativo.

ArrayIndexOutOfBoundsException.- Un intento de accesar un elemento de un arreglo más allá del límite del arreglo.

SecurityException.- Típicamente lanzada en un navegador, la clase SecurityManager lanza una excepción en los applets que intentan:
- Accesar un archivo local.
- Abrir un socket en un host diferente al que pertenece el applet.
- Ejecutar otro programa.

Creación de excepciones

Para fomentar la programación de código robusto, Java requiere que si un método hace algo que puede resultar en una excepción, entonces debe quedar claro que acción se debe tomar si el problema se presenta.

Hay dos cosas que el programador puede hacer para cumplir este requisito: el primero es usar el bloque try{}catch(){} donde se va a usar una subclase de Exception, aunque el bloque catch quede vacío; la segunda, es indicar que la excepción no es manejada en este método, y que por lo tanto será lanzada al método que hace la llamada. Esto se hace indicando la declaración del método como sigue:

```
public void metodo() throws Exception
```

Después de la palabra reservada throws va una lista de todas las excepciones que pueden ser lanzadas por el método, pero no tratar de manejarla en este lugar. Aunque solamente una excepción se muestra aquí, una lista separada por comas puede ser usada si hay múltiples posibles excepciones que no serán manejadas en el método.

Ejemplo de creación de excepciones:
(CantidadException - Subclase de Exception para crear una excepción.
DemoException - Clase que genera una excepción y la atrapa)

```
// CantidadException.java
public class CantidadException extends Exception
{
 private String mensaje;
 private int cantidad;

 public CantidadException(String mensaje, int cantidad)
 {
  this.mensaje=mensaje;
  this.cantidad=cantidad;
 }


 public String toString()
 {
  return mensaje+" "+cantidad;
 }
}
```

```
// DemoException.java
import java.lang.*;
import CantidadException;

public class DemoException
{
 public DemoException()
 {
  System.out.println("Ejemplo de una Exception");
 }

 public void imprimeNumero(int n) throws CantidadException
 {
  if(n > 10)
  {
   throw new CantidadException("Imposible imprimir el número",n);
  }
  else
  {
   System.out.println("x="+n);
  }
 }

 public static void main(String argv[])
 {
  DemoException demo = new DemoException();
  int x;

  for(x=0;x<15;x++)
  {
   try
   {
    demo.imprimeNumero(x);
   }
   catch(CantidadException e)
   {
    System.out.println(e.toString());
   }
  }
 }
}
```

Laboratorio 5

1. Realizar un programa que lea de la entrada estándar y que aviente una excepción que no sea manejada.
   Vea Cap5\Lee.java
2. Completar el programa anterior para que maneje la excepción.
   Vea Cap5\Lee2.java

*Nota: Para ambos programas utlilice el método "read" de la clase System.in para leer de la entrada estándar.

Capítulo 6 - Flujos de entrada/salida

¿Qué es un flujo (stream)?

Un flujo es una fuente de bytes o un destino para los bytes. El orden es importante. Así, por ejemplo, un programa que requiere entrada del teclado puede usar un flujo para hacer su tarea.
Las dos categorías básicas de flujos son: flujos de entrada y flujos de salida. Se puede leer de un flujo de entrada, pero no se puede escribir en él. De igual manera, se puede escribir en un flujo de salida, pero no se puede leer de él.

Métodos de flujo de entrada

```
int read()
int read(byte[])
int read(byte[],int,int)
```

Estos tres métodos proveen acceso a los datos entrantes. El método simple read() regresa un int que contiene ya sea un byte del flujo ó -1, si se alcanzó la condición de fin de archivo. Los otros dos métodos son para leer en un arreglo de bytes y regresan el número de bytes leídos. Los dos argumentos int en el tercer método indican un subrango en el arreglo destino que necesita ser llenado. Para lograr mayor eficiencia, se procurara leer datos en el bloque más grande posible.

```
void close()
```

Cuando se ha terminado de recibir datos, se debe cerrar el flujo.

```
int available()
```

Este método reporta el número de bytes que están inmediatamente disponibles para ser leídos del flujo.

```
skip(long)
```

Este método omite el número especificado de caracteres del flujo.

Métodos de flujo de salida

```
void write(int)
void write(byte[])
void write(byte[],int,int)
```

Estos métodos escriben a un flujo de salida. Como en la entrada, se procurara tratar de escribir datos en el mayor bloque posible.

```
void close()
```

Los flujos de salida deben cerrarse cuando ya no se van a usar.

```
void flush()
```

En ocasiones un flujo de salida acumula los datos antes de enviarlos. Este método permite forzar la escritura.

Objetos básicos de flujo de entrada y salida

Existen varias clases para flujos en el paquete java.io. A continuación se presenta un diagrama jerárquico de las clases que están en el paquete.

InputStream
- SequenceInputStream
- PipedInputStream
- FilterInputStream
    - DataInputStream
    - PushbackInputStream
    - BufferedInputStream
    - LineNumberInputStream
- StringBufferInputStream
- ByteArrayInputStream
- FileInputStream

FileInputStream y FileOutputStream

Estas clases usan archivos. Los constructores para estas clases permiten especificar la ruta al archivo al que serán conectados. Para construir un FileInputStream, al archivo asociado debe existir y ser leíble. Si se va a construir un FileOutputStream, el archivo de salida deberá ser sobrescribible si ya existe.

```
FileInputStream infile = new FileInputStream("myfile.dat");
FileOutputStream outfile = new FileOutputStream("results.dat");
```

BufferedInputStream y BufferedOutputStream

Estos son flujos con filtro y pueden ser usado para aumentar la eficiencia en operaciones de entrada/salida.

DataInputStream y DataOutputStream

Estos flujos permiten leer y escribir tipos de datos primitivos de Java. Existen métodos para cada tipo de dato. Por ejemplo:

- Métodos de DataInputStream
```
byte readByte()
long readLong()
double readDouble()
```

- Métodos de DataOutputStream
```
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

Estas clases tienen métodos para lectura y escritura de cadenas de caracteres, pero no deberían usarse. En vez de eso, hay clases lectores y escritores.

Lectores y escritores con buffer

Java usa Unicode para representar cadenas de caracteres y caracteres, y provee versiones de flujos de 16 bits para permitir que los caracteres sean tratados como caracteres, independientemente de la plataforma. Estas versiones de 16 bits son los llamados lectores y escritores. Existen varios disponibles en el paquete java.io.

Las versiones más importantes de los lectores y escritores son el InputStreamReader y el OutputStreamWriter. Estas clases son usadas como interfaces entre flujos de bytes y lectores y escritores de caracteres.

Existen las clases BufferedReader y BufferedWriter para facilitar la conversión de formatos, que dependen de la plataforma. Se pueden agregar al final de un constructor InputStreamReader o OutputStreamWriter.

En el siguiente ejemplo se muestra una técnica sencilla para leer una cadena de caracteres, un String, desde el teclado:

```
import java.io.*;

public class CharInput
{
 public static void main(String argv[]) throws java.io.IOException
 {
  String s;
  InputStreamReader ir;
  BufferedReader in;

  ir = new InputStreamReader(System.in);
  in = new BufferedReader(ir);

  while((s = in.readLine()) != null)
  {
   System.out.println("Se leys: " + s);
  }
 }
}
```

Archivos

La clase File ofrece varias utilerías para diferentes operaciones sobre los archivos.

```
File archivo = new File("datos.txt");

archivo = new File("./","datos.txt");
// mas útil si se usan variables en los parámetros

File directorio = new File("./nombres");
archivo = new File(directorio,"mujeres.txt");
```

El constructor a usar depende del número de archivos que se manipularán al mismo tiempo. Si se va a usar un sólo archivo se puede usar el primer ejemplo, pero si se van a usar varios archivos de un directorio en común, es mejor usar los ejemplos dos y tres.

Se puede usar un objeto File como el parámetro del constructor de FileInputStream y FileOutputStream, en lugar de un String. Esto permite mejor independencia de las convenciones del sistema de archivos local, y generalmente es más recomendable.

Métodos de la clase File

Una vez que se creó un objeto File, se pueden usar los siguientes métodos para obtener información del archivo:

- Nombres de archivo:
```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo(File nuevoNombre)
```

- Pruebas sobre archivos:
```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
```

- Información general y utilerías:
```
long lastModified()
long length()
boolean delete()
```

- Utilerías de directorio:
```
boolean mkdir()
String[] list()
```

Archivos de acceso aleatorio

Habrá ocasiones en las que se necesitará leer o escribir datos sin hacerlo de principio a fin del archivo. Se puede accesar un archivo de texto, como una base de datos, en el que se mueve a lo largo del archivo para leer un registro, luego otro, luego otro, que están en diferentes partes del archivo. El lenguaje Java ofrece la clase RandomAccessFile para hacer estas operaciones de entrada/salida.

Para crear un archivo de acceso aleatorio se tienen las siguientes opciones:

- Con el nombre del archivo:
```
myRAFile = new RandomAccessFile(String name, String mode);
```

- Con un objeto File:
```
myRAFile = new RandomAccessFile(File file, String mode);
```

El parámetro mode determina si se tiene acceso de solo lectura (r) o de lectura/escritura (rw) al archivo. Por ejemplo, para abrir un archivo de bases de datos para actualizar, se usa la siguiente sintaxis:

```
RandomAccessFile myRAFile;
myRAFile = new RandomAccessFile("db/almacen.dbf","rw");
```

Métodos de la clase RandomAccessFile

Un objeto RandomAccessFile espera leer y escribir información de la misma manera objetos de entrada y de salida. Se tiene acceso a todas las operaciones de read() y write() de las clases DataInputStream y DataOutputStream. Además se cuenta con los siguientes métodos:

`long getFilePointer()`.- Regresa la posición actual del apuntador a archivo.
`void seek(long pos)`.- Fija el apuntador al archivo en la posición absoluta especificada en pos. La posición está dada como un corrimiento de bytes desde el inicio del archivo. La posición 0 representa el inicio del archivo.
`long length()`.- Regresa el tamaño del archivo, además indica el fin del archivo.

Para agregar información en un archivo de acceso aleatorio, primero se tienen que posicionar el apuntador de archivo en el final:

```
myRAFile = new RandomAccessFile("db/results.dbf","rw");
myRAFile.seek(myRAFile.length());
// cada llamada a write agregara al archivo
```

Ejemplo de lectura de archivos:

```
// java Cat [archivo1 archivo2]
import java.io.*;
import java.lang.*;

public class Cat
{
 private String archivos[];

 public Cat(String archivos[])
 { // constructor
  this.archivos = archivos;
 }

 public static void main(String argv[])
 {
  if(argv.length > 0)
  {
   Cat programa = new Cat(argv);
   programa.leeArchivos();
  }
  else
  {
   System.out.println("Falto especificar archivo a leer");
  }
 }

 public void leeArchivos()
 {
  BufferedReader datos;

  for(int x = 0; x<archivos.length; x++)
  {
   try
   {
    datos = new BufferedReader(new FileReader(new File(archivos[x])));

    String buf = new String();
    while(buf != null)
    {
     try
     {
      buf = datos.readLine();
      if(buf != null) System.out.println(buf);
     }
     catch(IOException lee)
     {
      System.out.println("Error de lectura de archivo");
     }
    }
    datos.close();
   }
   catch(IOException e)
   {
    System.out.println("No se puede abrir el archivo o no existe\n"+
     e.toString());
   }
  }
 }
}
```

Laboratorio 6

1. Realizar un programa que lea y escriba archivos.
   Ver Cap6\ReadWrite.java

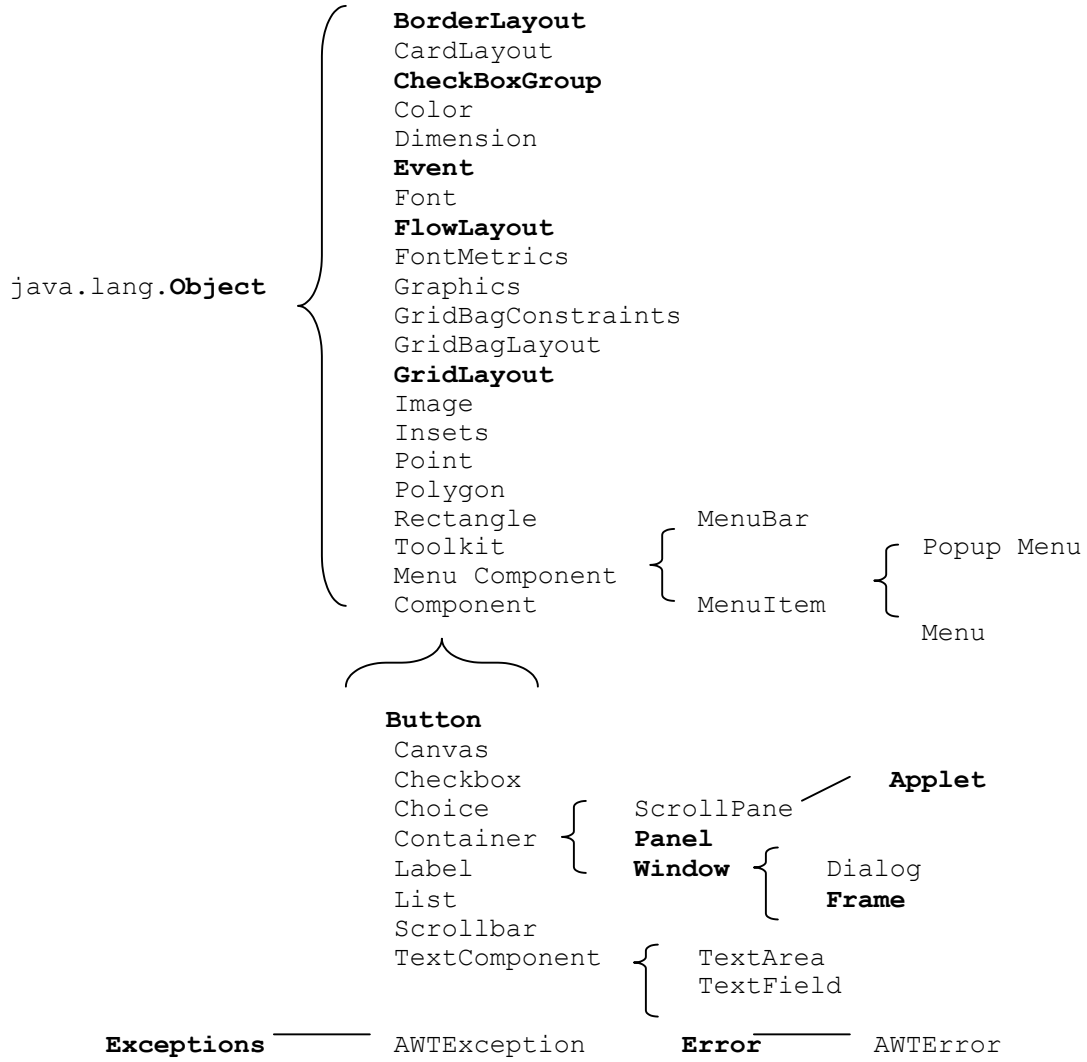
Capítulo 7 – AWT

Características del AWT

AWT provee los componentes GUI (Graphic User Interface) básicos que son usados en los applets de Java y las aplicaciones. El AWT provee una interfaz independiente de la máquina para aplicaciones. Esto asegura que lo que aparece en una computadora es comparable con lo que aparece en otra.

Cada componente GUI que aparece en la pantalla es una subclase de la clase abstracta *Component.* Esto significa que cada objeto gráfico que herede de la clase *Component* comparte un número de métodos y variables de instancia que les permiten operar.

Container es una subclase abstracta de *Component,* que permite que otros componentes sean anidados en él. Estos componentes pueden ser a su vez contenedores permitiendo que otros componentes también sean contenidos dentro de este, lo que crea una estructura jerárquica completa. Los Contenedores son útiles para arreglar a los componentes en la pantalla. El *Panel* es la subclase concreta más simple de *Container*. Otra subclase de *Container* es un *Window.*



Contenedores

Existen dos tipos principales de contenedores: *Window y Panel.*

Window es un objeto de *java.awt.Window*. Un *Window* es una ventana nativa que es independiente de otros contenedores.
Existen dos tipos importantes de *Window: Frame y Dialog. Frame* es una ventana con un título y esquinas reajustables. *Dialog* no tiene una barra de menú y aunque lo puedes mover, no lo puedes reajustar.

Panel es un objeto de *java.awt.Panel*. El *Panel* está contenido en otro contenedor, o dentro de la ventana de un navegador. *Panel* es un área rectangular en la que puedes poner otros componentes. Se debe poner un *Panel* dentro de un *Window* (o una subclase de esta) para que pueda ser desplegado.

Posicionamiento de Componentes

La posición y tamaño de un componente en un contenedor esta determinado por un *Layout Manager*. Un contenedor mantiene una referencia a una instancia particular de un Layout Manager. Cuando un contenedor necesita posicionar un componente, este invoca al Layout Manager para que realice la tarea. La misma delegación del trabajo ocurre cuando se quiere decidir el tamaño del componente. El Layout Manager toma el control completo sobre todos los componentes dentro de un contenedor. Es su responsabilidad definir el tamaño del objeto en el contexto del tamaño actual de la pantalla.

Cambiando el tamaño de los Componentes

Como el Layout Manager es generalmente el responsable del tamaño y la posición de los componentes en un contenedor, no debería (normalmente) tratar de definir el tamaño o la posición de los componentes por sí mismo. Si trata de hacerlo (usando los métodos `setLocation(), setSize o setBounds`) el Layout Manager sobrescribirá su decisión.

Si usted debe controlar el tamaño o la posición de los componentes en alguna manera que no pueda ser hecha usando los Layout Managers estándar, usted puede desactivarlos usando la siguiente llamada al método de su contendor:

```
Cont.setLayout(null);
```

Después de este paso, debe usar `setLocation(), setSize, o setBounds()` en todos los componentes para localizarlos en el contenedor. Debe tener en cuenta que esta solución es dependiente de la plataforma debido a la diferencia que existe entre los sistemas de ventanas y los tamaños de letra. Lo mejor seria crear una subclase de *LayoutManager.*

Frames

Frame es una subclase de *Window*. Es una ventana con título y esquinas reajustables. *Frame* hereda de *java.awt.Container* para que pueda agregar componentes al *Frame* usando el método *add()*. El Layout Manager por default para un Frame es el *BorderLayout*, aunque se puede cambiar usando el método *setLayout()*.

El constructor *Frame(String)* en la clase *Frame* crea un objeto *Frame* nuevo, invisible y con título *String*. Usted puede agregar componentes al Frame mientras siga invisible.

Ejemplo: El siguiente programa crea un *Frame* que tiene un título, un tamaño y un color de fondo específicos.

```
import java.awt.*;

public class MyFrame extends Frame
{
 public MyFrame (String str)
 {
  super (str);
 }
 public static void main (String argv[])
 {
  MyFrame fr=new MyFrame ("Hola Mundo!");
  fr.setSize(500,500);
  fr.setBackground(Color.blue);
  fr.setvisible(true);
 }
}
```

Panel

Panel, como *Frame*, provee de espacio para que se pueda poner cualquier componente GUI dentro de él, incluyendo otros paneles. Cada *Panel*, los cuales heredan de *java.awt.Container* puede tener su Layout Manager.
Cuando un *Panel* es creado, se debe agregar a un *Window* o *Frame* para que esté visible. Esto se hace por medio del método *add()* de la clase *Container*.

Ejemplo: El siguiente programa crea un pequeño *Panel* de color amarillo y lo agrega a un *Frame*.

```
import java.awt.*;

public class FrameWithPanel extends Frame
{
 public FrameWithPanel (String str)
 {
  super(str);
 }

 public static void main (String argv[])
 {
  FrameWithPanel fr=new FrameWithPanel ("Frame con Panel");
  Panel pan=new Panel();

  fr.setSize(200,200);
  setBackground(Color.blue);
  fr.setLayout(null);

  pan.setSize(100,100);
  pan.setBackground(Color.yellow);
  fr.add(pan);
  fr.setVisible(true);
 }
}
```

Layouts para Contenedores

El *Layout* de componentes en un contenedor es usualmente gobernado por un Layout Manager. Cada contendor (como una *Panel* o un *Frame*) tiene un Layout Manager asociado a él, pero puede ser modificado utilizando *setLayout()*.

El layout Manager es el responsable de decidir la política del Layout y los tamaños para cada uno de los componentes que existen en el contenedor.

Layout Managers

Los siguientes Layout Managers están incluidos en el lenguaje de Programación de Java:
`FlowLayout` – El Layout Manager por defecto de un *Panel y Applet*.
`BorderLayout` – El Layout Manager por defecto de *Window, Dialog y Frame.*
`GridLayout`
`CardLayout`
`GridBagLayout`

FlowLayout Manager

El *FlowLayout* posiciona los componentes en una base línea por línea. Cada vez que una línea se llena, una nueva línea es creada. A diferencia con otros Layout Managers, el *FlowLayout* no forza el tamaño de los componentes que maneja, sino que les permite tener su tamaño.

Los argumentos del constructor del *FlowLayout* le permiten controlar el flujo de los componentes a la izquierda, derecha o centrado (por defecto). Es importante hacer notar que cuando el usuario reajusta el tamaño del área que está siendo manejada por el *FlowLayout*, éste reacomoda los componentes y se ajusta al nuevo tamaño.

Ejemplo: El siguiente código añade varios botones al *FlowLayout* en un *Frame.*

```
import java.awt.*;

public class MyFlow
{
 private Frame f;
 private Button button1, button2, button3;

 public void go()
 {
  f=new Frame("Flow Layout");
  f.setLayout(new FlowLayout());
  button1=new Button ("Ok");

  button2=new Button ("Open");
  button2=new Button ("Close");
  f.add(button1);
  f.add(button2);
  f.add(button3);
  f.setSize(100,100);
  f.setVisible(true);
 }



public static void main (String argv[])
 {
  MyFlow mflow=new MyFlow();
```

```
  Mflow.go();
 }
}
```

BorderLayout Manager

El B*orderLayout Manager* provee un esquema un poco más complicado para el posicionamiento de los componentes en un contenedor. Es el Layout por defecto de *Frame y Dialog*. El B*orderLayout* Manager contiene 5 áreas distintas: NORTH (Norte), SOUTH (Sur), EAST (Este), WEST (Oeste) y CENTER (Centro), indicadas por BorderLayout.NORTH, y así sucesivamente.

NORTH ocupa la parte superior del Frame, EAST la parte derecha, y así sucesivamente. CENTER ocupa todo el espacio que no se ocupó una vez que se han definido NORTH, SOUTH, EAST y WEST. Cuando la ventana es reajustada verticalmente, también se reajustan las regiones EAST, WEST y CENTER, mientras que si la ventana se ajusta horizontalmente, las regiones NORTH, SOUTH y CENTER son las que se ajustan al nuevo tamaño.

Sólo se puede añadir un componente a cada una de las 6 regiones del *BorderLayoutManager*. Si se trata de agregar más de uno, solamente el último permanecerá visible. Usando contenedores intermedios se puede permitir que se encuentre más de un componente en una misma región del *BorderLayout.*

Nota: El Layout Manager mantiene la altura de los componentes en las regiones NORTH y SOUTH, pero los forza a ser tan anchos como el contenedor. En el caso de los componentes en WEST e EAST, el ancho se respeta mientras que la altura se forza.

Ejemplo: El siguiente código demuestra el comportamiento del *BorderLayout* Manager.

```
import java.awt.*;

public class MyBorder
{
 private Frame f;
 private Button bn, ns, bw, be, bc;

 public void go()
 {
  f=new Frame("Border Layout");
  bn=new Button ("Norte");
  bs=new Button ("Sur");
  bw=new Button ("Oeste");
  be=new Button ("Este");
  bc=new Button ("Centro");

  f.add(bn, BorderLayout.NORTH);
  f.add(bs, BorderLayout.SOUTH);
  f.add(bw, BorderLayout.WEST);
  f.add(be, BorderLayout.EAST);
  f.add(bc, BorderLayout.CENTER);
  f.setSize(200,200);
  f.setVisible(true);
 }



 public static void main (String argv[])
 {
  MyBorder mborder=new MyBorder();
```

```
  Mborder.go();
 }
}
```

GridLayout Manager

El *GridLayout* Manager provee flexibilidad para el posicionamiento de los componentes. Se crea un manager con un número de renglones y columnas. Los componentes se posicionan en las celdas definidas por el manager. Por ejemplo, un *GridLayout* con 3 renglones y 2 columnas creado por el enunciado *new GridLayout(3,2)* creará 6 celdas.

Como con el *BorderLayout* Manager, la posición relativa de los componentes no cambia cuando el área se reajuste. Únicamente los tamaños de los componentes cambian.

El *GridLayout* manager siempre ignora el tamaño que se haya dado a los componentes. El ancho de las celdas es idéntico y es determinado al dividir el ancho disponible entre el número de columnas. Similarmente, la altura de todas las celdas esta determinada por la división de la altura disponible entre el número de renglones.

El orden en el que los componentes son añadidos a la cuadrícula determina la celda que ocupan. Las líneas de celdas se llenan de izquierda a derecha (como el texto) y la página se llena de arriba abajo.

Ejemplo: El siguiente código demuestra el uso del *GridLayout.*

```
import java.awt.*;

public class MyGrid
{
 private Frama f;
 private Button b1, b2, b3, b4, b5, b6;

 public void go()
 {
  f=new Frame ("Grid Layout");
  f.setLayout(new GridLayout(3,2));

  b1=new Button("1");
  b2=new Button("2");
  b3=new Button("3");
  b4=new Button("4");
  b5=new Button("5");
  b6=new Button("6");

  f.add(b1);
  f.add(b2);
  f.add(b3);
  f.add(b4);
  f.add(b5);
  f.add(b6);
  f.pack();
  f.setVisible(true);
 }

 public static void main(String argv[])
 {
  MyGrid grid=new MyGrid();
```

```
  Grid.go();
 }
}
```

CardLayout Manager

El *CardLayout* manager permite tratar a la interfaz como una serie de cartas, donde sólo es posible ver una a la vez. Se usa el método add() para agregar cartas al *CardLayoutk,* dicho método toma un *String* como argumento e identifica el *Panel* en el programa. El método *show()* del *CardLayout* manager cambia a una nueva carta.

El siguiente código presenta un *Frame* que muestra 5 diferentes *Panels* con cada click del mouse.

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

public class CardTest implements MouseListener
{
 private Panel p1, p2, p, p4, p5;
 private Label lb1, lb2, lb3, lb4, lb5;

 //Se declara un objeto CardLayout para llamar a sus métodos.
 private CardLayout myCard;
 private Frame f;

 public void go()
 {
  f=new Frame ("Card Layout");
  myCard=new CardLayout();
  f.setLayout(myCard);

  //Creación de Paneles los cuales serán usados como cartas.
  p1=new Panel();
  p2=new Panel();
  p3=new Panel();
  p4=new Panel();
  p5=new Panel();

  //Creación de un Label para agregarlo a cada panel y cambio del color
  //de cada panel para que sean fácilmente distinguibles.

  lb1=new Label("Primer Panel");
  p1.setBackground(color.yellow);
  p1.add(lb1);

  lb2=new Label("Segundo Panel");
  p2.setBackground(color.green);
  p2.add(lb2);


  lb3=new Label("Tercer Panel");
  p3.setBackground(color.magenta);
  p3.add(lb3);



  lb4=new Label("Cuarto Panel");
  p4.setBackground(color.white);
```

```
  p4.add(lb4);

  lb5=new Label("Sexto Panel");
  p5.setBackground(color.cyan);
  p5.add(lb5);

  //Manejadores de Eventos

  p1.addMouseListener(this);
  p2.addMouseListener(this);
  p3.addMouseListener(this);
  p4.addMouseListener(this);
  p5.addMouseListener(this);

  //Agregado de cada panel al CardLayout

  f.add(p1, "Primero");
  f.add(p2, "Segundo");
  f.add(p3, "Tercero");
  f.add(p4, "Cuarto");
  f.add(p5, "Quinto");

  //Desplegado del Primer Panel

  myCard.show(f, "Primero");

  f.setSize(200,200);
  f.setVisible(true);
 }

 public void mousePressed(MouseEvent e)
 {
  myCard.next(f);
 }

 public void mouseReleased(MouseEvent e){}
 public void mouseClicked(MouseEvent){}
 public void mouseEntered(MouseEvent){}
 public void mouseExited(MouseEvent){}

 public static void main(String argv[])
 {
  CardTest ct=new CardTest();
  ct.go();
 }

}
```

Más Componentes

Button

Ya debe de estar familiarizado con éste componente. Provee una interfaz "presiona para activar" y puede ser construido con un texto que informa al usuario que uso tendrá.

```
f=new Frame ("Botón de Ejemplo");
b=new Button ("Ejemplo");
b.addActionListener (this);
f.add(b);
```

El método *actionPerformed()* de cualquier clase que implementa la interfaz *ActionListener (*la cual está registrada como *Listener)* es llamado cuando el botón es presionado por un click del mouse.

```
public void actionPerformed(ActionEvent ae)
{
 System.out.println("Se recibió una presión del botón");
 System.out.println("La acción del botón es: "+ae.getActionCommand());
}
```

El método *getActionCommand()* del *ActionEvent* es lanzado cuando el botón es presionado y regresa el texto que se encuentra en el botón por defecto.

Checkbox

El componente *Checkbox* provee un dispositivo básico de "prendido/apagado" con una etiqueta a un lado de éste.

Ejemplo:

```
f=new Frame("Ejemplo de Checkbox");
one=new Checkbox("Uno", true);
two=new Checkbox("Dos", false);
three=new Checkbox("Tres", false);

one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);

f.setLayout(new FlowLayout());
f.add(one);
f.add(two);
f.add(three);
```

La selección o no-selección de un *Checkbox* se manda a la interfaz *ItemListener.* El *ItemEvent* que es pasado, contiene el método *getStateChange()* el cual regresa *ItemEvent.DESELECTED o ItemEvent.SELECTED,* según sea el caso. El método *getItem()* regresa el checkbox que ha sido afectado como un objeto *String* que representa su etiqueta.

```
public void itemStateChanged(ItemEvent ev)
```

```
{
 String state= "deseleccionado";
 If(ev.getStateChange()==ItemEvent.SELECTED)
 {
  state= "seleccionado";
 }
 System.out.println(ev.getItem()+" "+state);
}
```

Checkbox Group – Radio Buttons

El *CheckboxGroup* provee la manera de agrupar múltiples elementos de un Checkbox en un conjunto de elementos mutuamente excluyentes. De esta manera solo un *Checkbox* en el conjunto podrá tener el valor de *true* en cualquier momento. El *Checkbox* con valor *true* es el que está actualmente seleccionado.

Se puede crear un *Checkbox* para que pertenezca a un grupo usando un constructor que recibe un argumento adicional, un *CheckboxGroup*. Es este objeto *CheckboxGroup* el que une cada elemento *Checkbox* en un conjunto. Si usted hace esto, la apariencia de los elementos *Checkbox* es modificada por el comportamiento "radio button".

Ejemplo:

```
f=new Frame("CheckboxGroup");
cbg=new CheckboxGroup();
one=new Checkbox("Uno", cbg, false);
two=new Checkbox("Dos", cbg, false);
three=new Checkbox("Tres", cbg, false);

f.setLayout(new FlowLayout());

one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);

f.add(one);
f.add(two);
f.add(three);
```

Choice

El componente *Choice* provee una entrada del tipo "selecciona de la lista".

Ejemplo:

```
f=new Frame("Ejemplo de Choice");
choice=new Choice();
choice.addItem("Primero");
choice.addItem("Segundo");
choice.addItem("Tercero");
choice.addItemListener(this);
f.add(choice, BorderLayout.CENTER);
```

Cuando se hace click sobre el *Choice*, este despliega la lista de elementos que han sido añadidos a él. Los elementos añadidos son objetos de tipo *String*.

La Interface *ItemListener* es usada para observar los cambios en el *Choice*. Los detalles son los mismos que en el *Checkbox*.

Canvas

El *Canvas* provee un espacio en blanco. Tiene un tamaño de cero por cero, a menos que se especifique lo contrario usando el método *setSize()*. Para especificar el tamaño, se debe de poner en un *Layout Manager* para especificar el tamaño.

Este espacio en blanco se puede utilizar para dibujar, escribir texto o recibir entradas del teclado o mouse. Generalmente un *Canvas* se usa para proveer un espacio general de dibujo o para un área de trabajo de un componente personalizado.

El *Canvas* puede "escuchar" todos los eventos que pueden ser aplicables a un componente en general. En particular, usted podría querer añadir objetos de tipo *KeyListener, MouseMotionListener, o MouseListener* a él para darle la capacidad de responder a las entradas que realiza el usuario de cierta manera.

Nota: Para recibir eventos del tipo *KeyEvent* en un *Canvas*, es necesario llamar al método *requestFocus() Canvas*. Si esto no se hace, no se podrán dirigir las presiones de las teclas hacia el *Canvas.*

Ejemplo: El siguiente código es un ejemplo de *Canvas*. Este programa cambia el color del *Canvas* cada vez que se presiona una tecla.

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

public class MyCanvas extends Canvas implements KeyListener
{
 private int index;
 Color colors[]={Color.red, Color.green, Color.blue};

 public void paint(Graphics g)
 {
  g.setColor(colors[index]);
  g.fillRect(0,0,getSize().width, getSize().height);
 }

 public void keyTyped(KeyEvent ev)
 {
  index++;
  if (index==colors.length)
  {
   index=0;
  }
  repaint();
 }

 //Métodos de KeyListener que no se usan
 public void keyPressed(KeyEvent ev){}
 public viod keyReleased(KeyEvent ev){}




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

```
  Frame f=new Frame ("Canvas");
  MyCanvas mc=new MyCanvas();
  mc.setSize(150,150);
  f.add(mc, BorderLayout.CENTER);
  mc.requestFocus();
  mc.addKeyListener(mc);
  f.pack();
  f.setVisible(true);
 }
}
```

Label

Un objeto *Label* despliega una línea de texto estático. El programa puede cambiar el texto pero el usuario no. No se usan ningún tipo de borde o decoración para delinear a un *Label*.

Ejemplo:

```
Frame f=new Frame ("Label");
Label lb=new Label("hola");
f.add(lb);
```

Un *Label* generalmente no necesita manejar eventos, pero se puede hacer de la misma manera que se haría con un *Canvas*.

TextField

El *TextField* es una línea de entrada de texto.

Ejemplo:

```
Frame f=new Frame ("Textfield");
TextField tf=new TextField("Una sola línea",30);
tf.addActionListener(this);
f.add(tf);
```

Ya que solo es posible tener una línea en un *TextField;* un *ActionListener* puede ser informado usando *actionPerformed()* cuando la tecla ENTER o RETURN ha sido presionada.

Nota: El segundo argumento que tiene el constructor de *TextField* es el número de caracteres visibles. No existe límite en el número de caracteres permitidos en un *TextField.* La barra de desplazamiento aparece cuando el texto se pasa del límite.

Ejemplo:
La aplicación de *TextField* puede usarse para enmascarar la entrada de ciertas teclas. El siguiente código crea un *TextField* que ignora el tecleo de dígitos.

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

public class SampleTextField
{
 private Frame f;
 private TextField tf;

 public void go()
 {
  f=new Frame("TextField");
  tf=new TextField("Una sola línea", 30);
  tf.addKeyListener(new NameHandler());
  f.add(tf, BorderLayout.CENTER);
  f.pack();
  f.setVisible(true);
 }

 class NameHandler extends KeyAdapter
 {
  public void keyPressed(KeyEvent e)
  {
   char c=e.getKeyChar();
   if (Character.isDigit(c))
   {
    e.consume();
   }
  }
 }

 public static void main (String args[])
 {
  SampleTextField txtf=new SampleTextField();
  txtf.go();
 }
}
```

TextArea

El *TextArea* es un dispositivo de entrada de múltiples renglones y columnas. Se puede ajustar a solo lectura usando el método de setEditable(boolean). Este despliega barras de desplazamiento horizontales y verticales.

El siguiente ejemplo muestra un *TextArea* de 4 renglones y 30 columnas con un texto inicial.

```
f=new Frame("TextArea");
ta=new TextArea ("Hello!",4,30);
f.add(ta,BorderLayout.CENTER);
```

El *listener* que se especifica con *addTextListener()* recibe notificación de la presión de teclas en la misma manera que lo hace el *TextField.*

Componentes de Texto

Tanto *TextArea como TextField* están documentados en dos partes. Si usted busca una clase llamada *TextComponent* encontrará varios métodos que *TextArea* y *TextField* tienen en común, por ejemplo *setEditable()*. Esto es posible ya que ambos son subclases de *TextComponent.*

Usted ha podido ver que los constructores de las clases *TextArea* y *TextField* le permiten especificar el número de columnas para el desplegado. Pero recuerde que el tamaño del componente depende del Layout Manager, así que estas referencias pueden ser ignoradas. Más aún, el número de columnas es interpretado en términos del ancho promedio de los caracteres de la letra que se está usando en ese momento, por lo tanto el número de caracteres que son realmente desplegados puede variar radicalmente dependiendo del espaciado que tenga la letra que esta siendo utilizada.

Ya que *TextComponent* implementa de *TextListener* las clases, como *TextField o TextArea* y otras subclases, tienen la capacidad de manejar eventos provocados por el teclado.

List

Un *List* presenta opciones de texto que son desplegadas en una región que permite que varios elementos sean visualizados al mismo tiempo. El *List* es desplazable y soporta selecciones simples o múltiples. Por Ejemplo:

```
List lst=new List(4,true);
lst.add("Hello");
lst.add("there");
lst.add("how");
```

El argumento numérico en el constructor define la altura de la lista en términos de renglones visibles. Como siempre, el Layout Manager puede modificar este valor. El argumento de tipo boolean con valor true indica que esta permitido que el usuario haga una selección múltiple.

Cuando un elemento de la lista es seleccionado o deseleccionado, AWT manda una instancia de *ItemEvent* a la lista. Cuando el usuario hace doble click en un elemento, un evento *ActionEvent* es generado por la lista en el modo de selección ya sea simple o múltiple. Los elementos pueden ser seleccionados de la lista dependiendo de las convenciones de la plataforma.

Dialog

El componente *Dialog* está asociado con *Frame.* Básicamente es una ventana con algunas decoraciones. Difiere de un *Frame* en que existen menos decoraciones y se bloquea la entrada al programa hasta que el *Dialog* haya desaparecido.

Ejemplo:

```
f=new Frame("Ejemplo de Dialog");
d=new Dialog(f, "Dialog", true);
d.setLayout(new GridLayout(2,1));
dl=new Label("Hola, soy un Dialog");
db1=new Button("OK");
d.add(dl);
d.add(db1);
d.pack();
```

El primer argumento en el constructor de *Dialog* designa el propietario que está siendo construido. En el ejemplo anterior *f* es el *Frame* al que pertenece el *Dialog.*

Un *Dialog* generalmente no se hace visible al usuario cuando es creado. Este se despliega en respuesta a una acción en la interfaz del usuario, como la presión de un botón. En este caso se debe agregar algo como lo siguiente:

```
public void actionPerformed(ActionEvent ae)
{
 d.setVisible(true);
}
```

Para esconder un *Dialog* se debe llamar al método *setVisible(false)*. Esto es posible al agregar un *WindowListener* y esperar una llamada al método *windowClosing()* en ese listener.

Menús

Un *Menú* es diferente de todos los demás componentes porque no puede ser añadido a un contenedor ordinario. Sólo se pueden añadir *Menús* a un *Menú Container*. Se puede empezar el "árbol" de un Menú incluyendo un *Menú Bar* en un *Frame* usando el método *setMenuBar()*. De ese punto, usted puede añadir Menus a la Barra de Menú o Elementos de Menú en un Menú.

Usando la Barra de Menú, se puede designar a un Menú para ser el Help Menú o Menú de Ayuda. Esto se hace usando el método *setHelpMenu(Menú)*. Usted debe añadir el Menú que será tratado como Menú de Ayuda a la barra de Menú. Esto hace que sea tratado de la misma manera que sería tratado un Menú de Ayuda en la plataforma local.

MenuBar

El componente *MenuBar* es un Menú horizontal. Solo puede ser añadido a un objeto *Frame,* y forma la raíz para todos los árboles de Menú. Un *Frame* sólo puede desplegar un *MenuBar* al mismo tiempo. Sin embargo, usted puede cambiar el *MenuBar* basado en el estado del programa para que puedan aparecer diferentes menús para varias ocasiones.

Ejemplo:

```
Frame f=new Frame("MenuBar");
MenuBar mb=new MenuBar();
f.setMenuBar(mb);
```

El *MenuBar* no soporta listeners. Como parte del comportamiento normal de un Menú, los eventos que ocurran en la región de una barra de menú son procesados automáticamente.

Menú

Este componente provee un menú desplegable básico. Se puede añadir ya sea a un *MenuBar* o a otro *Menú*.

Ejemplo:

```
f= new Frame("Menu");
mb=new MenuBar();
m1=new Menu("Archivo");
m2=new Menu("Edición");
m3=new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```

Menu Item

Los componentes *MenuItem* son los nodos hoja del árbol de menú. Se le añaden al Menú para completarlo.

Ejemplo:

```
mi1=new MenuItem("Nuevo");
mi2=new MenuItem("Guardar");
mi3=new MenuItem("Abrir");
mi4=new MenuItem("Salir");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.add(mi4);
```

Usualmente se agrega *ActionListener* a un objeto *MenuItem* para proveer un comportamiento para los menús.

Popup Menú

El *PopUpMenu* provee un menú independiente que puede ser desplegado sobre cualquier componente. Se pueden agregar elementos o inclusive otros menús en *Popup Menú.*

Ejemplo:

```
Frame f=new Frame("Popup Menú");
Button b=new Button("Presióname");
PopupMenu p=new PopupMenu("Popup");
MenuItem s=new MenuItem("Salvar");
MenuItem ld=new MenuItem("Abrir");
b.addActionListener(this);
f.add(b, BorderLayout.CENTER);
p.add(s);
p.add(ld);
f.add(p);
```

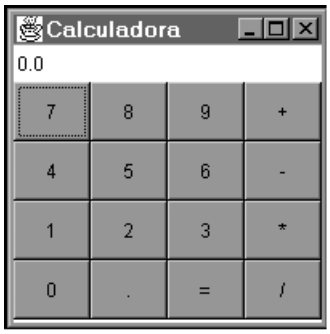
Para que el *PopupMenu* sea desplegado se debe llamar al método *show()*. Este método requiere una referencia a un componente para actuar como el origen de las coordenadas x, y. Se debe de añadir el *PopupMenu* a un componente "padre". Esto no es lo mismo que añadir componentes en contenedores. En el ejemplo, el *PopupMenu* ha sido añadido a un *Frame.*

```
public void actionPerformed(ActionEvent ev)
{
 //desplegado del popup en (10,10) relativo a b
 p.show(b,10,10);
}
```

Nota: La versión de Microsoft Windows del PopupMenu no esta disponible ya que se despliega un PopupMenu que contiene dos botones etiquetados como "Presióname" en lugar de uno.

Laboratorio 7

1. Realizar la interfaz gráfica (GUI) de una calculadora similar a la de la ilustración.
   Ver Cap7\Calculadora.java



Capítulo 8 - Eventos

Introducción a los eventos

¿Qué es un Evento?

Cuando un usuario realiza una acción al nivel de la interfaz de usuario (como el click en el mouse o la presión de una tecla), se origina un *evento*. Los eventos son objetos que describen lo que pasó. Existe gran cantidad de categorías de eventos para describir la acción del usuario.

Originadores de Eventos.

Un originador de eventos es el que hace que un evento ocurra. Por ejemplo, un click del mouse en un Botón genera un *ActionEvent* con el botón como originador. La instancia de *ActionEvent* es un objeto que contiene la información sobre el evento que acaba de llevarse a cabo.

Manejadores de Eventos.

Un manejador de eventos es un método que recibe un objeto evento, lo descifra y lo procesa para interactuar con el usuario.

C*ategorías de Eventos.*

Para cada categoría de Eventos, existe una interfaz que debe ser implementada por la clase que contenga los objetos que desean recibir eventos. Esta interfaz exige que sus métodos estén definidos también. Y esos métodos serán llamados cuando sucedan los eventos.
La siguiente tabla enlista las categorías, dando el nombre de la interface para cada una y los métodos que deberá contener:

Categoría	Nombre de la Interface	Métodos
Acción	*ActionListener*	*actionPerformed(ActionEvent)*
Item	*ItemListener*	*itemStateChanged(ItemEvent)*
Movimiento del Mouse	*MouseMotionListener*	*mouseDragged(MouseEvent)* *mouseMoved(MouseEvent)*
Botón del Mouse	*MouseListener*	*mousePressed(MouseEvent)* *mouseReleased(MouseEvent)* *mouseEntered(MouseEvent)* *mouseExited(MouseEvent)* *mouseClicked(MouseEvent)*
Tecla	*KeyListener*	*keyPressed(KeyEvent)* *keyReleased(KeyEvent)* *keyTyped(KeyEvent)*
Foco	*FocusListener*	*focusGained(FocusEvent)* *focusLost(FocusEvent)*
Ajuste	*AdjustmentListener*	*adjustmentValueChanged(AdjustmentEvent)*
Componente	*ComponentListener*	*componentMoved(ComponentEvent)* *componentHidden(ComponentEvent)* *componentResized(ComponentEvent)* *componentShown(ComponentEvent)*

Ventana	*WindowListener*	*windowClosing(WindowEvent)* *windowOpened(WindowEvent)* *windowIconfied(WindowEvent)* *windowDeiconfied(WindowEvent)* *windowClosed(WindowEvent)* *windowActivated(WindowEvent)* *windowDeactivates(WindowEvent)*
Contenedor	*ContainerListener*	*componentAdded(ContainerEvent)* *componentRemved(ContainerEvent)*
Texto	*TextListener*	*textValueChanged(TextEvent)*

Ejemplo de Eventos:

```
import java.awt.*;
import java.awt.event.*;
 public class TwoListen implements MouseMotionListener, MouseListener
{
 private Frame f;
 private TextField tf;

 public go()
 {
  f=new Frame("Ejemplo de dos listeners");
  f.add(new Label ("Presiona y libera el mouse"), new
                   BorderLayout.NORTH);
  tf=new TextField(30);
  f.add(tf, BroderLayout.SOUTH);

  f.addMouseMotionListener(this);
  f.addMouseListener(this);
  f.setSize(300,200);
  f.setVisible(true);
 }

 //Estos son los eventos del MouseMotionListener
 public void mouseDragged (MouseEvent e)
 {
  String s = "Arrastre del Mouse: X="e.getX()+"Y="+e.gety();
  tf.setText (s);
 }

 public void mouseEntered (MouseEvent e)
 {
  String s= "El mouse entró";
  tf.setText(s);
 }

 public void mouseExited (MouseEvent e)
 {
  String s="El mouse ha salido";
  tf.setText(s);
 }
```

```
//Métodos de MouseMotionListener que no se utilizan pero que se tienen
//que agregar.
 public void mouseMoved (MouseEvent e)
 {}

 //Métodos de MouseListener que no se utilizan.
 public void mousePressed (MouseEvent e)
 {}

 public void mouseclicked (MouseEvent e)
 {}

 public void mouseReleased(MouseEvent e)
 {}

 public static void main (String arg[])
 {
  TwoListen two=new TwoListen();
  two.go();
 }
}
```

Laboratorio 8

1. Modificar la Calculadora del capítulo pasado para que reciba eventos.
   Ver Cap8\Calculadora.java

Capítulo 9 - Applets

Introducción a los Applets

Un applet es una clase de Java que corre en un navegador. Difiere de una aplicación en la manera en que es ejecutado. Una aplicación empieza cuando el método main() es llamado.

El ciclo de vida de un applet es un poco más complicado. No es iniciado tecleando un comando. Primero se tiene que crear un archivo html, donde se indica al navegador que tiene que cargar y cómo correrlo. El navegador primero carga el documento html y después carga y ejecuta el applet.

Como los applets contienen código que es cargado a través de una red, representan un peligro, porque alguien podría escribir una clase que leyera el password del usuario y lo mandaría de regreso por la red.

Java incluye la clase SecurityManager que controla el acceso a casi cualquier llamada a nivel de sistema en la Máquina Virtual de Java (JVM). Esto es llamado modelo de seguridad de "caja de arena" (la JVM provee un área (caja de arena) para que el applet "juegue tranquilamente", pero si intenta dejar esa área, será prevenido).

Para escribir un applet se debe crear una subclase de Applet:

```
import java.applet.*;
public class HelloWorldApplet extends Applet
```

El applet debe ser declarado como public y el nombre de la clase debe ser idéntico al nombre del archivo (HelloWorldApplet.java).

Método init() y método start()

En una aplicación, el programa es teclado en el método main(). En un applet, no es así. El primer código que un applet ejecuta es el que está definido para su inicialización y su constructor.

Cuando el constructor termina, el navegador llama al método init() del applet. Este método realiza una inicialización básica del applet. Después de que init() termina, el navegador llama al método start(). Estos dos métodos son ejecutados antes de que el applet quede "vivo", y es por esto que no pueden ser usados para realizar tareas complejas, como presentar mensajes o imágenes.

El método start() avisa que el applet ya está "vivo". Esto también se aplica cuando el navegador es restaurado después de ser minimizado, o cuando el navegador regresa a la página que contiene el applet. Esto significa que el applet puede usar este método para empezar una animación o tocar sonidos.

Método paint()

Los applets son esencialmente un ambiente gráfico, y llamadas a System.out.println pueden hacerse, pero lo que se hace es escribir en una pantalla en un ambiente gráfico.

El método paint() sirve para presentar textos e imágenes. Este método es llamado por el navegador cuando se necesita refrescar la vista del applet, por ejemplo, cuando el navegador es restaurado después de ser minimizado.

El método toma un parámetro que es una instancia de la clase java.awt.Graphics. Este parámetro siempre es el contexto gráfico del applet. A continuación se presenta el ejemplo de "hola mundo" hecho applet:

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

public class HelloWorldApplet extends Applet
{
 public void paint(Graphics g)
 {
  g.drawString("Hello World!",5,15);
 }
}
```

Los parámetros numéricos de drawString indican las coordenadas x,y. La posición x representa la parte más izquierda del texto. La posición y representa la parte más baja del texto, sin tomar en cuenta letras como g, j, q, que tienen una parte inferior más saliente.

Método stop()

Este método es llamado cuando el applet deja de "vivir". Es la contraparte del método start(). Esto ocurre cuando el navegador es cerrado, minimizado, o abre otra página. En este método se puede detener una animación o sonido.

```
public void stop()
{
 musica.stop();
}
```

Método repaint() y update()

Además del ciclo de vida del applet, hay métodos importantes relacionados con su despliegue en pantalla. Estos métodos están declarados y documentados en la clase java.awt.Component. La actualización del despliegue se realiza por un proceso concurrente, y ocurre en dos situaciones:

1. La exposición; cuando parte del applet ha sido dañado y debe ser reemplazado. Este daño puede ocurrir en cualquier momento y el applet debe ser capaz de resolver esta situación.
2. Cuando el programa "decide" redibujar el despliegue con nuevos elementos. Este redibujo puede requerir que la imagen anterior sea removida primero.

Como se vio, el método paint() es llamado automáticamente. Dentro del código del applet se puede avisar al sistema cuando redibujarlo llamando al método repaint(), puede no llevar parámetros, o parámetros que indiquen una área rectangular a actualizar.

El método repaint() por su declaración, llama al método update(). El método update() generalmente limpia el despliegue actual y llama a paint().

La etiqueta <applet>

Para incluir un applet en un documento html se usa la etiqueta <applet>. Por ejemplo, para ver el applet HelloWorldApplet, se tiene que crear el siguiente archivo, demoapplet.html:

```
<html>
<body>
<applet code=HelloWorldApplet.class width=100 height=50>
</applet>
```

```
</body>
</html>
```

A continuación se presenta la sintaxis completa de <applet>:

```
<applet
 code=applet.class
 width=pixels height=pixels
 [codebase=ruta]
 [alt=texto_alterno]
 [name=nombre_de_la_instancia_del_applet]
 [align=alineacion]
 [vspace=pixels] [hspace=pixels]
>
[<param name=variable1 value=valor>]
[<param name=variable2 value=valor>]
...
</applet>
```

code.- Este atributo se tiene que especificar. Contiene el nombre de archivo de la clase. No se puede incluir la ruta del archivo, para eso se declara el atributo codebase.

width/height.- Estos atributos se tienen que especificar y representan el tamaño en pixeles del despliegue del applet.

codebase.- Este atributo opcional representa la ruta del applet (directorios en donde se encuentra).

alt.- Este atributo opcional representa cualquier texto que sería desplegado si el navegador entiende la etiqueta <applet>.

name.- Este atributo opcional representa un nombre para la instancia del applet, que hace posible la comunicación entre applets en la misma página.

align.- Este atributo opcional representa la alineación del applet. Los valores posibles son los mismos que para la etiqueta <img> de html: left, right, top, texttop, middle, absmiddle, baseline, bottom, y absbottom.

vspace/hspace.- Estos atributos opcionales especifican el número de pixeles encima y debajo, y en cada lado del applet. Tienen la misma función que en la etiqueta <img> de html.

<param name/value>.- Esta etiqueta es la única forma de especificar variables y valores que usara el applet. El applet puede leer estas variables con el método getParameter().

Inclusión de imágenes

El siguiente código muestra como incluir una imagen en un applet.

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

public class Imagen extends Applet
{
 Image duke;

 public void init()
 {
  duke = getImage(getDocumentBase(),"duke.gif");
 }

 public void paint(Graphics g)
 {
  g.drawImage(duke,10,10,this);
 }
}
```

Los argumentos de drawImage son:

- la imagen a dibujar
- la coordenada en x
- la coordenada en y
- el observador de la imagen. Un observador de imagen es una clase que es notificada si la imagen cambia

getImage() regresa un objeto Image y tiene dos parámetros: el primero es getDocumentBase(), que regresa un objeto URL que describe la página actual del navegador; el segundo es el nombre del archivo que contiene una imagen, ya sea gif o jpg, además se puede incluir la ruta relativa a getDocumentBase().

Inclusión de sonidos

El lenguaje Java incluye métodos para tocar archivos de sonido. Estos métodos residen en la case java.applet.AudioClip. Se necesita el hardware apropiado para escuchar los archivos de sonido. La forma más fácil de escuchar un sonido es usando el método play:

```
play(URL directorioSonido, String archivoSonido);
```

o simplemente:

```
play(URL urlSonido);
```

Ejemplo de inclusión de sonido en un applet:

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

public class Sonido extends Applet
{
 public void paint(Graphics g)
 {
  g.drawString("Prueba de sonido",15,15);
  play(getDocumentBase(),"sonidos/cucu.au");
 }
}
```

Otra forma de tocar un archivo de sonido es creando una instancia de AudioClip:

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

Y una vez que el archivo es cargado, se usa cualquiera de los métodos siguientes para activarlo: play, loop, stop. Estos métodos residen en la clase java.applet.AudioClip.

`sonido.play();`
Hace que inicie el archivo de sonido.

`sonido.loop();`
Hace que se escuche el archivo de sonido una y otra vez.

`sonido.stop();`
Detiene el archivo de sonido.

Ejemplo para escuchar un archivo de sonido repetidamente:

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

public class SonidoLoop extends Applet
{
 AudioClip musica;

 public void init()
 {
  musica = getAudioClip(getDocumentBase(),"cancion.au");
 }

 public void start()
 {
  musica.loop();
 }

 public void stop()
 {
  musica.stop();
 }
}
```

Laboratorio 9

1. Transformar la Calculadora del capítulo anterior en un applet.
   Ver Cap9\CalcApplet.java

Contenido

Créditos

Universidad La Salle
Directorio:

H. Raúl Valadez García, FSC
Rector

Ing. Edmundo Barrera Monsivais
Vicerrector Académico

Hno. Martín Rocha Pedrajo
Vicerrector de Formación

Ing. José Antonio Torres Hernández
Director de la Escuela de Ingeniería

Ing. Raúl Morales Farfán
Secretario de Talleres y Laboratorios

Ing. Luis M. Aguillón Banda
Jefe del Laboratorio de Cómputo de Ingeniería

1ra. Edición
Realización

Jaime Conde Rojas
Líder de Proyecto

2da. Edición
Revisión Técnica y Actualización

José Luis Gutierrez Herrera
Líder de Proyecto

Revisión y Edición

Liliana Vicenteño Loya
Coordinadora Académica L.C.I.

Laboratorio de Cómputo de Ingeniería
MMII