

Tutorial

básico de Java

Versión 3.0

**Sólo una tecnología puede impulsar
todos estos dispositivos**



**Y este tutorial puede ayudarte a
dar tus primeros pasos en ella**

Copyright

Copyright (c) 2007, Abraham Otero. Este documento puede ser distribuido solo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en <http://www.javahispano.org/licencias/>).

Para cualquier duda, consulta, insulto o tirón de orejas sobre este tutorial dirigirse a abraham@javahispano.org.

INDICE:

1	Java, el LENGUAJE	5
1.1	INTRODUCCIÓN.....	5
1.2	Características de Java	6
1.2.1	Simple.....	6
1.2.2	Orientado a Objetos	6
1.1.1	Distribuido.....	6
1.1.2	Robusto.....	6
1.1.3	Seguro.....	6
1.1.4	Portable.....	7
1.1.5	Arquitectura Neutral.....	7
1.1.6	Rendimiento medio.....	7
1.1.7	Multithread	8
1.2	Java frente a los demás lenguajes.....	8
2	J2Sdk, Java 2 Standard Development Kit	9
2.1	Javac	9
2.2	Java	9
2.3	appletviewer	10
2.4	JAVADOC	10
3	tipos de datos PRIMITIVOS en Java.....	12
3.1	Tipos de datos	12
3.1.1	Enteros.....	12
3.1.2	Reales	13
3.1.3	Caracteres	13
3.1.4	Boolean.....	13
3.2	DEFINICIÓN de Variables	14
3.3	Conversión entre tipos numéricos.....	15
3.4	Operadores.....	16
3.4.1	Exponenciación	17
3.4.2	Operadores lógicos	18
3.5	Cadenas de caracteres.....	20
3.5.1	Concatenación	20
3.5.2	Subcadenas	21
3.5.3	Comparación de cadenas	22
3.6	Ámbito de las variables	23
3.7	ARRAYS	24
3.8	Tipo enumerados	25
3.9	Java no es perfecto.....	26
4	Control de flujo en Java	28
4.1	Sentencias Condicionales	28
4.1.1	If then Else.....	28
4.1.2	Switch.....	30
4.2	Bucles.....	32
4.2.1	Bucle while.....	32
4.2.2	Bucle do while.....	32
4.2.3	Bucle for	33
4.2.4	Bucle for-each	34

4.2.5	Break y continue	35
4.3	return.....	36
5	Objetos y clases	37
5.1	Introducción.....	37
5.2	Clases y herencia.....	38
5.2.1	Definición de una clase.....	39
5.2.2	Modificadores de métodos y variables	42
5.2.3	Herencia.....	45
5.2.4	Creación y referencia a objetos.....	50
5.2.5	this	52
5.2.6	super	53
5.3	Interfaces.....	53
5.4	Nuestro primer Programa orientado a objetos.....	57
5.5	Aprendiendo a usar los paquetes.....	66
5.5.1	Un ejemplo de código con paquetes	70
5.6	El ejemplo de los marcianos con paquetes.....	77
6	Programación gráfica con swing	85
6.1	JFrame.....	86
6.2	Eventos	87
6.2.1	¿Qué es un evento?	87
6.2.2	El modelo de delegación de eventos.....	88
6.2.3	Un frame que se cierra.....	92
6.3	JPanel	94
6.4	Layaout.....	96
6.4.1	FlowLayout.....	98
6.4.2	GridLayout	99
6.4.3	BorderLayout.....	101
6.5	JButton	103
6.6	Dibujar en una ventana.....	108
6.6.1	Empezando a dibujar	109
6.6.2	El método paintComponent	111
6.7	Revisión de algunos componentes de Swing	114
6.7.1	JTextField	114
6.7.2	JTextArea	114
6.7.3	JPasswordField	115
6.7.4	JScrollBar	115
6.7.5	JLabel	115
6.7.6	JCheckBox.....	116
6.7.7	JRadioButton	116
6.7.8	JList	116
6.7.9	JComboBox	117
6.7.10	JMenu	117
7	JApplet	118
7.1	Cómo convertir una aplicación en un applet.....	118
7.2	Ciclo de vida de un applet.....	121
8	Threads	122
8.1	¿qué es un thread?.....	122
8.2	La vida de un thread	123
8.2.1	Recien nacido:	123

8.2.2	Ejecutable:	124
8.2.3	Corriendo	124
8.2.4	Bloqueado.....	124
8.2.5	Muerto	124
8.3	Threads en java.....	125
8.4	Un programa sin threads	126
8.5	Un programa con threads	130
9	Apéndice A : mejoras al código de la guerra	142
10	Apéndice B: ¿USO IDE para aprender Java? ¿Cual?	143
10.1	BlueJ http://www.bluej.org/.....	143
10.2	JCreator http://www.jcreator.com/	144
10.3	JBuilder http://www.codegear.com/tabid/102/Default.aspx	144
10.4	NetBeans http://www.netbeans.org/.....	145
10.5	Eclipse http://www.eclipse.org/.....	145
11	Apéndice C: Convenios de nomenclatura en Java	147

1 Java, el LENGUAJE

1.1 INTRODUCCIÓN

Estamos en unos días en los que cada vez más la informática invade más campos de nuestra vida, estando el ciudadano medio cada vez más familiarizado con términos del mundo informático, entre ellos, como no, los lenguajes de programación. A cualquier persona que haya empleado alguna vez un ordenador le resultará familiar alguno de estos nombres: C, Pascal, Cobol, Visual Basic, Java, Fortran y a una persona ya más introducida en ese mundillo posiblemente haya oído muchos otros: Oak, Prolog, Dbase, JavaScrip, Delphi, Simula, Smalltalk, Modula, Oberon, Ada, BCPL, Common LISP, Scheme... En la actualidad se podrían recopilar del orden de varios cientos de lenguajes de programación distintos, sino miles.

Cabe hacerse una pregunta: ¿Para qué tanto lenguaje de programación?. Toda esta multitud de nombres puede confundir a cualquier no iniciado que haya decidido aprender un lenguaje, quien tras ver las posibles alternativas no sabe cual escoger, al menos entre los del primer grupo, que por ser más conocidos deben estar más extendidos.

El motivo de esta disparidad de lenguajes es que cada uno ha sido creado para una determinada función, está especialmente diseñado para facilitar la programación de un determinado tipo de problemas, para garantizar seguridad de las aplicaciones, para obtener una mayor facilidad de programación, para conseguir un mayor aprovechamiento de los recursos del ordenador... Estos objetivos son muchos de ellos excluyentes: el adaptar un lenguaje a un tipo de problemas hará más complicado abordar mediante este lenguaje la programación de otros problemas distintos de aquellos para los que fue diseñado. El facilitar el aprendizaje al programador disminuye el rendimiento y aprovechamiento de los recursos del ordenador por parte de las aplicaciones programadas en este lenguaje; mientras que primar el rendimiento y aprovechamiento de los recursos del ordenador dificulta la labor del programador.

La pregunta que viene a continuación es evidente: ¿Para qué fue pensado Java?. A esto responderemos en el siguiente apartado.

1.2 Características de Java

A continuación haremos una pequeña redacción de las características del lenguaje, que nos ayudarán a ver para que tipo de problemas está pensado Java:

1.2.1 Simple

Es un lenguaje sencillo de aprender. Su sintaxis es la de C++ "simplificada". Los creadores de Java partieron de la sintaxis de C++ y trataron de eliminar de este todo lo que resultase complicado o fuente de errores en este lenguaje. La herencia múltiple, la aritmética de punteros, por la gestión de memoria dinámica (que en Java se elimina de modo transparente para el programador gracias al recogedor basura) son ejemplos de "tareas complicadas" de C++ y en Java se han eliminado poco simplificado.

1.2.2 Orientado a Objetos

Posiblemente sea el lenguaje más orientado a objetos de todos los existentes; en Java todo, a excepción de los tipos fundamentales de variables (int, char, long...) es un objeto.

1.1.1 Distribuido

Java está muy orientado al trabajo en red, soportando protocolos como TCP/IP, UDP, HTTP y FTP. Por otro lado el uso de estos protocolos es bastante sencillo comparandolo con otros lenguajes que los soportan.

1.1.2 Robusto

El compilador Java detecta muchos errores que otros compiladores solo detectarían en tiempo de ejecución o incluso nunca. A esclarecer así por ejemplo " if(a=b) then ... " o " int i; h=i*2; " son dos ejemplos en los que el compilador Java no nos dejaría compilar este código; sin embargo un compilador C compilaría el código y generaría un ejecutable que ejecutaría esta sentencia sin dar ningún tipo de error).

1.1.3 Seguro

Sobre todo un tipo de desarrollo: los Applet. Estos son programas diseñados para ser ejecutados en una página web. Java garantiza que ningún Applet puede escribir o leer de nuestro disco o mandar información del usuario que accede a la página a través de la red (como, por ejemplo, la dirección de correo electrónico). En general no permite realizar cualquier acción que pudiera dañar la máquina o violar la intimidad del que visita la página web.

1.1.4 Portable

En Java no hay aspectos dependientes de la implementación, todas las implementaciones de Java siguen los mismos estándares en cuanto a tamaño y almacenamiento de los datos.

Esto no ocurre así en C++, por ejemplo. En éste un entero, por ejemplo, puede tener un tamaño de 16, 32 o más bits, siendo lo única limitación que el entero sea mayor que un short y menor que un long int. Así mismo C++ bajo UNIX almacena los datos en formato little endian, mientras que bajo Windows lo hace en big endian. Java lo hace siempre en little edian para evitar confusiones.

1.1.5 Arquitectura Neutral

El código generado por el compilador Java es independiente de la arquitectura: podría ejecutarse en un entorno UNIX, Mac o Windows. El motivo de esto es que el que realmente ejecuta el código generado por el compilador no es el procesador del ordenador directamente, sino que este se ejecuta mediante una máquina virtual. Esto permite que los Applets de una web pueda ejecutarlos cualquier máquina que se conecte a ella independientemente de que sistema operativo emplee (siempre y cuando el ordenador en cuestión tenga instalada una máquina virtual de Java).

1.1.6 Rendimiento medio

Actualmente la velocidad de procesado del código Java es semejante a la de C++, hay ciertos pruebas estándares de comparación (benchmarks) en las que Java gana a C++ y viceversa. Esto es así gracias al uso de compiladores *just in time*, compiladores que traduce los bytecodes de Java en código para una determinada CPU, que no precisa de la máquina virtual para ser ejecutado, y guardan el resultado de dicha conversión, volviéndolo a llamar en caso de volverlo a necesitar, con lo que se evita la sobrecarga de trabajo asociada a la interpretación del bytecode.

No obstante por norma general el programa Java consume bastante más memoria que el programa C++, ya que no sólo ha de cargar en memoria los recursos necesario para la ejecución del programa, sino que además debe simular un sistema operativo y hardware virtuales (la máquina virtual). Por otro lado la programación gráfica empleando las librerías Swing es más lenta que el uso de componentes nativos en las interfaces de usuario.

En general en Java se ha sacrificado el rendimiento para facilitar la programación y sobre todo para conseguir la característica de neutralidad arquitectural, si bien es cierto que los avances en las máquinas virtuales remedian cada vez más estas decisiones de diseño.

1.1.7 Multithread

Soporta de modo nativo los threads, sin necesidad del uso de de librerías específicas (como es el caso de C++). Esto le permite además que cada Thread de una aplicación java pueda correr en una CPU distinta, si la aplicación se ejecuta en una máquina que posee varias CPU. Las aplicaciones de C++ no son capaces de distribuir, de modo transparente para el programador, la carga entre varias CPU.

1.2 JAVA FRENTE A LOS DEMÁS LENGUAJES

Java es un lenguaje relativamente moderno. Su primer uso en una “tarea seria” de programación fue la construcción del navegador HotJava por parte de la empresa Sun en mayo de 1995, y fue a principios de 1996 cuando Sun distribuye la primera versión de Java. Es esta corta edad lo que hace que Java esté más orientado al mundo web, que no existía cuando, por ejemplo, C fue desarrollado.

También es esto lo que ha hecho que soporte de modo nativo (no mediante el uso de librerías, como C) los threads, siendo posible aprovechar las ventajas de los sistemas multiprocesadores.

Las ventajas fundamentales de Java frente a otros lenguajes son el menor periodo de aprendizaje por parte del programador, llegando a ser un programador productivo en menos tiempo (sencillez) y siendo posible desarrollar aplicaciones más rápido que en otros lenguajes (sencillez y robustez), lo cual se traduce en el mundo empresarial en un ahorro de costes.

Sus cualidades de distribuido, seguro e independencia de la plataforma lo hacen ideal para aplicaciones relacionadas con el mundo web; precisamente a esto es a lo que Java debe su gran difusión y fama. El hecho de que sea independiente de la máquina y del sistema operativo permite que distintas máquinas con distintos sistemas operativos se conecten a una misma página web y ejecuten los mismos applets. Además la seguridad que garantiza Java para los applets impiden que alguien trate de averiguar información sobre los usuarios que se conectan a la página web o intente dañar sus máquinas.

En cuanto a su capacidad de soporte de threads y su capacidad de sacarle partido a sistemas multiprocesador lo convierten en un lenguaje más “orientado hacia el futuro “. Estas cualidades podrían dar pie a que algún día los rendimientos computacionales de Java sean comparables con los de C++ y otros lenguajes que hoy son computacionalmente más eficientes.

2 J2SDK, JAVA 2 STANDARD DEVELOPMENT KIT

En este capítulo haremos un breve repaso del entorno de programación distribuido por Sun, jdk. Dicho entorno de programación es suministrado por Sun de forma gratuita, pudiéndose encontrar en la dirección web: <http://Java.sun.com/j2se/>.

Es de consenso que el entorno jdk no es el más adecuado para el desarrollo de aplicaciones Java, debido a funcionar única y exclusivamente mediante comandos de consola, ya que hoy en día la programación se suele ayudar de entornos visuales, como JBuilder, JCreator o muchos otros, que facilitan enormemente la tarea (ver apéndice B). Sin embargo, puede ser un entorno bastante útil para aprender el lenguaje, ya que aunque los entornos visuales nos hagan mucho trabajo siempre es necesario ir al código para modificarlo y obtener el comportamiento deseado, lo cual quiere decir que necesitamos dominar el lenguaje y es más fácil llegar a este dominio escribiendo códigos completos en un entorno "hostil" que no nos ayuda, que simplemente remodelando códigos ya generados por entornos visuales.

2.1 JAVAC

Es el comando compilador de Java. Su sintaxis es:

```
| javac ejemplo.java
```

La entrada de este comando ha de ser necesariamente un fichero que contenga código escrito en lenguaje Java y con extensión .Java. El comando nos creará un fichero .class por cada clase que contenga el fichero Java (en el tema 5 se explicará qué es una clase).

Los ficheros .class contienen código bytecode, el código que es interpretado por la máquina virtual Java.

2.2 JAVA

Es el intérprete de Java. Permite ejecutar aplicaciones que previamente hayan sido compiladas y transformadas en ficheros .class. Su sintaxis es:

```
| java ejemplo
```

No es necesario aquí suministrar la extensión del fichero, ya que siempre ha de ser un fichero .class.

2.3 APPLETVIEWER

Se trata de un comando que verifica el comportamiento de un applet. La entrada del comando ha de ser una página web que contenga una referencia al applet que deseamos probar. Su sintaxis es:

```
| Appletviewer mipagina.html
```

El comando ignora todo el contenido de la página web que no sean applets y se limita a ejecutarlos. Un ejemplo de página web “mínima” para poder probar un applet llamado myapplet.class sería:

```
| <HTML>
  <TITLE>My Applet </TITLE>
  <BODY>
    <APPLET CODE="myapplet.class" WIDTH=180 HEIGHT=180>
  </APPLET>
  </BODY>
</HTML>
```

2.4 JAVADOC

Este útil comando permite generar documentación en formato html sobre el contenido de ficheros con extensión .Java. Su sintaxis es:

```
| javadoc ejemplo.java
```

En la documentación generada por este comando se puede ver que métodos y constructores posee una determinada clase, junto con comentarios sobre su uso, si posee inner classes, la versión y el autor de la clase....

Una explicación detallada de la sintaxis de los comentarios de javadoc, que aquí no abordaremos, se encuentra el capítulo 2 página 122 del libro Thinking in Java de Bruce Eckel (<http://www.bruceeckel.com/>).

A continuación suministramos el código Java de un ejemplo

```
//: c02:HelloDate.Java
import java.util.*;

/**Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
 */
public class HelloDate {
    /** Sole entry point to class & application
     * @param args array of string arguments
     * @return No return value
     * @exception exceptions No exceptions thrown
     */
    public static void main(String[] args) {
        //Esta línea imprime por consola la cadena de caracteres
        //"Hello it's"
        System.out.println("Hello, it's: ");
        //Esta sentencia imprime la fecha actual del equipo
        System.out.println(new Date());
    }
} ///:~
```

3 TIPOS DE DATOS PRIMITIVOS EN JAVA

En este tema trataremos las estructuras básicas de Java de datos, sus tipos y las variables, operadores. Aquellos que estén familiarizados con C, o C++, no encontrarán prácticamente nada nuevo en este tema, ya que, como se ha dicho en el primer tema, Java hereda toda su sintaxis de C, pudiendo considerarse la sintaxis de Java una versión simplificada de la de C. Sólo las operaciones con Strings pueden resultar un poco novedosas.

3.1 TIPOS DE DATOS

En Java toda variable declarada ha de tener su tipo, y además antes de poder emplearla hemos de inicializarla a un valor, si no es compilador se quejará y no generará los archivos .class. Esto por ejemplo en C no es necesario, siendo fuente de muchos errores al emplearse en operaciones variables que nos hemos olvidado de inicializar. A continuación pasamos a describir los tipos de datos:

3.1.1 Enteros

Almacenan como su propio nombre indica números enteros, sin parte decimal. Cabe destacar, como ya se indicó en el primer tema, que por razones de portabilidad todos los datos en Java tienen el mismo tamaño y formato. En Java hay cuatro tipos de enteros:

Tabla 1: tipo de datos enteros en Java

Tipo	Tamaño (bytes)	Rango
Byte	1	-128 a 127
Short	2	-32768 a 32767
Int	4	-2147483648 a 2147483647
Long	8	-9223372036854775808 a 9223372036854775807

Para indicar que una constante es de tipo long lo indicaremos con una L: 23L.

3.1.2 Reales

Almacenan número reales, es decir números con parte fraccionaria. Hay dos tipos:

Tabla 2: tipos de datos reales en Java

Tipo	Tamaño (bytes)	Rango
Float	4	+ 3.40282347E+38
Double	8	+ 179769313486231570E+308

Si queremos indicar que una constante es flotante: ej: 2.3 hemos de escribir: 2.3F, sino por defecto será double.

3.1.3 Caracteres

En Java hay un único tipo de carácter: char. Cada carácter en Java esta codificado en un formato denominado Unicode, en este formato cada carácter ocupa dos bytes, frente a la codificación en ASCII, dónde cada carácter ocupaba un solo byte.

Unicode es una extensión de ASCII, ya que éste último al emplear un byte por carácter sólo daba acogida a 256 símbolos distintos. Para poder aceptar todos los alfabetos (chino, japonés, ruso...) y una mayor cantidad de símbolos se creó el formato Unicode.

En Java al igual que en C se distingue la representación de los datos char frente a las cadenas de caracteres. Los char van entre comillas simples: char ch = 'a', mientras que las cadenas de caracteres usan comillas dobles.

3.1.4 Boolean

Se trata de un tipo de dato que solo puede tomar dos valores: "true" y "false". Es un tipo de dato bastante útil a la hora de realizar chequeos sobre condiciones. En C no hay un dato equivalente y para suplir su ausencia muchas veces se emplean enteros con valor 1 si "true" y 0 si "false". Otros lenguajes como Pascal sí tiene este tipo de dato.

3.2 DEFINICIÓN DE VARIABLES

Al igual que C, Java requiere que se declaren los tipos de todas las variables empleadas. La sintaxis de iniciación es la misma que C:

```
|         int i;
```

Sin embargo, y a diferencia que en C, se requiere inicializar todas las variables antes de usarlas, si no el compilador genera un error y aborta la compilación. Se puede inicializar y asignar valor a una variable en una misma línea:

```
|         int i = 0;
```

Asignación e inicialización pueden hacerse en líneas diferentes:

```
|         int i ;  
|         i = 0;
```

Es posible iniciar varias variables en una línea:

```
|         int i, j,k;
```

Después de cada línea de código, bien sea de iniciación o de código, al igual que en C va un ;.

En Java, al igual que en todo lenguaje de programación hay una serie de palabras reservadas que no pueden ser empleadas como nombres de variables (if, int, char, else, goto....); alguna de estas se emplean en la sintaxis del lenguaje, otras, como goto no se emplean en la actualidad pero se han reservado por motivos de compatibilidad por si se emplean en el futuro.

Los caracteres aceptados en el nombre de una variable son los comprendidos entre "A-Z", "a-z", "_", "\$ y cualquier carácter que sea una letra en algún idioma.

3.3 CONVERSIÓN ENTRE TIPOS NUMÉRICOS

Las normas de conversión entre tipos numéricos son las habituales en un lenguaje de programación: si en una operación se involucran varios datos numéricos de distintos tipos todos ellos se convierten al tipo de dato que permite una mayor precisión y rango de representación numérica; así, por ejemplo:

- Si cualquier operando es double todos se convertirán en double.
- Si cualquier operando es float y no hay ningún double todos se convertirán a float.
- Si cualquier operando es long y no hay datos reales todos se convertirán en long.
- Si cualquier operando es int y no hay datos reales todos ni long se convertirán en int.
- En cualquier otro caso el resultado será también un int.

Java solo tiene dos tipos de operadores enteros: uno que aplica para operar datos de tipo long, y otro que emplea para operar datos de tipo int. De este modo cuando operemos un byte con un byte, un short con un short o un short con un byte Java empleará para dicha operación el operador de los datos tipo int, por lo que el resultado de dicha operación será un int siempre.

La “jerarquía” en las conversiones de mayor a menor es:

double <- float <- long <- int <- short <- byte

Estas conversiones sólo nos preocuparán a la hora de mirar en que tipo de variable guardamos el resultado de la operación; esta ha de ser de una jerarquía mayor o igual a la jerarquía de la máxima variable involucrada en la operación. Si es de rango superior no habrá problemas.

Es posible convertir un dato de jerarquía “superior” a uno con jerarquía “inferior”, arriesgándonos a perder información en el cambio. Este tipo de operación (almacenar el contenido de una variable de jerarquía superior en una de jerarquía inferior) se denomina cast. Veamos una ejemplo para comprender su sintaxis:

```
public class Ejemplo1 {
    public static void main(String[] args) {
        int i = 9,k;
        float j = 47.9F;
        System.out.println("i: "+i + " j: " +j);
        k = (int)j; //empleo de un cast
        System.out.println("j: " + j + " k: " +k);
        j = k;//no necesita cast
        System.out.println("j: " + j + " k: " +k);
        float m = 2.3F;
        //float m = 2.3; daría error al compilar.
        System.out.println("m: "+m);
    }
} ///:~
```

3.4 OPERADORES

Los operadores básicos de Java son + , - , * , / para suma, resta, producto y división. El operador / representa la división de enteros si ambos operandos son enteros. Su módulo puede obtenerse mediante el operador % (7/4= 1; 7% 4=3).

Además existen los operadores decremento e incremento: -- y ++ respectivamente. La operación que realizan son incrementar y decrementar en una unidad a la variable a la que se aplican. Su acción es distinta según se apliquen antes (++a) o después (a++) de la variable. El siguiente programa ilustra estos distintos efectos:

```
public class Ejemplo2{
    public static void main(String[] args) {
        int i = 1;
        prt("i : " + i);
        prt("++i : " + ++i); // Pre-incremento, primero
        //incrementa y luego imprime por consola
        prt("i++ : " + i++); // Post-incremento, primero imprime
        //"2" por consola y luego incrementa i.
        prt("i : " + i);//i por lo tanto vale 3
    }
}
```



```
        prt("--i : " + --i); // Pre-decremento, primero
//decrementa i y luego lo imprime por consola
        prt("i-- : " + i--); // Post-decremento, primero imprime
//i por consola y luego de decrementa.
        prt("i : " + i); //Ahora i vale 1
    }
//esto nos ahorra teclear. Invocando a prt podremos
//imprimir por consola la cadena de caracteres que le pasemos
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

3.4.1 Exponenciación

En Java a diferencia que en otros lenguajes no existe el operador exponenciación. Tampoco existen los operadores Seno o Coseno. Si se desea realizar una operación de exponenciación se deberá invocar el método correspondiente de la clase Math de Java.lang. Estos métodos son estáticos, por lo que no es necesario crear un objeto de dicha clase. Su sintaxis general es:

```
Math.metodo(argumentos);
```

En el siguiente código aparecen ejemplos. Si alguna vez deseamos realizar algún otro tipo de operación y queremos ver si la soporta Java podemos hacerlo consultando la ayuda on-line de la clase Math, o bien la documentación que podemos descargar los desde la misma página que el jdk.

```
public class Ejemplo3 {
    public static void main(String[] args) {
        int i = 45, j=2;
//Imprime por consola la cadena de caracteres "Cos i : "
//concatenado con el resultado de calcular el coseno de i
        prt("Cos i : " + Math.cos(i));
        prt("Sen i : " + Math.sin(i));
    }
}
```

```
        prt("j^i : " + Math.pow(j,i));

    }

    //esto nos ahorrara teclear
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

3.4.2 Operadores lógicos

En la tabla a continuación recogemos los operadores lógicos disponibles en Java:

Tabla 3: operadores lógicos

Operador	Operación que realiza
!	Not lógico
==	Test de igualdad
!=	Test de desigualdad
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
&&	And lógico
	Or lógico

Con el siguiente ejemplo se muestra la función de estos operadores:

```
import java.util.*;

public class Ejemplo4 {
    public static void main(String[] args) {
        //Creamos un objeto de tipo Random, almacenado un puntero a
        //el en la variable rand. En el tema 5 se detallará como se
        //crean objetos.
        Random rand = new Random();
        //el método nextInt() del objeto Random creado (se invoca
        //como rand.nextInt()) genera un número aleatorio entero. En
        //el tema 5 se explica que son los métodos y como emplearlos.
        //El módulo 100 de un entero aleatorio será un entero
        //aleatorio entre 0 y 100.
        int i = rand.nextInt() % 100;
        int j = rand.nextInt() % 100;
        //Imprime i y j por consola
        prt("i = " + i);
        prt("j = " + j);
        //Imprime diversas operaciones binarias sobre i y j, junto
        //con su resultado.
        prt("i > j es " + (i > j));
        prt("i < j es " + (i < j));
        prt("i >= j es " + (i >= j));
        prt("i <= j es " + (i <= j));
        prt("i == j es " + (i == j));
        prt("i != j es " + (i != j));
        prt("(i < 10) && (j < 10) es "
            + ((i < 10) && (j < 10)) );
        prt("(i < 10) || (j < 10) es "
            + ((i < 10) || (j < 10)) );
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

Una posible salida de este programa es:

```
i = 85
j = 4
i > j es true
i < j es false
i >= j es true
i <= j es false
i == j es false
i != j es true
//true && false = false
(i < 10) && (j < 10) es false
//true || false = true
(i < 10) || (j < 10) es true
```

3.5 CADENAS DE CARACTERES

En Java no hay un tipo predefinido para cadenas de caracteres, en su lugar hay una clase, String, que es la que soporta las distintas operaciones con cadenas de caracteres. La definición de un String es:

```
String e ;    //no inicializado
String e = ""; //cadena vacía
String e = "Hola"; //inicialización y asignación juntas.
```

A continuación veremos algunas operaciones básicas soportadas por la clase String:

3.5.1 Concatenación

La concatenación en Java es increíblemente sencilla: se realiza con el operador +, es decir "sumando" cadenas de caracteres obtenemos la concatenación de estas. Lo ilustraremos con un ejemplo:

```
String saludo = "hola";
String nombre = "Pepe";
String saluda_pepe = "";
```

```
saluda_pepe = saludo + nombre;// saluda_pepe toma el valor  
holaPepe
```

La sencillez de Java en el manejo de cadenas de caracteres llega incluso más allá: si una cadena la intentamos encadenar con otro tipo de variable automáticamente se convierte la otra variable a String, de tal modo que es perfectamente correcto:

```
String saludo = "hola";  
int n = 5;  
saludo = saludo + " " + n;// saludo toma el valor "hola 5"
```

3.5.2 Subcadenas

En la clase String hay un método que permite la extracción de una subcadena de caracteres de otra. Su sintaxis es:

```
Nombre_String.substring((int)posición_inicial,(int)posición_f  
inal);
```

Donde posición_inicial y posición_final son respectivamente la posición del primer carácter que se desea extraer y del primer carácter que ya no se desea extraer.

```
String saludo = "hola";  
String subsaludo = "";  
Subsaludo = saludo.substring(0,2);// subsaludo toma el  
valor "ho"
```

Puede extraerse un char de una cadena, para ello se emplea el método charAt(posición), siendo posición la posición del carácter que se desea extraer.

3.5.3 Comparación de cadenas

Se empleo otro método de String: equals. Su sintaxis es:

```
| cadena1.equals(cadena2);
```

Devuelve true si son iguales y false si son distintos.

El siguiente ejemplo permitirá ilustrar estas operaciones con Strings:

```
public class Ejemplo5 {
    public static void main(String[] args) {
        String saludo = "Hola";
        String saludo2 ="hola";
        int n = 5;
        //Imprime por consola la subcadena formada por los caracteres
        //comprendidos entre el caracter 0 de saludo y hasta el
        //carácter 2, sin incluir el último
        prt(saludo.substring(0,2));
        //Concatena saludo con un espacio en blanco y con el valor de
        //la variable n
        prt(saludo +" " + n);
        //Imprime el resultado del test de igualdad entre saludo y
        //saludo2. Son distintos, en Java se distingue entre
        //mayúsculas y minúsculas.
        prt("saludo == saludo2 "+ saludo.equals(saludo2));
    }
    static void prt(String s) {
        System.out.println(s);
    }
}
```

3.6 ÁMBITO DE LAS VARIABLES

En este apartado vamos a tratar de ver cual es el ámbito de validez de una variable. Éste en Java viene dado por los corchetes: {}; una vez definida una variable en un código dejará de existir cuando se acabe el bloque de código en el que se definió. Los bloques de código empiezan con "{" y acaban en "}", por lo que la variable dejará de existir cuando se cierre el corchete que esté justo antes que ella en el código. Veámoslo con un ejemplo:

```
{
    int x = 12;
    /* solo x disponible */
    {
        int q = 96;
        /* x y q disponible */
    }
    /* solo x disponible */
    /* q "fuera de ámbito" */
}
```

Por otro lado si intentamos hacer lo siguiente:

```
{
    int x = 12;
    {
        int x = 96; /* ilegal en Java, no en C++ */
    }
}
```

El compilador se nos quejará diciendo que la variable x ya ha sido definida. En C++ esto si es posible, pero los diseñadores de Java decidieron no permitir este tipo de construcciones a para lograr más claridad en el código.

3.7 ARRAYS

En Java los arrays son un objeto. Como tales se crean mediante el comando new (se verá su uso en el tema 5). La sintaxis en la definición de un array es la siguiente:

```
Tipo_datos[] nombre_array = new  
Tipo_datos[tamaño_array];
```

Tipo_datos es el tipo de los datos que se almacenarán en el array (int, char, String... o cualquier objeto). Tamaño_array es tamaño que le queremos dar a este array. Veamos un ejemplo:

```
int[] edades = new int[10];
```

En este ejemplo hemos definido un array llamado edades, en el que podremos almacenar 10 datos tipo entero. El primer elemento de un array se sitúa en la posición 0, exactamente igual que en C. Si quisiésemos realizar un bucle que recorriese los elementos de este array escribiríamos un código del tipo:

```
public class Ejemplo5b {  
    public static void main(String[] args) {  
        int[] edades = new int[10];  
        for(int i= 0; i< 10; i++){  
            edades[i] = i;  
            System.out.println("Elemento " + i + edades[i]);  
        }  
        int sum = 0;  
        for(int i= 0; i< 10; i++){  
            sum = sum + edades[i];  
        }  
        System.out.println("Suma " + sum);  
    }  
}
```


3.8 TIPOS ENUMERADOS

Esta característica del lenguaje sólo está disponible en Java 5 y superior. Si estás empleando Java 1.4.X o inferior no podrás emplearla. Cuando escribo este tutorial la versión actual de Java es la 6.

Los tipos de datos enumerados son un tipo de dato definido por el programador (no como ocurre con los tipos de datos primitivos). En su definición el programador debe indicar un conjunto de valores finitos sobre los cuales las variables de tipo enumeración deberán tomar valores. La principal funcionalidad de los tipos de datos enumerados es incrementar la legibilidad del programa. La mejor forma de comprender lo qué son es viéndolo; para definir un tipo de dato enumerado se emplea la sintaxis:

```
| modificadores enum NombreTipoEnumerado{ VALOR1,VALOR2,... }
```

Los posibles valores de "modificadores" serán vistos en el tema 5. El caso más habitual es que modificadores tome el valor "public". El nombre puede ser cualquier nombre válido dentro de Java. Entre las llaves se ponen los posibles valores que podrán tomar las variables de tipo enumeración, valores que habitualmente se escriben en letras mayúsculas. Un ejemplo de enumeración podría ser:

```
| public enum Semana {LUNES, MARTES, MIÉRCOLES, JUEVES,  
| VIERNES, SÁBADO, DOMINGO}
```

Las definiciones de los tipos enumerados deben realizarse fuera del método main y, en general, fuera de cualquier método; es decir, deben realizarse directamente dentro del cuerpo de la clase. En el tema 5 se explicará detalladamente qué es una clase y qué es un método.

Para definir una variable de la anterior enumeración se emplearía la siguiente sintaxis:

```
| Semana variable;
```

y para darle un valor a las variables de tipo enumeración éstas deben asignarse a uno de los valores creados en su definición. El nombre del valor debe ir precedido del nombre de la propia enumeración:

```
| variable = Semana.DOMINGO;
```

Veamos un ejemplo de programa que emplea enumeraciones:

```
public class Ejemplo5c {
    //definimos un tipo enumerado
    //los tipos enumerados deben definirse siempre fuera
    //del main y, más en general, fuera de cualquier método
    public enum Semana {LUNES, MARTES, MIERCOLES, JUEVES,
VIERNES, SABADO, DOMINGO};
    public static void main(String[] args) {
        //definimos una variable que pertenece al tipo enumerado
        Semana
            //y le damos el valor que representa el día martes
            Semana hoy = Semana.MARTES;
        //si el día se cayese en el fin de semana no hay que trabajar
        //obs!rvase como gracias a la numeración del programa es
        facil del entender
        if(hoy == Semana.DOMINGO || hoy == Semana.SABADO){
            System.out.println("Hoy toca descansar");
        } else{
            System.out.println("Hoy toca trabajar");
        }
    }
}
```

3.9 JAVA NO ES PERFECTO

Ya hemos hecho hincapié en que Java esta diseñado para facilitar la labor del programador, disminuyendo los tiempos de aprendizaje y de programación gracias a su sencillez y a su capacidad de detectar errores; sin embargo, Java también tiene sus defectos: Java deja hacer overflow sin dar error ni en tiempo de compilación ni tan siquiera en tiempo de ejecución. Aquí tenéis la prueba:

```
public class ejemplo6 {
    public static void main(String[] args) {
        //0x significa que el número está en formato hexadecimal.
        // 0x7 = 0111 en binario
        // 0xf = 1111 en binario
        //El numero representado abajo será "0" + 31 "1". El primer
        //bit es el signo, al ser 0 indica que es un número
        //positivo. Es por lo tanto el numero entero (un int son 32
        //bits) más grande que podemos representar en Java
        int gran = 0x7fffffff; // maximo valor int
        prt("gran = " + gran);
        //Forzamos un overflow:
        int mas_grande = gran * 4;
        //No se produce una excepción y se ejecuta la siguiente línea
        prt("mas_grande = " + mas_grande);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:
```

4 CONTROL DE FLUJO EN JAVA

El modo de ejecución de un programa en Java en ausencia de elementos de control de flujo es secuencial, es decir una instrucción se ejecuta detrás de otra y sólo se ejecuta una vez. Esto nos permite hacer programas muy limitados; para evitarlo se introducen estructuras de control de flujo.

Las estructuras de control de flujo de Java son las típicas de cualquier lenguaje de programación, por lo que supondremos que todos estáis familiarizados con ellas y se explicaran con poco detenimiento.

4.1 SENTENCIAS CONDICIONALES

Ejecutan un código u otro en función de que se cumplan o no una determinada condición. Pasemos a ver sus principales tipos.

4.1.1 *If then Else*

Su modo más simple de empleo es:

```
|      If(condicion) {  
|          Grupo de sentencias}
```

Condición es un valor tipo boolean. El grupo de sentencias se ejecuta solo si la condición toma un valor true. En caso contrario se sigue ejecutando ignorando el Grupo de sentencias.

```
|      If(condicion) {  
|          Grupo de sentencias}  
|      else{  
|          Grupo2 de sentencias}
```

Si condición toma el valor true se ejecuta Grupo de sentencias, en caso contrario se ejecuta Grupo2 de sentencias. En ambos casos se continúa ejecutando el resto del código.

```
If(condicion) {  
    Grupo de sentencias}  
else if (condicion2){  
    Grupo2 de sentencias}  
else if (condicion3){  
    Grupo3 de sentencias}  
.  
.  
.  
else{  
    Grupo_n de sentencias}
```

Si condición toma el valor true se ejecuta Grupo de sentencias, si condicion2 toma el valor true se ejecuta Grupo2 de sentencias... y así sucesivamente hasta acabarse todas las condiciones. Si no se cumple ninguna se ejecuta Grupo_n de sentencias. Este último else es opcional. En ambos casos se continúa ejecutando el resto del código.

Ilustraremos esto con el siguiente ejemplo:

```
public class Ejemplo7 {  
    // Método que podremos invocar como test(int a, int b) y que  
    // devolverá -1 si a < b, +1 si a > b y 0 si a == b.  
    static int test(int val, int val2) {  
        int result = 0;  
        if(val > val2)  
            result = +1;  
        else if(val < val2)  
            result = -1;  
        else  
            result = 0;  
        return result;  
    }  
    public static void main(String[] args) {  
        //Imprimimos por consola el resultado de realizar unos  
        //cuantos test.
```

```
        System.out.println(test(10, 5));  
        System.out.println(test(5, 10));  
        System.out.println(test(5, 5));  
    }  
} ///:~
```

4.1.2 Switch

Los creadores de Java trataron de hacer de este lenguaje una versión simplificada y mejorada del lenguaje de C++. Su trabajo fue bastante bueno, pero no perfecto. Prueba de ello es esta sentencia: está tan poco flexible como en C++.

Explicemos su sintaxis antes de dar los motivos de esta crítica:

```
switch(selector) {  
    case valor1 : Grupo de sentencias1; break;  
    case valor2 : Grupo de sentencias2; break;  
    case valor3 : Grupo de sentencias3; break;  
    case valor4 : Grupo de sentencias4; break;  
    case valor5 : Grupo de sentencias5; break;  
                // ...  
    default: statement;  
}
```

Se compara el valor de selector con sentencias_n. Si el valor coincide se ejecuta su respectivo grupo de secuencias. Si no se encuentra ninguna coincidencia se ejecutan las sentencias de default. Si no se pusieran los break una vez que se encontrase un valor que coincidiera con el selector se ejecutarían todos los grupos de sentencias, incluida la del default.

Ha llegado el momento de justificar la crítica hecha a los creadores de Java. Este tipo de estructura tiene sus posibilidades muy limitadas, ya que en las condiciones sólo se admite la igualdad, no ningún otro tipo de condición (sería fácil pensar ejemplos dónde, por poner un caso, se le sacaría partido a esta sentencia si aceptase desigualdades). Además para colmo esta comparación de igualdad sólo admite valores tipo char o cualquier tipo de valores enteros menos long (si estás empleando Java 5 o posterior también se puede emplear un tipo enumerado). No podemos comparar contra reales, Strings....

También se le podría criticar el hecho de que una vez cumplidas una condición se ejecuten todas las sentencias si no hay instrucciones break que lo impidan. Esto es en muchas ocasiones fuente de errores, aunque también hay que reconocer que a veces se le puede sacar

partido, de hecho en el ejemplo que empleamos para ilustrar esta sentencia aprovechamos esta característica:

```
public class Ejemplo8 {
    public static void main(String[] args) {
        //Bucle for. Ejecutará 100 veces el código que tiene dentro.
        for(int i = 0; i < 100; i++) {
            //Math.random() es un método estático que genera un número real
            //aleatorio entre 0 y 1.
            //Math.random()*26 será un número real aleatorio entre 0 y
            //26. Al sumarle un carácter, 'a' el carácter se transforma a
            //un entero y se le suma. 'a' = 97.
            //Se transforma el número aleatorio entre 97 y 97 + 26 en el
            //carácter correspondiente a su parte entera. Será un carácter
            //aleatorio, que por la disposición de los caracteres Unicode
            //será un letra del abecedario.
            char c = (char)(Math.random() * 26 + 'a');
            System.out.print(c + ": ");
            switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    //Si el carácter es 'a', 'e', 'i', 'o' o 'u' imprimimos
                    //vocal.
                    System.out.println("vocal");
                    break;
                default:
                    //Si no era ninguna de las anteriores imprimimos consonante.
                    System.out.println("consonante");
            }
        }
    }
}
```

4.2 BUCLES

Son instrucciones que nos permiten repetir un bloque de código mientras se cumpla una determinada condición. Pasemos a ver sus tipos.

4.2.1 Bucle *while*

Cuando en la ejecución de un código se llega a un bucle *while* se comprueba si se verifica su condición, si se verifica se continua ejecutando el código del bucle hasta que esta deje de verificarse. Su sintaxis es:

```
while(condición){  
    Grupo de sentencias}
```

Ilustramos su funcionamiento con un ejemplo:

```
public class Ejemplo9 {  
    public static void main(String[] args) {  
        double r = 0;  
        //Mientras que r < 0.99 sigue ejecutando el cuerpo del bucle.  
        //La d significa double. No es necesaria  
        while(r < 0.99d) {  
            //Genera un nuevo r aleatorio entr 0 y 1.  
            r = Math.random();  
            //Lo imprime por consola.  
            System.out.println(r);  
        }  
    }  
} ///:~
```

4.2.2 Bucle *do while*

Su comportamiento es semejante al bucle *while*, sólo que aquí la condición va al final del código del bucle, por lo que tenemos garantizado que el código se va a ejecutar al menos una vez. Dependerá del caso concreto si es más conveniente emplear un bucle *while* o *do while*. La sintaxis de *do while* es:


```
do {  
    Grupo de sentencias;  
}while(condición);
```

Obsérvese como el ejemplo 9 implementado mediante un bucle do while independientemente del valor de r ejecutará al menos una vez el código de su cuerpo:

```
public class Ejemplo10 {  
    public static void main(String[] args) {  
        double r;  
        //Idéntico al ejemplo anterior, solo que aahora la condición  
        //está al final del bucle.  
        do {  
            r = Math.random();  
            System.out.println(r);  
        } while(r < 0.99d);  
    }  
} ///:~
```

4.2.3 Bucle for

Su formato es el siguiente:

```
for(expresion1;expresion2;expresion3){  
    Grupo de sentecias;
```

Expresion1 es una asignación de un valor a una variable, la variable-condición del bucle. Expresion2 es la condición que se le impone a la variable del bucle y expresion3 indica una operación que se realiza en cada iteración a partir de la primera (en la primera iteración el valor de la variable del bucle es el que se le asigna en expresion1) sobre la variable del bucle.

```
public class Ejemplo11 {  
    public static void main(String[] args) {  
        for( char c = 0; c < 128; c++)
```

```
//Imprime los caracteres correspondientes a los números
//enteros comprendidos entre 0 y 128. (int)c es el entero
//correspondiente al carácter c.

    System.out.println("valor: " + (int)c +
        " caracter: " + c);
}
} ///:~
```

4.2.4 Bucle for-each

Esta característica sólo está disponible en Java 5 y versiones posteriores. Se trata de un bucle diseñado con el propósito de recorrer un conjunto de objetos. Dado lo básico de este tutorial la única estructura que hemos visto que permite almacenar un conjunto de objetos son los arrays; por lo que para nosotros será el único caso en la cual tenga sentido emplearlo. Su sintaxis es:

```
for(Tipo elemento: colecciónElementos){
    Grupo de sentencias;}
```

Tipo es el tipo de dato de los elemento del conjunto; elemento es una variable auxiliar que la primera vez que se ejecute el bucle tomará el valor del primer elemento del conjunto, la segunda vez tomará el valor del segundo elemento del conjunto y así sucesivamente. La colecciónElementos es el conjunto de elementos sobre los cuales queremos iterar (un array para nosotros). El bucle se repetirá una vez para cada elemento de la colección. Veamos un ejemplo:

```
public class Ejemplo11b {

    public static void main(String[] args) {
        //Este arrays será la colección de elementos por la
        que iteraremos
        int array[] = new int[10];
        int suma = 0, contador = 0;
        //con este bucle damos valores a los elementos del
        array
        for (int i = 0; i < array.length; i++) {
            array[i]= 2*i;
        }
    }
}
```

```
        //este es el nuevo tipo de bucle. Va acumulando de la
variable
        //suma los valores de todo los elementos del array;
el bucle
        //se repetirá a una vez por cada elemento del array y
la variable e
        //ir a tomando como valores cada uno de los
contenidos de éste
        for (int e : array){ //para cada elemento del array
            suma = suma + e;
        }
        System.out.println(suma);
    }
}
```

4.2.5 Break y continue

No se tratan de un bucle, pero sí de una sentencia íntimamente relacionada con estos. El encontrarse una sentencia break en el cuerpo de cualquier bucle detiene la ejecución del cuerpo del bucle y sale de este, continuándose ejecutando el código que hay tras el bucle.

Esta sentencia también se puede usar para forzar la salida del bloque de ejecución de una instrucción condicional (esto ya se vio con switch).

Continue también detiene la ejecución del cuerpo del bucle, pero en esta ocasión no se sale del bucle, sino que se pasa a la siguiente iteración de este.

Observaremos el funcionamiento de ambos en un mismo ejemplo:

```
public class Ejemplo12 {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // terminamos aqui el bucle
//Salto a la siguiente iteración si i no es divisible entre 9
            if(i % 9 != 0) continue;
//Si i es divisible entre 9 se imprime
            System.out.println(i);
        }
        int i = 0;
        // Lazo infinito del cual se sale con break:
        while(true) {
            i++;
            if(i == 120) break; // Salimos del lazo
        }
    }
}
```

```
        System.out.println(i);  
    }  
}  
} ///:~
```

4.3 RETURN

Sus funciones son las mismas que en C++. Cuando se llama a un procedimiento (que en OOP se denominó método) al encontrarse con una sentencia return se pasa el valor especificado al código que llamó a dicho método y se devuelve el control al código invocador. Su misión tiene que ver con el control de flujo: se deja de ejecutar código secuencialmente y se pasa al código que invocó al método.

Esta sentencia también está profundamente relacionada con los métodos, ya que es la sentencia que le permite devolver al método un valor. Podíamos haber esperado ha hablar de métodos para introducir esta sentencia, pero hemos decidido introducirla aquí por tener una función relacionada, entre otras cosas, con control de flujo. En el ejemplo 7 ya se ha ejemplificado su uso.

5 OBJETOS Y CLASES

Como ya hemos comentado Java es un lenguaje totalmente orientado a objetos, mucho más que, por ejemplo, C++. En Java todo es un objeto, a excepción de los tipos básicos de variables enteras, reales y char. Pero bien, ¿qué es un objeto? y ¿qué es la programación orientada a objetos?.

Responder a estas preguntas no es en absoluto trivial, hay libros enteros escritos sobre objetos y metodologías de programación orientada a objetos sin abordar ningún lenguaje de programación en concreto . Aquí simplemente daremos unas nociones muy básicas de programación orientada a objetos que nos permitan empezar a adentrarnos en el mundo de Java. Si el lector está interesado en el tema le puede consultar los tutoriales Orientación a Objetos: Conceptos y Terminología (<http://javahispano.org/tutorials.item.action?id=25>) y Guía básica de Referencia sobre Casos de Uso (<http://javahispano.org/tutorials.item.action?id=28>).

5.1 INTRODUCCIÓN

En los años 60 la programación se realizaba de un modo “clásico” (no orientado a objetos). Un programa era un código que se ejecutaba, los trozos de código que se podían emplear en varias ocasiones a lo largo del programa (reusar) se escribían en forma de procedimientos que se invocaban desde el programa, y esta era la única capacidad de reuso de código posible.

Según los códigos se fueron haciendo más grandes y complejos este estilo de programación se hacía más inviable: es difícil programar algo de grandes dimensiones con este estilo de programación. La única posibilidad de repartir trozos de código relativamente independientes entre programadores son los procedimientos, y al final hay que juntar todos estos con el programa central que los llama, siendo frecuente encontrar problemas al unir estos trozos de código.

En los años 70 se empezó a imponer con fuerza otro estilo de programación: POO, programación orientada a objetos (en la literatura suele aparecer como OOP, Object Oriented Programming). Aquí un programa no es un código que llama a procedimientos, aquí un programa es un montón de objetos, independientes entre sí, que dialogan entre ellos pasándose mensajes para llegar a resolver el problema en cuestión.

A un objeto no le importa en absoluto como está implementado otro objeto, que código tiene o deja de tener, que variables usa.... sólo le importa a que mensajes es capaz de responder. Un mensaje es la invocación de un método de otro objeto. Un método es muy

semejante a un procedimiento de la programación clásica: a un método se le pasan uno, varios o ningún dato y nos devuelve un dato a cambio.

Si hay que repartir un programa de grandes dimensiones entre varios programadores a cada uno se le asignan unos cuantos objetos, y en lo único que tendrán que ponerse de acuerdo entre ellos es en los mensajes que se van a pasar; la forma en que un programador implemente sus objetos no influye en absoluto en lo que los demás programadores hagan. Esto es así gracias a que los objetos son independientes unos de otros (cuanta mayor sea la independencia entre ellos de mayor calidad serán).

Si analizamos lo que hemos dicho hasta aquí de los objetos veremos que estos parecen tener dos partes bastante diferenciadas: la parte que gestiona los mensajes, que ha de ser conocida por los demás, y que no podremos cambiar en el futuro sin modificar los demás objetos (sí es posible añadir nuevos métodos para dar nuevas funciones al objetos sin modificar los métodos ya existentes). La otra parte es el mecanismo por el cual se generan las acciones requeridas por los mensajes el conjunto de variables que se emplean para lograr estas acciones. Esta segunda parte es, en principio, totalmente desconocida para los demás objetos (a veces no es así, pero es lo ideal en un buena OOP). Por ser desconocida para los demás objetos podemos en cualquier momento modificarla sin que a los demás les importe, y además cada programador tendrá total libertad para llevarla a cabo como él considere oportuno.

La OOP permite abordar con más posibilidades de éxito y con un menor coste temporal grandes proyectos de software, simplificándole además la tarea al programador.

5.2 CLASES Y HERENCIA

Una clase es la “plantilla” que usamos para crear los objetos. Todos los objetos pertenecen a una determinada clase. Un objeto que se crea a partir de una clase se dice que es una instancia de esa clase. Las distintas clases tienen distintas relaciones de herencia entre si: una clase puede derivarse de otra, en ese caso la clase derivada o clase hija hereda los métodos y variables de la clase de la que se deriva o clase padre. En Java todas las clases tienen como primer padre una misma clase: la clase Object.

Por motivos de simplicidad y dada la corta duración de este curso ignoraremos la existencia del concepto de package en la explicación de los siguientes conceptos, o haremos breves referencias a este concepto sin dar demasiadas explicaciones. En general en lo que al alumno respecta se recomienda ignorar, al menos durante este curso, toda referencia al término “package”.

Vamos a continuación a profundizar en todos estos conceptos y a explicar su sintaxis en Java.

5.2.1 Definición de una clase

La forma más general de definición de una clase en Java es:

```
[Modificador] class nombreClase [extends nombreClasePadre]
[implements interface] {
    Declaración de variables;
    Declaración de métodos;
}
```

Los campos que van entre corchetes son optativos. nombreClase es el nombre que le queramos dar a nuestra clase, nombreClasePadre es el nombre de la clase padre, de la cual hereda los métodos y variables. En cuanto al contenido del último corchete ya se explicará más adelante su significado.

Los modificadores indican las posibles propiedades de la clase. Veamos que opciones tenemos:

5.2.1.1 Modificadores de clases

public: La clase es pública y por lo tanto accesible para todo el mundo. Sólo podemos tener una clase public por unidad de compilación, aunque es posible no tener ninguna.

Ninguno: La clase es “amistosa”. Será accesible para las demás clases del package. Sin embargo mientras todas las clases con las que estemos trabajando estén en el mismo directorio pertenecerán al mismo package y por ello serán como si fuesen públicas. Como por lo de ahora trabajaremos en un solo directorio asumiremos que la ausencia de modificador es equivalente a que la clase sea pública.

final: Indicará que esta clase no puede “tener hijo”, no se puede derivar ninguna clase de ella.

abstract: Se trata de una clase de la cual no se puede instanciar ningún objeto.

Veamos un ejemplo de clase en Java:

```
class Animal{
    int edad;
    String nombre;

    public void nace(){
        System.out.println("Hola mundo");
    }
    public void getNombre(){
        System.out.println(nombre);
    }
    public void getEdad(){
        System.out.println(edad);
    }
} ///:~
```

5.2.1.2 Sobrecarga de métodos

Java admite lo que se llama sobrecarga de métodos: puede haber varios métodos con el mismo nombre pero a los cuales se les pasan distintos parámetros. Según los parámetros que se le pasen se invocará a uno u otro método:

```
class Animal{
    int edad;
    String nombre;

    public void nace(){
        System.out.println("Hola mundo");
    }
    public void getNombre(){
        System.out.println(nombre);
    }
    public void getNombre(int i){
        System.out.println(nombre + " " + edad);
    }
}
```



```
    }  
    public void getEdad(){  
        System.out.println(edad);  
    }  
} ///:~
```

5.2.1.3 Constructores

Constructores son métodos cuyo nombre coincide con el nombre de la clase y que nunca devuelven ningún tipo de dato, no siendo necesario indicar que el tipo de dato devuelto es void. Los constructores se emplean para inicializar los valores de los objetos y realizar las operaciones que sean necesarias para la generación de este objeto (crear otros objetos que puedan estar contenidos dentro de este objeto, abrir un archivo o una conexión de internet.....).

Como cualquier método, un constructor admite sobrecarga. Cuando creamos un objeto (ya se verá más adelante como se hace) podemos invocar al constructor que más nos convenga.

```
class Animal{  
  
    int edad;  
    String nombre;  
  
    public Animal(){  
    }  
    public Animal(int _edad, String _nombre){  
        edad = _edad;  
        nombre = _nombre;  
    }  
    public void nace(){  
        System.out.println("Hola mundo");  
    }  
    public void getNombre(){  
        System.out.println(nombre);  
    }  
    public void getNombre(int i){  
        System.out.println(nombre + " " + edad);  
    }  
}
```

```
    }  
    public void getEdad(){  
        System.out.println(edad);  
    }  
} ///:~
```

En la Figura 1 podemos observar dos objetos de tipo Animal creados mediante BlueJ. El objeto que tiene el nombre animal4 dentro de BlueJ fue creado mediante el constructor Animal(int, String) pasándole como argumentos el entero 12 y el nombre "Tomás", mientras que el que tiene el nombre "animal3" fue creado con el mismo constructor, pero con los argumentos 2 y "Tobi".

5.2.2 Modificadores de métodos y variables

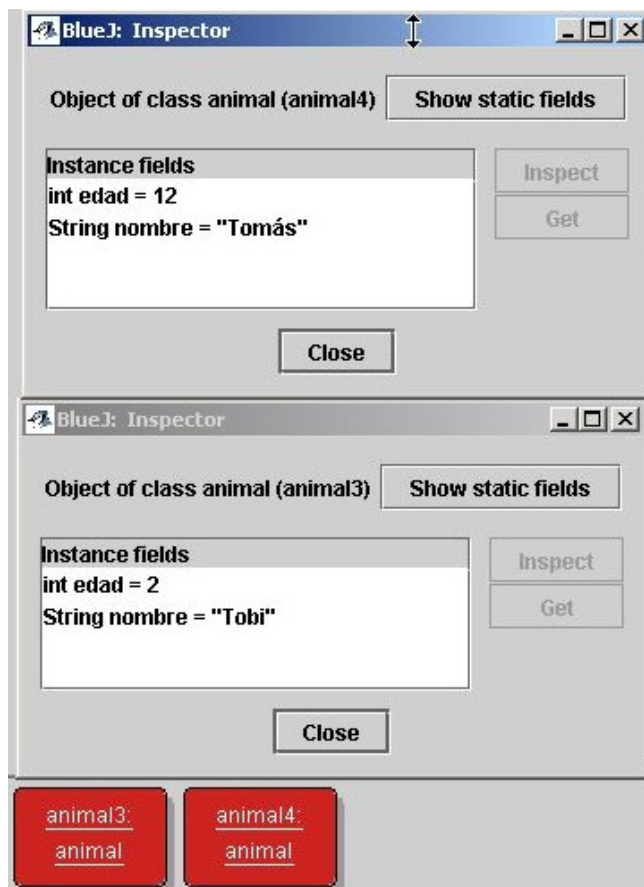


Figura 1 Dos objetos de tipo animal visualizados en BlueJ.

Antes de explicar herencia entre clases comentaremos cuales son los posibles modificadores que pueden tener métodos y variables y su comportamiento:

5.2.2.1 Modificadores de variables

public: Pública, puede acceder todo el mundo a esa variable.

Ninguno: Es "amistosa", puede ser accedida por cualquier miembro del package, pero no por otras clases que pertenezcan a otro package distinto

protected: Protegida, sólo pueden acceder a ella las clases hijas de la clase que posee la variable y las que estén en el mismo package.

private: Privada, nadie salvo la clase misma puede acceder a estas variables. Pueden acceder a ella todas las instancias de la clase (cuando decimos clase nos estamos refiriendo a todas sus posibles instancias)

static: Estática, esta variable es la misma para todas las instancias de una clase, todas comparten ese dato. Si una instancia lo modifica todas ven dicha modificación.

final: Final, se emplea para definir constantes, un dato tipo final no puede variar nunca su valor. La variable no tiene porque inicializarse en el momento de definirse, pero cuando se inicializa ya no puede cambiar su valor.

```
class Marciano {
    boolean vivo;
    private static int numero_marcianos = 0;
    final String Soy = "marciano";

    void quienEres(){
        System.out.println("Soy un " + Soy);
    }

    Marciano(){
        vivo = true;
        numero_marcianos++;
    }

    void muerto(){
        if(vivo){
            vivo = false;
            numero_marcianos--;
        }
    }
} //::~~
```

En la figura Figura 2 Podemos ver 4 objetos de tipo Marciano creados en BlueJ. Sobre el objeto "marciano3" se ha invocado el método *muerto*, lo cual ha provocado la muerte del marciano, es decir, la variable de tipo booleano vivo ha pasado a valer "false". Podemos observar también el resultado de la inspección de las variables estáticas de la clase, vemos que numero_marcianos vale 3. Hemos creado 4 marciano, con lo cual esa variable debiera tener el valor 4; hemos matado uno de los marcianos, con lo cual esa variable se decrementa y pasa a valer 3, como nos indica BlueJ.

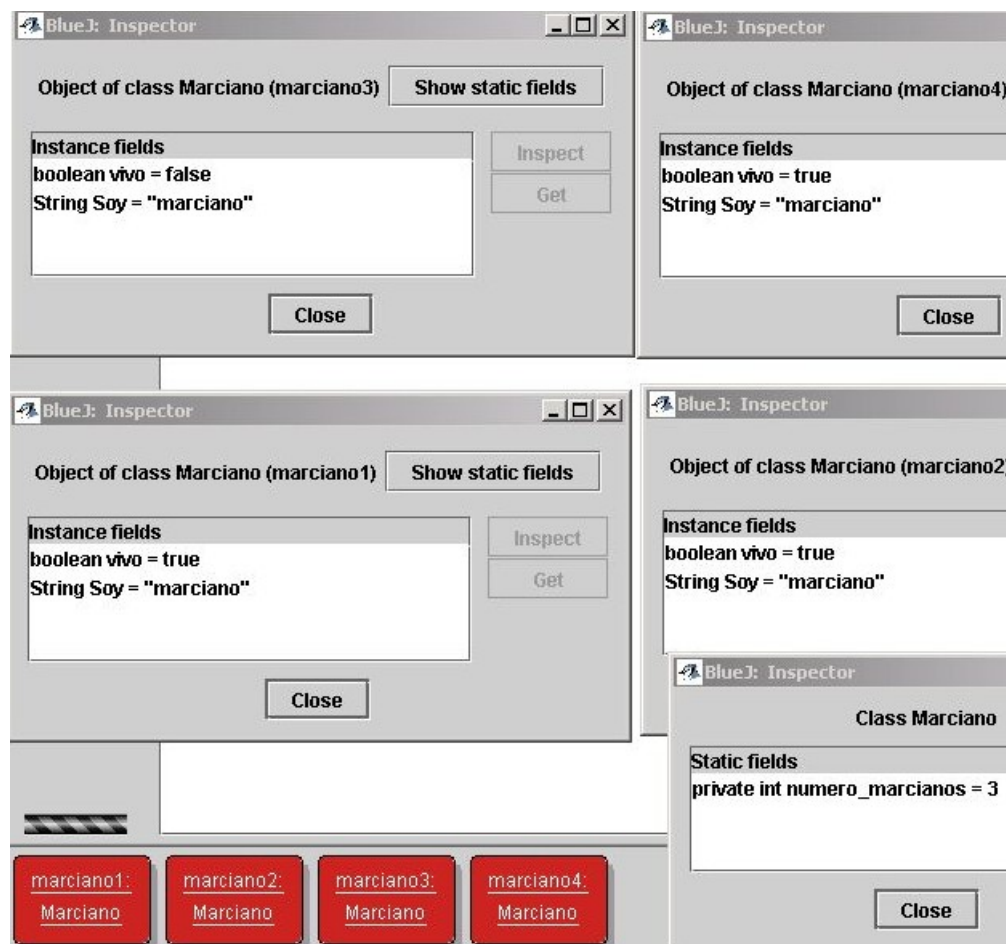


Figura 2 Cuatro objetos tipo Marciano creados e inspeccionados en BlueJ.

5.2.2.2 Modificadores de un método

public: Pública, puede acceder todo el mundo a este método.

Ninguno: Es "amistoso", puede ser accedida por cualquier miembro del package, pero no por otras clases que pertenecen a otro package.

protected: Protegido, sólo pueden acceder a ella las clases hijas de la clase que posea el método y las que estén en el mismo package.

private: Privada, nadie salvo la clase misma puede acceder a estos métodos.

static: Estática, es un método al cual se puede invocar sin crear ningún objeto de dicha clase. Math.sin, Math.cos son dos ejemplos de métodos estáticos. Desde un método estático sólo podemos invocar otros métodos que también sean estáticos.

final: Final, se trata de un método que no podrá ser cambiado por ninguna clase que herede de la clase donde se definió. Es un método que no se puede "sobrescribir". Más adelante se explicará que es esto.

```
class Mat{
    static int cuadrado(int i){
        return i*i;
    }
    static int mitad (int i){
        return i/2;
    }
} //::~~
```

5.2.3 Herencia

Cuando en Java indicamos que una clase "extends" otra clase estamos indicando que es una clase hija de esta y que, por lo tanto, hereda todos sus métodos y variables. Este es un poderoso mecanismo para la reusabilidad del código. Podemos heredar de una clase, por lo cual partimos de su estructura de variables y métodos, y luego añadir lo que necesitamos o modificar lo que no se adapte a nuestros requerimientos. Veamos un ejemplo:

```
class Animal{
    protected int edad;
    String nombre;
```

```
public Animal(){
}

public Animal(int _edad, String _nombre){
    edad = _edad;
    nombre = _nombre;
}

public void nace(){
    System.out.println("Hola mundo");
}

public void getNombre(){
    System.out.println(nombre);
}

public void getNombre(int i){
    System.out.println(nombre + " " + edad);
}

public void getEdad(){
    System.out.println(edad);
}
} ///:~
//La clase Perro extiende a Animal, heredando sus métodos y
//variables
public class Perro extends Animal{

    Perro(){
        edad = 0;
        nombre ="Tobi";
    }

    Perro(int edad, String nombre){
//Esta sentencia invoca a un constructor de la clase padre.
        super(edad,nombre);
    }
}
```

```

    }

    //Método estático que recibe un objeto de tipo Perro e
    //invoca a su método getEdad()
    static void get1(Perro dog){
    //El método getEdad() no está definido en Perro, lo ha
    //heredado de Animal.
        dog.getEdad();
    }

    public static void main (String[] args){
    //Creamos un objeto de tipo Perro
        Perro dog = new Perro(8,"Bambi");
    //Invocamos al método estático get1 pasándole el objeto //
    de tipo Perro creado.
        Perro.get1(dog);
    }
} ///:~

```

En la figura 3 podemos observar un objeto de tipo *Perro* creado invocando al constructor por



Figura 3 Inspección de un objeto tipo perro creado en BlueJ.

defecto *Perro()*. Gracias a los mecanismos de inspección de BlueJ podemos observar como la variable *edad* se ha inicializado a 0, y la variable *nombre* a "Tobi", como se indicó en el constructor. Así mismo vemos como el objeto *Perro* ha heredado todos los métodos definidos en la clase *Animal*.

Si un método no hace lo que nosotros queríamos podemos sobrescribirlo (overriding). Bastará para ello que definamos un método con el mismo nombre y argumentos.

Veámoslo sobre el ejemplo anterior:

```
class Animal{

    protected int edad;
    String nombre;

    public Animal(){
    }
    public Animal(int _edad, String _nombre){
        edad = _edad;
        nombre = _nombre;
    }
    public void nace(){
        System.out.println("Hola mundo");
    }
    public void getNombre(){
        System.out.println(nombre);
    }
    public void getNombre(int i){
        System.out.println(nombre + " " + edad);
    }
    public void getEdad(){
        System.out.println(edad);
    }
} ///:~

public class Perro2 extends Animal{
    Perro2(){
        edad = 0;
        nombre = "Tobi";
    }
    Perro2(int edad, String nombre){
        super(edad,nombre);
    }

    static void get1(Perro2 dog){
```



```

        dog.getEdad();
//Cuando ejecutemos este método en vez de ejecutarse el
//código de la clase padre se ejecutará el código de la clase
//hija, ya que ésta ha sobrescrito este método.
        dog.getNombre(11);
    }
//Sobreescribe al método de la clase padre.
    public void getNombre(int i){
        System.out.println(nombre + " " + i);
    }

    public static void main (String[] args){
        Perro2 dog = new Perro2(8,"hola");
        Perro2.get1(dog);
    }
} ///:~

```

En la figura podemos observar un objeto de tipo *Perro2*; esta vez ha sido creado invocando al constructor *Perro2(int, String)*, pausándole como argumentos el entero 6 y una cadena de



Figura 4 Objeto de tipo perro2 inspeccionado mediante BlueJ.

caracteres con valor "Tobi_2". Empleado los mecanismos de inspección de BlueJ podemos observar como la variable *edad* se ha inicializado a 6, y la variable *nombre* a "Tobi_2". Vemos como el objeto *Perro* ha heredado todos los métodos

definidos en la clase *Animal*, y como BlueJ nos advierte de que el método `getNombre(i)` de la clase *Animal*, ha sido sobrescrito (redefined) en la clase hija, por lo que cuando se invoque será el código del hijo, y no el del padre, el que se ejecute

5.2.4 Creación y referencia a objetos

Aunque ya hemos visto como se crea un objeto vamos a formalizarlo un poco. Un objeto en el ordenador es esencialmente un bloque de memoria con espacio para guardar las variables de dicho objeto. Crear el objeto es sinónimo de reservar espacio para sus variables, inicializarlo es darle un valor a estas variables. Para crear un objeto se utiliza el comando `new`. Veámoslo sobre un ejemplo:

```
class Fecha{

    int dia,mes,ano;

    Fecha(){
        dia=1;
        mes = 1;
        ano = 1900;
    }

    Fecha (int _dia, int _mes, int _ano){
        dia= _dia;
        mes = _mes;
        ano = _ano;
    }
} ///:~
```

```
Fecha hoy;
hoy = new Fecha();
```

Con el primer comando hemos creado un puntero que apunta a una variable tipo fecha, como está sin inicializar apuntará a null. Con el segundo inicializamos el objeto al que apunta

hoy, reservando espacio para sus variables. El constructor las inicializará, tomando el objeto hoy el valor 1-1-1900.

```
| Fecha un_dia = new Fecha(8,12,1999);
```

Con esta sentencia creamos una variable que se llama un_dia con valor 8-12-1999.

Una vez creado un objeto será posible acceder a todas sus variables y métodos públicos, así por ejemplo en el ejemplo anterior un_dia.dia = 8. Si la variable fuese privada solo podrían acceder a ella sus instancias:

```
class Fecha{
    private int dia,mes,ano;

    Fecha(){
        dia=1;
        mes = 1;
        ano = 1900;
    }

    Fecha (int ndia, nmes, nano){
        dia= ndia;
        mes = nmes;
        ano = nano;
    }
} ///:~
```

De este modo no podríamos acceder a las variables de la clase fecha, para acceder a ella tendríamos que hacerlo mediante métodos que nos devolviesen su valor. A esto se le denomina encapsulación. Esta es la forma correcta de programar OOP: no debemos dejar acceder a las variables de los objetos por otro procedimiento que no sea paso de mensajes entre métodos.

5.2.5 *this*

Es una variable especial de sólo lectura que proporciona Java. Contiene una referencia al objeto en el que se usa dicha variable. A veces es útil que un objeto pueda referenciarse a si mismo:

```
class Cliente{
    public Cliente(String n){
        //Llamamos al otro constructor. El empleo de this ha de ser
        //siempre en la primera línea dentro del constructor.
        this(n, Cuenta.nuevo_numero());
        .....
    }
    public Cliente (String n, int a){
        nombre = n;
        numero_cuenta = a;
    }
    .....
} ///:~
```

Otro posible uso de *this*, que ya se ha visto en ejemplos anteriores es diferenciar entre variables locales de un método o constructor y variables del objeto. En los códigos del cd correspondientes a los ejemplos Perro.java y Perro2.java el constructor de la clases Animal aunque realiza la misma función que los que se recogen en estos apuntes son ligeramente diferentes:

```
    public Animal(int edad, String nombre){
        //this.edad = variable del objeto Perro
        //edad = variable definida sólo dentro del constructor
        this.edad =edad;
        this.nombre=nombre;
    }
```

5.2.6 super

Del mismo modo que this apunta al objeto actual tenemos otra variable super que apunta a la clase de la cual se deriva nuestra clase

```
class Gato {  
  
    void hablar(){  
        System.out.println("Miau");  
    }  
  
} ///~  
  
class GatoMagico extends Gato {  
  
    boolean gentePresente;  
  
    void hablar(){  
        if(gentePresente)  
            //Invoca al método sobrescrito de la clase padre  
            super.hablar();  
  
        else  
            System.out.println("Hola");  
    }  
} ///:~
```

Uno de los principales usos de super es, como ya se empleó en un ejemplo, llamar al constructor de la clase padre.

5.3 INTERFACES

En Java no está soportada la herencia múltiple, esto es, no está permitido que una misma clase pueda heredar las propiedades de varias clases padres. En principio esto pudiera parecer una propiedad interesante que le daría una mayor potencia al lenguaje de programación, sin

embargo los creadores de Java decidieron no implementar la herencia múltiple por considerar que esta añade al código una gran complejidad (lo que hace que muchas veces los programadores que emplean programas que sí la soportan no lleguen a usarla).

Sin embargo para no privar a Java de la potencia de la herencia múltiple sus creadores introdujeron un nuevo concepto: el de interface. Una interface es formalmente como una clase, con dos diferencias: sus métodos están vacíos, no hacen nada, y a la hora de definirla en vez de emplear la palabra clave "class" se emplea "interface". Veámoslo con un ejemplo:

```
interface Animal{

    public int edad = 10;
    public String nombre = "Bob";

    public void nace();

    public void getNombre();

    void getNombre(int i);

} ///:~
```

Cabe preguntarnos cual es el uso de una interface si sus métodos están vacíos. Bien, cuando una clase implementa una interface lo que estamos haciendo es una promesa de que esa clase va a implementar todos los métodos de la interface en cuestión. Si la clase que implementa la interface no "sobrescribiera" alguno de los métodos de la interface automáticamente esta clase se convertiría en abstracta y no podríamos crear ningún objeto de ella. Para que un método sobrescriba a otro ha de tener el mismo nombre, se le han de pasar los mismos datos, ha de devolver el mismo tipo de dato y ha de tener el mismo modificador que el método al que sobrescribe. Si no tuviera el mismo modificador el compilador nos daría un error y no nos dejaría seguir adelante. Veámoslo con un ejemplo de una clase que implementa la anterior interface:

```
interface Animal{

    public int edad = 10;
    public String nombre = "Bob";

    public void nace();
    public void getNombre();
    void getNombre(int i);
} ///:~

public class Perro3 implements Animal{
    Perro3(){
        getNombre();
        getNombre(8);
    }
    //Compruévese como si cambiamos el nombre del método a nace()
    //no compila ya que no henos sobreescrito todos los métodos
    //de la interfaz.
    public void nace(){
        System.out.println("hola mundo");
    }

    public void getNombre(){
        System.out.println(nombre );
    }

    public void getNombre(int i){
        System.out.println(nombre +" " +i);
    }

    public static void main (String[] args){
        Perro3 dog = new Perro3();
        //Compruevese como esta línea da un error al compilar debido
        //a intentar asignar un valor a una variable final
        //    dog.edad = 8;
    }
}
```

```
| } ///:~
```

Las variables que se definen en una interface llevan todas ellas el atributo final, y es obligatorio darles un valor dentro del cuerpo de la interface. Además no pueden llevar modificadores private ni protected, sólo public. Su función es la de ser una especie de constantes para todos los objetos que implementen dicha interface.

Por último decir que aunque una clase sólo puede heredar propiedades de otra clase puede implementar cuantas interfaces se desee, recuperándose así en buena parte la potencia de la herencia múltiple.

```
| interface Animal1{
|
|     public int edad = 10;
|     public String nombre = "Bob";
|
|     public void nace();
| } ///:~
|
| interface Animal2{
|
|     public void getNombre();
| } ///:~
|
| interface Animal3{
|
|     void getNombre(int i);
|
| } ///:~
|
| public class Perro4 implements Animal1,Animal2,Animal3{
|     Perro4(){
|         getNombre();
|         getNombre(8);
|         //edad = 10; no podemos cambiar este valor
|     }
|     public void nace(){
```



```
        System.out.println("hola mundo");
    }

    public void getNombre(){
        System.out.println(nombre );
    }

    public void getNombre(int i){
        System.out.println(nombre +" " +i);
    }

    public static void main (String[] args){
        Perro4 dog = new Perro4();
    }
} ///:~
```

5.4 NUESTRO PRIMER PROGRAMA ORIENTADO A OBJETOS

Posiblemente este tema, del cual ya hemos finalizado la teoría, sea el más difícil de asimilar. La OOP no es sencilla en un principio, pero una vez que se empieza a dominar se ve su potencia y su capacidad para simplificar problemas complejos.

Para intentar hacer este tema un poco menos teórico y ver algo de lo que aquí se ha expuesto se ha realizado el siguiente programilla; en el se empieza un ficticia guerra entre dos naves, una de marcianos y otra de terrícolas, cada uno de los cuales va disparando, generando números aleatorios, y si acierta con el número asignado a algún componente de la otra nave lo "mata". En el se hacen ciertos usos adecuados de la herencia y ciertos usos no tan adecuados, para ejemplificar tanto lo que se debe como lo que no se debe hacer.

No pretende ser en absoluto ninguna maravilla de programa, simplemente se trata con el de romper el esquema de programación estructurada o clásica, en el cual el programa se realiza en el main llamando a funciones. En OOP un programa, digámoslo una vez más, es un conjunto de objetos que dialogan entre ellos pasándose mensajes para resolver un problema. Veremos como, ni más ni menos, esto es lo que aquí se hace para resolver un pequeño problema-ejemplo.

Antes de pasar al código, veamos la representación UML del diagrama de clases que componen nuestro problema visualizado en BlueJ:

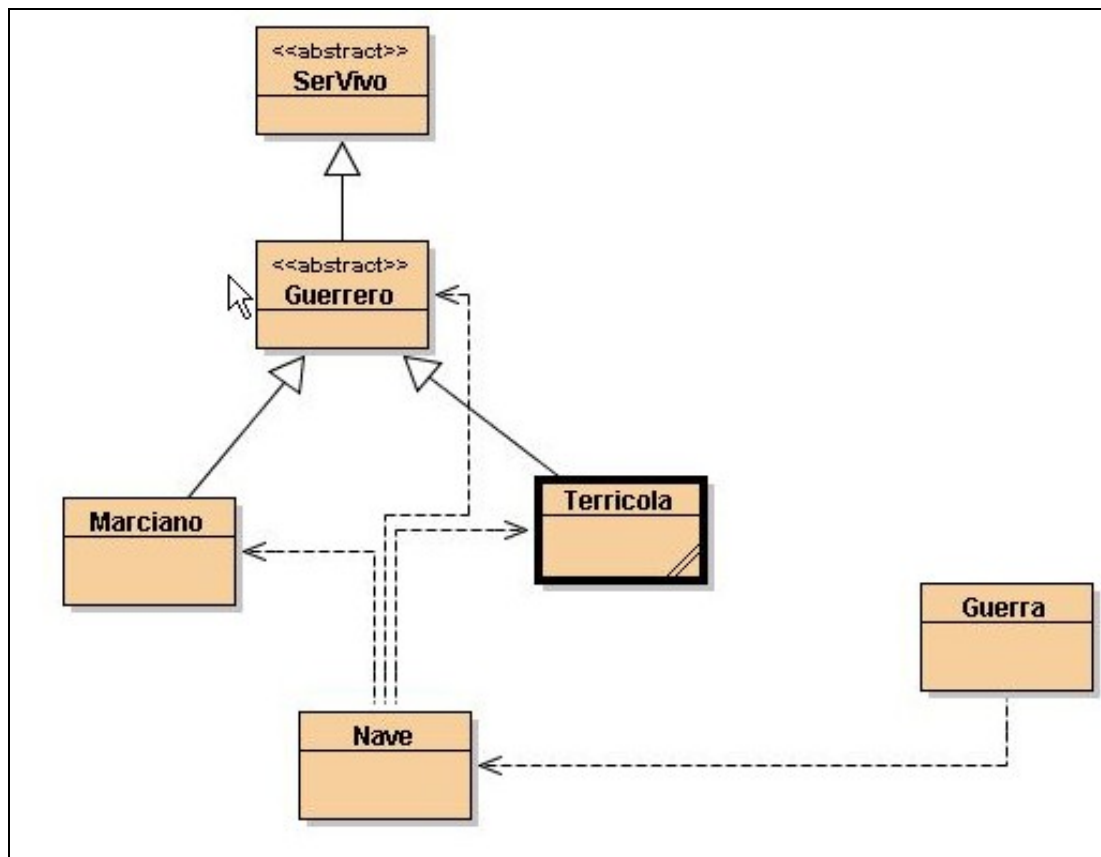


Figura 5.- Diagrama de clases de nuestro problema visualizado en BlueJ. Contamos con una clase Abstracta **SerVivo**, que representa a un ser vivo. **Guerrero** es nuevamente una clase abstracta que hereda de **SerVivo** (un guerrero es un ser vivo); en ella se sitúa el comportamiento de atacar. Tenemos dos clases, **Terrícola** y **Marciano**, que representarán a terrícolas y marcianos. La Clase **Nave** actúa como un contenedor cuya tripulación puede ser un conjunto de terrícolas o de marcianos. Por último **Guerra** es una clase auxiliar encargada de la creación de las naves y de la simulación de la guerra.

```
/**Esta clase está formada por una variable protegida de tipo
 *boolean llamada edad, con un método que devuelve su valor.
 *Representa as un SerVivo y es abstracta, al igual que el
 *concepto de ser vivo es abstracto: en el mundo hay plantas,
 *animales, personas... que son seres vivos, pero no "seres
 *vivos" de modo independiente*/
 *public abstract class SerVivo{

    public boolean isVivo(){
        return vivo;
    }

    protected boolean vivo = true;
} ///:~
```

```
/**Esta clase representa a un Guerrero, que nuevamente es un
 *concepto abstracto: los guerreros han de ser Terricolas o
 *Marcianos*/
public abstract class Guerrero extends SerVivo {
    /**Constructor. Almacena la cadena de caracteres que se le
    *pasa en la variable soy del objeto, e inicializa la variable
    *blanco empleando el método generablanc*/
    public Guerrero (String soy){
        blanco = generaBlanco();
        this.soy = soy;
    }

    /**Si el guerrero está vivo (la variable vivo la hereda de
    *ser vivo vale true) el guerrero "dispara" mediante este
    *método un número aleatorio entre 0 y 10. Si está muerto
    *dispara un 100, que nunca matará a nadie; de esa forma
    *modelamos que un muerto nunca mata a nadie.*/
    public int dispara (){
        if (vivo){
            int disparo = ((int)(Math.random()*10));
            System.out.println(soy + "Dispara nº " +disparo);
        }
    }
}
```

```
        return disparo;
    }
    else
        return 100;
    }

    /**Método que devuelve el valor de la variable blanco*/
    public int getBlanco(){
        return blanco;
    }

    /**Método privado, que por lo tanto sólo será accesible por
    *el propio objeto, que emplea para iniciar la variable
    *blanco*/
    private int generaBlanco (){
        return ((int)(Math.random()*10));
    }

    /*Variables del objeto, una cadena de caracteres dónde se
    *almacenará "Terricola" o "Marciano" según el guerrero sea un
    *terricola o un marciano, y un entero, que es el entero
    *aleatorio con el que han de acertar para matar a este
    *Guerrero*/
    protected int blanco;
    private final String soy;
} ///:~
```

```
/**Esta clase modela a un terricola*/
class Terricola extends Guerrero{
    /**Constructor*/
    Terricola(String soy){
        //Invoca al constructor del padre, al de Guerrero.
        super(soy);
        //Incrementa la variable estática total
        total++;
        //Almacena en el objeto la cadena de caracteres que se le ha
        //pasado al constructor.
        this.soy= soy;
    }
    /**Mediante este método se le comunica al Terricola que le
    *han disparado. Si el número del disparo coincide con el
    *valor de la variable blanco, que ha heredado de Guerrero, se
    *muere, decrementa la variable estática total,e imprime un
    mensaje por consola*/
    public void recibeDisparo(int i){
        if (vivo && blanco == i){
            vivo = false;
            total--;
            System.out.println (soy + " Muerto por disparo nº
" +i);
        }
    }
    /**Método que devuelve el valor de la variable total*/
    public int getTotal(){
        return total;
    }
    /**Variable estática que nos permite llevar cuenta de cuantos
    Terricola hay en cada momento, ya que se incrementa al crear
    un Terricola y se decrementa al morir*/
    private static int total = 0;
    /**Cadena de caracteres que almacenará "Terricola"*/
    private String soy;
} ///:~
```

```
/**Todos los comentarios que hay en la clase Terricola son
*igualmente válidos para la clase Guerrero. Sus códigos son
*idénticos. ¿No podíamos haber empleado la herencia para no
*replicar el código?. En este caso no ya que todos los
*métodos y el constructor que aquí hay acceden a la variable
*estática total, que debe ser definida al nivel de las clases
*Marciano y Terricola para contar con dos variables
*estáticas, cada una de las cuales lleva cuenta del número de
*marcianos y de terrícolas que hay en cada momento.*/
class Marciano extends Guerrero{
    Marciano(String soy){
        super(soy);
        this.soy =soy;
        total++;
    }

    public void recibeDisparo(int i){
        if (vivo && blanco == i){
            vivo = false;
            total--;
            System.out.println (soy + "Muerto por disparo nº
" +i);
        }
    }

    public int getTotal(){
        return total;
    }
    private static int total = 0;
    private String soy;

} ///::~~
```

```
/**Clase Nave, que actúa como contenedor de los Marcianos y
de los Terrícolas*/
class Nave {
/**Constructor, se le pasa una cadena de caracteres que será
"Terrícolas" si es una nave de Terrícolas, y "Marcianos" si
es la nave de los Marcianos*/
    public Nave (String somos){
        this.somos = somos;
        for (int i = 0; i<10; i++){
//Si es la nave de los terrícolas inicializa cada una de las
//posiciones del array de guerreros con Terrícolas, que son
//Guerreros, de ahí que puedan almacenarse en un contenedor
//de Guerreros
            if (somos.equals("Terrícolas")){
                tripulacion [i] = new Terricola(somos);
            }
//Si es la nave de los marcianos creamos una tripulación de
//Marcianos
            else{
                tripulacion [i] = new Marciano(somos);
            }
        }
//Imprimos un texto por consola para saber que nave se creó
        System.out.println("Creada nave de " + somos);
    }
/**Método que invocará un nave sobre la otra para notificarle
*que le han disparado*/
    public void recibeDisparo(int i){
        for (int j=0; j<10;j++){
//Si es la nave de los Terrícolas
            if (somos.equals("Terrícolas")){
//Hago un cast del Guerrero que está en la posición [j] del
//array de tripulantes a Terricola:
//((Terricola)(tripulacion[j])), de este modo puedo recuperar
```

```
//el método recibeDisparo de Terricola
((Terricola)(tripulacion[j])).recibeDisparo(i);
    }
    else{
//Si es la nave de los Marcianos el cast se hace a Marciano
((Marciano)(tripulacion[j])).recibeDisparo(i);
    }
}

}

/**Este método se invoca sobre la nave indicándole que el
*tripulante j ha de disparar*/
public int generaDisparo(int i){
//El método dispara() se definió en Guerrero, esta vez no es
//necesario relaizar ningún cast.
return tripulacion[i].dispara();
}

/**Método que indica cuantos tripulantes quedan en la nave*/
public int cuantosQuedan(){
//Si es la nave de los Terricolas
if (somos.equals("Terricolas")){
//Cojo al tripulante que está en la posición 1 y le hago un
//cast a Terricola, invocando el método getTotal(), que me
//devuelve el número de Terricolas que quedan vivos. Este es
//el valor que devuelve el método.
return ((Terricola)(tripulacion[1])).getTotal();
}
else{
//Idem para los Marcianos.
return ((Marciano)(tripulacion[1])).getTotal();
}
}

/**Array de Guerreros (tanto Marcianos como Terricolas son
*Guerreros). Es la tripulacion de la nave. Contendrá
*Terricolas si es la nave de Terricolas, Marcianos en caso
*contrario*/
private Guerrero[] tripulacion = new Guerrero[10];
private String somos;
```



```
} ///:~
```

```
/**Clase Guerra. Es una clase auxiliar que nos permitirá
simular la guerra*/
class Guerra {
/**Constructor. Constuye las dos naves e invoca al método
*empiezaGuerra*/
    public Guerra(){
        nave1 = new Nave("Terricolas");
        nave2 = new Nave("Marcianos");
        empiezaGuerra();
    }
/**Método que simula la guerra*/
    public void empiezaGuerra(){
//Bucle do que (ver la condición) se ejecuta mientras halla
//tripulantes vivos en ambas naves
        do{
            for(int i = 0; i<10;i++){
//Esta línea invoca el método generaDisparo sobre la nave de
//los Terricolas, método que devuelve un entero aleatorio que
//es el disparo del tripulante i de la nave de los Terricolas
//(nave1.generaDisparo(i), y le comunica el resultado a la
//nave de los Marcianos.
                nave2.recibeDisparo(nave1.generaDisparo(i));
//Idem, pero esta vez disparan los marcianos
                nave1.recibeDisparo(nave2.generaDisparo(i));
            }
        }while(nave1.cuantosQuedan(>0)&&nave2.cuantosQuedan(>0);
//Si hay tripulantes vivos en la nave de los Terricolas
//ganaron ellos
        if(nave1.cuantosQuedan(>0)){
            System.out.println("GANARON LOS
TERRICOLAS!!!!");
```

```
        }  
        //Si hay tripulantes vivos en la nave de los Marcianos  
        //ganaron ellos.  
        else if (nave2.cuantosQuedan()>0){  
            System.out.println("GANARON LOS MARCIANOS");  
        }  
    }  
    /**Método main, desde el se arranca el programa, creando un  
    *objeto Guerra, pero no se hace nada más, serán los objetos  
    *que se creen los que "dialogando entre ellos" resuelvan el  
    *programa.*/  
    public static void main(String[] args){  
        new Guerra();  
    }  
  
    private Nave nave1, nave2;  
} ///:~
```

NOTA: el código es perfectamente válido y funciona sin ningún problema. Sin embargo hay dos cosas que se podrían mejorar en el, ambas relacionadas con la herencia y con los modificadores de los métodos y las variables. Invito al lector a que busque como mejorar el código por su cuenta. Una vez que lo logre, o al menos que lo haya intentado, puede ver las posibles mejoras en el apéndice A.

5.5 APRENDIENDO A USAR LOS PAQUETES

A estas alturas deberías tener claro que una parte privada que oculta a los demás y que no es necesario conocer para poder acceder a la funcionalidad de la clase. Si hacemos cambios a la parte privada de la clase, mientras se respete la parte pública, cualquier código cliente que emplee la clase no se dará cuenta de dichos cambios.

Imagínate que tú y un compañero vais a construir en un programa complejo juntos. Os repartís el trabajo entre los dos y cada uno de vosotros implementa su parte como un montón de clases Java. Cada uno de vosotros en su código va a emplear parte de las clases del otro. Por tanto, os ponéis de acuerdo en las interfaces de esas clases. Sin embargo, cada uno de

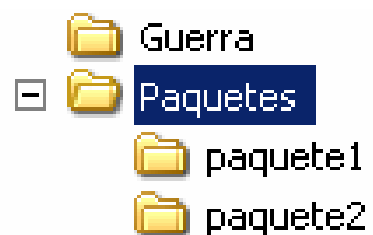
vosotros para construir la funcionalidad de esas clases probablemente se apoye en otras clases auxiliares. A tu compañero le dan igual las clases auxiliares que tú emplees. Es más, dado que el único propósito de esas clases es servir de ayuda para las que realmente constituyen la interface de tu parte del trabajo sería contraproducente que él pudiese acceder a esas clases que son detalles de implementación: tú en el futuro puedes decidir cambiar esos detalles de implementación y cambiar esas clases, modificándolas o incluso eliminándolas.

Dada esta situación ¿no sería interesante poder "empaquetar" tu conjunto de clases de tal modo que ese "paquete" sólo dejase acceder a tu compañero a las clases que tú quieras y oculte las demás?. Esas clases a las que se podría acceder serían la interface de ese "paquete". Serían "públicas". Dentro del paquete tú puedes meter cuantas más clases quieras. Pero esas no serán vistas por tu compañero y podrás cambiarlas en cualquier momento sin que él tenga que modificar su código. Es la misma idea que hay detrás de una clase pero llevada a un nivel superior: una clase puede definir cuáles de sus partes son accesibles y no accesibles para los demás. El paquete permitiría meter dentro cuantas clases quieras pero mostraría al exterior sólo aquellas que considere adecuado. Parece una buena idea ¿no?

Pues esa es precisamente la utilidad de los *package* en Java. Empaquetar un montón de clases y decidir cuáles serán accesibles para los demás y cuáles no. Para empaquetar las clases simplemente debemos poner al principio del archivo donde definimos la clase, en la primera línea que no sea un comentario, una sentencia que indique a qué paquete pertenece:

```
| package mipaquete;
```

Una clase que esté en el paquete "mipaquete" debe situarse dentro de un directorio con nombre "mipaquete". En Java los paquetes se corresponden con una jerarquía de directorios. Por tanto, si para construir un programa quiero emplear dos paquetes diferentes con nombres "paquete1" y "paquete2" en el directorio de trabajo debo crear dos subdirectorios con dichos nombres y colocar dentro de cada uno de ellos las clases correspondientes. En la figura, el directorio de trabajo desde el cual deberíamos compilar y ejecutar la aplicación es "paquetes". En cada uno de los dos subdirectorios colocaremos las clases del paquete correspondiente.



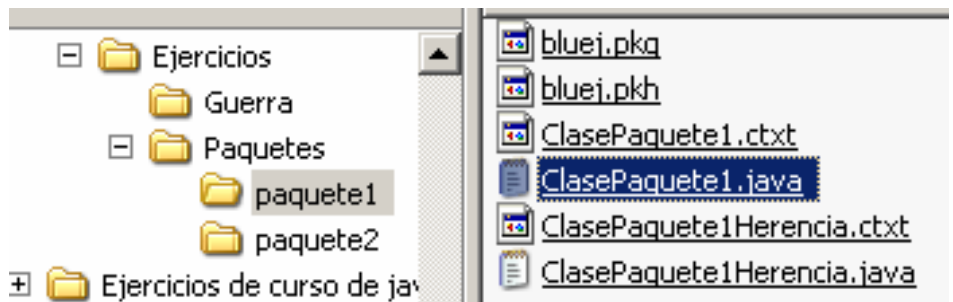
Cuando una clase se encuentra dentro de un paquete el nombre de la clase pasa a ser "NombrePaquete.NombreClase". Así, la clase "ClasePaquete1" que se encuentra físicamente en el directorio "paquete1" y cuya primera línea de código es:

```
package paquete1;
```

tendrá como nombre completo "paquete1.ClasePaquete1". Si deseamos, por ejemplo, ejecutar el método main de dicha clase debemos situarnos en el directorio "Paquetes" y teclear el comando:

```
java paquete1.ClasePaquete1
```

Para todos los efectos, el nombre de la clase es "paquete1.ClasePaquete1". En la figura, en el directorio "paquete1" se pueden observar más archivos a parte de los que contienen el código fuente Java. Son archivos de configuración de BlueJ.



Si el lector está trabajando con otra herramienta no tienen porqué estar ahí esos archivos y, en cualquier caso, no tienen nada que ver con java en sí sino con el entorno de desarrollo que he empleado para construir el ejemplo.

Cuando en una clase no se indica que está en ningún paquete, como hemos hecho hasta ahora en todos los ejemplos de este tutorial, esa clase se sitúa en el "paquete por defecto" (default package). En ese caso, el nombre de la clase es simplemente lo que hemos indicado después de la palabra reservada class sin precederlo del nombre de ningún paquete.

Es posible anidar paquetes; por ejemplo, en el directorio "paquete1" puedo crear otro directorio con nombre "paquete 11" y colocar dentro de él la clase "OtraClase". La primera línea de dicha clase debería ser:

```
package paquete1.paquete11;
```

y el nombre de la clase será "paquete1.paquete11.OtraClase".

¿Cómo indico qué clases serán visibles en un paquete y qué clases no serán visibles?. Cuando explicamos cómo definir clases vimos que antes de la palabra reservada class podíamos poner un modificador de visibilidad. Hasta ahora siempre hemos empleado el modificador public. Ese modificador significaría que la clase va a ser visible desde el exterior, forma parte de la

interfaz del paquete. Si no ponemos el modificador `públic` la clase tendrá visibilidad de paquete, es decir, no será visible desde fuera del paquete pero sí será visible para las demás clases que se encuentren en el mismo paquete que ella. Aunque hay más opciones para el modificador de visibilidad de una clase, para un curso básico como éste estas dos son suficientes.

Por tanto, poniendo o no poniendo el modificador `public` podemos decidir qué forma parte de la interfaz de nuestros paquetes y qué no.

¿Y cómo hacemos para emplear clases que se encuentren en otros paquetes diferentes al paquete en el cual se encuentra nuestra clase?. Para eso es precisamente para lo que vale la sentencia `import`. Para indicar que vamos a emplear clases de paquetes diferentes al nuestro. Así, si desde la clase "MiClase" que se encuentre definida dentro de "paquete1" quiero emplear la clase "OtraClase" que se encuentra en "paquete2" en "MiClase" debo añadir la sentencia:

```
| import paquete2.OtraClase;
```

A partir de ese momento, si OtraClase era pública, podré acceder a ella y crear instancias. El importar una clase sólo será posible si dicha clase forma parte de la interfaz pública del paquete. También podemos escribir la sentencia:

```
| import paquete2.*;
```

Que haría accesibles todas las clases públicas que se encuentren en "paquete2", y no sólo una como el ejemplo anterior.

Una opción alternativa a emplear la sentencia `import` es emplear el nombre completo de la clase cuando vayamos a acceder a ella para crear un objeto o para invocar uno de sus métodos estáticos. Así, si no hemos importado las clases del paquete2, para crear un objeto de una de sus clases deberemos escribir:

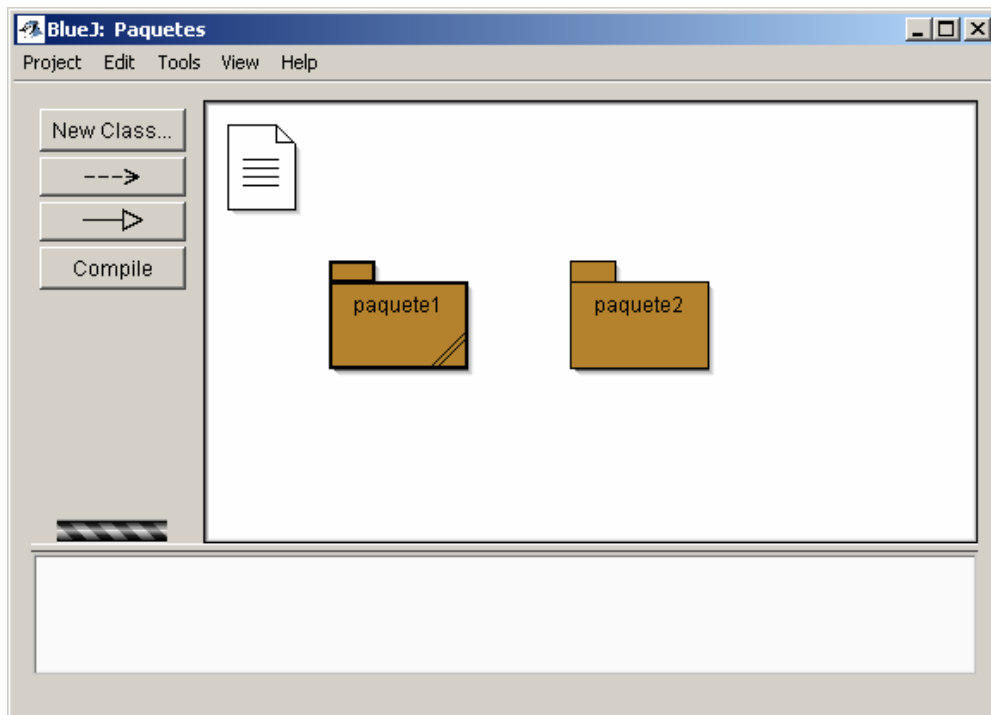
```
| paquete2.OtraClase objeto = new paquete2.OtraClase ();
```

Es posible que las clases que estén dentro de un paquete hereden de clases que forman la parte pública de otro paquete. En este caso, se aplican las normas que ya hemos presentado

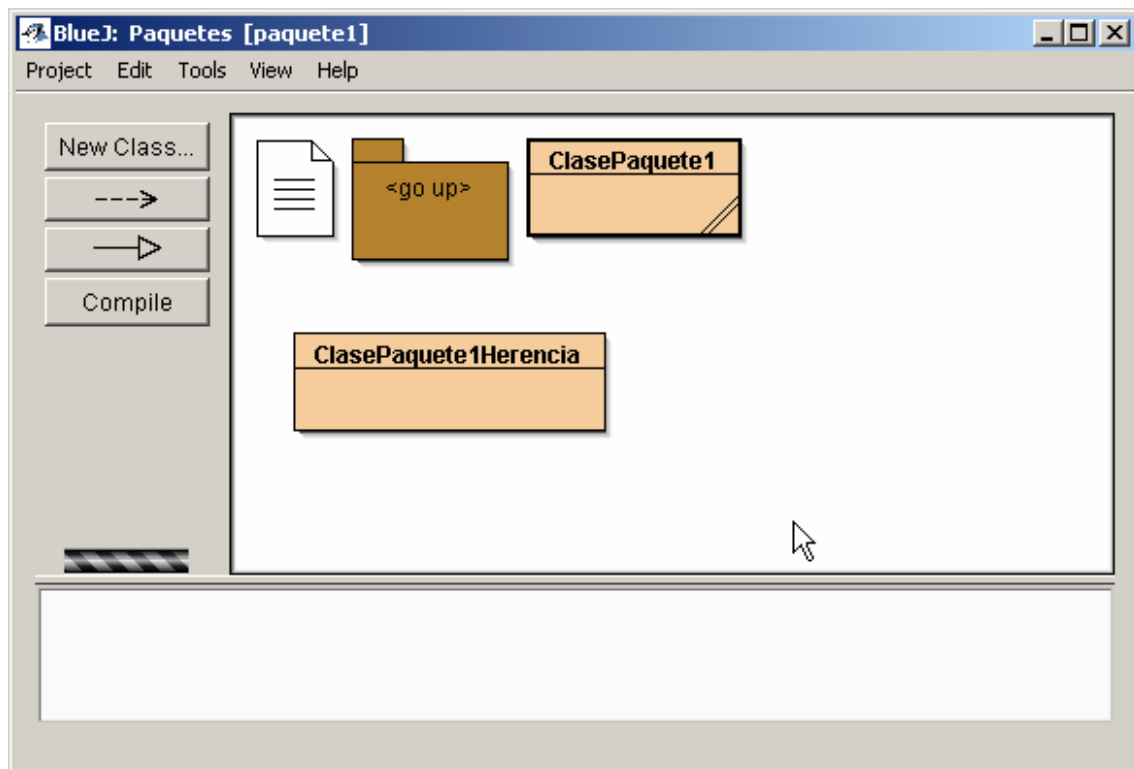
para la herencia: la clase hija podrá acceder a la parte pública, protegida y de visibilidad de paquete de la clase padre.

5.5.1 Un ejemplo de código con paquetes

Vamos a ver un código en el que se pone en un uso los conceptos que estamos presentando aquí. Este código se encuentra en el directorio "Paquetes" del directorio de ejercicios. Dicho directorio es un proyecto de BlueJ. El proyecto contiene dos paquetes, como se muestra en la imagen.

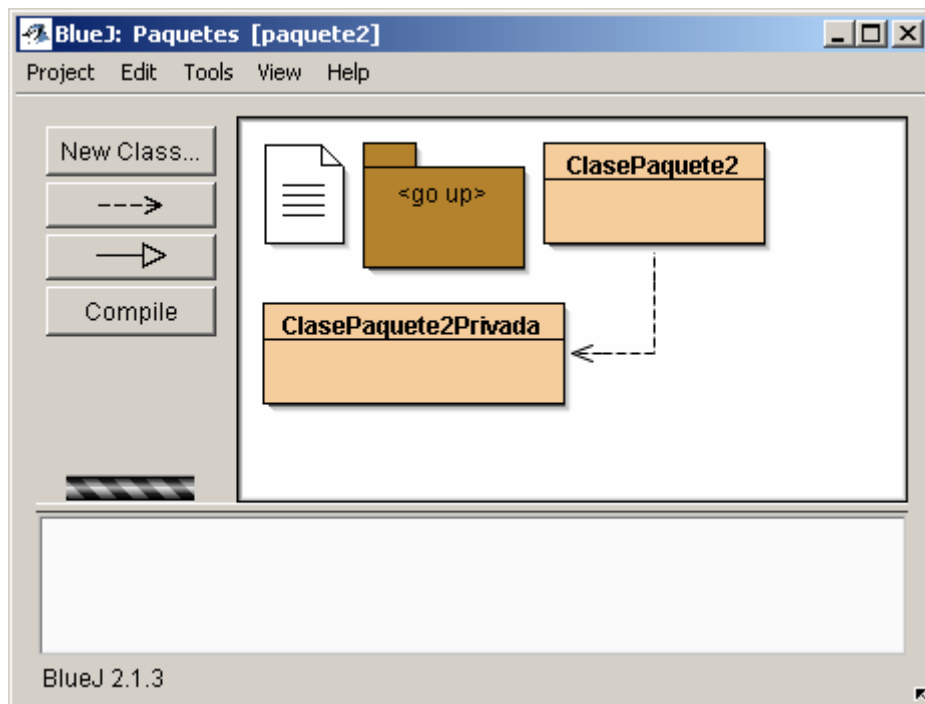


Para crear un paquete desde BlueJ se emplea el menú de edición, en el cual hay una entrada que permite crear paquetes. Para "meternos dentro de un paquete" hacemos doble clic sobre él y podemos ver su contenido. Por ejemplo, el contenido de "paquete1" es:



El icono con forma de paquete y con el texto "go up" permite subir un nivel en el anidamiento de los paquetes. En nuestro caso, permitirá ir al nivel raíz que contiene los dos paquetes. Cuando estamos visualizando el contenido de un paquete si creamos una nueva clase en BlueJ dicha clase se creará dentro de ese paquete. Las clases de este primer paquete son las que van a usar las clases del segundo paquete. En concreto, la clase "ClasePaquete1" empleará a una clase del segundo paquete (creará a una instancia de ella e invocará métodos) y la clase "ClasePaquete1Herencia" heredará de una clase del otro paquete.

El contenido del segundo paquete es:



Cómo podrás deducir a partir de los nombres de las clases, una de ellas es accesible desde fuera y por tanto será la interfaz de "paquete2", mientras que la otra no lo es. Eso sí, como también se muestra en la imagen, la clase accesible desde fuera del paquete emplea a la clase no accesible desde fuera. Esa clase es "un detalle de implementación" de este paquete. Si en el futuro la modificamos, la eliminamos, creamos más clases para repartir sus responsabilidades... ningún código que emplee "paquete2" se dará cuenta de dichos cambios ya que nunca conoció la existencia de esa clase.

Veamos ahora el código de cada una de las clases:

```
package paquete2;

//esta clase tiene visibilidad de paquete, formará parte de
//los detalles de implementación de este paquete.
class ClasePaquete2Privada
{
    void visibilidadPublica()
    {
        System.out.println("Mensaje del método con
visibilidad pública de la clase con visibilidad de paquete");
    }
}
```



```
        void visibilidadPaquete(){
            System.out.println("Mensaje del método con
visibilidad de paquete de la clase con visibilidad de
paquete");
        }

        protected void visibilidadProtegida(){
            System.out.println("Mensaje del método con
visibilidad protegida de la clase con visibilidad de
paquete");
        }

        private void visibilidadPrivada (){
            System.out.println("Mensaje del método privado de
la clase con visibilidad de paquete");
        }

    }
```

```
package paquete2;

/*Esta clase es pública, por lo tanto formará parte de la
interfaz del paquete*/

public class ClasePaquete2
{

    public void saludar()
    {
        System.out.println("Hola");
        //por supuesto, la clase puede acceder a sus
métodos privados
        this.privado();
        //protegidos
        this.visibilidadProtegida();
        //y de paquete
        this.visibilidadPaquete();
    }
}
```

```
//aquí usamos los "detalles de implementación" del
paquete

    ClasePaquete2Privada    objeto2    =    new
ClasePaquete2Privada();

    //por supuesto, pueda acceder a su parte pública
    objeto2.visibilidadPublica();

    //y, como estoy en el mismo paquete, a la parte
con visibilidad de paquete
    objeto2.visibilidadPaquete ();

    //también a la parte protegida porque estamos en
el mismo paquete
    objeto2.visibilidadProtegida ();

    //pero no la privada
    //
    objeto2.visibilidadPrivada ();
}

void visibilidadPaquete(){
    System.out.println("Mensaje del método con
visibilidad de paquete");
}

protected void visibilidadProtegida(){
    System.out.println("Mensaje del método con
visibilidad protegida");
}

private void privado (){
    System.out.println("Mensaje del método privado");
}
}
```

Ahora veamos el contenido de "paquete1". Sus clases tienen métodos main y, por tanto, se pueden ejecutar. Las dos clases de este paquete emplean la clase pública del paquete anterior, o bien porque crea un objeto de ella o bien porque heredan de ella.

```
package paquetel;

//para poder usar la clase del otro paquete
import paquete2.*;

public class ClasePaquetel
{
    public static void main (String[] args){
        //accedemos a la clase pública del otro paquete
        ClasePaquete2 objeto = new ClasePaquete2();
        //si quitas el comentario de esta línea obtendrás
un error al compilar
        //esta clase tenía visibilidad de paquete y no
puede ser cedida desde aquí
        //      ClasePrivadaPaquete2      objeto      =      new
ClasePrivadaPaquete2();
        //por supuesto, se siguen cumpliendo las normas de
siempre
        //para el acceso a las partes públicas, privadas,
protegidas y con visibilidad de paquete
        objeto.saludar();
        //no puedo acceder al método privado:
        //      objeto.privado ();
        //ni al que tiene visibilidad de paquete
        //      objeto.visibilidadPaquete ();
        //ni, por supuesto, al privado
        //      objeto.visibilidadPrivada()
    }
}
```

```
package paquetel;

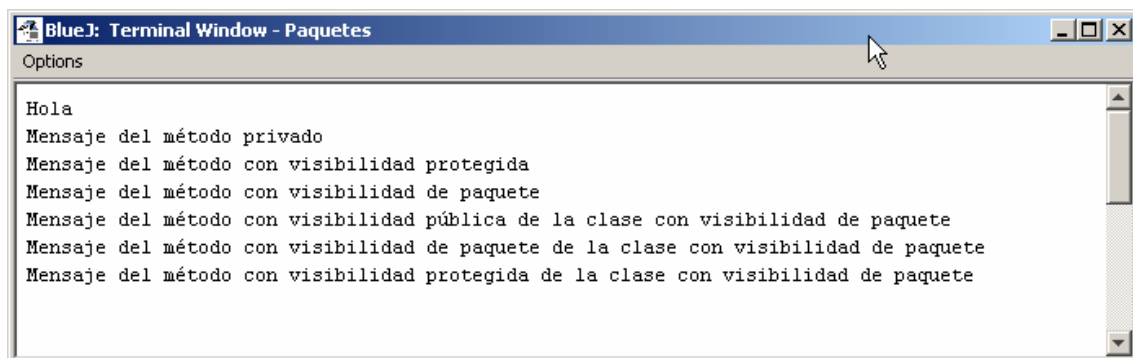
import paquete2.*;

//la clase hereda de ClasePaquete2, por tanto va a poder
//acceder a sus partes protegidas y con visibilidad de
paquete
```

```
//además de, por supuesto, a la parte pública
public class ClasePaquete1Herencia extends ClasePaquete2
{

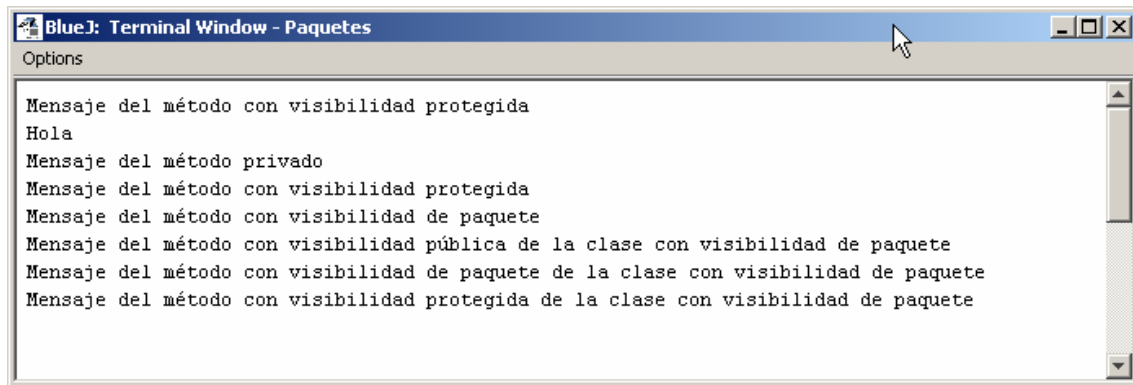
    public static void main (String[] args){
        //creamos una instancia de esta clase. No la hemos
dotado de ningún método,
        //pero podremos acceder a los métodos que hemos
heredado de ClasePaquete2
        ClasePaquete1Herencia      objeto      =      new
ClasePaquete1Herencia();
        //no puedo acceder al método privado:
//      objeto.visibilidadPrivada()
//ni al que tiene visibilidad de paquete
//      objeto.visibilidadPaquete();
//pero si al protegido
objeto.visibilidadProtegida();
objeto.saludar();
    }
}
```

La salida que produce la ejecución del método main de "ClasePaquete1" es:



```
BlueJ: Terminal Window - Paquetes
Options
Hola
Mensaje del método privado
Mensaje del método con visibilidad protegida
Mensaje del método con visibilidad de paquete
Mensaje del método con visibilidad pública de la clase con visibilidad de paquete
Mensaje del método con visibilidad de paquete de la clase con visibilidad de paquete
```

y la que produce el método main de " ClasePaquete1Herencia" será:

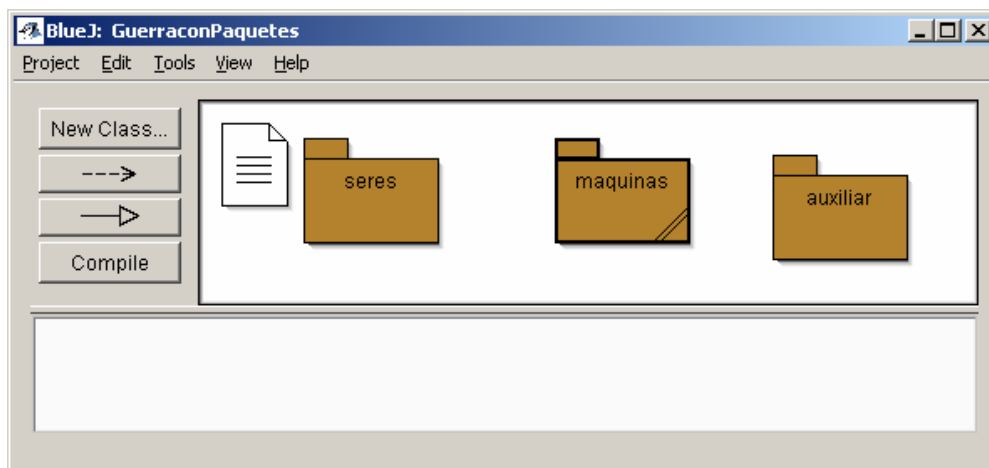


Si el lector desea borrar de vez en cuando la salida de la consola del programa BlueJ encontrará una entrada en el menú de opciones que le permite realizar esta acción.

5.6 EL EJEMPLO DE LOS MARCIANOS CON PAQUETES

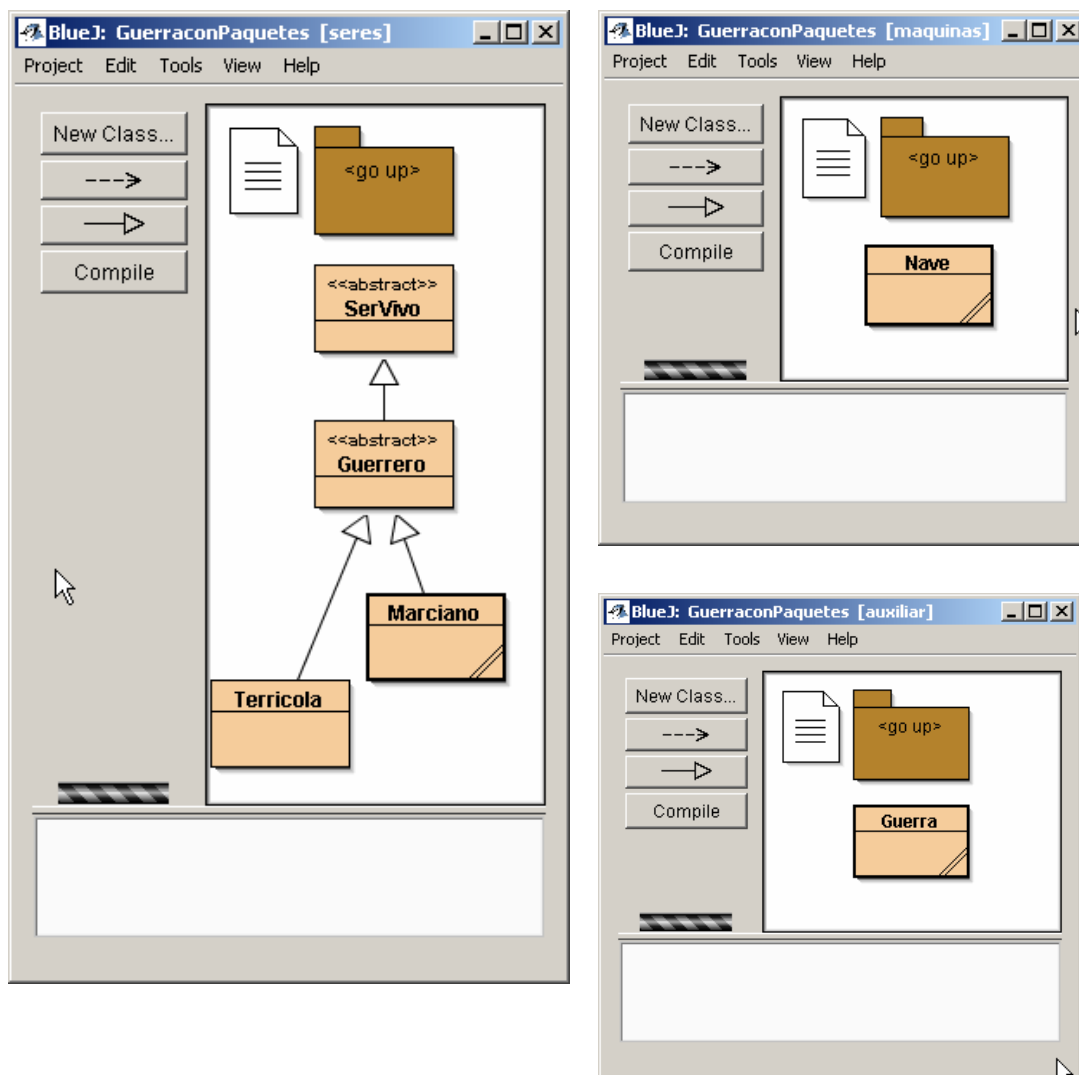
En esta sección vamos a reorganizar el código de la guerra entre marcianos y terrícolas para emplear paquetes. La principal funcionalidad de los paquetes es precisamente ayudar a organizar programas complejos. El cómo organizar un código en paquetes depende completamente del propósito de dicho código y de cómo se quiera diseñar el software. Con el tiempo irás ganando experiencia e irás aprendiendo a organizar tus clases en paquetes de modo adecuado.

Para un ejemplo tan trivial como éste podría perfectamente argumentarse que con emplear un único paquete es más que suficiente. No obstante, vamos a reorganizar el código empleando tres paquetes. Uno de los paquetes lo llamaremos "seres" porque en él será donde vayan todos los objetos del universo que consideramos "seres". En nuestro caso, irán Marciano, Terrícola, Guerrero y SerVivo. Crearemos un segundo paquete llamado "máquinas" donde irían todas las máquinas que implementásemos. En nuestro caso, la única máquina en el problema es Nave. Por último, creamos un paquete "auxiliar" en la cual colocamos la clase que contiene el método main.



Los únicos cambios que será necesario hacer en el código es incluir las sentencias que indican a qué paquete pertenece cada clase, colocar cada clase en el directorio adecuado y, cuando una clase emplee clases de otros paquetes diferentes al suyo, añadir la sentencia import correspondiente. Aunque a continuación vamos a mostrar el código fuente del ejemplo modificado, recomiendo al lector que, a modo de ejercicio, coja el código que ya conoce (el que podrá encontrar en el directorio "Guerra") e intente realizar las transformaciones que he descrito. El código modificado se encuentra en el directorio "GuerraconPaquetes".

La apariencia de los tres paquetes que forman el programa será:



Y el código fuente (última oportunidad para abandonar los apuntes e intentar realizar los cambios por tu cuenta sin antes mirar) es:

```
package seres;

//No necesitamos importar nada porque no usamos clases de
otros paquetes

public abstract class SerVivo {

    public boolean isVivo(){
        return vivo;
    }

    protected boolean vivo = true;
}
```

```
package seres;

//No necesitamos importar nada porque no usamos clases de
otros paquetes

public abstract class Guerrero extends SerVivo {

    public Guerrero (String soy){
        blanco = generaBlanco();
        this.soy = soy;
    }

    public int dispara (){
        if (vivo){
            int disparo = ((int)(Math.random()*10));
            System.out.println(soy + "Dispara n° "
+disparo);
            return disparo;
        }
    }
}
```

```
        else
            return 100;
    }

    public int getBlanco(){
        return blanco;
    }

    private int generaBlanco (){
        return ((int)(Math.random()*10));
    }

    protected int blanco;
    private final String soy;
}
```

```
package seres;

//No necesitamos importar nada porque no usamos clases de
otros paquetes

public class Terricola extends Guerrero {
    //ahora el constructor tiene que ser público para el clases
de otros
    //paquetes que no hereden ésta (Nave) puedan crear
instancias de Terricola
    public Terricola(String soy){
        super(soy);

        total++;
        this.soy= soy;
    }

    public void recibeDisparo(int i){
        if (vivo && blanco == i){
            vivo = false;
        }
    }
}
```



```
        total--;
        System.out.println (soy + " Muerto por disparo
nº " +i);
    }
}

public int getTotal(){
    return total;
}
private static int total = 0;
private String soy;

}
```

```
package seres;

//No necesitamos importar nada porque no usamos clases de
otros paquetes

public class Marciano extends Guerrero {
    //ahora el constructor tiene que ser público para el clases
de otros
    //paquetes que no hereden ésta (Nave) puedan crear
instancias de Marciano
    public Marciano(String soy){
        super(soy);
        this.soy =soy;
        total++;
    }

    public void recibeDisparo(int i){
        if (vivo && blanco == i){
            vivo = false;
            total--;
        }
    }
}
```

```
        System.out.println (soy + "Muerto por disparo  
nº " +i);  
    }  
}  
  
    public int getTotal(){  
        return total;  
    }  
    private static int total = 0;  
    private String soy;  
  
}
```

```
package maquinas;  
  
//necesitamos emplear clases del paquete seres  
import seres.*;  
  
public class Nave {  
    public Nave (String somos){  
        this.somos = somos;  
        for (int i = 0; i<10; i++){  
            if (somos.equals("Terricolas")){  
                tripulacion [i] = new Terricola(somos);  
            }  
            else{  
                tripulacion [i] = new Marciano(somos);  
            }  
        }  
        System.out.println("Creada nave de " + somos);  
    }  
  
    public void recibeDisparo(int i){  
        for (int j=0; j<10;j++){  
            if (somos.equals("Terricolas")){
```

```
((Terricola)(tripulacion[j])).recibeDisparo(i);
    }
    else{

((Marciano)(tripulacion[j])).recibeDisparo(i);

    }

}

}

public int generaDisparo(int i){
    return tripulacion[i].dispara();
}

public int cuantosQuedan(){
    if (somos.equals("Terricolas")){
        return
((Terricola)(tripulacion[1])).getTotal();
    }
    else{
        return ((Marciano)(tripulacion[1])).getTotal();
    }
}

private Guerrero[] tripulacion = new Guerrero[10];
private String somos;

}
```

```
package auxiliar;

//necesitamos emplear la clase Nave
import maquinas.*;

public class Guerra {
    public Guerra(){
        navel = new Nave("Terricolas");
        nave2 = new Nave("Marcianos");
    }
}
```

```
        empiezaGuerra();
    }

    public void empiezaGuerra(){
        do{
            for(int i = 0; i<10;i++){
                nave2.recibeDisparo(nave1.generaDisparo(i));
                nave1.recibeDisparo(nave2.generaDisparo(i));
            }
        }while(nave1.cuantosQuedan(>0)&& nave2.cuantosQuedan(>0);
        if(nave1.cuantosQuedan(>0)){
            System.out.println("GANARON LOS
TERRICOLAS!!!!");
        }
        else if (nave2.cuantosQuedan(>0){
            System.out.println("GANARON LOS MARCIANOS");
        }
    }
    public static void main(String[] args){
        new Guerra();
    }

    private Nave nave1, nave2;
}
```

6 PROGRAMACIÓN GRÁFICA CON SWING

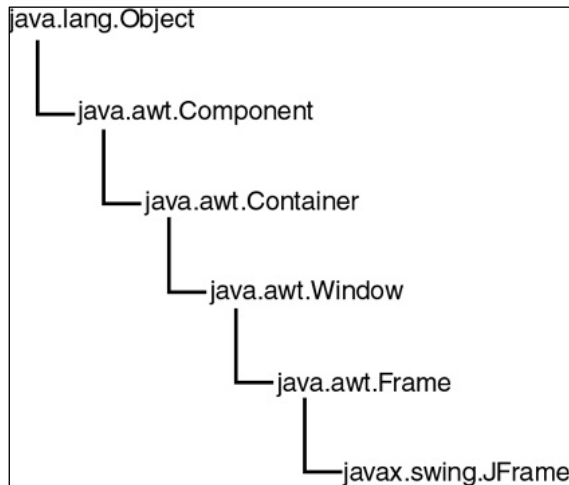
El objetivo de este tema es enseñar a diseñar pequeñas interfaces gráficas empleando para ello las librerías gráficas Swing. Una interfaz es lo que le permite a un usuario comunicarse con un programa, una interfaz es lo que nosotros vemos al arrancar, por ejemplo, un navegador de internet: un conjunto de menús, botones, barras.... que nos permiten activar un código que es el que realmente nos llevará a una página web, salvará la imagen en el disco duro....

Las librerías gráficas que usaremos vienen a sustituir a las antiguas AWT. Las nuevas librerías a parte de tener una mayor cantidad de opciones sobre los componentes (como distintas apariencias, control sobre el focus, mayor número de campos que modifican su aspecto, mayor facilidad para pintar al hacer el buffering transparente al usuario....) se diferencian de las anteriores radicalmente en su implementación.

En AWT cuando añadíamos un botón, por ejemplo, a nuestro diseño el compilador generaba código que le pedía al sistema operativo la creación de un botón en un determinado sitio con unas determinadas propiedades; en Swing ya no se pide al sistema operativo nada: se dibuja el botón sobre la ventana en la que lo queríamos. Con esto se eliminaron muchos problemas que existían antes con los códigos de las interfaces gráficas, que debido a depender del sistema operativo para obtener sus componentes gráficos, era necesario testar los programas en distintos sistemas operativos, pudiendo tener distintos bugs en cada uno de ellos.

Esto evidentemente iba en contra de la filosofía de Java, supuestamente un lenguaje que no dependía de la plataforma. Con Swing se mejoró bastante este aspecto: lo único que se pide al sistema operativa es una ventana, una vez que tenemos la ventana dibujamos botones, listas, scroll-bars... y todo lo que necesitemos sobre ella. Evidentemente esta aproximación gana mucho en lo que a independencia de la plataforma se refiere. Además el hecho de que el botón no sea un botón del S.O. sino un botón pintado por Java nos da un mayor control sobre su apariencia.

6.1 JFRAME



Es el contenedor que emplearemos para situar en él todos los demás componentes que necesitemos para el desarrollo de la interface de nuestro programa.

En el gráfico 1 se muestra la jerarquía de herencia de este componente desde Object, que como ya dijimos es el padre de todas las clases de Java. Los métodos de este componente estarán repartidos a lo largo de todos sus ascendientes, cosa que hemos de tener en cuenta cuando consultemos la ayuda on-line de esta clase. Así por ejemplo resulta intuitivo que debiera haber un método para cambiar el color de

Graduación de herencia de JFrame 1

fondo del frame, pero él no tiene ningún método para ello, lo tiene Component. Veamos el código necesario para crear un JFrame:

```
//Importamos una librería, un package
import javax.swing.*;

//Nuestra clase Frame extiende a JFrame
class Frame extends JFrame {
    //el constructor
    public Frame(){
        //Este es uno de los métodos que nuestra clase Frame ha
        //heredado de JFrame. Pone un título a la ventana
        setTitle("Hola!!!");
        //Igual que el anterior, pero le esta vez le da un tamaño
        setSize(300,200);
    }
}

//Esta es la clase auxiliar, tiene el main de la aplicación
public class Ejemplo13{
    public static void main (String[] args){
```

```
//Creamos un objeto de tipo Frame
    JFrame frame = new JFrame();
//invoco sobre este objeto uno de los métodos que ha heredado
//de JFrame: show. Los frames por defecto son "invisibles",
//este método los hace visibles.
    frame.show();
}
} ///:~
```

Si embargo nuestro código anterior tiene un problema: no podemos cerrar la ventana. La única forma de acabar con ella será mediante `^c` si hemos ejecutado el programa desde una consola o con `Control-Alt-Supr` y eliminando su tarea correspondiente si lo ejecutamos desde Windows. ¿Por qué no se cierra la ventana? porque no hemos escrito el código necesario para ello. Para que se cierre nuestro frame hemos de escribir un código que escuche los eventos de ventana, y que ante el evento de intentar cerrar la ventana reaccione cerrándose esta. A continuación y antes de seguir con componentes de la librería Swing veamos que es un evento y como gestionarlos.

6.2 EVENTOS

El sistema de gestión de eventos de Java 1.2 es el mismo que Java 1.1 y por lo tanto el mismo que para las librerías AWT. Aunque los desarrolladores de Java considerasen que para mejorar el lenguaje se necesitaba dejar a un lado las librerías AWT e introducir las Swing no sintieron lo mismo del sistema de gestión de eventos, consideraron que era lo suficientemente bueno.

Realmente este sistema de gestión de eventos es bastante elegante y sencillo, sobre todo si se compara con el sistema de gestión de eventos de Java 1.0, mucho más engorroso de usar y menos elegante.

6.2.1 ¿Qué es un evento?

Todos los sistemas operativos están constantemente atendiendo a los eventos generados por los usuarios. Estos eventos pueden ser pulsar una tecla, mover el ratón, hacer clic con el ratón, pulsar el ratón sobre un botón o menú (Java distingue entre simplemente pulsar el ratón en

un sitio cualquiera o hacerlo, por ejemplo, en un botón). El sistema operativo notifica a las aplicaciones que están ocurriendo estos eventos, y ellas deciden si han de responder o no de algún modo a este evento.

6.2.2 El modelo de delegación de eventos

El modelo de Java se basa en la delegación de eventos: el evento se produce en un determinado componente, por ejemplo un scroll. Dónde se produce el evento se denomina "fuente del evento". A continuación el evento se transmite a un "manejador de eventos" (event listener) que este asignado al componente en el que se produjo el evento. El objeto que escucha los eventos es el que se encargará de responder a ellos adecuadamente. Esta separación de código entre generación del evento y actuación respecto a él facilita la labor del programador y da una mayor claridad a los códigos.

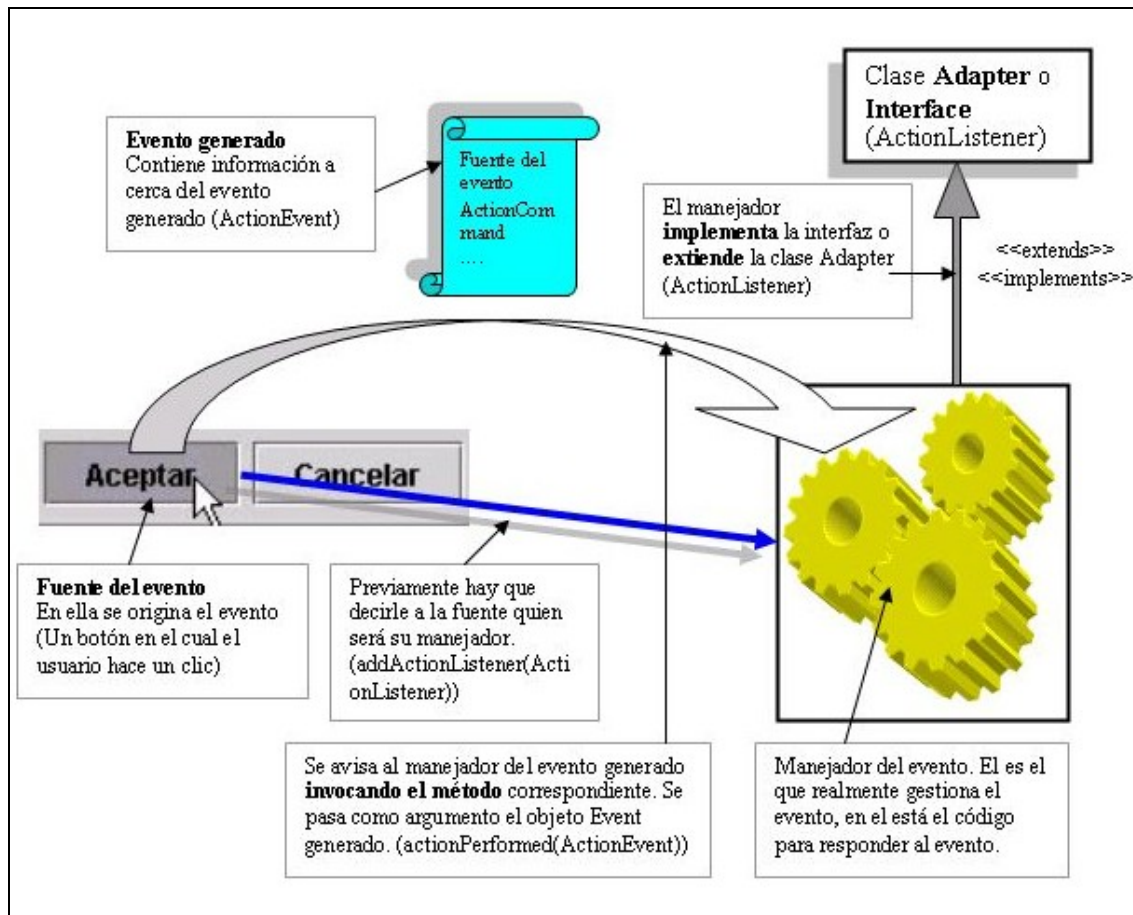


Figura 6.- Gestión de eventos en Java. A la *fente* del evento, en este caso un botón, le indicamos quién será su *manejador* de eventos, manejador que ha de *extender la clase Adapter* correspondiente o implementar *la interfaz Listener* (interfaz `ActionLitener` en este caso). Cuando el usuario genere el evento deseado (en este caso pulse el botón), el objeto fuente empaqueta información acerca del evento generando un objeto de tipo `Event` (`ActionEvent` en este caso) e invoca el método correspondiente del manejador (`actionPerformed(actionEvent)`) pasándole como información el objeto de tipo `Event` generado. Es responsabilidad del manejador, y no de la fuente, responder al evento, por ello se dice que la fuente delega la gestión del evento en el manejador.

Lo que la fuente de eventos le pasa al objeto encargado de escuchar los eventos es, como no, otro objeto. Es un objeto tipo Event. En este objeto va toda la información necesaria para la correcta gestión del evento por parte del objeto que escucha los eventos.

El objeto que escucha los eventos ha de implementar para ello una interface. El nombre de esta interface es siempre el nombre del evento más "Listener": para que un objeto escuche eventos de ratón ha de implementar la interface MouseListener, para que escuche eventos de teclado KeyListener.....

Para hacer que un objeto escuche los eventos de otro objeto se emplea el método add[nombre_evento]Listener, así si tuviésemos un JFrame llamado "frame" y quisiésemos que el objeto llamado "manejador" escuchase los eventos de ratón de "frame" lo haríamos del siguiente modo:

```
frame.addMouseListener(manejador);
```

manejador ha de pertenecer a una clase que implemente la interface MouseListener, que tiene un total de 7 métodos que ha de implementar.

A continuación en la siguiente tabla mostramos los eventos más comunes, junto a la interface que debe implementar el objeto que escuche esos eventos y el método para asociar un objeto para escuchar dichos eventos. En la columna de la derecha se presentarán diversos componentes que pueden generar dichos eventos.

Event, listener interface y métodos para ligar el objeto que escucha	Algunos componentes que generan este tipo de eventos.
ActionEvent ActionListener addActionListener()	JButton, JList, JTextField, JMenuItem, JCheckBoxMenuItem, JMenu, JpopupMenu.
AdjustmentEvent AdjustmentListener addAdjustmentListener()	JScrollbar y cualquier objeto que implemente la interface Adjustable.
ComponentEvent ComponentListener addComponentListener()	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel,

Event, listener interface y métodos para ligar el objeto que escucha	Algunos componentes que generan este tipo de eventos.
	JList, JScrollbar, JTextArea, JTextField.
ContainerEvent ContainerListener addContainerListener()	Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame.
FocusEvent FocusListener addFocusListener()	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
KeyEvent KeyListener addKeyListener()	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
MouseEvent MouseListener addMouseListener()	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
MouseEvent MouseMotionListener addMouseMotionListener()	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
WindowEvent WindowListener addWindowListener()	Window, JDialog, JFileDialog, and JFrame.
ItemEvent ItemListener addItemListener()	JCheckBox, JCheckBoxMenuItem, JComboBox, Jlist.
TextEvent TextListener addTextListener()	JTextComponent, JtextArea, JTextField.

Event, listener interface y métodos para ligar el objeto que escucha	Algunos componentes que generan este tipo de eventos.

Cabe preguntarse ahora por que métodos tiene cada interface, ya que hemos de implementar todos ellos, incluso aunque no los usemos, sino la clase que se encargaría de escuchar los eventos sería abstracta y no podríamos crear ningún objeto de ella. Parece un poco estúpido implementar métodos que no hagan nada sólo porque la interface de la que heredamos los tenga. Así por ejemplo si estamos interesados en escuchar clics de ratón hemos de crear una clase que implemente `MouseListener`, pero nosotros sólo estaremos interesados en un método de dicha interfase: `mouseClicked`.

Los creadores de Java también pensaron en esto y por ello para cada interface que tiene más de un método crearon una clase llamada `[nombre_evento]Adapter` (`MouseAdapter`), que lo que hace es implementar todos los métodos de la interface sin hacer nada en ellos. Nosotros lo único que tendremos que hacer es que nuestra clase que escuche eventos extienda esta clase y sobrescriba los métodos que nos interesen.

A continuación en la siguiente tabla damos un listado de las principales interfaces junto a sus respectivas clases "Adapter" y los métodos que poseen:

Listener interface y Adapter	Metodos
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent)

Listener interface y Adapter	Metodos
	mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

6.2.3 Un frame que se cierra

Pongamos en práctica lo que hemos visto haciendo un frame que se pueda cerrar:

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

class Frame extends JFrame {
    public Frame(){
        setTitle("Hola!!!");
        setSize(300,200);
        //Le indicamos al Frame quien será su manejador de eventos de
        //ventana: un objeto de tipo manejador que creamos en esta
        misma línea
        addWindowListener (new manejador());
    }
}
```

```
/*Clase manejadora de eventos de ventana. Implementa el
*inteface WindowListener, por lo que ha de sobrescribir
*todos sus métodos*/
class manejador implements WindowListener{
    public void windowClosing(WindowEvent e){
        System.out.println("sali");
//Esta sentencia termina la máquina virtual
        System.exit(0);
    }
//Métodos que no hacen nada,pero que he de sobrescribir por
//implementar el interface
    public void windowOpened(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}

}

public class Ejemplo15{
    public static void main (String[] args){
        JFrame frame = new Frame();
        frame.show();
    }
} ///:~
```

Aquí se ve como al hacer que la clase manejador implemente la interface MouseListener hemos de implementar sus siete métodos aunque sólo nos interesa el que está relacionado con el cierre de la ventana. A continuación rescribimos el ejemplo pero aprovechando las ventajas de las clases Adapter:

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

```

class Frame extends JFrame {
    public Frame(){
        setTitle("Hola!!!");
        setSize(300,200);
        //Igual que antes le indico a la ventana quién será su
        //manejador de eventos de ventana.
        addWindowListener (new manejador());
    }
}

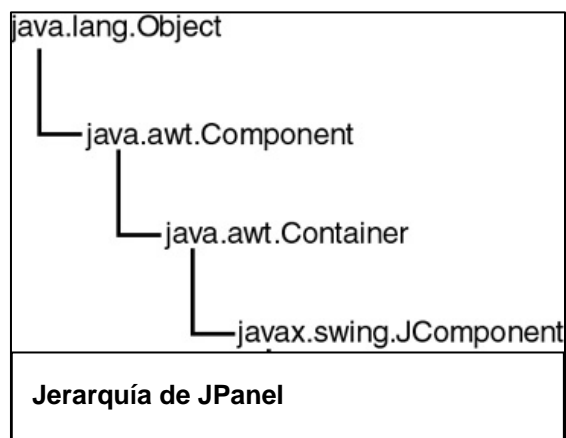
/**Esta vez la clase manejador extiende la clase adapter, por
*lo que sólo tengo que sobrescribir el método en el que
*estoy interesado*/
class manejador extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.out.println("sali");
        System.exit(0);
    }
}

public class Ejemplo16{
    public static void main (String[] args){
        JFrame frame = new Frame();
        frame.show();
    }
} ///:~

```

6.3 JPANEL

Ahora ya sabemos hacer una ventana (frame) que se cierra. Podríamos empezar a añadirle botones, scrolls, campos de texto ... y todo lo que necesitemos, pero no es considerado una buena práctica de programación añadir componentes directamente sobre un contenedor de "pesado" (frames y applets por



lo que a nosotros respecta). Lo correcto es añadir a este uno o varios paneles y añadir sobre los paneles lo que necesitemos.

Una de las ventajas de añadir paneles sobre nuestro frame es que los paneles al derivar de JComponent poseen el método paintComponent que permite dibujar y escribir texto sobre el panel de modo sencillo.

Para añadir un JPanel a nuestro frame primero obtenemos uno de los objetos que forman el frame: el "panel contenedor" (content pane). Para ello invocaremos al método getContentPane de nuestro JFrame. El objeto que nos devuelve será de tipo Container:

```
|         Container [nombre_del_contentpane] =  
| frame.getContentPane();
```

A continuación invocamos al método add del Container obtenido para añadir el panel, pasándole el propio panel al método:

```
|         [nombre_del_contentpane].add(nombre_del_panel);
```

Añadámosle un JPanel a nuestro frame:

```
| import javax.swing.*;  
| import java.awt.event.*;  
| import java.awt.*;  
  
| class Frame extends JFrame {  
|     public Frame(){  
|         setTitle("Hola!!!");  
|         setSize(300,200);  
|         addWindowListener (new manejador());  
|         //Le pido al Frame su objeto contenedor  
|         Container contentpane = getContentPane();  
|         //Creo un objeto de tipo JPanel  
|         JPanel panel = new JPanel();  
|         //Añado el panel en el objeto contenedor del frame  
|         contentpane.add(panel);  
|         //Pongo el color de fondo del panel de color rojo
```

```
        panel.setBackground(Color.red);

    }

}

class manejador extends WindowAdapter{

    public void windowClosing(WindowEvent e){

        System.out.println("sali");

        System.exit(0);

    }

}

public class Ejemplo17{

    public static void main (String[] args){

        JFrame frame = new Frame();

        frame.show();

    }

} ///:~
```

6.4 LAYOUT

Ya ha llegado casi el momento de empezar a añadir cosas a nuestra ventana, sólo una cosa queda pendiente: como controlar dónde añadimos los objetos. Por ejemplo como decirle a nuestro panel dónde tiene que colocar un botón, por ejemplo.

Una solución sería indicarle dónde colocar la esquina izquierda de arriba del botón y luego indicando el alto y ancho del botón. Esto es precisamente lo que hace el método `setBounds(int,int,int,int)` de la clase `Component`. Parece, en principio, una buena solución. Hagámoslo:

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

class Frame extends JFrame {
```



```
public Frame(){
    setTitle("Hola!!!");
    setSize(500,400);
    addWindowListener (new manejador());
    Container contentpane = getContentPane();
    JPanel panel = new JPanel();
    //Elimino el gestor de layouts del panel
    panel.setLayout(null);
    //Creo un objeto de tipo JButton (un botón)
    JButton boton = new JButton();
    //Mediante este método le indico al botón que se sitúe en las
    //coordenadas 300,300 del panel, con un tamaño de 50x50
    boton.setBounds(300,300,50,50);
    //Añado el botón al panel
    panel.add(boton);
    contentpane.add(panel);
    panel.setBackground(Color.red);
}
}

class manejador extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.out.println("sali");
        System.exit(0);
    }
}

public class Ejemplo18{
    public static void main (String[] args){
        JFrame frame = new Frame();
        frame.show();
    }
} ///:~
```

Si con este código probamos a cambiar de tamaño la ventana podremos ver como el botón no se mueve, desapareciendo si la ventana se hace muy pequeña y cambiando su posición relativa a los bordes de la ventana. Esto, en una aplicación real, desorientaría al usuario que nunca sabría dónde irlo a buscar al cambiar el tamaño de la ventana.

Para solucionar este inconveniente se crearon los Layout Manager: con ellos se especifican unas posiciones determinadas en un panel, frame o applet donde añadiremos nuestros componentes o un nuevo panel, al que también le podremos añadir un layout en cuyas posiciones podremos añadir componentes o más panels con layouts....

La posibilidad de añadir varios panels a un layout y fijar a estos nuevos layouts da un gran dinamismo a la hora de colocar los componentes.

6.4.1 *FlowLayout*

Es el que tienen los paneles por defecto. Los objetos se van colocando en filas en el mismo orden en que se añadieron al contenedor. Cuando se llena una fila se pasa a la siguiente. Tiene tres posibles constructores:

```
|      FlowLayout ( ) ;
```

Crea el layout sin añadirle los componentes, con los bordes de unos pegados a otros.

```
|      FlowLayout ( FlowLayout . LEFT [ RIGHT ] [ CENTER ] ) ;
```

Indica la alineación de los componentes: a la izquierda, derecha o centro.

```
|      FlowLayout ( FlowLayout . LEFT , gap_horizontal ,  
|      gap_vertical ) ;
```

Además de la alineación de los componentes indica un espaciado (gap) entre los distintos componentes, de tal modo que no aparecen unos pegados a otros.

He aquí un ejemplo del uso de FlowLayout:

```
import javax.swing.*;
import java.awt.*;

class panel extends JFrame{

    public panel() {
        setSize(300,200);
        Container container = this.getContentPane();
        FlowLayout fl = new FlowLayout(FlowLayout.LEFT, 5,
10);
        container.setLayout(fl);
        for (int i=0; i<4; i++) {
            JButton button = new JButton("Button"+(i+1));
            button.setPreferredSize(new Dimension(100,25));
            container.add(button);
        }
    }
}

public class Ejemplo19{
    public static void main (String[] args){
        panel t = new panel();
        t.show();
    }
} ///:~
```

6.4.2 GridLayout

Como su propio nombre indica crea un grid (malla) y va añadiendo los componentes a las cuadrículas de la malla de izquierda a derecha y de arriba abajo. Todas las cuadrículas serán del mismo tamaño y crecerán o se harán más pequeñas hasta ocupar toda el área del contenedor. Hay dos posibles constructores:

```
GridLayout(int filas, int columnas);
```

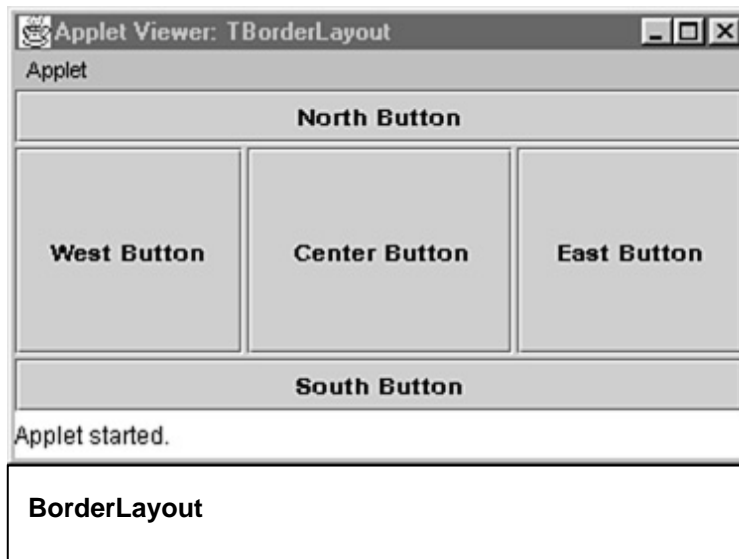
Crearé un layout en forma de malla con un número de columnas y filas igual al especificado.

```
        GridLayout(int columnas, int filas, int  
        gap_horizontal, int gap_vertical);
```

Especifica espaciados verticales y horizontales entre las cuadrículas. El espaciado se mide en píxeles. Veamos un ejemplo:

```
import javax.swing.*;  
import java.awt.event.*;  
import java.awt.*;  
  
class panel extends JFrame {  
    Container container = null;  
  
    public panel() {  
        setSize(350,150);  
        container = this.getContentPane();  
        GridLayout grid = new GridLayout(3,2, 5,5);  
        // filas = 3, columnas = 2, horizontal gap =5,  
        //vertical gap = 5  
        container.setLayout(grid);  
  
        for (int i=0; i<6; i++)  
            container.add(new JButton("Button"+(i+1)));  
    }  
}  
  
public class Ejemplo20{  
    public static void main (String[] args){  
        panel t = new panel();  
        t.show();  
    }  
} ///:~
```

6.4.3 BorderLayout



Este layout tiene cinco zonas predeterminadas: son norte (NORTH), sur (SOUTH), este (EAST), oeste (WEST) y centro (CENTER). Las zonas norte y sur al cambiar el tamaño del contenedor se estirarán hacia los lados para llegar a ocupar toda el área disponible, pero sin variar su tamaño en la dirección vertical. Las zonas este y oeste presentan el

comportamiento contrario: variarán su tamaño en la dirección vertical pero sin nunca variarlo en la dirección horizontal.

En cuanto a la zona central crecerá o disminuirá en todas las direcciones para rellenar todo el espacio vertical y horizontal que queda entre las zonas norte, sur, este y oeste.

Posee dos constructores:

```
| BorderLayout ( ) ;
```

que creará el layout sin más y

```
| BorderLayout(int gap_horizontal, int gap_vertical);
```

Crearé el layout dejando los gaps horizontales y verticales entre sus distintas zonas.

A la hora de añadir más paneles o componentes a este Layout hay una pequeña diferencia respecto a los otros dos: en los otros íbamos añadiendo componentes y el los iba situando en un determinado orden, aquí especificamos en el método add la región donde queremos añadir el componente:

```
| panel.add(componente_a_añadir, BorderLayout.NORTH);
```

Con esta llamada al método add añadiremos el componente en la área norte. Cambiando NORTH por SOUTH, EAST, WEST, CENTER lo añadiremos en la región correspondiente.

Veamos un ejemplo:

```
import java.awt.*;
import javax.swing.*;

class panel extends JFrame {

    public panel() {
        setSize(400,250);
        Container container = this.getContentPane();
        container.setLayout(new BorderLayout(2,2));
        // ((BorderLayout) container.getLayout()).setHgap(2);
        //(((BorderLayout) container.getLayout()).setVgap(2);

        String[] borderConsts = { BorderLayout.NORTH,
                                   BorderLayout.SOUTH,
                                   BorderLayout.EAST,
                                   BorderLayout.WEST,
                                   BorderLayout.CENTER };

        String[]  buttonNames  = {  "North  Button", "South
Button",

                                   "East Button", "West Button",
                                   "Center Button" };

        for (int i=0; i<borderConsts.length; i++) {
            JButton button = new JButton(buttonNames[i]);
            container.add(button, borderConsts[i]);
        }
    }
}

public class Ejemplo21{
    public static void main (String[] args){
```

```
        panel t = new panel();  
        t.show();  
    }  
} ///:~
```

6.5 JBUTTON

Ha llegado el momento de introducir un componente que no sea un mero contenedor. Empecemos por un botón. Crear un botón es tan sencillo como:

```
JButton boton = new JButton();
```

Si queremos que el botón aparezca con una etiqueta de texto:

```
JButton boton = new JButton("texto va aquí");
```

Veamos como añadirle funcionalidad a nuestro botón. Cada vez que hacemos clic sobre el botón se genera un evento del tipo `ActionEvent`. Para poder escuchar dichos eventos necesitaremos una clase que implemente la interface `ActionListener`, interface que tiene un solo método `actionPerformed (ActionEvent)`.

Haremos, por ejemplo que al hacer un clic sobre nuestro botón cambie el color de fondo de un panel. Reutilizaremos nuestro viejo ejemplo 17. Le pondremos un `BorderLayout` y haremos que el panel sea quien escuche los eventos del botón; para ello ha de implementar la interface `ActionListener`. Cuando se llame al método `actionPerformed` invocaremos al método `setBackground` del panel para cambiarlo a azul.

Por otro lado hemos de indicar que va a ser el frame el que escuche los eventos del botón:

```
boton.addActionListener(this);
```

El código resultante será:

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

//Ahora la clase frame implementa el interface
//ActionListener, para poder gestionar eventos de tipo
//ActionEvent
class Frame extends JFrame implements ActionListener{
//Ahora el panel lo he definido como una variable de la clase
//no local de un método. De este modo podré acceder a ella en
//cualquier método de la clase.
    JPanel panel = new JPanel();

    public Frame(){
        setTitle("Hola!!!");
        setSize(500,400);
        addWindowListener (new manejador());
        Container contentpane = getContentPane();
//Le pongo al panel un BorderLayout
        panel.setLayout(new BorderLayout());
        JButton boton = new JButton("Azul");
//Le indico al botón quien será su gestor de eventos de
//ventana: este propio objeto (this), es decir el frame
        boton.addActionListener(this);
//Creo un objeto de tipo dimensión, un objeto que contiene un
//par de valores enteros
        Dimension d = new Dimension();
//Inicializo ese par de valores enteros
        d.height = 40;
        d.width = 100;
//Le pongo al botón un tamaño preferido, empleando para ello
//el objeto dimensión que he creado. El BorderLayout
//respetará el alto preferido del botón al estar éste en su
//campo sur.
        boton.setPreferredSize(d);
//añado el layout al campo sur del panel
        panel.add(boton,BorderLayout.SOUTH);
        contentpane.add(panel);
```



```
        panel.setBackground(Color.red);

    }

    //La clase frame ha de sobrescribir este método, ya que
    //implementa la interface ActionListener
    public void actionPerformed (ActionEvent e){
        panel.setBackground(Color.blue);
    }
}

class manejador extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.out.println("sali");
        System.exit(0);
    }
}

public class Ejemplo22{
    public static void main (String[] args){
        JFrame frame = new Frame();
        frame.show();
    }
} ///:~
```

Haremos una aclaración sobre lo que se ha hecho en el ejemplo; se invoca a un método del botón, `setPreferredSize(Dimension)`. Este método fija un par de constantes del objeto botón a los valores indicados por el objeto `Dimension`, que no es más que un objeto que contiene dos enteros. Estos valores son el tamaño que preferentemente ha de tener el botón. `BorderLayout` respetará la altura, ya que al añadirlo en la posición sur no crecerá en esta dimensión. Si el botón se hubiese añadido en las posiciones este u oeste, donde no crece la anchura sería este el parámetro que respetaría `BorderLayout`.

El motivo de haber hecho esto es que en caso de no hacerlo el botón sería muy fino, con lo cual quedaría poco estético y además no se daría leído la etiqueta.

En cuanto al objeto `panel` no ha sido definido en el constructor porque en caso de haberlo hecho sólo existiría esa variable dentro del constructor y no podríamos invocar al método

`setBackgroundColor` en el método `ActionPerformed` por pertenecer a otro bloque distinto que el de inicialización.

Complicuemos un poco más el problema. Añadamos un botón en las posiciones norte, este y oeste y hagamos que según pulsemos un botón u otro cambie a un color distinto el fondo del panel. Haremos que sea el mismo panel el que escuche los eventos de todos los botones.

El problema que se nos plantea ahora es el siguiente: cada vez que un botón sea pulsando se generará un `ActionEvent` y se invocará al método `ActionPerformed`, pero ¿Cómo sabremos que botón fue accionado para saber de que color ha de tener el fondo del panel?. Esto lo lograremos gracias a que en el objeto evento (`ActionEvent`) hay información sobre quien produjo dicho evento: invocando al método `getSource()` de `ActionEvent` nos devuelve el nombre del componente que generó dicho evento. Así habremos resuelto nuestro problema:

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

class Frame extends JFrame implements ActionListener{
    private JPanel panel = new JPanel();
    //A parte del panel defino cuatro variables puntero a objetos
    //botón como variables de la clase
    private JButton azul,rosa,amarillo,verde;

    public Frame(){
        setTitle("Hola!!!");
        setSize(500,400);
        addWindowListener (new manejador());
        Container contentpane = getContentPane();
        panel.setLayout(new BorderLayout());
        //inicializo la variable azul
        azul = new JButton("Azul");
        //Hago que el frame escuche los eventos del botón azul
        azul.addActionListener(this);
        Dimension d = new Dimension();
        d.height = 40;
```

```
d.width = 100;
azul.setPreferredSize(d);

verde = new JButton("Verde");
verde.addActionListener(this);
verde.setPreferredSize(d);

amarillo = new JButton("Amarillo");
amarillo.addActionListener(this);
amarillo.setPreferredSize(d);

rosa = new JButton("Rosa");
rosa.addActionListener(this);
rosa.setPreferredSize(d);
//Añado los cuatro botones en los campos norte, sur, este y
//oeste del panel
panel.add(azul,BorderLayout.SOUTH);
panel.add(verde,BorderLayout.NORTH);
panel.add(amarillo,BorderLayout.EAST);
panel.add(rosa,BorderLayout.WEST);

contentpane.add(panel);
panel.setBackground(Color.red);

}

public void actionPerformed (ActionEvent e){
//Vamos a emplear información encapsulado en el evento de
//tipo(ActionEvent para saber que botón se pulsó. En el
//puntero source almacenamos la referencia al objeto fuente
//de este evento, es decir la referencia al botón que se
//pulsó.
    Object source = e.getSource();
//Si la referencia es igual a la puntero que apunta la botón
//azul es que fue este el que se pulsó
    if (source ==azul)
        panel.setBackground(Color.blue);
```

```
        if (source ==verde)
            panel.setBackground(Color.green);
        if (source ==amarillo)
            panel.setBackground(Color.yellow);
        if (source ==rosa)
            panel.setBackground(Color.pink);
    }
}

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

public class Ejemplo23{
    public static void main (String[] args){
        JFrame frame = new Frame();
        frame.show();
    }
} //::~~
```

6.6 DIBUJAR EN UNA VENTANA

En este apartado vamos a ver como dibujar sobre una ventana. Esto puede sernos útil tanto para mostrar resultados de un modo gráfico al usuario o para realizar animaciones que ilustren nuestras páginas web (una sorprendente demostración de las capacidades gráficas de java en animaciones la podemos encontrar en <http://www.anfyteam.com/>, página mantenida por una empresa cuyos programadores han recibido numerosos premios por sus desarrollos en java).

Si recordamos lo que ya se comentó al principio del tema, los componentes Swing están “dibujados” sobre una ventana que le pedimos al sistema operativo, se puede decir que “son de la máquina virtual Java” y no del sistema operativo. Esto no ocurría así con los

componentes de la librería AWT, en la cual los componentes “eran del S.O.”. Comentamos al principio de este tema que una de las ventajas de este hecho es un mayor control sobre la apariencia de estos componentes. Bien, aquí vamos a poder comprobar eso.

Si estuviésemos programando con las antiguas librerías AWT sólo hay un componente sobre el cual podemos dibujar: Canvas. Es el único objeto que el S.O. nos proporciona en el que podemos dibujar. Esto quiere decir, por ejemplo, que si no nos gusta la apariencia de un scroll o de un botón no podemos hacer nada para modificarla, hemos de aceptarla tal y como se nos da. Por este motivo una misma aplicación se veía diferente en los diferentes S.O., un scroll de Windows no tiene por que ser igual que un scroll de Linux o Mac. En las nuevas librerías Swing es posible hacer que una misma aplicación se vea igual en todos los S.O., mediante una característica de los componentes que se llama Look and Feel.

Por otra parte en las librerías Swing es posible pintar sobre cualquier componente que derive de JComponent, es posible por lo tanto modificar la apariencia de un botón o scroll a nuestro antojo (al margen de los distintos Looks and Feel disponibles. De todas formas no es una buena práctica de programación pintar sobre cualquier componente. En la documentación de Java se aconseja que si queremos mostrar al usuario un dibujo se aconseja que lo hagamos siempre sobre un JPanel.

6.6.1 Empezando a dibujar

A continuación vamos a ver qué tenemos que hacer para dibujar en Swing. Lo primero es hacerse con el objeto gráfico del componente sobre el cual queremos dibujar. El objeto gráfico es un objeto que posee toda la información que un componente necesita para dibujarse en pantalla. Para obtener el objeto gráfico empleamos el método `getGraphics()`, de JComponent.

```
| Graphics g = Componente.getGraphics()
```

Una vez obtenido dicho objeto gráfico procedemos a dibujar empleando los métodos que dicho objeto nos proporciona. Estos métodos son tan intuitivos como numerosos, por lo que no haremos aquí una revisión de ellos. Si deseamos dibujar algo lo que más normal es acudir a la documentación de la librería y buscar en ella los métodos que necesitemos. A modo de ejemplo, si queremos dibujar una línea y vamos a la documentación de la clase Graphics encontraremos un método `drawLine(int x1, int x2, int x3, int x4)` en el cual (x1, x2) es el punto de inicio de la línea y (x3, x4) el fin de la línea.

Veamos un ejemplo de esto.

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

public class Dibujar1 extends JFrame implements
ActionListener{
    public Dibujar1(){
        setSize(400,400);
        setTitle("Dibujo");
        Container contentpane = this.getContentPane();
        contentpane.setLayout (new BorderLayout());
        JPanel p =new JPanel();
        contentpane.add(p, BorderLayout.SOUTH);
        JButton boton = new JButton("Dibujar");
        boton.addActionListener(this);
//Hago que el actioncommand del botón de dibujar sea
//"Dibujar"
        boton.setActionCommand("Dibujar");
        p.add(boton);
        JButton boton2 = new JButton("Salir");
        boton2.addActionListener(this);
//Hago que el actioncommand del botón para salir sea "Salir"
        boton2.setActionCommand("Salir");
        p.add(boton2);
        contentpane.add(panel, BorderLayout.CENTER);
        this.setVisible(true);
    }
    public void actionPerformed(ActionEvent e){
//Obtengo el actioncommand de el objeto que produjo el evento
        String comando = e.getActionCommand();
//Si es salir , terminamos la máquina virtual
        if(comando.equals("Salir")){
            System.exit(0);
        }
//Si no le pedimos al panel el objeto Grafico
        Graphics g = panel.getGraphics();
```

```
//Pintaremos en color azul
    g.setColor(Color.blue);
//Dibujamos una línea d las coordenados (0,0) a (100,100)
    g.drawLine(0,0, 100, 100);
//Dibujamos otra línea desde las cooredenadas (150,150) hasta
//la esquina inferior izquierda del panel
    g.drawLine(150,150,(int)(this.getSize()).getWidth(),
        (int)(this.getSize()).getHeight());
//Dibujo un rectángulo
    g.drawRect(100, 80, 200, 200);
//Cambio de color a rojo
    g.setColor(Color.red);
//Dibujo otro rectángulo, esta vez relleno
    g.fillRect(110, 90, 150, 150);
}
JPanel panel = new JPanel();

public void main(String[] args){
    new Dibujar1();
}

} ///:~
```

Éste código sin embargo tiene un pequeño problema. Si, por ejemplo, minimizamos la ventana y la volvemos a maximizar el dibujo idesaparece!. En el siguiente apartado veremos porqué.

6.6.2 El método *paintComponent*

Cuando cualquier componente de las librerías Swing por cualquier razón necesita “redibujarse” en pantalla llama al método `paintComponent`. En este método se halla toda la información que se necesita para dibujar el componente en pantalla.

Causas externas a un componente que fuerzan que este se "redibuje" son que la ventana en la que está se minimiza y luego se maximiza o que otra ventana se ponga encima de la ventana que lo contiene. Además el componente se dibujará cuando se crea y cuando se invoque el método `repaint()`.

Cuando este método es invocado lo único que aparecerá en el componente es lo que se dibuje desde el, todo lo demás es "borrado". Este es el motivo por el cual en nuestro ejemplo anterior al minimizar y luego maximizar la pantalla dejamos de ver lo que habíamos dibujado. Si queremos que siempre que el componente se redibuje apareciesen nuestros dibujos hemos de sobrescribir el método `paintComponent` y escribir en el código necesario para realizar estos dibujos. Veamos como haríamos esto sobre el ejemplo Dibujar1:

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

public class Dibujar2 extends JFrame implements
ActionListener{
    public Dibujar2(){
        setSize(400,400);
        setTitle("Dibujo");
        Container contentpane = this.getContentPane();
        contentpane.setLayout (new BorderLayout());
        JPanel p =new JPanel();
        contentpane.add(p, BorderLayout.SOUTH);
        JButton boton = new JButton("Dibujar");
        boton.addActionListener(this);
        boton.setActionCommand("Dibujar");
        p.add(boton);
        JButton boton2 = new JButton("Salir");
        boton2.addActionListener(this);
        boton2.setActionCommand("Salir");
        p.add(boton2);
        contentpane.add(panel, BorderLayout.CENTER);
```



```
        this.setVisible(true);
    }
    public void actionPerformed(ActionEvent e){
        String comando = e.getActionCommand();
        if(comando.equals("Salir")){
            System.exit(0);
        }
        Graphics g = panel.getGraphics();
        g.setColor(Color.red);
        g.fillRect(110, 90, 150, 150);
    }
    //Esta vez el panel es de un clase nuestra que extiende a
    //JPanel
    MyPanel panel = new MyPanel();

    public void main(String[] args){
        new Dibujar2();
    }
}
//La clase JPanel extiende a JPanel y sobrescribe su método
//paintComponet()
class MyPanel extends JPanel{
    public void paintComponent(Graphics g){
        //Recupero la funcionalidad del método sobrescrito, ya que se
        //encarga de realizar ciertas funcionalidades como buffering
        super.paintComponent(g);
        //Dibujo ahora dentro de este método
        g.setColor(Color.blue);
        g.drawLine(0,0, 100, 100);
        g.drawLine(150,50,(int)this.getSize().getWidth(),
            (int)(this.getSize().getHeight()));
        g.drawRect(100, 80, 200, 200);
    }
} ///:~
```

Podemos observar como en el método `paintComponent` lo primero que hay es una llamada al método de la clase padre. Esto es necesario por que el método de la clase padre es el que tiene idea de cómo “dibujar” el componente, (nosotros nos limitamos a dibujar sobre el componente, pero no a dibujarlo a el) y además realiza ciertas tareas de necesarias para el correcto funcionamiento del componente, como el buffering, de un modo totalmente transparentes al usuario. Si no invocamos al método del padre la ventana no se dibujará de modo correcto, ha de ser siempre lo primero que hagamos en el método `paintComponent`.

Por último decir que nunca debemos llamar al método `paintComponent` de modo directo, hemos de llamar siempre a `repaint` y será él el que se encargue de llamar a `paintComponent` con el argumento correcto.

6.7 REVISIÓN DE ALGUNOS COMPONENTES DE SWING

En este apartado y por falta de más tiempo haremos un pequeño comentario sobre varios de los componentes de la librería Swing más usados.

6.7.1 *JTextField*

Está pensado para obtener texto del usuario, este tecleará en él y cuando pulse intro podremos disponer del texto que tecleó. Únicamente se puede recoger una línea de texto. Tiene métodos para recoger el texto del usuario, poner un texto en él , recoger solo el texto seleccionado, seleccionar una parte del texto, insertar texto, cortar texto, pegar texto

Un ejemplo de su manejo se ve en el programa: `TJTextField.java` que se halla en el archivo zip complemento de este tutorial.

6.7.2 *JTextArea*

Todo lo dicho para `JTextField` es válido aquí también; la diferencia entre ambos es que `JTextArea` permite al usuario introducir más de una línea de texto.

Para ver un ejemplo ver JTextArea.java que se halla en el archivo zip complemento de este tutorial.

6.7.3 JPasswordField

No es más que un JTextField en el cual al escribir no se ve lo que se escribe, sino un carácter (*, por ejemplo). Se emplea para pedirle passwords al usuario y evitar que puedan ser leídas por alguien.

Ejemplo en el archivo TJPasswordField.java que se halla en el archivo zip complemento de este tutorial.

6.7.4 JScrollBar

Se trata de un scroll con múltiples funciones. Puede servirnos para tomar una entrada numérica del usuario o sobre todo para scrolear a lo largo de regiones demasiado grandes para ser vistas en la pantalla o ventana en que representamos la información. Hay un componente de Swing: JScrollPane, que es un panel que ya lleva incorporados por defecto dos scrolls, nosotros lo único que tenemos que hacer es introducir en él la imagen o texto a lo largo del cual queremos scrolear y él se encargará de todo lo demás.

JScrollBar posee métodos para fijar el valor numérico correspondiente al mínimo y máximo de las posiciones del scroll, para ver qué valor posee el scroll en un determinado momento, para poner el scroll en un determinado valor...

Un ejemplo se puede ver en el archivo TJScrollBar.java que se halla en el archivo zip complemento de este tutorial.

6.7.5 JLabel

No es más que una etiqueta de texto que podemos colocar al lado de cualquier componente para darle una indicación al usuario de cual es la función de dicho componente. También se puede emplear a modo de título de ,por ejemplo, un applet.

6.7.6 JCheckBox

Se trata de un componente empleado para tomar información del usuario sobre cosas del tipo "sí", "no"; se emplean para seleccionar una opción entre un conjunto de opciones seleccionadas por defecto. Posee métodos para seleccionar o deselectar, o para indicar el estado.

Ver ejemplo en el archivo TJCheckBox.java que se halla en el archivo zip complemento de este tutorial.

6.7.7 JRadioButton

Debe su nombre a funcionar como los botones de una radio antigua: al seleccionar uno se deselecta el que antes estaba seleccionado. Cuando añadimos estos componentes a nuestro diseño se añaden por grupos; de entre todos los JRadioButtons que han sido añadidos a un grupo sólo puede haber uno seleccionado, la selección de uno distinto dentro de un grupo provoca la inmediata desección del que antes estaba seleccionado. Se emplean para darle a elegir al usuario entre un grupo de opciones mutuamente excluyentes.

Para ver un ejemplo ver TJRadioButton.java que se halla en el archivo zip complemento de este tutorial.

6.7.8 JList

Se trata de un componente con una función similar a los dos anteriores , sólo que aquí las posibles selecciones se encuentran en una lista, que normalmente lleva un scroll incorporado, y se seleccionan haciendo clic directamente sobre ellas. Se emplea cuando el número de opciones entre las que ha de escoger el usuario es demasiado grande para presentársela en forma de radiobuttons o checkboxes.

Posee métodos para permitir simple o múltiple selección, seleccionar o deselectar un componente, averiguar que componente está seleccionado...

Para ver un ejemplo ver TJList.java que se halla en el archivo zip complemento de este tutorial.

6.7.9 JComboBox

Su filosofía es idéntica a la de JList, pero en esta ocasión las opciones no se ven en un principio. El usuario ha de hacer un clic sobre una pestaña que desplegará una lista de opciones sobre las cuales escoge el usuario una mediante un clic.

Para ver un ejemplo ver TJComboBox1.java que se halla en el archivo zip complemento de este tutorial.

6.7.10 JMenu

Se trata de un componente que permite generar los típicos menús a los que todos estamos acostumbrados. En estos menús se pueden añadir CheckBoxes y RadioButtons.

Para ver un ejemplo ver TJCheckBoxMenuItem.java que se halla en el archivo zip complemento de este tutorial.

7 JAPPLET

Un applet es un programa Java con la capacidad de ser incluido en una página web y correr dentro de un navegador. Para que esto sea posible sin que el visitante de la página web sufra alguna violación de su intimidad o se arriesgue a sufrir daños en su máquina hay ciertas limitaciones que poseen los applets respecto a las aplicaciones normales; estas son fundamentalmente:

- Un applet nunca podrá ejecutar un programa local de nuestra máquina.
- Un applet sólo podrá comunicarse con la máquina servidora de la página web, pero no con ninguna otra máquina.
- Un applet no puede nunca acceder a nuestra sistema de ficheros, ni para lectura ni para escritura.
- Un applet es incapaz de averiguar información sobre la máquina en la que corre, aparte del sistema operativo, versión de la máquina virtual de Java y algún parámetro del sistema, como la resolución de la pantalla. Un applet nunca podría, por ejemplo, averiguar nuestra dirección de e-mail, el nombre del propietario de la máquina....

Recientemente ha surgido lo que se denominan applets firmados: son applets que incorporan una "firma" que nos garantiza quién hizo el programa. Si queremos podemos darle a este applet más privilegios, si nos fiamos del que lo programó, como de el escribir en nuestro disco duro, pero siempre hemos de tener en cuenta que un applet esté firmado sólo nos garantiza quién ha sido su creador, pero no que este applet no haya sido construido con intenciones maliciosas para averiguar información sobre nosotros o dañar nuestra máquina.

7.1 CÓMO CONVERTIR UNA APLICACIÓN EN UN APPLET

A continuación damos una pequeña receta para convertir una aplicación en un applet:

- Generar una página HTML con el código adecuado para cargar el applet.
- Eliminar el método main. Su contenido habitualmente suele ser el código necesario para crear un objeto del tipo frame. Esto no será necesario, el navegador se encargará de crear nuestro objeto, de tipo applet.
- Hacer que nuestra clase extienda a JApplet, en vez de a JFrame.
- Eliminar las llamadas a setSize, ya que será la página HTML la que determine el tamaño de nuestro applet.
- Eliminar las llamadas setTitle, será también el código HTML el que se encargará de ponerle título al applet.
- Eliminar las llamadas addWindowListener; un applet no puede ser cerrado, es el navegador el que se encarga de iniciarlo y cerrarlo.
- Reemplazar el constructor por un método llamado init. Este será el equivalente al constructor de una aplicación. Será automáticamente llamado por el applet.
- Recordar hacer pública la clase del applet.

Veamos como queda nuestro ejemplo 23 tras estos cambios:

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

public class Applet extends JApplet implements
ActionListener{
    private JPanel panel = new JPanel();
    private JButton azul,rosa,amarillo,verde;
//Método init del applet
    public void init(){
        Container contentpane = getContentPane();
        panel.setLayout(new BorderLayout());

        azul = new JButton("Azul");
        azul.addActionListener(this);
        Dimension d = new Dimension();
        d.height = 40;
        d.width = 100;
        azul.setPreferredSize(d);

        verde = new JButton("Verde");
```

```
        verde.addActionListener(this);
        d.height = 40;
        d.width = 100;
        verde.setPreferredSize(d);

        amarillo = new JButton("Amarillo");
        amarillo.addActionListener(this);
        d.height = 40;
        d.width = 100;
        amarillo.setPreferredSize(d);

        rosa = new JButton("Rosa");
        rosa.addActionListener(this);
        d.height = 40;
        d.width = 100;
        rosa.setPreferredSize(d);

        panel.add(azul, BorderLayout.SOUTH);
        panel.add(verde, BorderLayout.NORTH);
        panel.add(amarillo, BorderLayout.EAST);
        panel.add(rosa, BorderLayout.WEST);

        contentpane.add(panel);
        panel.setBackground(Color.red);
    }

    public void actionPerformed (ActionEvent e){
        Object source = e.getSource();
        if (source == azul)
            panel.setBackground(Color.blue);
        if (source == verde)
            panel.setBackground(Color.green);
        if (source == amarillo)
            panel.setBackground(Color.yellow);
        if (source == rosa)
            panel.setBackground(Color.pink);
    }
}
```



```
    }  
  
} ///:~
```

7.2 CICLO DE VIDA DE UN APPLET

Para entender el funcionamiento de un applet hay cuatro métodos que debemos conocer:

-Método `init()`, que es el constructor del applet.

-Método `start()`. Este método se vuelve a llamar además cada vez que volvemos a la página del applet. Aquí se suelen programar tareas que es necesario volver a arrancar cada vez que volvamos a cargar el applet, como por ejemplo una animación.

-Método `stop()`, se le llama cuando el navegador sale de la página del applet. su propósito es liberar los recursos del sistema que el applet pudiera estar consumiendo. Las tareas que este método detiene para liberar recursos del sistema suelen ser las mismas que se arrancan en el método `start()`. El propio navegador se encarga de llamar a este método, aunque no debe ser llamado directamente. Si nuestro applet no arranca ninguna animación, emplea audio o realiza cálculos en un thread por lo general no será necesario implementar ni este método ni `start()`.

-Método `destroy()`. Se llama cuando se cierra el navegador. Este método lo que hace es que el applet deje de estar en la caché del navegador de tal forma que la próxima vez que arranquemos dicho método será como si fuese la primera vez que lo arrancamos.

8 THREADS

8.1 ¿QUÉ ES UN THREAD?

Mis tres años de experiencia como docente me dicen que no es un concepto sencillo el de "Thread" la primera vez que un alumno se "enfrenta" a él. Empecemos por explicar algo más "simple de ver": qué es un proceso. Seguramente habrás oído alguna vez que tal sistema operativo es "multitarea" o "multiproceso". ¿Que quiere decir eso?. Esto quiere decir que ese sistema operativo es capaz de ejecutar más de una tarea (proceso) simultáneamente.

Cada aplicación que nosotros arrancamos es un proceso dentro del ordenador: si tenemos abiertos Internet Explorer, Word y Excel a la vez cada uno de ellos se ejecutará como un proceso. Esto no es posible en un sistema operativo monotarea: en el MSDOS no es posible ejecutar más de un comando a la vez; aquellos veteranos que os acercasteis a la informática en los días del MSDOS sabéis que si arrancábamos un programa teníamos que finalizarlo para poder arrancar otro. Evidentemente el ser multitarea es una cualidad muy deseable para un S.O., de no serlo no podríamos tener arrancada más de una aplicación a la vez.

Los distintos procesos están "protegidos unos de otros". Un proceso posee su propia región de memoria del ordenador que es "privada" para los demás procesos e inaccesible, por lo que un proceso no puede acceder (directamente en memoria) a las variables de otro proceso. Cuando la CPU cambia de proceso, es decir, para de ejecutar un proceso y empieza con otro, ha de cambiar su contador interno y el espacio de direccionamiento; además las variables que tuviese cacheadas no le serán útiles: pertenecían a otro proceso y este no las va a poder utilizar, por lo que tendrá que acceder a memoria principal no pudiendo hacer uso de la caché hasta que la renueve. Todo esto se traduce en un elevado coste de CPU al cambiar de un proceso a otro.

Sin embargo aún podemos ir más allá: es deseable que dentro de un proceso haya "procesos"; por ejemplo a todos nos gusta cuando ejecutamos una tarea que va a consumir mucho tiempo, como descargar algo desde internet mediante un FTP gráfico, podamos en cualquier momento detener esa tarea. Para ellos es necesario que dentro la aplicación del FTP (que se ejecuta en la máquina como un proceso) arranque un nuevo proceso que se encargue de la descarga del material, mientras que otro proceso sigue escuchando nuestros eventos y reaccionando a ellos (por ejemplo cancelando la descarga cuando pulsamos el botón de "cancelar").

Parece una cualidad muy deseable de un proceso que pudiese generar otros procesos para atender distintas tareas sin que perdamos control sobre él. Sin embargo el coste de cambio de

un proceso a otro es demasiado elevado para hacer que esto sea posible en la práctica. Por este motivo a principios de los 90 se idearon los threads.

Como este no es un curso de computación, si no de programación, no nos meteremos en detalles de las diferencias entre un proceso y un thread. El comportamiento de un thread es semejante a ser “un proceso de un proceso”; los threads son distintas líneas de ejecución de un proceso. ¿Qué diferencia a un thread de un proceso?. Primero un thread pertenece siempre a algún proceso, no pueden existir independientemente. Por otro lado los distintos threads de un proceso comparten el mismo espacio de memoria, es decir unos pueden acceder a las variables de otros. Esto no es así entre los distintos procesos, los cuales para comunicarse han de emplear complejos mecanismos, que tienen asociado un alto coste computacional.

Como un thread tiene el mismo espacio de memoria que sus “hermanos” (los que pertenecen al mismo proceso) el cambio de ejecución de un thread a otro lleva asociado un coste mucho menor que el cambio de un proceso a otro, y además la comunicación entre threads es más sencilla y menos costosa computacionalmente hablando.

Parece que con los threads recuperamos todas las ventajas de los procesos sin sus inconvenientes, pero no es así: los threads tienen un inconveniente: un thread puede machacar las variables de otro, por lo que tendremos que tener cuidado de que esto no ocurra. Esto entre proceso no podía ocurrir, ya que la región de memoria de cada uno es “privada”. Por lo tanto los threads no son tan seguros como los procesos.

8.2 LA VIDA DE UN THREAD

Para entender como trabaja un thread es necesario conocer los distintos estados en los cuales se puede encontrar:

8.2.1 Recien nacido:

Se ha reservado memoria para el thread y se han inicializado sus variables, pero no se puesto todavía en cola de ejecución. Dicho de otro modo: hemos creado un objeto de tipo Thread. En este estado le pueden ocurrir dos cosas: que pase a cola de ejecución, mediante el método `Start()`, o que se mate sin llegar a ejecutarse nunca, mediante el método `stop()`.

8.2.2 Ejecutable:

El thread está listo para ser ejecutado, se halla en cola de ejecución junto a los demás threads. Periódicamente la CPU cambia de thread de ejecución, cogiendo el thread que estaba ejecutando y poniéndolo en la cola de ejecución y escogiendo un nuevo thread de la cola de ejecución.

Los threads llevan asociada una prioridad, cuando la CPU a de elegir un thread de la cola de ejecución para ver cual ejecuta elige el de mayor prioridad, y de haber varios los va ejecutando secuencialmente.

8.2.3 Corriendo

El thread está siendo ejecutado en el procesador. Hay tres posibles causas por las que un thread que esta corriendo pueda liberar el control del procesador:

Se acaba su tiempo de ejecución y vuelve a la cola de procesos, pasando al estado de ejecutable.

Se ejecuta el método `sleep()` o `wait()`, pasando a estado bloqueado

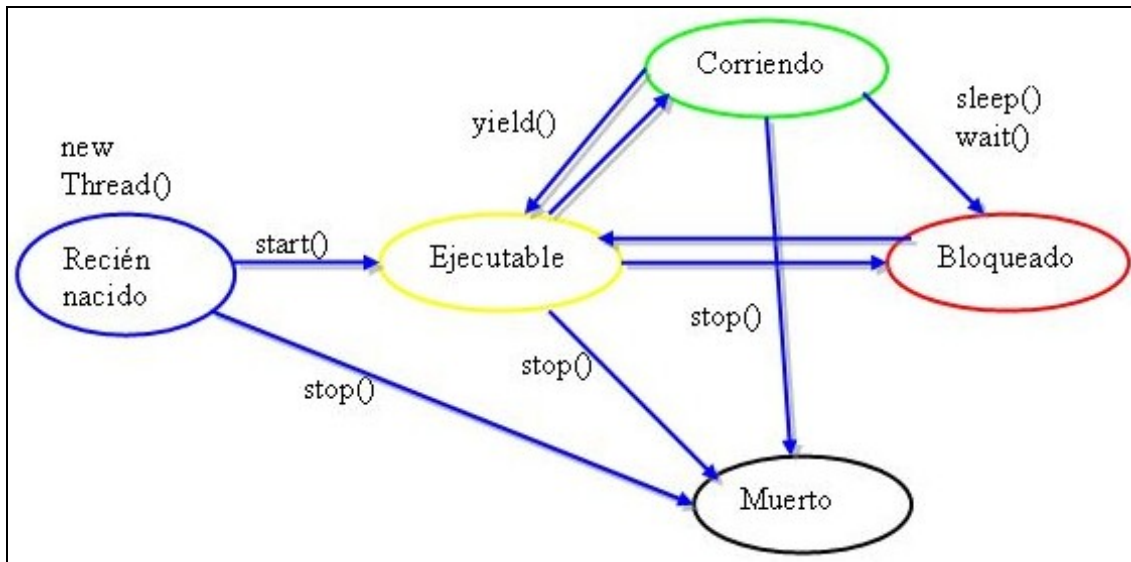
Termina su cuerpo de ejecución, o se invoca a su método `stop`, por lo que pasa al estado muerto.

8.2.4 Bloqueado

En este estado el thread no está en la cola de ejecución, está esperando a que ocurra un determinado evento o transcurra un determinado plazo de tiempo antes de poder acceder a la cola de ejecución.

8.2.5 Muerto

Como su nombre indica el thread deja de existir. Puede ocurrir bien por que ha terminado su cuerpo de ejecución o bien porque ha sido matado mediante el método `stop()`.

**Figura 7 Estados de un Thread**

8.3 THREADS EN JAVA

Como ya hemos comentado java soporta los threads de modo nativo, esto permite que si Java se ejecuta en un sistema multiprocesador reparta los distintos threads del programa entre los distintos procesadores de modo transparente para el programador. Esto no ocurre así con C++, por ejemplo. Este lenguaje no soporta de modo nativo los threads, los soporta mediante el uso de librerías, por lo que no posee capacidad para distribuir los threads entre varios procesadores de modo transparente para el programador.

La capacidad de que un programa cualquiera de Java, sin que halla sido programado pensando en ser ejecutado por un sistema multiprocesador, sea capaz de correr empleando varios procesadores es uno de las características que en un futuro pueden contribuir a aumentar el uso de este lenguaje de programación. Es de esperar que debido al coste cada vez superior de aumentar la escala de integración de los transistores de los chips llegue el día (en no muchos años) en que sea más barato un equipo con dos CPU que uno con una sola CPU y de potencia equivalente al anterior.

Existen dos formas de crear un thread en java, la primera es extender la clase Thread y sobrescribir su método run. En este método va el "cuerpo del thread", el código que se va a comportar como thread. El resto de los métodos de nuestra clase serán normales.

La segunda forma de crear un thread en java es implementando la interface Runnable, interface que posee un único método: run, que debemos sobrescribir.

Para empezar a ejecutar un thread nunca debemos llamar directamente al método run, hemos de llamar al método start y ya se encargará éste de llamar a run. Si llamamos directamente a run este se ejecutará como un método cualquiera y no en un nuevo thread.

Si queremos detener durante un intervalo de tiempo la ejecución de un thread empleamos el método sleep(int), dónde el argumento que se le pasa es el tiempo que ha de esperar en milisegundos.

8.4 UN PROGRAMA SIN THREADS

Vamos a suponer que tenemos que hacer un programa que al pulsar un botón haga aparecer una pelotita en una ventana y esta vaya rebotando en ella. Queremos que cada vez que se pulse el botón aparezca una nueva pelotita.

Sguiendo con mi interminable manía de mostrar primero lo que no se debe de hacer, para luego mostrar lo que si debemos hacer, supondremos en este código que no sabemos nada de threads .El código que escribiríamos sería algo así:

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

/**
 * Clase auxiliar que contiene el main y permite arrancar la
 * aplicación
 */
public class Thread1 {
    public static void main (String[] args){
        JFrame frame = new BounceFrame();
        frame.show();
    }
}

/**
 * JFrame sobre el cual dibujaremos
```

```
* */
class BounceFrame extends JFrame{
    public BounceFrame(){
        setSize(500,400);
        //Gestión de los eventos de ventana
        addWindowListener (new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });

        Container contentPane = getContentPane();
        panelDibujar = new JPanel();
        contentPane.add(panelDibujar, BorderLayout.CENTER);
        JPanel panelSur = new JPanel();
        JButton botonAnimar = new JButton("Animar");
        panelSur.add(botonAnimar);
        botonAnimar.addActionListener( new ActionListener(){
            //Cuando el usuario pulse este botón dibujamos la
            pelotita y la hacemos rebotar
            public void actionPerformed(ActionEvent evt){
                Pelotita b = new
                Pelotita(panelDibujar,Color.black);
                b.rebotar();
            }
        });
        JButton botonDetener = new JButton("Detener");
        panelSur.add(botonDetener);
        botonDetener.addActionListener( new ActionListener(){
            //Al pulsar este botón detenemos la animación y
            terminamos la máquina
            //Virtual.
            //Como sólo empleemos un thread comprobaremos que el
            botón no responde
            //hasta que la pelotita para de moverse.
            public void actionPerformed (ActionEvent evt){
                panelDibujar.setVisible(false);
            }
        });
    }
}
```

```
        System.exit(0);
    }
});
contentPane.add(panelSur,"South");
}
private JPanel panelDibujar;
}

class Pelotita {
    public Pelotita (JPanel b,Color c)
    { panel = b;
      color = c;
    }

    //Dibuja la pelotita la primera vez
    public void dibujarPelotita () {
        Graphics g = panel.getGraphics();
        g.setColor(color);
        g.fillOval(posicionX,posicionY,tamanoXPelotita,
tamanoYPelotita);
        g.dispose();
    }

    public void mover(){
        Graphics g = panel.getGraphics();
        g.setXORMode(panel.getBackground());
        g.setColor(color);
        //Pintamos en las anitguas coordenadas, borrando de
ese modo la pelotita,
        //Ya que estamos en modo XOR
        g.fillOval(posicionX,  posicionY,  tamanoXPelotita,
tamanoYPelotita);
        //Calculamos las nuevas coordenadas
        posicionX += incrementoX;
        posicionY +=incrementoY;
        Dimension d = panel.getSize();
        //Si la coordenada x es menor que 0 invertimos el
incremento en la
```



```
//coordenada x
if (posicionX<0){
    posicionX = 0;
    incrementoX = -incrementoX;
}
//Si la pelotita se sale del panel por la derecha
hacemos que "rebote"
//hacia la izquierda
if (posicionX+tamanoXPelotita >= d.width){
    posicionX = d.width-tamanoXPelotita; incrementoX
= - incrementoX;
}
//Si la coordenada y es menor que 0 invertimos el
incremento en la
//coordenada y
if(posicionY<0){
    posicionY=0;
    incrementoY = -incrementoY;
}
//Si la pelotita se sale del panel por abajo hacemos
que "rebote"
//hacia arriba
if (posicionY+ tamanoYPelotita >= d.height){
    posicionY = d.height-tamanoYPelotita;
    incrementoY = -incrementoY;
}
g.fillOval (posicionX, posicionY, tamanoXPelotita,
tamanoYPelotita);
g.dispose();
}
public void rebotar (){
    dibujarPelotita();
    //La pelotita se moverá 1000 veces
    for (int i = 1; i<= 1000; i++){
        mover();
    }
}
```

```
    }  
  
    private JPanel panel;  
    private static final int tamanoXPelotita = 10;  
    private static final int tamanoYPelotita = 10;  
    private int posicionX = 0;  
    private int posicionY = 0;  
    private int incrementoX = 2;  
    private int incrementoY = 2;  
    private Color color;  
  
} ///:~
```

8.5 UN PROGRAMA CON THREADS

El código anterior tiene un comportamiento nefasto: cada vez que generamos una pelota la aplicación deja de responder a nuestras órdenes hasta que la pelota deja de moverse. Además el movimiento es a una velocidad muy alta, obteniéndose como resultado un movimiento poco estético. Evidentemente no es aceptable una aplicación a la cual una vez que le mandas hacer algo no te hace caso hasta que acaba; imaginaos que vuestro navegador web fuese así, estáis en vuestra casa navegando y decidís bajar un archivo. Cuando habéis empezado a bajarlo os dais cuenta de que ocupa 100 Mb y le llevará con vuestro viejo módem a 28 kbps un día; intentáis cancelarlo, pero el proceso encargado de la descarga no responderá hasta que acabe su tarea!.

Para evitar este tipo de situaciones se emplean los threads. Volviendo a nuestro ejemplo de la pelotita lo que deberíamos hacer sería que cada pelotita fuese un thread, así podremos crear varias pelotitas a la vez y además la aplicación no dejará de responder cuando lo hagamos. Además al ser threads las pelotitas tenemos control sobre su velocidad de ejecución, podemos mandar los threads al estado bloqueado de vez en cuando y las pelotitas se moverán más lentas.

Para detener la ejecución de un Thread durante un intervalo de tiempo invocamos a su método `sleep(int milisegundos)`, el argumento que se le pasa es el tiempo en milisegundos que queremos que este Thread detenga su ejecución. Al transcurrir este tiempo este Thread pasa al estado de ejecutable.

Veamos el código anterior haciendo uso de threads:

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

/**
 * Clase auxiliar que contiene el main y permite arrancar la
 * aplicación
 */
public class Thread2 {
    public static void main (String[] args){
        JFrame frame = new BounceFrame();
        frame.show();
    }
}

/**
 * JFrame sobre el cual dibujaremos
 * */
class BounceFrame extends JFrame{
    public BounceFrame(){
        setSize(500,400);
        //Gestión de los eventos de ventana
        addWindowListener (new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });

        Container contentPane = getContentPane();
        panelDibujar = new JPanel();
        contentPane.add(panelDibujar, BorderLayout.CENTER);
        JPanel panelSur = new JPanel();
        JButton botonAnimar = new JButton("Animar");
        panelSur.add(botonAnimar);
        botonAnimar.addActionListener( new ActionListener(){
```

```
        //Cuando el usuario pulse este botón dibujamos la
        pelotita y la hacemos rebotar
        public void actionPerformed(ActionEvent evt){
            Pelotita b = new
Pelotita(panelDibujar,Color.black);
            //Una vez creada la pelotita (El Thread) la
            hacemos rebotar
            b.start();
        }
    });
    JButton botonDetener = new JButton("Detener");
    panelSur.add(botonDetener);
    botonDetener.addActionListener( new ActionListener(){
        //Al pulsar este botón detenemos la animación y
terminamos la máquina
        //Virtual.
        //Como sólo emplemos un thread comprobaremos que el
botón no responde
        //hasta que la pelotita para de moverse.
        public void actionPerformed (ActionEvent evt){
            panelDibujar.setVisible(false);
            System.exit(0);
        }
    });
    contentPane.add(panelSur,"South");
}
private JPanel panelDibujar;
}

class Pelotita extends Thread{
    public Pelotita (JPanel b,Color c)
    { panel = b;
      color = c;
    }
    //Dibuja la pelotita la primera vez
    public void dibujarPelotita () {
        Graphics g = panel.getGraphics();
```

```
        g.setColor(color);
        g.fillOval(posicionX,posicionY,tamanoXPelotita,
tamanoYPelotita);
        g.dispose();
    }

    public void mover(){
        Graphics g = panel.getGraphics();
        g.setXORMode(panel.getBackground());
        g.setColor(color);
        //Pintamos en las anitguas coordenadas, borrando de
ese modo la pelotita,
        //Ya que estamos en modo XOR
        g.fillOval(posicionX,  posicionY,  tamanoXPelotita,
tamanoYPelotita);
        //Calculamos las nuevas coordenadas
        posicionX += incrementoX;
        posicionY +=incrementoY;
        Dimension d = panel.getSize();
        //Si la coordenada x es menor que 0 invertimos el
incremento en la
        //coordenada x
        if (posicionX<0){
            posicionX = 0;
            incrementoX = -incrementoX;
        }
        //Si la pelotita se sale del panel por la derecha
hacemos que "rebote"
        //hacia la izquierda
        if (posicionX+tamanoXPelotita >= d.width){
            posicionX = d.width-tamanoXPelotita; incrementoX
= - incrementoX;
        }
        //Si la coordenada y es menor que 0 invertimos el
incremento en la
        //coordenada y
        if(posicionY<0){
```

```
        posicionY=0;
        incrementoY = -incrementoY;
    }
    //Si la pelotita se sale del panel por abajo hacemos
    que "rebote"
    //hacia arriba
    if (posicionY+ tamanoYPelotita >= d.height){
        posicionY = d.height-tamanoYPelotita;
        incrementoY = -incrementoY;
    }
    g.fillOval (posicionX, posicionY, tamanoXPelotita,
tamanoYPelotita);
    g.dispose();
}
public void rebotar (){
    dibujarPelotita();
    //La pelotita se moverá 1000 veces
    for (int i = 1; i<= 1000; i++){
        mover();
        try {
            //Esta sentencia duerme durante 5 milisegundos
            el thread,
            //Evitando que la pelotita se mueva demasiado
            rápido
            Thread.currentThread().sleep(5);
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
//Cuerpo del Thread. No hay ningún problem por invocar a
otros métodos
//desde el
public void run (){
    //Dibujamos la pelotita
```

```
        dibujarPelotita();
        //La movemos 10000 veces
        for (int i = 1; i<= 10000; i++){
            mover();
            //Entre movimiento y movimiento detenemos el
Thread 1 milisegundo,
            //así la pelotita no se moverá tan rápido. Para
ello empleamos el
            //método sleep de la clase Thread, que puede
lanzar una excepción,
            //por lo que hemos de invocar este método dentro
de un try catch.
            try {
                sleep(1);
            }
            catch (InterruptedException ex) {
            }
        }
    }

    private JPanel panel;
    private static final int tamanoXPelotita = 10;
    private static final int tamanoYPelotita = 10;
    private int posicionX = 0;
    private int posicionY = 0;
    private int incrementoX = 2;
    private int incrementoY = 2;
    private Color color;

} ///:~
```

Por último vamos a “jugar” un poco más con esta pequeña aplicación. Vamos a añadirle otro botón el cual al pulsarlo genera una pelotita distinta: esta pelotita será un thread de mayor prioridad. Veremos como estas pelotitas al tener mayor prioridad (las rojas) son llamadas más veces por la CPU para que sus respectivos threads se ejecuten y se mueven más rápido. Las

pelotitas normales llegarán incluso a pararse si hay muchas pelotitas rápidas (rojas) que no dejan moverse a las lentas.

La prioridad a los threads se les asigna mediante el método `setPriority(int)`, donde `int` es la prioridad. Los posibles valores de las prioridades están codificados como variables estáticas y final dentro de la clase `Thread`, como `"Thread.MIN_PRIORITY"`, que representa la prioridad mínima, `"Thread.NORM_PRIORITY"` prioridad normal, y `"Thread.MAX_PRIORITY"`, la máxima. Veamos el código:

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

/**
 * Clase auxiliar que contiene el main y permite arrancar la
 * aplicación
 */
public class Thread3 {
    public static void main(String[] args) {
        JFrame frame = new BounceFrame();
        frame.show();
    }
}

/**
 * JFrame sobre el cual dibujaremos
 */
class BounceFrame
    extends JFrame {
    public BounceFrame() {
        setSize(500, 400);
        //Gestión de los eventos de ventana
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```



```
});

Container contentPane = getContentPane();
panelDibujar = new JPanel();
contentPane.add(panelDibujar, BorderLayout.CENTER);
JPanel panelSur = new JPanel();
JButton botonAnimar = new JButton("Animar");
panelSur.add(botonAnimar);
botonAnimar.addActionListener(new ActionListener() {
    //Cuando el usuario pulse este botón dibujamos la
    pelotita y la hacemos rebotar
    public void actionPerformed(ActionEvent evt) {
        Pelotita b = new Pelotita(panelDibujar, Color.black);
        //Le ponemos una prioridad baja a esta pelotita
        b.setPriority(Thread.MIN_PRIORITY);
        //Una vez creada la pelotita (El Thread) la hacemos
    rebotar
        b.start();
    }
});

JButton botonExpress = new JButton("Express");
panelSur.add(botonExpress);
botonExpress.addActionListener(new ActionListener() {
    //Cuando el usuario pulse este botón dibujamos la
    pelotita y la hacemos rebotar
    public void actionPerformed(ActionEvent evt) {
        Pelotita b = new Pelotita(panelDibujar, Color.red);
        //Le ponemos una prioridad normal a esta pelotita
        b.setPriority(Thread.NORM_PRIORITY);
        //Una vez creada la pelotita (El Thread) la hacemos
    rebotar
        b.start();
    }
});

JButton botonDetener = new JButton("Detener");
panelSur.add(botonDetener);
```

```
        botonDetener.addActionListener(new ActionListener() {
            //Al pulsar este botón detenemos la animación y
terminamos la máquina
            //Virtual.
            //Como sólo emplemos un thread comprobaremos que el
botón no responde
            //hasta que la pelotita para de moverse.
            public void actionPerformed(ActionEvent evt) {
                panelDibujar.setVisible(false);
                System.exit(0);
            }
        });
        contentPane.add(panelSur, "South");

    }

    private JPanel panelDibujar;
}

class Pelotita
    extends Thread {
    public Pelotita(JPanel b, Color c) {
        panel = b;
        color = c;
    }

    //Dibuja la pelotita la primera vez
    public void dibujarPelotita() {
        Graphics g = panel.getGraphics();
        g.setColor(color);
        g.fillOval(posicionX,      posicionY,      tamanoXPelotita,
tamanoYPelotita);
        g.dispose();
    }

    public void mover() {
```

```
Graphics g = panel.getGraphics();
g.setXORMode(panel.getBackground());
g.setColor(color);
//Pintamos en las antiguas coordenadas, borrando de ese
modo la pelotita,
//Ya que estamos en modo XOR
g.fillOval(posicionX,      posicionY,      tamanoXPelotita,
tamanoYPelotita);
//Calculamos las nuevas coordenadas
posicionX += incrementoX;
posicionY += incrementoY;
Dimension d = panel.getSize();
//Si la coordenada x es menor que 0 invertimos el
incremento en la
//coordenada x
if (posicionX < 0) {
    posicionX = 0;
    incrementoX = -incrementoX;
}
//Si la pelotita se sale del panel por la derecha hacemos
que "rebote"
//hacia la izquierda
if (posicionX + tamanoXPelotita >= d.width) {
    posicionX = d.width - tamanoXPelotita;
    incrementoX = -incrementoX;
}
//Si la coordenada y es menor que 0 invertimos el
incremento en la
//coordenada y
if (posicionY < 0) {
    posicionY = 0;
    incrementoY = -incrementoY;
}
//Si la pelotita se sale del panel por abajo hacemos que
"rebote"
//hacia arriba
if (posicionY + tamanoYPelotita >= d.height) {
```

```
        posicionY = d.height - tamanoYPelotita;
        incrementoY = -incrementoY;
    }
    g.fillOval(posicionX,      posicionY,      tamanoXPelotita,
tamanoYPelotita);
    g.dispose();
}

public void rebotar() {
    dibujarPelotita();
    //La pelotita se moverá 1000 veces
    for (int i = 1; i <= 1000; i++) {
        mover();
        try {
            //Esta sentencia duerme durante 5 milisegundos el
thread,
            //Evitando que la pelotita se mueva demasiado rápido
            Thread.currentThread().sleep(5);
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

//Cuerpo del Thread. No hay ningún problem por invocar a
otros métodos
//desde el
public void run() {
    //Dibujamos la pelotita
    dibujarPelotita();
    //La movemos 10000 veces
    for (int i = 1; i <= 10000; i++) {
        mover();
        //Entre movimiento y movimiento detenemos el Thread 1
milisegundo,
```

```
        //así la pelotita no se moverá tan rápido. Para ello
        empleamos el
        //método sleep de la clase Thread, que puede lanzar una
        excepción,
        //por lo que hemos de invocar este método dentro de un
        try catch.
        try {
            sleep(1);
        }
        catch (InterruptedException ex) {
        }
    }

    private JPanel panel;
    private static final int tamanoXPelotita = 10;
    private static final int tamanoYPelotita = 10;
    private int posicionX = 0;
    private int posicionY = 0;
    private int incrementoX = 2;
    private int incrementoY = 2;
    private Color color;

} ///:~
```

9 APÉNDICE A : MEJORAS AL CÓDIGO DE LA GUERRA.

La primera mejora que se le puede hacer al código de la batalla entre marcianos y terrícolas tiene que ver con la herencia: en la clase Guerrero hay una variable que privada de tipo cadena de caracteres que se llama soy. Esta variable contenía la cadena de caracteres "Marciano" o "Terrícola" según el guerrero fuese un Marciano o un Terrícola. Tanto la clase Marciano como Terrícola poseen otra variable privada llamada soy que almacena la misma cadena de caracteres.

Las dos clases emplean esa cadena de caracteres para imprimir sentencias por la consola; la clase Guerrero para imprimir el texto soy + "Dispara nº " + disparo, y las clases hijas para imprimir soy + " Muerto por disparo nº " + i. Por cada objeto Marciano o Terrícola que creamos se van a almacenar dos cadenas de caracteres con el texto "Marciano", y "Terrícola" respectivamente, ya que las clases hijas no van a poder acceder a la clase privada de la superclase.

Esto se puede resolver simplemente haciendo protegida (protected) la variable soy en Guerrero, de tal modo que Terrícola y Marciano la hereden.

La segunda mejora tiene que ver con las líneas "return ((Terricola)(tripulacion[1])).getTotal();" y "return ((Marciano)(tripulacion[1])).getTotal();". El propósito de estas sentencias era averiguar el valor de las variables estáticas total definidas en Terrícola y Marciano respectivamente, variables que nos permitían llevar cuenta de cuantos marcianos y terrícola había vivos en cada momento.

La forma de acceder a esta información es muy extraña: cogemos al tripulante que está en la posición 1 del array y le preguntamos. ¿Por que a él y no a otro?. El resultado sería el mismo si lo invocásemos sobre otro tripulante, todos los métodos getTotal() de todos los tripulantes devuelven el mismo dato, la variable estática total. Cuando un método sólo accede a variables estáticas, como es el caso de getTotal(), lo más lógico es que el método también sea estático, ya que si sólo accede a información de la clase y no de una de las instancias lo lógico es preguntarle a la clase y no a una de las instancias por el.

Definiendo este método como estático (static) es Terrícola y Marciano podríamos haber cambiado las líneas "return ((Terricola)(tripulacion[1])).getTotal();" y "return ((Marciano)(tripulacion[1])).getTotal();" por "return Terricola.getTotal();" y "return Marciano.getTotal();", quedando el código mucho más claro.

10 APÉNDICE B: ¿USO IDE PARA APRENDER JAVA? ¿CUAL?

El propósito de esta sección es informar al lector de algunos de los entornos de desarrollo (IDE, Integrated Development Environment) existentes para el lenguaje Java. También se dan algunas directrices sobre cual es, desde mi punto de vista, el camino a seguir en cuanto al uso de IDEs, para alguien que está empezando a dar sus primeros pasos en Java.

10.1 BLUEJ [HTTP://WWW.BLUEJ.ORG/](http://www.bluej.org/)

Es el IDE que, sin duda, recomiendo al lector para que empiece a dar sus primeros pasos en Java. Esta herramienta freeware ha sido diseñada para introducir al estudiante en la programación orientada a objetos. Es un IDE “ligero”, posee la funcionalidad básica de cualquier IDE: editor, compilador y depurador. Siempre visualiza el código de nuestro proyecto en UML, mostrando las clases con las relaciones de herencia y dependencias entre ellas.

Posee ciertas capacidades para “jugar” con las clases permitiendo la creación de objetos, así como “jugar” con ellos invocando sus métodos, viendo el valor que toman sus variables... Estas cualidades carecen de valor para un programador experto, pero permiten a un programador novato familiarizarse con los conceptos de objeto, método, variable, herencia... a diferenciar entre objeto y clase...

No sólo recomiendo este IDE a los que empecéis a trabajar con java por su orientación didáctica; lo hago además porque considero que cualquier IDE “pesado”, de los cuales se recogen varios en este apéndice, no es apto para aprender a programar. Su elevada complejidad, si bien es fácilmente compensada con las obsesiones éstos ofrecen en el caso de un programador veterano, no lo es en un programador novato, quien ya tiene bastante con pelearse con el lenguaje para además tener que hacerlo con el IDE. Por otro lado los asistentes y el elevado grado de automatización de algunas tareas en los IDE pesados hacen que el programador novato haga cosas sin saber realmente lo que está haciendo, con lo cual, si bien si logra hacer pequeñas aplicaciones, su aprendizaje estará plagado de lagunas.

A los que sigan mi consejo y se decanten por este IDE para empezar a programar les recomiendo una cosa más: este tutorial: <http://www.bluej.org/tutorial/tutorial-spanish-2.pdf>, traducción al español del manual de usuario de BlueJ.

También les daré otro consejo: en cuanto empecéis a dominar la programación orientada a objetos y hayáis dado vuestros primeros pasos por Java abandonad este IDE. Los que se recogen abajo son mucho más completos y aumentaran notablemente vuestra productividad. Tampoco le recomiendo a nadie que se case con ninguna IDE: por motivos de trabajo, o por cambios en el mercado puede conveniros en cualquier momento cambiar de un IDE a otro. Para un programador experto que domina las bases del lenguaje unos días son más que suficientes para sentirse cómodo en un nuevo IDE.

10.2 JCREATOR [HTTP://WWW.JCREATOR.COM/](http://www.jcreator.com/)

Se trata de una IDE ligera, no posee, por ejemplo, capacidades para desarrollo de aplicaciones de modo gráfico. Sin embargo es interesante ya que requiere máquinas poco potentes para ejecutarse, mientras que las siguientes, e incluso BlueJ, requieren equipos bastante más potentes.

Yo la he empleado en algunos cursos de iniciación a Java como el primer IDE que emplean los alumnos, pero fue antes de conocer BlueJ y de enamorarme de él. No obstante, a diferencia de BlueJ este producto sí es un IDE que puede ser empleado por un programador profesional (BlueJ pierde todo interés más allá de la docencia), con lo cual los que empiecen directamente con él podrán “aguantar” más que los que se decidan por BlueJ.

Es un IDE comercial, con dos versiones:

JCreator : gratis.

JCreator Pro : 69\$ (35 \$ Licencia de estudiante).

10.3 JBUILDER [HTTP://WWW.CODEGEAR.COM/TABID/102/DEFAULT.ASPX](http://www.codegear.com/tabid/102/default.aspx)

Fue el mejor IDE para Java del mercado, aunque en el momento de escribir estas líneas (la herramienta ha dejado de ser propiedad de Borland para pasar a depender de CodeGear, una nueva compañía que se escinde de la primera) su futuro es un tanto incierto. Requiere máquinas potentes, la última versión (2007) se arrastra bastante con un equipo inferior a un Pentium IV a 1500 con 1024 megas de RAM.

Posee capacidades de desarrollo visual de aplicaciones, así como múltiples asistentes y está integrada con una gran cantidad de herramientas, unas Opensource (Ant, CVS, Struts,

Tomcat...) y otras propietarias (Borland Application Server, JDataStore, ClearCase, Visual SouceSafe ...).

Nuevamente es comercial, y los precios de las versiones son bastante prohibitivos:

JBuilder 2007 Professional: 350 \$

JBuilder 2007 Developer: 800 \$

JBuilder 2007 Enterprise : 2.500 \$

Existen versiones gratuitas de evaluación por 30 días de todos los productos.

10.4 NETBEANS [HTTP://WWW.NETBEANS.ORG/](http://www.netbeans.org/)

Quizás sea la opción más lógica a seguir después de BlueJ, ya que esta herramienta cuenta con una edición especial para facilitar la transición a los usuarios de BlueJ a un entorno de desarrollo profesional y no sólo académico.

Netbeans permite a el diseño de aplicaciones de modo visual a través de su herramienta Matisse siendo, en estos momentos, probablemente el mejor diseñador gráfico de interfaces de usuario existente para Java.

La herramienta proporciona soporte para el desarrollo de aplicaciones para dispositivos móviles (como, por ejemplo, tu teléfono móvil), para desarrollar aplicaciones web, de acceso a bases de datos, y un largo etcétera que al lector todavía le queda muy lejos. Es una herramienta que puede ser empleada por desarrolladores profesionales que se ganan el pan de cada día picando código. Esta herramienta Open Source está respaldada por Sun Microsystems.

10.5 ECLIPSE [HTTP://WWW.ECLIPSE.ORG/](http://www.eclipse.org/)

Es una IDE extensible en base a plugins (<http://www.javahispano.org/articles.article.action?id=75>). Esta filosofía, junto con la actividad que muestra su comunidad han convertido a esta herramienta en el líder indiscutible en cuanto a número de usuarios dentro de la plataforma Java. Sin embargo, para el autor, desde el punto de vista pedagógico deja bastante que desear. Su problema viene

sobre todo de uno de sus puntos fuertes: los plugins. Esta filosofía obliga al usuario a buscar y descargarse varios plugins o resignarse a carecer de bastante funcionalidad.

A un programador novato ya le basta con aprender Java para además tener que aprender qué plugins necesita y dónde conseguirlos.

Nota: Los datos sobre los precios, versiones y funcionalidad que se recogen en este documento están actualizados a febrero del 2007, y probablemente se queden obsoletos en un par de meses.

11 APÉNDICE C: CONVENIOS DE NOMENCLATURA EN JAVA

El lenguaje de programación Java probablemente haya sido el primero que desde un principio ha animado a los desarrolladores a emplear unos convenios de nomenclatura comunes. Emplear estos convenios trae enormes beneficios ya que facilita la comprensión de códigos ajenos al haber una regularidad en los nombres de las clases, métodos, variables, etc.

Los convenios de nomenclatura completos de lenguaje de programación Java son extensos. El lector podrá encontrarlos en "Convenciones de programación", <http://www.javahispano.org/tutorials.item.action?id=4>. Aquí simplemente recogemos un brevísimo resumen de estos convenios orientados al programador novato.

- Los nombres de las clases e interfaces deben empezar siempre por mayúscula (Nombre). Si el nombre surge de componer varias palabras la letra inicial de cada una de esas palabras irá en mayúscula (NombreCompuesto).
- Los nombres de las variables y los métodos empezarán siempre por minúscula (variable, método()). Si los nombres surgen de componer varias palabras la primera palabra empezará por minúscula y todas las demás empezarán por mayúscula (variableCompuesta, métodoCompuesto()).
- Los nombres de los paquetes irán siempre con todas sus letras en minúscula (paquete).
- Los distintos valores que pueden tomar las variables de tipo numeración se escriben con todas sus letras en mayúscula.
- Los métodos que permitan acceder al valor de una propiedad de un objeto (variable de una clase) se llamarán "getXXX". Así, el método que permite acceder a la propiedad "nombre" se llamará getNombre(). La única excepción es cuando la propiedad se representa mediante un boolean; en ese caso el método se llamará "isXXX". Así, el método que permite acceder a la propiedad de tipo boolean "vivo" se llamará isVivo().
- Los métodos que permitan modificar el valor de una propiedad de un objeto (variable de una clase) se llamarán "setXXX". Así, el método que permite modificar la propiedad "nombre" se llamará setNombre(...).

