

# PRÓLOGO

---

## ***Introducción***

Java a través de ejemplos presenta los conceptos básicos del lenguaje de programación Java y muestra sus posibilidades más relevantes para el desarrollo de aplicaciones. El libro se centra en la explicación detallada de los fundamentos del lenguaje, evitando los temas de carácter especializado.

La didáctica de esta publicación se basa en el aprendizaje guiado mediante ejemplos comentados y gráficos de apoyo a las explicaciones, evitándose las aclaraciones puramente teóricas y el uso de un nivel de detalle excesivo en la información suministrada.

Este libro, por tanto, resulta especialmente adecuado para los lectores que no poseen unos conocimientos previos del lenguaje o para aquellos cuyo principal objetivo es adquirir los fundamentos de Java para, posteriormente, poder afrontar con una base sólida el aprendizaje de aspectos específicos tales como las comunicaciones, acceso a bases de datos, presentaciones multimedia, etc.

Se incluyen dos apéndices con más de 500 preguntas y sus respuestas basadas en los conceptos fundamentales de Java. Estas cuestiones permiten que el lector pueda comprobar la corrección y completitud con los que va asimilando las diferentes materias.

Los contenidos del libro se han dividido en 9 capítulos que contienen 38 lecciones; cada una de estas lecciones forma una unidad de estudio que conviene afrontar de manera individualizada. Para facilitar la comprensión de estas unidades, las lecciones se han diseñado con un tamaño reducido: de unas 8 páginas por término medio.

En el siguiente apartado se presenta una relación de las lecciones, agrupadas en capítulos y con la referencia del número de página donde comienza cada una.

## ***Lecciones incluidas en la publicación***

### **Capítulo 1: toma de contacto, variables, tipos de datos y operadores**

LECCIÓN 1: Instalación del entorno de desarrollo

LECCIÓN 2: Primer programa en java: “Hola mundo”

LECCIÓN 3: Variables y tipos de datos

LECCIÓN 4: Operadores

### **Capítulo 2: estructuras de control**

LECCIÓN 5: El bucle FOR

LECCIÓN 6: El bucle WHILE

LECCIÓN 7: La instrucción condicional IF

LECCIÓN 8: La instrucción condicional SWITCH

LECCIÓN 9: Ejemplos

### **Capítulo 3: métodos y estructuras de datos**

LECCIÓN 10: Métodos

LECCIÓN 11: Strings

LECCIÓN 12: Matrices (arrays, vectores)

LECCIÓN 13: Ejemplos de programación

### **Capítulo 4: programación orientada a objetos usando clases**

LECCIÓN 14: Definición de clases e instancias

LECCIÓN 15: Sobrecarga de métodos y constructores

LECCIÓN 16: Ejemplos

LECCIÓN 17: Clases utilizadas como Parámetros

LECCIÓN 18: Propiedades y métodos de clase y de instancia

LECCIÓN 19: Paquetes y atributos de acceso

LECCIÓN 20: Ejemplo: máquina expendedora

### **Capítulo 5: programación orientada a objetos usando herencia**

LECCIÓN 21: Herencia

LECCIÓN 22: Ejemplos

LECCIÓN 23: Polimorfismo

LECCIÓN 24: Clases abstractas e interfaces

### **Capítulo 6: excepciones**

LECCIÓN 25: Excepciones predefinidas

LECCIÓN 26: Excepciones definidas por el programador

### **Capítulo 7: interfaz gráfico de usuario**

LECCIÓN 27: Creación de ventanas

LECCIÓN 28: Paneles y objetos de disposición (*layouts*)

LECCIÓN 29: Etiquetas, campos y áreas de texto

LECCIÓN 30: Cajas de verificación, botones de radio y listas

- LECCIÓN 31: Diseño de formularios
- LECCIÓN 32: Diálogos y menús

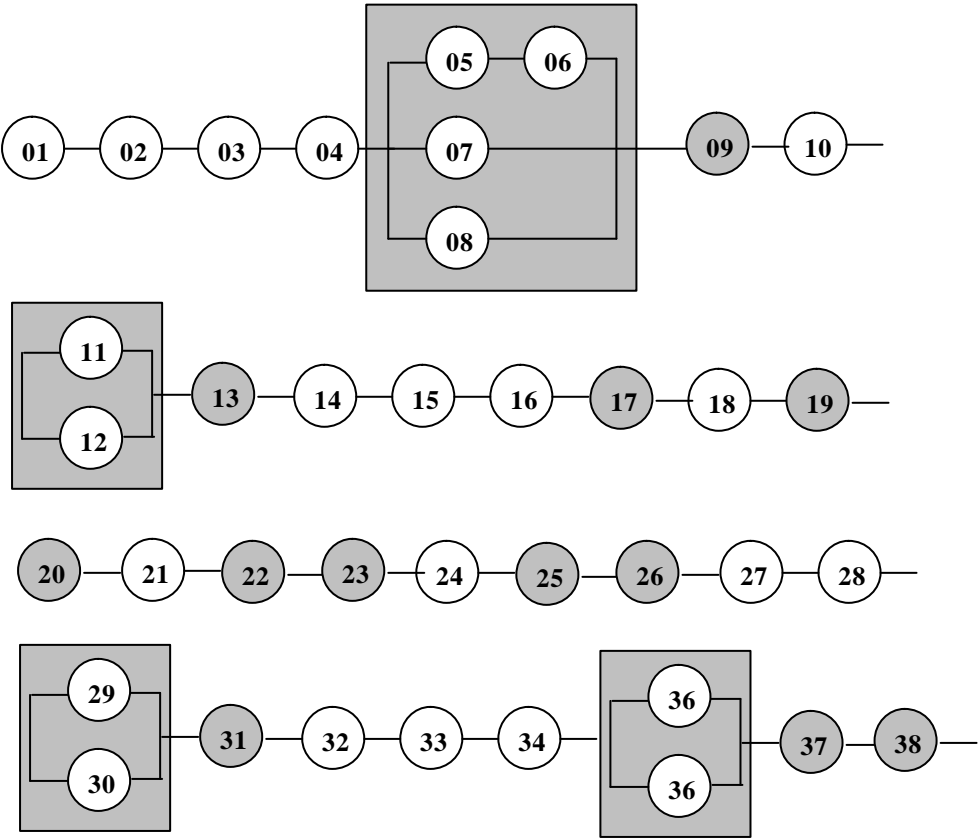
Capítulo 8: eventos

- LECCIÓN 33: Mecanismo de eventos en java
- LECCIÓN 34: Eventos de ratón y de movimiento de ratón
- LECCIÓN 35: Eventos de teclado y de ventana
- LECCIÓN 36: Eventos de acción, enfoque y elemento

Capítulo 9: aplicaciones de ejemplo

- LECCIÓN 37: Ejemplo: calculadora
- LECCIÓN 38: Ejemplo: editor

Planificación en el estudio de las lecciones



El diagrama presentado en la página anterior nos muestra la secuencia de las lecciones contenidas en el libro. El estudio de las lecciones que presentan fondo de color blanco es importante para conseguir un nivel adecuado de conocimientos en el lenguaje Java.

Las lecciones del gráfico que tienen fondo gris pueden ser omitidas en caso de necesidad; esto es posible debido a que se corresponden con ejercicios de consolidación de los temas anteriores o porque se refieren a características del lenguaje que se utilizan con menos frecuencia. En cualquier caso, el estudio de estas lecciones resulta importante para conseguir una mejor asimilación de los conceptos explicados, por lo que es muy conveniente acudir a ellas una vez comprendidas las lecciones que les preceden.

Los cuadrados grises contienen lecciones que pueden ser abordadas en cualquier orden, por ejemplo, después de la lección 10, podemos estudiar en primer lugar la 12 y después la 11 o viceversa.

Esperamos que estas indicaciones puedan ayudar a ahorrar tiempo a los lectores que desean estudiar en primer lugar las características más relevantes del lenguaje, pudiendo profundizar posteriormente en dichos conceptos y en algún otro menos utilizado en la programación general.

# ÍNDICE

---

<b>PRÓLOGO .....</b>	<b>XIII</b>
----------------------	-------------

<b>CAPÍTULO 1: TOMA DE CONTACTO, VARIABLES, TIPOS DE DATOS Y OPERADORES .....</b>	<b>1</b>
---	----------

1.1 Instalación del entorno de desarrollo.....	1
1.2 Primer programa en java: “Hola mundo” .....	5
1.3 Variables y tipos de datos.....	7
1.3.1 Variables .....	7
1.3.2 Tipos de datos .....	9
1.3.3 Tipos numéricos enteros .....	10
1.3.4 Tipos numéricos decimales .....	12
1.3.5 Tipo booleano .....	13
1.3.6 Tipo carácter .....	14
1.3.7 Conversión exp lícita de tipos (Casting).....	15
1.4 Operadores .....	16
1.4.1 Introducción.....	16
1.4.2 Operadores aritméticos.....	17
1.4.3 Operadores lógicos .....	18
1.4.4 Operadores de comparación.....	19

<b>CAPÍTULO 2: ESTRUCTURAS DE CONTROL .....</b>	<b>21</b>
---	-----------

2.1 El bucle FOR .....	21
2.1.1 Sintaxis .....	21
2.1.2 Ejemplos de aprendizaje .....	22
2.1.3 Situaciones erróneas.....	24
2.1.4 Ejemplos de resolución de problemas .....	26
2.2 El bucle WHILE .....	29
2.2.1 Sintaxis .....	30

2.2.2 Ejemplos de aprendizaje.....	30
2.2.3 Ejemplos de resolución de problemas .....	32
2.3 La instrucción condicional IF.....	35
2.3.1 Sintaxis .....	35
2.3.2 Ejemplos de aprendizaje.....	36
2.3.3 If anidados .....	37
2.3.4 Situaciones erróneas.....	38
2.3.5 Ejemplo de resolución de problemas .....	39
2.4 La instrucción condicional SWITCH.....	40
2.4.1 Sintaxis .....	40
2.4.2 Ejemplos de aprendizaje.....	41
2.4.3 Switch anidados.....	44
2.4.4 Situaciones erróneas.....	46
2.4.5 Ejemplo de resolución de problemas .....	47
2.5 Ejemplos.....	49
2.5.1 Cálculo de la hipotenusa de un triángulo .....	50
2.5.2 Punto de corte de dos rectas situadas en el espacio bidimensional.....	52
2.5.3 Soluciones de una ecuación de segundo grado.....	55

**CAPÍTULO 3: MÉTODOS Y ESTRUCTURAS DE DATOS ..... 57**

3.1 Métodos.....	57
3.1.1 Sintaxis .....	58
3.1.2 Ejemplo 1.....	59
3.1.3 Resultados del ejemplo 1 .....	60
3.1.4 Ejemplo 2.....	60
3.1.5 Resultados del ejemplo 2.....	61
3.1.6 Paso de argumentos por valor y por referencia .....	61
3.2 Strings.....	62
3.2.1 Sintaxis .....	62
3.2.2 Ejemplo básico.....	64
3.2.3 Resultado.....	65
3.2.4 Ejemplo de utilización de la clase String .....	65
3.2.5 Resultados.....	66
3.3 Matrices (arrays, vectores).....	66
3.3.1 Sintaxis .....	67
3.3.2 Acceso a los datos de una matriz.....	68
3.3.3 Ejemplo 1.....	69
3.3.4 Resultados del ejemplo 1 .....	70
3.3.5 Ejemplo 2.....	70
3.3.6 Resultados del ejemplo 2 .....	72
3.3.7 Ejemplo .....	72
3.4 Ejemplos de programación.....	74
3.4.1 Obtención de números primos .....	74
3.4.2 “Revienta claves”.....	76
3.4.3 Estadísticas .....	78

**CAPÍTULO 4: PROGRAMACIÓN ORIENTADA A OBJETOS  
USANDO CLASES ..... 83**

4.1 Definición de clases e instancias.....83

    4.1.1 Sintaxis .....83

    4.1.2 Representación gráfica .....84

    4.1.3 Instancias de una clase.....84

    4.1.4 Utilización de los métodos y propiedades de una clase.....86

    4.1.5 Ejemplo completo .....87

    4.1.6 Resultado.....89

4.2 Sobrecarga de métodos y constructores .....89

    4.2.1 Sobrecarga de métodos.....89

    4.2.2 Ejemplo .....90

    4.2.3 Resultado .....93

    4.2.4 Constructores.....93

4.3 Ejemplos.....96

    4.3.1 Figura genérica .....96

    4.3.2 Agenda de teléfono .....98

    4.3.3 Ejercicio de logística.....100

4.4 Clases utilizadas como Parámetros.....104

    4.4.1 Código .....105

    4.4.2 Resultados.....109

4.5 Propiedades y métodos de clase y de instancia .....109

    4.5.1 Propiedades de instancia .....109

    4.5.2 Propiedades de clase.....111

    4.5.3 Métodos de instancia .....113

    4.5.4 Métodos de clase.....113

    4.5.5 Ejemplo que utiliza propiedades de clase .....115

    4.5.6 Código del ejemplo .....116

    4.5.7 Resultados.....120

4.6 Paquetes y atributos de acceso.....120

    4.6.1 Definición de paquetes .....121

    4.6.2 Utilización de las clases de un paquete.....122

    4.6.3 Proceso de compilación cuando se utilizan paquetes .....123

    4.6.4 Atributos de acceso a los miembros (propiedades y métodos) de una clase.....124

4.7 Ejemplo: máquina expendedora .....125

**CAPÍTULO 5: PROGRAMACIÓN ORIENTADA A OBJETOS  
USANDO HERENCIA..... 137**

5.1 Herencia.....137

    5.1.1 Funcionamiento básico.....137

    5.1.2 Accesibilidad a los miembros heredados en una subclase.....140

    5.1.3 Constructores de las subclases .....141

    5.1.4 El modificador final.....145

5.2 Ejemplos.....146

5.2.1 Figuras geométricas .....	146
5.2.2 Vehículos .....	150
5.3 Polimorfismo .....	155
5.3.1 Conceptos .....	155
5.3.2 Ejemplo .....	157
5.4 Clases abstractas e interfaces .....	160
5.4.1 Métodos abstractos .....	160
5.4.2 Clases abstractas .....	161
5.4.3 Interfaces .....	164
5.4.4 Ejercicio .....	167
<b>CAPÍTULO 6: EXCEPCIONES .....</b>	<b>171</b>
6.1 Excepciones predefinidas .....	171
6.1.1 Introducción .....	171
6.1.2 Sintaxis y funcionamiento con una sola excepción .....	174
6.1.3 Ejemplo con una sola excepción .....	176
6.1.4 Sintaxis y funcionamiento con más de una excepción .....	177
6.1.5 El bloque finally .....	180
6.1.6 Propagación de excepciones .....	180
6.1.7 Estado de una excepción .....	182
6.2 Excepciones definidas por el programador .....	184
6.2.1 Introducción .....	184
6.2.2 Definición de una excepción definida por el programador .....	185
6.2.3 Utilización de una excepción definida por el programador .....	186
6.2.4 Ejemplo .....	187
<b>CAPÍTULO 7: INTERFAZ GRÁFICO DE USUARIO .....</b>	<b>193</b>
7.1 Creación de ventanas .....	193
7.1.1 Introducción .....	193
7.1.2 Utilización de la clase Frame .....	195
7.1.3 Creación y posicionamiento de múltiples ventanas .....	196
7.2 Paneles y objetos de disposición ( <i>layouts</i> ) .....	199
7.2.1 Introducción .....	199
7.2.2 Utilización básica de paneles .....	202
7.2.3 Objeto de disposición (Layout): <code>FlowLayout</code> .....	203
7.2.4 Objeto de disposición (Layout): <code>BorderLayout</code> .....	205
7.2.5 Objeto de disposición (Layout): <code>GridLayout</code> .....	208
7.3 Etiquetas, campos y áreas de texto .....	211
7.3.1 Introducción .....	211
7.3.2 Etiqueta ( <code>Label</code> ) .....	211
7.3.3 Campo de texto ( <code>TextField</code> ) .....	215
7.3.4 Área de texto ( <code>TextArea</code> ) .....	218
7.3.5 Fuentes ( <code>Font</code> ) .....	219



7.4 Cajas de verificación, botones de radio y listas..... 221

    7.4.1 Introducción ..... 221

    7.4.2 Cajas de verificación (Checkbox) ..... 221

    7.4.3 Botones de radio (CheckboxGroup)..... 223

    7.4.4 Lista (List)..... 226

    7.4.5 Lista desplegable (Choice)..... 229

7.5 Diseño de formularios..... 230

    7.5.1 Diseño del interfaz gráfico deseado ..... 230

    7.5.2 Implementación en una sola clase..... 232

    7.5.3 Implementación en varias clases..... 235

7.6 Diálogos y menús ..... 240

    7.6.1 Introducción ..... 240

    7.6.2 Diálogo (Dialog)..... 241

    7.6.3 Diálogo de carga / almacenamiento de ficheros (FileDialog) ..... 243

    7.6.4 Menús (Menu y MenuBar) ..... 245

**CAPÍTULO 8: EVENTOS..... 247**

8.1 Mecanismo de eventos en java..... 247

    8.1 Introducción..... 247

    8.2 Arquitectura de los eventos..... 248

    8.3 Interfaces que soportan el mecanismo de eventos ..... 250

    8.4 Esquema general de programación ..... 253

8.2 Eventos de ratón y de movimiento de ratón..... 257

    8.2.1 Introducción ..... 257

    8.2.2 Eventos de ratón..... 259

    8.2.3 Eventos de movimiento de ratón ..... 267

8.3 Eventos de teclado y de ventana..... 272

    8.3.1 Introducción ..... 272

    8.3.2 Eventos de teclado..... 274

    8.3.3 Eventos de ventana..... 276

8.4 Eventos de acción, enfoque y elemento ..... 278

    8.4.1 Introducción ..... 278

    8.4.2 Eventos de acción..... 280

    8.4.3 Eventos de enfoque..... 284

    8.4.4 Eventos de elemento..... 287

**CAPÍTULO 9: APLICACIONES DE EJEMPLO ..... 293**

9.1 Ejemplo: calculadora ..... 293

    9.1.1 Definición del ejemplo ..... 293

    9.1.2 Diseño del interfaz gráfico de usuario ..... 294

    9.1.3 Implementación del interfaz gráfico de usuario ..... 296

    9.1.4 Diseño del tratamiento de eventos..... 300

    9.1.5 Implementación de las clases de tratamiento de eventos..... 302

9.1.6	Diseño del control .....	305
9.1.7	Implementación del control .....	307
9.2	Ejemplo: editor.....	316
9.2.1	Definición del ejemplo .....	316
9.2.2	Estructura de la aplicación .....	318
9.2.3	Propiedades de la clase.....	321
9.2.4	Constructores .....	322
9.2.5	Método PreparaMenus.....	324
9.2.6	Método PreparaZonaInferior.....	325
9.2.7	Clase Colores .....	325
9.2.8	Tratamiento de las opciones de menú .....	327
9.2.9	Tratamiento de las opciones “Buscar” y “Reemplazar” .....	329
9.2.10	Tratamiento de los botones de radio “Color texto” y “Color fondo” .....	331
9.2.11	Tratamiento a las pulsaciones de los botones de colores .....	331

**APÉNDICES**

<b>A: CUESTIONES DE AUTOEVALUACIÓN .....</b>	<b>333</b>
<b>B: SOLUCIONES A LAS CUESTIONES DE AUTOEVALUACIÓN .....</b>	<b>359</b>
<b>ÍNDICE ALFABÉTICO .....</b>	<b>361</b>

# TOMA DE CONTACTO, VARIABLES, TIPOS DE DATOS Y OPERADORES

---

## 1.1 INSTALACIÓN DEL ENTORNO DE DESARROLLO

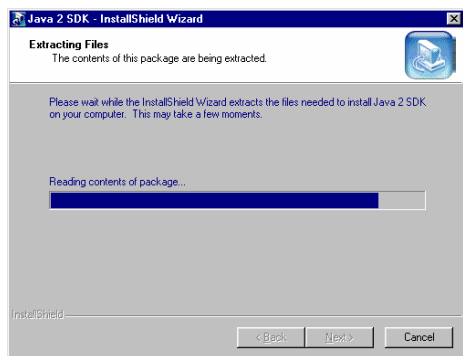
Para poder realizar programas en lenguaje Java es necesario disponer de un mínimo de herramientas que nos permitan editar, compilar e interpretar el código que diseñamos. Para escribir físicamente los programas podemos utilizar cualquier editor de texto (por ejemplo el *bloc de notas*, el *WordPad*, etc.). Para compilar y ejecutar los programas existen dos opciones:

- Utilizar un entorno integrado de desarrollo (por ejemplo *JBuilder* de *Borland*, *Visual J++* de *Microsoft*, etc.)
- Emplear el software básico de desarrollo (SDK) de *Sun Microsystems*

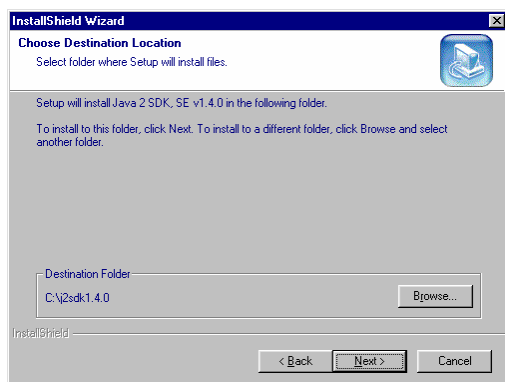
La primera opción resulta especialmente interesante para afrontar la creación de aplicaciones de manera eficiente, puesto que estos entornos facilitan enormemente el diseño, escritura y depuración de los programas. La segunda opción es mucho más adecuada para aprender a programar en Java, porque no existe la generación automática de código que incorporan los entornos integrados de desarrollo.

El SDK de *Sun Microsystems* puede ser obtenido de forma gratuita (es un software de libre distribución) en la dirección [www.sun.com](http://www.sun.com); una vez que se dispone del fichero con el SDK (por ejemplo *j2sdk-1\_4\_0-win.exe*) basta con ejecutarlo para conseguir que el software se instale en su ordenador.

En primer lugar se descomprime automáticamente la información:



Por defecto, el software se instalará a partir de un directorio cuyo nombre tiene que ver con la versión del SDK (en este caso *j2sdk1.4.0*), aunque siempre puede elegirse una ubicación personalizada (con la opción *Browse*).

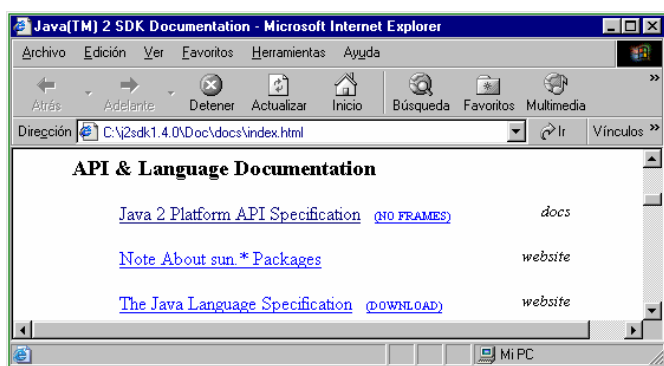


La propia instalación rellenará la variable de entorno *CLASSPATH* con los valores adecuados (donde el entorno debe buscar los ficheros compilados *.class*). La variable de entorno *PATH* la deberemos actualizar para que el sistema operativo encuentre los ficheros ejecutables del SDK, que en nuestro caso están ubicados en el directorio *c:\j2sdk1.4.0\bin* (suponiendo que hemos realizado la instalación sobre el disco *c*: y el directorio *j2sdk1.4.0*).

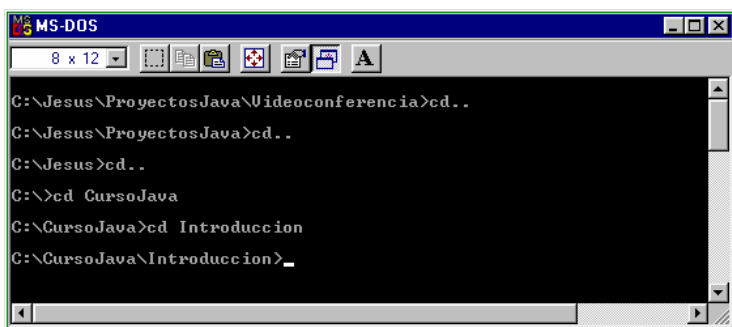
Para asegurarse de que la variable *CLASSPATH* se encuentra correctamente establecida, podemos consultar su valor de la siguiente manera:



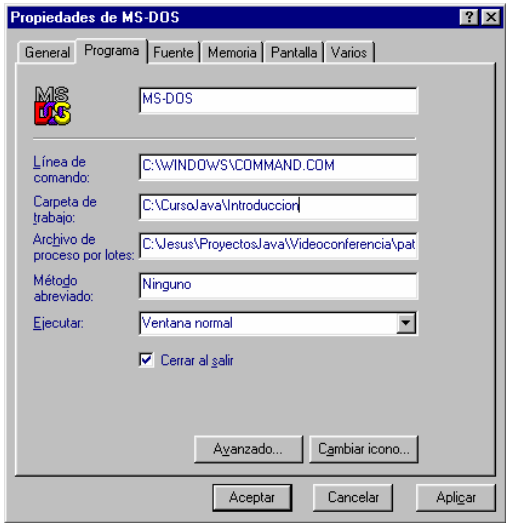
*Sun Microsystems* proporciona, además, ayuda en línea sobre las características del lenguaje y las clases más comunes para el desarrollo de aplicaciones. Escribiendo programas en Java resulta necesario instalar esta ayuda para poder consultar los métodos y clases disponibles. La ayuda puede ser obtenida en Internet a través del sitio Web de *Sun* ([www.sun.com](http://www.sun.com)). Una vez instalada, se accede a ella a través de su página principal (*index.html*). Para escribir programas en Java, el enlace más importante de esta página es: *Java 2 Platform API Specification*, correspondiente al apartado *API & Language Documentation*.



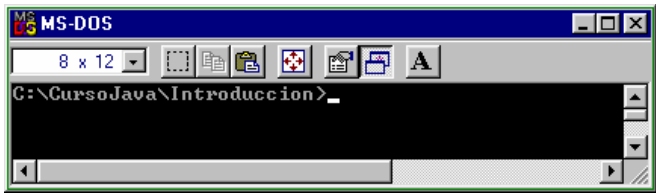
El software de desarrollo SDK funciona en modo texto sobre una ventana de consola del sistema operativo. En Windows, para acceder a la consola se debe pulsar sobre el símbolo de MS-DOS (habitualmente accesible a través del botón de *Inicio*). Una vez en la consola, dirigirse al directorio de trabajo (en el ejemplo *C:\CursoJava\Introducción*), tal y como aparece en la siguiente figura:



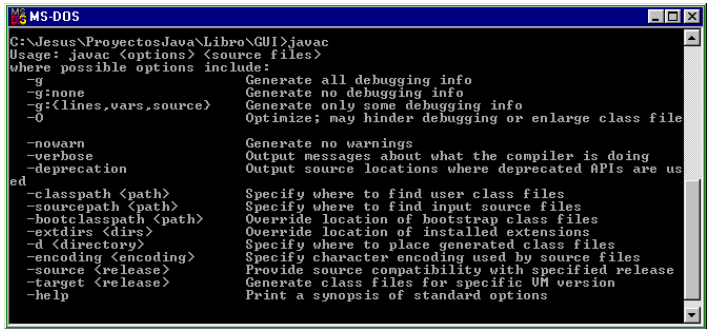
Podemos configurar la consola en cuanto al tamaño con el que aparece, memoria que utiliza, fuentes que emplea, etc. Para realizar esta configuración basta con acceder al símbolo de MS-DOS y, en lugar de pulsar con el botón izquierdo del ratón, hacerlo con el derecho y seleccionar *Propiedades*.



En concreto, nos interesa seleccionar la solapa *Programa* y variar el valor del campo de texto *Carpeta de trabajo*, introduciendo el directorio sobre el que solemos trabajar, con ello conseguimos que la consola se abra directamente en ese directorio. En nuestro ejemplo, hemos introducido el valor *C:\CursoJava\Introducción*.



Como veremos en el siguiente apartado, el fichero *javac.exe* es el compilador de java. Se encuentra en el directorio *c:\j2sdk1.4.0\bin* (hacia donde hemos apuntado la variable *PATH*). Si todo se encuentra correctamente al invocar el compilador sin parámetros nos indica la forma de uso correcta, tal y como aparece en la siguiente figura:



## 1.2 PRIMER PROGRAMA EN JAVA: “HOLA MUNDO”

El aprendizaje de todo lenguaje de programación pasa por la etapa obligatoria de realizar un primer programa, lo más sencillo posible, que muestre:

- La estructura sintáctica mínima a la que obliga el lenguaje
- La manera de introducir, traducir y ejecutar el programa
- La validez del entorno en el que se sustentarán los desarrollos (para nosotros el SDK)

Nuestro programa únicamente escribirá el texto “Hola mundo”, pero servirá para asentar cada uno de los puntos expuestos en el párrafo anterior.

Pasos detallados:

- 1 Abrir un editor de texto (por ejemplo el *WordPad*)

Habitualmente a través del botón de Inicio: (Inicio → Programas → Accesorios → WordPad).

- 2 Introducir el código (sin incluir los números de línea):

```
1 public class HolaMundo {
2     public static void main (String[] args) {
3         System.out.println("Hola Mundo");
4     }
5 }
```

Es necesario respetar la condición mayúscula/minúscula de cada letra del programa, puesto que en este lenguaje una letra en minúscula es diferente a su correspondiente en mayúsculas.

La línea 1 define la clase (objeto) *HolaMundo*. Java es un lenguaje orientado a objetos, donde toda las aplicaciones se estructuran en grupos de objetos (clases en Java). La clase se define como *public* (pública), indicando que será accesible a cualquier otra clase. El último carácter de la línea 1 es una llave de comienzo; indica el comienzo de la clase. La clase termina en la línea 5 (con la llave de fin).

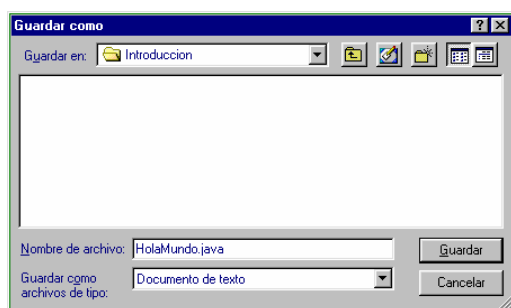
La línea 2 define un método (procedimiento, subrutina) de la clase *HolaMundo*. Este método es especial, le indica al entorno de Java el comienzo de nuestra aplicación. Su nombre (identificador) es: *main* (método principal). Este método siempre lleva un parámetro *String[]* que identifica un conjunto de literales (textos); por ahora no emplearemos esta característica, aunque debemos respetar su sintaxis. El método es público y estático (atributos que veremos en detalle en los temas siguientes).

El contenido del método *main*, en nuestro ejemplo, se encuentra delimitado entre la llave de inicio situada en la línea 2 y la llave de fin

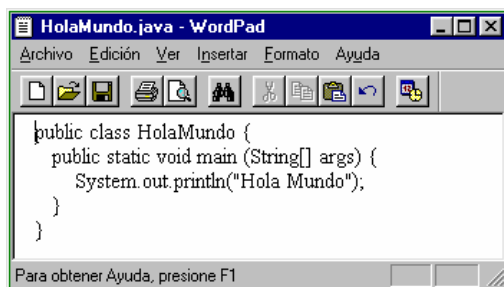
situada en la línea 4. Únicamente contiene la instrucción situada en la línea 3, que imprime el literal (texto) que aparece entre comillas y como parámetro de la llamada al método *println*. Obsérvese el carácter ‘;’ que se debe utilizar obligatoriamente para separar instrucciones.

El sangrado de cada una de las líneas no es necesario, pero resulta muy conveniente para hacer más legible los programas. Su función principal es facilitar la identificación visual de cada bloque de código.

### 3 Grabar el código en el fichero *HolaMundo.java* y con formato de texto



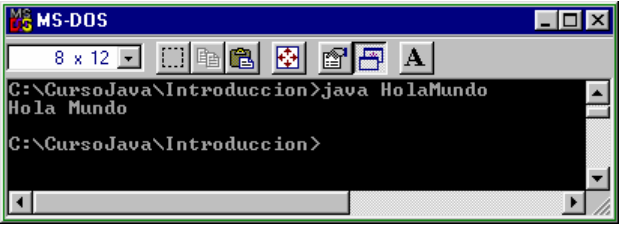
Es absolutamente necesario que la extensión sea *.java*, que el fichero este grabado con formato de texto y que el nombre de la clase coincida EXACTAMENTE con el nombre del fichero (en nuestro caso *HolaMundo*). Finalmente, en el *WordPad* nos quedará una ventana similar a la siguiente:



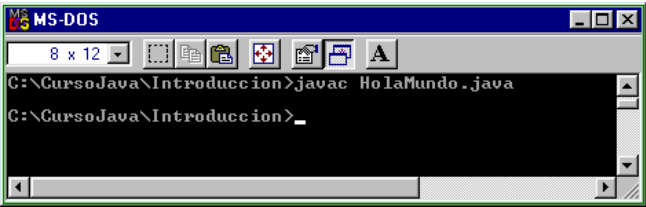
### 4 Compilar el programa

En la ventana de MS-DOS, en el directorio donde hemos grabado el fichero *HolaMundo.java*, debemos ejecutar el compilador (*javac*) poniendo como argumento el nombre del fichero CON LA EXTENSIÓN *.java* (*javac HolaMundo.java*). Si existen errores, se obtendrá un listado de los mismos. Si no los hay, como en nuestro caso, no aparece nada:



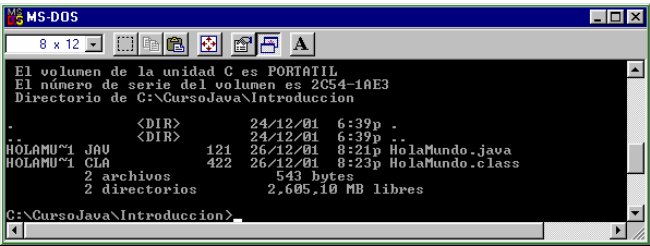


Tras la compilación con éxito del programa, obtenemos el fichero objeto *HolaMundo.class*:



5 Ejecutar el programa

En el directorio en el que estamos trabajando, ejecutar el intérprete de java (*java*) suministrando como parámetro el nombre de la clase (*HolaMundo*) SIN la extensión *.class* (*java HolaMundo*). Nos aparecerá el texto “Hola Mundo” correspondiente a la impresión de datos por consola que codificamos en la línea 3 de nuestro programa:



1.3 VARIABLES Y TIPOS DE DATOS

1.3.1 Variables

Las variables se utilizan para almacenar datos asociados a nombres. Cada variable tiene un nombre que sirve de referencia para introducir datos o acceder a los

misimos. Un ejemplo de utilización de variable es: *Edad\_Pedro* = 23; en este caso, introducimos el valor numérico 23 en la variable con nombre *Edad\_Pedro*. A partir de ahora podemos utilizar la variable a través de su nombre (*Edad\_Pedro*) para referirnos a su valor (23), por ejemplo estableciendo *Edad\_Ana* = *Edad\_Pedro* - 2; donde Ana podría ser la hermana pequeña de Pedro, que es dos años más joven que él.

Los nombres de las variables, en Java, pueden tener cualquier longitud y deben comenzar con una letra, el símbolo de subrayado “\_” o el dólar “\$”. El resto de los caracteres del nombre pueden ser cualquiera, salvo los que pueden dar lugar a confusión, como los operadores (+, -, \*, etc.). Por ejemplo, sería correcto el nombre *MiAmigoMasQuerido*, pero no el nombre *MiAmigo+Querido*, puesto que en este último caso el compilador interpretaría que hay que sumar el contenido de la variable *MiAmigo* con el contenido de la variable *Querido*.

Los nombres de variables no deben tener espacios en blanco, puesto que el compilador identificaría más de una variable; por ejemplo, *NumeroDeAciertos* es un nombre de variable correcto, sin embargo *Numero De Aciertos* no lo es, porque el compilador identificaría tres variables diferentes: *Numero*, *De* y *Aciertos*.

Por último, los nombres de variables no deben coincidir con palabras reservadas (tales como *public*, *static*, *class*, *void*, *main*, etc.), puesto que estos identificadores tienen un significado especial para el compilador.

En Java, los identificadores de variables suelen definirse empezando por un carácter en minúscula, por ejemplo *contador*. Si en el identificador existe más de una palabra, los comienzos del resto de las palabras se ponen con mayúsculas, ejemplo: *contadorDeAciertos*. En este curso seguiremos una notación parecida, aunque no idéntica; comenzaremos todas las palabras con mayúsculas: *ContadorDeAciertos*.

Para asentar los conceptos expresados en este apartado, veamos una serie de casos de identificadores de variables correctos e incorrectos:

*LaCasaDeLaPradera* → identificador correcto

*El Hombre Sin Rostro* → identificador incorrecto, no debe existir ningún espacio en blanco en un nombre de variable

*3Deseos* → identificador incorrecto, el nombre no empieza por una letra (sino por un número)

*TresDeseos* → identificador correcto

*\_4* → identificador correcto

*\$* → identificador correcto

*\$Ganado* → identificador correcto

*public* → identificador incorrecto, *public* es un nombre reservado por el lenguaje

Los identificadores deben intentar ser representativos de los datos que albergan, de esta manera *ValorAcumulado*, *NumeroAlumnos*, *CantidadEuros*, *Edad*, *Potencia* son variables que determinan de forma adecuada el significado de sus contenidos, mientras que los nombres *Variable*, *Valor*, *V1*, *V2*, no son representativos de sus contenidos.

La longitud de los identificadores no debe ser excesivamente larga, para no dificultar la legibilidad de las instrucciones, por ejemplo resulta mucho más legible *Stock = LibrosEditados – LibrosVendidos*, que *StockTotalDeLibrosEnAlmacen = LibrosEditadosEnElAñoEnCurso – LibrosVendidosYSacadosDelAlmacen*.

Los identificadores en mayúsculas se suelen reservar para nombres (normalmente cortos) de constantes, de esta manera las instrucciones (basadas mayoritariamente en operaciones con variables) son más legibles, al utilizarse minúsculas. Compárese *Stock = LibrosEditados – LibrosVendidos* con *STOCK = LIBROSEEDITADOS – LIBROSVENDIDOS*.

### 1.3.2 Tipos de datos

Las variables albergan datos de diferentes tipos (numérico decimal, numérico entero, caracteres, etc.). Para indicar el tipo de dato que contendrán las variables debemos “declarar” las mismas, indicando sus tipos. Este mecanismo permite que el traductor (compilador) realice comprobaciones estáticas de validez, como por ejemplo que no empleamos en el programa una variable que no ha sido declarada, que no asignemos un carácter a una variable de tipo numérico, que no sumemos un carácter a un valor numérico, etc.

A continuación se establece una relación de los tipos de datos **primitivos** (los que proporciona el lenguaje):

TIPOS PRIMITIVOS		
Nombre del tipo	Tamaño en bytes	Rango
Tipos numéricos enteros		
byte	1	-128 a 127

<b>short</b>	2	-32768 a 32767
<b>int</b>	4	-2 <sup>31</sup> a 2 <sup>31</sup>
<b>long</b>	8	-2 <sup>63</sup> a 2 <sup>63</sup>
Tipos numéricos decimales		
<b>float</b>	4	-3.4x10 <sup>38</sup> a 3.4x10 <sup>38</sup>
<b>double</b>	8	-1.7x10 <sup>308</sup> a 1.7x10 <sup>308</sup>
Tipo carácter		
<b>char</b>	2	Conjunto de caracteres
Tipo lógico (booleano)		
<b>boolean</b>	1	<i>true, false</i>

Para declarar una variable se emplea la sintaxis:

*tipo identificador;*  
*tipo identificador [=valor];*  
*tipo identificador1,identificador2,identificador3,etc.;*  
*tipo identificador1 = valor1,identificador2 = valor2, etc.;*

Por ejemplo:

*byte EdadPedro = 60;*  
*short SueldoMensual;*  
*float PrecioEnEuros, Cateto1, Cateto2, Hipotenusa;*  
*boolean Adquirido = false, Finalizado = true;*

1.3.3 Tipos numéricos enteros

A continuación se proporcionan diversos programas comentados que muestran la manera de utilizar los tipos numéricos enteros:

En el primer ejemplo *TiposNumericos1*, se declaran las variables de tipo *byte EdadJuan* y *EdadPedro* (líneas 3 y 4) y las variables de tipo *short SueldoBase* y *Complementos* (líneas 6 y 7). La línea 9 muestra, a modo de ejemplo, el valor de *SueldoBase*, que es 1980.

Obsérvese que se ha seleccionado el tipo *byte* para albergar edades, por lo que asumimos que nunca tendremos que introducir un valor superior a 127; del mismo modo empleamos el tipo *short* para contener sueldos (en euros), por lo que no podremos pasar de 32767, lo que puede ser válido si nos referimos a sueldos

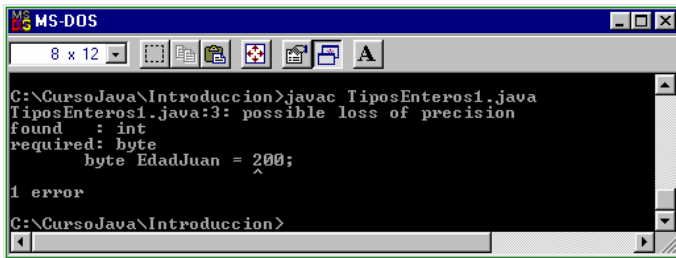
mensuales. Si estamos indicando sueldos anuales, sería conveniente pasar al siguiente tipo en tamaño (*int*).

```

1 public class TiposEnteros1 {
2     public static void main (String[] args) {
3         byte EdadJuan = 20;
4         byte EdadPedro = 22;
5
6         short SueldoBase = 1980;
7         short Complementos = 400;
8
9         System.out.println(SueldoBase);
10    }
11 }

```

Si al inicializar el valor de una variable nos pasamos en el rango permitido, el compilador nos dará un error, indicándonos el tipo que deberíamos emplear:



En el siguiente ejemplo *TiposEnteros2* se define una variable de tipo entero (*int*) y otra de tipo entero largo (*long*), cada una de ellas con valores adecuados al tipo de datos que representan. Es importante saber que los valores numéricos enteros no pueden ser representados con puntos (ejemplo 4.000.000 en lugar de 4000000), puesto que se confundirían con valores numéricos decimales.

En la línea 4 del ejemplo, se define el valor 5000000000L acabado en la letra *L*. Esta letra indica que el valor debe ser tomado como *long* antes de ser asignado a la variable. Si no ponemos esta indicación, el valor numérico se toma como *int* (por defecto) y el compilador muestra un error indicando que el número es demasiado grande para pertenecer a este tipo.

```

1 public class TiposEnteros2 {
2     public static void main (String[] args) {
3         int HabitantesEnMadrid = 4000000;
4         long HabitantesEnElMundo = 5000000000L;
5
6         System.out.println(HabitantesEnElMundo);

```

```
7    }  
8 }
```

### 1.3.4 Tipos numéricos decimales

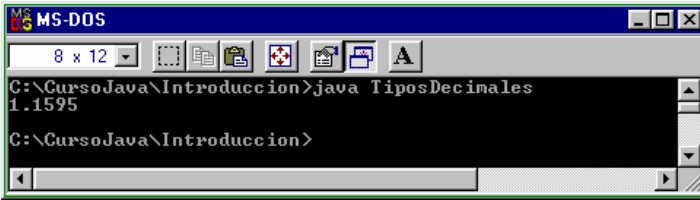
Existe una gran cantidad de valores numéricos que, por su naturaleza, requieren la utilización de decimales; Java proporciona los tipos de datos *float* y *double* para albergar este tipo de valores. Como ejemplos de valores que requieren decimales podemos establecer: el precio en euros (con centimos) de un artículo, el peso en kilos (con precisión de gramos) de una persona, la medida en metros (con precisión de centímetros) de un objeto, etc.

El siguiente ejemplo (*TiposDecimales*) muestra la manera de declarar y utilizar variables de tipo *float* y *double*. En las líneas 3 y 4 establecemos el precio de una pieza de pan en 0.87 euros y el del kilo de queso en 1.93 euros; la letra “f” con la que terminamos las declaraciones le indica al compilador que los literales numéricos (0.87 y 1.93) son de tipo *float*, si no pusiéramos esta indicación, el compilador los tomaría (por defecto) como de tipo *double*, con lo que la asignación *float* = *double* resultaría fallida.

En la línea de código 6 introducimos en la variable *\$Bocadillo* el precio de un bocadillo que contiene 150 gramos de queso.

Las líneas 8 y 9 muestran la manera de definir e inicializar variables que contienen números realmente grandes (6 elevado a 100) y números realmente pequeños (2.45 elevado a -95), para estos casos utilizamos el tipo *double*.

```
1 public class TiposDecimales {  
2     public static void main (String[] args) {  
3         float $PiezaPan = 0.87f;  
4         float $KiloQueso = 1.93f;  
5  
6         float $Bocadillo = $PiezaPan + $KiloQueso * 0.15f;  
7  
8         double NumeroHormigas = 6E+100;  
9         double DistanciaSubAtomica = 2.45E-95;  
10  
11         System.out.println($Bocadillo);  
12     }  
13 }
```



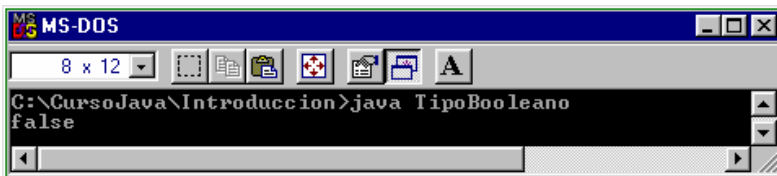
```
MS-DOS
8 x 12
C:\CursoJava\Introduccion>java TiposDecimales
1.1595
C:\CursoJava\Introduccion>
```

### 1.3.5 Tipo booleano

El tipo de datos lógico (booleano) nos permite declarar y definir variables cuyo contenido es binario. Sus únicos valores posibles son *true* (verdadero) y *false* (falso). Estas variables resultan especialmente útiles para definir condiciones lógicas aplicables al control de flujo de los programas, por ejemplo: si (*Encontrado*) hacer una cosa, en caso contrario hacer otra, siendo *Encontrado* una variable de tipo *boolean* que se modifica a lo largo del programa.

En el siguiente ejemplo (*TipoBooleano*) se utilizan diversas variables de este tipo. En la línea 3 se definen e inicializan (a *true* y *false*) dos variables booleanas. En la línea 5 se establece el contenido de una variable *Triste* como el contrario (negado) de la variable *Contento*, para ello se emplea el operador *!* que detallaremos más adelante.

```
1 public class TipoBooleano {
2     public static void main (String[] args) {
3         boolean Contento = true, MenorDeEdad = false;
4
5         boolean Triste = !Contento;
6
7         System.out.println(Triste);
8     }
9 }
```



```
MS-DOS
8 x 12
C:\CursoJava\Introduccion>java TipoBooleano
false
```

1.3.6 Tipo carácter

En Java tenemos la posibilidad de utilizar variables de tipo carácter, capaces de contener, cada variable, un carácter ('A', 'B', 'a', '\*', '8', etc.). En la línea 2 de la clase *TipoCaracter* se declaran las variables *AMayuscula* y *AMinuscula*, en donde introducimos los valores 'A' y 'a'.

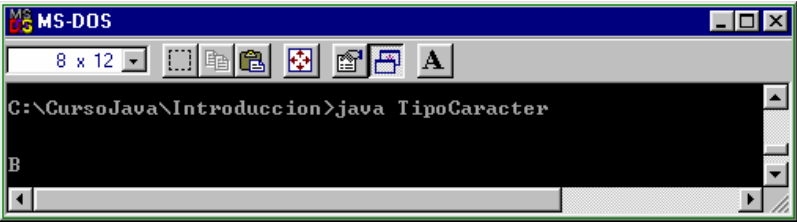
La línea 4 define la variable *Bmayuscula*, introduciendo el valor siguiente al de *AMayuscula*, para ello hay que realizar una conversión de tipos que nos sirve para ilustrar los conceptos que justamente se explican en el apartado siguiente. En la línea 6 se imprime una nueva línea y en la 7 el carácter 'B'.

```
1 public class TipoCaracter {
2     public static void main (String[] args) {
3         char AMayuscula = 'A', AMinuscula = 'a';
4         char BMayuscula = (char) (AMayuscula + 1);
5
6         System.out.println('\n');
7         System.out.println(BMayuscula);
8     }
9 }
```

También existe la posibilidad de utilizar caracteres que no podemos conseguir con facilidad en nuestro teclado, para ello se utiliza la secuencia de “escape”: \. Podemos, de esta manera, definir cualquier carácter de codificación “unicode” (codificación más extensa que ASCII) de la siguiente manera: ‘\uxxxx’, donde las equis se sustituyen por el valor numérico en hexadecimal del carácter. Por ejemplo ‘\u0041’ se corresponde con la A mayúscula.

Otros caracteres especiales son:

\b	espacio hacia atrás
\n	nueva línea
\r	retorno de carro
\t	tabulador





### 1.3.7 Conversión explícita de tipos (Casting)

Resulta muy habitual que en los programas nos encontremos con la necesidad de mezclar tipos de datos, por ejemplo al trabajar con una biblioteca matemática que obliga a utilizar tipos de datos *double*, cuando nosotros las variables las tenemos de tipo *float*; o más sencillo todavía, al intentar sumar un 1 (por defecto *int*) a una variable de tipo *short*. Java, en algunos casos, realiza una conversión implícita de datos, sin embargo, por regla general, nos veremos obligados a programar conversiones explícitas de tipos.

Para modificar el tipo de un valor, basta con indicar el nuevo tipo entre paréntesis antes del valor, por ejemplo:

*(byte)* 1 → convierte el 1 (*int*) a *byte*

*(double)* *MiVariableDeTipoFloat* → convierte a *double* una variable de tipo *float*

*(short)* (*VariableDeTipoByte* + *VariableDeTipoByte*) → convierte a *short* el resultado de sumar dos variables de tipo *byte*. Obsérvese que al sumarse dos variables de tipo *byte*, el resultado puede que no “quepa” en otra variable de tipo *byte*.

En el ejemplo siguiente: *Casting*, se realizan dos conversiones explícitas de tipo: la primera en la línea 4, donde al sumarse *EdadJuan* (*byte*) a 1 (*int*) el resultado es *int* y por lo tanto lo debemos convertir al tipo de la variable *EdadPedro* (*byte*); en este caso debemos estar muy seguros de que *EdadJuan* es distinto a 127 (límite del tipo *byte*). En la línea 11 convertimos a *short*, debido a que la operación de suma trabaja con el tipo (*int*) y por lo tanto realiza una conversión implícita de *short* a *int* en las variables *SueldoBase* y *Complementos*. El *casting* que realizamos nos convierte de *int* (resultado de la suma) a *short* (tipo de la variable *SueldoTotal*).

```

1 public class Casting {
2     public static void main (String[] args) {
3         byte EdadJuan = 20;
4         byte EdadPedro = (byte) (EdadJuan + 1);
5
6         short SueldoBase = 1980;
7         short Complementos = 400;
8
9         short SueldoTotal;
10
11        SueldoTotal = (short) (SueldoBase + Complementos);
12
13        System.out.println(SueldoTotal);
14    }
15 }
```

A menudo, en los programas escritos en Java se utiliza intensivamente el tipo *int*, cuando, en principio, sería más adecuado hacer uso de otros tipos que se ajusten mejor en tamaño a la naturaleza de los datos. La razón es evitar operaciones de conversión explícita de tipos.

## 1.4 OPERADORES

### 1.4.1 Introducción

Los operadores nos permiten realizar operaciones sobre los operandos estudiados en el apartado anterior. Existen operadores unarios y binarios; un ejemplo de operador unario que ya hemos utilizado es la negación lógica (!), lo vimos al explicar el tipo de datos booleano y lo empleamos en la instrucción: *boolean Triste = !Contento*; donde el operador negación se aplica a un único operando (la variable *Contento*). El ejemplo de operador binario que más hemos utilizado es la suma, que se aplica sobre dos operandos.

Los operadores tienen unas reglas de precedencia que resulta importante tener en cuenta. En el apartado anterior, calculábamos el precio de un bocadillo de la siguiente manera: *float \$Bocadillo = \$PiezaPan + \$KiloQueso \* 0.15f*; en este caso utilizamos dos operadores binarios: la suma (+) y la multiplicación (\*). El operador de multiplicación tiene mayor precedencia que el de suma, por lo que, de forma correcta, primero se obtiene el precio del queso y luego se realiza la suma del pan más el queso. Si el operador de suma se hubiera aplicado primero, se habría sumado *\$PiezaPan + \$KiloQueso* y, posteriormente, el resultado se habría multiplicado por 0.15f, lo que nos proporcionaría un resultado incorrecto. Cuando existen varios operadores con la misma precedencia, se aplican de izquierda a derecha.

La precedencia de los operadores se puede variar usando los paréntesis. Las operaciones que se encuentren entre paréntesis se realizan primero. En el ejemplo del bocadillo, si suponemos que ya contábamos con 0.1f kilos de queso, el nuevo precio lo podemos obtener de la siguiente manera: *float \$Bocadillo = \$PiezaPan + \$KiloQueso \* (0.15f - 0.1f)*; primero se realizará la resta, después la multiplicación y por último la suma.

Los operadores pueden ser clasificados atendiendo al tipo de operandos sobre los que actúan, de esta manera realizaremos una breve descripción de los operadores aritméticos, lógicos y de comparación, obviando a los operadores de bits que se utilizan con muy poca frecuencia.

### 1.4.2 Operadores aritméticos

Los operadores aritméticos más comunes son la suma (+) , resta (-), multiplicación (\*) y división (/) binarios, aunque también se utilizan los operadores unarios (+) y (-), y el operador binario que obtiene el módulo de una división (%). Finalmente, disponemos de operadores aritméticos de preincremento, postincremento, predecremento y postdecremento, que permiten acortar el tamaño de ciertas instrucciones y facilitar la optimización de código por parte del compilador.

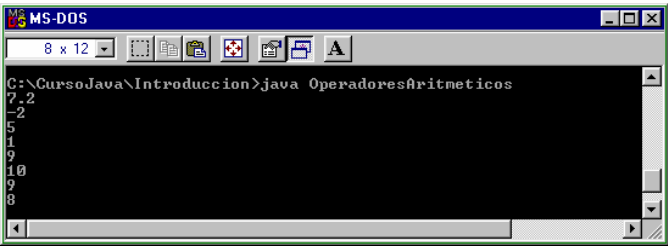
Operación	sintaxis	significado
Preincremento	++Variable;	Variable = Variable + 1; (antes de asignación)
Postincremento	Variable++;	Variable = Variable + 1; (después de asignación)
Predecremento	--Variable;	Variable = Variable – 1; (antes de asignación)
Postdecremento	Variable--;	Variable = Variable – 1; (después de asignación)

Para ilustrar los conceptos expuestos se proporciona la clase *OperadoresAritmeticos*. La línea 3 hace uso de la multiplicación y suma binarios, con resultado 7.2, mostrado en la línea 16. En la línea 4, primero actúa el operador unario (-), después el operador binario (/) y por último la suma; el resultado, mostrado en la línea 17 es -2. Las líneas de código 5 y 6 muestran, respectivamente, la división y resto enteros, obteniéndose resultados 5 y 1 en las líneas 18 y 19.

Las líneas 8 a 14 del código utilizan los operadores de pre/post incremento/decremento. A la variable *PostIncremento* de la línea 8 se le asigna el valor 9, debido a que primero se realiza la asignación *PostIncremento* = *Nueve* y posteriormente la operación de postincremento. La variable *PreIncremento* (en la línea 10) albergará el valor 10, puesto que primero se realiza el preincremento y después la asignación. Como cabe esperar, la variable *PostDecremento* recibe el valor 9 y la variable *PreDecremento* el valor 8.

```
1 public class OperadoresAritmeticos {
2     public static void main (String[] args) {
3         float Impuesto = 2.2f * 1.0f + 5.0f;
4         int    Impuesto2 = -8 + 12 / 2;
5         int    Cociente = 16 / 3;
6         int    Resto = 16 % 3;
7         int    Nueve = 9;
8         int    PostIncremento = Nueve++;
9         Nueve = 9;
10        int    PreIncremento = ++Nueve;
11        Nueve = 9;
12        int    PostDecremento = Nueve--;
13        Nueve = 9;
14        int    PreDecremento = --Nueve;
15    }
```

```
16     System.out.println(Impuesto);
17     System.out.println(Impuesto2);
18     System.out.println(Cociente);
19     System.out.println(Resto);
20     System.out.println(PostIncremento);
21     System.out.println(PreIncremento);
22     System.out.println(PostDecremento);
23     System.out.println(PreDecremento);
24 }
25 }
```



1.4.3 Operadores lógicos

Los operadores lógicos nos permiten combinar operandos booleanos, obteniendo, así mismo, resultados booleanos. Los operandos lógicos y sus significados son:

Operador	Sintaxis	Ejemplo	Funcionamiento		
Negación	!	Calor = !Frio	!		
			false		true
			true		false
Y	&&	Oportunidad = Bueno && Bonito && Barato	&&	false	true
			false	0	0
			true	0	1
O		Mojado = Llueve    Riego		false	true
			false	0	1
			true	1	1

En la clase *OperadoresLogicos* se programan las operaciones incluidas en la columna “Ejemplo” de la tabla anterior. El resultado en todos los casos es: *true*.

```
1 public class OperadoresLogicos {
2     public static void main (String[] args) {
3         boolean Calor, Frio = false, Oportunidad, Bueno = true,
4             Bonito = true, Barato = true, Llueve = true,
```

```
5         Riego = false;
6
7     Oportunidad = Bueno && Bonito && Barato;
8
9     System.out.println(!Frio);
10    System.out.println(Oportunidad);
11    System.out.println(Llueve || Riego);
12 }
13 }
```

1.4.4 Operadores de comparación

Los operadores de comparación se utilizan frecuentemente para dirigir la evolución del flujo de control en los programas. A continuación se proporciona una tabla en la que se definen estos operadores:

Operador	Sintaxis	Ejemplo
Menor	<	(EdadJuan < 18)
Menor o igual	<=	(EdadJuan <= EdadPedro)
Mayor	>	(Hipotenusa > 8.0f * 6.2f + 5.7f)
Mayor o igual	>=	(Cateto1 >= Cateto2)
Igual	==	(Contador == 8)
Distinto	!=	(Contador != 8)
Instancia de	instanceof	(Valor instanceof float)

El operador *instanceof* nos indica si una variable pertenece un tipo dado (siendo el tipo una clase o array). Las clases y arrays se explicarán más adelante. En el ejemplo siguiente se hace uso de las comparaciones mostradas en la tabla. Todos los resultados que se muestran en las líneas 6 a 11 son *true*, salvo la 10 que es *false*.

```
1 public class OperadoresComparacion {
2     public static void main (String[] args) {
3         int EdadJuan = 6, EdadPedro = 21, Contador = 14;
4         float Hipotenusa = 105.6f, Cateto1 = 13.2f,
5         Cateto2 = 5.7f;
6
7         System.out.println(EdadJuan < 18);
8         System.out.println(EdadJuan <= EdadPedro);
9         System.out.println(Hipotenusa > 8.0f * 6.2f + 5.7f);
10        System.out.println(Cateto1 >= Cateto2);
11        System.out.println(Contador == 8);
12        System.out.println(Contador != 8);
13
14    }
15 }
```



# ESTRUCTURAS DE CONTROL

---

## 2.1 EL BUCLE FOR

Hasta ahora, todos los programas que hemos visto son estrictamente secuenciales, es decir, a la ejecución de una instrucción inexorablemente le sigue la ejecución de la siguiente instrucción en secuencia. Esta situación es excesivamente restrictiva, puesto que lo habitual es que surja la necesidad de variar el flujo de control del programa para tomar decisiones y/o repetir cálculos.

En la mayor parte de los lenguajes de programación existe una serie de instrucciones que permiten variar la secuencialidad del flujo de control. En Java las podemos dividir en el grupo de instrucciones condicionales, el grupo de instrucciones repetitivas y las llamadas a métodos. El bucle *for* que se explica en este apartado pertenece al grupo de instrucciones repetitivas.

### 2.1.1 Sintaxis

Cuando deseamos ejecutar un grupo de instrucciones un número determinado de veces, la instrucción *for* es la que mejor se adapta a esta tarea. La sintaxis de esta instrucción es:

```
for (inicialización; condición de continuidad; expresión de variación) {  
    Instrucciones a ejecutar de forma repetitiva  
}
```

La semántica (significado) de la instrucción es la siguiente: se inicializa una variable (inicialización), se evalúa la condición de continuidad y, si se cumple, se ejecutan las instrucciones situadas entre las llaves, finalmente se ejecuta la expresión

de variación y se repite el ciclo hasta que la condición de continuidad se evalúa como *false*. Este proceso se puede entender mucho mejor con una serie de ejemplos:

2.1.2  Ejemplos de aprendizaje

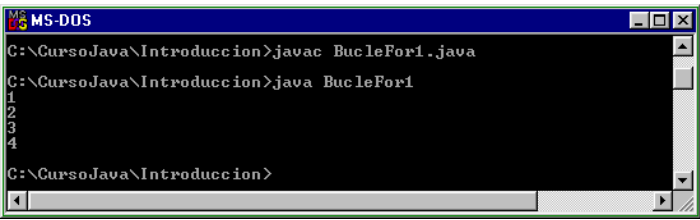
```
1 public class BucleFor1 {
2     public static void main (String[] args) {
3         int i;
4         for (i=1;i<=4;i=i+1) {
5             System.out.println(i);
6         }
7     }
8 }
```

En el ejemplo anterior (*BucleFor1*) se declara una variable de tipo *int* en la línea 3, que se utiliza en la línea 4 (*for*). El bucle *for* contiene sus tres secciones obligatorias:

- Inicialización: *i*=1
- Condición de continuidad: *i*<=4
- Expresión de incremento: *i*=*i*+1

De esta manera el bucle se ejecuta de la siguiente forma:

Primera iteración	<i>i</i> almacena el valor 1	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 1
Segunda iteración	<i>i</i> almacena el valor 2	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 2
Tercera iteración	<i>i</i> almacena el valor 3	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 3
Cuarta iteración	<i>i</i> almacena el valor 4	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 4
Finalización	<i>i</i> almacena el valor 5	<i>i</i> <=4 se evalúa como <i>false</i>	-----



En el ejemplo anterior podemos cambiar, si lo deseamos, las líneas de código 3 y 4 de la siguiente manera: `for (int i=1; i<=4; i++)`; en este caso declaramos la variable *i* dentro del bucle y utilizamos el operador de postincremento. También podemos prescindir de las llaves, puesto que el bloque de instrucciones que se ejecutan repetitivamente, en este caso, se compone de una única instrucción.



La expresión de variación puede definirse de cualquier manera que nos sea útil. Por ejemplo, si deseamos imprimir los 10 primeros números impares negativos, podemos programar el siguiente bucle:

```
1 public class BucleFor2 {
2     public static void main (String[] args) {
3         for (int i=-1;i>-20;i=i-2)
4             System.out.println(i);
5     }
6 }
```

Primera iteración	<i>i</i> almacena el valor -1	<i>i</i> >-20 se evalúa como <i>true</i>	Se imprime: -1
Segunda iteración	<i>i</i> almacena el valor -3	<i>i</i> >-20 se evalúa como <i>true</i>	Se imprime: -3
.....	.....	.....	.....
Décima iteración	<i>i</i> almacena el valor -19	<i>i</i> >-20 se evalúa como <i>true</i>	Se imprime: -19
Finalización	<i>i</i> almacena el valor -21	<i>i</i> >-20 se evalúa como <i>false</i>	-----

Es habitual, en programación, utilizar en la medida de lo posible bucles sencillos, aunque esto conlleve que las instrucciones interiores al bucle contengan expresiones un poco más complejas. Los bucles sencillos (secuenciales) facilitan la depuración del código cuando existen errores en los programas. El ejemplo *BucleFor2* puede ser codificado de la siguiente manera:

```
1 public class BucleFor3 {
2     public static void main (String[] args) {
3         for (int i=1;i<=10;i++)
4             System.out.println(1-i*2);
5     }
6 }
```

Otra posible solución es:

```
1 public class BucleFor4 {
2     public static void main (String[] args) {
3         int ImparNegativo = -1;
4         for (int i=0;i<=9;i++){
5             System.out.println(ImparNegativo);
6             ImparNegativo = ImparNegativo - 2;
7         }
8     }
9 }
```

En este caso no podemos prescindir de las llaves de comienzo y final (llaves delimitadoras) de bloque, puesto que si lo hiciéramos sólo se ejecutaría repetitivamente la instrucción 5. La instrucción 6 se ejecutaría una sola vez, al acabar el bucle.

### 2.1.3 Situaciones erróneas

En este apartado vamos a ilustrar una serie de actuaciones erróneas que son muy habituales entre las personas que empiezan a programar. Resulta especialmente conveniente prestar atención en este apartado para evitar, en la medida de lo posible, codificar los programas con errores.

La primera situación que vamos a analizar son los bucles infinitos (producto siempre de un error en la programación). En el siguiente ejemplo (*BucleFor5*) se produce esta situación, debido a que la condición de continuidad del bucle nunca se evalúa como *false*. Analizando el programa, llegamos fácilmente a la conclusión de que la variable *i* albergará únicamente valores pares positivos, por lo que la condición de continuidad *i!=21* nunca se evaluará como *false* y por lo tanto nunca se saldrá del bucle.

```
1 public class BucleFor5 {
2     public static void main (String[] args) {
3         for (int i=0;i!=21;i=i+2)
4             System.out.println(i);
5     }
6 }
```

Para cortar la ejecución de un programa basta con pulsar la combinación de teclas *Ctrl.+C* con la consola de MS-DOS activa (donde se ejecuta el programa). La tecla *Ctrl.* se debe pulsar primero, y manteniendo esta tecla pulsada presionar la tecla *C*.

El siguiente programa es correcto. Imprime los primeros 20 números enteros positivos:

```
1 public class BucleFor6 {
2     public static void main (String[] args) {
3         for (int i=1;i!=21;i=i+1)
4             System.out.println(i);
5     }
6 }
```

En *BucleFor6* la condición de continuidad es correcta; la iteración se produce para  $i=1, 2, 3, \dots, 20$ . Con  $i$  valiendo 21, la condición se evalúa como *false* ( $21!=21$  es *false*). Si pusiéramos como condición de finalización  $i < 21$ , la semántica del programa no variaría, sin embargo,  $i>20$ , por ejemplo, nos sacaría del bucle de forma inmediata, puesto que en la primera iteración la condición de continuidad se evaluaría como *false*. Aquí tenemos otro ejemplo típico de programación errónea: un bucle *for* cuya condición de evaluación, mal programada, nos saca del bucle en la primera iteración.

```
for (int i=1;i>20;i=i+1)
```

El siguiente programa (*BucleFor7*) contiene un error muy común y muy difícil de detectar. El problema reside en haber mezclado la expresión de postincremento ( $i++$ ) con la de asignación ( $i=i+1$ ), obteniendo la expresión, errónea,  $i=i++$ .

```
1 public class BucleFor7 {
2     public static void main (String[] args) {
3         for (int i=1;i!=21;i=i++)
4             System.out.println(i);
5     }
6 }
```

El programa codificado en la clase *BucleFor8* realiza la suma de los 1000 primeros números naturales ( $1+2+3+4+ \dots +1000$ ), imprimiendo por cada suma el resultado parcial obtenido. Este programa está correctamente codificado.

```
1 public class BucleFor8 {
2     public static void main (String[] args) {
3         int Suma = 0;
4         for (int i=1;i<=1000;i++) {
5             Suma = Suma + i;
6             System.out.println(Suma);
7         }
8     }
9 }
```

Las instrucciones que se repiten (1000 veces) son las que se encuentran entre los delimitadores `{ }` asociados a la instrucción repetitiva *for*; por ello tanto la instrucción 5 como la 6 se ejecutan 1000 veces. En cada vuelta del bucle se añade el valor de  $i$  (1,2,3,4, ..., 1000) al resultado de la suma anterior (guardado en la variable *Suma*). La evolución de las variables es la siguiente:

Primera iteración	$i$ almacena el valor 1	<i>Suma</i> almacena el valor 1	$i<=1000$ se evalúa como <i>true</i>	Se imprime: 1
-------------------	-------------------------	---------------------------------	--------------------------------------	---------------

Segunda iteración	<i>i</i> almacena el valor 2	<i>Suma</i> almacena el valor 3	<i>i</i> ≤1000 se evalúa como <i>true</i>	Se imprime: 3
Tercera iteración	<i>i</i> almacena el valor 3	<i>Suma</i> almacena el valor 6	<i>i</i> ≤1000 se evalúa como <i>true</i>	Se imprime: 6
.....	.....		.....	.....
Iteración 1000	<i>i</i> almacena el valor 1000	<i>Suma</i> almacena el valor 500500	<i>i</i> ≤1000 se evalúa como <i>true</i>	Se imprime: 500500
Finalización	<i>i</i> almacena el valor 1001	----- -----	<i>i</i> ≤1000 se evalúa como <i>false</i>	-----

Si en el programa anterior olvidáramos las llaves delimitadoras del ámbito (alcance) del bucle *for*, sólo se ejecutaría dentro del bucle la línea 5. En este caso obtenemos el mismo resultado que en el ejemplo anterior, pero sin la impresión de los resultados parciales de las sumas. La instrucción 6 se ejecuta al terminar el bucle y nos imprime únicamente el resultado final.

```
1 public class BucleFor9 {
2     public static void main (String[] args) {
3         int Suma = 0;
4         for (int i=1;i<=1000;i++)
5             Suma = Suma + i;
6             System.out.println(Suma);
7     }
8 }
```

Omitir las llaves de un bucle es un error habitual cuando se comienza a programar en Java, y normalmente las consecuencias son mucho peores que las ocurridas en el ejemplo anterior.

2.1.4 Ejemplos de resolución de problemas

Factorial de un número

El primer ejemplo que se propone en esta sección es hallar el factorial de un número. El valor factorial se consigue de la siguiente manera:

Factorial de *k* (*k*!) = *k* \* (*k*-1) \* (*k*-2) \* ... \* 2 \* 1  
Ejemplo: 4! = 4 \* 3 \* 2 \* 1 = 24

Solución:

```
1 public class Factorial {
2     public static void main (String[] args) {
3         int Numero = 6;
4         int Factorial = 1;
5         for (int i=2;i<=Numero;i++)
6             Factorial = Factorial * i;
7         System.out.println(Factorial);
8     }
9 }
```

El valor del que queremos obtener el factorial lo almacenamos en la variable *Numero* (línea 3), después inicializamos la variable *Factorial* a 1, que nos servirá de acumulador. El bucle for comienza en 2, aumenta secuencialmente y termina en el valor almacenado en *Numero*, de esta manera conseguimos pasar por la secuencia: 2, 3, 4, 5, 6, que son los valores que debemos multiplicar (el 1, realmente no es necesario).

En la línea 6 acumulamos los valores parciales de la multiplicación de la secuencia (2,3,4,5,6). La línea 7, ya fuera del bucle, nos imprime el resultado (720).

Para hallar el factorial de cualquier otro número, basta con variar el valor con el que inicializamos la variable *Numero*, en la línea 6. Obsérvese que el factorial de 1 (que es cero) no se saca con este método, y sobre todo, que si el valor de *Numero* es muy grande, podemos desbordar el rango del tipo *int*, por lo que sería conveniente emplear el tipo *long*.

## Problema de logística

Supongamos que una importante empresa de electrodomésticos nos contrata para resolver problemas de logística. El primer caso práctico que nos plantean es el siguiente:

En las grandes ciudades el precio del suelo es muy caro, por lo que comprar o alquilar grandes superficies de almacenamiento de electrodomésticos resulta prohibitivo en el centro de la ciudad. La solución es alejarse del núcleo urbano, sin embargo, cuanto más nos alejamos, más nos cuesta el precio de distribución que cada día hay que abonar a los transportistas que nos trasladan los electrodomésticos de la periferia al centro (donde se realizan la mayoría de las compras).

La estrategia que adoptaremos es la siguiente:

- 1 Adquirir un almacén pequeño en el centro de la ciudad (para 200 electrodomésticos, por término medio).
- 2 Adquirir almacenes en anillos concéntricos de 5 kilómetros a partir del centro de la ciudad, cada almacén podrá contener un stock del doble de electrodomésticos que el almacén anterior (es decir, de 400 electrodomésticos a 5 Km. del núcleo urbano, de 800 electrodomésticos a 10 kilómetros, etc.).

Se pide: indicar a cuantos kilómetros se encontrará el último almacén en una ciudad que requiere una capacidad total de 100000 electrodomésticos en stock. Los 100000 electrodomésticos estarán repartidos entre todos los almacenes adquiridos.

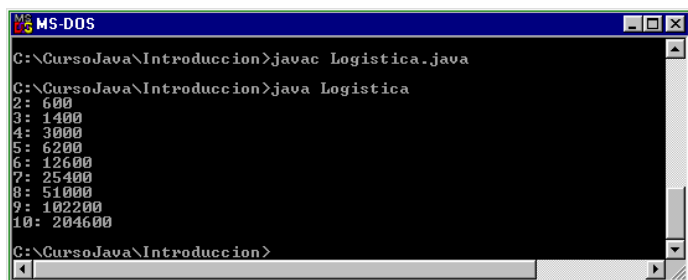
Solución:

```
1 public class Logistica {
2     public static void main (String[] args) {
3         int CapacidadAlmacen = 200;
4         int CapacidadTotal = CapacidadAlmacen;
5         for (int i=2;i<=10;i++) {
6             CapacidadAlmacen = CapacidadAlmacen * 2;
7             CapacidadTotal = CapacidadTotal + CapacidadAlmacen;
8             System.out.println(i+": "+CapacidadTotal);
9         }
10    }
11 }
```

En la línea 3 establecemos la capacidad del primer almacén (*CapacidadAlmacen*) a 200, en la línea 4 establecemos la capacidad total (*CapacidadTotal*) que por ahora tenemos como la capacidad del primer almacén (el del centro urbano). La línea 5 codifica un bucle que itera 10 veces: esperamos que con 10 almacenes consigamos la capacidad total de 100000 electrodomésticos, si no fuera así deberíamos aumentar el número de iteraciones.

La línea 6 actualiza la capacidad del siguiente almacén como el doble de la capacidad del almacén actual. La línea 7 acumula en la variable *CapacidadTotal* el número de electrodomésticos que sumaban bs almacenes anteriores con los que puede albergar el almacén actual. La línea 8 nos imprime los resultados parciales, obsérvese que podemos imprimir varias variables en la misma línea.

El resultado de la ejecución del programa se muestra en la siguiente ventana:



```
MS-DOS
C:\CursoJava\Introduccion>javac Logistica.java
C:\CursoJava\Introduccion>java Logistica
2: 600
3: 1400
4: 3000
5: 6200
6: 12600
7: 25400
8: 51000
9: 102200
10: 204600
C:\CursoJava\Introduccion>
```

Se puede observar como con 9 almacenes (incluido el del centro urbano) se alcanza la capacidad acumulada de 100000 electrodomésticos (concretamente 102200); por lo tanto hemos tenido que comprar 8 almacenes fuera del centro urbano, situándose el último a  $8 \times 5 = 40$  kilómetros de la ciudad.

Para realizar este ejercicio hubiera sido mucho más elegante emplear una estructura de control de flujo en bucle que nos permitiera iterar mientras (o hasta) que se cumpla una condición determinada: (que *CapacidadTotal*  $\geq 100000$ ). El bucle *for* no nos resuelve esta situación adecuadamente, puesto que está diseñado para indicar a priori el número de iteraciones, aunque tampoco resulta imposible utilizarlo de otra manera:

```
1 public class Logistica2 {
2     public static void main (String[] args) {
3         int CapacidadAlmacen = 200;
4         int CapacidadTotal = CapacidadAlmacen;
5         int i;
6         for (i=2;CapacidadTotal<100000;i++) {
7             CapacidadAlmacen = CapacidadAlmacen * 2;
8             CapacidadTotal = CapacidadTotal + CapacidadAlmacen;
9         }
10        System.out.println(i+": "+CapacidadTotal);
11    }
12 }
```

En cualquier caso, para resolver este ejercicio, resulta más apropiado utilizar las estructuras de control que se explican en el siguiente apartado.

## 2.2 EL BUCLE WHILE

El bucle *while* nos permite repetir la ejecución de una serie de instrucciones mientras que se cumpla una condición de continuidad. Su uso resulta recomendable cuando no conocemos a priori el número de iteraciones que debemos realizar.

### 2.2.1 Sintaxis

El bucle *while* tiene dos posibles sintaxis:

```
while (condición de continuidad) {  
    Instrucciones a ejecutar de forma repetitiva  
}
```

```
do {  
    Instrucciones a ejecutar de forma repetitiva  
} while (condición de continuidad);
```

En ambos casos se itera mientras que la “condición de continuidad” se cumpla, abandonándose el bucle cuando la condición se evalúa como *false*. En el primer caso puede ocurrir que las instrucciones interiores del bucle nunca se ejecuten (si la primera vez que se evalúa la condición resulta *false*); en el segundo caso las instrucciones interiores al bucle se ejecutan al menos una vez.

### 2.2.2 Ejemplos de aprendizaje

En el siguiente ejemplo se muestra una implementación muy sencilla del bucle *while* en la que se pretende imprimir los números 1, 2, 3 y 4. Puesto que conocemos a priori el número de iteraciones sería más adecuado utilizar un bucle *for*, pero se ha escogido este ejemplo sencillo para mostrar una primera implementación del bucle *while*.

```
1 public class BucleWhile1 {  
2     public static void main (String[] args) {  
3         int i=1;  
4         while (i<=4) {  
5             System.out.println(i);  
6             i++;  
7         }  
8     }  
9 }  
10 }
```

En *BucleWhile1* se declara una variable de tipo *int* en la línea 3 y se inicializa a 1; esta variable actuará como contador de iteraciones en el bucle. En la línea 4 se establece la condición de continuidad del bucle (se itera mientras que  $i \leq 4$ ). La línea 5 se encarga de imprimir el valor del índice y la línea 6 de



incrementarlo. Nótese como se están codificando las distintas expresiones y condiciones del bucle *for*: *for (i=1; i<=4;i++)*, en las líneas 3, 4 y 6.

Un error muy frecuente cuando se codifica un bucle *while* es olvidar incrementar el contador (línea 6), generando un bucle infinito. En nuestro ejemplo también crearíamos un bucle infinito se olvidáramos las llaves delimitadoras del ámbito del bucle.

Detalle de la ejecución de *BucleWhile1*:

Antes del <i>while</i>	<i>i</i> almacena el valor 1		
Primera iteración	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 1	<i>i</i> almacena el valor 2
Segunda iteración	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 2	<i>i</i> almacena el valor 3
Tercera iteración	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 3	<i>i</i> almacena el valor 4
Cuarta iteración	<i>i</i> <=4 se evalúa como <i>true</i>	Se imprime: 4	<i>i</i> almacena el valor 5
Finalización	<i>i</i> <=4 se evalúa como <i>false</i>	-----	-----

A continuación se muestra la forma alternativa de utilizar un bucle *while*: *do { } while(condición)*; la semántica en este ejemplo es la misma que en el anterior (se imprimen los valores 1 a 4). El detalle de la ejecución no varía respecto a la mostrada en *BucleWhile1*, salvo que la condición de continuidad se evalúa al final en lugar de al comienzo del bucle.

```
1 public class BucleWhile2 {
2     public static void main (String[] args) {
3         int i=1;
4         do {
5             System.out.println(i);
6             i++;
7         } while (i<=4);
8     }
9 }
```

Ahora vamos a resolver la siguiente cuestión: ¿Cuántos números naturales (1, 2, 3, 4...) debemos sumar en secuencia para obtener al menos un valor de 100000?, es decir: ¿hasta qué valor llegará el sumatorio 1+2+3+4+5+..... para que la suma alcance al valor 100000?

Este problema lo podemos resolver de forma muy simple haciendo uso del bucle *while*:

```
1 public class BucleWhile3 {
2     public static void main (String[] args) {
3         int Suma=0;
4         int i=0;
5         do {
```

```
6         i++;
7         Suma = Suma + i;
8     } while (Suma<100000);
9     System.out.println(i);
10 }
11 }
```

En la línea 3 se declara e inicializa el acumulador *Suma* al valor 0 y en la línea 4 el contador *i* al valor 0. En la línea 6 se incrementa *i* y en la línea 7 se acumula el valor del contador sobre la variable *Suma*. Las iteraciones continúan mientras *Suma*<100000 (línea 8).

Detalle de la ejecución de *BucleWhile3*:

Antes del <i>while</i>	<i>i</i> almacena el valor 0	<i>Suma</i> almacena el valor 0	
Primera iteración	<i>i</i> almacena el valor 1	<i>Suma</i> almacena el valor 1	<i>Suma</i> <100000 se evalúa como <i>true</i>
Segunda iteración	<i>i</i> almacena el valor 2	<i>Suma</i> almacena el valor 3	<i>Suma</i> <100000 se evalúa como <i>true</i>
.....	.....	.....	.....
Iteración 447	<i>i</i> almacena el valor 447	<i>Suma</i> almacena el valor 100128	<i>Suma</i> <100000 se evalúa como <i>false</i>

2.2.3 Ejemplos de resolución de problemas

Determinación de si un número es primo

Utilizando un algoritmo muy sencillo aunque poco eficaz, podemos saber si un número es primo de la siguiente manera: dividimos el número entre todos los anteriores (salvo el 1) y si no es divisible entre ninguno, entonces es primo. Una posible implementación de este algoritmo es:

```
1 public class Primo {
2     public static void main (String[] args) {
3         int PosiblePrimo = 17;
4         int Divisor = 2;
5         boolean Primo = true;
6
7         do {
8             Primo = (PosiblePrimo % Divisor)!=0;
9             Divisor++;
```

```
10      } while(Divisor<PosiblePrimo && Primo);
11      System.out.println(Primo);
12  }
13 }
```

En la línea 3 declaramos y definimos el valor del número del que deseamos saber si es primo. En la línea 4 inicializamos una variable *Divisor* a 2, éste será el contador por el que se irá dividiendo sucesivamente nuestro *PosiblePrimo*. En la línea 8 determinamos si *PosiblePrimo* es divisible entre *Divisor*, para ello obtenemos su módulo (operación %) y lo comparamos con 0, si no es divisible, por ahora el número puede ser primo. Vamos incrementando *Divisor* (en la línea 9) y continuamos el bucle (línea 10) mientras que *Divisor* sea menor que *PosiblePrimo* y *Primo* nos indique que por ahora el número puede ser primo.

Cuando se sale del bucle (línea 11) o bien *Divisor* ha alcanzado a *PosiblePrimo*, en cuyo caso el número es primo (*Primo* es *true*), o bien la variable *Primo* ha tomado el valor *false*, en cuyo caso *PosiblePrimo* no es un número primo.

Detalle de la ejecución de *BucleWhile3*:

Antes del <i>while</i>	<i>Primo</i> almacena el valor <i>true</i>	<i>Divisor</i> almacena el valor 2	
Primera iteración	<i>Primo</i> almacena el valor <i>true</i>	<i>Divisor</i> almacena el valor 3	La condición se evalúa como <i>true</i>
Segunda iteración	<i>Primo</i> almacena el valor <i>true</i>	<i>Divisor</i> almacena el valor 4	La condición se evalúa como <i>true</i>
.....	.....	.....	.....
Iteración 15	<i>Primo</i> almacena el valor <i>true</i>	<i>Divisor</i> almacena el valor 17	La condición se evalúa como <i>false</i>

Problema de logística

El problema que aquí se plantea ha sido resuelto en la lección anterior haciendo uso del bucle *for*, la solución utilizando el bucle *while* es muy parecida y mucho más apropiada: supongamos que una importante empresa de electrodomésticos nos contrata para resolver problemas de logística. El primer caso práctico que nos plantean es el siguiente:

En las grandes ciudades el precio del suelo es muy caro, por lo que comprar o alquilar grandes superficies de almacenamiento de electrodomésticos resulta prohibitivo en el centro de la ciudad. La solución es alejarse del núcleo urbano, sin

embargo, cuanto más nos alejamos, más nos cuesta el precio de distribución que cada día hay que abonar a los transportistas que nos trasladan los electrodomésticos de la periferia al centro (donde se realizan la mayoría de las compras).

La estrategia que adoptaremos es la siguiente:

- 1 Adquirir un almacén pequeño en el centro de la ciudad (para 200 electrodomésticos, por término medio).
- 2 Adquirir almacenes en anillos concéntricos de 5 kilómetros a partir del centro de la ciudad, cada almacén podrá contener un stock del doble de electrodomésticos que el almacén anterior (es decir, de 400 electrodomésticos a 5 Km. del núcleo urbano, de 800 electrodomésticos a 10 kilómetros, etc.).

Se pide: indicar a cuantos kilómetros se encontrará el último almacén en una ciudad que requiere una capacidad total de 100000 electrodomésticos en stock. Los 100000 electrodomésticos estarán repartidos entre todos los almacenes adquiridos.

Solución:

```
1 public class LogisticaWhile {
2     public static void main (String[] args) {
3         int CapacidadAlmacen = 200, Km = 0;
4         int CapacidadTotal = CapacidadAlmacen;
5         do {
6             CapacidadAlmacen = CapacidadAlmacen * 2;
7             CapacidadTotal = CapacidadTotal + CapacidadAlmacen;
8             Km = Km + 5;
9         } while(CapacidadTotal < 100000);
10        System.out.println(Km + ": " + CapacidadTotal);
11    }
12 }
```

En la línea 3 establecemos la capacidad del primer almacén (*CapacidadAlmacen*) a 200 y los kilómetros de distancia (*Km*) a 0, en la línea 4 establecemos la capacidad total (*CapacidadTotal*) que por ahora tenemos como la capacidad del primer almacén (el del centro urbano). Las líneas 5 y 9 codifican un bucle que itera mientras la capacidad acumulada (*CapacidadTotal*) sea menor que 100000.

La línea 6 actualiza la capacidad del siguiente almacén como el doble de la capacidad del almacén actual. La línea 7 acumula en la variable *CapacidadTotal* el número de electrodomésticos que sumaban los almacenes anteriores con los que puede albergar el almacén actual. La línea 8 actualiza los kilómetros a los que se encuentra el primer almacén.

El resultado de la ejecución del programa nos muestra 40 kilómetros y una capacidad acumulada de 102200 electrodomésticos.

## 2.3 LA INSTRUCCIÓN CONDICIONAL *IF*

Para poder programar aplicaciones no nos basta con ejecutar instrucciones secuencialmente, ni siquiera aunque tengamos la posibilidad de definir bucles; también resulta esencial poder tomar decisiones en base a condiciones. Las instrucciones condicionales nos permiten ejecutar distintas instrucciones en base a la evaluación de condiciones.

### 2.3.1 *Sintaxis*

La instrucción *if* puede emplearse de diversas maneras:

```
if (condición)  
    Instrucción
```

```
if (condición) {  
    Instrucciones  
}
```

```
if (condición)  
    Instrucción de la rama “then”  
else  
    Instrucción de la rama “else”
```

```
if (condición) {  
    Instrucciones de la rama “then”  
} else {  
    Instrucciones de la rama “else”  
}
```

En el primer caso, la instrucción se ejecuta sólo si la condición se evalúa como *true*. En el segundo caso, el conjunto de instrucciones sólo se ejecuta si la condición se evalúa como *true*. Como el lector habrá observado, el primer caso no es más que una situación particular del segundo, en el que, al existir una sola instrucción se pueden omitir las llaves (tal y como hacíamos con los bucles).

En las dos últimas sintaxis de la instrucción *if*, se introduce una “rama” *else*, cuyo significado es: si la condición se evalúa como *true* se ejecuta el grupo de instrucciones de la primera rama (llamémosla “then”), en caso contrario (la condición se evalúa como *false*) se ejecuta el grupo de instrucciones de la segunda rama (la rama “else”).

Obviamente se pueden programar situaciones en las que sólo hay una instrucción en la rama “then” y varias en la rama “else” o viceversa. En general podemos tomar como sintaxis la del último caso de los 4 presentados, sabiendo que la rama “else” es opcional y que si sólo existe una instrucción en alguna rama del *if*, podemos prescindir del uso de las llaves en esa rama.

### 2.3.2 Ejemplos de aprendizaje

El primer ejemplo presenta la forma más simple de instrucción condicional. Establecemos una condición sencilla y una instrucción que se ejecuta si la condición se evalúa como cierta.

```
1 public class If1 {
2     public static void main (String[] args) {
3         int EdadJuan = 20, EdadAna =25;
4
5         if (EdadJuan<EdadAna)
6             System.out.println ("Juan es mas joven que Ana");
7     }
8 }
```

Si empleamos las dos ramas del *if*:

```
1 public class If2 {
2     public static void main (String[] args) {
3         int EdadJuan = 20, EdadAna =25;
4
5         if (EdadJuan<EdadAna)
6             System.out.println ("Juan es mas joven que Ana");
7         else
8             System.out.println ("Juan no es mas joven que Ana");
9     }
10 }
```

Cuando necesitamos más de una instrucción en alguna rama, no debemos olvidar las llaves que delimitan estas instrucciones:

```
1 public class If3 {
2     public static void main (String[] args) {
3         float Presion = 2.3f;
4
5         if (Presion > 2f) {
6             System.out.println ("Abrir valvula de seguridad");
7             System.out.println ("Reducir la temperatura");
8         } else
9             System.out.println ("Todo en orden");
10    }
11 }
```

Las condiciones pueden programarse todo lo complejas que sea necesario:

```
1 public class If4 {
2     public static void main (String[] args) {
3         float Presion = 2.3f, Temperatura = 90f;
4
5         if (Presion > 2f && Temperatura > 200f) {
6             System.out.println ("Abrir valvula de seguridad");
7             System.out.println ("Reducir la temperatura");
8             System.out.println ("Llamar a los bomberos");
9         } else
10             System.out.println ("Todo en orden");
11    }
12 }
```

### 2.3.3 *if anidados*

En muchas ocasiones resulta conveniente insertar un *if* dentro de otro *if*. Si, por ejemplo, quisiéramos saber si Juan es mayor, menor o de la misma edad que Ana, normalmente recurriríamos a la utilización de *if* anidados:

```
1 public class If5 {
2     public static void main (String[] args) {
3         int EdadJuan = 30, EdadAna = 25;
4
5         if (EdadJuan < EdadAna)
6             System.out.println ("Juan es mas joven que Ana");
7         else
8             if (EdadJuan == EdadAna)
9                 System.out.println ("Juan tiene la edad de Ana");
10            else
```

```
11         System.out.println ("Juan es mayor que Ana");
12     }
13 }
```

En *If5*, en la línea 3, se declaran dos variables de tipo *int* y se inicializan a 30 y 25; representarán las edades de Juan y de Ana. En la línea 5 se pregunta si la edad de Juan es menor que la de Ana; si lo es, se imprime el mensaje adecuado, y si no, pueden ocurrir dos cosas: que tengan la misma edad o que Ana sea mayor; esto obliga a realizar en este punto una nueva pregunta: ¿Tienen la misma edad?, la línea 8 realiza esta pregunta dentro de la rama *else* del primer *if*.

El ejemplo anterior se podría haber resuelto sin hacer uso de sentencias condicionales anidadas:

```
1  public class If6 {
2      public static void main (String[] args) {
3          int EdadJuan = 30, EdadAna =25;
4
5          if (EdadJuan < EdadAna)
6              System.out.println ("Juan es mas joven que Ana");
7
8          if (EdadJuan > EdadAna)
9              System.out.println ("Juan es mayor que Ana");
10
11         if (EdadJuan == EdadAna)
12             System.out.println ("Juan tiene la edad de Ana");
13
14     }
15 }
```

La solución de la clase *If6* resuelve el mismo problema que el ejemplo de la clase *If5*; además la solución es más legible y fácil de depurar en caso de errores, sin embargo es importante darse cuenta de que en este último caso el ordenador siempre tiene que evaluar tres condiciones, mientras que en la solución aportada en *If5* basta con evaluar 1 ó 2 condiciones (dependiendo de las edades de Juan y Ana), por lo cual los *if* anidados proporcionan una solución más eficiente.

### 2.3.4 Situaciones erróneas

Utilizando instrucciones *if*, los principales errores de programación que se cometen son:



Situación errónea	Comentario
Omisión de los paréntesis en las condiciones	Los paréntesis son sintácticamente obligatorios en las condiciones de todas las instrucciones
Confundir el operador relacional == con el operador de asignación =	Este es un error muy típico y difícil de detectar, recordar que en las asignaciones se utiliza un símbolo de igual y en las comparaciones dos
Colocar mal los if anidados	Es conveniente repasar las llaves que se colocan en cada rama de un if anidado
No tener en cuenta el orden de precedencia de los operadores en las condiciones	Ante cualquier duda utilizar paréntesis

El segundo de los errores es tan común que merece la pena comentarlo en más detalle. En la línea 8 de la clase *If5* utilizamos el operador de comparación (==). Si en esta línea, por error, hubiéramos introducido el operador de asignación (=), obtendríamos un error de compilación (que nos facilitaría la depuración del código). En otros contextos no seríamos tan afortunados y se realizaría la asignación desvirtuando la comparación.

2.3.5 Ejemplo de resolución de problemas

Encontrar el menor de tres valores

Dados tres valores A, B y C, podemos determinar cual es el menor realizando las preguntas anidadas pertinentes. Basta con evaluar dos condiciones para obtener un resultado válido:

```
1 public class If7 {
2     public static void main (String[] args) {
3         int A = 10, B = 5, C = 20;
4
5         if (A < B)
6             if (A < C)
7                 System.out.println ("A es el menor");
8             else
9                 System.out.println ("B es el menor");
10        else
11            if (C < B)
12                System.out.println ("C es el menor");
13            else
14                System.out.println ("B es el menor");
15    }
16 }
```

Una solución más fácil de entender, pero mucho más costosa en ejecución (por el número de condiciones a evaluar) es:

```
1 public class If8 {
2     public static void main (String[] args) {
3         int A = 10, B = 5, C = 20;
4
5         if (A <= B && A <= C)
6             System.out.println ("A es el menor");
7
8         if (B <= A && B <= C)
9             System.out.println ("B es el menor");
10
11        if (C <= A && C <= B)
12            System.out.println ("C es el menor");
13
14    }
15 }
```

## 2.4 LA INSTRUCCIÓN CONDICIONAL SWITCH

Cuando en una condición existen diversas posibilidades, nos vemos obligados a programar usando *if* anidados, lo que complica la realización y depuración de código. Para facilitar la programación en estas situaciones, se proporciona la instrucción condicional *switch*, que permite definir un número ilimitado de ramas basadas en una misma condición.

### 2.4.1 Sintaxis

```
switch (expresión) {
    case valor1:
        Instrucciones;
        break;

    case valor1:
        Instrucciones;
        break;

    .....
}
```

```
default:
    Instrucciones;
break;
}
```

Cuando el flujo de control del programa llega a la instrucción *switch*, lo primero que se hace es evaluar la expresión, después se va comparando el valor de cada cláusula *case* con el resultado de la evaluación de la expresión. Cuando en una cláusula *case* coinciden los valores, se ejecutan las instrucciones asociadas hasta alcanzar la sentencia *break*. Si no se incluye el *break* en un *case*, se ejecutan todas las instrucciones siguientes (correspondientes a los siguientes grupos *case*) hasta que se encuentra un *break* o se termina la instrucción *switch*.

La cláusula *default* es muy útil, nos sirve para indicar que se ejecuten sus instrucciones asociadas en el caso de que no se haya ejecutado previamente ningún otro grupo de instrucciones.

La sentencia *break* asociada al último *case* (o *default*) no es necesaria, puesto que el final de la ejecución de instrucciones del *switch* viene marcado tanto por las instrucciones *break* como por el fin físico de la instrucción *switch*.

Es importante tener en cuenta que la expresión asociada a la instrucción *switch* sólo debe generar valores de tipo: *char*, *byte*, *short* o *int*.

## 2.4.2 Ejemplos de aprendizaje

Nuestro primer ejemplo (*Switch1*) nos muestra una instrucción *switch* (línea 5) con una expresión de tipo *int*. Según el valor de la expresión sea 1, 2 ó 3, se imprimirán diferentes mensajes. Cuando el valor de la expresión es diferente a 1, 2 y 3 (línea 18) se imprime un mensaje genérico (línea 19). En el ejemplo, si la instrucción 12 no existiera, se imprimirían dos mensajes: “Medalla de plata” y “Medalla de bronce”.

```
1 public class Switch1 {
2     public static void main (String[] args) {
3         int Puesto = 2;
4
5         switch (Puesto) {
6             case 1:
7                 System.out.println("Medalla de oro");
8                 break;
9
```

```
10         case 2:
11             System.out.println("Medalla de plata");
12             break;
13
14         case 3:
15             System.out.println("Medalla de bronce");
16             break;
17
18         default:
19             System.out.println("Gracias por participar");
20             break;
21     }
22 }
23
24 }
25 }
```

Nuestro segundo ejemplo implementa un control de accionamiento de accesos, donde se puede ordenar la apertura, cierre o comprobación de circuitos de una puerta. En un programa más amplio se le pediría al usuario la introducción de un carácter que indique la acción requerida, digamos ‘a’: abrir, ‘c’: cerrar, ‘t’: test de circuitos; en nuestro caso suplimos la introducción del comando por la línea 3.

La clase *Switch2* es muy parecida a la clase *Switch1*, salvo en el tipo de datos que utilizamos (*char*).

```
1 public class Switch2 {
2     public static void main (String[] args) {
3         char Caracter = 't';
4
5         switch (Caracter) {
6             case 'a':
7                 System.out.println("Abrir puerta");
8                 break;
9
10            case 'c':
11                System.out.println("Cerrar puerta");
12                break;
13
14            case 't':
15                System.out.println("Comprobar circuitos");
16                break;
17
18            default:
19                System.out.println("Opcion no contemplada");
20                break;
21        }
```

```
22     }
23
24 }
25 }
```

Podemos ampliar el ejemplo anterior para que admita también los comandos en mayúsculas. Obsérvese la nueva sintaxis:

```
1  public class Switch3 {
2      public static void main (String[] args) {
3          char Caracter = 'C';
4
5          switch (Caracter) {
6              case 'a':
7              case 'A':
8                  System.out.println("Abrir puerta");
9                  break;
10
11             case 'c':
12             case 'C':
13                 System.out.println("Cerrar puerta");
14                 break;
15
16             case 't':
17             case 'T':
18                 System.out.println("Comprobar circuitos");
19                 break;
20
21             default:
22                 System.out.println("Opcion no contemplada");
23                 break;
24         }
25     }
26
27 }
28 }
```

La expresión del *switch* puede programarse todo lo compleja que sea necesario:

```
1  public class Switch4 {
2      public static void main (String[] args) {
3          int Valor = 341;
4
5          switch ((4*Valor+17)%3) {
6              case 0:
```

```
7         System.out.println("Primera opcion");
8         break;
9
10        case 1:
11            System.out.println("Segunda opcion");
12            break;
13
14        case 2:
15            System.out.println("Tercera opcion");
16            break;
17
18    }
19
20 }
21 }
```

2.4.3 switch anidados

Resulta muy habitual tener que realizar selecciones basadas en dos niveles; el siguiente ejemplo muestra una de estas situaciones: estamos realizando las páginas Web de un concesionario de vehículos de la marca SEAT y en un momento dado el usuario puede escoger entre las siguientes opciones:

	Amarillo	Blanco	Rojo
Ibiza	SI	SI	SI
Córdoba	NO	SI	NO
Toledo	NO	SI	SI

Para programar la elección de opción por parte del usuario, resulta adecuado emplear una instrucción *switch* anidada:

```
1 public class Switch5 {
2     public static void main (String[] args) {
3         char Modelo = 'c', Color = 'r';
4         boolean Disponible = false;
5
6         switch (Modelo) {
7             case 'i':
8                 switch (Color){
9                     case 'a':
10                         Disponible = true;
```

```
11         break;
12     case 'b':
13         Disponible = true;
14         break;
15     case 'r':
16         Disponible = true;
17         break;
18     default:
19         System.out.println("Opcion no contemplada");
20         break;
21 }
22 break;
23
24 case 'c':
25     switch (Color) {
26     case 'a':
27         Disponible = false;
28         break;
29     case 'b':
30         Disponible = true;
31         break;
32     case 'r':
33         Disponible = false;
34         break;
35     default:
36         System.out.println("Opcion no contemplada");
37         break;
38     }
39     break;
40
41 case 't':
42     switch (Color) {
43     case 'a':
44         Disponible = false;
45         break;
46     case 'b':
47         Disponible = true;
48         break;
49     case 'r':
50         Disponible = true;
51         break;
52     default:
53         System.out.println("Opcion no contemplada");
54         break;
55     }
56     break;
57
58 default:
59     System.out.println("Opcion no contemplada");
```

```
60         break;
61     }
62
63     if (Disponible)
64         System.out.println("Opcion disponible");
65     else
66         System.out.println("Opcion no disponible");
67
68 }
69 }
```

En el ejercicio anterior (*Switch5*) la primera instrucción *switch* consulta el modelo del vehículo ('i': Ibiza, 'c':Córdoba, 't': Toledo), los *switchs* anidados se encargan de tratar la opción de color seleccionada ('a': amarillo, 'b': blanco, 'r':rojo). Aunque este ejercicio muestra adecuadamente la manera de anidar instrucciones *switch*, posiblemente el lector preferirá la siguiente solución:

```
1 public class Switch6 {
2     public static void main (String[] args) {
3         char Modelo = 'c', Color = 'r';
4
5         if ( (Modelo=='c' && (Color=='a' || Color=='r')) ||
6             (Modelo=='t' && Color=='a')
7         )
8             System.out.println("Opcion no disponible");
9         else
10             System.out.println("Opcion disponible");
11     }
12 }
```

2.4.4 Situaciones erróneas

Utilizando instrucciones *switch*, los principales errores de programación que se cometen son:

Situación errónea	Comentario
Omisión de los paréntesis en las condiciones	Los paréntesis son sintácticamente obligatorios en las condiciones de todas las instrucciones
Utilización de tipos no permitidos en la expresión	Sólo se permiten los tipos <i>char</i> , <i>byte</i> , <i>short</i> e <i>int</i> .
Olvidar alguna sentencia <i>break</i>	Si olvidamos una sentencia <i>break</i> , la ejecución de las instrucciones pasa a la siguiente cláusula case (en caso de que la haya)



### 2.4.5 Ejemplo de resolución de problemas

#### Realizar un tratamiento cada día de la semana

La estructura de la instrucción *switch* es muy sencilla en nuestro ejemplo, basta con una cláusula *case* por cada día laborable y una cláusula *default* para el sábado y el domingo. También podríamos haber puesto dos cláusulas *case* para los días 6 y 7 y emplear la opción *default* para tratar el posible error en el que *DiaSemana* contenga un valor distinto del 1 al 7.

```
1 public class Switch7 {
2     public static void main (String[] args) {
3         byte DiaSemana = 4;
4
5         switch (DiaSemana) {
6             case 1:
7                 System.out.println("¡Que duro es el lunes!");
8                 break;
9
10            case 2:
11                System.out.println("Dia de jugar al squash");
12                break;
13
14            case 3:
15                System.out.println("Mitad de la semana");
16                break;
17
18            case 4:
19                System.out.println("Famoso jueves del soltero");
20                break;
21
22            case 5:
23                System.out.println("¡Viernes bendito!");
24                break;
25
26            default:
27                System.out.println("El fin de semana a la playa");
28                break;
29
30        }
31    }
32 }
```



```
4      byte Mes = 2;
5      boolean Bisisesto = false;
6
7      switch (Mes) {
8          case FEBRERO:
9              if (Bisisesto)
10                 System.out.println(29);
11             else
12                 System.out.println(28);
13             break;
14
15             case ABRIL:
16             case JUNIO:
17             case SEPTIEMBRE:
18             case NOVIEMBRE:
19                 System.out.println(30);
20                 break;
21
22             case ENERO:
23             case MARZO:
24             case MAYO:
25             case JULIO:
26             case AGOSTO:
27             case OCTUBRE:
28             case DICIEMBRE:
29                 System.out.println(31);
30                 break;
31
32             default:
33                 System.out.println("Numero de mes erroneo");
34                 break;
35         }
36     }
37 }
```

## 2.5 EJEMPLOS

En esta lección se realizan tres ejemplos en los que se combinan buena parte de los conceptos explicados en las lecciones anteriores. Los ejemplos seleccionados son:

1. Cálculo de la hipotenusa de un triángulo
2. Soluciones de una ecuación de segundo grado

### 3. Obtención del punto de corte de dos rectas en el espacio bidimensional

Para permitir que el usuario pueda seleccionar, en cada caso, los parámetros necesarios (longitudes de los catetos, coordenadas de los vectores en las rectas, etc.) se proporciona una clase de entrada de datos por teclado desarrollada por el autor: la clase *Teclado*, que consta de los métodos:

- `public static String Lee_String()`
- `public static long Lee_long()`
- `public static int Lee_int()`
- `public static int Lee_short()`
- `public static byte Lee_byte()`
- `public static float Lee_float()`
- `public static double Lee_double()`

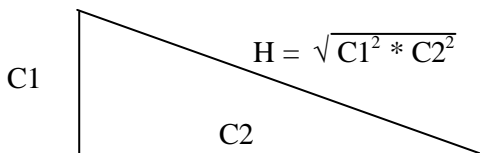
De esta manera, para leer un valor de tipo *byte*, podemos utilizar una instrucción similar a: *byte Edad = Teclado.Lee\_byte();*

En estos ejemplos también será necesario hacer uso de algún método de la biblioteca matemática que proporciona el entorno de desarrollo de Java. Esta biblioteca tiene como nombre *Math*, y entre sus métodos se encuentran:

- `public static double cos(double a) → coseno`
- `public static double sin(double a) → seno`
- `public static double sqrt(double a) → raíz cuadrada`
- `public static double pow(double a, double b) → a elevado a b`
- etc.

#### 2.5.1 Cálculo de la hipotenusa de un triángulo

En este ejercicio se pretende calcular la hipotenusa (*H*) de un triángulo equilátero en donde el usuario nos proporciona como datos (por teclado) el valor de cada uno de sus catetos (*C1* y *C2*). No es necesario tener en cuenta ninguna condición de datos de entrada erróneos.



## Plantilla para comenzar el código

```
1 public class Hipotenusa {
2     public static void main(String[] args) {
3     }
4 }
```

El código de la plantilla se corresponde a la estructura mínima necesaria para ejecutar un programa situado dentro del método *main*.

## Ejercicio resuelto

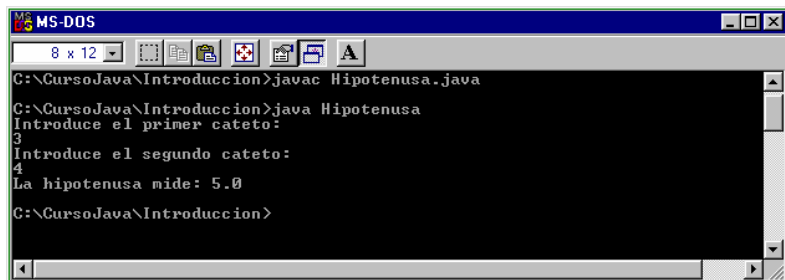
```
1 public class Hipotenusa {
2     public static void main(String[] args) {
3         System.out.println("Introduce el primer cateto: ");
4         double Cateto1 = Teclado.Lee_double();
5         System.out.println("Introduce el segundo cateto: ");
6         double Cateto2 = Teclado.Lee_double();
7
8         double Hipotenusa =
            Math.sqrt(Cateto1*Cateto1+Cateto2*Cateto2);
9
10        System.out.println("La hipotenusa mide: "
11                           + Hipotenusa);
12    }
13 }
```

## Comentarios

En las líneas 4 y 6 se introduce el valor de los dos catetos del triángulo, haciendo uso de nuestra clase de entrada de datos *Teclado*. Los valores que introduce el usuario se almacenan en las variables *Cateto1* y *Cateto2*, ambas de tipo *double* para posibilitar su uso directo en los métodos de la clase *Math*.

La línea de código número 8 introduce en la variable *Hipotenusa*, de tipo *double*, el resultado de la operación: raíz cuadrada de la suma de los catetos al cuadrado. La clase *Math* (*java.lang.Math*) permite realizar operaciones numéricas básicas, tales como funciones trigonométricas.

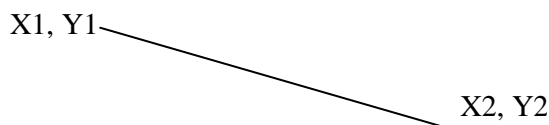
## Resultados



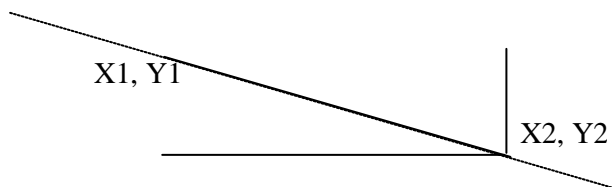
```
MS-DOS
8 x 12
C:\CursoJava\Introduccion>javac Hipotenusa.java
C:\CursoJava\Introduccion>java Hipotenusa
Introduce el primer cateto:
3
Introduce el segundo cateto:
4
La hipotenusa mide: 5.0
C:\CursoJava\Introduccion>
```

### 2.5.2 Obtención del punto de corte de dos rectas situadas en el espacio bidimensional

La determinación del punto de corte de dos rectas es matemáticamente muy fácil de enunciar: una recta definida por dos puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  se puede representar de la siguiente manera:



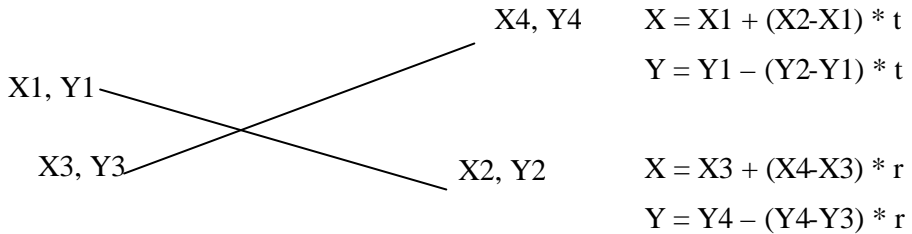
Cualquier punto de la recta puede obtenerse multiplicando un número real por los vectores  $(X_2 - X_1)$  e  $(Y_2 - Y_1)$ :



De esta manera, la ecuación de la recta, en notación paramétrica puede ser definida como:

- $X = X_1 + (X_2 - X_1) * t$
- $Y = Y_1 - (Y_2 - Y_1) * t$

Siendo  $t$  el valor real a partir del cual podemos obtener cualquier punto de la recta. Si dibujamos dos rectas no paralelas:



El punto de corte es el que cumple que la  $X$  y la  $Y$  en las dos rectas son los mismos:

$$X_1 + (X_2 - X_1) * t = X_3 + (X_4 - X_3) * r$$

$$Y_1 - (Y_2 - Y_1) * t = Y_4 - (Y_4 - Y_3) * r$$

Despejando:

$$t = [(Y_4 - Y_3) * (X_3 - X_1) - (Y_3 - Y_1) * (X_4 - X_3)] / [(Y_4 - Y_3) * (X_2 - X_1) - (Y_2 - Y_1) * (X_4 - X_3)]$$

## Ejercicio resuelto

```

1 public class PuntoDeCorte {
2     public static void main (String[] args) {
3
4         int X1, X2, Y1, Y2, X3, X4, Y3, Y4;
5         System.out.print("Introduce el valor de X1:");
6         X1 = Teclado.Lee_int();
7         System.out.println();
8
9         System.out.print("Introduce el valor de X2:");
10        X2 = Teclado.Lee_int();
11        System.out.println();
12
13        System.out.print("Introduce el valor de Y1:");
14        Y1 = Teclado.Lee_int();
15        System.out.println();
16
17        System.out.print("Introduce el valor de Y2:");
18        Y2 = Teclado.Lee_int();
19        System.out.println();
20
21        System.out.print("Introduce el valor de X3:");

```

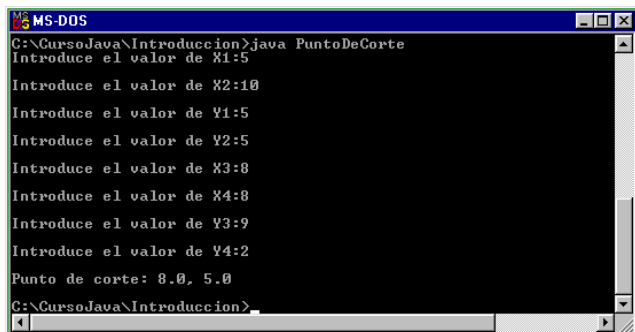
```
22     X3 = Teclado.Lee_int();
23     System.out.println();
24
25     System.out.print("Introduce el valor de X4:");
26     X4 = Teclado.Lee_int();
27     System.out.println();
28
29     System.out.print("Introduce el valor de Y3:");
30     Y3 = Teclado.Lee_int();
31     System.out.println();
32
33     System.out.print("Introduce el valor de Y4:");
34     Y4 = Teclado.Lee_int();
35     System.out.println();
36
37     int Denominador = (Y4-Y3)*(X2-X1) - (Y2-Y1)*(X4-X3);
38     if (Denominador == 0)
39         System.out.println("Las rectas son paralelas");
40     else {
41         int Numerador = (Y4-Y3)*(X3-X1) - (Y3-Y1)*(X4-X3);
42         float t = (float) Numerador / (float) Denominador;
43         float CorteX = X1 + (X2-X1)*t;
44         float CorteY = Y1 + (Y2-Y1)*t;
45         System.out.println("Punto de corte: " + CorteX +
46                             ",      "+ CorteY);
47     }
48
49 }
50 }
```

## Comentarios

En la línea 4 se declaran los 8 valores enteros que determinarán a las dos rectas. Entre las líneas 5 y 35 se pide al usuario que introduzca el valor de cada coordenada de cada punto de las rectas; obsérvese el uso de la clase *Teclado*. En la línea 37 se declara y calcula el denominador de la ecuación necesaria para obtener el parámetro  $t$ . Si el denominador es cero, no existe punto de corte, porque las rectas son paralelas (líneas 38 y 39), en caso contrario, calculamos el numerador (línea 41) y el parámetro  $t$  (línea 42), teniendo en cuenta que  $t$  no es entero, sino real (*float*). Finalmente, conocido  $t$ , podemos obtener las coordenadas  $X$  e  $Y$  correspondientes al punto de corte que nos piden (líneas 43 y 44).



## Resultados



```
MS-DOS
C:\CursoJava\Introduccion>java PuntoDeCorte
Introduce el valor de X1:5
Introduce el valor de X2:10
Introduce el valor de Y1:5
Introduce el valor de Y2:5
Introduce el valor de X3:8
Introduce el valor de X4:8
Introduce el valor de Y3:9
Introduce el valor de Y4:2
Punto de corte: 8.0, 5.0
C:\CursoJava\Introduccion>
```

### 2.5.3 Soluciones de una ecuación de segundo grado

Dada una ecuación genérica de grado 2:  $aX^2 + bX + c = 0$ , sus soluciones son:  $X = -b \pm (\sqrt{b^2 - 4ac}) / 2a$

Una ecuación de grado 2 define una parábola, que puede no cortar el eje x, cortarlo en un único punto o bien cortarlo en dos puntos.

### Ejercicio resuelto

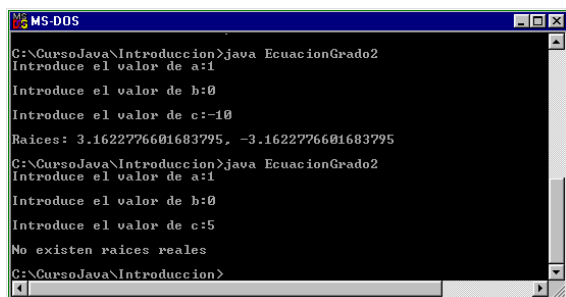
```
1 public class EcuacionGrado2 {
2     public static void main (String[] args) {
3
4         double a, b, c;
5         System.out.print("Introduce el valor de a:");
6         a = Teclado.Lee_double();
7         System.out.println();
8
9         System.out.print("Introduce el valor de b:");
10        b = Teclado.Lee_double();
11        System.out.println();
12
13        System.out.print("Introduce el valor de c:");
14        c = Teclado.Lee_double();
15        System.out.println();
16
17
18        double Auxiliari = b*b-4d*a*c;
```

```
19     if (Auxiliar<0)
20         System.out.println("No existen raices reales");
21     else
22         if (Auxiliar == 0d) {
23             System.out.print("Solo existe una raiz: ");
24             System.out.println(-b/2d*a);
25         } else { // Auxiliar mayor que cero
26             Auxiliar = Math.sqrt(Auxiliar);
27             double Raiz1 = (-b + Auxiliar) / (2 * a);
28             double Raiz2 = (-b - Auxiliar) / (2 * a);
29             System.out.println("Raices: " + Raiz1 +
30                               ", " + Raiz2);
31         }
32     }
33 }
34 }
```

## Comentarios

En la línea 4 se declaran los parámetros de la ecuación:  $a$ ,  $b$  y  $c$ . Las líneas 5 a 15 se emplean para que el usuario pueda introducir los valores deseados en los parámetros. La línea 18 declara y define el valor de una variable *Auxiliar* que contendrá el término sobre el que hay que calcular la raíz cuadrada. Si *Auxiliar* es menor que cero, no existen soluciones reales de la ecuación (la parábola se encuentra encima del eje X); si *Auxiliar* es cero existe una única solución (líneas 22 a 24) y por fin, si *Auxiliar* es mayor que cero, existen dos soluciones (dos puntos de corte de la parábola con el eje X), que se calculan en las líneas 26 a 28.

## Resultados



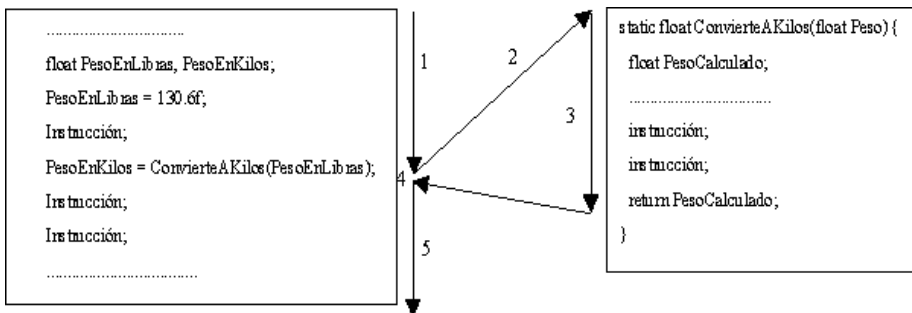
```
MS-DOS
C:\CursoJava\Introduccion>java EcuacionGrado2
Introduce el valor de a:1
Introduce el valor de b:0
Introduce el valor de c:-10
Raices: 3.1622776601683795, -3.1622776601683795
C:\CursoJava\Introduccion>java EcuacionGrado2
Introduce el valor de a:1
Introduce el valor de b:0
Introduce el valor de c:5
No existen raices reales
C:\CursoJava\Introduccion>
```

# MÉTODOS Y ESTRUCTURAS DE DATOS

## 3.1 MÉTODOS

Los métodos (procedimientos, funciones, subrutinas) nos permiten encapsular un conjunto de instrucciones de manera que puedan ser ejecutadas desde diferentes puntos de la aplicación. Por ejemplo, puede resultar útil crear un método que convierta de libras a kilos, de manera que, cada vez que se necesite realizar esta conversión se pueda invocar al método sin preocuparse de los detalles con los que está implementado.

Cuando se utiliza un método ya creado, se realiza una llamada al mismo, provocando la ejecución de sus instrucciones y devolviendo, posteriormente, el flujo de control al programa que llama al método. Gráficamente:



Obviando en este momento los detalles sintácticos, la evolución del flujo es el siguiente:

- 1. Se ejecutan las instrucciones del programa llamante hasta llegar a la llamada al método
- 2. Se hace la llamada al método (En el ejemplo “ConvierteAKilos” )
- 3. Se ejecutan las instrucciones del método
- 4. Se traspa el valor devuelto (si lo hay) al programa llamante. En nuestro ejemplo se asigna a la variable *PesoEnKilos*.
- 5. Se continúa la ejecución del programa principal a partir de la siguiente instrucción a la llamada.

No todos los métodos devuelven valores al programa principal, sólo lo hacen si es necesario. Por ejemplo, podemos crear una serie de métodos que dibujan figuras geométricas en pantalla:

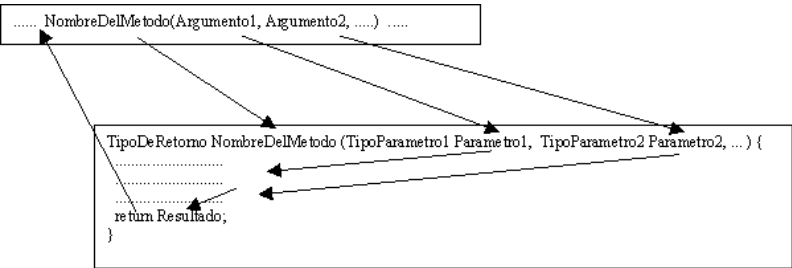
```
void DibujaCirculo(int XCentro, int YCentro, int Radio){
.....
}

void DibujaRecta(int X1, int X2, int Y1, int Y2) {
.....
}
.....
```

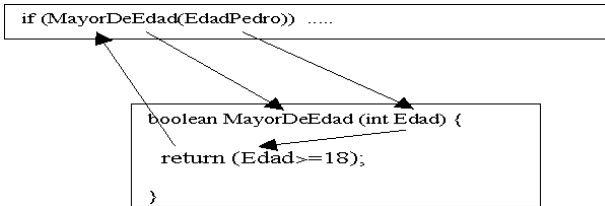
Ninguno de los métodos anteriores necesitan devolver un valor al programa que los llama. En este caso, en lugar de indicar un tipo concreto de dato antes del nombre del método (como en *ConvierteAKilos*), se pone la palabra reservada *void*.

Los métodos pueden contener parámetros o no. El método *ConvierteAKilos* contiene (entre los paréntesis) el parámetro *Peso*, de tipo *float*. Este parámetro se utiliza como base para conocer las libras correspondientes al argumento que se incluye en la llamada (en nuestro ejemplo *PesoEnLibras*). Aunque un método no incluya parámetros es necesario mantener los paréntesis después del nombre, de esta manera podemos diferenciar la llamada a un método de un nombre de variable: es diferente *Edad=EdadJuan* que *Edad=EdadJuan()*.

3.1.1 **Sintaxis**



Ejemplo:



No hay que confundir argumentos con parámetros; los parámetros son variables que utiliza el método como valores de partida para sus cálculos. Su visibilidad y ámbito (existencia) se limitan a los del propio método. Los argumentos son valores que se establecen en el programa llamante y que se traspasan (por valor o referencia como veremos más adelante) al método llamado.

La instrucción *return Resultado*; debe ser la última del método.

Cuando el método no devuelve un valor, como ya se ha comentado, hay que poner la palabra reservada *void* como tipo de retorno. En el método podemos prescindir de la instrucción *return* o utilizarla sin un valor asociado; es decir: *return*;

### 3.1.2 Ejemplo 1

El siguiente ejemplo (*Metodo1*) define un método que calcula el tamaño de la hipotenusa de un triángulo equilátero en función del tamaño de sus dos catetos. Este método es invocado dos veces por el programa principal.

```

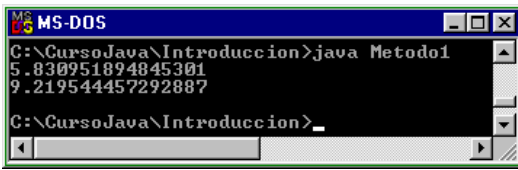
1  public class Metodo1 {
2
3      static double Hipotenusa(double Cateto1, double Cateto2) {
4          double Auxiliar;
5          Auxiliar = Cateto1*Cateto1 + Cateto2*Cateto2;
6          return Math.sqrt(Auxiliar);
7      }
8
9      public static void main(String[] args) {
10         System.out.println(Hipotenusa(5.0d,3.0d));
11         System.out.println(Hipotenusa(9.0d,2.0d));
12     }
13 }

```

En la línea 3 se define el método *Hipotenusa*, que devuelve el tamaño de la hipotenusa mediante un valor de tipo *double*. El atributo *static* se explicará con detalle en las siguientes lecciones; su uso en el ejemplo es obligatorio. El método tiene dos parámetros de tipo *double*: *Cateto1* y *Cateto2*. En las líneas 5 y 6 se calcula el valor de la hipotenusa, siendo en la línea 6 donde se devuelve el valor hallado al programa llamante.

En el programa principal (*main*), se realizan dos llamadas al método *Hipotenusa*: una en la línea 10 (con argumentos 5 y 3) y otra en la línea 11 (con argumentos 9 y 2).

### 3.1.3 Resultados del ejemplo 1



### 3.1.4 Ejemplo 2

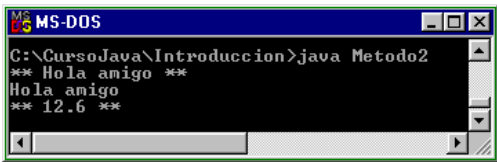
En el siguiente ejemplo se emplea un método que no devuelve ningún valor (tipo de retorno *void*). El método imprime en consola el *String* que se le pasa como argumento, precedido y seguido de dos asteriscos.

```
1 public class Metodo2 {
2
3     static void Imprimir(String Mensaje) {
4         System.out.println("** " + Mensaje + " **");
5     }
6
7     public static void main(String[] args) {
8         Imprimir("Hola amigo");
9         System.out.println("Hola amigo");
10        Imprimir("12.6");
11    }
12 }
```

En la línea 3 se define el método *Imprimir*, que no devuelve ningún valor y que tiene como parámetro un *String* (estructura de datos que veremos en breve). Su funcionalidad se codifica en la línea 4, imprimiéndose el parámetro *Mensaje* entre

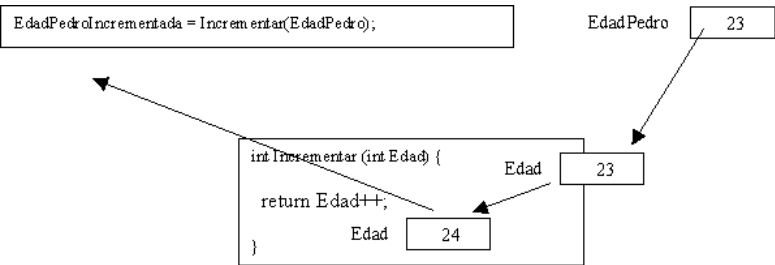
asteriscos. En las líneas 8 y 10 del programa llamante (método *main*) se invoca al método llamado con argumentos “Hola Amigo” y “12.6” (respectivamente).

3.1.5 Resultados del ejemplo 2

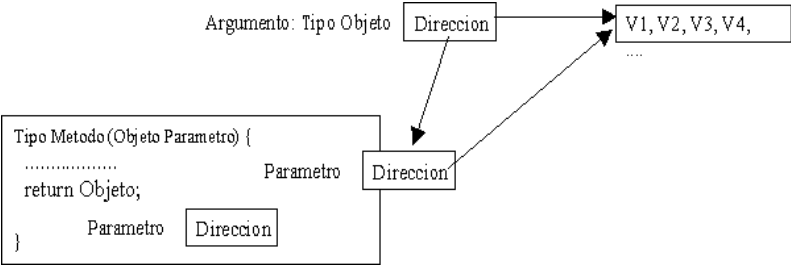


3.1.6 Paso de argumentos por valor y por referencia

Los argumentos de tipos básicos (primitivos) del lenguaje se pasan (a los parámetros) por valor, esto significa que no se traspasan los propios datos, sino una copia de los mismos. La idea central que hay que recordar, es que los argumentos de tipo *byte*, *short*, *int*, *long*, *char*, *float*, *double* y *boolean* nunca se modifican en el programa llamante (aunque sus copias varíen en el método llamado). En el siguiente diagrama, *EdadPedro* es un argumento de tipo *int*, que se pasa por valor (copia) al parámetro *Edad* del método *Incrementar*. El método varía su parámetro (que es una copia del argumento *EdadPedro*), tomando el valor 24; este valor es el que se devuelve al programa principal y se almacena en *EdadPedroIncrementada*. Tal y como cabía esperar, al finalizarse la llamada, *EdadPedro* tiene el valor 23, *EdadPedroIncrementada* tiene 24 y *Edad* ya no existe, puesto que el método termino de ejecutarse y se liberó la memoria que utilizaba.



En el paso de argumentos por referencia, lo que se copia no es el valor del argumento (en el siguiente ejemplo V1, V2, V3,... ), sino su apuntador (dirección) a la estructura de datos; cuando decimos que se modifica el valor del parámetro, realmente lo que queremos decir es: “se modifica el valor de la estructura de datos donde apunta el parámetro”, que es el mismo lugar donde apunta el argumento.



Cuando se realiza un paso de argumentos por referencia, los argumentos varían en la misma medida que varían los parámetros. Si queremos asegurarnos de que los parámetros no puedan modificarse podemos declararlos con el atributo *final*.

### 3.2 STRINGS

Los *Strings*, también llamados *literales* o *cadena de caracteres*, nos permiten declarar, definir y operar con palabras y frases. Su utilización es muy común en los programas, de hecho, nosotros, de una forma implícita los hemos usado en todos los ejemplos anteriores al escribir frases en consola por medio del método `System.out.println("El String a imprimir")`; también declaramos una estructura de datos basada en el *String* como parámetro del método *main*.

Los *Strings* no forman parte de los tipos nativos de Java, sino que existe una clase *String* (`java.lang.String`).

#### 3.2.1 Sintaxis

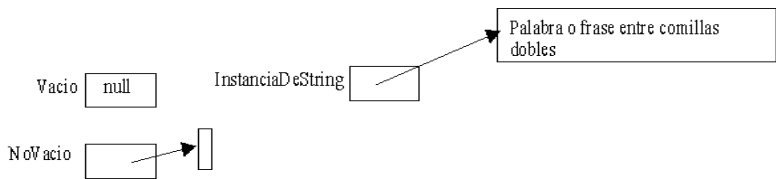
```
String InstanciaDeString = "Palabra o frase entre comillas dobles";
String OtroString = "El chico llamo 'pesado' a su amigo...";
String NoVacio = "";
String Vacio = null;
```

Un *String* se define como una serie de caracteres delimitados por comillas dobles. Las comillas simples pueden formar parte del contenido del *String*.

Una instancia de un objeto *String* es una posición de memoria que apunta hacia una estructura de datos que contiene el conjunto de caracteres que define el *String*, de esta manera, un *String* declarado, pero sin definir, todavía no apunta hacia



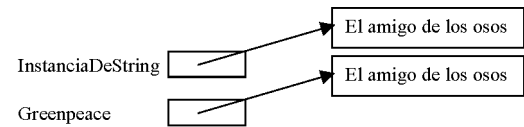
ninguna estructura de datos; su valor es *null*. Esto es diferente a una instancia del objeto *String* que apunta hacia un conjunto de caracteres vacío.



Es importante darse cuenta de que si tenemos dos instancias del objeto *String* apuntando hacia contenidos idénticos, eso no significa que sean iguales:

```
String InstanciaDeString = "El amigo de los osos";
String Greenpeace = "El amigo de los osos";

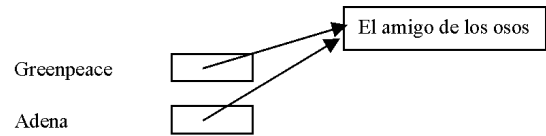
(InstanciaDeString == Greenpeace) ® falso
```



Dos *Strings* serán iguales (aplicando el operador de comparación) si apuntan hacia la misma estructura de datos:

```
String Greenpeace = "El amigo de los osos";
String Adena = Greenpeace;

(Adena == Greenpeace) ® verdadero
```



Para poder comparar dos *Strings* por su contenido (y no por su referencia), podemos utilizar el método *equals* que contiene la clase *String*. Este método compara carácter a carácter los contenidos de las dos referencias suministradas.

Ejemplo:

```
boolean Iguales = InstanciaDeString.equals(Greenpeace);  
System.out.println(Iguales); ® verdadero
```

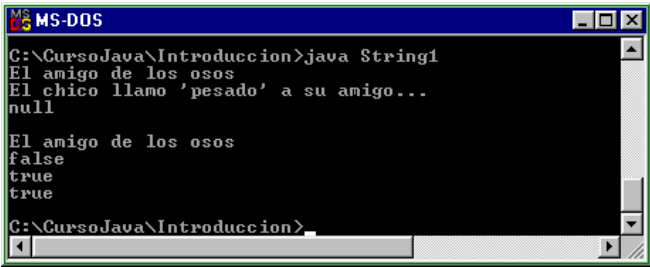
### 3.2.2 Ejemplo básico

El primer ejemplo que se proporciona implementa todos los conceptos expuestos en el apartado anterior:

```
1 public class String1 {  
2     public static void main (String[] args) {  
3         String InstanciaDeString = "El amigo de los osos";  
4         String OtroString = "El chico llamo 'pesado'  
5             a su amigo...";  
6         String Vacio = null;  
7         String NoVacio = "";  
8         String AnimalesPreferidos = "osos";  
9         String Greenpeace = "El amigo de los " +  
10             AnimalesPreferidos;  
11  
12         System.out.println(InstanciaDeString);  
13         System.out.println(OtroString);  
14         System.out.println(Vacio);  
15         System.out.println(NoVacio);  
16         System.out.println(Greenpeace);  
17         System.out.println(InstanciaDeString==Greenpeace);  
18         System.out.println(Adena==Greenpeace);  
19  
20         boolean Iguales = InstanciaDeString.equals(Greenpeace);  
21         System.out.println(Iguales);  
22  
23     }  
24 }
```

A parte de los conceptos explicados en el apartado anterior, el ejemplo incluye (en la línea 9) una característica fundamental de los *Strings*: se pueden concatenar utilizando la operación suma (+); esta es la razón por la que hemos podido poner en ejemplos anteriores líneas de código como: *System.out.println("Raices: " + Raiz1 + ", "+ Raiz2)*; en realidad estábamos concatenando *Strings* (e implícitamente realizando conversiones de distintos tipos de datos a *String*).

### 3.2.3 Resultado



### 3.2.4 Ejemplo de utilización de la clase String

El siguiente ejemplo muestra parte de los métodos más importantes de la clase *String*, que nos sirven para obtener información y manipular objetos de este tipo:

<code>public int length();</code>	Devuelve la longitud del <i>String</i>
<code>public String toLowerCase();</code>	Devuelve un <i>String</i> basado en el objeto base convertido en minúsculas
<code>public String toUpperCase();</code>	Devuelve un <i>String</i> basado en el objeto base convertido en mayúsculas
<code>public String substring(int Comienzo, int Final);</code>	Devuelve el substring formado por los caracteres situados entre las posiciones <i>Comienzo</i> y <i>Final-1</i> (posiciones numeradas a partir de 0)
<code>public int indexOf(char Carácter, int Comienzo);</code>	Devuelve la posición donde se encuentra la primera ocurrencia de <i>Carácter</i> , comenzando la búsqueda a partir de la posición <i>Comienzo</i> . Si no se encuentra ninguna ocurrencia de <i>Carácter</i> , se devuelve el valor -1

```

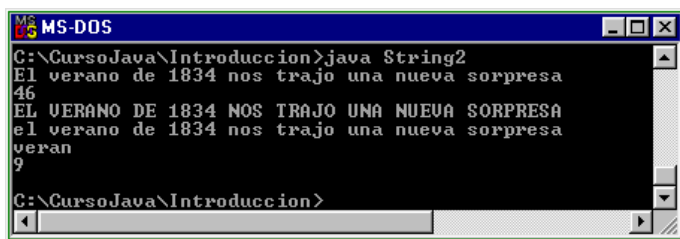
1 public class String2 {
2
3     public static void main(String[] args) {
4         String Frase = Teclado.Lee_String();
5         System.out.println(Frase.length());
6
7         String FraseMayusculas = Frase.toUpperCase();
8         System.out.println(FraseMayusculas);
9
10        String FraseMinusculas = Frase.toLowerCase();
11        System.out.println(FraseMinusculas);
12
13        String Caracteres3A18 = Frase.substring(3,8);

```

```
14      System.out.println(Caracteres3Al8);
15
16      int NumPalabras=0, Posicion=0;
17
18      while(Posicion!=-1) {
19          Posicion = Frase.indexOf(" ",Posicion+1);
20          NumPalabras++;
21      }
22      System.out.println(NumPalabras);
23
24  }
25 }
```

En la línea 4 se introduce en la variable *Frase* una sentencia que el usuario teclea. Inmediatamente (línea 8) se calcula su longitud, después se convierte en mayúsculas (líneas 7 y 8), en minúsculas (líneas 10 y 11) y se obtiene el *substring* situado entre las posiciones 3 y 7 (comenzando a contar desde cero). El bucle definido en la línea 18 se encarga de contar las palabras que tiene la frase introducida por el usuario; su funcionamiento se basa en el método *indexOf* aplicado al carácter en blanco, que actúa como separador de palabras. El bucle termina cuando no se hallan más caracteres en blanco (resultado -1).

### 3.2.5 Resultados



## 3.3 MATRICES (ARRAYS, VECTORES)

En muchas ocasiones es necesario hacer uso de un conjunto ordenado de elementos del mismo tipo. Cuando el número de elementos es grande, resulta muy pesado tener que declarar, definir y utilizar una variable por cada elemento. Pongamos por caso la gestión de notas de una asignatura en una universidad, donde puede haber 1000 alumnos matriculados en dicha asignatura. Sería inviable tener

que declarar 1000 variable diferentes, insertar las notas con 1000 instrucciones diferentes, etc.

Para solucionar este tipo de situaciones, los lenguajes de programación poseen estructuras de datos que permiten declarar (y utilizar) con un solo nombre un conjunto de variables ordenadas de un mismo tipo; estas estructuras de datos son las matrices.

### 3.3.1 *Sintaxis*

```
tipo[] Variable; // declara una matriz de variables del tipo indicado (ejemplo:
// float[] Notas;)
tipo Variable[]; // otra alternativa sintáctica para declarar la misma matriz
// que la línea anterior,
// en adelante utilizaremos únicamente la sintaxis de la primera línea

tipo[] Variable = new tipo[Elementos]; // declara y define una matriz de
// “Elementos” elementos y tipo “tipo”
// (ejemplo: float[] Notas = new float[1000];

tipo[] [] [] ... Variable =
new tipo[ElementosDim1][ElementosDim2][ElementosDim3].....; //Matriz
// multidimensional

tipo[] Variable = {Valor1, Valor2, Valor3, .....}; // Declaración e inicialización de
// una matriz
```

Ejemplos:

```
float[] Temperaturas; // Se declara una matriz Temperaturas de elementos  
// de tipo float  
Temperaturas = new float[12]; // Se define la variable Temperaturas como  
// una matriz de 12 elementos de tipo float
```

Obsérvese la utilización de la palabra reservada *new*, que se emplea para crear un objeto del tipo especificado. Las dos instrucciones anteriores se podrían fundir en:

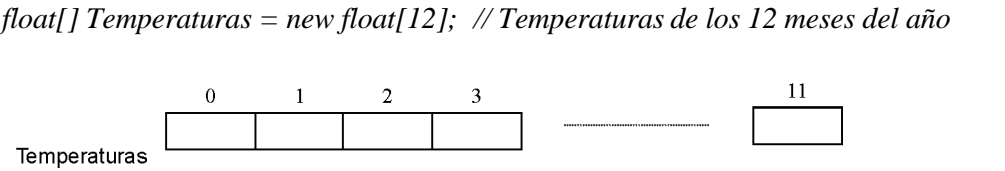
```
float[] Temperaturas = new float[12]; //Temperaturas de los 12 meses del año
float[] [] Temperaturas = new float[12] [31]; //Temperaturas de cada día en los
// 12 meses del año
```

```
float[] TemperaturasMadrid = {3.4f, 5.6f, 17.5f, 19.2f, 21.0f, 25.6f, 35.3f, 39.5f,
                               25.2f, 10.0f, 5.7f, 4.6f};
String[] Nombres = {"Pinta", "Niña", "Santamaría"};
```

Este es el momento de poder entender el parámetro que siempre utilizamos en el método *main*: *String[] args*, con el que definimos una matriz unidimensional de *Strings*. El nombre de la matriz es *args*. En esta matriz se colocan los posibles parámetros que deseamos pasar al invocar el programa.

3.3.2 Acceso a los datos de una matriz

En las matrices, los elementos se colocan de forma ordenada, linealmente, numerando las posiciones individuales que componen la matriz a partir del valor de índice 0. Por ejemplo, nuestra matriz *Temperaturas* se puede representar de la siguiente manera:



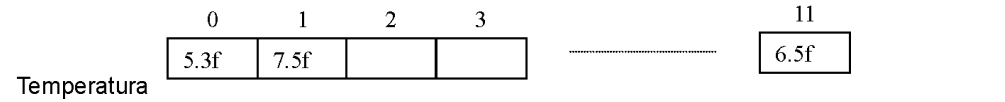
Para almacenar la temperatura media de febrero (pongamos 7.5 grados), utilizamos la sentencia de asignación:

```
Temperaturas[1] = 7.5f;
```

Para almacenar las de enero y diciembre:

```
Temperaturas[0] = 5.3f;
Temperaturas[11]=6.5f;
```

En este momento, la matriz *Temperaturas* tiene los siguiente valores:



Para hallar la temperatura media de los meses de invierno:

```
float MediaInvierno = (Temperaturas[0] +Temperaturas[1] +
                        Temperaturas[2]) / 3f;
```

Una de las grandes ventajas que presentan las matrices es la posibilidad que existe para recorrer sus elementos con una instrucción repetitiva:

```
for (int i=0; i<12; i++)  
    System.out.println(Temperaturas[i]);
```

### 3.3.3 Ejemplo 1

En el ejemplo *Matriz1* se define una matriz unidimensional de notas y otra de nombres, la primera de elementos de tipo *float* y la segunda de elementos de tipo *String*, ambas matrices dimensionadas a 10 elementos. Se introducen los nombres y las notas obtenidas, se imprime la lista y posteriormente se imprime el listado de aprobados:

```
1  public class Matriz1 {  
2      public static void main(String[] args) {  
3          float[] Notas =  
4              {5.8f,6.2f,7.1f,5.9f,3.6f,9.9f,1.2f,10.0f,4.6f,5.0f};  
5          String[] Nombres = new String[10];  
6          Nombres[0]="Pedro";Nombres[1]="Ana";Nombres[2]="Luis";  
7          Nombres[3]="Luis";Nombres[4]="Juan";Nombres[5]="Eva";  
8          Nombres[6]="Mari";Nombres[7]="Fran";Nombres[8]="Luz";  
9          Nombres[9]="Sol";  
10  
11         for (int i=0;i<Nombres.length;i++)  
12             System.out.println(Nombres[i] + " : " + Notas[i]);  
13  
14         System.out.println();  
15  
16         byte Aprobados = 0;  
17         String NombresAprobados = new String();  
18         for (int i=0;i<Nombres.length;i++)  
19             if (Notas[i]>=5.0){  
20                 Aprobados++;  
21                 NombresAprobados = NombresAprobados+" "+Nombres[i];  
22             }  
23         System.out.println("Aprobados: " + Aprobados);  
24         System.out.println("Aprobados:" + NombresAprobados);  
25  
26     }  
27 }
```

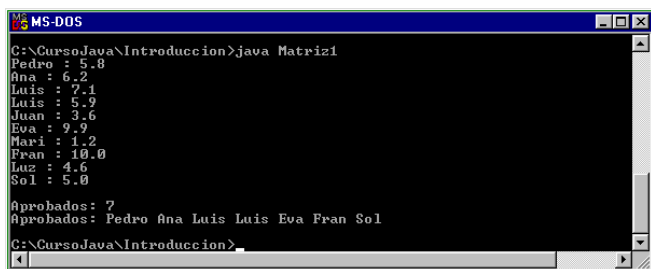
En la línea 3 se declara la matriz *Notas* y se inicializa con 10 valores *float* entre 0 y 10. En la línea 5 se declara la matriz *Nombres*, que contiene 10 variables ordenadas de tipo *String*. En las líneas 6 a 9 se introducen los 10 nombres de los alumnos.

La línea 11 implementa un bucle *for* que recorre el rango de posiciones de índice entre 0 y 9. Obsérvese el uso de la propiedad *length* del objeto *Array*, que nos permite conocer el número de elementos de una matriz unidimensional (en este caso 10). En la línea 12, dentro del bucle, se imprime el nombre y la nota de cada alumno.

Entre las líneas 16 y 22 se determina el número de alumnos que han aprobado, guardándose sus nombres en el *String*: *NombresAprobados*, que ha sido definido como un nuevo (*new*) objeto del tipo *String* (línea 17).

El bucle de la línea 18 itera 10 veces recorriendo la matriz *Notas*, consultándose en cada iteración si la nota del alumno en cuestión (*i*) es un aprobado (línea 19), en caso de que lo sea se incrementa un contador de aprobados: *Aprobados* y se concatena el nombre del nuevo “afortunado” a la lista de los aprobados anteriores (línea 21). Finalmente se imprimen los resultados (líneas 23 y 24).

### 3.3.4 Resultados del ejemplo 1



```
MS-DOS
C:\CursoJava\Introduccion>java Matriz1
Pedro : 5.8
Ana : 6.2
Luis : 7.1
Luis : 5.9
Juan : 3.6
Eva : 9.9
Mari : 1.2
Fran : 10.0
Luz : 4.6
Sol : 5.0
Aprobados: 7
Aprobados: Pedro Ana Luis Luis Eva Fran Sol
C:\CursoJava\Introduccion>
```

### 3.3.5 Ejemplo 2

Este ejemplo se basa en el anterior, aumentando sus funcionalidades. Se proporciona un método *Imprimir*, que visualiza por consola los contenidos de los elementos de la Matriz de tipo *String* que se le pasa como parámetro. También se codifica el método *EntreNotas*, que admite como parámetros dos notas y una matriz de *floats*, devolviendo el número de elementos de la matriz que se encuentran en el rango de notas suministrado.



El método *Imprimir*, que no devuelve ningún valor, se codifica entre las líneas 3 y 6. El método *EntreNotas*, que devuelve un valor de tipo *byte*, se encuentra entre las líneas 8 y 16; en la línea 11 se recorre toda la matriz de notas, preguntándose en cada iteración si la nota se encuentra en el rango especificado (línea 12). La variable *Contador* acumula el número de notas que cumplen las condiciones.

En el programa principal (*main*) se declaran e inicializan las matrices de nombres y notas, se hace una llamada al método *Imprimir* y tres al método *EntreNotas*: una para obtener los aprobados, otra para los suspensos y la última para las matrículas (líneas 28, 31 y 34).

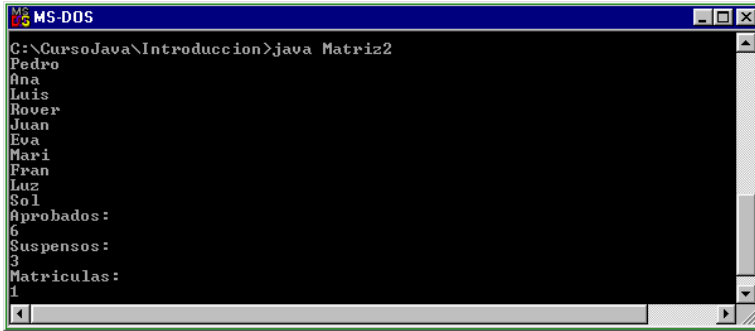
```
1 public class Matriz2 {
2
3     public static void Imprimir(String[] Nombres) {
4         for (int i=0;i<Nombres.length;i++)
5             System.out.println(Nombres[i]);
6     }
7
8     public static byte EntreNotas(float Notal, float Nota2,
9                                   float[] Notas) {
10        byte Contador = 0;
11        for (int i=1;i<Notas.length;i++){
12            if ((Notas[i]>=Notal)&&(Notas[i]<=Nota2))
13                Contador++;
14        }
15        return Contador;
16    }
17
18    public static void main(String[] args) {
19        float[] Notas =
20            {5.8f,6.2f,7.1f,5.9f,3.6f,9.9f,1.2f,10.0f,4.6f,5.0f};
21        String[] Nombres = new String[10];
22        Nombres[0]="Pedro";Nombres[1]="Ana";Nombres[2]="Luis";
23        Nombres[3]="Rover";Nombres[4]="Juan";Nombres[5]="Eva";
24        Nombres[6]="Mari";Nombres[7]="Fran";
25        Nombres[8]="Luz";Nombres[9]="Sol";
26
27        Imprimir(Nombres);
28        System.out.println("Aprobados: ");
29        System.out.println(EntreNotas(5.0f,10.0f,Notas));
30
31        System.out.println("Suspensos: ");
32        System.out.println(EntreNotas(0.0f,4.9f,Notas));
33
34        System.out.println("Matriculas: ");
35        System.out.println(EntreNotas(10.0f,10.0f,Notas));
```

```

36     }
37 }

```

### 3.3.6 Resultados del ejemplo 2



```

C:\CursoJava\Introduccion>java Matriz2
Pedro
Ana
Luis
Rover
Juan
Eva
Mari
Fran
Luz
Sol
Aprobados:
6
Suspendidos:
3
Matriculas:
1

```

### 3.3.7 Ejemplo

El último ejemplo de esta lección, basado en los anteriores, permite pasar como parámetro una matriz bidimensional a los métodos *Imprimir* y *EntreNotas*. También se utiliza una constante (atributo *final*) *NUM\_ALUMNOS*, definida en la línea 22, para dimensionar el tamaño de la matriz bidimensional.

La matriz se ha definido de tipo *String*, lo que encaja muy bien con los campos que tienen que albergar nombres, sin embargo, las notas hay que transformarlas de *String* a tipo *float*, lo que se realiza con facilidad gracias al método *parseFloat* de la clase *Float* (líneas 12 y 13). La operación contraria también nos resulta necesaria; en la línea 31 se convierte de *float* a *String* utilizando el método *toString* de la clase *Float*.

Obsérvese como los argumentos de tipo matriz se colocan sin corchetes (líneas 34, 37, 40 y 43), mientras que en los parámetros hay que definir tanto el tipo (con corchetes) como el nombre del parámetro (líneas 3 y 9).

A lo largo de todo el ejemplo aparece la manera de utilizar una matriz bidimensional, colocando adecuadamente los índices en cada dimensión (delimitadas por corchetes).

```

1  public class Matriz3 {
2
3      public static void Imprimir(String[][] Alumnos) {

```



## 3.4 EJEMPLOS DE PROGRAMACIÓN

En esta lección se presentan tres ejemplos resueltos y comentados, en los que se hace uso de la mayor parte de los conceptos explicados en las lecciones anteriores. En el primero de estos ejemplos se calculan números primos, en el segundo se generan claves utilizando bucles anidados y en el tercer ejemplo se hace uso de métodos que admiten matrices lineales como argumentos.

### 3.4.1 Obtención de números primos

En este ejercicio se pretende obtener un listado de los ‘n’ primeros números primos, siendo ‘n’ un valor que definimos como constante en el programa. Los números primos deben guardarse en una matriz lineal de enteros.

La obtención de los números primos se basará en la siguiente propiedad: para saber si un número es primo, no es necesario dividirlo por todos los valores anteriores, basta con constatar que no es divisible por ningún número primo menor que su propio valor, por ejemplo: el 17 es primo, porque no es divisible entre 2, 3, 5, 7, 11 y 13 (que son los primos anteriores); no es necesario dividir 17 entre 4, 6, 8, 9, 10, 12, 14, 15 y 16 para saber que es primo.

### Solución

```
1 public class GeneraPrimos {
2     public static void main (String[] args) {
3         final int NUM_PRIMOS = 60;
4         int[] Primos = new int[NUM_PRIMOS];
5         Primos[0] = 2;
6         int PrimosHallados = 1;
7         int PosiblePrimo = 3;
8         int Indice=0;
9         boolean Primo = true;
10
11     do {
12         while (Primo && (Indice<PrimosHallados)) {
13             if (PosiblePrimo % Primos[Indice] == 0)
14                 Primo = false;
15             else
16                 Indice++;
17         }
18
19         if (Primo) {
20             Primos[PrimosHallados] = PosiblePrimo;
21             PrimosHallados++;
```

```
22     }
23
24     Primo = true;
25     PosiblePrimo++;
26     Indice = 0;
27
28 } while (PrimosHallados<NUM_PRIMOS );
29
30 for (int i=0; i<NUM_PRIMOS;i++) {
31     System.out.print(Primos[i] + "  ");
32     if (i % 10 ==0)
33         System.out.println();
34 }
35
36 }
37 }
```

La clase *GeneraPrimos* calcula los primeros *NUM\_PRIMOS* (línea 3) primos. Estos primos se irán almacenando en una matriz *Primos* de valores enteros (línea 4). Como nuestro algoritmo se basa en dividir entre los primos anteriores, introducimos el primer primo (el número 2) que nos servirá de base para ir generando los demás; esta asignación se realiza en la línea 5. En la línea 6 se indica que ya tenemos un número primo almacenado en la matriz.

La línea 7 nos inicializa un contador *PosiblePrimo* a 3, que es el siguiente valor sobre el que tenemos que determinar si es primo. En la línea 8 se inicializa la variable *Indice*, que nos indica el siguiente primo hallado (y almacenado en la matriz *Primos*) por el que hay que dividir el valor *PosiblePrimo*. En la línea 9 se inicializa una variable *Primo* que nos indicará en todo momento si *PosiblePrimo* va siendo divisible entre los primos hallados hasta el momento.

El bucle situado en las líneas 11 y 28 va iterando hasta que se han hallado los *NUM\_PRIMOS* buscados. Cuando se sale de este bucle los primos han sido calculados y, a continuación, en el bucle *for* de la línea 30 se imprimen en grupos de 10.

El bucle *while* de la línea 12 comprueba si *PosiblePrimo* es o no un número primo,

La instrucción condicional de la línea 13 es el núcleo del programa: calcula el resto de la división de *PosiblePrimo* con cada primo hallado previamente (y almacenado en la matriz *Primos*). Si el resto es cero, *PosiblePrimo* no es un número primo (*Primo = false*); si el resto es distinto de cero habrá que volver a probar con el siguiente primo de nuestra matriz. Esta instrucción se ejecuta (en el *while* de la línea 12) mientras *PosiblePrimo* siga siendo primo (*Primo es true*) e *Indice* no haya alcanzado el número de primos que hemos hallado previamente.

Del bucle anterior se puede salir por dos causas: que *PosiblePrimo* haya pasado todas las comprobaciones de divisibilidad y por lo tanto es primo (*Primo* es *true*), o que no haya pasado las comprobaciones y por lo tanto no es primo (*Primo* es *false*). En el primer caso se actualiza la matriz *Primos* y el contador asociado: *PrimosHallados* (líneas 20 y 21). En cualquier caso, al salir del bucle, se preparan las variables para comprobar el siguiente valor en secuencia (líneas 24, 25 y 26).

## Resultados

```

C:\CursoJava\Introduccion>java GeneraPrimos
2
3 5 7 11 13 17 19 23 29 31
37 41 43 47 53 59 61 67 71 73
79 83 89 97 101 103 107 109 113 127
131 137 139 149 151 157 163 167 173 179
181 191 193 197 199 211 223 227 229 233
239 241 251 257 263 269 271 277 281
C:\CursoJava\Introduccion>

```

### 3.4.2 “Revienta claves”

Este ejemplo muestra como combinar caracteres, con el fin de generar todas las posibles claves que se pueden utilizar con un juego de caracteres permitido y un tamaño máximo establecido. El juego de caracteres que utilizaremos son los números y letras (mayúsculas y minúsculas), junto a unos pocos caracteres especiales de uso común.

Los caracteres de nuestro “abecedario” los podemos obtener con el siguiente programa en Java:

```

1 public class RevientaClaves2 {
2
3     public static void main (String[] args) {
4
5         char Caracter = '0';
6         for (int i=1;i!=76;i++) {
7             System.out.print(Caracter + " ");
8             Caracter++;
9         }
10    }
11 }

```

Empezando por el carácter numérico '0' (línea 5), imprimimos 75 caracteres en secuencia. El resultado es:

```

C:\CursoJava\Introduccion>java RevientaClaves2
0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z
C:\CursoJava\Introduccion>

```

## Solución

```

1  public class RevientaClaves {
2
3
4      static boolean Permiso(String Clave) {
5          return (Clave.equals("0A:g"));
6      }
7
8
9      public static void main (String[] args) {
10
11          String Clave = "";
12          char Car1 = '0', Car2 = '0', Car3 = '0', Car4 = '0';
13          boolean Encontrada = false;
14
15          do {
16              do {
17                  do {
18                      do {
19                          Clave = "" + Car1 + Car2 + Car3 + Car4;
20                          System.out.println("*"+Clave+"*");
21                          if (Permiso(Clave)) {
22                              System.out.println ("Permiso concedido");
23                              Encontrada = true;
24                          }
25                          Car4++;
26                      } while (Car4 != 'z' && !Encontrada);
27                      Car4 = '0';
28                      Car3++;
29                  } while (Car3 != 'z' && !Encontrada);
30                  Car3 = '0';
31                  Car2++;
32              } while (Car2 != 'z' && !Encontrada);
33              Car2 = '0';
34              Car1++;
35          } while (Car1 != 'z' && !Encontrada);
36

```

```
37  
38 }  
39 }
```

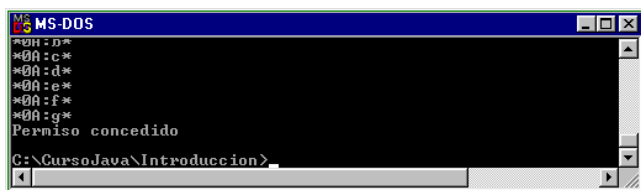
En las líneas 4 a 6 implementamos un método que simula la entrada al sistema: la línea 5 compara el *String* que se introduce como parámetro con una clave prefijada.

En la línea 11 se inicializa el *String Clave*, en el que introduciremos las diferentes claves que generemos por programa, y con las que supuestamente intentamos “introducimos” en un sistema. En la línea 12 se inicializan 4 variables de tipo carácter, con las que vamos a generar la clave (en este ejemplo, siempre de longitud 4). En la línea 13 establecemos una variable lógica *Encontrada*, que utilizaremos para finalizar las iteraciones cuando la clave se encuentre.

En las líneas 15, 16, 17 y 18 se inician 4 bucles, uno por cada carácter (1º, 2º, 3º, 4º) de cada clave generada, de tal forma que cada carácter va aumentando (líneas 25, 28, 31 y 34) en cada iteración de su bucle correspondiente. Cada iteración de un bucle exterior debe inicializar el carácter del bucle inferior (líneas 27, 30 y 33).

En el bucle más interno se intenta acceder al “sistema” (línea 21); cuando se consigue se imprime un mensaje y se almacena el valor *true* en la variable *Encontrada* (línea 23), con lo que los bucles dejan de iterar y el programa finaliza.

## Resultado



### 3.4.3 Estadísticas

En este ejemplo se muestra, especialmente, como utilizar matrices lineales de elementos. Se definen matrices de temperaturas medias de cada mes de un año en



Madrid y Granada (con valores inventados), también se define una matriz de 28 elementos que contiene presiones atmosféricas.

En el ejemplo se suministran tres métodos: uno para hallar la media de los valores de la matriz que se le pasa como parámetro, otro para dar una medida de variación respecto a la media y el tercero para obtener los valores extremos (mínimo y máximo) de la matriz. Estos métodos funcionan con corrección, independientemente del tamaño de la matriz que se les suministre.

## Solución

```
1 public class Estadisticas {
2
3     static float Media(float[] Matriz) {
4         float Suma = 0;
5         for (int i=0; i<Matriz.length; i++)
6             Suma = Suma + Matriz[i];
7         return Suma/Matriz.length;
8     }
9
10    static float Variacion(float[] Matriz) {
11        float Suma = 0;
12        float Media = Media(Matriz);
13        for (int i=0; i<Matriz.length; i++)
14            Suma = Suma + Math.abs(Media - Matriz[i]);
15        return Suma/Matriz.length;
16    }
17
18    static float[] MenorMayor(float[] Matriz) {
19        float[] Auxiliar = new float[2]; //Contendra el menor
                                         //y el mayor valor
20        Auxiliar[0] = Matriz[0]; //Por ahora el menor es
                                         //el primero
21        Auxiliar[1] = Matriz[0]; //Por ahora el mayor es
                                         //el primero
22        for (int i=0; i<Matriz.length; i++) {
23            if (Matriz[i] < Auxiliar[0])
24                Auxiliar[0] = Matriz[i];
25            if (Matriz[i] > Auxiliar[1])
26                Auxiliar[1] = Matriz[i];
27        }
28        return Auxiliar;
29    }
30
31    public static void main (String[] args) {
32
33        float[] TemperaturasMadrid =
```

```
34  {6.7f, 5.8f, 14.5f, 15.0f, 18.5f, 22.3f,
    35.6f, 38.0f, 28.6f, 19.4f, 14.6f, 8.5f};

35      float[] TemperaturasGranada =
36  {5.2f, 4.6f, 12.5f, 19.3f, 16.2f, 24.5f,
    37.2f, 37.0f, 29.3f, 18.2f, 10.3f, 7.5f};

37      float[] PresionFebrero =
38  {800.2f, 810.5f, 815.2f, 825.6f, 837.4f, 850.2f, 860.4f,
39  855.2f, 847.2f, 820.4f, 810.5f, 805.2f, 790.3f, 795.1f,
40  790.4f, 786.6f, 780.6f, 770.3f, 770.4f, 777.7f, 790.3f,
41  820.2f, 830.7f, 840.1f, 845.6f, 859.4f, 888.2f, 899.4f};
42
43      float[] mM = new float[2]; // Contendra el menor y
                                // el mayor valor
44      System.out.println("Media temperaturas de Madrid: " +
45                          Media(TemperaturasMadrid) );
46      System.out.println("Variacion temperaturas de Madrid:
47                          " + Variacion(TemperaturasMadrid) );
48      mM = MenorMayor(TemperaturasMadrid);
49      System.out.println("Menor y mayor temperatura de
50                          Madrid: "+ mM[0] + " " + mM[1] );
51      System.out.println();
52
53      System.out.println("Media temperaturas de Granada: " +
54                          Media(TemperaturasGranada) );
55      System.out.println("Variacion temperaturas de Granada:
56                          " + Variacion(TemperaturasGranada) );
57      mM = MenorMayor(TemperaturasGranada);
58      System.out.println("Menor y mayor temperatura de
59                          Granada: "+mM[0] + " " + mM[1] );
60      System.out.println();
61
62      System.out.println("Media presion de febrero:
63                          "+Media(PresionFebrero));
64      System.out.println("Variacion presion de febrero: " +
65                          Variacion(PresionFebrero) );
66      mM = MenorMayor(PresionFebrero);
67      System.out.println("Menor y mayor presion de febrero:
68                          " + mM[0] + " " + mM[1] );
69  }
```

En las líneas 33 a 41 del programa principal se declaran y definen las matrices que contienen las temperaturas medias de los meses de un año en Madrid y Granada y las presiones medias de cada día de febrero. Estas matrices (*TemperaturasMadrid*, *TemperaturasGranada*, *PresionFebrero*) se utilizan como argumentos que se les pasa a los métodos definidos antes del programa principal.

Los métodos implementados son *Media*, *Variación* y *MenorMayor*, este último devuelve una matriz lineal de valores *float*. El primer elemento contiene el menor valor de la matriz que se le ha pasado, mientras que en el segundo elemento se coloca el mayor de los valores de la matriz.

La línea 43 (en el programa principal) declara una matriz lineal (*mM*) de 2 valores de tipo *float*, que nos servirá para recoger los valores mínimos y máximos (*mM[0]* y *mM[1]*) que nos devuelve el método *MenorMayor*.

En la línea 45 se obtiene la temperatura media del año en “Madrid”, invocando al método *Media* con el argumento *TemperaturasMadrid*. En la línea 47 se obtiene una medida de la variación de la temperaturas del año en “Madrid”, invocando al método *Variacion* con el argumento *TemperaturasMadrid*. En la línea 48 se obtienen las temperaturas mínima y máxima de la media de los meses en Madrid, para ello se utiliza el método *MenorMayor*, con argumento *TemperaturasMadrid* y recogiendo el resultado en la matriz *mM*.

El resto de las líneas hasta el final del programa principal repiten la secuencia explicada, pero utilizando en este caso como argumentos las matrices *TemperaturasGranada* y *PresionFebrero*.

Del método *Media* (situado entre las líneas 3 y 8) cabe resaltar la utilización de la propiedad *length*, asociada al array *Matriz*. Utilizando esta propiedad, que nos indica el número de elementos de la matriz lineal, conseguimos que el método *Media* sea independiente del tamaño de la matriz: obsérvese como funciona adecuadamente con *TemperaturasMadrid*, de 12 elementos y con *PresionFebrero*, de 28 elementos. Esta propiedad la utilizamos también en los demás métodos del ejemplo.

El método *Variación*, implementado entre las líneas de código 10 y 16, necesita el valor medio de los valores de la matriz que se le suministra como argumento. Este valor lo obtiene en la línea 12, llamando al método *Media*. Obsérvese como no existe ambigüedad entre la variable *Media* y la llamada al método *Media*(.....). El funcionamiento de este método se codifica en la línea 14, hallándose la diferencia (variación), en valor absoluto, de cada valor de la matriz respecto a la media de todos los valores.

El método *MenorMayor* (línea 18) presenta la novedad de devolver una matriz (aunque sea de sólo dos elementos) en lugar de un valor de tipo primitivo del lenguaje. Para su implementación, inicialmente se considera el primer elemento de *Matriz* como el mayor y el menor de todo el array (líneas 20 y 21) y después se utiliza un bucle (línea 22) que va recorriendo la matriz y actualizando los valores del menor y mayor elementos encontrados (líneas 24 y 26). Finalmente, en la línea 28, se devuelve el array *Auxiliar* que contiene los dos valores buscados.

## Resultados



The screenshot shows a black MS-DOS command window with white text. The title bar reads 'MS-DOS'. The command prompt shows the directory 'C:\CursoJava\Introduccion' and the command 'java Estadisticas'. The output displays statistical data for Madrid and Granada, including average temperatures, temperature variations, and pressure for February.

```
MS-DOS
C:\CursoJava\Introduccion>java Estadisticas
Media temperaturas de Madrid: 18.958334
Variacion temperaturas de Madrid: 8.184722
Menor y mayor temperatura de Madrid: 5.8 38.0

Media temperaturas de Granada: 18.483334
Variacion temperaturas de Granada: 9.147222
Menor y mayor temperatura de Granada: 4.6 37.2

Media presion de febrero: 820.475
Variacion presion de febrero: 28.121428
Menor y mayor presion de febrero: 770.3 899.4

C:\CursoJava\Introduccion>
```

# PROGRAMACIÓN ORIENTADA A OBJETOS USANDO CLASES

---

## 4.1 DEFINICIÓN DE CLASES E INSTANCIAS

Las clases son los objetos que Java utiliza para soportar la programación orientada a objetos. Constituyen la estructura básica sobre la que se desarrollan las aplicaciones.

Una clase permite definir propiedades y métodos relacionados entre sí. Habitualmente, las propiedades son las variables que almacenan el estado de la clase y los métodos son los programas que se utilizan para consultar y modificar el contenido de las propiedades.

Un ejemplo de clase podría ser un semáforo de circulación, cuyo estado se guarde en una propiedad *EstadoSemaforo* de tipo *String* que pueda tomar los valores “Verde”, “Amarillo” y “Rojo”. Como métodos de acceso a la propiedad podríamos definir: *PonColor(String Color)* y *String DimeColor()*.

### 4.1.1 Sintaxis

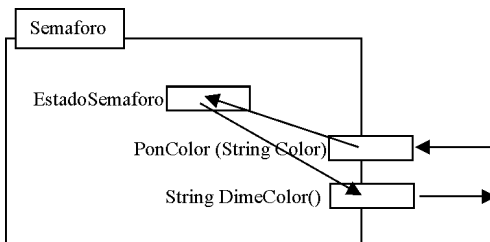
```
AtributoAcceso class NombreClase {  
    // propiedades y métodos  
}
```

Obviando el significado del atributo de acceso, que se explicará más tarde, un ejemplo concreto de clase en Java podría ser:

```
public class Semaforo {  
    private String EstadoSemaforo = "Rojo";  
  
    public void PonColor (String Color) {  
        EstadoSemaforo = Color;  
    }  
  
    public String DimeColor() {  
        return EstadoSemaforo;  
    }  
  
} // Fin de la clase Semaforo
```

#### 4.1.2 Representación gráfica

Gráficamente, la clase *Semaforo* la podríamos definir de la siguiente manera:



La propiedad *EstadoSemaforo*, con atributo *private*, no es accesible directamente desde el exterior de la clase, mientras que los métodos, con atributo *public*, si lo son. Desde el exterior de la clase podemos acceder a la propiedad *EstadoSemaforo* a través de los métodos *PonColor* y *DimeColor*.

#### 4.1.3 Instancias de una clase

Cuando definimos una clase, estamos creando una plantilla y definiendo un tipo. Con el tipo definido y su plantilla de código asociada (sus propiedades y métodos) podemos crear tantas entidades (instancias) de la clase como sean necesarias; de esta manera, en nuestro ejemplo, podemos crear varios semáforos

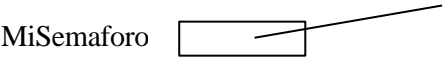
(instancias de la clase *Semáforo*), y hacer evolucionar el estado de estos “semáforos” de forma independiente.

Si deseamos disponer de diversos semáforos independientes entre sí, en el sentido de que cada semáforo pueda encontrarse en un estado diferente a los demás, obligatoriamente debemos crear (instanciar) cada uno de estos semáforos.

Para declarar un objeto de una clase dada, empleamos la sintaxis habitual:  
*Tipo Variable;*

En nuestro caso, el tipo se refiere al nombre de la clase:  
*Semaforo MiSemaforo;*

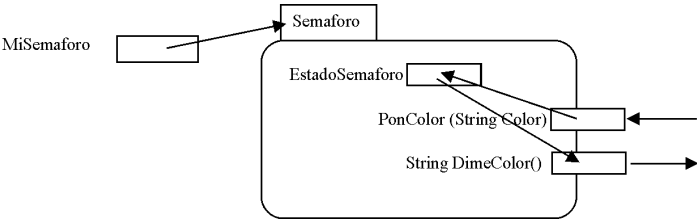
De esta manera hemos creado un apuntador (*MiSemaforo*) capaz de direccionar un objeto (una instancia) de la clase *Semaforo*:



Para crear una instancia de la clase *Semaforo*, empleamos la palabra reservada *new*, tal y como hacíamos para crear una instancia de una matriz; después invocamos a un método que se llame igual que la clase. Estos métodos se denominan constructores y se explicarán un poco más adelante.  
*MiSemaforo = new Semaforo();*

También podemos declarar e instanciar en la misma instrucción:  
*Semaforo MiSemaforo = new Semaforo();*

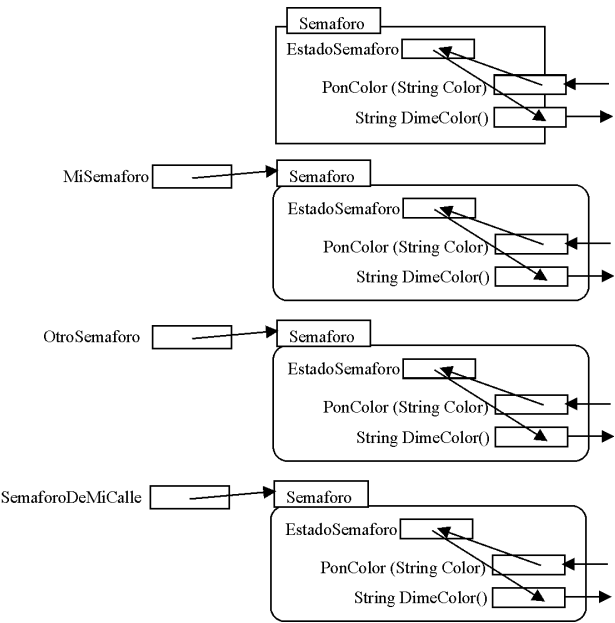
En cualquier caso, el resultado es que disponemos de una variable *MiSemaforo* que direcciona un objeto creado (instanciado) de la clase *Semaforo*:



Podemos crear tantas instancias como necesitemos:  
*Semaforo MiSemaforo = new Semaforo();*  
*Semaforo OtroSemaforo = new Semaforo();*

```
Semáforo SemaforoDeMiCalle = new Semaforo();
```

El resultado es que disponemos del tipo *Semaforo* (de la clase *Semaforo*) y de tres instancias (*MiSemaforo*, *OtroSemaforo*, *SemaforoDeMiCalle*) de la clase:



Es importante ser consciente de que en este momento existen tres variables diferentes implementando la propiedad *EstadoSemaforo*; cada una de estas variables puede contener un valor diferente, por ejemplo, cada semáforo puede presentar una luz distinta (“Verde”, “Rojo”, “Amarillo”) en un instante dado.

4.1.4 Utilización de los métodos y propiedades de una clase

Para designar una propiedad o un método de una clase, utilizamos la notación punto:

```
Objeto.Propiedad
Objeto.Metodo()
```



De esta forma, si deseamos poner en verde el semáforo *SemaforoDeMiCalle*, empleamos la instrucción:

```
SemaforoDeMiCalle.PonColor("Verde");
```

De igual manera podemos actuar con las demás instancias de la clase *Semaforo*:

```
MiSemaforo.PonColor("Rojo");  
OtroSemaforo.PonColor("Verde");
```

Para consultar el estado de un semáforo:

```
System.out.println( OtroSemaforo.DimeColor() );  
if (MiSemaforo.DimeColor().equals("Rojo"))  
String Luz = SemaforoDeMiCalle.DimeColor();
```

En nuestro ejemplo no podemos acceder directamente a la propiedad *EstadoSemaforo*, por ser privada. En caso de que fuera pública se podría poner:

```
String Luz = SemaforoDeMiCalle.EstadoSemaforo; // sólo si EstadoSemaforo es  
// accesible (en nuestro ejemplo NO lo es)
```

### 4.1.5 Ejemplo completo

En esta sección se presenta el ejemplo del semáforo que hemos ido desarrollando. Utilizaremos dos clases: la clase definida (*Semaforo*) y otra que use esta clase y contenga el método *main* (programa principal). A esta última clase la llamaremos *PruebaSemaforo*.

```
1 public class Semaforo {  
2     String EstadoSemaforo = "Rojo";  
3  
4     public void PonColor (String Color) {  
5         EstadoSemaforo = Color;  
6     }  
7  
8     public String DimeColor() {  
9         return EstadoSemaforo;  
10    }  
11  
12 } // Fin de la clase Semaforo
```

```

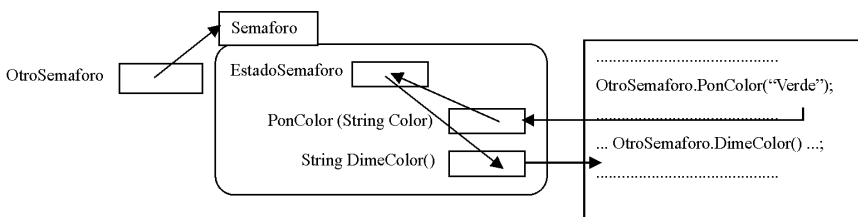
13 public class PruebaSemaforo {
14     public static void main (String[] args) {
15         Semaforo MiSemaforo = new Semaforo();
16         Semaforo SemaforoDeMiCalle = new Semaforo();
17         Semaforo OtroSemaforo = new Semaforo();
18
19         MiSemaforo.PonColor("Rojo");
20         OtroSemaforo.PonColor("Verde");
21
22         System.out.println( OtroSemaforo.DimeColor() );
23         System.out.println( SemaforoDeMiCalle.DimeColor() );
24
25         if (MiSemaforo.DimeColor().equals("Rojo"))
26             System.out.println ("No Pasar");
27     }
28 }
29 }

```

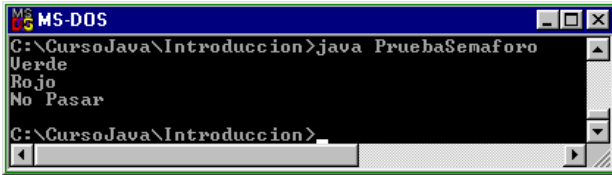
En las líneas 3, 4 y 5 de la clase *PruebaSemaforo* se declaran e instancian las variables *MiSemaforo*, *SemaforoDeMiCalle* y *OtroSemaforo*. En las líneas 7 y 8 se asignan los textos *Rojo* y *Verde* en las propiedades *EstadoSemaforo* de las instancias *MiSemaforo* y *OtroSemaforo*. La propiedad *EstadoSemaforo* de la instancia *SemaforoDeMiCalle* contendrá el valor “Rojo”, con el que se inicializa (línea 2 de la clase *Semaforo*).

En las líneas 10 y 11 se obtienen los valores de la propiedad *EstadoSemaforo*, a través del método *DimeColor()* y se imprimen. En la línea 13 se compara (*equals*) el valor obtenido en *DimeColor()* con el literal “Rojo”.

Cuando, por ejemplo, se invoca a los métodos *PonColor* y *DimeColor* de la instancia *OtroSemaforo*, internamente se produce el siguiente efecto: el literal “Verde” es el argumento en la llamada al método *PonColor*, asociado a la instancia *OtroSemaforo*; el argumento pasa al parámetro *Color* del método, y de ahí a la variable (propiedad) *EstadoSemaforo* (por la asignación de la línea 5 en la clase *Semaforo*). En sentido contrario, cuando se invoca al método *DimeColor()* asociado a la instancia *OtroSemaforo*, el valor de la propiedad *EstadoSemaforo* se devuelve (retorna) a la instrucción llamante en el programa principal (línea 9 de la clase *Semaforo*).



## 4.1.6 Resultado



```
MS-DOS
C:\CursoJava\Introduccion>java PruebaSemaforo
Verde
Rojo
No Pasar
C:\CursoJava\Introduccion>
```

## 4.2 SOBRECARGA DE MÉTODOS Y CONSTRUCTORES

### 4.2.1 Sobrecarga de métodos

La sobrecarga de métodos es un mecanismo muy útil que permite definir en una clase varios métodos con el mismo nombre. Para que el compilador pueda determinar a qué método nos referimos en un momento dado, los parámetros de los métodos sobrecargados no pueden ser idénticos.

Por ejemplo, para establecer las dimensiones de un objeto (anchura, profundidad, altura) en una medida dada (“pulgadas”, “centímetros”, ...) podemos definir los métodos:

```
Dimensiones(double Ancho, double Alto, double Profundo, String Medida)
Dimensiones(String Medida, double Ancho, double Alto, double Profundo)
Dimensiones(double Ancho, String Medida, double Alto, double Profundo)
Dimensiones(double Ancho, double Alto, String Medida, double Profundo)
```

Cuando realicemos una llamada al método *Dimensiones(...)*, el compilador podrá determinar a cual de los métodos nos referimos por la posición del parámetro de tipo *String*. Si definiéramos el siguiente nuevo método sobrecargado el compilador no podría determinar a qué método nos referimos al intentar resolver la llamada

```
Dimensiones(double Alto, double Ancho, double Profundo, String Medida)
```

Un método se determina por su firma. La firma se compone del nombre del método, número de parámetros y tipo de parámetros (por orden de colocación). De los 5 métodos sobrecargados que hemos definido, el primero y el último presentan la misma firma, por lo que el compilador generará un error al compilar la clase.

Como se ha mencionado, los métodos sobrecargados pueden contener distinto número de parámetros:

### Dimensiones(String Medida)

*Dimensiones(double Ancho, double Alto, double Profundo)*

Los últimos dos métodos definidos son compatibles con todos los anteriores y tendrían sentido si suponemos dos métodos adicionales que los complementen:

*Dimensiones3D(double Ancho, double Alto, double Profundo)*

*TipoMedida(String Medida)*

### 4.2.2 Ejemplo

[illegible]

```
33     Dimensiones(Ancho,Alto,Profundo,Medida);
34 }
35
36 public void Dimensiones(String Medida) {
37     TipoMedida(Medida);
38 }
39
40 public void Dimensiones(double Ancho, double Alto,
41                         double Profundo) {
42     Dimensiones3D(Ancho,Alto,Profundo);
43 }
44
45 public double DimeAncho() {
46     return X;
47 }
48
49 public double DimeAlto() {
50     return Y;
51 }
52
53 public double DimeProfundo() {
54     return Z;
55 }
56
57 public String DimeMedida() {
58     return TipoMedida;
59 }
60 } // Fin de la clase Objeto3D
```

En las líneas 2, 3, 4 y 5 se declaran y definen valores iniciales para las propiedades privadas *X*, *Y*, *Z* y *TipoMedida*. En la línea 7 se define el método *Dimensiones3D*, que permite asignar valores a las tres dimensiones espaciales de un objeto. En la línea 11 se define el método *TipoMedida*, que permite asignar un valor a la propiedad del mismo nombre.

La línea 15 define el primer método del grupo de 6 métodos sobrecargados *Dimensiones*. El cuerpo de este método (líneas 17 y 18) aprovecha la existencia de los métodos anteriores, para evitar la repetición de código. Los 3 métodos *Dimensiones* siguientes (líneas 21, 26 y 31) simplemente hacen una llamada al primero, ordenando adecuadamente los argumentos de la invocación.

Los dos últimos métodos sobrecargados *Dimensiones* (líneas 36 y 40) hacen llamadas a los métodos más convenientes para realizar su función.

Los últimos 4 métodos (*DimeAlto*, *DimeAncho*, *DimeProfundo*, *DimeMedida*) nos permiten conocer el valor de las propiedades de la clase, aumentando la funcionalidad de *Objeto3D*.

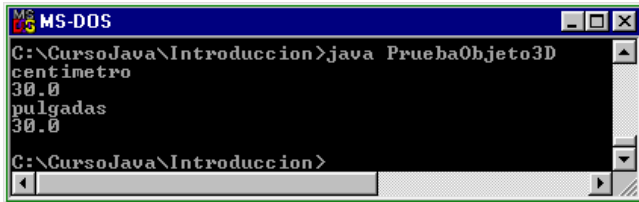
A continuación se muestra el código de una clase (*PruebaObjeto3D*) que utiliza a la clase *Objeto3D*:

```
1 public class PruebaObjeto3D {
2     public static void main (String[] args) {
3         Objeto3D Caja = new Objeto3D();
4         Objeto3D Esfera = new Objeto3D();
5         Objeto3D Bicicleta = new Objeto3D();
6
7         Caja.Dimensiones(20.0,12.5,30.2,"centimetros");
8         Esfera.Dimensiones(10.0,"pulgadas",10.0,10.0);
9         Bicicleta.Dimensiones(90.0,30.0,20.0);
10
11        System.out.println(Bicicleta.DimeMedida());
12        System.out.println(Bicicleta.DimeAlto());
13
14        Bicicleta.Dimensiones("pulgadas");
15
16        System.out.println(Bicicleta.DimeMedida());
17        System.out.println(Bicicleta.DimeAlto());
18    }
19 }
20 }
```

En las líneas 3, 4 y 5 se declaran y definen tres instancias (*Caja*, *Esfera*, *Bicicleta*) de la clase *Objeto3D*. En las líneas 7, 8 y 9 se invocan diversas ocurrencias del método sobrecargado *Dimensiones*. Como en la instancia *Bicicleta* no se define el tipo de sus medidas, prevalece “centímetro” que ha sido asignada en la instrucción 5 de la clase *Objeto3D*.

Las líneas 11 y 12 imprimen la medida y altura de la instancia *Bicicleta* (esperamos “centímetro” y 30.0). En la línea 14 se varía el tipo de medida empleada, lo que se reflejará en la línea 16.

### 4.2.3 Resultado



```
MS-DOS
C:\CursoJava\Introduccion>java PruebaObjeto3D
centimetro
30.0
pulgadas
30.0
C:\CursoJava\Introduccion>
```

### 4.2.4 Constructores

Los constructores son métodos que nos sirven para iniciar los objetos al definirse las instancias de los mismos. Habitualmente, el cometido de los constructores es asignar valores iniciales a las propiedades de la clase, es decir, situar a la clase instanciada en un estado concreto.

La sintaxis de los constructores es la misma que la de los métodos, salvo que no tienen la referencia del atributo de acceso y nunca devuelven ningún valor (tampoco se pone la palabra reservada *void*), además su nombre debe coincidir con el nombre de la clase.

Los constructores suelen estar sobrecargados, para permitir más posibilidades de inicialización de las instancias de las clases.

Los constructores nos permiten, a la vez, crear instancias y establecer el estado inicial de cada objeto instanciado, a diferencia de lo que hemos realizado en el ejercicio anterior, donde primero debíamos instanciar los objetos y posteriormente, en otras instrucciones, establecer su estado inicial.

El ejemplo anterior, utilizando constructores, nos quedaría de la siguiente manera:

```
1 public class Objeto3DConConstructor {
2     private double X = 0d;
3     private double Y = 0d;
4     private double Z = 0d;
5     private String TipoMedida = "centimetro";
6
7     public void Dimensiones3D(double Ancho, double Alto,
8                             double Profundo) {
9         X = Ancho; Y = Alto; Z = Profundo;
10    }
```

```
9      }
10
11     public void TipoMedida(String Medida) {
12         TipoMedida = Medida;
13     }
14
15     Objeto3DConConstructor(double Ancho, double Alto,
16                             double Profundo, String Medida) {
17         Dimensiones3D(Ancho,Alto,Profundo);
18         TipoMedida(Medida);
19     }
20
21     Objeto3DConConstructor(String Medida, double Ancho,
22                             double Alto, double Profundo) {
23         this(Ancho,Alto,Profundo,Medida);
24     }
25
26     Objeto3DConConstructor(double Ancho, String Medida,
27                             double Alto, double Profundo) {
28         this(Ancho,Alto,Profundo,Medida);
29     }
30
31     Objeto3DConConstructor(double Ancho, double Alto,
32                             String Medida, double Profundo) {
33         this(Ancho,Alto,Profundo,Medida);
34     }
35
36     Objeto3DConConstructor(String Medida) {
37         TipoMedida(Medida);
38     }
39
40     Objeto3DConConstructor(double Ancho, double Alto,
41                             double Profundo) {
42         Dimensiones3D(Ancho,Alto,Profundo);
43     }
44
45     public double DimeAncho() {
46         return X;
47     }
48
49     public double DimeAlto() {
50         return Y;
51     }
52
53     public double DimeProfundo() {
54         return Z;
55     }
```



```
56     public String DimeMedida() {
57         return TipoMedida;
58     }
59
60 } // Fin de la clase Objeto3DConConstructor
```

Las líneas 15, 21, 26, 31, 36 y 40 definen los constructores de la clase. Como se puede observar, se omite la palabra *void* en su definición; tampoco se pone el atributo de acceso. Por lo demás, se codifican como métodos normales, salvo el uso de la palabra reservada *this*: *this* se refiere a “esta” clase (en nuestro ejemplo *Objeto3DConConstructor*). En el ejemplo indicamos que se invoque a los constructores de la clase cuya firma coincida con la firma de las instrucciones llamantes.

La diferencia entre los constructores definidos y los métodos *Dimensiones* de la clase *Objeto3D* se aprecia mejor en las instanciaciones de los objetos:

```
1  public class PruebaObjeto3DConConstructor {
2      public static void main (String[] args) {
3
4          Objeto3DConConstructor Caja = new
5              Objeto3DConConstructor(20.0,12.5,30.2,"centimetros");
6
7          Objeto3DConConstructor Esfera = new
8              Objeto3DConConstructor(10.0,"pulgadas",10.0,10.0);
9
10         Objeto3DConConstructor Bicicleta = new
11             Objeto3DConConstructor(90.0,30.0,20.0);
12
13         System.out.println(Bicicleta.DimeMedida());
14         System.out.println(Bicicleta.DimeAlto());
15
16         Bicicleta.TipoMedida("pulgadas");
17
18         System.out.println(Bicicleta.DimeMedida());
19         System.out.println(Bicicleta.DimeAlto());
20
21     }
22 }
```

En las líneas 4 y 5 se declara una instancia *Caja* de la clase *Objeto3DConConstructor*. La instancia se inicializa utilizando el constructor cuya firma es: *Objeto3DConConstructor(double,double,double,String)*.

En las líneas 7 y 8 se declara una instancia *Esfera* de la clase *Objeto3DConConstructor*. La instancia se inicializa utilizando el constructor cuya firma es: *Objeto3DConConstructor(double,String,double,double)*.

En las líneas 10 y 11 se declara una instancia *Bicicleta* de la clase *Objeto3DConConstructor*. La instancia se inicializa utilizando el constructor cuya firma es: *Objeto3DConConstructor(double,double,double)*.

## 4.3 EJEMPLOS

En esta lección se presentan tres ejemplos: “figura genérica”, “agenda de teléfono” y “ejercicio de logística”, en los que se implementan clases que engloban las propiedades y métodos necesarios para facilitar la resolución de los problemas planteados.

### 4.3.1 Figura genérica

Este primer ejemplo muestra una clase *Figura*, en la que se puede establecer y consultar el color y la posición del centro de cada instancia de la clase.

En la línea 4 se declara la propiedad *ColorFigura*, de tipo *Color*. *Color* es una clase de Java (que se importa en la línea 1). En la línea 5 se declara el vector (matriz lineal) *Posición*, que posee dos componentes: *Posición[0]* y *Posición[1]*, que representan respectivamente al valor X e Y de la posición del centro de la figura.

En la línea 7 se define un constructor de la clase en el que se puede establecer el color de la figura. Este método hace una llamada a *EstableceColor* (línea 16) en donde se actualiza la propiedad *ColorFigura* con el valor del parámetro *color*.

En la línea 11 se define un segundo constructor en el que se establece el color y la posición del centro de la figura. Este constructor hace una llamada al método *EstableceCentro* (línea 24) en donde se actualiza la propiedad *Posición* de la clase (*this*) con el valor del parámetro *Posición*.

Para completar la clase *Figura*, se establecen los métodos de acceso *DimeColor* (línea 20) y *DimeCentro* (línea 29), a través de los cuales se puede obtener los valores de las propiedades *ColorFigura* y *Posición*. Nótese que desde el exterior de esta clase no se puede acceder directamente a sus propiedades, que tienen atributo de acceso *private*.

## Código

```
1  import java.awt.Color;
2
3  public class Figura {
4      private Color ColorFigura;
5      private int[] Posicion = new int[2];
6
7      Figura(Color color) {
8          EstableceColor(color);
9      }
10
11     Figura(Color color, int[] Posicion) {
12         EstableceColor(color);
13         EstableceCentro(Posicion);
14     }
15
16     public void EstableceColor(Color color) {
17         ColorFigura = color;
18     }
19
20     public Color DimeColor() {
21         return ColorFigura;
22     }
23
24     public void EstableceCentro(int[] Posicion) {
25         this.Posicion[0] = Posicion[0];
26         this.Posicion[1] = Posicion[1];
27     }
28
29     public int[] DimeCentro() {
30         return Posicion;
31     }
32
33 }
```

### 4.3.2 Agenda de teléfono

En este ejemplo se implementa el control que puede tener un teléfono para mantener las últimas llamadas realizadas y para poder acceder a las mismas.

#### Código

```
1 public class Telefono {
2     private int Max_Llamadas;
3     private String[] LlamadasHechas;
4
5     private int NumLlamadaHecha = -1;
6
7     Telefono(int Max_Llamadas) {
8         this.Max_Llamadas = Max_Llamadas;
9         LlamadasHechas = new String[Max_Llamadas];
10    }
11
12    public void Llamar(String Numero) {
13        // Hacer la llamada
14        NumLlamadaHecha = (NumLlamadaHecha+1)%Max_Llamadas;
15        LlamadasHechas[NumLlamadaHecha] = Numero;
16    }
17
18    public String UltimaLlamada() {
19        return Llamada(0);
20    }
21
22    public String Llamada(int n) { // La ultima llamada es n=0
23        if (n<=NumLlamadaHecha)
24            return LlamadasHechas[NumLlamadaHecha-n];
25        else
26            return LlamadasHechas[Max_Llamadas-(n-NumLlamadaHecha)];
27    }
28
29 }
```

En la línea 2 de la clase *Telefono* se declara una propiedad *Max\_Llamadas*, donde se guardará el número de llamadas que el teléfono almacena. En la línea siguiente se declara la matriz lineal (vector) de literales que contendrá los últimos *Max\_Llamadas* teléfonos marcados.

El constructor de la línea 7 permite definir el número de llamadas que almacena el teléfono. En la línea 8 se presenta una idea importante: cuando se utiliza

la estructura *this.Propiedad*, el código se refiere a la variable *Propiedad* de la clase (*this*). En nuestro ejemplo, en la línea 8, *this.Max\_Llamadas* hace referencia a la propiedad privada *Max\_Llamadas* de la clase *Telefono*, mientras que *Max\_Llamadas* hace referencia al parámetro del constructor.

En la línea 9 se crea y dimensiona el vector *LlamadasHechas*. Nótese que en esta implementación, para crear un registro de llamadas, hay que instanciar el objeto *Telefono* haciendo uso del constructor definido.

El método de la línea 12 introduce el *Numero* “marcado” en el vector *LlamadasHechas*. Para asegurarnos de que se almacenan las últimas *Max\_Llamadas* hacemos uso de una estructura de datos en forma de buffer circular, esto es, si por ejemplo *Max\_Llamadas* es 4, rellenaremos la matriz con la siguiente secuencia: *LlamadasHechas*[0], *LlamadasHechas*[1], *LlamadasHechas*[2], *LlamadasHechas*[3], *LlamadasHechas*[0], *LlamadasHechas*[1], etc. Este efecto 0, 1, 2, 3, 0, 1, ... lo conseguimos con la operación módulo (%) *Max\_Llamadas*, como se puede ver en la línea 14.

En el método *Llamar*, la propiedad *NumLlamadaHecha* nos sirve de apuntador a la posición de la matriz que contiene el número de teléfono correspondiente a la última llamada realizada.

El método situado en la línea 22 devuelve el número de teléfono al que llamamos en último lugar (*n=0*), penúltimo lugar (*n=1*), etc.

El método definido en la línea 18 (*UltimaLlamada*) devuelve el último número de teléfono llamado.

Para utilizar la clase *Telefono* se ha implementado la clase *PruebaTelefono*. En primer lugar se declaran y definen dos teléfonos (*ModeloBarato* y *ModeloMedio*) con capacidades de almacenamiento de llamadas 2 y 4 (líneas 3 y 4); posteriormente se realizan una serie de llamadas con *ModeloBarato*, combinadas con consultas de las mismas a través de los métodos *UltimaLlamada* y *Llamada*.

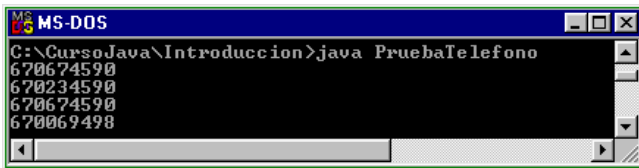
A partir de la línea 13 se hace uso del teléfono *ModeloMedio*, realizándose 6 llamadas y consultando en la línea 15 la última llamada y en la línea 20 la penúltima llamada.

## Código de prueba

```
1 public class PruebaTelefono {
2     public static void main(String[] args) {
```

```
3     Telefono ModeloBarato = new Telefono(2);
4     Telefono ModeloMedio = new Telefono(4);
5
6     ModeloBarato.Llamar("670879078");
7     ModeloBarato.Llamar("670674590");
8     System.out.println(ModeloBarato.UltimaLlamada());
9     ModeloBarato.Llamar("670234590");
10    ModeloBarato.Llamar("670069423");
11    System.out.println(ModeloBarato.Llamada(1));
12
13    ModeloMedio.Llamar("670879078");
14    ModeloMedio.Llamar("670674590");
15    System.out.println(ModeloMedio.UltimaLlamada());
16    ModeloMedio.Llamar("670234590");
17    ModeloMedio.Llamar("670069423");
18    ModeloMedio.Llamar("670069498");
19    ModeloMedio.Llamar("670069499");
20    System.out.println(ModeloMedio.Llamada(1));
21
22 }
23 }
```

## Resultados



### 4.3.3 Ejercicio de logística

En este ejemplo se plantea la siguiente situación: una empresa dispone de 3 almacenes de grandes contenedores. El primer almacén tiene una capacidad de 2 contenedores, el segundo de 4 y el tercero de 8. El primero se encuentra muy cerca de una vía (carretera) principal, el segundo se encuentra a 10 kilómetros de la carretera principal y el tercero a 20 kilómetros.

Los camioneros o bien llegan con un contenedor (uno solo por camión) o bien llegan con el camión vacío con la intención de llevarse un contenedor. En

cualquier caso, siempre ha existido un vigilante al comienzo del camino que lleva a los almacenes que le indicaba a cada camionero a que almacén debía dirigirse a depositar el contenedor que traía o a recoger un contenedor, en caso de llegar sin carga.

El vigilante, con muy buena lógica, siempre ha indicado a los camioneros el almacén más cercano donde podían realizar la operación de carga o descarga, evitando de esta manera largos trayectos de ida y vuelta a los almacenes más lejanos cuando estos desplazamientos no eran necesarios.

Como el buen vigilante está a punto de jubilarse, nos encargan la realización de un programa informático que, de forma automática, le indique a los camioneros el almacén al que deben dirigirse minimizando los costes de combustible y tiempo. Los camioneros, una vez que llegan a la barrera situada al comienzo del camino, pulsan un botón ('m') si van a meter un contenedor, o un botón 's' si lo van a sacar. El programa les indicará en un panel el almacén (1, 2 ó 3) al que se deben dirigir.

Para resolver esta situación, utilizaremos dos clases: una a la que llamaremos *LogisticaAlmacen*, que permitirá la creación de una estructura de datos almacén y sus métodos de acceso necesarios. Posteriormente crearemos la clase *LogisticaControlContenedor* que utilizando la primera implementa la lógica de control expuesta en el enunciado del ejercicio.

La clase *LogísticaAlmacen* puede implementarse de la siguiente manera:

## Código

```
1 public class LogisticaAlmacen {
2     private byte Capacidad;
3     private byte NumeroDeHuecos;
4
5     LogisticaAlmacen(byte Capacidad) {
6         this.Capacidad = Capacidad;
7         NumeroDeHuecos = Capacidad;
8     }
9
10    public byte DimeNumeroDeHuecos() {
11        return (NumeroDeHuecos);
12    }
13
14    public byte DimeCapacidad() {
15        return (Capacidad);
16    }
17 }
```

```
18     public boolean HayHueco() {
19         return  (NumeroDeHuecos != 0);
20     }
21
22     public boolean HayContenedor() {
23         return (NumeroDeHuecos != Capacidad);
24     }
25
26     public void MeteContenedor() {
27         NumeroDeHuecos--;
28     }
29
30     public void SacaContenedor() {
31         NumeroDeHuecos++;
32     }
33
34 } // LogisticaAlmacen
```

La clase *LogisticaAlmacen* es muy sencilla y muy útil. El estado del almacén puede definirse con dos propiedades: *Capacidad* y *NumeroDeHuecos* (líneas 2 y 3). En este caso hemos declarado las variables de tipo *byte*, por lo que la capacidad máxima de estos almacenes es de 256 elementos. Si quisiéramos una clase almacén menos restringida, podríamos cambiar el tipo de las variables a *short* o *int*.

Sólo hemos programado un constructor (en la línea 5), que nos permite definir la capacidad (*Capacidad*) del almacén. En la línea 6 asignamos a la propiedad *Capacidad* de la clase (*this.Capacidad*) el valor que nos indica el parámetro *Capacidad* del constructor. En la línea 7 determinamos que el almacén, inicialmente se encuentra vacío (tantos huecos como capacidad).

Los métodos 10 y 14 nos devuelven (respectivamente) los valores de las propiedades *NumeroDeHuecos* y *Capacidad*.

El método de la línea 18 (*HayHueco*) nos indica si tenemos la posibilidad de meter un elemento en el almacén, es decir *NumeroDeHuecos* es distinto de 0. El método de la línea 22 (*HayContenedor*) nos indica si existe al menos un elemento en el almacén, es decir, no hay tantos huecos como capacidad.

Finalmente, los métodos *MeteContenedor* y *SacaContenedor* (líneas 26 y 30) actualizan el valor de la propiedad *NumeroDeHuecos*. Obsérvese que estos métodos no realizan ninguna comprobación de si el almacén está lleno (en el primer caso) o vacío (en el segundo), esta comprobación la deberá realizar el programador que utilice la clase, invocando a los métodos *HayHueco* y *HayContenedor* (respectivamente).



A continuación se muestra el código de la clase que realiza el control de acceso a los almacenes, utilizando la clase *LogisticaAlmacen*:

## Código de prueba

```
1 public class LogisticaControlContenedor {
2     public static void main(String[] args){
3         LogisticaAlmacen Almacen1 = new
4             LogisticaAlmacen((byte)2);
5         LogisticaAlmacen Almacen2 = new
6             LogisticaAlmacen((byte)4);
7         LogisticaAlmacen Almacen3 = new
8             LogisticaAlmacen((byte)8);
9
10        String Accion;
11
12        do {
13            Accion = Teclado.Lee_String();
14            if (Accion.equals("m")) // meter contenedor
15                if (Almacen1.HayHueco())
16                    Almacen1.MeteContenedor();
17                else
18                    if (Almacen2.HayHueco())
19                        Almacen2.MeteContenedor();
20                    else
21                        if (Almacen3.HayHueco())
22                            Almacen3.MeteContenedor();
23                    else
24                        System.out.println("Hay que esperar a que
25                            vengán a quitar un contenedor");
26            else // sacar contenedor
27                if (Almacen1.HayContenedor())
28                    Almacen1.SacaContenedor();
29                else
30                    if (Almacen2.HayContenedor())
31                        Almacen2.SacaContenedor();
32                    else
33                        if (Almacen3.HayContenedor())
34                            Almacen3.SacaContenedor();
35                    else
36                        System.out.println("Hay que esperar a que
37                            vengán a poner un contenedor");
38            } while (!Accion.equals("Salir"));
39        }
40    } // clase
```

En las líneas 3, 4 y 5 se declaran e instancian los almacenes *Almacen1*, *Almacen2* y *Almacen3*, con capacidades 2, 4 y 8.

En la línea 9 se entra en un bucle, que normalmente sería infinito: *while* (*true*), aunque ha sido programado para terminar cuando se teclea el literal “Salir” (línea 35). En este bucle se está esperando a que el primer camionero que llegue pulse el botón “m” o el botón “s”, en nuestro caso que pulse la tecla “m” o cualquier otra tecla (líneas 10, 11 y 23).

Si se pulsa la tecla “m”, con significado “meter contenedor”, en primer lugar se pregunta si *HayHueco* en el *Almacen1* (línea 12), si es así se le indica al camionero que se dirija al primer almacén y se actualiza el estado del almacén invocando al método *MeteContenedor* (línea 13). Si no hay hueco en el primer almacén (línea 14), se “prueba” suerte con *Almacen2* (línea 15); en el caso de que haya hueco se mete el contenedor en este almacén (línea 16). Si no hay hueco en el almacén 2 se intenta en el tercer y último almacén (línea 18).

El tratamiento para sacar un contenedor (líneas 24 a 34) es análogo al de meter el contenedor.

## 4.4 CLASES UTILIZADAS COMO PARÁMETROS

Utilizar una clase como parámetro en un método hace posible que el método utilice toda la potencia de los objetos de java, independizando las acciones realizadas de los objetos (clases) sobre las que las realiza. Por ejemplo, podríamos escribir un método, llamado *Controllgnicion* (*Cohete MiVehiculoEspacial*), donde se realicen complejas operaciones sobre el componente software (clase) *MiVehiculoEspacial*, de la clase *Cohete* que se le pasa como parámetro.

En el ejemplo anterior, el mismo método podría simular el control de la ignición de distintos cohetes, siempre que estos estén definidos como instancias de la clase *Cohete*:

```
Cohete Pegasus, Ariane5;
.....
Controllgnicion(Ariane5);
Controllgnicion(Pegasus);
```

Como ejemplo de clases utilizadas como parámetros, resolveremos la siguiente situación:

Una empresa se encarga de realizar el control informático de la entrada-salida de vehículos en diferentes aparcamientos. Cada aparcamiento dispone de un número fijado de plazas y también de puertas de entrada/salida de vehículos.

Se pide realizar el diseño de un software orientado a objetos que controle los aparcamientos. Para simplificar no consideraremos peticiones de entrada-salida simultáneas (concurrentes).

#### 4.4.1 Código

En primer lugar se define una clase *Almacen*, con la misma funcionalidad que la clase *LogisticaAlmacen* explicada en la lección anterior:

```
1 public class Almacen {
2     private short Capacidad;
3     private short NumeroDeElementos = 0;
4
5     Almacen(short Capacidad) {
6         this.Capacidad = Capacidad;
7     }
8
9     public short DimeNumeroDeElementos() {
10         return (NumeroDeElementos);
11     }
12
13     public short DimeCapacidad() {
14         return (Capacidad);
15     }
16
17     public boolean HayElemento() {
18         return (NumeroDeElementos != 0);
19     }
20
21     public boolean HayHueco() {
22         return (NumeroDeElementos != Capacidad);
23     }
24
25     public void MeteElemento() {
26         NumeroDeElementos++;
27     }
28
29     public void SacaElemento() {
30         NumeroDeElementos--;
31     }
```

```
32
33     public void RellenaAlmacen() {
34         NumeroDeElementos = Capacidad;
35     }
36
37 } // clase
```

Puesto que cada aparcamiento que gestiona la empresa puede tener un número diferente de accesos (puertas), se define la clase *Puerta*. La clase *Puerta*, además del constructor, tiene únicamente dos métodos: *EntraVehiculo* y *SaleVehiculo*, donde se evalúan las peticiones de entrada y salida de los usuarios.

La clase *Puerta*, para ser útil, tiene que poder actuar sobre cada uno de los diferentes aparcamientos (para nosotros “almacenes” de vehículos). No nos interesa tener que implementar una clase *Puerta* por cada aparcamiento que gestiona la empresa.

Para conseguir que las acciones de los métodos implementados en la clase *Puerta* se puedan aplicar a los aparcamientos (almacenes) deseados, le pasaremos (como parámetro) a la clase *Puerta* la clase *Almacen* sobre la que tiene que actuar. El constructor situado en la línea 5 de la clase *Puerta* admite la clase *Almacen* como parámetro. La referencia del almacén suministrado como argumento se copia en la propiedad *Parking* de la clase *Puerta* (*this.Parking*).

Los métodos *EntraVehiculo* y *SaleVehiculo* (líneas 9 y 19) utilizan el almacén pasado como parámetro (líneas 10, 13 y 23). Estos métodos se apoyan en las facilidades que provee la clase *Almacen* para implementar la lógica de tratamiento de las peticiones de entrada y salida del aparcamiento realizadas por los usuarios.

```
1  public class Puerta {
2
3      Almacen Parking = null;
4
5      Puerta (Almacen Parking) {
6          this.Parking = Parking;
7      }
8
9      public void EntraVehiculo() {
10         if (Parking.HayHueco()) {
11             System.out.println ("Puede entrar");
12             // Abrir la barrera
13             Parking.MeteElemento();
14         }
15         else
```

```
16         System.out.println ("Aparcamiento completo");
17     }
18
19     public void SaleVehiculo() {
20         // Comprobar el pago
21         System.out.println ("Puede salir");
22         // Abrir la barrera
23         Parking.SacaElemento();
24     }
25 }
```

Finalmente, la clase *Aparcamiento* recoge las peticiones de entrada/salida de los usuarios por cada una de las puertas. Esto se simula por medio del teclado:

```
1  class Aparcamiento {
2      public static void main(String[] args){
3          char CPuerta, COperacion;
4          Puerta PuertaRequerida = null;
5
6          Almacen Aparcamiento = new Almacen( (short) 5 );
7          Puerta Puerta1 = new Puerta(Aparcamiento);
8          Puerta Puerta2 = new Puerta(Aparcamiento);
9
10         do {
11             CPuerta = IntroduceCaracter ("Puerta de acceso:
12                                     (1, 2): ");
13
14             switch (CPuerta) {
15                 case '1':
16                     PuertaRequerida = Puerta1;
17                     break;
18                 case '2':
19                     PuertaRequerida = Puerta2;
20                     break;
21                 default:
22                     System.out.println ("Puerta seleccionada no
23                                     valida");
24                     break;
25             }
26
27             COperacion = IntroduceCaracter ("Entrar/Salir
28                                     vehiculo (e, s): ");
29
30             switch (COperacion) {
31                 case 'e':
32                     PuertaRequerida.EntraVehiculo();
33                     break;
34                 case 's':
35                     PuertaRequerida.SaleVehiculo();
36             }
37         }
38     }
39 }
```

```
31             break;
32         default:
33             System.out.println ("Operacion seleccionada
                                   no valida");
34             break;
35     }
36
37     } while (true);
38
39 } // main
40
41
42 static public char IntroduceCaracter (String Mensaje) {
43     String Entrada;
44
45     System.out.print (Mensaje);
46     Entrada = Teclado.Lee_String();
47     System.out.println();
48     Entrada = Entrada.toLowerCase();
49     return Entrada.charAt(0);
50 }
51
52 }
```

En la línea 6 se declara el aparcamiento (*Almacen*) sobre el que se realiza la prueba de funcionamiento. En las líneas 7 y 8 se establece que este aparcamiento dispondrá de dos accesos: *Puerta1* y *Puerta2*, de tipo *Puerta*. Obsérvese como a estas dos puertas se les pasa como argumento el mismo *Aparcamiento*, de tipo *Almacen*. Con esto conseguiremos que las entradas y salidas de los vehículos se contabilicen en la propiedad *NumeroDeElementos* de una sola clase *Almacen*.

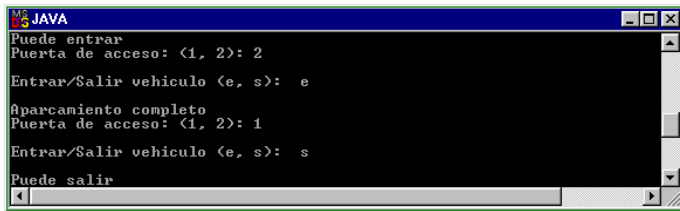
En la línea 10 nos introducimos en un bucle sin fin, donde pedimos a la persona que prueba esta simulación que introduzca el número de puerta del aparcamiento por donde desea entrar o salir un usuario. Para ello nos ayudamos del método *IntroduceCaracter* implementado a partir de la línea 42.

Si el valor introducido es un 1 ó un 2, se actualiza la propiedad *PuertaRequerida*, de tipo *Puerta*, con la referencia a la puerta correspondiente (*Puerta1* o *Puerta2*). Si el valor introducido es diferente a 1 y a 2, se visualiza un mensaje de error (línea 20).

A continuación se vuelve a emplear el método *IntroduceCaracter* para saber si se desea simular una entrada o bien una salida. Si es una entrada se invoca al método *EntraVehiculo* de la clase *Puerta*, a través de la referencia *PuertaRequerida*

(que sabemos que apunta o bien a la instancia *Puerta1*, o bien a la instancia *Puerta2*). Si el usuario desea salir, se invoca al método *SalVehiculo*.

## 4.4.2 Resultados



## 4.5 PROPIEDADES Y MÉTODOS DE CLASE Y DE INSTANCIA

En una clase, las propiedades y los métodos pueden definirse como:

- De instancia
- De clase

### 4.5.1 Propiedades de instancia

Las propiedades de instancia se caracterizan porque cada vez que se define una instancia de la clase, se crean físicamente una nuevas variables que contendrán los valores de dichas propiedades en la instancia creada. Es decir, cada objeto (cada instancia de una clase) contiene sus propios valores en las propiedades de instancia. Todos los ejemplos de clases que hemos realizado hasta ahora han utilizado variables de instancia.

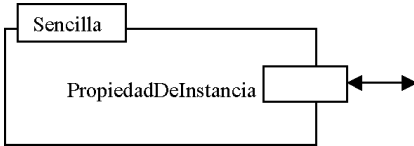
En este punto es importante resaltar el hecho de que hasta que no se crea una primera instancia de una clase, no existirá ninguna propiedad visible de la clase.

Gráficamente lo expuesto se puede representar como:

```

1 class Sencilla {
2     public int PropiedadDeInstancia;
3 }

```



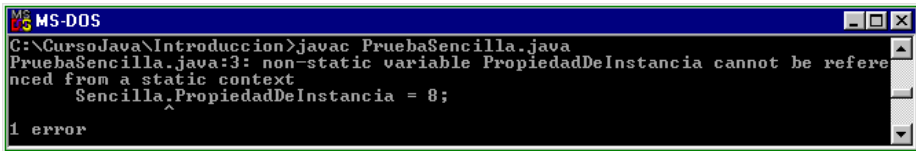
La clase *Sencilla* está definida, pero no instanciada, por lo que todavía no existe ninguna variable *PropiedadDeInstancia*. El gráfico nos muestra únicamente la estructura que tendrá una instancia de la clase *Sencilla* (cuando la definamos).

Si ahora intentásemos hacer uso de la propiedad *PropiedadDeInstancia* a través del nombre de la clase (*Sencilla*), el compilador nos daría un error:

```

1 class PruebaSencilla {
2     public static void main (String[] args) {
3         Sencilla.PropiedadDeInstancia = 8;
4     }
5 }

```



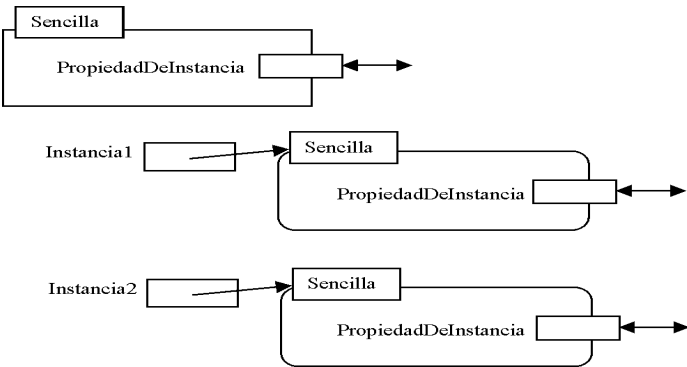
El compilador nos indica que la variable *PropiedadDeInstancia* es “no estática” y que existe un error. Para poder hacer uso de la variable *PropiedadDeInstancia*, obligatoriamente deberemos crear alguna instancia de la clase, tal y como hemos venido haciendo en las últimas lecciones:

```

1 class PruebaSencilla2 {
2     public static void main (String[] args) {
3         Sencilla Instancia1 = new Sencilla();
4         Sencilla Instancia2 = new Sencilla();
5         Instancia1.PropiedadDeInstancia = 8;
6         Instancia2.PropiedadDeInstancia = 5;
7     }
8 }

```





En este caso disponemos de dos propiedades de instancia, a las que podemos acceder como:

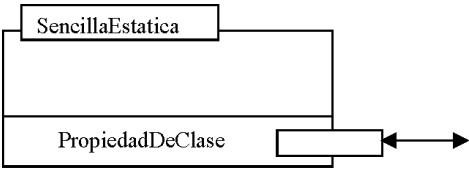
*Instancia1.PropiedadDeInstancia* e *Instancia2.PropiedadDeInstancia*

Todo intento de utilizar directamente la definición de la clase nos dará error:  
~~*Sencilla.PropiedadDeInstancia*~~

4.5.2 Propiedades de clase

Una propiedad de clase (propiedad estática) se declara con el atributo *static*:

```
1 class SencillaEstatica {
2     static public int PropiedadDeClase;
3 }
```



A diferencia de las propiedades de instancia, las propiedades de clase existen incluso si no se ha creado ninguna instancia de la clase. Pueden ser referenciadas directamente a través del nombre de la clase, sin tener que utilizar el identificador de ninguna instancia.

```

1 class PruebaSencillaEstatica {
2     public static void main (String[] args) {
3         SencillaEstatica.PropiedadDeClase = 8;
4     }
5 }

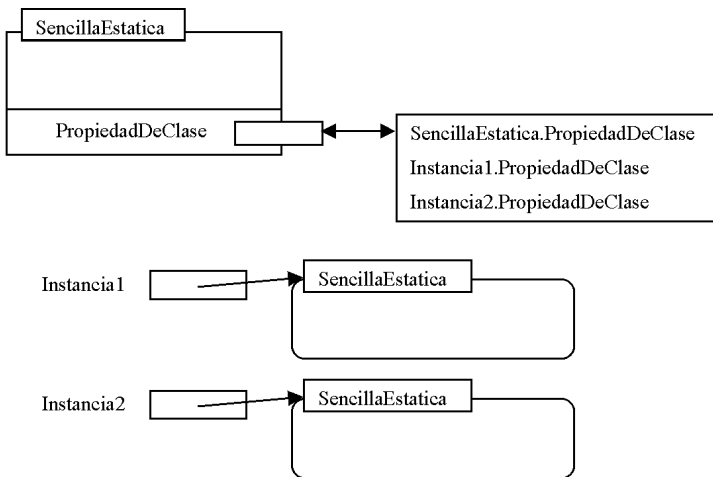
```

Las propiedades de clase son compartidas por todas las instancias de la clase. Al crearse una instancia de la clase, no se crean las variables estáticas de esa clase. Las variables estáticas (de clase) existen antes de la creación de las instancias de la clase.

```

1 class PruebaSencillaEstatica {
2     public static void main (String[] args) {
3         SencillaEstatica Instancial = new SencillaEstatica();
4         SencillaEstatica Instancia2 = new SencillaEstatica();
5         SencillaEstatica.PropiedadDeClase = 4;
6         Instancial.PropiedadDeClase = 8;
7         Instancia2.PropiedadDeClase = 5;
8     }
9 }

```



En el ejemplo anterior, *SencillaEstatica.PropiedadDeClase*, *Instancia1.PropiedadDeClase* e *Instancia2.PropiedadDeClase* hacen referencia a la misma variable (la propiedad estática *PropiedadDeClase* de la clase *SencillaEstatica*)

### 4.5.3 Métodos de instancia

Los métodos de instancia, al igual que las propiedades de instancia, sólo pueden ser utilizados a través de una instancia de la clase. Hasta ahora siempre hemos definido métodos de instancia (salvo el método *main*, que es estático).

La siguiente porción de código (obtenida de un ejemplo ya realizado) muestra el funcionamiento de los métodos de instancia (al que estamos habituados): en primer lugar se declaran y definen las instancias de las clases (líneas 2, 3 y 4) y posteriormente se hace uso de los métodos a través de las instancias (líneas 11 y 12).

Cualquier intento de acceder a un método de instancia a través del nombre de la clase (y no de una instancia de la clase) nos dará error de compilación.

En la línea 9 de la porción de código mostrada, podemos observar como hacemos una llamada a un método estático: utilizamos el método *Lee\_String* de la clase *Teclado*. Esta llamada funciona porque el método *Lee\_String* es estático, si no lo fuera obtendríamos un error de compilación en la línea 9.

```
1      .....
2      LogisticaAlmacen Almacen1 = new
3                                     LogisticaAlmacen((byte)2);
4      LogisticaAlmacen Almacen2 = new
5                                     LogisticaAlmacen((byte)4);
6      LogisticaAlmacen Almacen3 = new
7                                     LogisticaAlmacen((byte)8);
8
9      String Accion;
10
11     do {
12         Accion = Teclado.Lee_String();
13         if (Accion.equals("m")) // meter contenedor
14             if (Almacen1.HayHueco())
15                 Almacen1.MeteContenedor();
16         .....
```

### 4.5.4 Métodos de clase

Un método estático puede ser utilizado sin necesidad de definir previamente instancias de la clase que contiene el método. Los métodos estáticos pueden

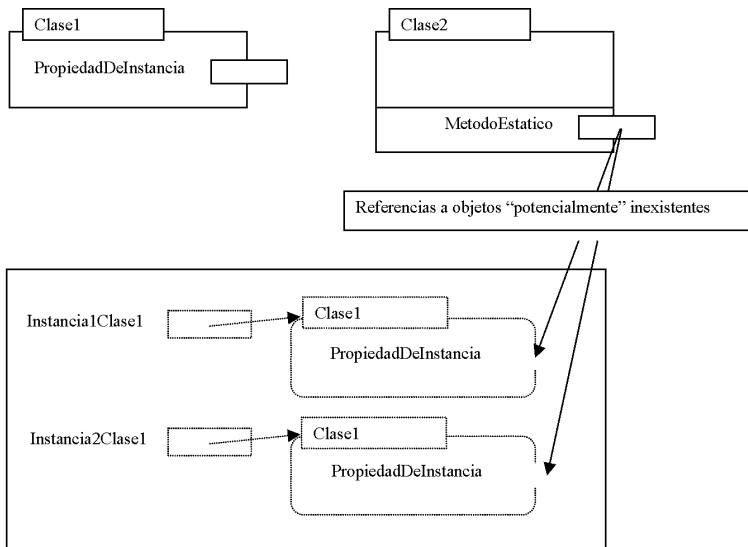
referenciarse a través del nombre de la clase (al igual que las propiedades estáticas). Esta posibilidad es útil en diversas circunstancias:

- Cuando el método proporciona una utilidad general  
Esta situación la hemos comprobado con los métodos de la clase *Math*. Si queremos, por ejemplo, realizar una raíz cuadrada, no nos es necesario crear ninguna instancia del método *Math*; directamente escribimos *Math.sqrt(Valor\_double)*. Esto es posible porque el método *sqrt* de la clase *Math* es estático.
- Cuando el método hace uso de propiedades estáticas u otros métodos estáticos  
Los métodos estáticos referencian propiedades y métodos estáticos.

No es posible hacer referencia a una propiedad de instancia o un método de instancia desde un método estático. Esto es así debido a que en el momento que se ejecuta un método estático puede que no exista ninguna instancia de la clase donde se encuentra la propiedad o el método de instancia al que referencia el método estático.

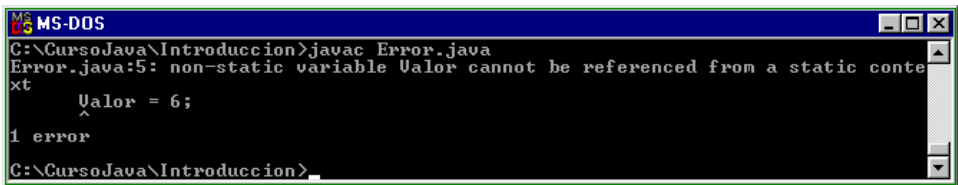
Los compiladores de Java comprueban estas situaciones y producen errores cuando detectan una referencia a un objeto no estático dentro de un método estático.

Gráficamente, lo explicado se puede mostrar de la siguiente manera:



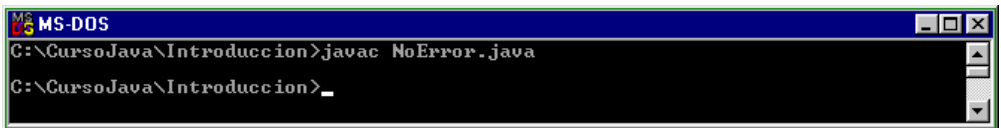
Una situación muy común cuando se empieza a programar es realizar una referencia a una propiedad no estática desde el método estático *main*. El siguiente ejemplo caracteriza este tipo de error tan extendido:

```
1 class Error {
2     int Valor = 8;
3
4     public static void main(String[] args){
5         Valor = 6;
6     } // main
7
8 } // clase
```



Si se quiere referenciar a la propiedad *Valor* de la clase, es necesario que esté declarada como estática:

```
1 class NoError {
2     static int Valor = 8;
3
4     public static void main(String[] args){
5         Valor = 6;
6     } // main
7
8 } // clase
```



#### 4.5.5 Ejemplo que utiliza propiedades de clase

Vamos a realizar el control de una votación en la que se puede presentar un número cualquiera de candidatos. En cada momento se puede votar a cualquier candidato y se pueden pedir los siguientes datos:

- Nombre de un candidato concreto y el número de votos que lleva hasta el momento
- Nombre del candidato más votado hasta el momento y número de votos que lleva conseguidos

La solución desarrollada parte de una clase *Votacion*, que permite almacenar el nombre de un candidato y el numero de votos que lleva, además de los métodos necesarios para actualizar el estado del objeto. Si instanciamos la clase 14 veces, por ejemplo, podremos llevar el control de votos de 14 candidatos.

La cuestión que ahora se nos plantea es: ¿Cómo contabilizar el número de votos y almacenar el nombre del candidato más votado hasta el momento? Una solución posible es crear una nueva clase “MasVotado” que se instancie una sola vez y contenga propiedades para almacenar estos valores, junto a métodos para consultar y actualizar los mismos.

La solución expuesta en el párrafo anterior funcionaría correctamente y no requiere del uso de propiedades estáticas, aunque existe una cuestión de diseño que hay que tener clara: la clase “MasVotado” tiene sentido si se instancia una sola vez.

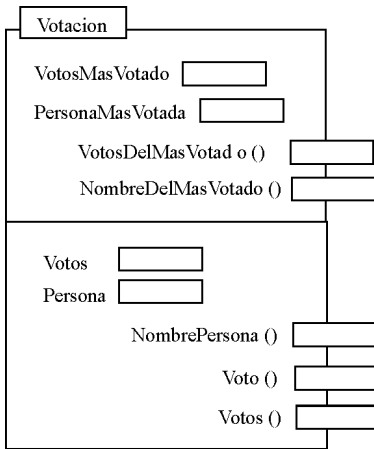
La solución que se aporta en este apartado no requiere del uso de una nueva clase “MasVotado” o similar, ni necesita un número fijo de instanciaciones para funcionar. La solución propuesta contiene las propiedades y métodos de acceso a la persona más votada dentro de la propia clase *Votacion*, en la que se vota a cada persona.

Como el nombre y número de votos de la persona mas votada hasta el momento es una información general, que no depende únicamente de los votos de un candidato, sino de los votos recibidos por todos los candidatos, estas propiedades deben ser accesibles, comunes y compartidas por todos. Estas variables deben ser estáticas (de clase).

#### 4.5.6 Código del ejemplo

```
1 class Votacion {
2     // Persona a la que se vota en esta instancia y el numero
3     // de votos que lleva
4     private String Persona = null;
5     private int Votos = 0;
6
7     // Persona mas votada de todas las instancias y el numero
```

```
8 // de votos que lleva
9 static private int VotosMasVotado = 0;
10 static private String PersonaMasVotada = null;
11
12 // Constructor
13 Votacion (String Persona) {
14     this.Persona = Persona;
15 }
16
17 // Se invoca cada vez que alguien vota a Persona
18 public void Voto() {
19     Votos++;
20     if (Votos > VotosMasVotado) {
21         PersonaMasVotada = Persona;
22         VotosMasVotado = Votos;
23     }
24 }
25
26 // Devuelve el nombre de Persona
27 public String NombrePersona() {
28     return Persona;
29 }
30
31 // Devuelve el numero de votos de Persona
32 public int Votos() {
33     return Votos;
34 }
35
36 // Devuelve el nombre de la persona mas votada
37 static public String NombreDelMasVotado() {
38     return PersonaMasVotada;
39 }
40
41 // Devuelve el numero de votos de la persona mas votada
42 static public int VotosDelMasVotado() {
43     return VotosMasVotado;
44 }
45
46 }
```



En las líneas de código 4 y 5 se definen las propiedades de instancia *Persona* y *Votos*, que contendrán el nombre completo de un candidato y el número de votos que lleva contabilizados. El nombre del candidato se asigna en la instanciación de la clase, a través del constructor situado en la línea 13. Una vez creada la instancia, se puede conocer el nombre del candidato utilizando el método *NombrePersona* (línea 27) y el número de votos contabilizados, utilizando el método *Votos* (línea 32).

Las líneas 9 y 10 declaran las propiedades de clase *VotosMasVotado* y *PersonaMasVotada*, que contendrán el número de votos de la persona más votada hasta el momento y su nombre completo. Al ser propiedades estáticas, no pertenecen a ninguna instancia, sino que tienen existencia desde que se empieza a ejecutar la aplicación. Sólo existe un espacio de almacenamiento de estas variables, a diferencia de las propiedades *Persona* y *Votos* que se van creando (replicando) a medida que se crean instancias de la clase.

El valor de *VotosMasVotado* se puede obtener en cualquier momento (y conviene resaltar las palabras “en cualquier momento”) a través del método estático *VotosDelMasVotado* (línea 42). Tanto la propiedad como el método, al ser estáticos, son accesibles desde el comienzo de la aplicación. Análogamente, el valor de *PersonaMasVotada* puede consultarse invocando el método estático *NombreDelMasVotado* (línea 37).

Cada vez que se contabiliza un voto del candidato con el que se ha instanciado la clase, se debe llamar al método *Voto* (línea 18), que aumenta el número de votos del candidato (línea 19). Además se comprueba si este voto convierte al candidato en el nuevo ganador temporal (línea 20) de la votación; si es así, se actualiza el nombre y número de votos del candidato más votado hasta el momento (líneas 21 y 22).



Con la clase *Votacion* implementada, podemos pasar a comprobar su funcionamiento en una votación simulada de tres candidatos: Juan, Ana y Adela. La clase *PruebaVotacion* realiza esta función:

```
1  class PruebaVotacion {
2      public static void main (String[] args) {
3
4          System.out.println (Votacion.NombreDelMasVotado() +
5                               ": " + Votacion.VotosDelMasVotado());
6
7          // Tenemos tres candidatos en esta votacion
8          Votacion Juan = new Votacion ("Juan Peire");
9          Votacion Ana = new Votacion ("Ana Garcia");
10         Votacion Adela = new Votacion ("Adela Sancho");
11
12         // empieza la votacion
13         Juan.Voto(); Ana.Voto(); Ana.Voto(); Ana.Voto();
14         Adela.Voto();
15         System.out.println (Votacion.NombreDelMasVotado() +
16                              ": " + Votacion.VotosDelMasVotado());
17
18         Juan.Voto(); Juan.Voto(); Juan.Voto(); Adela.Voto();
19         System.out.println (Votacion.NombreDelMasVotado() +
20                              ": " + Votacion.VotosDelMasVotado());
21
22         Adela.Voto(); Adela.Voto(); Ana.Voto(); Ana.Voto();
23         System.out.println (Votacion.NombreDelMasVotado() +
24                              ": " + Votacion.VotosDelMasVotado());
25
26         System.out.println (Juan.NombrePersona() + ": " +
27                              Juan.Votos() );
28         System.out.println (Ana.NombrePersona() + ": " +
29                              Ana.Votos() );
30         System.out.println (Adela.NombrePersona() + ": " +
31                              Adela.Votos() );
32     }
33 }
```

En la línea 4 se accede al nombre y número de votos del candidato más votado (y todavía no hemos creado ningún candidato). Esto es posible porque tanto los métodos invocados como las variables accedidas son estáticos, y tienen existencia antes de la creación de instancias de la clase. Obsérvese como se accede a los métodos a través del nombre de la clase, no a través del nombre de ninguna instancia de la misma (que ni siquiera existe en este momento). El valor esperado

impreso por la instrucción *System.out* es el de inicialización de las variables estáticas en la clase *Votacion* (*null* y 0).

En las líneas 8, 9 y 10 damos “vida” a nuestros candidatos: Juan, Ana y Adela, creando instancias de la clase *Votacion*. En este momento, además de las propiedades ya existentes: *VotosMasVotado* y *PersonaMasVotada*, se dispone de tres copias de las propiedades de instancia: *Persona* y *Votos*.

En la línea 13 se contabilizan los siguientes votos: uno para Juan, tres para Ana y uno para Adela. En la línea 14 se comprueba que llevamos un seguimiento correcto de la persona más votada (Ana García, con 3 votos). Seguimos referenciando a los métodos estáticos a través del nombre de la clase (no de una instancia), que es la forma de proceder más natural y elegante.

En la línea 17 se contabilizan nuevos votos, obteniéndose resultados en la 18. De igual forma se actúa en las líneas 21 y 22.

En las líneas 25, 26 y 27 se imprimen los nombres y votos obtenidos por cada candidato. Esta información se guarda en variables de instancia, accedidas (como no podría ser de otra manera) por métodos de instancia, por lo que la referencia a los métodos se hace a través de los identificadores de instancia (*Juan*, *Ana* y *Adela*).

## 4.5.7 Resultados



```
MS-DOS
C:\CursoJava\Introduccion>java PruebaVotacion
null: 0
Ana Garcia: 3
Juan Peire: 4
Ana Garcia: 5
Juan Peire: 4
Ana Garcia: 5
Adela Sancho: 4
```

## 4.6 PAQUETES Y ATRIBUTOS DE ACCESO

Los paquetes sirven para agrupar clases relacionadas, de esta manera, cada paquete contiene un conjunto de clases. Las clases que hay dentro de un paquete deben tener nombres diferentes para que puedan diferenciarse entre sí, pero no hay ningún problema en que dos clases que pertenecen a paquetes diferentes tengan el

mismo nombre; los paquetes facilitan tanto el agrupamiento de clases como la asignación de nombres.

Hasta ahora no hemos hecho un uso explícito de los paquetes en las clases que hemos creado. Cuando no se especifica el nombre del paquete al que pertenece una clase, esa clase pasa a pertenecer al “paquete por defecto”.

#### 4.6.1 Definición de paquetes

Definir el paquete al que pertenece una clase es muy sencillo: basta con incluir la sentencia *package Nombre\_Paquete*; como primera sentencia de la clase (obligatoriamente la primera). Ejemplo:

```
package Terminal;  
public class Telefono {  
.....  
}
```

```
package Terminal;  
public class Ordenador {  
.....  
}
```

```
package Terminal;  
public class WebTV {  
.....  
}
```

En el ejemplo anterior, el paquete *Terminal* contiene tres clases: *Telefono*, *Ordenador* y *WebTV*. El nombre de los ficheros que contienen las clases sigue siendo *Telefono.java*, *Ordenador.java* y *WebTV.java*. Estos ficheros, obligatoriamente, deben situarse en un directorio (carpeta) con el nombre del paquete: *Terminal*.

Las clases definidas como *public* son accesibles desde fuera del paquete, las que no presentan este atributo de acceso sólo son accesibles desde dentro del paquete (sirven para dar soporte a las clases publicas del paquete).

En resumen, en este ejemplo se definen tres clases *Telefono*, *Ordenador* y *WebTV*, pertenecientes a un mismo paquete *Terminal* y accesibles desde fuera del paquete, por ser publicas. Los ficheros que contienen las clases se encuentran en un

directorio de nombre *Terminal* (obligatoriamente el mismo nombre que el del paquete).

## 4.6.2 Utilización de las clases de un paquete

Cuando necesitamos hacer uso de todas o algunas de las clases de un paquete, debemos indicarlo de manera explícita para que el compilador sepa donde se encuentran las clases y pueda resolver las referencias a las mismas. Para ello utilizamos la sentencia *import*:

- *import Nombre\_Paquete.Nombre\_Clase;*
- *import NombrePaquete.\*;*

En el primer caso se especifica la clase que se va a utilizar (junto con la referencia de su paquete). En el segundo caso se indica que queremos hacer uso (potencialmente) de todas las clases del paquete. Conviene recordar que sólo podemos hacer uso, desde fuera de un paquete, de las clases definidas como públicas en dicho paquete.

Siguiendo el ejemplo del paquete *Terminal*, podríamos utilizar:

```
import Terminal.Ordenador;  
class TerminalOficinaBancaria {  
    // podemos utilizar las referencias deseadas a la clase Ordenador  
}
```

```
import Terminal.*;  
class TerminalOficinaBancaria {  
    // podemos utilizar las referencias deseadas a las 3 clases del paquete Terminal  
}
```

```
import Terminal.Telefono;  
import Terminal.Ordenador;  
import Terminal.WebTV;  
class TerminalOficinaBancaria {  
    // podemos utilizar las referencias deseadas a las 3 clases del paquete Terminal  
}
```

Podemos hacer uso de tantas sentencias *import* combinadas como deseemos, referidas a un mismo paquete, a diferentes paquetes, con la utilización de asterisco o sin él. Como se puede observar, el tercer ejemplo produce el mismo efecto que el segundo: la posibilidad de utilización de las tres clases del paquete *Terminal*.

En este momento cabe realizarse una pregunta: ¿Cómo es posible que en ejercicios anteriores hayamos hecho uso de la clase *Math* (en el paquete *java.lang*) sin haber puesto la correspondiente sentencia *import java.lang.Math*, y lo mismo con la clase *String*? ...es debido a que el uso de este paquete de utilidades es tan común que los diseñadores del lenguaje decidieron que no fuera necesario importarlo para poder utilizarlo.

Una vez que hemos indicado, con la sentencia *import*, que vamos a hacer uso de una serie de clases, podemos referenciar sus propiedades y métodos de manera directa. Una alternativa a esta posibilidad es no utilizar la sentencia *import* y referenciar los objetos con caminos absolutos:

```
Terminal.Telefono MiTelefono = new Terminal.Telefono(....);
```

Aunque resulta más legible el código cuando se utiliza la sentencia *import*:

```
import Terminal.Telefono;
```

```
.....  
Telefono MiTelefono = new Telefono(....);
```

### 4.6.3 Proceso de compilación cuando se utilizan paquetes

El proceso de compilación cuando se importan paquetes puede ser el mismo que cuando no se importan. Basta con realizar previamente la configuración necesaria.

Las clases de un paquete se encuentran en el directorio que tiene el mismo nombre que el paquete, y habitualmente compilaremos las clases en ese directorio, generando los objetos *.class*. Siguiendo nuestro ejemplo tendremos *Telefono.class*, *Ordenador.class* y *WebTV.class* en el directorio *Terminal*.

Para que el compilador funcione correctamente, debe encontrar los ficheros *.class* de nuestros programas (que se encontrarán en el directorio de trabajo que utilicemos) y el de las clases que importamos, que se encontrarán, en general, en los directorios con nombres de paquetes.

En nuestro ejemplo, suponiendo que trabajamos en el directorio *C:\CursoJava\Introduccion* y que hemos situado las clases *Telefono*, *Ordenador* y *WebTV* en el directorio *C:\Paquetes\Terminal*, el compilador deberá buscar los ficheros *.class* en estos dos directorios. Debemos actualizar la variable de entorno del sistema *CLASSPATH* (ver lección 1) con el siguiente valor:

```
CLASSPATH = C:\CursoJava\Introducción;C:\Paquetes\Terminal
```

Según el sistema operativo que utilicemos deberemos cambiar la variable de entorno de una manera u otra, por ejemplo, en Windows 98 lo podemos hacer a través del Panel de Control (Sistema) o bien añadiendo la línea:

```
set CLASSPATH = C:\CursoJava\Introducción;C:\Paquetes\Terminal al fichero
autoexec.bat
```

También podemos utilizar el atributo `-classpath` en el compilador de java:

```
javac -classpath .;C:\CursoJavaIntroduccion;C:\Paquetes\Terminal
TerminalOficinaBancaria.java
```

4.6.4 Atributos de acceso a los miembros (propiedades y métodos) de una clase

Hasta ahora hemos utilizado los atributos de acceso público (*public*) y privado (*private*) para indicar las posibilidades de acceso a las propiedades y métodos de una clase desde el exterior a la misma. Hemos empleado estos atributos según los principios básicos generales de la programación orientada a objetos: los atributos son de acceso privado y sólo se puede acceder a ellos (en consulta o modificación) a través de métodos públicos.

Si bien la manera con la que hemos actuado es, sin duda, la más habitual, adecuada y aconsejada, existen diferentes posibilidades que tienen que ver con la existencia de los paquetes, motivo por el cual hemos retrasado a este momento la explicación detallada de los atributos de acceso.

Existen 4 posibles atributos de acceso, que listamos a continuación según el nivel de restricción que imponen:

Tipo de acceso	Palabra reservada	Ejemplo	Acceso desde una clase del mismo paquete	Acceso desde una clase de otro paquete
Privado	<b>private</b>	private int PPrivada;	No	No
Sin especificar		int PSinEspecificar;	Sí	No
Protegido	<b>protected</b>	protected int PProtegida;	Sí	No
Publico	<b>public</b>	public int PPublica;	Sí	Sí

El acceso desde una clase perteneciente al mismo paquete sólo está prohibido si el miembro es privado. El acceso desde una clase perteneciente a otro

paquete sólo está permitido si el miembro es público. Los demás atributos de acceso tienen un mayor sentido cuando utilizamos el mecanismo de herencia, que se explicará un poco más adelante.

Ejemplo:

```
package ConversionDeMedidas;
public class ConversionDeDistancias {
    final public double LibrasAKilos = ...;
    .....
}

package VentaDeProductos;
import ConversionDeMedidas.ConversionDeDistancias;
class VentaDeNaranjas {
    .....
    double Kilos = Libras * LibrasAKilos;
    .....
}
```

La propiedad constante *LibrasAKilos* ha sido declarada como pública en la clase *ConversionDeDistancias*, dentro del paquete *ConversionDeMedidas*. Al declararse como pública, puede ser referenciada desde una clase (*VentaDeNaranjas*) situada en un paquete diferente (*VentaDeProductos*) al anterior. Esto no sería posible si *LibrasAKilos* tuviera un atributo de acceso diferente a *public*.

Cualquiera de los ejemplos que hemos realizado en las últimas lecciones nos sirve para ilustrar el uso de propiedades privadas situadas en clases pertenecientes a un mismo paquete. Puesto que no incluíamos ninguna sentencia *package*, todas nuestras clases pertenecían al “paquete por defecto”.

## 4.7 EJEMPLO: MÁQUINA EXPENDEDORA

En esta lección se desarrolla el software necesario para controlar el funcionamiento de una máquina expendedora sencilla. Esta máquina suministrará botellas de agua, naranja y coca-cola, permitiendo establecer los precios de cada producto. Así mismo admitirá monedas de un euro y de 10 céntimos de euro (0.1 euros). El diseño se realizará de tal manera que podamos definir con facilidad una máquina con cualquier número de productos.

Si analizamos el ejercicio con detalle, descubriremos que existe la necesidad de mantener la cuenta de:

- Cuantas botellas de agua nos quedan (en el depósito de botellas de agua)
- Cuantas botellas de naranja nos quedan (en el depósito de botellas de naranja)
- Cuantas botellas de coca-cola nos quedan (en el depósito de botellas de coca-cola)
- Cuantas monedas de un euro nos quedan (en el depósito de monedas de un euro)
- Cuantas monedas de un décimo de euro nos quedan (en el depósito de monedas de 10 céntimos de euro)

En definitiva, surge la necesidad de utilizar una clase que nos gestione un almacén de elementos. Esta clase la hemos implementado en ejercicios anteriores y podría ser reutilizada. A continuación mostramos el código de la misma, sin comentar sus propiedades y métodos, ya explicados en ejercicios anteriores.

```
1 public class MaquinaAlmacen {
2     private short Capacidad;
3     private short NumeroDeElementos = 0;
4
5     MaquinaAlmacen(short Capacidad) {
6         this.Capacidad = Capacidad;
7     }
8
9     public short DimeNumeroDeElementos() {
10         return (NumeroDeElementos);
11     }
12
13     public short DimeCapacidad() {
14         return (Capacidad);
15     }
16
17     public boolean HayElemento() {
18         return (NumeroDeElementos != 0);
19     }
20
21     public boolean HayHueco() {
22         return (NumeroDeElementos != Capacidad);
23     }
24
25     public void MeteElemento() {
26         NumeroDeElementos++;
27     }
28
29     public void SacaElemento() {
30         NumeroDeElementos--;
31     }
32 }
```



```
33     public void RellenaAlmacen() {
34         NumeroDeElementos = Capacidad;
35     }
36
37 } // MaquinaAlmacen
```

Una vez que disponemos de la clase *MaquinaAlmacen*, estamos en condiciones de codificar la clase *MaquinaModeloSencillo*, que definirá una máquina expendedora con tres almacenes de bebidas y dos almacenes de monedas. Además es necesario que en cada máquina instanciada se puedan poner precios personalizados, puesto que el precio al que se vende un producto varía dependiendo de donde se ubica la máquina.

La clase *MaquinaModeloSencillo* puede implementarse de la siguiente manera:

```
1  public class MaquinaModeloSencillo {
2
3      public MaquinaAlmacen DepositoEuro = new
4          MaquinaAlmacen((short)8);
5
6      public MaquinaAlmacen Deposito01Euro = new
7          MaquinaAlmacen((short)15);
8
9      public MaquinaAlmacen DepositoCocaCola = new
10         MaquinaAlmacen((short)10);
11
12     public MaquinaAlmacen DepositoNaranja = new
13         MaquinaAlmacen((short)5);
14
15     public MaquinaAlmacen DepositoAgua = new
16         MaquinaAlmacen((short)8);
17
18     private float PrecioCocaCola = 1.0f;
19     private float PrecioNaranja = 1.3f;
20     private float PrecioAgua = 0.6f; //precio recomendado
21
22     public void PonPrecios (float CocaCola, float Naranja,
23         float Agua) {
24         PrecioCocaCola = CocaCola;
25         PrecioNaranja = Naranja;
26         PrecioAgua = Agua;
27     }
28
29     public float DimePrecioCocaCola() {
30         return PrecioCocaCola;
31     }
32
33     public float DimePrecioNaranja() {
34         return PrecioNaranja;
35     }
36 }
```

```

26     }
27
28     public float DimePrecioAgua() {
29         return PrecioAgua;
30     }
31
32     public void MostrarEstadoMaquina() {
33         System.out.print("CocaColas: " +
34             DepositoCocaCola.DimeNumeroDeElementos() + "    ");
35         System.out.print("Naranjas: " +
36             DepositoNaranja.DimeNumeroDeElementos() + "    ");
37         System.out.println("Agua: " +
38             DepositoAgua.DimeNumeroDeElementos() + "    ");
39
40         System.out.print("1 Euro: " +
41             Deposito1Euro.DimeNumeroDeElementos() + "    ");
42         System.out.println("0.1 Euro: " +
43             Deposito01Euro.DimeNumeroDeElementos() + "    ");
44         System.out.println();
45     }
46
47 }

```

En la línea 3 se define el depósito de monedas de 1 euro (*Deposito1Euro*), inicializado con una capacidad de 8 elementos. En la línea 4 se hace una definición similar a la anterior para las monedas de 10 céntimos de euro (*Deposito01Euro*), con capacidad inicial para 15 monedas.

En la línea 6 se define el primer depósito de bebidas (*DepositoCocaCola*), con una capacidad para 10 recipientes. En las líneas 7 y 8 se definen *DepositoNaranja* y *DepositoAgua*, con capacidades de 5 y 8 recipientes.

Los 5 depósitos han sido declarados con el atributo de acceso *public*, de esta manera pueden ser referenciados directamente por las clases que instancien a *MaquinaModeloSencillo*. Una alternativa menos directa, pero más adaptada a la programación orientada a objetos sería declararlos como privados e incorporar los métodos necesarios para obtener sus referencias, por ejemplo:

```

public MaquinaModeloSencillo DameAlmacenAgua();
public MaquinaModeloSencillo DameAlmacenNaranja();
.....

```

o bien

```

public MaquinaModeloSencillo DameAlmacen(String TipoAlmacen);

```

Las líneas 10, 11 y 12 declaran las variables *PrecioCocaCola*, *PrecioNaranja* y *PrecioAgua*. Los valores por defecto de estos precios se establecen como 1, 1.3 y 0.6 euros. En la línea 14 se define el método *PonPrecios*, que permite modificar en cualquier momento el precio de los tres productos que admite una instancia de la clase *MaquinaSencilla*. Los métodos *DimePrecioCocaCola*, *DimePrecioNaranja* y *DimePrecioAgua* (líneas 20, 24 y 28) completan la funcionalidad necesaria para gestionar los precios de los productos.

Finalmente, en la línea 32, se implementa el método *MostrarEstadoMaquina*, que muestra el número de elementos que contiene cada uno de los depósitos.

De forma análoga a como hemos implementado la clase *MaquinaModeloSencillo*, podríamos crear nuevas clases que definieran más productos, o mejor todavía, podríamos ampliar esta clase con nuevos productos; esta última posibilidad se explica en el siguiente tema.

Las máquinas definidas, o las que podamos crear siguiendo el patrón de *MaquinaModeloSencillo*, necesitan un elemento adicional: el control de las monedas. Cuando un usuario trata de comprar una botella de un producto, no sólo es necesario verificar que se cuenta con existencias de ese producto, sino que también hay que realizar un control de las monedas que el usuario introduce y del cambio que hay que devolverle, pudiéndose dar la circunstancia de que no se le pueda suministrar un producto existente porque no se dispone del cambio necesario.

Para realizar el control de las monedas introducidas y del cambio que hay que devolver, se ha implementado la clase *MaquinaAutomataEuros*: el concepto más importante que hay que entender en esta clase es el significado de su primer método *IntroducciónMonedas*, que admite como parámetros una máquina de tipo *MaquinaModeloSencillo* y un precio. Este método, además de realizar todo el control monetario, devuelve *true* si el pago se ha realizado con éxito y *false* si no ha sido así.

Los detalles de esta clase pueden obviarse en el contexto general del ejercicio. En cualquier caso, los lectores interesados en su funcionamiento pueden ayudarse de las siguientes indicaciones:

El método *IntroduccionMonedas* admite las entradas de usuario (u, d, a) que simulan la introducción de una moneda de un **e**uro, de un **d**écimo de euro o de la pulsación de un botón “anular operación”. Existe una propiedad muy importante, *Acumulado*, donde se guarda la cantidad de dinero que el usuario lleva introducido; el método implementa un bucle de introducción de monedas (línea 15) hasta que *Acumulado* deje de ser menor que el precio del producto (línea 47) o el usuario anule la operación (líneas 38 a 41).

Cuando el usuario introduce un euro (líneas 17 y 20), se comprueba si existe hueco en el depósito de monedas de euro de *Maquina* (línea 21). Si es así se introduce el euro en el depósito (línea 22) y se incrementa la propiedad *Acumulado* (línea 23), en caso contrario se muestra un mensaje informando de la situación (línea 25) y el euro no introducido en el depósito lo podrá recoger el usuario.

La introducción de monedas de décimo de euro se trata de manera equivalente a la introducción de euros explicada en el párrafo anterior.

Al salirse del bucle (línea 48) se comprueba si la operación ha sido anulada, en cuyo caso se devuelve la cantidad de moneda introducida (*Acumulado*). Si la operación no fue anulada (línea 50) es necesario comprobar si tenemos cambio disponible (línea 51); si es así se devuelve la cantidad sobrante *Acumulado-Precio* en la línea 52. Si no hay cambio disponible (línea 53) se visualiza un mensaje indicándolo, se devuelve todo el dinero introducido y se almacena el valor *true* en la propiedad *Anulado*.

Finalmente, en la línea 58, devolvemos la indicación de operación con éxito (es decir, no anulada).

El análisis de los métodos *CambioDisponible* y *Devolver* se deja al lector interesado en los detalles de funcionamiento de esta clase.

```

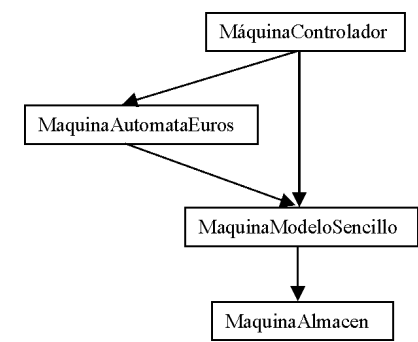
1  public class MaquinaAutomataEuros {
2
3      // *****
4      // * Recoge monedas en 'Maquina' para cobrar 'Precio'.
5      // * Devuelve 'true'
6      // * si el pago se ha realizado con exito y 'false' en
7      // * caso contrario
8      // *****
9      public static boolean IntroduccionMonedas
10         (MaquinaModeloSencillo Maquina, float Precio) {
11
12         String Accion;
13         char Car;
14         boolean Pagado=false, Anulado = false, CambioOK ;
15         float Acumulado = 0;
16
17         do {
18             System.out.println("-- u,d,a --");
19             Accion = Teclado.Lee_String();
20             Car = Accion.charAt(0);
21             switch (Car) {
22                 case 'u':
23                     if (Maquina.Deposito1Euro.HayHueco()) {

```

```
22         Maquina.Deposito1Euro.MeteElemento();
23         Acumulado = Acumulado + 1f;
24     } else
25         System.out.println("Temporalmente esta
26             maquina no cepta monedas de un euro");
27     break;
28
29     case 'd':
30         if (Maquina.Deposito01Euro.HayHueco()) {
31             Maquina.Deposito01Euro.MeteElemento();
32             Acumulado = Acumulado + 0.1f;
33         } else
34             System.out.println("Temporalmente esta
35                 maquina no acepta monedas de 0.1 euros");
36     break;
37
38     case 'a':
39         System.out.println("Operación anulada");
40         Anulado = true;
41         break;
42     }
43
44     Maquina.MostrarEstadoMaquina();
45
46     } while (Acumulado<Precio || Anulado);
47
48     if (Anulado)
49         Devolver(Maquina,Acumulado);
50     else
51         if (CambioDisponible(Maquina,Acumulado-Precio)) {
52             Devolver (Maquina,Acumulado-Precio);
53         } else {
54             System.out.println("La maquina no dispone del
55                 cambio necesario");
56             Devolver(Maquina,Acumulado);
57             Anulado = true;
58         }
59     return (!Anulado);
60 }
61
62
63 // *****
64 // * Indica si es posible devolver 'Cantidad' euros en
65 // * 'Maquina'
66 // *****
67 private static boolean CambioDisponible
68     (MaquinaModeloSencillo Maquina, float Cantidad) {
```

```
68
69     int Monedas1, Monedas01;
70
71     Cantidad = Cantidad + 0.01f; //Evita problemas de
                                   //falta de precision
72     Monedas1 = (int) Math.floor((double) Cantidad);
73     Cantidad = Cantidad - (float) Monedas1;
74     Monedas01 = (int) Math.floor((double) Cantidad*10f);
75     return {
        (Maquina.Deposito1Euro.DimeNumeroDeElementos())>=Monedas1)&&
        (Maquina.Deposito01Euro.DimeNumeroDeElementos())>=Monedas01));
76     }
77
78
79
80 // *****
81 // *   Devuelve la cantidad de dinero indicada,
82 // *   actualizando los almacenes de monedas
83 // *****
84 private static void Devolver (MaquinaModeloSencillo
85                               Maquina, float Cantidad) {
86
87     int Monedas1, Monedas01;
88     Cantidad = Cantidad + 0.01f; //Evita problemas de
                                   //falta de precision
89     Monedas1 = (int) Math.floor((double) Cantidad);
90     Cantidad = Cantidad - (float) Monedas1;
91     Monedas01 = (int) Math.floor((double) Cantidad*10f);
92
93     for (int i=1; i<=Monedas1; i++){
94         Maquina.Deposito1Euro.SacaElemento();
95         // Sacar 1 moneda de un euro
96     }
97
98     for (int i=1; i<=Monedas01; i++){
99         Maquina.Deposito01Euro.SacaElemento();
100        // Sacar 1 moneda de 0.1 euro
101    }
102    System.out.println("Recoja el importe: "+Monedas1+"
103                        monedas de un euro y "+Monedas01+
104                        " monedas de 0.1 euros");
105
106 }
107
108 } // clase
```

Las clases anteriores completan la funcionalidad de la máquina expendedora que se pretendía implementar en este ejercicio. Para finalizar nos basta con crear una clase que haga uso de las anteriores e interaccione con el usuario. A esta clase la llamaremos *MaquinaControlador*. La jerarquía de clases del ejercicio nos queda de la siguiente manera:



La clase *MaquinaControlador* define la propiedad *MiMaquina*, instanciando *MaquinaModeloSencillo* (línea 7); posteriormente se establecen los precios de los productos (línea 8), se introduce alguna moneda en los depósitos (una de un euro y dos de 0.1 euro) en las líneas 9 a 11 y se rellenan los almacenes de coca-cola y naranja (líneas 12 y 13). Suponemos que no disponemos de botellas de agua en este momento.

Nos metemos en un bucle infinito de funcionamiento de la máquina (líneas 17 a 70), aunque como se puede observar en la línea 70 hemos permitido salir del bucle escribiendo cualquier palabra que comience por ‘s’. En el bucle, el usuario puede pulsar entre las opciones ‘c’, ‘n’ y ‘a’, correspondientes a coca-cola, naranja y agua.

Si el usuario selecciona un producto concreto, por ejemplo naranja (línea 36), se muestra un mensaje con su elección (línea 37) y se comprueba si hay elementos en el almacén del producto (línea 38). Si no hay elementos de ese producto se muestra la situación en un mensaje (líneas 46 y 47), si hay existencias invocamos al método estático *IntroduccionMonedas* de la clase *MaquinaAutomataEuros* (línea 39); si todo va bien se extrae el elemento (la botella de naranja) y se le indica al usuario que la recoja (líneas 41 y 42), en caso contrario el método *IntroduccionMonedas* se encarga de realizar las acciones oportunas con las monedas.





```

                                naranja");
43         // Sacar físicamente la Naranja
44     }
45 }
46 else
47     System.out.println("Producto agotado");
48     break;
49
50 case 'a':
51     System.out.println("Ha seleccionado Agua");
52     if (MiMaquina.DepositoAgua.HayElemento()) {
53         if MaquinaAutomataEuros.IntroduccionMonedas
54             (MiMaquina, MiMaquina.DimePrecioAgua())) {
55             MiMaquina.DepositoAgua.SacaElemento();
56             System.out.println("No olvide coger su
                                agua");
57             // Sacar físicamente el agua
58         }
59     }
60     else
61         System.out.println("Producto agotado");
62         break;
63
64     default:
65         System.out.println("Error de seleccion,
                                intentelo de nuevo");
66         break;
67     }
68     MiMaquina.MostrarEstadoMaquina();
69
70 } while (!Accion.equals("s"));
71 }
72 }
```

A continuación se muestra una posible ejecución del software desarrollado:

En primer lugar se adquiere una botella de naranja (1.3 euros) suministrando el precio exacto. Obsérvese como se incrementan los depósitos de monedas y se decrementa el de botellas de naranja.

Posteriormente se trata de adquirir una coca-cola (1.1 euros) con dos monedas de euro. El almacén de euros va incrementando el número de elementos, pero posteriormente, cuando se descubre que la máquina no dispone del cambio necesario, se devuelven las monedas:

```
MC JAVA
C:\CursoJava\Introduccion>java MaquinaControlador
-- c.n.a.s --
naranja
Ha seleccionado Naranja
-- u.d.a --
unidad
CocaColas: 10 Naranjas: 5 Agua: 0
1 Euro: 2 0.1 Euro: 2
-- u.d.a --
decimo
CocaColas: 10 Naranjas: 5 Agua: 0
1 Euro: 2 0.1 Euro: 3
-- u.d.a --
decimo
CocaColas: 10 Naranjas: 5 Agua: 0
1 Euro: 2 0.1 Euro: 4
-- u.d.a --
decimo
CocaColas: 10 Naranjas: 5 Agua: 0
1 Euro: 2 0.1 Euro: 5
Recoja el importe: 0 monedas de un euro y 0 monedas de 0.1 euros
No olvide coger su naranja
CocaColas: 10 Naranjas: 4 Agua: 0
1 Euro: 2 0.1 Euro: 5
-- c.n.a.s --
c
Ha seleccionado Coca cola
-- u.d.a --
unidad
CocaColas: 10 Naranjas: 4 Agua: 0
1 Euro: 3 0.1 Euro: 5
-- u.d.a --
unidad
CocaColas: 10 Naranjas: 4 Agua: 0
1 Euro: 4 0.1 Euro: 5
La maquina no dispone del cambio necesario
Recoja el importe: 2 monedas de un euro y 0 monedas de 0.1 euros
CocaColas: 10 Naranjas: 4 Agua: 0
1 Euro: 2 0.1 Euro: 5
-- c.n.a.s --
```

# PROGRAMACIÓN ORIENTADA A OBJETOS USANDO HERENCIA

---

## 5.1 HERENCIA

### 5.1.1 *Funcionamiento básico*

La herencia es un mecanismo muy importante en la programación orientada a objetos. Gracias a la herencia podemos crear clases que especializan a otras previamente definidas.

Pongamos como ejemplo una clase *Publicacion* en la que se determinan propiedades básicas comunes a todas las publicaciones (libros, revistas, periódicos, etc.). Entre estas propiedades pueden definirse el número de páginas y el precio de la publicación:

```
1 class Publicacion {  
2     public int NumeroDePaginas;  
3     public float Precio;  
4 }
```

Partiendo de la clase base *Publicacion*, podemos especializar un libro de la siguiente manera:

```
1 class Libro extends Publicacion {  
2     public String Titulo;  
3     public String TipoPortada;  
4     public String ISBN;  
5     public String NombreAutor;  
6     public String Editorial;  
7 }
```

La clase *Libro* extiende (y se emplea la palabra reservada *extends*) a la clase *Publicación*. La clase *Libro* contiene ahora todas sus propiedades más las propiedades de la clase *Publicacion*. De esta manera, un libro se define por *NumeroDePaginas*, *Precio*, *Titulo*, *TipoPortada*, etc.

Se dice (en terminología orientada a objetos) que *Libro* es una clase derivada o subclase de *Publicación*, mientras que *Publicación* es la superclase de *Libro*.

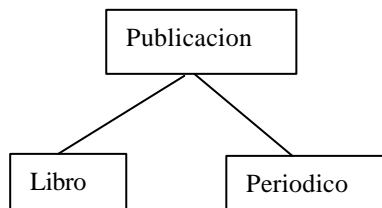
Para comprobar que un libro contiene tanto las propiedades de la superclase como las de la subclase realizamos el siguiente programa:

```
1 class PruebaLibro {
2     public static void main(String[] args) {
3         Libro MiLibro = new Libro();
4         MiLibro.NombreAutor = "Frederick Forsyth";
5         MiLibro.Titulo = "El manifiesto negro";
6         MiLibro.Editorial = "Circulo de lectores";
7         MiLibro.TipoPortada = "Dura";
8         MiLibro.NumeroDePaginas = 575;
9     }
10 }
```

Podemos comprobar como la clase *Libro* ha heredado las propiedades de la clase *Publicacion*.

Una superclase puede tener cualquier número de subclases:

```
1 class Periodico extends Publicacion {
2     public String Nombre;
3     public String Fecha;
4 }
```



El mecanismo de herencia no sólo actúa sobre las propiedades, lo hace sobre todos los miembros (métodos y propiedades) de las clases, de esta manera, las clases anteriores se pueden completar con métodos:

```
1 class Publicacion2 {
2     private int NumeroDePaginas;
3     private float Precio;
4
5     public int DimeNumeroDePaginas(){
6         return NumeroDePaginas;
7     }
8
9     public void PonNumeroDePaginas(int NumeroDePaginas){
10         this.NumeroDePaginas = NumeroDePaginas;
11     }
12
13     public float DimePrecio(){
14         return Precio;
15     }
16
17     public void PonPrecio(float Precio){
18         this.Precio = Precio;
19     }
20 }

```

```
1 class Periodico2 extends Publicacion2 {
2     private String Nombre;
3     private String Fecha;
4
5     public String DimeNombre() {
6         return Nombre;
7     }
8
9     public void PonNombre(String Nombre) {
10         this.Nombre = Nombre;
11     }
12
13     public String DimeFecha() {
14         return Fecha;
15     }
16
17     public void PonFecha(String Fecha) {
18         this.Fecha = Fecha;
19     }
20 }

```

En este caso la clase derivada (o subclase) *Periodico2* contiene tanto las propiedades como los métodos de la superclase *Publicacion2*, aunque como veremos en el siguiente apartado no todos los miembros son directamente accesibles. La clase *PruebaPeriodico2* nos muestra como podemos acceder a los valores de las propiedades de *Publicacion2*.

```
1 class PruebaPeriodico2 {
2     public static void main(String[] args) {
3         Periodico2 MiPeriodico = new Periodico2();
4         MiPeriodico.PonNumeroDePaginas(65);
5         MiPeriodico.PonPrecio(0.9f);
6         MiPeriodico.PonFecha("22/02/2003");
7         System.out.println(MiPeriodico.DimeNumeroDePaginas());
8     }
9 }
```

5.1.2 Accesibilidad a los miembros heredados en una subclase

Las subclases heredan todos los miembros de su superclase, aunque no todos los miembros tienen porque ser accesibles. En particular, los miembros privados de una superclase no son directamente accesibles en sus subclases. La siguiente tabla muestra las posibilidades existentes:

Tipo de acceso	Palabra reservada	Ejemplo	Acceso desde una subclase del mismo paquete	Acceso desde una subclase de otro paquete
Privado	<b>private</b>	private int PPrivada;	No	No
Sin especificar		int PSinEspecificar;	Sí	No
Protegido	<b>protected</b>	protected int PProtegida;	Sí	Sí
Publico	<b>public</b>	public int PPublica;	Sí	Sí

La única diferencia entre las posibilidades de acceso de una clase a los miembros de otra y de una subclase a los miembros de su superclase está en que los miembros protegidos de la superclase sí son accesibles a subclases situadas en paquetes diferentes a la superclase.

En nuestro ejemplo, en el que sólo se emplean propiedades privadas y métodos públicos, las propiedades privadas de la superclase *Publicacion2* (*NumeroDePaginas* y *Precio*) no son directamente accesibles desde la subclase *Periodico2*.

Existe la posibilidad de “ocultar” propiedades de la superclase, definiendo propiedades con el mismo nombre en la subclase. En este caso, en la subclase, al referenciar directamente el nombre nos referimos a la propiedad de la subclase; si

escribimos *super.Nombre* nos referimos a la propiedad de la superclase. Resulta conveniente no utilizar este mecanismo de ocultación de propiedades en el diseño de las clases.

Los métodos tienen un mecanismo similar al de ocultación de propiedades; se denomina “redefinición” y nos permite volver a definir el comportamiento de los métodos de la superclase. Para poder redefinir un método debe tener la misma firma que el equivalente en la superclase, además su atributo de acceso debe ser el mismo o menos restrictivo que el original.

Al igual que ocurre con las propiedades, nos podemos referir al método de la superclase utilizando la palabra reservada *super*.

### 5.1.3 Constructores de las subclases

Normalmente, entre otras posibles acciones, los constructores inicializan las propiedades de las clases, es decir establecen su estado inicial. En las clases derivadas de una superclase, los constructores, además de establecer el estado inicial de su propia subclase deben establecer el estado inicial de la superclase.

Para realizar la acción descrita en el párrafo anterior, los constructores de las subclases tienen la posibilidad de invocar a los constructores de sus superclases. La sintaxis es la siguiente:

*super (parámetros) // donde puede haber cero o mas parámetros*

La llamada *super*, en caso de utilizarse, debe ser (obligatoriamente) la primera sentencia del constructor.

Cuando en un constructor de una subclase no se utiliza la sentencia *super*, el compilador inserta automáticamente *super()* como primera instrucción. Esto puede dar lugar a errores de compilación, puesto que si en la superclase hemos definido algún constructor y no hemos definido *super()*, el compilador no encontrará este constructor. Si en la superclase no hemos definido ningún constructor, no existirán problemas, puesto que *super()* es el constructor por defecto que crea el compilador cuando nosotros no definimos otro. En cualquier caso, cuando se necesite, se recomienda poner explícitamente la llamada *super()*.

Repasemos estos conceptos introduciendo constructores en el ejemplo de las publicaciones que estamos siguiendo:

```
1  class Publicacion3 {
2      private int NumeroDePaginas;
3      private float Precio;
4
5      Publicacion3() {
6          NumeroDePaginas = 0;
7          Precio = 0f;
8      }
9
10     Publicacion3(int NumeroDePaginas) {
11         PonNumeroDePaginas(NumeroDePaginas);
12     }
13
14     Publicacion3(float Precio) {
15         PonPrecio(Precio);
16     }
17
18     Publicacion3(int NumeroDePaginas, float Precio) {
19         this(NumeroDePaginas);
20         PonPrecio(Precio);
21     }
22
23     Publicacion3(float Precio, int NumeroDePaginas) {
24         this(NumeroDePaginas, Precio);
25     }
26
27     public int DimeNumeroDePaginas(){
28         return NumeroDePaginas;
29     }
30
31     public void PonNumeroDePaginas(int NumeroDePaginas){
32         this.NumeroDePaginas = NumeroDePaginas;
33     }
34
35     public float DimePrecio(){
36         return Precio;
37     }
38
39     public void PonPrecio(float Precio){
40         this.Precio = Precio;
41     }
42 }
```

La clase *Publicacion3* contiene todos los constructores posibles para inicializar el valor de dos propiedades. Tenemos el constructor sin parámetros (al que se llamará automáticamente si no ponemos la sentencia *super* en algún



constructor de alguna subclase), constructores con un solo parámetro y constructores con dos parámetros.

Cada sentencia *this(parámetros)* invoca al constructor de la misma clase que coincide en firma con la llamada; por ejemplo, en la línea 19, se invoca al constructor de la línea 10 y en la línea 24 al constructor de la 18.

Ahora modificamos la subclase *Periodico2* para que contenga constructores:

```
1  class Periodico3 extends Publicacion3 {
2      private String Nombre;
3      private String Fecha;
4
5      Periodico3() {
6          super();
7          Nombre = null;
8          Fecha = null;
9      }
10
11     Periodico3(String Nombre, String Fecha) {
12         super();
13         this.Nombre = Nombre;
14         this.Fecha = Fecha;
15     }
16
17     Periodico3(int NumeroDePaginas, float Precio) {
18         super(NumeroDePaginas, Precio);
19         this.Nombre = null;
20         this.Fecha = null;
21     }
22
23     Periodico3(String Nombre, String Fecha,
24                 int NumeroDePaginas, float Precio) {
25         super(NumeroDePaginas, Precio);
26         this.Nombre = Nombre;
27         this.Fecha = Fecha;
28     }
29
30     public String DimeNombre() {
31         return Nombre;
32     }
33
34     public void PonNombre(String Nombre) {
35         this.Nombre = Nombre;
36     }
37
38     public String DimeFecha() {
```

```
39     return Fecha;
40 }
41
42 public void PonFecha(String Fecha) {
43     this.Fecha = Fecha;
44 }
45 }
```

La clase *Periodico3* contiene una serie de constructores representativos (poner todos los posibles constructores resulta innecesario e inviable dado el gran número de combinaciones posibles). Estos constructores nos permiten inicializar explícitamente todas las propiedades de la subclase y la superclase (constructor definido en la línea 23), sólo las propiedades de la subclase (línea 11), sólo las propiedades de la superclase (línea 17) o no definir explícitamente ninguna propiedad (constructor vacío). En este último caso, aunque no se implementase, el compilador realizaría implícitamente las mismas acciones.

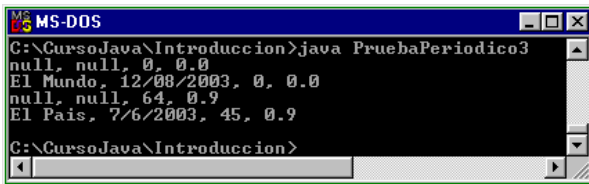
Para comprobar el funcionamiento de la clase *Periodico3* creamos la clase *PruebaPeriodico3*, que realiza diversas instancias que invocan a cada uno de los constructores definidos:

```
1  class PruebaPeriodico3 {
2      public static void main(String[] args) {
3          Periodico3 PeriodicoA = new Periodico3();
4          Periodico3 PeriodicoB = new Periodico3("El Mundo",
5                                                  "12/08/2003");
6          Periodico3 PeriodicoC = new Periodico3(64, 0.9f);
7          Periodico3 PeriodicoD = new
8              Periodico3("El Pais", "7/6/2003", 45, 0.9f);
9          Imprimir(PeriodicoA);
10         Imprimir(PeriodicoB);
11         Imprimir(PeriodicoC);
12         Imprimir(PeriodicoD);
13     }
14
15     private static void Imprimir(Periodico3 Periodico) {
16         String Nombre, Fecha;
17         int NumeroDePaginas;
18         float Precio;
19         Nombre = Periodico.DimeNombre();
20         Fecha = Periodico.DimeFecha();
21         NumeroDePaginas = Periodico.DimeNumeroDePaginas();
22         Precio = Periodico.DimePrecio();
23         System.out.println(Nombre+", "+Fecha+",
24                             "+NumeroDePaginas+", "+Precio);
25     }
26 }
```

En la clase *PruebaPeriodico3* se crea una instancia de *Periodico3* sin proporcionar ningún parámetro (línea 3), otra instancia proporcionando valores de las propiedades de la subclase (línea 4), otra con valores de la superclase (línea 5) y una última instancia que incluye valores iniciales de las propiedades de la subclase y la superclase (línea 6).

A partir de la línea 13 codificamos un método *Imprimir* que visualiza los contenidos de las 4 propiedades accesibles en la clase *Periodico3*, para ello utilizamos los métodos de la clase derivada *DimeNombre* y *DimeFecha* y los métodos de la superclase *DimeNumeroDePaginas* y *DimePrecio*.

Tras realizar las llamadas pertinentes al método *Imprimir*, obtenemos los resultados esperados, comprobándose el funcionamiento de los constructores de las clases derivadas:



```

MS-DOS
C:\CursoJava\Introduccion>java PruebaPeriodico3
null, null, 0, 0.0
El Mundo, 12/08/2003, 0, 0.0
null, null, 64, 0.9
El Pais, 7/6/2003, 45, 0.9
C:\CursoJava\Introduccion>
  
```

### 5.1.4 El modificador final

Si utilizamos el modificador *final* al definir una clase evitamos que se puedan construir clases derivadas de la misma:

```

public final class NombreClase {
    // contenido de la clase
}

public class ClaseDerivada extends NombreClase {
}
  
```

La utilización del modificador *final* asociado a las clases no es muy habitual, puesto que restringe la posibilidad de reutilizar las clases usando el mecanismo de herencia.

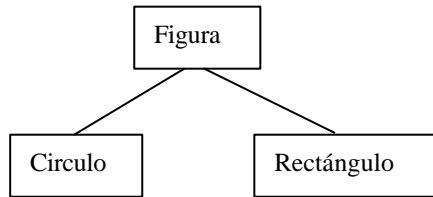
## 5.2 EJEMPLOS

En este apartado se muestran dos ejemplos sencillos de herencia. Los ejemplos son muy parecidos entre sí, con la intención de que el lector pueda realizar el segundo tomando como referencia el primero de ellos.

### 5.2.1 Figuras geométricas

En este primer ejemplo se pretende crear clases que representen diferentes figuras geométricas bidimensionales. Nosotros implementaremos el círculo y el rectángulo. Las figuras geométricas se caracterizarán por su color y su posición. En los círculos, además, se establece un radio; en los rectángulos es necesario definir la longitud de sus lados.

La estructura de las clases presenta el siguiente diagrama:



La clase *Figura*, que se muestra a continuación, contiene las propiedades privadas *ColorFigura* y *Posicion* (líneas 4 y 5). *ColorFigura* es del tipo *Color* (línea 1) y *Posicion* es un vector (matriz lineal) de dos componentes: *Posicion[0]* representando la coordenada X del centro de la figura y *Posicion[1]* representando la coordenada Y del centro de la figura.

```
1 import java.awt.Color;
2
3 public class Figura {
4     private Color ColorFigura;
5     private int[] Posicion = new int[2];
6
7     Figura() {
8         EstableceColor(Color.black);
9         Posicion[0] = 0;
10        Posicion[1] = 0;
11    }
12 }
```

```
13  Figura(Color color) {
14      EstableceColor(color);
15  }
16
17  Figura(Color color, int[] Posicion) {
18      EstableceColor(color);
19      EstableceCentro(Posicion);
20  }
21
22  public void EstableceColor(Color color) {
23      ColorFigura = color;
24  }
25
26  public Color DimeColor() {
27      return ColorFigura;
28  }
29
30  public void EstableceCentro(int[] Posicion) {
31      this.Posicion[0] = Posicion[0];
32      this.Posicion[1] = Posicion[1];
33  }
34
35  public int[] DimeCentro() {
36      return Posicion;
37  }
38
39 }
```

En la clase *Figura* se incluyen los métodos públicos básicos de consulta y actualización de las propiedades privadas: *EstableceColor* y *DimeColor* (líneas 22 y 26 respectivamente) y *EstableceCentro* y *DimeCentro* (líneas 30 y 35 respectivamente).

Los constructores de la clase hacen llamadas a los métodos *EstableceColor* y *EstableceCentro* para inicializar el contenido de las propiedades.

Una vez creada la clase *Figura*, podemos definir las diferentes subclases que especializan y amplían a esta superclase, representando diferentes figuras geométricas. A modo de ejemplo proporcionamos la clase derivada *Circulo*:

```
1  import java.awt.Color;
2
3  public class Circulo extends Figura {
4      private double Radio;
5
6      Circulo(double Radio) {
7          EstableceRadio(Radio);
```

```
8    }
9
10   Circulo(double Radio,Color color) {
11       super(color);
12       EstableceRadio(Radio);
13   }
14
15   Circulo(double Radio, Color color, int[] Posicion) {
16       super(color, Posicion);
17       EstableceRadio(Radio);
18   }
19
20   public void EstableceRadio(double Radio) {
21       this.Radio = Radio;
22   }
23
24   public double DimeRadio() {
25       return Radio;
26   }
27 }
```

*Circulo* es una clase derivada de *Figura* (línea 3) que incorpora una nueva propiedad *Radio* (línea 4) a las heredadas de su superclase (*ColorFigura* y *Posicion*). Los métodos públicos *EstableceRadio* y *DimeRadio* permiten la actualización y acceso a la propiedad privada *Radio*.

Se ha definido un constructor que permite instanciar un círculo con un radio determinado, dejando la posición central y color a sus valores por defecto (línea 6). El siguiente constructor (línea 10) permite establecer el radio y el color, mientras que el último constructor (línea 15) nos ofrece la posibilidad de asignar valores a todas las propiedades accesibles en la clase *Circulo*.

La última clase de este ejemplo especializa a *Figura*, definiendo objetos de tipo *Rectangulo*. Esta clase es similar a *Circulo*, aunque aquí se establece la longitud de los lados de un rectángulo en lugar del radio de un círculo. El código de la subclase es:

```
1  import java.awt.Color;
2
3  public class Rectangulo extends Figura {
4      private double[] Lados = new double[2];
5
6      Rectangulo(double[] Lados) {
7          EstableceLados(Lados);
8      }
9  }
```

```
10 Rectangulo(double[]Lados, Color color) {
11     super(color);
12     EstableceLados(Lados);
13 }
14
15 Rectangulo(double[] Lados, Color color, int[] Posicion) {
16     super(color, Posicion);
17     EstableceLados(Lados);
18 }
19
20 public void EstableceLados(double[] Lados) {
21     this.Lados[0] = Lados[0];
22     this.Lados[1] = Lados[1];
23 }
24
25 public double[] DimeLados() {
26     return Lados;
27 }
28 }
```

La propiedad privada *Lados* (línea 4) es un vector de dos componentes: la primera para definir un lado del rectángulo (por ejemplo el horizontal) y la segunda para definir el otro lado del rectángulo. El método *EstableceLados* permite modificar el contenido de la propiedad *Lados* y el método *DimeLados* sirve para consultar su valor.

Los constructores de la clase derivada permiten asignar valores iniciales a las propiedades de la superclase *Figura* y a la propiedad *Lados* de esta subclase.

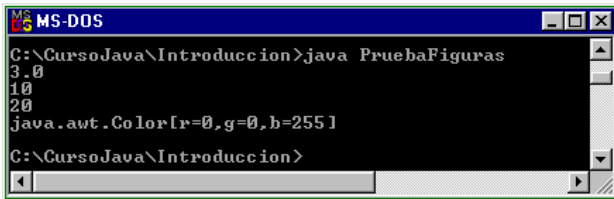
Para mostrar el funcionamiento de las clases heredadas de *Figura*, se muestra el siguiente código (*PruebaFiguras*):

```
1 import java.awt.Color;
2
3 public class PruebaFiguras {
4     public static void main(String[] args) {
5         int[] Posicion = {10,20};
6         double[] Lados = {50d,100d};
7         Circulo MiCirculo = new Circulo(3d,Color.red,Posicion);
8         Rectangulo MiRectangulo = new
9             Rectangulo(Lados,Color.blue,Posicion);
10
11         System.out.println(MiCirculo.DimeRadio());
12         int[] Centro = MiCirculo.DimeCentro();
13         System.out.println(Centro[0]);
14         System.out.println(Centro[1]);
15     }
```

```
15      System.out.println(MiRectangulo.DimeColor() );
16
17  }
18 }
```

En la línea 7 de la clase *PruebaFiguras* se instancia un círculo de radio 3, color rojo y posición central 10, 20. En la línea 8 se instancia un rectángulo con lados de longitud 50,100, color azul y posición 10, 20. En la línea 10 se invoca al método *DimeRadio* de la subclase *Circulo*, en la línea 11 al método *DimeCentro* de la superclase *Figura*, y en la línea 15 al método *DimeColor* de la superclase *Figura*.

El resultado es:



## 5.2.2 Vehículos

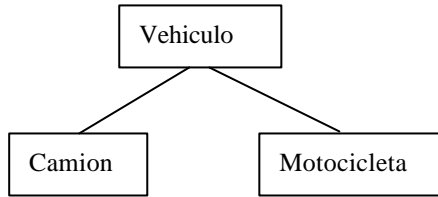
En este ejemplo se pretende definir clases que implementen algunas características de diferentes tipos de vehículos. Crearemos una clase *Vehiculo* que contendrá propiedades básicas comunes a todos los vehículos y actuará de superclase de las distintas clases derivadas (*Camion*, *Motocicleta*, etc.).

La superclase *Vehiculo* contendrá propiedades que establezcan el color del vehículo, el número de ruedas que utiliza, la cilindrada que tiene y la potencia que ofrece. Además contará con los métodos y constructores que se considere necesarios.

La clase derivada *Camion* establecerá una nueva propiedad que indique el número de ejes que posee el camión, junto a los constructores y métodos que se consideren adecuados.

La clase derivada *Motocicleta* contendrá una propiedad que albergue el número de ocupantes permitido que puede transportar cada motocicleta.





Finalmente se implementará una clase *PruebaVehiculos* que creará diversas instancias de camiones y motocicletas e invocará a diferentes métodos accesibles a las subclases.

Código de la superclase *Vehiculo*:

```
1  import java.awt.Color;
2
3  public class Vehiculo {
4      private Color ColorVehiculo;
5      private byte NumRuedas;
6      private short Cilindrada;
7      private short Potencia;
8
9      Vehiculo(Color color) {
10         EstableceColor(color);
11     }
12
13     Vehiculo (byte NumRuedas) {
14         this.NumRuedas = NumRuedas;
15     }
16
17     Vehiculo (short Cilindrada) {
18         this.Cilindrada = Cilindrada;
19     }
20
21     Vehiculo(Color color, byte NumRuedas) {
22         this(color);
23         this.NumRuedas = NumRuedas;
24     }
25
26     Vehiculo(Color color, byte NumRuedas, short Cilindrada) {
27         this(color,NumRuedas);
28         this.Cilindrada = Cilindrada;
29     }
30
31     Vehiculo(Color color, byte NumRuedas, short Cilindrada,
32         short Potencia) {
33         this(color,NumRuedas,Cilindrada);
34         this.Potencia = Potencia;
```

```
35  }
36
37  public void EstableceColor(Color color) {
38      ColorVehiculo = color;
39  }
40
41  public Color DimeColor() {
42      return ColorVehiculo;
43  }
44
45  public byte DimeNumRuedas() {
46      return NumRuedas;
47  }
48
49  public short DimeCilindrada() {
50      return Cilindrada;
51  }
52
53  public short DimePotencia() {
54      return Potencia;
55  }
56
57 }
```

Código de la subclase *Camion*:

```
1  import java.awt.Color;
2
3  public class Camion extends Vehiculo {
4
5      private byte NumeroDeEjes;
6
7      Camion(byte NumeroDeRuedas) {
8          super(NumeroDeRuedas);
9      }
10
11     Camion(Color color, byte NumeroDeRuedas) {
12         super(color, NumeroDeRuedas);
13     }
14
15     Camion(Color color, byte NumeroDeRuedas,
16         short Cilindrada) {
17         super(color, NumeroDeRuedas, Cilindrada);
18     }
19
20     Camion(Color color, byte NumeroDeRuedas,
21         short Cilindrada, short Potencia) {
22         super(color, NumeroDeRuedas, Cilindrada, Potencia);
23     }
24 }
```

```
23  }
24
25  Camion(Color color, byte NumeroDeRuedas,
26         byte NumeroDeEjes, short Cilindrada,
27         short Potencia) {
28      super(color, NumeroDeRuedas, Cilindrada, Potencia);
29      EstableceNumeroDeEjes(NumeroDeEjes);
30  }
31
32  public byte DimeNumeroDeEjes() {
33      return NumeroDeEjes;
34  }
35
36  public void EstableceNumeroDeEjes(byte NumeroDeEjes) {
37      this.NumeroDeEjes = NumeroDeEjes;
38  }
39
40 }
```

Código de la subclase *Motocicleta*:

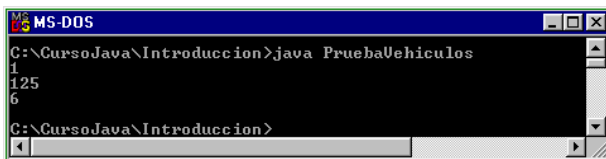
```
1  import java.awt.Color;
2
3  public class Motocicleta extends Vehiculo {
4
5      private byte NumeroDePlazas;
6
7      Motocicleta() {
8          super((byte)2);
9      }
10
11     Motocicleta(byte NumeroDePlazas) {
12         super((byte)2);
13         EstableceNumeroDePlazas(NumeroDePlazas);
14     }
15
16     Motocicleta(Color color) {
17         super(color, (byte)2);
18     }
19
20     Motocicleta(Color color, short Cilindrada) {
21         super(color, (byte)2, Cilindrada);
22     }
23
24     Motocicleta(Color color, short Cilindrada,
25                 short Potencia) {
26         super(color, (byte)2, Cilindrada, Potencia);
27     }
```

```
28
29 public byte DimeNumeroDePlazas() {
30     return NumeroDePlazas;
31 }
32
33 public void EstableceNumeroDePlazas(byte NumeroDePlazas) {
34     this.NumeroDePlazas = NumeroDePlazas;
35 }
36
37 }
```

Código de la clase *PruebaVehiculos*:

```
1 import java.awt.Color;
2
3 public class PruebaVehiculos {
4     public static void main(String[] args) {
5         Motocicleta MotoBarata = new
6             Motocicleta(Color.red,(short)125,(short)25);
7         Motocicleta MotoBarata2 = new
8             Motocicleta(Color.red,(short)125,(short)25);
9         Motocicleta MotoCara = new
10             Motocicleta(Color.yellow,(short)1000,(short)90);
11
12         Camion CamionNormal = new Camion(Color.red,(byte)4,
13             (byte)2,(short)4000,(short)300);
14         Camion CamionEnorme = new Camion(Color.red,(byte)24,
15             (byte)6,(short)15000,(short)800);
16
17         MotoBarata.EstableceNumeroDePlazas((byte)1);
18         System.out.println(MotoBarata.DimeNumeroDePlazas());
19         System.out.println(MotoBarata2.DimeCilindrada());
20         System.out.println(CamionEnorme.DimeNumeroDeEjes());
21     }
22 }
23 }
```

Resultado:

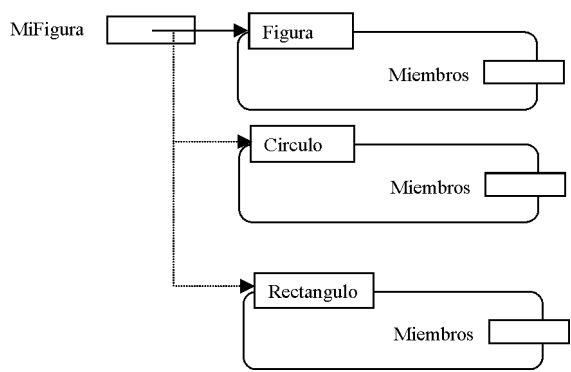


```
MS-DOS
C:\CursoJava\Introduccion>java PruebaVehiculos
1
125
6
C:\CursoJava\Introduccion>
```

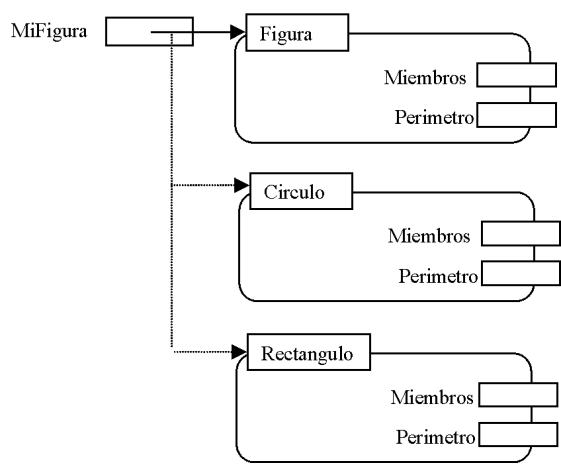
## 5.3 POLIMORFISMO

### 5.3.1 Conceptos

El polimorfismo, desde un punto de vista de orientación a objetos, permite que clases de diferentes tipos puedan ser referenciadas por una misma variable. En el siguiente ejemplo, una instancia de la clase *Figura* (*MiFigura*) podrá referenciar a propiedades y métodos implementados en las clases *Circulo* y *Rectángulo*.



Pongamos como ejemplo que las clases *Circulo* y *Rectangulo* implementan un método llamado *Perimetro* que calcula el perímetro de la figura geométrica:



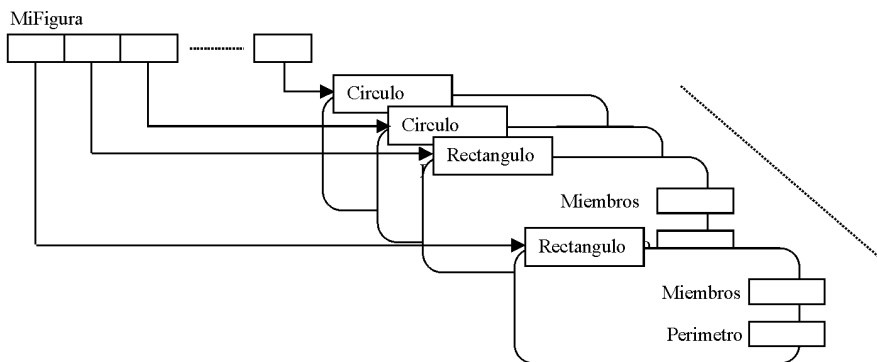
La referencia *MiFigura* (de tipo *Figura*) podrá invocar al método *Perimetro* de la clase *Circulo* o al método *Perimetro* de la clase *Rectangulo* , dependiendo de, si

en tiempo de ejecución, está referenciando a una instancia de la clase *Circulo* o está referenciando a una instancia de la clase *Rectangulo*.

Es importante resaltar que la determinación del método *Perimetro* que se va a invocar (el de la clase *Circulo* o el de la clase *Rectangulo*) no se realiza en la compilación, sino en tiempo de ejecución, lo que hace del polimorfismo un mecanismo de programación muy potente.

Para entender mejor la utilidad del polimorfismo, pongamos un ejemplo en el que su utilización es adecuada: supongamos que estamos realizando un editor gráfico en el que el usuario puede dibujar figuras geométricas bidimensionales tales como círculos, rectángulos, pentágonos, triángulos, etc. Además, dentro de las opciones de la herramienta, existe un botón que al ser pulsado nos muestra en una ventana el valor del perímetro de la figura seleccionada.

Nosotros no conocemos a priori las figuras que el usuario dibujará, eso es algo que ocurre en tiempo de ejecución, al igual que no podemos saber por adelantado de qué figura o figuras solicitará conocer el perímetro. En un momento dado, mientras un usuario utiliza el editor gráfico, existirá una estructura de datos que contendrá las figuras dibujadas:



El vector *MiFigura* contendrá referencias a objetos de tipo *Figura* (*Figura[] MiFigura = new Figura[100]*). Según el usuario, en tiempo de ejecución, seleccione una figura geométrica u otra se invocará a un método *Perimetro* u otro. Esto es posible gracias al polimorfismo.

El polimorfismo, en Java, debe implementarse respetando a las siguientes reglas:

- La invocación a un método de un subclase debe realizarse a través de una referencia a la superclase; es decir, es adecuado poner `p = LaFigura.Perimetro()`, pero no ~~`p = ElCirculo.Perimetro()`~~.
- El método llamado debe ser también un miembro de la superclase; en nuestro ejemplo, el método *Perimetro* debe existir también en la clase *Figura*.
- La firma del método debe ser igual en la superclase que en las clases derivadas; es decir, el nombre del método y el número y tipos de sus parámetros debe ser igual en las clases *Figura*, *Circulo*, *Rectangulo*, etc.
- El tipo de retorno del método (que se utilizará de forma polimórfica) debe ser igual en la superclase que en las subclases.
- El atributo de acceso del método no debe ser más restrictivo en las clases derivadas que en la superclase.

### 5.3.2 Ejemplo

Para acabar de ilustrar la forma de utilización del polimorfismo, vamos a ampliar las clases *Figura*, *Circulo* y *Rectangulo* implementadas en la lección anterior, de manera que contengan un método *Perimetro* que calcule el perímetro de cada tipo de figura (círculos y rectángulos):

La clase *Figura* debe contener un método *Perimetro* para satisfacer las reglas de utilización del polimorfismo en Java. Nótese que nunca se va a ejecutar el código de este método, se ejecutará el código de los métodos *Perimetro* de las subclases. Cuando expliquemos clases abstractas mejoraremos este aspecto.

```
1  import java.awt.Color;
2
3  public class PolimorfismoFigura {
4
5      // Los mismos miembros que en la clase Figura,
6      // modificando el nombre de los constructores
7
8      public double Perimetro() {
9          return 0d;
10     }
11
12 }
```

El método *Perimetro* de la clase *PolimorfismoCirculo* calcula el perímetro ( $2\pi r$ ) del círculo:

```
1  import java.awt.Color;
2
3  public class PolimorfismoCirculo extends PolimorfismoFigura
4  {
5      // Los mismos miembros que en la clase Circulo,
6      // modificando el nombre de los constructores
7
8      public double Perimetro() {
9          return 2.0d*Math.PI*Radio;
10     }
11
12 }
```

El método *Perimetro* de la clase *PolimorfismoRectangulo* calcula el perímetro ( $2*\text{lado1}+2*\text{lado2}$ ) del rectángulo:

```
1  import java.awt.Color;
2
3  public class PolimorfismoRectangulo extends
4                                     PolimorfismoFigura {
5      // Los mismos miembros que en la clase Rectangulo,
6      // modificando el nombre de los constructores
7
8      public double Perimetro() {
9          return 2d*Lados[0] + 2d*Lados[1];
10     }
11
12 }
```

En la clase *PolimorfismoPruebaFiguras* se utiliza el recurso del polimorfismo. En la línea 7 se declara una propiedad *MiCirculo* de tipo *PolimorfismoFigura*, pero instanciando la clase *PolimorfismoCirculo*. En la línea 9 se hace lo mismo con la referencia *MiRectangulo*, que siendo una instancia de la clase *PolimorfismoRectangulo*, es de tipo *PolimorfismoFigura*.

En la línea 12, cuando se invoca al método *Perimetro* con la referencia *MiCirculo*, en tiempo de ejecución se determina que la propiedad *MiCirculo* (de tipo *PolimorfismoFigura*) apunta a una instancia de la clase *PolimorfismoCirculo*, por lo que se invoca al método *Perimetro* de dicha clase (y no de *PolimorfismoRectangulo* o *PolimorfismoFigura*).



En la línea 13 ocurre lo mismo que en la 12, salvo que ahora se invoca al método *Perimetro* de la clase *PolimorfismoRectangulo*.

Ya fuera de lo que es el concepto de polimorfismo, cuando intentamos hacer una llamada a un método de una clase derivada utilizando una referencia de la superclase, nos da error, puesto que desde una superclase no se tiene visibilidad de los miembros de las subclases (aunque al contrario no haya problemas). Es decir no podemos poner *MiCirculo.DimeRadio()*, porque el método *DimeRadio* pertenece a la clase *PolimorfismoCirculo* y *MiCirculo* es una propiedad del tipo *PolimorfismoFigura*.

Si queremos conocer el radio de *MiCirculo*, tenemos dos posibilidades:

- Crear una instancia de la clase *PolimorfismoCirculo* y asignarle la referencia de *MiCirculo* (utilizando el mecanismo de *casting*). Esta solución se implementa en la línea 15 del código mostrado.
- Utilizar el mecanismo de *casting* sin crear explícitamente una instancia de la clase *PolimorfismoCirculo*. Esta solución se implementa en la línea 17.

```
1  import java.awt.Color;
2
3  public class PolimorfismoPruebaFiguras {
4      public static void main(String[] args) {
5          int[] Posicion = {10,20};
6          double[] Lados = {50d,100d};
7          PolimorfismoFigura MiCirculo = new
8              PolimorfismoCirculo(3d,Color.red,Posicion);
9          PolimorfismoFigura MiRectangulo = new
10             PolimorfismoRectangulo(Lados,Color.blue,Posicion);
11
12         System.out.println(MiCirculo.Perimetro());
13         System.out.println(MiRectangulo.Perimetro());
14
15         PolimorfismoCirculo InstanciaCirculo =
16             (PolimorfismoCirculo) MiCirculo;
17         System.out.println(InstanciaCirculo.DimeRadio());
18         System.out.println(((PolimorfismoCirculo)MiCirculo).
19             DimeRadio());
20     }
```

## 5.4 CLASES ABSTRACTAS E INTERFACES

### 5.4.1 Métodos abstractos

Un método abstracto es un método que se declara, pero no se define. La sintaxis es la misma que la de la declaración de un método no abstracto, pero terminando con punto y coma y sin poner las llaves de comienzo y final del método; además se utiliza la palabra reservada *abstract*.

Por ejemplo, uno de los métodos *Perimetro*, vistos anteriormente se declaraba y definía como:

```
public double Perimetro() {  
    return 2.0d*Math.PI*Radio;  
}
```

Este mismo método, declarado como abstracto adopta la forma:

```
public abstract double Perimetro();
```

Cuando hacemos uso del polimorfismo nos vemos obligados a definir un método en la superclase, sabiendo que nunca será llamado. En la lección anterior, en la clase *PolimorfismoFigura* escribíamos el método:

```
public double Perimetro() {  
    return 0d;  
}
```

Resultaría mucho más elegante utilizar la versión abstracta del método en casos como este.

Una puntualización interesante respecto a los métodos abstractos es que no deben ser definidos como *final*, puesto que los métodos *final* no pueden ser redefinidos en las clases derivadas, por lo tanto, si escribimos un método abstracto como *final*, las subclases siempre tendrán ese método como abstracto y nunca podrán ser instanciadas.

### 5.4.2 Clases abstractas

Cuando una clase contiene al menos un método abstracto, la clase es abstracta y debe declararse como tal:

```
public abstract class ClaseAbstracta {  
.....  
}
```

Se pueden declarar variables de clases abstractas, pero no instanciarlas:

```
ClaseAbstracta VariableClase;  
VariableClase = new ClaseAbstracta();
```

Pueden utilizarse clases abstractas como superclases:

```
public abstract class ClaseAbstractaDerivada extends ClaseAbstracta {  
    // podemos definir parte de los metodos abstractos  
}  
  
public class ClaseNoAbstracta extends ClaseAbstractaDerivada {  
    // definimos todos los metodos abstractos que tenga la superclase  
}  
  
ClaseNoAbstracta MiInstancia= new ClaseNoAbstracta();
```

Las clases abstractas proporcionan un mecanismo muy potente para facilitar el diseño y programación orientado a objetos; podemos diseñar aplicaciones que contengan una serie de clases abstractas y codificar las mismas sin entrar en la definición de los detalles del código de los métodos. La aplicación queda de esta manera bien estructurada, definida y consistente (podemos compilarla), a partir de este punto de partida resulta mucho más sencilla la fase de implementación, que puede ser llevada a cabo en paralelo por diversos programadores, conociendo cada uno los objetos que tiene que codificar y las clases relacionadas que puede emplear.

Veamos el ejemplo de los vehículos (motocicletas y camiones) empleando una clase abstracta. Crearemos la clase abstracta *AbstractoVehiculo* y dos clases derivadas: *AMotocicleta* y *ACamion*. La clase *AbstractoVehiculo* contendrá las propiedades necesarias para albergar el color, número de ruedas, cilindrada y potencia. Podremos inicializar estas propiedades a través de diferentes constructores

y consultarlas a través de métodos. También existirá un método abstracto *Impuesto* que será definido en cada subclase derivada.

El código de ejemplo realizado para la clase *Vehiculo* es:

```
1 import java.awt.Color;
2
3 abstract public class AbstractoVehiculo {
4
5     // Los mismos constructores y metodos que en la clase
6     // Vehiculo
7     // Tambien cambiamos el nombre de los constructores
8
9     abstract public float Impuesto();
10 }
```

El método *Impuesto* se deja abstracto en la superclase y se define en cada una de las subclases según sea necesario. El impuesto que pagan los camiones es diferente al que pagan las motocicletas, al igual que el perímetro de un círculo se calcula de manera diferente al de un rectángulo.

La clase derivada *ACamion* implementa el método *Impuesto*; esta clase no es abstracta, puesto que no contiene ningún método abstracto. Supondremos que el impuesto que pagan los camiones se calcula realizando el sumatorio de los términos: cilindrada/30, potencia\*20, Número de ruedas \*20 y número de ejes \* 50. El código de la clase nos queda de la siguiente manera:

```

1  import java.awt.Color;
2
3  public class ACamion extends AbstractoVehiculo {
4
5      // Los mismos constructores y metodos que en la clase
6      // Camion
7      // Tambien cambiamos el nombre de los constructores
8
9      public float Impuesto(){
10         return (super.DimeCilindrada())/30 +
11             super.DimePotencia()*20 +
12             super.DimeNumRuedas()*20 +
13             DimeNumeroDeEjes()*50);
14     }
15 }

```

Obsérvese el uso de la palabra reservada *super* para identificar la superclase de la clase derivada donde se emplea; de esta manera, *this* indica la propia clase y *super* indica la superclase.

La clase derivada *AMotocicleta* es similar a *ACamion*. Supondremos que el impuesto se calcula aplicando la fórmula:  $cilindrada/30 + potencia*30$ . El código nos queda de la siguiente manera:

```
1  import java.awt.Color;
2
3  public class AMotocicleta extends AbstractoVehiculo {
4
5      // Los mismos constructores y metodos que en la clase
      // Motocicleta
      // Tambien cambiamos el nombre de los constructores
6
7      public float Impuesto(){
8          return (super.DimeCilindrada()/30 +
9                  super.DimePotencia()*30);
10     }
11
12 }
```

Para probar las clases anteriores empleamos el fichero *APruebaVehiculos.java* que contiene declaraciones e instanciaciones de diversos objetos camión y motocicleta. Obsérvese como en la línea 5 se declara una variable de tipo *AbstractoVehiculo* (clase abstracta) y se instancia como tipo *AMotocicleta* (clase no abstracta), con el fin de poder utilizar posteriormente el método *Impuesto* de manera polimórfica. En las líneas 7, 9, 11 y 14 se realizan acciones similares a la descrita.

```
1  import java.awt.Color;
2
3  public class APruebaVehiculos {
4      public static void main(String[] args) {
5          AbstractoVehiculo MotoBarata =
6              new AMotocicleta(Color.red, (short)125, (short)25);
7          AMotocicleta MotoBarata2 =
8              new AMotocicleta(Color.red, (short)125, (short)25);
9          AbstractoVehiculo MotoCara = new
10              AMotocicleta(Color.yellow, (short)1000, (short)90);
11
12          AbstractoVehiculo CamionNormal = new
13              ACamion(Color.red, (byte)4, (byte)2,
14                  (short)4000, (short)300);
15          AbstractoVehiculo CamionEnorme = new
```

```
15             ACamion(Color.red,(byte)24,(byte)6,
16                 (short)15000,(short)800);
17
18     AMotocicleta InstanciaMotoBarata =
19         (AMotocicleta) MotoBarata;
20     InstanciaMotoBarata.EstableceNumeroDePlazas((byte)1);
21     System.out.println(InstanciaMotoBarata.
22         DimeNumeroDePlazas());
23     MotoBarata2.EstableceNumeroDePlazas((byte)1);
24     System.out.println(MotoBarata2.DimeNumeroDePlazas());
25     System.out.println(((ACamion)CamionEnorme).
26         DimeNumeroDeEjes());
27
28     System.out.println(MotoBarata.Impuesto());
29     System.out.println(MotoCara.Impuesto());
30     System.out.println(CamionNormal.Impuesto());
31     System.out.println(CamionEnorme.Impuesto());
32     System.out.println(InstanciaMotoBarata.Impuesto());
33 }
34 }
```

### 5.4.3 Interfaces

Los interfaces son colecciones de constantes y métodos abstractos. Los métodos son siempre públicos y abstractos (no es necesario definirlos como tal) y las constantes son siempre públicas, estáticas y, por supuesto, *final* (tampoco es necesario especificar sus atributos de acceso y modificadores).

Los interfaces proporcionan un mecanismo de herencia múltiple que no puede ser utilizado empleando únicamente clases. Es necesario saber que este mecanismo proporcionado por Java es muy limitado respecto a las posibilidades que ofrece la herencia múltiple de la programación orientada a objetos; sin embargo, permite afrontar de manera elegante diferentes situaciones de diseño y programación.

La sintaxis de definición de un interfaz es:

```
public interface Nombre {
    // constantes y metodos abstractos
}
```

Ejemplo de interfaz que contiene solo constantes:

```
1 public interface DatosCentroDeEstudios {
2     byte NumeroDePisos = 5;
3     byte NumeroDeAulas = 25;
4     byte NumeroDeDespachos = 10;
5     // resto de datos constantes
6 }
```

Ejemplo de interfaz que contiene solo métodos:

```
1 public interface CalculosCentroDeEstudios {
2     short NumeroDeAprobados(float[] Notas);
3     short NumeroDeSuspensos(float[] Notas);
4     float NotaMedia(float[] Notas);
5     float Varianza(float[] Notas);
6     // resto de metodos
7 }
```

Ejemplo de interfaz con constantes y métodos:

```
public interface CentroDeEstudios {
    byte NumeroDePisos = 5;
    byte NumeroDeAulas = 25;
    byte NumeroDeDespachos = 10;
    // resto de datos constantes

    short NumeroDeAprobados(float[] Notas);
    short NumeroDeSuspensos(float[] Notas);
    float NotaMedia(float[] Notas);
    float Varianza(float[] Notas);
    // resto de metodos
}
```

También podemos definir un interfaz basado en otro. Utilizamos la palabra reservada *extends*, tal y como hacemos para derivar subclases:

```
public interface CentroDeEstudios extends DatosCentroDeEstudios {
    short NumeroDeAprobados(float[] Notas);
    short NumeroDeSuspensos(float[] Notas);
    float NotaMedia(float[] Notas);
    float Varianza(float[] Notas);
    // resto de metodos
```

```
}
```

Al igual que hablamos de superclases y subclases, podemos emplear la terminología superinterfaces y subinterfaces. A diferencia de las clases, un interfaz puede extender simultáneamente a varios superinterfaces, lo que supone una aproximación a la posibilidad de realizar herencia múltiple:

```
1 public interface CentroDeEstudios extends
2     DatosCentroDeEstudios, CalculosCentroDeEstudios {
3     // otros posibles metodos y constantes
4 }
```

Para poder utilizar los miembros de un interfaz es necesario implementarlo en una clase; se emplea la palabra reservada *implements* para indicarlo:

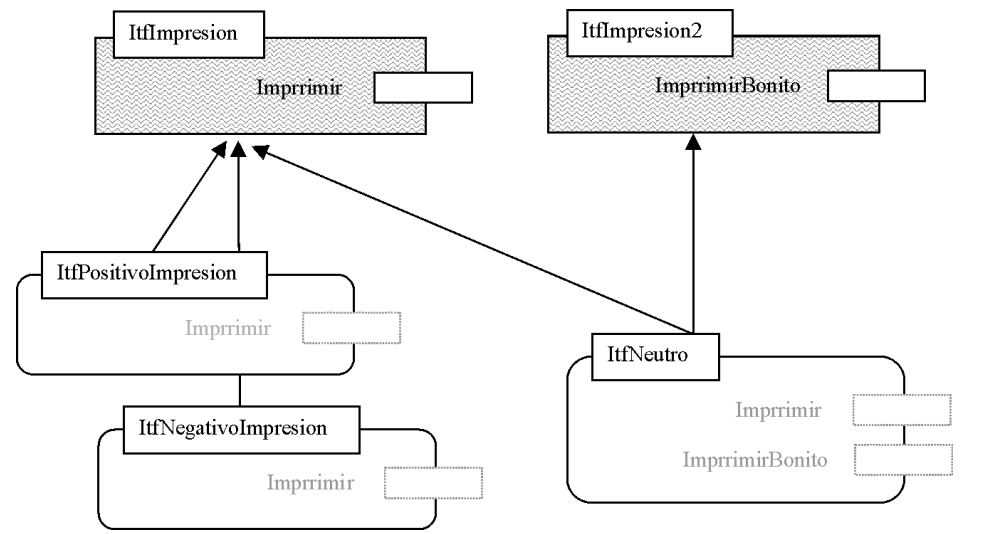
```
1 public class CCentroDeEstudios implements
2     CentroDeEstudios {
3
4     public short NumeroDeAprobados(float[] Notas) {
5         short NumAprobados = 0;
6         for (int i=0; i<Notas.length; i++)
7             if (Notas[i]>=5.0f)
8                 NumAprobados++;
9         return NumAprobados;
10    }
11
12    public short NumeroDeSuspendidos(float[] Notas) {
13        short NumSuspendidos = 0;
14        for (int i=0; i<Notas.length; i++)
15            if (Notas[i]<5.0f)
16                NumSuspendidos++;
17        return NumSuspendidos;
18    }
19
20    public float NotaMedia(float[] Notas) {
21        float Suma = 0;
22        for (int i=0; i<Notas.length; i++)
23            Suma = Suma + Notas[i];
24        return Suma/(float)Notas.length;
25    }
26
27    public float Varianza(float[] Notas) {
28        float Media = NotaMedia(Notas);
29        float Suma = 0;
30        for (int i=0; i<Notas.length; i++)
```



```
31         Suma = Suma + Math.abs(Media-Notas[i]);
32     return Suma/(float)Notas.length;
33 }
34
35 }
```

5.4.4 Ejercicio

Vamos a definir dos interfaces (*ItfImpresion* e *ItfImpresion2*) extremadamente sencillos. Cada uno de ellos consta de un solo método, diseñado para imprimir un texto: (*Imprimir* e *ImprimirBonito*). Después crearemos una serie de clases para practicar con el mecanismo de implementación de interfaces. El diagrama de clases e interfaces presenta el siguiente aspecto:



En el diagrama superior, los dos bloques con entramado de ondas representan a los interfaces, mientras que los tres bloques sin entramado representan a las distintas clases que implementan a los interfaces. Obsérvese como la clase *ItfNeutro* utiliza el mecanismo de herencia múltiple (implementando simultáneamente dos interfaces).

El interfaz *ItfImpresion* se utiliza como superinterfaz de diversas clases, al igual que una clase puede ser superclase de varias clases derivadas.

El código de los interfaces es:

```
1 public interface ItfImpresion {
2     void Imprimir();
3 }

1 public interface ItfImpresion2 {
2     void ImprimirBonito();
3 }
```

El código de las clases *ItfPositivoImpresion* e *ItfNegativoImpresion* se ha mantenido lo más sencillo posible:

```
1 public class ItfPositivoImpresion implements ItfImpresion {
2     public void Imprimir() {
3         System.out.println ("!Que buen tiempo hace!");
4     }
5 }

1 public class ItfNegativoImpresion implements ItfImpresion {
2     public void Imprimir() {
3         System.out.println ("!Odio los lunes!");
4     }
5 }
```

Para probar el funcionamiento de las clases, se codifica *ItfPrueba*:

```
1 public class ItfPrueba {
2     public static void main(String[] args){
3         ItfImpresion Positivo = new ItfPositivoImpresion();
4         ItfImpresion Negativo = new ItfNegativoImpresion();
5
6         Positivo.Imprimir();
7         Negativo.Imprimir();
8     }
9 }
```

En la línea 3 se declara una variable *Positivo* de tipo *ItfImpresion* (interfaz) y se instancia como *ItfPositivoImpresion* (clase); en la línea 4 se hace lo mismo con la variable *Negativo*. Estamos utilizando el mismo mecanismo que cuando

declaramos una variable de tipo superclase abstracta y la instanciamos como clase derivada.

En las líneas 6 y 7 se explota el polimorfismo que proporciona Java, haciendo llamadas al mismo método *Imprimir* definido de manera diferente en cada una de las clases que implementan el interfaz *ItfImpresion*. En el resultado, como cabe esperar, primero aparece *!Que buen tiempo hace!*, y posteriormente *!Odio los lunes!*

La clase *ItfNeutro* implementa los dos interfaces, consiguiendo herencia múltiple (de métodos abstractos):

```
1 public class ItfNeutro implements ItfImpresion,
                                   ItfImpresion2 {
2
3     public void Imprimir(){
4         System.out.print ("Las olas del mar");
5     }
6
7     public void ImprimirBonito(){
8         System.out.print("---- ");
9         Imprimir();
10        System.out.println(" ----");
11    }
12
13 }
```

La clase *ItfPrueba2* crea una instancia de la clase *ItfNeutro* (línea 3) y otra de la clase *ItfNegativoImpresion* (línea 4). Posteriormente, en las líneas 6, 7 y 8, se hace uso del polimorfismo (llamando al método *Imprimir* definido en clases diferentes) y de la posibilidad de acceso a los métodos *Imprimir* e *ImprimirBonito* diseñados en los interfaces *ItfImpresion* e *ItfImpresion2*.

```
1 public class ItfPrueba2 {
2     public static void main(String[] args){
3         ItfNeutro Instancia = new ItfNeutro();
4         ItfImpresion Negativo = new ItfNegativoImpresion();
5
6         Instancia.Imprimir();
7         Instancia.ImprimirBonito();
8         Negativo.Imprimir();
9     }
10 }
```

## EXCEPCIONES

---

### 6.1 EXCEPCIONES PREDEFINIDAS

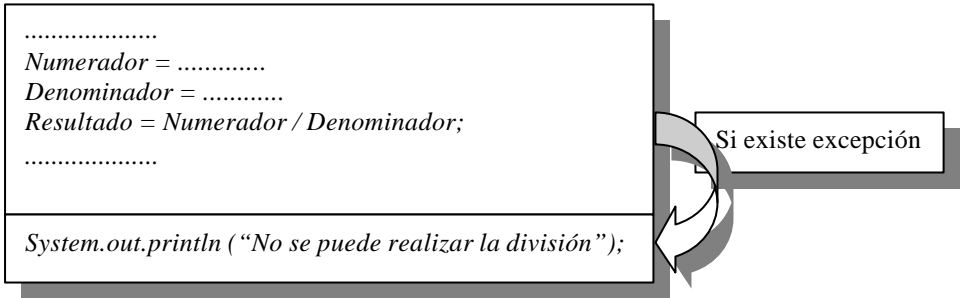
#### 6.1.1 Introducción

Las excepciones señalan errores o situaciones poco habituales en la ejecución de un programa, por ejemplo una división de un valor entre cero, un intento de acceso a un *String* declarado, pero no instanciado, etc.

Habitualmente, en programación, se incluyen tantas instrucciones condicionales como sea necesario para conseguir que una aplicación sea robusta, de esta manera, por ejemplo, en cada división de un valor entre una variable, antes se comprueba que el denominador no sea cero:

```
.....  
Denominador = .....  
if (Denominador != 0) {  
    Numerador = .....  
    Resultado = Numerador / Denominador;  
}  
else  
    System.out.println ("No se puede realizar la división");  
.....
```

Utilizando el mecanismo de excepciones que proporciona Java, en nuestro ejemplo, en lugar de incluir una serie de instrucciones condicionales para evitar las distintas divisiones entre cero que se puedan dar, se escribe el programa sin tener en cuenta esta circunstancia y, posteriormente, se escribe el código que habría que ejecutar si la situación “excepcional” se produce:



Al hacer uso de excepciones, el bloque que codifica la porción de aplicación resulta más sencillo de entender, puesto que no es necesario incluir las instrucciones condicionales que verifican si puede darse la situación de excepción. En definitiva, si utilizamos el mecanismo de excepciones, podemos separar la lógica del programa de las instrucciones de control de errores, haciendo la aplicación más legible y robusta.

Las excepciones son objetos (clases) que se crean cuando se produce una situación extraordinaria en la ejecución del programa. Estos objetos almacenan información acerca del tipo de situación anormal que se ha producido y el lugar donde ha ocurrido. Los objetos excepción se pasan automáticamente al bloque de tratamiento de excepciones (el inferior de nuestro gráfico) para que puedan ser referenciados.

La superclase de todas las excepciones es la clase *Throwable*. Sólo las instancias de esta clase o alguna de sus subclasses pueden ser utilizadas como excepciones. La clase *Throwable* tiene dos clases derivadas: *Error* y *Exception*.

La clase *Exception* sirve como superclase para crear excepciones de propósito específico (adaptadas a nuestras necesidades), por ejemplo, si estamos diseñando una clase que lee secuencialmente bytes en una cinta digital de datos, podemos crear la excepción: *FinDeCinta* que se produce cuando el dispositivo físico ha alcanzado el final de la cinta. Otro ejemplo podría ser la implementación de una clase de envío de datos a un satélite no geostacionario, donde convendría incluir una excepción *FueraDeCobertura* que se produzca cuando el satélite se encuentre fuera del alcance de nuestra antena parabólica. Un último ejemplo: si escribimos un driver de impresora podemos crear una clase derivada de *Exception* para que nos avise de la situación excepcional *FinDePapel*.

La clase *Error* sirve de superclase para una serie de clases derivadas ya definidas que nos informan de situaciones anormales relacionadas con errores de muy difícil recuperación producidos en el sistema.

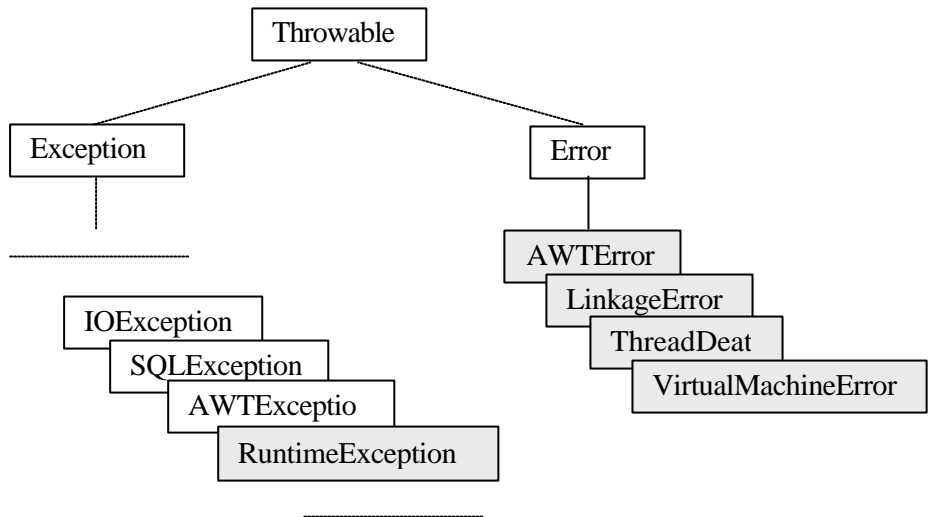
No es obligatorio tratar las excepciones derivadas de la clase *Error*, ya que en la ejecución de una aplicación probablemente nunca se produzcan estas situaciones anormales, sin embargo, sí que es obligatorio hacer un tratamiento explícito de las excepciones derivadas de la clase *Exception* (no podemos ignorar la falta de cobertura de un satélite, el fin del papel en una impresora, el fin de datos en una lectura secuencial, etc.). En una aplicación bien diseñada e implementada es muchísimo más probable que se produzcan excepciones de tipo *Exception* que excepciones de tipo *Error*.

La clase *Exception* tiene un amplio número de clases derivadas proporcionadas por el SDK, por ejemplo existen excepciones predefinidas para el uso de ficheros, de SQL, etc. De todas estas subclases, *RuntimeException* tiene una característica propia: no es necesario realizar un tratamiento explícito de estas excepciones (de todas las demás clases derivadas de *Exception* si es necesario). Esto es debido a que, al igual que con las excepciones derivadas de *Error*, existen pocas posibilidades de recuperar situaciones anómalas de este tipo.

Entre las clases derivadas de *RuntimeException* se encuentran:

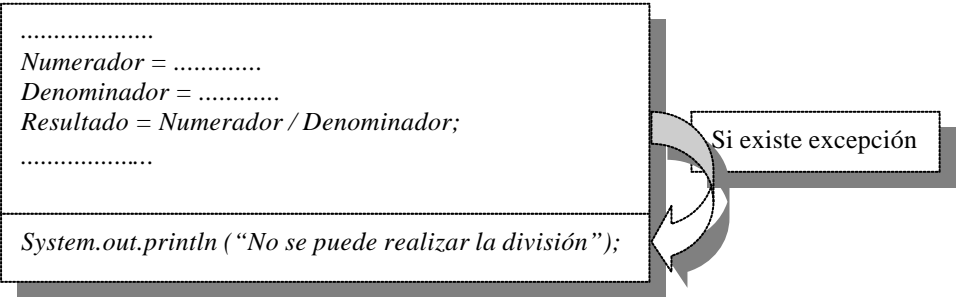
Clase	Situación de excepción
ArithmeticException	Cuando ocurre una operación aritmética errónea, por ejemplo división entre cero; con los valores reales no se produce esta excepción.
ArrayStoreException	Intento de almacenar un valor de tipo erróneo en una matriz de objetos
IllegalArgumentException	Se le ha pasado un argumento ilegal o inapropiado a un método
IndexOutOfBoundsException	Cuando algún índice (por ejemplo de array o String) está fuera de rango
NegativeArraySizeException	Cuando se intenta crear un array con un índice negativo
NullPointerException	Cuando se utiliza como apuntador una variable con valor null

El siguiente gráfico muestra las clases más importantes en el uso de excepciones y su jerarquía. En las clases no sombreadas es obligatorio realizar un tratamiento explícito de las excepciones, en las clases sombreadas no es necesario este tratamiento



6.1.2 Sintaxis y funcionamiento con una sola excepción

En el apartado anterior adelantábamos el diseño básico del tratamiento de excepciones: separar el código de los programas del código de control de situaciones excepcionales en los programas, y los ilustrábamos gráficamente con el siguiente ejemplo:



El bloque superior representa el código del programa y se le denomina bloque de "intento" (try). Aquí se introduce el código del programa susceptible de causar cierto tipo de excepciones. Las instrucciones se incluyen en un bloque con la siguiente sintaxis:

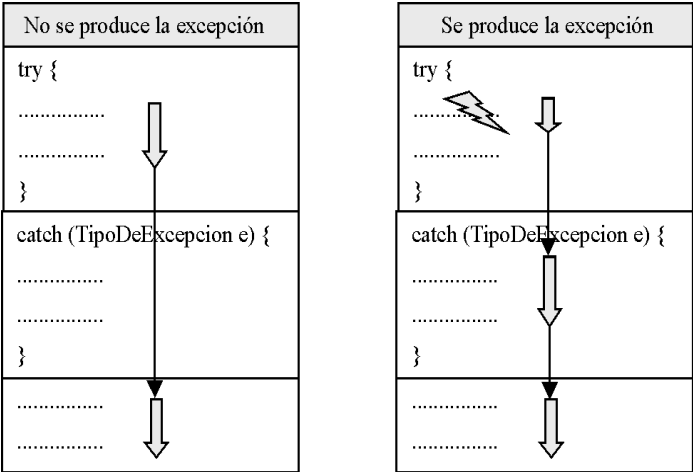
```
try {  
    // instrucciones susceptibles de causar cierto tipo de excepciones  
}
```

El bloque inferior contiene las instrucciones de control de situaciones excepcionales. La sintaxis es:

```
catch (TipoDeExcepcion Identificador) {  
    // instrucciones de control de situaciones excepcionales  
}
```

Entre el bloque `try` y el bloque `catch` no puede haber ninguna instrucción. Ambos bloques se encuentran ligados en ejecución.

El funcionamiento es el siguiente: se ejecutan las instrucciones del bloque `try` hasta que se produzca la situación de excepción de tipo `TipoDeExcepcion`; nótese que, habitualmente, no se producirá la excepción y se ejecutarán todas las instrucciones del bloque `try`. Si se da la excepción, se pasa a ejecutar las instrucciones del bloque `catch` y, posteriormente, las instrucciones siguientes al bloque `catch`. Si no se produce la excepción no se ejecutan las instrucciones del bloque `catch`. Gráficamente:





### 6.1.3 Ejemplo con una sola excepción

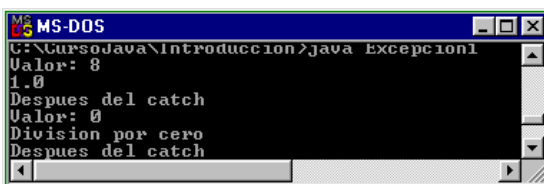
En este ejemplo capturaremos la excepción *ArithmeticException*, que es una subclase de *RuntimeException*; aunque no es obligatorio tratar las excepciones que derivan de *RuntimeException*, en ciertos casos resulta conveniente.

La clase *Excepcion1* implementa un bucle sin fin (líneas 3 y 18) que recoge un valor entero suministrado por el usuario (línea 7) y lo utiliza como denominador en una división (línea 8). Todo ello dentro de un bloque *try* (líneas 5 y 10).

El bloque *catch* atiende únicamente a excepciones de tipo *ArithmeticException* (línea 12), imprimiendo el mensaje “Division por cero” en la consola (línea 13). La instrucción situada en la línea 16 nos muestra como se ejecuta el programa tras los bloques ligados *try-catch*.

```
1 public class Excepcion1 {
2     public static void main(String[] args) {
3         do {
4
5             try {
6                 System.out.print("Valor: ");
7                 int Valor = Teclado.Lee_int();
8                 float Auxiliar = 8/Valor;
9                 System.out.println(Auxiliar);
10            }
11
12            catch (ArithmeticException e) {
13                System.out.println("Division por cero");
14            }
15
16            System.out.println("Despues del catch");
17
18        } while (true);
19    }
20 }
```

Al ejecutar el programa e introducir diversos valores, obtenemos la siguiente salida:



```
MS-DOS
C:\CursoJava\Introduccion>java Excepcion1
Valor: 8
1.0
Despues del catch
Valor: 0
Division por cero
Despues del catch
```

Si introducimos un valor distinto de cero se ejecutan todas las instrucciones del bloque *try*, ninguna del *catch* (no se produce la excepción) y la instrucción siguiente al *catch*. Si introducimos el valor cero se ejecutan las instrucciones del *try* hasta que se produce la excepción (línea 8), por lo que la línea 9 no se ejecuta; posteriormente se ejecutan las instrucciones del *catch* y finalmente las posteriores al bloque *catch*.

### 6.1.4 Sintaxis y funcionamiento con más de una excepción

Una porción de código contenida en un bloque *try*, potencialmente puede provocar más de una excepción, por lo que Java permite asociar varios bloques *catch* a un mismo bloque *try*. La sintaxis es la siguiente:

```
try {  
    // instrucciones susceptibles de causar cierto tipo de excepciones  
}  
catch (TipoDeExcepcion1 Identificador) {  
    // instrucciones que se deben ejecutar si ocurre la excepcion de tipo  
    TipoDeExcepcion1  
}  
catch (TipoDeExcepcion2 Identificador) {  
    // instrucciones que se deben ejecutar si ocurre la excepcion de tipo  
    TipoDeExcepcion2  
}  
.....  
catch (TipoDeExcepcionN Identificador) {  
    // instrucciones que se deben ejecutar si ocurre la excepcion de tipo  
    TipoDeExcepcionN  
}
```

No se pueden poner instrucciones entre los bloques *catch*.

Cuando “se levanta” (ocurre) una excepción en las instrucciones del bloque *try*, se recorren en orden los bloques *catch* hasta que se encuentra uno con el mismo tipo o un tipo que es superclase de la excepción que ha llegado. Se ejecutan únicamente las instrucciones del bloque *catch* que cumple los requisitos (si es que alguno los cumple).

De esta manera sería correcto implementar la secuencia:

```
try {  
    //instrucciones susceptibles de causar cierto tipo de excepciones  
}  
catch (NullPointerException e) {  
    //instrucciones que se deben ejecutar si ocurre la excepcion de tipo  
    NullPointerException  
}  
catch (RuntimeException e) {  
    //instrucciones que se deben ejecutar si ocurre la excepcion de tipo  
    RunTimeException  
}  
catch (Exception e) {  
    //instrucciones que se deben ejecutar si ocurre la excepcion de tipo Exception  
}
```

Si ocurre una excepción del tipo *NullPointerException* se ejecutan las instrucciones asociadas al primer bloque *catch*. Si ocurre una excepción del tipo *RunTimeException* o de una clase derivada del mismo (distinta a *NullPointerException*) se ejecutan las instrucciones asociadas al segundo bloque *catch*. Si ocurre una excepción del tipo *Exception* o de una clase derivada del mismo (distinta a *RuntimeException*) se ejecutan las instrucciones asociadas al tercer bloque *catch*.

Si cambiásemos el orden (en cualquiera de sus posibles combinaciones) de los bloques *catch* del ejemplo nos encontraríamos con un error de compilación, puesto que al menos un bloque *catch* no podría ejecutarse nunca.

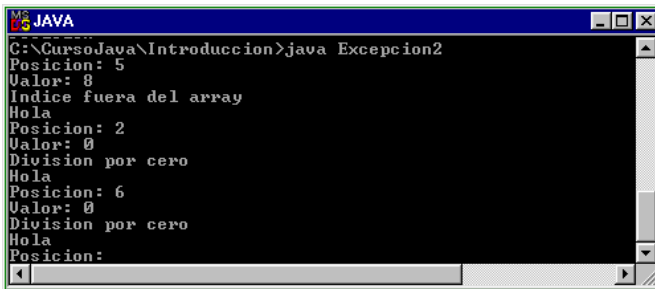
Resulta habitual utilizar varios bloques *catch* correspondientes a excepciones con un mismo nivel de herencia, por ejemplo con subclases de *RuntimeException*:

```
1 public class Excepcion2 {  
2     public static void main(String[] args) {  
3         int Posicion=0;  
4         do{  
5             try {  
6                 float[] Valores = new float[5];  
7                 System.out.print("Posicion: ");  
8                 Posicion = Teclado.Lee_int();  
9                 System.out.print("Valor: ");  
10                int Valor = Teclado.Lee_int();  
11  
12                float Auxiliar = 8/Valor;  
13                Valores[Posicion] = Auxiliar;  
14            }  
15        }  
16    }  
17 }
```

```
15
16     catch (ArithmeticException e) {
17         System.out.println("Division por cero");
18     }
19
20     catch (IndexOutOfBoundsException e) {
21         System.out.println("Indice fuera del array");
22     }
23
24     System.out.println("Hola");
25
26     } while (Posicion!=10);
27 }
28 }
```

El código de la clase *Excepcion2* contiene un bloque *try* (líneas 5 a 14) y dos bloques *catch* (líneas 16 a 18 y 20 a 22). En el bloque *try* se define un vector de 5 posiciones *float* (línea 6) y se piden dos valores enteros; el primero (línea 8) servirá de índice para almacenar un dato en la posición indicada del vector *Valores* (línea 13), el segundo valor (obtenido en la línea 10) actuará de denominador en una división (línea 12).

Como se puede comprobar, las instrucciones del bloque *try* pueden generar, al menos, dos excepciones: *ArithmeticException* e *IndexOutOfBoundsException*. Si introducimos un índice (*Posición*) mayor que 4 (el vector abarca de 0 a 4) nos encontramos con una excepción del segundo tipo, mientras que si introducimos como denominador (valor) un cero, nos encontramos con una excepción aritmética.



```
JAVA
G:\CursoJava\Introduccion>java Excepcion2
Posicion: 5
Valor: 8
Indice fuera del array
Hola
Posicion: 2
Valor: 0
Division por cero
Hola
Posicion: 6
Valor: 0
Division por cero
Hola
Posicion:
```

Como se puede observar analizando la ejecución mostrada, una vez que se produce una excepción se termina la ejecución en el bloque *try* y se pasa al bloque *catch* correspondiente (en caso de que exista).

### 6.1.5 El bloque *finally*

Como hemos visto, una vez que llega una excepción dentro de un bloque *try*, las instrucciones siguientes (en el bloque) a la que causa la excepción no se ejecutan. Esto puede dar lugar a que dejemos el sistema en un estado no deseado (ficheros abiertos, comunicaciones sin terminar, recursos bloqueados, etc.). Para evitar este tipo de situaciones, Java proporciona la posibilidad de incluir un bloque (*finally*) cuyas instrucciones siempre se ejecutan después de las del *catch* seleccionado.

La sintaxis es:

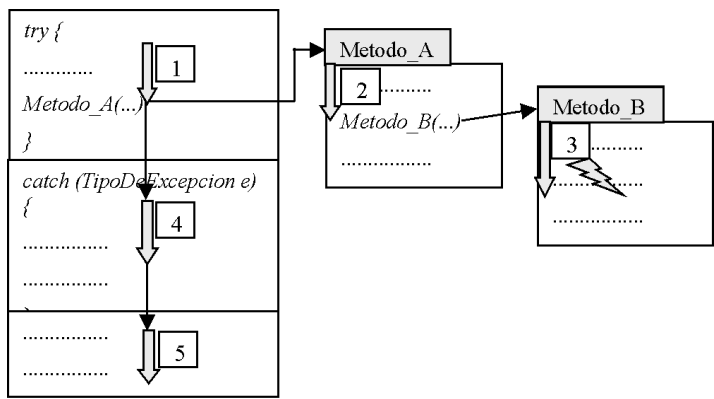
```
try {  
.....  
}  
  
catch (TipoExcepcion1 Identificador) {  
.....  
}  
.....  
catch (TipoExcepcionN Identificador) {  
.....  
}  
  
finally {  
    // instrucciones que siempre se ejecutan  
}
```

Las instrucciones del bloque *finally* siempre se ejecutan, independientemente de que se trate o no una excepción. Si en el bloque *try* existe una sentencia *return*, las instrucciones del bloque *finally* también se ejecutan antes que el *return*.

### 6.1.6 Propagación de excepciones

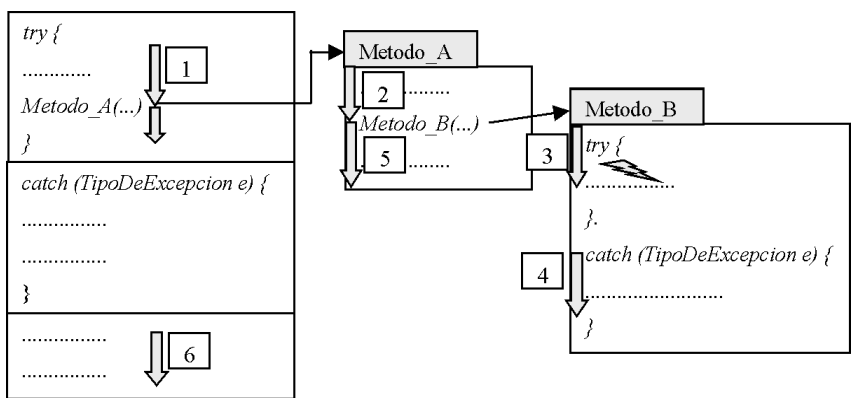
Las instrucciones que tenemos dentro de un bloque *try* a menudo contienen llamadas a métodos que a su vez pueden realizar llamadas a otros métodos y así sucesivamente. Cualquiera de los métodos llamados puede provocar una excepción y cualquiera de los métodos puede, o no, tratarla (con bloques *catch*).

Una excepción no tratada en un bloque se propaga hacia el bloque llamante. Este mecanismo de propagación continúa mientras no se trate la excepción o se llegue al método de nivel superior. Si la excepción no se trata en el método de nivel superior, se imprime un mensaje de error por consola.



En el gráfico anterior se presenta el caso de un bloque *try* que contiene una llamada a un método *Metodo\_A* que a su vez hace una llamada a *Metodo\_B*, donde se produce una excepción. Como la excepción no es tratada en *Metodo\_B* se propaga al bloque llamante (*Metodo\_A*); a su vez la excepción no es tratada en *Metodo\_A*, por lo que se propaga al bloque *try*, donde sí que es capturada por un *catch*.

En el caso de que un método capture y trate la excepción, el mecanismo de propagación se termina y la ejecución de los métodos llamados y el bloque *try* del método llamante continúan como si la excepción no se hubiera producido:



En muchas ocasiones, después de tratar la excepción a un nivel, resulta conveniente que también sea tratada a niveles superiores; por ejemplo, si estando leyendo de un fichero origen y escribiendo su contenido en un fichero destino nos encontramos con un error de lectura, además de tratar la excepción a ese nivel (cerrar los ficheros, etc.) sería deseable propagar la excepción al programa llamante para que informe al usuario y le permita hacer un nuevo intento si lo desea.

Para propagar de forma explícita una excepción se emplea la palabra reservada *throw* seguida del objeto excepción. Ejemplo:

```
try {
    // codigo
    Origen.CopiaFichero(Destino);
    // codigo
}
catch (IOException e) {
    System.out.println ("Error de lectura ¿Desea intentarlo de Nuevo?");
    .....
}

public void CopiaFichero (TipoFichero Destino) {
    try {
        // codigo
    }
    catch (IOException e) {
        // cerrar ficheros, etc.
        throw e;
    }
}
```

En el ejemplo anterior, el método *CopiaFichero*, después de tratar la excepción, la propaga (*throw*) al método llamante, que a su vez hace otro tratamiento de la excepción.

### 6.1.7 Estado de una excepción

Como hemos visto, todas las excepciones derivan de la clase *Throwable* y tienen acceso a sus dos constructores y sus 7 métodos. El estado de las instancias de esta clase se compone de un *String* que sirve de mensaje indicando las características de la excepción y una pila de ejecución que contiene la relación de métodos que se encuentran en ejecución en el momento en el que se produce la excepción.

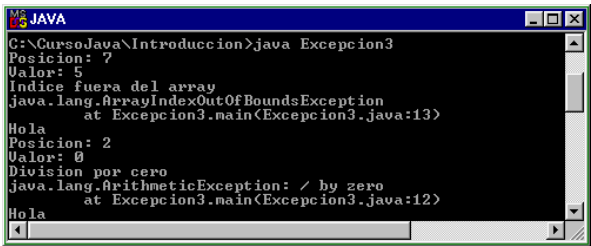
Los métodos más significativos de la clase *Throwable* son:

Método	Descripción
getMessage()	Devuelve (como <i>String</i> ) el mensaje de error almacenado en este objeto. Este mensaje se compone del nombre de la excepción más una breve referencia
printStackTrace()	Imprime por el dispositivo de salida de errores (normalmente la consola) el mensaje y la pila de ejecución almacenados en el objeto <i>Throwable</i> . Este mismo método está sobrecargado para poder sacar el resultado por diferentes salidas: <code>printStackTrace(PrintStream s)</code> y <code>printStackTrace(PrintWriter s)</code>
toString()	Devuelve (como <i>String</i> ) una descripción corta del objeto

Para ilustrar la forma de uso de estos métodos, en la clase *Excepcion2* modificamos el contenido de los bloques *catch* para imprimir la pila de ejecución:

```
16      catch (ArithmeticException e) {
17          System.out.println("Division por cero");
18          e.printStackTrace();
19      }
20
21      catch(IndexOutOfBoundsException e) {
22          System.out.println("Indice fuera del array");
23          e.printStackTrace();
24      }
```

A la nueva clase la hemos llamado *Excepcion3*. La instrucción donde se realiza la división se encuentra en la línea 12 del método *main*, mientras que la instrucción donde se asigna el valor a la posición concreta del array se encuentra en la línea 13 del método *main*. A continuación se muestra el resultado de una posible ejecución del programa:



Como se puede observar, cada instrucción *e.printStackTrace()* provoca la impresión de un breve mensaje acerca de la excepción, seguido de la pila de ejecución de los métodos del programa (en este caso sólo el método *main*) con indicación de la línea de cada método donde se ha producido la excepción.



## 6.2 EXCEPCIONES DEFINIDAS POR EL PROGRAMADOR

### 6.2.1 Introducción

Las excepciones predefinidas cubren las situaciones de error más habituales con las que nos podemos encontrar, relacionadas con el propio lenguaje y el hardware. Cuando se desarrollan aplicaciones existen otras situaciones de error de más ‘alto nivel’ relacionadas con la funcionalidad de nuestros programas.

Imaginemos una aplicación informática que controla la utilización de los remotes de una estación de esquí: los pases de acceso a los remotes son personales e intransferibles y dispondrán de un código de barras que los identifica. Cada vez que un usuario va a hacer uso de un remote debe introducir su pase de acceso en una máquina de validación, que acciona un torno y devuelve el pase. El sistema puede constar de un ordenador central al que le llegan telemáticamente los datos correspondientes a los códigos de barras de los pases que en cada momento se están introduciendo en cada máquina de validación de cada remote; si un código de barras está en regla, el ordenador envía una orden de liberar el torno para permitir al usuario acceder al remote. El ordenador central habitualmente recibirá códigos correctos utilizados en momentos adecuados, sin embargo, en ciertas ocasiones nos encontraremos con situaciones anómalas:

- 1 Código de barras ilegible
- 2 Código de barras no válido (por ejemplo correspondiente a un pase caducado)
- 3 Código de barras utilizado en otro remote en un periodo de tiempo demasiado breve
- 4 etc.

Ninguna de las situaciones anteriores se puede detectar utilizando excepciones predefinidas, puesto que su naturaleza está estrechamente relacionada con los detalles de la aplicación (no del lenguaje o el sistema informático), por lo que tendremos que recurrir al método tradicional de incluir condiciones (if, case, ...) de comprobación o bien hacer uso de excepciones definidas por nosotros mismos (excepciones no predefinidas).

Antes de nada tenemos que tener en cuenta que el mecanismo de excepciones es muy lento en ejecución comparado con la utilización de instrucciones condicionales. Aunque el mecanismo de excepciones es elegante,

debemos utilizarlo con prudencia: únicamente en situaciones que realmente son excepcionales.

La primera situación anómala de nuestro ejemplo (código de barras ilegible) podría ser tratado como excepción o bien a través de alguna instrucción condicional, depende de la frecuencia con la que se nos presente esta situación. Si ocurre, por ejemplo, en aproximadamente un 0.5% de los casos, tendremos un claro candidato a ser tratado como excepción. Si ocurre, digamos en un 14% de los casos, deberíamos pensar en tratarlo con rapidez por medio de instrucciones condicionales. Para tratar la segunda situación anómala del ejemplo (código de barras no valido) deberíamos aplicar un razonamiento equivalente al que acabamos de realizar.

Nuestro tercer caso nos presenta la situación de que un mismo pase de pistas ha podido ser duplicado (intencionadamente o por error), puesto que resulta físicamente imposible hacer uso de un remonte en un instante y volver a utilizarlo en otro remonte lejano medio minuto después. Probablemente esta situación se presenta muy rara vez y resultaría muy adecuado tratarla como excepción propia.

Una vez razonada la utilidad de este tipo de excepciones, veremos la manera de definir las en Java

## 6.2.2 Definición de una excepción definida por el programador

En programación orientada a objetos lo más adecuado es que las excepciones sean objetos, por lo que en Java definiremos las excepciones como clases. Nuestras clases de excepción, en general, heredarán de la clase *Exception*.

En las clases que nos creemos a partir de *Exception*, incluiremos al menos el constructor vacío y otro que contenga un *String* como argumento. Este *String* se inicializa automáticamente con el nombre de la clase; la inicialización se realiza en la superclase *Throwable*. El texto que pongamos al hacer uso del segundo constructor se añadirá al nombre de la clase insertado por la superclase.

A continuación se presenta una clase (*ExPropia*) muy simple que define una excepción propia:

```
1 public class ExPropia extends Exception {  
2  
3     ExPropia () {  
4         super("Esta es mi propia excepcion");  
5     }  
}
```

```
6
7  ExPropia (String s) {
8      super(s);
9  }
10
11 }
```

En la línea 1 se define nuestra clase *ExPropia* que hereda de *Exception*. La línea 3 define el constructor vacío, que en este caso añade el texto “Esta es mi propia excepcion” al nombre de la clase, quedando “ExPropia: Esta es mi propia excepcion”. La línea 7 define el constructor con un *String* como argumento; nos servirá para añadir el texto que deseemos al instanciar la clase.

### 6.2.3 Utilización de una excepción definida por el programador

Una vez que disponemos de una excepción propia, podremos programar la funcionalidad de nuestras aplicaciones provocando (lanzando) la excepción cuando detectemos alguna de las situaciones anómalas asociadas.

En el siguiente ejemplo (*ExPropiaClase*) se presenta un método (línea 2) que levanta (*throw*) la excepción *ExPropia* cuando se lee un cero (línea 4). La excepción *ExPropia* podría levantarse en diferentes secciones de código de esta u otras clases. No debemos olvidar indicar que nuestro método es susceptible de lanzar la excepción (línea 2).

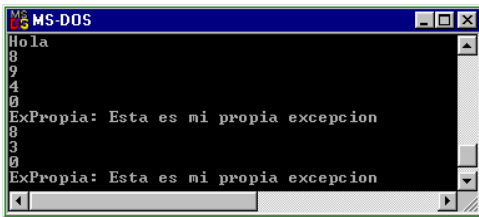
```
1  public class ExPropiaClase {
2      public void Metodo() throws ExPropia {
3          if (Teclado.Lee_int() == 0)
4              throw new ExPropia();
5      }
6      // otros metodos
7  }
```

Finalmente, en clases de más alto nivel podemos programar secciones de código que recojan la excepción que hemos creado:

```
1  public class ExPropiaPrueba {
2      public static void main (String[] args) {
3          System.out.println("Hola");
4          do
5              try {
6                  ExPropiaClase Instancia = new ExPropiaClase();
```

```
7         Instancia.Metodo();
8     }
9     catch(ExPropia e) {
10         System.out.println(e);
11     }
12     while(true);
13 }
14 }
```

Como se puede observar en la clase *ExPropiaPrueba*, podemos insertar código en un bloque *try* asociándole un bloque *catch* que recoge la excepción que hemos definido (*ExPropia*). A continuación se muestra un posible resultado de la ejecución de *ExPropiaPrueba*.



### 6.2.4 Ejemplo

En los apartados anteriores hemos definido y utilizado una excepción con sus constructores básicos, pero sin propiedades ni métodos adicionales. En ocasiones con el esquema seguido es suficiente, sin embargo, a menudo es adecuado dotar de un estado (propiedades) a la clase que define la excepción. Veamos este concepto con un ejemplo sencillo: deseamos programar la introducción por teclado de matrículas de vehículos en un país donde las matrículas se componen de 8 caracteres, siendo obligatoriamente el primero de ellos una letra. Si alguna matrícula introducida por el usuario no sigue el formato esperado se recogerá la interrupción oportuna y se escribirá un aviso en la consola.

En el código siguiente se presenta la clase *ExMatricula* que hereda la funcionalidad de *Exception* (línea 1). Los constructores situados en las líneas 7 y 10 permiten iniciar el estado de la clase a través de los constructores de su superclase.

En la línea 3 se define la propiedad *MalFormada*, que contiene información sobre la razón por la que el formato de la matrícula es incorrecto: tamaño inadecuado (línea 4) o inexistencia de la letra inicial (línea 5).

El constructor de la línea 14 permite crear una excepción *ExMatricula* indicando la naturaleza del problema que obligará a levantar (*throw*) esta excepción (*ExMatricula.MAL\_TAMANIO* o *ExMatricula.MAL\_LETRA*).

En la línea 18 se suministra un método que permitirá, a los programa que reciben la excepción, saber la razón que ha provocado la misma.

```
1 public class ExMatricula extends Exception {
2
3     private int MalFormada = 0;
4     static final int MAL_TAMANIO = -1;
5     static final int MAL_LETRA = -2;
6
7     ExMatricula()
8     {}
9
10    ExMatricula(String s) {
11        super(s);
12    }
13
14    ExMatricula(int MalFormada) {
15        this.MalFormada = MalFormada;
16    }
17
18    public int DimeProblema() {
19        return MalFormada;
20    }
21
22 }
```

Una vez creada la excepción *ExMatricula* podemos escribir el código que valida las matrículas. La siguiente clase (*ExMatriculaValidar*) realiza este cometido: el método *Validar* situado en la línea 7 comprueba el formato del parámetro suministrado y si es necesario tendrá capacidad de levantar (*throws*) la excepción *ExMatricula*.

Si la longitud de la matrícula es distinta de 8 caracteres (línea 8) se crea y levanta la excepción *ExMatricula* con estado inicial *ExMatricula.MAL\_TAMANIO* (línea 9). Si el carácter inicial no es una letra (línea 11) se crea y levanta la excepción *ExMatricula* con estado inicial *ExMatricula.MAL\_LETRA* (línea 12).

El método privado (sólo es visible en esta clase) *UnaLetra* aísla el primer carácter y comprueba que es una letra empleando el método *matches* y la expresión regular “[A-Za-z]”. La utilización de expresiones regulares se incluye a partir de la versión 1.4 del JDK.

```
1 public class ExMatriculaValidar {
2
3     private boolean UnaLetra(String Matricula) {
4         return Matricula.substring(0,1).matches("[A-Za-z]");
5     }
6
7     public void Validar(String Matricula) throws ExMatricula {
8         if (Matricula.length()!=8)
9             throw new ExMatricula(ExMatricula.MAL_TAMANIO);
10        else
11            if (!UnaLetra(Matricula))
12                throw new ExMatricula(ExMatricula.MAL_LETRA);
13            else
14                {} // matricula bien formada
15        }
16
17 }
```

Con el método de validación de matrículas implementado, capaz de levantar la excepción *ExMatricula*, ya podemos escribir una aplicación que recoja las matrículas que introducen los usuarios. La recuperación de los errores de formato se escribirá en el bloque *catch* correspondiente:

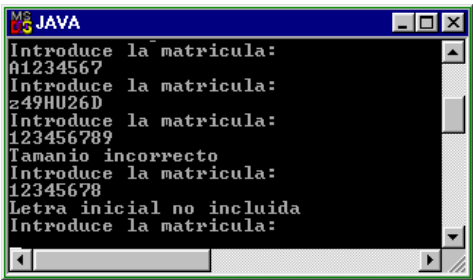
```
1 public class ExMatriculaPrincipal {
2
3     public static void main (String[] args) {
4         ExMatriculaValidar LaMatricula = new
5                                     ExMatriculaValidar();
6
7         do
8             try {
9                 System.out.println("Introduce la matricula: ");
10                String Matricula = Teclado.Lee_String();
11                LaMatricula.Validar(Matricula);
12            }
13            catch(ExMatricula e) {
14                switch (e.DimeProblema()) {
15                    case ExMatricula.MAL_TAMANIO:
16                        System.out.println("Tamano incorrecto");
17                        break;
18                    case ExMatricula.MAL_LETRA:
19                        System.out.println("Letra inicial no
20                                         incluida");
21                        break;
22                    default:
23                        System.out.println("Matricula correcta");
24                        break;
25                }
26            }
27        }
28    }
29 }
```

```
23     }
24     }
25     while(true);
26 }
27 }
```

En el código anterior existe un bucle infinito (líneas 6 y 25) que permite la validación de matrículas (líneas 8, 9 y 10). Habitualmente estas tres líneas se ejecutan repetitivamente; esporádicamente el usuario introducirá una matrícula con el formato erróneo y el método *Validar* levantará la excepción *ExMatricula*, que será recogida y tratada en el bloque *catch* (línea 12).

En la línea 13 sabemos que se ha producido una excepción *ExMatricula*, pero no conocemos su causa exacta (tamaño incorrecto o primer carácter distinto de letra). El método *DimeProblema* que incluimos en la clase *ExMatricula* nos indica el estado de la excepción. Si el estado de la excepción es *MAL\_TAMANIO* le indicamos esta situación al usuario que introdujo la matrícula con un tamaño incorrecto (líneas 14 y 15). En las líneas 17 y 18 hacemos lo propio con *MAL\_LETRA*.

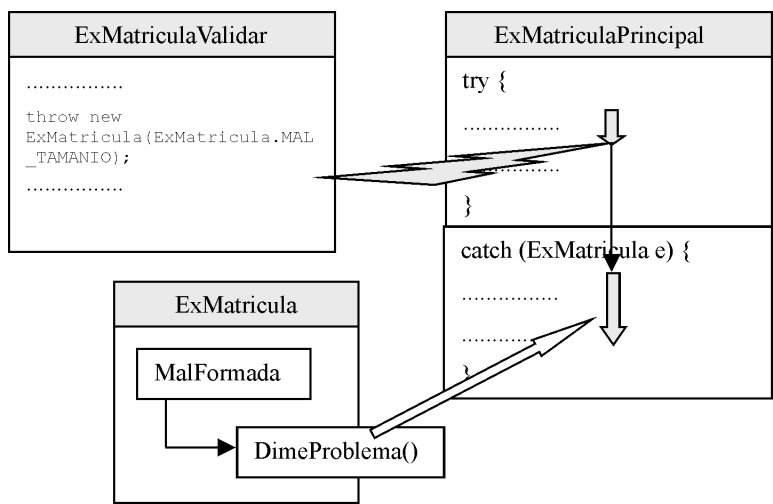
La ventana siguiente muestra un ejemplo de ejecución del programa:



Los objetos utilizados en el ejemplo se presentan en la siguiente tabla:

Objeto	Cometido	Nombre de la clase
Excepción	Define la situación de error	<i>ExMatricula</i>
Clase que levanta la excepción	Reconocimiento y comunicación de la situación de error, junto con la creación del objeto excepción	<i>ExMatriculaValidar</i>
Clase que recoge la excepción	Recibe la situación de error y trata de recuperar la situación	<i>ExMatriculaPrincipal</i>

La estructura de clases se representa en el siguiente diagrama que relaciona los objetos:





# INTERFAZ GRÁFICO DE USUARIO

---

## 7.1 CREACIÓN DE VENTANAS

### 7.1.1 Introducción

Hasta ahora, todos los ejemplos que se han desarrollado se han ejecutado en modo consola, es decir, las entradas de datos y la visualización de resultados se realiza a través de una ventana de MS-DOS. Las aplicaciones normalmente requieren de un interfaz de usuario más sofisticado, basado en ventanas gráficas y distintos componentes (botones, listas, cajas de texto, etc.) en el interior de estas ventanas.

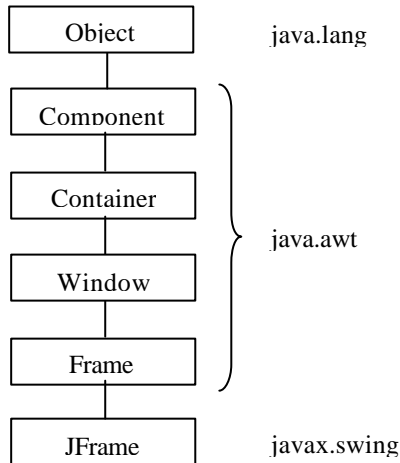
Java proporciona una serie de paquetes estándar que nos facilitan la labor de crear los Interfaces Gráficos de Usuario (GUI) que necesitamos. Estos paquetes se encuentran englobados dentro de Java Foundation Classes (JFC), que proporciona:

- Soporte de componentes de entrada/salida a través de AWT (Abstract Window Toolkit) y Swing
- Imágenes 2D
- Servicio de impresión
- Soporte para funciones Drag-and-Drop (arrastrar y pegar)
- Funciones de accesibilidad, etc.

De los paquetes anteriores, los más utilizados, con diferencia, son AWT y Swing. AWT fue el primero en aparecer y proporciona los componentes básicos de entrada/salida; Swing recoge la mayor parte de la funcionalidad de AWT, aportando clases más complejas y especializadas que las ya existentes, así como otras de nueva creación.

El elemento más básico que se suele emplear en un interfaz gráfico de usuario es la ventana. AWT proporciona la clase *Window*, aunque para simplificar los desarrollos acudiremos a la clase *Frame* (marco) derivada de *Window* y por lo tanto más especializada.

La jerarquía existente en torno a las ventanas de Java es:



La clase *Object* es la raíz de toda la jerarquía de clases, por lo que todos los objetos tienen a *Object* como superclase. La clase abstracta *Component* proporciona una representación gráfica susceptible de ser representada en un dispositivo de salida; como se puede observar es una clase muy genérica que no utilizaremos directamente, sino a través de sus clases derivadas.

La clase abstracta *Container* deriva de *Component* y añade la funcionalidad de poder contener otros componentes AWT (por ejemplo varios botones, una caja de texto, etc.). La clase *Window* proporciona una ventana gráfica sin bordes ni posibilidad de incluir barras de menú, por ello haremos uso de su clase derivada *Frame*, que no presenta las restricciones del objeto *Window*.

*JFrame* es la versión de *Frame* proporcionada por *Swing*. Al igual que muchos otros componentes de *Swing*, ofrece posibilidades más amplias que las implementadas por sus superclases de AWT, aunque también presenta una mayor complejidad de uso.

Nosotros haremos uso de las clases más útiles que nos ofrece AWT, empezando por la definición de ventanas mediante el objeto *Frame*.

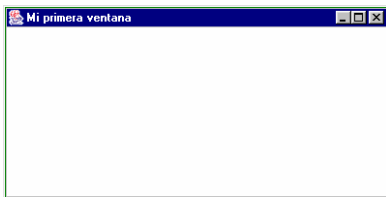
### 7.1.2 Utilización de la clase *Frame*

Una manera muy sencilla de crear y visualizar una ventana en Java se muestra en el siguiente ejemplo (*Marco1*): en primer lugar se importa la clase *Frame* de AWT (línea 1); más adelante, en otros ejemplos, utilizaremos conjuntamente diferentes clases de AWT, por lo que nos resultará más sencillo utilizar la instrucción `import java.awt.*`.

La línea 6 nos crea la instancia *MiMarco* de tipo *Frame*; en la línea 7 se asigna un tamaño de ventana de 400 (horizontal) por 200 (vertical) pixels; en la línea 8 se define el título de la ventana y, por último, la línea 9 hace visible (representa) el marco creado y definido en las líneas anteriores.

La clase *Frame* contiene 4 constructores; uno de ellos permite definir el título, por lo que las líneas 6 y 8 pueden sustituirse por la sentencia `Frame MiMarco = new Frame("Mi primera ventana");`.

```
1  import java.awt.Frame;
2
3  public class Marco1 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          MiMarco.setSize(400,200);
8          MiMarco.setTitle("Mi primera ventana");
9          MiMarco.setVisible(true);
10     }
11 }
```



No existe ningún constructor de la clase *Frame* que admita como parámetros el tamaño de la ventana, lo que nos ofrece la posibilidad de crearnos nuestra propia clase (*Marco2*) que herede de *Frame* y ofrezca esta posibilidad:

```
1  import java.awt.Frame;
2
3  public class Marco2 extends Frame {
4
5      Marco2(String Titulo) {
```

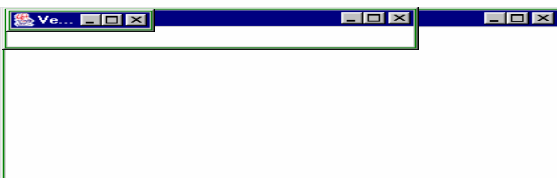
```
6      this.setTitle(Titulo);
7      this.setVisible(true);
8  }
9
10     Marco2(String Titulo, int Ancho, int Alto) {
11         this(Titulo);
12         this.setSize(Ancho, Alto);
13     }
14
15     Marco2() {
16         this("Mi primera ventana",400,200);
17     }
18
19 }
```

La clase *Marco2* que hemos diseñado permite utilizar el constructor vacío (línea 15), obteniendo una ventana de tamaño 400 x 200 con el título “Mi primera ventana”. También se puede incorporar el título (línea 5), consiguiéndose una ventana visible (línea 7) con el título indicado. Finalmente se proporciona un constructor que permite incluir título y dimensiones (línea 10); en este caso nos apoyamos en el constructor que admite título (palabra reservada *this* en la línea 11) y el método *setSize* del objeto *Frame* (línea 12).

Una vez definida la clase *Marco2*, podemos utilizarla en diferentes programas como el mostrado a continuación:

```
1  public class Marco2main {
2
3      public static void main(String[] args) {
4          Marco2 MiVentana1 = new Marco2();
5          Marco2 MiVentana2 = new Marco2("Ventana2",300,50);
6          Marco2 MiVentana3 = new Marco2("Ventana3");
7      }
8  }
```

*Marco2main* crea tres instancias de la clase *Marco2*, que al ser subclase de *Frame* representa marcos de ventanas; las líneas 4 a 6 muestran diferentes maneras de instanciar *Marco2*. El resultado obtenido son tres ventanas superpuestas de diferentes tamaños y títulos:



### 7.1.3 Creación y posicionamiento de múltiples ventanas

Los interfaces gráficos de usuario (GUI) no tienen por qué restringirse al uso de una única ventana para entrada de datos, mostrar información, etc. Siguiendo el mecanismo explicado en los apartados anteriores podemos crear y hacer uso de un número ilimitado de ventanas.

Cuando se crean varias ventanas a menudo resulta útil situar cada una de ellas en diferentes posiciones del dispositivo de salida, de forma que adopten una configuración práctica y estética, evitándose que se oculten entre sí, etc.

Para situar una ventana en una posición concreta utilizamos el método *setLocation*, heredado de la clase *Component*. Este método está sobrecargado y admite dos tipos de parámetros de entrada:

- *setLocation* (int x, int y)
- *setLocation* (Point p)

En ambos casos es necesario suministrar las coordenadas bidimensionales que sitúen la esquina superior izquierda de la ventana respecto a su componente padre (la pantalla, el objeto contenedor del *Frame*, etc.). El siguiente ejemplo (*Marco3*) ilustra la manera de utilizar el método que usa la clase *Point* de AWT como argumento:

```
1  import java.awt.Frame;
2  import java.awt.Point;
3
4  public class Marco3 {
5
6      public static void main(String[] args) {
7          Frame MiMarco = new Frame();
8          MiMarco.setSize(400,200);
9          MiMarco.setTitle("Mi primera ventana");
10         MiMarco.setLocation(new Point(100,220));
11         MiMarco.setVisible(true);
12     }
13 }
```

El resultado de la clase *Marco3* es una ventana situada en la posición (100,220) respecto al origen del *GraphicsDevice*, que habitualmente es la pantalla gráfica del ordenador.

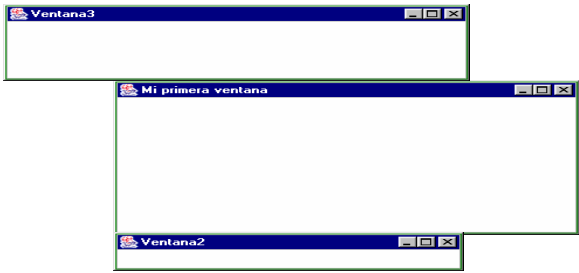
A continuación se presenta la clase *Marco4*, que surge de *Marco2*, añadiendo la posibilidad de indicar la posición de la ventana instanciada:

```
1  import java.awt.Frame;
2  import java.awt.Point;
3
4  public class Marco4 extends Frame {
5
6      Marco4(String Titulo) {
7          this.setTitle(Titulo);
8          this.setVisible(true);
9      }
10
11     Marco4(String Titulo, int Ancho, int Alto) {
12         this(Titulo);
13         this.setSize(Ancho, Alto);
14     }
15
16     Marco4(String Titulo, int Ancho, int Alto,
17             int PosX, int PosY) {
18         this(Titulo,Ancho, Alto);
19         this.setLocation(new Point(PosX,PosY));
20     }
21
22     Marco4() {
23         this("Mi primera ventana",400,200,100,100);
24     }
25
26 }
```

El constructor definido en la línea 16 permite situar el marco creado en la posición deseada (línea 19). La clase *Marco4main* instancia varias ventanas con diferentes tamaños y posiciones iniciales:

```
1  public class Marco4main {
2
3      public static void main(String[] args) {
4          Marco4 MiVentana1 = new Marco4();
5          Marco4 MiVentana2 = new
6                                  Marco4("Ventana2",300,50,100,300);
7          Marco4 MiVentana3 = new Marco4("Ventana3",400,100);
8      }
```

La posición relativa de las 3 ventanas involucradas en el ejemplo es la siguiente:



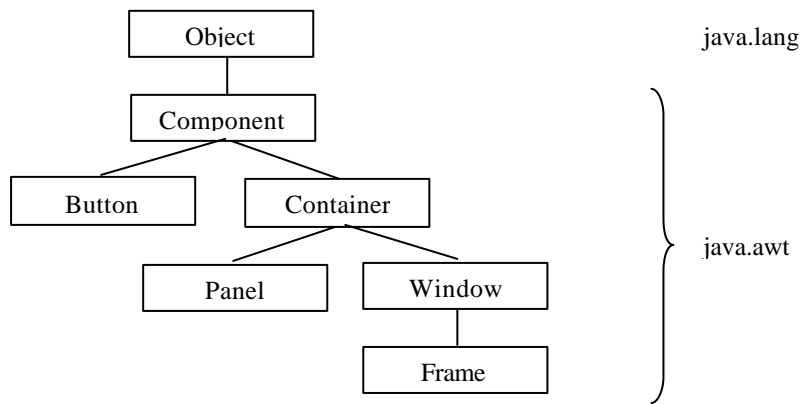
## 7.2 PANELES Y OBJETOS DE DISPOSICIÓN (LAYOUTS)

### 7.2.1 Introducción

Las ventanas tienen utilidad si podemos añadirles diversos componentes de interacción con los usuarios. El elemento que en principio se adapta mejor a esta necesidad es la clase *Container*; como ya se explicó: “La clase abstracta *Container* deriva de *Component* y añade la funcionalidad de poder contener otros componentes AWT (por ejemplo varios botones, una caja de texto, etc.)”.

En lugar de utilizar directamente la clase *Container*, resulta mucho más sencillo hacer uso de su clase heredada *Panel*. Un panel proporciona espacio donde una aplicación puede incluir cualquier componente, incluyendo otros paneles.

Para ilustrar el funcionamiento y necesidad de los paneles emplearemos un componente muy útil, conocido y fácil de utilizar: los botones (*Button*). Una forma de crear un botón es a través de su constructor *Button(String Texto)*, que instancia un botón con el “Texto” interior que se le pasa como parámetro.



Nuestro primer ejemplo (*Boton1*) no hace uso de ningún panel, situando un botón directamente sobre una ventana. En la línea 8 se crea la instancia *BotonHola* del objeto *Button*, con el texto “Hola” como etiqueta. La línea 10 es muy importante, añade el componente *BotonHola* al contenedor *MiMarco*; además de crear los componentes (en nuestro caso el botón) es necesario situar los mismos en los contenedores donde se vayan a visualizar.

A continuación del código se representa el resultado del programa: un botón que ocupa toda la ventana. Si hubiéramos creado varios botones y los hubiésemos añadido a *MiMarco*, el resultado visible sería la ventana ocupada por entero por el último botón añadido.

```
1  import java.awt.Frame;
2  import java.awt.Button;
3
4  public class Boton1 {
5
6      public static void main(String[] args) {
7          Frame MiMarco = new Frame();
8          Button BotonHola = new Button("Hola");
9
10         MiMarco.add(BotonHola);
11
12         MiMarco.setSize(400,200);
13         MiMarco.setTitle("Ventana con botón");
14         MiMarco.setVisible(true);
15     }
16 }
```

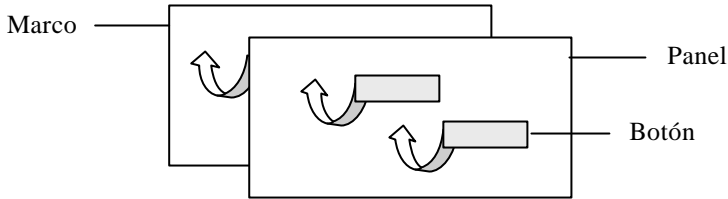


Si no utilizamos paneles (y/o *layouts*), los componentes (tales como botones) se representarán en las ventanas de una manera inadecuada. Los paneles y *layouts* nos permiten añadir componentes visuales con diferentes disposiciones en el plano.

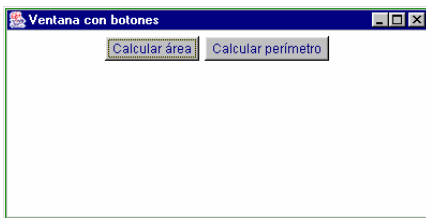
El ejemplo anterior puede ser ampliado con facilidad para que el marco incorpore un panel. Usando el panel conseguiremos que los componentes que añadamos al mismo sigan una disposición establecida (por defecto ordenación secuencial).



La clase *Boton2* instancia un panel en la línea 9 y lo añade al marco en la línea 13. Los botones creados en las líneas 10 y 11 se añaden (secuencialmente, por defecto) al panel en las líneas 14 y 15:

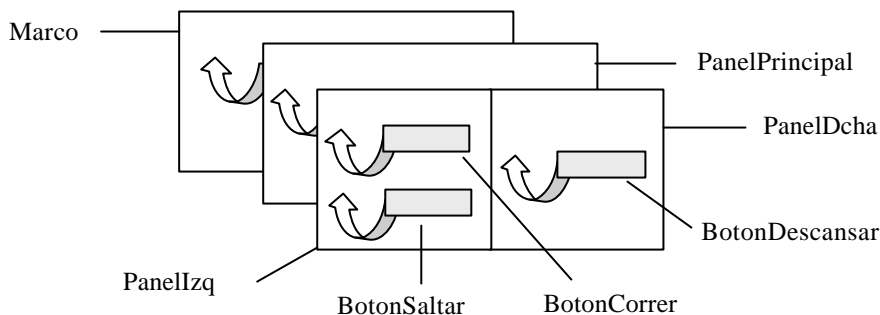


```
1  import java.awt.Frame;
2  import java.awt.Button;
3  import java.awt.Panel;
4
5  public class Boton2 {
6
7      public static void main(String[] args) {
8          Frame MiMarco = new Frame();
9          Panel MiPanel = new Panel();
10         Button BotonArea = new Button("Calcular área");
11         Button BotonPerimetro = new Button("Calcular
                                         perímetro");
12
13         MiMarco.add(MiPanel);
14         MiPanel.add(BotonArea);
15         MiPanel.add(BotonPerimetro);
16
17         MiMarco.setSize(400,200);
18         MiMarco.setTitle("Ventana con botones");
19         MiMarco.setVisible(true);
20     }
21 }
```



## 7.2.2 Utilización básica de paneles

Los paneles son contenedores que pueden albergar componentes (incluidos otros paneles). En el siguiente ejemplo (*Panel1*) emplearemos la disposición de elementos que se muestra a continuación:



```

1  import java.awt.Frame;
2  import java.awt.Button;
3  import java.awt.Panel;
4
5  public class Panel1 {
6
7      public static void main(String[] args) {
8          Frame MiMarco = new Frame();
9          Panel PanelPrincipal = new Panel();
10         Panel PanelIzq = new Panel();
11         Panel PanelDcha = new Panel();
12
13         Button BotonCorrer = new Button("Correr");
14         Button BotonSaltar = new Button("Saltar");
15         Button BotonDescansar = new Button("Descansar");
16
17         MiMarco.add(PanelPrincipal);
18         PanelPrincipal.add(PanelIzq);
19         PanelPrincipal.add(PanelDcha);
20         PanelIzq.add(BotonCorrer);
21         PanelIzq.add(BotonSaltar);
22         PanelDcha.add(BotonDescansar);
23
24         MiMarco.setSize(400,200);
25         MiMarco.setTitle("Ventana con paneles");
26         MiMarco.setVisible(true);
27     }
28 }

```

La ventana de la izquierda representa un posible resultado de este programa. Dependiendo de la definición de pantalla en pixels que tenga el usuario la relación de tamaños entre los botones y la ventana puede variar. Si los botones no caben a lo ancho, la disposición secuencial hará “saltar de línea” al panel derecho, obteniéndose una visualización como la mostrada en la ventana de la derecha.



La idea subyacente en el diseño del paquete AWT es que los interfaces gráficos de usuario puedan visualizarse independientemente de las características y configuración de los dispositivos de salida, así como del tamaño de las ventanas de visualización (en muchos casos los navegadores Web). Para compaginar este objetivo con la necesidad de dotar al programador de la flexibilidad necesaria en el diseño de GUI's se proporcionan las clases “Layouts” (*FlowLayout*, *BorderLayout*, *GridLayout*, etc.).

Cada panel que creamos tiene asociado, implícita o explícitamente, un tipo de disposición (“layout”) con la que se visualizan los componentes que se le añaden. Si no especificamos nada, la disposición por defecto (implícita) es *FlowLayout*, con los elementos centrados en el panel, tal y como aparecen los botones del ejemplo anterior.

### 7.2.3 Objeto de disposición (Layout): *FlowLayout*

Para establecer una disposición de manera explícita, podemos recurrir al método *setLayout* (heredado de la clase *Container*) o bien instanciar nuestro panel indicando directamente el “layout” deseado. Las siguientes instrucciones ilustran las posibles maneras de asociar un *FlowLayout* a un panel:

```
Panel MiPanel = new Panel (new FlowLayout());
```

```
OtroPanel.setLayout(new FlowLayout());
```

El siguiente ejemplo (*FlowLayout1*) muestra una posible forma de crear una ventana con 4 botones dispuestos horizontalmente. En la línea 11 creamos el objeto

*PosicionamientoSecuencial*, de tipo *FlowLayout*. En la línea 17 asociamos este objeto al panel instanciado en la línea 10 (*MiPanel*).

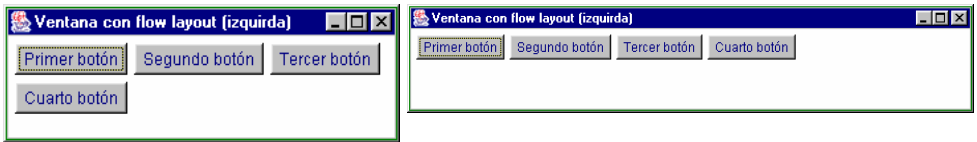
```
1  import java.awt.Frame;
2  import java.awt.Button;
3  import java.awt.Panel;
4  import java.awt.FlowLayout;
5
6  public class FlowLayout1 {
7
8      public static void main(String[] args) {
9          Frame MiMarco = new Frame();
10         Panel MiPanel = new Panel();
11         FlowLayout PosicionamientoSecuencial = new
                                                    FlowLayout();
12         Button BotonA = new Button("Primer botón");
13         Button BotonB = new Button("Segundo botón");
14         Button BotonC = new Button("Tercer botón");
15         Button BotonD = new Button("Cuarto botón");
16
17         MiPanel.setLayout(PosicionamientoSecuencial);
18
19         MiMarco.add(MiPanel);
20         MiPanel.add(BotonA);
21         MiPanel.add(BotonB);
22         MiPanel.add(BotonC);
23         MiPanel.add(BotonD);
24
25         MiMarco.setSize(300,100);
26         MiMarco.setTitle("Ventana con flow layout");
27         MiMarco.setVisible(true);
28     }
29 }
```

La ventana que se visualiza contiene cuatro botones posicionados consecutivamente y centrados respecto al panel. Dependiendo del tamaño con el que dimensionemos la ventana, los botones cabrán horizontalmente o bien tendrán que “saltar” a sucesivas líneas para tener cabida, tal y como ocurre con las líneas en un procesador de texto:



Los objetos *FlowLayout* pueden instanciarse con posicionamiento centrado (*FlowLayout.CENTER*), izquierdo (*FlowLayout.LEFT*), derecho (*FlowLayout.RIGHT*) y los menos utilizados *LEADING* y *TRAILING*. La disposición por defecto es *CENTER*.

Si en el ejemplo anterior sustituimos la línea 11 por: *FlowLayout PosicionamientoSecuencial = new FlowLayout(FlowLayout.LEFT);* el resultado obtenido es:



## 7.2.4 Objeto de disposición (Layout): BorderLayout

Un *BorderLayout* sitúa y dimensiona los componentes en 5 “regiones cardinales”: norte, sur, este, oeste y centro. La clase que se presenta a continuación (*BorderLayout1*) crea un *BorderLayout* (línea 11) y lo asocia al panel *MiPanel* (línea 18). En las líneas 21 a 25 se añaden cinco botones al panel, cada uno en una zona “cardinal”.

```

1  import java.awt.Frame;
2  import java.awt.Button;
3  import java.awt.Panel;
4  import java.awt.BorderLayout;
5
6  public class BorderLayout1 {
7
8      public static void main(String[] args) {
9          Frame MiMarco = new Frame();
10         Panel MiPanel = new Panel();
11         BorderLayout PuntosCardinales = new BorderLayout();
12         Button BotonNorte = new Button("Norte");
13         Button BotonSur = new Button("Sur");
14         Button BotonEste = new Button("Este");
15         Button BotonOeste = new Button("Oeste");
16         Button BotonCentro = new Button("Centro");
17
18         MiPanel.setLayout(PuntosCardinales);
19

```

```

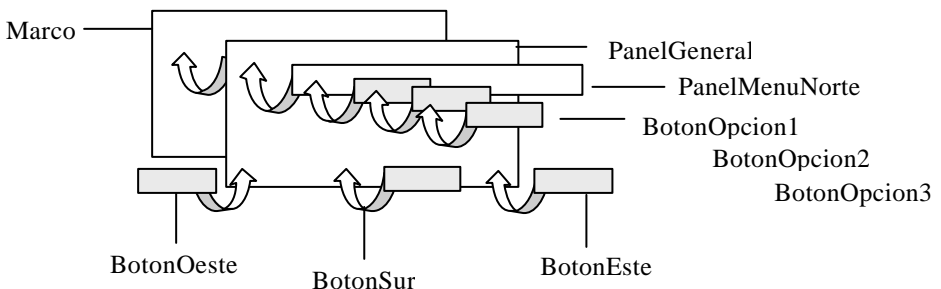
20     MiMarco.add(MiPanel);
21     MiPanel.add(BotonNorte, BorderLayout.NORTH);
22     MiPanel.add(BotonSur, BorderLayout.SOUTH);
23     MiPanel.add(BotonEste, BorderLayout.EAST);
24     MiPanel.add(BotonOeste, BorderLayout.WEST);
25     MiPanel.add(BotonCentro, BorderLayout.CENTER);
26
27     MiMarco.setSize(300,100);
28     MiMarco.setTitle("Ventana con BorderLayout");
29     MiMarco.setVisible(true);
30 }
31 }

```

Tal y como muestran los dos siguientes resultados (correspondientes a la ventana con el tamaño original y la misma ventana agrandada) los componentes (botones) insertados en el panel se sitúan en sus respectivas regiones del *BorderLayout*.



Cada región de un *BorderLayout* admite un único componente, pero nada impide que ese componente sea un contenedor que admita una nueva serie de componentes. Vamos a modificar la clase *BorderLayout1* para que la región norte sea ocupada por tres botones dispuestos en posición horizontal. El diseño de la solución es el siguiente:



En la clase *BorderLayout2* se crea el panel *PanelGeneral* (línea 8) al que se asocia el *BorderLayout PuntosCardinales* (línea 9) mediante el método *setLayout* (línea 19). Análogamente, se crea el panel *PanelMenuNorte* (línea 7) al que se asocia el *FlowLayout OpcionesMenu* (línea 10) mediante el método *setLayout* (línea 20).

A *PanelMenuNorte* se le añaden tres botones (líneas 24 a 26), mientras que el propio panel (con los tres botones incluidos) se añade a la región norte del *PanelGeneral* (línea 23). De esta manera, el botón *BotonNorte* del ejemplo anterior queda sustituido por la secuencia (en flujo secuencial) de botones *BotonOpcion1*, *BotonOpcion2* y *BotonOpcion3*.

```
1  import java.awt.*;
2
3  public class BorderLayout2 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel PanelMenuNorte = new Panel();
8          Panel PanelGeneral = new Panel();
9          BorderLayout PuntosCardinales = new BorderLayout();
10         FlowLayout OpcionesMenu = new FlowLayout();
11         Button BotonOpcion1 = new Button("Opción 1");
12         Button BotonOpcion2 = new Button("Opción 2");
13         Button BotonOpcion3 = new Button("Opción 3");
14         Button BotonSur = new Button("Sur");
15         Button BotonEste = new Button("Este");
16         Button BotonOeste = new Button("Oeste");
17         Button BotonCentro = new Button("Centro");
18
19         PanelGeneral.setLayout(PuntosCardinales);
20         PanelMenuNorte.setLayout(OpcionesMenu);
21
22         MiMarco.add(PanelGeneral);
23         PanelGeneral.add(PanelMenuNorte, BorderLayout.NORTH);
24         PanelMenuNorte.add(BotonOpcion1);
25         PanelMenuNorte.add(BotonOpcion2);
26         PanelMenuNorte.add(BotonOpcion3);
27         PanelGeneral.add(BotonSur, BorderLayout.SOUTH);
28         PanelGeneral.add(BotonEste, BorderLayout.EAST);
29         PanelGeneral.add(BotonOeste, BorderLayout.WEST);
30         PanelGeneral.add(BotonCentro, BorderLayout.CENTER);
31
32         MiMarco.setSize(400,150);
33         MiMarco.setTitle("Ventana con BorderLayout");
34         MiMarco.setVisible(true);
35     }
36 }
```



### 7.2.5 Objeto de disposición (Layout): *GridLayout*

Un *GridLayout* coloca y dimensiona los componentes en una rejilla rectangular. El contenedor se divide en rectángulos de igual tamaño, colocándose un componente en cada rectángulo.

Los objetos *GridLayout* se pueden instanciar indicando el número de filas y columnas de la rejilla:

```
GridLayout Matriz = new GridLayout(3,2);
```

También se puede hacer uso del constructor vacío y posteriormente utilizar los métodos *setRows* y *setColumns*:

```
GridLayout Matriz = new GridLayout();
```

```
Matriz.setRows(3);
```

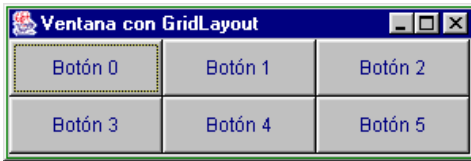
```
Matriz.setColumns(2);
```

La clase *GridLayout1* crea una ventana con 6 botones dispuestos matricialmente en 2 filas y 3 columnas, para ello se instancia un objeto de tipo *GridLayout* con dicha disposición (línea 9). En la línea 14 se asocia el *GridLayout* al panel, añadiéndose posteriormente 6 botones al mismo (líneas 15 y 16).

```
1 import java.awt.*;
2
3 public class GridLayout1 {
4
5     public static void main(String[] args) {
6         Frame MiMarco = new Frame();
7         Panel MiPanel = new Panel();
8
9         GridLayout Matriz = new GridLayout(2,3);
```

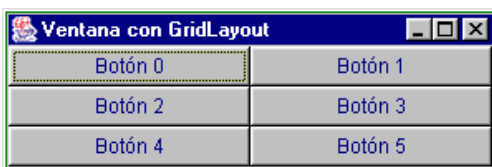


```
10     Button[] Botones = new Button[6];
11     for (int i=0;i<6;i++)
12         Botones[i] = new Button("Botón "+i);
13
14     MiPanel.setLayout(Matriz);
15     for (int i=0;i<6;i++)
16         MiPanel.add(Botones[i]);
17
18     MiMarco.add(MiPanel);
19     MiMarco.setSize(300,100);
20     MiMarco.setTitle("Ventana con GridLayout");
21     MiMarco.setVisible(true);
22 }
23 }
```



En *GridLayout2* se crea una clase muy sencilla en la que se asocia un *GridLayout* de 3 x 2 directamente al marco de la aplicación:

```
1  import java.awt.*;
2
3  public class GridLayout2 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7
8          MiMarco.setLayout(new GridLayout(3,2));
9          for (int i=0;i<6;i++)
10             MiMarco.add(new Button("Botón "+i));
11
12         MiMarco.setSize(300,100);
13         MiMarco.setTitle("Ventana con GridLayout");
14         MiMarco.setVisible(true);
15     }
16 }
```



Finalmente, en *GridLayout3* se utiliza un *GridLayout* (línea 8) en el que, en una celda, se añade un componente *Panel* (*Vertical*) con disposición *BorderLayout* (línea 7). Al panel *Vertical* se le añaden tres botones en las regiones cardinales: norte, centro y sur (líneas 10 a 12).

```
1  import java.awt.*;
2
3  public class GridLayout3 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel Vertical = new Panel(new BorderLayout());
8          MiMarco.setLayout(new GridLayout(2,3));
9
10         Vertical.add(new Button("Arriba"), BorderLayout.NORTH);
11         Vertical.add(new Button("Centro"),
12                        BorderLayout.CENTER);
13
14         Vertical.add(new Button("Abajo"), BorderLayout.SOUTH);
15
16         MiMarco.add(Vertical);
17         for (int i=1;i<6;i++)
18             MiMarco.add(new Button("Botón "+i));
19
20         MiMarco.setSize(300,160);
21         MiMarco.setTitle("Ventana con Layouts Grid y Border");
22         MiMarco.setVisible(true);
23     }
24 }
```

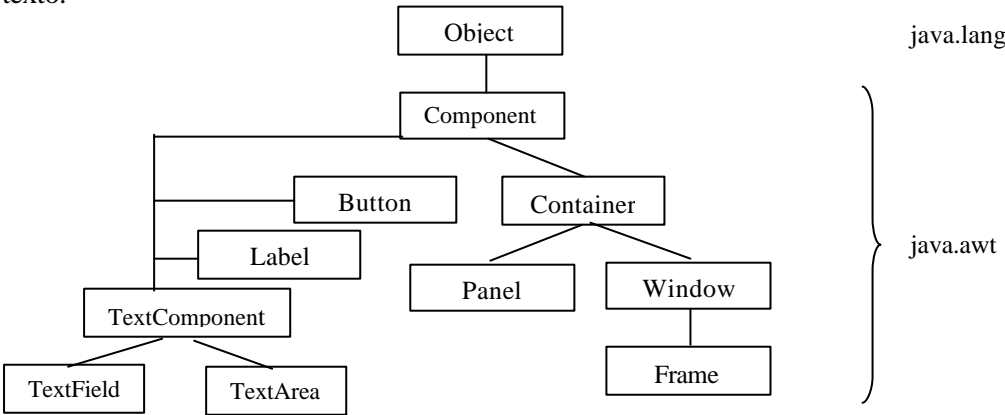


## 7.3 ETIQUETAS, CAMPOS Y ÁREAS DE TEXTO

### 7.3.1 Introducción

Una vez que somos capaces de crear marcos, añadirles paneles con diferentes “layouts” e insertar componentes en los mismos, estamos preparados para realizar diseños de interfaces de usuario, sin embargo, todavía nos falta conocer los diferentes componentes básicos de entrada/salida que podemos utilizar en Java.

Un componente que hemos empleado en todos los ejemplos anteriores es el botón (*Button*). Ahora se explicarán las etiquetas, los campos de texto y las áreas de texto.



### 7.3.2 Etiqueta (Label)

Las etiquetas permiten situar un texto en un contenedor. El usuario no puede editar el texto, aunque si se puede variar por programa.

Los constructores de la clase *Label* son:

- Label ()*
- Label (String Texto)*
- Label (String Texto, int Alineacion)*

Entre los métodos existentes tenemos:

*setText (String Texto)*

*setAlignment (int Alineacion)*

De esta manera podemos definir e instanciar etiquetas de diversas maneras:

*Label Saludo = new Label ("Hola", Label.LEFT);*

*Label OtroSaludo = new Label ("Buenos días");*

*OtroSaludo.setAlignment (Label.CENTER);*

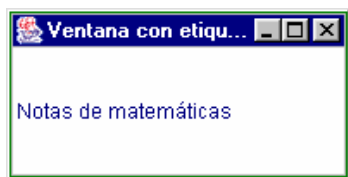
*Label Cabecera = new Label ();*

*Cabecera.setAlignment (Label.RIGHT);*

*Cabecera.setText ("Ingresos totales");*

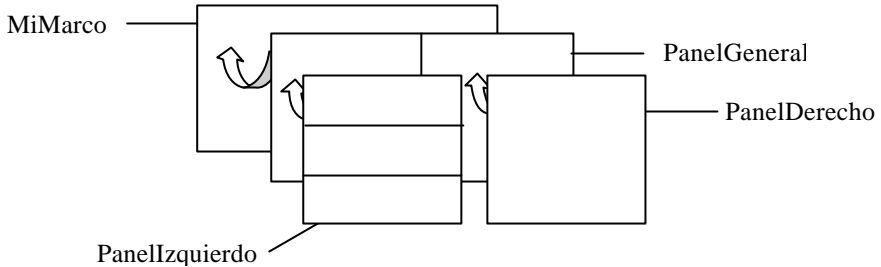
En la clase *Etiqueta1* se presenta la manera más simple de visualizar una etiqueta:

```
1  import java.awt.Frame;  
2  import java.awt.Label;  
3  
4  public class Etiqueta1 {  
5  
6      public static void main(String[] args) {  
7          Frame MiMarco = new Frame();  
8          Label Titulo = new Label("Notas de matemáticas");  
9  
10         MiMarco.add(Titulo);  
11  
12         MiMarco.setSize(200,100);  
13         MiMarco.setTitle("Ventana con etiqueta");  
14         MiMarco.setVisible(true);  
15     }  
16 }
```



El siguiente ejemplo que se presenta (*Etiqueta2*) combina el uso de diferentes “layouts” con el de etiquetas. Se instancia un *PanelGeneral* con un *GridLayout* 1 x 2 asociado (línea 7). Al *PanelGeneral* se le añaden los paneles *PanelIzquierdo* y *PanelDerecho* (líneas 11 y 12), el primero con disposición

*GridLayout* 3 x 1 (línea 8) y el segundo con disposición *FlowLayout* (línea 9). En las líneas 13 a 17 se definen y añaden etiquetas utilizando su constructor más completo.



```

1  import java.awt.*;
2
3  public class Etiqueta2 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel PanelGeneral = new Panel(new GridLayout(1,2));
8          Panel PanelIzquierdo = new Panel(new GridLayout(3,1));
9          Panel PanelDerecho = new Panel(new FlowLayout());
10
11         PanelGeneral.add(PanelIzquierdo);
12         PanelGeneral.add(PanelDerecho);
13         PanelIzquierdo.add(new Label("Ford",Label.CENTER));
14         PanelIzquierdo.add(new Label("Opel",Label.CENTER));
15         PanelIzquierdo.add(new Label("Audi",Label.CENTER));
16         PanelDerecho.add(new Label("Coupe"));
17         PanelDerecho.add(new Label("Cabrio"));
18
19         MiMarco.add(PanelGeneral);
20         MiMarco.setSize(250,100);
21         MiMarco.setTitle("Ventana con etiqueta");
22         MiMarco.setVisible(true);
23     }
24 }

```



En el último ejemplo de etiquetas (*Etiqueta3*) se imprime la tabla de multiplicar con cabeceras, para ello se crea un *GridLayout* de 11 x 11 (línea 7) que albergará los resultados de la tabla de multiplicar (10 x 10 resultados) más la cabecera de filas y la cabecera de columnas.

Los valores de las cabeceras de filas y columnas se crean como sendas matrices unidimensionales de etiquetas *CabeceraFila* y *CabeceraColumna* (líneas 8 y 9).

En las líneas 11 a 14 se asignan valores a los elementos del array *CabeceraFila*, y se les asocia el color rojo mediante el método *setBackground* perteneciente a la clase *Component*. En las líneas 18 a 20 se hace lo propio con el array *CabeceraColumna*.

Las etiquetas de las celdas centrales de la tabla (los resultados) se calculan y añaden al panel *Tabla* en las líneas 22 y 23, dentro del bucle que recorre cada fila (línea 18).

```
1  import java.awt.*;
2
3  public class Etiqueta3 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel Tabla = new Panel(new GridLayout(11,11));
8          Label[] CabeceraFila = new Label[11];
9          Label[] CabeceraColumna = new Label[11];
10
11         Tabla.add(new Label(""));
12         for (int i=1;i<=10;i++) {
13             CabeceraFila[i] = new Label(""+i);
14             CabeceraFila[i].setBackground(Color.red);
15             Tabla.add(CabeceraFila[i]);
16         }
17
18         for (int i=1;i<=10;i++) {
19             CabeceraColumna[i] = new Label(""+i);
20             CabeceraColumna[i].setBackground(Color.red);
21             Tabla.add(CabeceraColumna[i]);
22             for (int j=1;j<=10;j++)
23                 Tabla.add(new Label(""+i*j));
24         }
25
26         MiMarco.add(Tabla);
27         MiMarco.setSize(400,400);
28         MiMarco.setTitle("Tabla de multiplicar");
```

```

29     MiMarco.setVisible(true);
30 }
31 }

```



	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

### 7.3.3 Campo de texto (TextField)

Un campo de texto es un componente que permite la edición de una línea de texto. Sus constructores permiten que su tamaño tenga un número de columnas determinado y que el campo presente un texto inicial.

En la clase *CampoDeTexto1* se instancian tres campos de texto: *Nombre*, *Apellidos* y *Nacionalidad* (líneas 8 a 10); *Nombre* se define con un tamaño de edición de 15 columnas y sin texto inicial, *Apellidos* con 60 columnas y sin texto inicial, finalmente *Nacionalidad* con 15 columnas y texto inicial “Española”.

```

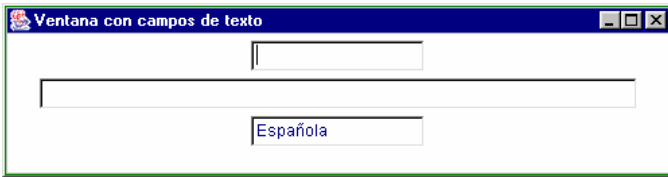
1  import java.awt.*;
2
3  public class CampoDeTexto1 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel EntradaDeDatos = new Panel(new FlowLayout());
8          TextField Nombre = new TextField(15);
9          TextField Apellidos = new TextField(60);
10         TextField Nacionalidad = new TextField("Española",15);
11
12         EntradaDeDatos.add(Nombre);
13         EntradaDeDatos.add(Apellidos);
14         EntradaDeDatos.add(Nacionalidad);

```

```

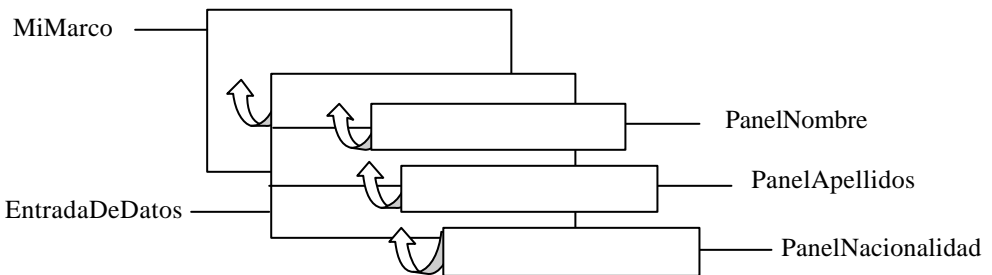
15
16     MiMarco.add(EntradaDeDatos);
17     MiMarco.setSize(500,130);
18     MiMarco.setTitle("Ventana con campos de texto");
19     MiMarco.setVisible(true);
20 }
21 }

```



En el resultado de la clase *CampoDeTexto1* se puede observar que los campos de texto no incluyen un literal que indique lo que hay que introducir, por lo que habitualmente se utilizan etiquetas antes de los campos de texto para realizar esta función. Por otra parte, la alineación *CENTER* que por defecto presenta la disposición *FlowLayout* no siempre es adecuada. El siguiente ejemplo (*CampoDeTexto2*) resuelve ambas situaciones.

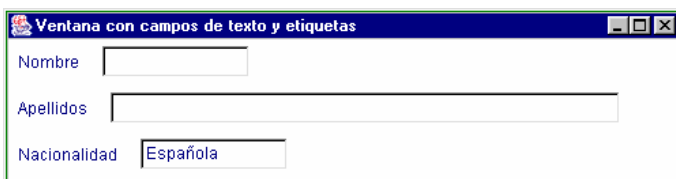
En la clase *CampoDeTexto2* se crea el panel *EntradaDeDatos* como *GridLayout* de 3 x 1, es decir, con 3 celdas verticales (línea 7). En cada una de estas celdas se añade un panel con disposición *FlowLayout* y sus componentes alineados a la izquierda (líneas 8, 9 y 10).



En *PanelNombre* se añaden (líneas 22 y 23) la *EtiquetaNombre* y el *CampoNombre*, definidos en las líneas 15 y 12; en *PanelApellidos* se hace lo mismo con *EtiquetaApellidos* y *CampoApellidos*; en *PanelNacionalidad* se realiza el mismo proceso. El resultado esperado se puede observar en la ventana situada tras el código.

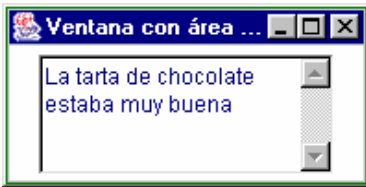


```
1 import java.awt.*;
2
3 public class CampoDeTexto2 {
4
5     public static void main(String[] args) {
6         Frame MiMarco = new Frame();
7         Panel EntradaDeDatos = new Panel(new GridLayout(3,1));
8         Panel PanelNombre = new Panel(new
9             FlowLayout(FlowLayout.LEFT));
10        Panel PanelApellidos = new Panel(new
11            FlowLayout(FlowLayout.LEFT));
12        Panel PanelNacionalidad = new Panel(new
13            FlowLayout(FlowLayout.LEFT));
14        TextField CampoNombre = new TextField(12);
15        TextField CampoApellidos = new TextField(50);
16        TextField CampoNacionalidad = new
17            TextField("Española",12);
18        Label EtiquetaNombre = new Label("Nombre",Label.LEFT);
19        Label EtiquetaApellidos = new
20            Label("Apellidos",Label.LEFT);
21        Label EtiquetaNacionalidad = new
22            Label("Nacionalidad",Label.LEFT);
23
24        EntradaDeDatos.add(PanelNombre);
25        EntradaDeDatos.add(PanelApellidos);
26        EntradaDeDatos.add(PanelNacionalidad);
27        PanelNombre.add(EtiquetaNombre);
28        PanelNombre.add(CampoNombre);
29        PanelApellidos.add(EtiquetaApellidos);
30        PanelApellidos.add(CampoApellidos);
31        PanelNacionalidad.add(EtiquetaNacionalidad);
32        PanelNacionalidad.add(CampoNacionalidad);
33
34        MiMarco.add(EntradaDeDatos);
35        MiMarco.setSize(500,130);
36        MiMarco.setTitle("Ventana con campos de texto
37            y etiquetas");
38        MiMarco.setVisible(true);
39    }
40 }
```





```
12     Comentarios.append(" de chocolate estaba buena");
13     Comentarios.insert(" muy", 28);
14
15     MiPanel.add(Comentarios);
16     MiMarco.add(MiPanel);
17     MiMarco.setSize(200,100);
18     MiMarco.setTitle("Ventana con área de texto");
19     MiMarco.setVisible(true);
20 }
21 }
```



### 7.3.5 Fuentes (Font)

Las fuentes no son componentes; son clases que heredan de forma directa de *Object* (la clase inicial de la jerarquía). El conocimiento de la clase *Font* es importante debido a que nos permite variar el aspecto de los textos involucrados en los componentes que estamos estudiando. Podemos cambiar el tamaño y el tipo de letra de los caracteres contenidos en una etiqueta, un campo de texto, un área de texto, etc.

Su constructor más utilizado es:

*Font (String Nombre, int Estilo, int Tamaño)*

El nombre, que puede ser lógico o físico, indica el tipo de letra con el que se visualizará el texto (*Serif*, *SansSerif*, *Monospaced*...). El estilo se refiere a letra itálica, negrilla, normal o alguna combinación de ellas: *Font.ITALIC*, *Font.BOLD*, *Font.PLAIN*.

La clase de ejemplo *Fuente* ilustra el modo con el que se puede utilizar el objeto *Font*. En las líneas 7 y 8 se instancian dos tipos de letras (de fuentes) a los que denominamos *UnaFuente* y *OtraFuente*, teniendo la segunda un tamaño superior a la primera (40 frente a 20).

Las líneas 14 y 17 establecen el tipo de fuente de los objetos (las etiquetas *HolaAmigo1* y *HolaAmigo2*) a los que se aplica el método *setFont*, que pertenece a la clase *Component*. Obsérvese que la mayor parte de las clases de AWT que estamos estudiando son subclases de *Component* y por lo tanto heredan su método *setFont*. Esta es la clave que nos permite aplicar fuentes a componentes. Lo mismo ocurre con el objeto *Color* y los métodos *setForeground* (líneas 15 y 18) y *setBackground*.

```
1  import java.awt.*;
2
3  public class Fuente {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Font UnaFuente = new Font("SansSerif",Font.BOLD,20);
8          Font OtraFuente = new Font("Serif",Font.ITALIC,40);
9
10         Panel Sencillo = new Panel();
11         Label HolaAmigo1 = new Label("Hola amigo");
12         Label HolaAmigo2 = new Label("Hola amigo");
13
14         HolaAmigo1.setFont(UnaFuente);
15         HolaAmigo1.setForeground(Color.red);
16
17         HolaAmigo2.setFont(OtraFuente);
18         HolaAmigo2.setForeground(Color.orange);
19
20         Sencillo.add(HolaAmigo1);
21         Sencillo.add(HolaAmigo2);
22
23         MiMarco.add(Sencillo);
24         MiMarco.setSize(500,130);
25         MiMarco.setTitle("Ventana con etiquetas y fuentes");
26         MiMarco.setVisible(true);
27     }
28 }
```

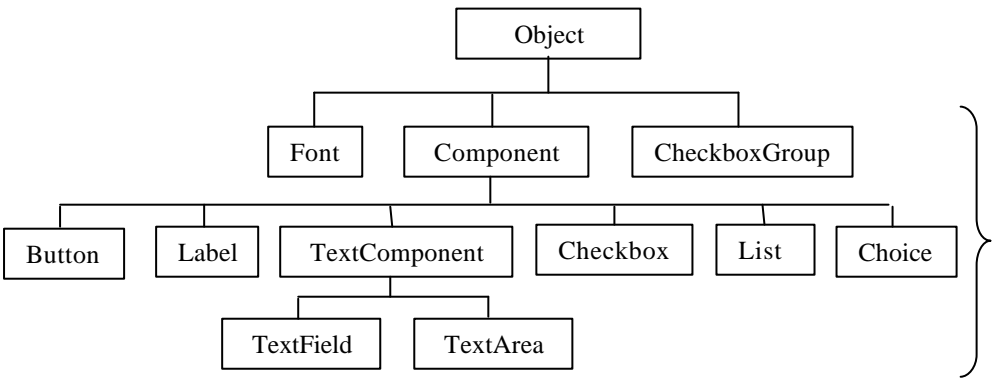


## 7.4 CAJAS DE VERIFICACIÓN, BOTONES DE RADIO Y LISTAS

### 7.4.1 Introducción

Entre los componentes habituales en el diseño de interfaces gráficas de usuario (GUI) se encuentran los botones, etiquetas, campos de texto y áreas de texto, todos ellos previamente estudiados. Ahora nos centraremos en las cajas de verificación (*Checkbox*), los botones de radio (*CheckboxGroup*), las listas (*List*) y las listas desplegables (*Choice*).

El siguiente diagrama sitúa todas estas clases en la jerarquía que proporciona Java:



### 7.4.2 Cajas de verificación (*Checkbox*)

Una caja de verificación es un componente básico que se puede encontrar en dos posibles estados: activado (*true*) y desactivado (*false*). Cuando se pulsa en una caja de verificación se conmuta de estado.

Los constructores más simples de la clase *Checkbox* son:

```
Checkbox()  
Checkbox(String Etiqueta)  
Checkbox(String Etiqueta, boolean Estado)
```

Entre los métodos existentes tenemos:

*setLabel (String Etiqueta)*  
*setState (boolean Estado)*

De esta manera podemos definir e instanciar cajas de verificación de diversas formas:

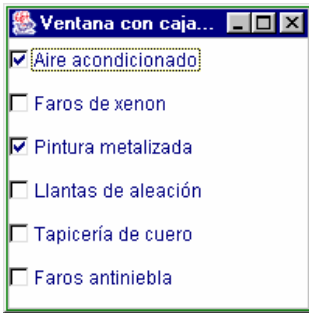
```
Checkbox MayorDeEdad = new Checkbox ("Mayor de edad", true);
```

```
Checkbox MayorDeEdad = new Checkbox ();  
MayorDeEdad.setLabel ("Mayor de edad");  
MayorDeEdad.setState ("true");
```

En la clase *CajaDeVerificación* se instancian varios componentes de tipo *Checkbox*, añadiéndolos a un panel *Sencillo* con disposición *GridLayout* 6 x 1. En la ventana obtenida se puede observar como los componentes instanciados con estado *true* aparecen inicialmente marcados.

```
1  import java.awt.*;  
2  
3  public class CajaDeVerificacion {  
4  
5      public static void main(String[] args) {  
6          Frame MiMarco = new Frame();  
7          Panel Sencillo = new Panel(new GridLayout(6,1));  
8  
9          Checkbox AireAcondicionado = new Checkbox("Aire  
10                                     acondicionado",true);  
11          Checkbox FarosXenon          = new Checkbox("Faros de  
12                                     xenon",false);  
13          Checkbox PinturaMetalizada = new Checkbox("Pintura  
14                                     metalizada",true);  
15          Checkbox LlantasAleacion    = new Checkbox("Llantas de  
16                                     aleación",false);  
17          Checkbox TapiceriaCuero     = new Checkbox("Tapicería de  
18                                     cuero",false);  
19          Checkbox FarosAntiniebla    = new Checkbox("Faros  
20                                     antiniebla",false);  
21  
22          Sencillo.add(AireAcondicionado);  
23          Sencillo.add(FarosXenon);  
24          Sencillo.add(PinturaMetalizada);  
25          Sencillo.add(LlantasAleacion);  
26          Sencillo.add(TapiceriaCuero);  
27          Sencillo.add(FarosAntiniebla);  
28  
29          MiMarco.add(Sencillo);  
30      }  
31  }
```

```
24     MiMarco.setSize(200,200);
25     MiMarco.setTitle("Ventana con cajas de verificación");
26     MiMarco.setVisible(true);
27 }
28 }
```



### 7.4.3 Botones de radio (*CheckboxGroup*)

La clase *CheckboxGroup* se utiliza para agrupar un conjunto de cajas de verificación. *CheckboxGroup* no es un componente (no deriva de la clase *Component*), es una subclase de *Object*. Cuando definimos un grupo de botones de radio utilizamos varias instancias del componente *Checkbox* asociadas a un objeto *CheckboxGroup*.

Los botones de radio presentan la característica de que la activación de un elemento implica directamente la desactivación automática de los demás botones del grupo, es decir no puede haber dos o más botones activos al mismo tiempo.

Sólo existe el constructor sin argumentos del objeto *CheckboxGroup*: *CheckboxGroup()*.

Cada caja de verificación se puede asociar a un *CheckboxGroup* de dos posibles maneras:

- Instanciando la caja de verificación con uno de los siguientes constructores:

*Checkbox (String Etiqueta, boolean Estado, CheckboxGroup GrupoDeBotonesDeRadio)*

*Checkbox (String Etiqueta, CheckboxGroup GrupoDeBotonesDeRadio, boolean Estado)*

- Utilizando el método `setCheckboxGroup` (`CheckboxGroup GrupoDeBotonesDeRadio`) sobre la caja de verificación.

El siguiente ejemplo (*BotonesDeRadio1*) instancia un grupo de botones de radio *Colores* en la línea 8. En las líneas 10 a 15 se crean seis cajas de verificación asociadas al grupo *Colores*. Estas 6 cajas de verificación se comportarán como un grupo de botones de radio (con selección excluyente).

```

1  import java.awt.*;
2
3  public class BotonesDeRadio1 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel Sencillo = new Panel(new GridLayout(6,1));
8          CheckboxGroup Colores = new CheckboxGroup();
9
10         Sencillo.add(new Checkbox("Rojo",false,Colores));
11         Sencillo.add(new Checkbox("Azul",false,Colores));
12         Sencillo.add(new Checkbox("Verde",false,Colores));
13         Sencillo.add(new Checkbox("Amarillo",false,Colores));
14         Sencillo.add(new Checkbox("Negro",false,Colores));
15         Sencillo.add(new Checkbox("Gris",false,Colores));
16
17         MiMarco.add(Sencillo);
18         MiMarco.setSize(200,200);
19         MiMarco.setTitle("Ventana con botones de radio");
20         MiMarco.setVisible(true);
21     }
22 }

```



Vamos a combinar el ejemplo anterior con el uso de fuentes (*Font*), de manera que las etiquetas de las cajas de verificación presenten un tamaño mayor y el color que representan. Los cambios más significativos respecto a nuestro ejemplo base son:



- En la línea 25 se crea una fuente *MiFuente* con tipo de letra *SansSerif*, plana (ni itálica ni negrilla) y de tamaño 25.
- En la línea 5 se define un método *EstableceVisualizacion* con argumentos: un objeto *Checkbox*, un objeto *Color* y un objeto *Font*. En la línea 8 se establece como fuente del objeto *Checkbox* el objeto *Font* que se pasa como parámetro. Análogamente, en la línea 9, se establece como color del objeto *Checkbox* el objeto *Color* que se pasa como parámetro.
- Las líneas 27 a 32 invocan al método *EstableceVisualizacion* con cada una de las cajas de verificación instanciadas en las líneas 18 a 23. En todos los casos se utiliza la misma fuente (*MiFuente*).

```
1  import java.awt.*;
2
3  public class BotonesDeRadio2 {
4
5      private static void EstableceVisualizacion(Checkbox
6          Elemento, Color ColorSeleccionado,
7          Font FuenteSeleccionada){
8          Elemento.setFont(FuenteSeleccionada);
9          Elemento.setForeground(ColorSeleccionado);
10     }
11
12
13     public static void main(String[] args) {
14         Frame MiMarco = new Frame();
15         Panel Sencillo = new Panel(new GridLayout(6,1));
16         CheckboxGroup Colores = new CheckboxGroup();
17
18         Checkbox Rojo      = new Checkbox("Rojo",false,Colores);
19         Checkbox Azul      = new Checkbox("Azul",false,Colores);
20         Checkbox Verde     = new
21             Checkbox("Verde",false,Colores);
22         Checkbox Amarillo  = new
23             Checkbox("Amarillo",false,Colores);
24         Checkbox Negro     = new
25             Checkbox("Negro",false,Colores);
26         Checkbox Gris      = new Checkbox("Gris",false,Colores);
27
28         Font MiFuente = new Font("SansSerif",Font.PLAIN,25);
29
30         EstableceVisualizacion(Rojo,Color.red,MiFuente);
31         EstableceVisualizacion(Azul,Color.blue,MiFuente);
32         EstableceVisualizacion(Verde,Color.green,MiFuente);
33         EstableceVisualizacion(Amarillo,Color.yellow,MiFuente);
34         EstableceVisualizacion(Negro,Color.black,MiFuente);
35         EstableceVisualizacion(Gris,Color.gray,MiFuente);
36     }
```

```
33
34     Sencillo.add(Rojo);
35     Sencillo.add(Azul);
36     Sencillo.add(Verde);
37     Sencillo.add(Amarillo);
38     Sencillo.add(Negro);
39     Sencillo.add(Gris);
40
41     MiMarco.add(Sencillo);
42     MiMarco.setSize(200,200);
43     MiMarco.setTitle("Ventana con botones de radio");
44     MiMarco.setVisible(true);
45 }
46 }
```



#### 7.4.4 Lista (List)

El componente lista le ofrece al usuario la posibilidad de desplazarse por una secuencia de elementos de texto. Este componente se puede configurar de manera que se puedan realizar selecciones múltiples o bien que sólo se pueda seleccionar un único elemento de la lista.

Los constructores que admite esta clase son:

*List ()*

*List (int NumeroDeFilas)*

*List (int NumeroDeFilas, boolean SeleccionMultiple)*

Haciendo uso del segundo constructor podemos configurar el tamaño de la lista para que se visualicen *NumeroDeFilas* elementos de texto. Si hay más elementos en la lista se puede acceder a ellos por medio de una barra de desplazamiento vertical. Con el tercer constructor indicamos si deseamos permitir la selección múltiple de elementos.

Podemos variar en tiempo de ejecución el modo de selección de elementos, utilizando el método *setMultipleMode(boolean SeleccionMultiple)* de la clase *List*.

La manera de incorporar elementos de texto en una lista es mediante uno de los siguientes métodos:

```
add (String Elemento)
add (String Elemento, int Posicion)
```

Con el primer método se añade el elemento especificado al final de la lista; si se desea insertarlo en una posición determinada se utiliza el segundo método.

En el siguiente ejemplo (*Lista1*) se instancia la lista *Islas* con 5 filas y sin posibilidad de selección múltiple (línea 8). Las líneas 10 a 16 añaden elementos a la lista.

```
1  import java.awt.*;
2
3  public class Lista1 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel MiPanel = new Panel();
8          List Islas = new List(5,false);
9
10         Islas.add("Tenerife");
11         Islas.add("Lanzarote");
12         Islas.add("Gran Canaria");
13         Islas.add("Hierro");
14         Islas.add("La Gomera");
15         Islas.add("Fuerteventura");
16         Islas.add("La Palma");
17
18         MiPanel.add(Islas);
19         MiMarco.add(MiPanel);
20         MiMarco.setSize(200,200);
21         MiMarco.setTitle("Ventana con lista");
22         MiMarco.setVisible(true);
23     }
24 }
```



La clase *Lista2* parte del ejemplo anterior. En este caso se utiliza un constructor más sencillo (línea 8) y se complementa haciendo uso del método *setMultipleMode* (línea 9), con el que se indica que se permite selección múltiple (necesario, por ejemplo si existe la posibilidad de paquetes turísticos combinados entre varias islas).

Las líneas 19 y 20 utilizan el método *select*, que permite seleccionar por programa el elemento indicado por la posición que se le pasa como parámetro. En este caso se han seleccionado “Gran Canaria” y “Lanzarote”. Obsérvese que la posición de los elementos se numera a partir de cero.

```
1  import java.awt.*;
2
3  public class Lista2 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel MiPanel = new Panel();
8          List Islas = new List(3);
9          Islas.setMultipleMode(true);
10
11         Islas.add("Fuerteventura");
12         Islas.add("La Gomera");
13         Islas.add("Gran Canaria");
14         Islas.add("Hierro");
15         Islas.add("Lanzarote");
16         Islas.add("Tenerife");
17         Islas.add("La Palma",5);
18
19         Islas.select(2);
20         Islas.select(4);
21
22         MiPanel.add(Islas);
23         MiMarco.add(MiPanel);
24         MiMarco.setSize(200,200);
25         MiMarco.setTitle("Ventana con lista de selección
                               múltiple");
26         MiMarco.setVisible(true);
27     }
28 }
```



### 7.4.5 Lista desplegable (Choice)

La clase *Choice* presenta un menú desplegable de posibilidades. La posibilidad seleccionada aparece como título del menú. No existe la posibilidad de realizar selecciones múltiples.

Esta clase sólo admite el constructor sin parámetros: *Choice()* y sus elementos se añaden mediante el método *add(String Elemento)*.

El ejemplo *ListaDesplegable1* ilustra la manera de hacer un uso sencillo del componente *Choice*: en la línea 8 se instancia la lista desplegable *Ciudades* y en las líneas 10 a 15 se añaden los elementos deseados.

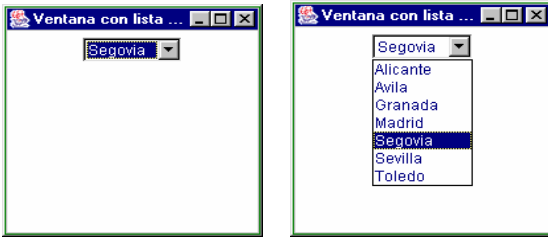
```
1  import java.awt.*;
2
3  public class ListaDesplegable1 {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7          Panel MiPanel = new Panel();
8          Choice Ciudades = new Choice();
9
10         Ciudades.add("Alicante");
11         Ciudades.add("Avila");
12         Ciudades.add("Granada");
13         Ciudades.add("Segovia");
14         Ciudades.add("Sevilla");
15         Ciudades.add("Toledo");
16
17         MiPanel.add(Ciudades);
18         MiMarco.add(MiPanel);
19         MiMarco.setSize(200,200);
20         MiMarco.setTitle("Ventana con lista desplegable");
21         MiMarco.setVisible(true);
22     }
23 }
```



Si en el ejemplo anterior introducimos entre las líneas 15 y 17 las siguientes instrucciones:

```
Ciudades.insert("Madrid",3); // inserta "Madrid" en la cuarta posición de la lista  
// desplegable "Ciudades"  
Ciudades.select("Segovia"); // selecciona el elemento de texto "Segovia"
```

Obtendremos el resultado:

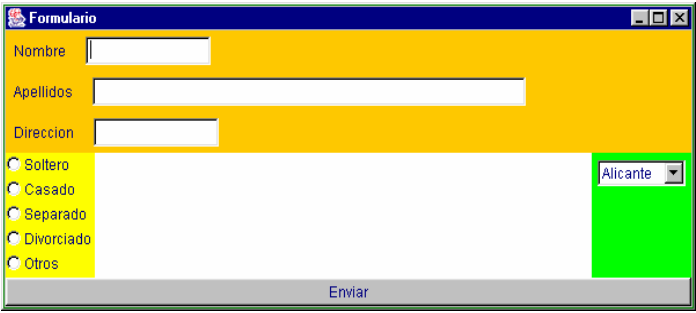


## 7.5 DISEÑO DE FORMULARIOS

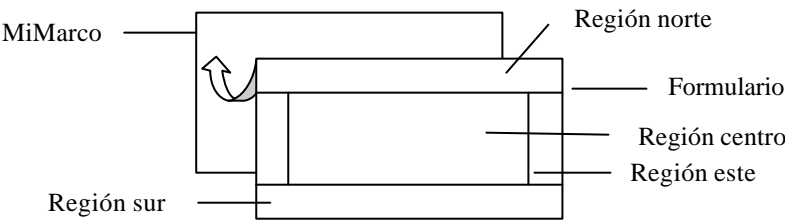
### 7.5.1 Diseño del interfaz gráfico deseado

Supongamos que deseamos crear una pantalla de entrada de datos que nos permita recibir información personal de usuarios, por ejemplo contribuyentes fiscales residentes en España.

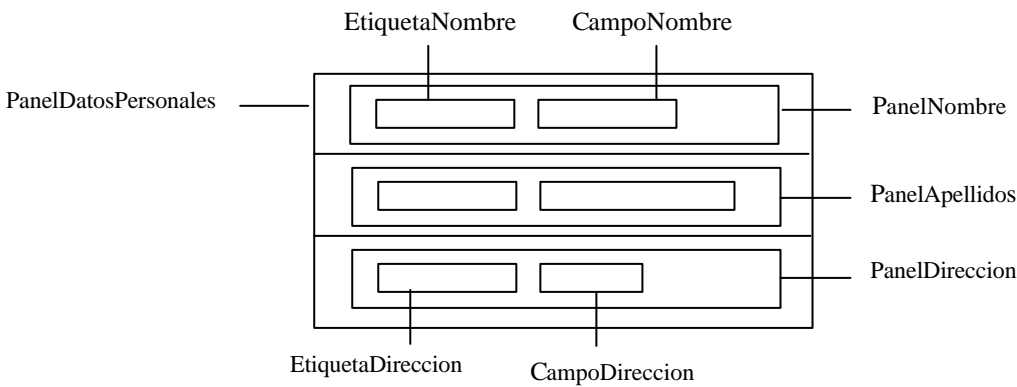
En la parte superior de la ventana deseamos preguntar el nombre y los apellidos, así como la nacionalidad, que por defecto se rellenará como española, puesto que sabemos que la gran mayoría de los contribuyentes poseen dicha nacionalidad. En la zona izquierda de la ventana se consulta el estado civil y en la derecha la ciudad de residencia; la zona central se deja libre para incluir el logotipo del ministerio de hacienda, de lo que se encargará un equipo de diseño gráfico, y la inferior contendrá el botón de enviar/grabar datos. El resultado esperado tendrá un aspecto similar a la ventana siguiente:



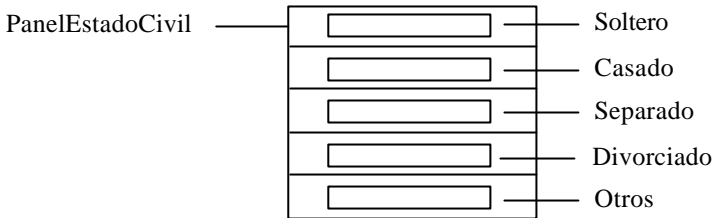
Antes de la programación deberíamos diseñar el formulario en base a contenedores (paneles) y disposiciones (*layouts*). En primer lugar necesitamos una ventana, que conseguiremos mediante un objeto *Frame* al que podemos denominar, igual que en los ejemplos anteriores, *MiMarco*. A *MiMarco* le añadiremos un panel “Formulario” con disposición *BorderLayout*, que encaja perfectamente con el diseño pedido.



El grupo de datos nombre, apellidos y nacionalidad requieren, cada uno, de una etiqueta y un campo de texto dispuestos en secuencia horizontal (*FlowLayout*) y alineados a la izquierda. A su vez estos tres grupos de componentes se deben situar en secuencia vertical, para lo que resulta adecuado un *GridLayout* 3 x 1.



Los botones de radio que sirven para seleccionar el estado civil se pueden añadir a un único panel “PanelEstadoCivil” asociado a un *GridLayout* 5 x 1:



El resto de los componentes no requiere de una disposición especial, pudiendo añadirse directamente a sus regiones correspondientes en el panel *Formulario*.

### 7.5.2 Implementación en una sola clase

Si pensamos que los elementos de nuestro formulario no se van a poder reutilizar en otras ventanas de GUI podemos implementar todo el diseño en una sola clase, el siguiente ejemplo (*Formulario1*) codifica el diseño realizado:

En primer lugar se instancian el marco y los paneles que hemos previsto (líneas 5 a 10), como variables globales a la clase, privadas a la misma y estáticas para poder ser usadas desde el método estático *main*.

En el método principal (*main*), en primer lugar se hacen las llamadas a los métodos privados *PreparaDatosPersonales*, *PreparaEstadoCivil* y *PreparaProvincia* (líneas 66 a 68), que añaden los componentes necesarios a los paneles *PanelDatosPersonales*, *PanelEstadoCivil* y *PanelProvincia*. Estos métodos podrían haber admitido un panel como parámetro, evitándose las variables globales y estáticas, que en este caso se habrían declarado en el propio método *main*.

En las líneas 70 a 72 se asignan los colores deseados a cada uno de los paneles principales. En las líneas 74 a 77 se añaden los paneles principales a las regiones deseadas en el panel *Formulario*. El código de cada uno de los métodos privados se obtiene de manera directa sin más que seguir los gráficos diseñados en el apartado anterior.

```
1 import java.awt.*;
2
```



```
3 public class Formulario1 {
4
5     private static Frame MiMarco = new Frame();
6     private static Panel Formulario = new Panel(new
7         BorderLayout());
8     private static Panel PanelDatosPersonales =
9         new Panel(new
10             GridLayout(3,1));
11     private static Panel PanelEstadoCivil = new Panel(new
12         GridLayout(5,1));
13     private static Panel PanelProvincia = new Panel();
14
15     private static void PreparaDatosPersonales(){
16         Panel PanelNombre = new Panel(new
17             FlowLayout(FlowLayout.LEFT));
18         Panel PanelApellidos = new Panel(new
19             FlowLayout(FlowLayout.LEFT));
20         Panel PanelDireccion = new Panel(new
21             FlowLayout(FlowLayout.LEFT));
22
23         TextField CampoNombre = new TextField(12);
24         TextField CampoApellidos = new TextField(50);
25         TextField CampoDireccion = new TextField(12);
26         Label EtiquetaNombre = new Label("Nombre",Label.LEFT);
27         Label EtiquetaApellidos = new
28             Label("Apellidos",Label.LEFT);
29         Label EtiquetaDireccion = new
30             Label("Direccion",Label.LEFT);
31
32         PanelDatosPersonales.add(PanelNombre);
33         PanelDatosPersonales.add(PanelApellidos);
34         PanelDatosPersonales.add(PanelDireccion);
35         PanelNombre.add(EtiquetaNombre);
36         PanelNombre.add(CampoNombre);
37         PanelApellidos.add(EtiquetaApellidos);
38         PanelApellidos.add(CampoApellidos);
39         PanelDireccion.add(EtiquetaDireccion);
40         PanelDireccion.add(CampoDireccion);
41     }
42
43     private static void PreparaEstadoCivil() {
44         CheckboxGroup EstadoCivil = new CheckboxGroup();
45
46         Checkbox Soltero = new
47             Checkbox("Soltero",false,EstadoCivil);
48         Checkbox Casado = new
49             Checkbox("Casado",false,EstadoCivil);
50         Checkbox Separado = new
```

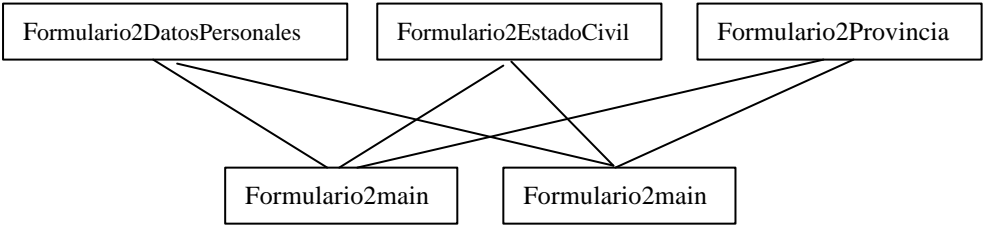
```

                                Checkbox("Separado", false, EstadoCivil);
41     Checkbox Divorciado = new
                                Checkbox("Divorciado", false, EstadoCivil);
42     Checkbox Otros      = new
                                Checkbox("Otros", false, EstadoCivil);
43
44     PanelEstadoCivil.add(Soltero);
45     PanelEstadoCivil.add(Casado);
46     PanelEstadoCivil.add(Separado);
47     PanelEstadoCivil.add(Divorciado);
48     PanelEstadoCivil.add(Otros);
49 }
50
51 private static void PreparaProvincia() {
52     Choice Ciudades = new Choice();
53
54     Ciudades.add("Alicante");
55     Ciudades.add("Avila");
56     Ciudades.add("Granada");
57     Ciudades.add("Madrid");
58     Ciudades.add("Segovia");
59     Ciudades.add("Sevilla");
60     Ciudades.add("Toledo");
61
62     PanelProvincia.add(Ciudades);
63 }
64
65 public static void main(String[] args) {
66     PreparaDatosPersonales();
67     PreparaEstadoCivil();
68     PreparaProvincia();
69
70     PanelDatosPersonales.setBackground(Color.orange);
71     PanelEstadoCivil.setBackground(Color.yellow);
72     PanelProvincia.setBackground(Color.green);
73
74     Formulario.add(PanelDatosPersonales,
                    BorderLayout.NORTH);
75     Formulario.add(PanelEstadoCivil, BorderLayout.WEST);
76     Formulario.add(PanelProvincia, BorderLayout.EAST);
77     Formulario.add(new
                    Button("Enviar"), BorderLayout.SOUTH);
78
79     MiMarco.add(Formulario);
80     MiMarco.setSize(600, 250);
81     MiMarco.setTitle("Formulario");
82     MiMarco.setVisible(true);
83 } }
```

### 7.5.3 Implementación en varias clases

El formulario diseñado contiene secciones que posiblemente nos interese reutilizar en otras ventanas de interfaz gráfico, por ejemplo, el panel que permite introducir el nombre, apellidos y nacionalidad con toda probabilidad podrá ser utilizado de nuevo en la misma aplicación o bien en otras diferentes.

Para facilitar la reutilización al máximo, crearemos una clase por cada uno de los paneles principales y después crearemos una o varias clases que hagan uso de las anteriores. El esquema que seguiremos es el siguiente:



En primer lugar presentamos la clase *Formulario2DatosPersonales*, en donde creamos la propiedad privada *PanelDatosPersonales* (línea 5). Esta propiedad es accesible a través del método público *DamePanel()* situado en la línea 18.

La clase tiene un constructor con argumento de tipo *Color* (línea 7), lo que nos permite asignar el color seleccionado al panel *PanelDatosPersonales* (línea 11). El segundo constructor de la clase (línea 14) no presenta parámetros; internamente llama al primero con el parámetro de color blanco.

El contenido del constructor se obtiene directamente del método *PreparaDatosPersonales()*.

```
1  import java.awt.*;
2
3  public class Formulario2DatosPersonales {
4
5      private Panel PanelDatosPersonales = new Panel(new
6                                          GridLayout(3,1));
7      Formulario2DatosPersonales(Color ColorDelPanel){
8
9          // *** Codigo del metodo PreparaDatosPersonales() ***
10
11          PanelDatosPersonales.setBackground(ColorDelPanel);
```

```
12  }
13
14  Formulario2DatosPersonales(){
15      this(Color.white);
16  }
17
18  Panel DamePanel() {
19      return PanelDatosPersonales;
20  }
21
22 }
```

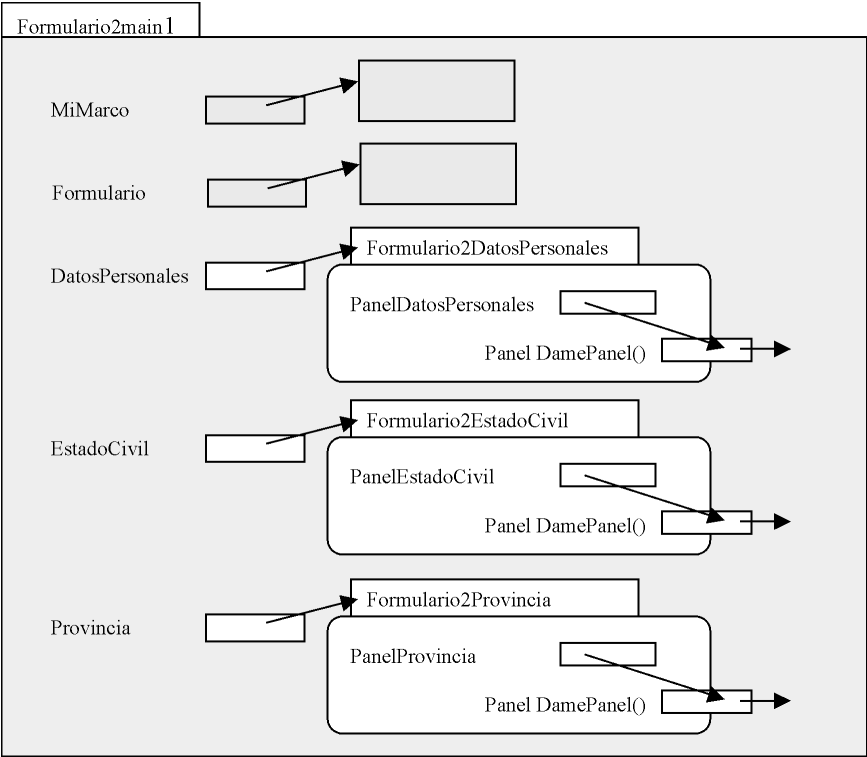
La clase *Formulario2EstadoCivil* se diseña con los mismos principios que *Formulario2DatosPersonales*. A continuación se muestra su código abreviado:

```
1  import java.awt.*;
2
3  public class Formulario2EstadoCivil {
4
5      private Panel PanelEstadoCivil = new Panel(new
                                                GridLayout(5,1));
6
7      Formulario2EstadoCivil(Color ColorDelPanel) {
8
9          // ***Codigo del metodo PreparaEstadoCivil() ***
10
11          PanelEstadoCivil.setBackground(ColorDelPanel);
12      }
13
14      Formulario2EstadoCivil(){
15          this(Color.white);
16      }
17
18      Panel DamePanel() {
19          return PanelEstadoCivil;
20      }
21
22 }
```

Finalmente se presenta el código resumido de la clase *Formulario2Provincia*:

```
1  import java.awt.*;
2
3  public class Formulario2Provincia {
4
5      private Panel PanelProvincia = new Panel();
```

```
6
7  Formulario2Provincia(Color ColorDelPanel){
8
9      // *** Codigo del metodo PreparaProvincia() ***
10
11      PanelProvincia.setBackground(ColorDelPanel);
12  }
13
14  Formulario2Provincia() {
15      this(Color.white);
16  }
17
18  Panel DamePanel() {
19      return PanelProvincia;
20  }
21
22 }
```



La clase *Formulario2main1* presenta exactamente el mismo resultado que la clase *Formulario1*, por ello no representamos de nuevo la ventana de resultado.

En primer lugar se crean las propiedades de clase *MiMarco* y *Formulario* (también se podrían haber creado como variables de instancia dentro del método *main*), después, en el método *main*, se crean sendas instancias de los objetos *Formulario2DatosPersonales*, *Formulario2EstadoCivil* y *Formulario2Provincia* (líneas 9 a 13), cada uno con el color deseado.

En este momento se encuentran preparados (como estructuras de datos) los paneles *PanelDatosPersonales*, *PanelEstadoCivil* y *PanelProvincia*, cada uno como propiedad privada de su objeto correspondiente (*DatosPersonales*, *EstadoCivil* y *Provincia*).

En las líneas 15 a 17 se obtienen los paneles *PanelDatosPersonales*, *PanelEstadoCivil* y *PanelProvincia* invocando al método *DamePanel()* en las instancias *DatosPersonales*, *EstadoCivil* y *Provincia*. En estas mismas líneas se añaden los paneles obtenidos a las regiones deseadas del panel *Formulario*.

```
1  import java.awt.*;
2
3  public class Formulario2main1 {
4
5      private static Frame MiMarco = new Frame();
6      private static Panel Formulario = new Panel(new
                                   BorderLayout());
7
8      public static void main(String[] args) {
9          Formulario2DatosPersonales DatosPersonales =
10              new Formulario2DatosPersonales(Color.orange);
11          Formulario2EstadoCivil EstadoCivil =
12              new Formulario2EstadoCivil(Color.yellow);
13          Formulario2Provincia Provincia = new
                                   Formulario2Provincia(Color.green);
14
15          Formulario.add(DatosPersonales.DamePanel(),
                                   BorderLayout.NORTH);
16          Formulario.add(EstadoCivil.DamePanel(),
                                   BorderLayout.WEST);
17          Formulario.add(Provincia.DamePanel(),
                                   BorderLayout.EAST);
18          Formulario.add(new Button("Enviar"),
                                   BorderLayout.SOUTH);
19
20          MiMarco.add(Formulario);
21          MiMarco.setSize(600,250);
22          MiMarco.setTitle("Formulario");
```

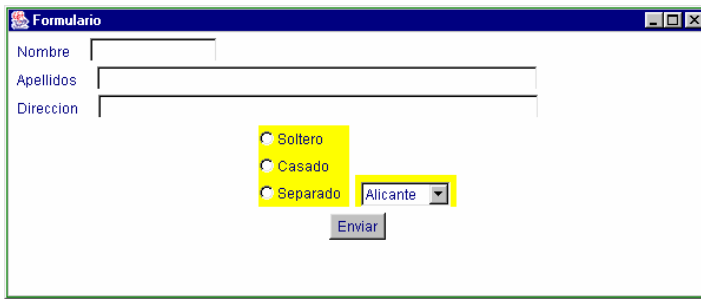
```

23     MiMarco.setVisible(true);
24 }
25 }

```

La clase *Formulario2main2* hace uso de los objetos *Formulario2DatosPersonales*, *Formulario2EstadoCivil* y *Formulario2Provincia* para crear un interfaz de usuario con la misma funcionalidad que la del ejemplo anterior, pero con una disposición diferente de los componentes.

La ventana de resultado se presenta a continuación y la naturaleza del código es muy similar a la del ejemplo anterior. Al reutilizar las clases básicas tenemos la posibilidad de cambiar la apariencia final de la ventana de interfaz de usuario.



```

1  import java.awt.*;
2
3  public class Formulario2main2 {
4
5      private static Frame MiMarco = new Frame();
6      private static Panel Formulario = new Panel(new
                                           GridLayout(3,1));
7
8      public static void main(String[] args) {
9
10         Panel EstadoCivil_Provincia =
11             new Panel(new FlowLayout(FlowLayout.CENTER));
12         Panel BotonEnviar = new Panel();
13
14         Formulario2DatosPersonales DatosPersonales =
15             new Formulario2DatosPersonales();
16         Formulario2EstadoCivil EstadoCivil =
17             new Formulario2EstadoCivil(Color.yellow);
18         Formulario2Provincia Provincia = new
19             Formulario2Provincia(Color.yellow);

```

```
20     EstadoCivil_Provincia.add(EstadoCivil.DamePanel());
21     EstadoCivil_Provincia.add(Provincia.DamePanel());
22     BotonEnviar.add(new Button("Enviar"));
23
24     Formulario.add(DatosPersonales.DamePanel());
25     Formulario.add(EstadoCivil_Provincia);
26     Formulario.add(BotonEnviar);
27
28     MiMarco.add(Formulario);
29     MiMarco.setSize(600,250);
30     MiMarco.setTitle("Formulario");
31     MiMarco.setVisible(true);
32 }
33 }
```

Este mismo ejercicio se podría haber resuelto con un enfoque más elegante desde el punto de vista de programación orientada a objetos: podríamos haber definido las clases *Formulario2DatosPersonales*, *Formulario2EstadoCivil* y *Formulario2Provincia* como subclases de *Panel*. En ese caso no necesitaríamos crear instancias del objeto *Panel* dentro de cada una de las tres clases, ni tampoco haría falta definir y utilizar el método *DamePanel()*.

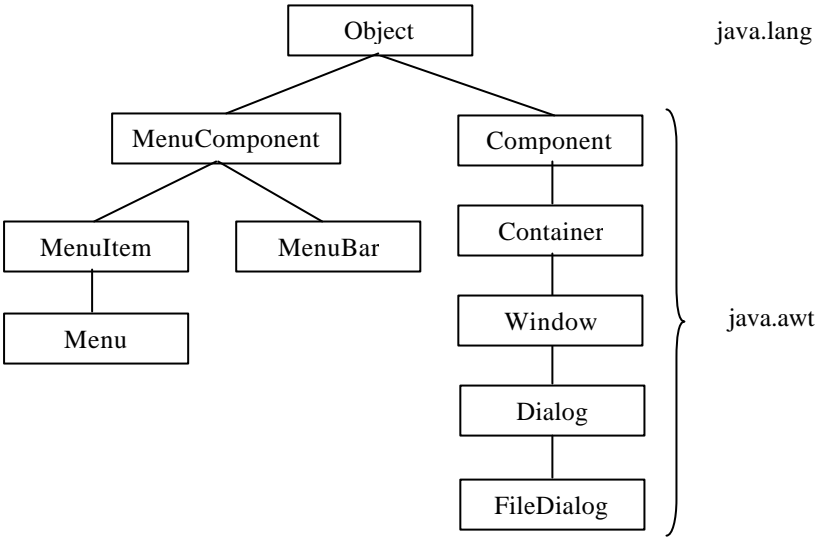
## 7.6 DIÁLOGOS Y MENÚS

### 7.6.1 Introducción

Los siguientes componentes de AWT que vamos a estudiar son los diálogos, con los que podemos crear ventanas adicionales a la principal, especializadas en diferentes tipos de interacción con el usuario (por ejemplo selección de un fichero). También nos centraremos en la manera de crear menús.

Los diálogos los basaremos en las clases *Dialog* y *FileDialog*, mientras que los menús los crearemos con los objetos *MenuBar* y *Menu*. El siguiente diagrama sitúa todas estas clases en la jerarquía que proporciona Java:





7.6.2 *Diálogo (Dialog)*

Un diálogo es una ventana que se usa para recoger información del usuario; su disposición por defecto es *BorderLayout*. Para poder crear un diálogo hay que indicar quién es su propietario (un marco o bien otro diálogo).

Las ventanas de diálogo pueden ser modales o no modales (opción por defecto). Las ventanas de diálogo modales se caracterizan porque realizan un bloqueo de las demás ventanas de la aplicación (salvo las que hayan podido ser creadas por el propio diálogo).

Existen 5 constructores de la clase *Dialog*, los más utilizados son:

- Dialog (Frame Propietario, String Titulo, boolean Modal)*
- Dialog (Dialog Propietario, String Titulo, boolean Modal)*
- Dialog (Frame Propietario)*
- Dialog (Dialog Propietario)*

Entre los métodos existentes tenemos:

- setTitle (String Titulo)*
- setModal (boolean Modal)*
- hide() // oculta el diálogo*
- show() // muestra el diálogo*
- setResizable(boolean CambioTamano) // permite o no el cambio de tamaño de la ventana por parte del usuario*

El primer ejemplo del componente diálogo (*Dialogo1*), en la línea 9 instancia un diálogo *Dialogo* cuyo propietario es el marco *MiMarco*, el título “Ventana de diálogo” y su naturaleza no modal (parámetro a *false*).

La línea 14 invoca al método *show* del objeto *Dialogo* para conseguir que la ventana se haga visible. El diálogo que se obtiene es una ventana vacía y sin dimensiones.

```
1  import java.awt.*;
2
3  public class Dialogo1 {
4
5      public static void main(String[] args) {
6          final boolean NO_MODAL = false;
7          Frame MiMarco = new Frame();
8
9          Dialog Dialogo = new Dialog(MiMarco,"Ventana de
                                diálogo",NO_MODAL);
10
11         MiMarco.setSize(200,100);
12         MiMarco.setTitle("Ventana con diálogo");
13         MiMarco.setVisible(true);
14         Dialogo.show();
15     }
16 }
```



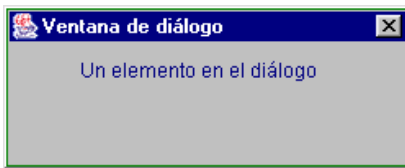
En la clase *Dialogo2* se crea un diálogo modal de propietario *MiMarco* (línea 11). Con el diálogo (*Dialogo*) instanciado trabajamos de manera análoga a como lo hemos venido haciendo con los marcos. En el ejemplo se le añade un panel (línea 12), se le asigna un tamaño (línea 13) y también una posición (línea 14). Al igual que en el ejemplo anterior, debemos mostrar el componente (línea 19).

```
1  import java.awt.*;
2
3  public class Dialogo2 {
4
5      public static void main(String[] args) {
6          final boolean MODAL = true;
7          Frame MiMarco = new Frame();
8          Panel MiPanel = new Panel();
9          MiPanel.add(new Label("Un elemento en el diálogo"));
10
11         Dialog Dialogo = new Dialog(MiMarco,"Ventana de
```

```

12         dialogo.add(MiPanel);
13         dialogo.setSize(250,100);
14         dialogo.setLocation(new Point(50,80));
15
16         MiMarco.setSize(200,100);
17         MiMarco.setTitle("Ventana con diálogo");
18         MiMarco.setVisible(true);
19         dialogo.show();
20     }
21 }

```



### 7.6.3 Diálogo de carga / almacenamiento de ficheros (FileDialog)

*FileDialog* es una subclase de *Dialog*. Como clase especializada de *Dialog*, *FileDialog* gestiona la selección de un fichero entre los sistemas de ficheros accesibles. Los diálogos de fichero son modales y sus propietarios deben ser marcos.

Los constructores de la clase son:

*FileDialog (Frame Propietario)*

*FileDialog (Frame Propietario, String Titulo)*

*FileDialog (Frame Propietario, String Titulo, int ModoCargaOGrabacion)*

Entre los métodos más utilizados de esta clase están:

*getMode()* y *setMode(int ModoCargaOGrabacion)*

*getFile()* y *setFile(String NombreFichero)* // para seleccionar por programa el  
 // fichero (o consultar cuál  
 // es el fichero que ha sido seleccionado)

*getDirectory* y *setDirectory(String NombreDirectorio)* // para seleccionar el  
 // directorio inicial que

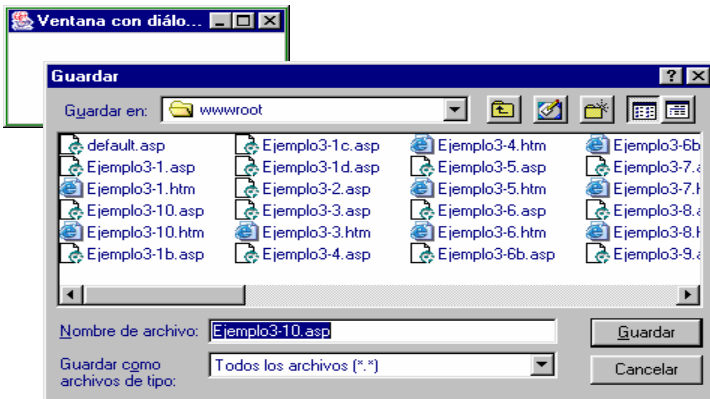
// utilizará el diálogo (o consultarlo)  
*getFilenameFilter()* y *setFilenameFilter(FilenameFilter Filtro)* // para mostrar  
 // únicamente los ficheros que pasan el filtro

En el siguiente ejemplo (*DialogoFichero*) se crea una instancia *Grabar* de la clase *FileDialog* (línea 8). El diálogo tiene como propietario el marco *MiMarco*, como título el texto “Guardar” y como modo *SAVE*. La línea 9 (comentada) ilustra la forma de crear un diálogo de fichero en modo *LOAD*.

En la línea 15 se invoca al método *show*, perteneciente a la clase *Dialog* (superclase de *FileDialog*).

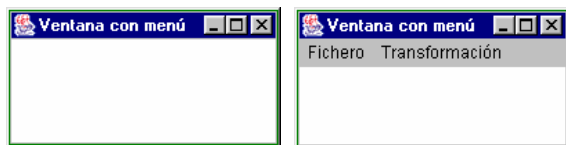
```

1  import java.awt.*;
2
3  public class DialogoFichero {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7
8          FileDialog Grabar = new
9              FileDialog(MiMarco, "Guardar", FileDialog.SAVE);
10         // FileDialog Cargar =
11             new FileDialog(MiMarco, "Cargar",
12                 FileDialog.LOAD);
13
14         MiMarco.setSize(200,100);
15         MiMarco.setTitle("Ventana con diálogo de carga
16                             de fichero");
17         MiMarco.setVisible(true);
18         Grabar.show();
19     }
20 }
```



### 7.6.4 Menús (*Menu* y *MenuBar*)

La clase *MenuBar* se utiliza para dotar de una barra principal de menú a un marco. A continuación se presenta un marco sin barra de menú y otro con una barra de menú compuesto de dos menús.



*MenuBar* solo tiene el constructor sin argumentos. En la línea 8 de la clase *Menus* se instancia una barra de menú con identificador *MenuPrincipal*.

Para añadir una barra de menú a un marco se emplea el método *setMenuBar* (no *add*), tal y como aparece en la línea 24 del ejemplo; *setMenuBar* es un método de la clase *Frame*.

Una vez que disponemos de un objeto barra de menú, podemos añadirle menús (con el método *add*), tal y como se realiza en las líneas 21 y 22 del ejemplo. Veamos primero qué es y como se emplea un menú: un menú es un componente desplegable que parte de una barra de menús. Su representación gráfica se puede ver en las ventanas situadas al final del código de la clase *Menus*.

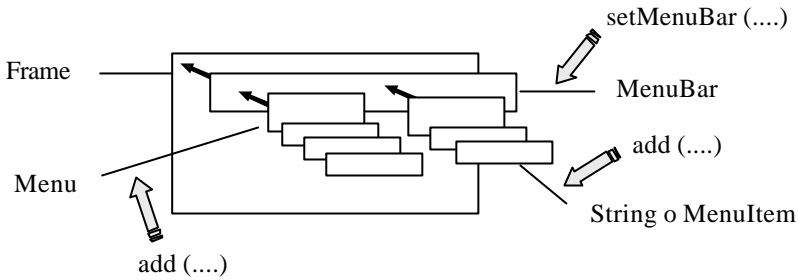
El constructor más utilizado de la clase menú es:

*Menu* (*String* *Etiqueta*)

El constructor definido crea un menú vacío con la etiqueta de título que se le pasa como parámetro; en nuestro ejemplo tenemos un menú *Fichero* con etiqueta “Fichero” (línea 9) y un menú *Transformaciones* con etiqueta “Transformación” (línea 10). La etiqueta representa el texto que se visualiza en la barra de menús.

En las líneas 12 a 19 del ejemplo se añaden (usando el método *add*) las diferentes opciones que el usuario puede seleccionar en los menús.

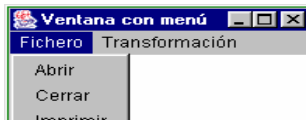
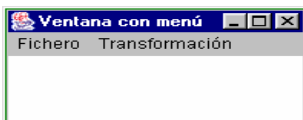
El siguiente esquema muestra la manera en la que se disponen las clases involucradas, así como los métodos necesarios para asociarlas entre sí.



```

1  import java.awt.*;
2
3  public class Menus {
4
5      public static void main(String[] args) {
6          Frame MiMarco = new Frame();
7
8          MenuBar MenuPrincipal = new MenuBar();
9          Menu Fichero = new Menu("Fichero");
10         Menu Transformaciones = new Menu("Transformación");
11
12         Fichero.add("Abrir");
13         Fichero.add("Cerrar");
14         Fichero.add("Imprimir");
15         Fichero.add("Guardar");
16
17         Transformaciones.add("Rotación");
18         Transformaciones.add("Traslación");
19         Transformaciones.add("Cambio de escala");
20
21         MenuPrincipal.add(Fichero);
22         MenuPrincipal.add(Transformaciones);
23
24         MiMarco.setMenuBar(MenuPrincipal);
25         MiMarco.setSize(200,100);
26         MiMarco.setTitle("Ventana con menú");
27         MiMarco.setVisible(true);
28     }
29 }

```



# EVENTOS

---

## 8.1 MECANISMO DE EVENTOS EN JAVA

### 8.1 *Introducción*

Java es un lenguaje orientado a objetos, por lo que los objetos (las clases) son los elementos más importantes en el diseño y desarrollo de una aplicación. También podemos afirmar que Java es un lenguaje orientado a eventos, puesto que nos proporciona todos los elementos necesarios para definirlos y utilizarlos; los eventos son objetos fundamentales en el desarrollo de la mayor parte de las aplicaciones.

Los eventos de Java proporcionan un mecanismo adecuado para tratar situaciones que habitualmente se producen de manera asíncrona a la ejecución del programa; situaciones normalmente producidas desde el exterior de la aplicación, por ejemplo cuando se pulsa una tecla. Cuando llega un evento, en ocasiones nos interesa tratarlo (por ejemplo la pulsación de un número en una aplicación calculadora) y otras veces no deseamos tratar el evento con ninguna acción (por ejemplo cuando el usuario pulsa con el ratón sobre un texto al que no hemos asignado ninguna información complementaria).

Para afianzar el significado de los eventos resulta conveniente enumerar algunos de los más comunes que se pueden producir en la ejecución de una aplicación que proporcione interfaz gráfico de usuario (GUI):

- Pulsación de ratón
- El puntero del ratón “entra en” (se sitúa sobre) un componente gráfico (por ejemplo sobre un botón o una etiqueta)
- El puntero del ratón “sale de” un componente gráfico
- Se pulsa una tecla
- Se cierra una ventana

- etc.

De esta manera, a modo de ejemplo, podemos desear que en una aplicación que consta de GUI, cuando se pulse en un botón *Calcular* se ejecute un método *CalculoEstructura* que determina el diámetro mínimo que debe tener una columna de hormigón para soportar una estructura determinada. En este caso, cuando el usuario de la aplicación pulse en el botón *Calcular*, esperamos que se produzca un evento y que este evento me permita invocar al método *CalculoEstructura*.

## 8.2 Arquitectura de los eventos

Los eventos habitualmente se originan desde el exterior de la aplicación, producidos por los usuarios que hacen uso de la misma. El simple movimiento del ratón genera multitud de eventos que pueden ser tratados por nuestros programas Java. Veamos con mayor detalle como se pasa de las acciones de los usuarios a las ejecuciones que deseamos que se produzcan en los programas:

- 1 El usuario interacciona con la aplicación por medio de dispositivos de entrada/salida (ratón, teclado, etc.)
- 2 Los dispositivos de entrada/salida generan señales eléctricas (que codifican información) que son recogidas por los controladores (habitualmente placas controladoras de puerto serie, paralelo, USB, etc.)
- 3 Los drivers (manejadores) de cada dispositivo recogen las señales eléctricas, las codifican como datos y traspasan esta información al procesador (CPU). En concreto activan los pines de interrupción (excepción) de la CPU.
- 4 La CPU habitualmente deja de ejecutar la acción en curso (salvo que esta acción tenga mayor prioridad que la asociada a la interrupción que le llega) y pasa a ejecutar la rutina de tratamiento de interrupción (excepción) de entrada/salida que le ha asignado el sistema operativo.
- 5 El sistema operativo determina si tiene que tratar él mismo esta interrupción o bien si tiene que pasársela a alguna aplicación que se ejecuta sobre él (este sería nuestro caso). Si la interrupción se produjo sobre una ventana de nuestra aplicación Java, el sistema operativo traspasa a nuestra máquina virtual Java (JVM) la información de la interrupción.
- 6 La máquina virtual Java determina el componente sobre el que se ha producido la interrupción (pensemos en este momento en un botón que ha sido pulsado)
- 7 La máquina virtual Java consulta si se ha definido alguna acción a realizar como respuesta a esa interrupción sobre dicho componente. Si



no es así no se hace nada y la CPU continúa con la acción que estaba ejecutando cuando llego la interrupción.

- 8 Si se ha definido alguna respuesta:
  - a. JVM crea un objeto evento con la información de la acción
  - b. Se pasa el evento al objeto (clase) que tratará la interrupción
  - c. Se pasa el flujo de control (de ejecución) a la clase Java que hemos creado para tratar la interrupción.
  - d. Se pasa el flujo de control a la CPU para que continúe con la acción que estaba ejecutando cuando llego la interrupción.

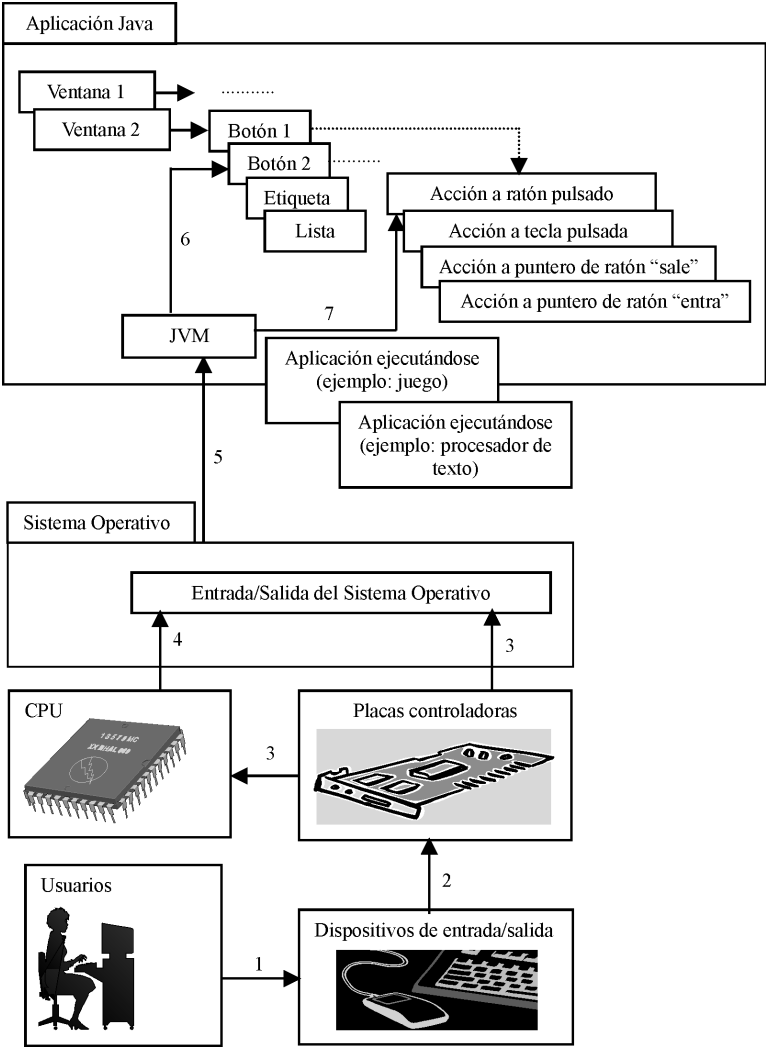
La figura siguiente muestra un esquema gráfico de las 7 primeras etapas expuestas.

Podemos observar como las interrupciones físicas que le llegan a la CPU (etapa 3) se convierten en eventos de alto nivel (etapa 8) que son tratados por la propia aplicación Java. Este mecanismo es adecuado, puesto que el sistema operativo puede tratar interrupciones relacionadas con el sistema (por ejemplo interrupciones de DMA indicando la finalización de la lectura de una pista del disco duro), pero no puede procesar lo que hay que hacer cuando se pulsa el botón *Calcular* de una aplicación Java.

La máquina virtual Java (JVM) realiza un papel fundamental en el mecanismo de eventos: recoge la información de interrupción del sistema operativo (etapa 5), determina la ventana y el componente de la ventana donde se ha producido el evento (etapa 6) y ejecuta la acción asociada al tipo de evento que llega, en caso de haber alguna (etapa 7).

En este momento se pueden obtener 3 conclusiones importantes relacionadas con los mecanismos de programación de eventos en Java:

- 1 Necesitamos objetos (clases) que nos definan las posibles acciones a realizar cuando llegan distintos tipos de eventos.
- 2 No es necesario definir acciones para todos los eventos ni para todos los objetos (por ejemplo podemos definir etiquetas sin ninguna acción asociada, también podemos definir una acción para el evento pulsar en un botón, pero dejar sin acción los eventos “al entrar” y “al salir” con el puntero del ratón sobre dicho botón).
- 3 Necesitamos un mecanismo para asociar acciones a los distintos eventos que se puedan producir sobre cada componente de nuestro GUI.



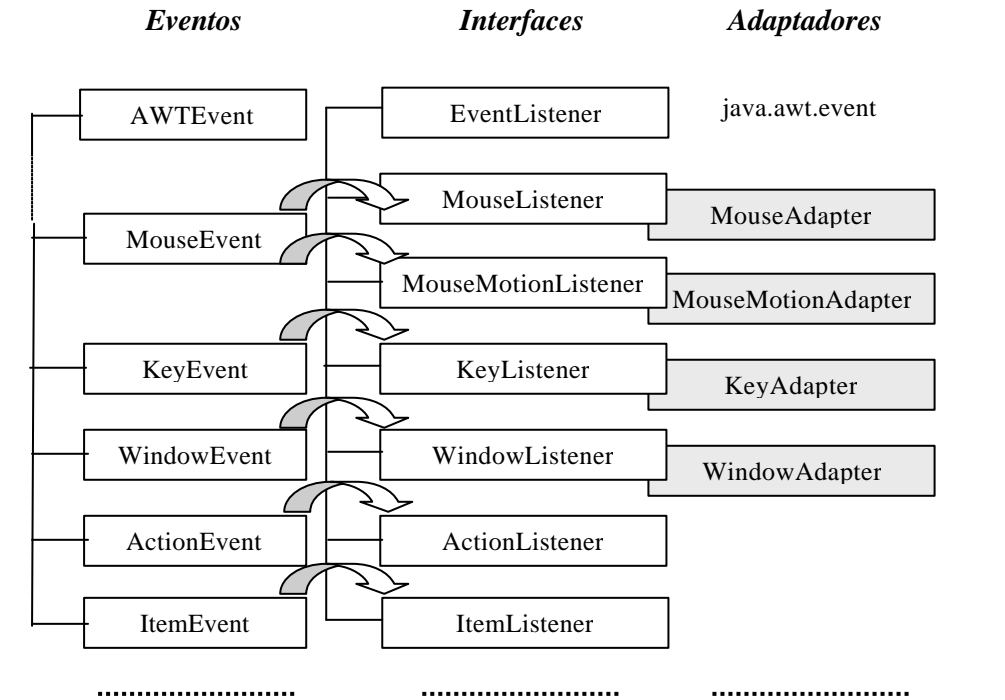
8.3 Interfaces que soportan el mecanismo de eventos

Java proporciona una serie de interfaces que agrupan métodos relacionados para el tratamiento de eventos. Los interfaces más comunes, normalmente relacionados con los GUI's basados en AWT, se encuentran en el paquete *java.awt.event*. Más adelante se detallará con ejemplos el funcionamiento de los más

utilizados; en este apartado se presentan y describen los principales interfaces y sus clases asociadas.

El mecanismo básico de eventos de Java se basa en la existencia de las clases de eventos y los interfaces “listeners” (escuchadores). Los listeners son interfaces que debemos implementar, colocando las acciones deseadas en sus métodos; también podemos basarnos en las implementaciones con métodos vacíos que proporciona el SDK: los “adaptadores”.

AWT proporciona una amplia gama de eventos que pueden ser recogidos por los métodos existentes en las implementaciones que hagamos de los listeners (o las clases que especializemos a partir de los adaptadores). El siguiente gráfico muestra los eventos, interfaces y adaptadores más utilizados en las aplicaciones de carácter general.



Para entender con mayor facilidad la manera con la que se tratan los eventos en Java, vamos a analizar uno de los interfaces existentes: *MouseListener*. Este interfaz contiene los siguientes métodos:

- `void mousePressed (MouseEvent e)`

- *void mouseReleased (MouseEvent e) )*
- *void mouseClicked (MouseEvent e)*
- *void mouseEntered (MouseEvent e)*
- *void mouseExited (MouseEvent e)*

El primer método se invoca cuando un botón del ratón ha sido pulsado en un componente, el segundo método se invoca en la acción contraria (soltar el botón). El tercer método se invoca cuando se han producido las dos acciones consecutivas anteriores (pulsar y soltar). Los dos últimos métodos se activan al entrar y salir, respectivamente, de un componente con el puntero del ratón.

De esta forma, si deseamos que un texto se ponga de color rojo al situarnos sobre él y de color gris al salir del mismo (con el ratón), deberemos crear una clase que implemente los métodos *mouseEntered* y *mouseExited*; el primer método se encargará de poner el texto en rojo y el segundo método se encargará de poner el texto en gris. Si deseamos que al hacer ‘click’ (subir y bajar) con el ratón un componente botón de un GUI se ejecute una acción, entonces debemos implementar la acción en el método *mouseClicked*.

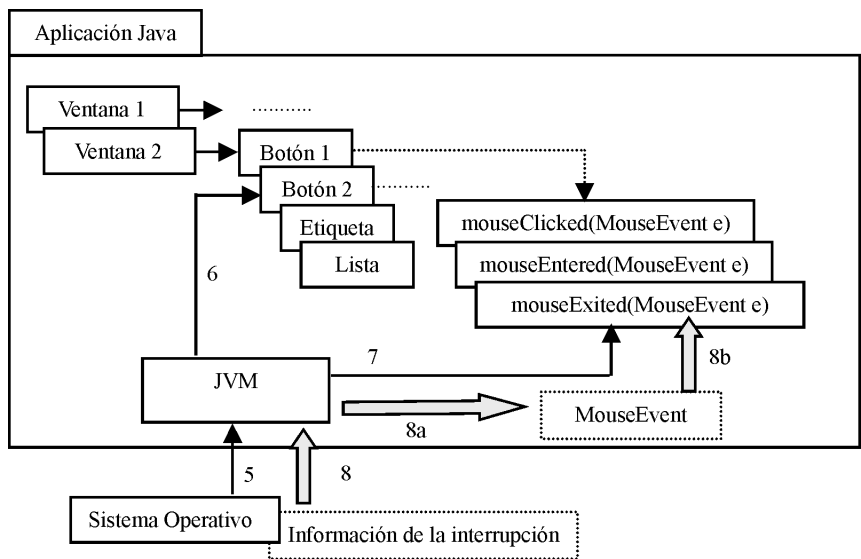
En todos estos casos podemos implementar el interface *MouseListener* o bien extender el adaptador *MosuseAdapter*. Nótese que aunque utilizar el adaptador es más cómodo, también es menos seguro. El problema que nos podemos encontrar es que hayamos escrito mal el nombre de algún método utilizado, por ejemplo *mousePresed(MouseEvent e)* en lugar de *mousePressed(MouseEvent e)*; en este caso no existe ningún problema sintáctico detectable por el compilador, pero el método *mousePressed(MouseEvent e)* nunca será invocado porque no ha sido definido. Si utilizamos los interfaces el compilador puede detectar este tipo de errores.

Todos los métodos que hemos analizado incluyen un parámetro de tipo *MouseEvent*. Como se puede apreciar en el gráfico anterior, *MouseEvent* es una clase que se le suministra a los métodos de los interfaces *MouseListener* y *MouseMotionListener*. El mecanismo de creación y paso de esta clase ha sido descrito en la etapa 8 del apartado anterior (arquitectura de eventos).

En el siguiente gráfico, las flechas de color gris representan:

1. La manera en la que la información de la interrupción se traspasa a la máquina virtual Java (8)
2. La creación de la instancia del tipo *MouseEvent* que JVM produce a partir de la información de la interrupción (8a)

- 3. El traspaso de la instancia creada al método invocado, vía el parámetro *e* de tipo *MouseEvent* (8b)



Los objetos de tipo evento, en nuestro ejemplo *MouseEvent*, nos permiten consultar información relativa a la interrupción que genera el evento. La clase *MouseEvent* nos proporciona entre otra información: el botón que ha sido pulsado, las coordenadas X e Y en pixels en las que se encontraba el puntero del ratón respecto al componente que ha generado la interrupción (el evento), el componente que ha generado la interrupción, etc.

8.4 Esquema general de programación

A lo largo de los apartados anteriores se ha explicado el funcionamiento general del mecanismo de eventos y se han mostrado los objetos que Java proporciona para su utilización. En este apartado desarrollaremos un esquema de programación que capture eventos y nos sirva como introducción a las siguientes explicaciones en las que se realizarán ejemplos completos de programación.

El esquema se basará en una porción de aplicación que permita las siguientes acciones:

- Cuando se pulse un botón determinado se realizará una acción

- Cuando el puntero del ratón se coloque sobre el botón, el color de fondo del mismo se pondrá rojo
- Cuando el puntero del ratón salga del botón, el color de fondo del mismo se pondrá gris

En primer lugar crearemos una clase que asigne las acciones deseadas a los eventos que hemos determinado. Podemos utilizar el interface *MouseListener* o la clase *MouseAdapter*.

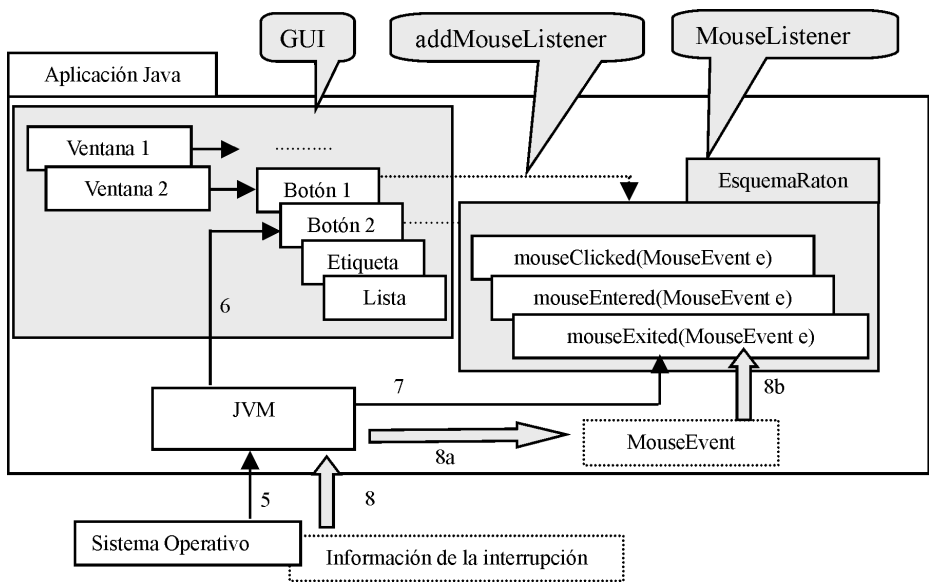
Utilizando el interface:

```
1 import java.awt.event.*;
2 import java.awt.Color;
3
4 public class EsquemaRaton implements MouseListener {
5
6     public void mouseClicked(MouseEvent EventoQueLlega){
7         // aqui se implementa la accion deseada
8     }
9
10    public void mousePressed(MouseEvent EventoQueLlega){
11    }
12
13    public void mouseReleased(MouseEvent EventoQueLlega){
14    }
15
16    public void mouseEntered(MouseEvent EventoQueLlega){
17        EventoQueLlega.getComponent().setBackground(Color.red);
18    }
19
20    public void mouseExited(MouseEvent EventoQueLlega){
21        EventoQueLlega.getComponent().
22                                setBackground(Color.gray);
23    }
24 }
```

Cuando se implementa un interfaz es obligatorio definir todos sus métodos, por lo que incluimos *mousePressed* y *mouseReleased* (líneas 10 y 13) aunque no programemos ninguna acción asociada a los mismos. En el método *mouseClicked* (línea 6) programaríamos la acción deseada, mientras que *mouseEntered* y *mouseExited* (líneas 16 y 20) se encargan de variar el color de fondo del componente que ha generado el evento.

Obsérvese como a través del objeto *MouseEvent* podemos conocer información que proviene del sistema operativo y la máquina virtual Java, en este caso el componente que ha generado el evento (a través del método *getComponent* en las líneas 17 y 21).

Para finalizar el esquema nos falta implementar un interfaz gráfico de usuario que contenga al menos un botón y enlazar de alguna manera los componentes deseados con la clase *EsquemaRaton*. La manera genérica de realizar este enlace es mediante los métodos *addMouseListener*, *addMouseMotionListener*, *addKeyListener*, etc. La mayoría de estos métodos pertenecen a la clase *Component* y son heredados por los distintos componentes de un GUI. Existen otros métodos “de enlace” que pertenecen a componentes concretos (por ejemplo la clase *Button* contiene el método *addActionListener*).



Siguiendo nuestro ejemplo, podemos implementar una clase que contenga un GUI con dos botones y asignar el “listener” *EsquemaRaton* a cada uno de ellos:

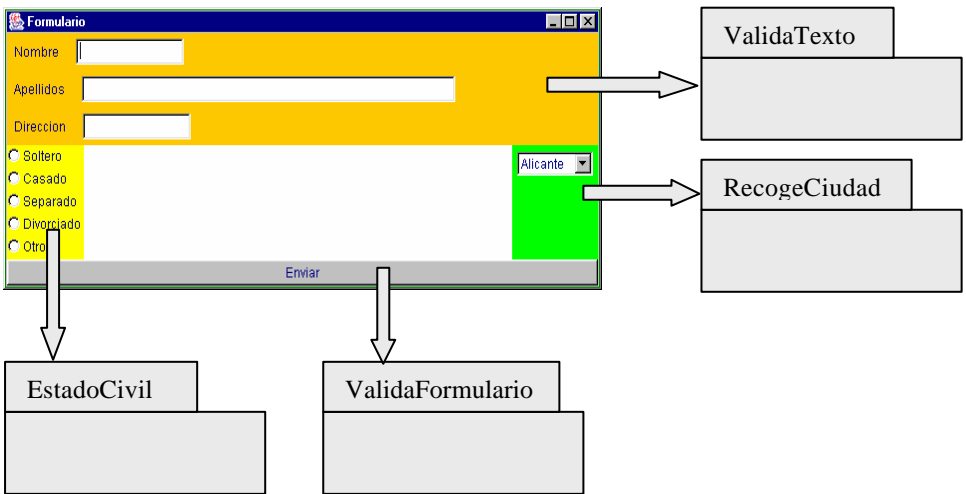
```
1 import java.awt.*;
2
3 public class PruebaEsquemaRaton {
4
5     public static void main(String[] args) {
6         Frame MiFrame = new Frame("Esquema de eventos");
7         Panel MiPanel = new Panel();
```

```

8      Button Hola = new Button("Saludo");
9      Button Adios = new Button("Despedida");
10     MiPanel.add(Hola); MiPanel.add(Adios);
11     MiFrame.add(MiPanel);
12     MiFrame.setSize(200,100);
13     MiFrame.show();
14
15     Hola.addMouseListener (new EsquemaRaton());
16     Adios.addMouseListener(new EsquemaRaton());
17
18 }
19 }
    
```

*PruebaEsquemaRaton* contiene los botones *Hola* y *Adios* (líneas 8 y 9) a los que se ha asociado el mismo “listener” *EsquemaRaton* (líneas 15 y 16). El resultado son dos botones que se comportan de idéntica forma: se ponen de color rojo o gris según se entre o se salga de cada uno de ellos.

Podemos implementar tantas clases “listeners” como comportamientos diferentes deseemos y asignar diferentes componentes a distintos “listeners”:

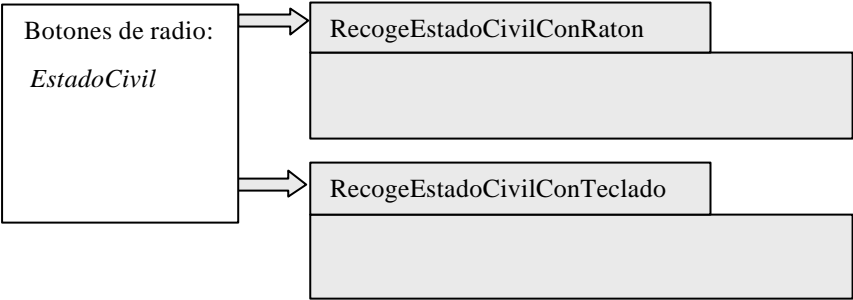


En el gráfico anterior, *EstadoCivil*, *ValidaFormulario*, *ValidaTexto* y *RecogeCiudad* son 4 “listeners” diferentes que realizarán las acciones adecuadas respecto a los componentes del GUI. Las flechas grises representan los distintos métodos “addxxxxListener” utilizados para asociar componentes con comportamientos.



Un componente no tiene por que estar limitado a la funcionalidad de un solo “listener”, por ejemplo podemos hacer que el estado civil pueda seleccionarse con el ratón o con el teclado, utilizando una letra significativa de cada estado civil posible; de esta manera el componente “EstadoCivil” podría asociarse a un *MouseListener* y a un *KeyListener*:

```
EstadoCivil.addMouseListener(new RecogeEstadoCivilConRaton());
EstadoCivil.addKeyListener(new RecogeEstadoCivilConTeclado());
```



## 8.2 EVENTOS DE RATÓN Y DE MOVIMIENTO DE RATÓN

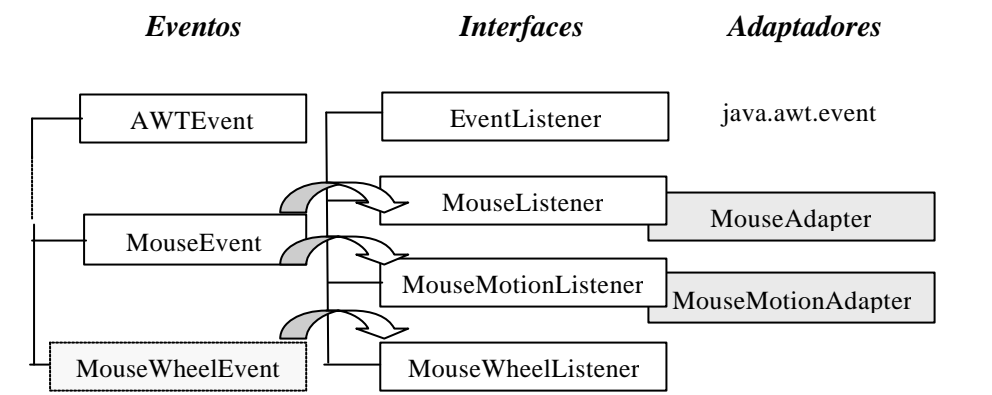
### 8.2.1 Introducción

Los eventos de ratón se capturan haciendo uso de los siguientes interfaces (o adaptadores) y eventos:

Interface	Adaptador	Evento
MouseListener	MouseAdapter	MouseEvent
MouseMotionListener	MouseMotionAdapter	MouseEvent
MouseWheelListener		MouseWheelEvent

Los dos primeros interfaces y adaptadores se usan mucho más a menudo que *MouseWheelListener*, por lo que centraremos las explicaciones en ellos. Como se puede observar en la tabla anterior, el evento *MouseEvent* se utiliza en los métodos pertenecientes a los interfaces *MouseListener* y *MouseMotionListener*. El interfaz

*MouseWheelListener* solo consta de un método, por lo que no resulta interesante incorporarle un adaptador.



A continuación se presenta un resumen de los objetos más importantes en el proceso de tratamiento de eventos de ratón:

MouseListener	
Método	El método se invoca cuando...
mouseClicked (MouseEvent e)	Se hace click (presión y liberación) con un botón del ratón
mouseEntered (MouseEvent e)	Se introduce el puntero del ratón en el interior de un componente
mouseExited (MouseEvent e)	Se saca el puntero del ratón del interior de un componente
mousePressed (MouseEvent e)	Se presiona un botón del ratón
mouseReleased (MouseEvent e)	Se libera un botón del ratón (que había sido presionado)

MouseMotionListener	
Método	El método se invoca cuando...
mouseDragged (MouseEvent e)	Se presiona un botón y se arrastra el ratón
mouseMoved (MouseEvent e)	Se mueve el puntero del ratón en un componente (sin pulsar un botón)

MouseWheelListener	
Método	El método se invoca cuando...
MouseWheelMoved (MouseWheelEvent e)	Se mueve la rueda del ratón

MouseEvent	
Métodos más utilizados	Explicación
int getButton()	Indica qué botón del ratón ha cambiado su estado (en caso de que alguno haya cambiado de estado). Se puede hacer la consulta comparando con los valores BUTTON1, BUTTON2 y BUTTON3
int getClickCount()	Número de clicks asociados con este evento
Point getPoint()	Devuelve la posición <i>x</i> , <i>y</i> en la que se ha generado este evento (posición relativa al componente)
int getX()	Devuelve la posición <i>x</i> en la que se ha generado este evento (posición relativa al componente)
int getY()	Devuelve la posición <i>y</i> en la que se ha generado este evento (posición relativa al componente)
Object getSource()	Método perteneciente a la clase <i>EventObject</i> (superclase de <i>MouseEvent</i> ). Indica el objeto que produjo el evento

### 8.2.2 Eventos de ratón

En este apartado vamos a desarrollar una serie de ejemplos que muestran el uso del interfaz *MouseListener*, la clase *MouseAdapter* y el evento *MouseEvent*. Utilizaremos ejemplos muy sencillos que ayuden a afianzar el mecanismo que Java ofrece para capturar eventos.

En primer lugar se presenta la clase *InterrupcionDeRaton1*, que implementa el interfaz *MouseListener* (línea 6). En la línea 1 se importan los objetos pertenecientes al paquete *java.awt.event*, donde se encuentran los que necesitamos en este momento: *MouseListener* y *MouseEvent*.

La clase *InterrupcionDeRaton1* implementa los 5 métodos del interfaz *MouseListener*, condición necesaria para que el compilador no nos genere un mensaje de error (a no ser que la declaremos abstracta). En este primer ejemplo la única acción que realizamos es imprimir por consola un mensaje identificando el evento que ha llegado.

```
1 import java.awt.event.*;
2
3 // Clase que recoge los eventos de raton mediante sus
  // metodos "mousePressed,
4 // mouseReleased, mouseClicked, etc." con parametro
  // "MouseEvent"
```

```
5
6 public class InterrupcionDeRaton1 extends Object implements
                                MouseListener {
7
8     public void mouseClicked(MouseEvent EventoQueLlega){
9         System.out.println("Click de raton");
10    }
11
12    public void mousePressed(MouseEvent EventoQueLlega){
13        System.out.println("Presion de raton");
14    }
15
16    public void mouseReleased(MouseEvent EventoQueLlega){
17        System.out.println("Se ha levantado el boton del
                                raton");
18    }
19
20    public void mouseEntered(MouseEvent EventoQueLlega){
21        System.out.println("'Focus' de raton");
22    }
23
24    public void mouseExited(MouseEvent EventoQueLlega){
25        System.out.println("'Blur' de raton");
26    }
27 }
```

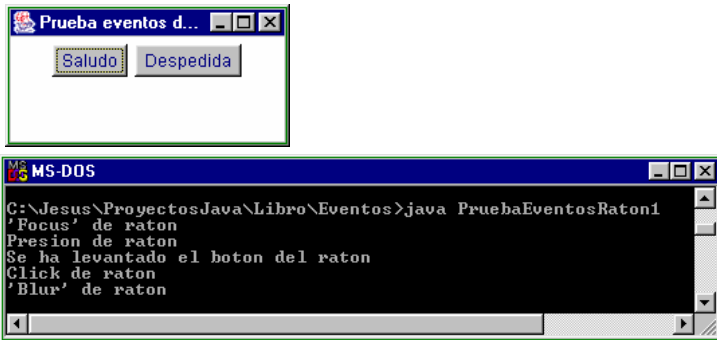
Para conseguir que la clase *InterrupcionDeRaton1* actúe sobre uno o varios componentes, debemos instanciarla y “enlazarla” con los componentes deseados. Esto lo vamos a realizar en la clase *PruebaEventosRaton1*:

```
1 import java.awt.*;
2
3 public class PruebaEventosRaton1 {
4
5     public static void main(String[] args) {
6         Frame MiFrame = new Frame("Prueba eventos de raton");
7         Panel MiPanel = new Panel();
8         Button Hola = new Button("Saludo");
9         Button Adios = new Button("Despedida");
10        MiPanel.add(Hola); MiPanel.add(Adios);
11        MiFrame.add(MiPanel);
12        MiFrame.setSize(200,100);
13        MiFrame.show();
14
15        Hola.addMouseListener (new InterrupcionDeRaton1());
16        Adios.addMouseListener(new InterrupcionDeRaton1());
17    }
18 }
```

La clase *PruebaEventosRaton1* crea un interfaz gráfico de usuario que contiene un marco, un panel y dos botones (líneas 6 a 13), posteriormente crea una instancia de la clase *InterrupcionDeRaton1* y la “enlaza” al botón *Hola* mediante el método *addMouseListener* (línea 15). En la línea 16 se hace lo mismo con el botón *Adios*; de esta manera ambos botones presentan el mismo comportamiento respecto a los eventos de ratón.

El método *addMouseListener (MouseListener l)* pertenece a la clase *Component*, por lo que podemos capturar eventos de ratón en todos los objetos subclases de *Component* (*Button*, *Label*, *List*, *Choice*, etc.).

Una posible ejecución del ejemplo anterior nos da el siguiente resultado:



La clase *IntrrupcionDeRaton1* puede ser utilizada por otras aplicaciones, en el ejemplo *PruebaEventosRaton2* se asocia su comportamiento a un panel situado sobre un marco (línea 12):

```
1  import java.awt.*;
2
3  public class PruebaEventosRaton2 {
4
5      public static void main(String[] args) {
6          Frame MiFrame = new Frame("Prueba eventos de raton");
7          Panel MiPanel = new Panel();
8          MiFrame.add(MiPanel);
9          MiFrame.setSize(200,100);
10         MiFrame.show();
11
12         MiPanel.addMouseListener (new InterrupcionDeRaton1());
13
14     }
15 }
```

Para modificar el comportamiento de algunos componentes de un interfaz gráfico de usuario nos basta con variar, en esos componentes, la clase que implementa el interfaz 'Listener'. En el ejemplo siguiente (*PruebaEventosRaton3*) se utiliza el mismo GUI que presenta la clase *PruebaEventosRaton1*, pero ahora empleamos el objeto *InterrupcionDeRaton2*.

```
1  import java.awt.*;
2
3  public class PruebaEventosRaton3 {
4
5      public static void main(String[] args) {
6          // como en PruebaEventosRaton1
7
8          Hola.addMouseListener (new InterrupcionDeRaton2());
9          Adios.addMouseListener(new InterrupcionDeRaton2());
10
11     }
12 }
```

La clase *InterrupcionDeRaton2* aumenta la funcionalidad de *InterrupcionDeRaton1*. Los métodos *mousePressed* (línea 11) y *mouseReleased* (línea 17) utilizan los métodos *getX* y *getY* de la clase *EventoQueLlega*, de tipo *MouseEvent*; de esta manera podemos conocer las coordenadas del puntero del ratón (respecto al origen del componente) cuando se pulsa y cuando se suelta el botón.

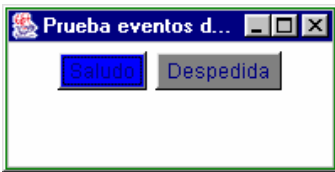
Los métodos *mouseEntered* (línea 23) y *mouseExited* (línea 29) hacen uso del método *getSource* para obtener el objeto que generó el evento. Conociendo el objeto, podemos cambiar sus propiedades (en este ejemplo su color de fondo). En este ejemplo, en una serie de dos botones, conseguimos cambiar a azul el color de fondo del botón sobre el que nos situamos con el ratón, dejándolo en gris al salir del mismo.

```
1  import java.awt.event.*;
2  import java.awt.Component;
3  import java.awt.Color;
4
5  public class InterrupcionDeRaton2 extends Object implements
                                   MouseListener {
6
7      public void mouseClicked(MouseEvent EventoQueLlega){
8          System.out.println("Click de raton");
9      }
10
11     public void mousePressed(MouseEvent EventoQueLlega){
12         System.out.println("Presion de raton");
13         System.out.println(EventoQueLlega.getX());
14     }
15
16     public void mouseReleased(MouseEvent EventoQueLlega){
17         System.out.println("Suelta de raton");
18     }
19
20     public void mouseEntered(MouseEvent EventoQueLlega){
21         System.out.println("Entrada de raton");
22     }
23
24     public void mouseExited(MouseEvent EventoQueLlega){
25         System.out.println("Salida de raton");
26     }
27 }
```

```

14     System.out.println(EventoQueLlega.getY());
15 }
16
17 public void mouseReleased(MouseEvent EventoQueLlega){
18     System.out.println("Se ha levantado el boton del
19                             raton");
20     System.out.println(EventoQueLlega.getX());
21     System.out.println(EventoQueLlega.getY());
22 }
23
24 public void mouseEntered(MouseEvent EventoQueLlega){
25     System.out.println("'Focus' de raton");
26     Component Boton = (Component)EventoQueLlega.getSource();
27     Boton.setBackground(Color.blue);
28 }
29
30 public void mouseExited(MouseEvent EventoQueLlega){
31     System.out.println("'Blur' de raton");
32     Component Boton = (Component)EventoQueLlega.getSource();
33     Boton.setBackground(Color.gray);
34 }

```

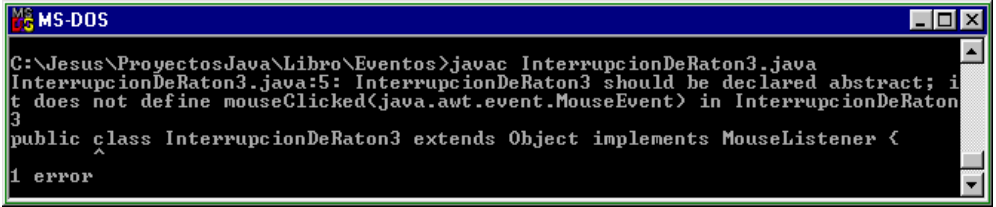


No debemos olvidar que al implementar un interfaz con una clase no abstracta debemos implementar todos y cada uno de sus métodos, de esta forma la clase *InterrupcionDeRaton3* no puede ser compilada sin errores:

```

1  import java.awt.event.*;
2  import java.awt.Component;
3  import java.awt.Color;
4
5  public class InterrupcionDeRaton3 extends Object implements
6                                     MouseListener {
7
8      // public void mouseClicked(MouseEvent EventoQueLlega){
9      //     System.out.println("Click de raton");
10     // }
11
12     // Como en la clase InterrupcionDeRaton2

```



```
MS-DOS
C:\Jesus\ProyectosJava\Libro\Eventos>javac InterrupcionDeRaton3.java
InterrupcionDeRaton3.java:5: InterrupcionDeRaton3 should be declared abstract; i
t does not define mouseClicked(java.awt.event.MouseEvent) in InterrupcionDeRaton
3
public class InterrupcionDeRaton3 extends Object implements MouseListener {
1 error
```

Java proporciona el adaptador *MouseAdapter*, que implementa todos los métodos del interfaz *MouseListener*; de esta manera, si usamos *MouseAdapter*, podremos sobrecargar únicamente los métodos que deseemos. La clase *InterrupcionDeRaton4* funciona de esta manera.

```
1 import java.awt.event.*;
2 import java.awt.Component;
3 import java.awt.Color;
4
5 public class InterrupcionDeRaton4 extends MouseAdapter {
6
7     public void mouseEntered(MouseEvent EventoQueLlega){
8         System.out.println("'Focus' de raton");
9         Component Boton = (Component)EventoQueLlega.getSource();
10        Boton.setBackground(Color.blue);
11    }
12
13    public void mouseExited(MouseEvent EventoQueLlega){
14        System.out.println("'Blur' de raton");
15        Component Boton = (Component)EventoQueLlega.getSource();
16        Boton.setBackground(Color.gray);
17    }
18 }
```

El último ejemplo de este apartado *PruebaEventosRaton5* permite crear un número arbitrario de botones, pudiendo desplazarse por los mismos provocando su cambio de color. Cuando se pulsa en cualquiera de ellos aparecerá un mensaje aclarativo; este mensaje podría sustituirse con facilidad por la ejecución de un método que tuviese asociado cada botón.

En la clase *PruebaEventosRaton5* se crea un GUI con una etiqueta *Mensaje* (líneas 7 y 20) y *NUM\_OPCIONES* botones (líneas 8 a 15, 21 y 22). Nótese el uso del método *setName* (líneas 12 y 14) para asignar un nombre a cada botón (no confundir el nombre del botón con su identificador o su texto). En las líneas 28 y 29 se asigna a todos los botones instancias de la misma clase de tratamiento de eventos de ratón: *InterrupcionDeRaton5*. Esta clase admite un constructor con un parámetro de tipo *Label*, donde pasamos el apuntador al objeto *Mensaje*.



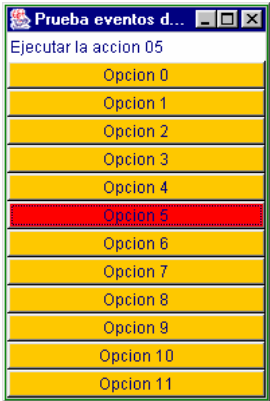
```
1  import java.awt.*;
2
3  public class PruebaEventosRaton5 {
4
5      public static void main(String[] args) {
6          final int NUM_OPCIONES = 12;
7          Label Mensaje = new Label("Mensaje asociado al botón
                                   pulsado");
8          Button[] Botones = new Button[NUM_OPCIONES];
9          for (int i=0;i<NUM_OPCIONES;i++) {
10             Botones[i] = new Button("Opcion " + i);
11             if (i<10)
12                 Botones[i].setName("0"+i);
13             else
14                 Botones[i].setName(String.valueOf(i));
15         }
16
17         Frame MiFrame = new Frame("Prueba eventos de raton");
18         Panel PanelPrincipal = new Panel(new
                                   GridLayout(NUM_OPCIONES+1,1));
19
20         PanelPrincipal.add(Mensaje);
21         for (int i=0;i<NUM_OPCIONES;i++)
22             PanelPrincipal.add(Botones[i]);
23
24         MiFrame.add(PanelPrincipal);
25         MiFrame.setSize(200,300);
26         MiFrame.show();
27
28         for (int i=0;i<NUM_OPCIONES;i++)
29             Botones[i].addMouseListener(new
                                   InterrupcionDeRaton5(Mensaje));
30
31     }
32 }
33 }
```

La clase *InterrupcionDeRaton5* extiende el adaptador *MouseAdapter* (línea 6) y contiene un constructor (línea 11) a través del cual se puede indicar una etiqueta donde los métodos de la clase podrán asignar diferentes textos.

Los métodos *mouseEntered* (línea 22) y *mouseExited* (línea 27) se encargan de mantener los colores de los botones según pasamos el puntero del ratón por los mismos. El método *mouseClicked* (línea 15) en primer lugar determina el componente (en nuestro caso un botón) que ha generado el evento (línea 16), después obtiene su nombre mediante el método *getName* (línea 17), posteriormente (línea 18) aísla los dos últimos dígitos del nombre (que en nuestro ejemplo indican

el número del botón), finalmente se modifica el texto de la etiqueta que ha recibido la clase a través de su constructor (línea 19).

```
1  import java.awt.event.*;
2  import java.awt.Component;
3  import java.awt.Label;
4  import java.awt.Color;
5
6  public class InterrupcionDeRaton5 extends MouseAdapter {
7
8      private Label Mensaje;
9      private Component ComponenteQueInvoca;
10
11     public InterrupcionDeRaton5(Label Mensaje) {
12         this.Mensaje = Mensaje;
13     }
14
15     public void mouseClicked(MouseEvent EventoQueLlega){
16         Component ComponenteQueInvoca = (Component)
17                                     EventoQueLlega.getSource();
18         String Nombre = ComponenteQueInvoca.getName();
19         String Opcion = Nombre.substring(Nombre.length()
20                                     -2,Nombre.length());
21         Mensaje.setText("Ejecutar la accion "+Opcion);
22         // invocar a un método
23     }
24
25     public void mouseEntered(MouseEvent EventoQueLlega){
26         Component ComponenteQueInvoca = (Component)
27                                     EventoQueLlega.getSource();
28         ComponenteQueInvoca.setBackground(Color.red);
29     }
30
31     public void mouseExited(MouseEvent EventoQueLlega){
32         Component ComponenteQueInvoca = (Component)
33                                     EventoQueLlega.getSource();
34         ComponenteQueInvoca.setBackground(Color.orange);
35     }
36 }
```



Una última consideración respecto a este último ejemplo se centra en la decisión de pasar la etiqueta del GUI como argumento al constructor de la clase de tratamiento de eventos. Si analizamos lo que hemos hecho, desde un punto de vista orientado a objetos no es muy adecuado, puesto que la etiqueta no debería ser visible en la clase *InterrupcionDeRaton5*. Este es un problema clásico en el diseño y desarrollo de interfaces de usuario. Habitualmente se resuelve uniendo en un solo fichero ambas clases, aunque esta solución es menos modular y reutilizable que la adoptada. Java proporciona medios de mayor complejidad para establecer una solución más elegante a esta interacción GUI/Adaptador, aunque, en general, estos medios no son muy utilizados.

### 8.2.3 Eventos de movimiento de ratón

Los eventos de movimiento de ratón nos permiten asociar acciones a cada movimiento del ratón a través de un componente, de esta manera, por ejemplo, podríamos implementar una aplicación que dibuje la trayectoria del ratón, o que vaya dibujando las rectas con origen el punto donde pulsamos el ratón y con final cada posición por donde movemos el ratón con el botón pulsado.

Tal y como vimos anteriormente, los métodos involucrados en el interfaz *MouseMotionListener* son: *mouseMoved* y *mouseDragged*. En el siguiente ejemplo *PruebaEventosMovimientoRaton1* vamos a utilizar de una manera muy sencilla estos métodos. La clase *PruebaEventosMovimientoRaton1* contiene un GUI con un panel *MiPanel* al que se le añade una instancia de la clase de tratamiento de eventos de movimiento de ratón: *InterrupciónDeMovimientoDeRaton1* (línea 12).

```

1  import java.awt.*;
2
3  public class PruebaEventosMovimientoRaton1 {
4
5      public static void main(String[] args) {
6          Frame MiFrame = new Frame("Prueba eventos de movimiento
                                   de raton");
7
8          Panel MiPanel = new Panel();
9          MiFrame.add(MiPanel);
10         MiFrame.setSize(200,100);
11         MiFrame.show();
12
13         MiPanel.addMouseListener (new
                                   InterrupcionDeMovimientoDeRaton1());
14     }
15 }

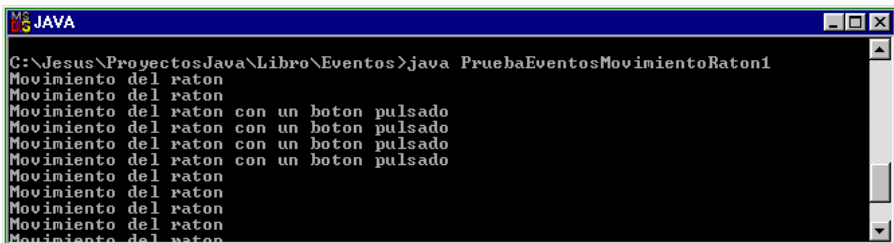
```

La clase *InterrupcionDeMovimientoDeRaton1* implementa el interfaz *MouseEvent* (línea 3), por lo que también implementa sus métodos *mouseMoved* (línea 5) y *mouseDragged* (línea 9). Ambos métodos, en este primer ejemplo, simplemente sacan por consola un mensaje representativo del evento que se ha producido (líneas 6 y 10).

```

1  import java.awt.event.*;
2
3  public class InterrupcionDeMovimientoDeRaton1 implements
                                   MouseEvent {
4
5      public void mouseMoved(MouseEvent EventoQueLlega){
6          System.out.println("Movimiento del raton");
7      }
8
9      public void mouseDragged(MouseEvent EventoQueLlega){
10         System.out.println("Movimiento del raton con un boton
                               pulsado");
11     }
12
13 }

```

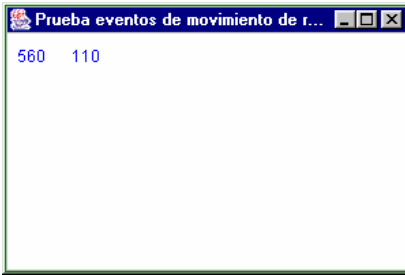


```

MC JAVA
C:\Jesus\ProyectosJava\Libro\Eventos>java PruebaEventosMovimientoRaton1
Movimiento del raton
Movimiento del raton
Movimiento del raton con un boton pulsado
Movimiento del raton con un boton pulsado
Movimiento del raton con un boton pulsado
Movimiento del raton
Movimiento del raton
Movimiento del raton
Movimiento del raton
Movimiento del raton

```

El segundo ejemplo de este apartado muestra en una ventana las posiciones x,y que va tomando el puntero del ratón según el usuario lo va moviendo por la pantalla. Las posiciones x, y son relativas a la esquina superior izquierda del componente que genera el evento, en nuestro caso el panel *MiPanel* (líneas 7, 11, 12 y 16). El color del texto en las etiquetas será de color azul si tenemos pulsado un botón del ratón y naranja si no lo tenemos.



La clase *PruebaEventosMovimientoRaton2* crea un GUI en el que existe un marco (línea 6), un panel (línea 7) y dos etiquetas (líneas 9 y 10). En la línea 16 se añade una instancia de la clase de tratamiento de eventos de movimiento de ratón: *InterrupcionDeMovimientoDeRaton2*; esta clase contiene un constructor que recoge las referencias de las etiquetas, con el fin de poder variar sus valores.

```

1  import java.awt.*;
2
3  public class PruebaEventosMovimientoRaton2 {
4
5      public static void main(String[] args) {
6          Frame MiFrame = new Frame("Prueba eventos de movimiento
                                   de raton");
7          Panel MiPanel = new Panel(new
                                   FlowLayout(FlowLayout.LEFT));
8          MiFrame.add(MiPanel);
9          Label PosicionX = new Label("000");
10         Label PosicionY = new Label("000");
11         MiPanel.add(PosicionX);
12         MiPanel.add(PosicionY);
13
14         MiFrame.setSize(300,200);
15         MiFrame.show();
16         MiPanel.addMouseMotionListener(new
            InterrupcionDeMovimientoDeRaton2(PosicionX,PosicionY));
17     }
18 }

```

La clase *InterrupcionDeMovimientoDeRaton2* (línea 5) implementa el interfaz de tratamiento de eventos de movimiento de ratón (*MouseMotionListener*). Su único constructor contiene como parámetros dos etiquetas (línea 10).

Los métodos *mouseMoved* y *mouseDragged* (líneas 22 y 28) asignan el color adecuado a las etiquetas e invocan (líneas 25 y 31) al método privado *AsignaPosicion* (línea 15). Este método obtiene la posición en pixels del puntero del ratón, usando los métodos *getX* y *getY* (líneas 16 y 18) pertenecientes a la clase *MouseEvent*. Estos valores de tipo *int* se convierten a *String* y se asignan a las etiquetas (líneas 17 y 19).

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import java.lang.String;
4
5  public class InterrupcionDeMovimientoDeRaton2 implements
6                                     MouseMotionListener {
7
8      private Label EtiquetaX, EtiquetaY;
9
10     InterrupcionDeMovimientoDeRaton2(Label EtiquetaX, Label
11                                     EtiquetaY) {
12         this.EtiquetaX = EtiquetaX;
13         this.EtiquetaY = EtiquetaY;
14     }
15     private void AsignaPosicion(MouseEvent EventoQueLlega) {
16         String PosicionX =
17             String.valueOf(EventoQueLlega.getX());
18         EtiquetaX.setText(PosicionX);
19         String PosicionY =
20             String.valueOf(EventoQueLlega.getY());
21         EtiquetaY.setText(PosicionY);
22     }
23     public void mouseMoved(MouseEvent EventoQueLlega){
24         EtiquetaX.setForeground(Color.orange);
25         EtiquetaY.setForeground(Color.orange);
26         AsignaPosicion(EventoQueLlega);
27     }
28     public void mouseDragged(MouseEvent EventoQueLlega){
29         EtiquetaX.setForeground(Color.blue);
30         EtiquetaY.setForeground(Color.blue);
31         AsignaPosicion(EventoQueLlega);
32     }
33 }
```

Si deseamos evitar el uso del constructor con parámetros podemos unificar el GUI y el adaptador en un solo fichero en el que las etiquetas son propiedades globales al adaptador:

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import java.lang.String;
4
5  public class PruebaEventosMovimientoRaton2b {
6      Frame MiFrame = new Frame("Prueba eventos de movimiento
                                de raton");
7      Panel MiPanel = new Panel(new
                                FlowLayout(FlowLayout.LEFT));
8      Label PosicionX = new Label("000");
9      Label PosicionY = new Label("000");
10
11     PruebaEventosMovimientoRaton2b() {
12         MiFrame.add(MiPanel);
13         MiPanel.add(PosicionX);
14         MiPanel.add(PosicionY);
15         MiFrame.setSize(300,200);
16         MiFrame.show();
17         MiPanel.addMouseListener(new
18             InterrupcionDeMovimientoDeRaton2b());
19     } // Constructor
20
21
22     class InterrupcionDeMovimientoDeRaton2b implements
23         MouseMotionListener {
24         private void AsignaPosicion(MouseEvent EventoQueLlega) {
25             String X = String.valueOf(EventoQueLlega.getX());
26             PosicionX .setText(X);
27             String Y = String.valueOf(EventoQueLlega.getY());
28             PosicionY .setText(Y);
29         }
30
31         public void mouseMoved(MouseEvent EventoQueLlega){
32             PosicionX .setForeground(Color.orange);
33             PosicionY .setForeground(Color.orange);
34             AsignaPosicion(EventoQueLlega);
35         }
36
37         public void mouseDragged(MouseEvent EventoQueLlega){
38             PosicionX .setForeground(Color.blue);
39             PosicionY .setForeground(Color.blue);
40             AsignaPosicion(EventoQueLlega);
```

```
41     }
42
43 } // InterrupcionDeMovimientoDeRaton2b
44
45 } // PruebaEventosMovimientoRaton2b
```

Para ejecutar la aplicación instanciamos la clase *PruebaEventosMovimientoRaton2b*:

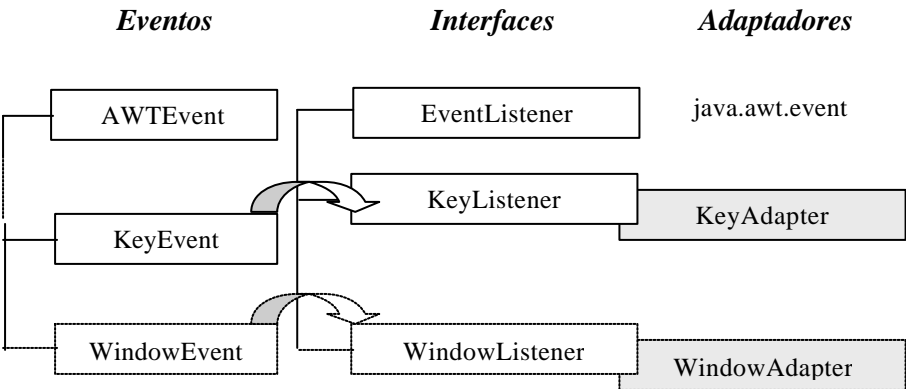
```
1 public class Prueba {
2     public static void main(String[] args) {
3         PruebaEventosMovimientoRaton2b Instancia =
4             new PruebaEventosMovimientoRaton2b();
5     }
6 }
```

### 8.3 EVENTOS DE TECLADO Y DE VENTANA

#### 8.3.1 Introducción

Los eventos de teclado se capturan haciendo uso de los siguientes interfaces (o adaptadores) y eventos:

Interface	Adaptador	Evento
KeyListener	KeyAdapter	KeyEvent
WindowListener	WindowAdapter	WindowEvent





A continuación se presenta un resumen de los objetos más importantes en el proceso de tratamiento de eventos de teclado y de ventana:

KeyListener	
Método	El método se invoca cuando...
keyPressed (KeyEvent e)	Se ha presionado una tecla
keyReleased (KeyEvent e)	Se ha terminado la presión de una tecla
keyTyped (KeyEvent e)	Se ha presionado (en algunas ocasiones presionado y soltado) una tecla

WindowListener	
Método	El método se invoca cuando...
windowActivated (WindowEvent e)	La ventana pasa a ser la activa
windowDeactivated (WindowEvent e)	La ventana deja de ser la activa
windowOpened (WindowEvent e)	La primera vez que la ventana se hace visible
windowClosing (WindowEvent e)	El usuario indica que se cierre la ventana
windowClosed (WindowEvent e)	La ventana se ha cerrado
windowIconified (WindowEvent e)	La ventana pasa de estado normal a un estado minimizado
windowDeiconified (WindowEvent e)	La ventana pasa de estado minimizado a un estado normal

KeyEvent	
Métodos más utilizados	Explicación
char getKeyChar()	Devuelve el carácter asociado con la tecla pulsada
int getKeyCode()	Devuelve el valor entero que representa la tecla pulsada
String getKeyText()	Devuelve un texto que representa el código de la tecla
Object getSource()	Método perteneciente a la clase <i>EventObject</i> . Indica el objeto que produjo el evento

WindowEvent	
Métodos más utilizados	Explicación
Window getWindow()	Devuelve la ventana que origina el evento
Window getOppositeWindow()	Devuelve la ventana involucrada en el cambio de activación o de foco
int getNewState()	Para WINDOW_STATE_CHANGED indica el nuevo estado de la ventana
int getOldState()	Para WINDOW_STATE_CHANGED indica el antiguo estado de la ventana
Object getSource()	Método perteneciente a la clase <i>EventObject</i> . Indica el objeto que produjo el evento

### 8.3.2 Eventos de teclado

En este apartado vamos a desarrollar dos ejemplos que muestran el uso del interfaz *KeyListener*, la clase *KeyAdapter* y el evento *KeyEvent*.

En el primer ejemplo utilizamos una clase de tratamiento de eventos de teclado (*InterrupcionDeTeclado1*) que escribe un texto por consola cada vez que se activa uno de los tres métodos que proporciona el interfaz *KeyListener*. Los métodos *keyTyped*, *keyPressed* y *keyReleased* se implementan en las líneas 5, 9 y 13.

```
1  import java.awt.event.*;
2
3  public class InterrupcionDeTeclado1 implements KeyListener
4  {
5      public void keyTyped(KeyEvent EventoQueLlega){
6          System.out.println("Tecla pulsada y soltada");
7      }
8
9      public void keyPressed(KeyEvent EventoQueLlega){
10         System.out.println("Tecla pulsada");
11     }
12
13     public void keyReleased(KeyEvent EventoQueLlega){
14         System.out.println("Tecla soltada");
15     }
16
17 }
```

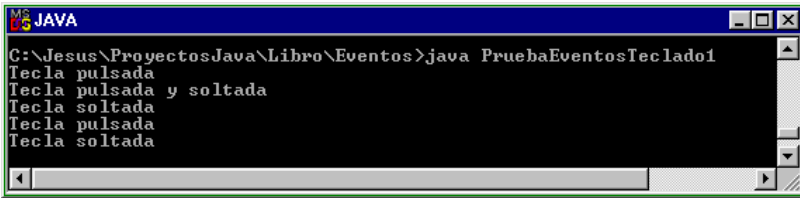
Para probar el funcionamiento de la clase anterior creamos *PruebaEventosTeclado1* (línea 3), que incorpora un GUI en el que al panel *MiPanel* le añadimos una instancia del “listener” *InterrupcionDeTeclado1* (línea 13).

```
1  import java.awt.*;
2
3  public class PruebaEventosTeclado1 {
4
5      public static void main(String[] args) {
6
7          Frame MiFrame = new Frame("Prueba eventos de raton");
8          Panel MiPanel = new Panel();
9          MiFrame.add(MiPanel);
10         MiFrame.setSize(200,300);
11         MiFrame.show();
12
13         MiPanel.addKeyListener(new InterrupcionDeTeclado1());
```

```

14
15  }
16  }

```



Según el tipo de tecla pulsada (letras, teclas de función, etc.) la activación del evento *keyTyped* puede variar.

Nuestro segundo ejemplo imprime por consola los caracteres tecleados hasta que pulsamos el asterisco, momento en el que abandonamos de la aplicación. Para conseguir esta funcionalidad nos basta con hacer uso de un solo método del interfaz *KeyListener*. Para facilitarnos la labor empleamos el adaptador *KeyAdapter* (línea 3 de la clase *InterrupcionDeTeclado2*), sobrecargando únicamente el método *keyTyped* (cualquiera de los otros dos métodos nos hubiera servido de igual manera).

En el método *keyTyped* (línea 5) imprimimos (línea 6) el carácter correspondiente a la tecla pulsada; este carácter lo obtenemos invocando al método *getKeyChar* perteneciente a la clase *KeyEvent*. Posteriormente consultamos si su valor es un asterisco (línea 7), en cuyo caso abandonamos la aplicación (línea 8).

```

1  import java.awt.event.*;
2
3  public class InterrupcionDeTeclado2 extends KeyAdapter {
4
5      public void keyTyped(KeyEvent e){
6          System.out.print(e.getKeyChar());
7          if (e.getKeyChar()=='*')
8              System.exit(0);
9      }
10
11 }

```

Tal y como hemos venido haciendo en ejercicios anteriores, implementamos una clase que incorpora un GUI con algún elemento al que se le añade la rutina de tratamiento de interrupción. En este caso la clase es *PruebaEventosTeclado2* (línea 3) y el método *addKeyListener* se utiliza en la línea 17.

```

1  import java.awt.*;
2

```

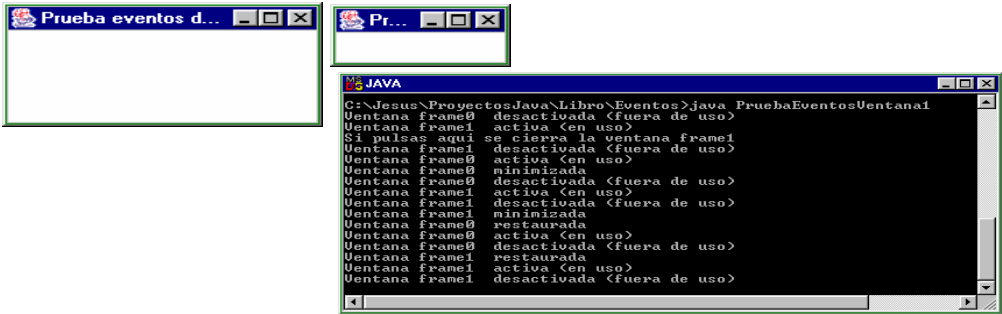


```
9     public void windowClosing(WindowEvent e) {
10         System.out.println("Si pulsas aqui se cierra la
                               ventana " + e.getWindow().getName() );
11         // System.exit(0);
12     }
13
14     public void windowClosed(WindowEvent e) {
15         System.out.println("Ventana " + e.getWindow().getName()
                               + " cerrada");
16     }
17
18     public void windowActivated(WindowEvent e) {
19         System.out.println("Ventana " + e.getWindow().getName()
                               + " activa (en uso)");
20     }
21
22     public void windowDeactivated(WindowEvent e) {
23         System.out.println("Ventana " + e.getWindow().getName()
                               + " desactivada (fuera de uso)");
24     }
25
26     public void windowIconified(WindowEvent e) {
27         System.out.println("Ventana " + e.getWindow().getName()
                               + " minimizada");
28     }
29
30     public void windowDeiconified(WindowEvent e) {
31         System.out.println("Ventana " + e.getWindow().getName()
                               + " restaurada");
32     }
33
34 }
```

El programa de prueba *PruebaEventosVentana1* crea dos marcos (líneas 7 y 11) y les añade instancias de la clase de tratamiento de eventos de ventana *InterrupcionDeVentana1*, haciendo uso del método *addWindowListener* (líneas 16 y 17).

```
1  import java.awt.*;
2
3  public class PruebaEventosVentana1 {
4
5      public static void main(String[] args) {
6
7          Frame MiFrame = new Frame("Prueba eventos de ventana");
8          MiFrame.setSize(200,100);
9          MiFrame.show();
10 }
```

```
11      Frame OtroFrame = new Frame("Prueba eventos de
                                   ventana");
12      OtroFrame.setSize(100,50);
13      OtroFrame.setLocation(200,0);
14      OtroFrame.show();
15
16      MiFrame.addWindowListener(new InterrupcionDeVentana1());
17      OtroFrame.addWindowListener(new InterrupcionDeVentana1());
18
19  }
20 }
```



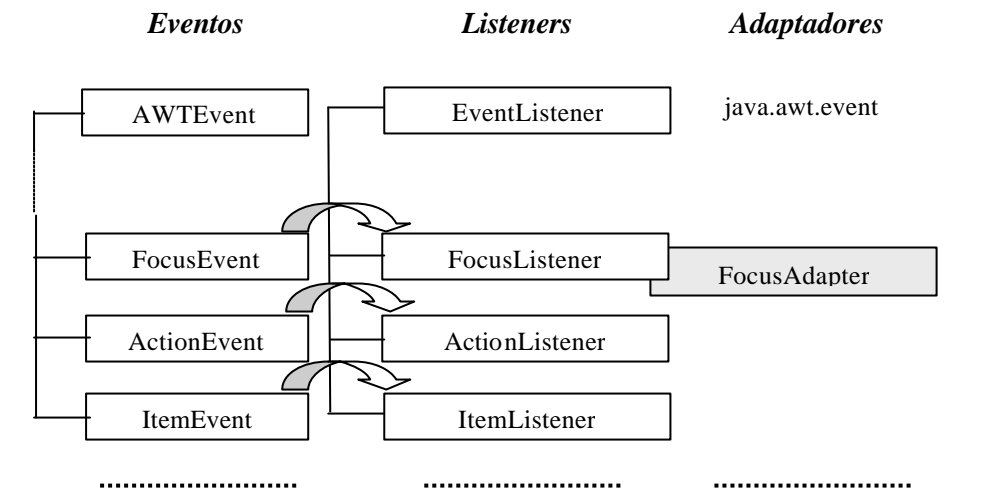
## 8.4 EVENTOS DE ACCIÓN, ENFOQUE Y ELEMENTO

### 8.4.1 Introducción

Los eventos de acción, enfoque y elemento se capturan haciendo uso de los siguientes interfaces (o adaptadores) y eventos:

Interface	Adaptador	Evento
ActionListener		ActionEvent
FocusListener	FocusAdapter	FocusEvent
ItemListener		ItemEvent

Estos eventos tienen un nivel de abstracción superior a los de ratón y teclado; en estos tres casos el evento se produce cuando ocurre una acción sobre un componente, independientemente de la causa física que produce ese evento. Por ejemplo, podemos pulsar un botón haciendo uso del ratón, del teclado, e incluso por programa.



Empleando los “listeners” vistos hasta ahora, si quisiéramos, por ejemplo, ejecutar un método asociado a la pulsación de un botón, tendríamos que añadir a ese botón dos instancias de tratamiento de eventos: una de tipo *MouseListener* para capturar pulsaciones de ratón sobre el botón y otra de tipo *KeyListener* para capturar pulsaciones de teclado. En este caso sería mucho más cómodo utilizar el interfaz *ActionListener* que proporciona un método que se activa con independencia de la causa física o lógica que “activa” el componente.

A continuación se presenta un resumen de los objetos más importantes en el proceso de tratamiento de eventos de acción, enfoque y elemento:

ActionListener	
Método	El método se invoca cuando...
actionPerformed (ActionEvent e)	Ocurre una acción sobre el elemento

FocusListener	
Método	El método se invoca cuando...
focusGained (FocusEvent e)	Nos posicionamos sobre el componente con el teclado
focusLost (FocusEvent e)	Salimos del elemento haciendo uso del teclado

ItemListener	
Método	El método se invoca cuando...
itemStateChanged (ItemEvent e)	Un elemento ha sido seleccionado o deseleccionado por el usuario

ActionEvent	
Métodos más utilizados	Explicación
String getActionCommand()	Identifica/explica la acción
long getWhen()	Momento en el que se produce el evento
Object getSource()	Método perteneciente a la clase <i>EventObject</i> . Indica el objeto que produjo el evento

FocusEvent	
Métodos más utilizados	Explicación
String getOppositeComponent()	Indica el componente involucrado en el cambio de foco. Si el tipo de evento es FOCUS_GAINED se refiere al componente que ha perdido el foco, si es FOCUS_LOST al que ha ganado el foco.
int getID()	Método perteneciente a la clase <i>AWTEvent</i> . Indica el tipo de evento
Object getSource()	Método perteneciente a la clase <i>EventObject</i> . Indica el objeto que produjo el evento

ItemEvent	
Métodos más utilizados	Explicación
int getStateChange()	Tipo de cambio producido (SELECTED, DESELECTED)
Object getItem()	Objeto afectado por el evento (por ejemplo una lista)
ItemSelectable getItemSelectable()	Elemento que ha originado el evento (por ejemplo un elemento de una lista)
Object getSource()	Método perteneciente a la clase <i>EventObject</i> . Indica el objeto que produjo el evento

8.4.2 Eventos de acción

En este apartado vamos a desarrollar dos ejemplos que muestran el uso del interfaz *ActionListener* y el evento *ActionEvent*.

En el primer ejemplo utilizamos una clase de tratamiento de eventos de acción (*InterrupcionDeAccionI*) que escribe por consola información del evento cada vez que se activa *actionPerformed* (el único método que proporciona el interfaz *ActionListener*).



En la línea 6 se implementa el método *actionPerformed*, que admite como parámetro un objeto de tipo *ActionEvent*. En primer lugar (línea 7) obtenemos el componente que generó el evento, para ello hacemos uso del método *getSource*. En la línea 8 obtenemos un literal que identifica el comando (en este caso el texto asociado al objeto); utilizamos el método *getActionCommand* sobre la instancia del evento que nos ha llegado. En las líneas 10 y 11 se imprime la información obtenida y el nombre del componente (como no hemos asignado explícitamente nombre a los componentes, conseguimos los que JVM asigna implícitamente).

```
1  import java.awt.event.*;
2  import java.awt.*;
3
4  public class InterrupcionDeAccion1 implements
                                   ActionListener {
5
6      public void actionPerformed(ActionEvent Evento){
7          Component Componente = (Component) Evento.getSource();
8          String AccionRealizada = Evento.getActionCommand();
9
10         System.out.println("Componente:  " +
                               Componente.getName());
11         System.out.println("Suceso:  " + AccionRealizada);
12         System.out.println();
13     }
14 }
```

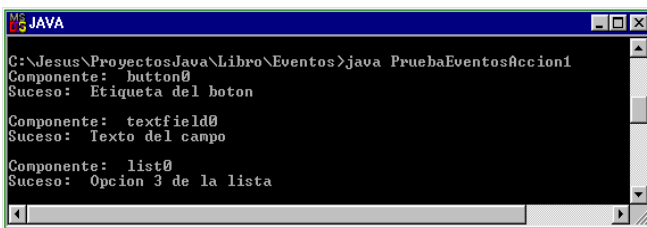
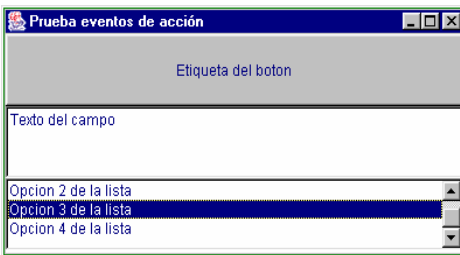
Para probar nuestra implementación del interfaz *ActionListener*, creamos una clase *PruebaEventosAccion1* (línea 3) que incorpora un GUI con un botón, un campo de texto y una lista, a estos tres elementos se les asocian sendas instancias de la clase *InterrupcionDeAccion1* (líneas 20 a 22); el resultado de una posible ejecución se muestra tras el código de la clase.

```
1  import java.awt.*;
2
3  public class PruebaEventosAccion1 {
4
5      public static void main(String[] args) {
6
7          Frame MiFrame = new Frame("Prueba eventos de acción");
8          Panel MiPanel = new Panel(new GridLayout(3,1));
9
10         Button Boton = new Button("Etiqueta del boton");
11         TextField CampoDeTexto = new TextField("Texto del
                                                    campo",8);
12
13         List Lista = new List(3);
14         Lista.add("Opcion 1 de la lista");
15         Lista.add("Opcion 2 de la lista");
16     }
17 }
```

```

14     Lista.add("Opcion 3 de la lista");
15     Lista.add("Opcion 4 de la lista");
16     MiPanel.add(Boton);
17     MiPanel.add(CampoDeTexto);
18     MiPanel.add(Lista);
19
20     Boton.addActionListener(new InterrupcionDeAccion1());
21     CampoDeTexto.addActionListener(new
22                                     InterrupcionDeAccion1());
23     Lista.addActionListener(new InterrupcionDeAccion1());
24
25     MiFrame.add(MiPanel);
26     MiFrame.setSize(400,200);
27     MiFrame.show();
28 }
29 }

```



En el segundo ejemplo de eventos de acción se crea un GUI con tres botones: “rojo”, “verde” y “azul”; al pulsar en cada uno de ellos el color del fondo varía al color seleccionado. La clase *PruebaEventosAccion2* proporciona el interfaz gráfico de usuario con los tres botones (líneas 10 a 12) y les añade instancias de un objeto de tratamiento de eventos de acción: *InterrupcionDeAccion2* (líneas 18 a 20); a través del constructor de esta clase se pasa la referencia del panel que registrará los cambios de color a medida que se produzcan los eventos.

```

1  import java.awt.*;
2
3  public class PruebaEventosAccion2 {

```

```

4
5 public static void main(String[] args) {
6
7     Frame MiFrame = new Frame("Prueba eventos de acción");
8     Panel MiPanel = new Panel();
9
10    Button Rojo  = new Button("Rojo");
11    Button Verde  = new Button("Verde");
12    Button Azul   = new Button("Azul");
13
14    MiPanel.add(Rojo);
15    MiPanel.add(Verde);
16    MiPanel.add(Azul);
17
18    Rojo.addActionListener(new
                                InterrupcionDeAccion2(MiPanel));
19    Verde.addActionListener(new
                                InterrupcionDeAccion2(MiPanel));
20    Azul.addActionListener(new
                                InterrupcionDeAccion2(MiPanel));
21
22    MiFrame.add(MiPanel);
23    MiFrame.setSize(400,200);
24    MiFrame.show();
25
26 }
27 }

```

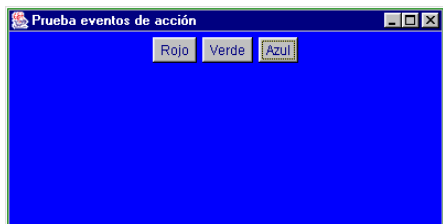
La clase *InterrupcionDeAccion2* (línea 4) implementa el interfaz *ActionListener* e incorpora un constructor (línea 8) que permite recoger la referencia de un panel. El único método del interfaz se define en la línea 12, su implementación consiste en identificar la fuente del evento (línea 16) y obtener su etiqueta (línea 17) con la ayuda de los métodos *getSource* y *getLabel*. Una vez conocida la etiqueta resulta sencillo modificar adecuadamente el color del panel (líneas 19 a 27).

```

1 import java.awt.event.*;
2 import java.awt.*;
3
4 public class InterrupcionDeAccion2 implements
                                ActionListener {
5
6     private Panel PanelPrincipal;
7
8     InterrupcionDeAccion2 (Panel PanelPrincipal) {
9         this.PanelPrincipal = PanelPrincipal;
10    }
11
12    public void actionPerformed(ActionEvent Evento){

```

```
13
14     Color ColorFondo;
15
16     Button Componente = (Button) Evento.getSource();
17     String ColorSeleccionado = Componente.getLabel();
18
19     if (ColorSeleccionado=="Rojo")
20         ColorFondo = Color.red;
21     else
22         if (ColorSeleccionado=="Verde")
23             ColorFondo = Color.green;
24         else
25             ColorFondo = Color.blue;
26
27     PanelPrincipal.setBackground(ColorFondo);
28 }
29
30
31 }
```



### 8.4.3 Eventos de enfoque

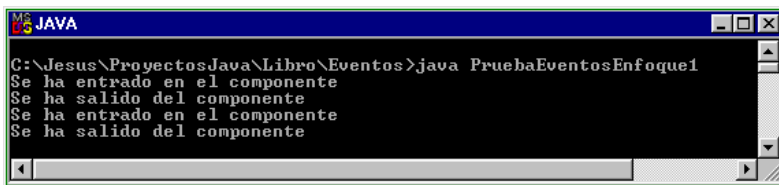
En este apartado vamos a desarrollar dos ejemplos que muestran el uso del interfaz *FocusListener*, el adaptador *FocusAdapter* y el evento *FocusEvent*.

En el primer ejemplo utilizamos una clase de tratamiento de eventos de enfoque (*InterrupcionDeEnfoque1*) que escribe un texto identificativo por consola cada vez que se activa alguno de los métodos disponibles en el interfaz *FocusListener*: *focusGained* y *focusLost*. Además hacemos uso de la clase *PruebaEventosEnfoque1* que define dos botones (líneas 9 y 10) y añade a uno de ellos el tratamiento de eventos de enfoque desarrollado (línea 17).

```
1 import java.awt.*;
2
```

```
3 public class PruebaEventosEnfoque1 {
4
5     public static void main(String[] args) {
6
7         Frame MiFrame = new Frame("Prueba eventos de enfoque");
8         Panel MiPanel = new Panel();
9         Button Boton1 = new Button("Componente 1");
10        Button Boton2 = new Button("Componente 2");
11        MiPanel.add(Boton1);
12        MiPanel.add(Boton2);
13        MiFrame.add(MiPanel);
14        MiFrame.setSize(300,200);
15        MiFrame.show();
16
17        Boton2.addFocusListener(new InterrupcionDeEnfoque1());
18    }
19 }
20 }

1 import java.awt.event.*;
2
3 public class InterrupcionDeEnfoque1 implements
                                FocusListener {
4
5     public void focusGained(FocusEvent Evento){
6         System.out.println("Se ha entrado en el componente");
7     }
8
9     public void focusLost(FocusEvent Evento){
10        System.out.println("Se ha salido del componente");
11    }
12
13 }
```



El segundo ejemplo de este apartado muestra la manera de obtener información acerca del objeto origen cuando nos llega un evento que indica que se ha “ganado” enfoque. De igual manera podríamos haber realizado la implementación con el evento de pérdida de enfoque.

La clase *PruebaEventosEnfoque2* define un interfaz gráfico de usuario con un conjunto de botones y de campos de texto; a todos estos elementos se les añade instancias del adaptador *InterrupcionDeEnfoque2*, de tipo *FocusListener* (líneas 19 y 20).

```
1  import java.awt.*;
2
3  public class PruebaEventosEnfoque2 {
4
5      public static void main(String[] args) {
6
7          final int NUM_FILAS = 5;
8          Button[] Boton = new Button[NUM_FILAS];
9          TextField[] Campo = new TextField[NUM_FILAS];
10
11         Frame MiFrame = new Frame("Prueba eventos de enfoque");
12         Panel MiPanel = new Panel(new GridLayout(NUM_FILAS,2));
13
14         for (int i=0;i<NUM_FILAS;i++) {
15             Boton[i] = new Button("Boton "+i);
16             Campo[i] = new TextField(15);
17             MiPanel.add(Boton[i]);
18             MiPanel.add(Campo[i]);
19             Boton[i].addFocusListener(new InterrupcionDeEnfoque2());
20             Campo[i].addFocusListener(new InterrupcionDeEnfoque2());
21         }
22
23         MiFrame.add(MiPanel);
24         MiFrame.setSize(400,200);
25         MiFrame.show();
26
27     }
28 }
```

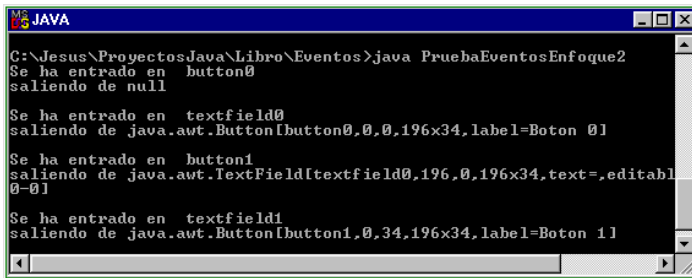
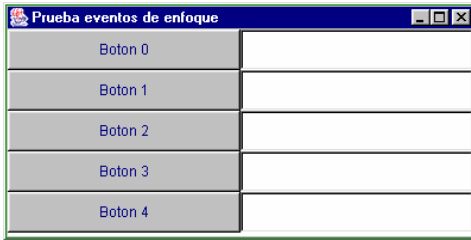
La clase *InterrupcionDeEnfoque2* extiende el adaptador *FocusAdapter* (línea 4) con el fin de sobrecargar únicamente uno de sus métodos: *focusGained* (línea 6). En la línea 7 se obtiene el elemento que ha generado el evento (usando el método *getSource*); en la línea 8 se imprime su nombre y una descripción del elemento que ha perdido el foco (usando el método *getOppositeComponent*).

```
1  import java.awt.event.*;
2  import java.awt.*;
3
4  public class InterrupcionDeEnfoque2 extends FocusAdapter {
5
6      public void focusGained(FocusEvent Evento){
7          Component Componente = (Component) Evento.getSource();
```

```

8      System.out.println("Se ha entrado en " +
                          Componente.getName() + "\nsaliendo de " +
                          Evento.getOppositeComponent() + "\n");
9  }
10
11 }

```



### 8.4.4 Eventos de elemento

En este apartado vamos a desarrollar dos ejemplos que muestran el uso del interfaz *ItemListener* y el evento *ItemEvent*. En el primer ejemplo utilizamos la clase *PruebaEventosElemento1* que crea cuatro cajas de texto (líneas 10 a 13) y les añade la funcionalidad implementada en la clase *InterrupcionDeElemento1* (líneas 20 a 23).

```

1  import java.awt.*;
2
3  public class PruebaEventosElemento1 {
4
5      public static void main(String[] args) {
6
7          Frame MiFrame = new Frame("Prueba eventos de
                                elemento");

```

```
8      Panel MiPanel = new Panel(new GridLayout(4,1));
9
10     Checkbox Diesel = new Checkbox("Diesel", true);
11     Checkbox FarosXenon = new Checkbox("Faros de Xenon",
12                                         false);
13     Checkbox LlantasAleacion = new Checkbox("Llantas de
14                                             aleacion", false);
15     Checkbox PinturaMetalizada = new Checkbox("Pintura
16                                             Metalizada", true);
17
18     MiPanel.add(Diesel);
19     MiPanel.add(FarosXenon);
20     MiPanel.add(LlantasAleacion);
21     MiPanel.add(PinturaMetalizada);
22
23     Diesel.addItemListener(new InterrupcionDeElemento1());
24     FarosXenon.addItemListener(new
25                                 InterrupcionDeElemento1());
26     LlantasAleacion.addItemListener(new
27                                     InterrupcionDeElemento1());
28     PinturaMetalizada.addItemListener(new
29                                     InterrupcionDeElemento1());
30
31     MiFrame.add(MiPanel);
32     MiFrame.setSize(400,200);
33     MiFrame.show();
34
35 }
```

La clase *InterrupcionDeElemento1* implementa el único método del interfaz *ItemListener*: *itemStateChanged* (línea 6). En primer lugar obtenemos la caja de texto que ha producido el evento (línea 8), para ello utilizamos el método *getSource* sobre el evento que nos llega. Podemos consultar el estado de la caja de texto utilizando el método *getState* perteneciente a la clase *Checkbox* (línea 9) o bien usando *getStateChange* perteneciente a la clase *ItemEvent* (línea 10). El elemento que ha generado el evento se puede obtener, además de mediante *getSource*, utilizando *getItem* (línea 12).

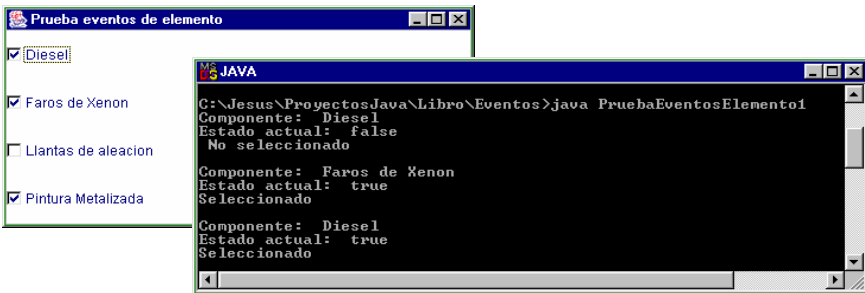
```
1  import java.awt.event.*;
2  import java.awt.*;
3
4  public class InterrupcionDeElemento1 implements
5                                     ItemListener {
6      public void itemStateChanged(ItemEvent Evento){
7
```



```

8      Checkbox Componente = (Checkbox) Evento.getSource();
9      boolean Estado = Componente.getState();
10     int NuevoEstado = Evento.getStateChange();
11
12     System.out.println("Componente: " + Evento.getItem());
13     System.out.println("Estado actual: " + Estado);
14     if (NuevoEstado==ItemEvent.SELECTED)
15         System.out.println("Seleccionado");
16     else
17         System.out.println(" No seleccionado");
18     System.out.println();
19 }
20
21 }

```



El segundo ejemplo de este apartado muestra la manera de programar opciones y validaciones interactivas en interfaces gráficas de usuario. La clase *PruebaEventosElemento2* crea un GUI con 5 cajas de verificación (líneas 13 a 18), una lista (línea 20 a 24), una lista desplegable (líneas 26 a 29) y una etiqueta (línea 31). Todos estos elementos se asignan a un vector de componentes (líneas 34 a 41) y se les añaden instancias de la clase de tratamiento de eventos de elemento *InterrupcionDeElemento2* (líneas 54 a 60).

```

1  import java.awt.*;
2
3  public class PruebaEventosElemento2 {
4
5      public static void main(String[] args) {
6
7          Frame MiFrame = new Frame("Prueba eventos de
                                elemento");
8          Panel PanelPrincipal = new Panel(new
                                FlowLayout(FlowLayout.LEFT));
9          Panel PanelIzq = new Panel(new GridLayout(6,1));
10
11          Component[] Componentes = new Component[8];

```

```
12
13     Checkbox Diesel = new Checkbox("Diesel", true);
14     Checkbox AsientosDeportivos = new Checkbox("Asientos
15         deportivos", false);
16     Checkbox TapiceriaCuero = new Checkbox("Tapiceria de
17         cuero", false);
18     Checkbox LlantasAleacion = new Checkbox("Llantas de
19         aleación", false);
20     Checkbox PinturaMetalizada = new Checkbox("Pintura
21         metalizada", true);
22
23     Choice Llantas = new Choice();
24     Llantas.add("Tres radios");
25     Llantas.add("Cinco radios");
26     Llantas.add("Siete radios");
27     Llantas.setEnabled(false);
28
29     List ColoresMetalizados = new List(2);
30     ColoresMetalizados.add("Rojo");
31     ColoresMetalizados.add("Azul");
32     ColoresMetalizados.add("Verde");
33
34     Label Seleccionado = new Label("Seleccionado:");
35     Seleccionado.setForeground(Color.red);
36
37     Componentes[0] = Diesel;
38     Componentes[1] = AsientosDeportivos;
39     Componentes[2] = TapiceriaCuero;
40     Componentes[3] = LlantasAleacion;
41     Componentes[4] = PinturaMetalizada;
42     Componentes[5] = Llantas;
43     Componentes[6] = ColoresMetalizados;
44     Componentes[7] = Seleccionado;
45
46     PanelPrincipal.add(PanelIzq);
47     PanelPrincipal.add(ColoresMetalizados);
48     PanelPrincipal.add(Llantas);
49
50     PanelIzq.add(Diesel);
51     PanelIzq.add(AsientosDeportivos);
52     PanelIzq.add(TapiceriaCuero);
53     PanelIzq.add(LlantasAleacion);
54     PanelIzq.add(PinturaMetalizada);
55     PanelIzq.add(Seleccionado);
56
57     Diesel.addItemListener(new
58         InterrupcionDeElemento2(Componentes));
59     AsientosDeportivos.addItemListener(new
```

```

        InterrupcionDeElemento2(Componentes));
56    TapiceriaCuero.addItemListener(new
        InterrupcionDeElemento2(Componentes));
57    LlantasAleacion.addItemListener(new
        InterrupcionDeElemento2(Componentes));
58    PinturaMetalizada.addItemListener(new
        InterrupcionDeElemento2(Componentes));
59    Llantas.addItemListener(new
        InterrupcionDeElemento2(Componentes));
60    ColoresMetalizados.addItemListener(new
        InterrupcionDeElemento2(Componentes));
61
62    MiFrame.add(PanelPrincipal);
63    MiFrame.setSize(500,200);
64    MiFrame.show();
65
66 }

```

La clase *InterrupcionDeElemento2* implementa el interfaz *ItemListener* (línea 4) y admite un constructor (línea 8) que recoge el vector de componentes con el fin de actuar sobre los mismos. El único método del interfaz (*itemStateChanged*) se define en la línea 12; su funcionalidad es la siguiente:

Si el elemento que ha generado el evento está definido como “Llantas de aleación”, la lista *Llantas* (*Componentes[5]*) se habilita en caso de que la caja de verificación esté seleccionada (*SELECTED*) o se deshabilita en caso de que la caja de verificación no esté seleccionada (líneas 16 a 20). La misma lógica se aplica con la caja de verificación “Pintura metalizada”, a través de la cual se controla si la lista desplegable *ColoresMetalizados* (*Componentes[6]*) se hace visible o no (líneas 22 a 26).

La activación de la caja de verificación “Asientos Deportivos” (línea 28) controla el estado y disponibilidad de la caja de verificación “Tapicería de cuero” (*Componentes[2]*), de esta manera, si la primera se selecciona (línea 30), la segunda se selecciona automáticamente (línea 31) y se limita su disponibilidad (línea 32).

La etiqueta *Seleccionado* muestra en cada instante un texto relativo al elemento que ha sido seleccionado más recientemente (líneas 38 y 39).

```

1  import java.awt.event.*;
2  import java.awt.*;
3
4  public class InterrupcionDeElemento2 implements
                                     ItemListener {
5
6      Component[] Componentes = new Component[8];

```

```

7
8  InterrupcionDeElemento2(Component[] Componentes) {
9      this.Componentes = Componentes;
10 }
11
12 public void itemStateChanged(ItemEvent Evento){
13
14     int Estado = Evento.getStateChange();
15
16     if (Evento.getItem().equals("Llantas de aleación"))
17         if (Estado==ItemEvent.SELECTED)
18             Componentes[5].setEnabled(true);
19         else
20             Componentes[5].setEnabled(false);
21
22     if (Evento.getItem().equals("Pintura metalizada"))
23         if (Estado==ItemEvent.SELECTED)
24             Componentes[6].setVisible(true);
25         else
26             Componentes[6].setVisible(false);
27
28     if (Evento.getItem().equals("Asientos deportivos")) {
29         Checkbox Asientos = (Checkbox) Componentes[2];
30         if (Estado==ItemEvent.SELECTED) {
31             Asientos.setState(true);
32             Componentes[2].setEnabled(false);
33         }
34         else
35             Componentes[2].setEnabled(true);
36     }
37
38     Label Seleccionado = (Label) Componentes[7];
39     Seleccionado.setText(" "+Evento.getItem());
40 }
41
42 }

```



## APLICACIONES DE EJEMPLO

---

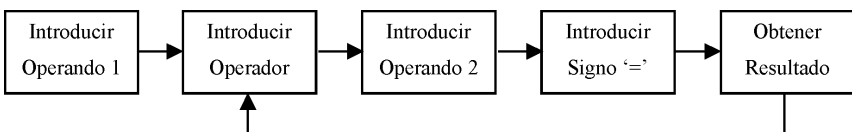
### 9.1 EJEMPLO: CALCULADORA

#### 9.1.1 Definición del ejemplo

Vamos a implementar una aplicación que ofrezca el interfaz y el comportamiento de una calculadora sencilla. Nuestra calculadora contendrá los diez dígitos (del cero al nueve), el punto decimal, el signo de igual y los operadores de suma, resta, multiplicación y división; además se proporcionará un espacio para visualizar las pulsaciones del usuario y los resultados obtenidos. Nuestro interfaz gráfico de usuario debe, por tanto, mostrar un aspecto similar al siguiente:

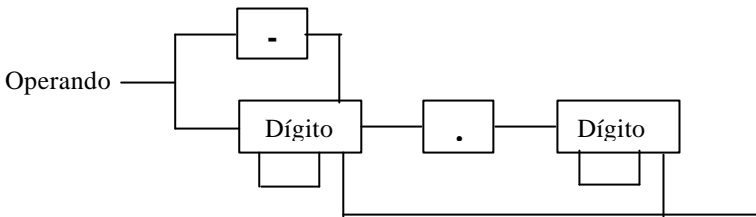


El comportamiento de la calculadora también será sencillo. Su esquema básico de funcionamiento lo programaremos con el siguiente patrón:



Por ejemplo, podemos ir pulsando la secuencia:  $8*4=$ , obteniendo el resultado 32; posteriormente ir pulsando la secuencia  $-2=$ , obteniendo 30 y así sucesivamente.

Los operandos podrán ser valores numéricos (enteros o decimales) con signo, por ejemplo 8, 8.2, -3, -309.6, 108.42345. Un operando del tipo definido lo podemos modelizar de la siguiente manera:



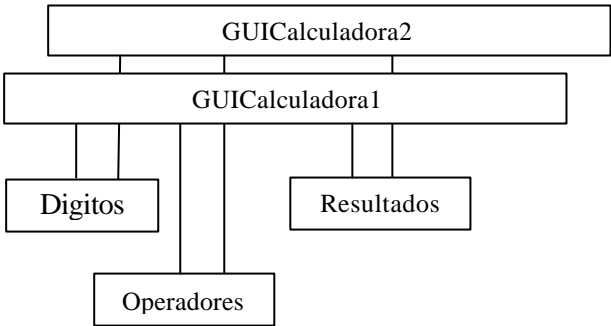
Por último, deseamos que la calculadora cumpla las siguientes condiciones:

- Diseñada según los principios básicos de la programación orientada a objetos
- Utilización amplia de las posibilidades de Java que han sido estudiadas
- Los botones pueden definirse de diversos colores
- Cuando nos situamos sobre un botón, éste debe cambiar de color y al salir del mismo recuperar el color original. Este comportamiento se debe cumplir tanto al usar el teclado como al usar el ratón
- Cuando pulsemos una opción no válida (por ejemplo un dígito después del signo igual, dos operadores seguidos, etc.) nos muestre una indicación de error en el área de resultados y el botón pulsado de color rojo

### 9.1.2 Diseño del interfaz gráfico de usuario

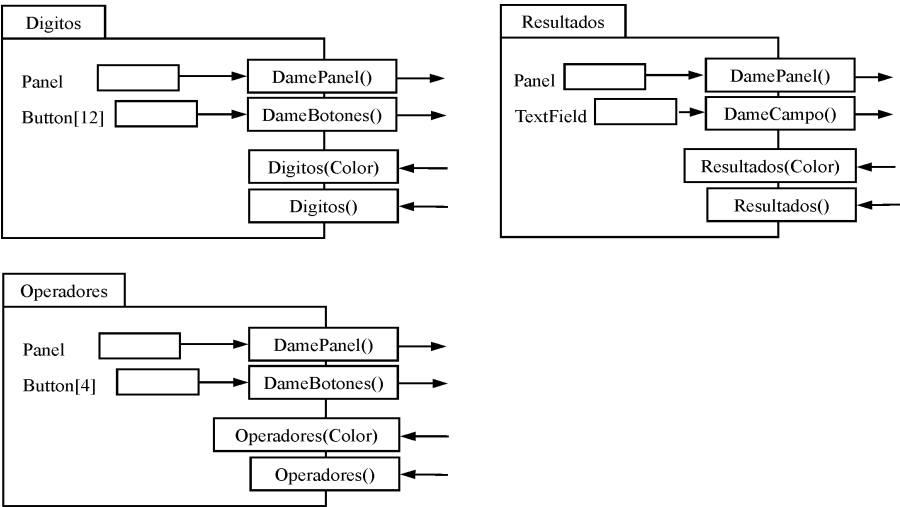
Para realizar el diseño de la aplicación estableceremos en primer lugar la forma de crear el interfaz gráfico de usuario, posteriormente decidiremos como estructurar las clases que implementarán su comportamiento. El interfaz gráfico de usuario lo vamos a basar en tres clases muy sencillas: *Digitos*, *Operadores* y *Resultados*, la primera define un panel con botones que representan los dígitos del 0 al 9, el signo punto y el igual; la clase operadores define un panel con 4 botones de signos: +, -, \*, /; *Resultados* se encarga del visor de la calculadora.

Utilizando las clases *Digitos*, *Operadores* y *Resultados* podemos definir diferentes representaciones de calculadoras, implementando clases *GUICalculadora1*, *GUICalculadora2*, etc. El esquema de clases, hasta el momento, nos queda de la siguiente manera:



Las clases *GUICalculadora<sub>x</sub>* necesitan tener acceso a los botones individuales que se definen en las clases *Digitos* y *Operadores*, de esta manera se podrá asignar diferentes objetos de tratamiento de eventos a los botones. También resulta necesario que las clases *Digitos* y *Operadores* proporcionen sus respectivos paneles con la estructura de botones. Finalmente, para permitir que los botones de cada panel presenten el color deseado, se proporciona un constructor que admite un parámetro de tipo *Color*.

La clase *Resultados* proporciona un campo de texto deshabilitado que hará las veces de visor de la calculadora. A continuación se muestra el diseño de las clases *Digitos*, *Operadores* y *Resultados*:



### 9.1.3 Implementación del interfaz gráfico de usuario

La clase *Digitos* crea un panel (línea 8) en el que introduce 12 botones (líneas 9, 10 y 23). El panel es accesible a través del método *DamePanel* (línea 31) y los botones son accesibles a través del método *DameBotones* (línea 35). El color de los botones se determina en la llamada al constructor con parámetro (líneas 12 y 22); el constructor vacío asigna el color gris claro (líneas 27 y 28).

Los botones se sitúan en una disposición *GridLayout* de 4 x 3 (líneas 13 y 20); las tres primeras filas albergan los dígitos de 0 a 8 (líneas 14 a 16) y la tercera el dígito 9, el punto y el igual (líneas 17 a 19).

```
1  import java.awt.Button;
2  import java.awt.Panel;
3  import java.awt.GridLayout;
4  import java.awt.Color;
5
6  public class Digitos {
7
8      private Panel MiPanel = new Panel();
9      private final int NUM_DIGITOS=12;
10     private Button[] Digito = new Button[NUM_DIGITOS];
11
12     Digitos(Color ColorBotones) {
13         GridLayout LayoutBotones = new GridLayout(4,3);
14         for (int fila=0;fila<3;fila++)
15             for (int col=0;col<3;col++)
16                 Digito[fila*3+col] = new
17                     Button(String.valueOf(fila*3+col));
18         Digito[9] = new Button("9");
19         Digito[10] = new Button(".");
20         Digito[11] = new Button("=");
21         MiPanel.setLayout(LayoutBotones);
22         for (int i=0;i<NUM_DIGITOS;i++) {
23             Digito[i].setBackground(ColorBotones);
24             MiPanel.add(Digito[i]);
25         }
26
27     Digitos() {
28         this(Color.lightGray);
29     }
30
31     public Panel DamePanel() {
32         return MiPanel;
33     }
34 }
```



```
35 public Button[] DameBotones() {
36     return Digito;
37 }
38
39 }
```

Las clases *Operadores* y *Resultados* siguen el mismo esquema que *Digitos*:

```
1 import java.awt.Button;
2 import java.awt.Panel;
3 import java.awt.GridLayout;
4 import java.awt.Color;
5
6 public class Operadores {
7
8     private Panel MiPanel = new Panel();
9     private final int NUM_OPERADORES=4;
10    private Button[] Operador = new Button[NUM_OPERADORES];
11
12    Operadores(Color ColorBotones) {
13        GridLayout LayoutBotones = new
14                                GridLayout(NUM_OPERADORES,1);
15        Operador[0] = new Button("+");
16        Operador[1] = new Button("-");
17        Operador[2] = new Button("*");
18        Operador[3] = new Button("/");
19        MiPanel.setLayout(LayoutBotones);
20        for (int i=0;i<NUM_OPERADORES;i++) {
21            Operador[i].setBackground(ColorBotones);
22            MiPanel.add(Operador[i]);
23        }
24
25    Operadores() {
26        this(Color.lightGray);
27    }
28
29    public Panel DamePanel() {
30        return MiPanel;
31    }
32
33    public Button[] DameBotones() {
34        return Operador;
35    }
36
37 }
```

```
1  import java.awt.*;
2
3  public class Resultados {
4
5      private Panel MiPanel = new Panel();
6      private TextField Resultado = new TextField("",10);
7
8      Resultados(Color ColorEtiqueta) {
9          FlowLayout LayoutResultado = new
                                FlowLayout(FlowLayout.LEFT);
10         MiPanel.setLayout(LayoutResultado);
11         Resultado.setForeground(ColorEtiqueta);
12         MiPanel.add(Resultado);
13         Resultado.setEnabled(false);
14     }
15
16     Resultados() {
17         this(Color.black);
18     }
19
20     public Panel DamePanel() {
21         return MiPanel;
22     }
23
24     public TextField DameCampo() {
25         return Resultado;
26     }
27
28 }
```

La clase *GUICalculadora1* implementa el interfaz de calculadora que se ha presentado en el primer apartado: “Definición del ejemplo”. Primero se define un panel con disposición *BorderLayout* (líneas 7 a 9), posteriormente se crea una instancia de la clase *Digitos*, otra de la clase *Operadores* y una tercera de la clase *Resultados* (líneas 11 a 13). Finalmente se obtienen los paneles de cada una de las instancias y se sitúan en las posiciones este, centro y norte de nuestro panel (líneas 16 a 18).

```
1  import java.awt.*;
2
3  public class GUICalculadora1 {
4
5      GUICalculadora1() {
6          Frame MiMarco = new Frame();
7          Panel MiPanel = new Panel();
8          BorderLayout PuntosCardinales = new BorderLayout();
9          MiPanel.setLayout(PuntosCardinales);
10 }
```

```

11     Digitos InstanciaDigitos = new Digitos(Color.orange);
12     Operadores InstanciaOperadores = new
        Operadores(Color.magenta);
13     Resultados InstanciaResultados = new Resultados();
14
15     MiMarco.add(MiPanel);
16     MiPanel.add(InstanciaOperadores.DamePanel(),
        BorderLayout.EAST);
17     MiPanel.add(InstanciaDigitos.DamePanel(),
        BorderLayout.CENTER);
18     MiPanel.add(InstanciaResultados.DamePanel(),
        BorderLayout.NORTH);
19
20     // Aqui prepararemos el tratamiento de eventos
21
22     MiMarco.setSize(150,150);
23     MiMarco.setTitle("Calculadora");
24     MiMarco.setVisible(true);
25 }
26 }

1 public class Calculadora1 {
2
3     public static void main(String[] args) {
4         GUICalculadora1 MiCalculadora = new GUICalculadora1();
5     }
6 }

```

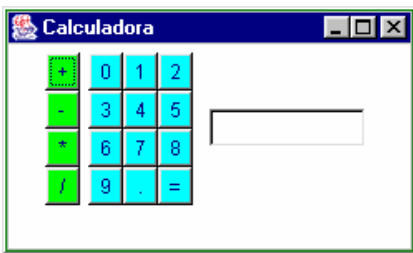
Utilizando las clases *Digitos*, *Operadores* y *Resultados* podemos crear de forma muy sencilla nuevas apariencias de calculadoras:

```

1 import java.awt.*;
2
3 public class GUICalculadora2 {
4
5     GUICalculadora2() {
6         Frame MiMarco = new Frame();
7         Panel MiPanel = new Panel();
8         FlowLayout TodoSeguido = new
        FlowLayout(FlowLayout.CENTER);
9         MiPanel.setLayout(TodoSeguido);
10
11         Digitos PanelDigitos = new Digitos(Color.cyan);
12         Operadores PanelOperadores = new
        Operadores(Color.green);
13         Resultados PanelResultados = new Resultados(Color.red);
14
15         MiMarco.add(MiPanel);

```

```
16     MiPanel.add(PanelOperadores.DamePanel());
17     MiPanel.add(PanelDigitos.DamePanel());
18     MiPanel.add(PanelResultados.DamePanel());
19
20     MiMarco.setSize(250,150);
21     MiMarco.setTitle("Calculadora");
22     MiMarco.setVisible(true);
23 }
24 }
```



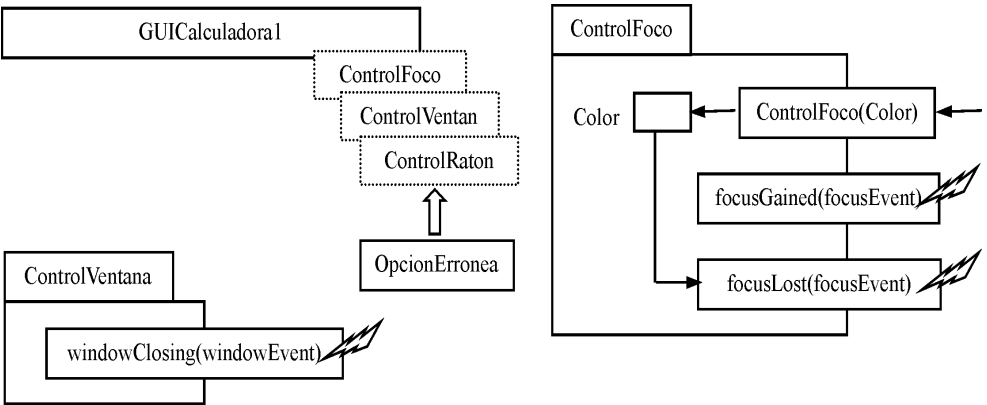
### 9.1.4 Diseño del tratamiento de eventos

La calculadora que estamos desarrollando en este ejemplo va a atender a tres tipos de eventos diferentes:

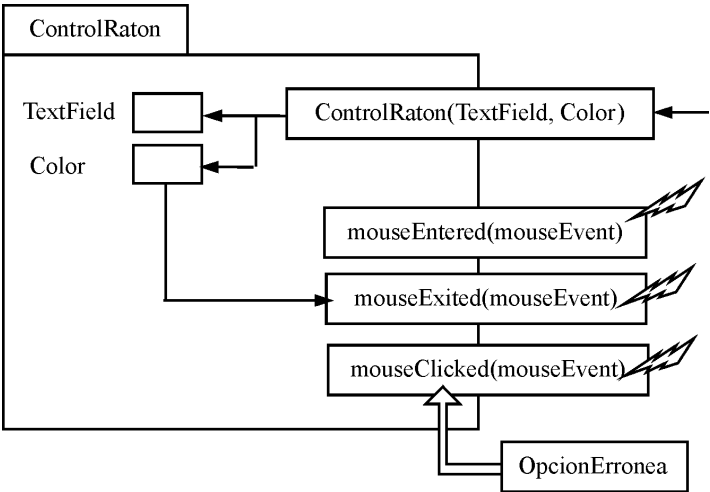
- Eventos de ventana
- Eventos de enfoque
- Eventos de ratón

Los dos primeros tipos de eventos van a recibir un tratamiento muy sencillo, mientras que el tercer tipo, eventos de ratón, va a incorporar la mayor parte del dinamismo de la aplicación.

La clase de tratamiento de eventos de ventana atenderá al método *windowClosing*, con el fin de permitir la finalización de la aplicación. La clase de tratamiento de eventos de enfoque implementará los dos métodos del interfaz: *focusGained* y *focusLost*, de esta manera conseguiremos cambiar el color de los botones cuando nos situemos sobre ellos con el teclado: verde al colocarse sobre cada uno de ellos y su color original (proporcionado a través del constructor) al salir de los mismos. El diseño de las clases queda de la siguiente manera:



La clase de tratamiento de eventos de ratón (*ControlRaton*) contiene un constructor que permite indicar el color del fondo de los botones y el campo de texto que hace de visor de la calculadora. El primer parámetro sirve para que el método *mouseExited* restituya el color de cada botón al salir del mismo (*mouseEntered* pone de color verde cada botón en el que se coloca el puntero del ratón). El segundo parámetro (*TextField*) permite que el método *mouseClicked* sitúe en el visor un mensaje de situación errónea (circunstancia que conoce por la llegada de la excepción *OpcionErronea*).



### 9.1.5 Implementación de las clases de tratamiento de eventos

La clase *controlVentana* se encarga de finalizar la aplicación (línea 6) cuando el usuario selecciona una opción de cierre de la ventana (línea 5).

```
1 import java.awt.event.*;
2
3 public class ControlVentana extends WindowAdapter {
4
5     public void windowClosing(WindowEvent EventoQueLlega){
6         System.exit(0);
7     }
8
9 }
```

La clase *controlFoco* se encarga de variar el color de los botones a medida que el usuario se desplaza por los mismos usando el teclado. El método *focusGained* (línea 12) pone los botones por los que se pasa en color verde y el método *focusLost* (línea 17) los pone, al salir, del color original (línea 19).

```
1 import java.awt.event.*;
2 import java.awt.*;
3
4 public class ControlFoco implements FocusListener {
5
6     private Color ColorBoton;
7
8     ControlFoco(Color ColorBoton) {
9         this.ColorBoton = ColorBoton;
10    }
11
12    public void focusGained(FocusEvent EventoQueLlega){
13        Button Boton = (Button) EventoQueLlega.getSource();
14        Boton.setBackground(Color.green);
15    }
16
17    public void focusLost(FocusEvent EventoQueLlega){
18        Button Boton = (Button) EventoQueLlega.getSource();
19        Boton.setBackground(ColorBoton);
20    }
21
22 }
```

El último de los controladores de eventos utilizados en nuestro ejemplo es *ControlRaton*, de tipo *MouseAdapter*. Su constructor (línea 9) permite indicar el color de los botones y el visor (de tipo *TextField*) que se está utilizando. Los

métodos *mouseEntered* (línea 26) y *mouseExited* (línea 31) realizan en esta clase la misma acción que *focusGained* y *focusLost* en la clase anterior, de esta forma, el color de los botones también varía al desplazarnos con el ratón.

El método *mouseClicked* (línea 14) obtiene el botón que generó el evento (línea 15) y posteriormente extrae el primer carácter de su etiqueta (línea 16). El carácter obtenido (0,1,...,9,+,-,\*,/,.,=) se le pasa a un método de la clase *Automata* para que lo procese; esta clase se explica en el siguiente apartado; se puede producir una excepción del tipo *OpcionErronea* si el botón seleccionado es inadecuado (línea 20) en cuyo caso se coloca un mensaje de error sobre el visor (línea 21) y se asigna el color rojo al botón que ha sido pulsado (línea 22).

```
1  import java.awt.event.*;
2  import java.awt.*;
3
4  public class ControlRaton extends MouseAdapter {
5
6      private TextField Resultado;
7      private Color ColorBoton;
8
9      ControlRaton(TextField Resultado,Color ColorBoton) {
10         this.Resultado = Resultado;
11         this.ColorBoton = ColorBoton;
12     }
13
14     public void mouseClicked(MouseEvent EventoQueLlega){
15         Button Boton = (Button) EventoQueLlega.getSource();
16         char Car = Boton.getLabel().charAt(0);
17         System.out.print(Car);
18         try {
19             Automata.CaracterIntroducido(Car);
20         } catch(OpcionErronea e) {
21             Resultado.setText(e.getMessage());
22             Boton.setBackground(Color.red);
23         }
24     }
25
26     public void mouseEntered(MouseEvent EventoQueLlega){
27         Button Boton = (Button) EventoQueLlega.getSource();
28         Boton.setBackground(Color.green);
29     }
30
31     public void mouseExited(MouseEvent EventoQueLlega){
32         Button Boton = (Button) EventoQueLlega.getSource();
33         Boton.setBackground(ColorBoton);
34     }
35 }
```

Para finalizar este apartado se presenta la clase *GUICalculadora1* completa, con los tratamientos de eventos asociados a los botones. En las líneas 20 a 22 se obtienen las referencias de los botones y el campo de texto que componen la calculadora, para ello se hace uso de los métodos *DameBotones* y *DameCampo* pertenecientes a las clases *Digitos*, *Operadores* y *Resultados*. En la línea 23 se instancia el autómatas de control que veremos en el próximo apartado.

En las líneas 25 a 35 se asocian instancias de las clases de tratamientos de eventos *ControlRaton* y *ControlFoco* a todos los botones de la calculadora; finalmente, en la línea 37 se asocia una instancia de la clase *ControlVentana* al marco principal de la aplicación (*MiMarco*).

```
1  import java.awt.*;
2
3  public class GUICalculadora1 {
4
5      GUICalculadora1() {
6          Frame MiMarco = new Frame();
7          Panel MiPanel = new Panel();
8          BorderLayout PuntosCardinales = new BorderLayout();
9          MiPanel.setLayout(PuntosCardinales);
10
11          Digitos InstanciaDigitos = new Digitos(Color.orange);
12          Operadores InstanciaOperadores = new
13              Operadores(Color.magenta);
14          Resultados InstanciaResultados = new Resultados();
15
16          MiMarco.add(MiPanel);
17          MiPanel.add(InstanciaOperadores.DamePanel(),
18              BorderLayout.EAST);
19          MiPanel.add(InstanciaDigitos.DamePanel(),
20              BorderLayout.CENTER);
21          MiPanel.add(InstanciaResultados.DamePanel(),
22              BorderLayout.NORTH);
23
24          Button[] BotonesDigitos =
25              InstanciaDigitos.DameBotones();
26          Button[] BotonesOperadores =
27              InstanciaOperadores.DameBotones();
28          TextField Resultado = InstanciaResultados.DameCampo();
29          Automata InstanciaAutomata = new Automata(Resultado);
30
31          for (int i=0;i<BotonesDigitos.length;i++) {
32              BotonesDigitos[i].addMouseListener(new
33                  ControlRaton(Resultado,Color.orange));
34              BotonesDigitos[i].addFocusListener(new
35                  ControlFoco(Color.orange));
36          }
37      }
38  }
```



```

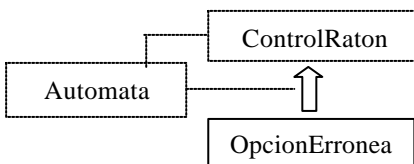
29     }
30
31     for (int i=0;i<BotonesOperadores.length;i++) {
32         BotonesOperadores[i].addMouseListener(new
33             ControlRaton(Resultado,Color.magenta));
34         BotonesOperadores[i].addFocusListener(new
35             ControlFoco(Color.magenta));
36     }
37     MiMarco.addWindowListener(new ControlVentana());
38
39     MiMarco.setSize(150,150);
40     MiMarco.setTitle("Calculadora");
41     MiMarco.setVisible(true);
42 }
43
44 }

```

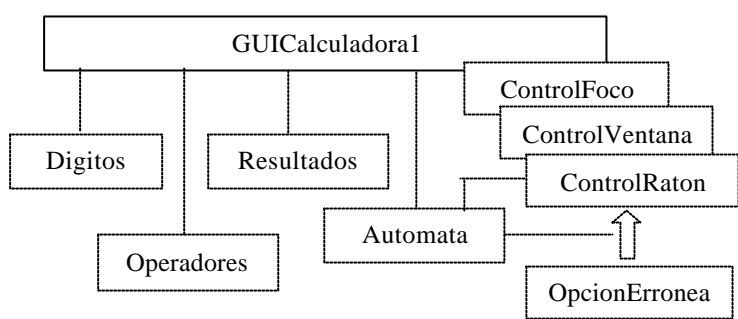
### 9.1.6 Diseño del control

Hasta ahora hemos resuelto la manera de visualizar la calculadora y la forma de interactuar con el usuario recogiendo sus pulsaciones de ratón, modificando los colores de los botones a medida que se pasa por ellos, etc. Lo único que nos queda por hacer es implementar la lógica de control de la calculadora: utilizar las pulsaciones del usuario como entrada y proporcionar resultados como salida.

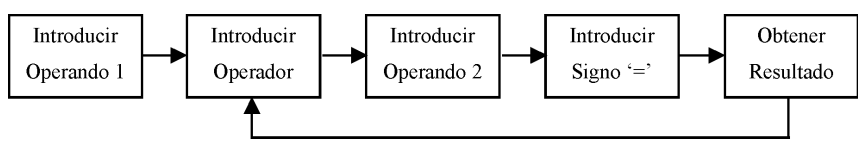
La clase que recoge las pulsaciones que realiza el usuario es *ControlRaton*, que se encarga del tratamiento de eventos de ratón; para no mezclar su función básica (recoger eventos) con el tratamiento (control) de los mismos crearemos una clase *Automata* que realice esta última labor; *ControlRaton* le pasará en forma de caracteres ('0', '1', ..., '9', '.', '+', '-', '\*', '/', '=') las pulsaciones del usuario y *Automata* las procesará. En caso de que el usuario pulse un botón inadecuado (por ejemplo dos operandos seguidos) *Automata* generará una excepción *OpcionErronea* y *ControlRaton* se encargará de recogerla. *OpcionErronea* es una excepción que nos definimos nosotros mismos.



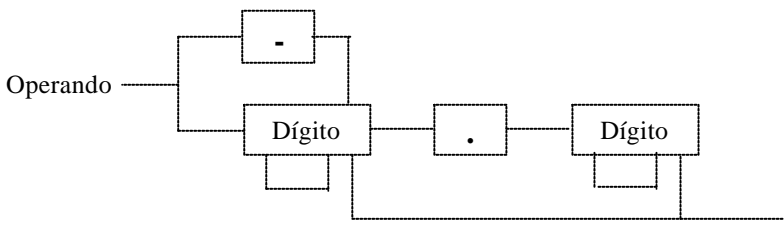
El esquema completo de clases de la aplicación nos queda de la siguiente manera:



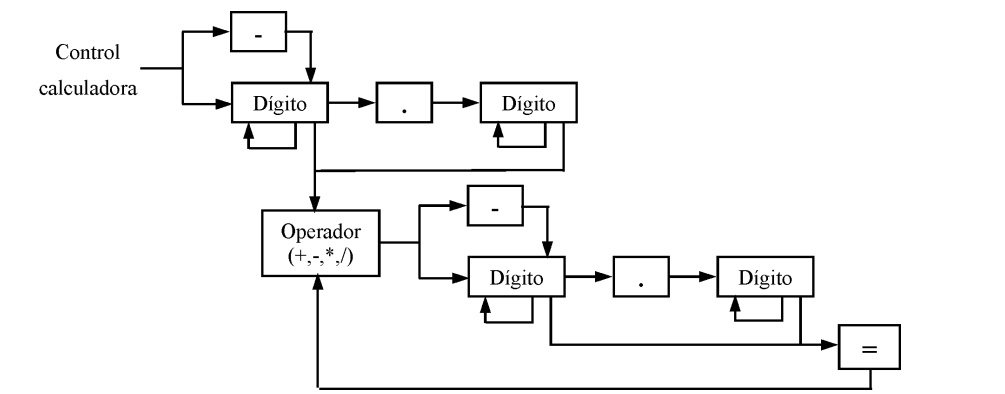
Siguiendo el modelo gráfico con el que fomalizabamos el formato de un número, podemos establecer el control completo de la calculadora. Recordemos que el esquema general es:



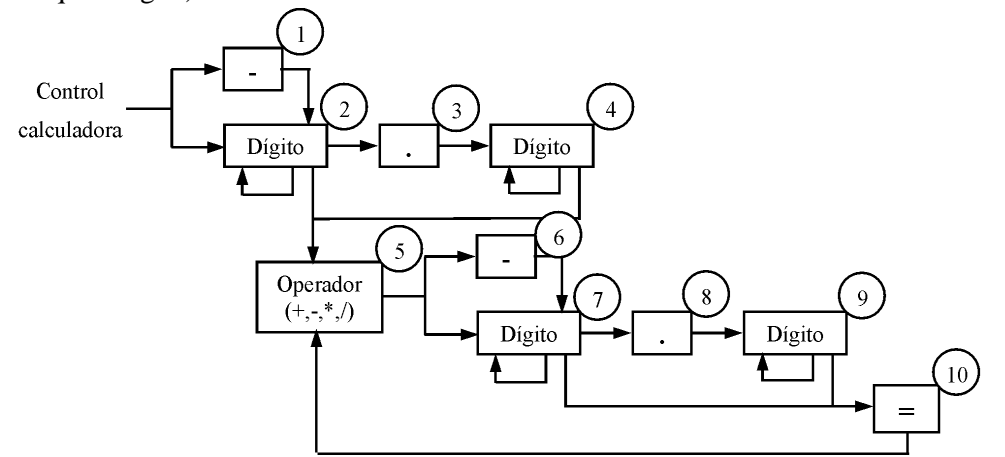
... y el de un número:



Combinando los dos grafos anteriores podemos obtener con facilidad el autómata de estados en el que nos vamos a basar para implementar el control de la calculadora. Las cajas representan los diferentes estados en los que nos podemos encontrar y las flechas son las transiciones permitidas desde cada uno de esos estados. Numerando los estados, en el gráfico siguiente, podemos expresar, por ejemplo, que en el estado 5 hemos recibido un número y un operador y estamos a la espera de un dígito o del signo menos.



Siguiendo el esquema nos resulta muy sencillo saber como vamos evolucionando según el usuario pulsa botones; también resulta inmediato conocer las pulsaciones permitidas en cada situación (por ejemplo, en el estado 7 solo podemos admitir como válidas las pulsaciones en los botones punto, igual o cualquier dígito).



9.1.7 Implementación del control

La clase *Automata* sigue fielmente el grafo de estados que hemos diseñado. La clase *ControlRaton* invoca al método *CaracterIntroducido* (línea 15) por cada pulsación de botón que realiza el usuario. La cuestión más importante en este momento es darse cuenta de que el estado del autómata debe mantenerse entre

pulsación y pulsación del usuario, por lo que debemos emplear propiedades que tengan existencia durante toda la aplicación: no tendría sentido guardar el estado del autómata, el valor de los operandos, etc. en variables locales al método *CaracterIntroducido*.

Las propiedades que necesitaremos para almacenar el estado del autómata son: *Estado* (indica el valor numérico del estado en el que nos encontramos), *Visor* (contiene los últimos caracteres pulsados), *Operador* (operador seleccionado), *Operando1* y *Operando2* (valores a partir de los que se calcula el resultado).

A modo de ejemplo hemos definido estas propiedades como estáticas, así como el método *CaracterIntroducido* que las utiliza: la aplicación funciona correctamente y evitamos tener que pasar la referencia de la instancia del autómata hasta la clase *ControlRaton*, sin embargo debemos tener en cuenta que si en una misma aplicación deseamos crear más de una calculadora, el control no funcionaría, puesto que todas las calculadoras de la aplicación compartirían las mismas propiedades de estado. Si cada calculadora se encontrase en una aplicación separada sí funcionaría, puesto que cada calculadora se ejecutaría sobre una JVM diferente.

Evitar las propiedades estáticas en nuestra aplicación es muy sencillo, basta con pasar a la clase *ControlRaton* la referencia del objeto *Automata* y, en su línea 19, utilizar la referencia en lugar del nombre de la clase.

El código de la clase *Automata* sigue el diseño realizado: su método *CaracterIntroducido* consulta en primer lugar el estado en el que se encuentra el autómata (línea 19) para realizar un tratamiento u otro (marcado por el grafo de estados desarrollado en el apartado anterior); se consulta el carácter que nos llega (líneas 22, 48, etc.), correspondiente al último botón pulsado por el usuario y se actúa en consecuencia.

Por ejemplo, si estamos en el estado 1 (línea 47) y nos llega un dígito (líneas 49 a 58), pasamos al estado 2 (tal y como marca el grafo de estados) y actualizamos el visor de la calculadora (línea 60); si nos llega un carácter distinto a un dígito (línea 62) nos encontramos ante un error del usuario, por lo que volvemos al estado 0 del autómata (línea 63) y levantamos la excepción *OpcionErronea* (línea 64).

Los demás estados se tratan de una manera análoga al explicado, en todos se levanta la misma excepción (*OpcionErronea*) cuando llega un carácter equivocado. Resultaría muy sencillo y útil pasarle a la excepción un texto indicando los caracteres permitidos en cada caso: por ejemplo en el estado 10: *throw new OpcionErronea("+-\*")*, incorporando un constructor con parámetro de tipo String a la excepción *OpcionErronea*.

Los resultados se calculan cuando nos encontramos en el estado 9 (línea 260) y nos llega el carácter '=' (línea 262): pasamos al estado 10 (línea 263), obtenemos el segundo operando a partir de los caracteres almacenados en el visor (línea 264), calculamos el resultado (línea 265), lo visualizamos (línea 266) y asignamos el resultado como primer operando para la siguiente operación (línea 267).

La clase *OpcionErronea* se ha mantenido lo más sencilla posible; únicamente almacena el texto "No valido" como indicación de la causa de la excepción, que siempre levantamos cuando el usuario pulsa un botón inapropiado. El código de esta clase se muestra al final del apartado.

```
1  import java.awt.TextField;
2
3  public class Automata {
4
5      private static byte Estado=0;
6      private static TextField Visor;
7      private static double Operando1=0d;
8      private static double Operando2=0d;
9      private static char Operador=' ';
10
11     Automata(TextField Visor) {
12         this.Visor = Visor;
13     }
14
15     public static void CaracterIntroducido(char Car) throws
16                                     OpcionErronea {
17         if (Visor.getText().equals("No valido"))
18             Visor.setText("");
19
20         switch(Estado) {
21             case 0:
22                 switch(Car) {
23                     case '-':
24                         Estado=1;
25                         Visor.setText(Visor.getText()+Car);
26                         break;
27                     case '0':
28                     case '1':
29                     case '2':
30                     case '3':
31                     case '4':
32                     case '5':
33                     case '6':
34                     case '7':
```

```
35         case '8':
36         case '9':
37             Estado=2;
38             Visor.setText(Visor.getText()+Car);
39             break;
40         default:
41             Iniciar();
42             throw new OpcionErronea();
43     }
44     break;
45
46
47     case 1:
48         switch(Car) {
49             case '0':
50             case '1':
51             case '2':
52             case '3':
53             case '4':
54             case '5':
55             case '6':
56             case '7':
57             case '8':
58             case '9':
59                 Estado=2;
60                 Visor.setText(Visor.getText()+Car);
61                 break;
62             default:
63                 Iniciar();
64                 throw new OpcionErronea();
65         }
66     break;
67
68
69     case 2:
70         switch(Car) {
71             case '.':
72                 Estado=3;
73                 Visor.setText(Visor.getText()+Car);
74                 break;
75             case '+':
76             case '-':
77             case '*':
78             case '/':
79                 Estado=5;
80                 Operador=Car;
81                 Operandol=Double.parseDouble(Visor.getText());
82                 Visor.setText("");
```

```
83         break;
84     case '0':
85     case '1':
86     case '2':
87     case '3':
88     case '4':
89     case '5':
90     case '6':
91     case '7':
92     case '8':
93     case '9':
94         Estado=2;
95         Visor.setText(Visor.getText()+Car);
96         break;
97     default:
98         Iniciar();
99         throw new OpcionErronea();
100    }
101    break;
102
103
104    case 3:
105        switch(Car) {
106            case '0':
107            case '1':
108            case '2':
109            case '3':
110            case '4':
111            case '5':
112            case '6':
113            case '7':
114            case '8':
115            case '9':
116                Estado=4;
117                Visor.setText(Visor.getText()+Car);
118                break;
119            default:
120                Iniciar();
121                throw new OpcionErronea();
122        }
123    break;
124
125
126    case 4:
127        switch(Car) {
128            case '+':
129            case '-':
130            case '*':
```

```
131         case '/':
132             Estado=5;
133             Operador=Car;
134             Operando1=Double.parseDouble(Visor.getText());
135             Visor.setText("");
136             break;
137         case '0':
138         case '1':
139         case '2':
140         case '3':
141         case '4':
142         case '5':
143         case '6':
144         case '7':
145         case '8':
146         case '9':
147             Estado=4;
148             Visor.setText(Visor.getText()+Car);
149             break;
150         default:
151             Iniciar();
152             throw new OpcionErronea();
153     }
154     break;
155
156
157     case 5:
158         switch(Car) {
159             case '-':
160                 Estado=6;
161                 Visor.setText(Visor.getText()+Car);
162                 break;
163             case '0':
164             case '1':
165             case '2':
166             case '3':
167             case '4':
168             case '5':
169             case '6':
170             case '7':
171             case '8':
172             case '9':
173                 Estado=7;
174                 Visor.setText(Visor.getText()+Car);
175                 break;
176             default:
177                 Iniciar();
178                 throw new OpcionErronea();
```



```
179         }
180         break;
181
182
183     case 6:
184         switch(Car) {
185             case '0':
186             case '1':
187             case '2':
188             case '3':
189             case '4':
190             case '5':
191             case '6':
192             case '7':
193             case '8':
194             case '9':
195                 Estado=7;
196                 Visor.setText(Visor.getText()+Car);
197                 break;
198             default:
199                 Iniciar();
200                 throw new OpcionErronea();
201         }
202         break;
203
204
205     case 7:
206         switch(Car) {
207             case '.':
208                 Estado=8;
209                 Visor.setText(Visor.getText()+Car);
210                 break;
211             case '=':
212                 Estado=10;
213                 Operando2=Double.parseDouble(Visor.getText());
214                 double Resultado=ObtenerResultado();
215                 Visor.setText(String.valueOf(Resultado));
216                 Operando1=Resultado;
217                 break;
218             case '0':
219             case '1':
220             case '2':
221             case '3':
222             case '4':
223             case '5':
224             case '6':
225             case '7':
226             case '8':
```

```
227         case '9':
228             Estado=7;
229             Visor.setText(Visor.getText()+Car);
230             break;
231         default:
232             Iniciar();
233             throw new OpcionErronea();
234     }
235     break;
236
237
238     case 8:
239         switch(Car) {
240             case '0':
241             case '1':
242             case '2':
243             case '3':
244             case '4':
245             case '5':
246             case '6':
247             case '7':
248             case '8':
249             case '9':
250                 Estado=9;
251                 Visor.setText(Visor.getText()+Car);
252                 break;
253             default:
254                 Iniciar();
255                 throw new OpcionErronea();
256         }
257         break;
258
259
260     case 9:
261         switch(Car) {
262             case '=':
263                 Estado=10;
264                 Operando2=Double.parseDouble(Visor.getText());
265                 double Resultado=ObtenerResultado();
266                 Visor.setText(String.valueOf(Resultado));
267                 Operando1=Resultado;
268                 break;
269             case '0':
270             case '1':
271             case '2':
272             case '3':
273             case '4':
274             case '5':
```

```
275         case '6':
276         case '7':
277         case '8':
278         case '9':
279             Estado=9;
280             Visor.setText(Visor.getText()+Car);
281             break;
282         default:
283             Iniciar();
284             throw new OpcionErronea();
285     }
286     break;
287
288
289     case 10:
290         switch(Car) {
291             case '+':
292             case '-':
293             case '*':
294             case '/':
295                 Estado=5;
296                 Operador=Car;
297                 Visor.setText("");
298                 break;
299             default:
300                 Iniciar();
301                 throw new OpcionErronea();
302         }
303         break;
304
305     }
306
307 }
308
309
310 private static void Iniciar(){
311     Estado=0;
312     Visor.setText("");
313     Operando1=0d;
314     Operando2=0d;
315     Operador=' ';
316 }
317
318
319 private static double ObtenerResultado() {
320     switch(Operador) {
321         case '+':
322             return Operando1+Operando2;
```

```
323         case '-':
324             return Operando1-Operando2;
325         case '*':
326             return Operando1*Operando2;
327         case '/':
328             return Operando1/Operando2;
329         default:
330             return 0d;
331     }
332 }
333
334 }
```

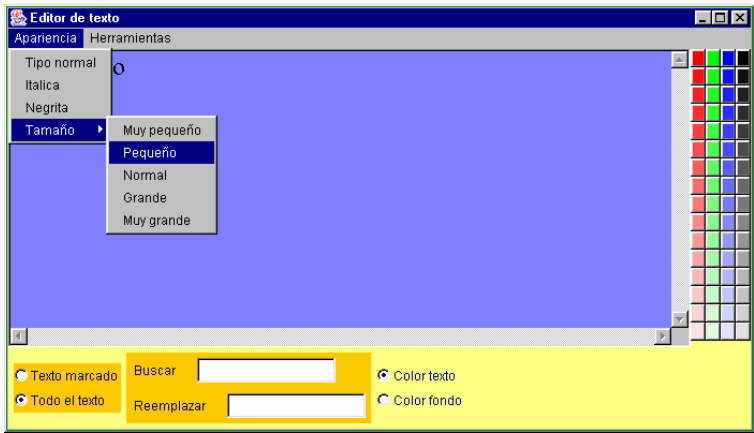
```
1 public class OpcionErronea extends Exception {
2
3     OpcionErronea(){
4         super("No valido");
5     }
6
7 }
```

## 9.2 EJEMPLO: EDITOR

### 9.2.1 Definición del ejemplo

La aplicación consiste en una primera aproximación a un editor de texto sencillo basado en la utilización de componentes AWT. En este ejemplo se hace uso de diversos interfaces y adaptadores de tratamiento de eventos. El interfaz gráfico que incorpora la aplicación presenta el aspecto que muestra la figura siguiente.

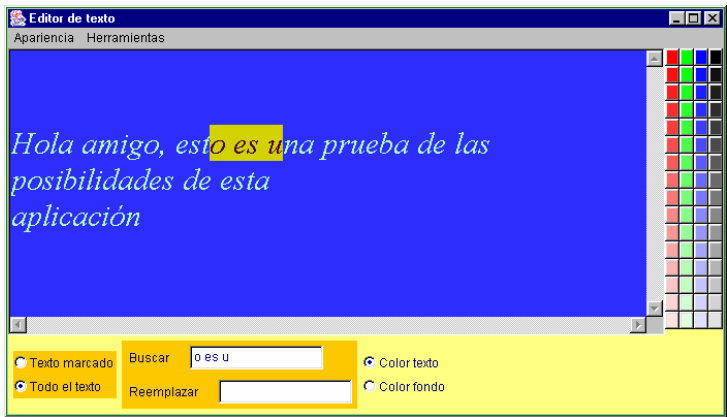
El área central permite la edición de texto y está implementada con un componente “área de texto”. La barra de menús situada en la parte superior de la ventana contiene las opciones “Apariencia” y “Herramientas”; el primer menú permite cambiar el tamaño y el estilo (itálica, negrita) del texto, el segundo hace posible pasar todo el texto a mayúsculas o a minúsculas.



La parte derecha de la ventana contiene un conjunto de botones que pueden ser utilizados para seleccionar colores; cada columna presenta una gama de gradación hacia el blanco partiendo del rojo, verde, azul y negro. La selección de un color actúa en combinación con los botones de radio “Color texto” y “Color fondo”, de manera que podemos conseguir variar el color del texto o del fondo del área de edición de texto.

El campo de texto “Buscar” permite hacer una búsqueda del literal suministrado sobre el texto editado; esta búsqueda se realiza sobre todo el texto editado o bien sobre la parte seleccionada (con el ratón o teclado) del mismo, para ello disponemos de los botones de radio “Texto marcado” y “Todo el texto”. La búsqueda se realiza hasta que se encuentra la primera ocurrencia del texto, seleccionándose automáticamente en el área de texto.

El campo de texto “Reemplazar” funciona en combinación con “Buscar” para reemplazar porciones de texto. Se reemplaza la primera ocurrencia que se encuentre coincidiendo con el texto suministrado en “Buscar”. Esta opción también actúa atendiendo a los botones de radio “Texto marcado” y “Todo el texto”.



La figura anterior muestra el interfaz de la aplicación tras haber sido empleadas algunas de sus posibilidades.

## 9.2.2 Estructura de la aplicación

A continuación se muestra un “esqueleto” de código que representa la estructura básica con la que se ha implementado la aplicación.

En primer lugar se declaran las propiedades que utilizaremos en la aplicación (componentes de AWT usados, constantes empleadas, variables auxiliares, etc.); posteriormente se implementa un constructor que admite como argumento un color que será utilizado como color de fondo del área de edición; después se invoca al método *PreparaMenus* (línea 10 y 27), que define la barra de menús y los propios menús desplegables.

El método *PreparaZonaInferior* (líneas 11 y 34) se encarga de presentar los botones de radio y cajas de texto en la parte inferior de la ventana. En la línea 12 se crea una instancia de la clase *Colores*; esta clase se ha implementado fuera de nuestra aplicación para facilitar su reutilización en otros programas. En la línea 13 se hace una llamada al método *DameBotones* de la clase *Colores*, de esta manera hacemos posible la asociación de manejadores de eventos a los botones que representan colores.

El resto del constructor se encarga de definir el GUI de la aplicación, tanto a nivel estático (visualización) como a nivel dinámico (enlace de los componentes con las clases de tratamiento de eventos).

En la línea 39 se declara una clase que permite que la aplicación finalice cuando el usuario cierra la ventana. El resto de las clases se encargan del tratamiento de eventos de la aplicación, ofreciendo la parte más instructiva de este ejemplo.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class GUIEditor {
5
6      // Propiedades de la clase
7
8      GUIEditor(Color ColorFondoEdicion) {
9
10         PreparaMenus();
```

```
11     PreparaZonaInferior();
12     Colores ZonaColores = new Colores(NUM_COLORES);
13     Button[][] Botones = ZonaColores.DameBotones();
14
15     // definición del interfaz grafico de usuario (GUI)
16
17     // Enlace de los components con las clases de
18     // tratamiento de eventos
19 }
20
21
22 GUIEditor() {
23     this(Color.white);
24 }
25
26
27 private void PreparaMenus() {
28
29     // Defincion y presentacion de los menus declarados
30     // previamete
31 }
32
33
34 private void PreparaZonaInferior() {
35     // Definicion y presentacion de los botones de radio y
36     // cajas de texto
37 }
38
39 private class CerrarVentana extends WindowAdapter {
40     public void windowClosing(WindowEvent e) {
41         System.exit(0);
42     }
43 }
44
45
46 private class EventosAccion implements ActionListener {
47     public void actionPerformed(ActionEvent e) {
48
49         // Tratamiento de las opciones de menu
50
51     } // actionPerformed
52 } // EventosAccion
53
54
55
```

```
56 private class EventoBuscaReemplaza implements
                                   ActionListener {
57     public void actionPerformed(ActionEvent e) {
58
59         // Tratamiento de las opciones "Buscar" y
           // "Reemplazar"
60
61     }
62 } // EventoBuscaReemplaza
63
64
65
66 private class EventosElemento implements ItemListener {
67     public void itemStateChanged(ItemEvent e) {
68
69         // Tratamiento de los botones de radio "Colortexto" y
           // "Color fondo"
70
71     } // itemStateChanged
72 } // EventosElemento
73
74
75
76 private class EventosRaton extends MouseAdapter {
77     public void mouseClicked(MouseEvent e) {
78
79         // Tratamiento a la pulsacion de los botones de
           // colores
80
81     } // mouseClicked
82 } // EventosRaton
83
84 }
```

A continuación se explican los detalles más relevantes de cada “bloque” de la aplicación:

- Propiedades de la clase
- Constructores
- Método *PreparaMenus*
- Método *PreparaZonaInferior*
- Clase *Colores*
- Clases de tratamiento de eventos
  - Tratamiento de las opciones de menú



- Tratamiento de las opciones “Buscar” y “Reemplazar”
- Tratamiento de los botones de radio “Color texto” y “Color fondo”
- Tratamiento a las pulsaciones de los botones de colores

### 9.2.3 Propiedades de la clase

La mayor parte de las propiedades de la clase definen los diferentes componentes del interfaz gráfico. En las 2 primeras líneas de código se declaran y definen el marco y su barra de menús; en las líneas 6, 7 y 8 se hace lo propio con los menús.

En la línea 3 se define el panel principal de la aplicación y en la línea 4 el panel inferior que contendrá los botones de radio y las cajas de texto. La línea 10 crea el área de introducción de texto.

En las líneas 11 y 12 se declaran las variables necesarias para saber a partir de qué posición (y hasta qué posición) del texto hay que hacer las búsquedas/reemplazamientos. En las líneas 13 y 14 se definen constantes que nos servirán para consultar la propiedad *TextoFondo* (línea 15) y saber si hay que asignar el color seleccionado al fondo del área de edición o bien al texto.

En las líneas 17 a 25 se definen los dos grupos de botones de radio; en las líneas 27 y 28 las dos cajas de texto. En las líneas 30 y 31 se declaran e inicializan variables que permiten mantener el estilo y tamaño del texto. Por fin, en la línea 33 se declara una constante con el número de colores (en cada columna) que deseamos que el usuario pueda seleccionar.

```

1    private Frame MiMarco = new Frame();
2    private MenuBar MenuPrincipal = new MenuBar();
3    private Panel PanelPrincipal = new Panel(new
                                           BorderLayout());
4    private Panel ZonaInferior = new Panel(new
                                           FlowLayout(FlowLayout.LEFT));
5
6    private Menu Apariencia = new Menu("Apariencia");
7    private Menu Tamano = new Menu("Tamaño ");
8    private Menu Herramientas = new Menu("Herramientas");
9
10   private TextArea AreaEdicion = new TextArea("Hola
                                                amigo");
11   private int PosicionInicial=0;
```

```
12 private int PosicionFinal=0;
13 private final int TEXTO = 1;
14 private final int FONDO = 2;
15 private int TextoFondo = TEXTO;
16
17 private CheckboxGroup Alcance = new CheckboxGroup();
18 private Checkbox TextoMarcado = new Checkbox("Texto
19                                     marcado", false, Alcance);
20 private Checkbox TextoCompleto = new Checkbox("Todo el
21                                     texto", true, Alcance);
22
23 private CheckboxGroup Marcado = new CheckboxGroup();
24 private Checkbox ColorTexto = new Checkbox("Color
25                                     texto",true, Marcado);
26 private Checkbox ColorFondo = new Checkbox("Color
27                                     fondo",false, Marcado);
28
29 private TextField Buscar = new TextField(15);
30 private TextField Reemplazar = new TextField(15);
31
32 private int TamanioFuente = 18;
33 private int TipoFuente = Font.PLAIN;
34
35 private final int NUM_COLORES = 16;
```

## 9.2.4 Constructores

El primer constructor invoca a los métodos auxiliares *PreparaMenus* y *PreparaZonaInferior* (líneas 3 y 4), instancia un objeto de la clase *Colores* (línea 5) y obtiene sus botones (línea 6); posteriormente asigna un nombre a la caja de texto *Buscar* (línea 9) y establece el color de fondo y de texto inicial del área de texto (líneas 10 a 15).

En las líneas 17 a 19 se añaden al panel principal: el área de texto, el panel *ZonaInferior* y el panel que proporciona la instancia de la clase *colores*. Las líneas 21 a 24 completan los pasos necesarios para dimensionar y hacer visible la ventana.

El resto del constructor se dedica a asignar los manejadores de eventos más adecuados para los componentes de la aplicación: cerrando la ventana podremos finalizar la aplicación (línea 26), los menús los controlamos con eventos de acción (líneas 27 a 29), las cajas de texto “Buscar y “reemplazar” también las controlamos con eventos de acción, pero definidos en una clase diferente.

Aunque no sería necesario asociar un tratamiento de eventos a los botones de radio “Color texto” y “Color fondo”, hemos optado por asignarles un tratamiento de eventos de elemento para mostrar su funcionamiento (líneas 32 y 33). Finalmente, las líneas 34 a 36 se encargan de asignar una instancia de tratamiento de eventos de ratón a cada botón de colores proporcionado por la clase *Colores*.

```

1  GUIEditor(Color ColorFondoEdicion) {
2
3      PreparaMenus();
4      PreparaZonaInferior();
5      Colores ZonaColores = new Colores(NUM_COLORES);
6      Button[][] Botones = ZonaColores.DameBotones();
7
8      PanelPrincipal.setBackground(new Color(255,255,128));
9      Buscar.setName("Busca");
10     AreaEdicion.setFont(new
11         Font("Serif",TipoFuente,TamanoFuente));
12     AreaEdicion.setBackground(ColorFondoEdicion);
13     if (ColorFondo.equals(Color.black))
14         AreaEdicion.setForeground(Color.white);
15     else
16         AreaEdicion.setForeground(Color.black);
17
18     PanelPrincipal.add(AreaEdicion, BorderLayout.CENTER);
19     PanelPrincipal.add(ZonaColores.DamePanel(),
20         BorderLayout.EAST);
21     PanelPrincipal.add(ZonaInferior, BorderLayout.SOUTH);
22
23     MiMarco.add(PanelPrincipal);
24     MiMarco.setSize(700,400);
25     MiMarco.setTitle("Editor de texto");
26     MiMarco.setVisible(true);
27
28     MiMarco.addWindowListener(new CerrarVentana());
29     Apariencia.addActionListener(new EventosAccion());
30     Tamano.addActionListener(new EventosAccion());
31     Herramientas.addActionListener(new EventosAccion());
32     Buscar.addActionListener(new EventoBuscaReemplaza());
33     Reemplazar.addActionListener(new
34         EventoBuscaReemplaza());
35
36     ColorTexto.addItemListener(new EventosElemento());
37     ColorFondo.addItemListener(new EventosElemento());
38     for (int i=0;i<NUM_COLORES;i++)
39         for (int j=0;j<=3;j++)
40             Botones[i][j].addMouseListener(new EventosRaton());
41 }

```

```
40  GUIEditor() {
41      this(Color.white);
42  }
```

### 9.2.5 Método *PreparaMenus*

Las líneas 2 a 6 se utilizan para definir las opciones del menú *Tamano*. Las líneas 8 a 11 definen las opciones del menú *Apariencia*; obsérvese como una opción de menú puede incluir a su vez un nuevo menú (línea 11). En las líneas 13 y 14 se definen las dos opciones del menú *Herramientas*.

En las líneas 16 y 17 se añaden los menús *Apariencia* y *Herramientas* a la barra de menús *MenuPrincipal*, que a su vez se asigna al marco *MiMarco* mediante el método *setMenuBar* (línea 19).

```
1      private void PreparaMenus() {
2          Tamano.add("Muy pequeño");
3          Tamano.add("Pequeño");
4          Tamano.add("Normal");
5          Tamano.add("Grande");
6          Tamano.add("Muy grande");
7
8          Apariencia.add("Tipo normal");
9          Apariencia.add("Italica");
10         Apariencia.add("Negrita");
11         Apariencia.add(Tamano);
12
13         Herramientas.add("Todo mayúsculas");
14         Herramientas.add("Todo minúsculas");
15
16         MenuPrincipal.add(Apariencia);
17         MenuPrincipal.add(Herramientas);
18
19         MiMarco.setMenuBar(MenuPrincipal);
20     }
```

### 9.2.6 Método *PreparaZonaInferior*

```
1 private void PreparaZonaInferior() {
2     Panel PanelAlcance = new Panel(new GridLayout(2,1));
3     Panel PanelBuscar = new Panel(new GridLayout(2,1));
4     Panel PanelMarcado = new Panel(new GridLayout(2,1));
5     Panel PanelBuscarArriba = new
        Panel(new FlowLayout(FlowLayout.LEFT));
6     Panel PanelBuscarAbajo = new
        Panel(new FlowLayout(FlowLayout.LEFT));
7
8     ZonaInferior.add(PanelAlcance);
9     ZonaInferior.add(PanelBuscar);
10    ZonaInferior.add(PanelMarcado);
11
12    PanelAlcance.setBackground(Color.orange);
13    PanelAlcance.add(TextoMarcado);
14    PanelAlcance.add(TextoCompleto);
15
16    PanelBuscar.setBackground(Color.orange);
17    PanelBuscar.add(PanelBuscarArriba);
18    PanelBuscar.add(PanelBuscarAbajo);
19
20    PanelMarcado.setBackground(new Color(255,255,128));
21    PanelMarcado.add(ColorTexto);
22    PanelMarcado.add(ColorFondo);
23
24    PanelBuscarArriba.add(new Label("Buscar "));
25    PanelBuscarArriba.add(Buscar);
26
27    PanelBuscarAbajo.add(new Label("Reemplazar "));
28    PanelBuscarAbajo.add(Reemplazar);
29 }
```

### 9.2.7 Clase *Colores*

La clase *Colores* proporciona un conjunto de botones de colores situados en un panel; se ha implementado como una clase independiente con el fin de facilitar su reutilización.

Existen tres propiedades globales en la clase:



```
20         BotonesColores[c][1].setBackground(new
                                   Color(Factor,255,Factor));
21         BotonesColores[c][2].setBackground(new
                                   Color(Factor,Factor,255));
22         BotonesColores[c][3].setBackground(new
                                   Color(Factor,Factor,Factor));
23     }
24 }
25
26 public Panel DamePanel() {
27     return PanelColores;
28 }
29
30 public Button[][] DameBotones() {
31     return BotonesColores;
32 }
33
34 }
```

### 9.2.8 Tratamiento de las opciones de menú

Los objetos *Menu* y *MenuItem* admiten la utilización del interfaz *ActionListener* para tratar eventos. En el constructor de la clase *GUIEditor* hemos asignado instancias de la clase *EventosAccion* (línea 1) a los menús *Apariencia*, *Tamaño* y *Herramientas*, de esta forma, cuando se ejecuta el método *actionPerformed* (línea 3), sabemos que ha sido seleccionada alguna opción de menú.

Para saber cuál de las opciones de menú ha sido seleccionada, consultamos el método *getActionCommand* (línea 5) perteneciente a la clase *ActionEvent* (línea 3); comparando (*equals*) el resultado con el texto que define cada opción de menú (“Negrita”, “Italica”, etc.) podemos determinar la opción pulsada.

Las tres primeras sentencias condicionales (líneas 5, 10 y 15) realizan el tratamiento de cada posible selección de estilo de letra. La variable *TipoFuente* se actualiza (líneas 6, 11 y 16) y se aplica una nueva fuente al texto del área de edición (líneas 7, 12 y 17).

Las siguientes 5 instrucciones condicionales se encargan de aplicar el tamaño de letra seleccionado. Como se puede observar, el tratamiento es igual al explicado, pero en este caso lo que variamos es la propiedad *TamañoFuente*.

Las últimas dos instrucciones condicionales se encargan de poner el texto en mayúsculas o minúsculas haciendo uso de los métodos *toUpperCase* y *toLowerCase*, pertenecientes a la clase *String*.

```
1 private class EventosAccion implements ActionListener {
2
3     public void actionPerformed(ActionEvent e) {
4
5         if (e.getActionCommand().equals("Negrita")) {
6             TipoFuente=Font.BOLD;
7             AreaEdicion.setFont(new
8                 Font("Serif",TipoFuente,TamanoFuente));
9         }
10        if (e.getActionCommand().equals("Italica")) {
11            TipoFuente=Font.ITALIC;
12            AreaEdicion.setFont(new
13                Font("Serif",TipoFuente,TamanoFuente));
14        }
15        if (e.getActionCommand().equals("Tipo normal")) {
16            TipoFuente=Font.PLAIN;
17            AreaEdicion.setFont(new
18                Font("Serif",TipoFuente,TamanoFuente));
19        }
20        if (e.getActionCommand().equals("Muy pequeño")) {
21            TamanoFuente=10;
22            AreaEdicion.setFont(new
23                Font("Serif",TipoFuente,TamanoFuente));
24        }
25        if (e.getActionCommand().equals("Pequeño")) {
26            TamanoFuente=14;
27            AreaEdicion.setFont(new
28                Font("Serif",TipoFuente,TamanoFuente));
29        }
30        if (e.getActionCommand().equals("Normal")) {
31            TamanoFuente=18;
32            AreaEdicion.setFont(new
33                Font("Serif",TipoFuente,TamanoFuente));
34        }
35        if (e.getActionCommand().equals("Grande")) {
36            TamanoFuente=23;
37            AreaEdicion.setFont(new
38                Font("Serif",TipoFuente,TamanoFuente));
```



```
38         return;
39     }
40     if (e.getActionCommand().equals("Muy grande")) {
41         TamanioFuente=30;
42         AreaEdicion.setFont(new
                               Font("Serif",TipoFuente,TamanioFuente));
43         return;
44     }
45     if (e.getActionCommand().equals("Todo mayúsculas")) {
46         AreaEdicion.setText(AreaEdicion.getText().
                               toUpperCase());
47         return;
48     }
49     if (e.getActionCommand().equals("Todo minúsculas")) {
50         AreaEdicion.setText(AreaEdicion.getText().
                               toLowerCase());
51         return;
52     }
53
54     } // actionPerformed
55
56     } // EventosAccion
```

## 9.2.9 Tratamiento de las opciones “Buscar” y “Reemplazar”

En el constructor de la clase *GUIEditor* se crearon 2 instancias de la clase *EventoBuscaReemplaza* y se “añadieron” a las cajas de texto *Buscar* y *Reemplazar*. Obsérvese como no existe ningún problema en utilizar varias clases que implementen un mismo “Listener” (en nuestra aplicación *EventosAccion* y *EventoBuscaReemplaza*).

En la línea 5 del método *actionPerformed* se consulta si el botón de radio *TextoCompleto* está activo (*getState*), si es así asignamos el valor 0 a *PosicionInicial* (de búsqueda) y el tamaño del texto escrito en el área a *PosicionFinal*, es decir se realiza la búsqueda en todo el texto; si *TextoCompleto* no está activo (línea 8), las posiciones de búsqueda inicial y final se obtienen consultando el comienzo (*getSelectionStart*) y final (*getSelectionEnd*) del área seleccionada en *AreaEdicion*.

Posteriormente, en la línea 13 obtenemos la longitud (*length*) del texto buscado (*TamanioTexto*) y en la línea 15 la selección que ha hecho el usuario (búsqueda o reemplazamiento).

El bucle situado en la línea 17 itera desde la posición inicial hasta la posición final (menos el tamaño del propio texto buscado); nos servirá para comparar, a partir de cada posición, el texto siguiente en *AreaEdicion* con el texto buscado. Esta comparación se realiza en la línea 18 (y 19) utilizando el método *equals* sobre los literales: texto de *Buscar* y porción de texto (*substring*) situado entre la posición actual y la actual más *TamanoTexto*.

Si encontramos una coincidencia realizamos la búsqueda/reemplazamiento (líneas 21 a 28) y terminamos el método (línea 30). Si la opción que ha elegido el usuario es “Busca” (línea 21) se selecciona el texto a partir de la posición (i) encontrada (línea 22). Si el usuario ha seleccionado reemplazar (línea 23), aislamos el texto anterior (línea 24) y posterior (línea 25) al que deseamos reemplazar y creamos el nuevo texto (línea 27).

```

1    private class EventoBuscaReemplaza implements
                                   ActionListener {
2
3    public void actionPerformed(ActionEvent e) {
4
5        if (TextoCompleto.getState()) {
6            PosicionInicial=0;
7            PosicionFinal=AreaEdicion.getText().length();
8        } else {
9            PosicionInicial=AreaEdicion.getSelectionStart();
10           PosicionFinal  =AreaEdicion.getSelectionEnd();
11        }
12
13        int TamanoTexto=Buscar.getText().length();
14
15        TextField Accion = (TextField) e.getSource();
16
17        for (int i=PosicionInicial;i<=(PosicionFinal-
                                   TamanoTexto);i++)
18            if
19            (AreaEdicion.getText().substring(i,i+TamanoTexto).
20             equals(Buscar.getText())){
21                if (Accion.getName().equals("Busca")) // Busqueda
22                    AreaEdicion.select(i,i+TamanoTexto);
23                else { //Reemplaza
24                    String Antes  =
25                        AreaEdicion.getText().substring(0,i);
26                    String Despues = AreaEdicion.getText().
27                        substring(i+TamanoTexto,
28                                AreaEdicion.getText().length());
29                    AreaEdicion.setText(Antes+Reemplazar.getText()
29                                        +Despues);

```

```
28         }
29
30         return;
31     }
32 }
33 } // EventoBuscaReemplaza
```

### 9.2.10 Tratamiento de los botones de radio “Color texto” y “Color fondo”

La clase *EventosElemento* (línea 1) implementa el único método del interfaz *ItemListener*: *itemStateChanged* (línea 3). Usando el método *getItem* (línea 4) perteneciente a la clase *ItemEvent*, podemos conocer el botón de radio que ha pulsado el usuario; si ha sido “Color texto” (línea 4) asignamos la constante *TEXTO* a la propiedad *TextoFondo* (línea 5), y si ha sido “Color fondo” hacemos lo propio en la línea 9.

```
1  private class EventosElemento implements ItemListener {
2
3      public void itemStateChanged(ItemEvent e) {
4          if (e.getItem().equals("Color texto")) {
5              TextoFondo=TEXTO;
6              return;
7          }
8          if (e.getItem().equals("Color fondo")) {
9              TextoFondo=FONDO;
10             return;
11         }
12     }
13 } // itemStateChanged
14 } // EventosElemento
```

### 9.2.11 Tratamiento a las pulsaciones de los botones de colores

La clase *EventosRaton* ha sido utilizada para crear todas las instancias de los tratamientos de eventos de ratón de los botones de colores. El único método redefinido del adaptador *MouseAdapter* (línea 1) ha sido *mouseClicked* (línea 3).

En primer lugar se obtiene el botón que ha generado el evento (línea 4). Si los botones de radio indican que hay que modificar el color de fondo del área de edición (línea 5) se realiza esta acción (línea 6); idéntico tratamiento se hace para modificar el color del texto (líneas 7 y 8).

Nótese que modificando la línea 5 por la sentencia: *if (ColorFondo.getState())* el comportamiento de la aplicación no varía y evitamos la implementación de la clase *EventosElemento*. Esta clase ha sido incluida únicamente para mostrar un uso habitual del interfaz *itemListener*.

```
1    private class EventosRaton extends MouseAdapter {
2
3        public void mouseClicked(MouseEvent e) {
4            Button Boton = (Button) e.getSource();
5            if (TextoFondo==FONDO)
6                AreaEdicion.setBackground(Boton.getBackground());
7            else // TEXTO
8                AreaEdicion.setForeground(Boton.getBackground());
9        } // mouseClicked
10
11
12    } // EventosRaton
```