

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

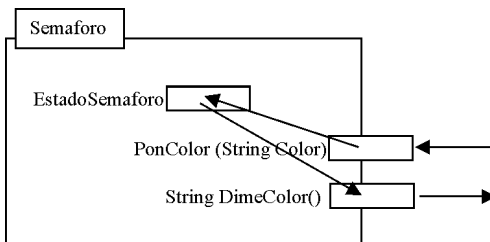
```
AtributoAcceso class NombreClase {  
    // propiedades y métodos  
}
```

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

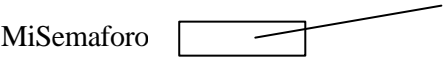
(instancias de la clase *Semáforo*), y hacer evolucionar el estado de estos “semáforos” de forma independiente.

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*:

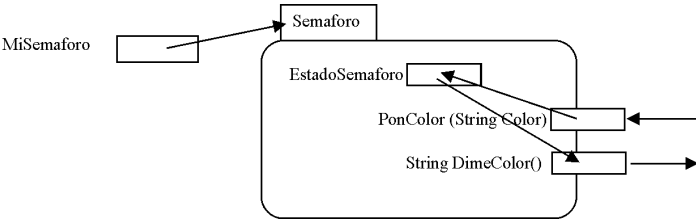


Para crear una instancia de la clase *Semaforo*, empleamos la palabra reservada *new*, tal y como hacíamos para crear una instancia de una matriz; después invocamos a un método que se llame igual que la clase. Estos métodos se denominan constructores y se explicarán un poco más adelante.

MiSemaforo = new Semaforo();

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

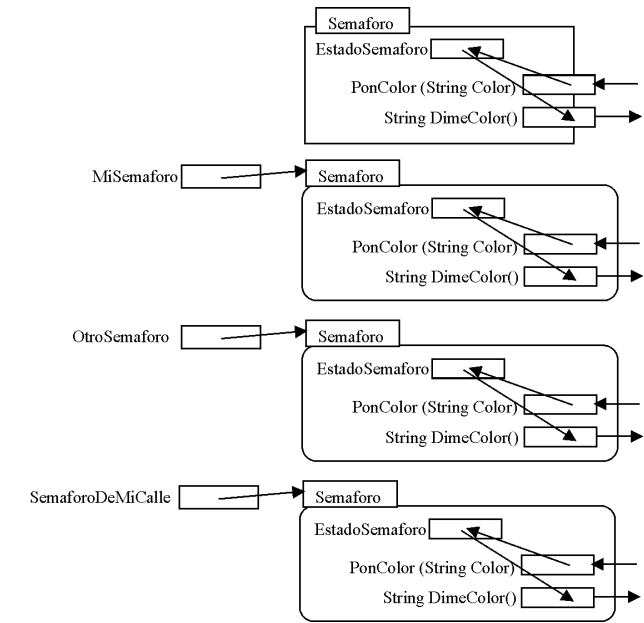
En cualquier caso, el resultado es que disponemos de una variable *MiSemaforo* que direcciona un objeto creado (instanciado) de la clase *Semaforo*:



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
```

```

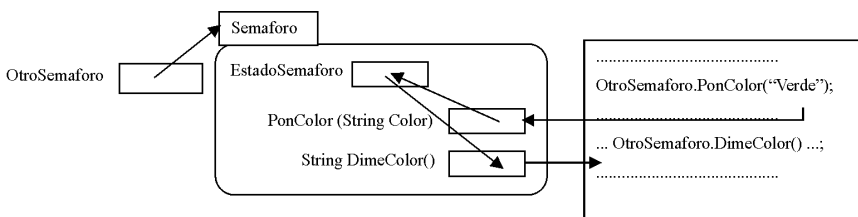
13 public class PruebaSemaforo {
14     public static void main (String[] args) {
15         Semaforo MiSemaforo = new Semaforo();
16         Semaforo SemaforoDeMiCalle = new Semaforo();
17         Semaforo OtroSemaforo = new Semaforo();
18
19         MiSemaforo.PonColor("Rojo");
20         OtroSemaforo.PonColor("Verde");
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 }

```

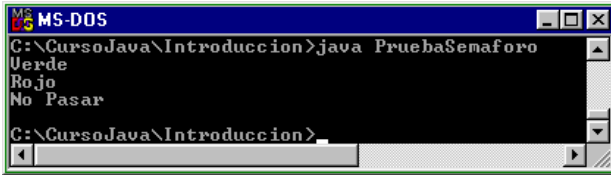
En las líneas 3, 4 y 5 de la clase *PruebaSemaforo* se declaran e instancian las variables *MiSemaforo*, *SemaforoDeMiCalle* y *OtroSemaforo*. En las líneas 7 y 8 se asignan los textos *Rojo* y *Verde* en las propiedades *EstadoSemaforo* de las instancias *MiSemaforo* y *OtroSemaforo*. La propiedad *EstadoSemaforo* 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*.

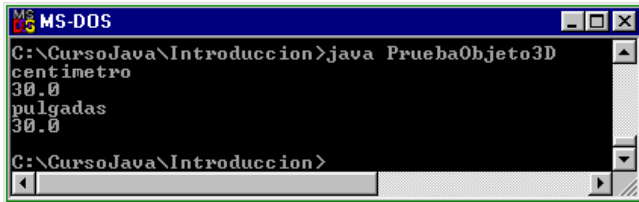
A continuación se muestra el código de una clase (*PruebaObjeto3D*) que utiliza a la clase *Objeto3D*:

```
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*.

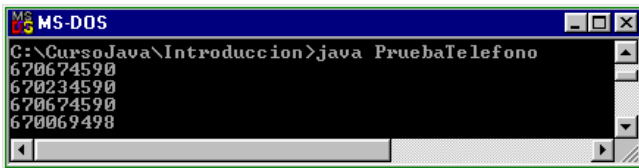
A partir de la línea 13 se hace uso del teléfono *ModeloMedio*, realizándose 6 llamadas y consultando en la línea 15 la última llamada y en la línea 20 la penúltima llamada.

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                            vengan 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                            vengan 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*.

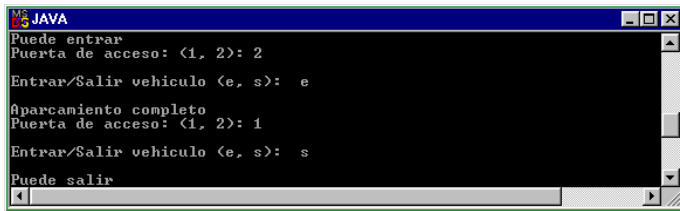
En la línea 10 nos introducimos en un bucle sin fin, donde pedimos a la persona que prueba esta simulación que introduzca el número de puerta del aparcamiento por donde desea entrar o salir un usuario. Para ello nos ayudamos del método *IntroduceCaracter* implementado a partir de la línea 42.

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.

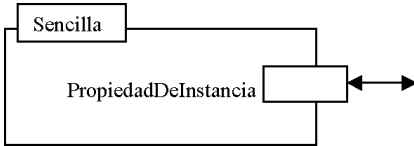
En este punto es importante resaltar el hecho de que hasta que no se crea una primera instancia de una clase, no existirá ninguna propiedad visible de la clase.

Gráficamente lo expuesto se puede representar como:

```

1 class Sencilla {
2     public int PropiedadDeInstancia;
3 }

```



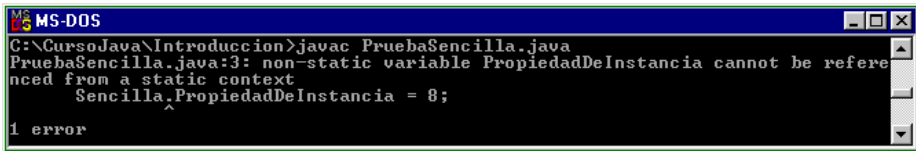
La clase *Sencilla* está definida, pero no instanciada, por lo que todavía no existe ninguna variable *PropiedadDeInstancia*. El gráfico nos muestra únicamente la estructura que tendrá una instancia de la clase *Sencilla* (cuando la definamos).

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 }

```

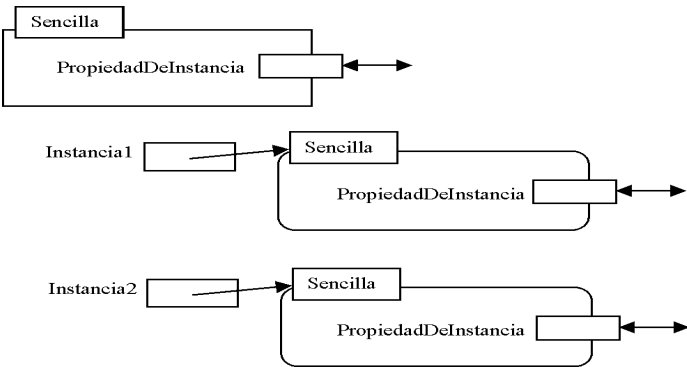


El compilador nos indica que la variable *PropiedadDeInstancia* es “no estática” y que existe un error. Para poder hacer uso de la variable *PropiedadDeInstancia*, obligatoriamente deberemos crear alguna instancia de la clase, tal y como hemos venido haciendo en las últimas lecciones:

```

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*

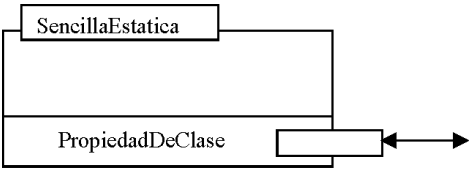
Todo intento de utilizar directamente la definición de la clase nos dará error:

~~*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 }

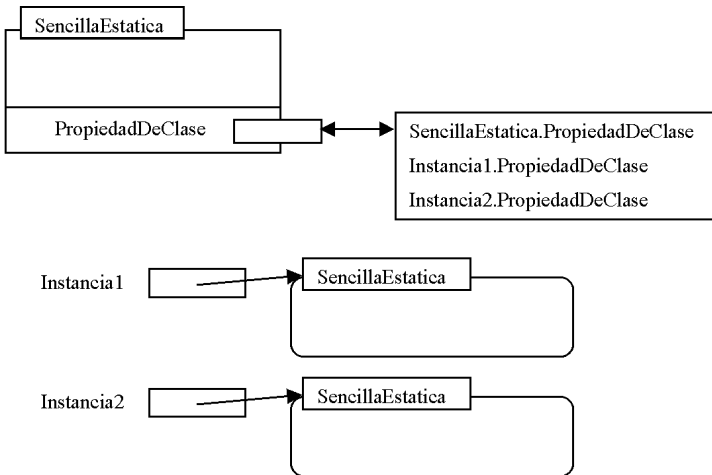
```

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

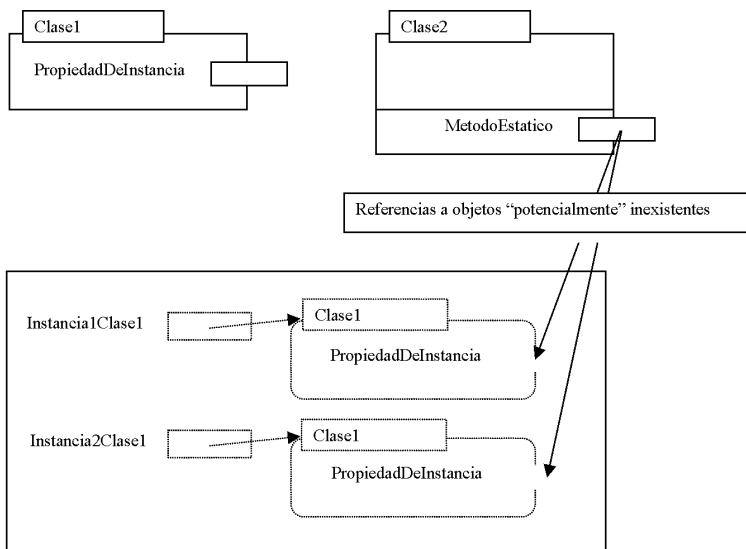
referenciarse a través del nombre de la clase (al igual que las propiedades estáticas). Esta posibilidad es útil en diversas circunstancias:

- 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.

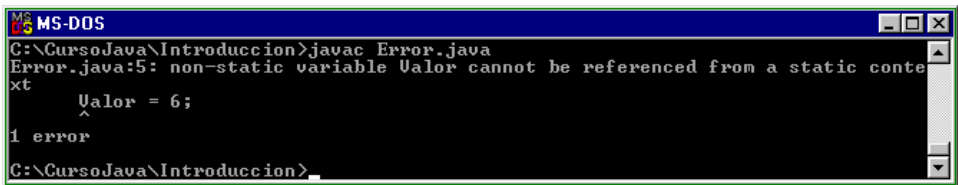
Los compiladores de Java comprueban estas situaciones y producen errores cuando detectan una referencia a un objeto no estático dentro de un método estático.

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



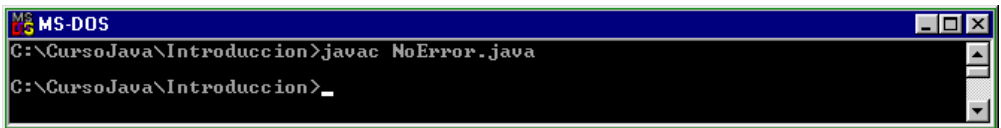
Una situación muy común cuando se empieza a programar es realizar una referencia a una propiedad no estática desde el método estático *main*. El siguiente ejemplo caracteriza este tipo de error tan extendido:

```
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.

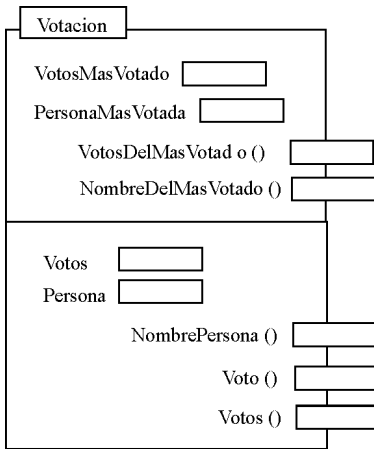
La solución que se aporta en este apartado no requiere del uso de una nueva clase “MasVotado” o similar, ni necesita un número fijo de instanciaciones para funcionar. La solución propuesta contiene las propiedades y métodos de acceso a la persona más votada dentro de la propia clase *Votacion*, en la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

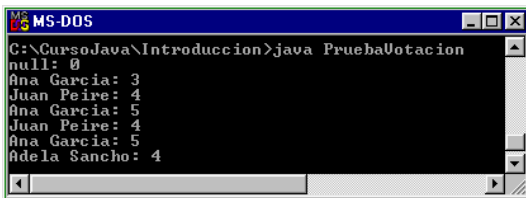
En las líneas 8, 9 y 10 damos “vida” a nuestros candidatos: Juan, Ana y Adela, creando instancias de la clase *Votacion*. En este momento, además de las propiedades ya existentes: *VotosMasVotado* y *PersonaMasVotada*, se dispone de tres copias de las propiedades de instancia: *Persona* y *Votos*.

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	private	private int PPrivada;	No	No
Sin especificar		int PSinEspecificar;	Sí	No
Protegido	protected	protected int PProtegida;	Sí	No
Publico	public	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()) {

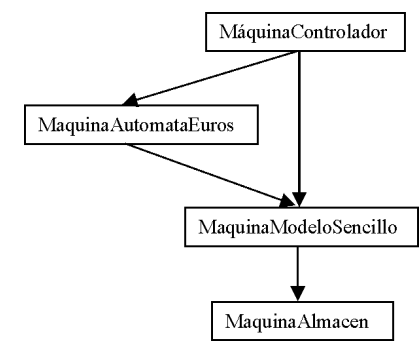
```



```
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 --
```