9. Objetos y clases

Objetivos:
a) Presentar el concepto de objeto, clase, atributo, método e instancia
b) Interpretar el código fuente de una aplicación Java donde aparecen implementados los conceptos anteriores
c) Construir una aplicación Java sencilla, convenientemente especificada, que emplee los conceptos anteriores.

Aunque parezca una obviedad, la base de la Programación Orientada a Objetos es el *objeto*. En la vida real todos los *objetos* tienen una serie de características y un comportamiento. Por ejemplo, una puerta tiene color, forma, dimensiones, material... (goza de una serie de características) y puede abrirse, cerrarse... (posee un comportamiento). En Programación Orientada a Objetos, un objeto es una combinación de unos datos específicos y de las rutinas que pueden operar con esos datos. De forma que los dos tipos de componentes de un objeto son:

a. Campos o *atributos*: componentes de un objeto que almacenan datos. También se les denomina *variables miembro*. Estos datos pueden ser de tipo primitivo (`boolean`, `int`, `double`, `char`...) o, a su vez, de otro tipo de objeto (lo que se denomina *agregación* o *composición* de objetos). La idea es que un atributo representa una propiedad determinada de un objeto.
b. Rutinas o *métodos*: es una componente de un objeto que lleva a cabo una determinada acción o tarea con los atributos.

En principio, todas las variables y rutinas de un programa de Java deben pertenecer a una clase. De hechom en Java no hay noción de programa principal y los subrutinas no existen como unidades modulares independientes, sino que forman siempre parte de alguna clase.

9.1. Clases

Una *clase* representa al conjunto de objetos que comparten una estructura y un comportamiento comunes. Una clase es una combinación específica de atributos y métodos y puede considerarse un tipo de dato de cualquier tipo **no** primitivo. Así, una clase es una especie de *plantilla* o *prototipo* de objetos: define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos. Aunque, por otro lado, una clase también puede estar compuesta por métodos estáticos que no necesitan de objetos (como las clases construidas en los capítulos anteriores que contienen un método estático `main`). La declaración de una clase sigue la siguiente sintaxis:

```
[modificadores] class IdentificadorClase {
    // Declaraciones de atributos y metodos
    . . .
}
```

Convención de los programadores en Java: Los identificadores de las clases deberían ser simples, descriptivos y sustantivos y, en el caso de nombres compuestos, con la primera letra de cada uno en mayúsculas. Es conveniente utilizar las palabras completas y evitar los acrónimos, a menos que la abreviatura sea mucho más utilizada que la forma no abreviada como en URL o HTML.

9.2. Instancias

Una instancia es un elemento tangible (ocupa memoria durante la ejecución del programa) generado a partir de una definición de clase. Todos los objetos empleados en un programa han de pertenecer a una clase determinada.

Aunque el término a veces se emplea de una forma imprecisa, un objeto es una *instancia* de una clase predefinida en Java o declarada por el usuario y *referenciada* por una variable que almacena su dirección de memoria. Cuando se dice que *Java no tiene punteros* simplemente se indica que Java no tiene punteros que el programador pueda *ver*, ya que todas las referencias a objeto son de hecho punteros en la representación interna.

En general, el *acceso* a los atributos se realiza a través del operador punto, que separa al identificador de la referencia del identificador del atributo (*`idReferencia.idAtributo`*). Las *llamadas* a los métodos para realizar las distintas acciones se llevan a cabo separando los identificadores de la referencia y del método correspondiente con el operador punto (*`idReferencia.idMetodo(parametros)`*).

Ejemplo sencillo de clase y de instancia

El siguiente código muestra la declaración de la clase `Precio`. La clase `Precio` consta de un único atributo (`euro`) y dos métodos: uno que asigna un valor al atributo (`pone`) sin devolver ningún valor y otro que devuelve el valor del atributo (`da`).

```
/**
 *  Ejemplo de declaracion de la clase Precio
 *  double da()              --> devuelve el valor almacenado en euros
 *  void pone( double x )    --> almacena valor en euros
 *  A. Garcia-Beltran - octubre, 2005
 */
public class Precio {
    // Atributo o variable miembro
    public double euros;
    // Metodos
    public double da() { return euros; }
    public void pone(double x) { euros=x; }
}
```

Gráficamente, una clase puede representarse como un rectángulo según muestra la Figura 9.1.

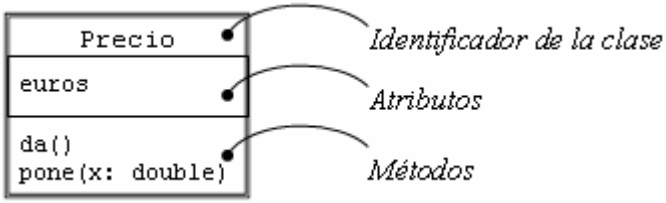

Figura 9.1. Representación gráfica de la clase `Precio`

El anterior código puede compilarse...

```
$>javac Precio.java↵
```

generando el archivo de *bytecodes* `Precio.class`. Este archivo no es directamente ejecutable por el intérprete, ya que el código fuente no incluye ningún método principal (`main`). Para poder probar el código anterior, puede construirse otro archivo con el código fuente que se muestra a continuación:

```
/**
 *  Ejemplo de uso de la clase Precio
 *  A. Garcia-Beltran - abril, 2005
 */
public class PruebaPrecio {
    public static void main (String [] args ) {
        Precio p;                       // Crea una referencia de la clase Precio
        p = new Precio();               // Crea el objeto de la clase Precio
        p.pone(56.8);                   // Llamada al metodo pone
                                        // que asigna 56.8 al atributo euros
        System.out.println("Valor = " + p.da());    // Llamada al metodo da
                                        // que devuelve el valor de euros
        Precio q = new Precio();        // Crea una referencia y el objeto
        q.euros=75.6;                   // Asigna 75.6 al atributo euros
        System.out.println("Valor = " + q.euros);
    }
}
```

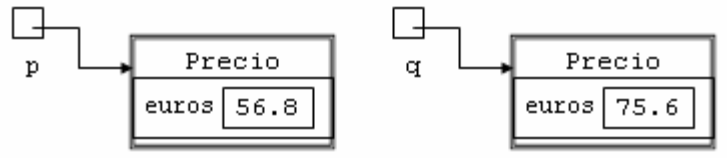

Figura 9.2. Representación gráfica del espacio de la memoria utilizado por las referencias e instancias de la clase `Precio` durante la ejecución del método `main` de la clase `PruebaPrecio`

El código anterior puede compilarse y ejecutarse, mostrando la siguiente salida por pantalla:

```
$>javac PruebaPrecio.java↵          Compilacion

$>java PruebaPrecio↵                Ejecución
Valor = 56.8                        Salida por pantalla
Valor = 75.6                        Salida por pantalla
```

Explicación del ejemplo anterior

Para poder trabajar con objetos se tendrá que seguir un proceso de dos pasos. Lo primero que debe hacer el programa es crear una referencia o puntero de la clase `Precio` con el identificador `p` (Figura 9.3). De forma similar a cómo se declara una variable de un tipo primitivo, la declaración del identificador de la referencia se realiza con la sintaxis:

```
identificadorClase identificadorReferencia;
// En el ejemplo anterior: Precio p;
```

Figura 9.3. Creación de la referencia `p`

La referencia o puntero, `p`, tiene como misión almacenar la dirección de memoria de (*apuntar a*) los componentes de la instancia que todavía no ha sido creada ni referenciada. En este momento se dice que la referencia `p`, recien creada, almacena una dirección de memoria *nula* (que no corresponde a objeto alguno) o `null`. El segundo paso del proceso para trabajar con objetos lleva a la creación de una nueva instancia referenciada por `p` (Figura 9.4), que se realiza con la sentencia:

```
identificadorReferencia = new identificadorClase();
// Ejemplo: p = new Precio();
```

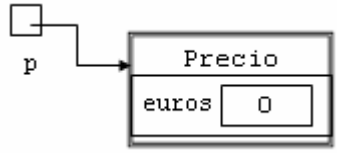

Figura 9.4. Creación de la nueva instancia de la clase `Precio` referenciado por `p`

A esta operación se le denomina también ***instanciación***. Aunque las dos operaciones anteriores (creación de la referencia y creación de la instancia referenciada) pueden realizarse conjuntamente (Figura 9.5) en la misma línea de código:

```
identificadorClase identificadorReferencia = new identificadorClase();
// En el codigo del ejemplo anterior:        Precio q = new Precio();
```

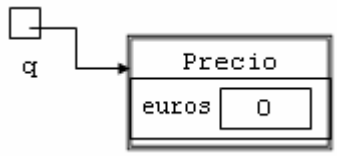

Figura 9.5. Creación de la referencia `q` y de la nueva instancia de la clase `Precio` referenciado por `q`

El resultado de la ejecución del código anterior son dos nuevas instancias de la clase `Precio` referenciados respectivamente por `p` y `q` (Figura 9.2). El atributo `euros` de cada una de las nuevas instancias de la clase `Precio` es accesible a través del identificador de la referencia y del operador punto (`p.euros` y `q.euros`). Los métodos `da` y `pone` pertenecientes a la clase `Precio` son accesibles a través del identificador de la referencia y del operador punto (`p.da()` y `p.pone(56.8)` y `q.da()` y `q.pone(75.6)`, respectivamente). En el caso de los métodos, la instancia mediante la cual se realiza la llamada correspondiente actúa como un parámetro o argumento implícito del método.

Si se asigna una referencia a otra mediante una sentencia de asignación, no se copian los valores de los atributos, sino que se tiene como resultado una única instancia apuntada por dos referencias distintas (Figura 9.6). Por ejemplo:

```
q = p;                    // Ahora p y q referencian al mismo objeto
```

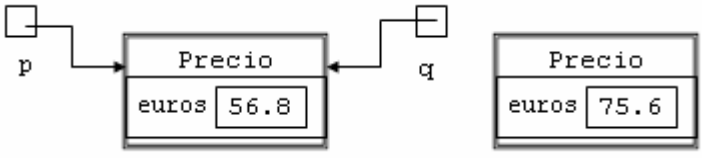

Figura 9.6. Resultado de la asignación de valores entre referencias

En este caso ¿qué ocurre con la instancia referenciada previamente por `q`? Dicha instancia se queda sin referencia (inaccesible). Esto puede ser un problema en algunos lenguajes de programación, como es el caso de Pascal o de C, que utilizan variables *dinámicas* y que necesitan liberar explícitamente el espacio en memoria reservado para las variables que van a dejar de ser referenciadas. La gestión *dinámica* de la memoria suele ser una tarea engorrosa para el programador y muy dada a la proliferación de errores de ejecución. Para evitar tales inconvenientes, Java permite crear tantas instancias como se desee (con la única limitación de la memoria que sea capaz de gestionar el sistema), sin que el programador tenga que preocuparse de destruirlas o liberarlas cuando ya no se necesiten. El entorno de ejecución de Java elimina automáticamente las instancias cuando detecta que no se van a usar más (cuando dejan de estar referenciadas). A este proceso se le denomina recogida o recolección de basura (*garbage collection*).

9.3. Modificadores de visibilidad

El modificador `public` indica que la componente del método es accesible fuera del código de la clase a la que pertenece la componente a través del operador punto. El modificador `private` indica que la componente solamente es accesible a través de los métodos de la propia clase. El modificador `protected` se verá posteriormente. En el siguiente código se declara el atributo `euros` con el modificador `private`.

```
/**
 *  Ejemplo de declaracion de la clase PrecioPrivado
 *  double da()              --> devuelve el valor almacenado en euros
 *  void pone( double x )    --> almacena valor en euros
 *  euros                    --> Atributo de acceso privado
 *  A. Garcia-Beltran - octubre, 2005
 */

public class PrecioPrivado {

  // Atributo o variable miembro
  private double euros;

  // Metodos publicos
  public double da() { return euros; }
  public void pone(double x) { euros=x; }
}
```

Si se construye otro código que intente utilizar directamente el atributo `euros`:

```
/**
 *  Ejemplo de uso de la clase PrecioPrivado
 *  double da()              --> devuelve el valor almacenado en euros
 *  void pone( double x )    --> almacena valor en euros
 *  euros                    --> Atributo de acceso privado
 *  A. Garcia-Beltran - octubre, 2005
 */

public class PruebaPrecioPrivado {
  public static void main (String [] args ) {
    precioPrivado p = new precioPrivado();    // Crea instancia
    p.pone(56.8);                             // Asigna 56.8 a euros
    System.out.println("Valor = " + p.da());
    p.euros=75.6;                             // Asigna 75.6 a euros - ERROR
    System.out.println("Valor = " + p.euros);              // Tambien ERROR
  }
}
```

se producirá un error de compilación:

```
$>javac PruebaPrecioPrivado.java↵                    Compilacion

pruebaPrecioPrivado.java:15: euros has private access in precioPrivado
    p.euros=75.6;
     ^
pruebaPrecioPrivado.java:16: euros has private access in precioPrivado
    System.out.println("Valor = " + p.euros);
                                    ^
```

ya que el atributo `euros` sólo es accesible a través de los métodos de la clase `da` y `pone`.

La utilización del modificador `private` sirve para implementar una de las características de la programación orientada a objetos: el ocultamiento de la información o **encapsulación**. Estrictamente hablando, la declaración como público de un atributo de una clase no respeta este principio de ocultación de información. Declarándolos como privados, no se tiene acceso directo a los atributos del objeto fuera del código de la clase correspondiente y sólo puede accederse a ellos de forma indirecta a través de los métodos proporcionados por la propia clase. Una de las ventajas prácticas de obligar al empleo de un método para modificar el valor de un atributo es asegurar la consistencia de la operación. Por ejemplo, un método que asigne valor al atributo `euros` de un objeto de la clase `Precio` puede garantizar que no se le asignará un valor negativo.

9.4. Clases anidadas e internas

Una clase B se puede definir como miembro de otra clase A. La estructura sintáctica es la siguiente:

```
class ClaseA {
    ...
    class ClaseB {
        ...
    }
}
```

Se dice que `ClaseB` es una clase anidada en la `ClaseA`. La clase anidada sólo puede emplearse dentro de la clase *contenedora*. Este tipo de clases sólo se construyen cuando la clase anidada sólo se emplea o tiene sentido dentro de la clase contenedora. La clase anidada puede declararse como `static`. En este caso la clase anidada se denomina clase *anidada estática*. En caso contrario se denomina clase *interna*.

9.5. El operador instanceof

El operador `instanceof` devuelve verdadero o falso si un objeto pertenece o no a una clase determinada. Sintaxis:

```
identificadorInstancia instanceof identificadorClase
```

9.6. Javadoc

Javadoc es un programa proporcionado en el Kit de Desarrollo de Java que permite generar automáticamente documentación para las clases. El resultado de javadoc es un conjunto de documentos en formato HTML que puede ser visualizados con un navegador. El archivo fuente de

java puede añadir comentarios incluidos entre las secuencias de caracteres `/**` ... `*/` a la documentación generada con javadoc. Existe también un conjunto de datos específicos que pueden incluirse en los comentarios de tipo javadoc. Por ejemplo: `@author`, `@param`, `@return` y `@exception`. `@author` debe preceder al código de la declaración de la clase mientras que `@param` y `@return` deben preceder a la declaración del método como muestra el siguiente código:

```
// Ejemplo de declaracion de una clase para documentar con javadoc
/**
 * Clase para trabajar con una variable que almacena un precio
 * @author Angel Garcia Beltran
 * Ultima version: octubre, 2005
 */
public class Precio2
{
  // Variable miembro
  public double euros;

  /**
   * Devuelve el valor almacenado en euros
   * @return el valor almacenado
   */
  public double da() { return euros; }

  /**
   * Almacena un valor
   * @param x el valor a almacenar
   */
  public void pone(double x) { euros=x; }
}
```

La ejecución del programa javadoc se realiza de la siguiente manera:

```
$>javadoc Precio2.java↵

Loading source file Precio2.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating index.html...
Generating packages.html...
Generating precio2.html...
Generating serialized-form.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
```

La salida de javadoc está formada fundamentalmente por los comentarios incluidos en el código fuente. El compilador no comprueba que dichos comentarios estén implementados. En la Figura 9.7 se muestra uno de los documentos HTML generados por javadoc.

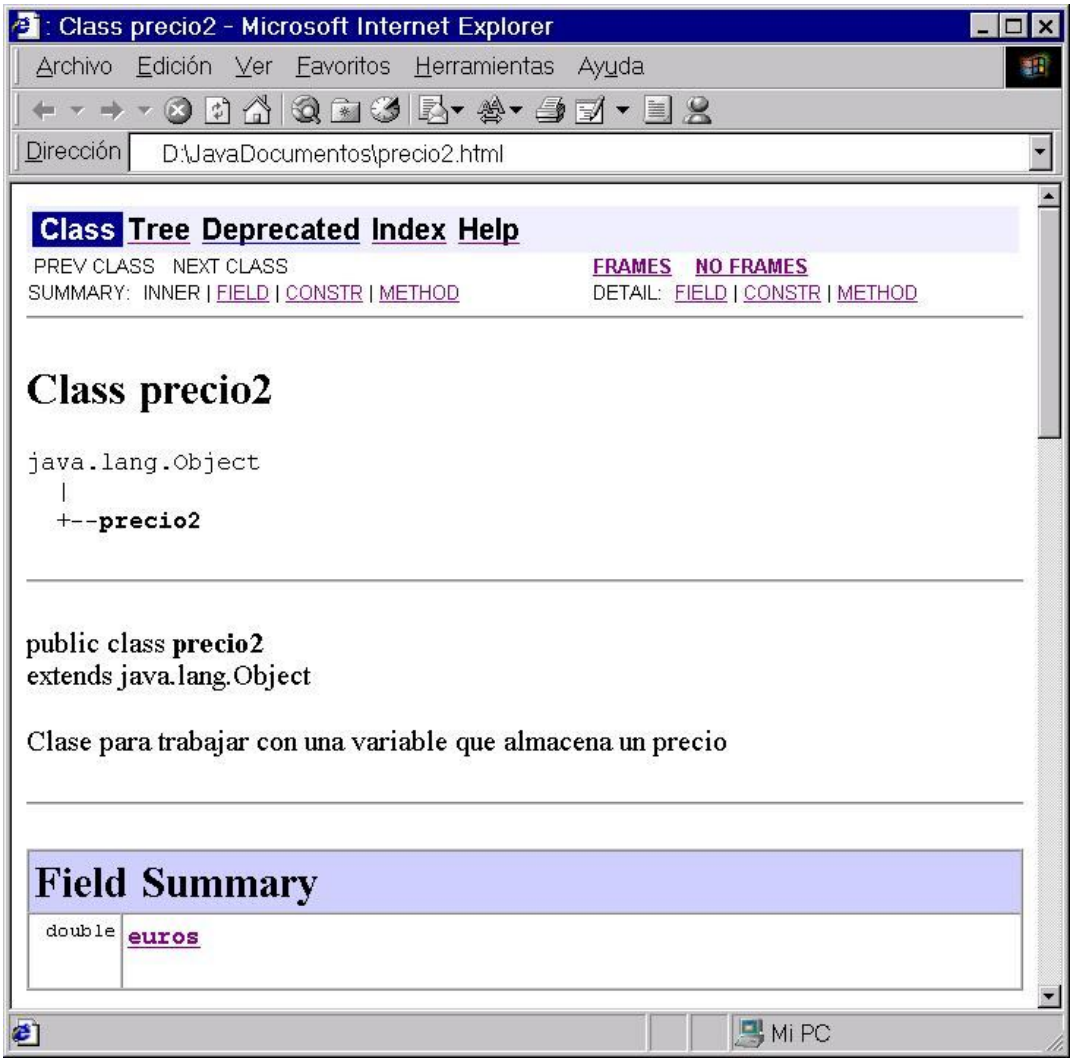

Figura 9.7. Visualización de uno de los documentos generados por Javadoc