CAPÍTULO 4

ESTRUCTURA DE UN PROGRAMA

En este capítulo estudiará cómo es la estructura de una programa Java. Partiendo de un programa ejemplo sencillo analizaremos cada una de las partes que componen su estructura, así tendrá un modelo para realizar sus propios programas. También veremos cómo se construye un programa a partir de varios módulos de clase. Por último, estudiaremos los conceptos de ámbito y accesibilidad de las variables.

ESTRUCTURA DE UNA APLICACIÓN JAVA

Puesto que Java es un lenguaje orientado a objetos, un programa Java se compone solamente de objetos. Recuerde que un objeto es la concreción de una clase, y que una clase equivale a la generalización de un tipo específico de objetos. La clase define los atributos del objeto así como los métodos para manipularlos. Muchas de las clases que utilizaremos pertenecen a la biblioteca de Java, por lo tanto ya están escritas y compiladas. Pero otras tendremos que escribirlas nosotros mismos, dependiendo del problema que tratemos de resolver en cada caso.

Toda aplicación Java está formada por al menos una clase que define un método nombrado **main,** como se muestra a continuación:

```
public class CMiAplicacion
{
  public static void main(String[] args)
  {
    // escriba aquí el código que quiere ejecutar
  }
}
```

Una clase que contiene un método **main** es una plantilla para crear lo que vamos a denominar objeto aplicación, objeto que tiene como misión iniciar y fi-

nalizar la ejecución de la aplicación. Precisamente, el método **main** es el punto de entrada y de salida de la aplicación.

Según lo expuesto, la solución de cualquier problema no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de objetos naturales del problema mismo, abstraídos de alguna manera, que darán lugar a los objetos que intervendrán en la solución del programa. El empleo de este modelo de desarrollo de programas, nos conduce al diseño y programación orientada a objetos, modelo que ha sido empleado para desarrollar todos los ejemplos de este libro.

Para explicar cómo es la estructura de un programa Java, vamos a plantear un ejemplo sencillo de un programa que presente una tabla de equivalencia entre grados centígrados y grados *fahrenheit*, como indica la figura siguiente:

```
-30 C     -22.00 F
-24 C     -11.20 F
  .
  .
  .
 90 C     194.00 F
 96 C     204.80 F
```

La relación entre los grados centígrados y los grados *fahrenheit* viene dada por la expresión *grados fahrenheit = 9/5 * grados centígrados + 32*. Los cálculos los vamos a realizar para un intervalo de –30 a 100 grados centígrados con incrementos de 6.

Analicemos el problema. ¿De qué trata el programa? De grados. Entonces podemos pensar en objetos "grados" que encapsulen un valor en grados centígrados y los métodos necesarios para asignar al objeto un valor en grados centígrados, así como para obtener tanto el dato grados centígrados como su equivalente en grados *fahrenheit*. En base a esto, podríamos escribir una clase *CGrados* como se puede observar a continuación:

```
class CGrados
{
  private float gradosC; // grados centígrados

  public void CentígradosAsignar(float gC)
  {
    // Establecer el atributo grados centígrados
    gradosC = gC;
  }
```

```
  public float FahrenheitObtener()
  {
    // Retornar los grados fahrenheit equivalentes a gradosC
    return 9F/5F * gradosC + 32;
  }

  public float CentígradosObtener()
  {
    return gradosC; // retornar los grados centígrados
  }
}
```

El código anterior muestra que un objeto de la clase *CGrados* tendrá una estructura interna formada por el atributo:

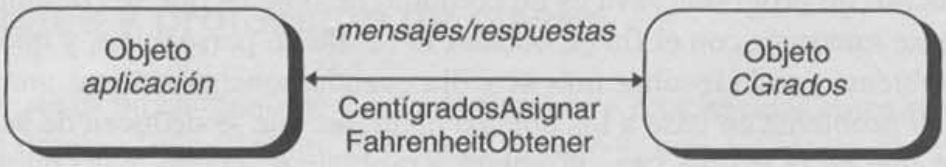
- *gradosC*, grados centígrados,

y una interfaz de acceso formada por los métodos:

- *CentígradosAsignar* que permite asignar a un objeto un valor en grados centígrados.
- *FahrenheitObtener* que permite retornar el valor grados *fahrenheit* equivalente a *gradosC* grados centígrados.
- *CentígradosObtener* que permite retornar el valor almacenado en el atributo *gradosC*.

Sin casi darnos cuenta estamos abstrayendo (separando por medio de una operación intelectual) los elementos naturales que intervienen en el problema a resolver y construyendo objetos que los representan.

Recordando lo visto anteriormente, una aplicación Java tiene que tener un objeto aplicación, que aporte un método **main**, por donde empezará y terminará la ejecución de la aplicación, además de otros que consideramos necesarios. ¿Cómo podemos imaginar esto de una forma gráfica? La figura siguiente da respuesta a esta pregunta:



Entonces, ¿qué tiene que hacer el objeto aplicación? Pues, visualizar cuántos grados *fahrenheit* son –30 C, –24 C, ..., *n* grados centígrados, ..., 96 C. Y, ¿cómo hace esto? Enviando al objeto *CGrados* los mensajes *CentígradosAsignar* y *FahrenheitObtener* una vez para cada valor desde –30 a 100 grados centígrados con

incrementos de 6. El objeto *CGrados* responderá ejecutando los métodos vinculados con los mensajes que recibe. Según esto, el código de la clase que dará lugar al objeto aplicación puede ser el siguiente:

```
import java.lang.System; // importar la clase System

public class CApGrados
{
  // Definición de constantes
  final static int limInferior = -30;
  final static int limSuperior = 100;
  final static int incremento = 6;

  public static void main(String[] args)
  {
    // Declaración de variables
    CGrados grados = new CGrados();
    int gradosCent = limInferior;
    float gradosFahr = 0;

    while (gradosCent <= limSuperior) // mientras ... hacer:
    {
      // Asignar al objeto grados el valor en grados centígrados
      grados.CentigradosAsignar(gradosCent);
      // Obtener del objeto grados los grados fahrenheit
      gradosFahr = grados.FahrenheitObtener();
      // Escribir la siguiente línea de la tabla
      System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
      // Siguiente valor
      gradosCent += incremento;
    }
  }
}
```

Seguro que pensará que todo el proceso se podría haber hecho utilizando solamente el objeto aplicación, escribiendo todo el código en el método **main**, lo cual es cierto. Pero, lo que se pretende es que pueda ver de una forma clara que, en general, un programa Java es un conjunto de objetos que se comunican entre sí mediante mensajes con el fin de obtener el resultado perseguido, y que la solución del problema puede resultar más sencilla cuando consiga realizar una representación del problema en base a los objetos naturales que se deducen de su enunciado. Piense que en la realidad se enfrentará a problemas mucho más complejos y, por lo tanto, la descomposición en objetos será vital para resolverlos.

Una vez analizado el problema, cree una nueva aplicación desde su entorno de desarrollo y escriba la clase *CGrados*; observe que no es pública. A continuación, escriba la clase aplicación *CApGrados* en el mismo fichero fuente; observe

que es pública. Después, guarde la aplicación que ha escrito en un fichero utilizando como nombre el de la clase aplicación; esto es, *CApGrados.java*. Finalmente, compile y ejecute la aplicación.

No se preocupe si no entiende todo el código. Ahora lo que importa es que aprenda cómo es la estructura de un programa, no por qué se escriben unas u otras sentencias, cuestión que aprenderá más tarde en éste y en sucesivos capítulos. Este ejemplo le servirá como plantilla para inicialmente escribir sus propios programas. Posiblemente su primera aplicación utilice solamente un objeto aplicación, pero con este ejemplo tendrá un concepto más real de lo que es una aplicación Java.

En el ejemplo realizado podemos observar que una aplicación Java consta de:

- Sentencias **import** (para establecer vínculos con otras clases de la biblioteca Java o realizadas por nosotros).
- Una clase aplicación pública (la que incluye el método **main**).
- Otras clases no públicas.

Sabemos también que una clase encapsula los atributos de los objetos que describe y los métodos para manipularlos. Pues bien, cada método consta de:

- Definiciones y/o declaraciones.
- Sentencias a ejecutar.

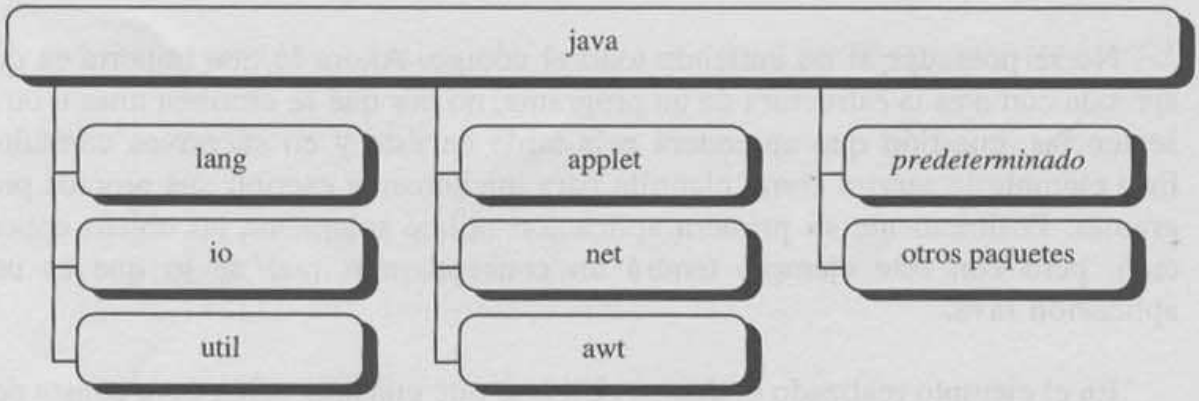
En un fichero se pueden incluir tantas definiciones de clase como se desee, pero sólo una de ellas puede ser declarada como pública (**public**). Recuerde que cada clase pública debe ser guardada en un fichero con su mismo nombre y extensión *.java*.

Los apartados que se exponen a continuación explican brevemente cada uno de estos componentes que aparecen en la estructura de un programa Java.

Paquetes y protección de clases

Un paquete es un conjunto de clases, lógicamente relacionadas entre sí, agrupadas bajo un nombre (por ejemplo, el paquete **java.io** agrupa las clases que permiten a un programa realizar la entrada y salida de información); incluso, un paquete puede contener a otros paquetes. Análogamente a como las carpetas o directorios ayudan a organizar los ficheros en un disco duro, los paquetes ayudan a organizar las clases en grupos para facilitar el acceso a las mismas cuando las necesitemos en un programa. Aprenderá a crear paquetes más adelante, ahora es suficiente con que aprenda a utilizar los paquetes de la biblioteca de Java.

La propia biblioteca de clases de Java está organizada en paquetes dispuestos jerárquicamente. En la figura siguiente se muestran algunos de ellos:



El nivel superior se denomina **java**. En el siguiente nivel tenemos paquetes como **lang**, **applet** o **io**.

Para referirnos a una *clase* de un paquete, tenemos que hacerlo utilizando su nombre completo, excepto cuando el paquete haya sido importado implícita o explícitamente, como veremos a continuación. Por ejemplo, **java.lang.System** hace referencia a la *clase* **System** del paquete **java.lang** ("java.lang" es el nombre completo del paquete **lang**).

Las clases que guardamos en un fichero cuando escribimos un programa, pertenecen al paquete *predeterminado* sin nombre. Por ejemplo, las clases *CGrados* y *CApGrados* de la aplicación anterior pertenecen, por omisión, a este paquete. De esta forma Java asegura que toda clase pertenece a un paquete.

Protección de una clase

La protección de una clase determina la relación que tiene con otras clases de otros paquetes. Distinguimos dos niveles de protección: de *paquete* y *público*. Una clase con nivel de protección de *paquete* sólo puede ser utilizada por las clases de su paquete (no está disponible para otros paquetes, ni siquiera para los subpaquetes). En cambio, una clase pública puede ser utilizada por cualquier otra clase de otro paquete. ¿Qué se entiende por utilizar? Que una clase puede crear objetos de otra clase y manipularlos utilizando sus métodos.

Por omisión una clase tiene el nivel de protección de *paquete*; por ejemplo, la clase *CGrados* del ejemplo anterior tiene este nivel de protección. En cambio, cuando se desea que una clase tenga protección *pública*, hay que calificarla como tal utilizando la palabra reservada **public**; la clase *CApGrados* del ejemplo anterior tiene este nivel de protección. Otro ejemplo: echando un vistazo a la docu-

mentación de Java, se puede observar que la clase **System** del paquete **java.lang** es pública, razón por la cual se ha podido utilizar en la aplicación *CApGrados*.

Sentencia import

Una clase de un determinado paquete puede hacer uso de otra clase de otro paquete de dos formas:

1. Utilizando su nombre completo en todas las partes del código donde haya que referirse a ella. Por ejemplo:

```
java.lang.System.out.println(gradosFahr);
```

2. Importando la clase, como se indica en el párrafo siguiente, lo que posibilita referirse a ella simplemente por su nombre. Por ejemplo:

```
System.out.println(gradosFahr);
```

Para importar una clase de un paquete desde un programa utilizaremos la sentencia **import**. En un programa Java puede aparecer cualquier número de sentencias **import**, las cuales deben escribirse antes de cualquier definición de clase. Por ejemplo:

```
import java.lang.System; // importar la clase System

public class CApGrados
{
  // ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
  // ...
}
```

Como se puede comprobar en el ejemplo anterior, importar una clase permite al programa referirse a ella más tarde sin utilizar el nombre del paquete. Esto es, la sentencia **import** sólo indica al compilador e intérprete de Java dónde encontrar las clases, no trae nada dentro del programa Java actual.

En el caso concreto del ejemplo expuesto, si eliminamos la sentencia **import**, todo seguirá funcionando igual. Esto es así porque las clases del paquete **java.lang** son importadas de manera automática para todos los programas, no sucediendo lo mismo con el resto de los paquetes, que tienen que ser importados explícitamente.

```
public class CApGrados
{
  // ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
  // ...
}
```

También puede importar un paquete completo de clases utilizando como comodín un asterisco en lugar del nombre específico de una clase. Por ejemplo:

```
import java.lang.*; // importar las clases públicas de este paquete
                    // a las que se refiera el código

public class CApGrados
{
  // ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
  // ...
}
```

En realidad, para ser exactos, la sentencia **import** del ejemplo anterior importa todas las clases *públicas* del paquete **java.lang** que realmente se usen en el código del programa.

Definiciones y declaraciones

Una declaración introduce uno o más identificadores en un programa. Una declaración es una definición, a menos que no haya asignación de memoria.

Toda variable debe ser definida antes de ser utilizada. La definición de una variable, declara la variable y además le asigna memoria:

```
int gradosCent;
float gradosFahr;

gradosCent = limInferior;
gradosFahr = 0;
```

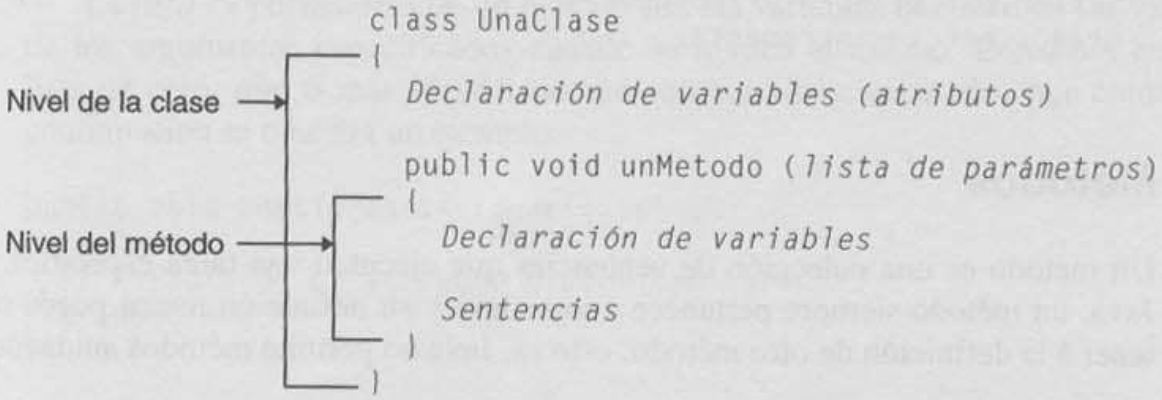
Además, una variable puede ser iniciada en la propia definición:

```
int gradosCent = limInferior;
float gradosFahr = 0;
```

La definición de un método, declara el método y además incluye el cuerpo del mismo. En cambio, la declaración de un método se corresponde con la cabecera de dicho método (su aplicación podrá verla en clases abstractas e interfaces).

```
public float FahrenheitObtener()
{
  // Retornar los grados fahrenheit equivalentes a gradosC
  return 9F/5F * gradosC + 32;
}
```

La declaración o la definición de una variable pueden realizarse a *nivel de la clase* (atributos de la clase) o a *nivel del método* (dentro de la definición de un método). Pero, la definición de un método, siempre ocurre a nivel de la clase.



En un método, las definiciones o declaraciones se pueden realizar en cualquier lugar; o mejor dicho, en el lugar justo donde se necesiten y no necesariamente al principio del método, antes de todas las sentencias.

Sentencia simple

Una *sentencia simple* es la unidad ejecutable más pequeña de un programa Java. Las sentencias controlan el flujo u orden de ejecución. Una sentencia Java puede formarse a partir de: una palabra clave (**for**, **while**, **if ... else**, etc.), expresiones, declaraciones o llamadas a métodos. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un punto y coma (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas una de otra por un punto y coma, aunque esta forma de proceder no es aconsejable porque va en contra de la claridad que se necesita cuando se lee el código de un programa.
- Una sentencia nula consta solamente de un punto y coma. Cuando veamos la sentencia **while**, podrá ver su utilización.

Sentencia compuesta o bloque

Una *sentencia compuesta* o bloque, es una colección de sentencias simples incluidas entre llaves - { } -. Un bloque puede contener a otros bloques. Un ejemplo de una sentencia de este tipo es el siguiente:

```
{
  grados.CentigradosAsignar(gradosCent);
  gradosFahr = grados.FahrenheitObtener();
  System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
  gradosCent += incremento;
}
```

Métodos

Un método es una colección de sentencias que ejecutan una tarea específica. En Java, un método siempre pertenece a una clase y su definición nunca puede contener a la definición de otro método; esto es, Java no permite métodos anidados.

Definición de un método

La definición de un método consta de una *cabecera* y del *cuerpo* del método encerrado entre llaves. La sintaxis para escribir un método es la siguiente:

```
[modificador] tipo-resultado nombre-método ([lista de parámetros])
{
  declaraciones de variables locales;
  sentencias;
  [return [(]expresión[)]];
}
```

Las variables declaradas en el cuerpo del método son locales a dicho método y por definición solamente son accesibles dentro del mismo.

Un *modificador* es una palabra clave que modifica el nivel de protección predeterminado del método. Véase el apartado "Protección de los miembros de una clase" expuesto un poco más adelante.

El *tipo del resultado* especifica qué tipo de valor retorna el método. Éste, puede ser cualquier tipo primitivo o referenciado. Para indicar que no se devuelve nada, se utiliza la palabra reservada **void**. El resultado de un método es devuelto a la sentencia que lo invocó, por medio de la siguiente sentencia:

```
return [(]expresión[)];
```

La sentencia **return** puede ser o no la última y puede aparecer más de una vez en el cuerpo del método. En el caso de que el método no retorne un valor (**void**), se puede omitir o especificar simplemente **return**. Por ejemplo:

```
void mEscribir()
{
  // ...
  return;
}
```

La *lista de parámetros* de un método son las variables que reciben los valores de los argumentos especificados cuando se invoca al mismo. Consisten en una lista de cero, uno o más identificadores con sus tipos, separados por comas. A continuación se muestra un ejemplo:

```
public void CentígradosAsignar(float gC)
{
  // Establecer el atributo grados centígrados
  gradosC = gC;
}
```

Método main

Toda aplicación Java tiene un método denominado **main**, y sólo uno. Este método es el punto de entrada a la aplicación y también el punto de salida. Su definición es como se muestra a continuación:

```
public static void main(String[] args)
{
  // Cuerpo del método
}
```

Como se puede observar, el método **main** es público (**public**), estático (**static**), no devuelve nada (**void**) y tiene un argumento de tipo **String** que almacenará los argumentos pasados en la línea de órdenes cuando se invoca a la aplicación para su ejecución, concepto que estudiaremos posteriormente en otro capítulo. Para más detalles, puede ver un poco más adelante los apartados "Protección de los miembros de una clase" y "Miembro de un objeto o de una clase".

Crear objetos de una clase

Sabemos que las clases son plantillas para crear objetos. Pero, ¿cómo se crea un objeto? Para crear un objeto de una clase hay que utilizar el operador **new**, análogamente a como muestra el ejemplo siguiente:

```
CGrados grados = new CGrados();
```

En este ejemplo se observa que para crear un objeto de la clase *CGrados* hay que especificar a continuación del operador **new** el nombre de la clase del objeto seguido de paréntesis. ¿Por qué paréntesis? ¿Es acaso *CGrados* un método? Así es. Más adelante aprenderá que toda clase tiene al menos un método predeterminado especial denominado igual que ella, que es necesario invocar para crear un objeto; ese método se denomina *constructor* de la clase.

Otro ejemplo; ahora con una clase de la biblioteca Java. El paquete **java.util** proporciona una clase denominada **GregorianCalendar**. Un objeto de esta clase representa una fecha, incluyendo también opcionalmente la hora. El siguiente código crea tres objetos de esta clase, *fh1*, *fh2* y *fh3*, iniciados, el primero con la fecha y hora actual por omisión, el segundo con la fecha especificada, y el tercero con la fecha y hora especificadas:

```
GregorianCalendar fh1 = new GregorianCalendar();
GregorianCalendar fh2 = new GregorianCalendar(2001, 1, 21);
GregorianCalendar fh3 = new GregorianCalendar(2001, 1, 21, 12, 30, 15);
```

Las sentencias anteriores son válidas porque, como puede comprobar si lo desea, la clase **GregorianCalendar** proporciona varias formas de construir un objeto: sin utilizar parámetros, con tres parámetros (año, mes y día), con seis parámetros (año, mes, día, hora, minutos y segundos), etc.

Cuando se crea un nuevo objeto utilizando **new**, Java asigna automáticamente la cantidad de memoria necesaria para ubicar ese objeto. Si no hubiera suficiente espacio de memoria disponible, el operador **new** lanzará una excepción **OutOfMemoryError** cuyo estudio posponemos. Después de saber esto quizás se pregunte: ¿Quién libera esa memoria y cuándo lo hace? La respuesta es otra vez la misma: Java se encarga de hacerlo en cuanto el objeto no se utilice, cosa que ocurre cuando ya no exista ninguna referencia al objeto. Por ejemplo, en el código que se muestra a continuación, la memoria asignada a los objetos *fh1*, *fh2* y *fh3* será liberada cuando finalice la ejecución del método **main**.

```
import java.util.*;
public class CFechaHora
{
  public static void main(String[] args)
  {
    GregorianCalendar fh1 = new GregorianCalendar();
    GregorianCalendar fh2 = new GregorianCalendar(2001, 1, 21);
    GregorianCalendar fh3 = new GregorianCalendar(2001, 1, 21, 12, 30, 15);
    // ...
  }
}
```

Ahora basta con que sepa que Java cuenta con una herramienta denominada *recolector de basura* que busca objetos que no se utilizan con el fin de destruirlos liberando la memoria que ocupan. Más adelante aprenderá sobre este mecanismo.

Cómo acceder a los miembros de un objeto

Para acceder desde un método de la clase aplicación o de cualquier otra clase a un miembro (atributo o método) de un objeto de otra clase diferente se utiliza la sintaxis siguiente: *objeto.miembro*. Por ejemplo:

```
miObjeto.atributo;
miObjeto.metodo([argumentos]);
```

Lógicamente, como pueden existir varios objetos de la misma clase, es necesario especificar de quién es el miembro. Si el miembro es a su vez un objeto, la sintaxis se extiende siguiendo la misma sintaxis: *objeto.mbroObjeto.miembro*. Recuerde que el operador punto (.) se evalúa de izquierda a derecha.

Cuando el miembro accedido es un método, la interpretación que se hace en programación orientada a objetos es que el objeto ha recibido un mensaje, el especificado por el nombre del método, y responde ejecutando ese método. Los mensajes que puede recibir un objeto se corresponden con los nombres de los métodos de su clase. Por ejemplo, una sentencia como:

```
grados.CentígradosAsignar(gradosCent);
```

se interpreta como que el objeto *grados* recibe el mensaje *CentígradosAsignar*. Entonces el objeto responde a ese mensaje ejecutando el método de su clase que tenga el mismo nombre. Lógicamente, como el método se ejecuta para un objeto concreto, el cuerpo del mismo no necesita especificar explícitamente de qué objeto es el miembro accedido. Esto es, en el ejemplo siguiente se sabe que *gradosC* pertenece al objeto que está respondiendo al mensaje *CentígradosAsignar*.

```
public void CentígradosAsignar(float gC)
{
  // Establecer el atributo grados centígrados
  gradosC = gC;
}
```

Es importante asimilar que un programa orientado a objetos sólo se compone de objetos que se comunican mediante mensajes. Desde este conocimiento, no tiene sentido pensar que un método se pueda invocar aisladamente, esto es, sin que exista un objeto para el que es invocado. Por ejemplo, si en el método **main** de nuestra aplicación ejemplo pudiéramos escribir:

```
CentígradosAsignar(gradosCent);
```

seguro que nos preguntaríamos ¿a quién se asigna el valor *gradosCent*? Los métodos **static** que estudiaremos más tarde son una excepción a la regla.

Protección de los miembros de una clase

Los miembros de una clase son los atributos y los métodos, y su nivel de protección determina quién puede acceder a los mismos. Los niveles de protección a los que nos referimos son: de *paquete*, *público*, *privado* y *protegido*. De este último hablaremos en un capítulo posterior.

Por ejemplo, en la clase *CGrados* de la aplicación realizada al principio de este capítulo, hemos definido los atributos privados y los métodos, públicos:

```
class CGrados
{
  private float gradosC; // grados centígrados

  public void CentígradosAsignar(float gC)
  {
    // Establecer el atributo grados centígrados
    gradosC = gC;
  }
  // ...
}
```

Un miembro de una clase declarado *privado* puede ser accedido únicamente por lo métodos de su clase. En el ejemplo anterior se puede observar que el atributo *gradosC* es privado y es accedido por el método *CentígradosAsignar*.

Si un método de otra clase, por ejemplo el método **main** de la clase *CApGrados*, incluyera una sentencia como la siguiente,

```
grados.gradosC = 30;
```

el compilador Java mostraría un error indicando que el miembro *gradosC* no es accesible desde esta clase, por tratarse de un miembro privado de *CGrados*.

Un miembro de una clase declarado *público* es accesible desde cualquier método definido dentro o fuera de la clase o paquete actual. Por ejemplo, en la clase *CApGrados*, se puede observar cómo el objeto *grados* de la clase *CGrados* creado en el método **main** accede a su método *CentígradosAsignar* con el fin de modificar el valor de su miembro privado *gradosC*.

```
public class CApGrados
{
  // ...
  public static void main(String[] args)
  {
    // Declaración de variables
    CGrados grados = new CGrados();
    // ...
    while (gradosCent <= limSuperior) // while ... hacer:
    {
      // Asignar al objeto grados el valor en grados centígrados
      grados.CentigradosAsignar(gradosCent);
      // ...
    }
  }
}
```

Generalmente los atributos de una clase de objetos se declaran privados, estando así ocultos para otras clases, siendo posible el acceso a los mismos únicamente a través de los métodos públicos de dicha clase. El mecanismo de ocultación de miembros se conoce en la programación orientada a objetos como *encapsulación*: proceso de ocultar la estructura interna de datos de un objeto y permitir el acceso sólo a través de la interfaz pública definida, entendiendo por interfaz pública el conjunto de miembros públicos de una clase. ¿Qué beneficios reporta la encapsulación? Que un usuario de una determinada clase no pueda escribir código en base a la estructura interna del objeto, sino sólo en base a la interfaz pública; esta forma de proceder obliga a pensar en objetos y a trabajar con ellos. En el capítulo "Clases" que expondremos más adelante abundaremos más sobre lo dicho y sobre otras muchas cuestiones.

El nivel de protección predeterminado para un miembro de una clase es el de *paquete*. Un miembro de una clase con este nivel de protección puede ser accedido desde todas las otras clases del mismo paquete.

Miembro de un objeto o de una clase

Sabemos que una clase agrupa los atributos y los métodos que definen a los objetos de esa clase. Pero, cada objeto que creemos de esa clase ¿mantiene una copia tanto de los atributos como de los métodos? Lógicamente, cada objeto mantiene su propia copia de los atributos para almacenar sus datos particulares; pero, de los métodos sólo hay una copia para todos los objetos, lo cual también es lógico, porque cada objeto sólo requiere utilizarlos; por ejemplo, cuando necesite modificar sus atributos. Desde este análisis se dice que los miembros son del objeto; esto es, un mismo atributo tiene un valor específico para cada objeto, y un objeto ejecuta un método en respuesta a un mensaje recibido.

Esta forma de concebir los objetos puede suponer, en ocasiones, un desperdicio de espacio de almacenamiento; por ejemplo, volviendo a la clase *COrdenador* que expusimos en el capítulo 2, podríamos pensar en añadir un nuevo atributo que fuera el tiempo de garantía. Si suponemos que este tiempo es el mismo para todos los ordenadores, sería más eficiente definir un atributo que no formara parte de la estructura de cada objeto, sino que fuera compartido por todos los objetos. En este caso, diremos que el atributo *es de la clase* de los objetos, no del objeto.

Un atributo de la clase almacena información común a todos los objetos de esa clase. Se define agregándole previamente la palabra reservada **static**, y existe aunque no haya objetos definidos de la clase.

Para acceder a un atributo **static** de la clase puede utilizar un objeto de la clase, o bien el nombre de la clase como puede verse en el ejemplo siguiente:

```
class COrdenador
{
  String Marca;
  String Procesador;
  String Pantalla;
  static byte Garantía;
  boolean OrdenadorEncendido;
  boolean Presentación;
  // ...

  public static void main (String[] args)
  {
    // Garantía existe aunque no haya objetos definidos de la clase
    COrdenador.Garantía = 1;
  }
}
```

Utilizar una expresión como *MiOrdenador.Garantía*, siendo *MiOrdenador* un objeto de la clase *COrdenador*, aunque sea correcta, no se aconseja porque puede resultar engañosa. Parece que nos estamos refiriendo al atributo *Garantía* del objeto *MiOrdenador*, cuando en realidad nos estamos refiriendo a todos los objetos que el programa haya creado de la clase *COrdenador*.

Análogamente, un método declarado **static** es un método de la clase. Este tipo de métodos no se ejecutan para un objeto particular, sino que se aplican en general donde se necesiten, lo que impide que puedan acceder a un miembro del objeto. Una aplicación puede acceder a un método estático de la misma forma que se ha expuesto para un atributo estático.

En el ejemplo que se muestra a continuación se puede observar que para establecer el valor del atributo privado *Garantía* se ha utilizado un método estático. Si

se hubiera utilizado un método no **static**, tendría que ser invocado a través de un objeto de la clase, lo que, siendo correcto, resultaría engañoso.

```
public class CMiAplicacion
{
  public static void main (String[] args)
  {
    COrdenador.EstablecerGarantia((byte)3);
  }
}

class COrdenador
{
  private String Marca;
  private String Procesador;
  private String Pantalla;
  private static byte Garantía;
  private boolean OrdenadorEncendido;
  private boolean Presentación;
  // ...

  public static void EstablecerGarantia(byte g)
  {
    Garantía = g; // Garantía es un miembro de la clase
  }
}
```

El método *EstablecerGarantía* del ejemplo anterior puede acceder a *Garantía* porque es un miembro estático pero no podría incluir, por ejemplo, una sentencia como *Marca = "Ast"* porque *Marca* no es **static**.

Ahora puede comprender por qué el método **main** es **static**: para que pueda ser invocado aunque no exista un objeto de su clase. Por ejemplo, el método **main** de la clase *CMiAplicacion* anterior, es invocado cuando se ejecuta la aplicación, independientemente de que exista un objeto de esa clase.

Referencias a objetos

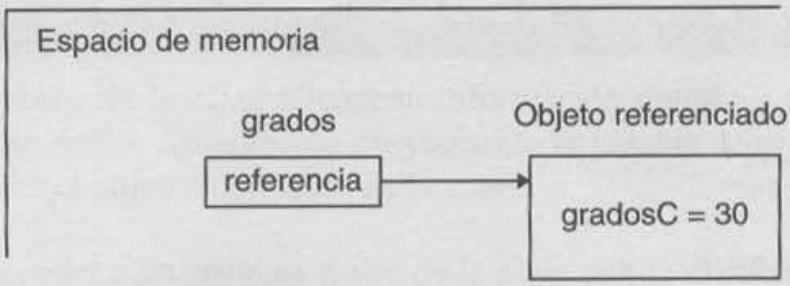
Según lo que hemos aprendido hasta ahora, para crear un objeto de una clase hay que hacerlo explícitamente utilizando el operador **new**. Por ejemplo:

```
CGrados grados = new CGrados();
```

El operador **new** devuelve una referencia al nuevo objeto, que se almacena en una variable del tipo del objeto. En el ejemplo anterior, la referencia devuelta por

el operador **new** es almacenada en la variable *grados* del tipo *CGrados*. La clase *CGrados* se encuadra dentro de lo que hemos denominado *tipos referenciados*.

Gráficamente puede imaginarse una referencia y el objeto referenciado, ubicados en algún lugar del espacio de memoria correspondiente a su aplicación, así:



En realidad una referencia es la posición de memoria donde se localiza un objeto. Observará que anteriormente nos hemos referido a la referencia *grados* como el objeto *grados*. Esto es una forma de abreviar que no crea confusión, ya que *grados* es única y referencia un único objeto *CGrados*. Una expresión como "el objeto referenciado por la variable *grados*" resulta demasiado larga y no aporta más información. Expresándonos en estos términos, cuando se asigne un objeto a otro, o bien se pasen objetos como argumentos a métodos, lo que se están copiando son referencias, no el contenido de los objetos.

El siguiente ejemplo aclarará este concepto. Se trata de la aplicación *CRacional* que expusimos al final del capítulo 2, compuesta por la clase *CRacional* a la que hemos añadido un nuevo método estático que permite sumar dos números racionales, devolviendo como resultado el número racional resultante de la suma.

```
class CRacional
{
  private int Numerador;
  private int Denominador;

  public void AsignarDatos(int num, int den)
  {
    Numerador = num;
    if (den == 0) den = 1; // el denominador no puede ser cero
    Denominador = den;
  }

  public void VisualizarRacional()
  {
    System.out.println(Numerador + "/" + Denominador);
  }
```

```
    public static CRacional Sumar(CRacional a, CRacional b)
    {
      CRacional r = new CRacional();
      int num = a.Numerador * b.Denominador +
                a.Denominador * b.Numerador;
      int den = a.Denominador * b.Denominador;
      r.AsignarDatos(num, den);
      return r;
    }

    public static void main (String[] args)
    {
      // Punto de entrada a la aplicación
      CRacional r1, r2;
      r1 = new CRacional(); // crear un objeto CRacional
      r1.AsignarDatos(2, 5);
      r2 = r1;

      r1.AsignarDatos(3, 7);
      r1.VisualizarRacional(); // se visualiza 3/7
      r2.VisualizarRacional(); // se visualiza 3/7

      CRacional r3;
      r2 = new CRacional(); // crear un objeto CRacional
      r2.AsignarDatos(2, 5);
      r3 = CRacional.Sumar(r1, r2); // r3 = 3/7 + 2/5
      r3.VisualizarRacional(); // se visualiza 29/35
    }
}
```

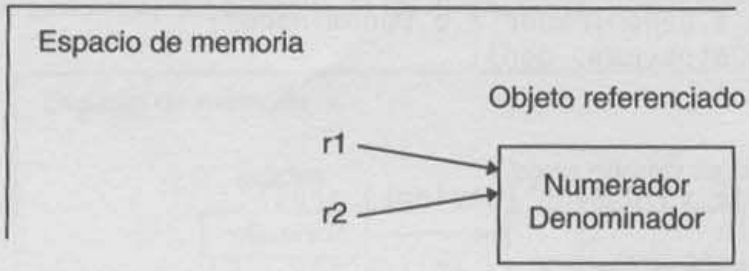
La clase *CRacional* encapsula una estructura de datos formada por dos enteros: *numerador* y *denominador*; y para acceder a esta estructura proporciona la interfaz pública formada por los métodos:

- *AsignarDatos* que permite establecer el numerador y el denominador de un número racional.
- *VisualizarRacional* que permite visualizar un racional en forma de quebrado.
- *Sumar* que devuelve el número racional resultante de sumar otros dos pasados como argumentos.

Analizada la clase *CRacional* pasemos a estudiar el método **main**. La primera parte de este método declara dos variables *r1* y *r2* de tipo *CRacional*, crea un nuevo objeto *r1* de tipo *CRacional* asignándole el valor 2/5, y asigna el valor de *r1* a *r2*.

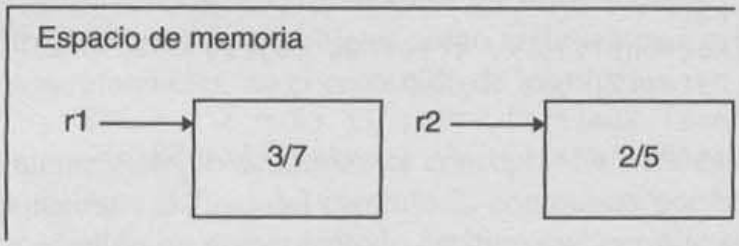
A continuación, asigna a *r1* un nuevo valor 3/7, ¿cuál es el valor de *r2*? Comprobamos que es el mismo que el de *r1*. ¿Qué ha ocurrido? Que cuando se asignó

r1 a *r2*, simplemente se creó una nueva referencia al mismo objeto referenciado por *r1*. Por lo tanto, modificar el objeto al que se refiere *r1* es modificar el objeto al que se refiere *r2* porque *r1* y *r2* referencian el mismo objeto.



Si realmente lo que deseamos es que *r1* y *r2* señalen a objetos separados, hay que utilizar **new** con ambas referencias para crear objetos separados:

```
r1 = new CRacional(); // crear un objeto CRacional r1
r1.AsignarDatos(3, 7);
r2 = new CRacional(); // crear un objeto CRacional r2
r2.AsignarDatos(2, 5);
```



Como hemos visto, una variable de un tipo referenciado se puede asignar a otra del mismo tipo. En cambio, no existe aritmética de referencias (por ejemplo, a una referencia no se le puede sumar un entero) ni tampoco se puede asignar directamente un entero a una referencia.

Pasando argumentos a los métodos

La segunda parte del método **main** del ejemplo anterior, crea un objeto *r2* y le asigna el valor 2/5. A continuación, invoca al método estático *Sumar* pasándole como argumentos los objetos *r1* y *r2* que queremos sumar. El resultado devuelto por *Sumar* será un objeto *CRacional* que quedará referenciado por *r3*.

```
// ...
CRacional r3;
r2 = new CRacional(); // crear un objeto CRacional
r2.AsignarDatos(2, 5);
```

```
r3 = CRacional.Sumar(r1, r2); // r3 = 3/7 + 2/5
r3.VisualizarRacional(); // se visualiza 29/35
```

Analicemos el método *Sumar*. Este método tiene dos parámetros de tipo *CRacional*. Después de que el método ha sido invocado desde **main**, *a* y *b* señalan a los mismos objetos que *r1* y *r2*. Esto significa que los objetos pasados a los parámetros de un método son siempre referencias a dichos objetos, lo cual significa que cualquier modificación que se haga a esos objetos dentro del método afecta al objeto original. En cambio, las variables de un tipo primitivo pasan por valor, lo cual significa que se pasa una copia, por lo que cualquier modificación que se haga a esas variables dentro del método no afecta a la variable original.

Cuando se invoca a un método, el primer argumento es pasado al primer parámetro, el segundo argumento es pasado al segundo parámetro y así sucesivamente. En Java todos los argumentos que son objetos son pasados por referencia.

```
public static CRacional Sumar(CRacional a, CRacional b)
{
  CRacional r = new CRacional();
  int num = a.Numerador * b.Denominador +
            a.Denominador * b.Numerador;
  int den = a.Denominador * b.Denominador;
  r.AsignarDatos(num, den);
  return r;
}
```

A continuación, el método *Sumar* utiliza **new** para crear un nuevo objeto *r* al que asigna el resultado de la suma de los objetos *a* y *b*. Finalmente devuelve *r*. Otra vez más lo que se devuelve es una referencia que se copia en *r3*. Finalizado este proceso la variable *r* desaparece por ser local, no sucediendo lo mismo con el objeto que señalaba, ya que ahora está señalado por *r3*.

El recolector de basura de Java eliminará un objeto cuando no exista ninguna referencia al mismo.

PROGRAMA JAVA FORMADO POR MÚLTIPLES FICHEROS

Según lo que hemos visto, un programa Java es un conjunto de objetos que se comunican entre sí. Para crear los objetos, escribimos plantillas que denominamos clases. Por ejemplo, en la aplicación acerca de números racionales escribimos una sola clase, pero en la aplicación acerca de conversión de grados centígrados a *Fahrenheit*, escribimos dos clases. En ambas aplicaciones, almacenamos todo su código en un único fichero *.java*. Esto no debe inducirnos a pensar que todo programa tiene que estar escrito en un único fichero. De hecho no es así, ya que ge-

neralmente se almacena cada clase en un único fichero para favorecer su mantenimiento y posterior reutilización.

Como ejemplo, reconstruyamos la aplicación *CRacional* creando ahora dos clases separadas: *CRacional* y *CAplicacion*.

La clase *CRacional* incluirá su estructura de datos y su interfaz pública, excepto el método **main** que será ahora incluido en *CAplicacion*. Cuando haya escrito la clase *CRacional* guárdela en el fichero *CRacional.java*.

```
public class CRacional
{
  private int Numerador;
  private int Denominador;

  public void AsignarDatos(int num, int den)
  {
    Numerador = num;
    if (den == 0) den = 1; // el denominador no puede ser cero
    Denominador = den;
  }

  public void VisualizarRacional()
  {
    System.out.println(Numerador + "/" + Denominador);
  }

  public static CRacional Sumar(CRacional a, CRacional b)
  {
    CRacional r = new CRacional();
    int num = a.Numerador * b.Denominador +
              a.Denominador * b.Numerador;
    int den = a.Denominador * b.Denominador;
    r.AsignarDatos(num, den);
    return r;
  }
}
```

Escriba ahora la clase *CAplicacion* que se muestra a continuación y guárdela en el fichero *CAplicacion.java*.

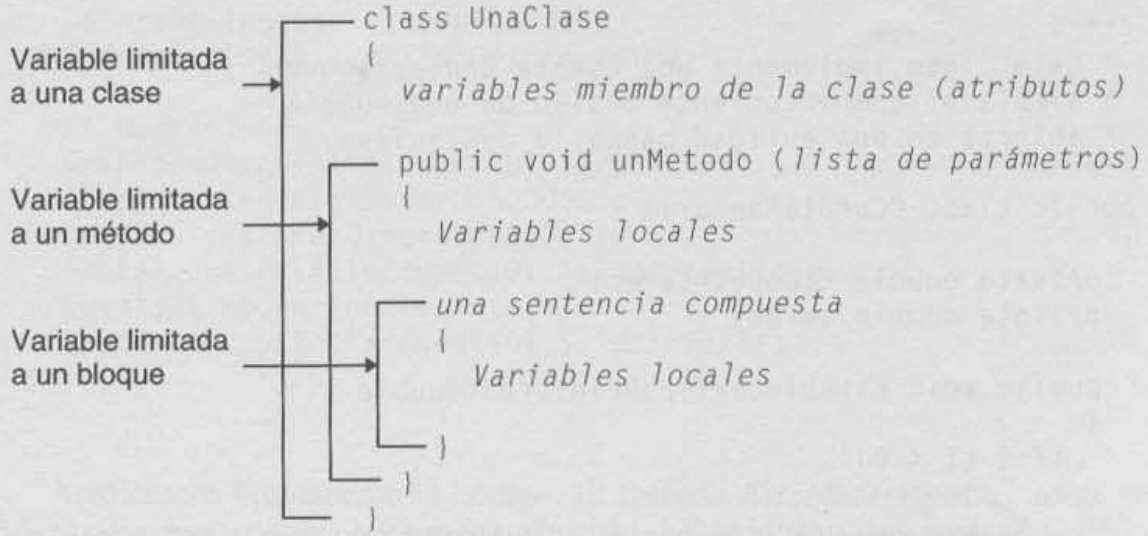
```
public class CAplicacion
{
  public static void main (String[] args)
  {
    // Punto de entrada a la aplicación
    CRacional r1, r2;
```

```
        r1 = new CRacional();           // crear un objeto CRacional
        r1.AsignarDatos(2, 5);
        r2 = r1;

        r1.AsignarDatos(3, 7);
        r1.VisualizarRacional();        // se visualiza 3/7
        r2.VisualizarRacional();        // se visualiza 3/7

        CRacional r3;
        r2 = new CRacional();           // crear un objeto CRacional
        r2.AsignarDatos(2, 5);
        r3 = CRacional.Sumar(r1, r2); // r3 = 3/7 + 2/5
        r3.VisualizarRacional();        // se visualiza 29/35
    }
}
```

Cuando se compile *CAplicacion*, que por omisión pertenece al paquete predeterminado, puesto que necesita utilizar la clase *CRacional*, buscará también ésta en el mismo paquete, lo que supone buscar su fichero compilado, o en su defecto su fichero fuente, en el directorio actual de trabajo. Por lo tanto, antes de compilar la aplicación asegúrese de que el fichero *CRacional.class* o *CRacional.java* está en el mismo directorio que *CAplicacion.java*.

ACCESIBILIDAD DE VARIABLES

Aunque este tema ya ha sido tratado, realizamos ahora un resumen. Se denomina *ámbito* de una variable a la parte de un programa donde dicha variable puede ser referenciada por su nombre. Una variable puede ser limitada a una clase, a un método, o a un bloque de código correspondiente a una sentencia compuesta.



Una variable miembro de una clase puede ser declarada en cualquier sitio dentro de la clase siempre que sea fuera de todo método. La variable está disponible para todo el código de la clase.

Una variable declarada dentro de un método es una variable local al método. Los parámetros de un método son también variables locales al método. Y una variable declarada dentro de un bloque correspondiente a una sentencia compuesta también es una variable local a ese bloque.

En general, una *variable local* existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definida. Cada vez que se ejecuta el bloque que la contiene, la variable local es nuevamente definida, y cuando finaliza la ejecución del mismo, la variable local deja de existir. Un elemento con carácter local es accesible solamente dentro del bloque al que pertenece.

EJERCICIOS RESUELTOS

Con los conocimientos que hemos adquirido hasta ahora vamos a realizar una aplicación sencilla para simular una cuenta bancaria.

Una cuenta bancaria vista como un objeto tiene, por una parte, atributos que definen su estado, como *Tipo de interés* y *Saldo*, y por otra, operaciones que definen su comportamiento, como *Establecer tipo de interés*, *Ingresar dinero*, *Retirar dinero*, *Saldo actual* o *Abonar intereses*.

Una vez abstraídas las características generales de la clase de objetos cuentas bancarias, el paso siguiente es escribir el código que da lugar a la implementación de dicha clase. Ésta puede ser más o menos así:

```
/**
 * Esta clase implementa una cuenta bancaria que
 * simula el comportamiento básico de una cuenta
 * abierta en una entidad bancaria cualquiera.
 */
public class CCuentaBancaria
{
  private double tipoDeInterés;
  private double saldo;

  public void EstablecerTipoDeInterés(double ti)
  {
    if ( ti < 0)
    {
      System.out.println("El tipo de interés no puede ser negativo");
      return; // retornar
    }
```

```
    tipoDeInterés = ti;
  }

  public void IngresarDinero(double ingreso)
  {
    saldo += ingreso;
  }

  public void RetirarDinero(double cantidad)
  {
    if ( saldo - cantidad < 0)
    {
      System.out.println("No tiene saldo suficiente");
      return;
    }
    // Hay saldo suficiente. Retirar la cantidad.
    saldo -= cantidad;
  }

  public double SaldoActual()
  {
    return saldo;
  }

  public void AbonarIntereses()
  {
    saldo += saldo * tipoDeInterés / 100;
  }

  public static void main (String[] args)
  {
    // Abrir una cuenta con 1.000.000 a un 2%
    CCuentaBancaria Cuenta01 = new CCuentaBancaria();
    Cuenta01.IngresarDinero(1000000);
    Cuenta01.EstablecerTipoDeInterés(2);

    // Operaciones
    System.out.println(Cuenta01.SaldoActual());
    Cuenta01.IngresarDinero(500000);
    Cuenta01.RetirarDinero(200000);
    System.out.println(Cuenta01.SaldoActual());
    Cuenta01.AbonarIntereses();
    System.out.println(Cuenta01.SaldoActual());
  }
}
```

Analicemos brevemente el código. El método *EstablecerTipoDeInterés* verifica si el valor pasado como argumento es negativo, en cuyo caso lo notifica y termina. Si es positivo, lo asigna al miembro *tipoDeInterés*.

El método *IngresarDinero* acumula la cantidad pasada como argumento sobre el saldo actual.

El método *RetirarDinero* verifica si hay suficiente dinero como para poder retirar la cantidad solicitada. En caso negativo lo notifica y termina; en caso positivo, resta del saldo la cantidad retirada.

El método *SaldoActual* devuelve el valor del saldo actual en la cuenta.

El método *AbonarIntereses* acumula los intereses sobre el saldo actual.

Finalmente, el método **main** crea e inicia un objeto de la clase *CCuentaBancaria* y realiza sobre el mismo las operaciones programadas con el fin de comprobar su correcto funcionamiento.

EJERCICIOS PROPUESTOS

1. Escriba la aplicación *CApGrados.java* y compruebe los resultados.

2. En el capítulo 1 hablamos acerca del depurador. Si su entorno integrado favorito aporta la funcionalidad necesaria para depurar un programa, pruebe a ejecutar la aplicación *CApGrados.java* paso a paso y verifique los valores que van tomando las variables a lo largo de la ejecución.

3. Modifique los límites inferior y superior de los grados centígrados, el incremento, y ejecute de nuevo la aplicación.

4. Cargue en su entorno de desarrollo integrado la aplicación *CApGrados.java* y modifique la sentencia:

   ```
   return 9F/5F * gradosC + 32;
   ```

   correspondiente al método *FahrenheitObtener* de la clase *CGrados*, como se muestra a continuación:

   ```
   return 9/5 * gradosC + 32;
   ```

   Después, compile y ejecute la aplicación. Explique lo que sucede.

5. Reconstruya la aplicación *CApGrados.java* para que cada clase esté almacenada en un fichero: la clase *CGrados* en el fichero *CGrados.java* y la clase *CApGrados* en el fichero *CApGrados.java*.