Programación Avanzada
UD3

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

Copyright ©2012- P. Pablo Garrido Abenza

ÍNDICE GENERAL

Copyright ©2012- P. Pablo Garrido Abenza

CAPÍTULO 3

PROGRAMACIÓN ORIENTADA A OBJETOS (I)

Copyright © 2012 P. Pablo Garrido Abenza

En los temas anteriores hemos presentado la sintáxis básica del lenguaje, que permite el desarrollo de programas de forma muy similar a como lo permiten otros lenguajes como C, en modo texto o consola. Vimos que no había grandes diferencias entre Java y C, sin embargo, para programar aplicaciones más complejas se requiere de técnicas o métodos que faciliten todo el ciclo de desarrollo.

En este capítulo se presentan los conceptos básicos de la Programación Orientada a Objetos (POO) para el análisis y diseño de programas bien estructurados. Esta metodología de programación es la más ampliamente utilizada en la actualidad, y es la que utiliza el lenguaje Java. A lo largo de todo el capítulo se describe la sintaxis que utiliza Java para la implementación de estos conceptos.

3.1. Introducción

En este primer apartado se hace un breve repaso a la evolución de la programación a lo largo de toda su historia (unos 60 años), y los distintos paradigmas de programación aparecidos. Se hace una rápida introducción al ciclo de vida del desarrollo de un proyecto software, utilizando la metodología de la Programación Orientada a Objetos, presentando los conceptos básicos (clase, objeto) y sus propiedades (abstracción, encapsulación, polimorfismo, herencia, ...). En las secciones siguientes ya se utilizará el lenguaje Java para ampliar estos conceptos.

Un nuevo paradigma de programación

La Programación Orientada a Objetos (POO) es un "nuevo" paradigma[1] en la programación, es decir, una nueva metodología para la programación, para la resolución de problemas mediante un ordenador. Sin embargo, existen otros paradigmas de programación, anteriores a la POO:

[1] El concepto de paradigma fue aplicado en los años 60 por Thomas Kuhn (1922-1996) a la ciencia, con referencia a que el pensamiento científico avanza de dos posibles formas: (1) por acumulacióon de conocimiento, y (2) creando una nueva forma de pensamiento (revolución intelectual) o cambio de paradigma.

- **Programación funcional (declarativa)**: se basa en el uso de funciones aritméticas, utilizando la composición para resolver problemas complejos a partir de otros más sencillos, por ejemplo, de forma recursiva. Como ejemplo de lenguaje funcional tenemos el lenguaje LISP (*LISt Processing*), que debido a la gran cantidad de paréntesis utilizados al anidar múltiples llamadas a funciones también se le conoce como *Lots of Insipid, Stupid Parentheses*.

- **Programación lógica (declarativa)**: un programa se compone de una serie de axiomas[2], a los que se añade una serie de condiciones o ecuaciones para deducir la solución de un problema, pero sin indicar exactamente los pasos a seguir para encontrar la solución. Dos ejemplos de lenguajes declarativos son el PROLOG (*PROgrammation en LOGique*), utilizado para temas relacionados con la inteligencia artificial, y el SQL (*Structured Query Language*), utilizado en gran medida en la actualidad para realizar acciones sobre bases de datos relacionales.

- **Programación imperativa o procedural**: un programa se compone de una serie de sentencias u órdenes que que se ejecutan de forma secuencial y pueden modificar el valor de una serie de variables (estado), es decir, se indica la tarea a realizar paso a paso (algoritmo). Este paradigma ha sido el más utilizado y duradero, pues data de la creación del lenguaje FORTRAN (*FORmula TRANslating System*) en 1957, y todavía perdura. Existen lenguajes tanto de bajo nivel como el lenguaje ensamblador, como de alto nivel. Dentro de la programación imperativa se encuentra la **programación estructurada**, que se basa en el uso de ciertas sentencias de control de flujo (bifurcación y bucles), como sería el caso del lenguaje BASIC (*Beginner's All-purpose Symbolic Instruction Code*), y la **programación modular**, que fue una evolución de la programación estructurada que permitía dividir un programa grande en varios módulos, como el lenguaje Modula, Pascal, o el C.

- **Programación Orientada a Objetos (POO)**: un programa se compone de objetos (instancias de clases), que interaccionan entre sí. La POO comenzó en 1967 con el lenguaje Simula'67 y en los años 70 con SmallTalk, pero no fue hasta los años 80 cuando aumentó el interés por ellos, debido a que requerían más recursos que otros lenguajes, así como por la aparición de interfaces gráficos GUI (*Graphical User Interface*) de tipo WIMP (*Windows, Icons, Mice, Pull-Down/Pointers*). En realidad, la POO es bastante similar a la programación imperativa, solucionando los problemas existentes en dicho paradigma. En POO, las clases de las que se instancian los objetos son como un conjunto de funciones, en los que se describe un algoritmo paso a paso. De hecho, los primeros lenguajes orientados a objetos fueron modificaciones de lenguajes de 3ª generación existentes: C++, Objetive-C, Object-Pascal, etc. La razón de esto es que los lenguajes diseñados exclusivamente para utilizar la POO eran menos eficientes que los lenguajes de programación estructurada y de este modo se podía utilizar ambas metodologías.

La adaptación al paradigma de la POO proporciona una serie de **ventajas**:

- Reutilización de código: gracias a los mecanismos de abstracción y ocultación de información en las clases, las cuales tendrán, además, una función concreta.

- Se facilita el mantenimiento (extensibilidad): podremos modificar y ampliar el comportamiento de los programas sin tener que modificar módulos ya existentes, gracias a la mayor independencia entre módulos (clases).

[2] Un axioma es un conocimiento que se considera cierto (p.e. $0! = 1$).

Copyright © 2012 P. Pablo Garrido Abenza

Sin embargo, utilizar la POO puede conllevar también **inconvenientes**:

- Se requiere un **esfuerzo de aprendizaje**, un cambio de paradigma. Esto significa que el programador tiene que adoptar una nueva forma de pensar a la hora de diseñar un nuevo programa, esto es, cambiar la forma de concebir un problema y programar su solución. El hecho de que en tan corto espacio de tiempo haya habido varios cambios de paradigma, unido a innumerables cambios en las tecnologías y lenguajes, hace que la disciplina de la programación tenga un dinamismo muy grande, lo cual provoca que mucha gente se quede obsoleta en poco tiempo.
- La **velocidad de ejecución** de un lenguaje orientado a objetos es menor que en otros lenguajes. Por ello, la POO no ha sustituido a la programación estructurada, la cual se sigue utilizando para ciertos problemas que requieren una mayor eficiencia.

POO - Conceptos básicos

El principal **objetivo** de la POO es crear modelos de la realidad, es decir, la estructura de un programa debe ser lo más parecida al problema real que se está modelando. Esto es bastante más difícil de conseguir con los lenguajes típicos de programación estructurada como C.

Clase y Objeto

Una **clase** es una generalización o plantilla en donde se define los datos (atributos) y el comportamiento (métodos) que tendrán todos los objetos que se instancien (definan) a partir de ella. Una vez definida, podremos usarla exactamente del mismo modo que el resto de clases que incorpora el lenguaje.

En otras palabras, una clase es una construcción software a partir de la cual podremos declarar objetos, de la misma forma que a partir de un tipo primitivo podemos definirnos todas las variables que necesitemos. Cuando escribimos una nueva clase estamos añadiendo un nuevo "tipo" al lenguaje, aunque con alguna diferencia; a partir de ese momento podremos declarar variables de ese nuevo "tipo", sólo que ahora ya no se llaman variables sino objetos.

Los **objetos** son modelos de programación que reflejan objetos cotidianos, los cuales poseen un comportamiento y un estado en cada instante. Los métodos o funciones para representar el comportamiento y las variables necesarias para almacenar el estado de los objetos es lo que se escribe en una clase. Dicho de otro modo, un **objeto** es una instancia (particularización) de una clase, que tendrá todo lo que se defina en la clase pero con unos valores concretos en las variables de la clase (estado). Al igual que dos variables de un tipo concreto pueden tener valores diferentes, dos objetos de la misma clase también podrán tener valores distintos.

Una clase es el modelo de la realidad para un tipo de objetos reales. Por ejemplo, en la realidad existen las figuras geométricas (cuadrados, círculos, triángulos, etc.), que quedan definidas mediante ciertos atributos, distintos en cada caso. Por ejemplo:

- Cuadrados: se necesita almacenar un color, y el tamaño del lado (los cuatro lados serán iguales).
- Círculos: quedan definidos por un color y un radio que determina su tamaño.
- Triángulos: tendrán un color y es necesario conocer el tamaño de cada uno de los 3 lados (a, b, c).

Copyright © 2012 P. Pablo Garrido Abenza

Por tanto, y de forma muy simplificada, para modelar estos objetos necesitaremos escribir 3 clases como las siguientes:

```
1 class Cuadrado {
2    String color;
3    double lado;
4 }
```

```
1 class Circulo {
2    String color;
3    double radio;
4 }
```

```
1 class Triangulo {
2    String color;
3    double a,b,c;
4 }
```

Una vez creadas las clases para los objetos a modelizar, se compilarían, y ya podrían utilizarse como cualquier otra clase incluida en el lenguaje Java. Pueden crearse objetos de ellas (cosa que se explicará un poco más adelante) con unos determinados valores. Por ejemplo, podríamos crear varios círculos, cada uno de ellos con un color y un tamaño diferente.

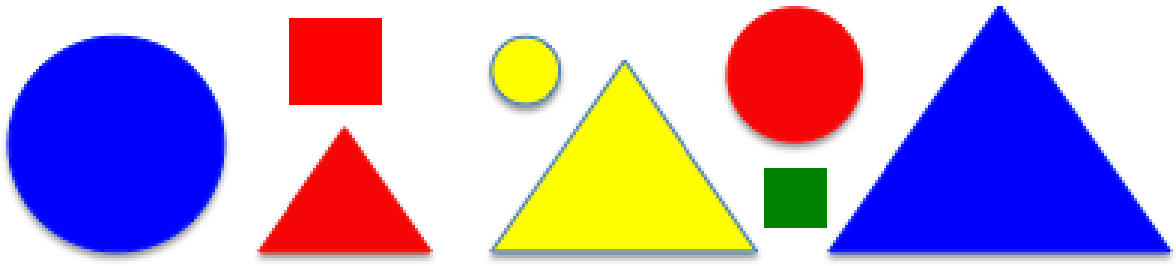

Figura 3.1: Varios objetos de las tres clases anteriores

Copyright ©2012 P. P.

3.2. Clases y sus componentes

Las clases son los bloques de código fundamentales para la construcción de aplicaciones. En este apartado se explica cómo escribir clases, y los distintos componentes que puede tener: atributos (variables de clase, variables de instancia, constantes), constructores, finalizadores, y métodos. Estos componentes son los que integra cualquier clase, tal cual puede comprobarse al consultar la documentación en línea de la librería de clases de Java (API Specification).

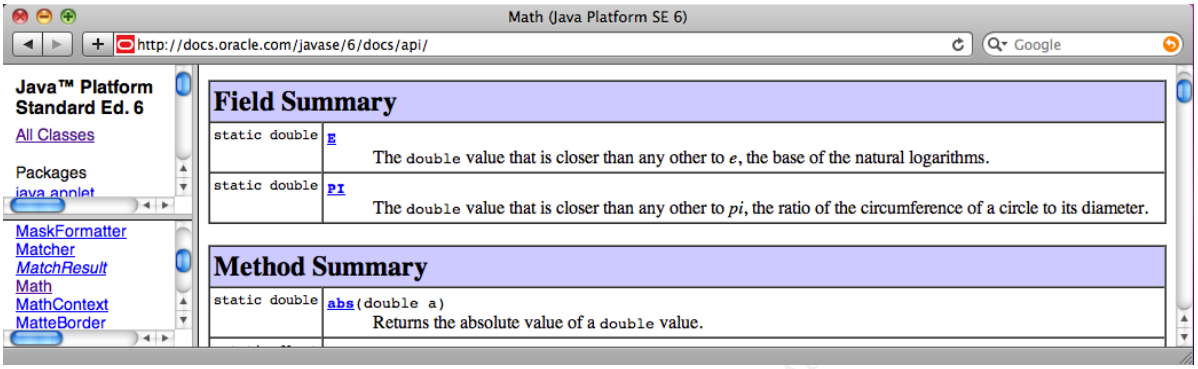

Figura 3.2: Documentación Java API

Viendo la documentación en línea (figura 3.2), podemos distingur varias partes:

- Field Summary / Detail: atributos (variables o constantes)
- Constructor Summary / Detail: constructores
- Method Summary / Detail: métodos (y finalizador, si lo hubiera)

La sintaxis de una clase en Java podría resumirse así:

```
[<ModificadoresClase>] class <NombreClase> [extends ...] [implements ...]
{
    ...
}
```

Por convenio, el nombre de las clases deben comenzar con mayúsculas. Los modificadores de clase opcionales pueden ser: `public`, `private`, `abstract`, o `final`, que de momento no utilizamos. La claúsula `extends` es opcional, y se utiliza para heredar de otra clase. Del mismo modo, la claúsula `implements` también es opcional, y se utiliza para especificar que se deben implementar una o varias interfaces. Tanto la herencia como el uso de interfaces se explicará más adelante.

Field (atributos)

Los **atributos** son los datos que contiene y encapsula una clase, es decir, almacenan el **estado** de cada uno de los objetos se instancien de la clase. Cada objeto tendrá un estado propio, es decir, cada objeto podrá tener valores distintos en sus datos, que reciben el nombre de variables de instancia, porque son propios a cada

instancia. También existen las variables de clase (o estáticas), que son iguales que las anteriores salvo que son compartidas por todos los objetos que se hayan instanciado de una clase, y se distinguen de aquellas en que éstas llevan un modificador `static` delante. Junto con las variables de instancia y de clase, las constantes también se consideran atributos de la clase, y normalmente se escriben en la parte superior de la clase.

Las clases `Cuadrado`, `Circulo`, y `Triangulo` presentadas anteriormente contenían unicamente variables de instancia; la siguiente clase amplía la clase `Circulo` con una constante (compartida por todos los objetos creados de la clase ya que es estática), y una variable de clase (también compartida por todos los objetos):

```
1 class Circulo {
2     final static double PI = 3.14159265D;    // Constante (compartida)
3     static int numObjetos;                   // Variable de clase (compartida)
4     String color;                            // Variable de instancia
5     double radio;                            // Variable de instancia
6 }
```

Constructores

Los **constructores** son bloques de código que se ejecutan justo en el momento de crear un objeto, y sirven para inicializar las variables de clase a algún valor, crear algún objeto o array, etc. Aunque se escriben de forma muy parecida a los métodos, se utilizan de forma diferente, y se distinguen porque los constructores deben tener como nombre el nombre de la clase. Todo esto se tratará más adelante.

Métodos (funciones)

Los **métodos** (llamados funciones en otros lenguajes) definen el **comportamiento** de una clase, es decir, las operaciones que pueden realizarse sobre los objetos que se instancien de ella.

Dentro de este apartado también se incluyen el finalizador de la clase (`finalize`), si lo hubiera. Tanto de los métodos como de los constructores y finalizadores se hablará más adelante, donde se verá la forma en la que se invocan (caso de los métodos), o el momento en que se ejecutan (caso de constructores y finalizadores).

Copyright ©2012- P. Pablo Garrido Abenza

3.3. Creación de objetos

Podría establecerse una doble equivalencia entre tipo y clase, y entre variable y objeto, ya que una variable se declara de un tipo primitivo, de la misma forma que un objeto se declara de una clase. Sin embargo, hay una diferencia: una variable puede utilizarse inmediatamente haya sido declarada (p.e. `int n=5;`), pero un objeto para poder utilizarse debe de crearse, lo cual se hace mediante el operador `new`. Mientras un objeto no se haya creado, su valor será `null`, y si se utilizase el objeto se generaría un error en tiempo de ejecución (*NullPointerException*).

Por ejemplo, dada una clase llamada `MiClase`, podemos declarar y crear un objeto llamado `obj` del siguiente modo:

```
1 MiClase obj;          // Declaracion del objeto
2 obj = new MiClase();  // Creacion del objeto
```

Al crear el objeto mediante el operador new se realizan varias acciones:

- **Reservar memoria** para las variables necesarias para almacenar el estado del objeto, es decir, para las variables de instancia declaradas en esa clase.
- **Inicializar** variables de instancia al valor por defecto según el tipo primitivo (ver tabla 3.1). En el caso de que la clase declare algún objeto de otra clase (por ejemplo un objeto `String`), el valor que se asigna es `null`, es decir, el valor que tiene cualquier objeto que aun no se ha creado.
- Ejecutar un **constructor**, lo cual se explicará en el apartado 3.5.

Tipo	Valor por defecto
`char`	'\0' o '\u0000'
`byte`	0
`short`	0
`int`	0
`long`	0L
`float`	0.0F
`double`	0.0D
`boolean`	`false`
Objetos (p.e. `String`)	`null`

Cuadro 3.1: Valores por defecto para cada tipo primitivo y objetos

Respecto las **variables de clase compartidas** por todos los objetos, las que son estáticas (`static`), comentar que sólo se reserva memoria para ellas y se inicializan en el momento en que se crea el primer objeto de la clase; los siguientes objetos que se creen utilizarán esas variables compartidas con los valores que tengan en ese instante.

El siguiente programa crea un objeto de la clase `Circulo` y otro de la clase `Triangulo`, y le asigna unos valores a sus valores de instancia:

Copyright © 2012 P. Pablo Garrido Abenza

```
1  class Figuras {
2     public static void main (String args[]) {
3        Circulo   c;
4        Triangulo t;
5
6        c = new Circulo();
7        c.radio = 10.0;
8        t = new Triangulo();
9        t.a=2.0; t.b=2.0; t.c=2.0;
10    }
11 }
```

NOTA: las **cadenas de caracteres** también son objetos en Java, en este caso de la clase `String`. Esto significa que para utilizar una cadena de caracteres habría que crearla mediante el operador `new`, como cualquier otro objeto. Sin embargo, dado que las cadenas de caracteres son muy utilizadas, se ha simplificado el lenguaje, y únicamente para este caso particular podemos hacerlo de otra forma más sencilla. Por ejemplo, el siguiente fragmento de código crea dos cadenas de dos formas equivalentes:

```
1  String hola  = "Hola";
2  String adios = new String("Adios");
```

```
1  String hola  = "";
2  String adios = new String();
```

Copyright ©2012- P. Pablo Garrido Abenza

3.4. Métodos (funciones)

Como se dijo antes, los **métodos** definen el **comportamiento** de una clase, es decir, las operaciones que pueden realizarse sobre los objetos que se instancien de ella. Los métodos son los bloques de código que componen las clases; cualquier código (salvo alguna declaración de variables de clase) se debe escribir dentro de un método, el cual se incluye dentro de una clase. En este apartado se explica la sintaxis necesaria para el uso de métodos, tanto para su escritura como para su invocación o llamada desde otro método.

Invocar un método implica utilizar su **nombre**, el paso de **argumentos** al método llamado, y el **retorno de un valor** desde este al método llamador. Por tanto, los componentes de cualquier método son los siguientes:

- Un método debe tener un **nombre**, que será el que se utilice para invocarlo. Este nombre debe expresar bien la tarea que realiza; esto hace más comprensibles los programas sin tener que escribir demasiados comentarios (*código autodocumentado*). Al igual que para el caso de las variables, el nombre de un método puede incluir caracteres especiales (Unicode), y no tiene límite de longitud. Por convenio, se aconseja que su nombre empiece por minúsculas, y cada palabra distinta comience por mayúsculas, sin utilizar ningún carácter separador entre ellas (guión, subrayados, ...).

- Los **argumentos** consisten en una lista de parámetros separados por coma, donde cada uno se especifica como si fuera una variable, con un tipo y un nombre, todo ello entre paréntesis. En caso de no especificar parámetros, se deben especificar también los paréntesis, aunque vacíos `()`.

- En cuanto al **valor retornado** por un método, se indica antes del nombre, y puede ser cualquier tipo primitivo o cualquier clase (pueden retornarse objetos). En el caso de que el método no retorne ningún valor, el tipo de retorno a especificar es la palabra reservada `void` (del inglés, vacío).

- Después de todo lo anterior, ya vendría el **código** del método entre llaves `{...}`. Dependiendo del tipo de retorno especificado, en el caso de que se haya especificado un tipo o clase, se debe obligatoriamente utilizar una sentencia `return(...);` para retornar un valor del tipo apropiado, asegurándonos de que siempre exista una sentencia `return(...);` independientemente del flujo de ejecución; en caso contrario, el compilador nos dará un error de ejecución: "*missing return*". Por otra parte, si el tipo de retorno era `void`, no se retorna ningún valor; no obstante, si a mitad del código se quiere finalizar su ejecución y retornar al método llamador puede utilizarse `return;`, pero sin especificar ningún valor.

En próximos capítulos se verá que puede indicarse también una lista de excepciones, que de momento no utilizamos.

Es importante también que un método no sea demasiado grande, pues seguramente realizará varias tareas, es decir, que podrá dividirse en varios métodos más pequeños, lo cual facilitará su comprensión. Sin embargo, cada llamada a un método lleva un coste computacional, tanto en tiempo de ejecución como de recursos (memoria), por lo que si un método se va a invocar muchas veces (por ejemplo, desde dentro de un bucle) puede que convenga poner el código desde donde se invoca en lugar de llamar al método.

En POO, cuando se invoca a un método para realizar una acción o solicitar cierta información se dice que se realiza un *paso de mensaje*.

Copyright © 2012 P. Pablo Garrido Abenza

Vamos a ampliar de nuevo la clase `Circulo` para incluir un método para calcular el area de cualquier círculo, que como se sabrá, se calcula con la fórmula $\pi * r^2$:

```
class Circulo {
   final static double PI = 3.14159265D;    // Constante (compartida)
   static int numObjetos;                   // Variable de clase (compartida)
   String color;                            // Variable de instancia
   double radio;                            // Variable de instancia

   // Cálculo del area de un circulo
   double area () {
      return (PI*radio*radio);              // PI por radio al cuadrado
   }
}
```

En el siguiente ejemplo se declaran y crean en una sóla sentencia dos círculos (objetos de la clase `Circulo`) en una sóla sentencia. Después se asignan valores a sus atributos: color (rojo o azul) y un radio (10.0 o 20.0). Finalmente se accede al método `area()` que calcula el area de cada uno y se muestra por pantalla:

```
class Figuras2 {
   public final static void main (String args[]) {
      Circulo c1 = new Circulo();
      Circulo c2 = new Circulo();
      c1.color = "rojo"; c1.radio = 10.0;
      c2.color = "azul"; c2.radio = 20.0;
      System.out.println ("Area1 = " + c1.area());
      System.out.println ("Area2 = " + c2.area());
   }
}
```

Respecto a la llamada al método `area()`, en ambos casos se ejecutará el mismo código escrito en la clase `Circulo`, que si recordamos no recibía ningún argumento. Entonces surge la pregunta de qué valor de radio utiliza para calcular el area; viendo el código de dicho método, se aprecia que se accede al valor de una variable `radio`, que será el valor que cierto objeto tenga en esa variable de instancia. Pues bien, dicho objeto es aquel con el que se invoca al método, es decir, `c1.area()` calculará el area de un círculo de radio 10.0, mientras que `c2.area()` calculará el area de un círculo de radio 20.0.

El resultado (aproximado) de la ejecución del programa anterior será:

```
Area1 = 314.1592654
Area2 = 1256.637061
```

Copyright © 2012 P. Pablo Garrido Abenza

Métodos de instancia vs. métodos de clase

Existen métodos de instancia y métodos de clase (`static`), los cuales se invocan de forma ligeramente distinta. La forma de saber si un método de Java es de un tipo u otro es mirando si lleva el modificador `static`: los que lo lleven serán métodos de clase, los que no, serán métodos de instancia.

- Los **métodos de instancia** se llaman así porque pueden acceder a las variables de instancia, propias del objeto con el que se invocan. Por ello, la única posibilidad de ejecutar un método de instancia es mediante un objeto. El siguiente fragmento de código invoca al metodo de instancia `area()` de la clase `Circulo`, el cual accedía al valor del `radio` del objeto con el que se invoca (en este caso `c`):

```
1 Circulo c = new Circulo("azul", 15.0);
2 double  s = c.area();  // area() era un metodo de instancia
```

- Los **métodos de clase o estáticos** no pueden acceder a las variables de instancia, todos los valores que necesiten se tendrán que pasar como argumento. Para invocar a estos métodos basta con utilizar el nombre de la clase, es decir, no hace falta crear objetos previamente (aunque también sería válido). Por ejemplo, la clase matemática `Math` tiene una serie de métodos matemáticos: para calcular la potencia de un valor elevado a otro `pow()`, el valor absoluto `abs()`, etc. Se invocarían así:

```
1 double picuadrado = Math.pow(Math.PI, 2.0);  // PI al cuadrado
2 int    diferencia = Math.abs(5-8);           // Resultado = 3
```

Polimorfismo

El concepto de **polimorfismo** consiste en escribir varios métodos con el mismo nombre dentro de una misma clase, por lo que se les denomina *métodos sobrecargados*. La forma de distinguirlos a la hora de invocarlos es por los argumentos que se pasan, bien por el número de argumentos o por sus tipos. Por ejemplo, en la clase matemática `Math` que hemos comentado en la sección anterior, el método para calcular el valor absoluto `abs()` está definido 4 veces (ver tabla 3.2), las 4 con el mismo nombre pero variando el tipo del argumento que recibe: `double`, `float`, `long`, o `int`. La ventaja de esto es que es más fácil de recordar que si cada uno tuviera un nombre distinto, como ocurre en el lenguaje C: `abs(int a)`, `labs(long int a)`, `fabs(double a)`, `cabs(complex a)`, ...

Método	Descripción
`static int abs(int a)`	Retorna el valor absoluto de un valor `int`.
`static long abs(long a)`	Retorna el valor absoluto de un valor `long`.
`static float abs(float a)`	Retorna el valor absoluto de un valor `float`.
`static double abs(double a)`	Retorna el valor absoluto de un valor `double`.

Cuadro 3.2: Clase `Math` - métodos para el valor absoluto

La **sobrecarga de operadores** también es una forma de polimorfismo. Se trata de darle más funcionalidad a ciertos operadores aritméticos existentes (+, -, ...), para poder realizar operaciones con objetos de cualquier clase, además de con los tipos primitivos habituales. El lenguaje Java no lo soporta.

Copyright © 2012 P. Pablo Garrido Abenza

3.5. Constructores (inicializadores)

En las secciones 3.2 y 3.3 se nombraron a los constructores, los cuales son bloques de código que se ejecutan justo en el momento de crear un objeto mediante el operador `new`. Un constructor se escribe de forma muy similar a un método, con las siguientes excepciones:

1. Tiene que tener el mismo **nombre** que la clase (obligatoriamente).
2. No especifica **tipo de retorno** (ni siquiera `void`).

Puede no haber ningún constructor en una clase o pueden haber varios:

- **Ningún constructor**: se añade automáticamente un constructor llamado "*constructor por defecto*", el cual no recibe parámetros y no hace nada (código vacío), de lo contrario no se podrían crear objetos de la clase. Por ejemplo, aunque en las clases `Circulo`, `Cuadrado`, y `Triangulo` anteriores no aparecía ningún constructor, esto no significa que no podamos crear objetos de dichas clases (con `new`). Este constructor por defecto se puede utilizar para inicializar los datos a algún valor distinto del valor por defecto.
- **Varios constructores**: podemos escribir tantos constructores como necesitemos, pero puesto que todos los constructores se llamarán de la misma forma (el nombre de la clase), para distinguirlos es necesario que cambie el número o el tipo de argumentos recibidos, al igual que ocurría con los métodos (polimorfismo, ver sección 3.4). En este caso no se añade automáticamente el constructor por defecto, por lo que si se necesita será necesario escribirlo nuevamente, aunque no tenga nada de código.

A continuación completamos la clase `Circulo` con 4 constructores, lo cual nos permitirá crear objetos de esta clase de 4 posibles formas. Podemos fijarnos en el primer constructor, el constructor por defecto, que se invoca cuando no se pasan argumentos al crear el objeto `Circulo` con `new`; lo que hace es asignar al radio un valor aleatorio entre 0 y 100, es decir, que si cada vez que creemos un objeto Circulo, éste tendrá un tamaño distinto de los otros. Esto es un ejemplo, pero como se puede imaginar, puede asignarse cualquier valor o no hacer nada (dejarlo en blanco), o incluso no escribirlo, aunque se aconseja escribirlo siempre.

```
1  class Circulo {
2     final static double PI = 3.14159265D;   // Constante (compartida)
3     static int numObjetos;                  // Variable de clase (compartida)
4     String color;                           // Variable de instancia
5     double radio;                           // Variable de instancia
6
7     // Constructor por defecto (sin parametros)
8     Circulo () {
9        radio = Math.random() * 100.0;
10    }
11
12    // Constructor conociendo el color
13    Circulo (String color) {
14       this.color = color;
15    }
```

Copyright © 2012 P. Pablo Garrido Abenza

```
16
17      // Constructor conociendo el radio
18      Circulo (double r) {
19         if (r >= 0.0) radio = r;     // El this no es necesario aqui
20      }
21
22      // Constructor conociendo el color y el radio
23      Circulo (String color, double r) {
24         this.color = color;
25         if (r >= 0.0) radio = r;     // El this no es necesario aqui
26      }
27
28      // Cálculo del area de un circulo
29      double area () {
30         return (PI*radio*radio);     // PI por radio al cuadrado
31      }
32  }
```

Como se puede observar, los 4 constructores difieren en los argumentos recibidos (número y/o tipos). La ventaja de haber definido varios constructores es que podemos crear objetos de la clase `Circulo` de 4 posibles formas, y dependiendo de los parámetros que se especifiquen, el compilador puede deducir cuál de ellos invocar. Aunque parece que hay que escribir más código, la finalidad de los constructores es todo lo contrario, ya que el código que se escriba en la clase ya no tendrá que repetirse en cualquier programa que la utilice: cada vez que se requiera crear un objeto de esta clase se invoca al constructor que se necesite (mediante `new`), en lugar de crearlo primero (con `new`) y luego escribir su código.

A continuación se muestran varios fragmentos de código con distintas posibilidades.

```
1  Circulo c = new Circulo();
2  c.color = "rojo";
3  c.radio = 10.0;
```

```
1  Circulo c = new Circulo("rojo");
2  ...
3  c.radio = 10.0;
```

```
1  Circulo c = new Circulo(10.0);
2  ...
3  c.color = "rojo";
```

```
1  Circulo c =
2     new Circulo("rojo", 10.0);
```

El uso de constructores, además de ahorrar código es más seguro que cuando se asignan valores manualmente a las variables, ya que podría darse el caso de que se asignen valores incorrectos. Al hacerlo dentro de los constructores (o métodos, como se verá más adelante) se gana en seguridad, pues puede comprobarse si el valor es correcto o no antes de realizar la asignación, como para el caso de intentar asignar un valor negativo para un radio (p.e. `c.radio = -10.0;`). En el código mostrado de los constructores, en ocasiones se utiliza un `this`; esto se explica en la sección 3.5.

Copyright © 2012 P. Pablo Garrido Abenza

El siguiente ejemplo muestra dos programas en los que se crean dos objetos de la clase `Circulo` de un color y un tamaño concreto, en la parte de la izquierda suponiendo que no se han escrito constructores (como antes), y en la parte derecha utilizándolos. Se puede apreciar que al final, los programas se simplifican gracias al uso de constructores.

```
1  class Figuras2a {
2     public final static void main (
          String args[]) {
3        Circulo c1 = new Circulo();
4        Circulo c2 = new Circulo();
5        c1.color = "rojo";
6        c1.radio = 10.0;
7        c2.color = "azul";
8        c2.radio = 20.0;
9        System.out.println
10          ("Area1 = " + c1.area());
11       System.out.println
12          ("Area2 = " + c2.area());
13    }
14 }
```

```
1  class Figuras2b {
2     public final static void main (
          String args[]) {
3        Circulo c1 =
4           new Circulo("rojo", 10.0);
5        Circulo c2 =
6           new Circulo("azul", 20.0);
7        System.out.println
8           ("Area1 = " + c1.area());
9        System.out.println
10          ("Area2 = " + c2.area());
11    }
12 }
```

Las clases de Java suelen tener varios constructores, facilitando la tarea al programador. Por ejemplo, consultando la documentación en línea de la librería de clases de Java (API Specification) se puede comprobar que la clase `String` dispone de un buen número de constructores (unos 15): para crear una cadena vacía, desde un array de `byte`, desde un array de `char`, desde otro `String`, ... Las cadenas de caracteres se explican con más detalle en un capítulo posterior.

Referencia al objeto actual: `this`

Java tiene una palabra reservada llamada `this` que tiene dos posibles usos:

- **Como referencia al objeto actual**: como ya se ha comentado, cuando se invoca un método utilizando distintos objetos (p.e. `c1.area();`, `c2.area();` ...), siempre se ejecuta el mismo trozo de código, pero cambian los datos utilizados dentro de él, que serán las del objeto con el que se invoca al método (`c1`, `c2`, ...). ¿Habría alguna forma de tener una referencia a dicho objeto dentro del método? Sí, con `this`, que se utilizará como un nombre de objeto genérico dentro de la clase, y será equivalente a utilizar un nombre de un objeto concreto desde fuera. Pero, si la ejecución ya está dentro de un método de una clase, ¿no podemos acceder ya a todos sus datos?, ¿para qué serviría entonces utilizar `this`? La respuesta es, para acceder a los datos del objeto en el caso de que haya conflicto con algún parámetro del método o con alguna variable local. Si nos fijamos en la clase `Circulo` de la sección anterior, en algunos constructores (como el que se muestra a continuación) se utilizaba la palabra clave `this` en un caso así: el constructor recibe un argumento llamado `color`, y otro llamado `r`, pero la clase ya tenía una variable de instancia que se llamaba `color`, entrando pues en conflicto. El criterio seguido para solventar el conflicto consiste en considerar siempre el identificador más cercano, es decir, en este ca-

Copyright © 2012 P. Pablo Garrido Abenza

so el parámetro `color`. Pero como ese valor hay que asignarlo a la variable de clase de mismo nombre, no podría escribirse `color=color;`, ya que no estaríamos haciendo nada (asignaremos el valor de una variable a sí misma). Para estos casos es para lo que puede utilizarse la referencia `this` como se muestra en el código. En otras palabras, si se utiliza `this.color` se hace referencia a la variable de clase, mientras que si se utiliza simplemente `color` se hace referencia al parámetro (tiene prioridad).

```
1  class Circulo {
2     String color;                // Variable de instancia
3     double radio;                // Variable de instancia
4     ...
5     // Constructor conociendo el color y el radio
6     Circulo (String color, double r) {
7        this.color = color;
8        if (r >= 0.0) radio = r;  // El this no es necesario aqui
9     }
10 }
```

En cuanto al parámetro `r` no hay problema, ya que la variable de clase se llamaba `radio`, es decir, al llamarse de forma distinta no hay conflicto, por lo que podemos acceder a dicha variable de clase sin problemas y hacer la asignación sin utilizar la referencia `this` con `radio=r;`.

- **Invocar a un constructor desde otro de la misma clase**: en ocasiones, dos o más constructores tienen que realizar prácticamente la misma tarea, teniendo que duplicar varias líneas de código en ellos. En la clase `Circulo` que utilizamos de ejemplo no era mucho problema pues tan sólo tienen una o dos líneas, pero en otras clases la cantidad de código a duplicar (o triplicar, o ...) podría llegar a ser considerable. La solución a esto sería utilizar `this(...);` para invocar a otro constructor de su misma clase, especificando los parámetros adecuados. Del mismo modo que al utilizar el operador `new`, según los argumentos especificados se ejecuta un constructor u otro, al utilizar `this(...);` pasa lo mismo. Por ejemplo, vamos a modificar la clase `Circulo` para que al utilizar el constructor por defecto, se cree un objeto círculo de color verde y radio 15.0. En vez de realizar esas dos asignaciones tal cual en el constructor, vamos a invocar a otro constructor para que lo haga:

```
1  class Circulo {
2     ...
3     // Constructor por defecto
4     Circulo () {
5        this ("verde", 15.0);  // Invocacion de otro constructor ¿cual?
6     }
7
8     // Constructor conociendo el color y el radio
9     Circulo (String color, double r) {
10       this.color = color;
11       if (r >= 0.0) radio = r;  // El this no es necesario aqui
12    }
13 }
```

Copyright © 2012 P. Pablo Garrido Abenza

Ahora, cuando desde una clase externa se crea un objeto mediante `Circulo c = new Circulo();`, la ejecución salta al constructor por defecto, y desde éste, al constructor que recibe un `String` y un `double`, el cual realiza finalmente las inicializaciones de las variables.

Puesto que los constructores se llaman como el nombre de la clase, quizás habría sido más inmediato de entender utilizar simplemente dicho nombre con los parámetros adecuados (p.e. `Circulo (‘‘verde’’, 15.0);`), pero eso no es válido, se requiere utilizar `this(...);`.

Un constructor sólo puede invocarse así desde otro constructor, no desde un método. Además, es imprescindible que el uso de `this(...);` se haga en la primera línea del constructor, incluso antes que la declaración de cualquier variable local que se necesitase. Lo mismo ocurre con el uso de `super(...);`, que se utiliza para invocar otro constructor pero de una superclase de la que se hereda, aunque esto se explicará en otro capítulo.

Copyright ©2012- P. Pablo Garrido Abenza

3.6. Referencias a objetos

Para comprender el concepto de **referencia a un objeto** es necesario distinguir si un objeto que se ha declarado se crea mediante el operador `new` o se inicializa a otro objeto ya existente. En el primer caso será un objeto nuevo, que tendrá reservada una zona de memoria concreta y una referencia a ella, pero en el segundo caso, sólo será una referencia, es decir, se tendrá un único objeto pero con dos referencias al mismo.

El siguiente ejemplo declara 3 objetos, los dos primeros (`c1` y `c2`) se crean como se ha visto hasta ahora (mediante el operador `new`), pero el tercero (`c3`) no se crea, sino que simplemente se le asigna la referencia al objeto `c2`. Por tanto, utilizar `c2` o `c3` es indiferente, puesto que se trata en realidad del mismo objeto.

```
1 class Figuras2c {
2    public final static void main (String args[]) {
3       Circulo c1 = new Circulo("azul", 15.0);
4       Circulo c2 = new Circulo("rojo", 15.0);
5       Circulo c3 = c2;
6       ...
7    }
8 }
```

Un ejemplo de uso de referencias sería el siguiente. Imaginemos que tenemos un conjunto de círculos, es decir, un array de objetos `Circulo` (ya creados), y queremos recorrerlos todos mediante un bucle. En algún momento nos hará falta tener una referencia al objeto `Circulo` correspondiente a cada iteración. En cada nueva iteración, dicha referencia apuntará al siguiente objeto de la lista, pero esto no supone que se esté creando un nuevo objeto (no se ha utilizado el operador `new`).

Copyright © 2012 P. Pablo Garrido Abenza

Comparación de objetos

Para comparar si dos variables son iguales, esto es, tienen el mismo valor, se utiliza el operador `'=='`. Sin embargo, cuando se trata de comparar si dos objetos son iguales hay que distinguir si lo que se compara es la referencia (si apuntan al mismo objeto, misma "dirección") o el contenido.

1. **Para comparar la referencia**, es decir, si dos referencias apuntan al mismo objeto: operador `'=='`.

```
1 Circulo c1, c2;
2 ...
3 if (c1 == c2) {
4    // Los objetos c1 y c2 son el mismo objeto
5    ...
6 } else {
7    // Los objetos c1 y c2 son distintos objetos
8    ...
9 }
```

2. **Para comparar el contenido**: método `equals()`. Este método está en la clase raíz de la jerarquía de clases de Java (`Object`), y lo debemos sobreescribir nosotros. Hasta que no se llegue al capítulo donde se explica la herencia no puede comprenderse esto, aunque de momento basta con saber que se trata de escribir un método del siguiente estilo dentro de la clase, cuyo código escribe el programador de tal manera que se retorne `true` cuando se estime que dos objetos son iguales, comparando las variables de instancia que se estimen convenientes de ambos objetos, y `false` en caso contrario. Para comparar el contenido dos objetos `c1` y `c2` se escribe `c1.equals(c2)`, aunque también sería equivalente escribir `c2.equals(c1)`, del mismo modo que para comparar `c1==c2` sería equivalente escribir `c2==c1`.

```
1 class MiClase {
2     ...
3     public boolean equals (Object o) {
4         ...
5     }
6 }
```

IMPORTANTE:

- El método `equals()` debe escribirse siempre así, es decir, retornando un valor de tipo `boolean`, y recibiendo un objeto `Object`. Al igual que pueden realizarse conversiones de tipo mediante la técnica del *casting*, también puede aplicarse esta técnica para realizar conversiones de clase. Puesto que se recibe un objeto de la clase `Object`, es necesario realizar *casting* a la clase que estamos comparando, la clase actual. Los dos objetos a comparar son el que se recibe y el actual (`this`).
- Si una clase no declara el método `equals()`, éste se podrá utilizar, ya que se hereda de la superclase raíz `Object`, pero su comportamiento será como si comparásemos con el operador `'=='`.

Copyright © 2012 P. Pablo Garrido Abenza

Como ejemplo, ampliamos de nuevo la clase `Circulo` con la función `equals()`, que nos permitirá comparar el contenido de dos objetos. En este caso, consideramos que dos círculos son iguales si tienen el mismo tamaño (radio)[3], ignorando si tienen el mismo color o no. Vemos que en este caso realizamos casting a la clase `Circulo` desde el objeto `Object` que recibe.

```
1  class Circulo {
2     String color;                              // Variable de instancia
3     double radio;                              // Variable de instancia
4     ...
5     public boolean equals (Object o) {
6        Circulo otro = (Circulo) o;
7        if (this.radio == otro.radio) {
8           return (true);
9        } else {
10          return (false);
11       }
12    }
13 }
```

Ahora completamos también el ejemplo que declaraba 3 objetos `Circulo`, y creaba los 2 primeros. Los comparamos utilizando tanto el operador `'=='` como el método `equals()` que se ha definido.

```
1  class Figuras3 {
2     public final static void main (String args[]) {
3        Circulo c1 = new Circulo ("azul", 15.0);
4        Circulo c2 = new Circulo ("rojo", 15.0);
5        Circulo c3 = c2;
6        if (c1==c2) {
7           System.out.println ("c1 y c2 son el mismo objeto");
8        }
9        if (c1.equals(c2)) {
10          System.out.println ("c1 y c2 son iguales");
11       }
12       if (c2==c3) {
13          System.out.println ("c2 y c3 son el mismo objeto");
14       }
15       if (c2.equals(c3)) {
16          System.out.println ("c2 y c3 son iguales");
17       }
18    }
19 }
```

[3]Los valores de tipo `double` o `float` no deberían compararse si son iguales con el operador `'=='`, ya que podría haber pequeñas pérdidas de precisión por algún motivo; aquí se hace por sencillez.

Copyright © 2012 P. Pablo Garrido Abenza

El resultado de la ejecución del programa anterior será el siguiente. Puede comprobarse que `c1` y `c2` dice que son iguales, ya que tienen el mismo radio, aunque no dice que sean el mismo objeto; sin embargo, `c2` y `c3` sí que lo son, ambos referencian al mismo objeto, y por tanto, su contenido tendrá que ser el mismo:

```
c1 y c2 son iguales
c2 y c3 son el mismo objeto
c2 y c3 son iguales
```

Referencia nula: `null`

Como ya se sabrá, antes de utilizar cualquier objeto es necesario crearlo mediante el operador `new`. Desde que se declara un objeto hasta que se crea, el valor de un objeto es una constate llamada `null`. Cualquier intento de utilizar un objeto no creado generará un error en tiempo de ejecución: `NullPointerException`.

Como ya se comentó anteriormente, las variables de instancia definidas en una clase se inicializan automáticamente a un valor por defecto, dependiendo de su tipo. Pues bien, el valor por defecto de los objetos que se hayan declarado es justo este valor `null`, es decir, que los objetos deben crearse siempre de forma explícita mediante el operador `new`.

En ocasiones es inmediato saber si un objeto se ha creado o no simplemente mirando el código, pero hay otras en que no es tan sencillo, y puede que necesitemos averiguar si un objeto ya se ha creado o no, esto es, si es `null` o no. Sería el caso, por ejemplo, de una función que recibe un objeto y podría ser que en alguna invocación el objeto recibido valga `null`, pudiendo generar el error `NullPointerException` mencionado.

Tenemos dos **formas de comprobar si un objeto se ha creado** o no:

1. Comparando el objeto con el valor `null` (referencia nula) mediante una sentencia `if`.

```
1  // Una funcion que recibe un objeto
2  void unaFuncion (Circulo c) {
3     // ¿Se ha creado el objeto c?
4     if (c == null) {
5        // No
6        System.out.println ("El objeto no se ha creado");
7     } else {
8        // Si, ya podemos acceder a sus datos o metodos
9        System.out.println (c.area());
10    }
11 }
```

2. Capturando la excepción `NullPointerException`, cosa que se explica en otro capítulo.

Copyright © 2012 P. Pablo Garrido Abenza

El siguiente es un ejemplo de uso de la constante `null`, en el que se comprueba si un objeto es distinto de `null` antes y después de crearlo con `new`. Vemos que el objeto es `null`.

```
1 // Referencia nula mientras no creemos el objeto
2 Circulo c;
3 System.out.println ("¿Se ha creado el objeto? " + ((a!=null) ? "Si" : "No"));
4 c = new Circulo();
5 System.out.println ("¿Se ha creado el objeto? " + ((a!=null) ? "Si" : "No"));
```

La salida del programa anterior será:

```
¿Se ha creado el objeto? No
¿Se ha creado el objeto? Si
```

En la siguiente sección 3.7 se explica otro uso de la palabra reservada `null`, consistente en eliminar objetos ya creados, aunque este uso no es necesario en Java.

Copyright ©2012- P. Pablo Garrido Abenza

3.7. Destrucción de objetos

En este apartado se explica la eliminación de objetos, en concreto, el uso de finalizadores y el algoritmo de recolección de basura (*garbage collection*) que integra la plataforma Java.

Lo primero que hay que comentar es que **en Java no hay que preocuparse** por la destrucción de objetos, puesto que se destruyen automáticamente cuando se detecta que ya no son necesarios, al salir la ejecución de su ámbito (visibilidad). Por ejemplo, si dentro de un método se declara y se crea un objeto, al finalizar la ejecución del método el objeto ya no tiene sentido que siga existiendo, por lo que puede destruirse. Pero esto se realiza de forma transparente para el programador, es decir, al contrario que en otros lenguajes, la memoria dinámica es gestionada automáticamente por el intérprete de Java (JVM), liberando al programador de tener que liberar toda la memoria que haya sido reservada previamente.

Sin embargo, también es posible solicitar la destrucción de un objeto de **forma explícita**, lo cual se consigue asignando el valor `null` al objeto:

```
1 Circulo c = new Circulo();
2 ...
3 c = null;
4 ...
```

En el caso de que un objeto tenga **varias referencias**, el objeto no se marcará para eliminar hasta que se hayan eliminado todas ellas. Cada objeto tiene un contador interno de referencias que apuntan a él, que se incrementa cada vez que se asigna una nueva referencia, y se decrementa cada vez que una de esas referencias se elimina (asignando `null`, por ejemplo). En el siguiente fragmento de código, se crea un objeto `Circulo` y su primera referencia `c1`; luego se crea una nueva referencia `c2`, por lo que el contador de referencias pasará a valer 2; posteriormente se van eliminando estas referencias, en este caso de forma manual asignando `null`, y el contador irá decrementándose. En el momento en que se hayan eliminado todas las referencias, el contador valdrá 0, marcándose para ser eliminado.

```
1 Circulo c1 = new Circulo();  // contador=1
2 Circulo c2 = c1;             // contador=2
3 c2 = null;                   // contador=1
4 c1 = null;                   // contador=0
```

Recolector de basura (*garbage collector*)

Tanto en el caso de destrucción automática de referencias (por salir de su ámbito de visibilidad) como en el caso de destrucción manual (asignando `null`), y cuando un objeto haya perdido todas sus referencias a él, dicho objeto se marca para ser destruido, pero no se destruye inmediatamente. La memoria utilizada por el objeto se liberará cuando el recolector de basura o *garbage collector* destruya finalmente el objeto.

Java tiene un algoritmo de *recolección de basura* que se ejecuta como un hilo (subproceso) de forma paralela (concurrente) a nuestra aplicación, limpiando los objetos desreferenciados en segundo plano, liberando la memoria ocupada por ellos.

Finalizadores

El finalizador de una clase es lo contrario a un constructor. Se utiliza para especificar una serie de acciones para la finalización de un objeto. Puede no escribirse ninguno, pero en caso de escribirse, éste deberá ser único y llamarse `finalize()`. El finalizador siempre tendrá la siguiente sintaxis:

```
1 // Finalizador
2 public void finalize () {
3     ...
4 }
```

Permiten liberar recursos utilizados por objetos: arrays, cerrar archivos que pudieran estar abiertos, cerrar *sockets* de comunicación abiertos, etc. No se ejecuta cuando el objeto pierde una referencia, o cuando se marca para ser destruido, sino que se ejecuta justo antes de que el recolector de basura vaya a liberar definitivamente su memoria.

Sin embargo, puesto que el recolector de basura no se está ejecutando siempre de forma contínua, cabe la posibilidad de que el programa termine antes de que se hayan liberado ciertos objetos marcados para eliminar, es decir, que los finalizadores correspondientes no se lleguen a ejecutar.

Por ello, existen varias formas de forzar a que se ejecute el recolector de basura, aunque aun así, no se garantiza que se ejecuten realmente todos los finalizadores:

- `System.gc()`: solicitar al intérprete Java (JVM) que ejecute el recolector de basura para liberar la memoria de objetos pendientes de liberar. Invocar a `System.gc()` es exactamente lo mismo que invocar a `Runtime.getRuntime().gc()`.
- `System.runFinalization()`: solicitar al intérprete Java (JVM) que ejecute el finalizador de los objetos pendientes de ser liberados. Invocar a `System.runFinalization()` es exactamente lo mismo que invocar a `Runtime.getRuntime().runFinalization()`.
- `System.runFinalizersOnExit(true)`: habilitar (`true`) o deshabilitar (`false`) el que un programa intente ejecutar los finalizadores antes de terminar su ejecución (por defecto está deshabilitado). Este método es obsoleto (*deprecated*), ya que no es una operación segura, es decir, puede dejarse finalizadores sin ejecutar, ejecutar finalizadores de objetos vivos, o incluso producirse un interbloqueo entre los distintos hilos (*threads*).

Para ilustrar el uso de estos métodos, vamos a ampliar la clase `Circulo` con un finalizador (mostramos la clase completa). También utilizaremos una variable de clase compartida (`static`) `numObjetos` que ya habíamos añadido anteriormente, pero que aun no habíamos utilizado, la cual servirá para llevar la cuenta de cuantos objetos de esa clase se han creado: se incrementará en los constructores, y se decrementará en el finalizador. Notese que la variable `numObjetos` se inicializa a 0 por claridad, aunque no sería necesario ya que se trata de una variable de clase y se inicializa automaticamente a su valor por defecto que es 0. Notese también que para no repetir código en los constructores, ahora todos ellos invocan al que recibe dos argumentos mediante `this(...)`, y es ahí donde se incrementa la variable `numObjetos`, la cual se decrementa en el finalizador, imprimiéndose el valor en ese instante para saber si realmente se ejecuta o no el finalizador.

Copyright ©2012 P. Pablo Garrido Abenza

```
1  class Circulo {
2     final static double PI = 3.14159265D;   // Constante (compartida)
3     static int numObjetos = 0;              // Variable de clase (compartida)
4     String color;                           // Variable de instancia
5     double radio;                           // Variable de instancia
6
7     // Constructor por defecto
8     public Circulo () {
9        this ("verde", 15.0);
10    }
11
12    // Constructor conociendo el color (radio por defecto)
13    public Circulo (String color) {
14       this (color, 15.0);
15    }
16
17    // Constructor conociendo el radio (color por defecto)
18    public Circulo (double r) {
19       this ("verde", r);
20    }
21
22    // Constructor conociendo el color y el radio
23    public Circulo (String color, double r) {
24       this.color = color;
25       if (r >= 0.0) radio = r;
26       numObjetos++;
27       System.out.println ("Incremento de numObjetos = " + numObjetos);
28    }
29
30    // Finalizador
31    public void finalize () {
32       numObjetos--;
33       System.out.println ("Decremento de numObjetos = " + numObjetos);
34    }
35
36    // Cálculo del area de un circulo
37    double area () {
38       return (PI*radio*radio);    // PI por radio al cuadrado
39    }
40
41    // Comparacion de objetos (compactado)
42    public boolean equals (Object o) {
43       return (this.radio == ((Circulo) otro).radio);
44    }
45 }
```

Copyright © 2012 P. Pablo Garrido Abenza

Ahora escribimos un programa de prueba en el que creamos varios objetos y los destruimos, imprimiendo el valor de la variable compartida `numObjetos`. Al ser ésta una variable compartida, en vez de escribir `c1.numObjetos`, `c2.numObjetos`, ... (que sería válido), podemos escribir `Circulo.numObjetos`. Se supone que al finalizar la ejecución el resultado será 0, cosa que indicaría que los finalizadores se han ejecutado, y los correspondientes objetos se han liberado, pero comprobaremos que esto no es así siempre.

```
class TestCirculo1 {
   public final static void main (String args[]) {
      Circulo c1, c2, c3;

      // Creacion de objetos (incremento de numObjetos)
      c1 = new Circulo ("azul", 15.0);
      c2 = new Circulo ("rojo", 15.0);
      c3 = new Circulo ("verde", 15.0);

      // Destrucción de objetos (decremento de numObjetos?)
      c1 = null;
      c2 = null;
      c3 = null;

      System.out.println ("numObjetos = " + Circulo.numObjetos);
   }
}
```

El resultado de la ejecución del programa anterior es el siguiente, es decir, no se ejecutan los finalizadores (no aparece nunca el mensaje que se debería mostrar en ellos):

```
Incremento de numObjetos = 1
Incremento de numObjetos = 2
Incremento de numObjetos = 3
numObjetos = 3
```

Ahora le añadimos la línea `System.gc();` o `System.runFinalization();`, y se puede comprobar que sí. Sin embargo, como el recolector de basura es un hilo, y la asignación de procesador a los distintos hilos la controla el sistema operativo, es posible que el resultado sea distinto, dependiendo de en qué momento realmente se ejecute.

```
class TestCirculo1 {
   public final static void main (String args[]) {
      Circulo c1, c2, c3;

      // Creacion de objetos (incremento de numObjetos)
      c1 = new Circulo ("azul", 15.0);
      c2 = new Circulo ("rojo", 15.0);
```

Copyright © 2012 P. Pablo Garrido Abenza

```
        c3 = new Circulo ("verde", 15.0);

        // Destrucción de objetos (decremento de numObjetos?)
        c1 = null;
        c2 = null;
        c3 = null;

        System.out.println ("numObjetos = " + Circulo.numObjetos);
        // Forzamos ejecución del recolector de basura
        System.gc();
        //System.runFinalization();
        System.out.println ("numObjetos = " + Circulo.numObjetos);
    }
}
```

Lo siguiente son dos posibles resultados del programa, pero en ambos se ve que los finalizadores se han ejecutado. En la izquierda, el recolector de basura se ejecuta practicamente al invocarlo, antes del último `println()`, pero en la derecha, primero se hace el `println()` y luego se ejecutan los finalizadores:

```
Incremento de numObjetos = 1
Incremento de numObjetos = 2
Incremento de numObjetos = 3
numObjetos = 3
Decremento de numObjetos = 2
Decremento de numObjetos = 1
Decremento de numObjetos = 0
numObjetos = 0
```

```
Incremento de numObjetos = 1
Incremento de numObjetos = 2
Incremento de numObjetos = 3
numObjetos = 3
numObjetos = 3
Decremento de numObjetos = 2
Decremento de numObjetos = 1
Decremento de numObjetos = 0
```

Para finalizar, con la última opción `System.runFinalizersOnExit(true);` también funcionaría para este ejemplo, aunque para otros ejemplos más complejos podría no funcionar o funcionar incorrectamente.

```
class TestCirculo2 {
    public final static void main (String args[]) {
        Circulo c1, c2, c3;

        // Creacion de objetos (incremento de numObjetos)
        c1 = new Circulo ("azul", 15.0);
        c2 = new Circulo ("rojo", 15.0);
        c3 = new Circulo ("verde", 15.0);

        // Destrucción de objetos (decremento de numObjetos?)
        c1 = null;
        c2 = null;
        c3 = null;

        // Habilitamos la ejecucion de finalizadores antes de terminar
        System.runFinalizersOnExit (true);
        System.out.println ("numObjetos = " + Circulo.numObjetos);
```

Copyright © 2012 P. Pablo Garrido Abenza

```
18      }
19  }
```

En este caso se puede comprobar que cuando se habilita la ejecución de los finalizadores al salir sí se ejecutan, pero siempre al final, una vez finalizada siempre la función `main()`.

```
Incremento de numObjetos = 1
Incremento de numObjetos = 2
Incremento de numObjetos = 3
numObjetos = 3
Decremento de numObjetos = 2
Decremento de numObjetos = 1
Decremento de numObjetos = 0
```

CONSEJO: debido al uso problemático de los finalizadores en cuanto a que no se tiene la certeza de su ejecución no se recomienda utilizarlos, o al menos, no escribir código que sea crítico, como sería el caso de cerrar algún archivo (podrían perderse datos).

Copyright ©2012- P. Pablo Garrido Abenza

3.8. Encapsulación

La **encapsulación** es un mecanismo que nos permite ocultar información al exterior de una clase sobre las datos y detalles de implementación de la misma, por lo que también se denomina ocultación de información o abstracción. Se basa en la idea de modularidad o uso de módulos, que son bloques de software independientes pero que interaccionan entre si mediante algún mecanismo. La programación modular es anterior a la POO, pero ahora es mucho más clara pues cada módulo será una clase y tendrá una función concreta, ya que reflejará el comportamiento de un objeto de la realidad.

Abstracción es la representación de las características esenciales de algo sin incluir detalles irrelevantes, con el objeto de tener una visión global. A mayor nivel de abstracción (niveles altos de abstracción), tan solo se verá el interfaz con el exterior, es decir, las propiedades y métodos públicos (caja negra); a menor nivel de abstracción (niveles bajos de abstracción), podremos ver todos los detalles de la implementación.

En el desarrollo de una aplicación existirán diferentes personas implicadas, necesitando cada una de ellas un nivel de abstracción distinto. Por ejemplo, un programador necesitará una vista detallada de los atributos y métodos; en cambio, un analista junto al usuario final necesitarán una vista mucho más escueta de las mismas clases. En este contexto, el programador está trabajando a un menor nivel de abstracción, y el analista y el usuario final lo hacen a un nivel mayor, pero todos trabajan sobre el mismo modelo. En el caso de que exista un grupo de programadores que trabajen en equipo, normalmente cada uno trabajará en ciertas clases que luego compartirá con el resto del equipo. Mediante la encapsulación, el desarrollador de una clase puede proporcionar una descripción sobre el uso de dicha clase (interfaz), pero los detalles de la implementación permanecen encapsulados y ocultos a otros programadores.

Copyright ©2012 P. Pablo Garrido Abenza

Para conseguir la ocultación de información (encapsulación), Java dispone de dos mecanismos, aunque básicamente nos centraremos en el primero:

- Mediante el uso de modificadores de acceso.
- Mediante clases anidadas o internas.

Modificadores de acceso

El lenguaje Java permite especificar unos modificadores de acceso cuando se declara cualquier componente de una clase (variables, constantes, constructores, y métodos), los cuales se utilizan para decidir explícitamente qué componentes van a estar visibles y cuales ocultos, y desde dónde puede accederse (ver tabla 3.3). En resumen, podemos declarar un componente como:

- Público (`public`): será visible desde cualquier parte (la propia clase, subclases, y cualquier otras clases).
- Protegido (`protected`): será visible desde la propia clase, y subclases, pero no desde otras clases.
- Acceso a nivel de paquete (si no se especifica nada): será visible desde la propia clase, y subclases del mismo paquete.
- Privado (`private`): será visible sólo desde la propia clase.

Modificador	Misma clase	Subclases (mismo paquete)	Subclases (otro paquete)	Otras clases
`public`	Si	Si	Si	Si
`protected`	Si	Si	Si	
Nada (paquete)	Si	Si		
`private`	Si			

Cuadro 3.3: Modificadores de acceso

Clases anidadas o internas (*inner classes*)

Normalmente, el código de cada clase es independiente de las otras, pero una forma de conseguir ocultación de información consiste en escribir una clase dentro de otra. Esto es lo que se conoce como clase anidada o interna (*inner class*). Una clase interna tan sólo puede ser utilizada por la clase en la que reside; ninguna otra clase puede acceder a ella (no será visible).

El uso de clases anidadas (*inner classes*) permite simplificar el código en ocasiones: puesto que la clase interna puede referenciar los datos y métodos de la clase externa en la que está anidada, no es necesario pasar la referencia del objeto al constructor de la clase interna.

Se utilizan mucho para la captura de eventos en aplicaciones gráficas, en las llamadas clases adaptadoras anónimas. Cuando se llegue a dicho capítulo se mostrarán varios ejemplos.

Métodos setXXX() y getXXX()

Si una variable de instancia es pública (`public`), cualquier método desde fuera de la clase podrá leerla o establecerla a voluntad. Sin embargo, esto puede hacer que se establezcan valores incorrectos. Para evitar esto, es muy recomendable convertir los atributos en privados (`private`) y utilizar dos métodos para acceder a los valores, uno para establecer un nuevo valor (`setXXX()`) y otro para consultar dicho valor (`getXXX()`), donde `XXX` sería el nombre de la variable a establecer o consultar, respectivamente. No es necesario que estos métodos se llamen así, pero por convenio se suele hacer así. El acceso a datos privados (`private`) a través de estos métodos protege a las variables para que no reciban valores no válidos.

Copyright © 2012 P. Pablo Garrido Abenza

3.9. Estructura de un programa

Un programa Java se compone, salvo casos triviales, de más de una clase, tanto a nivel de código fuente (archivos .java) como código ya compilado (archivos .class). Esto difiere respecto otros lenguajes como C, que aunque el código fuente esté dividido en varios módulos (archivos .c), el ejecutable final se compone de un único archivo. Por ejemplo, en Windows un archivo ejecutable se distingue por la extensión .exe [4].

Todas las clases ya compiladas que componen un programa escrito en Java (archivos .class) normalmente estarán en un mismo directorio, aunque no tiene por qué. La única restricción es que estén localizables por medio de las rutas especificadas en la variable de entorno `CLASSPATH`.

Sin embargo, para simplificar la distribución de un programa (despliegue o *deployment*), conviene empaquetar todas las clases junto con otros recursos (iconos, sonidos, etc.) dentro un único archivo JAR (extensión .jar). Para ello puede utilizar la orden `jar` que se incluye en el mismo JDK, o bien, si se utiliza el entorno de desarrollo NetBeans, el archivo JAR se crea automáticamente cada vez que se compila la aplicación, almacenándose en el subdirectorio `\build\dist` de nuestro proyecto.

Copyright ©2012- P. Pablo Garrido Abenza

[4] Un ejecutable en Windows también puede hacer llamadas a archivos de librerías dinámicas o .dll.

GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

Copyright © 2012 P. Pablo Garrido Abenza

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere

Copyright ©2012- P. Pablo Garrido Abenza