4: Inicialización y limpieza

A medida que progresa la revolución computacional, la programación "insegura" se ha convertido en uno de los mayores culpables del encarecimiento de la programación.

Dos de estos aspectos de seguridad son la *inicialización* y la *limpieza*. Muchos de los fallos que se dan en C ocurren cuando el programador olvida inicializar una variable. Esto es especialmente habitual con las bibliotecas, cuando los usuarios no saben cómo inicializar un componente de una biblioteca, o incluso cuándo deben hacerlo. La limpieza o eliminación es un problema especial porque es fácil olvidarse de un elemento una vez que ya no se utiliza, puesto que ya no tiene importancia. Por consiguiente, los recursos que ese elemento utilizaba quedan reservados y es fácil acabar quedándose sin recursos (y el más importante, la memoria).

C++ introdujo el concepto de *constructor*, un método especial invocado automáticamente en la creación de un objeto. Java también adoptó el constructor, y además tiene un recolector de basura que libera automáticamente recursos de memoria cuando dejan de ser utilizados. Este capítulo examina los aspectos de inicialización y eliminación, y su soporte en Java.

Inicialización garantizada con el constructor

Es posible imaginar la creación de un método denominado **inicializar( )** para cada clase que se escriba. El nombre se debe invocar antes de utilizar el objeto. Por desgracia, esto significa que el usuario debe recordar llamar al método. En Java, el diseñador de cada clase puede garantizar que se inicialice cada objeto proporcionando un método especial llamado *constructor*. Si una clase tiene un constructor, Java llama automáticamente al constructor cuando se crea un objeto, antes de que los usuarios puedan siquiera pensar en poner sus manos en él. Por consiguiente, la inicialización queda garantizada.

El siguiente reto es cómo llamar a este método. Hay dos aspectos. El primero es que cualquier nombre que se use podría colisionar con un nombre que nos gustaría utilizar como miembro en una clase. El segundo es que dado que el compilador es el responsable de invocar al constructor, debe saber siempre qué método invocar. La solución de C++ parece la mejor y más lógica, por lo que se utiliza también en Java: el nombre del constructor es el mismo que el nombre de la clase. Tiene sentido que un método así se invoque automáticamente en la inicialización.

He aquí hay una clase con un constructor simple:

```
//: c04:ConstructorSimple.java
// Muestra de un constructor simple.
```

```
class Roca {
  Roca() { // Éste es el constructor
    System.out.println("Creando Roca");
  }
}

public class ConstructorSimple {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Roca();
  }
} ///:~
```

Ahora, al crear un objeto:

```
new Roca();
```

se asigna almacenamiento y se invoca al constructor. Queda garantizado que el objeto será inicializado de manera adecuada antes de poder poner las manos sobre él.

Fíjese que el estilo de codificación de hacer que la primera letra de todos los métodos sea minúscula no se aplica a los constructores, dado que el nombre del constructor debe coincidir *exactamente* con el nombre de la clase.

Como cualquier método, el constructor puede tener parámetros para permitir especificar *cómo* se crea un objeto. El ejemplo de arriba puede cambiarse sencillamente de forma que el constructor reciba un argumento:

```
//: c04:ConstructorSimple2.java
// Los constructores pueden tener parámetros.

class Roca2 {
  Roca2(int i) {
    System.out.println(
      "Creando la roca numero " + i);
  }
}

public class ConstructorSimple2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Roca2(i);
  }
} ///:~
```

Los parámetros del constructor proporcionan un medio para pasar parámetros a la inicialización de un objeto. Por ejemplo, si la clase **Arbol** tiene un constructor que toma un número entero que indica la altura del árbol, crearíamos un objeto **Arbol** como éste:

```
Arbol a = new Arbol(12); // Un árbol de 12 metros
```

Si **Arbol(int)** es el único constructor, entonces el compilador no permitirá crear un objeto **Arbol** de ninguna otra forma.

Los constructores eliminan un montón de problemas y simplifican la lectura del código. En el fragmento de código anterior, por ejemplo, no se verá ninguna llamada explícita a ningún método **inicializar( )** que esté conceptualmente separado, por definición. En Java, la definición e inicialización son conceptos que están unidos —no se puede tener uno sin el otro.

El constructor es un tipo inusual de método porque no tiene valor de retorno. Esto es muy diferente al valor de retorno **void**, en el que el método no devuelve nada pero se sigue teniendo la opción de hacer que devuelva algo más. Los constructores no devuelven nada y no es necesario tener ninguna opción. Si hubiera un valor de retorno, y si se pudiera seleccionar el propio, el compilador, de alguna manera, necesitaría saber qué hacer con ese valor de retorno.

Sobrecarga de métodos

Uno de los aspectos más importantes de cualquier lenguaje de programación es el uso de los nombres. Al crear un objeto, se da un nombre a cierta región de almacenamiento. Un método es un nombre que se asigna a una acción. Al utilizar nombres para describir el sistema, se crea un programa más fácil de entender y de modificar por la gente. Es como escribir en prosa —la meta es comunicarse con los lectores.

Para hacer referencia a objetos y métodos se usan nombres. Los nombres bien elegidos hacen más sencillo que todos entiendan un código.

Surge un problema cuando se trata de establecer una correspondencia entre el concepto de matiz del lenguaje humano y un lenguaje de programación. A menudo, la misma palabra expresa varios significados —se ha *sobrecargado*. Esto es útil, especialmente cuando incluye diferencias triviales. Se dice "lava la camisa", "lava el coche" y "lava el perro". Sería estúpido tener que decir "lavaCamisas la camisa", "lavaCoche el coche" y "lavaPerro el perro" simplemente para que el que lo escuche no tenga necesidad de intentar distinguir entre las acciones que se llevan a cabo. La mayoría de los lenguajes humanos son redundantes, por lo que incluso aunque se te olviden unas pocas palabras, se sigue pudiendo entender. No son necesarios identificadores únicos —se puede deducir el significado del contexto.

La mayoría de los lenguajes de programación (C en particular) exigen que se tenga un identificador único para cada función. Así, no se podría tener una función llamada **print( )** para imprimir enteros si existe ya otra función llamada **print( )** para imprimir decimales —cada función requiere un nombre único.

En Java (y C++) otros factores fuerzan la sobrecarga de los nombres de método: el constructor. Dado que el nombre del constructor está predeterminado por el nombre de la clase, sólo puede haber un nombre de constructor. Pero ¿qué ocurre si se desea crear un objeto de más de una manera? Por ejemplo, suponga que se construye una clase que puede inicializarse a sí misma de manera estándar o leyendo información de un archivo. Se necesitan dos constructores, uno que no tome argumentos (el constructor *por defecto*, llamado también constructor *sin parámetros*), y otro que tome como parámetro un **String**, que es el nombre del archivo con el cual inicializar el objeto. Ambos son constructores, por lo que deben tener el mismo nombre —el nombre de la clase. Por consiguiente, la *sobrecarga de métodos* es esencial para permitir que se use el mismo nombre de métodos con distintos tipos de parámetros. Y aunque la sobrecarga de métodos es una necesidad para los constructores, es bastante conveniente y se puede usar con cualquier método.

He aquí un ejemplo que muestra métodos sobrecargados, tanto constructores como ordinarios:

```
//: c04:Sobrecarga.java
// Muestra de sobrecarga de métodos
// tanto constructores como ordinarios.
import java.util.*;

class Arbol {
  int altura;
  Arbol() {
    visualizar("Plantando un retoño");
    altura = 0;
  }
  Arbol(int i) {
    visualizar("Creando un nuevo arbol que tiene "
        + i + " metros de alto");
    altura = i;
  }
  void info() {
    visualizar("El arbol tiene " + altura
        + " metros de alto");
  }
  void info(String s) {
    visualizar(s + ": El arbol tiene "
        + altura + " metros de alto");
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
}

public class sobrecarga {
  public static void main(String[] args) {
```

```
    for(int i = 0; i < 5; i++) {
      Arbol t = new Arbol(i);
      t.info();
      t.info("metodo sobrecargado");
    }
    // Constructor sobrecargado:
    new Arbol();
  }
} ///:~
```

Se puede crear un objeto **Arbol,** bien como un retoño, sin argumentos, o como una planta que crece en un criadero, con una altura ya existente. Para dar soporte a esto, hay dos constructores, uno que no toma argumentos (a los constructores sin argumentos se les llama *constructores por defecto*[1]), y uno que toma la altura existente.

Podríamos también querer invocar al método **info( )** de más de una manera. Por ejemplo, con un parámetro **String** si se tiene un mensaje extra para imprimir, y sin él si no se tiene nada más que decir. Parecería extraño dar dos nombres separados a lo que es obviamente el mismo concepto. Afortunadamente, la sobrecarga de métodos permite usar el mismo nombre para ambos.

Distinguir métodos sobrecargados

Si los métodos tienen el mismo nombre, ¿cómo puede saber Java qué método se debe usar en cada caso? Hay una regla simple: cada método sobrecargado debe tomar una única lista de tipos de parámetros.

Si se piensa en esto por un segundo, tiene sentido: ¿de qué otra forma podría un programador distinguir entre dos métodos que tienen el mismo nombre si no fuera por los tipos de parámetros?

Incluso las diferencias en el orden de los parámetros son suficientes para distinguir ambos métodos: (Aunque normalmente este enfoque no es necesario, pues produce un código difícil de mantener.)

```
//: c04:OrdenSobrecarga.java
// Sobrecarga basada en el
// orden de los parámetros.

public class OrdenSobrecarga {
  static void print(String s, int i) {
    System.out.println(
      "cadena: " + s +
      ", entero: " + i);
  }
```

[1] En algunos documentos sobre Java de Sun, por el contrario, se refieren a éstos con el poco elegante pero descriptivo nombre de "constructores sin parámetros". El término "constructor por defecto" se ha usado durante muchos años, por lo que será el que utilizaremos.

```
  static void print(int i, String s) {
    System.out.println(
      "Entero: " + i +
      ", Cadera: " + s);
  }
  public static void main(String[] args) {
    print("Primero cadena", 11);
    print(99, "Primero entero");
  }
} ///:~
```

Ambos métodos **print( )** tienen los mismos argumentos, pero distinto orden, y eso es lo que los hace diferentes.

Sobrecarga con tipos primitivos

Un tipo primitivo puede ser promocionado automáticamente de un tipo menor a otro mayor, y esto puede ser ligeramente confuso si se combina con la sobrecarga. El ejemplo siguiente demuestra lo que ocurre cuando se pasa un tipo primitivo a un método sobrecargado:

```
//: c04:SobrecargaPrimitivo.java
// Promoción de tipos primitivos y sobrecarga.

public class SobrecargaPrimitivo {
  // los booleanos no pueden convertirse automáticamente
  static void visualizar(String s) {
    System.out.println(s);
  }

  void f1(char x) { visualizar("f1(char)"); }
  void f1(byte x) { visualizar("f1(byte)"); }
  void f1(short x) { visualizar("f1(short)"); }
  void f1(int x) { visualizar("f1(int)"); }
  void f1(long x) { visualizar("f1(long)"); }
  void f1(float x) { visualizar("f1(float)"); }
  void f1(double x) { visualizar("f1(double)"); }

  void f2(byte x) { visualizar("f2(byte)"); }
  void f2(short x) { visualizar("f2(short)"); }
  void f2(int x) { visualizar("f2(int)"); }
  void f2(long x) { visualizar("f2(long)"); }
  void f2(float x) { visualizar("f2(float)"); }
  void f2(double x) { visualizar("f2(double)"); }

  void f3(short x) { visualizar("f3(short)"); }
```

```
void f3(int x) { visualizar("f3(int)"); }
void f3(long x) { visualizar("f3(long)"); }
void f3(float x) { visualizar("f3(float)"); }
void f3(double x) { visualizar("f3(double)"); }

void f4(int x) { visualizar("f4(int)"); }
void f4(long x) { visualizar("f4(long)"); }
void f4(float x) { visualizar("f4(float)"); }
void f4(double x) { visualizar("f4(double)"); }

void f5(long x) { visualizar("f5(long)"); }
void f5(float x) { visualizar("f5(float)"); }
void f5(double x) { visualizar("f5(double)"); }

void f6(float x) { visualizar("f6(float)"); }
void f6(double x) { visualizar("f6(double)"); }

void f7(double x) { visualizar("f7(double)"); }

void pruebaValoresConstante() {
  visualizar("Probando con el 5");
  f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}
void pruebaChar() {
  char x = 'x';
  visualizar("parametro char:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaByte() {
  byte x = 0;
  visualizar("parametro byte:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaShort() {
  short x = 0;
  visualizar("parametro short:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaInt() {
  int x = 0;
  visualizar("parametro int:");
  f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void pruebaLong() {
  long x = 0;
```

```
    visualizar("parametro long:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void pruebaFloat() {
    float x = 0;
    visualizar("parametro float:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void pruebaDouble() {
    double x = 0;
    visualizar("parametro double:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  public static void main(String[] args) {
    SobrecargaPrimitivo p =
      new SobrecargaPrimitivo();
    p.pruebaValoresConstante();
    p.pruebaChar();
    p.pruebaByte();
    p.pruebaShort();
    p.pruebaInt();
    p.pruebaLong();
    p.pruebaFloat();
    p.pruebaDouble();
  }
} ///:~
```

Si se observa la salida de este programa, se verá que el valor constante 5 se trata como un **int**, de forma que si hay disponible un método sobrecargado que tome un **int**, será el utilizado. En todos los demás casos, si se tiene un tipo de datos menor al parámetro del método, ese tipo de dato será promocionado. Un dato de tipo **char** produce un efecto ligeramente diferente, pues si no encuentra una coincidencia exacta de **char**, se promociona a **int**.

¿Qué ocurre si el parámetro es *mayor* que el que espera el método sobrecargado? La respuesta la proporciona una modificación del programa anterior:

```
//: c04:Degradacion.java
// Degradación de tipos primitivos y sobrecarga.

public class Degradacion {
  static void visualizar(String s) {
    System.out.println(s);
  }

  void f1(char x) { visualizar("f1(char)"); }
  void f1(byte x) { visualizar("f1(byte)"); }
```

```
  void f1(short x) { visualizar("f1(short)"); }
  void f1(int x) { visualizar("f1(int)"); }
  void f1(long x) { visualizar("f1(long)"); }
  void f1(float x) { visualizar("f1(float)"); }
  void f1(double x) { visualizar("f1(double)"); }

  void f2(char x) { visualizar("f2(char)"); }
  void f2(byte x) { visualizar("f2(byte)"); }
  void f2(short x) { visualizar("f2(short)"); }
  void f2(int x) { visualizar("f2(int)"); }
  void f2(long x) { visualizar("f2(long)"); }
  void f2(float x) { visualizar("f2(float)"); }

  void f3(char x) { visualizar("f3(char)"); }
  void f3(byte x) { visualizar("f3(byte)"); }
  void f3(short x) { visualizar("f3(short)"); }
  void f3(int x) { visualizar("f3(int)"); }
  void f3(long x) { visualizar("f3(long)"); }

  void f4(char x) { visualizar("f4(char)"); }
  void f4(byte x) { visualizar("f4(byte)"); }
  void f4(short x) { visualizar("f4(short)"); }
  void f4(int x) { visualizar("f4(int)"); }

  void f5(char x) { visualizar("f5(char)"); }
  void f5(byte x) { visualizar("f5(byte)"); }
  void f5(short x) { visualizar("f5(short)"); }

  void f6(char x) { visualizar("f6(char)"); }
  void f6(byte x) { visualizar("f6(byte)"); }

  void f7(char x) { visualizar("f7(char)"); }

  void pruebaDouble() {
    double x = 0;
    visualizar("parametro double:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Degradacion p = new Degradacion();
    p.pruebaDouble();
  }
} ///:~
```

Aquí, los métodos toman valores primitivos de menor tamaño. Si el parámetro es de mayor tamaño es necesario *convertir* el parámetro al tipo necesario poniendo entre paréntesis el nombre del tipo. Si no se hace esto, el compilador mostrará un mensaje de error.

Uno debería ser consciente de que ésta es una *conversión reductora*, que significa que podría conllevar una pérdida de información durante la conversión. Éste es el motivo por el que el compilador obliga a hacerlo —para marcar la conversión reductora.

Sobrecarga en los valores de retorno

Es común preguntarse "¿Por qué sólo los nombres de las clases y las listas de parámetros de los métodos? ¿Por qué no distinguir entre métodos basados en sus valores de retorno?" Por ejemplo, estos dos métodos, que tienen el mismo nombre y parámetros, se distinguen fácilmente el uno del otro:

```
void f() {}
int f() {}
```

Esto funciona bien cuando el compilador puede determinar de manera inequívoca el significado a partir del contexto, como en **int x = f( )**. Sin embargo, se puede llamar a un método e ignorar el valor de retorno; a esto se le suele llamar *invocar a un método por su efecto lateral*, dado que no hay que tener cuidado sobre el valor de retorno y sí desear los otros efectos de la llamada al método. Por tanto, si se llama al método de la siguiente manera:

```
f();
```

¿cómo puede determinar Java qué **f( )** invocar? ¿Y cómo podría alguien más que lea el código verlo también? Debido a este tipo de problemas, no se pueden usar los tipos de valores de retorno para distinguir los métodos sobrecargados.

Constructores por defecto

Como se mencionó anteriormente, un constructor por defecto (un constructor sin parámetros) es aquél que no tiene parámetros, y se utiliza para crear un "objeto básico". Si se crea una clase que no tiene constructores, el compilador siempre creará un constructor por defecto. Por ejemplo:

```
//: c04:ConstructorPorDefecto.java

class Pajaro {
  int i;
}

public class ConstructorPorDefecto {
  public static void main(String[] args) {
    Pajaro nc = new Pajaro(); // ¡por defecto!
  }
} ///:~
```

La línea

```
new Pajaro();
```

crea un objeto nuevo e invoca a la función constructor, incluso aunque ésta no se haya definido explícitamente. Sin ella no habría ningún método a invocar para construir el objeto. Sin embargo, si se define algún constructor (con o sin parámetros) el compilador *no* creará uno automáticamente:

```
class Arbusto {
      Arbusto (int i) {}
      Arbusto (double d) {}
}
```

Ahora, si se escribe

```
new Arbusto();
```

el compilador se quejará por no poder encontrar un constructor que coincida. Es como si no se pusiera ningún constructor, y el compilador dice "Debes necesitar *algún* constructor, por lo que crearé uno". Pero si escribes un constructor, el compilador dice "Has escrito un constructor por lo que ya sabes lo que estás haciendo; si no hiciste un constructor por defecto es porque no lo necesitabas".

La palabra clave **this**

Si se tiene dos objetos del mismo tipo llamados **a** y **b**, nos pondríamos preguntar cómo es que se puede invocar a un método **f( )** para ambos objetos:

```
class Platano { void f (int i) { /* ... */ } }
Platano a = new Platano(), b = new Platano();
a.f(1);
b.f(2);
```

Si sólo hay un método llamado **f( )**, ¿cómo puede este método saber si está siendo invocado para el objeto **a** o **b**?

Para permitir la escritura de código con una sintaxis adecuada orientada a objetos en la que "se envía un menaje a un objeto", el compilador se encarga del código clandestino. Hay un primer parámetro secreto que se pasa al método **f( )**, y ese parámetro es la referencia al objeto que está siendo manipulado. Por tanto, las dos llamadas a método anteriores, se convierten en algo parecido a:

```
Platano.f(a,1);
Platano.f(b,2);
```

Esto es interno y uno no puede escribir estas expresiones y hacer que el compilador las acepte, pero da una idea de lo que está ocurriendo.

Supóngase que uno está dentro de un método y que desea conseguir la referencia al objeto actual. Dado que esa referencia se pasa *de forma secreta* al compilador, no hay un identificador para él. Sin embargo, para este propósito hay una palabra clave: **this**. Esta palabra clave —que puede usarse sólo dentro de un método— produce la referencia al objeto por el que se ha invocado al método. Uno puede tratar esta referencia como cualquier otra referencia a un objeto. Hay que recordar que si se está invocando a un método de una clase desde dentro de un método de esa misma clase, no es necesario utilizar **this**; uno puede simplemente invocar al método. Por consiguiente, se puede decir:

```
Class Albaricoque{
      void tomar() { /* ... */ }
      void deshuesar() { tomar(); /* ... */}
}
```

Dentro de **deshuesar( )**, uno *podría* decir **this.tomar( ),** pero no hay ninguna necesidad. El compilador lo hace automáticamente. La palabra clave **this** sólo se usa para aquellos casos especiales en los que es necesario utilizar explícitamente la referencia al objeto actual. Por ejemplo, se usa a menudo en sentencias **return** cuando se desea devolver la referencia al objeto actual:

```
//: c04:Hoja.java
// Utilización simple de la palabra clave "this".

public class Hoja {
  int i = 0;
  Hoja incrementar() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Hoja x = new Hoja();
    x.incrementar().incrementar().incrementar().print();
  }
} ///:~
```

Dado que **incrementar( )** devuelve la referencia al objeto actual, a través de la palabra clave **this**, pueden ejecutarse múltiples operaciones con el mismo objeto.

Invocando a constructores desde constructores

Cuando se escriben varios constructores para una clase, hay veces en las que uno quisiera invocar a un constructor desde otro para evitar la duplicación de código. Esto se puede lograr utilizando la palabra clave **this**.

Normalmente, cuando se dice **this**, tiene el sentido de "este objeto" o "el objeto actual", y por sí mismo produce la referencia al objeto actual. En un constructor, la palabra clave **this** toma un significado diferente cuando se le da una lista de parámetros: hace una llamada explícita al constructor que coincida con la lista de parámetros. Por consiguiente, hay una manera directa de llamar a otros constructores:

```
//: c04:Flor.java
// Invocación a constructores con "this".

public class Flor {
  int numeroPetalos = 0;
  String s = new String("null");
  Flor(int petalos) {
    numeroPetalos = petalos;
    System.out.println(
      "Constructor w/ parametro entero solo, Numero de petalos = "
      + numeroPetalos);
  }
  Flor(String ss) {
    System.out.println(
      "Constructor w/ parametro cadera solo, s=" + ss);
    s = ss;
  }
  Flor(String s, int petalos) {
    this(petalos);
//!    this(s); // ¡No se puede invocar dos!
    this.s = s; // Otro uso de "this"
    System.out.println("cadena y entero Parámetros");
  }
  Flor() {
    this("Hola", 47);
    System.out.println(
      "constructor por defecto (sin parametros)");
  }
  void print() {
//!    this(11); // ;No dentro de un no-constructor!
    System.out.println(
      "Numero de Petalos = " + numeroPetalos + " s = "+ s);
  }
  public static void main(String[] args) {
    Flor x = new Flor();
    x.print();
  }
} ///:~
```

El constructor **Flor(String s, int petalos)** muestra quese puede invocar a un constructor utilizando **this**, pero no a dos. Además, la llamada al constructor debe ser la primera cosa que se haga o se obtendrá un mensaje de error del compilador.

El ejemplo también muestra otra manera de ver el uso de **this**. Dado que el nombre del parámetro **s** y el nombre del atributo **s** son el mismo, hay cierta ambigüedad. Se puede resolver diciendo **this.s** para referirse al dato miembro. A menudo se verá esta forma en código Java, que también se usa en muchas partes de este libro.

En **print( )** se puede ver que el compilador no permite invocar a un constructor desde dentro de otro método que no sea un constructor.

El significado de estático (static)

Teniendo en cuenta la palabra clave this, uno puede comprender completamente qué significa hacer un método **estático**. Significa que no hay un **this** para ese método en particular. No se puede invocar a métodos no **estático** desde dentro de métodos **estáticos**[2] (aunque al revés sí que es posible), y se puede invocar al método **estático** de la propia clase, sin objetos. De hecho, esto es principalmente el fin de un método **estático**. Es como si se estuviera creando el equivalente a una función global (de C). La diferencia es que las funciones globales están prohibidas en Java, y poner un método **estático** dentro de una clase permite que ésta acceda a otros métodos **estáticos** y a campos **estáticos**.

Hay quien discute que los métodos **estáticos** no son orientados a objetos, puesto que tienen la semántica de una función global; con un método **estático** no se envía un mensaje a un objeto, puesto que no hay **this**. Esto probablemente es un argumento justo, y si uno acaba usando *un montón* de métodos estáticos, seguro que tendrá que replantearse su estrategia. Sin embargo, los métodos **estáticos** son pragmáticos y hay veces en las que son genuinamente necesarios, por lo que el hecho de que sean o no "POO pura" se deja para los teóricos. Sin duda, incluso Smalltalk tiene un equivalente en sus "métodos de clase ".

Limpieza: finalización y recolección de basura

Los programadores conocen la importancia de la inicialización, pero a menudo se les olvida la importancia de la limpieza. Después de todo, ¿quién necesita eliminar un **int**? Pero con las bibliotecas, dejar que un objeto simplemente "se vaya" una vez que se ha acabado con él, no es siempre seguro. Por supuesto, Java tiene el recolector de basura para recuperar la memoria de los objetos que ya no se usan. Considere ahora un caso muy inusual. Supóngase que los objetos asignan memoria "especial" sin utilizar **new**. El recolector de basura sólo sabe liberar la memoria asignada con **new**, por lo que ahora no sabrá cómo liberar esa memoria "especial" del objeto. Para hacer frente a este caso, Java proporciona un método denominado **finalize( )** que se puede definir en cada clase. He aquí cómo se *supone* que funciona. Cuando el recolector de basura está preparado para liberar el espacio de almacenamiento uti-

[2] El único caso en el que esto podría ocurrir es si se pasa una referencia a un objeto dentro del método **estático**. Después, a través de la referencia (que ahora es **this**) se puede invocar a métodos no **estáticos** y acceder a campos no **estáticos**. Pero generalmente si se desea hacer algo así, simplemente se hará un método ordinario no **estático**.

lizado por el objeto, primero invocará a **finalize( )**, y sólo recuperará la memoria del objeto durante la pasada del recolector de basura. Por tanto, si se elige usar **finalize( )**, éste te proporciona la habilidad de llevar a cabo alguna limpieza importante *a la vez que la recolección de basura.*

Éste es un error potencial de programación porque algunos programadores, especialmente los de C++, podrían confundir **finalize( )** con el *destructor* de C++, que es una función que siempre se invoca cuando se destruye un objeto. Pero es importante distinguir entre C++ y Java en este caso, pues en C++ *los objetos siempre se destruyen* (en un programa sin errores), mientras que los objetos de Java no siempre son eliminados por el recolector. O, dicho de otra forma:

La recolección de basura no es destrucción

Si se recuerda esto, se evitarán los problemas. Lo que significa es que si hay alguna actividad que debe llevarse a cabo antes de que un objeto deje de ser necesario, hay que llevar a cabo esa actividad por uno mismo. Java no tiene un destructor o un concepto similar, por lo que hay que crear un método ordinario para hacer esta limpieza. Por ejemplo, supóngase que en el proceso de creación de un objeto, éste se dibuja a sí mismo en la pantalla. Si no se borra explícitamente esta imagen de la pantalla, podría ser que éste no se elimine nunca. Si se pone algún tipo de funcionalidad eliminadora dentro de **finalize( )**, si un objeto es eliminado por el recolector de basura, la imagen será eliminada en primer lugar de la pantalla, pero si no lo es, la imagen permanecerá. Por tanto, un segundo punto a recordar es:

Los objetos podrían no ser eliminados por el recolector de basura

Uno podría averiguar que el espacio de almacenamiento de un objeto nunca se libera porque el programa nunca llega a quedarse sin espacio de almacenamiento. Si el programa se completa y el recolector de basura nunca llega a ejecutarse para liberar el espacio de almacenamiento de ningún objeto, éste será devuelto por completo al sistema operativo en el momento en que acaba el programa. Esto es bueno, porque el recolector de basura tiene algo de sobrecarga, y si nunca se ejecuta, no hay que incurrir en ese gasto.

¿Para qué sirve **finalize( )**?

Uno podría pensar en este punto que no deberíamos utilizar **finalize( )** como un método de limpieza de propósito general. ¿Cómo de bueno es?

Un tercer punto para recordar es:

La recolección de basura sólo tiene que ver con la memoria

Es decir, la única razón para la existencia de un recolector de basura, es recuperar la memoria que un programa ha dejado de utilizar. Por tanto, cualquier actividad asociada a la recolección de basura, especialmente el método **finalize( )** debe estar relacionada también sólo con la memoria y su desasignación.

¿Significa esto que si un objeto contiene otros objetos **finalize( )** debería liberar explícitamente esos objetos? Pues no —el recolector de basura cuida de la liberación de toda la memoria de los objetos independientemente de cómo se creará el objeto. Resulta que la necesidad de **finalize( )** se limita a casos especiales, en los que un objeto puede reservar espacio de almacenamiento de forma distinta a la creación de un objeto. Pero, podríamos pensar: en Java todo es un objeto, así que ¿cómo puede ser?

Parecería que **finalize( )** tiene sentido debido a la posibilidad de que se haga algo de estilo C, asignando memoria utilizando un mecanismo distinto al normal de Java. Esto puede ocurrir principalmente a través de *métodos nativos*, que son la forma de invocar a código no-Java desde Java. (Los métodos nativos se discuten en el Apéndice B.) C y C++ son los únicos lenguajes actualmente soportados por los métodos nativos, pues dado que pueden llamar a subprogramas escritos en otros lenguajes, pueden efectivamente invocar a cualquier cosa. Dentro del código no-Java, se podría invocar a la familia de funciones de **malloc( )** de C para asignar espacio de almacenamiento, provocando una pérdida de memoria. Por supuesto, **free( )** es una función de C y C++, por lo que sería necesario invocarla en un método nativo desde el **finalize( )**.

Después de leer esto, probablemente se tendrá la idea de que no se usará mucho **finalize( )**. Es correcto: no es el sitio habitual para que ocurra una limpieza normal. Por tanto, ¿dónde debería llevarse a cabo la limpieza normal?

Hay que llevar a cabo la limpieza

Para eliminar un objeto, el usuario debe llamar a un método de limpieza en el punto en el que se desee. Esto suena bastante directo, pero colisiona un poco con el concepto de destructor de C++. En este lenguaje, se destruyen todos los objetos. O mejor dicho, *deberían* eliminarse todos los objetos. Si se crea el objeto C++ como local (por ejemplo, en la pila —lo cual no es posible en Java), la destrucción se da al cerrar la llave del ámbito en el que se ha creado el objeto. Si el objeto se creó usando **new** (como en Java) se llama al destructor cuando el programador llame al operador **delete** de C++ (que no existe en Java). Si el programador de C++ olvida invocar a **delete**, no se llama nunca al destructor, y se tiene un fallo de memoria, y además las otras partes del objeto no se borran nunca. Este tipo de fallo suele ser muy difícil de localizar.

Por el contrario, Java no permite crear objetos locales —siempre hay que usar **new**. Pero en Java, no hay un "eliminar" al que invocar para liberar el objeto, dado que el recolector de basura se encarga de liberar el espacio de almacenamiento. Por tanto, desde un punto de vista simplista, se podría decir que por culpa del recolector de basura, Java no tiene destructor. Se verá a medida que se vaya avanzando en el libro, que la presencia de un recolector de basura no elimina la necesidad de, o la utilidad de los destructores (y no se debería invocar a **finalize( )** directamente, pues ésta no es la solución más adecuada). Si se desea llevar a cabo algún tipo de limpieza distinta a la liberación de espacio de almacenamiento, hay que *seguir* llamando explícitamente al método apropiado en Java, que es el equivalente al destructor de C++ , sea o no lo más conveniente.

Una de las cosas para las que puede ser útil **finalize( )** es para observar el proceso de recolección de basura. El ejemplo siguiente resume las descripciones anteriores del recolector de basura:

```
//: c04:Basura.java
// Demostración de recolector de
// basura y finalización

class Silla {
  static boolean ejecrecol = false;
  static boolean f = false;
  static int creadas = 0;
  static int finalizadas = 0;
  int i;
  Silla() {
    i = ++creadas;
    if(creadas == 47)
      System.out.println("Creadas 47");
  }
  public void finalize() {
    if(!ejecrecol) {
      // La primera vez se invoca a finalize():
      ejecrecol = true;
      System.out.println(
        "Comenzando a finalizar tras haber creado " +
        creadas + " sillas");
    }
    if(i == 47) {
      System.out.println(
        "Finalizando la silla #47, " +
        "Poniendo el indicada que evita la creacion de mas sillas");
      f = true;
    }
    finalizadas++;
    if(finalizadas >= creadas)
      System.out.println(
        "Las " + finalizadas + " han sido finalizadas");
  }
}

public class Basura {
  public static void main(String[] args) {
    // Mientras no se haya puesto el flag,
    // hacer sillas y cadenas de texto:
    while(!Silla.f) {
      new Silla();
      new String("Coger cspacio");
    }
    System.out.println(
      "Despues de haber creado todas las sillas:\n" +
```

```
        "creadas en total = " + Silla.creadas +
        ", finalizadas total = " + Silla.finalizadas);
      // Parámetros opcionales fueran la recolección
      // de basura y finalización:
      if(args.length > 0) {
        if(args[0].equals("rec") ||
           args[0].equals("todo")) {
          System.out.println("gc():");
          System.gc();
        }
        if(args[0].equals("finalizar") ||
           args[0].equals("todo")) {
          System.out.println("runFinalization():");
          System.runFinalization();
        }
      }
      System.out.println("adios!");
    }
} ///:~
```

El programa anterior crea muchos objetos **Silla**, y en cierto momento después de que el recolector de basura comience a ejecutarse, el programa deja de crear objetos de tipo **Silla**. Dado que el recolector de basura puede ejecutarse en cualquier momento, uno no sabe exactamente cuando empezará, y hay un indicador denominado **ejecrecol** para indicar si el recolector de basura ha comenzado ya su ejecución o no. Un segundo indicador **f** es la forma de que **Silla** le comunique al bucle **main( )** que debería dejar de hacer objetos. Ambos indicadores se ponen dentro de **finalize( )**, que se invoca durante la recolección de basura.

Otras dos variables **estáticas**, **creadas** y **finalizadas**, mantienen el seguimiento del número de objetos de tipo **Silla** creadas frente al número de finalizadas por el recolector de basura. Finalmente, cada **Silla** tiene su propio (no **estático**) **int i**, por lo que se hace un seguimiento de qué número es. Cuando finalice la **Silla** número 47, el indicador se pone a **true** para detener el proceso de creación de objetos de tipo **Silla**.

Todo esto ocurre en el método **main( )**, en el bucle

```
while(!Silla.f) {
      new Silla( );
      new String("Coger espacio");
}
```

Uno podría preguntarse cómo conseguir finalizar este bucle, dado que no hay nada dentro del bucle que cambie el valor de **Silla.f**. Sin embargo, el proceso **finalize( )** se supone que lo hará cuando finalice el número 47.

La creación de un objeto **String** en cada iteración es simplemente la asignación de almacenamiento extra para animar al recolector de basura a actuar, lo que hará cuando se empiece a poner nervioso por la cantidad de memoria disponible.

Cuando se ejecute el programa, se proporciona un parámetro de línea de comandos que pueden ser "rec", "finalizar" o "todo". El argumento "rec" invocará al método **System.gc( )** (para forzar la ejecución del recolector de basura). La utilización del parámetro "finalizar" invoca a **System.runFinalization( )** que —en teoría— hará que finalicen los objetos que no lo hayan hecho. Y "todo" hace que se llame a los dos métodos.

El comportamiento de este programa y de la versión de la primera edición de este libro muestra que todo lo relacionado con el recolector de basura y la finalización ha evolucionado, habiendo ocurrido mucha de esta evolución detrás del telón. De hecho, para cuando se lea esto, puede que el comportamiento del programa haya vuelto a cambiar.

Si se invoca a **System.gc( )**, se finalizan todos los objetos. Esto no era necesario en el caso de las implementaciones previas del JDK, aunque la documentación decía otra cosa. Además, se verá que no parece haber ninguna diferencia si se invoca o no a **System.runFinalization( )**.

Sin embargo, se verá que sólo si se invoca a **System.gc( )** después de crear y descartar todos los objetos se invocará a todos los finalizadores. Si no se invoca a **System.gc( )**, entonces sólo se finalizan algunos de los objetos. En Java 1.1, se introdujo un método **System.runFinalizersOnExit( )** que hacía que los programas ejecutaran todos los finalizadores al salir, pero el diseño resultó tener errores y se desechó el método. Esto puede ser otro de los motivos por los que los diseñadores de Java siguen dándole vueltas al problema de la recolección de basura y la finalización. Esperamos que este asunto se termine de resolverse en Java 2.

El programa precedente muestra que la promesa de que todos los finalizadores se ejecuten siempre es verdadera, pero sólo uno fuerza explícitamente el que suceda. Si no se fuerza la invocación a **System.gc( )**, se logra una salida como:

```
Creadas 47
Comenzando a finalizar tras haber creado 3486
Finalizando la silla #47
Poniendo el indicador que evita la creacion de mas sillas
Despues de haber creado todas las sillas:
total creadas = 3881, total finalizadas = 2684
adios!
```

Por consiguiente, no se invoca a todos los finalizadores cuando acaba el programa. Si se llama a **System.gc( )**, acabará y destruirá todos los objetos que no estén en uso en ese momento.

Recuérdese que ni el recolector de basura ni la finalización están garantizadas. Si la Máquina Virtual Java (JVM) no está a punto de quedarse sin memoria, entonces (sabiamente) no malgastará tiempo en recuperar memoria mediante el recolector de basura.

La condición de muerto

En general, no se puede confiar en que se invoque a **finalize( )**, y es necesario crear funciones de "limpieza" aparte e invocarlas explícitamente. Por tanto, parece que **finalize( )** solamente es útil para limpiezas oscuras de memoria que la mayoría de programadores nunca usarán. Sin embargo,

hay un uso muy interesante de **finalize( )** que no confía en ser invocada siempre. Se trata de la verificación de la *condición de muerte*[3] de un objeto.

En el momento en que uno deja de estar interesado en un objeto —cuando está listo para ser eliminado— el objeto debería estar en cierto estado en el que su memoria pueda ser liberada de manera segura. Por ejemplo, si el objeto representa un fichero abierto, ese fichero debería ser cerrado por el programador antes de que el objeto sea eliminado por el recolector de basura. Si no se eliminan correctamente ciertas porciones del objeto, se tendrá un fallo en el programa que podría ser difícil de encontrar. El valor de **finalize( )** es que puede usarse para descubrir esta condición, incluso si no se invoca siempre. Si una de las finalizaciones acaba revelando el fallo, se descubre el problema, que es de lo que verdaderamente hay que cuidar.

He aquí un ejemplo simple de cómo debería usarse:

```
//: c04:CondicionMuerte.java
// Utilización de finalize() para detectar un
// objeto que no ha sido eliminado correctamente.

class Libro {
  boolean comprobado = false;
  Libro(boolean comprobar) {
    comprobado = comprobar;
  }
  void correcto() {
    comprobado = false;
  }
  public void finalize() {
    if(comprobado)
      System.out.println("Error: comprobado");
  }
}

public class CondicionMuerte {
  public static void main(String[] args) {
    Libro novela = new Libro(true);
    // Eliminación correcta:
    novela.correcto();
    // Cargarse la referencia, olvidando la limpieza:
    new Libro(true);
    // Forzar la recolección de basura y finalización:
    System.gc();
  }
} ///:~
```

La condición de muerte consiste en que todos los objetos **Libro** supuestamente serán comprobados antes de ser recogidos por el recolector de basura, pero en el método **main( )** un error del pro-

[3] Un término acuñado por Hill Venners *(www.artima.com)* durante un seminario que él y yo impartimos conjuntamente.

gramador no comprueba alguno de los libros. Sin **finalize( )** para verificar la condición de muerte, este error podría ser difícil de encontrar.

Nótese que se usa **System.gc( )** para forzar la finalización (y se debería hacer esto durante el desarrollo del programa para forzar la depuración). Pero incluso aunque no se use, es muy probable descubrir objetos de tipo **Libro** errantes a lo largo de ejecuciones repetidas del programa (asumiendo que el programa asigna suficiente espacio de almacenamiento para hacer que se ejecute el recolector de basura).

Cómo funciona un recolector de basura

Si se realiza en un lenguaje de programación en el que la asignación de objetos en el montículo es cara, hay que asumir naturalmente que el esquema de Java de asignar todo (excepto los datos primitivos) en el montículo es caro.

Sin embargo, resulta que el recolector de basura puede tener un impacto significativo en un *incremento* de la velocidad de creación de los objetos. Esto podría sonar un poco extraño al principio —que la liberación de espacio de almacenamiento afecte a la asignación de espacio— pero es la manera en que trabajan algunas JVM, y significa que la asignación de espacio para objetos del montículo en Java pueda ser casi tan rápida como crear espacio de almacenamiento en la pila en otros lenguajes.

Por ejemplo, se puede pensar que el montículo de C++ es como un terreno en el que cada objeto toma un fragmento de suelo. Puede ser que este espacio sea abandonado tiempo después, y haya que reutilizarlo. En algunas JVM, el montículo de Java es bastante distinto; es más como una cinta transportadora que avanza cada vez que se asigna un nuevo objeto. Esto significa que la asignación de espacio de almacenamiento de los objetos es notoriamente rápida. El "puntero del montículo" simplemente se mueve hacia delante en territorio virgen, así que es exactamente lo mismo que la asignación de pila de C++. (Por supuesto, hay una pequeña sobrecarga por el mantenimiento de espacios, pero no hay nada como buscar espacio de almacenamiento.)

Ahora uno puede observar que el montículo no es, de hecho, una cinta transportadora, pues si se trata como tal podría comenzar eventualmente una paginación excesiva de memoria (que constituye un factor de rendimiento importante), e incluso más tarde la memoria podría agotarse. El truco es que el recolector de basura va paso a paso, y mientras recolecta la basura, compacta todos los objetos de la pila de forma que el resultado es que se ha movido el "puntero del montículo" más cerca del principio de la cinta transportadora y más lejos de un fallo de página. El recolector de basura reorganiza los elementos y hace posible usar un modelo de montículo de alta velocidad y árbol infinito, durante la asignación de espacio de almacenamiento.

Para entender cómo funciona esto es necesario tener una idea un poco mejor de la manera en que funcionan los diferentes esquemas de recolección de basura (GC, *Garbage Collector*). Una técnica simple pero lenta de GC es contar referencias. Esto significa que cada ejemplo tiene un contador de referencias, y cada vez que se adjunte una referencia a un objeto se incrementa en uno el contador de referencias. Cada vez que una referencia cae fuera del ámbito o se pone **null** se decrementa el contador de referencias. Por consiguiente, la gestión de contadores de referencias supone una carga constante y pequeña que se va produciendo durante toda la vida del programa. El recolector de

basura va recorriendo toda la lista de objetos y al encontrar alguno con el contador de referencias a cero, libera el espacio de almacenamiento que tenía asignado. El inconveniente radica en que si los objetos tienen referencias circulares entre sí es posible no encontrar contadores de referencias a cero, que, sin embargo, pueden ser basura. La localización de estos grupos auto-referenciados requiere de una carga de trabajo significativa por parte del recolector de basura. La cuenta de referencias se usa frecuentemente para explicar un tipo de recolección de basura, pero parece no estar implementada en ninguna Máquina Virtual de Java.

En esquemas más rápidos, la recolección de basura no se basa en la cuenta de referencias. Se basa, en cambio, en la idea de que cualquier objeto no muerto podrá recorrerse, realizar una traza en última instancia, hasta una referencia que resida bien en la pila o bien en espacio de almacenamiento estático. La cadena podría atravesar varias capas de objetos. Por consiguiente, si se comienza en la pila y en el área de almacenamiento estático y se van recorriendo todas las referencias, será posible localizar todos los objetos vivos. Por cada referencia que se encuentre, es necesario hacer un recorrido traceo hasta localizar el objeto al que apunta y después seguir todas las referencias a *ese* objeto, recorriendo todos los objetos a los que apunta, etc., hasta haber recorrido toda la red que se originó con la referencia de la pila o del almacenamiento estático. Cada objeto que se recorra debe seguir necesariamente vivo. Fíjese que no hay ningún problema con los grupos auto-referenciados —simplemente no son localizados en el recorrido, trazado, por lo que se considerarán basura automáticamente.

En la aproximación descrita, la Máquina Virtual de Java usa un esquema de recolección de basura adaptativo, y lo que hace con los objetos vivos que encuentra depende de la variante que se haya implementado. Una de estas variaciones es la de *parar-y-copiar*. Esto significa que —por razones que pronto parecerán evidentes— el programa se detiene en primer lugar (este esquema no implica recolección en segundo plano). Posteriormente, cada objeto vivo que se encuentre se copia de un montículo a otro, dejando detrás toda la basura. Además, a medida que se copian los ejemplos al nuevo montículo, se empaquetan de extremo a extremo, compactando por consiguiente el nuevo montículo (y permitiendo recorrer rápida y simplemente el nuevo almacenamiento hasta el final, como se describió previamente).

Por supuesto, cuando se mueve un objeto de un lugar a otro, hay que cambiar todas las referencias que apuntan a ese objeto. Las referencias que vayan del montículo o del área de almacenamiento estática a un objeto pueden cambiarse directamente, pero puede haber otras referencias que apunten a este objeto y que se encuentren más tarde durante la "búsqueda". Éstas se van recomponiendo a medida que se encuentren (podría imaginarse una tabla que establezca una relación entre las direcciones viejas y las nuevas).

También hay dos aspectos que hacen ineficientes a estos denominados "recolectores de copias". El primero es la idea de que son necesarios dos montículos y se maneja por toda la memoria adelante y atrás entre estos dos montículos separados, manteniendo el doble de memoria de la que de hecho se necesita. Algunas Máquinas Virtuales de Java siguen este esquema asignando el montículo por bloques a medida que son necesarios y haciendo simplemente copias de bloques.

El segundo aspecto es la copia. Una vez que el programa se vuelve estable, debería generar poca o ninguna basura. Además de esto, un recolector de copias seguiría copiando toda la memoria de un

sitio a otro, lo que es una pérdida de tiempo y recursos. Para evitar esto, algunas Máquinas Virtuales de Java detectan que no se esté generando nueva basura y pasan a un esquema distinto (ésta es la parte "adaptativa"). Este otro esquema denominado *marcar y barrer*[4], es el que usaban las primeras versiones de la Máquina Virtual de Java de Sun. Para uso general, el esquema de marcar y barrer es bastante lento, pero si se genera poca o ninguna basura es rápido.

El marcar y barrer sigue la misma lógica de empezar rastreando a través de todas las referencias, a partir de la pila y el almacenamiento estático, para encontrar objetos vivos. Sin embargo, cada vez que encuentra un objeto vivo, lo marca poniendo a uno cierto indicador, en vez de recolectarlo. Sólo cuando acaba el proceso de marcado, se da el barrido. Durante el barrido, se liberan los objetos muertos. Sin embargo, no se da ninguna copia, de forma que si el recolector elige recolectar un montículo fragmentado, lo hace reordenando todos los objetos.

El "parar-y-copiar" se refiere a la idea de que este tipo de recolector de basura *no* se hace en segundo plano; sino que, por el contrario, se detiene el programa mientras se ejecuta el recolector de basura. En la documentación de Sun se encuentran muchas referencias a la recolección de basura como un proceso de segundo plano de baja prioridad, pero resulta que el recolector de basura no está implementado así, al menos en las primeras versiones de la Máquina Virtual de Java de Sun. En vez de esto, el recolector de basura de Sun se ejecutaba cuando quedaba poca memoria. Además el marcado y barrido requiere la detención del programa.

Como se mencionó previamente, en la Máquina Virtual de Java aquí descrita, la memoria se asigna por bloques grandes. Si se asigna un objeto grande, éste se hace con un bloque propio. El parar-y-copiar estricto exige copiar todos los objetos vivos del montículo fuente a un montículo nuevo antes de poder liberar el viejo, lo que se traduce en montones de memoria. Con los objetos, el recolector de basura puede en ocasiones usar los bloques muertos para copiar los objetos al ir recolectando. Cada bloque tiene un *contador de generación* para mantener información sobre si está o no vivo. En circunstancias normales, sólo se compactan los bloques creados desde la última recolección. Así se maneja la gran cantidad de objetos temporales de vida corta. Periódicamente, se hace un barrido completo —se siguen sin copiar los objetos grandes, y se copian y compactan todos los bloques que tienen objetos pequeños. La Máquina Virtual de Java monitoriza la eficiencia de la recolección de basura y si se convierte en una pérdida de tiempo porque todos los objetos tienen vida larga, pasa al esquema de marcar-y-barrer. De manera análoga, la Máquina Virtual de Java mantiene un registro del éxito del marcar-y-borrar, y si el montículo comienza a estar fragmentado vuelve de nuevo al parar-y-copiar. Éste es el momento en que interviene la parte "adaptativa", de forma que finalmente se tiene un nombre kilométrico: "Marcado-y-borrado con parada-y-copia adaptativo generacional".

Hay varias técnicas que permiten acelerar la velocidad de la Máquina Virtual de Java. Una especialmente importante se refiere a la operación del cargador y del compilador "justo-a-tiempo" (JIT). Cuando hay que cargar una clase (generalmente, la primera vez que se desea crear un objeto de esa clase), se localiza el fichero **.class** y se lleva a memoria el "codigo byte" de esa clase. En ese momento, un enfoque sería compilar JIT todo el código, pero esto tiene dos inconvenientes: lleva un poco más de tiempo, lo cual, extrapolado a toda la vida del programa puede ser significativo; y aumenta el tamaño del ejecutable (los "códigos byte" son bastante más compactos que el código JIT

[4] N. Del traductor: En inglés, *mark and sweep*.

expandido), lo que podría causar paginación, que definitivamente ralentizaría el programa. Un enfoque alternativo lo constituye la *evaluación perezosa*, que quiere decir que el código no se compila JIT hasta que es necesario. Por tanto, el código que no se ejecute nunca será compilado por JIT .

Inicialización de miembros

Java sigue este camino para garantizar que se inicialicen correctamente todas las variables antes de ser utilizadas. En el caso de variables definidas localmente en un método, esta garantía se presenta en forma de error de tiempo de compilación, de forma que si se dice:

```
void f() {
      int i;
      i++;
}
```

se obtendrá un mensaje de error que dice que **i** podría no haber sido inicializada. Por supuesto, el compilador podría haber asignado a **i** un valor por defecto, pero es más probable que se trate de un error del programador, que un valor por defecto habría camuflado. Al forzar al programador a dar un valor de inicialiación es más fácil detectar el fallo.

Sin embargo, las cosas son algo distintas en el caso de atributos de tipo primitivo de una clase. Dado que cualquier método puede inicializar o usar ese dato, podría no ser práctico obligar al usuario a inicializarlo a su valor apropiado antes de usar el dato. Sin embargo, es poco seguro dejarlo con un valor basura, por lo que se garantiza que tendrá un valor inicial. Estos valores pueden verse aquí:

```
//: c04:ValoresIniciales.java
// Muestra los valores iniciales por defecto.

class Medida {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void escribir() {
    System.out.println(
      "Tipo dato        Valor inicial\n" +
      "boolean          " + t + "\n" +
      "char             [" + c + "] "+ (int)c +"\n"+
      "byte             " + b + "\n" +
      "short            " + s + "\n" +
      "int              " + i + "\n" +
```

```
        "long              " + l + "\n" +
        "float             " + f + "\n" +
        "double            " + d);
  }
}

public class ValoresIniciales {
  public static void main(String[] args) {
    Medida d = new Medida();
    d.escribir();
    /* En este caso también podría decirse:
    new Medida().escribir();
    */
  }
} ///:~
```

La salida del programa será:

```
Tipo dato  Valor inicial
boolean    false
char       [ ] 0
byte       0
short      0
int        0
long       0
float      0.0
double     0.0
```

El valor **char** es un cero, que se imprime como un espacio.

Veremos más adelante que al definir una referencia a un objeto dentro de una clase sin inicializarla a un nuevo objeto, la referencia recibe el valor especial **null** (que es una palabra clave de Java).

Puede incluso verse que, aunque no se especifiquen los valores, se inicializan automáticamente. De esta forma, al menos, no hay amenaza de que se llegue a trabajar con valores sin inicializar.

Especificación de la inicialización

¿Qué ocurre si se quiere dar un valor inicial a una variable? Una manera directa de hacerlo consiste simplemente en asignar el valor al definir la variable en la clase. (Téngase en cuenta que esto no se puede hacer en C++, aunque los novatos en C++ siempre intentan hacerlo). Aquí se han cambiado las definiciones de la clase **Medida** para que proporcionen valores iniciales:

```
class Medida {
  boolean b = true;
  char c = 'x';
```

```
byte b = 47;
short s = 0xff;
int i = 999;
long l = 1;
float f = 3.14f;
double d = 3.14159;
// . . .
```

También se pueden inicializar de la misma manera objetos no primitivos. Si **Profundidad** es una clase, se puede insertar una variable e inicializarla así:

```
class Medida {
  Profundidad o = new Profundidad();
  boolean b = true;
  // . . .
```

Si no se ha dado a **o** un valor inicial e intenta usarlo de cualquier forma, se obtendrá un error de tiempo de ejecución denominado *excepción* (del que se hablará en el Capítulo 10).

Se puede incluso invocar a un método para proporcionar un valor de inicialización:

```
class CInit {
      int i = f();
      //...
}
```

El método puede, por supuesto, tener parámetros, pero éstos no pueden ser sino miembros de la clase que no han sido aún inicializados. Por consiguiente, se puede hacer esto:

```
class CInit {
      int i = f();
      int j = g(i);
      //...
}
```

Pero no se puede hacer esto:

```
class CInit {
      int j = g(i);
      int i = f();
      //...
}
```

Éste es un punto en el que el compilador se *queja*, con razón, del referenciado hacia delante, pues es un error relacionado con el orden de la inicialización y no con la manera de compilar el programa.

Este enfoque de inicialización es simple y directo. Tiene la limitación de que *todo objeto* de tipo **Medida** tendrá los mismos valores de inicialización. Algunas veces esto es justo lo que se necesita, pero otras veces se necesita mayor flexibilidad.

Inicialización de constructores

El constructor puede usarse para llevar a cabo la inicialización, lo que da una flexibilidad mayor en la programación, puesto que se puede invocar a métodos para llevar a cabo acciones en tiempo de ejecución que determinen los tiempos de ejecución. Sin embargo, hay que recordar siempre que no se está excluyendo la inicialización automática, que se da antes de entrar en el constructor. Así, por ejemplo, si se dice:

```
class Contador {
      int i;
      Contador() { i = 7; }
      // . . .
```

se inicializa primero la **i** a 0, y después a 7. Esto es cierto con todos los tipos primitivos y con las referencias a objetos, incluyendo aquéllos a los que se da inicialización explícita en el momento de su definición. Por esta razón, el compilador no intenta forzar la inicialización de elementos del constructor en ningún lugar en concreto, o antes de que se usen —la inicialización ya está garantizada[4].

Orden de inicialización

Dentro de una clase, el orden de inicialización lo determina el orden en que se definen las variables dentro de la clase. Las definiciones de variables pueden estar dispersas a través y dentro de las definiciones de métodos, pero las variables se inicializan antes de invocar a ningún método —incluido el constructor. Por ejemplo:

```
//: c04:OrdenDeInicializacion.java
// Demuestra el orden de inicialización.

// Cuando se invoque al constructor para crear un
// objeto Etiqueta, se verá un mensaje:
class Etiqueta {
  Etiqueta(int marcador) {
    System.out.println("Etiqueta(" + marcador + ")");
  }
}

class Tarjeta {
  Etiqueta t1 = new Etiqueta(1); // Antes del constructor
```

[4] En contraste, C++ tiene la *lista de inicializadores del constructor* que hace que se dé la inicialización antes de entrar en el cuerpo del constructor, y se fuerza para los objetos. Ver *Thinking in C++*, 2.ª edición (disponible en el CD ROM de este libro, y en *http://www.BruceEckel.com*).

```
  Tarjeta() {
    // Indicar que estamos en el constructor:
    System.out.println("Tarjeta()");
    t3 = new Etiqueta(33); // Reiniciar t3
  }
  Etiqueta t2 = new Etiqueta(2); // Después del constructor
  void f() {
    System.out.println("f()");
  }
  Etiqueta t3 = new Etiqueta(3); // Al final
}

public class OrdenDeInicializacion {
  public static void main(String[] args) {
    Tarjeta t = new Tarjeta();
    t.f(); // Muestra que se ha acabado la construcción
  }
} ///:~
```

En **Tarjeta**, la definición de los objetos **Etiqueta** se han dispersado intencionadamente para probar que todos se inicializarán antes de que se llegue a entrar al constructor u ocurra cualquier otra cosa. Además, **t3** se reinicia dentro del constructor. La salida es:

```
Etiqueta(1)
Etiqueta(2)
Etiqueta(3)
Tarjeta()
Etiqueta(33)
f()
```

Por consiguiente, la referencia **t3** se inicializa dos veces, una antes y otra durante la llamada al constructor. (El primer objeto se desecha, de forma que posteriormente podrá ser eliminado por el recolector de basura.) Esto podría parecer ineficiente a primera vista, pero garantiza una inicialización correcta —¿Qué ocurriría si se definiera un constructor sobrecargado que *no* inicializara **t3** y no hubiera una inicialización "por defecto" para **t3** en su definición?

Inicialización de datos estáticos

Cuando los datos son **estáticos** ocurre lo mismo; si se trata de un dato primitivo y no se inicializa, toma los valores iniciales estándares de los tipos primitivos. Si se trata de una referencia a un objeto, es **null,** a menos que se cree un objeto nuevo al que se asocie la referencia.

Si se desea realizar una inicialización en el momento de la definición, ocurre lo mismo que con los no **estáticos**. Sólo hay un espacio de almacenamiento para un dato **estático** independientemente de cuántos objetos se creen. Pero las dudas surgen cuando se inicializa el espacio de almacenamiento de un dato **estático**. Un ejemplo puede aclarar esta cuestión:

```
//: c04:InicializacionStatic.java
// Especificando los valores iniciales en una
// definición de clase.

class Bolo {
  Bolo(int marcador) {
    System.out.println("Bolo(" + marcador + ")");
  }
  void f(int marcador) {
    System.out.println("f(" + marcador + ")");
  }
}

class Mesa {
  static Bolo b1 = new Bolo(1);
  Mesa() {
    System.out.println("Mesa()");
    b2.f(1);
  }
  void f2(int marcador) {
    System.out.println("f2(" + marcador + ")");
  }
  static Bolo b2 = new Bolo(2);
}

class Armario {
  Bolo b3 = new Bolo(3);
  static Bolo b4 = new Bolo(4);
  Armario() {
    System.out.println("Armario()");
    b4.f(2);
  }
  void f3(int marcador) {
    System.out.println("f3(" + marcador + ")");
  }
  static Bolo b5 = new Bolo(5);
}

public class InicializacionStatic {
  public static void main(String[] args) {
    System.out.println(
      "Creando nuevo Armario() en el método main");
    new Armario();
    System.out.println(
      "Creando nuevo Armario() en el método main");
```

```
    new Armario();
    t2.f2(1);
    t3.f3(1);
  }
  static Mesa t2 = new Mesa();
  static Armario t3 = new Armario();
} ///:~
```

Bolo permite ver la creación de una clase, y **Mesa** y **Armario** crean miembros **estáticos** de **Bolo** dispersos por sus definiciones de clases. Fíjese que **Armario** crea un **Bolo** no **estático** antes de las definiciones **estáticas**. La salida muestra lo que ocurre:

```
Bolo(1)
Bolo(2)
Mesa()
f(1)
Bolo(4)
Bolo(5)
Bolo(3)
Armario()
f(2)
Creando nuevo Armario() el método main
Bolo(3)
Armario()
f(2)
Creando nuevo Armario() el método main
Bolo(3)
Armario()
f(2)
f2(1)
f3(1)
```

La inicialización **estática** sólo se da si es necesaria. Si no se crea un objeto **Mesa** y nunca se hace referencia a **Mesa.b1** o **Mesa.b2**, los objetos estáticos de tipo **Bolo b1** y **b2** no se crearán nunca. Sin embargo, se inicializan sólo cuando se cree el *primer* objeto **Mesa** (o se dé el primer acceso **estático**). Después de eso, los objetos **estáticos** no se reinician.

Se inicializan primero los objetos **estáticos**, si todavía no han sido inicializados durante la creación anterior de un objeto, y posteriormente los objetos no estáticos. Se puede ver la prueba de esto en la salida del programa anterior.

Es útil resumir el proceso de creación de un objeto. Considérese una clase llamada **Perro**:

1. La primera vez que se cree un objeto de tipo **Perro**, *o* la primera vez que se acceda a un método **estático** o un campo **estático** de la clase **Perro**, el intérprete de Java debe localizar **Perro.class**, que lo hace buscando a través de las trayectorias de clases.

2. Al cargar **Perro.class** (creando un objeto **Class**, del que se hablará más adelante), se ejecutan todos sus inicializadores **estáticos**. Por consiguiente, la inicialización sólo tiene lugar una vez, al cargar el objeto **Class** la primera vez.
3. Cuando se crea un **new Perro( )**, el proceso de construcción de un objeto **Perro** asigna, en primer lugar, el espacio de almacenamiento suficiente para un objeto **Perro** del montículo.
4. Este espacio de almacenamiento se pone a cero, poniendo automáticamente todos los datos primitivos del objeto **Perro** con sus valores por defecto (cero para los números y su equivalente para los **boolean** o **char**) y las referencias a **null**.
5. Se ejecuta cualquier inicialización que se dé en el momento de la definición de campos.
6. Se ejecutan los constructores. Como se verá en el Capítulo 6, esto podría implicar de hecho una cantidad de actividad considerable, especialmente cuando esté involucrada la herencia .

Inicialización estática explícita

Java permite agrupar todas las inicializaciones **estáticas** dentro de una "cláusula de construcción **estática**" (llamada a veces *bloque estático*) dentro de una clase. Tiene la siguiente apariencia:

```
class Cuchara {
      static int i;
      static {
            i = 47;
      }
      // . . .
```

Parece un método, pero es simplemente la palabra clave **static** seguida de un cuerpo de método. Este código, como otras inicializaciones **estáticas**, se ejecuta sólo una vez, la primera vez que se cree un objeto de esa clase *o* la primera vez que se acceda a un miembro **estático** de esa clase (incluso si nunca se llega a hacer un objeto de esa clase). Por ejemplo:

```
//: c04:StaticExplicito.java
// Inicialización explícita estática
// con la cláusula "static".

class Taza {
   Taza(int marcador) {
      System.out.println("Taza(" + marcador + ")");
   }
   void f(int marcador) {
      System.out.println("f(" + marcador + ")");
   }
}

class Tazas {
```

```
  static Taza c1;
  static Taza c2;
  static {
    c1 = new Taza(1);
    c2 = new Taza(2);
  }
  Tazas() {
    System.out.println("Tazas()");
  }
}

public class StaticExplicito {
  public static void main(String[] args) {
    System.out.println("Dentro de main()");
    Tazas.c1.f(99);   // (1)
  }
  // static Tazas x = new Tazas();   // (2)
  // static Tazas y = new Tazas();   // (2)
} ///:~
```

Los inicializadores **estáticos** de **Tazas** se ejecutan cuando se da el acceso al objeto **estático c1** en la línea marcada (1), si la línea (1) se marca como un comentario, y se quita el signo de comentario de las líneas marcadas como (2). Si tanto (1) como (2) se consideran comentarios, la inicialización **estática** de **Tazas** no se realizará nunca. Además, no importa si una o las dos líneas marcadas (2) dejan de ser comentarios; la inicialización sólo ocurre una vez.

Inicialización de instancias no estáticas

Java proporciona una sintaxis similar para la inicialización de variables no **estáticas** de cada objeto. He aquí un ejemplo:

```
//: c04:Jarras.java
// Java "Inicialización de Instancias."

class Jarra {
  Jarra(int marcador) {
    System.out.println("Jarra(" + marcador + ")");
  }
  void f(int marcador) {
    System.out.println("f(" + marcador + ")");
  }
}

public class Jarras {
  Jarra c1;
  Jarra c2;
```

```
  {
    c1 = new Jarra(1);
    c2 = new Jarra(2);
    System.out.println("c1 y c2 inicializadas");
  }
  Jarras() {
    System.out.println("Jarras()");
  }
  public static void main(String[] args) {
    System.out.println("Dentro de main()");
    Jarras x = new Jarras();
  }
} ///:~
```

Se puede ver que la cláusula de inicialización de instancias:

```
{
    c1 = new Jarra(1);
    c2 = new Jarra(2);
    System.out.println("c1 y c2 inicializadas");
}
```

tiene exactamente la misma apariencia que la cláusula de inicialización estática excepto porque no está la palabra clave **static**. Esta sintaxis es necesaria para dar soporte a la inicialización de clases *internas anónimas* (ver Capítulo 8).

Inicialización de arrays

La inicialización de arrays en C suele ser fuente de errores y tediosa. C++ usa la *inicialización agregada* para hacerla más segura[5]. Java no tiene "agregados" como C++, puesto que en Java todo es un objeto. Tiene arrays, y éstos se soportan con la inicialización de arrays.

Un array es simplemente una secuencia, bien de objetos o bien de datos primitivos, todos del mismo tipo, empaquetados juntos bajo un único identificador. Los arrays se definen y utilizan con el *operador de indexación* entre corchetes **[ ]**. Para definir un array simplemente hay que colocar corchetes vacíos seguidos del nombre del tipo de datos:

```
int [] a1;
```

También se puede poner los corchetes tras el identificador para lograr exactamente el mismo significado:

```
int a1[];
```

[5] *Thinking in C++*, 2.ª edición, para obtener una descripción completa de la inicialización agregada.

Esto satisface las expectativas de los programadores de C y C++. El estilo anterior, sin embargo, es probablemente una sintaxis más sensata, puesto que dice que el tipo es "un array de **int**". Éste será el estilo que se use en este libro.

El compilador no permite especificar el tamaño del array. Esto nos devuelve al aspecto de las "referencias". Todo lo que se tiene en este momento es una referencia a un array, para el que no se ha asignado espacio de almacenamiento. Para crear espacio de almacenamiento para el array es necesario escribir una expresión de inicialización. En el caso de los arrays, la inicialización puede aparecer en cualquier lugar del código, pero puede usarse un tipo especial de expresión de inicialización que debe situarse en el mismo lugar en que el que se crea el array. Esta inicialización especial es un conjunto de valores encerrados entre llaves. Es el compilador el que se encarga de la asignación de espacio (el equivalente a usar **new**). Por ejemplo:

```
int [ ] a1 = { 1, 2, 3, 4, 5 };
```

Por tanto ¿por qué puede definirse una referencia a un array sin un array?

```
int [ ] a2;
```

Bien, es posible asignar un array a otro en Java, por lo que puede decirse:

```
a2 = a1;
```

Lo que se está haciendo realmente es copiar una referencia, como se demuestra a continuación:

```
//: c04:Arrays.java
// Arrays de datos primitivos.

public class Arrays {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

Puede verse que se da a **a1** un valor de inicialización mientras que a **a2**, no; **a2** se asigna más tarde —en este caso, a otro array.

Aquí hay algo nuevo: todos los arrays tienen un miembro intrínseco (bien sean arrays de objetos o arrays de tipos primitivos) por el que se puede preguntar —pero no modificar— para saber cuántos elementos hay en el array. Este miembro es **length**. Dado que los arrays en Java, como en C y C++, empiezan a contar desde elemento 0, el elemento más lejano que se puede indexar es **length - 1**.

Si se sale de rango, no produce error, siendo esto la fuente de muchos errores graves. Sin embargo, Java le protege de esos problemas originando un error en tiempo de ejecución (una *excepción*, el tema del Capítulo 10) al intentar acceder más allá de los límites. Por supuesto, la comprobación de todos los accesos a arrays supone tiempo y código, y no hay manera de desactivarse, lo que significa que los accesos a arrays podrían ser una fuente de ineficiencia en un programa si se dan en una situación crítica. Los diseñadores de Java pensaron que este sacrificio merecía la pena en aras de la seguridad de Internet y la productividad del programador.

¿Qué ocurre si al escribir el programa se desconocen cuántos elementos son necesarios que tenga el array? Simplemente se utiliza **new** para crear elementos del array. Aquí, **new** funciona incluso aunque se esté creando un array de datos primitivos (**new** no creará datos primitivos que no sean elementos de un array):

```
//: c04:NuevoArray.java
// Creando arrays con new.
import java.util.*;

public class NuevoArray {
  static Random aleatorio = new Random();
  static int pAleatorio(int modulo) {
    return Math.abs(aleatorio.nextInt()) % modulo + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pAleatorio(20)];
    System.out.println(
      "longitud de = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

Dado que el tamaño del array se elige al azar (utilizando el método **pAleatorio( )**) está claro que la creación del array se está dando en tiempo de ejecución. Además, se verá en la salida de este programa que los elementos del array de tipos primitivos se inicializan automáticamente a valores "vacíos". (En el caso de valores numéricos y **carácter**, este valor es cero, y en el caso de los **boolean**, es **false**.)

Por supuesto, el array también podría haberse definido e inicializado en la misma sentencia:

```
int [] a = new int[pAleatorio(20)];
```

Si se está tratando con un array de objetos no primitivos, siempre es necesario usar **new**. Aquí, vuelve a surgir el tema de las referencias porque lo que se crea es un array de referencias. Considérese el tipo **Integer**, que es una clase y no un tipo primitivo:

```
//: c04:ObjetoClaseArray.java
// Creando un array de objetos no primitivos.
```

```
import java.util.*;

public class ObjetoClaseArray {
  static Random aleatorio = new Random();
  static int pAleatorio(int modulo) {
    return Math.abs(aleatorio.nextInt()) % modulo + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pAleatorio(20)];
    System.out.println(
      "longitud de a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pAleatorio(500));
      System.out.println(
        "a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

Aquí, incluso tras llamar a **new** para crear el array:

```
Integer[] a = new Integer[pAleatorio(20)];
```

se trata sólo de un array de referencias, y no se completa la inicialización hasta que se inicializa la propia referencia creando un nuevo objeto **Integer**:

```
a[i] = new Integer(pAleatorio(500));
```

Si se olvida crear el objeto, sin embargo, se obtiene una excepción en tiempo de ejecución al intentar leer la localización vacía del array.

Eche un vistazo a la creación del objeto **String** dentro de las sentencias de impresión. Puede observarse que la referencia al objeto **Integer** se convierte automáticamente para producir un **String** que representa el valor dentro del objeto.

También es posible inicializar el array de objetos utilizando la lista encerrada entre llaves. Hay dos formas:

```
//: c04:InicializacionArray.java
// Inicialización de arrays.

public class InicializacionArray {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
```

```
    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

Esto es útil en ocasiones, pero es más limitado, pues se determina el tamaño del array en tiempo de compilación. La coma final de la lista de inicializadores es opcional. (Esta característica permite un mantenimiento más sencillo de listas largas.)

La segunda forma de inicializar arrays proporciona una sintaxis adecuada para crear y llamar a métodos que pueden producir el mismo efecto que las *listas de parámetros variables* de C (conocidas en este lenguaje como "parametros-variables"). Éstas pueden incluir una cantidad de parámetros desconocida además de tipos desconocidos. Dado que todas las clases se heredan en última instancia de la clase raíz común **Object** (un tema del que se aprenderá más a medida que progrese el libro), se puede crear un método que tome un array de **Object** e invocarlo así:

```
//: c04:ParametrosVariables.java
// Utilizando la sintaxis de arrays para crear
// listas de parámetros variables.

class A { int i; }

public class ParametrosVariables {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new ParametrosVariables(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"un", "dos", "tres" });
    f(new Object[] {new A(), new A(), new A()});
  }
} ///:~
```

En este punto, no hay mucho que pueda hacerse con estos objetos desconocidos, y el programa usa la conversión automática **String** para hacer algo útil con cada **Object**. En el Capítulo 12, que cubre la *identificación de tipos en tiempo de ejecución* (*Run-time type identification*, RTTI), se aprenderá a descubrir el tipo exacto de objetos así, de forma que se pueda hacer algo más interesante con ellos.

Arrays multidimensionales

Java permite crear fácilmente arrays multidimensionales:

```
//: c04:ArrayMultidimensional.java
// Creando arrays multidimensionales.
import java.util.*;

public class ArrayMultidimensional {
  static Random aleatorio = new Random();
  static int pAleatorio(int modulo) {
    return Math.abs(aleatorio.nextInt()) % modulo + 1;
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
        visualizar("a1[" + i + "][" + j +
            "] = " + a1[i][j]);
    // array 3-D de longitud fija:
    int[][][] a2 = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length;
            k++)
          visualizar("a2[" + i + "][" +
              j + "][" + k +
              "] = " + a2[i][j][k]);
    // array 3-D con vectores de longitud variable:
    int[][][] a3 = new int[pAleatorio(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pAleatorio(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pAleatorio(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
```

```
            visualizar("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
      // Array de objetos no primitivos:
      Integer[][] a4 = {
        { new Integer(1), new Integer(2)},
        { new Integer(3), new Integer(4)},
        { new Integer(5), new Integer(6)},
      };
      for(int i = 0; i < a4.length; i++)
        for(int j = 0; j < a4[i].length; j++)
          visualizar("a4[" + i + "][" + j +
              "] = " + a4[i][j]);
      Integer[][] a5;
      a5 = new Integer[3][];
      for(int i = 0; i < a5.length; i++) {
        a5[i] = new Integer[3];
        for(int j = 0; j < a5[i].length; j++)
          a5[i][j] = new Integer(i*j);
      }
      for(int i = 0; i < a5.length; i++)
        for(int j = 0; j < a5[i].length; j++)
          visualizar("a5[" + i + "][" + j +
              "] = " + a5[i][j]);
    }
} ///:~
```

El código utilizado para imprimir utiliza el método **length**, de forma que no depende de tamaños fijos de array.

El primer ejemplo muestra un array multidimensional de tipos primitivos. Se puede delimitar cada vector del array por llaves:

```
int[][] a1 = {
        { 1, 2, 3, },
        { 4, 5, 6, },
      };
```

Cada conjunto de corchetes nos introduce en el siguiente nivel del array.

El segundo ejemplo muestra un array de tres dimensiones asignado con **new**. Aquí, se asigna de una sola vez todo el array:

```
int[][][] a2 = new int[2][2][4];
```

Pero el tercer ejemplo muestra que cada vector en los arrays que conforman la matriz pueden ser de cualquier longitud:

```
int[][][] a3 = new int[pAleatorio(7)][][];
     for(int i = 0; i < a3.length; i++) {
       a3[i] = new int[pAleatorio(5)][];
       for(int j = 0; j < a3[i].length; j++)
         a3[i][j] = new int[pAleatorio(5)];
     }
```

El primer **new** crea un array con un primer elemento de longitud aleatoria, y el resto, indeterminados. El segundo **new** de dentro del bucle **for** rellena los elementos pero deja el tercer índice indeterminado hasta que se acometa el tercer **new**.

Se verá en la salida que los valores del array que se inicializan automáticamente a cero si no se les da un valor de inicialización explícito.

Se puede tratar con arrays de objetos no primitivos de forma similar, lo que se muestra en el cuarto ejemplo, que demuestra la habilidad de englobar muchas expresiones **new** entre llaves:

```
Integer[][] a4 = {
       { new Integer(1), new Integer(2)},
       { new Integer(3), new Integer(4)},
       { new Integer(5), new Integer(6)},
     };
```

El quinto ejemplo muestra cómo se puede construir pieza a pieza un array de objetos no primitivos:

```
Integer[][] a5;
     a5 = new Integer[3][];
     for(int i = 0; i < a5.length; i++) {
       a5[i] = new Integer[3];
       for(int j = 0; j < a5[i].length; j++)
         a5[i][j] = new Integer(i*j);
     }
```

El **i*j** es simplemente para poner algún valor interesante en el **Integer**.

Resumen

El constructor, mecanismo aparentemente elaborado de inicialización, proporciona un importante mecanismo para realizar la inicialización. Cuando Stroustrup estaba diseñando C++, una de las primeras observaciones que hizo sobre la productividad de C era relativa a la inicialización de las variables erróneas que causan un porcentaje significativo de los problemas de programación. Estos tipos de fallos son difíciles de encontrar, y hay aspectos similares que pueden aplicarse a la limpieza errónea. Dado que los constructores permiten *garantizar* la inicialización correcta y la limpieza (el compilador no permitirá que un objeto se cree sin los constructores pertinentes), se logra un control y seguridad completos.

En C++, la destrucción es bastante importante porque los objetos creados con **new** deben ser destruidos explícitamente. En Java, el recolector de basura libera automáticamente la memoria de todos

los objetos, por lo que el método de limpieza equivalente es innecesario en Java en la mayoría de ocasiones. En los casos en los que no es necesario un comportamiento al estilo de un destructor, el recolector de basura de Java simplifica enormemente la programación, y añade un elevado y necesario nivel de seguridad a la gestión de memoria. Algunos recolectores de basura pueden incluso limpiar otros recursos como los gráficos y los manejadores de ficheros. Sin embargo, el recolector de basura añade un coste en tiempo de ejecución, cuyo gasto es difícil de juzgar, debido a la lentitud de los intérpretes de Java existentes en el momento de escribir el presente libro. Cuando cambie esto, se podrá descubrir si la sobrecarga del recolector de basura excluirá el uso de Java para determinados tipos de programas. (Uno de los aspectos es la falta de predicción del recolector de basura.)

Dado que se garantiza la construcción de todos los objetos, de hecho, hay más aspectos que los aquí descritos. En particular, al crear nuevas clases usando la *agregación* o la *herencia* también se mantiene la garantía de construcción, aunque es necesaria cierta sintaxis para dar soporte a esto. Se aprenderá todo lo relativo a la agregación, la herencia y cómo afectan éstas operaciones a los constructores en los capítulos siguientes.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Crear una clase con el constructor por defecto (el que no tiene parámetros) que imprima un mensaje. Crear un objeto de esta clase.
2. Añadir un constructor sobrecargado al Ejercicio 1, que tome un **String** como parámetro y lo imprima junto con el mensaje.
3. Crear un array de referencias a objetos de la clase creada en el Ejercicio 2, pero no crear los objetos a asignar al array. Al ejecutar el programa, tomar nota de si se imprimen los mensajes de inicialización del constructor.
4. Completar el Ejercicio 3 creando los objetos a asociar al array de referencias.
5. Crear un array de objetos **String** y asignar una cadena de caracteres a cada elemento. Imprimir el array utilizando un bucle **for**.
6. Crear una clase **Perro** con un método **ladrar( )** sobrecargado. Este método debería sobrecargarse en base a varios tipos de datos primitivos, e imprimir distintos tipos de ladridos, aullidos, etc. dependiendo de la versión sobrecargada que se invoque. Escribir un método **main( )** que llame a todas las distintas versiones.
7. Modificar el Ejercicio 6 de forma que dos de los métodos sobrecargados tengan dos argumentos (de dos tipos distintos), pero en orden inverso entre sí. Verificar que funciona.
8. Crear una clase sin constructor, y crear un objeto de esa clase en **main( )** para verificar que el constructor por defecto se invoca automáticamente.
9. Crear una clase con dos métodos. Dentro del primer método, invocar al segundo dos veces: la primera vez sin utilizar **this**, y la segunda, usando **this**.

10. Crear una clase con dos constructores (sobrecargados). Utilizando **this**, invocar al segundo constructor dentro del primero.

11. Crear una clase con un método **finalize( )** que imprima un mensaje. En **main( )**, crear un objeto de esa clase. Explicar el funcionamiento del programa.

12. Modificar el Ejercicio 11 de forma que siempre se llame a **finalize( )**.

13. Crear una clase llamada **Tanque** que pueda rellenarse y vaciarse, y que tenga una *condición de muerte* que tenga que estar vacía al eliminar el objeto. Escribir un **finalize( )** que verifique esta condición de muerte. En el método **main( )**, probar los escenarios posibles que puedan ocurrir al usar **Tanque**.

14. Crear una clase que contenga un **int** y un **char** no inicializados, e imprimir sus valores para verificar que Java realiza la inicialización por defecto.

15. Crear una clase que contenga una referencia **String** sin inicializar. Demostrar que Java inicializa esta referencia a **null**.

16. Crear una clase con un campo **String** que se inicialice en el momento de la definición y otra que inicialice el constructor. ¿Cuál es la diferencia entre los dos enfoques?

17. Crear una clase con un campo **estático String** que se inicialice en el momento de la definición, y otra que sea inicializada por un bloque **estático**. Añadir un método **estático** que imprima ambos campos y demuestre que ambos se inicializan antes de usarse.

18. Crear una clase con un **String** que se inicialice usando "inicialización de instancias". Describir un uso de esa característica (una descripción distinta de la que se especifica en este libro).

19. Escribir un método que cree e inicialice un array bidimensional de datos de tipo **double**. El tamaño del array vendrá determinado por los parámetros del método, y los valores de inicialización vendrán determinados por un rango delimitado por sus valores superior e inferior, parámetros ambos también del método. Crear un segundo método que imprima el array generado por el primer método. En el método **main( )** probar los métodos creando e imprimiendo varios arrays de distintos tamaños.

20. Repetir el Ejercicio 19 para un array tridimensional.

21. Comentar la línea marcada (1) en **StaticExplicito.java** y verificar que la cláusula de inicialización estática no es invocada. Ahora, quitar la marca de comentario de alguna de las líneas marcadas (2) y verificar que se invoca a la cláusula de inicialización estática. Ahora quitar la marca de comentario de la otra línea marcada (2) y verificar que la inicialización estática sólo se da una vez.

22. Experimentar con **Basura.java** ejecutando el programa utilizando los argumentos "rec", "finalizar" o "todo". Repetir el proceso y ver si detecta patrones en la salida. Cambiar el código de forma que se llame a **System.runFinalization( )** *antes* que a **System.gc( )** y observar los resultados.