4: Inicialización y Limpieza

Con el avance de la revolución informática, la "inestabilidad" de la programación se ha convertido en uno de los factores principales en el incremento del coste del software.

Dos de estos elementos de estabilidad son la *inicialización* y el *purgado* , o liberación de recursos del sistema cuando estos dejan de ser necesarios. Muchos de los errores en C se producen cuando el programador olvida inicializar una variable.Esto es cierto sobre todo cuando se trabaja con librerías ya que los usuarios a menudo desconocen como inicializar sus componentes o incluso si deben ser inicializados o no. El purgado presenta un problema especial ya que es fácil olvidarse de un elemento cuando ha dejado de usarse y ha dejado de ser interesante. Es así como los recursos usados por ese elemento no son liberados con lo que se puede acabar por agotar los recursos del sistema (normalmente la memoria).

C++ introdujo el concepto de *constructor* , un método especial que es invocado automáticamente cuando se crea un objeto. Java ha adaptado el constructor y además tiene un recogedor de basura que automáticamente libera las zonas de memoria que ya no están en uso. Este capítulo examina los conceptos de inicialización y purgado y su implementación en Java.

El constructor garantiza la inicialización

Se puede imaginar que hubiese que crear un método llamado **inicializar()** cada una de las clases que se escribiese. El nombre del es una indicación de que el método debebería ser invocado antes de usar el objeto. Pero esto significaría que el usuario debe acordarse siempre de llamar a este método. En Java el diseñador de una clase puede garantizar que todos los objetos son inicializados definiendo un método especial llamado *constructor* . Si una clase tiene un constructor Java invocará automáticamente ese constructor cuando el objeto es creado, antes de que el usuario pueda ponerle las manos encima.

El siguiente problema a resolver es que nombre debería darse a este método. Hay que considerar dos cuestiones. La primera es que, cualquier nombre que se elija podría coincidir con el que se desease usar un miembro de la clase. La segunda cuestión es que el compilador debe generar la invocación al constructor por lo que deberá saber en todo momento que método llamar. La solución que da C++ parece la más sencilla y la más lógica por lo que ha sido adoptada por Java: el nombre del constructor es el mismo que el de la clase. Es lógico que así se

invocado durante la inicialización.

Veamos el constructor de una clase sencilla:

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

class Roca {
   Roca () { // Este 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();
   }

} ///:-
```

Entonces, cuando se crea un objeto:

```
new Roca();
```

se reserva un área de memoria y se invoca el constructor. De este modo queda garantizado que el objeto es inicializado adecuadamente antes de que nadie pueda tocarlo

Se observa que, la regla habitual de poner en minúscula la primera letra del nombre de los métodos no se aplica a los constructores, ya que estos deben tener *exactamente* con el nombre de su clase.

Como cualquier otro método los constructores pueden tener argumentos que permiten especificar como se ha de crear el objeto.El ejemplo anterior puede ser fácilmente modificado que el constructor tenga un argumento.

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

class Roca2 {
   Roca2(int i) {
      System.out.println(
```

```
         "Creando Roca numero " + i);
   }
}

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

} ///:-
```

Los argumentos del constructor proporcionan al programador un medio parametrizar la creación de objetos. Por ejemplo, si la clase **Arbol** tiene un constructor con un único argumento entero representando la altura, se puede crear un objeto **Arbol** del siguiente modo:

```
Arbol a = new Arbol (12);  // árbol de 12 pies.
```

Si **Arbol(int)** es el único constructor, el compilador no permitirá que se creen objetos de ninguna otra manera.

Los constructores eliminan una amplia categoría de errores y hacen que el código sea más fácil de leer. Por ejemplo, en el fragmento de código precedente no aparece ninguna llamada explícita a ningún método **inicializar()** separado conceptualmente de la definición. En Java la definición y la inicialización son dos conceptos que van de la mano: no se puede tener uno sin el otro.

El constructor es un método atípico ya que no devuelve ningún valor. Esto es completamente diferente de un valor **void** en que un método no devuelve nada pero en un método ordinario siempre se tiene la posibilidad de elegir entre devolver un valor o no. Los constructores no devuelven nada y, además, no es posible modificar este comportamiento. Si se pudiese devolver un valor y se pudiese elegir cual, el compilador debería saber que hacer con ese valor.

Sobrecarga de métodos

Un aspecto importante en cualquier lenguaje de aplicación es el uso de los nombres. Cuando se crea un objeto se le asigna un nombre a una zona de almacenamiento. El uso de nombres describir un sistema permite crear programas que son más fáciles de entender y modificar. Es parecido a escribir en prosa, el objetivo es comunicarse con los lectores.

Se usan nombres referirse a los objetos y a los métodos. La elección de nombres

adecuados facilita la comprensión del código uno mismo y los demás.

Cuando se intenta trasladar el concepto de matiz en el lenguaje humano a un lenguaje de programación surgen problemas. A menudo una simple palabra tiene más de un posible significado, es decir, esta *sobrecargada* . Esto resulta especialmente útil cuando las diferencias entre los posibles significados son solo de detalle. Decimos "lavar la camisa", "lavar el coche" o "lavar al perro". Sería absurdo tener que decir "camisa-lavar la camisa", "coche-lavar el coche" o "perro-lavar al perro" porque el que escucha no necesita que se haga ninguna distinción entre las diferentes acciones. La mayoría de los lenguajes humanos son redundantes por lo que es posible reconstruir el significado de una frase incluso cuando se han perdido unas pocas palabras; podemos deducir el significado por el contexto.

La mayoría de los lenguajes de programación (C en particular) obligan a que cada función tenga un nombre único. Por lo tanto no se puede tener una función **print ()** imprimir enteros y otra función **print()** números en coma flotante. Cada función debe tener un nombre único.

En Java (y en C++) hay otra razón que obliga a la sobrecarga en los nombres de los métodos. Como el nombre del constructor debe se el mismo que el de la clase, solo existe un nombre posible él. Pero qué sucede si se desea poder construir un objeto en más de un modo diferente? Por ejemplo, si se diseña una clase que pueda inicializarse de un modo estándar o leyendo la información necesaria de un fichero, se necesitará dos constructores: uno sin argumentos (el constructor *predeterminado* , también llamado constructor *no-args*) y otro que tenga un argumento de tipo **String** que contenga el nombre del fichero. Ambos son constructores por lo que deben tener el mismo nombre: el nombre de la clase. Por lo tanto la *sobrecarga de métodos* es esencial que mismo nombre pueda se usado por dos procedimientos con argumentos diferentes. La sobrecarga de métodos es necesaria los constructores y en general es un mecanismo conveniente que puede usarse con cualquier método.

En el ejemplo siguiente muestra tanto constructores como métodos ordinarios sobrecargados:

```
//: c04:Sobrecargando.java
// Ejemplo de sobrecarga en constructores
// y metodos ordinarios.
import java.util.*;

class Arbol {

   int altura;
   Arbol() {
      prt("Plantando una semilla");
      altura=0;
```

```
   }
   Arbol(int i) {
      prt("Creando un arbol que mide "
          + i + " pies de alto");
      altura = i;
   }
   void info() {
      prt("El arbol mide " + altura +
          " pies de alto");
   }
   void info() {
      prt(s + ": El arbol mide "
          + height + " pies de alto");
   }
   static void prt(String s) {
      system.out.println(s);
   }
}

public class Sobrecargando {
   public static void main (String[] args) {
      for (int i = 0; i < 5; i++) {
         Arbol a = new Arbol(i);
         a.info();
         a.info("metodo sobrecargado");
      }
      // Constructor sobrecargado:
      new Arbol();
   }
} ///:~
```

Se puede crear un **Arbol** a partir de una semilla o a partir de un árbol joven criado en un invernadero. que ésto sea posible hay dos constructores; uno sin argumentos (a los constructores sin argumentos se les denomina *constructores predeterminados*) y otro que tiene por argumento la altura inicial.

También se puede querer llamar al método **info()** de más de una manera.Por ejemplo con un argumento **String** , si de desea presentar un mensaje extra, o sin argumentos ,si no se tiene nada que añadir. Resultaría extraño tener que usar nombres diferentes dos cosas que representan el mismo concepto. La sobrecarga de métodos nos permite usar el mismo nombre ambos.

Decidiendo entre métodos sobrecargados

Si es posible tener varios métodos con el mismo nombre Ücómo puede Java determinar que método ha de ser invocado? La regla es sencilla: cada método debe tener una lista de argumentos diferente.

Si se reflexiona sobre ello, tiene sentido: de qué otra manera podría el programador indicar la diferencia entre los dos métodos si no es mediante la lista de argumentos?

Se puede distinguir entre dos métodos incluso por el orden de los argumentos (Sin embargo, no es conveniente usar esta opción ya que conduce a un código difícil de mantener).

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

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

Los dos métodos **print()** tienen los mismos argumentos, pero en diferente orden lo que les hace diferentes.

Sobrecarga con primitivas

Una primitiva puede ser elevada de modo automático a un tipo más general lo que puede dar lugar a una cierta confusión cuando se combina con la sobrecarga de métodos. El ejemplo siguiente muestra que es lo que ocurre cuando una primitiva es usada como argumento en un método sobrecargado.

```
//: c04:SobrecargaConPrimitivas.java
// Promocion de primitivas y sobrecarga.

public class SobrecargaConPrimitivas {
   // La conversion automatica no se aplica
   // a booleanos
```

```
static void prt(String s) {
   System.out.println(s);
}

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

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

void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }

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

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

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

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

void TestValConst() {
   prt("Probando con 5");
   f1(5);f2(5);f3(5);f4(5);f6(5);f7(5);
}
void testChar() {
   char x = 'x';
   prt("Argumento char:");
   f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
}
void testByte() {
   byte x = 0;
```

```
      prt{"Argumento byte:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testShort() {
      short x = 0;
      prt{"Argumento short:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testInt() {
      int x = 0;
      prt{"Argumento int:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testLong() {
      long x = 0;
      prt{"Argumento long:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testFloat() {
      float x = 0;
      prt{"Argumento float:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testDouble() {
      double x = 0;
      prt{"Argumento double:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   public static void main(String[] args) {
      SobrecargaConPrimitivas p =
        new SobrecargaConPrimitivas();
      p.testValConst();
      p.testChar();
      p.testByte();
      p.testShort();
      p.testInt();
      p.testLong();
      p.testFloat();
      p.testDouble();
   }
} ///:~
```

Si se examina la salida que produce este programa se puede ver que el valor constante 5 es tratado como un **int** , por lo tanto se usa el método sobrecargado con argumento **int** siempre que este disponible. En el resto de los casos, si el tipo que es inferior que el del argumento del método, es promovido al tipo superior. El tipo **char** se comporta de un modo ligeramente diferente ya que, cuando no se puede encontrar un método que tenga argumento **char** , es promovido a **int** .

Un variación del ejemplo anterior muestra que ocurre cuando el argumento es de un tipo superior al esperado por un método sobrecargado:

```
//: c04:Degradado.java
// Degradado de primitivas y sobrecarga.

public class Degradado {
   static void prt(String s) {
      System.out.println(s);
   }

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

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

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

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

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

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

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

   void testDouble() {
      double x = 0;
```

```
      prt("Argumento 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) {
      Degradado p = new Degradado();
      p.testDouble();
   }
} ///:~
```

En este caso hay métodos que tienen argumentos con tipos inferiores a los de los argumentos reales. Cuando se pasa a un método un valor de un tipo superior su argumento formal es necesario reducirlo al tipo aceptado por el método. ello se ha de poner entre paréntesis el nombre del tipo del argumento del método. Si no se hace así el compilador generará un mensaje de error.

Se debe tener muy en cuenta que se trata de una *conversión a un tipo inferior* por lo que se puede perder informacíon.Por ese motivo el compilador fuerza a que la transformación se haga de modo explícito.

Sobrecarga por valores de retorno

Es común preguntarse "?Por qué solo los nombres de las clases y la lista de argumentos? ?Por qué no distinguir entre métodos también usando el tipo de valor que devuelven? Por ejemplo los dos métodos siguientes tienen el mismo nombre y la misma lista de argumentos y pueden ser distinguidos fácilmente:

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

Esto funciona cuando el compilador puede determinar unívocamente el significado a partir del contexto, como en **int x=f()** . Sin embargo, es posible invocar un método e ignorar el valor que devuelve, lo que frecuentemente se llama *llamar al método por su efecto lateral* , ya que no lo importante no es el valor que el método devuelve sino otros efectos que produce la invocación. Por lo tanto si se invoca el método de la siguiente forma:

```
f();
```

Java no podría determinar cual de los dos métodos **f()** debe ser llamado. Nadie que leyese el código podría diferenciar entre uno y otro. Es por este tipo de problemas que no se puede usar el tipo de los valores devueltos por los métodos

como un criterio distinguir entre métodos sobrecargados.

Constructores predeterminados

Como ya se dijo más arriba el constructor predeterminado (también conocido como constructor "no-args") no tiene argumentos y se utiliza construir "objetos vainilla". Si se crea una clase que no tenga constructores, el compilador generará automáticamente un constructor predeterminado. Por ejemplo:

```
//: c04:CosntructorPredeterminado.java

class Pajaro {
   int i;
}

public class ConstructorPredeterminado {
   public static void main(String[] args) {
      Pajaro nc = new Pajaro(); // predeterminado!
   }
} ///:~
```

La línea:

```
new Pajaro();
```

crea un objeto nuevo y llama al constructor predeterminado, aunque no se haya definido explícitamente ninguno. Sin él no habría ningún método que invocar crear el objeto. Sin embargo, si se define algún constructor (con o sin argumentos) el compilador *no* genera ninguno por si mismo:

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

Si se intenta:

```
new Arbusto();
```

el compilador se quejará diciendo que no puede encontrar un constructor adecuado. Es como si, cuando no se incluye constructores, el compilador dijese: "Vas a necesitar *algún* constructor, así que déjame que prepare uno ti". Pero si se define algún constructor el compilador dirá: "Tienes un constructor, así que sabes lo que estás haciendo. Si no has definido un constructor predeterminado es por que no quieres que haya uno".

La palabra clave this

Si se tiene dos objetos, **a** y **b** , del mismo tipo cabe preguntarse como es posible invocar el mismo método **f()** cada uno de ellos:

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

Dado que solamente existe un método **f()** , éste debe ser capaz de determina cual de los dos objetos **a** o **b** ha sido invocado.

que sea posible escribir el código usando una sintaxis "orientada a objetos" en la que "se envía un mensaje a un objeto", el compilador debe hacer un poco de trabajo entre bastidores. Existe un argumento secreto que es pasado al método **f ()** en primer lugar. Este argumento es una referencia al objeto que se esta manipulando. Por lo tanto, las llamadas anteriores se traducen en algo así como:

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

Esta transformación se realiza internamente.No se puede escribir y compilar las dos líneas anteriores pero dan una buena idea de lo que ocurre.

Supongamos que dentro de un método se quiere obtener una referencia al objeto que se esta manipulando. Ya que el compilador ha pasado la referencia en *secreto* no hay ningún identificador que la contenga. Sin embargo, existe una palabra clave ello: **this** . La palabra clave **this** - que puede ser usada únicamente dentro de un método - devuelve la referencia al objeto el que el método ha sido invocado. Se puede usar esta referencia como cualquier otra referencia a un objeto. Conviene observar que, si se invoca un método de una clase desde otro método de la misma clase, no es necesario usar **this** ya que se puede invocar el método directamente. La referencia **this** es usada de modo automático. Por lo tanto es posible escribir:

```
class Melocoton {
   void coger() { /*...*/ }
   void pelar() { coger(); /*...*/ }
}
```

Dentro de **pelar()** se *puede* poner **this.coger()** pero no es necesario. El compilador lo hace automáticamente. La palabra clave **this** se usa únicamente en aquellos casos especiales en los que es necesita usar una referencia explícita al objeto que se está manipulando. Por ejemplo, es usando frecuentemente en la cláusula **return** cuando se quiere devolver una referencia al objeto en uso:

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

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

Como **incremento()** devuelve la referencia al objeto en uso usando la palabra clave **this** se pueden realizar varias operaciones sobre el mismo objeto de un modo sencillo.

Invocación de constructores dentro de constructores

En algunas ocasiones, cuando una clase tiene varios constructores, resulta conveniente llamar a un constructor desde dentro de otro no tener que duplicar el código. La palabra clave **this** permite realizarlo.

Normalmente, **this** se usa con el significado de "este objeto" o "el objeto actual" y produce una referencia al objeto con el que se está trabajando. Dentro de un constructor **this** toma otra significado cuando se le añade una lista de argumentos. En este caso representa una llamada explícita al constructor que tiene los mismos argumentos que los incluidos en la lista. Proporciona por lo tanto

un medio directo de invocar otros constructores:

```
//: c04:Flor.java
// Invocacion a constructores usando "this."

public class Flor {
   int contPetalos = 0;
   String s = new String("null");
   Flor (int petalos) {
      contPetalos = petalos;
      System.out.println(
        "Constructor con arg int solo, contPetalos= "
        + contPetalos);
   }
   Flor(String ss) {
      System.out.println(
        "Constructor con arg String solo, s=" + ss);
      s=ss;
   }
   Flor(String s, int petalos) {
     this(petalos);
//!     this(s);  // No se puede hacer dos veces.
      this.s = s; // Otro uso de "this"
      System.out.println("args String y int");
   }
   Flor() {
      this("hi",47);
      System.out.println(
        "constructor predeterminado (sin argumentos)");
   }
   void print() {
//!      this(11);  // Fuera de un constructor!
      System.out.println(
      "contPetalos = " + contPetalos + " s = " + s);
   }
   public static void main(String[] args) {
      Flor x = new Flor();
      x.print();
   }
} ///:~
```

El constructor **Flor(String s, int petalos)** muestra como se puede invocar un constructor usando **this** y que no se puede hacer dos veces. Además la llamada al constructor debe ser la primera sentencia o el compilador generará un error.

El ejemplo muestra también otra situación en la que se usa **this** . Como el nombre del argumento **s** y el nombre del dato miembro **s** es el mismo, se produce una

ambigüedad. Esta ambigüedad puede ser resuelta usando **this.s** referirse al dato miembro. Es frecuente el uso de esta forma en Java y aparece en numerosos lugares dentro de este libro.

En **print()** se ve que el compilador no permitirá que se llame a un constructor desde un método que no sea él mismo un constructor.

El significado de static

Con la palabra clave **this** en mente, se puede comprender mejor que significa que un método sea **static** . los métodos **static** no existe **this** . No es posible invocar un método no- **static** desde un método **static** (aunque lo contrario sí es posible) y se puede invocar un método **static** usando la clase, sin objeto alguno. De hecho, los métodos **static** son existen principalmente eso. Es el equivalente a un función global (de C), pero las funciones globales no están permitidas en Java. Definiendo un método **static** dentro de una clase se puede acceder a otros métodos **static** y a campos **static**

Hay quien opina que los métodos **static** no son orientados a objeto ya que significan lo mismo que una función global; con un método **static** no es posible enviar un mensaje a un objeto ya que no existe una referencia **this** asociada. Es un argumento razonable y si se empieza a usar *con frecuencia* es recomendable replantearse la estrategia que se está usando. Sin embargo, los métodos y campos **static** son prácticos y en ocasiones son realmente necesarios. Luego, si son o no programación orientada a objetos en estado puro, es una cuestión que se puede dejar a los teóricos. En realidad incluso Smalltalk tiene algo equivalente con sus "métodos de clase".

Purgado: finalización y recogida de basura

Los programadores son conscientes de la importancia de la inicialización, pero a menudo se olvidan de la importancia del purgado. Después de todo, ?quién necesita deshacerse de un **int** ?. Pero cuando se trabaja con librerías, la política de "déjalo estar" cuando se ha terminado con un objeto no es siempre segura. Por supuesto, Java tiene un recogedor de basura que se encarga de reclamar la memoria que los objetos ya no usan. Consideremos ahora un caso muy extraño. Supongamos que nuestros objetos reservan memoria "especial" sin usar **new** . El recogedor de basura solo sabe como liberar la memoria que ha sido reservada usando **new** , por lo que no sabrá como liberar la memoria "especial" de los objetos. estos casos Java proporciona un método llamado **finalize()** que puede ser definido dentro de una clase. Veamos como *se supone* que este método trabaja. Cuando el recogedor de basura está listo liberar la zona de memoria de memoria usada por un objeto invoca en primer lugar su **finalize()** y solo en el siguiente ciclo de recogida de basura reclamará la memoria del objeto. Por lo tanto si se usa **finalize()** es posible realizar alguna operación de limpieza importante durante *la recogida de basura* .

Esto puede ser un problema por que algunos programadores, especialmente para aquellos que vienen de C++, tienden a confundir **finalize()** con el destructor de C++: una función que es invocada automáticamente cuando un objeto es destruido. Es importante entender desde ahora mismo las diferencias entre C++ y Java en este punto: en C++ *los objetos son siempre destruidos* (en un programa sin errores) mientras que en Java los objetos no siempre terminan por ser arrojados a la basura. En otras palabras:

Recogida de basura no es lo mismo que destrucción

Si se recuerda esto es posible mantenerse al margen de los problemas. El enunciado anterior implica que, si es necesario hacer alguna cosa antes de que deje de ser necesario un objeto, ese "algo" debe ser hecho por uno mismo. Java no dispone de destructores o de cualquier otra construcción similar, por lo que es necesario crear métodos ordinarios que se encargue de esta tarea de purgado. Pongamos el ejemplo de un objeto se pinta a si mismo en la pantalla del ordenador al ser creado. Si la imagen no es borrada explícitamente de la pantalla, puede que el objeto nunca llegue a ser purgado del sistema. Solo si se introduce dentro de **finalize()** las instrucciones de borrado pertinentes, la imagen será borrada cuando el objeto sea enviado a la basura, de otro modo la imagen permanecerá pintada en la pantalla. Por lo tanto, el segundo punto a recordar es:

Es posible que los objetos no sean procesados por el recogedor de basura

Si un programa nunca llega a un punto en el que los recursos disponibles empiecen a escasear, el espacio ocupado por los objetos no es liberado. Si el programa termina y el recogedor de basura no ha liberado espacio para ninguno de los objetos del mismo, todo el espacio reservado por los objetos durante la ejecución del programa el liberado y devuelto al sistema operativo de una sola vez a la finalización del programa. Dado que la recogida de basura supone un trabajo extra que es necesario realizar, resulta adecuado evitarla cuando sea posible.

¿Para qué sirve finalize() ?

En este punto de la discusión, es posible pensar que **finalize()** no debería ser usado como un método de borrado de propósito general. ¿Qué tal es?

El tercer punto a recordar es:

La recogida de basura afecta solo a la memoria.

La única razón de exitir del recogedor de basura es recuperar la memoria que el programa ha dejado de utilizar. Por lo tanto cualquier actividad relacionada con la recogida de basura, en particular el método **finalize()** , debe ocuparse únicamente de la memoria y de su liberación.

Esto no significa que, si un objeto contiene otros objetos, **finalize()** debe eliminar explícitamente todos estos objetos: el recogedor de basura se encarga de liberar

la memoria asignada a cada objeto independientemente de como hayan sido creados. El método **finalize()** se necesita únicamente en aquellos casos especiales en los que un objeto reserve espacio para si de algún modo que no sea la creación de otros objetos. Pero si en Java todo es un objeto, ¿cómo puede ser esto posible?

Parece como si **finalize()** estuviese ahí para el caso en que, en el estilo de C, se reservase memoria usando un mecanismo diferente del normal en Java. Esto se puede hacer, principalmente, usando *métodos nativos* que permiten invocar desde Java código escrito en otro lenguaje de programación. (En el Apéndice B se puede encontrar un amplia descripción de los métodos nativos). De momento, C y C++ son los únicos lenguajes que pueden utilizarse con los métodos nativos pero, dado que desde C y C++ es realizar llamadas a subprogramas escritos en otros lenguajes, en realidad se pueden hacer llamadas a cualquier cosa. En un fragmento de código que no sea Java, se puede usar la familia de funciones **malloc()** de C para reservar espacio. A menos que se use luego la función **free()** el espacio nunca será liberado lo que producirá una pérdida de memoria (memory leak). Como **free()** es una función de C y C++, se necesita un método nativo para poder llamarla desde dentro de **finalize()** .

De la lectura de los párrafos anteriores se desprende que **finalize()** no se usa con mucha frecuencia. No es le lugar adecuando para purgar el sistema de los objetos que ya no están en uso. Entonces ¿donde debe hacerse este purgado?

El purgado es necesario

Para eliminar un objeto, el usuario de dicho objeto debe invocar un método de purgado cuando se desee hacerlo. Esto parece bastante claro pero se contradice con el concepto de destructor en C++. En C++ todos los objetos son destruidos. O mejor dicho: todos los objetos *deberian ser* destruidos. Si, en C++, un objeto es creado localmente (es decir, en la pila, lo que no es posible en Java), su destrucción tiene lugar en el punto en el que se cierra el ámbito de definición (scope) en que fue credo el objeto. Si, en cambio, el objeto fue creado usando **new** (como en Java) el destructor es invocado cuando el programador llama al operador **delete** de C++ (que no tiene equivalente en Java). Si el programador de C++ olvida incluir la llamada a **delete** el destructor nunca será invocado y se producirá una fuga de memoria; además los otros elementos del objetos no podrá ser eliminados nunca, Los errores de este tipo son muy difíciles de detectar.

Al revés que C++, Java no permite la creación de objetos locales: es necesario usar siempre **new** . Pero en Java no existen un "delete" que deba ser llamado para eliminar el objeto ya que cuenta con el recogedor de basura que se encarga de liberar el espacio previamente reservado. Por lo tanto de puede decir simplificando que Java no necesita destructores por que tiene un recogedor de basura. El lector comprobará, según avance en la lectura de este libro, que la presencia de un recogedor de basura no elimina la necesidad o utilidad de los destructores. (además nunca debería llamarse directamente **finalize()** , por que

ese no es el camino adecuado hacia una solución) Si se quiere realizar algún tipo de purgado, además de la liberación de espacio, se debe hacer de modo explícito llamando al método apropiado desde Java, lo que equivale a un destructor de C++ sin comodidad que éste proporciona.

Una de las cosas para las que puede ser útil **finalize()** , es para ver como funciona la recogida de basura. El ejemplo siguiente muestra que es lo que ocurre y resume las descripciones de la recogida se basura dadas hasta ahora:

```
//: c04:Basura.java
// Demostracion del funcionamiento del
// recogedor de basura y la finalizacion

class Silla {
   static boolean gcrun = false;
   static boolean f = false;
   static int creadas = 0;
   static int finalizadas = 0;
   int i;
   Silla() {
      i = ++creadas;
      if (creadas == 47)
         System.out.println("Creada 47");
   }

   public void finalize() {
      if(!gcrun) {
         // La primera que finalize es llamado:
         gcrun = true;
         System.out.println(
           "Empezando a finalizar despues de que " +
           creadas + " Sillas hayan sido creadas");
      }
      if(i == 47) {
         System.out.println(
           "Finalizando Silla #47, " +
           "Fijando indicador para detener la creacion de Sillas");
         f = true;
      }
      finalizadas++;
      if (finalizadas >= creadas)
         System.out.println(
           "Todas " + finalizadas + " finalizadas");
   }
}

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

```
      // Mientras no se fije el indicador,
      // hacer Sillas y Strings:
      while(!Silla.f) {
         new Silla();
         new String("Para gastar un poco de espacio");
      }
      System.out.println(
         "Despues de que todas las sillas han sido creadas:\n" +
         "Total creadas = " + Silla.creadas +
         ", total finalizadas = " + Silla.finalizadas);
       // Argumentos opcionales para forzar
       // recogida de basura y finalizacion
       if(args.length > 0) {
          if(args[0].equals("gc") ||
             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 algún momento el recogedor de basura comienza a ejecutarse, el programa detiene al creación de **Silla** s. Como el recogedor de basura puede ejecutarse en cualquier momento, no se puede saber con exactitud cuando arrancará. Por este motivo se usa el indicador **gcrun** para señalar si el recogedor de basura se ha comenzado a ejecutar ya. El segundo indicador **f** sirve para que **Silla** le diga a bucle en **main()** que debe dejar de crear objetos. Ambos indicadores son activados dentro de **finalize()** que es invocado durante la recogida de basura.

Otras dos variables **static** , **creadas** y **finalizadas** llevan la cuenta del número de **Silla** s creadas frente al de finalizadas por el recogedor de basura. Finalmente, cada **Silla** tiene su propia (no- **static**) **int i** , de modo que puede llevar la cuenta de que número hace dentro de la secuencia. Cuando la **Silla** número 47 es finalizada, el indicador **f** se le asigna el valor **true** para detener el proceso de creación de **Silla** .

Todo esto sucede en **main()** , dentro del bucle:

```
while(!Silla.f) {
```

```
        new Silla();
        new String("Para gastar un poco de espacio");
    }
```

El lector puede preguntarse como puede detenerse este bucle si no hay nada dentro de él que cambie el valor de **Silla.f** . Sin embargo, será **finalize()** quien lo haga con el tiempo, tras finalizar la **Silla** número 47.

El objeto **String** que se crea en cada iteración sirve tan solo para gastar un poco de espacio y acelerar la entrada del recogedor de basura, lo que ocurrirá en cuanto empiece a ponerse nervioso sobre la cantidad de memoria disponible.

Al lanzar el programa desde la línea de comandos es posible incluir un parámetro: "gc", "finalizar", o "todo". Con el argumento "gc" se llama al método **System.gc()** (para forzar la ejecución del recogedor de basura). Con "finalizar" se llama a **Systen.runFinalization()** que - en teoría - hará que sean finalizados todos los objetos que no lo estuviesen ya. Con el argumento "all" se llama a ambos métodos.

El comportamiento de este programa y la versión en la primera edición de este libro, ponen de manifiesto que todo el asunto de la recogida de basura y la finalizacion ha evolucionado y que, la mayor parte de esta evolución se ha producido en secreto. De echo, puede ser que el comportamiento del programa haya cambiado de nuevo cuando el lector este leyendo esta líneas.

Si se llama a **System.gc()** , todos los objetos son finalizados. Éste no era necesariamente el caso con la implementaciones anteriores del JDK, aunque la documentación insista en lo contrario. Además, se puede comprobar que no hay ninguna diferencia aparente si la llamada se hace a **System.runFinalization()**

Sin enbargo, se puede observar que, solamente si **System.gc()** es llamado después de que todos los objetos estén creados y descartados, serán invocados todos los finalizadores. Si no se llama a **System.gc()** solo seran finalizados algunos de los objetos creados. En Java 1.1 se introdujo el método **System.runFinalizersOnExit()** que forzaba a los programas a invocar a todos los finalizadores antes de termina. Sin embargo, el diseño resulto incorrecto y el método fue retirado (deprecated). Este es otro indicio más de que los diseñadores de Java seguían trabajado para intentar resolver el problema de la recogida de basura y la finlización. Solo cabe esperar que las cosas funcionen en Java2.

El ejemplo precedente muestra que el compromiso de que los finalizadores son ejecutados siempre, continúa siendo cierta, pero únicamente si se fuerza explícitamente. Si no se fuerza la llamada a **System.gc()** , se obtiene una salida como la siguiente:

```
Creada 47
```

```
Empezando a finalizar despues de que 3486 Sillas hayan sido creadas
Finalizando Silla #47, Fijando indicador para detener la creacion de S
Despues de que todas las sillas hayan sido creadas:
Total creadas = 3881, total finalizadas = 2684
Adios!
```

Por lo tanto, no todos los finalizadores son llamados cuando el programa termina. Si **System.gc()** es llamado, finalizará y destruirá todos los objetos que ya no estén en uso en ese momento.

Hay que recordar que ni la recogida de basura ni la finalización están garantizadas. Si la Máquina Virtual Java (JVM - Java Virtual Machine) no corre el riesgo de quedarse sin menoria, no malgastará el tiempo recuperando memoria a través de la recogida de basura.

Death Condition

En general, no se puede confiar en que **finalize()** sea llamado, y se deben crear funciones independientes de "purgado" que han de ser llamadas explícitamente. Luego, parece que **finalize()** es útil únicamente en extraños purgados de memoria que la mayoría de los programadores nunca usan. Existe, sin embargo, un uso muy interesante de **finalize()** que no depende en que sea llamada todas las veces. Este uso es la comprobación de la *death condition* de un objeto.

En el momento en que un objeto deja de ser de interés - cuando el está listo para ser eliminado - debe encontrarse en un estado tal que la memoria a él asignada pueda ser liberada de un modo seguro. Por ejemplo, si el objeto representa un fichero abierto, ese fichero debería ser cerrado por el programador antes de que el objeto sea enviado a la basura. Si alguno de los componentes del objeto no es eliminado adecuadamente se produce un error en el programa que puede ser muy difícil de encontrar. El valor de **finalize()** puede ser usado para descubrir este tipo de situaciones, aún cuando no sea llamado en todas la ocasiones. Si una de la finalizaciones revela el error, el problema queda al descubierto, que es precisamente lo que se pretende.

El siguiente ejemplo muestra como usar **finalize()** con esta finalidad:

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

class Libro {
   boolean retirado = false;
   Libro(boolean retirar) {
      retirado = retirar;
   }
```

```
   void ingresar() {
      retirado =  false;
   }
   public void finalize() {
      if (retirado)
         System.out.println("Error: libro retirado");
   }
}

public class DeathCondition {
   public static void main(String[] args) {
      Libro novela = new Libro(true);
      // Purgado correcto
      novela.ingresar();
      // Desprecia la referencia, olvida limpiar
      new Libro(true);
      // Fuerza la recogida de basura y la finalizacion
      System.gc();
   }
} ///:~
```

En este caso se supone que todo **Libro** tiene que ser ingresado antes que pueda ser enviado a la basura. Pero en **main()** el programador no registra un de los libros por error. Si **finalize()** no verificase la death condition, este error sería muy difícil de descubrir.

System.gc() se usa para forzar la finalización (lo que resulta conveniente durante el desarrollo para facilitar la depuración de errores). Pero, incluso aunque no estuviese, es más que probable que el **Libro** errante termine por aparecer durante alguna de las ejecuciones del programa (supuesto que se llegue a reservar memoria suficiente para que el recogedor de basura entre en acción).

El funcionamiento del recogedor de basura

Para aquellos que vengan de lenguajes de programación en los que la creación de objetos en el heap es una operación costosa, pueden pensar que el esquema de crear todo (salvo las primitivas) en heap adoptado por Java, es ineficiente.

Resulta, sin embargo, que la existencia del recogedor de basura puede tener un impacto significativo en la rapidez a la que se crean los objetos. Puede sonar un poco raro al principio que la liberación de memoria afecte a reserva de espacio, pero es como trabajan algunas JVMs y significa, al final, que la creación de objetos en el heap puede ser tan rápida en Java como la creación de objetos en la pila en otros lenguajes de programación.

Por ejemplo, en C++ el heap puede ser visto como un jardín en el que cada objeto se ocupa de cada su propio trozo de césped. Esta propiedad puede que ser abandonada y, con el tiempo, reutilizada por otro objeto. En varias JVMs, el heap

es bastante diferente: se asemeja más a una cinta transportadora que se mueve hacia adelante cada vez que se crea un objeto nuevo. Esto implica que la creación de objetos es notablemente rápida. El "puntero del heap" simplemente se mueve hacia adelante, adentrándose en territorio virgen, por lo que, a final de cuentas, es lo mismo que la reserva de espacio en la pila con C++ (Naturalmente, hay un pequeño trabajo extra que realizar para mantener la información acerca de la ocupación en el heap, pero es nada en comparación com andar buscando espacio libre por el heap).

El heap, sin embargo, no es una cinta transportadora y, si se manejase como tal tarde o temprano el programa empezaría a paginar (lo que influye muy negativamente en el rendimiento) y acabaría por agotar la memoria disponible. El truco está en que el recogedor de basura, al mismo tiempo que elimina la basura, compacta todos los objetos en el heap con lo que el "puntero del heap" es desplazado hacia el comienzo de la cinta transportadora y evitando el fallo de página. El recogedor de basura reordena las cosas y hace posible que el modelo del heap infinito de alta velocidad pueda ser utilizado para la asignación de espacio.

Para entender como funciona el recogedor de basura (GC del inglés, "garbage collector"), es necesario conocer mejor como trabajan los diferentes algoritmos de recogida de basura. Una técnica simple pero lenta de recogida de basura es el conteo de referencias (reference counting). En este modelo cada objeto debe contener un contador de referencias que se incrementa cada vez que se asigna una nueva referencia a dicho objeto. Cada vez que el programa abandona el ámbito de validez de una referencia o se le asigna el valor **null** , el contador de referencias del objeto asociado se decrementa en una unidad. La gestión de los contadores de referencias supone por lo tanto una pequeña, pero constante, carga adicional que se entiende durante toda la vida del programa. El recogedor de basura recorre toda la lista de objetos y, cuando encuentra uno cuyo contador de referencias es cero, libera el espacio reservado para dicho objeto. Esta técnica, sin embargo, no resulta adecuada para identificar conjuntos de objetos que contenga referencias de unos a otros de una manera circular de modo que, aun siendo solo basura, sus contadores de referencias sean distintos de cero. La identificación de estos grupos autoreferenciados requiere del recogedor de basura un trabajo extra considerable. El conteo de referencias se usa con frecuencia para describir un tipo concreto de recogida de basura pero probablemente no se use en ninguna implementación de la JVM.

Otros esquemas de recogida de basura más eficientes no usan la técnica de conteo de referencias. Es su lugar usan la idea de que cualquier objeto que permanezca vivo debe, en última instancia, ser rastreado hacia atrás hasta encontrar una referencia en la pila o en la zona de almacenamiento estático. La cadena puede extenderse a través de varias capas de objetos. Por lo tanto, es posible identificar que objetos están vivos sin más que seguir las referencias que se encuentran en la pila y en la zona de almacenamiento estático. Por cada referencia que se encuentre, hay que entrar dentro del objeto apuntado por ella, identificar todas las

referencias contenidas en *ese* en ese objeto, y entrar en los objetos apuntados por estas referencias, y así sucesivamente hasta recorrer por completo red de objetos generada a partir de las referencias en la pila y en la zona de almacenamiento estático. Cada objeto que se atraviese en este proceso debe ser un objeto vivo. En este caso, los conjuntos autoreferenciados no presentan ningún problema; simplemente no aparecen por ningún lado, y son, por lo tanto, enviados automáticamente a la basura.

En el enfoque descrito aquí, la JVM utiliza un esquema *adaptativo* de recogida de basura: lo que haga con los objetos vivos que encuentre dependerá de la variante que se encuentre activa en ese momento. Una de estas variantes es *para-y-copiar (stop-and-copy)* . Esto significa que - por razones que se harán evidentes más adelante - el programa es parado en primer lugar (por lo tanto no se trata de un esquena de recogida de basura en segundo plano). A continuación, cada objeto vivo que se encuentre en el heap es copiado en otro heap, con lo que la basura se queda en el primero. Además, los objetos son copiados en el nuevo heap uno a continuación del otro con lo que se optimiza de paso la utilización del espacio (y permitiendo así que los objetos que se creen puedan ser simplemente añadidos al final del heap como ya se dijo más arriba)

Naturalmente, cuando se mueve un objeto de un lugar a otro, todas las referencias que apuntan al (es decir esa *refencia*) objeto deben ser actualizadas. Las referencias que van del heap????? o la zona de almacenamiento estático al objeto pueden ser actualizadas inmediatamente; pero puede que existan otras referencias, apuntando a este objeto, que serán encontradas más tarde durante el paseo. Estas referencias son corregidas según se van encontrando (el lector puede imaginar una tabla conteniendo la correspondencia entre las direcciones nuevas y antiguas)

Hay dos razones por las que estos "recogedores por copia" (copy collectors") resultan ineficientes. La primera es que se necesitan dos heaps y copiar la información de uno a otro continuamente, por lo que hay mantener dos veces mas memoria de la que es realmente necesaria. Algunas JVMs tratan de aliviar el problema realizando las asignaciones de espacio en el heap por segmentos, según va siendo necesario, y copiando de un segmento a otro.

La segunda razón tiene que ver con la copia de un heap al otro. Una vez que el programa se ha estabilizado puede ser que genere muy poca basura o incluso que no genere basura en absoluto. Aún así, el recogedor por copia continuará todo el contenido de la memoria de un lugar a otro. Para evitar esta situación, algunas JVMs son capaces de detectar que no se está generando basura nueva y cambiar su modo de funcionamiento (de aquí lo de *adaptativa*). El segundo modo de trabajo recibe el nombre de *marcar y barrer (mark and sweep)* y fue el método usado por Sun en las primeras versiones de su JVM. Este método, usado en condiciones generales, es un poco lento pero, si se sabe que la cantidad de basura que se está generando es pequeña, es bastante rápido.

Marcar y barrer sigue la misma lógica de comenzar en la pila y en la zona de

almacenamiento estático y seguir todas la referencias para encontrar todos los objetos vivos. Cada objeto vivo encontrado se marca activando un indicador dentro del objeto. Solo cuando el proceso de marcado ha finalizado, tiene lugar el barrido, en el que los objetos muertos son eliminados. En cualquier caso, no se realiza copia de objetos, como con el método anterior, y cuando el recogedor de basura decide compactar el heap, lo hace desplazando los objetos dentro del heap.

El nombre "parar-y-copiar" hace referencia a hecho de que este tipo de recogida de basura no se realiza en segundo plano. El programa ha de ser detenido mientras el GC cumple con su misión. En la documentación de SUN se puede encontrar multitud de referencias en la que se habla de la recogida de basura como un proceso de prioridad baja que se ejecuta en segundo plano. Sin embargo, el GC nunca ha sido implementado de ese modo, al menos en las primeras versiones de la JVM de Sun. El recogedor de basura de Sun se ejecutaba siempre que la cantidad de memoria disponible alcanzaba un mínimo. Además el esquema marcar-y-barrer requiere que el programa sea detenido.

Como ya se dijo mas arriba, en la JVM que se está describiendo aquí, la memoria se asigna en bloques grandes. Cuando se crea un objeto grande, recibe su propio bloque. En un sistema en el que se aplique la política parar-y-copiar de un modo estricto, hay que copiar cada objeto vivo del heap original al nuevo antes de que se puedan liberar la memoria asignada el primero; esto implica grandes cantidades de memoria. Los bloques permiten al CG copiar los objetos vivos que va recogiendo en bloques muertos. Cada bloque tiene un *numero de generación* para llevar la cuenta de si está vivo o no. Normalmente, solo los bloques creados desde le ultima ejecución del GC son compactados; con el resto de los bloques se aumenta su numero de generación si se ha hecho alguna referencia a ellos. Esta política resulta adecuada para el caso en que se tenga muchos objetos de vida breve. Periódicamente, se realiza un barrido completo; los objetos grandes no son copiados (tan solo sus números de generación son actualizados) y los bloques con objetos pequeños son copiados y compactados. La JVM monitoriza el rendimiento del GC; si detecta que este caído bajo un cierto limite, debido a que la mayoría de los objetos tienen un tiempo de vida largo, cambia al modo marcar-y-barrer; si el heap comienza a estar fragmentado, cambia de nuevo al modo parar-y-copiar. Por este motivo se dice que el esquema es adaptativo, por lo que al final se tiene el bonito nombre de "adaptativo generacional parar-y-copiar marcar-y-barrer".

Hay varias posibilidades de optimizar el proceso de la recogida de basura dentro de la JVM. Una de las más importantes está relacionada con el funcionamiento del cargador y el compilador Just-In-Time (JIT). Cuando una clase tiene que ser cargada en el sistema (normalmente cuando se va a crear el primer objeto de la clase), ha de localizarse el fichero **.class** y código de bytes tiene que ser cargado en memoria. En ese momento, se puede compilar, usando el JIT, todo el código pero hay dos inconvenientes. El primero es que se introducen pequeños retardos que pueden combinarse unos con otros relentizando, al final, la ejecución del programa. El segundo inconveniente es que aumenta el tamaño de los ejecutables (el código en bytes en mucho más compacto que el código que genera el JIT) lo

que puede provocar que el programa empiece a paginar reduciendo drásticamente la velocidad de ejecución. Una estrategia alternativa es la *evaluacion débil (lazy evaluation)* que consiste en no compilado por el JIT hasta que no sea necesario. De este modo, el código que no llegue nunca a ser ejecutado, no será compilado por el JIT.

Inicialización de miembros

Java hace todo lo necesario para garantizar que las variables son inicializadas correctamente antes de que sean usadas. En el caso de que la variable esté definida localmente dentro de un método, el compilador generará un mensaje de error si no ha sido inicializada explícitamente. Por ejemplo, al intentar compilar:

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

se obtiene un mensaje de error diciendo que **i** no pudo ser inicializada. El compilador podría asignar a **i** un valor por defecto, pero lo más probable es que se trate de un error del programador que quedaría oculto si se asignase un valor por defecto. Es más fácil localizar un error obligando al programador a inicializar las variables.

Las cosas son un poco diferentes si la primitiva es un dato miembro. Como cualquier método puede iniciar o use el dato, no seria practico obligar al usuario a inicializarlo con un valor adecuado antes de utilizarlo. Pero, dejar que el dato contuviese basura es peligroso, así que Java se encarga de que cada dato miembro de la clase reciba un valor inicial. El siguiente ejemplo muestra cuales son estos valores:

```
//: 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 print() {
```

```
      System.out.println(
         "Tipo          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.print();
      /* En este caso se puede poner tambien:
      new Medidas().print();
      */
   }
} ///:~
```

El programa produce la siguiente salida:

```
Tipo          Valor Inicial\n" +
boolean       false
char          [ ] 0
byte          0
short         0
int           0
long          0
float         0.0
double        0.0
```

El valor de **char** es cero, que aparece como un espacio.

Más adelante se verá que cuando se define una referencia a un objeto dentro de una clase sin inicializarla con un objeto, la referencia toma el valor especial **null** (que es una palabra clave de Java).

El lector puede comprobar que, aunque no se especifique ningún valor, todas las variables son inicializadas. Por lo tanto, no hay peligro de trabajar con variables no inicializadas.

Especificando la inicialización

¿Qué ocurre si se desea dar un valor inicial a una variable? Una forma de hacerlo es simplemente asignar el en donde la variable es definida dentro de la clase (No es posible hacerlo en C++ aunque los programadores novatos lo intentan siempre). En el siguiente fragmento de código se han modificado la definición de los campos de la clase **Medida** para asignarles 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;
   //. . .
```

No solo las primitivas, cualquier objeto puede ser inicializado del mismo modo. Si **Profundidad** es una clase, se puede insertar una variable e inicializarla como sigue:

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

Si se intenta usar **o** sin haberle inicializado, se obtiene en tiempo de ejecución un error que recibe el nombre de *excepción(exception)* . Las excepciones son tratadas en captítulo 10)

También se puede incluir la llamada a un método en la inicialización:

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

El método puede tener argumentos, pero los argumentos no pueden ser miembros de la clase que todavía no hayan sido inicializados. Luego el siguiente segmento

de código es correcto:

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

En este caso el compilador se quejará, apropiadamente, de que ha encontrado una forward reference, ya que el problema está en el orden de la inicialización y en como es compilado el programa.

Esta modo de inicialización es simple y directo. Tiene, por contra, la limitación de que *todos* los objetos del tipo **Medida** serán inicializados con los mismos valores.Unas veces es esto justamente lo que se quiere; en otras ocasiones se necesita más flexibilidad.

Inicialización en el constructor

El constructor se puede usar realizar la inicialización. Esta posibilidad, da una gran flexibilidad a la hora de programar, ya que se puede hacer llamadas a métodos y realizar otras acciones en tiempo de ejecución determinar los valores iniciales. De cualquier modo, hay que tener siempre en mente que de este modo no se evita la inicialización automática que tiene lugar inclusa antes de que se haya invocado el constructor. Así, en el ejemplo siguiente,

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

la variable **i** será inicializada primero con 0 y luego con 7. Este principio se aplica a todos los tipos primitivos y a las referencias a objetos, incluso en los casos en que se hace una inicialización explícita en el momento de la definición. Por esta razón, el compilador no obliga a inicializar elementos en el constructor o antes de que sean usados: la inicialización está, por lo tanto, garantizada.

Orden de inicialización

Dentro de una clase, el orden de inicialización queda determinado por el orden en que las variables son definidas dentro de la clase. Las definiciones de las variables pueden estar esparcidas por o entre la definición de los métodos pero las variables son inicializadas antes de que cualquier método pueda ser invocado; incluido el

constructor. Por ejemplo:

```
//: c04:OrdenDeInicializacion.java
// Muestra cual es el orden de inicialización.

// Cuando el constructor es llamado crea
// un objeto Etiqueta, se vera el mensaje:
class Etiqueta {
   Etiqueta(int marcador) {
      System.out.println("Etiqueta(" + marcador + ")");
   }
}

class Tarjeta {
   Etiqueta t1 = new Etiqueta(1); // Antes del constructor
   Tarjeta() {
      // Indica que estamos dentro del constructor:
      System.out.println("Tarjeta()");
      t3 = new Etiqueta(33);      // Reinicializacion de t3
   }
   Etiqueta t2 = new Etiqueta(2); // Despues de 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();   // Demuestra que se ha completado el constructor
   }
} ///:~
```

En **Tarjeta** las definiciones de los objetos **Etiqueta** se han desperdigado demostrar que todos ellos son inicializados antes de que se ejecute el constructor y antes de que pueda ocurrir cualquier otra cosa. Además, **t3** es reinicializado dentro del constructor. La salida es:

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

De este modo, la referencia **t3** es inicializado dos veces; la primera antes de la llamada al constructor y la segunda durante ella. (El primer objeto es abandonado , por lo que podrá ser eliminado por el recogedor de basura más adelante). Esta estrategia puede parecer poco eficiente, que garantiza que la inicialización se realiza adecuadamente; por que ¿qué ocurriría si se definiese un segundo constructor que *no* inicializase **t3** y no hubiese hubiese una inicialización "por defecto" en su definición?

Inicialización de datos estáticos

Lo espuesto en el párrafo anterior se aplica también a los datos estáticos. Si una primitiva no es inicializada explícitamente, recibe el valor inicial estándar que le corresponda. Si es una referencia a un objeto, se le asigna el valor **null** a menos que se cree un objeto y se le asocie dicha referencia.

Si se quiere inicializar un datos en el mismo sitio en que se define, se hace como el caso de datos no estáticos. Hay solo una zona de memoria un dato estático, independientemente de cuantos objetos sean creados, lo que plantea la cuestión de cuando es inicializada la zona de memoria que corresponde al datos estático. El ejemplo siguiente aclara esta cuestión:

```
//: c04:InicializacionEstatica.java
// Especificacion de valores iniciales
// en la definicion de una clase.

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

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

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

public class InicializacionEstatica {
   public static void main(String[] args) {
      System.out.println(
         "Creando nuevo Armario() en main");
      new Armario();
      System.out.println(
         "Creando nuevo Armario() en main");
      new Armario();
      t2.f2(1);
      t3.f3(1);
   }
   static Mesa t2 = new Mesa();
   static Armario t3 = new Armario();
} ///:~
```

Bol permite ver la creación de la clase y **Mesa** y **Armario** crean miembros del tipo **Bol** en diferentes posiciones dentro de la definición de sus clases respectivas. Se observa que **Armario** crea el objeto no estático **Bol b3** antes que las definiciones **static** . El programa produce la siguiente salida:

```
Bol(1)
Bol(2)
Mesa()
f(1)
Bol(4)
Bol(5)
Bol(3)
Armario()
f(2)
Creando nuevo Armario() en main
Bol(3)
Armario()
f(2)
```

```
Creando nuevo Armario() en main
Bol(3)
Armario()
f(2)
f2(1)
f3(1)
```

Los datos estáticos son inicializados únicamente si es necesario. Si no se crea ninguna **Mesa** o si nunca se accede a **Mesa.b1** o **Mesa.b2** miembros estáticos **b1** y **b2** del tipo **Bol** nunca serán creados. Además, serán inicializados únicamente una vez: la *primera* que un objeto *Mesa* sea creado (o cuando se produzca el primer acceso estático) Después los objetos estáticos no vuelven a ser inicializados.

Los miembros estáticos son, pues, los primeros en ser inicializados, si no lo han sido ya como consecuencia de la creación un objeto anterior, y a continuación se inicializan los datos no estáticos, como se puede verificar examinando la salida del ejemplo anterior.

Resulta útil resumir aquí el proceso de creación de un objeto. Considérese la clase **Perro** :

1. La primera vez que un objeto del tipo **Perro** es creado, *o* la primera vez que se accede a un método estático o a un campo estático, el interprete de Java debe localizar el fichero **Perro.class** buscando por entre las diferentes localizaciones indicadas por *classpath*
2. A medida que **Dog.class** es cargado (creando un objeto **Class** como se verá más adelante) se ejecutan todos sus inicializadores estáticos. De este modo, la inicialización estática se realiza solo una vez: mientras el objeto **Class** es cargado por primera vez.
3. Cuando se crea un nuevo objeto **Perro** , mediante **new Perro()** , el proceso de construcción del objeto **Perro** reserva, en primer lugar, espacio suficiente en el heap contener el objeto.
4. El espacio reservado en el heap es automáticamente con ceros, lo que automáticamente asigna a las primitivas definidas dentro de **Perro** sus valores por defecto (cero los números y su equivalente los campos **boolean** y **char** y **null** a las referencias.
5. Se ejecutan todas las inicializaciones especificadas junto a la definición de los campos.
6. Se ejecutan los constructores. Como se verá en el capítulo 6 esto puede significar una gran cantidad de trabajo, especialmente cuando hay involucrada herencia.

Inicialización estática explícita

Java cuenta con una "construcción estática" que permite agrupar varias inicializaciones de datos **static** en un solo lugar dentro de una clase. Esta construcción recibe a veces el nombre de *bloque estático (static block)* . El siguiente ejemplo muestra un ejemplo de esta construcción.

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

Aunque parece la definición de un método, se trata solamente de la palabra clave **static** seguida por el cuerpo de método. Este fragmento de código, como cualquier otra inicialización estática, se ejecuta tan solo 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 la clase (incluso cuando no se llegue a crear un solo objeto de esa clase). Por ejemplo:

```
//: c04:ExplicitaEstatica.java
// Inicializacion explicita de datos
// estaticos con una clausula "static"

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

class Tazas {
   static Taza t1;
   static Taza t2;
   static {
      t1 = new Taza(1);
      t2 = new Taza(2);
   }
   Tazas() {
      System.out.println("Tazas()");
   }
}

public class ExplicitaEstatica {
   public static void main(String[] args) {
      System.out.println("Dentro de main()");
      Tazas.t1.f(99);   // (1)
   }
   // static Tazas x = new Tazas();  // (2)
   // static Tazas y = new Tazas();  // (2)
```

```
} ///:~
```

Los inicializadores estáticos de **Copas** se ejecutan cuando se accede al objeto estático **t1** en la línea (1) o si la línea (1) se comenta la línea (1) y se descomentan las líneas marcadas con (2). Si tanto la línea marcada con (1) como las marcadas con (2) son comentadas, la inicialización estática en **Tazas** no llega nunca a tener lugar. Por otro lado, no cambia nada si se quitan las marcas de comentario de ambas o solo una de las líneas (2): la inicialización estática solo ocurre una vez.

Inicialización de instancias no estáticas

La sintaxis en Java para inicializar las variables de cada objeto que no sean estáticas, es similar a la del caso estático. Por ejemplo:

```
//: c04:Tazones.java
// Inicializacion de instancias en Java.

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

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

Puede comprobarse que la clausula de inicializacion de la instancia:

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

parece exactamente igual que la clausula de inicialización estática salvo que no contiene la palabra clave **static** . Esta sintaxis es necesaria para permitir la inicialización de *clases internas anónimas (anonymous inner class)* como se verá en el capítulo 8.

Inicialización de arrays

La inicialización de arrays en C es una tarea tediosa en la que es fácil cometer errores. En su lugar, C++ utiliza inicialización de agregados (aggregate initialization) lo que hace la tarea más segura. Java no tiene "agregados" como C++ ya que en Java todo es siempre un objeto. Java tiene arrays y además proporciona los medios necesarios para su inicialización.

Un array no es más que una secuencia de primitivas u objetos, todos del mismo tipo y agrupados bajo un único identificador. Los arrays se definen y se utilizan haciendo uso del *operador indexación (indexing operator)* **[]** . Un array se puede definir, simplemente, añadiendo un par de paréntesis cuadrados detrás del nombre del tipo:

```
int[] a1;
```

También se pueden poner los paréntesis detrás del identificador: el efecto es el mismo:

```
int a1[];
```

Esta forma de la notación satisfacera las expectativas de los programadores de C y C++. La primera en cambio, es quizás más adecuada ya que se puede leer como "el tipo es un array de **int** . En este libro se utilizará por lo tanto este estilo cuando se definan arrays.

El compilador no permite indicar cual será el tamaño del array, lo que nos lleva de vuelta a asunto de las "referencias". Después de la definición del array lo único que se tiene es una referencia a un array, y además, no se reservará espacio alguno para contener el array. Para reserva espacio para el array se necesita

escribir una expresión de inicialización. La inicialización de un array puede aparecer en cualquier lugar dentro del código, pero también se puede usar una tipo especial de expresión de inicialización que tiene que hacerse en el mismo lugar donde se crea el array. Este tipo especial de inicialización consiste en un conjunto de valores encerrados entre llaves. En este caso es el compilador el que se encarga de reservar el espacio necesario para el array:

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

Según esto, ¿Por qué motivo va a querer nadie crear una referencia sin un array?

```
int [] a2;
```

Una razón posible, es que se quiera asignar un array a otro, lo que es una operación permitida en Java. Por ejemplo:

```
a2 = a1;
```

Lo que se está copiando realmente en este caso es una referencia como revela este ejemplo:

```
//: c04:Arrays.java
// Arrays de primitivas.

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]);
   }
} ///:~
```

En este ejemplo, a **a1** se le asigna un valor de inicialización pero no a **a2** , a la

que se le asigna posteriormente otro array.

En el ejemplo anterior aparece también algo nuevo: todos los arrays tiene un miembro intrínseco (tanto si son arrays de objetos como si son arrays de primitivas), que puede examinar - pero no modificar -, que contiene el número de elementos del array. Este miembro es **length** . Como, al igual que en C y C++, en Java los índices de los arrays comienzan a contar desde cero el índice del último elemento del array es **length - 1** . C y C++ aceptarán calladamente que un programa sobrepase los límites del array y acceda de este modo al resto de la zona de memoria reservada para él, lo que es una fuente de continuos problemas. Por contra, Java generará un error en tiempo de ejecución (es decir una *excepción* , tema del capítulo 10) si el programa intenta sobrepasar los límites del array. Naturalmente, el control de todos los accesos a arrays tiene un coste, tanto en tiempo como en código, que no se puede evitar, por lo que el acceso a arrays puede perjudicar el rendimiento del programa, especialmente si estos accesos ocurren en una juncture crítica. Los diseñadores de Java pensaron que, la seguridad en Internet y el aumento de productividad de los programadores, justificaban el sobrecarga que supone el control de los accesos a los arrays.

Pero ¿qué ocurre si se desconoce, cuando se esta escribiendo el programa, cuantos elementos tendrá que contener? Para crear nuevos elementos del array se usa simplemente **new** . En este ejemplo, **new** funciona también creando creando un array de primitivas (**new** no creará ??????????????????):

```
//: c04:ArrayNuevo.java
// Creacion de arrays con new.
import java.util.*;

public class ArrayNuevo {
   static Random rand = new Random();
   static int pRand(int mod) {
          return Math.abs(rand.nextInt()) % mod + 1;
   }
   public static void main(String[] args) {
            int[] a;
            a = new int[pRand(20)];
            System.out.println(
                 "num. elementos de a = " + a.length);
              for(int i = 0; i < a.length; i ++)
                System.out.println(
                  "a[" + i + "] = " + a[i]);
   }
} ///:~
```

Como el tamaño del array es elegido de modo aleatorio (usando el método **pRand ()**), queda claro que la creación del array se está haciendo en tiempo de

ejecución. Además, se puede comprobar examinando la salida del programa anterior, los elementos del array que son de un tipo primitivo son inicializadas automáticamente con sus valores "vacíos"(para números y **char** este valor es cero y para **boolean** es **false**)

El array podría haber sido definido e inicializado en la misma línea:

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

Si se está trabajando con arrays de objetos que no sean primitivas, el necesario usar siempre el operador **new** . Vuelve a aparecer aquí el tema de la referencia ya que lo que realmente se crea es un array de referencias. Así, por ejemplo, con el tipo wrapper **Integer** , que es una clase y no una primitiva:

```
//: c04:ArrayClassObj.java
// Creacion de un array de objetos
// de tipo no primitivo.
import java.util.*;

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

En este ejemplo, incluso después de haber llamado a **new** crear el array:

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

lo único que se tiene es un array de referencias, y la inicialización, por lo tanto, no

está completa hasta que no se inicializa la referencia misma creando un nuevo objeto **Integer** .

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

Si el objeto no es creado, se obtendrá una excepción en tiempo de ejecución al intentar leer la posición vacía del array.

Si se examina como se construye el objeto **String** dentro de las sentencias de impresión, se puede ver que la referencia al objeto **Integer** es transformada en una referencia a un objeto **String** que contiene el valor del objeto **Integer**

También se pueden inicializar arrays usando una lista de elementos encerrada entre llaves. Existen dos formas diferentes:

```
//: c04:ArrayInit.java
// Inicializacion de arrays.

public class ArrayInit {
   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),
      };
   }
} ///:~
```

Esta manera de inicializar arrays puede ser útil en ocasiones, pero tiene la limitación de que el tamaño del array queda fijado durante la compilación. La coma al final de la lista de los inicializadores es opcional; su única función es hacer más sencillo el mantenimiento de listas largas.

La segunda forma de inicialización de arrays proporciona una sintaxis adecuada crear e invocar métodos que pueden producir el mismo efecto que las *listas variables de argumentos* de C (en C reciben el nombre de "varargs"). Estas listas pueden contener un número indeterminado de argumentos que, además pueden ser de cualquier tipo. Ya que todas las clases derivan de la misma clase raíz

Object (el lector podrá aprender más sobre este tema según prosiga en la lectura de este libro), es posible crear un método que tenga como argumento un array de **Object** e invocarlo del siguiente modo:

```
//: c04:VarArgs.java
// Ejemplo de uso de un array para crear
// una lista variable de argumentos.

class A { int i; }

public class VarArgs {
   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 VarArgs(),
         new Float(3.14), new Double(111.11) });

      f(new Object[] {"one", "two", "three" });
      f(new Object[] {new A(), new A(), new A()});
   }
} ///:~
```

De momento no hay mucho más que se pueda hacer con estos objetos de tipo desconocido, y el programa solo utiliza la conversión automática a **String** poder hacer lago útil con cada **Object** . En el capítulo 12, que trata de la identificación de tipos durante la ejecución del programa (*run-time type identification* - RTTI), se verá como se puede averiguar el tipo preciso de cada objeto de modo que se posible hacer con ellos algo más interesante.

Arrays multidimensionales

Java permite crear arrays multidimensionales fácilmente:

```
//: c04:ArrayMultiDim.java
// Creacion de arrays multidimensionales
import java.util.*;

public class ArrayMultiDim {
   static Random rand = new Random();
   static int pRand(int mod) {
      return Math.abs(rand.nextInt()) % mod +1 ;
   }
```

```
static void prt(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++)
         prt("a1[" + i + "][" + j +
             "] = " + a1[i][j]);
   // Array 3-D con dimesiones fijas.
   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++)
            prt("a2[" + i + "][" +
                j + "][" + k +
                "] = " + a2[i][j][k]);
   // Array 3-D con vectores de longitud variable
   int [][][] a3 = new int[pRand(7)][][];
   for (int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for (int j = 0; j < a3[i].length; j++)
         a3[i][j] = new int[pRand(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++)
            prt("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
   // Arrays 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++)
         prt("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++)
            prt("a5[" + i + "][" + j +
                "] = " + a5[i][j]);

   }
} ///:~
```

En el programa anterior se utiliza **length** en los bucles de escritura, lo que muestra que **length** se puede usar también con arrays que no tengan un tamaño fijo.

El primer ejemplo muestra un array multidimensional de primitivas. Cada vector dentro del array es delimitado mediante el uso de llaves:

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

Cada pareja de corchetes define una nueva dimensión dentro del array.

El segundo ejemplo muestra un array tridimensional que se construye usando el operador **new** . En este caso, el espacio para el array es reservado de una sola vez:

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

El tercer ejemplo muestra como cada uno de los vectores que forman el array puede tener su propia longitud:

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

El primer operador **new** crea un array en el que el primer elemento tiene una

longitud aleatoria y deja el resto indeterminado. El segundo operador **new** , dentro del bucle **for** rellena los elementos pero deja el tercer índice sin definir hasta que se alcanza el tercer operador **new** .

Se pude verificar, examinando la salida que produce el programa anterior, que los valores de los arrays son inicializados con valor cero si no se les asigna explícitamente un valor de inicialización.

El cuarto ejemplo muestra que es posible trabajar con objetos no primitivos de una manera similar y como se puede agrupar varios operadores **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, por último, muestra como se puede construir un array de objetos no primitivos elemento a elemento:

```
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);
      }
```

donde **i*j** se utiliza simplemente para poner un valor interesante en la inicialización de **Integer** .

Resumen

Lo .....elaborado mecanismo de inicialización, el constructor, da una idea de la prioridad que se ha puesto en lo relativo a la inicialización en el lenguaje. Cuando Stroustrup estaba diseñando C++, una de las primeras observaciones que hizo acerca de la productividad en C, fue que la inicialización inadecuada de variables está en el origen de muchos de los problemas que se presentan en la programación. Estos errores son, además difíciles de descubrir. Lo mismo se aplica a un purgado inadecuado. Dado que los constructores garantizan la inicialización y el purgado adecuados (el compilador no permite que un objeto sea creado sin que se invoque el constructor adecuado),y de este modo, el programador retiene todo

el control en la creación y destrucción de objetos.

En C++ la destrucción de objetos es muy importante ya que los objetos creados mediante el operador **new** tienen que ser destruidos explícitamente. En Java, el recolector de basura es el encargado de liberar la memoria reservada por los objetos. Por lo tanto, no es necesario, en muchos casos, definir un método de purgado equivalente al destructor de C++, lo que simplifica en gran medida la programación en Java, y añade la tan necesitada seguridad en la gestión de memoria. Algunos recolectores de basura pueden incluso pueden incluso gestionar otros recursos como gráficos y ficheros de .....(handles). Pero la utilización de un recolector de basura conlleva un sobrecarga en la ejecución de los programas. La importancia que esta sobrecarga tenga es difícil de valorar debido a la lentitud de los interpretes de Java en el momento en que este libro esta siendo escrito. Solo cuando esta situación mejore, será posible determinar si el uso de recolector de basura permitirá, o no, el uso de Java en algún tipo de aplicaciones (uno de los asuntos a considerar es el carácter, impredecible a priori, del recolector de basura).

Dado que la construcción de objetos en Java está garantizada, hay muchos mas asuntos relativos a la construcción de objetos de los que se han tratado en este capítulo. En particular, cuando se crea una nueva clase usando *composición (composition)* o *herencia (inheritance)* , la garantía sobre la inicialización se sigue cumpliendo, por lo que es necesario ampliar la sintaxis de modo que estos casos queden cubiertos. La composición, la herencia y su efecto en como un objeto es construido, son tratados en otros capítulos, más adelante, en este libro.

Ejercicios

La soluciones a los ejercicios seleccionados pueden ser encontradas en el documento electrónico *The Thinking in Java Annotated Solution Guide* , disponible en *www.BruceEckel.com* a un precio muy asequible.

1. Crear una clase con un constructor por defecto (es decir, un constructor sin argumentos) que imprima un mensaje. Crear un objeto de esta clase.
2. Añadir a la clase del ejercicio 1 un constructor sobrecargado que tenga un argumento de tipo **String** y que imprima la cadena junto con el mensaje anterior.
3. Crear un array de referencias a objetos de la clase creada en el ejercicio anterior, pero si llegar a crear los objetos. Al ejecutar el programa observar si se imprimen los mensajes de inicialización.
4. Completar el ejercicio 3 creando objetos para asignarlos a las referencias contenidas en el array.
5. Crear un array de objetos **String** y asignar cadenas a cada elemento. Imprimir los resultados usando un bucle **for** .
6. Crear una clase **Perro** con un método **ladrar()** sobrecargado. El método debe ser sobrecargado usando diferentes tipos de primitivas e imprimir diferentes tipos de

ladridos, auyidos, etc., dependiendo de que versión del método es invocada. Escribir un método **main()** que llame a todas las versiones.

7. Modificar el Ejercicio 6 para que dos de los métodos sobrecargados tengan dos argumentos (cada uno de un tipo diferente) pero en distinto orden. Verificar que la sobrecarga de métodos funciona también en este caso.
8. Crear una clase sin constructores, crear un objeto de esta clase dentro del método **main ()** y verificar que el constructor por defecto es sintetizado automáticamente.
9. Crear una clase con dos métodos. Desde el primer método llamar al segundo dos veces: la primera sin usar **this** y la segunda con **this**
10. Crear una clase con dos constructores sobrecargados. Usar **this** para llamar al segundo constructor desde dentro del primero.
11. Crear una clase llamada **Tanque** que puede ser llenada y vaciada, y que además tiene como *"death condition"* que ha de estar vacía cuando el objeto es eliminado. Escribir un método **finalize()** que compruebe que se cumple la "death condition". Comprobar en el método **main()** todas las posibles situaciones que pueden darse.
12. Crear una clase que tenga un **int** y un **char** que no sean inicializados e imprimir su valores comprobar que Java realiza la inicialización por defecto.
13. Crear una clase que tenga un referencia a **String** sin inicializar. Demostrar que la referencia es inicializada con **null** .
14. Crear una clase con un campo **String** que sea inicializado en la definición y otro que se inicializado por el constructor. ¿Cual es la diferencia entre estos dos planteamientos?
15. Crear una clase con dos campos **static Sring** de modo que uno sea inicializado en la definición y el otro dentro de bloque **static** . Añadir un método **static** que imprima ambos campos y demuestre que ambos son inicializados antes de ser usados.
16. Crear una clase con un **String** que es inicializado usando "inicialización de instancia". Describir una utilidad que puede darse a esta construcción (diferente de la apuntada en este libro)
17. Escribir un método que cree e inicializa un array bidimensional de **double** . El tamaño del array debe venir definido los argumentos del método, y los valores de inicialización deben estar dentro de un rango cuyos valores máximo y mínimo sean también parámetros del método. Crear un segundo método que imprima el array generado por el primer método. En **main()** verificar el correcto funcionamiento de los dos métodos creando e imprimiendo arrays de diferentes tamaños.
18. Repetir el ejercicio 19 con un array tridimensional
19. Comentar la línea marcada con (1) en **ExplicitaEstatica** y comprobar que la cláusula de inicialización estática no es invocada. Quitar la marca de comentario de una de las líneas marcadas con (2) y comprobar que *sí* es invocada. Hacer lo mismo con la otra línea marcada con (2) y comprobar que la inicialización estática solo tiene lugar una vez.
20. Experimentar con **Basura.java** ejecutando el programa pasándole los argumentos "gc", "finalizar" o "todo". Repetir el proceso si se observa que se repite algún modelo en la salida. Modificar el código que **System.runFinalization()** sea llamado *antes* que **System.gc()** y observar el resultado.