6: Reutilizando clases

Una de las características más atractivas de Java es la reutilización de código. Pero para ser revolucionario, es necesario poder hacer muchísimo más que copiar código y cambiarlo.

Este es el enfoque que se utiliza en los lenguajes procedurales como C, pero no ha funcionado muy bien. Como todo en Java, la solución está relacionada con la clase. Se reutiliza código creando nuevas clases, pero en vez de crearlas de la nada, se utilizan clases ya existentes que otra persona ya ha construido y depurado.

El truco es usar clases sin manchar el código existente. En este capítulo se verán dos formas de lograrlo. La primera es bastante directa: simplemente se crean objetos de las clase existente dentro de la nueva clase. A esto se le llama *composición*, porque la clase nueva está compuesta de objetos de clases existentes. Simplemente se está reutilizando la funcionalidad del código, no su forma.

El segundo enfoque es más sutil. Crea una nueva clase como *un tipo de* una clase ya existente. Literalmente se toma la forma de la clase existente y se le añade código sin modificar a la clase ya existente. Este acto mágico se denomina *herencia*, y el compilador hace la mayoría del trabajo. La herencia es una de las clases angulares de la programación orientada a objetos y tiene implicaciones adicionales como se verá en el Capítulo 7.

Resulta que mucha de la sintaxis y comportamiento son similares, tanto para la herencia, como para la composición (lo cual tiene sentido porque ambas son formas de construir nuevos tipos a partir de tipos existentes). En este capítulo, se aprenderá sobre estos mecanismos de reutilización de código.

Sintaxis de la composición

Hasta ahora, la composición se usaba con bastante frecuencia. Simplemente se ubican referencias a objetos dentro de nuevas clases. Por ejemplo, suponga que se desea tener un objeto que albergue varios objetos de tipo **cadena de caracteres**, un par de datos primitivos, y un objeto de otra clase. En el caso de los objetos no primitivos, se ponen referencias dentro de la nueva clase, pero se definen los datos primitivos directamente:

```
//: c06:Aspersor.java
// Composición para la reutiliación de código.

class FuenteAgua {
  private String s;
  FuenteAgua() {
    System.out.println("FuenteAgua()");
    s = new String("Construida");
  }
  public String toString() { return s; }
```

```
}

public class Aspersor {
  private String valvula1, valvula2, valvula3, valvula4;
  FuenteAgua fuente;
  int i;
  float f;
  void escribir() {
    System.out.println("valvula1 = " + valvula1);
    System.out.println("valvula2 = " + valvula2);
    System.out.println("valvula3 = " + valvula3);
    System.out.println("valvula4 = " + valvula4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("fuente = " + fuente);
  }
  public static void main(String[] args) {
    Aspersor x = new Aspersor();
    x.escribir();
  }
} ///:~
```

Uno de los métodos definidos en **FuenteAgua( )** es especial: **toString( )**. Se aprenderá más adelante que todo objeto no primitivo tiene un método **toString( )**, y es invocado en situaciones especiales cuando el compilador desea obtener un objeto como cadena de caracteres. Por tanto, en la expresión:

```
System.out.println("fuente = " + fuente);
```

el compilador ve que se está intentando añadir un objeto **String** (“**fuente** =”) a un objeto **FuenteAgua.** Esto no tiene sentido porque sólo se puede “añadir” un **String** a otro **String**, por lo que dice: “¡Convertiré **fuente** en un **String** invocando a **toString( )**! Después de hacer esto se pueden combinar los dos objetos de tipo **String** y pasar el **String** resultante a **Sytem.out.println( )**. Siempre que se desee este comportamiento con una clase creada, sólo habrá que escribir un método **toString( )**.

A primera vista, uno podría asumir —siendo Java tan seguro y cuidadoso como es— que el compilador podría construir automáticamente objetos para cada una de las referencias en el código de arriba; por ejemplo, invocando al constructor por defecto para **FuenteAgua** para inicializar **fuente**. La salida de la sentencia de impresión es de hecho:

```
valvula1 = null
valvula2 = null
valvula3 = null
valvula4 = null
i = 0
f = 0.0
fuente = null
```

Los datos primitivos son campos de una clase que se inicializan automáticamente a cero, como se indicó en el Capítulo 2. Pero las referencias a objetos se inicializan a **null**, y si se intenta invocar a métodos de cualquiera de ellos, se sigue obteniendo una excepción. De hecho, es bastante bueno (y útil) poder seguir imprimiéndolos sin lanzar excepciones.

Tiene sentido que el compilador no sólo cree un objeto por defecto para cada referencia, porque eso conllevaría una sobrecarga innecesaria en la mayoría de los casos. Si se quieren referencias inicializadas, se puede hacer:

1. En el punto en que se definen los objetos. Esto significa que siempre serán inicializados antes de invocar al constructor.
2. En el constructor de esa clase.
3. Justo antes de que, de hecho, se necesite el objeto. A esto se le llama *inicialización perezosa*. Puede reducir la sobrecarga en situaciones en las que no es necesario crear siempre el objeto.

A continuación, se muestran los tres enfoques:

```
//: c06:Banio.java
// Inicialización de constructores con composición.

class Jabon {
  private String s;
  Jabon() {
    System.out.println("Jabon()");
    s = new String("Construido");
  }
  public String toString() { return s; }
}

public class Banio {
  private String
    // Inicializando en el momento de la definición:
    s1 = new String("Contento"),
    s2 = "Contento",
    s3, s4;
  Jabon pastilla;
  int i;
  float juguete;
  Banio() {
    System.out.println("Dentro del banio()");
    s3 = new String("Gozo");
    i = 47;
    juguete = 3.14f;
    pastilla = new Jabon();
  }
```

```
  void escribir() {
    // Inicialización tardía:
    if(s4 == null)
      s4 = new String("Temporal");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("juguete = " + juguete);
    System.out.println("pastilla = " + pastilla);
  }
  public static void main(String[] args) {
    Banio b = new Banio();
    b.escribir();
  }
} ///:~
```

Fíjese que en el constructor **Banio** se ejecuta una sentencia antes de que tenga lugar ninguna inicialización. Cuando no se inicializa en el momento de la definición, sigue sin haber garantías de que se lleve a cabo ningún tipo de inicialización antes de que se envíe un mensaje a una referencia a un objeto —excepto la inevitable excepción en tiempo de ejecución.

He aquí la salida del programa:

```
Dentro del Banio()
Jabon()
S1 = Contento
S2 = Contento
S3 = Gozo
S4 = Gozo
I = 47
Juguete = 3.14
pastilla = Construido
```

Cuando se invoca al método **escribir( )** éste rellena **s4** para que todos los campos estén inicializados correctamente cuando se usen.

Sintaxis de la herencia

La herencia es una parte integral de Java (y de todos los lenguajes de POO en general). Resulta que siempre se está haciendo herencia cuando se crea una clase, pero a menos que se herede explícitamente de otra clase, se hereda implícitamente de la clase raíz estándar de Java **Object**.

La sintaxis para la composición es obvia, pero para llevar a cabo herencia se realiza de distinta forma. Cuando se hereda, se dice: "Esta clase nueva es como esa clase vieja". Se dice esto en el códi-

go dando el nombre de la clase, como siempre, pero antes de abrir el paréntesis del cuerpo de la clase, se pone la palabra clave **extends** seguida del nombre de la *clase base*. Cuando se hace esto, automáticamente se tienen todos los datos miembro y métodos de la clase base. He aquí un ejemplo:

```
//: c06:Detergente.java
// Sintaxis y propiedades de la herencia.

class ProductoLimpieza {
  private String s = new String("Producto de Limpieza");
  public void aniadir(String a) { s += a; }
  public void diluir() { aniadir(" diluir()"); }
  public void aplicar() { aniadir(" aplicar()"); }
  public void frotar() { aniadir(" fregar()"); }
  public void escribir() { System.out.println(s); }
  public static void main(String[] args) {
    ProductoLimpieza x = new ProductoLimpieza();
    x.diluir(); x.aplicar(); x.frotar();
    x.escribir();
  }
}

public class Detergente extends ProductoLimpieza {
  // Cambiar un método:
  public void frotar() {
    aniadir(" Detergente.frotar()");
    super.frotar(); // Llamar a la versión de la clase base
  }
  // Añadir métodos al interfaz:
  public void aclarar() { aniadir(" aclarar()"); }
  // Probar la nueva clase:
  public static void main(String[] args) {
    Detergente x = new Detergente();
    x.diluir();
    x.aplicar();
    x.frotar();
    x.aclarar();
    x.escribir();
    System.out.println("Probando la clase base:");
    ProductoLimpicza.main(args);
  }
} ///:~
```

Esto demuestra un gran número de aspectos. En primer lugar, en el método **aniadir( ) de la clase ProductoLimpieza**, se concatenan **Cadenas de caracteres** a **s** utilizando el operador +=, que

es uno de los operadores (junto con '+') que los diseñadores de Java "sobrecargaron" para que funcionara con **Cadenas de caracteres**.

Segundo, tanto **ProductoLimpieza** como **Detergente** contienen un método **main( )**. Se puede crear un método **main( )** por cada clase que uno cree, y se recomienda codificar de esta forma, de manera que todo el código de prueba esté dentro de la clase. Incluso si se tienen muchas clases en un programa, sólo se invocará al método **main( )** de la clase invocada en la línea de comandos. (Dado que **main( )** es **público**, no importa si la clase a la que pertenece es o no **pública**.) Por tanto, en este caso, cuando se escriba **java Detergente**, se invocará a **Detergente.main( )**. Pero también se puede hacer que **ProductoLimpieza** invoque a **ProductoLimpieza.main( )**, incluso aunque **ProductoLimpieza** no sea una clase **pública**. Esta técnica de poner un método **main( )** en cada clase permite llevar a cabo pruebas para cada clase de manera sencilla. Y no es necesario eliminar el método **main( )** cuando se han acabado las pruebas; se puede dejar ahí por si hubiera que usarlas para otras pruebas más adelante.

Aquí, se puede ver que **Detergente.main( )** llama a **ProductoLimpieza.main( )** explícitamente, pasándole los mismos argumentos de la línea de comandos (sin embargo, se podría pasar cualquier array de **Cadenas de caracteres**).

Es importante que todos los métodos de **ProductoLimpieza** sean **públicos**. Recuerde que si se deja sin poner cualquier modificador de miembro, el miembro será por defecto "amistoso", lo cual permite acceder sólo a los miembros del paquete. Por consiguiente, *dentro de este paquete*, cualquiera podría usar esos métodos si no hubiera modificador de acceso. **Detergente** no tendría problemas, por ejemplo. Sin embargo, si se fuera a heredar desde **ProductoLimpieza** una clase de cualquier otro paquete, ésta sólo podría acceder a las clases **públicas**. Por tanto, al planificar la herencia, como regla general, deben hacerse todos los campos **privados** y todos los miembros **públicos**. (Los miembros **protegidos** también permiten accesos por parte de clases derivadas; esto se aprenderá más adelante.) Por supuesto, en los casos particulares hay que hacer ajustes, pero ésta es una regla útil.

Fíjese que **ProductoLimpieza** tiene un conjunto de métodos en su interfaz: **aniadir( )**, **diluir( )**, **aplicar( )**, **frotar( )** y **escribir( )**. Dado que **Detergente** se *hereda de* **ProductoLimpieza** (mediante la palabra clave **extends**) automáticamente se hace con estos métodos en su interfaz, incluso aunque no se encuentren explícitamente definidos en **Detergente**. Se puede pensar que la herencia, por tanto, es una *reutilización del interfaz*. (La implementación también se hereda, pero esto no es lo importante.)

Como se ha visto en **frotar( )**, es posible tomar un método que se haya definido en la clase base y modificarlo. En este caso, se podría desear llamar al método desde la clase base dentro de la nueva versión. Pero dentro de **frotar( )** no se puede simplemente invocar a **frotar( )**, dado que eso produciría una llamada recursiva, que no es lo que se desea. Para solucionar este problema Java tiene la palabra clave **super** que hace referencia a la "superclase" de la cual ha heredado la clase actual. Por consiguiente, la expresión **super.frotar( )** llama a la versión que tiene la clase base del método **frotar( )**.

Al heredar, uno no se limita a usar los métodos de la clase base. También se pueden añadir nuevos métodos a la clase derivada, exactamente de la misma manera que se introduce un método en una clase: simplemente se definen. El método **aclarar( )** es un ejemplo de esta afirmación.

En **Detergente.main( )** se puede ver que, para un objeto **Detergente,** se puede invocar a todos los métodos disponibles, también en **ProductoLimpieza** y en **Detergente** (por ejemplo, **aclarar( )**).

Inicializando la clase base

Dado que ahora hay dos clases involucradas —la clase base y la clase derivada— en vez de simplemente una, puede ser un poco confuso intentar imaginar el objeto resultante producido por una clase derivada. Desde fuera, parece que la nueva clase tiene la misma interfaz que la clase base, y quizás algunos métodos y campos adicionales. Pero la herencia no es una simple copia de la interfaz de la clase base. Cuando se crea un objeto de la clase derivada, éste contiene dentro de él un *sub objeto* de la clase base. Este subobjeto es el mismo que si se hubiera creado un objeto de la clase base en sí. Es simplemente que, desde fuera, el subobjeto de la clase base está envuelto dentro del objeto de la clase derivada.

Por supuesto, es esencial que el subobjeto de la clase base se inicialice correctamente y sólo hay una forma de garantizarlo: llevar a cabo la inicialización en el constructor, invocando al constructor de la clase base, que tiene todo el conocimiento y privilegios apropiados para llevar a cabo la inicialización de la clase base. Java inserta automáticamente llamadas al constructor de la clase base en el constructor de la clase derivada. El ejemplo siguiente muestra este funcionamiento con tres niveles de herencia:

```
//: c06:Animacion.java
// Llamadas al constructor durante la herencia.

class Arte {
  Arte() {
    System.out.println("Constructor de arte");
  }
}

class Dibujo extends Arte {
  Dibujo() {
    System.out.println("Constructor de dibujo");
  }
}

public class Animacion extends Dibujo {
  Animacion() {
    System.out.println("Constructor de animacion");
  }
  public static void main(String[] args) {
    Animacion x = new Animacion();
  }
} ///:~
```

La salida de este programa muestra las llamadas automáticas:

```
Constructor de arte
Constructor de dibujo
Constructor de animacion
```

Se puede ver que la construcción se da desde la base "hacia fuera", de forma que se inicializa la clase base antes de que los constructores de la clase derivada puedan acceder a ella.

Incluso si no se crea un constructor para **Animacion( )**, el compilador creará un constructor por defecto que invoque al constructor de la clase base.

Constructores con parámetros

El ejemplo de arriba tiene constructores por defecto; es decir, no tienen ningún parámetro. Para el compilador es fácil invocarlos porque no hay ningún problema que resolver respecto al paso de parámetros. Si una clase no tiene parámetros por defecto, o si se desea invocar a un constructor de una clase base que tiene parámetros, hay que escribir explícitamente la llamada al constructor de la clase base usando la palabra clave **super** y la lista de parámetros apropiada:

```
//: c06:Ajedrez.java
// Herencia, constructores y parámetros.

class Juego {
  Juego(int i) {
    System.out.println("Constructor de juego");
  }
}

class JuegoMesa extends Juego {
  JuegoMesa(int i) {
    super(i);
    System.out.println("Constructor de JuegoMesa");
  }
}

public class Ajedrez extends JuegoMesa {
  Ajedrez() {
    super(11);
    System.out.println("Constructor de Ajedrez");
  }
  public static void main(String[] args) {
    Ajedrez x = new Ajedrez();
  }
} ///:~
```

Si no se invoca al constructor de la clase base de **JuegoMesa( )**, el compilador se quejará al no poder encontrar un constructor de la forma **Juego( )**. Además, la llamada al constructor de la clase base *debe* ser lo primero que se haga en el constructor de la clase derivada. (El compilador así lo recordará cuando no se haga correctamente.)

Capturando excepciones del constructor base

Como se acaba de indicar, el compilador obliga a ubicar la llamada al constructor de la clase base, primero dentro del cuerpo del constructor de la clase derivada. Esto simplemente quiere decir que no puede aparecer nada antes de esta llamada. Como se verá en el Capítulo 10, esto también evita que un constructor de una clase derivada capture excepciones que provengan de una clase base. Esto puede suponer un inconveniente en algunas ocasiones.

Combinando la composición y la herencia

Es muy frecuente usar la composición y la herencia juntas. El ejemplo siguiente muestra la creación de una clase más compleja, utilizando tanto la herencia como la composición, junto con la inicialización necesaria del constructor:

```
//: c06:PonerMesa.java
// Combinando la composición y la herencia.

class Plato {
  Plato(int i) {
    System.out.println("Constructor de plato");
  }
}

class PlatoCena extends Plato {
  PlatoCena(int i) {
    super(i);
    System.out.println(
      "Constructor de PlatoCena");
  }
}

class Utensilio {
  Utensilio(int i) {
    System.out.println("Constructor de utensilio");
  }
}

class Cuchara extends Utensilio {
```

```
  Cuchara(int i) {
    super(i);
    System.out.println("Constructor de cuchara");
  }
}

class Tenedor extends Utensilio {
  Tenedor(int i) {
    super(i);
    System.out.println("Constructor de tenedor");
  }
}

class Cuchillo extends Utensilio {
  Cuchilllo(int i) {
    super(i);
    System.out.println("Constructor de cuchillo");
  }
}

// Una manera costrumbrista de hacer algo:
class Costumbre {
  Costumbre(int i) {
    System.out.println("Constructor de costumbre");
  }
}

public class PonerMesa extends Costumbre {
  Cuchara cc;
  Tenedor tnd;
  Cuchillo cch;
  PlatoCena pc;
  PonerMesa(int i) {
    super(i + 1);
    cc = new Cuchara(i + 2);
    tnd = new Tenedor(i + 3);
    cch = new Cuchillo(i + 4);
    pc = new PlatoCena(i + 5);
    System.out.println(
      "Constructor de PonerMesa");
  }
  public static void main(String[] args) {
    PonerMesa x = new PonerMesa(9);
  }
} ///:~
```

Cuando el compilador obliga a inicializar la clase base, y requiere que se haga justo al principio del constructor, no se asegura de que inicialicemos los objetos miembro, por lo que es conveniente prestar especial atención a esto.

Garantizar una buena limpieza

Java no tiene el concepto de método *destructor* de C++. Este método se invoca automáticamente al destruir un objeto. La razón de su ausencia es probablemente que en Java lo habitual es simplemente olvidarse de esos objetos, más que destruirlos, permitiendo que el recolector de basura reclame esta memoria cuando sea necesario.

En muchas ocasiones, esto es bueno, pero hay veces en las que una clase tiene que hacer algunas actividades durante su vida que requieren de limpieza. Como se mencionó en el Capítulo 4, no se puede saber cuándo se invocará al recolector de basura, o incluso, si éste será invocado. Por tanto, si se desea que se limpie algún espacio para una clase, hay que escribir explícitamente un método especial que lo haga, y asegurarse de que el programador cliente sepa que hay que invocar a este método. Por encima de esto —como se describe en el Capítulo 10 ("Manejo de Errores con Excepciones")— hay que protegerse de las excepciones poniendo este tipo de limpieza en una cláusula **finally**.

Considere un ejemplo de un sistema de diseño asistido por computador que dibuja en la pantalla:

```
//: c06:SistemaDAC.java
// Asegurando una limpieza adecuada.
import java.util.*;

class Forma {
  Forma(int i) {
    System.out.println("Constructor de forma");
  }
  void limpiar() {
    System.out.println("Limpieza de forma");
  }
}

class Circulo extends Forma {
  Circulo(int i) {
    super(i);
    System.out.println("Dibujando un circulo");
  }
  void limpiar() {
    System.out.println("Borrando un circulo");
    super.limpiar();
  }
}
```

```
class Triangulo extends Forma {
  Triangulo(int i) {
    super(i);
    System.out.println("Dibujando un triangulo");
  }
  void limpiar() {
    System.out.println("Borrando un triangulo");
    super.limpiar();
  }
}

class Linea extends Forma {
  private int inicio, fin;
  Linea(int inicio, int fin) {
    super(inicio);
    this.inicio = inicio;
    this.fin = fin;
    System.out.println("Dibujando una linea: " +
            inicio + ", " + fin);
  }
  void limpiar() {
    System.out.println("Borrando una linea: " +
            inicio + ", " + fin);
    super.limpiar();
  }
}

public class SistemaDAC extends Forma {
  private Circulo c;
  private Triangulo t;
  private Linea[] lineas = new Linea[10];
  SistemaDAC(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      lineas[j] = new Linea(j, j*j);
    c = new Circulo(1);
    t = new Triangulo(1);
    System.out.println("Constructor combinado");
  }
  void limpiar() {
    System.out.println("SistemaDAC.limpiar()");
    // El orden de eliminación es inverso al
    // orden de inicialización
    t.limpiar();
    c.limpiar();
```

```
    for(int i = lineas.length - 1; i >= 0; i--)
      lineas[i].limpiar();
    super.limpiar();
  }
  public static void main(String[] args) {
    SistemaDAC x = new SistemaDAC(47);
    try {
      // Código y manejo de excepciones...
    } finally {
      x.limpiar();
    }
  }
} ///:~
```

Todo en este sistema es algún tipo de **Forma** (que en sí es un tipo de **Objeto** dado que está implícitamente heredada de la clase raíz). Cada clase redefine el método **limpiar( )** de **Forma** además de invocar a la versión de ese método de la clase base haciendo uso de **super**. Las clases **Forma** específicas —**Círculo**, **Triángulo** y **Línea**— tienen todas constructores que "dibujan", aunque cualquier método invocado durante la vida del objeto podría ser el responsable de hacer algo que requiera de limpieza. Cada clase tiene su propio método **limpiar( )** para restaurar cosas a la forma en que estaban antes de que existiera el objeto.

En el método **main( )** se pueden ver dos palabras clave nuevas, y que no se presentarán oficialmente hasta el capítulo 10: **try** y **finally**. La palabra clave **try** indica que el bloque que sigue (delimitado por llaves) es una *región vigilada*, lo que quiere decir que se le da un tratamiento especial. Uno de estos tratamientos especiales consiste en que el código de la cláusula **finally** que sigue a esta región vigilada se ejecuta *siempre*, sin que importe cómo se salga del bloque **try**. (Con el manejo de excepciones, es posible dejar un bloque **try** de distintas formas no ordinarias.) Aquí, la cláusula **finally** dice: "Llama siempre a **limpiar( )** para **x**, sin que importe lo que ocurra". Estas palabras claves se explicarán con detalle en el Capítulo 10.

Fíjese que en el método de limpieza hay que prestar atención también al orden de llamada de los métodos de limpieza de la clase base y los objetos miembros, en caso de que un subobjeto dependa de otro. En general, se debería seguir la forma ya impuesta por el compilador de C++ para sus destructores: en primer lugar se lleva a cabo todo el trabajo de limpieza específico a nuestra clase, en orden inverso de creación. (En general, esto requiere que los elementos de la clase base sigan siendo accesibles.) Después, se llama al método de limpieza de la clase base, como se ha demostrado aquí.

Puede haber muchos casos en los que el aspecto de la limpieza no sea un problema; simplemente se deja actuar al recolector de basura. Pero cuando es necesario hacerlo explícitamente se necesita tanto diligencia como atención.

Orden de recolección de basura

No hay mucho en lo que se pueda confiar en lo referente a la recolección de basura. Puede que ni siquiera se invoque nunca al recolector de basura. Cuando se le invoca, puede reclamar objetos en el orden que quiera. Es mejor no confiar en la recolección de basura para nada que no sea reclamar

memoria. Si se desea que se dé una limpieza, es mejor que cada uno construya sus propios métodos de limpieza, y no confiar en el método **finalize( )**. (Como se mencionó en el Capítulo 4, puede obligarse a Java a invocar a todos los métodos **finalize( )**.)

Ocultación de nombres

Sólo los programadores de C++ podrían sorprenderse de la ocultación de nombres, puesto que funciona distinto en ese lenguaje. Si una clase base de Java tiene un nombre de método sobrecargado varias veces, la redefinición de ese nombre de método en la clase derivada *no* esconderá ninguna de las versiones de la clase base. Por consiguiente, la sobrecarga funciona independientemente de si el método se definió en el nivel actual o en una clase base:

```
//: c06:Ocultar.java
// Sobrecargando un nombre de método de una clase base
// en una clase derivada que no oculta
// las versiones de la clase base.

class Homer {
  char realizar(char c) {
    System.out.println("realizar(char)");
    return 'd';
  }
  float realizar(float f) {
    System.out.println("realizar(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void realizar(Milhouse m) {}
}

class Ocultar {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.realizar(1); // realizar(float) usado
    b.realizar('x');
    b.realizar(1.0f);
    b.realizar(new Milhouse());
  }
} ///:~
```

Como se verá en el siguiente capítulo, es bastante más común reescribir métodos del mismo nombre utilizando exactamente el mismo nombre, parámetros y tipo de retorno que en la clase base. De otra manera pudiera ser confuso (que es la razón por la que C++ no permite esto, para evitar que se haga lo que probablemente es un error).

Elección entre composición y herencia

Tanto la composición como la herencia, permiten ubicar subobjetos dentro de una nueva clase. Habría que preguntarse por la diferencia entre ambas, y cuándo elegir una en vez de la otra.

La composición suele usarse cuando se quieren mantener las características de una clase ya existente dentro de la nueva, pero no su interfaz. Es decir, se empotra un objeto de forma que se puede usar para implementar su funcionalidad en la nueva clase, pero el usuario de la nueva la clase ve la interfaz que se ha definido para la nueva clase en vez de la interfaz del objeto empotrado. Para lograr este efecto, se empotran objetos **privados** de clases existentes dentro de la nueva clase.

En ocasiones, tiene sentido permitir al usuario de la clase acceder directamente a la composición de la nueva clase; es decir, hacer a los objetos miembro **públicos**. Los objetos miembro usan por sí mismos la ocultación de información, de forma que esto es seguro. Cuando el usuario sabe que se está ensamblando un conjunto de partes, construye una interfaz más fácil de entender. Un objeto **coche** es un buen ejemplo:

```
//: c06:Coche.java
// Composición con objetos públicos.

class Motor {
  public void arrancar() {}
  public void acelerar() {}
  public void parar() {}
}

class Rueda {
  public void inflar(int psi) {}
}

class Ventana {
  public void subir() {}
  public void bajar() {}
}

class Puerta {
  public Ventana ventana = new Ventana();
  public void abrir() {}
  public void cerrar() {}
}
```

```
public class Coche {
  public Motor motor = new Motor();
  public Rueda[] rueda = new Rueda[4];
  public Puerta izquierda = new Puerta(),
        derecha = new Puerta(); // 2-puerta
  public Coche() {
    for(int i = 0; i < 4; i++)
      rueda[i] = new Rueda();
  }
  public static void main(String[] args) {
    Coche coche = new Coche();
    coche.izquierda.ventana.subir();
    coche.rueda[0].inflar(72);
  }
} ///:~
```

Dado que la composición de un coche es parte del análisis del problema (y no simplemente parte del diseño subyacente), hacer sus miembros **públicos** ayuda al entendimiento por parte del programador cliente de cómo usar la clase, y requiere menos complejidad de código para el creador de la clase. Sin embargo, hay que ser consciente de que éste es un caso especial y en general los campos se harán **privados**.

Cuando se hereda, se toma una clase existente y se hace una versión especial de la misma. En general, esto significa que se está tomando una clase de propósito general y especializándola para una necesidad especial. Simplemente pensando un poco se verá que no tendría sentido componer un coche utilizando un objeto vehículo —un coche no contiene un vehículo, *es* un vehículo. La relación *es-un* se expresa con herencia, y la relación *tiene-un* se expresa con composición.

Protegido (protected)

Ahora que se ha presentado el concepto de herencia, tiene sentido finalmente la palabra clave **protected**. En un mundo ideal, los miembros **privados** siempre serían irrevocablemente **privados**, pero en los proyectos reales hay ocasiones en las que se desea hacer que algo quede oculto del mundo en general, y sin embargo, permitir acceso a miembros de clases derivadas. La palabra clave **protected** es un nodo de pragmatismo. Dice: "Esto es **privado** en lo que se refiere al usuario de la clase, pero está disponible para cualquiera que herede de esta clase o a cualquier otro de este **paquete**. Es decir, **protegido** es automáticamente "amistoso" en Java.

La mejor conducta a seguir es dejar los miembros de datos **privados** —uno siempre debería preservar su derecho a cambiar la implementación subyacente. Posteriormente se puede permitir acceso controlado a los descendientes de la clase a través de los métodos **protegidos**:

```
//: c06:Malvado.java
// La palabra clave protected.
import java.util.*;
```

```
class Villano {
  private int i;
  protected int leer() { return i; }
  protected void poner(int ii) { i = ii; }
  public Villano(int ii) { i = ii; }
  public int valor(int m) { return m*i; }
}

public class Malvado extends Villano {
  private int j;
  public Malvado(int jj) { super(jj); j = jj; }
  public void cambiar(int x) { poner(x); }
} ///:~
```

Se puede ver que **cambiar( )** tiene acceso a **poner( )** porque es **protegido**.

Desarrollo incremental

Una de las ventajas de la herencia es que soporta el *desarrollo incremental* permitiendo introducir nuevo código sin introducir *errores* en el código ya existente. Esto también aísla nuevos fallos dentro del nuevo código. Pero al heredar de una clase funcional ya existente y al añadirle nuevos atributos y métodos (y redefiniendo métodos ya existentes), se deja el código existente —que alguien más podría estar utilizando— intacto y libre de errores. Si se da un fallo, se sabe que éste se encuentra en el nuevo código, que es mucho más corto y sencillo de leer que si hubiera que modificar el cuerpo del código existente.

Es bastante sorprendente la independencia de las clases. Ni siquiera se necesita el código fuente de los métodos para reutilizar el código. Como máximo, simplemente habría que importar el paquete. (Esto es cierto, tanto en el caso de la herencia, como en el de la composición.)

Es importante darse cuenta de que el desarrollo de un programa es un proceso incremental, al igual que el aprendizaje humano. Se puede hacer tanto análisis como se quiera, pero se siguen sin conocer todas las respuestas cuando comienza un proyecto. Se tendrá mucho más éxito —y una realimentación mucho más inmediata— si empieza a "crecer" el proyecto como una criatura evolucionaria, orgánica, en vez de construirlo de un tirón como si fuera un rascacielos de cristal.

Aunque la herencia puede ser una técnica útil de cara a la experimentación, en algún momento, una vez que las cosas se estabilizan es necesario echar un nuevo vistazo a la jerarquía de clases definida intentando encajarla en una estructura con sentido. Recuérdese que bajo todo ello, la herencia simplemente pretende expresar una relación que dice: "Esta nueva clase es un *tipo* de esa otra clase". Al programa no deberían importarle los bits, sino el crear y manipular objetos de varios tipos para expresar un modelo en términos que provengan del espacio del problema.

Conversión hacia arriba

El aspecto más importante de la herencia no es que proporcione métodos para la nueva clase. Es la relación expresada entre la nueva clase y la clase base. Esta relación puede resumirse diciendo: "La nueva clase es *un tipo de* la clase existente".

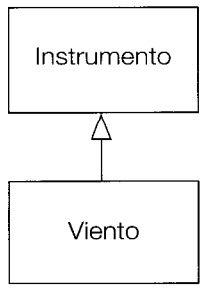
Esta descripción no es simplemente una forma elegante de explicar la herencia —está soportada directamente por el lenguaje. Como ejemplo, considérese una clase base denominada **Instrumento** que representa los instrumentos musicales, y una clase derivada denominada **Viento**. Dado que la herencia significa que todos los métodos de la clase base también están disponibles para la clase derivada, cualquier mensaje que se pueda enviar a la clase base podrá ser también enviado a la clase derivada. Si la clase **Instrumento** tiene un método **tocar( )**, también lo tendrán los instrumentos **Viento**. Esto significa que se puede decir con precisión que un objeto **Viento** es también un tipo de **Instrumento**. El ejemplo siguiente muestra cómo soporta este concepto el compilador:

```
//: c06:Viento.java
// Herencia y conversión hacia arriba.
import java.util.*;

class Instrumento {
  public void tocar() {}
  static void afinar(Instrumento i) {
    // ...
    i.tocar();
  }
}

// Los objetos de viento son instrumentos
// porque tienen la misma interfaz:
class Viento extends Instrumento {
  public static void main(String[] args) {
    Viento flauta = new Viento();
    Instrumento.afinar(flauta); // Conversión hacia arriba
  }
} ///:~
```

Lo interesante de este ejemplo es el método **afinar( )**, que acepta una referencia a **Instrumento**. Sin embargo, en **Viento.main( )**, se llama al método **afinar( )** proporcionándole una referencia a **Viento**. Dado que Java tiene sus particularidades en la comprobación de tipos, parece extraño que un método que acepte un tipo llegue a aceptar otro tipo, hasta que uno se da cuenta de que un objeto **Viento** es también un objeto **Instrumento**, y no hay método al que pueda invocar **afinar( )** para un **Instrumento** que no esté en **Viento**. Dentro de **afinar( )**, el código funciona para **Instrumento** y cualquier cosa que se derive de **Instrumento**, y al acto de convertir una referencia a **Viento** en una referencia a **Instrumento** se le denomina *hacer conversión hacia arriba*.

¿Por qué "conversión hacia arriba"?

La razón para el término es histórica, y se basa en la manera en que se han venido dibujando tradicionalmente los diagramas de herencia: con la raíz en la parte superior de la página, y creciendo hacia abajo. (Por supuesto, se puede dibujar un diagrama de cualquier manera que uno considere útil.) El diagrama de herencia para **Viento.java** es, por consiguiente:



La conversión de clase derivada a base se mueve hacia *arriba* dentro del diagrama de herencia, por lo que se denomina *conversión hacia arriba*. Esta operación siempre es segura porque se va de un tipo más específico a uno más general. Es decir, la clase derivada es un superconjunto de la clase base. Podría contener más métodos que la clase base, pero debe contener *al menos* los métodos de ésta última. Lo único que puede pasar a la interfaz de clases durante la conversión hacia arriba es que pierda métodos en vez de ganarlos. Ésta es la razón por la que el compilador permite la conversión hacia arriba sin ningún tipo de conversión especial u otras notaciones especiales.

También se puede llevar a cabo lo contrario a la conversión hacia arriba, denominado *conversión hacia abajo*, pero implica el dilema en el que se centra el Capítulo 12.

De nuevo composición frente a herencia

En la programación orientada a objetos, la forma más probable de crear código es simplemente empaquetando juntos datos y métodos en una clase, y usando los objetos de esa clase. También se utilizarán clases existentes para construir nuevas clases con composición. Menos frecuentemente, se usará la herencia. Por tanto, aunque la herencia captura gran parte del énfasis durante el aprendizaje de POO, esto no implica que se deba hacer en todas partes en las que se pueda. Por el contrario, se debe usar de una manera limitada, sólo cuando está claro que es útil. Una de las formas más claras de determinar si se debería usar composición o herencia es preguntar si alguna vez habrá que hacer una conversión hacia arriba desde la nueva clase a la clase base. Si se debe hacer una conversión hacia arriba, entonces la herencia es necesaria, pero si no se necesita, se debería mirar más con detalle si es o no necesaria. El siguiente capítulo (polimorfismo) proporciona una de las razones de más peso para una conversión hacia arriba, pero si uno recuerda preguntar: "¿Necesito una conversión hacia arriba?" obtendrá una buena herramienta para decidir entre la composición y la herencia.

La palabra clave **final**

La palabra clave **final** de Java tiene significados ligeramente diferentes dependiendo del contexto, pero en general dice: "Esto no puede cambiarse". Se podría querer evitar cambios por dos razones: diseño o eficiencia. Dado que estas dos razones son bastante diferentes, es posible utilizar erróneamente la palabra clave **final**.

Las secciones siguientes discuten las tres posibles ubicaciones en las que se puede usar **final**: para datos, métodos y clases.

Para datos

Muchos lenguajes de programación tienen una forma de indicar al compilador que cierta parte de código es "constante". Una constante es útil por varias razones:

1. Puede ser una *constante en tiempo de compilación* que nunca cambiará.
2. Puede ser un valor inicializado en tiempo de ejecución que no se desea que se llegue a cambiar.

En el caso de una constante de tiempo de compilación, el compilador puede "manejar" el valor constante en cualquier cálculo en el que se use; es decir, se puede llevar a cabo el cálculo en tiempo de compilación, eliminando parte de la sobrecarga de tiempo de ejecución. En Java, estos tipos de constantes tienen que ser datos primitivos y se expresan usando la palabra clave **final**. A este tipo de constantes se les debe dar un valor en tiempo de definición.

Un campo que es **estático** y **final** sólo tiene un espacio de almacenamiento que no se puede modificar.

Al usar **final** con referencias a objetos en vez de con datos primitivos, su significado se vuelve algo confuso. Con un dato primitivo, **final** convierte el *valor* en constante, pero con una referencia a un objeto, **final** hace de la *referencia* una constante. Una vez que la referencia se inicializa a un objeto, ésta nunca se puede cambiar para que apunte a otro objeto. Sin embargo, se puede modificar el objeto en sí; Java no proporciona ninguna manera de convertir un objeto arbitrario en una constante. (Sin embargo, se puede escribir la clase, de forma que sus objetos tengan el efecto de ser constantes.) Esta restricción incluye a los arrays, que también son objetos.

He aquí un ejemplo que muestra el funcionamiento de los campos **final**:

```
//: c06:DatosConstantes.java
// El efecto de final en campos.

class Valor {
  int i = 1;
}

public class DatosConstantes {
```

```
  // Pueden ser constantes de tiempo de compilación
  final int i1 = 9;
  static final int VAL_DOS = 99;
  // Típica constante pública:
  public static final int VAL_TRES = 39;
  // No pueden ser constantes en tiempo de compilación:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Valor v1 = new Valor();
  final Valor v2 = new Valor();
  static final Valor v3 = new Valor();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void escribir(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    DatosConstantes fd1 = new DatosConstantes();
    //! fd1.i1++; // Error: no se puede cambiar el valor
    fd1.v2.i++; // ¡El objeto no es constante!
    fd1.v1 = new Valor(); // OK -- no es final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // ¡El objeto no es una constante!
    //! fd1.v2 = new Valor(); // Error: No se puede
    //! fd1.v3 = new Valor(); // cambiar ahora la referencia
    //! fd1.a = new int[3];

    fd1.escribir("fd1");
    System.out.println("Creando un nuevo DatosConstantes");
    DatosConstantes fd2 = new DatosConstantes();
    fd1.escribir("fd1");
    fd2.escribir("fd2");
  }
} ///:~
```

Dado que **i1** y **VAL_DOS** son datos primitivos **final** con valores de tiempo de compilación, ambos pueden usarse como constantes de tiempo de compilación y su uso no difiere mucho. **VAL_TRES** es la manera más usual en que se verán definidas estas constantes: **pública** de forma que puedan ser utilizadas fuera del paquete, **estática** para hacer énfasis en que sólo hay una, y **final** para indicar que es una constante. Fíjese que los datos primitivo **static final** con valores iniciales constantes (es decir, las constantes de tiempo de compilación) se escriben con mayúsculas por acuerdo,

además de con palabras separadas por guiones bajos (es decir, justo como las constantes de C, que es de donde viene el acuerdo). La diferencia se muestra en la salida de una ejecución:

```
fd1:  i4 = 15; i5 = 9
Creando un nuevo DatosConstante
fd1:  i4 = 15; i5 = 9
fd2:  i4 = 10; i5 = 9
```

Fíjese que los valores de **i4** para **fd1** y **fd2** son únicos, pero el valor de **i5** no ha cambiado al crear el segundo objeto **DatosConstante**. Esto es porque es **estático** y se inicializa una vez en el momento de la carga y no cada vez que se crea un nuevo objeto.

Las variables de **v1** a **v4** demuestran el significado de una referencia **final**. Como se puede ver en **main( )**, justo porque **v2** sea **final,** no significa que no se pueda cambiar su valor. Sin embargo, no se puede reubicar **v2** a un nuevo objeto, precisamente porque es **final**. Eso es lo que **final** significa para una referencia. También se puede ver que es cierto el mismo significado para un array, que no es más que otro tipo de referencia. (No hay forma de convertir en **final** las referencias a array en sí.) Hacer las referencias **final** parece menos útil que hacer **final** a las primitivas.

Constantes blancas

Java permite la creación de *constantes blancas*, que son campos declarados como **final** pero a los que no se da un valor de inicialización. En cualquier caso, se *debe* inicializar una constante blanca antes de utilizarla, y esto lo asegura el propio compilador. Sin embargo, las constantes blancas proporcionan mucha mayor flexibilidad en el uso de la palabra clave **final** puesto que, por ejemplo, un campo **final** incluido en una clase puede ahora ser diferente para cada objeto, y sin embargo, sigue reteniendo su cualidad de inmutable. He aquí un ejemplo:

```
//: c06:CostanteBlanca.java
// Miembros de datos "Constantes blancas".

class Elemento { }

class ConstanteBlanca {
  final int i = 0; // Constante inicializada
  final int j; // Constante blanca
  final Elemento p; // Referencia a constante blanca
  // Las constantes blancas DEBEN inicializarse
  // en el constructor:
  ConstanteBlanca() {
    j = 1; // Inicializar la la constante blanca
    p = new Elemento();
  }
  ConstanteBlanca(int x) {
    j = x; // Inicializar la constante blanca
    p = new Elemento();
```

```
  }
  public static void main(String[] args) {
    ConstanteBlanca bf = new ConstanteBlanca();
  }
} ///:~
```

Es obligatorio hacer asignaciones a **constantes,** bien con una expresión en el momento de definir el campo o en el constructor. De esta forma, se garantiza que el campo **constante** se inicialice siempre antes de ser usado.

Parámetros de valor constante

Java permite hacer parámetros **constantes** declarándolos con la palabra final en la lista de parámetros. Esto significa que dentro del método no se puede cambiar aquello a lo que apunta la referencia al parámetro:

```
//: c06:ParametrosConstante.java
// Utilizando "final" con parámetros de métodos.

class Artilugio {
  public void girar() {}
}

public class ParametrosConstante {
  void con(final Artilugio g) {
    //! g = new Artilugio(); // Ilegal -- g es constante
  }
  void sin(Artilugio g) {
    g = new Artilugio(); // OK -- g no es constante
    g.girar();
  }
  // void f(final int i) { i++; } // No puede cambiar
  // Sólo se puede leer de un tipo de dato primitivo:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    ParametrosConstante bf = new ParametrosConstante();
    bf.sin(null);
    bf.con(null);
  }
} ///:~
```

Fíjese que se puede seguir asignando una referencia **null** a un parámetro **constante** sin que el compilador se dé cuenta, al igual que se puede hacer con un parámetro no **constante**.

Los métodos **f( )** y **g( )** muestran lo que ocurre cuando los parámetros primitivos son **constante**: se puede leer el parámetro pero no se puede cambiar.

Métodos constante

Hay dos razones que justifican los métodos **constante**. La primera es poner un "bloqueo" en el método para evitar que cualquier clase heredada varíe su significado. Esto se hace por razones de diseño cuando uno se quiere asegurar de que se mantenga el comportamiento del método durante la herencia, evitando que sea sobreescrito.

La segunda razón para los métodos **constante** es la eficiencia. Si se puede hacer un método **constante** se está permitiendo al compilador convertir cualquier llamada a ese método en llamadas *rápidas*. Cuando el compilador ve una llamada a un método **constante** puede (a su discreción) saltar el modo habitual de insertar código para llevar a cabo el mecanismo de invocación al método (meter los argumentos en la pila, saltar al código del método y ejecutarlo, volver al punto del salto y eliminar los parámetros de la pila, y manipular el valor de retorno) o, en vez de ello, reemplazar la llamada al método con una copia del código que, de hecho, se encuentra en el cuerpo del método. Esto elimina la sobrecarga de la llamada al método. Por supuesto, si el método es grande, el código comienza a aumentar de tamaño, y probablemente no se aprecien ganancias de rendimiento en la sustitución, puesto que cualquier mejora se verá disminuida por la cantidad de tiempo invertido dentro del método. Está implícito el que el compilador de Java sea capaz de detectar estas situaciones, y elegir sabiamente. Sin embargo, es mejor no confiar en que el compilador sea capaz de hacer esto siempre bien, y hacer un método **constante** sólo si es lo suficientemente pequeño o se desea evitar su modificación explícitamente.

constante y privado

Cualquier método **privado** de una clase es implícitamente **constante**. Dado que no se puede acceder a un método **privado**, no se puede modificar (incluso aunque el compilador no dé un mensaje de error si se intenta modificar, no se habrá modificado el método, sino que se habrá creado uno nuevo). Se puede añadir el modificador **final** a un método **privado** pero esto no da al método ningún significado extra.

Este aspecto puede causar confusión, porque si se desea modificar un método **privado** (que es implícitamente **constante**) parece funcionar:

```
//: c06:AparienciaModificacionConstante.java
// Sólo parece que se puede modificar
// un método privado o privado constante.

class ConConstantes {
  // Idéntico a únicamente "privado":
  private final void f() {
    System.out.println("ConConstantes.f()");
  }
  // También automáticamente "constante":
  private void g() {
    System.out.println("ConConstantes.g()");
  }
}
```

```
class ModificacionPrivado extends ConConstante {
  private final void f() {
    System.out.println("ModificacionPrivado.f()");
  }
  private void g() {
    System.out.println("ModificacionPrivado.g()");
  }
}

class ModificacionPrivado2
  extends ModificacionPrivado {
  public final void f() {
    System.out.println("ModificacionPrivado2.f()");
  }
  public void g() {
    System.out.println("ModificacionPrivado2.g()");
  }
}

public class AparienciaModificacionConstante {
  public static void main(String[] args) {
    ModificacionPrivado2 op2 =
      new ModificacionPrivado2();
    op2.f();
    op2.g();
    // Se puede hacer conversión hacia arriba:
    ModificacionPrivado op = op2;
    // Pero no se puede invocar a los métodos:
    //! op.f();
    //! op.g();
    // Lo mismo que aquí:
    ConCostantes wf = op2;
    //! wf.f();
    //! wf.g();
  }
} ///:~
```

La "modificación" sólo puede darse si algo es parte de la interfaz de la clase base. Es decir, uno debe ser capaz de hacer conversión hacia arriba de un objeto a su tipo base e invocar al mismo método (la esencia de esto se verá más clara en el siguiente capítulo). Si un método es **privado**, no es parte de la interfaz de la clase base. Es simplemente algún código oculto dentro de la clase, y simplemente tiene ese nombre, pero si se crea un método **público**, **protegido** o "amistoso" en la clase derivada, no hay ninguna conexión con el método que pudiese llegar a tener ese nombre en la clase base. Dado que un método **privado** es inalcanzable y a efectos invisible, no influye en nada más que en la organización del código de la clase para la que se definió.

Clases constantes

Cuando se dice que una clase entera es **constante** (precediendo su definición de la palabra clave **final**) se establece que no se desea heredar de esta clase o permitir a nadie más que lo haga. En otras palabras, por alguna razón el diseño de la clase es tal que nunca hay una necesidad de hacer cambios, o por razones de seguridad no se desea la generación de subclases. De manera alternativa, se pueden estar tratando aspectos de eficiencia, y hay que asegurarse de que cualquier actividad involucrada con objetos de esta clase sea lo más eficiente posible.

```
//: c06:Jurasico.java
// Convirtiendo una clase entera en final.

class CerebroPequenio {}

final class Dinosaurio {
  int i = 7;
  int j = 1;
  CerebroPequenio x = new CerebroPequenio();
  void f() {}
}

//! class SerEvolucionado extends Dinosaurio {}
// error: No pueda heredar de la clase constante 'Dinosaurio'

public class Jurasico {
  public static void main(String[] args) {
    Dinosaurio n = new Dinosaurio();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Fíjese que los atributos pueden ser **constantes** o no, como se desee. Las mismas reglas se aplican a los atributos independientemente de si la clase se ha definido como **constante**. Definiendo la clase como **constante** simplemente evita la herencia —nada más. Sin embargo, dado que evita la herencia, todos los métodos de una clase **constante** son implícitamente **constante**, puesto que no hay manera de modificarlos. Por tanto, el compilador tiene las mismas opciones de eficiencia como tiene si se declara un método explícitamente **constante**.

Se puede añadir el especificador **constante** a un método en una clase **constante**, pero esto no añade ningún significado.

Precaución con constantes

Puede parecer sensato hacer un método **constante** mientras se está diseñando una clase. Uno podría sentir que la eficiencia es muy importante al usar la clase y que nadie podría posiblemente desear modificar estos métodos de ninguna manera. En ocasiones esto es cierto.

Pero hay que ser cuidadoso con las suposiciones. En general, es difícil anticipar cómo se reutilizará una clase, especialmente en el caso de clases de propósito general. Si se define un método como **constante** se podría evitar la posibilidad de reutilizar la clase a través de la herencia en otros proyectos de otros programadores simplemente porque su uso fuera inimaginable.

La biblioteca estándar de Java es un buen ejemplo de esto. En particular, la clase **Vector** de Java 1.0/1.1 se usaba comúnmente y podría haber sido incluso más útil si, en aras de la eficiencia, no se hubieran hecho **constante** todos sus métodos. Es fácil de concebir que se podría desear heredar y superponer partiendo de una clase tan fundamentalmente útil, pero de alguna manera, los diseñadores decidieron que esto no era adecuado. Esto es irónico por dos razones. La primera, que la clase **Stack** hereda de **Vector**, lo que significa que un **Stack** *es* un **Vector**, lo que no es verdaderamente cierto desde el punto de vista lógico. Segundo, muchos de los métodos más importantes de **Vector**, como **addElement( )** y **elementAt( )** están sincronizados (**synchronized**). Como se verá en el Capítulo 14, esto incurre en una sobrecarga considerable que probablemente elimine cualquier ganancia proporcionada por **final**. Esto da credibilidad a la teoría de que los programadores suelen ser normalmente malos a la hora de adivinar dónde deberían intentarse las optimizaciones. Es muy perjudicial que haya un diseño tan poco refinado en una biblioteca con la que todos debemos trabajar. (Afortunadamente, la biblioteca de Java 2 reemplaza **Vector** por **ArrayList**, que se comporta mucho más correctamente. Desgraciadamente, se sigue escribiendo mucho código nuevo que usa la biblioteca antigua.)

También es interesante tener en cuenta que **Hashtable**, otra clase de biblioteca estándar importante, *no* tiene ningún método **constante**. Como se mencionó en alguna otra parte de este libro, es bastante obvio que algunas clases se diseñaron por unas personas y otras por personas completamente distintas. (Se verá que los nombres de método de **Hashtable** son mucho más breves que los de **Vector**, lo cual es otra prueba de esta afirmación.) Este es precisamente el tipo de aspecto que *no* debería ser obvio a los usuarios de una biblioteca de clases. Cuando los elementos son inconsistentes, simplemente el usuario final tendrá que trabajar más. Otra alabanza más al valor del diseño y de los ensayos de código. (Fíjese que la biblioteca de Java 2 reemplaza **Hashtable** por **HashMap**.)

Carga de clases e inicialización

En lenguajes más tradicionales, los programas se cargan de una vez como parte del proceso de arranque. Éste va seguido de la inicialización y posteriormente comienza el programa. El proceso de inicialización en estos lenguajes debe controlarse cuidadosamente de forma que el orden de inicialización de los datos **estáticos** no cause problemas. C++, por ejemplo, tiene problemas si uno de los datos **estáticos** espera que otro dato **estático** sea válido antes de haber inicializado el segundo.

Java no tiene este problema porque sigue un enfoque diferente en la carga. Dado que todo en Java es un objeto, muchas actividades se simplifican, y ésta es una de ellas. Como se aprenderá más en profundidad en el siguiente capítulo, el código compilado de cada clase existe en su propio archivo separado. El archivo no se carga hasta que se necesita el código. En general, se puede decir que "El código de las clases se carga en el momento de su primer uso". Esto no ocurre generalmente hasta que se construye el primer objeto de esa clase, pero también se da una carga cuando se accede a un dato o método **estático**.

El momento del primer uso es también donde se da la inicialización **estática**. Todos los objetos **estáticos** y el bloque de código **estático** se inicializarán en orden textual (es decir, el orden en que se han escrito en la definición de la clase) en el momento de la carga. Los datos **estáticos**, por supuesto, se inicializan únicamente una vez.

Inicialización con herencia

Ayuda a echar un vistazo a todo el proceso de inicialización, incluyendo la herencia, para conseguir una idea global de lo que ocurre. Considérese el siguiente código:

```
//: c06:Escarabajo.java
// El proceso de inicialización completo.

class Insecto {
  int i = 9;
  int j;
  Insecto() {
    visualizar("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    visualizar("static Insecto.x1 inicializado");
  static int visualizar(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Escarabajo extends Insecto {
  int k = visualizar("Escarabajo.k inicializado");
  Escarabajo() {
    visualizar("k = " + k);
    visualizar("j = " + j);
  }
  static int x2 =
    visualizar("static escarabajo.x2 inicializado");
  public static void main(String[] args) {
```

```
    visualizar("Constructor de Escarabajos ");
    Escarabajo b = new Escarabajo();
  }
} ///:~
```

La salida de este programa es:

```
static Insecto.x1 inicializado
static Escarabajo.x2 inicializado
Constructor de Escarabajos i = 9, j = 0
Escarabajo.k inicializado
k = 47
j = 39
```

Lo primero que ocurre al ejecutar **Escarabajo** bajo Java es que se intenta acceder a **Escarabajo.main( )** (un método **estático**), de forma que el cargador sale a buscar el código compilado de la clase **Escarabajo** (que resulta estar en un fichero denominado **Escarabajo.class**). En el proceso de su carga, el cargador se da cuenta de que tiene una clase base (que es lo que indica la palabra clave **extends**), y por consiguiente, la carga. Esto ocurrirá tanto si se hace como si no un objeto de esa clase. (Intente comentar la creación del objeto si se desea demostrar esto.)

Si la clase base tiene una clase base, las segunda clase base se cargaría también, y así sucesivamente. Posteriormente, se lleva a cabo la inicialización **estática** de la clase base raíz (en este caso **Insecto**), y posteriormente la siguiente clase derivada, y así sucesivamente. Esto es importante porque la inicialización estática de la clase derivada podría depender de que se inicialice adecuadamente el miembro de la clase base.

En este momento, las clases necesarias ya han sido cargadas de forma que se puede crear el objeto. Primero, se ponen a sus valores por defecto todos los datos primitivos de este objeto, y las referencias a objetos se ponen a **null** —esto ocurre en un solo paso poniendo la memoria del objeto a ceros binarios. Después se invoca al constructor de la clase base. En este caso, la llamada es automática, pero también se puede especificar la llamada al constructor de la clase base (como la primera operación en el constructor de **Escarabajo( )**) utilizando **super**. La construcción de la clase base sigue el mismo proceso en el mismo orden, como el constructor de la clase derivada. Una vez que acaba el constructor de la clase base se inicializan las variables de instancia en orden textual. Finalmente se ejecuta el resto del cuerpo del constructor.

Resumen

Tanto la herencia como la composición, permiten crear un nuevo tipo a partir de tipos ya existentes. Generalmente, sin embargo, se usa la composición para reutilizar tipos ya existentes como parte de la implementación subyacente del nuevo tipo, y la herencia cuando se desee reutilizar la interfaz. Dado que la clase derivada tiene la interfaz de la clase base, se le puede hacer una *conversión hacia arriba* hasta la clase base, lo que es crítico para el polimorfismo, como se verá en el siguiente capítulo.

A pesar del gran énfasis que la programación orientada a objetos pone en la herencia, al empezar un diseño debería generalmente preferirse la composición durante el primer corte, y la herencia sólo cuando sea claramente necesaria. La composición tiende a ser más flexible. Además, al utilizar la propiedad añadida de la herencia con un tipo miembro, se puede cambiar el tipo exacto y, por tanto, el comportamiento de aquellos objetos miembro en tiempo de ejecución. Por consiguiente, se puede cambiar el comportamiento del objeto compuesto en tiempo de ejecución.

Aunque la reutilización de código mediante la composición y la herencia es útil para el desarrollo rápido de proyectos, generalmente se deseará rediseñar la jerarquía de clases antes de permitir a otros programadores llegar a ser dependientes de ésta. La meta es una jerarquía en la que cada clase tenga un uso específico y no sea demasiado grande (agrupando tanta funcionalidad sería demasiado difícil de manejar) ni demasiado pequeño (no se podría usar por sí mismo o sin añadirle funcionalidad).

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 dos clases, **A** y **B**, con constructores por defecto (listas de parámetros vacías) que se anuncien a sí mismas. Heredar una nueva clase **C** a partir de **A**, y crear un miembro de la clase **B** dentro de **C**. No crear un constructor para **C**. Crear un objeto de la clase **C** y observar los resultados.

2. Modificar el Ejercicio 1 de forma que **A** y **B** tengan constructores con parámetros en vez de constructores por defecto. Escribir un constructor para **C** y llevar a cabo toda la inicialización dentro del constructor **C**.

3. Crear una clase simple. Dentro de una segunda clase, definir un campo para un objeto de la primera clase. Utilizar inicialización perezosa para instanciar este objeto.

4. Heredar una nueva clase de la clase **Detergente**. Superponer **frotar( )** y añadir un nuevo método denominado **esterilizar( )**.

5. Tomar el archivo **Animacion.java** y comentar el constructor de la clase **Animación**. Explicar qué ocurre.

6. Tomar el archivo **Ajedrez.java** y comentar el constructor de la clase **Ajedrez**. Explicar qué ocurre.

7. Probar que se crean constructores por defecto por parte del compilador.

8. Probar que los constructores de una clase base (a) siempre son invocados y, (b) se invocan antes que los constructores de la clase derivada.

9. Crear una clase base con sólo un constructor distinto del constructor por defecto, y una clase derivada que tenga, tanto un constructor por defecto, como uno que no lo sea. En los constructores de la clase derivada, invocar al de la clase base.

10. Crear una clase llamada **Raíz** que contenga una instancia de cada clase (que también se deben crear) denominadas **Componente1**, **Componente2**, y **Componente3**. Derivar una clase **Tallo** a partir de **Raíz** que también contenga una instancia de cada "componente". Todas las clases deberían tener constructores por defecto que impriman un mensaje relativo a ellas.

11. Modificar el Ejercicio 10 de forma que cada clase sólo tenga un constructor que no sea por defecto.

12. Añadir una jerarquía correcta de métodos **limpiar( )** a todas las clases del Ejercicio 11.

13. Crear una clase con un método sobrecargado tres veces. Heredar una nueva clase, añadir una nueva sobrecarga del método y mostrar que los cuatro métodos están disponibles para la clase derivada.

14. En **Coche.java** añadir un método **revisar( )** a **Motor** e invocar a este método en el método **main( )**.

15. Crear una clase dentro de un paquete. La clase debería contener un método **protegido**. Fuera del paquete, intentar invocar al método **protegido** y explicar los resultados. Ahora heredar de la clase e invocar al método **protegido** desde dentro del método de la clase derivada.

16. Crear una clase llamada **Anfibio**. Desde ésta, heredar una clase llamada **Rana**. Poner métodos apropiados en la clase base. En el método **main( )**, crear una **Rana** y hacer una conversión hacia **Anfibio**. Demostrar que todos los métodos siguen funcionando.

17. Modificar el Ejercicio 16 de forma que **Rana** superponga las definiciones de métodos de la clase base (proporciona nuevas definiciones usando los mismos nombres de método). Fijarse en lo que ocurre en el método **main( )**.

18. Crear una clase con un campo **estático constante** y un campo **constante**, y demostrar la diferencia entre los dos.

19. Crear una clase con una referencia constante **blanca** a un objeto. Llevar a cabo la inicialización de la constante *blanca* **final** dentro del método (no en el constructor) justo antes de usarlo. Demostrar que debe inicializarse la **constante** antes de usarlos, y que no puede cambiarse una vez inicializada.

20. Crear una clase con un método **constante**. Heredar desde esa clase e intentar superponer ese método.

21. Crear una clase **constante** e intentar heredar de ella.

22. Probar que la carga de clases sólo se da una vez. Probar que la carga puede ser causada, bien por la creación de la primera instancia de esa clase, o por el acceso a un miembro **estático**.

23. En **Escarabajo.java**, heredar un tipo específico de escarabajo de la clase **Escarabajo**, siguiendo el mismo formato que el de la clase existente. Hacer un seguimiento y explicar la salida.