7: Polimorfismo

El polimorfismo es la tercera característica esencial de los lenguajes de programación orientados a objetos, después de la abstracción de datos y la herencia.

Proporciona otra dimensión de separación de la interfaz de la implementación, separa el *qué* del *cómo*. El polimorfismo permite una organización de código y una legibilidad del mismo mejorada, además de la creación de programas *ampliables* que pueden "crecer", no sólo durante la creación original del proyecto sino también cuando se deseen añadir nuevas características.

La encapsulación crea nuevos tipos de datos mediante la combinación de características y comportamientos. La ocultación de la implementación separa la interfaz de la implementación haciendo los detalles **privados**. Este tipo de organización mecánica tiene bastante sentido para alguien con un trasfondo procedural de programación. Pero el polimorfismo tiene que ver con la separación en términos de *tipos*. En el capítulo anterior se vio como la herencia permite el tratamiento de un objeto como si fuera de sus propio tipo *o* del tipo base. Esta característica es crítica porque permite que varios tipos (derivados de un mismo tipo base) sean tratados como si fueran uno sólo, y un único fragmento de código se puede ejecutar de igual forma en todos los tipos diferentes. La llamada a un método polimórfico permite que un tipo exprese su distinción de otro tipo similar, puesto que ambos se derivan del mismo tipo base. Esta distinción se expresa a través de diferencias en comportamiento de los métodos a los que se puede invocar a través de la clase base.

En este capítulo, se aprenderá lo relacionado con el polimorfismo (llamado también *reubicación dinámica*, *reubicación tardía* o *reubicación en tiempo de ejecución*) partiendo de la base, con ejemplos simples que prescinden de todo, menos del comportamiento polimórfico del programa.

De nuevo la conversión hacia arriba

En el Capítulo 6 se vio cómo un objeto puede usarse con su propio tipo o como un objeto de su tipo base. Tomar una referencia a un objeto y tratarla como una referencia a su clase base se denomina *conversión hacia arriba*, debido a la forma en que se dibujan los árboles de herencia, en los que la clase base se coloca siempre en la parte superior.

También se vio que surge un problema, como se aprecia en:

```
//: c07:musica:Musica.java
// Herencia y conversión hacia arriba.

class Nota {
  private int valor;
  private Nota(int val) { valor = val; }
  public static final Nota
    DO_MAYOR = new Nota(0),
    DO_SOSTENIDO  = new Nota(1),
```

```
    SI_BEMOL    = new Nota(2);
} // Etc.

class Instrumento {
  public void tocar(Nota n) {
    System.out.println("Instrumento.tocar()");
  }
}

// Los objetos de viento son instrumentos
// dado que tienen la misma interfaz:
class Viento extends Instrumento {
  // Redefinir el metodo interfaz:
  public void tocar(Nota n) {
    System.out.println("Viento.tocar()");
  }
}

public class Musica {
  public static void afinar(Instrumento i) {
    // ...
    i.tocar(Nota.DO_SOSTENIDO);
  }
  public static void main(String[] args) {
    Viento flauta = new Viento();
    afinar(flauta); // Conversión hacia arriba
  }
} ///:~
```

El método **Musica.afinar( )** acepta una referencia a **Instrumento**, pero también cualquier cosa que se derive de **Instrumento**. En el método **main( )**, se puede ver que ocurre esto pues se pasa una referencia **Viento** a **afinar( )**, sin que sea necesaria ninguna conversión. Esto es aceptable; la interfaz de **Instrumento** debe existir en **Viento,** puesto que **Viento** se hereda de **Instrumento**. Hacer una conversión hacia arriba de **Viento** a **Instrumento** puede "estrechar" esa interfaz, pero no puede reducirlo a nada menos de lo contenido en la interfaz de **Instrumento**.

Olvidando el tipo de objeto

Este programa podría parecer extraño. ¿Por qué debería alguien *olvidar* intencionadamente el tipo de objeto? Esto es lo que ocurre cuando se hace conversión hacia arriba, y parece que podría ser mucho más directo si **afinar( )** simplemente tomara una referencia **Viento** como argumento. Esto presenta un punto esencial: si se hiciera esto se necesitaría escribir un nuevo método **afinar( )** para cada tipo de **Instrumento** del sistema. Supóngase que se sigue este razonamiento y se añaden los instrumentos de **Cuerda** y **Metal**:

```
//: c07:musica2:Musica2.java
// Sobrecarga en vez de conversión hacia arriba.

class Nota {
  private int valor;
  private Nota(int val) { valor = val; }
  public static final Nota
    DO-MAYOR = new Nota(0),
    DO-SOSTENIDO = new Nota(1),
    SI-BEMOL = new Nota(2);
} // Etc.

class Instrumento {
  public void tocar(Nota n) {
    System.out.println("Instrumento.tocar()");
  }
}

class Viento extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Viento.tocar()");
  }
}

class Cuerda extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Cuerda.tocar()");
  }
}

class Metal extends Instrumento {
  public void tocar(Nota n) {
    System.out.println("Metal.tocar()");
  }
}

public class Musica2 {
  public static void afinar(Viento i) {
    i.tocar(Nota.DO-MAYOR);
  }
  public static void afinar(Cuerda i) {
    i.tocar(Nota.DO-MAYOR);
  }
  public static void afinar(Metal i) {
    i.tocar(Nota.DO-MAYOR);
```

```
  }
  public static void main(String[] args) {
    Viento flauta = new Viento();
    Cuerda violin = new Cuerda();
    Metal trompeta = new Metal();
    afinar(flauta); // Sin conversión hacia arriba
    afinar(violin);
    afinar(trompeta);
  }
} ///:~
```

Esto funciona, pero hay un inconveniente: se deben escribir métodos específicos de cada tipo para cada clase **Instrumento** que se añada. En primer lugar esto significa más programación, pero también quiere decir que si se desea añadir un método nuevo como **afinar( )** o un nuevo tipo de **Instrumento**, se tiene mucho trabajo por delante. Añadiendo el hecho de que el compilador no emitirá ningún mensaje de error si se olvida sobrecargar alguno de los métodos, el hecho de trabajar con tipos podría convertirse en inmanejable.

¿No sería muchísimo mejor si simplemente se pudiera escribir un único método que tomara como parámetro la clase base, y no cualquiera de las clases específicas derivadas? Es decir, ¿no sería genial que uno se pudiera olvidar de que hay clases derivadas, y escribir un código que sólo tratara con la clase base?

Esto es exactamente lo que permite hacer el polimorfismo. Sin embargo, la mayoría de programadores que provienen de lenguajes procedurales, tienen problemas para entender el funcionamiento de esta característica.

El cambio

La dificultad con **Musica.java** se puede ver ejecutando el programa. La salida es **Viento.tocar( )** Ésta es, ciertamente, la salida deseada, pero no parece tener sentido que funcione de esa forma. Obsérvese el método **afinar( )**:

```
  public static void afinar(Instrumento i) {
    // ...
    i.tocar(Nota.DO-MAYOR);
  }
```

Recibe una referencia a **Instrumento**. Por tanto, ¿cómo puede el compilador saber que esta referencia a **Instrumento** apunta a **Viento** en este caso, y no a **Cuerda** o **Metal**? El compilador de hecho no puede saberlo. Para lograr un entendimiento más profundo de este aspecto, es útil echar un vistazo al tema de la *ligadura*.

La ligadura en las llamadas a métodos

La conexión de una llamada a un método se denomina *ligadura*. Cuando se lleva a cabo la *ligadura* antes de ejecutar el programa (por parte del compilador y el montador, cuando lo hay) se denomina *ligadura temprana*. Puede que este término parezca extraño pues nunca ha sido una opción con los lenguajes procedurales. Los compiladores de C tienen un único modo de invocar a un método utilizando la ligadura temprana.

La parte confusa del programa de arriba no se resuelve fácilmente con la *ligadura temprana* pues el compilador no puede saber el método correcto a invocar cuando sólo tiene una referencia a un **Instrumento**.

La solución es la *ligadura tardía*, que implica que la correspondencia se da en tiempo de ejecución, basándose en el tipo de objeto. La *ligadura tardía* se denomina también *dinámica* o *en tiempo de ejecución*. Cuando un lenguaje implementa la ligadura tardía, debe haber algún mecanismo para determinar el tipo de objeto en tiempo de ejecución e invocar al método adecuado. Es decir, el compilador sigue sin saber el tipo de objeto, pero el mecanismo de llamada a métodos averigua e invoca al cuerpo de método correcto. El mecanismo de *la ligadura tardía* varía de un lenguaje a otro, pero se puede imaginar que es necesario instalar algún tipo de información en los objetos.

Toda ligadura de métodos en Java se basa en la ligadura tardía a menos que se haya declarado un método como **constante**. Esto significa que ordinariamente no es necesario tomar decisiones sobre si se dará la ligadura tardía, sino que esta decisión se tomará automáticamente.

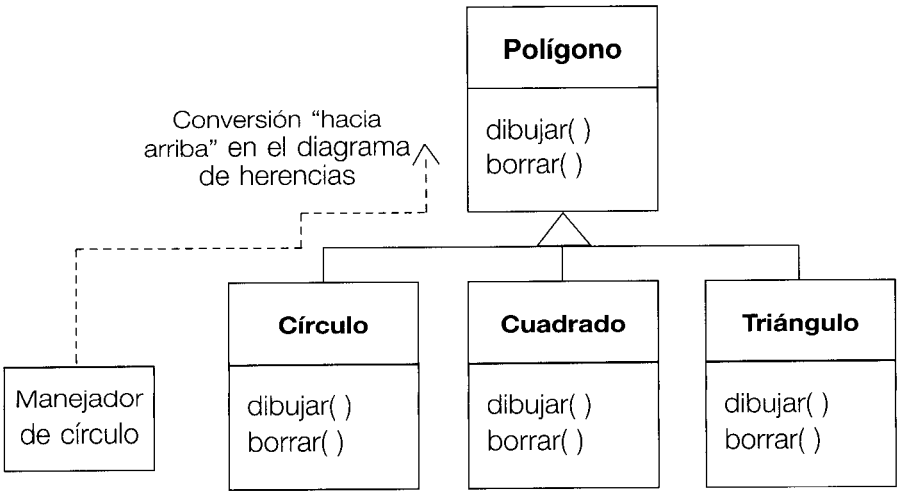
¿Por qué declarar un método como **constante**? Como se comentó en el capítulo anterior, evita que nadie superponga el método. Todavía más importante, "desactiva" ligadura dinámica, o mejor, es que, le dice al compilador que este tipo de ligadura no es necesaria. Esto permite al compilador generar código ligeramente más eficiente para llamadas a métodos **constantes**. Sin embargo, en la mayoría de los casos no se obtendrá ninguna mejora global de rendimiento del programa, por lo que es mejor usar métodos **constantes** únicamente como una decisión de diseño, y no para intentar mejorar el rendimiento.

Produciendo el comportamiento adecuado

Una vez que se sabe que toda la ligadura de métodos en Java se da de forma polimórfica a través de ligadura tardía, se puede escribir código que trate la clase base y saber que todas las clases derivadas funcionarán correctamente al hacer uso de ese mismo código. Dicho de otra forma, se "envía un mensaje a un objeto y se deja que éste averigüe la opción correcta a realizar".

El ejemplo clásico de POO es el ejemplo de los "polígonos". Éste se usa frecuentemente porque es fácil de visualizar, pero desgraciadamente puede confundir a los programadores novatos, haciéndoles pensar que la POO sólo se usa en programación de gráficos, y esto no es cierto.

El ejemplo de los polígonos tiene una clase base denominada **Polígono** y varios tipos derivados: **Círculo**, **Cuadrado** y **Triángulo**, etc. La razón por la que el ejemplo funciona tan bien es porque se puede decir sin problema "un círculo es un tipo de polígono" y se entiende. El diagrama de herencia muestra las relaciones:



La conversión hacia arriba podría darse en una sentencia tan simple como:

```
Poligono s = new Circulo();
```

Aquí, se crea un objeto **Círculo** y la referencia resultante se asigna directamente a un **Polígono**, lo que podría parecer un error (asignar un tipo a otro); y sin embargo, está bien porque un **Círculo** *es un* **Polígono** por herencia. Por tanto, el compilador se muestra de acuerdo con la sentencia y no muestra ningún mensaje de error.

Supóngase que se invoca a uno de los métodos de la clase base (que han sido superpuestos en clases derivadas):

```
s.dibujar();
```

De nuevo, se podría esperar que se invoque al método **dibujar( )** de **Polígono** porque se trata, después de todo, de una referencia a **Polígono** —por tanto, ¿cómo podría el compilador saber que tiene que hacer otra cosa? Y sin embargo, se invoca al **Círculo.dibujar( )** correcto debido a la ligadura tardía (polimorfismo).

El ejemplo siguiente hace lo propio de una manera ligeramente distinta:

```
//: c07:Poligonos.java
// Polimorfismo en Java.

class Poligono {
  void dibujar() {}
  void borrar() {}
}

class Circulo extends Poligono {
  void dibujar() {
    System.out.println("Circulo.dibujar()");
```

```
    }
    void borrar() {
      System.out.println("Circulo.borrar()");
    }
}

class Cuadrado extends Poligono {
    void dibujar() {
      System.out.println("Cuadrado.dibujar()");
    }
    void borrar() {
      System.out.println("Cuadrado.borrar()");
    }
}

class Triangulo extends Poligono {
    void dibujar() {
      System.out.println("Triangulo.dibujar()");
    }
    void borrar() {
      System.out.println("Triangulo.borrar()");
    }
}

public class Poligonos {
    public static PoligonoAleatorio () {
      switch((int)(Math.random() * 3)) {
        default:
        case 0: return new Circulo();
        case 1: return new Cuadrado();
        case 2: return new Triangulo();
      }
    }
    public static void main(String[] args) {
      Poligono[] s = new Poligono[9];
      // Rellenar el array con Polígonos:
      for(int i = 0; i < s.length; i++)
        s[i] = PoligonoAleatorio();
      // Hacer llamadas a métodos polimórficos:
      for(int i = 0; i < s.length; i++)
        s[i].dibujar();
    }
} ///:~
```

La clase base **Polígono** establece la interfaz común a cualquier cosa heredada de **Polígono** —es decir, se pueden borrar y dibujar todos los polígonos. La clase derivada superpone estas definiciones para proporcionar un comportamiento único para cada tipo específico de polígono.

La clase principal **Polígonos** contiene un método **estático** llamado **poligonoAleatorio( )** que produce una referencia a un objeto **Polígonos** seleccionado al azar cada vez que se le invoca. Fíjese que se realiza una conversión hacia arriba en cada sentencia **return** que toma una referencia a un **Círculo**, **Cuadrado** o **Triángulo,** y lo envía fuera del método con tipo de retorno **Polígonos**. Así, al invocar a este método no se tendrá la opción de ver de qué tipo específico es el valor devuelto, dado que siempre se obtendrá simplemente una referencia a **Polígono**.

El método **main( )** contiene un array de referencias **Polígono** rellenadas mediante llamadas a **poligonoAleatorio( )**. En este punto se sabe que se tienen objetos de tipo **Polígono**, pero no se sabe nada sobre nada más específico que eso (y tampoco el compilador). Sin embargo, cuando se recorre este array y se invoca al método **dibujar( )** para cada uno de sus objetos, mágicamente se da el comportamiento correcto específico de cada tipo, como se puede ver en un ejemplo de salida:

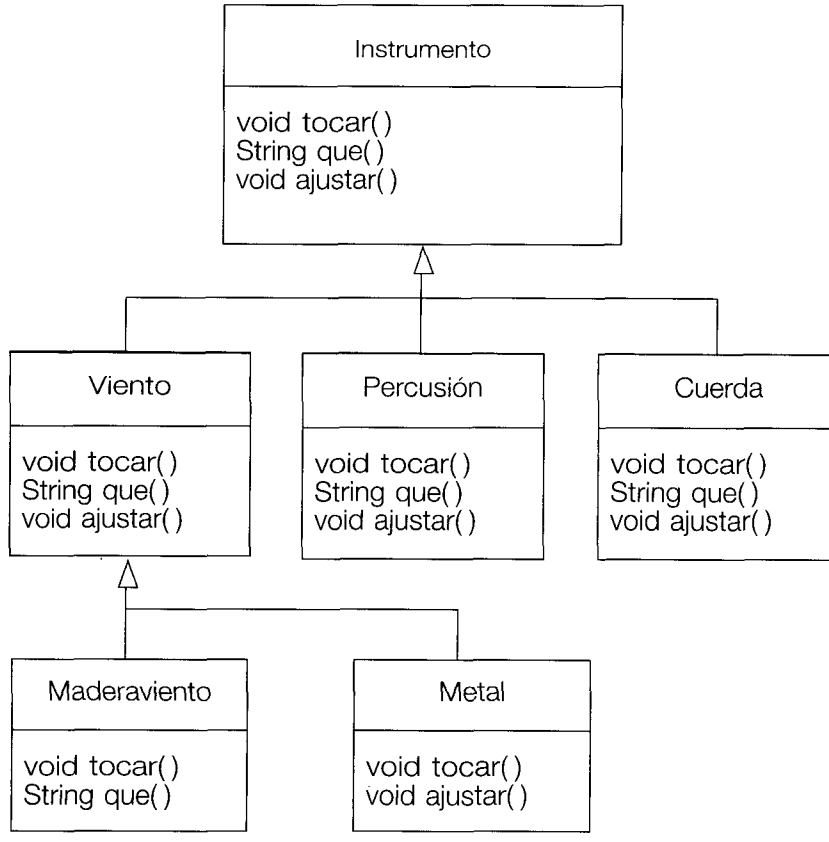
```
Circulo.dibujar()
Triangulo.dibujar()
Circulo.dibujar()
Circulo.dibujar()
Circulo.dibujar()
Cuadrado.dibujar()
Triangulo.dibujar()
Cuadrado.dibujar()
Cuadrado.dibujar()
```

Por supuesto, dado que los polígonos se eligen cada vez al azar, cada ejecución tiene resultados distintos. El motivo de elegir los polígonos al azar es abrirse paso por la idea de que el compilador no puede tener ningún conocimiento que le permita hacer las llamadas correctas en tiempo de compilación. Todas las llamadas a **dibujar( )** se hacen mediante ligadura dinámica.

Extensibilidad

Ahora, volvamos al ejemplo de los instrumentos musicales. Debido al polimorfismo, se pueden añadir al sistema tantos tipos como se desee sin cambiar el método **afinar( )**. En un programa POO bien diseñado, la mayoría de métodos deberían seguir el modelo de **afinar( )** y comunicarse sólo con la interfaz de la clase base. Un programa así es *extensible* porque se puede añadir nueva funcionalidad heredando nuevos tipos de datos de la clase base común. Los métodos que manipulan la interfaz de la clase base no necesitarán ningún cambio si se desea acomodarlos a las nuevas clases.

Considérese qué ocurre si se toma el ejemplo de los instrumentos y se añaden nuevos métodos a la clase base y varias clases nuevas. He aquí el diagrama:



Todas estas nuevas clases funcionan correctamente con el viejo método **afinar( )** sin tocarlo. Incluso si **afinar( )** se encuentra en un archivo separado y se añaden métodos de la interfaz de **Instrumento**, **afinar( )** funciona correctamente sin tener que volver a compilarlo. He aquí una implementación del diagrama de arriba:

```
//: c07:musica3:Musica3.java
// Un programa extensible.
import java.util.*;

class Instrumento {
  public void tocar() {
    System.out.println("Instrumento.tocar()");
  }
  public String que() {
    return "Instrumento";
  }
  public void ajustar() {}
}

class Viento extends Instrumento {
```

```
  public void tocar() {
    System.out.println("Viento.tocar()");
  }
  public String que() { return "Viento"; }
  public void ajustar() {}
}

class Percusion extends Instrumento {
  public void tocar() {
    System.out.println("Percusion.tocar()");
  }
  public String que() { return "Percusion"; }
  public void ajustar() {}
}

class Cuerda extends Instrumento {
  public void tocar() {
    System.out.println("Cuerda.tocar()");
  }
  public String que() { return "Cuerda"; }
  public void ajustar() {}
}

class Metal extends Viento {
  public void tocar() {
    System.out.println("Metal.tocar()");
  }
  public void ajustar() {
    System.out.println("Metal.ajustar()");
  }
}

class Maderaviento extends Viento {
  public void tocar() {
    System.out.println("Maderaviento.tocar()");
  }
  public String que() { return "Maderaviento"; }
}

public class Musica3 {
  // No le importa el tipo por lo que los nuevos tipos
  // que se aniadan al sistema seguirán funcionando bien:
  static void afinar(Instrumento i) {
    // ...
    i.tocar();
```

```
  }
  static void afinarTodo(Instrumento[] e) {
    for(int i = 0; i < e.length; i++)
      afinar(e[i]);
  }
  public static void main(String[] args) {
    Instrumento[] orquesta = new Instrumento[5];
    int i = 0;
    // Conversión hacia arriba durante inserción en el array:
    orquesta[i++] = new Viento();
    orquesta[i++] = new Percusion();
    orquesta[i++] = new Cuerda();
    orquesta[i++] = new Metal();
    orquesta[i++] = new Maderaviento();
    afinarTodo(orquesta);
  }
} ///:~
```

Los nuevos métodos son **que( )**, que devuelve una referencia a una cadena de caracteres con una descripción de la clase, y **ajustar( )**, que proporciona alguna manera de ajustar cada instrumento.

En **main( )**, cuando se coloca algo dentro del array **Instrumento** se puede hacer una conversión hacia arriba automáticamente a **Instrumento**.

Se puede ver que el método **afinar( )** ignora por completo todos los cambios de código que hayan ocurrido alrededor, y sigue funcionando correctamente. Esto es exactamente lo que se supone que proporciona el polimorfismo. Los cambios en el código no causan daño a partes del programa que no deberían verse afectadas. Dicho de otra forma, el polimorfismo es una de las técnicas más importantes que permiten al programador "separar los elementos que cambian de aquellos que permanecen igual".

Superposición frente a sobrecarga

Tomemos un enfoque distinto del primer enfoque de este capítulo. En el programa siguiente, se cambia la interfaz del método **tocar( )** en el proceso de sobrecarga, lo que significa que no se ha *superpuesto* el método, sino que se ha *sobrecargado*. El compilador permite sobrecargar métodos de forma que no haya quejas. Pero el comportamiento no es probablemente lo que se desea. He aquí un ejemplo:

```
//: c07:ErrorViento.java
// Cambiando la interfaz accidentalmente.

class NotaX {
  public static final int
    DO_MAYOR_C = 0, DO_SOSTENIDO = 1, SI_BEMOL = 2;
```

```
}

class InstrumentoX {
  public void tocar(int NotaX) {
    System.out.println("InstrumentoX.tocar()");
  }
}

class VientoX extends InstrumentoX {
  // Cambia la interfaz del método:
  public void tocar(NotaX n) {
    System.out.println("VientoX.tocar(NotaX n)");
  }
}

public class ErrorViento {
  public static void afinar(InstrumentoX i) {
    // ...
    i.tocar(NoteX.DO_MAYOR);
  }
  public static void main(String[] args) {
    VientoX flauta = new VientoX();
    afinar(flauta); // ¡No es el comportamiento deseado!
  }
} ///:~
```

Hay otro aspecto confuso en este caso. En **InstrumentoX**, el método **tocar( )** toma un dato **entero** con el identificador **NotaX**. Es decir, incluso aunque **NotaX** es un nombre de clase, también puede usarse como identificador sin problemas. Pero en **VientoX**, **tocar( )** toma una referencia a **NotaX** que tiene un identificador **n**. (Aunque podría incluso decirse **tocar(NotaX NotaX)** sin que diera error.) Por consiguiente parece que el programador pretendía superponer **tocar( )** pero equivocó los tipos del método. El compilador, sin embargo, asumió que se pretendía una sobrecarga y no una superposición. Fíjese que si se sigue la convención de nombres estándar de Java, el identificador de parámetros sería **notaX** ('n' minúscula), que lo distinguiría del nombre de la clase.

En **afinar**, se envía al **InstrumentoX i** el mensaje **tocar( )**, con uno de los miembros de **NotaX** (**DO_MAYOR**) como parámetro. Dado que **NotaX** contiene definiciones **int,** se invoca a la versión ahora sobrecargada del método **tocar( )**, y dado que ése *no* ha sido superpuesto, se emplea la versión de la clase base.

La salida es:

```
InstrumentoX.tocar()
```

Ciertamente esto no parece ser una llamada a un método polimórfico. Una vez que se entiende lo que está ocurriendo, se puede solventar el problema de manera bastante sencilla, pero imagínese lo

difícil que podría ser encontrar el fallo cuando se encuentre inmerso en un programa de tamaño significativo.

Clases y métodos abstractos

En todos los ejemplos de instrumentos, los métodos de la clase base **Instrumento** eran métodos "falsos". Si se llega a invocar alguna vez a estos métodos daría error. Esto es porque la intención de **Instrumento** es simplemente crear una *interfaz común* para todas las clases que se derivan del mismo.

La única razón para establecer esta interfaz común es que ésta se pueda expresar de manera distinta para cada subtipo diferente. Establece una forma básica, de forma que se puede decir qué tiene en común con todas las clases derivadas.

Otra manera de decir esto es llamar a la clase **Instrumento**, una *clase base abstracta* (o simplemente *clase abstracta*). Se crea una clase abstracta cuando se desea manipular un conjunto de clases a través de una interfaz común. Todos los métodos de clases derivadas que encajen en la declaración de la clase base se invocarán utilizando el mecanismo de ligadura dinámica. (Sin embargo, como se vio en la sección anterior, si el nombre del método es el mismo que en la clase base, pero los parámetros son diferentes, se tiene sobrecarga, lo cual probablemente no es lo que se desea.)

Si se tiene una clase abstracta como **Instrumento**, los objetos de esa clase casi nunca tienen significado. Es decir, **Instrumento** simplemente tiene que expresar la interfaz, y no una implementación particular, de forma que no tiene sentido crear objetos de tipo **Instrumento**, y probablemente se desea evitar que ningún usuario llegue a hacerlo. Esto se puede lograr haciendo que todos los métodos de **Instrumento** muestren mensajes de error, pero de esta forma se retrasa la información hasta tiempo de ejecución, y además es necesaria una comprobación exhaustiva y de confianza por parte del usuario. Siempre es mejor capturar los problemas en tiempo de ejecución.

Java proporciona un mecanismo para hacer esto, denominado el *método abstracto*[1]. Se trata de un método incompleto; tiene sólo declaración faltándole los métodos. La sintaxis para una declaración de método abstracto es:

```
abstract void f();
```

Toda clase que contenga uno o más métodos abstractos, se califica de **abstracto**. (De todos modos, el compilador emite un mensaje de error).

Si una clase abstracta está incompleta, ¿qué se supone que debe hacer el compilador cuando alguien intenta crear un objeto de esa clase? No se puede crear un objeto de una clase abstracta de forma segura, por lo que se obtendrá un mensaje de error del compilador. De esta manera, el compilador asegura la pureza de la clase abstracta, y no hay que preocuparse de usarla mal.

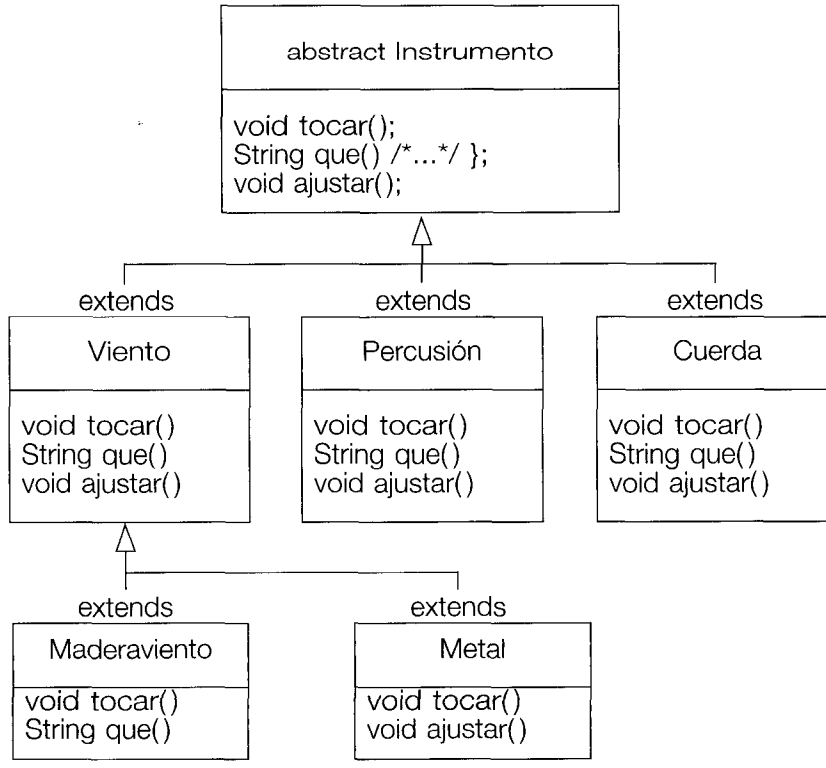
Si se hereda de una clase abstracta y se desea hacer objetos del nuevo tipo, hay que proporcionar definiciones de métodos para todos los métodos que en la clase base eran abstractos. Si no se hace así (y uno puede elegir no hacerlo) entonces la clase derivada también será abstracta y el compilador obligará a calificar *esa* clase con la palabra clave **abstract.**

---

[1] Para los programadores de C++, esto es análogo a la *función virtual pura* de C++.

Es posible crear una clase **abstracta** sin incluir ningún método **abstracto** en ella. Esto es útil cuando se desea una clase en la que no tiene sentido tener métodos **abstractos**, y se desea evitar que existan instancias de esa clase.

La clase **Instrumento** puede convertirse fácilmente en una clase **abstracta**. Sólo serán **abstractos** alguno de los métodos, puesto que hacer una clase **abstracta** no fuerza a hacer **abstractos** todos sus métodos. Quedará del siguiente modo:



He aquí el código de la orquesta modificado para que use clases y métodos **abstractos**:

```
//: c07:musica4:Musica4.java
// Clases y métodos abstractos.
import java.util.*;

abstract class Instrumento {
  int i; // almacenamiento asignado a cada uno
  public abstract void tocar();
  public String que() {
    return "Instrumento";
  }
  public abstract void ajustar();
}
```

```
class Viento extends Instrumento {
  public void tocar() {
    System.out.println("Viento.tocar()");
  }
  public String que() { return "Viento"; }
  public void ajustar() {}
}

class Percusion extends Instrumento {
  public void tocar() {
    System.out.println("Percusion.tocar()");
  }
  public String que() { return "Percusion"; }
  public void ajustar() {}
}

class Cuerda extends Instrumento {
  public void tocar() {
    System.out.println("Cuerda.tocar()");
  }
  public String que() { return "Cuerda"; }
  public void ajustar() {}
}

class Metal extends Viento {
  public void tocar() {
    System.out.println("Metal.tocar()");
  }
  public void ajustar() {
    System.out.println("Metal.ajustar()");
  }
}

class Maderaviento extends Viento {
  public void tocar() {
    System.out.println("Maderaviento.tocar()");
  }
  public String que() { return "Maderaviento"; }
}

public class Musica4 {
  // No le importa el tipo, por lo que los nuevos tipos
  // que se aniadan al sistema seguirán funcionando correctamente:
  static void afinar(Instrumento i) {
```

```
    // ...
    i.tocar();
  }
  static void afinarTodo(Instrumento[] e) {
    for(int i = 0; i < e.length; i++)
      afinar(e[i]);
  }
  public static void main(String[] args) {
    Instrumento[] orquesta = new Instrumento[5];
    int i = 0;
    // Conversión hacia arriba durante la inversión en el array:
    orquesta[i++] = new Viento();
    orquesta[i++] = new Percusion();
    orquesta[i++] = new Cuerda();
    orquesta[i++] = new Metal();
    orquesta[i++] = new Maderaviento();
    afinarTodo(orquesta);
  }
} ///:~
```

Se puede ver que realmente no hay cambios más que en la clase base.

Ayuda crear clases y métodos **abstractos** porque hacen que esa abstracción de la clase sea explícita, e indican, tanto al usuario, como al compilador cómo se tiene que usar.

Constructores y polimorfismo

Como es habitual, los constructores son distintos de otros tipos de métodos. Esto también es cierto cuando se ve involucrado el polimorfismo. Incluso aunque los constructores no sean polimórficos (aunque se puede tener algún tipo de "constructor virtual", como se verá en el Capítulo 12), es importante entender la forma en que trabajan los constructores en jerarquías complejas y con polimorfismo. Esta idea ayudará a evitar posteriores problemas.

Orden de llamadas a constructores

El orden de las llamadas a los constructores se comentó brevemente en el Capítulo 4, y de nuevo en el Capítulo 6, pero esto fue antes de introducir el polimorfismo.

En el constructor de una clase derivada siempre se invoca a un constructor de la clase base, encadenando la jerarquía de herencias de forma que se invoca a un constructor de cada clase base. Esto tiene sentido porque el constructor tiene un trabajo especial: ver que el objeto se ha construido correctamente. Una clase derivada tiene acceso, sólo a sus propios miembros, y no a aquéllos de la clase base (cuyos miembros suelen ser generalmente **privados**). Sólo el constructor de la clase base tiene el conocimiento adecuado y el acceso correcto para inicializar sus propios elementos. Por consiguiente, es esencial que se llegue a invocar a todos los constructores, si no, no se construiría

el objeto entero. Ésta es la razón por la que el compilador realiza una llamada al constructor por cada una de las clases derivadas. Si no se llama explícitamente al constructor de la clase base en el cuerpo del constructor de la clase derivada, llamará al constructor por defecto. Si no hay constructor por defecto, el compilador se quejará. (En el caso en que una clase no tenga constructores, el compilador creará un constructor por defecto automáticamente.)

Echemos un vistazo a un ejemplo que muestra los efectos de la composición, la herencia y el polimorfismo en el orden de construcción:

```
//: c07:Bocadillo.java
// Orden de llamadas a constructores.

class Comida {
  Comida() { System.out.println("Comida()"); }
}

class Pan {
  Pan() { System.out.println("Pan()"); }
}

class Queso {
  Queso() { System.out.println("Queso()"); }
}

class Lechuga {
  Lechuga() { System.out.println("Lechuga()"); }
}

class Almuerzo extends Comida {
  Almuerzo() { System.out.println("Almuerzo()");}
}

class AlmuerzoPortable extends Almuerzo {
  AlmuerzoPortable() {
    System.out.println("AlmuerzoPortable ()");
  }
}

class Bocadillo extends AlmuerzoPortable {
  Pan b = new Pan();
  Queso c = new Queso();
  Lechuga l = new Lechuga();
  Bocadillo() {
    System.out.println("Bocadillo()");
  }
```

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

Este ejemplo crea una clase compleja a partir de las otras clases, y cada clase tiene un constructor que la anuncia a sí misma. La clase principal es **Bocadillo**, que refleja tres niveles de herencia (cuatro, si se cuenta la herencia implícita de **Object**) y tres objetos miembro. Cuando se crea un objeto **Bocadillo** en el método **main( )**, la salida es:

```
Comida()
Almuerzo()
AlmuerzoPortable()
Pan()
Queso()
Lechuga()
Bocadillo()
```

Esto significa que el orden de las llamadas al constructor para un objeto completo es como sigue:

1. Se invoca al constructor de la clase base. Este paso se repite recursivamente de forma que se construya primero la raíz de la jerarquía, seguida de la siguiente clase derivada, etc. y así hasta que se llega a la última clase derivada.
2. Se llama a los inicializadores de miembros en el orden de declaración.
3. Se llama al cuerpo del constructor de la clase derivada.

El orden de las llamadas a los constructores es importante. Cuando se hereda, se sabe todo lo relativo a la clase base y se puede acceder a cualquier miembro **público** y **protegido** de la clase base. Esto significa que debemos ser capaces de asumir que todos los miembros de la clase base sean válidos cuando se está en la clase derivada. En un método normal, la construcción ya ha tenido lugar, de forma que se han construido todos los miembros de todas las partes del objeto. Dentro del constructor, sin embargo, hay que ser capaz de asumir que se han construido todos los miembros que se usan. La única garantía de esto es que se llame primero al constructor de la clase base. Después, en el constructor de la clase derivada, se inicializarán todos los miembros a los que se puede acceder en la clase base. "Saber que son válidos todos los miembros" dentro del constructor es otra razón para, cuando sea posible, inicializar todos los objetos miembro (es decir, los objetos ubicados en la clase utilizando la composición) al definir la clase (por ejemplo, b, c y l en el ejemplo anterior). Si se sigue esta práctica, se ayudará a asegurar que se han inicializado todos los miembros de la clase base y los objetos miembro del objeto actual. Desgraciadamente, esto no gestiona todos los casos, como se verá en la siguiente sección.

Herencia y **finalize( )**

Cuando se usa composición para crear una clase nueva, no hay que preocuparse nunca de finalizar los objetos miembros de esa clase. Cada miembro es un objeto independiente, y por consiguiente, será eliminado por el recolector de basura de modo independiente. Con la herencia, sin embargo,

hay que superponer el método **finalize( )** de la clase derivada si se tiene alguna limpieza especial a realizar como parte de la recolección de basura. Cuando se superpone el método **finalize( )** en una clase heredada, es importante que recordemos invocar a la versión de la clase base de **finalize( )**, puesto que de otra forma no se finalizará la clase base. El siguiente ejemplo lo prueba:

```
//: c07:Rana.java
// Probando finalize con herencia.

class HacerFinalizacionBase {
  public static boolean indicador = false;
}

class Caracteristica {
  String s;
  Caracteristica(String c) {
    s = c;
    System.out.println(
      "Creando Caracteristica " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizando Caracteristica " + s);
  }
}

class CriaturaViviente {
  Caracteristica p =
    new Caracteristica("esta vivo");
  CriaturaViviente() {
    System.out.println("CriaturaViviente()");
  }
  protected void finalize() throws Throwable {
    System.out.println(
      "Finalizando CriaturaViviente");
    // ¡Llamar a la versión de la clase base al FINAL!
    if(HacerFinalizacionBase.indicador)
      super.finalize();
  }
}

class Animal extends CriaturaViviente {
  Caracteristica p =
    new Caracteristica("tiene corazon");
  Animal() {
```

```
    System.out.println("Animal()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Finalizando Animal");
    if(HacerFinalizacionBase.indicador)
      super.finalize();
  }
}

class Anfibio extends Animal {
  Caracteristica p =
    new Caracteristica("puede vivir en el agua");
  Anfibio() {
    System.out.println("Anfibio()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Finalizando Anfibio");
    if(HacerFinalizacionBase.indicador)
      super.finalize();
  }
}

public class Rana extends Anfibio {
  Rana() {
    System.out.println("Rana()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Finalizando Rana");
    if(HacerFinalizacionBase.indicador)
      super.finalize();
  }
  public static void main(String[] args) {
    if(args.length != 0 &&
       args[0].equals("finalizar"))
       HacerFinalizacionBase.indicador = true;
    else
      System.out.println("No finalizando las bases");
    new Rana(); // Se convierte en basura automáticamente
    System.out.println("¡Adios!");
    // Forzar la invocación de todas las funciones:
    System.gc();
  }
} ///:~
```

La clase **HacerFinalizacionBase** simplemente guarda un *indicador* que informa a cada clase de la jerarquía si debe llamar a **super.finalize( )**. Este *indicador* se pone a uno con un parámetro de línea de comandos, de forma que se puede ver el comportamiento con y sin finalización de la clase base.

Cada clase de la jerarquía también contiene un objeto miembro de clase **Característica**. Se verá que independientemente de si se llama a los finalizadores de la clase base, siempre se finalizan los objetos miembros **Característica**.

Cada **finalize( )** superpuesto debe tener acceso, al menos a los miembros **protegidos** puesto que el método **finalize( )** de la clase **Object** es **protegido** y el compilador no permitirá reducir el acceso durante la herencia. ("Amistoso" es menos accesible que **protegido.**)

En **Rana.main( )**, se configura el indicador de **HacerFinalizacionBase** y se crea un único objeto **Rana**. Recuerde que el recolector de basura —y en particular la finalización— podrían no darse para algún objeto en particular, por lo que para fortalecer la limpieza, la llamada a **System.gc( )** dispara el recolector de basura, y por consiguiente, la finalización. Sin la finalización de la clase base, la salida es:

```
No finalizando las bases
Creando Caracteristica esta vivo
CriaturaViviente()
Creando Caracteristica tiene corazon
Animal()
Creando Caracteristica puede vivir en el agua
Anfibio()
Rana()
¡Adios!
finalizando Rana
finalizando Caracteristica esta vivo
finalizando Caracteristica tiene corazon
finalizando Caracteristica puede vivir en el agua
```

Se puede ver que, sin duda, no se llama a los finalizadores para las clases base de **Rana** (los miembros objeto *son* finalizados, como se esperaba). Pero si se añade el parámetro "finalizar" en la línea de comandos, se tiene:

```
Creando Caracteristica esta vivo
CriaturaViviente()
Creando Caracteristica tiene corazon
Animal()
Creando Caracteristica puede vivir en el agua
Anfibio()
Rana()
¡Adios!
finalizando Rana
finalizando Anfibio
finalizando CriaturaViviente
Finalizando Caracteristica esta vivo
```

```
finalizando Caracteristica tiene corazon
finalizando Caracteristica puede vivir en el agua
```

Aunque el orden en que finalizan los objetos miembro es el mismo que el de su creación, técnicamente el orden de finalización de los objetos no se especifica. Con las clases base, sin embargo, se tiene control sobre el orden de finalización. El mejor a usar es el que se muestra aquí, que es el inverso al orden de inicialización. Siguiendo la forma que se usa en los destructores de C++, se debería hacer primero la finalización de la clase derivada, y después la finalización de la clase base. Esto es porque la finalización de la clase derivada podría llamar a varios métodos de la clase base que requieran que los componentes de la clase base sigan vivos, por lo que no hay que destruirlos prematuramente.

Comportamiento de métodos polimórficos dentro de constructores

La jerarquía de llamadas a constructores presenta un interesante dilema. ¿Qué ocurre si uno está dentro de un constructor y se invoca a un método de ligadura dinámica del objeto que se está construyendo? Dentro de un método ordinario se puede imaginar lo que ocurrirá —la llamada que conlleva una ligadura dinámica se resuelve en tiempo de ejecución, pues el objeto no puede saber si pertenece a la clase dentro de la que está el método o a alguna clase derivada de ésta. Por consistencia, se podría pensar que esto es lo que debería pasar dentro de los constructores.

Éste no es exactamente el caso. Si se invoca a un método de ligadura dinámica dentro de un constructor, se utiliza la definición superpuesta de ese método. Sin embargo, el *efecto* puede ser bastante inesperado, y puede ocultar errores difíciles de encontrar.

Conceptualmente, el trabajo del constructor es crear el objeto (lo que es casi una proeza). Dentro de cualquier constructor, el objeto entero podría formarse sólo parcialmente —sólo se puede saber que se han inicializado los objetos de clase base, pero no se puede saber qué clases se heredan. Una llamada a un método de ligadura dinámica, sin embargo, se "sale" de la jerarquía de herencias. Llama a un método de una clase derivada. Si se hace esto dentro del constructor, se llama a un método que podría manipular miembros que no han sido aún inicializados —lo que ocasionará problemas.

Se puede ver este problema en el siguiente ejemplo:

```
//: c07:ConstructoresMultiples.java
// Los constructores y el polimorfismo
// no producen lo que cabría esperar.

abstract class Grafica {
  abstract void dibujo();
  Grafica() {
    System.out.println("Grafica() antes de dibujar()");
    dibujar();
    System.out.println("Grafica() despues de dibujar()");
```

```
    }
}

class GraficaCircular extends Grafica {
    int radio = 1;
    GraficaCircular(int r) {
        radio = r;
        System.out.println(
            "GraficaCircular.GraficaCircular(), radio = "
            + radio);
    }
    void dibujar() {
        System.out.println(
            "GraficaCircular.dibujar(), radio = " + radio);
    }
}

public class PoliConstructors {
    public static void main(String[] args) {
        new GraficaCircular(5);
    }
} ///:~
```

En **Gráfica**, el método **dibujar( )** es **abstracto**, pero está diseñado para ser superpuesto. Sin duda, uno se ve forzado a superponerlo en **GraficaCircular**. Pero el constructor **Gráfica** llama a este método, y la llamada acaba en **GraficaCircular.dibujar( )**, que podría parecer ser lo pretendido. Pero, veamos la salida:

```
Grafica() antes de dibujar()
GraficaCircular.dibujar(), radio = 0
Grafica() despues de dibujar()
GraficaCircular.GraficaCircular(), radio = 5
```

Cuando el constructor **Gráfica( )** llama a **dibujar( )**, el valor de **radio** ni siquiera es el valor inicial por defecto 1. Es 0. Esto podría provocar que se dibuje un punto en la pantalla, dejándole a uno atónito, tratando de averiguar por qué el programa no funciona.

El orden de inicialización descrito en la sección previa no es del todo completo, y ahí está la clave para solucionar el misterio. De hecho, el proceso de inicialización es:

1. Se inicializa el espacio de almacenamiento asignado al objeto a ceros binarios antes de que ocurra nada más.

2. Se invoca a los constructores de clase base como se describió previamente. En este momento, se invoca al método **dibujar( )** superpuesto (sí, *antes* de que se invoque al constructor **GraficaCircular**) que descubre un valor de cero para **radio**, debido al punto 1.

3. Se llama a los inicializadores de los miembros en el orden de declaración.
4. Se invoca al cuerpo del constructor de la clase derivada.

Hay algo positivo en todo esto, y es que todo se inicializa al menos a cero (o lo que cero sea para cada tipo de datos en particular) y no se deja simplemente como si fuera basura. Esto incluye referencias a objetos empotradas dentro de clases a través de composición, que se convertirán en **null**. Por tanto, si se olvida inicializar esa referencia, se logrará una referencia en tiempo de ejecución. Todo lo demás se pone a cero, lo que generalmente es un valor revelador al estudiar la salida.

Por otro lado, uno podría acabar horrorizado al ver la salida de su programa. Se ha hecho algo perfectamente lógico, sin quejas por parte del compilador, y sin embargo el comportamiento es misteriosamente erróneo. (C++ produce un comportamiento bastante más racional en esta situación.) Fallos como éste podrían quedar fácilmente enterrados y llevaría mucho tiempo descubrirlos.

Como resultado, una buena guía para los constructores sería, "Haz lo menos posible para dejar el objeto en un buen estado, y en la medida de lo posible, no llames a ningún método". Los únicos métodos seguros a los que se puede llamar dentro de un constructor son aquéllos que sean **constantes** dentro de la clase base. (Esto también se aplica a métodos **privados**, que son automáticamente **constantes**). Éstos no pueden ser superpuestos, y por consiguiente, no pueden producir este tipo de sorpresa.

Diseño con herencia

Una vez que se ha aprendido lo relativo al polimorfismo, puede parecer que todo debería ser heredado, siendo como es el polimorfismo una herramienta tan inteligente. Esto puede cargar los diseños; de hecho, si se elige la herencia en primer lugar cuando se esté usando una clase para construir otra nueva, las cosas pueden volverse innecesariamente complicadas.

Un mejor enfoque es elegir primero la composición, cuando no es obvio qué es lo que debería usarse. La composición no fuerza un diseño en una jerarquía de herencias. Y además, es más flexible dado que es posible elegir dinámicamente un tipo (y por tanto, un comportamiento) al usar la composición, mientras que la herencia requiere conocer un tipo exacto en tiempo de compilación. El ejemplo siguiente lo ilustra:

```
//: c07:Transformar.java
// Cambia dinámicamente el comportamiento de
// un objeto a través de la composición.

abstract class Actor {
  abstract void actuar();
}

class ActorFeliz extends Actor {
  public void actuar() {
    System.out.println("ActorFeliz");
  }
```

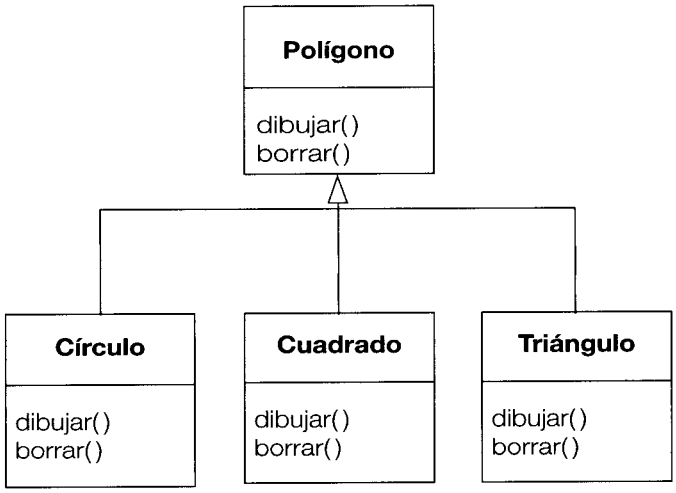
```
}

class ActorTriste extends Actor {
  public void actuar() {
    System.out.println("ActorTriste");
  }
}

class Escenario {
  Actor a = new ActorFeliz();
  void cambiar() { a = new ActorTriste(); }
  void ir() { a.actuar(); }
}

public class Transformar {
  public static void main(String[] args) {
    Escenario s = new Escenario();
    s.ir(); // Imprime "ActorFeliz"
    s.cambiar();
    s.cambiar(); // Imprime "ActorTriste"
  }
} ///:~
```

Un objeto **Escenario** contiene una referencia a un **Actor**, que se inicializa a un objeto **ActorFeliz**. Esto significa que **ir( )** produce un comportamiento particular. Pero dado que se puede reasignar una referencia a un objeto distinto en tiempo de ejecución, en escenario **a** puede sustituirse por una referencia a un objeto **ActorTriste**, con lo que cambia el comportamiento producido por **ir( )**. Por tanto, se gana flexibilidad dinámica en tiempo de ejecución. (A esto también se le llama el *Patrón Estado*. Véase *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*). Por contra, no se puede decidir heredar de forma distinta en tiempo de ejecución; eso debe determinarse completamente en tiempo de compilación.

Una guía general es "Utilice la herencia para expresar diferencias de comportamiento, y campos para expresar variaciones de estado". En el ejemplo de arriba, se usan ambos: se heredan dos clases distintas para expresar la diferencia en el método **actuar( )**, y **Escenario** usa la composición para permitir que varíe su estado. En este caso, ese cambio de estado viene a producir un cambio de comportamiento.

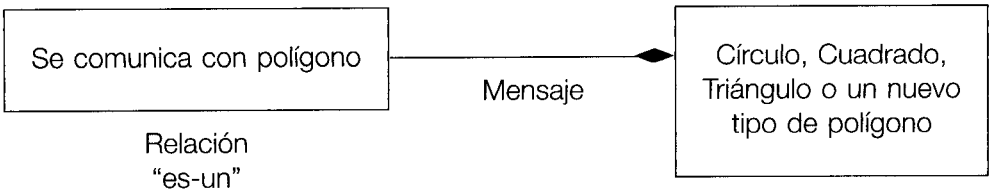
Herencia pura frente a extensión

Cuando se estudia la herencia, podría parecer que la forma más limpia de crear una jerarquía de herencias es seguir el enfoque "puro". Es decir, sólo se pueden superponer en la clase derivada los métodos que se han establecido en la clase base o la **interfaz**, como se muestra en este diagrama:



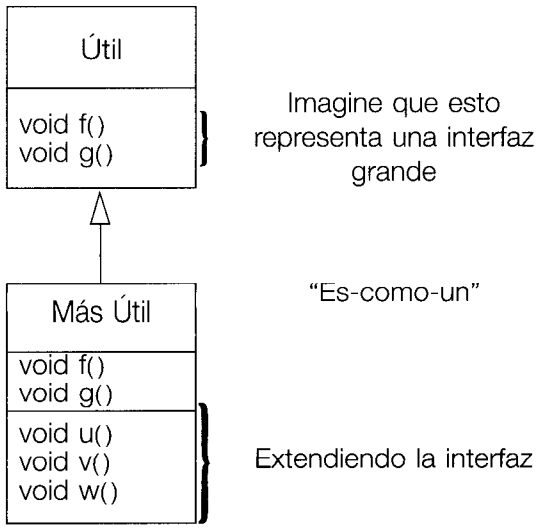
Se podría decir que ésta es una relación "es-un" pura porque el interfaz de la clase establece qué es. La herencia garantiza que cualquier clase derivada tendrá la interfaz de la clase base. Si se sigue el diagrama de arriba, las clases derivadas tampoco tendrán *nada más* que la interfaz de la clase base.

Podría pensarse que esto es una *sustitución pura*, porque los objetos de la clase derivada pueden sustituir perfectamente a la clase base, no siendo necesario conocer en estos casos ninguna información extra de las subclases cuando éstas se usan.

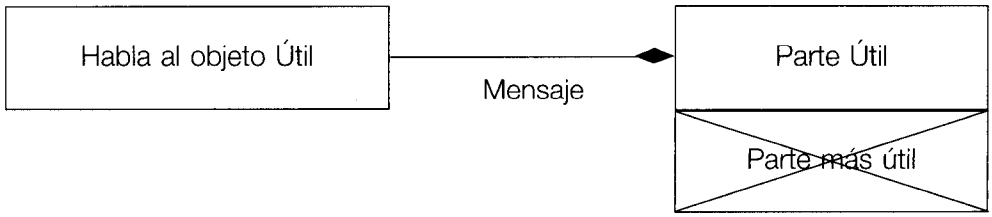


Es decir, la clase base puede recibir cualquier mensaje que se pueda enviar a la clase derivada porque ambas tienen exactamente la misma interfaz. Todo lo que se necesita es hacer una conversión hacia arriba desde la clase derivada y nunca volver a mirar con qué tipo exacto de objeto se está tratando. Todo se maneja mediante el polimorfismo.

Cuando se ve esto así, parece que una relación "es-un" pura es la única manera sensata de hacer las cosas, y cualquier otro diseño indica pensamiento desordenado y es por definición, un problema. Esto también es una trampa. En cuanto se empieza a pensar así, uno llega a descubrir que extender la interfaz (a lo que desafortunadamente parece animar la palabra clave **extends**) es la solución perfecta a un problema particular. Esto podría denominarse relación "es-como-un" porque la clase derivada es *como* la clase base —tiene la misma interfaz fundamental— pero tiene otros aspectos que requieren la implementación de métodos adicionales:



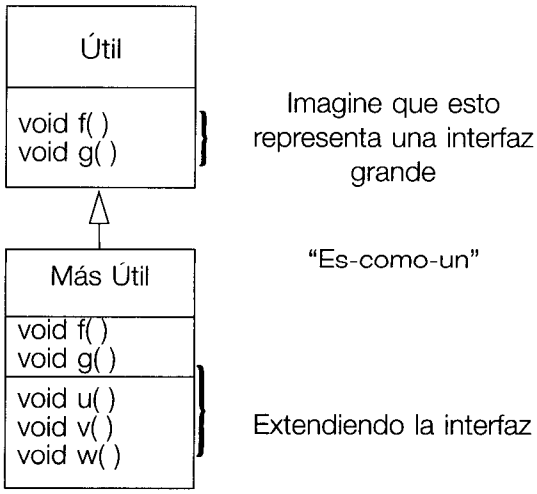
Mientras éste es también un enfoque sensato (dependiendo de la situación) tiene su desventaja. La parte extendida de la interfaz de la clase derivada no está disponible desde la clase base, de forma que una vez que se hace la conversión hacia arriba, no se puede invocar a los nuevos métodos:



Si en este caso no se está haciendo una conversión hacia arriba, no importa, pero a menudo nos meteremos en una situación en la que son necesario redescubrir el tipo exacto del objeto de forma que se pueda acceder a los métodos extendidos de ese tipo. La sección siguiente muestra cómo se ha hecho esto.

Conversión hacia abajo e identificación de tipos en tiempo de ejecución

Dado que a través de una conversión *hacia arriba* se pierde información específica de tipos (al moverse hacia arriba por la jerarquía), tiene sentido hacer una conversión hacia abajo si se quiere recuperar la información de tipos —es decir, moverse de nuevo *hacia abajo* por la jerarquía. Sin embargo, se sabe que una conversión hacia arriba es siempre segura; la clase base no puede tener una interfaz mayor que la de la clase derivada, por consiguiente, se garantiza que todo mensaje que se envíe a través de la interfaz de la clase base sea aceptado. Pero con una conversión hacia abajo, verdaderamente no se sabe que un polígono (por ejemplo) es, de hecho, un círculo. En vez de esto, podría ser un triángulo, un cuadrado o cualquier otro tipo.



Para solucionar este problema, debe haber alguna manera de garantizar que una conversión hacia abajo sea correcta, de forma que no se hará una conversión accidental a un tipo erróneo, para después enviar un mensaje que el objeto no pueda aceptar. Esto sería bastante inseguro.

En algunos lenguajes (como C++) hay que llevar a cabo una operación especial para conseguir una conversión hacia abajo segura en lo que a tipos se refiere, pero en Java ¡se comprueban *todas las conversiones*! Así, aunque parece que se está llevando a cabo una conversión ordinaria entre paréntesis, en tiempo de ejecución se comprueba esta conversión para asegurar que, de hecho, es del tipo que se cree que es. Si no lo es, se obtiene una **ClassCastException**. A esta comprobación de tipos en tiempo de ejecución se le denomina *identificación de tipos en tiempo de ejecución*[2]. El ejemplo siguiente demuestra el comportamiento de esta identificación de tipos:

```
//: c07:ITTE.java
// Conversión hacia abajo e Identificación de tipos
// en Tiempo de ejecución (ITTE)
import java.util.*;

class Util {
  public void f() {}
  public void g() {}
}

class MasUtil extends Util {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
```

[2] N. de T.: RTTI: Run-Time Type Identification.

```
}

public class ITTE {
  public static void main(String[] args) {
    Util[] x = {
      new Util(),
      new MasUtil()
    };
    x[0].f();
    x[1].g();
    // Tiempo de compilación: método no encontrado en útil:
    //! x[1].u();
    ((MasUtil)x[1]).u(); // Conversión hacia abajo/ITTE
    ((MasUtil)x[0]).u(); // Se lanza una Excepción
  }
} ///:~
```

Como en el diagrama, **MasUtil** extiende la interfaz de **Util**. Pero dado que es heredada, también puede tener una conversión hacia arriba hasta **Util**. Se puede ver cómo ocurre esto en la inicialización del array **x** en **main( )**. Dado que ambos objetos del array son de clase **Util**, se pueden enviar los métodos **f( )** y **g( )** a ambos, y si se intenta llamar a **u( )** (que sólo existe en **MasUtil**) se obtendrá un mensaje de error de tiempo de compilación.

Si se desea acceder a la interfaz extendida de un objeto **MasUtil**, se puede intentar hacer una conversión hacia abajo. Si es del tipo correcto, tendrá éxito. En caso contrario, se obtendrá una **ClassCastException**. No es necesario escribir ningún código extra para esta excepción, dado que indica un error del programador que podría ocurrir en cualquier lugar del programa.

Hay más que una simple conversión en la identificación de tipos en tiempo de ejecución. Por ejemplo, hay una forma de ver con qué tipo se está tratando *antes* de intentar hacer una conversión hacia abajo. Todo el Capítulo 12 está dedicado al estudio de distintos aspectos de la identificación de tipos en tiempo de ejecución de Java.

Resumen

Polimorfismo quiere decir "formas diferentes". En la programación orientada a objetos se tiene un mismo rostro (la interfaz común de la clase base) y distintas formas de usar ese rostro: las diferentes versiones de los métodos de la ligadura dinámica.

Hemos visto en este capítulo que es imposible entender, o incluso crear, un ejemplo de polimorfismo sin utilizar abstracción de datos y herencia. El polimorfismo es una faceta que no se puede ver aislada (como sí se podría hacer una sentencia **switch**, por ejemplo), pero sin embargo, funciona sólo dentro de un contexto, como parte de la "gran figura" que conforman las relaciones de clases. Las personas suelen confundirse con otras características de Java no orientadas a objetos, como la

sobrecarga de métodos, que en ocasiones se presenta como orientada a objetos. No se dejen engañar: si no hay ligadura tardía, no hay polimorfismo.

Para usar polimorfismo —y por consiguiente, técnicas de orientación a objetos— de manera efectiva en los programas, hay que ampliar la visión que se tiene de la programación para que incluya no sólo a los miembros y mensajes de una clase individual, sino también a aquellos elementos comunes a distintas clases, y sus inter-relaciones. Aunque esto requiere de un esfuerzo significativo, merece la pena, pues los resultados son un desarrollo de programas más rápido, una mejor organización del código, programas ampliables y un mantenimiento más sencillo del código.

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. Añadir un nuevo método a la clase base de **Poligonos.java** que imprima un mensaje, pero no superponerlo en la clase derivada. Explicar lo que ocurre. Ahora superponerlo en una de las clases derivadas pero no en las otras. Ver qué ocurre. Finalmente, superponerlo en todas las clases derivadas.

2. Añadir un nuevo tipo de **Polígono** a **Poligonos.java** y verificar en el método **main( )** que el polimorfismo funciona para el nuevo tipo como lo hace para los viejos.

3. Cambiar **Musica3.java** de forma que el método **que( )** se convierta en el método **toString( )** del objeto raíz **Object**. Intentar imprimir los objetos **Instrumento** haciendo uso de **System.out.println( )** (sin hacer uso de ningún tipo de conversión).

4. Añadir un nuevo tipo de **Instrumento** a **Musica3.java** y verificar que el polimorfismo funciona para el nuevo tipo.

5. Modificar **Musica3.java** de forma que cree objetos **Instrumento** al azar de la misma manera que lo hace **Poligonos.java**.

6. Crear una jerarquía de herencia de **Roedor**: **Ratón, Jerbo**, **Hamster**, etc. En la clase base, proporcionar métodos comunes a todas las clases de tipo **Roedor**, y superponerlos en las clases derivadas para que lleven a cabo distintos comportamientos en función del tipo de **Roedor** específico. Crear un array de objetos de tipo **Roedor**, rellenarlo con tipos de **Roedor** diferentes, e invocar a los métodos de la clase base para ver qué pasa.

7. Modificar el Ejercicio 6 de forma que **Roedor** sea una clase **abstracta**. Convertir en **abstractos** todos los métodos de **Roedor** que sea posible.

8. Crear una clase **abstracta** sin incluir ningún método **abstracto**, y verificar que no se pueden crear instancias de esa clase.

9. Añadir una clase **Escabeche** a **Bocadillo.java**.

10. Modificar el Ejercicio 6, de forma que demuestre el orden de inicialización de las clases base y las clases derivadas. Ahora, añadir objetos miembros, tanto a la clase base, como a las derivadas, y mostrar el orden en que se da la inicialización durante su construcción.

11. Crear una jerarquía de herencia de 3 niveles. Cada clase de la jerarquía debería tener un método **finalize( )**, y debería llamar adecuadamente a la versión de **finalize( )** de la clase base. Demostrar que la jerarquía funciona adecuadamente.

12. Crear una clase base con dos métodos. En el primer método, llamar al segundo método. Heredar una clase y superponer el segundo método. Crear un objeto de la clase derivada, hacer una conversión hacia arriba de la misma al tipo base, e invocar al primer método. Explicar lo que ocurre.

13. Crear una clase base con un método **abstracto escribir( )** superpuesta en una clase derivada. La versión superpuesta del método imprime el valor de una variable **entera** definida en la clase derivada. En el momento de la definición de esta variable, darle un valor distinto de cero. En el constructor de la clase base, invocar a este método. En el método **main( )** crear un objeto del tipo derivado, y después llamar a su método **escribir( )**. Explicar los resultados.

14. Siguiendo el ejemplo de **Transformar.java**, crear una clase **Estrella** que contenga una referencia a **EstadosAlerta** que pueda indicar tres estados diferentes. Incluir métodos para cambiar los estados.

15. Crear una clase **abstracta** sin métodos. Derivar una clase y añadir un método. Crear un método **estático** que toma una referencia a la clase base, haga una conversión hacia abajo y llame al método. Demostrar que funciona utilizando el método **main( )**. Ahora poner la declaración **abstracta** del método en la clase base, eliminando por consiguiente la necesidad de la conversión hacia abajo.