8: Interfaces y clases internas

Los interfaces y las clases internas proporcionan formas más sofisticadas de organizar y controlar los objetos de un sistema.

C++, por ejemplo, no contiene estos mecanismos, aunque un programador inteligente podría simularlos. El hecho de que existan en Java indica que se consideraban lo suficientemente importantes como para proporcionarles soporte directo en forma de palabras clave del lenguaje.

En el Capítulo 7, se aprendió lo relativo a la palabra clave **abstract**, que permite crear uno o más métodos sin definición dentro de una clase —se proporciona parte de la interfaz sin proporcionar la implementación correspondiente, que será creada por sus descendientes. La palabra clave **interface** produce una clase completamente abstracta, que no tiene ningún tipo de implementación. Se aprenderá que una **interfaz** es más que una clase abstracta llevada al extremo, pues permite llevar a cabo una variación de la "herencia múltiple" de C++, creando una clase sobre la que se puede hacer conversión hacia arriba a más de un tipo base.

Al principio, las clases internas parecen como un simple mecanismo de ocultación de código: se ubican clases dentro de otras clases. Se aprenderá, sin embargo, que una clase interna hace más que esto —conoce y puede comunicarse con las clases que le rodean— y el tipo de código que se puede escribir con clases internas es más elegante y claro, aunque para la mayoría de personas constituye un concepto nuevo. Lleva algún tiempo habituarse al diseño haciendo uso de clases internas.

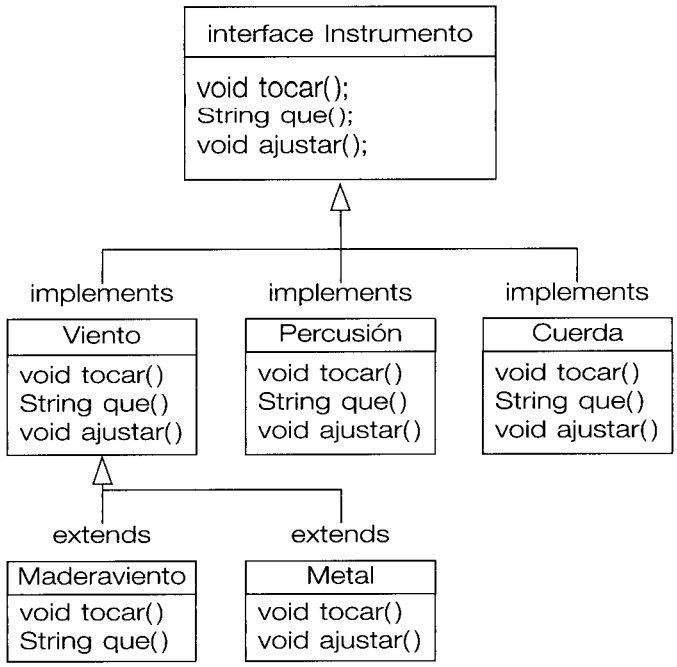
Interfaces

La palabra clave **interfaz** lleva el concepto de **abstracción** un paso más allá. Se podría pensar que es una clase **abstracta** "pura". Permite al creador establecer la forma de una clase: nombres de métodos, listas de parámetros, y tipos de retorno, pero no cuerpos de métodos. Una **interfaz** también puede contener campos, pero éstos son implícitamente **estáticos** y **constantes**. Una **interfaz** proporciona sólo la forma, pero no la implementación.

Una **interfaz** dice: "Ésta es la apariencia que tendrán todas las clases que implementen esta **interfaz**". Por consiguiente, cualquier código que use una **interfaz** particular sabe qué métodos deberían ser invocados por esa **interfaz**, y eso es todo. Por tanto se usa la **interfaz** para establecer un "protocolo" entre clases. (Algunos lenguajes de programación orientada a objetos tienen la palabra clave *protocolo* para hacer lo mismo.)

Para crear una **interfaz**, se usa la palabra clave **interface** en vez de la palabra clave **class**. Al igual que una clase, se le puede anteponer la palabra **public** a **interface** (pero sólo si esa **interfaz** se definió en un archivo con el mismo nombre), o dejar que se le dé el status de "amistoso" de forma que sólo se podrá usar dentro del mismo paquete.

Para hacer una clase que se ajuste a una **interfaz** particular (o a un grupo de **interfaces**), se usa la palabra clave **implements**. Se está diciendo "La **interfaz** contiene la apariencia, pero ahora voy a decir cómo *funciona*". Por lo demás, es como la herencia. El diagrama del ejemplo de los instrumentos lo muestra:



Una vez implementada una **interfaz**, esa implementación se convierte en una clase ordinaria que puede extenderse de forma normal.

Se puede elegir manifestar explícitamente las declaraciones de métodos de una **interfaz** como **pública**. Pero son **públicas** incluso si no se dice. Por tanto, cuando se **implementa** una **interfaz**, deben definirse como **públicos** los métodos de la **interfaz**. De otra forma, se pondrían por defecto a "amistoso", y se estaría reduciendo la accesibilidad a un método durante la herencia, lo que no permite el compilador de Java.

Se puede ver esto en la versión modificada del ejemplo **Instrumento**. Fíjese que todo método de la **interfaz** es estrictamente una declaración, que es lo único que permite el compilador. Además, ninguno de los métodos de **Instrumento** se declara como **público**, pero son **público** automáticamente:

```
//: c08:musica5:Musica5.java
// Interfaces.
import java.util.*;

interface Instrumento {
```

```
  // Tiempo de compilación constante:
  int i = 5; // estático y constante
  // No puede tener definiciones de métodos:
  void tocar(); // Automáticamente public
  String que();
  void ajustar();
}

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

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

class Cuerda implements Instrumento {
  public void tocar() {
    System.out.println("Cuerda.tocar()");
  }
  public Cuerda 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()");
  }
```

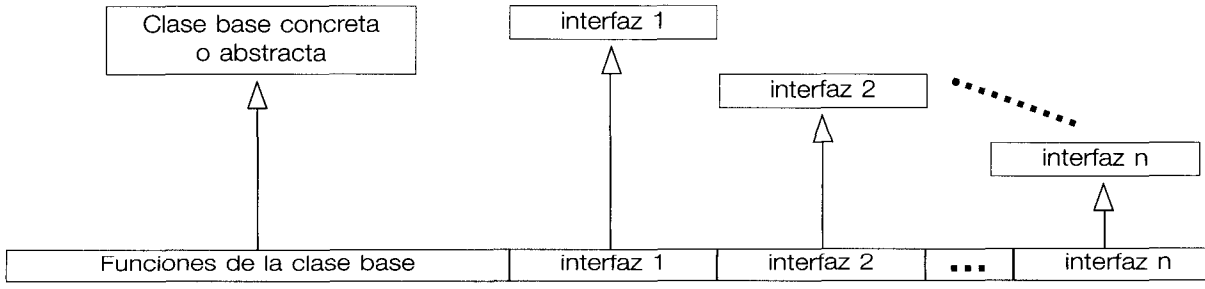
```
  public String que() { return "Maderaviento"; }
}

public class Musica5 {
  // 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;
    // Haciendo conversión hacia arriba durante la 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);
  }
} ///:~
```

El resto del código funciona igual. No importa si se está haciendo un conversión hacia arriba a una clase "regular" llamada **Instrumento**, una clase **abstracta** llamada **Instrumento**, o a una **interfaz** denominada **Instrumento**. El comportamiento es el mismo. De hecho, se puede ver en el método **afinar( )** que no hay ninguna prueba de que **Instrumento** sea una clase "regular", una clase **abstracta** o una **interfaz**. Ésta es la intención: cada enfoque da al programador un control distinto sobre la forma de crear y utilizar los objetos.

"Herencia múltiple" en Java

La **interfaz** no es sólo una forma "más pura" de clase **abstracta**. Tiene un propósito mayor. Dado que una **interfaz** no tiene implementación alguna —es decir, no hay espacio de almacenamiento asociado con una **interfaz**— no hay nada que evite que se combinen varias **interfaces**. Esto es muy valioso, pues hay veces en las que es necesario decir "una **x** es una **a** *y* una **b** *y* una **c**". En C++, a este acto de combinar múltiples interfaces de clases se le denomina *herencia múltiple*, y porta un equipaje bastante pegajoso porque puede que cada clase tenga una implementación. En Java, se puede hacer lo mismo, pero sólo una de las clases puede tener una implementación, por lo que los problemas de C++ no ocurren en Java al combinar múltiples interfaces:



En una clase derivada, no hay obligación de tener una clase base que puede ser **abstracta** o "concreta" (aquélla sin métodos **abstractos**). Si se *hereda* desde una no-**interfaz**, se puede heredar sólo de una. Todo el resto de elementos base deben ser **interfaces**. Se colocan todos los nombres de interfaz después de la palabra clave **implements**, separados por comas. Se pueden tener tantas **interfaces** como se desee —cada uno se convierte en un tipo independiente al que se puede hacer conversión hacia arriba. El ejemplo siguiente muestra una clase concreta combinada con varias **interfaces** para producir una nueva clase:

```
//: c08:Aventura.java
// Múltiples interfaces.
import java.util.*;

interface PuedeLuchar {
  void luchar();
}

interface PuedeNadar {
  void nadar();
}

interface PuedeVolar {
  void volar();
}

class PersonajeDeAccion {
  public void luchar() {}
}

class Heroe extends PersonajeDeAccion
    implements PuedeLuchar, PuedeNadar, PuedeVolar {
  public void nadar() {}
  public void volar() {}
}

public class Aventura {
```

```
  static void t(PuedeLuchar x) { x.luchar(); }
  static void u(PuedeNadar x) { x.nadar(); }
  static void v(PuedeVolar x) { x.volar(); }
  static void w(PersonajeDeAccion x) { x.luchar(); }
  public static void main(String[] args) {
    Heroe h = new Heroe();
    t(h); // Tratarlo como un PuedeLuchar
    u(h); // Tratarlo como un PuedeNadar
    v(h); // Tratarlo como un PuedeVolar
    w(h); // Tratarlo como un PersonajeDeAccion
  }
} ///:~
```

Se puede ver que **Héroe** combina la clase concreta **PersonajeDeAccion** con las interfaces **PuedeLuchar**, **PuedeNadar** y **PuedeVolar**. Cuando se combina una clase concreta con interfaces de esta manera, hay que poner primero la clase concreta, y después las interfaces. (Sino, el compilador dará error.)

Fíjese que la sintaxis del método **luchar()** es la misma en la **interfaz PuedeLuchar** y en la clase **PersonajeDeAccion**, y que *no* hay ninguna definición para **luchar()** en **Héroe**. La regla de una **interfaz** es que se puede heredar de ella (como se verá en breve) pero se obtiene otra **interfaz**. Si se desea crear un objeto del nuevo tipo, éste debe ser una clase a la que se proporcionen todas sus definiciones. Incluso aunque **Héroe** no proporciona explícitamente una definición para **luchar( )**, ésta viene junto con **PersonajeDeAccion**, por lo que ésta se proporciona automáticamente y es posible crear objetos de **Heroe**.

En la clase **Aventura**, se puede ver que hay cuatro métodos que toman como parámetros las distintas interfaces y la clase concreta. Cuando se crea un objeto **Héroe**, se le puede pasar a cualquier de estos métodos, lo que significa que se le está haciendo una conversión hacia arriba a cada **interfaz**. Debido a la forma de diseñar las interfaces en Java, esto funciona sin problemas ni esfuerzos por parte del programador.

Recuérdese que la razón principal de las interfaces se muestra en el ejemplo de arriba: poder hacer una conversión hacia arriba a más de un tipo base. Sin embargo, una segunda razón para el uso de interfaces es la misma que se da al usar una clase **abstracta**: evitar que el programador cliente haga objetos de esta clase y hacer que ésta no sea más que una interfaz. Esto provoca una pregunta: ¿debería usarse una **interfaz** o una clase **abstracta**? Una **interfaz** proporciona los beneficios de una clase **abstracta** *y* los beneficios de una **interfaz**, por lo que si es posible crear la clase base sin definiciones de métodos o variables miembro, siempre se debería preferir las **interfaces** a las clases **abstractas**. De hecho, si se sabe que algo va a ser una clase base, la primera opción debería ser convertirla en **interfaz**, y sólo si uno se ve forzado a tener definiciones de métodos o variables miembros habrá que cambiar a una clase **abstracta**, o si fuera necesario una clase concreta.

Colisiones de nombre al combinar interfaces

Se puede encontrar una pequeña pega al implementar múltiples interfaces. En el ejemplo de arriba, tanto **PuedeLuchar** como **PersonajeDeAccion** tienen un método **void luchar()** idéntico. Esto no

es un problema al ser el método idéntico en ambos casos pero, ¿qué ocurre si no es así? He aquí un ejemplo:

```
//: c08:ColisionInterfaces.java
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // sobrecargado
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // sobrecargado
}

class C4 extends C implements I3 {
  // Idéntica, sin problemas:
  public int f() { return 1; }
}

// Los métodos sólo difieren en el tipo de retorno:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

La dificultad se da porque se mezclan la sobrecarga, la implementación y la superposición, y las funciones sobrecargadas no pueden diferir sólo en el valor de retorno. Cuando se quita el comentario de las dos últimas líneas, los mensajes de error dicen:

```
ColisionInterfaces.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found     : int
required: void
ColisionInterfaces.java:24: interfaces I3 and I1 are
incompatible; both define f
(), but with different return type
```

Utilizando los mismos nombres de método en interfaces diferentes que se pretende combinar, suele causar también confusión en la legibilidad del código. Hay que tratar de evitarlo.

Extender una interfaz con herencia

Se pueden añadir nuevas declaraciones de método a una **interfaz** haciendo uso de la herencia, y también se pueden combinar varias **interfaces** en una nueva **interfaz** gracias a la herencia. En ambos casos se consigue una nueva **interfaz**, como se ve en el ejemplo siguiente:

```
//: c08:EspectaculoDeMiedo.java
// Extendiendo una interfaz con herencia.

interface Monstruo {
  void amenaza();
}

interface MonstruoPeligroso extends Monstruo {
  void destruir();
}

interface Letal {
  void matar();
}

class Dragon implements MonstruoPeligroso {
  public void amenaza() {}
  public void destruir() {}
}

interface Vampiro
    extends MonstruoPeligroso, Letal {
  void beberSangre();
}

class EspectaculoDeMiedo {
  static void u(Monstruo b) { b.amenaza(); }
  static void v(MonstruoPeligroso d) {
    d.amenaza();
    d.destuir();
  }
  public static void main(String[] args) {
    Dragon if2 = new Dragon();
    u(if2);
    v(if2);
  }
} ///:~
```

MonstruoPeligroso es una simple extension de **Monstruo** que produce una nueva **interfaz**. Éste se implementa en **Dragon**.

La sintaxis utilizada en **Vampiro** *sólo* funciona cuando se heredan interfaces. Normalmente, se puede usar **herencia** sólo con una única clase, pero dado que una **interfaz** puede estar hecha de otras múltiples interfaces, **extends** puede referirse a múltiples interfaces a base al construir una nueva **interfaz**. Como se puede ver, los nombres de **interfaz** simplemente se separan con comas.

Constantes de agrupamiento

Dado que cualquier campo que se ponga en una interfaz se convierte automáticamente en **estático** y **constante**, la **interface** es una herramienta conveniente para la creación de grupos de valores constantes, en gran medida al igual que se haría con un **enumerado** en C o C++. Por ejemplo:

```
//: c08:Meses.java
// Utilizando interfaces para crear grupos de constantes.
package c08;

public interface Meses {
  int
    ENERO = 1, FEBRERO = 2, MARZO = 3,
    ABRIL = 4, MAYO = 5, JUNIO = 6, JULIO = 7,
    AGOSTO = 8, SEPTIEMBRE = 9, OCTUBRE = 10,
    NOVIEMBRE = 11, DICIEMBRE = 12;
} ///:~
```

Fíjese que el estilo de Java de usar en todo letras mayúsculas (con guiones bajos para separar múltiples palabras dentro de un único identificador) para datos **estáticos** y **constantes** que tienen inicializadores constantes.

Los campos de una **interfaz** son automáticamente **públicos**, por lo que no es necesario especificarlo.

Ahora se pueden usar constantes de fuera del paquete, importándolas de **c08.*** o **c08.Meses** justo como se haría con cualquier otro paquete, y hacer referencia a los valores de expresiones como **Mes.ENERO**. Por supuesto, lo que se consigue es simplemente un **entero**, por lo que no existe la seguridad de tipos extra que tiene el **enumerado** de C++, pero esta técnica (comúnmente usada) es verdaderamente una mejora sobre la codificación ardua de números en un programa. (A este enfoque se le suele denominar cómo hacer uso de "números mágicos" y produce código muy difícil de mantener.)

Si se desea seguridad extra con los tipos, se puede construir una clase como[1]:

```
//: c08:Mes2.java
// Un sistema de enumeracion mas robusto.
package c08;

public final class Mes2 {
  private String nombre;
```

[1] Este enfoque se basa en un e-mail que me envió Rich Hoffarth.

```
  private Mes2(String nm) { nombre = nm; }
  public String toString() { return nombre; }
  public final static Mes2
    ENE = new Mes2("Enero"),
    FEB = new Mes2("Febrero"),
    MAR = new Mes2("Marzo"),
    ABR = new Mes2("Abril"),
    MAY = new Mes2("Mayo"),
    JUN = new Mes2("Junio"),
    JUL = new Mes2("Julio"),
    AGO = new Mes2("Agosto"),
    SEP = new Mes2("Septiembre"),
    OCT = new Mes2("Octubre"),
    NOV = new Mes2("Noviembre"),
    DIC = new Mes2("Diciembre");
  public final static Mes2[] mes = {
    ENE, ENE, FEB, MAR, ABR, MAY, JUN,
    JUL, AGO, SEP, OCT, NOV, DIC
  };
  public static void main(String[] args) {
    Mes2 m = Mes2.ENE;
    System.out.println(m);
    m = Mes2.mes[12];
    System.out.println(m);
    System.out.println(m == Mes2.DIC);
    System.out.println(m.equals(Mes2.DIC));
  }
} ///:~
```

La clase se llama **Mes2**, dado que ya hay una clase **Mes** (Month) en la biblioteca estándar Java. Es una clase **constante** con un constructor **privado** por lo que nadie puede heredar de la misma o hacer instancias de ella. Las únicas instancias son las **constante estáticas** creadas en la propia clase: **ENE, FEB, MAR**, etc. Estos objetos también pueden usarse en el array **mes**, que permite elegir los números por número en vez de por nombre. (Fíjese que el **ENE** extra del array proporciona un desplazamiento de uno, de forma que diciembre sea el mes número 12.) En el método **main( )** se puede ver la seguridad de tipos: **m** es un objeto **Mes2**, por lo que puede ser asignado sólo a **Mes2**. El ejemplo anterior **Meses.java** sólo proporcionaba valores **enteros**, por lo que una variable **entera** implementada con el fin de representar un mes, podría recibir cualquier valor entero, lo que no sería muy seguro.

Este enfoque también permite usar == o **equals( )** indistintamente, como se muestra al final del método **main( )**.

Inicializando atributos en interfaces

Los atributos definidos en las interfaces son automáticamente **estáticos** y **constantes**. Éstos no pueden ser "constantes blancas", pero pueden inicializarse con expresiones no constantes. Por ejemplo:

```
//: c08:ValoresAleatorios.java
// Inicializando atributos de interfaz con
// inicializadores no constantes.
import java.util.*;

public interface ValoresAleatorios {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

Dado que los campos son **estáticos**, se inicializan cuando se carga la clase por primera vez, lo que ocurre cuando se accede a cualquiera de los atributos por primera vez. He aquí una simple prueba:

```
//: c08:PruebaValoresAleatorios.java
public class PruebaValoresAleatorios {
  public static void main(String[] args) {
    System.out.println(PruebaValoresAleatorios.rint);
    System.out.println(PruebaValoresAleatorios.rlong);
    System.out.println(PruebaValoresAleatorios.rfloat);
    System.out.println(PruebaValoresAleatorios.rdouble);
  }
} ///:~
```

Los atributos, por supuesto, no son parte de la interfaz, pero se almacenan, sin embargo, en el área de almacenamiento **estático** de esa interfaz.

Interfaces anidados

Se pueden anidar interfaces dentro de clases y dentro de otras interfaces. Esto revela un número de aspectos muy interesantes[2]:

```
//: c08:InterfacesAnidadas.java
class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
```

[2] Gracias a Martin Danner por preguntar esto durante un seminario.

```
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void recibirD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // "public" es redundante:
  public interface H {
    void f();
  }
  void g();
  // No puede ser privado dentro de una interfaz
  //! private interface I {}
}

public class InterfacesAnidados {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
```

```
    public void f() {}
  }
  // No se puede implementar una interfaz anidada excepto que esté
  // dentro de la definición de una clase:
  //! class DImp implements A.D {
  //!   public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // No se puede acceder a A.D:
    //! A.D ad = a.obtenerD();
    // No devuelve nada más que A.D:
    //! A.DImp2 di2 = a.obtenerD();
    // No se puede acceder a un miembro de la interfaz:
    //! a.obtenerD().f();
    // Sólo otro A puede hacer algo con obtenerD():
    A a2 = new A();
    a2.recibirD(a.obtenerD());
  }
} ///:~
```

La sintaxis para anidar una interfaz dentro de una clase es razonadamente obvia, y al igual que con las interfaces no anidadas, éstas pueden tener visibilidad **pública** o "amistosa". También se puede ver que ambas interfaces anidadas **pública** y "amistosa" pueden implementarse como clases anidadas **pública**, "amistosa" y **privada**.

Como novedad, las interfaces también pueden ser **privadas** como se ve en **A.D** (se usa la misma sintaxis de calificaciones que en las clases anidadas).

¿Qué tiene de bueno una interfaz **pública** anidada? Se podría adivinar que sólo puede implementarse como una clase **privada** anidada como en **DImp**, pero **A.DImp2** muestra que también puede implementarse como una clase **pública**. Sin embargo, **A.DImp2** sólo puede ser usada como ella misma. No se permite mencionar el hecho de que implementa la interfaz **privada**, por lo que im-

plementar una interfaz **privada** es una manera de forzar la definición de métodos de esa interfaz sin añadir ninguna información de tipos (es decir, sin permitir conversiones hacia arriba).

El método **obtenerD( )** produce un dilema aún mayor en lo relativo a la interfaz **privada**: es un método **público** que devuelve una referencia a una interface **privada**. ¿Qué se puede hacer con el valor de retorno de este método? En el método **main( )**, se pueden ver varios intentos de usar el valor de retorno, pero todos en balde. Lo único que funciona es pasar el valor de retorno a un objeto que tenga permiso para usarlo —en este caso, otro **A**, a través del método **recibir( )**.

La interfaz E muestra que es posible anidar interfaces una dentro de la otra. Sin embargo, las reglas sobre las interfaces —en particular, que todos los elementos de una interfaz deban ser públicos— se vuelven en este caso muy estrictas, de forma que una interfaz anidada dentro de otra se convierte en **pública** automáticamente y no puede declararse como **privada**.

InterfacesAnidadas muestra las distintas maneras de implementar interfaces anidadas. En particular, fíjese que al implementar una interfaz, no es obligatorio implementar las interfaces que tenga anidadas. Tampoco las interfaces **privadas** pueden implementarse fuera de las clases en que se han definido.

Inicialmente, estas características pueden parecer añadidos para conseguir consistencia sintáctica, pero generalmente encontramos que una vez que se conocen, se descubren a menudo sitios en los que son útiles.

Clases internas

Es posible colocar una definición de clase dentro de otra definición de clase. A la primera se le denomina clase *interna*. Este tipo de clases son una característica valiosa, pues permite agrupar clases que lógicamente están relacionadas, además de controlar la visibilidad de una con la otra. Sin embargo, es importante entender que las clases internas son fundamentalmente distintas de la composición.

A menudo, al aprender clases internas, no se ve su necesidad inmediatamente. Al final de esta sección, una vez que se hayan descrito toda la sintaxis y semántica de las clases internas, se verán ejemplos que deberían aclarar los beneficios de estas clases.

Se crea una clase interna como uno esperaría —ubicando su definición dentro de una clase envolvente:

```
//: c08:Paquete1.java
// Creando clases internas.

public class Paquete1 {
  class Contenidos {
    private int i = 11;
    public int valor() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aDonde) {
      etiqueta = aDonde;
    }
```

```
    String leerEtiqueta() { return etiqueta; }
  }
  // Usar clases internas es igual que usar
  // otras clases, dentro de Paquete1:
  public void enviar(String dest) {
    Contenidos c = new Contenidos();
    Destino d = new Destino(dest);
    System.out.println(d.leerEtiqueta());
  }
  public static void main(String[] args) {
    Paquete1 p = new Paquete1();
    p.enviar("Tanzania");
  }
} ///:~
```

Las clases internas, cuando se usan dentro de **enviar()** tienen la misma apariencia que muchas otras clases. Aquí, la única diferencia práctica es que los nombres se anidan dentro de **Paquete1**. Se verá en breve que ésta no es la única diferencia.

Generalmente, la clase externa tendrá un método que devuelva una referencia a una clase interna, como ésta:

```
//: c08:Paquete2.java
// Devolviendo una referencia a una clase interna.

public class Paquete2 {
  class Contenidos {
    private int i = 11;
    public int valor() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aDonde) {
      etiqueta = aDonde;
    }
    String leerEtiqueta() { return etiqueta; }
  }
  public Destino para(String s) {
    return new Destino(s);
  }
  public Contenidos cont() {
    return new Contenidos();
  }
  public void enviar(String dest) {
    Contenidos c = cont();
    Destino d = para(dest);
    System.out.println(d.leerEtiqueta());
```

```
  }
  public static void main(String[] args) {
    Paquete2 p = new Paquete2();
    p.enviar("Tanzania");
    Paquete2 q = new Paquete2();
    // Definir referencias a clases internas:
    Paquete2.Contenidos c = q.cont();
    Paquete2.Destino d = q.para("Borneo");
  }
} ///:~
```

Si se desea hacer un objeto de la clase interna en cualquier sitio que no sea un método no **estático** de la clase externa, hay que especificar el tipo de ese objeto como *NombreClaseExterna.NombreClaseInterna*, como se ha visto en el método **main( )**.

Clases internas y conversiones hacia arriba

Hasta ahora, las clases no parecen excesivamente espectaculares. Después de todo, si uno pretende ocultar, Java ya tiene un buen mecanismo de ocultamiento —simplemente es necesario dejar que la clase sea "amistosa" (visible sólo dentro de un paquete) en vez de crearla como clase interna.

Sin embargo, las clases internas tienen su verdadera razón de ser al comenzar a hacer conversión hacia arriba hacia una clase base, y en particular a una **interfaz**. (El efecto de producir una referencia a una interfaz desde un objeto que lo implementa es esencialmente el mismo que hacer una conversión hacia una clase base.) Esto se debe a que la clase interna —la implementación de la **interfaz**— puede ocultarse completamente y no estará disponible para nadie, lo cual es adecuado para ocultar la implementación. Todo lo que se logra a cambio es una referencia a la clase base o a la **interfaz**.

En primer lugar, se definirán las interfaces en sus propios archivos de forma que puedan ser usados en todos los ejemplos:

```
//: c08:Dentro.java
public interface Dentro {
  String leerEtiqueta();
} ///:~


//: c08:Contenidos.java
public interface Contenidos {
  int valor();
} ///:~
```

Ahora **Contenidos** y **Dentro** representan las interfaces disponibles para el programador cliente. (La **interfaz**, recuérdese, convierte sus miembros en **públicos** automáticamente.)

Cuando se obtiene de vuelta una referencia a la clase base o a la **interfaz**, es posible que se pueda incluso averiguar el tipo exacto, como se muestra a continuación:

```
//: c08:Paquete3.java
// Devolviendo una referencia a una clase interna.

public class Paquete3 {
  private class PContenidos implements Contenido {
    private int i = 11;
    public int valor() { return i; }
  }
  protected class PDestino
      implements Destino {
    private String etiqueta;
    private PDestino(String aDonde) {
      etiqueta = aDonde;
    }
    public String leerEtiqueta() { return etiqueta; }
  }
  public Destino dest(String s) {
    return new PDestino(s);
  }
  public Contenidos cont() {
    return new PContenidos();
  }
}

class Prueba {
  public static void main(String[] args) {
    Paquete3 p = new Paquete3();
    Contenidos c = p.cont();
    Destino d = p.dest("Tanzania");
    // Ilegal -- no se puede acceder a la clase privada:
    //! Paquete3.PContenidos pc = p.new PContenidos();
  }
} ///:~
```

Fíjese que, dado que **main( )** está en **Prueba**, si se desea ejecutar este programa no hay que ejecutar **Paquete3**, sino:

```
java Prueba
```

En el ejemplo, **main( )** debe estar en una clase separada para demostrar la privacidad de la clase interna **PContenidos**.

En **Paquete3**, se ha añadido algo nuevo: la clase interna **PContenidos** es **privada** de forma que nadie sino **Paquete3** puede acceder a ella. **PDestino** es **protegido**, por lo que nadie sino **Paquete3**, las clases contenidas en el paquete **Paquete3** (dado que **protegido** también permite acceso a nivel de paquete (es decir, protegido también es "amistoso"), y los herederos de **Paquete3**

pueden acceder a **PDestino**. Esto significa que el programador cliente tiene conocimiento y acceso restringidos a estos miembros. De hecho, no se puede hacer una conversión hacia abajo a una clase interna **privada** (o a una clase **protegida** interna a menos que se sea un descendiente), dado que no se puede acceder al nombre, como se puede ver en la **clase Prueba**. Por consiguiente, la clase interna **protegida** proporciona una forma para que el diseñador evite cualquier dependencia de codificación de tipos y oculte los detalles sobre implementación. Además, la extensión de una **interfaz** es inútil desde el punto de vista del programador cliente, dado que éste no puede acceder a ningún método adicional que no sea parte de la **interfaz pública** de la clase. Esto también proporciona una oportunidad para que el compilador Java genere código más eficiente.

Las clases normales (no internas) no pueden ser **privadas** o **protegidas** —sino sólo **públicas** o "amistosas".

Ámbitos y clases internas en métodos

Lo que se ha visto hasta la fecha abarca el uso típico de las clases internas. En general, el código que se escriba y lea relativo a las clases internas será clases internas "planas", simples y fáciles de entender. Sin embargo, el diseño de las clases internas es bastante completo y hay otras formas oscuras de usarlas: las clases internas pueden crearse dentro de un método o incluso en un ámbito arbitrario. Hay dos razones para hacer esto:

1. Como se ha visto previamente, se está implementando una interfaz de algún tipo, de forma que se puede crear y devolver una referencia.
2. Se está resolviendo un problema complicado y se desea crear una clase que ayude en la solución, aunque no se desea que ésta esté públicamente disponible.

En los ejemplos siguientes, se modificará el código anterior para utilizar:

1. Una clase definida dentro de un método.
2. Una clase definida dentro del ámbito de un método.
3. Una clase anónima que implementa una interfaz.
4. Una clase anónima que extienda una clase que no tenga un constructor por defecto.
5. Una clase anónima que lleve a cabo la inicialización de campos.
6. Una clase anónima que lleve a cabo la construcción usando inicialización de instancias (las clases internas anónimas no pueden tener constructores).

Aunque es una clase ordinaria con una implementación, también se usa **Envoltorio** como una "interfaz" común a sus clases derivadas:

```
//: c08:Envoltorio.java
public class Envoltorio {
  private int i;
```

```
  public Envoltorio(int x) { i = x; }
  public int valor() { return i; }
} ///:~
```

Se verá que **Envoltorio** tiene un constructor que necesita un parámetro, para hacer las cosas un poco más interesantes.

El primer ejemplo muestra la creación de una clase entera dentro del ámbito de un método (en vez de en el ámbito de otra clase):

```
//: c08:Paquete4.java
// Anidando una clase dentro de un método.

public class Paquete4 {
  public Destino dest(String s) {
    class PDestino
        implements Destino {
      private String etiqueta;
      private PDestino(String aDonde) {
        etiqueta = aDonde;
      }
      public String leerEtiqueta() { return etiqueta; }
    }
    return new PDestino(s);
  }
  public static void main(String[] args) {
    Paquete4 p = new Paquete4();
    Destino d = p.dest("Tanzania");
  }
} ///:~
```

La clase **Pdestino** es parte de **dest( )** más que de **Paquete4**. (Fíjese también que se podría usar el identificador de clase **PDestino** para una clase interna dentro de cada clase del mismo subdirectorio sin que haya colisión de nombres.) Por consiguiente, **PDestino** no puede ser accedida fuera del método **dest( )**. Fíjese que la conversión hacia arriba se da en la sentencia de retorno —nada viene de fuera de **dest( )** excepto una referencia a **PDestino**, la clase base. Por supuesto, el hecho de que el nombre de la clase **PDestino** se ubique dentro de **dest( )** no quiere decir que **PDestino** no sea un objeto válido una vez que **dest( )** devuelva su valor.

El siguiente ejemplo muestra cómo se puede anidar una clase interna dentro de cualquier ámbito arbitrario:

```
//: c08:Paquete5.java
// Anidando una clase dentro de un ámbito.

public class Paquete5 {
  private void rastreoInterno(boolean b) {
```

```
    if(b) {
      class RealizarRastreo {
        private String id;
        RealizarRastreo(String s) {
          id = s;
        }
        String obtenerId() { return id; }
      }
      RealizarRastreo ts = new RealizarRastreo("slip");
      String s = ts.obtenerId();
    }
    // ¡No se puede usar aquí, fuera del rango!
    //! RealizarRastreo ts = new RealizarRastreo("x");
  }
  public void rastrear() { rastreoInterno(true); }
  public static void main(String[] args) {
    Paquete5 p = new Paquete5();
    p.rastrear();
  }
} ///:~
```

La clase **RealizarRastreo** está anidada en el ámbito de una sentencia **if**. Esto no significa que la clase se cree condicionalmente —se compila junto con todo lo demás. Sin embargo, no está disponible fuera del rango en el que se definió. Por lo demás, tiene exactamente la misma apariencia que una clase ordinaria.

Clases internas anónimas

El siguiente ejemplo parece un poco extraño:

```
//: c08:Paquete6.java
// Un método que devuelve una clase interna anónima.

public class Paquete6 {
  public Contenidos cont() {
    return new Contenidos() {
      private int i = 11;
      public int valor() { return i; }
    }; // En este caso es necesario el punto y coma
  }
  public static void main(String[] args) {
    Paquete6 p = new Paquete6();
    Contenidos c = p.cont();
  }
} ///:~
```

¡El método **cont( )** combina la creación del valor de retorno con la definición de la clase que representa ese valor de retorno! Además, la clase es anónima —no tiene nombre. Para empeorar aún más las cosas, parece como si se estuviera empezando a crear un objeto **Contenidos**:

```
return new Contenidos()
```

Pero entonces, antes del punto y coma, se dice: "Pero espera, creo que me convertiré en una definición de clase":

```
return new Contenidos() {
    private int i = 11;
    public int valor() { return i; }
};
```

Lo que esta sintaxis significa es: "Crea un objeto de una clase anónima heredada de **Contenidos**". A la referencia que devuelva la expresión **new** se le hace un conversión hacia arriba automáticamente para convertirla en una referencia a **Contenidos**. La sintaxis de clase interna anónima es una abreviación de:

```
Class MisContenidos implements Contenidos {
  private int i = 11;
  public int valor() { return i; }
}
return new MisContenidos();
```

En la clase interna anónima, se crea **Contenidos** utilizando un constructor por defecto. El código siguiente muestra qué hacer si la clase base necesita un constructor con un argumento:

```
//: c08:Paquete7.java
// Una clase interna anónima que llama al
// constructor de la clase base.

public class Paquete7 {
  public Envoltorio envolver(int x) {
    // Llamada al constructor base:
    return new Envoltorio(x) {
      public int valor() {
        return super.valor() * 47;
      }
    }; // Punto y coma obligatorio
  }
  public static void main(String[] args) {
    Paquete7 p = new Paquete7();
    Envoltura w = p.envolver(10);
  }
} ///:~
```

Es decir, simplemente se pasa el argumento adecuado al constructor de la clase base, en este caso, se pasa **x** en **new Envoltorio(x)**. Una clase anónima no puede tener un constructor donde normalmente se invocaría a **super( )**.

En los dos ejemplos anteriores, el punto y coma no delimita el final del cuerpo de la clase, (como en C++). En cambio, marca el final de la expresión que viene a contener la clase anónima. Por consiguiente, es idéntico al uso de un punto y coma en cualquier otro sitio.

¿Qué ocurre si se necesita llevar a cabo algún tipo de inicialización de algún objeto de una clase interna anónima? Dado que es anónima, no se puede dar ningún nombre al constructor —por lo que no se puede tener un constructor. Se puede, sin embargo, llevar a cabo inicializaciones en el momento de definición de los campos:

```
//: c08:Paquete8.java
// Una clase interna anónima que lleva a cabo
// una inicialización. Versión abreviada de
// Paquete5.java.

public class Paquete8 {
  // El argumento debe ser constante para usarse en una
  // clase anónima interna:
  public Destino dest(final String dest) {
    return new Destino() {
      private String etiqueta = dest;
      public String leerEtiqueta() { return etiqueta; }
    };
  }
  public static void main(String[] args) {
    Paquete8 p = new Paquete8();
    Destino d = p.dest("Tanzania");
  }
} ///:~
```

Si se está definiendo una clase anónima interna y se desea utilizar un objeto definido fuera de la clase interna anónima, el compilador exige que el objeto externo sea **constante**. Ésta es la razón por la que el argumento pasado a **dest( )** es **constante**. Si se olvida, se obtendrá un mensaje de error en tiempo de compilación.

Mientras sólo se esté asignando un campo, el enfoque de arriba está bien. Pero ¿qué ocurre si se desea llevar a cabo alguna actividad al estilo constructor? Con la *inicialización de instancias*, se puede, en efecto, crear un constructor para una clase anónima interna.

```
//: c08:Paquete9.java
// Utilizando "inicialización de instancias" para llevar a cabo
// la construcción de una clase interna anónima.

public class Paquete9 {
```

```
  public Destino
  dest(final String dest, final float precio) {
    return new Destino() {
      private int coste;
      // Inicialización de instancias para cada objeto:
      {
        coste = Math.round(precio);
        if(cost > 100)
          System.out.println("¡Por encima del presupuesto!");
      }
      private String etiqueta = dest;
      public String leerEtiqueta() { return etiqueta; }
    };
  }
  public static void main(String[] args) {
    Paquete9 p = new Paquete9();
    Destino d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

Dentro del inicializador de instancias, se puede ver el código que no podría ser ejecutado como parte de un inicializador de campos (es decir, la sentencia **if**). Por tanto, en efecto, un inicializador de instancias es el constructor de una clase interna anónima. Por supuesto, está limitado; no se pueden sobrecargar inicializadores de instancias, por lo que sólo se puede tener uno de estos constructores.

El enlace con la clase externa

Hasta ahora, parece que las clases internas son solamente una ocultación de nombre y un esquema de organización de código, lo cual ayuda pero no convence. Sin embargo, hay otra alternativa. Cuando se crea una clase interna, un objeto a esa clase interna tiene un enlace al objeto contenedor que la hizo, y así puede acceder a los miembros del objeto contenedor —*sin* restricciones especiales. Además, las clases internas tienen derechos de acceso a todos los elementos de la clase contenedor[3]. El ejemplo siguiente lo demuestra:

```
//: c08:Secuencia.java
// Tiene una secuencia de objetos.

interface Selector {
  boolean fin();
  Object actual();
  void siguiente();
}
```

[3] Este enfoque varía mucho del diseño de las *clases anidadas* en C++, en el que estas clases son simplemente un mecanismo de ocultación de nombres. En C++, no hay ningún enlace al objeto contenedor ni permisos implícitos.

```
public class Secuencia {
  private Object[] obs;
  private int siguiente = 0;
  public Secuencia(int tamanio) {
    obs = new Object[tamanio];
  }
  public void aniadir(Object x) {
    if(siguiente < obs.length) {
      obs[siguiente] = x;
      siguiente++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean fin() {
      return i == obs.length;
    }
    public Object actual() {
      return obs[i];
    }
    public void siguiente() {
      if(i < obs.length) i++;
    }
  }
  public Selector obtenerSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Secuencia s = new Secuencia(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.obtenerSelector();
    while(!sl.fin()) {
      System.out.println(sl.actual());
      sl.siguiente();
    }
  }
} ///:~
```

La **Secuencia** es simplemente un array de tamaño fijo de **Objetos** con una clase que lo envuelve. Para añadir un nuevo **Objeto** al final de la secuencia (si queda sitio) se llama a **aniadir( )**. Para buscar cada uno de los objetos de **Secuencia** hay una interfaz denominada **Selector** que permite ver si se está en el **fin( )**, echar un vistazo al método **actual( )**, y **siguiente( )** que permite moverse al siguiente objeto de la **Secuencia**. Dado que **Selector** es una **interfaz**, ésta puede ser implementada por otras muchas clases, y además los métodos podrían tomar la **interfaz** como parámetro, para crear código genérico.

Aquí, el **SSelector** es una clase **privada** que proporciona funcionalidad de **Selector**. En el método **main( )**, se puede ver la creación de una **Secuencia** seguida de la inserción de cierto número de objetos **String**. Después, se crea un **Selector** para llamar a **obtenerSelector( )** y éste se usa para moverse a través de la **Secuencia** y seleccionar cada elemento.

Al principio, la creación de **SSelector** parece simplemente otra clase interna. Pero examínela cuidadosamente. Fíjese que cada uno de los métodos **fin( )**, **actual( )**, y **siguiente( )** se refieren a **obs**, que es una referencia que no es parte de **SSelector**, sino un campo **privado** de la clase contenedora. Sin embargo, la clase interna puede acceder a métodos y campos de la clase contenedora como si les pertenecieran. Esto resulta ser muy conveniente, como se puede ver en el ejemplo de arriba.

Por tanto, una clase interna tiene acceso automático a los miembros de la clase contenedora. ¿Cómo puede ser esto? La clase interna debe mantener una referencia al objeto particular de la clase contenedora que era responsable de crearlo. Después, cuando se hace referencia al miembro de la clase contenedora, se usa esa referencia (oculta) para seleccionar ese miembro. Afortunadamente, el compilador se encarga de todos estos detalles, pero también podemos entender ahora que se pueda crear un objeto de una clase interna, sólo en asociación con un objeto de la clase contenedora. La construcción del objeto de la clase interna precisa de una referencia al objeto de la clase contenedora, y el compilador se quejará si no puede acceder a esa referencia. La mayoría de veces ocurre esto sin ninguna intervención por parte del programador.

Clases internas estáticas

Si no se necesita una conexión entre el objeto de la clase interna y el objeto de la clase externa, se puede hacer **estática** la clase interna. Para entender el significado de **estático** aplicado a clases internas, hay que recordar que el objeto de una clase interna ordinaria mantiene implícitamente una referencia al objeto de la clase contenedora que lo creó. Esto sin embargo no es cierto, cuando se dice que una clase interna es **estática**. Que una clase interna sea **estática** quiere decir que:

1. No se necesita un objeto de la clase externa para crear un objeto de una clase interna **estática**.
2. No se puede acceder a un objeto de una clase externa desde un objeto de una clase interna **estática**.

Las clases internas **estáticas** son distintas de las clases internas no **estáticas** también en otros aspectos. Los campos y métodos de las clases internas no **estáticas** sólo pueden estar en el nivel más externo de una clase, por lo que las clases internas no **estáticas** no pueden tener datos **estáticos**, campos **estáticos** o clases internas **estáticas**. Sin embargo, las clases internas **estáticas** pueden tener todo esto:

```
//: c08:Paquete10.java
// Clases internas estáticas.

public class Paquete10 {
  private static class PContenidos
  implements Contenidos {
    private int i = 11;
```

```
    public int valor() { return i; }
  }
  protected static class PDestino
      implements Destino {
    private String etiqueta;
    private PDestino(String aDonde) {
      etiqueta = aDonde;
    }
    public String leerEtiqueta() { return etiqueta; }
    // Las clases internas estáticas pueden tener
    // otros elementos estáticos:
    public static void f() {}
    static int x = 10;
    static class OtroNivel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destino dest(String s) {
    return new PDestino(s);
  }
  public static Contenidos cont() {
    return new PContenidos();
  }
  public static void main(String[] args) {
    Contenidos c = cont();
    Destinos d = dest("Tanzania");
  }
} ///:~
```

En el método **main( )** no es necesario ningún objeto de **Paquete10**; en cambio, se usa la sintaxis normal para seleccionar un miembro **estático** para invocar a los métodos que devuelven referencias a **Contenidos** y **Destino**.

Como se verá en breve, en una clase interna ordinaria (no **estática**) se logra un enlace al objeto de la clase externa con una referencia especial **this**. Una clase interna **estática** no tiene esta referencia **this** especial, lo que la convierte en análoga a un método **estático**.

Normalmente no se puede poner código en una **interfaz**, pero una clase interna **estática** puede ser parte de una **interfaz**. Dado que la clase es **estática** no viola las reglas de las interfaces —la clase interna **estática** sólo se ubica dentro del espacio de nombres de la interfaz:

```
//: c08:InterfazI.java
// Clases internas estáticas dentro de interfaces.

interface InterfazI {
```

```
  static class Interna {
    int i, j, k;
    public Interna() {}
    void f() {}
  }
} ///:~
```

Anteriormente sugerimos en este libro poner un método **main( )** en todas las clases para que actuara como banco de pruebas para cada una de ellas. Un inconveniente de esto es la cantidad de código compilado extra que se debe manejar. Si esto es un problema, se puede usar una clase interna **estática** para albergar el código de prueba:

```
//: c08:PruebaComa.java
// Poniendo código de pruebas en una clase estática interna.

class PruebaComa {
  PruebaComa() {}
  void f() { System.out.println("f()"); }
  public static class Probar {
    public static void main(String[] args) {
      PruebaComa t = new PruebaComa();
      t.f();
    }
  }
} ///:~
```

Esto genera una clase separada denominada **PruebaComa$Probar** (para ejecutar el programa, hay que decir **java PruebaComa$Probar**). Se puede usar esta clase para pruebas, pero no es necesario incluirla en el producto final.

Referirse al objeto de la clase externa

Si se necesita producir la referencia a la clase externa, se nombra la clase externa seguida por un punto y **this**. Por ejemplo, en la clase **Secuencia.SSelector**, cualquiera de sus métodos puede producir la referencia almacenada a la clase externa **Secuencia** diciendo **Secuencia.this**. La referencia resultante es automáticamente del tipo correcto. (Esto se conoce y comprueba en tiempo de compilación, por lo que no hay sobrecarga en tiempo de ejecución.)

En ocasiones, se desea decir a algún otro objeto que cree un objeto de una de sus clases internas. Para hacer esto hay que proporcionar una referencia al objeto de la otra clase externa en la expresión **new**, como en:

```
//: c08:Paquete11.java
// Creando instancias de clases internas.

public class Paquete11 {
```

```
  class Contenidos {
    private int i = 11;
    public int valor() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aDonde) {
      etiqueta = aDonde;
    }
    String leerEtiqueta() { return etiqueta; }
  }
  public static void main(String[] args) {
    Paquete11 p = new Paquete11();
    // Debe usar instancia de la clase externa
    // para crear una instancia de la clase interna:
    Paquete11.Contenidos c = p.new Contenidos();
    Paquete11.Destino d =
      p.new Destino("Tanzania");
  }
} ///:~
```

Para crear un objeto de la clase interna directamente, no se obra igual refiriéndose a la clase externa **Paquete11** como cabría esperar, sino que se une un *objeto* de la clase externa para construir un objeto de la clase interna:

```
Paquete11.Contenidos c = p.new Contenidos();
```

Por tanto, no es posible crear un objeto de la clase interna a menos que ya se tenga un objeto de la case externa. Esto se debe a que el objeto de la clase interna está conectado al objeto de la clase externa del que fue hecho. Sin embargo, si se hace una clase interna **estática**, entonces no es necesaria una referencia al objeto de la clase externa.

Acceso desde una clase múltiplemente anidada

No importa lo profundo que se anide una clase interna —puede acceder transparentemente a todos los miembros de todas las clases en los que esté anidada, como se muestra aquí:[3]

```
//: c08:AccesoAnidamientoMultiple.java
// Las clases anidadas pueden acceder a todos los miembros de todos
// niveles de las clases en las que están anidadas.

class AAM {
```

[3] Gracias de nuevo a Martin Danner.

```
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class AccesoAnidamientoMultiple {
  public static void main(String[] args) {
    AAM aam = new AAM();
    AAM.A aama = aam.new A();
    AAM.A.B aamab = aama.new B();
    aamab.h();
  }
} ///:~
```

Se puede ver que en **AAM.A.B**, los métodos **g( )** y **f( )** pueden ser invocados sin ningún tipo de restricción (a pesar de que sean **privados**). Este ejemplo también demuestra la sintaxis necesaria para crear objetos de clases internas múltiplemente anidadas cuando se crean los objetos en una clase distinta. La sintaxis "**.new**" produce el ámbito correcto por lo que no es necesario restringir el nombre de la clase en la llamada al constructor.

Heredar de clases internas

Dado que hay que adjuntar el constructor de la clase interna a la referencia del objeto de la clase contenedora, las cosas son ligeramente complicadas cuando se trata de heredar de una clase interna. El problema es que se debe inicializar la referencia "secreta" al objeto contenedor, y además en la clase derivada deja de haber un objeto por defecto al que adjuntarla. La respuesta es usar una sintaxis propucsta para hacer la asociación explícita:

```
//: c08:HerenciaInterna.java
// Heredando una clase interna.

class ConInterna {
  class Interna {}
}

public class HerenciaInterna
    extends HerenciaInterna.Interna {
  //! HerenciaInterna() {} // No compila
```

```
  HerenciaInterna(ConInterna wi) {
    wi.super();
  }
  public static void main(String[] args) {
    ConInterna wi = new ConInterna();
    HerenciaInterna ii = new HerenciaInterna(wi);
  }
} ///:~
```

Se puede ver que **HerenciaInterna** sólo está extendiendo la clase interna, y no la externa. Pero cuando llega la hora de crear un constructor, el constructor por defecto no es bueno y no se puede simplemente pasar una referencia a un objeto contenedor. Además, hay que usar la sintaxis:

```
referenciaClaseContenedora.super();
```

dentro del constructor. Esto proporciona la referencia necesaria para que el programa compile.

¿Pueden superponerse las clases internas?

¿Qué ocurre cuando se crea una clase interna, se hereda de la clase contenedora y se redefine la clase interna? Es decir, ¿es posible superponer una clase interna? Esto sería un concepto poderoso, pero la "superposición" en una clase interna como si fuera otro método de la clase externa verdaderamente no sirve para nada:

```
//: c08:HuevoGrande.java
// Una clase interna no se superpone como un método.

class Huevo {
  protected class Yema {
    public Yema() {
      System.out.println("Huevo.Yema()");
    }
  }
  private Yema y;
  public Huevo() {
    System.out.println("New Huevo()");
    y = new Yema();
  }
}

public class HuevoGrande extends Huevo {
  public class Yema {
    public Yema() {
      System.out.println("HuevoGrande.Yema()");
    }
  }
```

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

El compilador crea automáticamente el constructor por defecto, y éste llama al constructor por defecto de la clase base. Se podría pensar, que dado que se está creando **HuevoGrande**, debería usarse la versión "superpuesta" de **Yema**, pero éste no es el caso. La salida es:

```
New_Huevo()
Huevo.Yema()
```

Este ejemplo simplemente muestra que no hay ninguna magia extra propia de la clase interna cuando se hereda desde la clase externa. Las dos clases internas constituyen entidades completamente separadas, cada una en su propio espacio de nombres. Sin embargo, sigue siendo posible heredar explícitamente desde la clase interna:

```
//: c08:HuevoGrande2.java
// Herencia correcta de una clase interna.

class Huevo2 {
  protected class Yema {
    public Yema() {
      System.out.println("Huevo2.Yema()");
    }
    public void f() {
      System.out.println("Huevo2.Yema.f()");
    }
  }
  private Yema y = new Yema();
  public Huevo2() {
    System.out.println("New Huevo2()");
  }
  public void insertarYema(Yema yy) { y = yy; }
  public void g() { y.f(); }
}

public class HuevoGrande2 extends Huevo2 {
  public class Yema extends Huevo2.Yema {
    public Yema() {
      System.out.println("HuevoGrande2.Yema()");
    }
    public void f() {
      System.out.println("HuevoGrande2.Yema.f()");
    }
  }
```

```
  public HuevoGrande2() { insertaYema(new Yema()); }
  public static void main(String[] args) {
    Huevo2 e2 = new HuevoGrande2();
    e2.g();
  }
} ///:~
```

Ahora **HuevoGrande2.Yema** hereda explícitamente de **Huevo2.Yema** y superpone sus métodos. El método **insertaYema( )** permite a **HuevoGrande2** hacer una conversión hacia arriba a uno de sus propios objetos **Yema** hacia la referencia **y** de **Huevo2**, de forma que cuando **g( )** llama a **y.f( )** se usa la versión superpuesta de **f( )**. La salida es:

```
Huevo2.Yema()
New Huevo2()
Huevo2.Yema()
HuevoGrande2.Yema()
HuevoGrande2.Yema.f()
```

La segunda llamada a **Huevo2.Yema( )** es la llamada al constructor de la clase base del constructor **HuevoGrande2.Yema**. Se puede ver que se usa la versión superpuesta de **f( )** al llamar a **g( )**.

Identificadores de clases internas

Dado que toda clase produce un archivo **.class** que mantiene toda la información de como crear objetos de ese tipo (esta información produce una "meta-clase" llamada objeto **Class**), se podría adivinar que también las clases internas deben producir archivos **.class** para contener la información de *sus* objetos **Class**. Los nombres de estos archivos/clases tienen una fórmula estricta: el nombre de la clase contenedora, seguida de un "$", seguida del nombre de la clase interna. Por ejemplo, los ficheros **.class** creados por **HerenciaInterna.java** incluyen:

```
HerenciaInterna.class
ConInterna$Interna.class
ConInterna.class
```

Si las clases internas son anónimas, el compilador simplemente genera números como identificadores de las clases internas. Si las clases internas están anidadas dentro de clases internas, sus nombres simplemente se añaden tras un **"$"** y los identificadores de la clase externa.

Aunque este esquema de generación de nombres internos es simple y directo, también es robusto y maneja la mayoría de situaciones[5]. Dado que éste es el esquema de nombres estándar de Java, los ficheros generados son directamente independientes de la plataforma. (Nótese que el compilador de Java cambia las clases internas hasta hacerlas funcionar.)

---

[5] Por otro lado, "$" es un metacarácter para el *shell* de Unix, por lo que en ocasiones habrá problemas para listar las clases **.class**. Esto es un poco extraño viniendo de Sun, una compañía basada en Unix. Adivinamos que no tuvieron este aspecto en cuenta, y sin embargo, pensarían que había que centrarse en los ficheros de código fuente.

¿Por qué clases internas?

Hasta ahora se ha visto mucha sintaxis y semántica que describen el funcionamiento de las clases internas, pero esto no contesta a la pregunta de por qué existen. ¿Por qué Sun se metió en tanto lío para añadir esta característica fundamental del lenguaje?

Generalmente, la clase interna hereda de una clase o implementa una **interfaz**, y el código de la clase interna manipula el objeto de la clase externa en la que se ha creado. Por tanto, se podría decir que una clase interna proporciona una especie de ventana dentro de la clase externa.

Una pregunta que llega al corazón de las clases internas es: si simplemente se necesita una referencia a una **interfaz** ¿por qué no hacer simplemente que la clase externa implemente esa **interfaz**? La respuesta es: "Si eso es todo lo que necesitas, entonces así deberías hacerlo". Entonces, ¿qué es lo que distingue una clase interna que implementa una **interfaz** de una clase externa que implemente la misma **interfaz**? La respuesta es que siempre se puede tener la comodidad de las **interfaces** —algunas veces se trabaja con implementaciones. Por tanto, la razón más convincente para las clases internas es:

> *Cada clase interna puede heredar independientemente de una implementación. Por consiguiente, la clase interna no está limitada por el hecho de que la clase externa pueda estar ya heredando de una implementación.*

Sin la habilidad que las clases internas proporcionan para heredar —de hecho— desde más de una clase concreta o **abstracta**, algunos diseños y problemas de programación serían intratables. Por tanto, una forma de mirar a la clase interna es como la terminación de la solución del problema de la herencia múltiple. Las interfaces solucionan parte del problema, pero las clases internas permiten la "herencia de implementación múltiple" de manera efectiva. Es decir, las clases internas permiten heredar de forma efectiva de más de otro no-**interfaz**.

Para ver esto con mayor detalle, considérese una situación en que se tengan dos interfaces que de alguna manera deban implementarse dentro de una clase. Debido a la flexibilidad de las interfaces, se tienen dos alternativas: una única clase o una clase interna:

```
//: c08:InterfacesMultiples.java
// Dos formas en las que una clase puede
// implementar múltiples interfaces.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B creaB() {
    // Clase interna anónima:
    return new B() {};
  }
```

```
}

public class InterfacesMultiples {
  static void tomaA(A a) {}
  static void tomaB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    tomaA(x);
    tomaA(y);
    tomaB(x);
    tomaB(y.creaB());
  }
} ///:~
```

Por supuesto, esto asume que la estructura del código tiene sentido de alguna forma. Sin embargo, generalmente se tendrá algún tipo de guía, en la propia naturaleza del problema, sobre si usar una clase o una clase interna. Pero sin más restricciones, en el ejemplo de arriba el enfoque que se sigue no es muy diferente desde el punto de vista de la implementación. Ambos funcionan.

Sin embargo, si se tienen clases **abstractas** o concretas en vez de **interfaces**, nos limitamos repentinamente a usar clases internas si la clase debe implementar de alguna manera las otras dos:

```
//: c08:ImplementacionesMultiples.java
// Con clases concretas o abstractas, las clases
// internas son la única manera de producir el efecto de
// "herencia de implementación múltiple".

class C {}
abstract class D {}

class Z extends C {
  D crearD() { return new D() {}; }
}

public class ImplementacionesMultiples {
  static void tomarC(C c) {}
  static void tomarD(D d) {}
  public static void main(String[] args) {
    Z z = new Z();
    tomarC(z);
    tomarD(z.crearD());
  }
} ///:~
```

Si no se deseara resolver el problema de la "herencia de implementación múltiple", se podría codificar posiblemente todo lo demás sin la necesidad de clases internas. Pero con las clases internas se tienen estas características adicionales:

1. La clase interna tiene múltiples instancias, cada una con su propia información de estado que es independiente de la información del objeto de la clase externa.
2. En una clase externa se pueden tener varias clases internas, cada una de las cuales implementa la misma **interfaz** o hereda de la misma clase de distinta forma. En breve se mostrará un ejemplo de esto.
3. El momento de creación del objeto de la clase interna no está atado a la creación del objeto de la clase externa.
4. No hay relaciones "es-un" potencialmente confusas dentro de la clase interna; se trata de una entidad separada.

Como ejemplo, si **Secuencia.java** no usara clases internas, habría que decir que "una **Secuencia** es un **Selector**", y sólo se podría tener un **Selector** para una **Secuencia** particular. Además, se puede tener un segundo método, **obtenerRSelector( )**, que produce un **Selector** que se mueve hacia atrás por la secuencia. Este tipo de flexibilidad sólo está disponible con clases internas.

Cierres (closures) y Retrollamadas (Callbacks)

Un *cierre* es un objeto invocable que retiene información del ámbito en el que se creó. Por definición, se puede ver que una clase interna es un *cierre* orientado a objetos, porque no se limita a contener cada fragmento de información del objeto de la clase externa ("el ámbito en el que ha sido creada"), sino que mantiene automáticamente una referencia de vuelta al objeto completo de la clase externa, en el que tiene permiso para manipular todos los miembros, incluso los **privados**.

Uno de los argumentos más convincentes, hechos para incluir algún tipo de mecanismo apuntador en Java, era permitir las llamadas hacia atrás o *retrollamadas*. Con una *retrollamada*, se da a otro objeto de java un fragmento de información que le permite invocar al objeto que lo originó en algún momento. Éste es un concepto muy potente, como se verá en los Capítulos 13 y 16. Si se implementa una retrollamada utilizando un puntero, sin embargo, hay que confiar en que el programador se comporte adecuadamente y no use incorrectamente el puntero. Como se ha visto hasta ahora, Java tiende a ser más cuidadoso, de forma que no se incluyen punteros en el propio lenguaje.

El cierre proporcionado por la clase interna es la solución perfecta; más flexible y mucho más segura que un puntero. He aquí un ejemplo simple:

```
//: c08:Retrollamadas.java
// Utilizando clases internas para retrollamadas

interface Incrementable {
  void incrementar();
}
```

```
// Muy simple para simplemente implementar la interfaz:
class Llamada1 implements Incrementable {
  private int i = 0;
  public void incrementar() {
    i++;
    System.out.println(i);
  }
}

class MiIncremento {
  public void incremento() {
    System.out.println("Otra operacion");
  }
  public static void f(MiIncremento mi) {
    mi.incrementar();
  }
}

// Si tu clase debe implementar incrementar() de alguna u otra
// manera, hay que usar una clase interna:
class Llamada2 extends MiIncremento {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Cierre implements Incrementable {
    public void incrementar() { incr(); }
  }
  Incrementable obtenerReferenciaRetrollamada() {
    return new Cierre();
  }
}

class Visita {
  private Incrementable referenciaRetrollamada;
  Visita(Incrementable cbh) {
    referenciaRetrollamada = cbh;
  }
  void realizar() {
    referenciaRetrollamada.incrementar();
  }
}

public class Retrollamada {
```

```
  public static void main(String[] args) {
    Llamada1 c1 = new Llamada1();
    Llamada2 c2 = new Llamada2();
    MiIncremento.f(c2);
    Visita visita1 = new Visita(c1);
    Visita visita2 =
      new Visita(c2.obtenerReferenciaRetrollamada());
    visita1.realizar();
    visita1.realizar();
    visita2.realizar();
    visita2.realizar();
  }
} ///:~
```

Este ejemplo también proporciona una distinción aún mayor entre implementar una interfaz en una clase externa o hacerlo en una interna. **Llamada1** es claramente la solución más simple en lo que a código se refiere. **Llamada2** hereda de **MiIncremento**, que ya tiene un método **incrementar( )** diferente que hace algo no relacionado con lo que se espera de la interfaz **Incrementable**. Cuando se hereda **Llamada2.incrementar** de **MiIncremento**, no se puede superponer **incrementar( )** para ser usado por parte de **Incrementable**, por lo que uno se ve forzado a proporcionar una implementación separada utilizando una clase interna. Fíjese también que cuando se crea una clase interna no se añade o modifica la interfaz de la clase externa.

Téngase en cuenta que en **Llamada2** todo menos **obtenerReferenciaRetrollamada( )** es **privado**. Para permitir *cualquier* conexión al mundo exterior, es esencial la **interfaz Incrementable**. Aquí se puede ver cómo las **interfaces** permiten una completa separación de la interfaz de la implementación.

La clase interna **Cierre** simplemente implementa **Implementable** para proporcionar un anzuelo de vuelta a **Llamada2** —pero un anzuelo seguro. Quien logre una referencia **Incrementable** puede, por supuesto, invocar sólo a **incrementar( )** y no tiene otras posibilidades (a diferencia de un puntero, que permitiría acceso total).

Visita toma una referencia **Incrementable** en su constructor (aunque la captura de la referencia a la retrollamada podría darse en cualquier momento) y entonces, algo después, utiliza la referencia para hacer una "llamada hacia atrás" a la clase **Llamada**.

El valor de una retrollamada reside en su flexibilidad —se puede decidir dinámicamente qué funciones serán invocadas en tiempo de ejecución. El beneficio de esto se mostrará más evidentemente en el Capítulo 13, en el que se usan retrollamadas en todas partes para implementar la funcionalidad de la interfaz gráfica de usuario (IGU).

Clases internas y sistema de control

Un ejemplo más concreto del uso de las clases internas sería lo que llamamos *sistema de control.*

Un *sistema de aplicación* es una clase o conjunto de clases diseñado para solucionar un tipo particular de problema. Para aplicar un sistema de aplicación, se hereda de una o más clases y se su-

perponen algunos de los métodos. El código que se escribe en los métodos superpuestos particulariza la solución proporcionada por el sistema de aplicación, para solucionar un problema específico. El sistema de control es un tipo particular de sistema de aplicación dominado por la necesidad de responder a eventos; un sistema que responde principalmente a eventos se denomina *sistema dirigido por eventos*. Uno de los problemas más importantes en la programación de aplicaciones es la interfaz gráfica de usuario (IGU), que está dirigida a eventos casi completamente. Como se verá en el Capítulo 13, la biblioteca Swing de Java es un sistema de control que soluciona el problema del IGU de forma elegante, utilizando intensivamente clases internas.

Para ver cómo las clases internas permiten la creación y uso de forma simple de sistemas de control, considérese uno cuyo trabajo es ejecutar eventos siempre que éstos estén "listos". Aunque "listos" podría significar cualquier caso, en este caso se usará su significado por defecto, basado en el reloj. Lo que sigue es un sistema de control que no contiene información específica sobre qué se está controlando. En primer lugar, he aquí una interfaz que describe cualquier evento de control. Es una clase **abstracta** en vez de una **interfaz** porque su comportamiento por defecto es llevar a cabo el control basado en el tiempo, por lo que se puede incluir ya alguna implementación:

```
//: c08:controlador:Evento.java
// Los métodos comunes para cualquier evento de control.
package c08.controlador;

abstract public class Evento {
  private long instEvento;
  public Evento(long instanteEvento) {
    instEvento = instanteEvento;
  }
  public boolean listo() {
    return System.currentTimeMillis() >= instEvento;
  }
  abstract public void accion();
  abstract public String descripcion();
} ///:~
```

El constructor simplemente captura el instante de tiempo en el que que se desea que se ejecute el **Evento**, mientras que **listo( )** dice cuándo es hora de ejecutarlo. Por supuesto, podría superponerse **listo( )** en alguna clase derivada de la clase **Descripción**.

El método **accion( )** es el que se invoca cuando el **Evento** está **listo( )**, y **descripción** da información textual sobre el **Evento**.

El siguiente fichero contiene el sistema de control que gestiona eventos de incendios. La primera clase es realmente una clase "ayudante" cuyo trabajo es albergar objetos **Evento**. Se puede sustituir por cualquier contenedor apropiado, y en el Capítulo 9 se descubrirán otros contenedores que proporcionarán este truco sin exigir la escritura de código extra:

```
//: c08:controlador:Controlador.java
// Junto con Evento, el sistema genérico
// para todos los sistemas de control:
```

```
package c08.controlador;

// Esto es simplemente una forma de guardar objetos Evento.
class ConjuntoEventos {
  private Evento[] eventos = new Evento[100];
  private int indice = 0;
  private int siguiente = 0;
  public void aniadir(Evento e) {
    if(indice >= eventos.length)
      return; // (En realidad, lanzar una excepción)
    eventos[indice++] = e;
  }
  public Evento obtenerSiguiente() {
    boolean vuelta = false;
    int primero = siguiente;
    do {
      siguiente = (siguiente + 1) % eventos.length;
      // Ver si ha vuelto al principio:
      if(primero == siguiente) vuelta = true;
      // Si va más allá de primero, la lista está vacía:
      if((siguiente == (primero + 1) % eventos.length)
          && eventos)
        return null;
    } while(eventos[siguiente] == null);
    return eventos[siguiente];
  }
  public void eliminarActual() {
    eventos[siguiente] = null;
  }
}

public class Controlador {
  private ConjuntoEventos es = new ConjuntoEventos();
  public void aniadirEvento(Evento c) { es.aniadir(c); }
  public void ejecutar() {
    Evento e;
    while((e = es.obtenerSiguiente()) != null) {
      if(e.listo()) {
        e.accion();
        System.out.println(e.descripcion());
        es.eliminarActual();
      }
    }
  }
} ///:~
```

ConjuntoEventos mantiene arbitrariamente 100 **objetos de tipo Evento**. (Si se usara un contenedor de los que veremos en el Capítulo 9 no haría falta preocuparse por su tamaño máximo, puesto que se recalcularía por sí mismo.) El **índice** se usa para mantener información del espacio disponible, y **siguiente** se usa cuando se busca el siguiente **Evento** de la lista, para ver si ya se ha dado la vuelta o no. Esto es importante durante una llamada a **obtenerSiguiente( )**, porque los objetos **Evento** se van eliminando de la lista (utilizando **eliminarActual( )**) una vez que se ejecutan, por lo que **obtenerSiguiente( )** encontrará espacios libres en la lista al recorrerla.

Nótese que **eliminarActual( )** no se dedica simplemente a poner algún indicador que muestre que el objeto ya no está en uso. En vez de esto, pone la referencia a **null**. Esto es importante porque si el recolector de basura ve una referencia que sigue estando en uso, no puede limpiar el objeto. Si uno piensa que sus referencias podrían quedarse colgadas (como ocurre aquí), es buena idea ponerlas a **null** para dar permiso al recolector de basura y que los limpie.

Es en **Controlador** donde se da el verdadero trabajo. Utiliza un **ConjuntoEventos** para mantener sus objetos **Evento**, y **aniadirEvento( )** permite añadir nuevos eventos a esta lista. Pero el método importante es **ejecutar( )**. Este método se mete en un bucle en **ConjuntoEventos**, buscando un objeto **Evento** que esté **listo( )** para ser ejecutado. Por cada uno que encuentre **listo( )**, llama al método **accion( )**, imprime la **descripcion( )**, y quita el **Evento** de la lista.

Fíjese que hasta ahora en este diseño no se sabe nada de *qué* hace un **Evento**. Y esto es lo esencial del diseño; cómo "separa las cosas que cambian de las que permanecen igual". O, usando el término, el "vector de cambio" está formado por las diferentes acciones de varios tipos de objetos **Evento**, y uno expresa acciones diferentes creando distintas subclases **Evento**.

Es aquí donde intervienen las clases internas, que permiten dos cosas:

1. Crear la implementación completa de una aplicación de sistema de control en una única clase, encapsulando, por tanto, todo lo que sea único de la implementación. Se usan las clases internas para expresar los distintos tipos de **accion( )** necesarios para resolver el problema. Además, el ejemplo siguiente usa clases internas **privadas**, por lo que la implementación está completamente oculta y puede ser cambiada con impunidad.

2. Las clases internas hacen que esta implementación no sea excesivamente complicada porque se puede acceder sencillamente a cualquiera de los miembros de la clase exterior. Sin esta capacidad, el código podría volverse tan incómodo de manejar que se acabaría buscando otra alternativa.

Considérese una implementación particular del sistema de control diseñada para controlar las funciones de un invernadero[6]. Cada acción es completamente distinta: encender y apagar las luces, agua y termostatos, hacer sonar timbres, y reinicializar el sistema. Pero el sistema de control está diseñado para aislar fácilmente este código diferente. Las clases internas permiten tener múltiples versiones derivadas de la misma clase base, **Evento**, dentro de una única clase. Por cada tipo de acción se hereda una nueva clase interna **Evento**, y se escribe el código de control dentro de **accion( )**.

---

[6] Por algún motivo, éste siempre ha sido un problema que nos ha gustado resolver; viene en el libro *C++ Inside & Out*, pero Java permite una solución mucho más elegante.

Como es típico con un sistema de aplicación, la clase **ControlesInvernadero** se hereda de **Controlador**:

```
//: c08:ControlesInvernadero.java
// Aplicación especifica del sistema de
// control, toda ella en una única clase. Las clases internas
// permiten encapsular diferentes funcionalidades
// por cada tipo de evento.
import c08.controlador.*;

public class ControlesInvernadero
     extends Controlador {
  private boolean luz = false;
  private boolean agua = false;
  private String termostato = "Dia";
  private class EncenderLuz extends Evento {
    public EncenderLuz(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      // para encender físicamente la luz.
      luz = true;
    }
    public String descripcion() {
      return "Luz encencida";
    }
  }
  private class ApagarLuz extends Evento {
    public ApagarLuz(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      // para apagar físicamente la luz.
      luz = false;
    }
    public String descripcion() {
      return "Luz apagada";
    }
  }
  private class EncenderAgua extends Evento {
    public EncenderAgua(long instanteEvento) {
      super(instanteEvento);
    }
```

```
    public void accion() {
      // Poner aquí el código de control de hardware
      agua = true;
    }
    public String descripcion() {
      return "Agua del invernadero encendida";
    }
  }
  private class ApagarAgua extends Evento {
    public ApagarAgua(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      agua = false;
    }
    public String descripcion() {
      return "Agua del invernadero apagada";
    }
  }
  private class TermostatoNoche extends Evento {
    public TermostatoNoche(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      termostato = "Noche";
    }
    public String descripcion() {
      return "Termostato activado para la noche";
    }
  }
  private class TermostatoDia extends Evento {
    public TermostatoDia(long instanteEvento) {
      super(instanteEvento);
    }
    public void accion() {
      // Poner aquí el código de control de hardware
      termostato = "Dia";
    }
    public String descripcion() {
      return "Termostato activado para el dia";
    }
  }
  // Ejemplo de una acción() que inserta una
```

```
// nueva acción dentro de la lista de eventos:
private int timbres;
private class Campana extends Evento {
  public Campana(long instanteEvento) {
    super(instanteEvento);
  }
  public void accion() {
    // Sonar cada 2 segundos, 'timbres':
    System.out.println("¡Ring!");
    if(--timbres > 0)
      aniadirEvento(new Campana(
        System.currentTimeMillis() + 2000));
  }
  public String descripcion() {
    return "Sonar la campana";
  }
}
private class Rearrancar extends Evento {
  public rearrancar(long instanteEvento) {
    super(instanteEvento);
  }
  public void accion() {
    long tm = System.currentTimeMillis();
    // En vez de cableado se podría poner información
    // de configuración de un fichero de texto aquí:
    timbres = 5;
    aniadirEvento(new TermostatoNoche(tm));
    aniadirEvento(new EncenderLuz(tm + 1000));
    aniadirEvento(new ApagarLUz(tm + 2000));
    aniadirEvento(new EncenderAgua(tm + 3000));
    aniadirEvento(new ApagarAgua(tm + 8000));
    aniadirEvento(new Campana(tm + 9000));
    aniadirEvento(new TermostatoDia(tm + 10000));
    // ¡Incluso se puede añadir un objeto rearrancar!
    aniadirEvento(new Rearrancar(tm + 20000));
  }
  public String descripcion() {
    return "Reiniciando el sistema";
  }
}
public static void main(String[] args) {
  ControlesInvernadero gc =
    new ControlesInvernadero();
  long tm = System.currentTimeMillis();
  gc.aniadirEvento(gc.new Rearrancar(tm));
```

```
      gc.ejecutar();
    }
} ///:~
```

Fíjese que **luz**, **agua**, **termostato** y **campanas** pertenecen a la clase externa **ControlesInvernadero**, y sin embargo las clases internas pueden acceder a esos campos sin restricciones o permisos especiales. Además, la mayoría de los métodos **accion( )** implican algún tipo de control hardware, que con mucha probabilidad incluirán llamadas a código no Java.

La mayoría de clases **Evento** tienen la misma apariencia, pero **Campana** y **Rearrancar** son especiales. La **Campana** suena, y si no ha sonado aún el número suficiente de veces, añade un nuevo objeto **Campana** a la lista de eventos, de forma que volverá a sonar más tarde. Téngase en cuenta cómo las clases internas parecen una herencia múltiple: **Campana** tiene todos los métodos de **Evento** y también parece tener todos los métodos de la clase externa **ControlesInvernadero**.

Rearrancar es la responsable de inicializar el sistema, de forma que añade todos los eventos apropiados. Por supuesto, se puede lograr lo mismo de forma más flexible evitando codificar los eventos y en vez de ello leerlos de un archivo. (Un ejercicio que se pide en el Capítulo 11 es modificar este ejemplo para que haga justamente eso.) Dado que **Rearrancar( )** es simplemente otro objeto **Evento( )**, se puede añadir también un objeto **Rearrancar** dentro de **Rearrancar.accion( )**, de forma que el sistema se inicie a sí mismo regularmente. Y todo lo que se necesita hacer en el método **main( )** es crear un objeto **ControlesInvernadero** y añadir un objeto **Rearrancar** para que funcione.

Este ejemplo debería transportar al lector un gran paso hacia adelante al apreciar el valor de las clases internas, especialmente cuando se usan dentro de un sistema de control. Sin embargo, en el Capítulo 13 se verá cómo se usan estas clases elegantemente para describir las acciones de una interfaz gráfica de usuario. Para cuando se acabe ese capítulo, todo lector sabrá manejarlos.

Resumen

Las interfaces y las clases internas son conceptos más sofisticados que lo que se encontrará en la mayoría de lenguajes de POO. Por ejemplo, no hay nada igual en C++. Juntos, solucionan el mismo problema que C++ trata de solucionar con su característica de herencia múltiple[6]. Sin embargo, la MI de C++ resulta ser bastante difícil de usar, mientras que las clases internas e interfaces de Java son, en comparación, mucho más accesibles.

Aunque las características por sí mismas son bastante directas, usarlas es un aspecto de diseño, al igual que ocurre con el polimorfismo. Con el tiempo, uno será capaz de reconocer mejor las situaciones en las que hay que usar una interfaz, o una clase interna, o ambas. Pero en este punto del libro, deberíamos habernos familiarizado con su sintaxis y semántica. A medida que se vea el uso de estas características, uno las irá haciendo propias.

---

[6] N. del traductor: *Múltiple Inheritance* (MI) en C++.

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. Probar que los métodos de una **interfaz** son implícitamente **estáticos** y **constantes**.
2. Crear una **interfaz** que contenga tres métodos en su propio **paquete**. Implementar la interfaz en un **paquete** diferente.
3. Probar que todos los métodos de una **interfaz** son automáticamente **públicos**.
4. En **c07:Bocadillo.java**, crear una interfaz denominada **ComidaRapida** (con métodos apropiados) y cambiar **Bocadillo** de forma que también implemente **ComidaRapida**.
5. Crear tres **interfaces**, cada uno con dos métodos. Heredar una nueva **interfaz** de las tres, añadiendo un nuevo método. Crear una clase implementando la nueva **interfaz** y heredando además de una clase concreta. Ahora escribir cuatro métodos, cada uno de los cuales toma una de las cuatro **interfaces** como parámetro. En el método **main( )**, crear un objeto de la nueva clase y pasárselo a cada uno de los métodos.
6. Modificar el Ejercicio 5, creando una clase **abstracta** y heredándola en la clase derivada.
7. Modificar **Musica5.java** añadiendo una **interfaz Tocable**. Eliminar la declaración **tocar( )** de **Instrumento**. Añadir **Tocable** a las clases derivadas incluyéndolo en la lista de interfaces que **implementa**. Cambiar **afinar( )** de forma que tome un **Tocable** en vez de un **Instrumento**.
8. Cambiar el Ejercicio 6 del Capítulo 7 de forma que **Roedor** sea una **interfaz**.
9. En **Aventura.java**, añadir una **interfaz** denominada **PuedeTrepar** siguiendo el modelo de las otras interfaces.
10. Escribir un programa que importe y use **Mes2.java**.
11. Siguiendo el ejemplo de **Mes2.java**, crear una enumeración de los días de la semana.
12. Crear una **interfaz** con al menos un método, en su propio paquete. Crear una clase en otro paquete. Añadir una clase interna **protegida** que implemente la **interfaz**. En un tercer paquete, heredar de la nueva clase y, dentro de un método, devolver un objeto de la clase interna **protegida**, haciendo un conversión hacia arriba a la **interfaz** durante este retorno.
13. Crear una **interfaz** con al menos un método, e implementar esa **interfaz** definiendo una clase interna dentro de un método, que devuelva una referencia a la **interfaz**.
14. Repetir el Ejercicio 13, pero definir la clase interna dentro del ámbito de un método.
15. Repetir el Ejercicio 13 utilizando una clase interna anónima.
16. Crear una clase interna **privada** que implemente una **interfaz pública**. Escribir un método que devuelva una referencia a una instancia de la clase interna **privada**, hacer un conversión

hacia arriba a la **interfaz**. Mostrar que la clase interna está totalmente oculta intentando hacer una, conversión hacia abajo de la misma.

17. Crear una clase con un constructor distinto del constructor por defecto, y sin éste. Crear una segunda clase que tenga un método que devuelva una referencia a la primera clase. Crear el objeto a devolver haciendo una clase interna anónima que herede de la primera clase.

18. Crear una clase con un atributo **privado** y un método **privado**. Crear una clase interna con un método que modifique el atributo de la clase externa y llame al método de la clase externa. En un segundo método de la clase externa, crear un objeto de la clase interna e invocar a su método, después mostrar el efecto en el objeto de la clase externa.

19. Repetir el Ejercicio 18 utilizando una clase interna anónima.

20. Crear una clase que contenga una clase interna **estática**. En el método **main( )**, crear una instancia de la clase interna.

21. Crear una **interfaz** que contenga una clase interna **estática**. Implementar esta **interfaz** y crear una instancia de la clase interna.

22. Crear una clase que contenga una clase interna que contenga a su vez otra clase interna. Repetir lo mismo usando clases internas **estática**. Fijarse en los nombres de los archivos **.class** producidos por el compilador.

23. Crear una clase con una clase interna. En una clase separada, hacer una instancia de la clase interna.

24. Crear una clase con una clase interna que tiene un constructor distinto del constructor por defecto. Crear una segunda clase con una clase interna que hereda de la primera clase interna.

25. Reparar el problema de **ErrorViento.java**.

26. Modificar **Secuencia.java** añadiendo un método **obtenerRSelector( )** que produce una implementación diferente de la **interfaz Selector** que se mueve hacia atrás de la secuencia desde el final al principio.

27. Crear una **interfaz U** con tres métodos. Crear una clase **A** con un método que produce una referencia a **U** construyendo una clase interna anónima. Crear una segunda clase **B** que contenga un array de **U**. **B** debería tener un método que acepte y almacene una referencia a **U** en el array, un segundo método que establece una referencia dentro del array (especificada por el parámetro del método) a **null,** y un tercer método que se mueve a través del array e invoca a los métodos de **U**. En el método **main( )**, crear un grupo de objetos **A** y un único **B**. Rellenar el **B** con referencias **U** producidas por los objetos **A**. Utilizar el **B** para invocar de nuevo a todos los objetos **A**. Eliminar algunas de las referencias **U** de **B**.

28. En **ControlesInvernadero.java**, añadir clases internas **Evento** que enciendan y apaguen ventiladores.

29. Mostrar que una clase interna tiene acceso a los elementos **privados** de su clase externa. Determinar si también se cumple a la inversa.