5: Ocultar la implementación

Una consideración primordial del diseño orientado a objetos es "la separación de aquellas cosas que varían de aquéllas que permanecen constantes".

Esto es especialmente importante en el caso de las bibliotecas. El usuario (el *programador cliente*) de la biblioteca debe ser capaz de confiar en la parte que usa, y saber que no necesita reescribir el código si se lanza una nueva versión de esa biblioteca. Por otro lado, el creador de la biblioteca debe tener la libertad para hacer modificaciones y mejoras con la certeza de que el código del programador cliente no se verá afectado por estos cambios.

Esto puede lograse mediante una convención. Por ejemplo, el programador de la biblioteca debe acordar no eliminar métodos existentes al modificar una clase de la biblioteca, dado que eso destruiría el código del programador cliente. El caso contrario sería más problemático. En el caso de un atributo ¿cómo puede el creador de la biblioteca saber qué atributos son los que los programadores clientes han usado? Esto también ocurre con aquellos métodos que sólo son parte de la implementación de la clase, pero que no se diseñaron para ser usados directamente por el programador cliente. Pero, ¿qué ocurre si el creador de la biblioteca desea desechar una implementación antigua y poner una nueva? El cambio de cualquiera de esos miembros podría romper el código de un programador cliente. Por consiguiente, el creador de la biblioteca se encuentra limitado, y no puede cambiar nada.

Para solucionar este problema, Java proporciona *especificadores de acceso* para permitir al creador de la biblioteca decir qué está disponible para el programador cliente, y qué no. Los niveles de control de acceso desde el "acceso máximo" hasta el "acceso mínimo" son **public**, **protected**, "friendly" (para el que no existe palabra clave), y **private**. Por el párrafo anterior podría pensarse que, al igual que el diseñador de la biblioteca, se deseará mantener "private" tanto como sea posible, y exponer únicamente los métodos que se desee que use el programador cliente. Esto es completamente correcto, incluso aunque frecuentemente no es intuitivo para aquéllos que programan en otros lenguajes (especialmente C), y se utilizan para acceder a todo sin restricciones. Para cuando acabe este capítulo, el lector debería convencerse del valor del control de accesos en Java.

Sin embargo, el concepto de una biblioteca de componentes y el control sobre quién puede acceder a los componentes de esa biblioteca están completos. Todavía queda la cuestión de cómo se empaquetan los componentes para formar una unidad cohesiva. Esto se controla en Java con la palabra clave **package**, y los especificadores de acceso se ven en la medida en que una clase se encuentre en un mismo o distinto paquete. Por tanto, para empezar este capítulo, se aprenderá cómo ubicar los componentes de las bibliotecas en paquetes. Posteriormente, uno será capaz de comprender el significado completo de los especificadores de acceso.

El paquete: la unidad de biblioteca

Un paquete es lo que se obtiene al utilizar la palabra clave **import** para importar una biblioteca completa, como en:

```
import java.util.*;
```

Esto trae la biblioteca de utilidades entera, que es parte de la distribución estándar de Java. Dado que, por ejemplo, la clase **ArrayList** se encuentra en **java.util** es posible especificar el nombre completo **java.util. ArrayList** (lo cual se puede hacer sin la sentencia **import**) o bien se puede simplemente decir **ArrayList** (gracias a la sentencia **import**).

Si se desea incorporar una única clase, es posible nombrarla sin la sentencia **import**:

```
import java.util.ArrayList;
```

Ahora es posible hacer uso de **ArrayList**, aunque no estarán disponibles ninguna de las otras clases de **java.util**.

La razón de todas estas importaciones es proporcionar un mecanismo para gestionar los "espacios de nombres". Los nombres de todas las clases miembros están aislados unos de otros. Un método **f( )** contenido en la clase **A** no colisionará con un método **f( )** que tiene la misma lista de argumentos, dentro de la clase **B**. Pero, ¿qué ocurre con los nombres de clases? Supóngase que se crea una clase **Pila** que se instala en una máquina que ya tiene una clase **Pila** escrita por otra persona. Con Java en Internet, esto podría incluso ocurrir sin que el usuario lo sepa, dado que es posible que algunas clases se descarguen automáticamente en el proceso de ejecutar un programa Java.

Esta potencial colisión de nombres justifica la necesidad de tener control sobre los espacios de nombre en Java, y de tener la capacidad de crear un nombre completamente único sin que importen las limitaciones de Internet.

Hasta ahora, la mayoría de los ejemplos de este libro se incluían en un único fichero y están diseñados para un uso local, por lo que no han tenido que hacer uso de los nombres de paquetes. (En este caso el nombre de clase se ubica en el "paquete por defecto".) Ésta es ciertamente una opción, y con motivo de mantener la máxima simplicidad, se usará este enfoque siempre que sea posible en todo el resto del libro. Sin embargo, si se planifica crear bibliotecas o programas que se relacionan con otros programas Java de la misma máquina, hay que pensar en evitar las colisiones entre nombres de clases.

Cuando se crea un fichero de código fuente en Java, se crea lo que comúnmente se denomina una *unidad de compilación* (en ocasiones se denomina una *unidad de traducción*). Cada una de estas unidades tiene un nombre que acaba en **.java**, y dentro de la unidad de compilación puede haber una única clase **pública**, sino, el compilador se quejará. El resto de clases de esa unidad de compilación, si es que hay alguna, quedan ocultas para todo lo exterior al paquete al *no* ser **pública**, y constituyen clases de "apoyo" para la clase **pública** principal.

Cuando se compila un fichero **.java**, se obtiene un fichero de salida que tiene exactamente el mismo nombre pero tiene extensión **.class** *por cada clase* del fichero **.java**. Por tanto, se puede acabar teniendo bastantes ficheros **.class** partiendo de un número pequeño de ficheros **.java**. Si se programa haciendo uso de un lenguaje compilado, puede que uno esté acostumbrado a que el compilador devuelva un fichero en un formato intermedio (generalmente un fichero "obj") que se empaqueta junto con otros de su misma clase utilizando, bien un montador (para crear un fichero ejecutable) o una biblioteca. Java no funciona así. Un programa en acción es un compendio de ficheros **.java**, que pueden empaquetarse y comprimirse en un fichero JAR (utilizando la herramienta **jar** de Java).

El intérprete de Java es el responsable de encontrar, cargar e interpretar estos ficheros[1].

Una biblioteca también es un conjunto de estos ficheros de clase. Cada fichero tiene una clase que es **pública** (no es obligatorio introducir una clase **pública**, pero lo habitual es hacerlo así), de forma que hay un componente por cada fichero. Si se desea indicar que todos estos componentes (que se encuentran en sus propios ficheros separados **.java** y **.class**) permanezcan unidos, es necesaria la intervención de la palabra clave **package**.

Cuando se dice:

```
package mipaquete;
```

(al principio de un archivo), si se usa la sentencia **package**, ésta *debe* aparecer en la primera línea que no sea un comentario del fichero), se está indicando que esa unidad de compilación es parte de una biblioteca de nombre **mipaquete**. O, dicho de otra forma, se está diciendo que el nombre de la clase **pública** incluida en esa unidad de compilación se encuentra bajo el paraguas del nombre, **mipaquete**, y si alguien quiere utilizar el nombre, deben, o bien especificar completamente el nombre o bien usar la palabra clave **import** en combinación con **mipaquete** (utilizando las opciones descritas previamente). Fíjese que la convención para los nombres de paquete de Java dice que se usen únicamente letras minúsculas, incluso cuando hay más de una palabra.

Por ejemplo, supóngase que el nombre del fichero es **MiClase.java**. Esto significa que puede haber una y sólo una clase **pública** en ese fichero, y el nombre de esa clase debe ser **MiClase** (incluidas las mayúsculas y minúsculas):

```
package mipaquete;
public class MiClase {
  // . . .
```

Ahora, si alguien desea usar **MiClase** o, por cualquier motivo, cualquiera de las clases **públicas** de **mipaquete**, debe usar la palabra clave **import** para lograr que estén disponibles el/los nombres de **mipaquete**. La alternativa es dar el nombre completo:

```
mipaquete.MiClase m = new mipaquete.MiClase();
```

[1] No hay nada en Java que obligue al uso de un intérprete. Existen compiladores de código nativo Java que generan un único fichero ejecutable.

La palabra clave **import** puede lograr lo mismo pero de manera bastante más clara:

```
import mipaquete
// . . .
MiClase m = MiClase();
```

Merece la pena recordar que lo que las palabras clave **package** e **import** permiten hacer, como diseñador de bibliotecas, es dividir el espacio de nombres único y global, de forma que no se tengan colisiones de nombres, sin que importe cuánta gente se conecte a Internet y empiece a escribir clases en Java.

Creando nombres de paquete únicos

Podría observarse que, dado que un paquete nunca se llega a "empaquetar" en un fichero único, un mismo fichero podría estar constituido por muchos ficheros **.class**, y esto podría ser fuente de desorden y confusión. Para evitarlo, algo lógico es ubicar todos los ficheros **.java** de un paquete particular en un mismo directorio; es decir, hacer uso de la estructura de ficheros jerárquica del sistema operativo y sacar provecho de ella. Ésta es una de las maneras en que Java referencia el problema del desorden; se verá otra manera después, cuando se presente la utilidad **jar**.

La agrupación de los ficheros de paquete en un único subdirectorio soluciona otros dos problemas: la creación de nombres de paquete únicos, y la localización de esas clases que podrían estar enterradas en algún lugar de la estructura de directorios. Esto se logra, tal y como se presentó en el Capítulo 2, codificando camino de localización del fichero **.class** en el nombre del **paquete**. El compilador obliga a que esto sea así, pero por convención, la primera parte del nombre de un **paquete** es el nombre del dominio Internet del creador de la clase, eso sí, dado la vuelta. Dado que está garantizado que los nombres de dominio de Internet sean únicos, *si* se sigue esta convención se garantiza que el nombre del **paquete** sea único y, por consiguiente, nunca habrá colisiones de nombres (es decir, hasta que se pierde el nombre de dominio y alguien se hace con él y empieza a escribir código Java con los mismos nombres de ruta con los que lo hizo). Por supuesto, si se dispone de un nombre de dominio propio, es necesario fabricar en primer lugar una combinación única (como, por ejemplo, la formada por el nombre y apellidos) para crear nombres de paquete únicos. Si se ha decidido comenzar a publicar código Java merece la pena el esfuerzo, relativamente pequeño, de conseguir en primer lugar un nombre de dominio.

La segunda parte de este truco es la resolución del nombre de **paquete** en un directorio de la máquina, de forma que cuando se ejecuta un programa Java y necesita cargar el fichero **.class** (lo que ocurre dinámicamente, en el punto en el que el programa necesite crear un objeto de esa clase en particular, o la primera vez que se accede a un miembro **estático** de la clase), pueda localizar el directorio en el que reside el fichero **.class**.

El intérprete de Java procede de la siguiente forma. En primer lugar, encuentra la variable de entorno CLASSPATH (establecida mediante el sistema operativo, a veces por parte del programa de instalación de Java, o una herramienta basada en Java de la propia máquina). CLASSPATH contiene uno o más directorios utilizados como raíz para la búsqueda de ficheros **.class**. A partir de esa raíz, el intérprete toma el nombre de paquete y reemplaza cada punto por una barra para generar un

nombre relativo a la raíz CLASSPATH (de forma que el **paquete foo.bar.baz** se convierte en **foo\bar\baz** o en **foo/bar/baz** en función del sistema operativo instalado). A continuación, se concatena este nombre con las distintas entradas de la variable CLASSPATH. Es en este momento cuando se busca por el archivo **.class** que coincida en nombre con la clase que se está intentando crear (también busca algunos directorios estándares relativos al directorio en el que reside el intérprete Java).

Para entenderlo, considérese mi nombre de dominio, que es **bruceeckel.com**. Dando la vuelta a esto, **com.bruceeckel** establece el único nombre global a utilizar en todas mis clases. (La extensión com, edu, org, etc., se ponía en mayúsculas en las primeras versiones de los paquetes Java, pero esto se ha cambiado en Java 2, de forma que todo el nombre de paquete se escribe en minúsculas.) Posteriormente se puede subdividir este nombre diciendo que se quiere crear una biblioteca llamada **simple**, por lo que acabaremos con un nombre de paquete:

```
package com.bruceeckel.simple;
```

Ahora, este nombre de paquete puede usarse como un espacio de nombre paraguas para los siguientes dos archivos:

```
//: com:bruceeckel:simple:Vector.java
// Creando un paquete.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.bruceeckel.util.Vector");
  }
} ///:~
```

Cuando uno crea sus propios paquetes, se descubre que la sentencia **package** debe ser la primera del archivo de código que no sea un comentario dentro del archivo. El segundo archivo es muy parecido:

```
//: com:bruceeckel:simple:Lista.java
// Creando un paquete.
package com.bruceeckel.simple;

public class Lista {
  public Lista() {
    System.out.println(
      "com.bruceeckel.util.Lista");
  }
} ///:~
```

Ambos ficheros se encuentran ubicados en el subdirectorio:

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Si se empieza a recorrer esta trayectoria se puede componer el nombre de paquete **com.bruceeckel.simple**, pero ¿qué ocurre con la primera parte de la trayectoria? De esto se encarga la variable de entorno CLASSPATH:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Puede verse que la variable CLASSPATH puede contener más de un directorio de búsqueda, todos ellos alternativos.

Sin embargo, hay una variación cuando se usan archivos JAR. Se debe poner el nombre del archivo JAR en la trayectoria de clases CLASSPATH, no sólo la trayectoria en la que se encuentra. Así, para un JAR de nombre **uva.jar**, esta variable será:

```
CLASSPATH=.;D:\JAVA\LIB;C:\sabores\uva.jar
```

Una vez que se ha establecido correctamente el valor de esta variable, buscará el archivo en cualquiera de sus directorios:

```
//: c05:PruebaBiblioteca.java
// Utiliza la biblioteca.
import com.bruceeckel.simple.*;

public class PruebaBiblioteca {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

Cuando el compilador encuentra la sentencia **import**, empieza a buscar en los directorios especificados por CLASSPATH, buscando el subdirectorio com\bruceeckel\simple, y buscando después los ficheros compilados de nombres adecuados (**Vector.class** para **Vector** y **List.class** para **List**). (Fíjese que, tanto las clases, como los métodos deseados de **Vector** y **List,** deben ser **públicos**).

Establecer la variable CLASSPATH era tan problemático para los usuarios de Java principiantes (como lo era para mí cuando empecé) que Sun ha hecho el JDK de Java 2 algo más inteligente. Se descubrirá que, al instalarlo, incluso si no se establece un CLASSPATH, se podrán compilar y ejecutar programas básicos de Java. Para compilar y ejecutar el paquete código de este libro (disponible en el CD ROM empaquetado junto con este libro, o en *www. BruceEckel.com*), sin embargo, se necesitará hacer algunas modificaciones al CLASSPATH (éstas se explican en el paquete de código fuente).

Colisiones

¿Qué ocurre si se importan dos bibliotecas vía * que incluyen los mismos nombres? Por ejemplo, supóngase que un programa hace:

```
import com.bruceeckel.simple.*;
import java.util.*;
```

Dado que **java.util.*** también contiene una clase **Vector**, esto causa una colisión potencial. Sin embargo, mientras no se escriba el código que, de hecho, cause la colisión, todo va bien —esto es bueno porque de otra forma, uno podría acabar tecleando multitud de código para evitar colisiones que nunca ocurrirían.

La colisión *ocurre* si ahora se intenta crear un **Vector**:

```
Vector v = new Vector();
```

¿A qué **Vector** se refiere? El compilador no puede saberlo, y tampoco puede el lector. Por tanto el compilador se queja y obliga a especificar. Si se desea el **Vector** estándar de Java, por ejemplo, hay que decir:

```
java.util.Vector v = new java.util.Vector();
```

Dado que esto (junto con la variable CLASSPATH) especifica completamente la localización de ese **Vector**, no hay necesidad de la sentencia **import.java.util.***, a menos que se esté utilizando algo más de **java.util**.

Una biblioteca de herramientas a medida

Con estos conocimientos, ahora cada uno puede crear sus propias bibliotecas de herramientas para reducir o eliminar el código duplicado. Considérese, por ejemplo, que se está creando un alias para **System.out.println( )** para reducir el código a teclear. Éste podría ser parte de un paquete llamado **herramientas**:

```
//: com:bruceeckel:herramientas:P.java
// El atajo P.rint y P.rintln.
package com.bruceeckel.herramientas;

public class P {
  public static void rint(String s){
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
} ///:~
```

Se puede usar este atajo para usar un **String**, bien con un retorno de carro al final (**P.rintln( )**) o sin él (**P.rint( )**).

Se puede adivinar que este archivo debe estar ubicado en un directorio de los especificados en CLASSPATH, y que continúe por **com/bruceeckel/herramientas**. Una vez compilado, el fichero **P.class** puede usarse en cualquier lugar del sistema con una sentencia **import**:

```
//: c05:PruebaHerramienta.java
// Utiliza la biblioteca herramientas.
```

```
import com.bruceeckel.herramientas.*;

public class PruebaHerramienta {
  public static void main(String[] args) {
    P.rintln("¡Disponible de ahora en adelante!");
    P.rintln("" + 100); // Obligar a que sea un String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
  }
} ///:~
```

Obsérvese que se puede forzar a cualquier objeto a transformarse en una representación en forma de **String**, poniéndolos en una expresión **String**; en el caso anterior, se hace uso de un truco: comenzar la expresión con un **String** vacío. Pero esto recuerda una observación interesante. Si se invoca a **System.out.println(100)**, funciona sin tener que convertirlo a **String**. Con algo de sobrecarga, se puede conseguir que la clase **P** haga también esto (planteado como ejercicio al final del presente capítulo).

Por tanto, de ahora en adelante, cuando construya una nueva utilidad, se puede añadir al directorio **herramientas**. (O al directorio **util** o **herramientas** de cada uno.)

Utilizar el comando import para cambiar el comportamiento

Una característica que Java no ha heredado de C es la *compilación condicional*, que permite modificar un switch y obtener distintos comportamientos sin necesidad de variar ninguna otra parte del código. La razón por la que esta característica no se incluyó en Java es probablemente el hecho de que se utiliza en C fundamentalmente para resolver problemas de multiplataforma: se compilan distintas porciones de código en función de la plataforma para la que se está compilando cada código. Puesto que se pretende que Java sea multiplataforma automáticamente, una característica así no es necesaria.

Sin embargo, hay otros usos de gran valor en la compilación condicional. Un uso muy común es la depuración de código. Los aspectos de depuración se habilitan durante el desarrollo, y se deshabilitan en el lanzamiento del producto. A Allen Holub (*www.holub.com*) se le ocurrió la idea de utilizar paquetes para simular la compilación condicional. Hizo uso de esta idea para crear una versión Java del *mecanismo de afirmaciones* —tan útil en C—, mediante el que se puede decir "esto debería ser verdad" o "esto debería ser falso" y si la sentencia no está de acuerdo con el afirmación, ya se averiguará. Este tipo de herramienta supone una gran ayuda durante la fase de depuración.

He aquí la clase que se utilizará para depuración:

```
//: com:bruceeckel:herramientas:depurar:Afirmacion.java
// Herramienta de aserto para la depuración;
package com.bruceeckel.tools.debug;
```

```
public class Afirmacion {
  private static void error(String msg) {
    System.err.println(msg);
  }
  public final static void es_cierto(boolean exp) {
    if(!exp) error("Fallo la afirmacion");
  }
  public final static void es_falso(boolean exp) {
    if(exp) error("Fallo la afirmacion");
  }
  public final static void
        es_cierto(boolean exp, String mensaje){
    if (!exp) error("Fallo la afirmacion: " + mensaje);
  }
  public final static void
  es_falso(boolean exp, String msg) {
    if(exp) error("Fallo la afirmacion: " + mensaje);
  }
} ///:~
```

Esta clase simplemente encapsula pruebas de valores lógicos, que imprimen mensajes de error si fallan. En el Capítulo 10, se conocerá una herramienta más sofisticada para tratar con errores, denominada *manejo de excepciones*, pero el método **error( )** será suficiente mientras tanto.

La salida se imprime en el "flujo de datos" de la consola de *error estándar* escribiendo en **System.err**.

Cuando se desee hacer uso de esta clase, basta con añadir en el programa la línea:

```
import com.bruceeckel.herramientas.depurar.*;
```

Para retirar las afirmaciones que pueda lanzar el código, se crea una segunda clase **Afirmacion**, pero en un paquete distinto:

```
//: com:bruceeckel:herramientas:depurar:Afirmacion.java
// Desactivar la salida de la afirmacion
// de forma que se pueda lanzar el programa.
Package com.bruceeckel.herramientas;

public class Afirmacion {
  public final static void es_cierto(boolean exp) {}
  public final static void es_falso(boolean exp) {}
  public final static void
  es_cierto(boolean exp, String mensaje) {}
  public final static void
  es_falso(boolean exp, String mensaje) {}
} //:~
```

Ahora, si se cambia la sentencia **import** anterior a:

```
import com.bruceeckel.herramientas.*;
```

El programa dejará de imprimir afirmaciones. He aquí un ejemplo:

```
//: c05:PruebaAfirmacion.java
// Demostrando la herramienta de afirmación.
// Comentar y quitar el comentario
// de la línea siguiente para cambiar
// el comportamiento del aserto:
import com.herramientas.depurar.debug.*;
// import com.bruceeckel.herramientas.*;

public class PruebaAfirmacion {
  public static void main(String[] args) {
    afirmacion.es_cierto((2 + 2) == 5);
    afirmacion.es_falso((1 + 1) == 2);
    afirmacion.es_cierto((2 + 2) == 5, "2 + 2 == 5");
    afirmacion.es_falso((1 + 1) == 2, "1 +1 != 2");
  }
} ///:~
```

Al cambiar el **paquete** que se importa, se cambia el código de la versión en depuración a la versión de producción. Esta técnica puede usarse para cualquier tipo de valor condicional.

Advertencia relativa al uso de paquetes

Merece la pena recordar que cada vez que se cree un paquete, implícitamente se está especificando una estructura de directorios al dar un nombre a un paquete. El paquete *debe* residir en el directorio indicado por su nombre, que debe ser un directorio localizable a partir de CLASSPATH.

Experimentar con la palabra clave **package**, puede ser un poco frustrante al principio, puesto que, a menos que se adhiera al nombre del paquete la regla de trayectorias de directorios, se obtendrán numerosos mensajes en tiempo de ejecución que indican que no es posible localizar una clase en particular, incluso si esa clase reside en ese mismo directorio. Si se obtiene uno de estos mensajes, debe tratar de modificar la sentencia *package*, y cuando funcione se sabrá dónde residía el problema.

Modificadores de acceso en Java

Al utilizarlos, los modificadores de acceso **public**, **protected** y **private** se ubican delante de cada definición de cada miembro de la clase, sea un atributo o un método. Cada modificador de acceso controla el acceso sólo para esa definición en particular. Éste es diferente a C++, lenguaje en el que

el controlador de acceso controla todas las definiciones que lo sigan hasta la aparición del siguiente modificador de acceso.

De una manera u otra, todo tiene asignado algún tipo de modificador de acceso. En las secciones siguientes, se aprenderán los distintos tipos de accesos, comenzando por el acceso por defecto.

"Amistoso" ("Friendly")

¿Qué ocurre si no se indica ningun tipo de especificador de acceso, como en todos los ejemplos anteriores de este capítulo? El acceso por defecto no tiene ninguna palabra clave asociada, pero generalmente se le denomina acceso "amistoso". Significa que todas las demás clases del paquete actual tienen acceso al miembro amistoso, pero de cara a todas las clases de fuera del paquete, el miembro aparenta ser **privado**. Dado que una unidad de compilación —un fichero— puede pertenecer sólo a un único paquete, todas las clases de una única unidad de compilación son automáticamente "Amistosas" entre sí. Por consiguiente, se dice que los elementos "Amistosos" tienen *acceso paquete*.

El acceso amistoso permite agrupar clases relacionadas en un mismo paquete de forma que éstas puedan interactuar entre sí de manera sencilla. Al poner clases juntas en un paquete (garantizando por consiguiente el acceso mutuo "Amistoso" a sus miembros; por ejemplo, marcándolos como amigos) se "posee" el código de ese paquete. Tiene sentido que el único código que se posee debería tener acceso amistoso al resto de código propio. Podría decirse que el acceso amistoso da un significado o razón para agrupar juntas las clases de un paquete. En muchos lenguajes, la forma de organizar las definiciones de los ficheros puede ser obligatoria, pero en Java obliga a que cada uno las organice de manera sensata. Además, probablemente se excluirán las clases que no deberían tener acceso a las clases que están siendo definidas en el paquete actual.

La clase controla qué código tiene acceso a sus miembros. No hay ningún truco para "irrumpir" en ella. No se puede mostrar el código de otros paquetes y decir: "¡Hola, soy un amigo de **Bob**!", y esperar que se vean los miembros **protegidos**, "amistosos", y **privados** de **Bob**. La única manera de garantizar los accesos a un miembro es:

1. Hacer el miembro **público**. Posteriormente, todo el mundo, en todas partes, podría acceder a él.

2. Hacer el miembro amistoso no indicando ningún especificador de acceso, y poner las otras clases en el mismo paquete. Así, las otras clases pueden acceder al miembro.

3. Como se verá en el Capítulo 6, cuando se presente la herencia, una clase heredada puede acceder a un miembro **protegido** al igual que a un miembro **público** (pero no a los miembros **privados**). Puede acceder a los miembros "amistosos" sólo si las dos clases se encuentran en el mismo paquete. Pero no hay que preocuparse de esto ahora.

4. Proporcionar métodos "obtener/establecer" *("get/set")* que lean y cambien el valor. Éste es el enfoque más habitual en términos de POO, y es fundamental para los JavaBeans como se verá en el Capítulo 13.

public: acceso a interfaces

Cuando se usa la palabra clave **public**, significa que la declaración de miembro que continúe inmediatamente a **public** estará disponible a todo el mundo, y en especial al programador cliente que hace uso de la biblioteca. Supóngase que se define un paquete **postre**, que contiene la siguiente unidad de compilación:

```
//: c05:postre:Galleta.java
// Crea una biblioteca.
package c05.postre;

public class Galleta {
  public Galleta() {
    System.out.println("Constructor de Galleta");
  }
  void morder() { System.out.println("morder"); }
} ///:~
```

Recuérdese que **Galleta.java** debe residir en un subdirectorio de nombre **postre**, en un directorio bajo **c05** (que se corresponde con el Capítulo 5 de este libro) que debe estar bajo uno de los directorios de CLASSPATH. No hay que cometer el error de pensar que Java siempre buscará en el directorio actual como uno de los directorios de partida para la búsqueda. Si no se tiene un '.' como una de las rutas del CLASSPATH, Java no buscará ahí.

Ahora, si se crea un programa que haga uso de **Galleta**:

```
//: c05:Cena.java
// Hace uso de la biblioteca.
import c05.postre.*;

public class Cena {
  public Cena() {
    System.out.println("Constructor cena");
  }
  public static void main(String[] args) {
    Galleta x = new Galleta();
    //! x.morder(); // No se puede acceder
  }
} ///:~
```

se puede crear un objeto **Galleta**, dado que su constructor es **público** y la clase es **pública**. (Se profundizará más tarde en el concepto de **público**.) Sin embargo, el método **morder( )** es inaccesible dentro de **Cena.java** puesto que **morder( )** es amistoso sólo dentro del paquete **postre**.

El paquete por defecto

Uno podría sorprenderse de descubrir que el código siguiente compila, incluso aunque aparenta transgredir las reglas:

```
//: c05:Tarta.java
// Accede a una clase de una
// unidad de compilación distinta.

class Tarta {
  public static void main(String[] args) {
    Pastel x = new Pastel();
    x.f();
  }
} ///:~
```

En un segundo archivo del mismo directorio:

```
//: c05:Pastel.java
// La otra clase.

class Pastel {
  void f() { System.out.println("Pastel.f()"); }
} ///:~
```

Inicialmente uno podría pensar que se trata de archivos completamente independientes, y sin embargo, **Tarta** es incluso capaz de crear un objeto **Pastel**, e invocar a su método **f( )**. (Fíjese que debe tenerse el '.' en CLASSPATH para que los archivos se compilen.) Normalmente se pensaría que **Pastel** y **f( )** son amistosos y por consiguiente, no están disponibles para **Tarta**. *Son* amistosos —hasta ahí es correcto. La razón por la que están disponibles en **Tarta.java** es que se encuentran en el mismo directorio y no tienen ningún nombre de paquete explícito. Java trata a los archivos así como si fueran parte implícita del "paquete por defecto" de ese directorio, y por consiguiente, amistoso para el resto de ficheros del directorio.

private: ¡eso no se toca!

La palabra clave **private** significa que nadie puede acceder a ese miembro excepto a través de los métodos de esa clase. Otras clases del mismo paquete no pueden acceder a miembros **privados**, de forma que es como si se estuviera incluso aislando la clase contra uno mismo. Por otro lado, no es improbable que un paquete esté construido por varias personas que colaboran juntas, de forma que **privado** permite cambiar libremente ese miembro sin necesidad de preocuparse de si el cambio influirá a otras clases del mismo paquete.

El acceso "amistoso" al paquete por defecto proporciona un nivel de ocultación bastante elevado; recuerde, un miembro "amistoso" es inaccesible para el usuario del paquete. Esto está bien, dado que el acceso por defecto es el que se usa normalmente (y el que se lograría si se olvida añadir

algún control de acceso). Por consiguiente, uno generalmente pensaría en lo referente al acceso a los miembros de un programa, que habría que hacer éstos explícitamente **públicos**, y como resultado, puede que inicialmente no se piense en usar la palabra clave **private** a menudo. (Lo cual es distinto en C++.) Sin embargo, resulta que el uso consistente de **private** es muy importante, especialmente cuando está involucrada la ejecución multihilo. (Como se verá en el Capítulo 14.)

He aquí un ejemplo del uso de **private**:

```
//: c05:Helado.java
// Demuestra el uso de la palabra clave "private".

class Vainilla {
  private Vainilla() {}
  static Vainilla prepararVainilla() {
    return new Vainilla();
  }
}

public class Helado {
  public static void main(String[] args) {
    //! Vainilla x = new Vainilla();
    Vainilla x = Vainilla.prepararVainilla();
  }
} ///:~
```

Esto muestra un ejemplo de cómo el modificador **privado** resulta útil: se podría querer controlar cómo se crea un objeto y evitar que alguien pueda acceder directamente a un constructor particular (o a todos ellos). En el ejemplo de arriba, no se puede crear un objeto **Vainilla** a través de su constructor; por el contrario, debe invocarse al método **prepararVainilla( )**[2].

Puede declararse **privado** cualquier método del que tengamos la seguridad de que no es más que un método "ayudante" para esa clase, para asegurar que no se use accidentalmente en ningún otro lugar del paquete, y por consiguiente, prohibir a uno mismo cambiar o eliminar el método. Construir un método **privado** garantiza que se conserve esta opción.

Lo mismo es válido para un campo **privado** dentro de una clase. A menos que se deba exponer la implementación subyacente (lo cual es una situación mucho más rara de lo que se podría pensar), deberían hacerse **privados** todos los campos. Sin embargo, sólo porque una referencia a un objeto sea **privado** dentro de su clase, no es imposible que cualquier otro objeto pueda tener una referencia **pública** al mismo objeto. (Apéndice A para aspectos relativos al "uso de alias".)

---

[2] Hay otro efecto en este caso: dado que el constructor por defecto es el único definido, y éste es **privado**, evitará la herencia de esta clase. (Un aspecto que se detallará en el Capítulo 6.)

protected: "un tipo de amistades"

Entender el especificador de acceso **protegido** supone ir algo más allá. En primer lugar, uno debería ser consciente de que no necesita entender esta sección para continuar a lo largo de este libro hasta llegar a la herencia (Capítulo 6). Pero de manera comparativa, he aquí una breve descripción y ejemplo utilizando **protected**.

La palabra clave **protected** está relacionada con un concepto denominado *herencia*, que toma una clase existente y le añade nuevos miembros sin tocar la clase ya existente, a la que se denomina *clase base*. También se puede cambiar el comportamiento de los miembros existentes de la clase. Para heredar de una clase existente, se dice que la nueva clase **hereda** de una ya existente, como:

```
class FOO extends Bar {
```

El resto de la definición de la clase es exactamente igual.

Si se crea un nuevo paquete y se hereda desde una clase de otro paquete, los únicos miembros a los que se tiene acceso son los miembros **públicos** del paquete original. (Por supuesto, si se lleva a cabo la herencia dentro del *mismo* paquete, se tiene el acceso de paquete normal a todos los miembros "amistosos".) Algunas veces, el creador de la clase base desea tomar un miembro particular y garantizar el acceso a las clases derivadas, pero no a todo el mundo. Esto es lo que hace el modo **protegido**. Si se hiciera referencia de nuevo al fichero **Galleta.java**, la siguiente clase *no puede* acceder al miembro "amistoso":

```
//: c05:GalletaChocolate.java
// No puede acceder a un miembro amistoso.
// de otra clase.
import c05.postre.*;

public class GalletaChocolate extends Galleta {
  public GalletaChocolate() {
    System.out.println(
      "Constructor de GalletaChocolate");
  }
  public static void main(String[] args) {
    GalletaChocolate x = new GalletaChocolate();
    //! x.morder(); // No se puede acceder a morder.
  }
} ///:~
```

Una de las cosas más interesantes de la herencia es que si existe un método **morder( )** en la clase **Galleta**, también existe en cualquier clase heredada de **Galleta**. Pero dado que **morder( )** es "amistoso" para los otros paquetes, no podrá utilizarse en éstos. Por supuesto, se puede hacer que sea **público**, pero entonces todo el mundo tendría acceso y quizás eso no es lo que se desea. Si se cambia la clase **Galleta**, como sigue:

```
public class Galleta {
```

```
  public Galleta() {
    System.out.println("Constructor de galletas");
  }
  protected void morder() {
    System.out.println("morder");
}
} ///:~
```

entonces **morder( )** sigue teniendo acceso "amistoso" dentro del paquete **postre**, pero también es accesible a cualquiera que herede de **Galleta**. Sin embargo, *no* es **público**.

Interfaz e implementación

El control de accesos se suele denominar *ocultación de la información*. Al hecho de envolver datos y miembros dentro de las clases, en combinación con el ocultamiento de la información, se le suele denominar *encapsulación*[3]. El resultado es un tipo de datos con sus propias características y comportamientos.

El control de accesos pone límites dentro de un tipo de datos por dos razones importantes. La primera es establecer qué es lo que pueden y lo que no pueden usar los programadores cliente. Se pueden construir los mecanismos internos dentro de la estructura sin tener que preocuparse de que los programadores clientes traten de manipular accidentalmente las interioridades como parte de la interfaz, que es lo que deberían estar usando.

Esto nos presenta directamente en la segunda razón, que es separar la interfaz de la implementación. Si la estructura se utiliza en un conjunto de programas, los programadores clientes no pueden hacer nada más que enviar mensajes al interfaz **público**, entonces es posible cambiar cualquier cosa que *no* sea **público** (por ejemplo, "amistoso", **protegido** o **privado**) sin necesidad de requerir modificaciones en el código cliente.

Ahora nos encontramos en el mundo de la programación orientada a objetos, donde una **clase** describe, de hecho, "una clase de objetos", tal y como se describiría una clase de pescados o una clase de pájaros. Cualquier objeto que pertenezca a esta clase compartirá estas características y comportamientos. La clase es una descripción de lo que parecen y de cómo se comportan los objetos de este tipo.

En el lenguaje original de POO, Simula-67, la palabra clave **class** se utilizaba para describir un nuevo tipo de datos. La misma palabra clave se ha venido utilizando en la mayoría de lenguajes orientados a objetos. Éste es el punto más importante de todo el lenguaje: la creación de nuevos tipos de datos que son más que simples cajas contenedoras de datos y métodos.

La clase es el concepto fundamental en Java. Es una de las palabras clave que *no* se pondrá en negrita en este libro —pues resultaría molesto hacerlo con una palabra que se repite tan a menudo.

---

[3] Sin embargo, la gente suele denominar "encapsulación" únicamente al ocultamiento de información.

Por claridad, puede que se prefiera un estilo de creación de clases que ponga los miembros **públicos** al principio, seguidos de los miembros **protegidos**, amistosos y **privados**. La ventaja es que el usuario de la clase puede ir leyendo de arriba hacia abajo y ver primero lo que más le importa (los miembros **públicos**, que es a los que puede acceder desde fuera del archivo) y dejar de leer cuando encuentre los miembros no **públicos**, que son parte de la implementación interna.

```
public class x {
  public void pub1() { /* . . . */}
  public void pub2() { /* . . . */}
  public void pub3() { /* . . . */}
  private void priv1() { /* . . . */ }
  private void priv2() { /* . . . */ }
  private void priv3() { /* . . . */ }
  private int i;
  // . . .
}
```

Esto la hará simplemente un poco más fácil de leer, puesto que la interfaz y la implementación siguen estando entremezclados. Es decir, sigue siendo necesario ver el código fuente —la implementación, porque está justo ahí, dentro de la clase. Además, la documentación en forma de comentarios soportada por javadoc (descrito en el Capítulo 2) resta la importancia de la legibilidad del código para el programador cliente. Mostrar la interfaz al consumidor de una clase es verdaderamente el trabajo del *navegador de clases* o *class browser*, una herramienta cuyo trabajo es mirar en todas las clases disponibles y mostrar lo que se puede hacer con ellas (por ejemplo, qué miembros están disponibles) de forma útil. Para cuando se lea el presente texto, cualquier buena herramienta de desarrollo Java debería incluir este tipo de navegadores.

Acceso a clases

En Java, los especificadores de acceso pueden usarse también para determinar qué clases estarán disponibles *dentro* de una biblioteca para los usuarios de esa biblioteca. Si se desea que una clase esté disponible para un programador cliente, se coloca la palabra clave **public** en algún lugar antes de la llave de apertura del cuerpo de la clase. Esto controla si el programador cliente puede incluso crear objetos de esa clase.

Para controlar el acceso a una clase, debe aparecer el especificador antes de la palabra clave **class**. Por consiguiente, se puede decir:

```
public class Componente {
```

Ahora, si el nombre de la biblitoeca es **mibiblioteca**, cualquier programador cliente puede acceder a **Componente** diciendo

```
import mibiblioteca.Componente;
```

o

```
import mibiblioteca.*;
```

Sin embargo, hay un conjunto de restricciones extra:

1. Solamente puede haber una clase **pública** por cada unidad de compilación o fichero. La idea es que cada unidad de compilación tenga una única interfaz pública representada por esa clase **pública**. Puede tener tantas clases "amistosas" de soporte como se desee. Si se quiere tener más de una clase **pública** dentro de una unidad de compilación, el compilador mostrará un mensaje de error.

2. El nombre de la clase **pública** debe coincidir exactamente con el nombre del archivo que contenga la unidad de compilación, incluyendo las mayúsculas. Por tanto, para **componente**, el nombre del archivo debe ser **Componente.java** y no **componente.java** o **COMPONENTE.java**. De nuevo, si éstos tampoco coinciden, se obtendrá también un error de tiempo de compilación.

3. Es posible, aunque no habitual, que exista alguna unidad de compilación sin ninguna clase **pública**. En este caso, se puede dar al archivo el nombre que se desee.

¿Qué ocurre si se tiene una clase dentro de **mibiblioteca** que se está utilizando para llevar a cabo las tareas que hace **Componente** o cualquier otra clase **pública** de **mibiblioteca**? Nadie desea llegar hasta el punto de tener que crear documentación para el programador cliente, y pensar que algún tiempo después podría desearse cambiar completamente las cosas y arrancar todas esas clases, para sustituirlas por otras. Para tener esta flexibilidad, hay que asegurar que ningún programador cliente se vuelva dependiente de unos detalles de implementación particulares incluidos dentro de **mibiblioteca**. Para lograr esto, simplemente se quita la palabra **public** de la clase, en cuyo caso se convierte en "amistosa". (La clase puede usarse únicamente dentro de ese paquete.)

Fíjese que una clase no puede ser **privada** (pues esto la convertiría en inaccesible para alguien que no sea la propia clase), ni **protegida**[4]. Por tanto, sólo se tienen dos opciones para los accesos a clases: "amistosa" o **pública**. Si no se desea que nadie más tenga acceso a esa clase, se pueden hacer todos los constructores **privados**, evitando así que nadie más que uno mismo pueda crear un objeto de esa clase[5], dentro de un miembro **estático** de la clase. He aquí un ejemplo:

```
//: c05:Almuerzo.java
// Muestra el funcionamiento de los especificadores de acceso a clases.
// Hace una clase verdaderamente privada
// con constructores privados:

class Sopa {
  private Sopa() {}
  // (1) Permitir la creación a través de un método estático:
  public static Sopa hacerSopa() {
    return new Sopa();
  }
  // (2) Crear un objeto estático nuevo y
  // devolver una referencia bajo demanda.
```

[4] De hecho, una *clase interna* puede ser **privada** o **protegida**, pero se trata de un caso especial. Éstos se presentarán en el Capítulo 7.

[5] También se puede hacer esto por herencia (Capítulo 6) desde esa clase.

```
  // (El patrón "singular"):
  private static Sopa ps1 = new Sopa();
  public static Sopa acceso() {
    return ps1;
  }
  public void f() {}
}

class Bocadilo { // Usa Almuerzo
  void f() { new Almuerzo(); }
}

// Sólo se permite una clase pública por fichero:
public class Almuerzo {
  void prueba() {
    // ¡Esto no se puede hacer! Constructor privado:
    //! Sopa priv1 = new Sopa();
    Sopa priv2 = Sopa.hacerSopa();
    Bocadillo f1 = new Bocadillo();
    Sopa.acceso().f();
  }
} ///:~
```

Hasta ahora, la mayoría de métodos devolvían **void** o un tipo primitivo, por lo que la definición:

```
public static Sopa acceso() {
  return ps1;
}
```

podría parecer algo confusa a primera vista. La palabra antes del nombre del método (**acceso**) indica qué devuelve el método. Hasta la fecha, ésta ha sido la mayoría de las veces vacía (**void**), que quiere decir que no se devuelve nada. Pero también se puede devolver una referencia a un objeto, que es lo que ocurre aquí. Este método devuelve una referencia a un objeto de la clase **Sopa**.

La **clase Sopa** muestra como evitar la creación directa de una clase haciendo **privados** todos los constructores. Recuérdese que si no se crea al menos un constructor explícitamente, se creará automáticamente el constructor por defecto (un constructor sin parámetros). Si se escribiera el constructor por defecto, éste no se creará automáticamente. Al hacerlo **privado**, nadie puede crear un objeto de esa clase. Pero ahora ¿cómo puede alguien usarla? El ejemplo de arriba presenta dos opciones. En primer lugar se crea un método **estático** que crea un nuevo objeto **Sopa** y devuelve una referencia al mismo. Esto podría ser útil si se desea hacer alguna operación extra con la **Sopa** antes de devolverla, o si se desea mantener la cuenta de cuántos objetos **Sopa** crear (quizás para restringir la población de objetos de este tipo).

La segunda opción usa lo que se denomina un *patrón de diseño*, que se describe en *Thinking in Patterns with Java*, descargable de *www.BruceEckel.com*. Este patrón en particular se denomina un "singular", porque sólo permite la creación de un único objeto. El objeto de clase **Sopa** se crea como

un miembro **estático privado** de **Sopa**, por lo que hay uno y sólo uno, y solamente se puede conseguir a través del método **público** de nombre **acceso( )**.

Como se mencionó previamente, si no se desea poner un modificador de acceso para el acceso a una clase, éste es por defecto "amistoso". Esto significa que cualquier otra clase del paquete puede crear un objeto de esa clase, pero no desde fuera del paquete. (Recuérdese que todos los archivos del mismo directorio que no tengan declaraciones explícitas de **paquete** son implícitamente parte del paquete por defecto de ese directorio.) Sin embargo, si un miembro **estático** de esa clase es **público**, el programador cliente puede seguir accediendo al miembro **estático** incluso aunque no pueda crear un objeto de esa clase.

Resumen

En cualquier relación es importante tener unos límites que sean respetados por todas las partes involucradas. Cuando se crea una biblioteca, se establece una relación con el usuario de esa biblioteca —el programador cliente— que es otro programador, que en vez de esto, se encarga unir diverso código para construir una aplicación, o bien de utilizar su biblioteca para construir una aplicación aún más grande.

Sin reglas, los programadores cliente pueden hacer lo que quieran con todos los miembros de una clase, incluso si se desea que no manipulen directamente algunos de estos miembros. Todo aparece desnudo al mundo.

Este capítulo revisaba cómo se construyen clases a partir de bibliotecas; en primer lugar, se explica cómo se empaquetan clases dentro de una biblioteca, y en segundo, cómo controla la clase el acceso a sus miembros.

Se estima que un proyecto de programación en C se empieza a romper entre las 50K y las 100K líneas porque C tiene un único "espacio de nombres" y los nombres empiezan a colisionar, causando una sobrecarga extra de gestión. En Java, la palabra clave **package**, el esquema de nombrado de paquetes *(package)* y la palabra clave *import* dan un control completo sobre los nombres, de manera que se evita de manera sencilla el aspecto de posibles colisiones entre nombres.

Hay dos razones por las que controlar el acceso a los miembros. El primero es mantener las manos de los usuarios alejadas de lo que no deberían tocar; las herramientas que son necesarias para las maquinaciones internas de los tipos de datos, pero no forman parte de la interfaz que los usuarios necesitan para resolver sus problemas. Por tanto, hacer los métodos y campos **privados** es un servicio a los usuarios porque pueden ver fácilmente qué es importante para ellos y qué pueden ignorar. Esto simplifica su grado de entendimiento de la clase.

La segunda y más importante razón para controlar el acceso es permitir al diseñador de bibliotecas cambiar los funcionamientos internos de la clase sin tener que preocuparse de cómo afectará esto al programador cliente. Uno podría construir una clase inicialmente de una forma, y después descubrir que reestructurando el código se logra un aumento considerable de velocidad. Si la interfaz y la implementación están claramente separados y protegidos, se puede acometer este cambio sin forzar al usuario a reescribir su código.

Los modificadores de accesos dan en Java un control muy valioso al creador de la clase. Los usuarios de la clase pueden ver clara y exactamente qué es lo que pueden usar y qué ignorar. Y lo que es más importante, la capacidad para asegurar que ningún usuario se vuelva dependiente de ninguna parte de la implementación subyacente de una clase. Si se conoce ésta, como creador de la misma, se puede cambiar la implementación subyacente con el conocimiento de que ningún programador cliente se verá afectado por los cambios, pues éstos no pueden acceder a esa parte de la clase.

Cuando se tiene la capacidad de cambiar la implementación subyacente, no sólo se puede mejorar su diseño más tarde, sino que también se tiene la libertad de cometer errores. Sin que importe lo cuidadosamente que se haga la planificación y el diseño, se cometerán errores. Sabiendo que cometer estos errores significan seguro que uno experimentará más, aprenderá mejor y acabará antes su proyecto.

La interfaz pública de una clase es lo que el usuario *de hecho, ve*, de forma que conseguir que es lo más importante de una clase es acabar haciéndola "bien" durante el análisis y el diseño. E incluso eso permite alguna libertad de acción de cara al cambio. Si no se logra una interfaz la primera vez, se pueden *añadir* nuevos métodos, siempre que no se elimine ninguno que los programadores cliente se hayan podido usar en sus códigos.

Ejercicios

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

1. Escribir un programa que cree un objeto **ListaArray** sin exportaciones explícitas de **java.util.***.
2. En la sección "El paquete: la unidad de biblioteca", cambiar los códigos de fragmento relacionados con **mipaquete** en un conjunto de ficheros Java compilables y ejecutables.
3. En la sección "Colisiones", cambiar los fragmentos de código por un programa, y verificar que verdaderamente se dan colisiones.
4. Generalizar la clase **P** definida en este capítulo añadiendo todas las versiones sobrecargadas de **rint( )** y **rintln( )** necesarias para manejar todos los tipos básicos de Java.
5. Cambiar la sentencia import de **PruebaAfirmacion.java** para habilitar y deshabilitar el mecanismo de afirmaciones.
6. Crear una clase con miembros de datos y métodos **públicos**, **privados**, **protegidos** y "amistosos". Crear un objeto de esa clase y ver qué tipo de mensajes de compilación se obtienen al intentar acceder a todos los miembros de la clase. Ser conscientes de que las clases del mismo directorio son parte del paquete "por defecto".
7. Crear una clase con datos **protegidos**. Crear una segunda clase en el mismo archivo con un método que manipule los datos **protegidos** de la primera clase.

8. Cambiar la clase **Galleta** como se especifica en la sección "**protected**: «tipo de amistad»". Verificar que **morder( )** no es **público**.

9. En la sección titulada "Acceso a clases" se encontrarán fragmentos de código que describen **mibilioteca** y **Componente**. Crear esta biblioteca y posteriormente crear un **Componente** en una clase que no sea parte del paquete **mibiblioteca**.

10. Crear un nuevo directorio y editar la variable CLASSPATH para que incluya ese nuevo directorio. Copiar el archivo **P.class** (producido al compilar **com.burceeckel.herramientas.P.java**) al nuevo directorio y cambiar los nombres del fichero, la clase **P** y los nombres de los métodos. (A lo mejor también se desea añadir alguna salida adicional para ver cómo funciona.) Crear otro programa en un directorio diferente que haga uso de la nueva clase.

11. Siguiendo la forma del ejemplo **Almuerzo.java**, crear una clase denominada **GestorConexion** que gestione un array fijo de objetos **Conexión**. El programador cliente no debe ser capaz de crear explícitamente objetos **Conexión**, sino que solamente puede crear objetos a través de un método estático de **GestorConexion**. Cuando el **GestorConexion** se quede sin objetos, devolverá una referencia **null**. Probar las clases de **main( )**.

12. Crear el siguiente fichero en el directorio c05/local (presumiblemente en el CLASSPATH):

```
///: c05:local:ClaseEmpaquetada.java
package c05.local;
class ClaseEmpaquetada {
  public ClaseEmpaquetada() {
    System.out.println (
      "Creando una clase empaquetada");
  }
}///:~
```

Posteriormente, crear el directorio siguiente en un directorio distinto:

```
///: c05:exterior:Exterior.java
package c05.exterior;
import c05.local.*;
public class Exterior {
  public static void main (String[] args) {
      ClaseEmpaquetada ce = new ClaseEmpaquetada();
  }
} ///:~
```

Explicar por qué el compilador genera un error. ¿Hacer que la clase **Exterior** sea parte del paquete **c05.local** cambiaría algo?