5:Ocultando la Implementación

Una consideración fundamental en el diseño orientado a objetos es "separar las cosas que cambian de las cosas que permanecen igual".

Esto es especialmente importante para las librerías. El usuario *(programador cliente)* de esa librería tiene que confiar en la parte que usa, y saber que no necesitará reescribir código si una nueva versión de la librería aparece. En la cara opuesta, el creador de la librería debe tener libertad para realizar modificaciones y mejoras con la seguridad que el código de los programadores no será afectado por esos cambios.

Esto se puede conseguir gracias a la convención. Por ejemplo, el programador de la librería debe estar de acuerdo en no eliminar métodos existentes cuando modifique una clase de la librería, ya que eso estropearía el código del programador cliente. Sin embargo, la situación contraria es más espinosa. En el caso de un dato miembro, ¿cómo puede saber el creador de la librería que datos miembros han sido accedidos por los programadores cliente? Esto también ocurre con métodos que son sólo parte de la implementación de una clase, y no significa que sean usados directamente por el programador cliente. Pero, y si el creador de la librería quiere deshacerse de una vieja implementación y poner una nueva? Cambiar cualquiera de esos miembros podría estropear el código del programador cliente. Por tanto, el creador de la librería lleva una camisa de fuerza y no puede cambiar ninguna cosa.

Para solucionar este problema, Java proporciona *especificadores de acceso* que permiten al creador de la librería decir que está disponible al cliente programador, y que no está. Los niveles de control de acceso son, en orden creciente de permisividad, **public** (público), **protected** (protegido), "friendly" (amiga, no tiene palabra reservada) y **private** (privado). Del párrafo anterior se podría pensar que el diseñador de librerías mantendrá todo cuanto sea posible como privado (private) y dejar accesibles los métodos que quiere que el cliente programador use. Esto es totalmente correcto, incluso aunque frecuentemente no sea intuitivo para la gente que programaba en otros lenguajes (especialmente C) y accedía a todo sin restricciones. Al final del capítulo, usted deberá ser convencido de la importacncia del control de acceso en Java.

No obstante, el concepto de librería de componentes y el control sobre quien puede acceder a los componentes de esa librería no está completo. Todavía está la pregunta de cómo los integrantes son envueltos en una unidad de librería (library unit). Esto se controla con la palabra reservada **package** en Java, y los especificadores de acceso influyen en si una clase está en el mismo paquete o en

paquetes separados. Asi que para empezar este capítulo, aprenderá cómo los integrantes de la librería son alojados en paquetes. Luego, será capaz de comprender el significado completo de los especificadores de acceso.

paquetes: la librería unidad

Un paquete (package) es lo que se obtiene cuando usa la palabra reservada **import** para importar una librería entera, tal como la instrucción

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

Esto importa la librería de utilidades que es parte de la distribución estándar de Java. Ya que, por ejemplo, la clase **ArrayList** está en **java.util** , usted ahora puede especificiar el nombre completo **java.util.ArrayList** (lo cual puede hacer sin la sentencia import), o simplemente decir **ArrayList** (debido al **import**).

Si quiere importar sólo una clase, puede nombrar esa clase en la sentencia **import**

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

Ahora puede usare ArrayList sin limitación. Sin embargo, ninguna de las otras clases en java.util puede usarse.

La razón de hacer esto es proporcionar un mecanismo que gestione los "espacios de nombres". Los nombres de todos los miembros de clase están aislados unos de otros. Un método **f( )** dentro de una clase A no colisionará con **f( )** que tenga la misma signatura (lista de argumentos) de una clase B. Pero, Qué ocurre con el nombre de las clases? Suponga que crea una clase **stack** que está instalada en una máquina que ya tiene otra clase **stack** escrita por otra persona. Con Java en Internet, esto puede ocurrir sin que el usuario lo sepa, ya que las clases pueden descargarse automáticamente en el proceso de arranque de un programa en Java.

Esta posible colisión de nombres es la causa de la importancia de tener control completo sobre los nombres de espacio en Java, y ser capaz de crear un nombre completo único a pesar de todas las restricciones de Internet.

Además, la mayor parte de los ejemplos de este libro han estado en un solo fichero y han sido diseñados para uso local, y no se preocupan de los nombre de paquetes. (En este caso el nombre de la clase es situado en el "package default"). Esto es con certeza una opción, y por simplicidad esta aproximación será usada cuando sea posible durante el resto de l libro. Sin embargo, si está planeando crear librerias o programas que son "amigos" con otros programas Java en la

misma máquina, debe preocuparse de evitar conflictos de nombres de clases.

Cuando crea un fichero de código fuente para Java, se le conoce comúnmente como *unidad de compilación* (a veces unidad de translación). Cada unidad de compilación tiene que tener un nombre terminado **.java** , y dentro de la unidad de compilación puede haber una clase **public** que debe tener el mismo nombre que el fichero (incluso si está en mayúsculas, aunque no hay que añadir la extensión **.java** al nombre de la clase). Puede haber sólo una clase **public** en cada unidad de compilación, de otro modo el compilador dará error. El resto de las clases en esa unidad de compilación, si hay otras, estarán ocultas del mundo exterior a ese paquete, porque *no* son **public** , y comprenden clases de "apoyo" para la clase principal **public** (pública).

Cuando compila un fichero **.java** , obtiene otro fichero de salida con el mismo nombre pero de extensión **.class** *para cada clase en el fichero* **.java** . Así, puede terminar con bastantes ficheros **.class** de un pequeño número de ficheros **.java** files. Si ha programado con un lenguaje compilado, tal vez el compilador generaba una forma intermedia (normalmente un fichero "obj") que es luego empaquetado junto con otros de su mismo tipo usando un enlazador (para crear un fichero ejecutable) o un generador de librerías (para crear una librería). Así no es como funciona Java. Un programa funcionando es un grupo de ficheros **.class** , que pueden ser empaquetados y comprimidos dentro de un fichero JAR (usando el archivador **jar** de Java).

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

Una librería es también un grupo de ficheros class. Cada fichero tiene una clase que es public (no está obligado de tener una clase public class, pero es lo típico), por lo que hay un componente para cada fichero. Si quiere decir que todos esos componentes (que están en su correspondiente fichero .java y .class) belong together, that's where the package keyword comes in.

Cuando dice:

```
package mypackage;
```

al comienzo del fichero (si usa una sentencia **package** , debe aparecer al principio sin comentarios en el fichero), está indicando que esta unidad de compilación es parte de una librería denominada **mypackage** . O, dicho de otra forma, está diciendo que la clase **public** dentro de esta unidad de compilación está bajo la sombrilla del nombre **mypackage** , y si alguien quiere usarla tienen que especificar completamente el nombre de la clase o usar la palabra reservada **import** en combinación con **mypackage** (usando las instrucciones dadas anteriormente). Note que la convención para los nombres de paquete en java es

usar minúsculas, incluso para palabras intermedias.

Por ejemplo, supongamos que el nombre del fichero es **MyClass.java** . Esto quiere decir que puede haber una y solo una clase **public** en ese fichero, y el nombre de esa clase debe ser **MyClass** (incluyendo las que sean mayúsculas):

```
package mypackage;
public class MyClass {
// . . .
```

Ahora, si alguien quiere usar **MyClass** o, del mismo modo para cualquiera otra clase **public** en **mypackage** , tiene que usar la palabra reservada **import** para tener acceso al nombre o nombres en **mypackage** . La alternativa es dar el nombre completo:

```
mypackage.MyClass m = new mypackage.MyClass();
```

---

1 No hay nada en Java que obligue el uso de un intérprete. Existen compiladores de código nativo en Java que generan ficheros ejecutable.

La palabra import puede hacer esto más claramente:

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

Es bueno tener en mente que las palabras reservadas **package** e **import** lo que le permiten hacer, como diseñador de librerías, es dividir todo el espacio de nombres para que no se produzcan colisiones, sin importar cuanta gente tenga e Internet y comience a escribir clases en Java.

Creación de nombres de paquete único

Puede observar que, ya que un package nunca "empaqueta" todo realmente en un único fichero, puede estar compuesto de muchos ficheros .class y eso podría dar lugar a un poco de desorden. Para evitar esto, algo lógico de hacer es situar todos los ficheros **.class** de un paquete en particular en un solo directorio; esto es, usar la estructura jerárquica de ficheros del sistema operativo para su aprovechamiento. Esto es un modo en que Java da solución al problema del desorden; verá otra modo más tarde cuando se comente la utilidad **jar** .

Reunir los ficheros de los paquetes en un solo directorio resuelve otros dos problemas: creación de nombres de paquetes únicos, y encontrar aquellas clases que pudieran estar enterradas en algún lugar de la estrucutura del directorio. Esto está realizado, ya que fue presentado en el Capítulo 2, codificando la ruta de localización del fichero **.class** dentro del nombre del **package** . El compilador obliga a esto, pero por convención, la primera parte del nombre del **package** es el nombre de dominio de Internet del creador de la clase, revertido. Ya que está garantizado que los nombres de dominio de Internet sean únicos, si sigue esta convención está garantizando que su nombre de **package** será único y así nunca tendrá un conflicto de nombres. (Esto es, hasta que pierda el nombre de dominio y éste vaya a parar a alguien que además empiece a escribir código Java con la misma ruta que usó usted.) Por supuesto, si no tiene su propio nombre de dominio entonces debe fabricarse una combinación que difícilmente coincida con otra (como su nombre y apellidos) para crear nombres de paquete únicos. Si ha decidido empezar a publicar código Java merece la pena el relativamente pequeño esfuerzo de conseguir un nombre de dominio.

La segunda parte de este truco es tomar como nombre de **package** un directorio de su máquina, así cuando el programa en Java se ejecute y necesite cargar el fichero **.class** (lo cual se hace dinámicamente, en un punto del programa donde se necesita crear un objeto de esa clase particular, o la primera vez que accede a un miembro estático de la clase), puede localizar el directorio donde el fichero **.class** reside. El intérprete de Java procede como sigue. Primero, encuentra la variable de ambiente CLASSPATH (colocada por el sistema operativo, a veces por el programa de instalación que instala Java o una herramienta basada en Java de su máquina). CLASSPATH contiene uno o más directorios que son usados como raíz para la búsqueda de ficheros **.class** . Comenzando en esa raíz, el intérprete tomará el nombre del paquete y reemplazará cada punto con un paréntesis para generar la ruta desde la raíz del CLASSPATH (asi **package foo.bar.baz** se convierte en foo\bar\baz o foo/bar/baz o en alguna otra cosa, dependiendo de su sistema operativo) Esto es luego concatenado para varias entradas en el CLASSPATH. Ahí es donde busca el fichero .class con el nombre correspondiente a la clase que está intentando crear. (También busca algún directorio estándar relativo a donde el intérprete de Java reside).

Para entender esto, considere mi nombre de dominio, que es **bruceeckel.com** . Inviertiendo esto, **com.bruceeckel** establece mi nombre global único para mis clases. (Las extensiones com, edu, org, etc. fueron antiguamente puestas en mayúsculas en los paquetes Java, pero esto fue cambiando en Java 2 asi que el nombre del paquete entero es en minúsculas) Puedo además subdividir esto, decidiendo que quiero crear un librería llamada **simple** , así terminaré con un nombre de paquete:

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

Ahora este nombre de paquete puede ser usado como como una sombrilla de espacio de nombres para los siguientes dos ficheros:

```
//: 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 cree sus propios paquetes, descubrirá que la sentencia **package** tiene que ser la primera línea de código sin comentar en el fichero. El segundo fichero se parece mucho:

```
//: com:bruceeckel:simple:List.java
// Creando un paquete.
package com.bruceeckel.simple;
public class List {
   public List() {
      System.out.println(
         "com.bruceeckel.util.List");
   }
} ///:~
```

Ambos ficheros son situados en el subdirectorio de mi sistema:

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

Si retrocede, puede ver el nombre de paquete **com.bruceeckel.simple** , pero Qué ocurre con la primera parte de la ruta? Eso es tomado en cuenta en la variable de ambiente CLASSPATH , que es, en mi máquina:

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

Observe que el CLASSPATH puede contener un número alternativo de rutas de

búsqueda.

Sin embargo, hay una diferencia cuando se usan ficheros JAR. Debe poner el nombre del ficher JAR en el classpath, no solo la ruta donde se encuentra. Así, para un JAR llamado **grape.jar** su classpath incluiría:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Una vez que el classpath está instalado correctamente, el siguiente fichero puede ser situado en cualquier directorio:

```
//: c05:LibTest.java
// Usa la libreria.
import com.bruceeckel.simple.*;
public class LibTest {
   public static void main(String[] args) {
      Vector v = new Vector();
      List l = new List();
   }
} ///:~
```

Cuando el compilador encuentra la sentencia **import** , comienza investigando en los directorios especificados por CLASSPATH, buscando el subdirectorio com\bruceeckel\simple, y hallando los ficheros compilados con el nombre adecuado (**Vector.class** para **Vector** y **List.class** para **List**).

Observe que tanto las clases como los métodos utilizados in **Vector** y **List** deben ser **public** .

Ajustar el CLASSPATH ha sido tal aventura para los usuarios novatos en Java (para mí lo fue, cuando empecé) que Sun hizo el JDK en Java 2 un poco más listo. Encontrará que cuando lo instale, incluso si no establece un CLASSPATH será capaz de compilar y ejecutar programas básicos en Java. Sin embargo, para compilar y ejecutar el paquete del código fuente para este libro (disponible en el CD ROM incluido con este, o en www.BruceEckel.com), necesitará hacer algunas modificaciones a su CLASSPATH (estas son explicadas en el paquete del código fuente).

Conflictos

Qué ocurre si 2 librerias son importadas via * e incluyen los mismos nombres? Por ejemplo, suponga un programa que hace esto:

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

Ya que **java.util.*** también contiene una clase **Vector** , esto causa un conflicto en potencia. No obstante, mientras no escriba código que no cause conflictos es correcto- de esta forma no terminará escribiendo mucho para evitar colisiones que tal vez no se produciesen.

El conflicto ocurre si ahora intenta crear un **Vector** :

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

A qué clase **Vector** se hace referencia? El compilador no lo sabe, y el lector no puede saberlo tampoco. Por tanto el compilador da error y le obliga a ser explícito. Si quiero el **Vector** Java, por ejemplo, debo decir:

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

Ya que esto (junto con el CLASSPATH) especifica completamente la localización de ese **Vector** , no hay necesidad para la sentencia **import java.util.*** a menos que yo esté usando otra cosa de **java.util** .

Una librería de herramientas personalizada

Con lo que ya sabe, puede crear sus propias librerías de herramientas para reducir o eliminar código duplicado. Considere, por ejemplo, la creación de un alias para **System.out.println( )** y así no tener que escribir algo tan largo. Esto puede ser parte de una paquete llamado **tools:**

```
//: com:bruceeckel:tools:P.java
// Las abreviaturas P.rint y P.rintln.
package com.bruceeckel.tools;
public class P {
   public static void rint(String s) {
      System.out.print(s);
   }
   public static void rintln(String s) {
      System.out.println(s);
   }
} ///:~
```

Puede utilizar esta contracción para imprimir un **String** con salto de línea (**P.rintln( )**) or sin salto (**P.rint( )**).

Puede apostar que la ubicación de ese fichero debe ser en un subdirectorio que cominece en una de las ubicaciones del CLASSPATH y que luego continua con com/bruceeckel/tools. Después de compilar, el fichero P.class puede ser utilizado en cualquier lugar de su sistema con la sentencia import:

```
//: c05:ToolTest.java
// Usa la libreía tools.
import com.bruceeckel.tools.*;
public class ToolTest {
   public static void main(String[] args) {
      P.rintln("Disponible de ahora en adelante!");
      P.rintln("" + 100); // Obligada a ser un String
      P.rintln("" + 100L);
      P.rintln("" + 3.14159);
   }
} ///:~
```

Observe que todos los objetos pueden fácilmente ser convertidos al tipo **String** poniéndolos en una expresión de tipo **String** ; en el caso de arriba, comenzar la expresión con una cadena vacía sirve de truco. Pero esto nos trae una interesante observación.Si llama a **System.out.println(100)** , esto funciona sin convertir a cadena(**String**). Con un poco de sobrecarga, puede hacer que la clase **P** haga esto también (esto es un ejercicio al final de este capítulo).

Por tanto, de ahora en adelante, siempre que aparezca algo que sea de utilidad, puede añadirlo al directorio tools. (O a su directorio personal util o tools.)

Usando importaciones para cambiar el comportamiento

Una carácterística que está ausente de Java es la *compilación condicional* de C, que le permite cambiar un parámetro (switch) y conseguir diferentes comportamientos sin cambiar ningún otro código. La razón de que tal característica haya sido dejada en Java es probablemente el que la mayor parte del uso que se le da en C es para resolver cuestiones de cambio de plataforma: diferentes partes de código son compiladas dependiendo de la plataforma donde se compile. Ya que se pretende que Java sea multiplataforma, tal característica no debería ser necesaria.

Si embargo, hay otros usos importantes para la compilación condicional. Uno muy común es para depurar código. Las características de depurado son activadas durante el desarrollo y desactivadas en el producto final. Allen Holub

(www.holub.com) propuso la idea de utilizar paquetes para simular la compilación condicional. EL usó esto para crear una versión Java del muy útil mecanismo de afirmación de C, donde usted usted decir "esto debería ser verdadero" o "esto debería ser falso" y si la sentencia no coincide con su afirmación, lo sabrá. Tal herramienta es bastante útil durante la depuración.

Aquí está la clase que usará para depurar:

```
//: com:bruceeckel:tools:debug:Assert.java
// Herramienta de afirmación para depurar.
package com.bruceeckel.tools.debug;
public class Assert {
   private static void perr(String msg) {
      System.err.println(msg);
   }
   public final static void is_true(boolean exp) {
      if(!exp) perr("Assertion failed");
   }
   public final static void is_false(boolean exp){
      if(exp) perr("Assertion failed");
   }
   public final static void
   is_true(boolean exp, String msg) {
      if(!exp) perr("Assertion failed: " + msg);
   }
   public final static void
   is_false(boolean exp, String msg) {
      if(exp) perr("Assertion failed: " + msg);
   }
} ///:~
```

Esta clase simplemente encapsula un test para los Booleanos, que imprime un mensaje de error si falla. En el capítulo 10, aprenderá una herramienta más sofisticada para tratar con errores llamada *manejo de excepciones* , pero el método **perr()** funcionará bien mientras tanto.

La salida es impresa al dispositivo estándar de error (*standard error stream*) escribiendo a **System.err** .

Cuando quiera usar este clase, añada una linea en su programa:

```
import com.bruceeckel.tools.debug.*;
```

Para eliminar las las afirmaciones y poder despachar el código , se crea una

segunda clase **Assert** , pero en un paquete diferente:

```
//: com:bruceeckel:tools:Assert.java
// Desactivando la salida de la afirmación
// para poder despachar el programa.
package com.bruceeckel.tools;
public class Assert {
   public final static void is_true(boolean exp){}
   public final static void is_false(boolean exp){}
   public final static void
   is_true(boolean exp, String msg) {}
   public final static void
   is_false(boolean exp, String msg) {}
} ///:~
```

Ahora si cambia la anterior sentecia **import** :

```
import com.bruceeckel.tools.*;
```

El programa ya no imprimirá las afirmaciones. Aquí tiene un ejemplo:

```
//: c05:TestAssert.java
// Demostración de la herramienta de afirmación.
// Comente y descomente las líneas siguientes
// para cambiar el comportamiento de afirmación:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;
public class TestAssert {
   public static void main(String[] args) {
      Assert.is_true((2 + 2) == 5);
      Assert.is_false((1 + 1) == 2);
      Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
      Assert.is_false((1 + 1) == 2, "1 +1 != 2");
   }
} ///:~
```

Al cambiar el **package** que es importado, puede cambiar su código de la versión en pruebas a la version final. Esta técnica puede ser usada para todo tipo de código condicional.

Package caveat

Merece la pena recordar que siempre que cree un paquete, especifica implícitamente una estructura de directororio al dar nombre al paquete. El paquete debe residir en el directorio indicado por su nombre, el cual debe ser un directorio que se pueda buscar comenzando desde el CLASSPATH.

Experimentar con la palabra reservada **package** puede ser un poco frustrante al principio, porque a menos que ajuste el nombre del paquete a las reglas anteriores obtendrá muchos mensajes en tiempo de ejecución que hablan de no ser capaz de encontrar una clase, incluso si esa clase esta ahí en el mismo directorio. Si obtiene un mensaje asi, intente poner entre comentarios la sentencia **package** y si así funciona, sabrá donde está el problema.

Especificadores de acceso en Java

Cuando son usados, los especificadores de acceso en Java **public** , **protected** , y **private** se colocan delante de cada declaración para cada miembro de su clase, si se trata de un campo o un método. Cada especificador controla el acceso pero solo para una definición en concreto. Esto es una diferencia con C++, en el que los especificadores de acceso afectan todas las definiciones que aparecen tras ellos hasta que otro especificador apararezca.

De una u otra forma, todo los elementos tienen especificado algun tipo de acceso. En las siguientes secciones, aprenderá varios tipos de acceso, empezando con el acceso por defecto.

"Friendly"

Qué ocurre si no pone especificador de acceso tal y como ocurre en los ejemplos anteriores a este capítulo? El acceso por defecto no tiene palabra resarvada, pero es normalmente conocido como "friendly" (amiga). Significa que todo el resto de clases del paquete actual tienen acceso a un miembro friendly, pero para el resto de clases fuera del paquete, el miembro aparece como **private** (privado). Ya que una unidad de compilación -un fichero- puede pertenecer a un solo paquete, todas las clases dentro de una unidad de compilación son automáticamente amigas (friendly) unas de otras. Así, los elementos friendly también se dice que tienen *acceso de paquete* .

El acceso Friendly le permite agrupar clases relacionadas en un paquete para que puedan interactuar fácilmente unas con otras. Cuando coloca clases juntas en un paquete (garantizando así el acceso mutuo a sus miembros friendly; haciéndolos "amigos") usted "posee" el código en ese paquete. Tiene sentido que sólo el código que usted posee pueda tener acceso a oltro código de su propiedad. Usted podría decir que el acceso amistoso le d aun signifcicado o una razón al agrupamiento de clases un un paquete. En muchos lenguajes la forma en que

usted organiza sus definiciones en archivos puede ser willy-nilly, pero en Java está obligado a hacerlo en una forma sensible. En suma, probablemente quiera excluir clases que no deberían tener acceso a las clases definidas en el paquete actual.

La clase contorla qué código tiene acceso a sus miembros. No hay froma mágica de "entrar a la fuerza". El código de otro paquete no puede aparecer y decir "Hola, soy amigo de **Bob** !" y esperar ver los miembros **protected** , friendly, y **private** de **Bob** . El único modo de conceder acceso a un miembro es:

1. Hacer el miembro **public** . Entonces todo el mundo, en cualquier lado, puede acceder a él.
2. Hacer el miembro friendly al no indicar ningún especificador de acceso, y poniendo las otras clases en el mismo paquete. Entonces las otras clases pueden acceder al miembro.
3. Como verá en el Capítulo 6, cuando comentemos la herencia, una clase derivada puede acceder a un miembro **protected** igual que a un miembro **public** (pero no a miembros **private**). Puede acceder a los miembros friendly sólo si las dos clases están en el mismo paquete. Pero no se preocupe por eso ahora.
4. Proporcionar métodos para "acceso/mutación" (también conocidos como métodos "get/set") que leen y cambian el valor. Esta es la aproximación más civilizada en terminos de POO, y es fundamental con JavaBeans, como verá en el Capítulo13.

public : interfaz de acceso

Cuando usa la palabra reservada **public** , significa que la declaración del identificador que va detrás de public está disponible para todo el mundo, en particular para el programador cliente que usa la librería. Suponga que define un paquete **dessert** que contiene la siguiente unidad de compilación:

```
//: c05:dessert:Cookie.java
// Crea una librería.
package c05.dessert;
public class Cookie {
   public Cookie() {
      System.out.println("Constructor de Cookie");
      void bite() { System.out.println("bite"); }
   } ///:~
```

Recuerde, **Cookie.java** debe estar en un subdirectorio llamado **dessert** , en un directorio bajo **c05** (indicando Capítulo 5 de este libro) que debe estar bajo uno de los directorios del CLASSPATH. No cometa el error de pensar que Java siempre mirará en el directorio actual como uno de los puntos de comienzo a la hora de buscar. Si no tiene un punto '.'como ruta en su CLASSPATH, Java no mirará ahí. Ahora si crea un programa que utilice **Cookie** :

```
//: c05:Dinner.java
// Usa la librería
import c05.dessert.*;
public class Dinner {
   public Dinner() {
      System.out.println("Constructor de Dinner");
   }
   public static void main(String[] args) {
      Cookie x = new Cookie();
      //! x.bite(); // No puede acceder
   }
} ///:~
```

Puede crear un objeto **Cookie** , ya que su constructor es **public** y la clase es **public** . (Profundizaremos en el concepto de clase **public** más tarde.) Pero, el miembro **bite( )** no es accesible dentro de **Dinner.java** ya que **bite( )** es amigo solo dentro del paquete **dessert** .

El paquete por defecto

Puede que le sorprenda descubrir que el siguiente código compila, aunque a primera vista parece que no cumple las reglas:

```
//: c05:Cake.java
// Accede a una clase en una
// unidad de compilación separada.
class Cake {
   public static void main(String[] args) {
      Pie x = new Pie();
      x.f();
   }
} ///:~
```

En un segundo fichero, en el mismo directorio:

```
//: c05:Pie.java
// La otra clase.
class Pie {
   void f() { System.out.println("Pie.f()"); }
} ///:~
```

Podría inicialmente ver esto como ficheros totalmente ajenos, y sin embargo la

clase **Cake** es capaz de crear un objeto **Pie** y llamar a su método **f( )** !! (Observe que debe tener '.' en su CLASSPATH para que los ficheros compilen.) Normalmente pensaría que **Pie** y **f( )** son friendly y por lo tanto no disponible para **Cake** . Son friendly-eso es correcto. La razón de que esté disponible en **Cake.java** es porque están en el mismo directorio y tienen nombre de paquete no explícito. Java trata ficheros así como si fueran parte del "paquete por defecto" para ese directorio, y por lo tanto friendly a otros ficheros de ese directorio.

private: no puedes tocar eso!

La palabra reservada **private** significa que nadie puede acceder a ese miembro excepto la clase en sí, dentro de los métodos de la misma. Las otras clases en el mismo paquete no pueden acceder a miebros **private** , de manera que es como si la estuviera aislando de usted mismo. Por otro lado, es probable que un paquete pueda ser creado por varias personas colaborando juntas, asi que **private** le permite cambiar ese miembro sin preocuparse de que eso afectará a otra clase del mismo paquete. El acceso de paquete por defecto "friendly" a menudo proporciona una adecuado grado de ocultación; recuerde, un miembro "friendly" es inaccesible para el usuario del paquete. Esto está bien, ya que el acceso por defecto es el que normalmete usa (y el que obtendrá si olvida añadir cualquier control de acceso). Así, normalmente pensará en el acceso para los miembros que quiera hacer explícitamente **public** para el programador cliente y, como resultado, podría pensar incialmente que no usará la palabra clave **private** frecuentemente, ya que es tolerable estar sin ella. (Este es un contraste distinto con C++). Sin embargo, ocurre que el uso consistente de **private** es muy importante, especialmente cuando nos concierne el multithreading. (Como verá en el Capítulo 14.)

Aquí tiene un ejemplo del uso de **private** :

```
//: c05:IceCream.java
// Demuestra la palabra clave "private".
class Sundae {
   private Sundae() {}
   static Sundae makeASundae() {
      return new Sundae();
   }
}
public class IceCream {
   public static void main(String[] args) {
      //! Sundae x = new Sundae();
      Sundae x = Sundae.makeASundae();
   }
} ///:~
```

Esto muestra un ejemplo en el cual **private** viene bien: podría querer controlar

como un objeto es creado y evitar que alguien acceda a un constructor en particular (o a todos). En el ejemplo de arriba, no puede crear un objeto **Sundae** a través de su constructor; en cambio debe llamar al método **makeASundae( )** para que lo haga por usted[2].

Cualquier método que que está seguro que es sólo un método de ayuda para esa clase puede ser **private** , para asegurar que no lo use accidentalmente en otro lugar del paquete y prohibiéndose así de cambiar o remover el método. Hacer un método **private** garantiza que retiene esta opción.

---

2 Hay otro efecto en este caso:ya que el constructor por defecto es el único definido y es **private** , se evitará la herencia de esta clase. (A tema que será presentado en el Capítulo 6.)

Lo mismo es cierto para un campo **private** dentro de una clase. A menos que tenga que exponer la implementación subyacente (lo cual es una situación más rara de lo que usted podría pensar), debe hacer todos los campos private. Si enmbargo, solo porque una referencia a un objeto dentro de una clase es private no significa que otro objeto no puede tener una referencia public al mismo objeto. (Ver Apéndice A para temas acerca de aliasing.)

protected : "algo parecido a friendly"

Necesita leer más adelante para entender el especificador de acceso **protected** . En primer lugar, debe ser consciente de que no necesita entender esta sección para continuar con el libro pasando a la herencia (Capítulo 6). Pero para no dejarse nada atrás aquí tiene una breve descripción y un ejemplo usando **protected** . La palabra reservada **protected** se relaciona con un concepto llamado *herencia* , que toma una clase creada y añade nuevos miembros a esa clase sin modificarla, y a la que nos refereriremos como clase base. Puede también cambiar el comportamiento de los métodos creados de la clase. Para heredar de una clase creada, se dice que nuestra nueva clase extiende (extends) una clase existente, de este modo:

```
class Foo extends Bar {
```

El resto de la definición de la clase no varía. Si crea un nuevo paquete y hereda de una clase en otro paquete, a los únicos miembros que tiene acceso son los miembros public del paquete original. (Por supuesto, si realiza la herencia en el *mismo* paquete, tiene el acceso normal a todos los miembros "friendly".) A veces el creador de la clase base, le gustaría tomar un miembro en particular y garantizar acceso a las clases derivadas pero no al resto. Eso es lo que **protected** hace. Si nos fijamos en el anterior fichero **Cookie.java** , la siguiente clase no puede acceder al miembro "friendly":

```
//: c05:ChocolateChip.java
// No puede acceder a un miembro friendly
// de otra clase.
import c05.dessert.*;
public class ChocolateChip extends Cookie {
   public ChocolateChip() {
      System.out.println(
         "Constructor de ChocolateChip");
   }
   public static void main(String[] args) {
      ChocolateChip x = new ChocolateChip();
      //! x.bite(); // No puede acceder a bite
   }
} ///:~
```

Una de las cosas interesantes de la herencia es que sin un método **bite( )** existe en la clase **Cookie** , también existe luego en cualquier clase derivada de **Cookie** . Pero dado que **bite()** es "friendly" en un paquete distinto, no está disponible para nosotros en este. Por supuesto, podría hacerlo **public** , pero luego todos tendrían acceso y quizás no quiera eso. Si cambiamos la clase **Cookie** como sigue:

```
public class Cookie {
   public Cookie() {
      system.out.println("Constructor de Cookie");
   }
   protected void bite() {
      System.out.println("bite");
   }
}
```

entonces **bite()** todavía tiene acceso "friendly" dentro del paquete **dessert** , pero es también accesible para todos los que hereden de **Cookie** . Sin embargo, no es **public** .

Interfaz e implementación

El control de acceso está con frecuencia se refiere al *ocultamiento dela implementación* . Envolver datos y métodos dentro de clases combinado con la ocultación de la implementación es con frecuencia llamado *encapsulamiento* [3]. El resultado es un tipo de dato con características y comportamientos.

---

3 No obstante, la gente frecuentemente se refiere al ocultamiento de la implementación sólo como encapsulamiento.

El control de acceso crea fronteras dentro de un tipo de datos por dos importantes razones. La primera es establecer lo que el programador cliente puede y no puede usar. Puede construir mecanismos internos a la estructura sin preocuparse de que los programadores cliente accidentalmente traten las partes internas como parte de la interfaz que deberían usar.

Esto nos condunce directamente a la segunda razon, que es separar la interface de la implementación. Si se usa la estructura en un conjunto de programas, pero los programadores cliente no pueden hacer nada aparte de enviar mensajes a la interfaz pública, entonces usted puede cambiar cualquier cosa que no sea pública ("friendly", protected, o private) sin requerir modificaciones al código del cliente.

Estamos ahora en el mundo de la programación orientada a objetos, donde una **clase** en realidad se describe como "una clase de objetos", tal y como describiría una clase de pescados o pajaros. Cualquier objeto perteneciente a esta clase compartirá estas características y comportamientos. La clase es una descripción del estado de los objetos de ese tipo y de su forma de actuar.

En el primer lenguaje OO, Simula-67, la palabra reservada **class** era usada para describir un nuevo tipo de datos. La misma palabra ha sido usada por la mayor parte de los lenguajes orientado a objetos. Este el punto clave de todos los lenguajes: la creación de nuevos tipos de datos que son más que cajas conteniendo datos y métodos.

La clase es el concepto OO fundamental en Java. Es una de las palabras reservadas (class) que *no* será puesta en negrita en este libro-llega a ser molesta una palabra tantas veces repetida como "class".

Por claridad, podría preferir un estilo de creación de clases que ponga los miembros **public** al principio, seguido por los miembros **protected** , friendly, y **private** . La ventaja es que el usuario de la clase puede leer de abajo a arriba y ver al principio que es lo importante (los miembros **public** , porque ellos pueden ser accedidos fuera del fichero), y dejar de leer cuando encuentran los miembros no **public** , 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 solo lo hará fácil de leer en parte porque la interfaz y la implementación están mezcladas aún. De ese modo, verá el código fuente -la implementación- porque está justo ahí en la clase. Además, la documentación comentada soportada por javadoc (descrito en el Capítulo 2) reduce la importancia de la habilidad de leer código por el cliente programador. Enseñar la interfaz al consumidor de la clase es en realidad trabajo del *class browser* , una herramienta cuyo trabajo es mirar todas las clases disponibles y mostrarle que puede hacer con ellas(por ejemplo, los miebros que están disponibles) de un modo útil. En el momento en que lee esto, los browsers deben ser una parte a incluir en cualquier buen entorno de desarrollo Java.

Acceso a las clases

En Java, los especificadores de acceso pueden también ser usados para determinar las clases de una librería que estarán disponibles para los usuarios de esa librería. Si quiere que una clase esté disponible para un programador cliente, coloque la palabra **public** en algún lugar antes del la llave de apertura del cuerpo de la clase. Esto controla incluso si el programador cliente puede crear un objeto de esa clase.

Para controlar el acceso de una clase, el especifiacador debe aparecer antes de class. Así, puede poner:

```
public class Widget {
```

ahora si el nombre de su librería es **mylib** cualquier programador cliente puede acceder a **Widget** diciendo

```
import mylib.Widget;
```

o

```
import mylib.*;
```

No obstante, hay un conjunto de restricciones extra:

1. Puede haber solo un una clase **public** por unidad de compilación (fichero). La idea es que cada unidad de compilación tiene una única interfaz pública, representada por esa clase **public** . Puede tener muchas clases "friendly" de apoyo comoa quiera. Si tiene más

de una clase **public** dentro de una unidad de compilación, el compilador le dará un mensaje de error.

2. El nombre de la clase **public** debe ser exactamente el mismo que el del fichero que contiene la unidad de compilación, distinguiendo mayúsculas de minúsculas. Así para **Widget** , el nombre del fichero debe ser **Widget.java** , no **widget.java** o **WIDGET.java** . Una vez más, obtendría un error en tiempo de compilación si son diferentes.
3. Es posible, aunque no habitual, tener una unidad de compilación sin clases **public** . En ese caso, puede llamar al fichero como más le guste.

Que pasa si usted tiene una clase dentro de **mylib** que está usando sólo para llevar a cabo las tareas realizadas por **Widget** o alguna otra clase pública en **mylib** ? Usted no quiere tomarse la molestia de escribir documentación para el programador cliente y piensa que en algún momento más tarde podría querer cambiar completamente las cosas y desmenuzar completamente su clase, sustituyéndola por una diferente. Para conseguir esta flexibilidad, necesita asegurase de que ningún programador cliente se vuelva dependiente de sus detalles particulares de implementación ocultos dentro de **mylib** , Para llevar a cabo esto, sólo quite la palabra clave **public** de la clase, de manera que se vuelva "friendly" (Esta clase puede ser usada sólo dentro de este paquete).

Note que la clase no puede ser **private** (que la haría inaccesible a todos, excepto a ella misma), o **protected** 4. De manera que sólo tiene dos opciones para el acceso a la clase: "friendly" o **public** . Si no quiere que otro tenga acceso a esa clase, puede hacer que todos los constructores sean **private** , previniendo consecuentemente a todos de crear un objeto de esa clase, excepto a usted desde dentro de un miembro **static** de la clase5. Aquí hay un ejemplo:

```
//: c05:Lunch.java
// Demuestra los especificadores de acceso a clase.
// Hace una clase efectivamente privada
// con constructores privados:
class Soup {
   private Soup() {}
   // (1) Permite la creación por medio de un método estático:
   public static Soup makeSoup() {
      return new Soup();
   }
   // (2) Crea un objeto estático y
   // retorna una referencia al requerimiento.
   // (El patrón "Singleton"):
   private static Soup ps1 = new Soup();
   public static Soup access() {
      return ps1;
   }
   public void f() {}
```

```
}
class Sandwich { // Usa Lunch
   void f() { new Lunch(); }
}
// Sólo se permite una clase pública por archivo:
public class Lunch {
   void test() {
      // No permitido! Constructor privado:
      //! Soup priv1 = new Soup();
      Soup priv2 = Soup.makeSoup();
      Sandwich f1 = new Sandwich();
      Soup.access().f();
   }
} ///:~
```

---

4 Realmente, una clase interna puede ser privada o protegida, pero es un caso especial. Estas serán presentadas en el capítulo 7.

---

5 También lo puede hacer heredando (Capítulo 6) de esa clase.

Hasta ahora, la mayoría de los métodos han estado retornando **void** o un tipo primitivo, de manera que la definición:

```
public static Soup access() {
   return ps1;
}
```

puede parecer un poco confusa al principio. La palabra previa al nombre del método (**access**) dice qué retorna este método. Hasta ahora, esto ha sido en su mayor parte **void** , lo que significa que no retorna nada. Pero además se puede retornar una referencia a un objeto, que es lo que ocurre aquí. Este método retorna una referencia a un objeto de clase **Soup** .

La clase **Soup** muestra como prevenir la creación directa de una clase haciendo todos los constructores **private** . Recuerde que si no crea al menos un constructor explícitamente, el constructor por defecto (un constructor sin argumentos) será creado automáticamente. Escribiendo el constructor por defecto, no será creado automáticamente. Haciéndolo **private** , nadie puede crear un objeto de esa clase. Pero ahora Cómo usa alguien esta clase? El ejemplo anterior muestra dos opciones. Primero, se escribe un método **static** que crea un **Soup** nuevo y retorna una referencia al mismo. Esto puede ser útil si quiere hacer alguna operación extra sobre el **Soup** antes de retornarlo o si quiere mantener la cuenta de cuantos objetos **Soup** crea (quizás para restringir su población).

La segunda opción usa lo que se llama un *patrón de diseño* , el cual es cubierto en *Thinking in Patterns with Java* , descargable de *www.BruceEckel.com* . Este patrón

particular se llama "singleton" porque permite que se cree sólo un único (single) objeto. El objeto de clase **Soup** es creado como un miembro **static private** de **Soup** , de manera que hay sólo uno, y no puede obtenerlo a no ser por el método **access()** público.

Como se mencionó previamente, si no coloca un especificador de acceso para la clase, por defecto es "friendly". Esto significa que un objeto de esa clase puede ser creado por cualquier otra clase en el paquete, pero no por una externa al mismo. (Recuerde que todos los archivos del mismo directorio que no tienen una declaración de **package** explícita son implícitamente parte del paquete por defecto para ese directorio). Por supuesto, si un miembro **static** de esa clase es **public** , el programador cliente puede todavía acceder a ese miembro aunque no puedan crear un objeto de esa clase.

Resumen

En toda relación es importante tener límites que sean respetados por todas las partes involucradas. Cuando usted crea una librería, establece una relación con el usuario de la misma -el programador cliente- quien es otro programador, pero que está reuniendo una aplicación o usando su librería para construir una mayor.

Sin reglas, los programadores cliente pueden hacer lo que quieran con los miembros de una clase, incluso si usted prefiere que no manipulen directamente algunos miembros. Todo está desnudo al mundo.

Este capítulo dio un vistazo a cómo construir clases para formar librerías; primero, la forma en que un grupo de clases es empaquetado dentro de una librería y, segundo, la forma en que la clase controla el acceso a sus miembros.

Se estima que un proyecto de programación en C comienza a averiarse al llegar a algún lugar entre las 50.000 y 100.000 líneas de código porque C tiene un "espacio de nombres" único, de manera que los nombres comienzan a conflictuar, causando una sobrecarga extra en la gestión. En Java, la palabra clave **package** , el esquema de nombres de paquetes y la palabra clave **import** le dan un control completo sobre los nombres, de manera que la cuestión del conflicto de nombres es fácilmente evitada.

Hay dos razones para controlar el acceso a los nombres. La primera es para mantener las manos de los usuarios lejos de las herramientas que no deberían tocar; las herramientas son necesarias para las maquinaciones internas del tipo de dato, pero no una parte de la interfaz que los usuarios necesitan para resolver sus problemas particulares. De manera que hacer los **private** los métodos y campos es un servicio al usuario porque pueden ver fácilmente qué es importante para ellos y qué pueden ignorar. Simplifica su entendimiento de la clase.

La segunda razón más importante para el control de acceso es permitirle al diseñador de la librería cambiar el trabajo interno de la clase sin preocuparse acerca de cómo afectará al programador cliente. Podría construir una clase de una

forma primero y luego descubrir que reestructurando su código se obtendrá una velocidad mayor. Si la interfaz y la implementación son separadas y protegidas claramente, usted puede realizar esto sin forzar al usuario a reescribir su código.

Los especificadores de acceso en Java le dan al creador de la clase un control valioso. El usuario de la clase puede ver claramente qué puede usar y qué puede ignorar. Más importante aún, es la capacidad de asegurar que ningún usuario se vuelva dependiente de cualquier parte de la implementación subyacente de una clase. Si usted, como creador de la clase, sabe esto puede cambiar la implementación subyacente con el conocimiento de que ningún programador cliente será afectado por los cambios porque ellos no pueden acceder a esa parte de la clase.

Cuando tiene la capacidad de cambiar la implementación subyacente, usted puede no sólo mejorar su diseño más tarde, sino que además tiene la libertad de cometer errores. No importa que tan cuidadosamente planee o diseñe, cometerá errores. Saber que es relativamente fácil cometer esos errores significa que será más experimental, aprenderá más rápido y terminará su proyecto más pronto.

La interfaz pública de una clase es lo que el usuario *ve* , de manera que la parte más importante de la clase a hacer bien durante el análisis y diseño. Incluso esto brinda algo de libertad para el cambio. Si usted no hace la interfaz bien la primera vez, puede *agregar* métodos, en cuanto no remueva ninguno de los que los programadores cliente han usado ya en su código

Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico The Thinking in Java Annotated Solution Guide, disponible por un pequeño monto en www.BruceEckel.com.

1. Escriba un programa que cree un objeto **ArrayList** sin importar explícitamente **java.util.*** .
2. En la sección titulada "paquetes: la librería unidad" convierta los fragmentos de código concernientes a **mypackage** en un conjunto de archivos Java que compilen y corran.
3. En la sección titulada "Conflictos" tome los fragmentos de código y conviértalos en un programa y verifique los conflictos que de hecho ocurren.
4. Haga más general la clase **P** definida en este capítulo agregando todas las versiones sobrecargadas de **rint( )** y **rintln( )** necesarias para manejar todos los tipos básicos de Java.
5. Cambie la sentencia import en **TestAssert.java** para habilitar y deshabilitar el mecanismo de afirmación.
6. Construya una clase con datos y métodos miembro que sean **public** , **private** , **protected** y "friendly". Cree un objeto de esta clase y vea que tipo de errores de compilador obtiene cuando intenta acceder a todos los miembros de la clase. Sea consiente de que las clases del mismo directorio son parte del mismo paquete "por

defecto".

7. Construya una clase con datos **protected** . Construya una segunda clase en el mismo archivo que manipule los datos **protected** de la primera clase.
8. Cambie la clase **Cookie** como se especifica en la sección "protected: algo parecido a friendly". Verifique que **bite( )** no es **public** .
9. En la sección titulada "Acceso a las clases" encontrará fragmentos de código que describen a **mylib** y a **Widget** . Construya esta librería, luego cree un **Widget** en una clase que no sea parte del paquete **mylib** .
10. Cree un nuevo directorio y edite su CLASSPATH para que lo incluya. Copie el archivo **P.class** (producido compilando **com.bruceeckel.tools.P.java**) a su nuevo directorio y luego cambie los nombres del archivo, la clase **P** adentro y los nombres de los métodos. (También podría agregar salida adicional para ver cómo funciona). Construya otro programa en un directorio diferente que use su nueva clase.
11. Siguiendo la forma del ejemplo **Lunch.java** , construya una nueva clase llamada **ConnectionManager** que gestione un arreglo fijo de objetos **Connection** . El programador cliente debe ser ahora capaz de crear objetos **Connection** explícitamente, pero sólo puede obtenerlos vía un método **static** en **ConnectionManager** . Cuando **ConnectionManager** se queda sin objetos, retorna una referencia nula (**null**). Pruebe las clases en **main( )** .
12. Construya el siguiente archivo en el directorio c05/local (presumiblemente en su CLASSPATH):

```
package c05.local;
class PackagedClass {
   public PackagedClass() {
      System.out.println(
         "Creación de una clase empaquetada");
   }
} ///:~
```

Luego construya el siguiente archivo en un directorio distinto de c05:

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```