**CAPITULO 4 Interacciones de objetos**

Como aprendió en el Capítulo 3, los objetos son los componentes básicos de un software orientado a objetos sistema. En tal sistema, los objetos colaboran entre sí para lograr un sistema común metas, similares a las hormigas en un hormiguero, o los empleados de una corporación, o las células en su cuerpo. Cada objeto tiene una estructura y una misión específicas; estas respectivas misiones complementan unos a otros para lograr la misión general del sistema en su conjunto.

En este capítulo, aprenderá

• Cómo se utilizan los métodos para especificar el comportamiento de un objeto

• Los diversos elementos de código que componen un método

• Cómo los objetos publicitan sus métodos como servicios entre sí

• Cómo los objetos se comunican entre sí para solicitar los servicios de los demás en orden para colaborar

• Cómo los objetos mantienen sus datos y cómo protegen sus datos para garantizar su integridad.

• El poder de una función de lenguaje orientada a objetos conocida como ocultación de información, y cómo la información La ocultación se puede utilizar para limitar el efecto dominó en el código de una aplicación cuando los detalles de implementación de una clase cambian inevitablemente

• Cómo se puede utilizar un tipo especial de función conocida como constructor para inicializar el estado de un objeto cuando se instancia por primera vez Los eventos impulsan la colaboración de objetos En su forma más simple, el proceso de desarrollo de software orientado a objetos implica lo siguiente cuatro pasos básicos:

1. Establecer adecuadamente los requisitos funcionales y la misión general de una solicitud

2. Diseñar las clases adecuadas: sus estructuras de datos, comportamientos y relaciones unos con otros, necesarios para cumplir con estos requisitos y misión

3. Creación de instancias de estas clases para crear los tipos y el número adecuados de instancias de objetos

4. Poner estos objetos en movimiento a través de eventos de activación externos

Piense en un hormiguero: a primera vista, es posible que no vea ninguna actividad aparente. Pero si tu deje caer una barra de chocolate cerca, una ráfaga de actividad comienza repentinamente cuando las hormigas se apresuran a reunirse las "golosinas", así como para reparar cualquier daño que pueda haber sido causado si dejó caer el barra de chocolate demasiado cerca del hormiguero! Dentro de una aplicación OO (el "hormiguero"), los objetos ("hormigas") pueden ser puestos en movimiento por un evento externo como

• El clic de un botón en la interfaz gráfica de usuario de SRS, que indica el deseo de un estudiante de registrarse para un curso en particular

• La recepción de información de algún otro sistema automatizado, como cuando el SRS

recibe una lista de todos los estudiantes que han pagado su matrícula del sistema de facturación de la universidad

Tan pronto como un sistema OO detecta un evento desencadenante de este tipo, los objetos apropiados reaccionar, realizar servicios ellos mismos y / o solicitar servicios de otros objetos en reacción en cadena moda, hasta que se haya logrado algún objetivo general de la aplicación. Por ejemplo, La solicitud para registrarse en un curso realizado por un usuario estudiante a través de la GUI de la aplicación SRS implican la colaboración de muchos objetos diferentes, como se ilustra en la Figura 4-1:

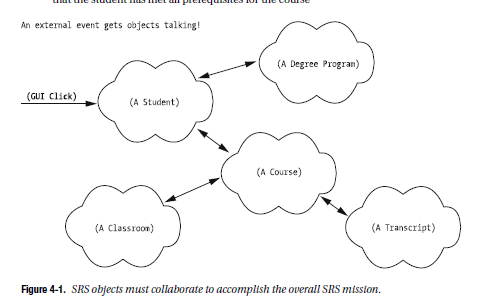
• Un objeto Student (una abstracción del usuario estudiante real)

• Un objeto del programa de grado, para asegurar que el curso solicitado sea realmente requerido para el estudiante graduarse

• El objeto del curso apropiado, para asegurarse de que haya un asiento disponible para el estudiante en ese curso

• Un objeto Classroom, que representa la sala en la que se reunirá el curso, para verificar su capacidad de asientos

• Un objeto de transcripción, específicamente, la transcripción del estudiante de interés, para garantizar que el alumno ha cumplido con todos los requisitos previos del curso



Mientras tanto, un estudiante que usa el SRS es felizmente ignorante de todos los objetos que están "corriendo alrededor” detrás de escena para lograr su objetivo. El estudiante simplemente completa algunos campos y hace clic en un botón en la GUI de SRS, y unos momentos más tarde ve un mensaje que confirma o rechaza su solicitud de registro.

Una vez que se ha logrado el objetivo final de una cadena de eventos (por ejemplo, registrar a un estudiante para un curso), los objetos de una aplicación quedan inactivos y pueden permanecer así hasta el próximo tal evento desencadenante ocurre. Una aplicación orientada a objetos es en cierto modo similar a un juego de billar: golpea la bola blanca con tu taco, y (¡con suerte!) golpea otra bola, que podría chocar con otras tres bolas y así sucesivamente. Eventualmente, sin embargo, todas las bolas se detendrán hasta la bola blanca se vuelve a golpear.

**Métodos de declaración**

Hablemos con un poco más de detalle sobre cómo especificamos formalmente los comportamientos de un objeto como métodos Java.

Recuerde del Capítulo 3 que los comportamientos de un objeto pueden considerarse como servicios que el objeto puede realizar. Para que un objeto A solicite algún servicio de un objeto B, A necesita saber el idioma específico con el que comunicarse con B. Es decir,

• El objeto A debe ser claro en cuanto a exactamente cuál de los métodos / servicios de B quiere que B realice.

Piense en usted mismo como objeto A y un perro como objeto B. ¿Quiere que su perro se siente? ¿Permanecer? ¿Tacón? ¿Ir a buscar?

• Dependiendo de la solicitud de servicio, es posible que el objeto A deba proporcionar a B información adicional.

para que B sepa exactamente cómo proceder. Si le dice a su perro que vaya a buscar, el perro necesita saber qué traer: ¿una pelota? ¿Un palo? ¿El gato del vecino?

• El objeto B, a su vez, necesita saber si el objeto A espera que B informe el resultado. de lo que se le ha pedido que haga. En el caso de un comando para buscar algo, su perro con suerte le traerá el artículo solicitado como resultado. Sin embargo, si su perro está en otra habitación y gritas el comando "¡Siéntate!", no verás el resultado de tu comando; tienes que confiar en que el perro ha hecho lo que le pediste.

Nos encargamos de especificar / definir estos tres aspectos de cada método declarando un método Luego debemos programar la lógica detrás de escena para saber cómo B realizará el servicio en el cuerpo del método.

Para los lectores familiarizados con el lenguaje de programación C, una declaración de método Java es prácticamente la misma sintácticamente como una declaración de función C. La única diferencia filosófica entre una función en un lenguaje que no es OO como C y un método en un lenguaje OO como Java es el contexto en el que se ejecutan: un no OO La función es ejecutada por el entorno de programación como un todo, mientras que un método en un lenguaje OO es ejecutado por un objeto en particular. Exploraremos esta diferencia con más detalle a medida que se desarrolla este capítulo. Primero veamos los encabezados de los métodos.

**Encabezados de método**

El encabezado de un método es una especificación formal (desde el punto de vista de la programación) de cómo ese método debe invocarse. Un encabezado de método, como mínimo, consta de

• El tipo de retorno de un método, es decir, el tipo de información que devolverá objeto B al objeto A, si lo hay, cuando el método de B termina de ejecutarse.

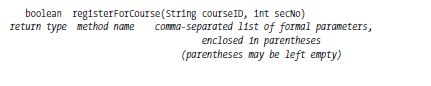
• El nombre de un método.

• Una lista opcional de parámetros formales separados por comas (especificando sus tipos y names) para pasar al método, entre paréntesis. Si no se necesitan parámetros pasado, se usa un conjunto vacío de paréntesis; se dice que tales métodos "no toman parámetros", y nos referiremos a ellos como sin parámetros.

Como ejemplo, aquí hay un encabezado de método típico que podríamos definir para la clase Student:

boolean registerForCourse (String courseID, int secNo)

tipo de retorno nombre del método lista separada por comas de parámetros formales, entre paréntesis (los paréntesis pueden dejarse en blanco)



Al referirse casualmente a un método como registerForCourse en texto narrativo, algunos autores adjuntan una conjunto vacío de paréntesis, (), al nombre del método, por ejemplo, registerForCourse (). Esto no necesariamente Sin embargo, implica que el encabezado formal no tiene parámetros.

**Convenciones de nomenclatura de métodos**

Los nombres de los métodos Java se elaboran utilizando el estilo de carcasa de camello; recuerde del Capítulo 2 esa variable Los nombres también se elaboran con tripa de camello. A modo de repaso, con tripa de camello

• La primera letra del nombre del método está en minúsculas.

• La primera letra de cada palabra concatenada subsiguiente en el nombre del método está en mayúsculas, y los caracteres restantes están en minúsculas.

• No utilizamos ningún carácter de "puntuación" (guiones, guiones bajos, etc.) para separar estas palabras.

Como ejemplo, chooseAdvisor es un nombre de método apropiado, mientras que ninguno de los siguientes sería apropiado: Elija asesor ("C" mayúscula), elija asesor ("a" minúscula), choose\_advisor (separando el guión bajo).

**Pasar argumentos a métodos**

El propósito de pasar argumentos a un método es doble:

• Proporcionarle el "combustible" (opcional) necesario para hacer su trabajo

• Para orientar su comportamiento de alguna manera Con el método registerForCourse mostrado anteriormente, por ejemplo, es necesario decir el objeto Student específico que realiza el método en el que queremos registrarlo; bien Hágalo pasando dos argumentos, un ID de curso (p. ej., "MATH 101") y un número de sección (p. ej., 10, que se reúne los lunes por la noche de 8:00 a 10:00 p.m.), como se ilustra aquí:

boolean registerForCourse (String courseID, int secNo)

Si hubiéramos declarado en su lugar el encabezado del método registerForCourse con una lista de parámetros vacía

registro booleanoForCourse ()

la solicitud sería ambigua, porque el objeto Student que realiza este método no tengo idea de en qué sección se espera que se registre.

Sin embargo, no todos los métodos requieren tal "combustible"; algunos métodos pueden producir resultados basándose únicamente en la información almacenada internamente dentro de un objeto como valores de atributo, en los quencaso no se necesita orientación adicional en forma de argumentos. Por ejemplo, el método

int getAge ()

está diseñado para no tener parámetros porque un objeto Student presumiblemente puede decirnos su edad sin tener que recibir cualquier información calificativa, tal vez comparando el valor de su fecha de nacimiento atributo con la fecha del sistema. Sin embargo, supongamos que queríamos que un objeto Student pudiera informar su edad expresada en años o en meseshs; en tal caso, podríamos querer declarar el método getAge de la siguiente manera:

int getAge (int ageType)

permitiéndonos pasar un argumento int (eger) para que sirva como bandera de control para informar al estudiante objeto de cómo queremos que se devuelva la respuesta. Es decir, podríamos programar el método getAge así que eso

• Si pasamos un valor de 1, significa que queremos que la respuesta se devuelva en términos de años.

• Si pasamos un valor de 2, queremos que la respuesta se devuelva en términos de meses (por ejemplo, un estudiante de 21 años respondería que tiene 252 meses).

Una forma alternativa de manejar el requisito de recuperar la edad de un objeto Student en dos formas diferentes serían definir dos métodos separados, como quizás el siguiente:

int getAgeInYears ()

int getAgeInMonths ()

pero en la programación orientada a objetos, es una práctica común controlar el comportamiento de un método a través de los valores (y tipos) de argumentos.

**UN POCO DE TERMINOLOGÍA**

Para asegurarse de que todos comprendan la diferencia entre "parámetro" y "argumento", ambos son términos genéricos que son igualmente aplicables a los lenguajes de programación OO y no OO, definiremos ambos términos aquí.

Un parámetro es una variable de ámbito local, declarada en un encabezado de método, que aparece temporalmente mientras se ejecuta un método. Por ejemplo, cuando el método de raíz cuadrada (lenguaje C), declarado de la siguiente manera, se invoca:

// Código C.

doble sqrt (doble x) {...}

el parámetro x (de tipo double) aparece temporalmente mientras se ejecuta el método. x se inicializa con el valor del argumento que se pasa cuando se invoca el método:

// Código C.

doble y = 39,7;

Aquí, estamos invocando la función sqrt, pasando un valor de argumento de 39,7.

doble z = raíz cuadrada (y);

En este caso, dado que estamos invocando el método sqrt con un valor de argumento de 39.7, el parámetro x se inicializa con el valor 39,7.

Mientras se ejecuta el método, el cuerpo del método puede hacer referencia a la variable x como mejor le parezca; sin embargo, La variable x deja de existir en lo que respecta al compilador, es decir, x sale del alcance (un concepto que discutido en el Capítulo 2) - cuando el método sale.

doble sqrt (doble x) {

// Usa el valor de x según corresponda.

si (x <0) {...}

más {...}

// Pseudocódigo.

devolver resultado;

}

// "x" sale del alcance aquí ... solo se define localmente en el método sqrt.

Aunque los términos "parámetro" y "argumento" se utilizan indistintamente con frecuencia, la conclusión es que son conceptos diferentes, es decir, los argumentos son valores; los parámetros son variables.

**Tipos de retorno de método**

Se muestra que el método registerForCourse como se declaró anteriormente tiene un tipo de retorno de booleano, lo que implica que este método devolverá uno de los siguientes dos valores:

• Un valor de verdadero, para señalar "misión cumplida", es decir, que el objeto Student ha se registró con éxito para el curso en el que se le indicó que se registrara.

• Un valor de falso, para indicar que la solicitud de registro ha sido denegada por algún motivo. Quizás la sección deseada estaba llena o el alumno no cumplía con los requisitos previos del curso, o el curso / sección solicitada ha sido cancelada, etc.

En el Capítulo 13, aprenderá técnicas para comunicarse y determinar con precisión por qué la misión de un método ha fallado cuando discutimos el manejo de excepciones.

Tenga en cuenta que un método puede estar diseñado para no devolver nada, es decir, puede funcionar su negocio en silencio, sin necesidad de informar el resultado de sus esfuerzos. Si es así, se declara tienen un tipo de retorno de vacío (otra de las palabras clave de Java). Como ejemplo, considere el Student encabezado del método:

void setName (String newName)

Este método requiere un argumento: una cadena que representa el nuevo nombre que queremos. Objeto de estudiante para asumir, y realiza "silenciosamente" estableciendo el nombre interno del objeto de estudiante atributo a cualquier valor que se esté pasando al método, sin devolver ninguna respuesta en respuesta. Aquí hay un ejemplo adicional de un encabezado de método que podríamos declarar para la clase Student. con un tipo de retorno vacío:

void switchMajor (String newDepartment, Professor newAdvisor)

Este método representa una solicitud para que un objeto Student cambie su campo principal de estudio, que implica designar tanto un nuevo departamento académico (por ejemplo, "BIOLOGÍA") como una referencia al profesor objeto que es servir como asesor del estudiante en este nuevo departamento. El ejemplo anterior demuestra que podemos declarar parámetros de cualquier tipo, incluidos los tipos definidos por el usuario como Professor. Lo mismo es cierto para el tipo de retorno de un método: por ejemplo, un método con el siguiente encabezado:

Profesor getAdvisor ()

podría usarse para preguntar a un objeto Student quién es su asesor. En lugar de simplemente devolver el nombre del asesor, el objeto Student devuelve una referencia al objeto Professor como un todo (como se registra

por el atributo interno del asesor de profesores del estudiante; aprenderá a informar a un objeto Student cuyo objeto Profesor es tservir como su consejero de la facultad un poco más adelante en el capítulo).

Tenga en cuenta que un método puede devolver como máximo un resultado, lo que puede parecer limitante. ¿Y si, Por ejemplo, queremos pedirle a un objeto Student una lista de todos los cursos que el alumno ha tomado: ¿debemos pedirlos uno por uno a través de múltiples llamadas a métodos? Afortunadamente no: el El resultado devuelto por un método puede ser en realidad una referencia a un objeto de complejidad arbitraria, incluyendo un tipo especial de objeto llamado colección que puede contener referencias a múltiples otros objetos. Hablaremos sobre colecciones en profundidad en el Capítulo 6.

**Una analogía**

Usemos una analogía para ayudar a ilustrar lo que hemos discutido hasta ahora acerca de los métodos. Con Respeto a las tareas del hogar, digamos que una persona es capaz de

• Sacando la basura

• Cortar el césped

• Lavando el carro

Expresando esta noción en código Java, quizás declararíamos tres métodos para la clase Person, uno para cada tarea (servicio):

• sacar la basura

• cortar el césped

• lava el auto

En el caso del método takeOutTheTrash, no necesitamos proporcionar ningún detalle de calificación en la forma de los argumentos, ni esperamos que la persona que realiza este servicio (método) informarnos, por lo que declaramos el encabezado del método con un tipo de retorno de void y un vacío lista de parámetros:

anular takeOutTheTrash ()

En el caso del método mowTheLawn, nos gustaría que quien esté cortando el césped informe a nosotros en cuanto a si él o ella ve alguna hierba de cangrejo, pero nuevamente, no necesitamos proporcionar ninguna calificación detalles en forma de argumentos, por lo que declaramos el encabezado del método con un tipo de retorno de booleano (donde verdadero significa que la persona vio hierba de cangrejo, y falso significa que la persona no lo vio) y una lista de parámetros vacía:

booleano mowTheLawn ()

Finalmente, en el caso del método washTheCar, podríamos poseer varios autos diferentes, por lo que Necesitamos especificar qué automóvil se va a lavar pasando una referencia al automóvil de interés. Nosotros Sin embargo, no necesitamos ningún tipo de respuesta de la persona que lava la ropa, por lo que podríamos elabore el siguiente encabezado de método:

vacío washTheCar (Coche c)

Revisaremos esta analogía de las "tareas" para ampliarla más adelante en este capítulo.

**Cuerpos de métodos**

Cuando diseñamos y programamos los métodos de una clase, declarar los encabezados de los métodos por sí solos no es suficiente: también debemos programar los detalles internos de cómo debe comportarse cada método cuando se invoca.

Estos detalles de programación interna, conocidos como el cuerpo del método, se incluyen entre llaves {...} inmediatamente después del encabezado del método, de la siguiente manera:

estudiante de clase pública {

// Atributos.

Nombre de cadena;

doble gpa;

// En este ejemplo se han omitido otras declaraciones de atributos de Student ...

// Declaramos un encabezado de método ...

boolean isHonorsStudent () {

// ... y programe los detalles de lo que debe hacer este método

// entre llaves adjuntas ... este es el cuerpo del método.

// Aquí, accedemos al valor de "gpa", declarado como

// atributo de la clase Student anterior.

si (gpa> = 3.5) {

// Devolver el valor "verdadero" indica "sí, esto es

// un estudiante de honor ".

devuelve verdadero;

}

else {

// Devolver el valor "falso" indica "no, esto no es

// un estudiante de honor ".

falso retorno;

}

}

// Seguirían otras declaraciones de métodos para la clase Student ...

// detalles omitidos.

}

Por tanto, podemos ver que un método es una función, una función que realiza un objeto específico, pero una función no obstante.

**Las características pueden declararse en cualquier orden**

Tenga en cuenta que el orden relativo en el que se declaran las características dentro de una clase Java no importa. Es decir, se nos permite hacer referencia a una característica A desde el método B aunque la declaración de la característica A viene después de la declaración del método B en la declaración de clase general.

Por ejemplo, en la siguiente clase simple, declaramos dos métodos, foo y bar, y uno atributo, x. El método foo puede invocar el método bar, a pesar de que la declaración de bar viene después de la declaración de foo en la clase:

public class Simple {

// Atributos.

int x;

// Métodos.

void foo () {

// Invocar bar () desde foo.

bar();

}

// bar () se declara DESPUÉS de foo ().

barra vacía () {

System.out.println (x);

}

}

No todos los idiomas son iguales en este sentido; en C ++, por ejemplo, solo puede hacer referencia a una característica si ha sido previamente declarado. Por lo tanto, invocar la barra de foo generaría un error de compilación si el

El ejemplo anterior fue un ejemplo de C ++ versus Java.

De manera similar, las declaraciones de atributos no necesitan preceder a las declaraciones de métodos para una clase; es por lo tanto Está permitido reescribir nuestra clase de estudiantes de la siguiente manera:

estudiante de clase pública {

// Aquí, EMPEZAMOS con declaraciones de métodos ...

void foo () {

bar();

}

barra vacía () {

// Podemos hacer referencia al atributo 'x' a pesar del hecho

// que su declaración no

// ha sido "visto" por el compilador todavía.

System.out.println (x);

}

// ... y END con declaraciones de atributos.

int x;

}

Sin embargo, es una práctica común consolidar todas las declaraciones de atributos al principiode una clase, antes de declarar cualquiera de sus métodos.

**declaraciones de devolución**

Una declaración de retorno es una declaración de salto que se usa para salir de un método:

void doSomething () {

// Pseudocódigo.

haga lo que sea requerido por este método ...

regreso;

}

Siempre que se encuentra una declaración de retorno, el método deja de ejecutarse a partir de esa línea de código, y el control de ejecución vuelve inmediatamente al código que invocó el método en el primer lugar.

Para los métodos con un tipo de retorno de vacío, la palabra clave de retorno se utiliza por sí misma, como un declaración:

Return;

Sin embargo, resulta que para los métodos con un tipo de retorno de vacío, el uso de un retorno; declaración es opcional. Si se omite, una devolución; La declaración está implícita como la última línea del método. Es decir, las siguientes dos versiones del método doSomething son equivalentes:

void doSomething () {

int x = 3;

int y = 4;

int z = x + y;

}

and

void doSomething () {

int x = 3;

int y = 4;

int z = x + y;

return;

}

Los cuerpos de métodos con un tipo de retorno no nulo, por otro lado, deben incluir en al menos una declaración de retorno explícita. La palabra clave return en tal caso debe ir seguida de una expresión que se evalúa como un valor compatible con el tipo de retorno declarado del método. Por ejemplo, si un método está definido para tener un tipo de retorno de int, entonces cualquiera de los siguientes retornos declaraciones serían aceptables:

return 0; // devolviendo un valor entero constante

return x; // devolviendo el valor de x (asumiendo que x

// ha sido previamente declarado como un int)

return x + y; // devolviendo el valor de la expresión "x + y" (aquí,

// asumimos que "x + y" se evalúa como un valor int)

return (int) z; // lanzar el valor de z (suponga que z se declaró como un doble)

// a un valor int

Etcétera. Como otro ejemplo, si un método está definido para tener un tipo de retorno booleano, entonces cualquiera de las siguientes declaraciones de devolución sería aceptable:

falso retorno; // devolviendo un valor constante booleano

resultado de retorno; // devolviendo el valor del resultado variable

// (asumiendo que el resultado ha sido previamente

// declarado de tipo booleano)

retorno (x <3); // devolviendo el valor booleano que resulta cuando

// el valor (numérico) de x se compara con 3:

// si x es menor que 3, este método devuelve un

// valor de verdadero; de lo contrario, devuelve falso.

El cuerpo de un método puede incluir más de una declaración de devolución. Buena programación La práctica, sin embargo, es tener solo una declaración de retorno en un método, al final. Veamos una vez nuevamente en el método isHonorsStudent discutido anteriormente, que tiene dos declaraciones de retorno:

boolean isHonorsStudent () {

si (gpa> = 3.5) {

devuelve verdadero; // primera declaración de retorno

}

else {

falso retorno; // segunda declaración de retorno

}

}

Reescribamos este método para usar una variable booleana declarada localmente, result, para capturar la respuesta verdadera / falsa que finalmente se devolverá. Devolveremos el valor del resultado con un solo declaración de retorno al final del método:

boolean isHonorsStudent () {

// Declara una variable local para realizar un seguimiento del resultado; arbitrariamente

// inicializarlo en falso.

resultado booleano = falso;

si (gpa> = 3.5) {

// En lugar de devolver verdadero, registramos el valor en nuestro "resultado"

// variable:

resultado = verdadero;

}

else {

// En lugar de devolver falso, registramos el valor en nuestro "resultado"

// variable:

resultado = falso;

}

// Ahora tenemos una única declaración de retorno al final de nuestro método para devolver el

// resultado.

devolver resultado;

}

Como resultado, dado que inicialmente asignamos el valor falso al resultado, configurándolo en falso explícitamente en la cláusula else es innecesario; por lo tanto, podríamos simplificar el isHonorsStudent

método de la siguiente manera:

boolean isHonorsStudent () {

// Declara una variable local para realizar un seguimiento del resultado; arbitrariamente

// inicializarlo en falso.

resultado booleano = falso;

si (gpa> = 3.5) {

resultado = verdadero;

}

// Tenga en cuenta que hemos eliminado la cláusula 'else' ... si la prueba "if"

// falla, la variable "resultado" ya tiene un valor de falso.

devolver resultado;

}

Sin embargo, existe una situación en la que se consideran múltiples declaraciones de retorno aceptable, y es entonces cuando un método necesita realizar una serie de operaciones donde falla en cualquier paso del camino constituye el fracaso como un todo. Esta situación se ilustra a través de pseudocódigo:

// Pseudocódigo.

boolean someMethod () {

// Realizar una prueba ... si falla, deseamos abortar el método como

// entero.

si (la primera prueba falla) devuelve falso;

// Si pasamos la primera prueba, hacemos algún procesamiento adicional ...

haz algo interesante ...

// Entonces, tal vez realicemos una segunda prueba, donde nuevamente falla el

// prueba justifica rendirse inmediatamente en nuestra "misión".

si (la segunda prueba falla) devuelve falso;

// Si pasamos la segunda prueba, hacemos algún procesamiento adicional ...

// detalles omitidos.

// Si llegamos a este punto en nuestro código, devolvemos un valor de verdadero

// ¡para señalar que llegamos a la meta!

devuelve verdadero;

}

Tenga en cuenta que el compilador de Java verificará que todas las rutas lógicasa través de un método de retorno un resultado debidamente mecanografiado. Por ejemplo, el siguiente método generará un error del compilador porque solo se alcanzará una declaración de retorno adecuada si la prueba if tiene éxito; si la prueba if falla,

la declaración de retorno se omite:

booleano xGreaterThanThree (int x) {

si (x <= 3) devuelve falso;

}

El mensaje de error del compilador específico en este caso sería el siguiente: sentencia devuelta faltante:

booleano xGreaterThanThree (int x) {

**Los métodos implementan reglas comerciales**

La lógica contenida en el cuerpo de un método define la lógica empresarial, también conocida como reglas, por una abstracción. Por ejemplo, en el método isHonorsStudent

boolean isHonorsStudent () {

resultado booleano = falso;

si (gpa> = 3.5) {

resultado = verdadero;

}

devolver resultado;

}

se expresa una sola regla comercial para determinar si un estudiante es o no un estudiante con honores, a saber,

Si un estudiante tiene un promedio de calificaciones (GPA) de 3.5 o más, entonces él o ella es un estudiante de honor.

Si las reglas de negocio subyacentes a este método fueran más complejas, digamos, si las reglas fueran las siguientes: Para que un estudiante sea considerado un estudiante con honores, el estudiante debe

(a) Tener un promedio de calificaciones (GPA) de 3.5 o más

(b) Haber tomado al menos tres cursos

(c) No haber recibido una calificación inferior a "B" en ninguno de estos cursos.

entonces la lógica de nuestro método sería necesariamente más compleja:

boolean isHonorsStudent () {

resultado booleano = falso;

// Pseudocódigo.

si ((gpa> = 3.5) &&

(número de cursos tomados> = 3) &&

(no se han recibido calificaciones inferiores a una B)) {

resultado = verdadero;

}

devolver resultado;

}

En cierto sentido, incluso el encabezado de un método expresa una forma simple de regla / requisito comercial; en este caso particular, que existe una noción como un "estudiante de honor" en primer lugar. Pero, los detalles de las reglas de negocio de una aplicación están codificados en los cuerpos de métodos de sus diversas clases.

**Objetos como contexto para la invocación de métodos**

Como se mencionó de pasada un poco antes en el capítulo, los métodos en un OOPL difieren de las funciones en un no OOPL en que

• Las funciones son ejecutadas por el entorno de programación como un todo, mientras que

• Los métodos son ejecutados por objetos específicos

Es decir, podemos invocar una función C "en el vacío" de la siguiente manera:

// Un programa en C.

vacío principal() {

hacer algo (42.0); // invocar la función doSomething ...

// etc.

}

mientras que en un OOPL como Java, normalmente debemos calificar la llamada al método prefijándola con el nombre de la variable de referencia que representa el objeto que va a realizar el método, seguido por un punto (punto). Esto se ilustra para el método registerForCourse de la siguiente manera:

// Crea una instancia de dos objetos Student.

Estudiante x = nuevo Estudiante ();

Estudiante y = nuevo Estudiante ();

// Invocar el método registerForCourse en el objeto Student x, pidiéndole que

// registrarse para el curso MATH 101, sección 10; El estudiante no se ve afectado.

x.registerForCourse ("MATEMÁTICAS 101", 10);

Nos referimos a una expresión de la forma referenceVariable.methodName (args) como un mensaje.

Es decir, esta línea de código

x.registerForCourse ("MATEMÁTICAS 101", 10);

se puede interpretar como "invocar un método en el objeto x" o como "enviar un mensaje al objeto x". De cualquier manera, dicho código debe verse como una solicitud del objeto x para realizar un método como servicio, en nombre de la aplicación a la que pertenece el objeto.

La terminología "enviar un mensaje a un objeto" se originó con el lenguaje Smalltalk y se utiliza cuando hablando de OOPL de forma genérica. Cuando se habla de Java específicamente, la terminología "invocando un método en se prefiere un objeto ”. De manera similar, la alternativa específica de Java para el término genérico de OOPL "mensaje" es "método invocación." A lo largo del libro, alternaré entre las formas genérica y específica de Java para referirme a estas nociones, pero tienden a favorecer la nomenclatura genérica de "mensaje".

Debido a que usamos un "punto" para agregar una llamada a un método a una variable de referencia particular, informalmente consulte la notación referenceVariable.methodName (args) como notación de puntos.

Otra forma informal de pensar en la notación x.methodName (args) es que estamos "hablando con"objeto x; específicamente, que estamos "hablando con" el objeto x para solicitarle que realice un método particular /Servicio. Regresemos a la analogía de las tareas domésticas presentada anteriormente en el capítulo para ilustre este punto.

Recuerde que una persona es capaz de realizar las siguientes tareas domésticas:

• Sacando la basura

• Cortar el césped

• Lavando el carro

A continuación, se muestra una expresión de esta abstracción como código Java:

Persona de clase pública {

// Atributos omitidos de este fragmento ...

// Métodos.

void takeOutTheTrash () {...}

boolean mowTheLawn () {...}

void washTheCar (Coche c) {...}

}

Decidimos que queremos que nuestros hijos adolescentes Larry, Moe y Curly hagan cada uno uno de estostres quehaceres. ¿Cómo les pediríamos que hicieran esto? Si tuviéramos que decir simplemente

• "Por favor, lava el Camry".

• "Por favor, saque la basura".

• "Por favor, corte el césped y avíseme si ve hierba de cangrejo".

lo más probable es que ninguno de loslas tareas domésticas se harían, porque no le hemos encargado a un hijo específico cumpliendo cualquiera de estas solicitudes! Larry, Moe y Curly probablemente permanecerán pegados a la televisión, porque ninguno de ellos reconocerá que se les ha dirigido una solicitud específicamente. Por otro lado, si en cambio dijéramos

• "Larry, lava el Camry".

• "Moe, saca la basura".

• "Curly, por favor, corte el césped y avíseme si ve hierba de cangrejo".

dirigiríamos cada solicitud a un hijo específico; de nuevo, usando la sintaxis de Java, esto podría ser expresado como sigue:

// Declaramos e instanciamos tres objetos Person:

Persona larry = nueva Persona ();

Persona moe = nueva Persona ();

Persona rizada = nueva Persona ();

// Y, ya que estamos en eso, ¡un objeto Car también!

Coche camry = coche nuevo ();

// Enviamos un mensaje a cada hijo, indicando el servicio que deseamos

// cada uno de ellos para realizar:

larry.washTheCar (camry);

moe.takeOutTheTrash ();

CrabgrassFound booleano = curly.mowTheLawn ();

if (crabgrassFound) {

// Pseudocódigo.

manejar la hierba de cangrejo ...

}

Al aplicar cada llamada de método a un "hijo" específico (referencia de objeto de persona), no hay ambigüedad en cuanto a qué objeto se le pide que realice qué servicio.

Suponiendo que takeOutTheTrash es un método definido para la clase Person como anteriormente ilustrado, el siguiente código no se compilará en Java (o, para el caso, en cualquier OOPL) porque la llamada al método no está calificada, es decir, falta la notación de puntos:

public class BadCode {

public static void main (String [] args) {

// La siguiente línea no se compilará, ¿dónde está el "punto"? Es decir, que objeto

// estamos hablando ???

sacar la basura();

}

}

Se informará del siguiente error de compilación:

no puedo encontrar el símbolo

símbolo: método takeOutTheTrash ()

ubicación: clase BadCode

Sin embargo, en un lenguaje que no es OOPL como C, no hay noción de objetos o clases, por lo que Las funciones en dichos lenguajes siempre se invocan "en el vacío" (es decir, en el entorno de programación como un todo):

// Un programa en C.

vacío principal() {

hacer algo (42.0);

// etc.

}

**Expresiones Java, revisadas**

Cuando discutimos las expresiones de Java en el Capítulo 2, había una forma de expresión que era omitidos de la lista, es decir, mensajes, porque aún no habíamos hablado de objetos. He repitió la lista de lo que constituye expresiones de Java aquí, agregando expresiones de mensaje a la mezcla:

• Una constante: 7, falsa

• Un carácter literal (acter): 'A', '&'

• Un literal de cadena: "foo"

• El nombre de cualquier variable declarada de uno de los tipos predefinidos que hemos visto

hasta ahora: myString, x

• Cualquiera de los anteriores que sea modificado por uno de los operadores unarios de Java: i ++

• Una invocación de método ("mensaje"): z.length ()

• Dos de los anteriores que se combinan con uno de los operadores binarios de Java:

longitud z () + 2

• Cualquiera de las expresiones simples anteriores entre paréntesis: (z.length () + 2)

El tipo de expresión de mensaje es el tipo de resultado que devuelve el método. Por ejemplo, si length () es un método con un tipo de retorno de int, entonces la expresión z.length () es una expresión de tipo int y, si registerForCourse es un método con un tipo de retorno booleano, entonces la expresión s.registerForCourse (...) es una expresión de tipo booleano.

**Capturando el valor devuelto por un método**

Siempre que invocamos un método con un tipo de devolución no nulo, depende de nosotros elegir ignorar o reaccionar al valor que devuelve el método. En un ejemplo anterior, declaramos la clase Student registerForCourse para tener un tipo de retorno de booleano:

boolean registerForCourse (String courseId, int sectionNumber)

pero no prestamos atención al valor booleano que se devolvió cuando invocamos el método:

x.registerForCourse ("MATEMÁTICAS 101", 10);

Si deseamos reaccionar al valor devuelto por un método no nulo, podemos optar por capturar el valor de una variable declarada del tipo apropiado, como en el siguiente ejemplo:

booleano registrado con éxito = x.registerForCourse ("MATH 101", 10);

if (! SuccessRegistered) {// o: if (SuccessRegistered == false)

// Pseudocódigo.

acción a tomar si el registro falla ...

}

Sin embargo, si solo planeamos usar el valor devuelto de un método una vez en nuestro código, entonces vamos a la molestia de declarar una variable explícita como SuccessRegistered para capturar el el resultado es exagerado. En su lugar, podemos reaccionar al resultado simplemente anidando una expresión de mensaje dentro de una declaración más compleja. Por ejemplo, podemos reescribir el fragmento de código anterior en eliminar la variable correctamente Registrada de la siguiente manera:

if (! (x.registerForCourse ("MATH 101", 10))) {

// Pseudocódigo.

acción a tomar si el registro falla ...

}

Dado que el método registerForCourse devuelve un valor booleano, el mensaje x.registerForCourse (...) es una expresión booleana y se puede utilizar dentro de la cláusula if de una si declaración. Además, podemos aplicar el! ("No") a la expresión, como en el ejemplo anterior.

A menudo combinamos llamadas a métodos con otros tiposde declaraciones al desarrollar objetos orientados a aplicaciones, por ejemplo, al devolver valores de métodos:

estudiante de clase pública {

Profesor consejero;

// Detalles omitidos.

public String getAdvisorsDepartment () {

return advisor.getDepartment (); // una expresión de cadena

}

// etc.

}

o al imprimir en la ventana de comandos:

Estudiante s = nuevo Estudiante ();

// Detalles omitidos.

System.out.println ("El estudiante llamado" + s.getName () +

"tiene un GPA de" + s.getGPA ());

etc.

**Firmas de métodos**

Ya aprendió que el encabezado de un método consiste, como mínimo, en el retorno del método tipo, nombre y lista de parámetros formales:

void switchMajor (String newDepartment, Professor newAdvisor)

Sin embargo, desde el punto de vista del código utilizado para invocar un método en un objeto, el tipo de retorno y los nombres de los parámetros no son inmediatamente evidentes tras la inspección:

Estudiante s = nuevo Estudiante ();

Profesor p = nuevo profesor ();

// Detalles omitidos ...

s.chooseMajor ("MATEMÁTICAS", p);

Podemos inferir de la inspección de la última línea de código que

• chooseMajor es el nombre de un método definido para la clase Student; de lo contrario, el compilador rechazaría esta línea.

• El método chooseMajor declara dos parámetros de tipo String y Professor, respectivamente, porque esos son los tipos de argumentos que transmitimos: específicamente, un literal String y una referencia a un objeto Professor.

Sin embargo, lo que no podemos determinar al inspeccionar este código es (a) cómo los parámetros formales fueron nombrados en el encabezado del método correspondiente, o (b) cuál es el tipo de retorno de este método declarado ser; puede ser nulo, o el método puede estar devolviendo un resultado no nulo que hemos simplemente elegido para ignorar.

Por esta razón, nos referimos a la firma de un método como aquellos aspectos de un encabezado de método que son "detectables" mediante la inspección del código utilizado para invocar el método, es decir,

• El nombre del método

• El orden, los tipos y la cantidad de parámetros declarados por el método pero excluyendo

• Los nombres de los parámetros

• El tipo de retorno del método

Además, presentaremos la firma del argumento de terminología informal para referirnos a eso subconjunto de una firma de método que consta del orden, los tipos y el número de argumentos, pero excluyendo el nombre del método.

"Firma de argumento" no es un término estándar de la industria, pero es útil. Lo usaremos en todas partes el libro.

Algunos ejemplos de encabezados de método y sus correspondientes firmas de método / argumento son como sigue:

• Method header: int getAge (int ageType)

• Method signature: getAge (int)

• Argument signature: (int)

Method header: void chooseMajor (String newDepartment, Professor newAdvisor)

• Method signature: ChooseMajor (String, Professor)

• Argument signature: (String, Professor)

• Method header: String getName ()

• Method signature: getName ()

• Argument signature: ()

**Elección de nombres de métodos descriptivos**

Asignar nombres descriptivos e intuitivos a nuestros métodos ayuda a que el código de una aplicación se autodocumente.

Cuando se combina con nombres de variables cuidadosamente elaborados, como los elegidos en En el siguiente ejemplo de código, los comentarios son (virtualmente) innecesarios:

public class IntuitiveNames {

public static void main (String [] args) {

Estudiante estudiante;

Profesor profesor;

Curso course1;

Course course2;

Course course3;

// Más adelante en el programa ...

// ¡Este código es bastante sencillo de entender!

// Un alumno elige a un profesor como su asesor ...

student.chooseAdvisor (profesor);

// ... y se inscribe para el primero de tres cursos.

student.registerForCourse (curso1);

// etc.

Ahora, contrasta el código anterior con el código mucho más "difuso" que sigue:

public class FuzzyNames {

public static void main (String [] args) {

Estudiante s;

Profesor p;

Curso c1;

Curso c2;

Curso c3;

// Más adelante en el programa ...

// Sin comentarios, este siguiente fragmento de código no es tan intuitivo.

s.elija (p);

s.reg (c1);

// etc.

**Sobrecarga de métodos**

La sobrecarga es un mecanismo de lenguaje que permite dos o más métodos diferentes pertenecientes al misma clase para tener el mismo nombre siempre que tengan diferentes firmas de argumentos. Sobrecarga es compatible con numerosos lenguajes que no son OO como C, así como con lenguajes OO como Java.

Por ejemplo, la clase de Estudiante puede definir legítimamente las siguientes cinco impresiones diferentes encabezados de método:

void print (String fileName) {... // versión # 1

void print (int detailLevel) {... // versión # 2

void print (int detailLevel, String fileName) {... // versión # 3

int print (String reportTitle, int maxPages) {... // versión # 4

boolean print () {... // versión # 5

y por lo tanto se dice que el método de impresión está sobrecargado. Tenga en cuenta que los cinco métodos difieren en términos de sus firmas de argumentos:

• El primero toma una sola cadena como argumento.

• El segundo toma un solo int.

• El tercero toma dos argumentos: un int seguido de un String.

• El cuarto toma dos argumentos: una cadena seguida de un int (aunque estos son los

mismos tipos de parámetros que en el encabezado anterior, están en un orden diferente). El quinto no admite ningún argumento.

Por lo tanto, estos cinco encabezados representan métodos válidos y diferentes, y todos pueden coexistir felizmente dentro de la clase Student sin ninguna queja del compilador.

Luego podemos elegir cuál de estos cinco "sabores" de método de impresión nos gustaría un objeto Student realizar según la forma de mensaje que enviamos a un objeto Student:

Estudiante s = nuevo Estudiante ();

// Invocando la versión de print que toma un solo argumento String.

s.print ("salida.rpt");

// Invocando la versión de print que toma un único argumento int.

s.print (2);

// Invocando la versión que toma dos argumentos, un int seguido de un String.

s.print (2, "salida.rpt");

// etc.

El compilador puede hacer coincidir sin ambigüedades qué versión del método de impresión se está utilizando.

llamado en cada instancia en función de las firmas de los argumentos.

Este ejemplo ilustra por qué los métodos sobrecargados deben tener firmas de argumentos únicas:

si se nos permitiera introducir el siguiente método de impresión adicional como sexto método de

Estudiante

boolean print (int levelOfDetail) {... // versión # 6

•a pesar del hecho de que la firma de su argumento (un único int) duplica la firma del argumento de uno de los otros cinco métodos de impresión

void print (int detailLevel) {... // versión # 2

entonces el compilador no podría determinar qué versión del método de impresión, # 2 o # 6, estamos intentando invocar con la siguiente línea de código:

s.print (3); // ¿Qué versión queremos ejecutar: # 2 o # 6? ¡¡¡AYUDA!!!

Entonces, para simplificar la vida, el compilador evita que este tipo de ambigüedad surja en el primer lugar al evitar que las clases declaren métodos con nombres similares con argumentos idénticos firmas. El error del compilador que generaríamos si intentáramos declarar la versión n. ° 6 de la impresión El método junto con las otras cinco versiones sería el siguiente:

print (int) ya está definido en Student

impresión booleana (int levelOfDetail) {

^

La capacidad de sobrecargar métodos nos permite crear una familia completa de nombres similares métodos que hacen esencialmente el mismo trabajo. Piense en el Capítulo 2 donde discutimos el Sistema.

método out.println, que se utiliza para mostrar la salida impresa en la ventana de comandos. Como lo resulta que no hay una, sino muchas versiones del método System.out.println; cada uno sobrecargado La versión acepta un tipo de argumento diferente (println (int), println (String), println (doble), etc.).

Usar un método System.out.println sobrecargado es mucho más simple y ordenado que tener que use métodos separados llamados printlnString, printlnInt, printlnDouble, etc.

Tenga en cuenta que no existe la sobrecarga de atributos; es decir, si una clase intenta declarar dos atributos con el mismo nombre

estudiante de clase pública {

private String studentId;

private int studentId;

// etc.

el compilador generará un mensaje de error en la segunda declaración:

studentId ya está definido en Student

**Mensaje que pasa entre objetos**

Veamos ahora un ejemplo de transmisión de mensajes que involucra dos objetos. Supongamos que tenemos dos clases definidas — Estudiante y Curso — y que se definen los siguientes métodos para cada una.

• Para la clase de estudiantes:

booleano completado con éxito (curso c)

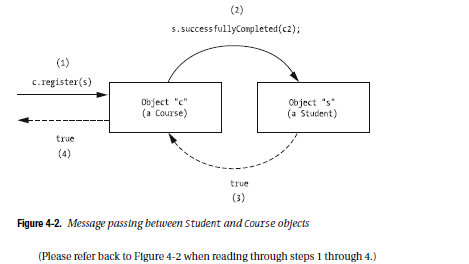
Dada una referencia c a un objeto de curso en particular, le pedimos al objeto Student que recibe este mensaje para confirmar que el alumno ha realizado efectivamente el curso en cuestión y recibió una calificación aprobatoria.

• Para la clase del curso:

registro booleano (Student s)

Dada una referencia a un objeto Student en particular, le pedimos al objeto Course que recibe este mensaje para hacer lo que sea necesario para registrar al alumno. En este caso, esperamos un curso para finalmente responder verdadero o falso para indicar éxito o fracaso del registro solicitud.

La Figura 4-2 refleja un posible intercambio de mensajes entre un objeto del curso cy un estudiante objetos; cada paso numerado del diagrama se narra en el texto que sigue. Flechas de línea continua representar mensajes que se pasan / métodos que se invocan; las flechas de línea discontinua representan valores siendo devuelto por métodos.



(Consulte la Figura 4-2 cuando lea los pasos 1 a 4.)

1. Un objeto de curso c recibe el mensaje

c. registro (s);

donde s representa un objeto Student en particular. (Por ahora, no nos preocuparemos por el origen de este mensaje; Lo más probable es que se haya desencadenado por la interacción de un usuario con la GUI de SRS. Bien ver el contexto de código completo de cómo se emiten todos estos mensajes más adelante en este capítulo, en la sección titulada "Objetos como clientes y proveedores").

2. Para que el objeto del curso c determine oficialmente si se debe permitir o no s para registrarse, c envía el mensaje

s.successfullyCompleted (c2);

a Student s, donde c2 representa una referencia a un objeto de curso diferente que ocurre ser un requisito previo del curso c. (No se preocupe por cómo el Curso c sabe que c2 es uno de sus prerrequisitos; esto implica interactuar con el atributo de requisitos previos internos de c, que todavía no hemos hablado. Además, el curso c2 no se muestra en la figura 4-2 porque, estrictamente hablando, c2 no participa en esta "discusión" entre los objetos cy s. c2 se está hablando sobre, pero no está hablando en sí mismo!) Figura 4-2. Transmisión de mensajes entre los objetos Student y Course Figura 4-3. Un solicitante solo ve los detalles externos de un intercambio de mensajes.

3. El objeto de estudiante s responde ac con el valor verdadero, lo que indica que s se completó con éxito el curso de prerrequisito. (Por el momento ignoraremos los detalles sobre cómo determina esto; implica interactuar con el atributo de transcripción interna de s, que no hemos cubierto la estructura de todavía).

4. Convencido de que el estudiante ha cumplido con los requisitos previos para el curso, el objeto del curso c finaliza el trabajo de registrar al estudiante (se omiten los detalles internos por ahora) y confirma el registro respondiendo con un valor de verdadero al originador de la solicitud de servicio.

Este ejemplo fue demasiado simplista; En realidad, el Curso c pudo haber tenido que hablar con numerosos otros objetos también:

• Un objeto Classroom (elsala en la que se va a realizar el curso, para asegurarse de que ha espacio suficiente para otro estudiante)

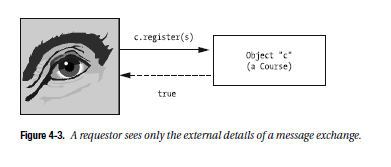
• Un objeto del Programa de Grado (el título buscado por el estudiante, para asegurarse de que el de hecho, se requiere un curso para el título que el estudiante está cursando) y así sucesivamente, antes de enviar una respuesta verdadera para indicar que la solicitud para registrar Student sse había cumplido.

Veremos una versión un poco más compleja de este intercambio de mensajes más adelante en el capítulo.

**Delegación**

Si se hace una solicitud de un objeto A y, al cumplir con la solicitud, A a su vez solicita asistencia de otro objeto B, esto se conoce como delegación de A a B. El concepto de delegación entre objetos es exactamente lo mismo que la delegación entre personas en el mundo real: si su otro ”le pide que corte el césped mientras él o ella están haciendo recados, y usted a su vez contrata un adolescente del vecindario para cortar el césped, entonces, en lo que respecta a su pareja, el césped ha sido cortado. El hecho de que haya delegado la actividad a otra persona es (¡con suerte!) Irrelevante.

El hecho de que la delegación se haya producido entre objetos suele ser transparente para el iniciador. de un mensaje, también. En nuestro ejemplo anterior de transmisión de mensajes, el Curso c delegó parte del trabajo de inscribir a los estudiantes de nuevo a s cuando c les pidió que verificaran haber tomado un curso de prerrequisito. Sin embargo, desde la perspectiva del originador de la solicitud de registro, c. Registro (s); esto parece una interacción simple: a saber, el solicitante pidió a c que registrara a un estudiante, y ¡también lo hizo! Todos los detalles detrás de escena de lo que c tuvo que hacer para lograr esto están ocultos del solicitante (consulte la Figura 4-3).



**Obtención de identificadores de objetos**

La única forma en que un objeto A puede pasar un mensaje a un objeto B es si A tiene acceso a una referencia a / manejar en B. Esto puede suceder de varias formas diferentes.

• El objeto A podría mantener una referencia a B como uno de los atributos de A. Por ejemplo, aquí está el ejemplo del Capítulo 3 de un objeto Student que tiene una referencia de profesor como atributo:

estudiante de clase pública {

// Atributos.

Nombre de cadena;

Profesor consejero de la facultad;

// etc.

(De nuevo, aprenderá a informar a un objeto Student de qué objeto Professor debe servir como su consejero de la facultad un poco más adelante en el capítulo).

A modo de analogía, esto es como una persona A que registra "permanentemente" el número de teléfono de la persona B en su o su libreta de direcciones para que A pueda buscar y llamar a B siempre que A necesite interactuar con B.

• Al objeto A se le puede entregar una referencia a B como argumento de uno de los métodos de A. Esto es cómo el objeto de curso c obtuvo acceso a los objetos de estudiante en el mensaje anterior pasando ejemplo, cuando se llamó al método de registro de c:

c. registro (s);

Esto es análogo a la persona A que recibe una hoja de papel con el número de teléfono de la persona B, de modo que A puede llamar a B.

• Una referencia al objeto B puede estar "disponible globalmente" para toda la aplicación, como que todos los demás objetos pueden acceder a él. Analizaremos las técnicas para hacerlo más adelante en el libro, y empleará tales técnicas en la construcción del SRS.

Esto es análogo a anunciar el número de teléfono de la persona B en una valla publicitaria para que cualquiera pueda llamar.

• El objeto A puede tener que solicitar explícitamente un identificador / referencia a B llamando a un método en algún tercer objeto C. Dado que esta es potencialmente la forma más compleja para que A obtenga un identificador en B, lo ilustraremos con un ejemplo.

Esto es análogo a que la persona A tenga que llamar a la persona C para pedirle a C el número de teléfono de la persona B.

Figura 4-4. Un ejemplo más complejo de transmisión de mensajes que involucra tres objetos Volviendo a la interacción de ejemplo entre el objeto de curso cy los objetos de estudiante de Hace unas páginas, compliquemos un poco la interacción.

• Primero, presentaremos un tercer objeto: un objeto Transcripción t, que representa un registro de todos cursos tomados por Student object s.

• Además, asumiremos que Student s mantiene un control sobre Transcript t como uno de los atributos de s (específicamente, el atributo de transcripción) y, a la inversa, esa Transcripción t mantiene un control sobre su "propietario", Student s, como uno de los atributos de t:

estudiante de clase pública {

// Atributos.

Transcripción de la transcripción;

// etc.

}

Transcripción de la clase pública {

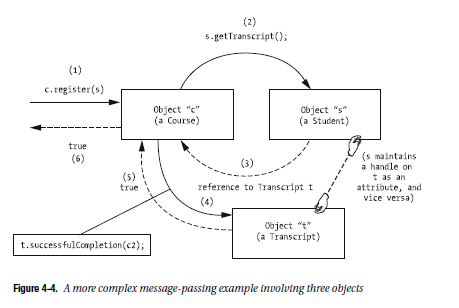
// Atributos.

Propietario del estudiante;

// etc.

}

La Figura 4-4 refleja este intercambio de mensajes más elaborado entre el Curso c, los Estudiantes, y Transcripción t; cada paso numerado del diagrama se narra en el texto que sigue. De nuevo, las flechas de línea continua representan los mensajes que se pasan / los métodos que se invocan; flechas de línea discontinua representan valores devueltos por métodos.



(Consulte la Figura 4-4 cuando lea los pasos del 1 al 6.)

1. En esta interacción mejorada de objetos, el primer paso es exactamente como se describió anteriormente: a saber, un objeto del curso c recibe el mensaje c. registro (s); donde s representa un objeto Student.

2. Ahora, en lugar de que el Curso c envíe el mensaje "Completado con éxito (c2)" a los Estudiantes como antes, dondec2 representa un curso prerrequisito, el objeto del curso c en su lugar envía el mensaje s.getTranscript (); al Estudiante, porque c quiere comprobar la transcripción de s de primera mano. Este mensaje corresponde Transcripción getTranscript ()Tenga en cuenta que este método está definido para devolver una referencia de objeto Transcript, específicamente, un identificador del objeto Transcript que pertenece a este estudiante.

3. Debido a que Student s mantiene un identificador en su objeto Transcript como atributo, es un chasquido para que s responda a este mensaje pasando un identificador en t al objeto Course c.

4. Ahora que el curso c tiene su propio control temporal en la transcripción t, el objeto c puede hablar directamente a t. El objeto c procede a preguntar t si t tiene algún registro del prerrequisito de c el curso c2 ha sido completado con éxito por los estudiantes pasando el mensaje

t.successfulCompletion (c2);

Esto implica que hay un método definido para la clase Transcript con el encabezado

boolean SuccessCompletion (Curso c)

5. El objeto de transcripción t responde con el valor verdadero al Curso c, lo que indica que Student s ha completado con éxito el curso de prerrequisito en cuestión. (Tenga en cuenta que El estudiante s no se da cuenta de que c le está hablando a t; s sabe que c le pidió que devolviera un identificador

t en un mensaje anterior, pero s no tiene ninguna idea de por qué c pidió el identificador).

Esto no es diferente a la situación del mundo real en la que la persona A le pide a la persona C el número de teléfono de la persona B,sin decirle a C por qué quieren llamar a B.

6. Satisfecho de que el estudiante ha cumplido con sus requisitos previos, el objeto del curso c termina el trabajo de registrar al estudiante (detalles internos omitidos por ahora) y confirma el registro respondiendo con un valor de verdadero al autor del registro solicitud que surgió por primera vez en el paso 1. Ahora que c ha terminado con esta transacción, descarta su mango (temporal) en t.

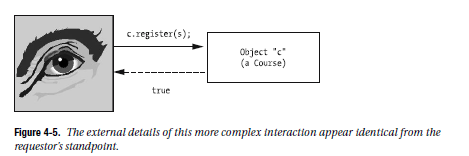


Figura 4-5. Los detalles externos de esta interacción más compleja parecen idénticos desde elpunto de vista del solicitante.

Tenga en cuenta que, desde la perspectiva de quien envió el mensaje originalmc. registro (s); para el Curso c, esta interacción más complicada parece idéntica a la interacción anterior, más simple, como se muestra en la Figura 4-5. Todo lo que sabe el remitente del mensaje original es que el Curso c eventualmente respondió con un valor de verdadero a la solicitud.

**Objetos como clientes y proveedores**

En el ejemplo anterior de transmisión de mensajes entre un objeto de curso y un objeto de alumno, podemos considerar que el objeto de curso c es un cliente de los objetos de estudiante s, porque c solicita que s realizar uno de sus métodos, a saber, getTranscript, como un servicio para c. Esto es idéntico a el concepto del mundo real de usted, como cliente, solicitando los servicios de un contador o un abogado o arquitecto. De manera similar, c es un cliente de Transcript t cuando c le pide a t que realice su método de ejecución exitosa. Por lo tanto, nos referimos al código que invoca un método en un objeto X como código de cliente relativo a X porque dicho código se beneficia de los servicios prestados por X.

Veamos algunos ejemplos de código de cliente correspondientes al ejemplo de transmisión de mensajes. que involucran un objeto de curso, estudiante y transcripción de algunas páginas atrás. Este primer código ejemplo, tomado del método principal de una aplicación, instancia dos objetos: Course cy

Student s — e invoca un método en uno de ellos, que los hace "hablar":

public class MyApp {

public static void main (String [] args) {

Curso c = nuevo curso ();

Estudiante s = nuevo Estudiante ();

// detalles omitidos ...

// Invocar un método en el objeto Course c.

// (Esto está etiquetado como mensaje (1) en la figura anterior; el

// valor, etiquetado como (6) en esa figura, se captura en booleano

// variable "éxito".)

éxito booleano = c.register (s);

// etc.

}

}

En este ejemplo, se considera que el cuerpo del método principal es el código del cliente en relación con el objeto Course c porque el método principal llama a c para realizar su método de registro como un servicio.

Veamos ahora el código que implementa el cuerpo del método de registro, dentro del Clase del curso:

Curso de clase pública {

// Detalles del atributo omitidos ...

registro booleano público (Estudiantes) {

resultado booleano = falso;

// Solicite un identificador del objeto Transcript del estudiante.

// (Esto está etiquetado como mensaje (2) en la figura anterior).

Transcripción t = s.getTranscript ();

// (El valor de retorno de este método está etiquetado como (3) en

// la figura anterior.)

// Ahora, solicite un servicio en ese objeto Transcript.

// (Suponga que c2 es un control de algún curso prerrequisito ...)

// (Esto está etiquetado como mensaje (4) en la figura anterior).

if (t.successfulCompletion (c2)) {

// (Este próximo valor de retorno está etiquetado como (5) en la figura anterior).

resultado = verdadero;

}

else {

resultado = falso;

}

resultado de retorno;

}

// etc.

Vemos que el cuerpo del método de registro de la clase del curso se considera relativo al código del cliente tanto al objeto Student como al objeto Transcript t porque este código llama s sobre s y t para cada uno realizar un servicio: s.getTranscript () y t.successfulCompletion (c2).

Siempre que un objeto A es un cliente del objeto B, el objeto B, a su vez, puede considerarse un proveedor a A. Tenga en cuenta que los roles de cliente y proveedor no son absolutos entre dos objetos; tales roles sólo son relevantes durante la duración de un evento de transmisión de mensajes en particular. Si te pido que me pases el pan, yo soy su cliente y usted es mi proveedor; y si un momento despues me pides que pase usted la mantequilla, entonces usted es mi cliente y yo soy su proveedor.

La noción de objetos como clientes y proveedores se analiza con más detalle en Construcción de software orientado a objetos por Bertrand Meyer (Prentice Hall, 2000).

**Ocultar información / Accesibilidad**

Así como hemos estado usando la notación de puntos para formular mensajes a objetos, también podemos usar puntos notación para referirse a los atributos de un objeto. Por ejemplo, si declaramos que una variable de referencia x es de tipo Student, podemos hacer referencia a cualquiera de los atributos de Student x desde el código de cliente mediante el siguiente notación: x.attribute\_name donde el punto se usa para calificar el nombre del atributo de interés con el nombre del variable de referencia que representa el objeto al que pertenece: x.name, x.gpa, etc. A continuación, se muestran algunos ejemplos adicionales:

// Crea una instancia de dos objetos.

Estudiante x = nuevo Estudiante ();

Estudiante y = nuevo Estudiante ();

// Utilice la notación de puntos para acceder a los atributos como variables.

// Asignar el nombre del alumno x ...

x.name = "John Smith";

// ... y el nombre del alumno y.

y.name = "Joe Blow";

// Compara las edades de los dos estudiantes.

if (x.age == y.age) {...}

Sin embargo, el hecho de que podamos acceder a los atributos de esta manera no significa que debamos hacerlo.

Hay muchas razones por las que queremos restringir el acceso a los datos de un objeto para otorgar la objeto de control completo sobre cuándo y cómo se alteran sus datos, y varios mecanismos para cómo podemos obtener la ayuda del compilador de Java para hacer cumplir tales restricciones.

En la práctica, los objetos a menudo restringen el acceso a algunas de sus características (atributos o métodos). Esta restricción se conoce como ocultación de información. En una aplicación orientada a objetos bien diseñada, una clase normalmente publica lo que pueden hacer sus objetos, es decir, los servicios que los objetos son capaces de proporcionar, como se declara a través de los encabezados del método de la clase, pero oculta los detalles internos tanto de cómo realizan estos servicios, así como de los datos (atributos) que mantienen internamente para apoyar estos servicios.

A modo de analogía, piense en un anuncio de las páginas amarillas de una tintorería. Tal anuncio promover los servicios que brinda la tintorería, es decir, lo que pueden hacer por usted: “Limpiamos ropa formal ”,“ Nos especializamos en limpiar alfombras de área ”, etc. Sin embargo, el anuncio normalmente no divulgar los detalles de cómo hacen la limpieza, por ejemplo, qué productos químicos o equipos específicos que utilizan, porque usted, el cliente potencial, no necesita conocer esos detalles para poder Determine si una tintorería en particular puede brindarle los servicios que necesita. Usamos el término accesibilidad para referirnos a si una característica particular de un objeto puede ser accedido fuera de la clase en la que se declara, es decir, si es accesible desde código de cliente mediante notación de puntos. La accesibilidad de una característica se establece colocando un acceso palabra clave modificadora al comienzo de su declaración:

public class MyClass {

// Atributos.

modificador de acceso int x;

// etc.

// Métodos.

modificador de acceso void foo () {...}

// etc.

}

Java define varios modificadores de acceso diferentes. Exploremos las implicaciones de usar el dos modificadores de acceso principales: privado y público.

Hay un tercer modificador de acceso, protegido, que pospondremos discutir hasta el Capítulo 5.

**Accesibilidad pública**

Cuando se declara que una función tiene accesibilidad pública, se puede acceder a ella libremente desde el código de cliente usando notación de puntos. Por ejemplo, si declaramos el atributo de nombre de la clase Student como ser de acceso público colocando la palabra clave public justo antes del tipo de atributo en la declaración

estudiante de clase pública {

nombre de cadena pública;

// etc.

otorgamos permiso al código de cliente para acceder directamente al atributo de nombre de un objeto Student mediante notación de puntos; es decir, sería perfectamente aceptable escribir el código de cliente de la siguiente manera:

public class MyProgram {

public static void main (String [] args) {

Estudiante x = nuevo Estudiante ();

// Debido a que el nombre es un atributo público de la clase Student, podemos acceder

// mediante notación de puntos del código del cliente.

x.name = "Fred Schnurd"; // asigna un valor al atributo de nombre de x

// o:

System.out.println (x.name); // recupera el valor del atributo de nombre de x

// etc.

}

}

De manera similar, si declaramos que el método isHonorsStudent de Student tiene accesibilidad, que hacemos agregando la palabra clave public al comienzo del encabezado del método declaración

estudiante de clase pública {

// Detalles de atributos omitidos en este ejemplo.

// Métodos.

public boolean isHonorsStudent () {...}

// etc.

}

otorgamos permiso al código de cliente para invocar el método isHonorsStudent en un objeto Student mediante notación de puntos; es decir, sería perfectamente aceptable escribir el código de cliente de la siguiente manera:

public class MyProgram {

public static void main (String [] args) {

Estudiante x = nuevo Estudiante ();

// Debido a que isHonorsStudent es un método público, podemos acceder a él

// mediante notación de puntos desde el código del cliente.

if (x.isHonorsStudent ()) {...}

// etc.

**Accesibilidad privada**

Cuando se declara que una función tiene accesibilidad privada, por otro lado, no es accesible fuera de la clase en la que se declara, es decir, no podemos usar la notación de puntos para acceder a tales una característica del código de cliente. Por ejemplo, si declaramos el atributo ssn del Student clase para tener accesibilidad privada

estudiante de clase pública {

nombre de cadena pública;

Private String ssn;

// etc.

entonces no se nos permite acceder a SSN directamente a través de la notación de puntos desde el código del cliente. En el siguiente ejemplo de código, se produciría un error del compilador en la línea que está en negrita:

public class MyProgram {

public static void main (String [] args) {

Estudiante x = nuevo Estudiante ();

// ¡No permitido desde el código del cliente! ssn es privado para el

// Clase de estudiante, por lo que esto no se compilará.

x.ssn = "123-45-6789";

// etc.

El mensaje de error resultante sería ssn tiene acceso privado en Student

Lo mismo ocurre con los métodos que se declaran privados, es decir, tales métodos no pueden ser invocado desde el código del cliente. Por ejemplo, si declaramos el método printInfo de Student como privado

estudiante de clase pública {

// Detalles de atributo omitidos en este ejemplo.

// Métodos.

public boolean isHonorsStudent () {...}

printInfo vacío privado () {...}

// etc.

}

entonces no sería posible invocar el método printInfo en un objeto Student desde dentro codigo del cliente. En el siguiente fragmento de código, se produciría un error del compilador en la línea en negrita:

public class MyProgram {

public static void main (String [] args) {

Estudiante x = nuevo Estudiante ();

// Debido a que printInfo () es un método privado, no podemos acceder a él

// mediante notación de puntos desde el código del cliente; esto no se compilará:

x.printInfo ();

// etc.

El mensaje de error resultante sería

printInfo () tiene acceso privado en Student

**Servicios de publicidad**

Resulta que los métodos de una clase generalmente se declaran públicos porque un objeto (clase) necesita publicitar sus servicios (como en la analogía del anuncio de las páginas amarillas) para que el código del cliente puede solicitar estos servicios. Por el contrario, la mayoría de los atributos suelen declararse privados (y efectivamente "oculto"), de modo que un objeto pueda mantener el control final sobre sus datos. Miraremos varios ejemplos detallados más adelante en este capítulo de cómo un objeto lo hace.

Aunque no se declara explícitamente como tal, el código interno que implementa cada método (es decir, el cuerpo del método) también es, en cierto sentido, implícitamente privado. Cuando un objeto cliente A pregunta a otro objeto B para realizar uno de sus métodos, A no necesita conocer los detalles detrás de escena de cómo B está haciendo lo que está haciendo; El objeto A necesita simplemente confiar en que el objeto B realizará el Servicio "anunciado". Esto se representa conceptualmente en la Figura 4-6, donde esos aspectos de una clase / Los objetos considerados privados se representan como aislados del código del cliente por una impenetrable pared de ladrillo.

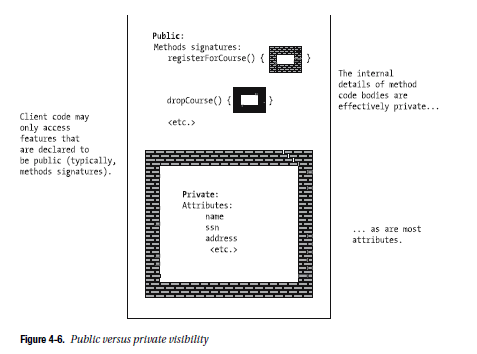


Figura 4-6. Visibilidad pública versus privada

**Encabezados de método, revisados**

Modifiquemos la definición de un encabezado de método de un poco antes en este capítulo. Un encabezado de método en realidad consiste en lo siguiente:

• Modificador de acceso de un método

• El tipo de retorno de un método, es decir, el tipo de datos de la información que se va a devuelto por el objeto B al objeto A, si lo hay, cuando el método ha terminado de ejecutarse

• El nombre de un método

• Una lista opcional de parámetros formales separados por comas (especificando sus tipos y nombres) para pasar al método, entre paréntesis Como ejemplo, aquí hay un encabezado de método típico que podríamos definir para la clase Student,

con el modificador de acceso incluido:

public boolean registerForCourse (String courseID, int secNo)

acceso tipo de retorno nombre del método lista de formales separados por comas

parámetros de modificador, entre paréntesis (los paréntesis pueden dejarse en blanco)

**Acceder a las funciones de una clase desde sus propios métodos**

Tenga en cuenta que podemos acceder a todas las funciones de una clase determinada, independientemente de su accesibilidad, desde dentro de cualquiera de los cuerpos de métodos propios de esa clase; es decir, las designaciones públicas / privadas solo afectan acceso a una característica desde fuera de la clase misma (es decir, desde el código del cliente).

Estudiemos el siguiente ejemplo para ver cómo se puede acceder a una función de una clase desde dentro de otro:

estudiante de clase pública {

// Algunos atributos privados.

nombre de cadena privada;

Private String ssn;

Préstamos privados dobles totales;

matrícula doble privada

// Se proporcionarían métodos get / set para todos estos atributos; // detalles omitidos ...

public void printStudentInfo () {

// Accediendo a los atributos de la clase Student.

System.out.println ("Nombre:" + nombre);

System.out.println ("ID de estudiante:" + ssn);

// etc.

}

public boolean allBillsPaid () {

respuesta booleana = falso;

// Accediendo a otro método de la clase Student.

doble cantidad = moneyOwed ();

si (amt == 0.0) {

respuesta = verdadero;

}

else {

respuesta = falso;

}

devolver respuesta;

}

private double moneyOwed () {

// Accediendo a los atributos de la clase Student.

devolver totalLoans + tuitionOwed;

}

}

Lo primero que observamos es que no necesitamos utilizar la notación de puntos para acceder a ninguna de las funciones de la clase Student desde dentro de los métodos Student. El compilador lo entiende automáticamente que una clase está accediendo a una de sus propias características cuando un nombre simple, es decir, un nombre sin se utiliza un prefijo de notación de puntos, también conocido como nombre no calificado, por ejemplo:

public void printStudentInfo () {

// Aquí, estamos accediendo al atributo "nombre" sin notación de puntos.

System.out.println ("Nombre:" + nombre);

// etc.

}

y

public boolean allBillsPaid () {

respuesta booleana = falso;

// Aquí, estamos accediendo al método "moneyOwed" sin notación de puntos.

doble cantidad = moneyOwed ();

// etc.

}

Dicho esto, la palabra clave de Java this se puede utilizar en forma de notación de puntos:

this.featureName: dentro de cualquiera de los métodos de una clase para enfatizar el hecho de que estamos accediendo otra característica de esta misma clase. He reescrito el ejemplo de Student de antes para tomar ventaja de esta palabra clave:

estudiante de clase pública {

// Algunos atributos privados.

nombre de cadena privada;

Private String ssn;

Préstamos privados dobles totales;

matrícula doble privada

// Se proporcionarían métodos get / set para todos estos atributos;

// detalles omitidos ...

public void printStudentInfo () {

// Hemos agregado el prefijo "esto".

System.out.println ("Nombre:" + este.nombre);

System.out.println ("ID de estudiante:" + this.ssn);

// etc.

}

public boolean allBillsPaid () {

respuesta booleana = falso;

// Hemos agregado el prefijo "esto".

doble importe = this.moneyOwed ();

si (amt == 0.0) {

respuesta = verdadero;

}

else {

respuesta = falso;

}

devolver respuesta;

}

private double moneyOwed () {

// Hemos agregado el prefijo "esto".

devolver this.totalLoans + this.tuitionOwed;

}

}

Cualquiera de los dos enfoques: prefije las referencias de características internas con esto. u omitiendo tal calificación prefijo — es aceptable; La práctica común es renunciar al uso de this. prefijo excepto cuando necesario para eliminar la ambigüedad de un parámetro de método de un atributo con un nombre similar. Es decir, es permitido declarar un parámetro de método con el mismo nombre que un atributo, como lo ilustra el siguiente código:

estudiante de clase pública {

principal de cadena privada;

// Otros atributos omitidos.

// Tenga en cuenta que hemos utilizado "major" como el nombre de un parámetro

// al siguiente método - esto duplica el nombre de

// el atributo "mayor" anterior. Sin embargo, esto está bien si

// usamos "esto". dentro del cuerpo del método a continuación para eliminar la ambigüedad

// los dos.

public void updateMajor (String major) {

// En la siguiente línea de código, "this.major" en el lado izquierdo

// de la sentencia de asignación se refiere al PARÁMETRO

// llamado "mayor", mientras que "mayor" en el lado derecho de

// la sentencia de asignación se refiere al PARÁMETRO

// llamado "mayor".

this.major = mayor;

}

// etc.

}

Por supuesto, podríamos evitar tener que usar esto. como prefijo simplemente eligiendo una alternativa nombre para nuestro parámetro de método:

estudiante de clase pública {

principal de cadena privada;

// Otros atributos omitidos.

public void updateMajor (String m) {

// ¡Sin ambigüedad!

mayor = m;

}

// etc.

}

Es importante evitar dar accidentalmente nombres de parámetros / variables locales que dupliquen los nombres de atributos, ya que esto puede provocar errores que son difíciles de diagnosticar. Por ejemplo, en la clase Student que sigue son tanto un atributo como una variable local denominada major. Consulte los comentarios en el ejemplo de código para una explicación de por qué esto es problemático.

estudiante de clase pública {

// Atributos.

principal de cadena privada;

public void updateMajor () {

// Inadvertidamente hemos declarado una variable local, "mayor", con

// el MISMO nombre como atributo de esta clase. ¡Esta es una mala idea!

// Tenga en cuenta que este código se compilará SIN ERROR ...

Cadena mayor = nulo;

// Más adelante en el método:

// CREEMOS que estamos actualizando el valor de ATTRIBUTE "major" a continuación,

// pero en su lugar estamos actualizando LOCAL VARIABLE "major", que

// sale del alcance tan pronto como finaliza este método;

// mientras tanto, ¡el valor de ATTRIBUTE "major" no cambia!

major = major.toUppercase ();

// etc.

}

}

Veremos otros usos para esta palabra clave, que incluyen la reutilización de código y la autorreferencia de objetos, más adelante en el libro.

**Acceso a funciones privadas desde el código de cliente**

Si no se puede acceder a las funciones privadas fuera de los métodos propios de un objeto, ¿cómo funciona el código del cliente? alguna vez los manipulas? ¡A través de funciones públicas, por supuesto!

La buena práctica de programación de OO exige proporcionar métodos de acceso público mediante los cuales los clientes de un objeto pueden manipular eficazmenteular atributos privados seleccionados para leer o modificar sus valores. ¿Por qué es esto? Para que podamos empoderar a un objeto para que tenga la "última palabra" sobre si no es válido lo que el código de cliente intenta hacer con sus atributos. Es decir, queremos que un objeto sea involucrado en determinar si alguna de las reglas de negocio definidas por su clase está siendo violado. Antes de ver ejemplos específicos que ilustran por qué esto es tan importante, primero discutir la mecánica de cómo declaramos los métodos de acceso.

**Declaración de métodos de acceso**

El siguiente código, extraído de la clase Student, ilustra el descriptor de acceso convencional métodos, conocidos informalmente como métodos "get" y "set", que podríamos escribir para lectura / escribir el valor de dos atributos privados de la clase del estudiante llamados nombre y asesor de la facultad, respectivamente:

estudiante de clase pública {

// Los atributos normalmente se declaran privados.

nombre de cadena privada;

profesor privado consejero de la facultad;

// otros atributos omitidos de este ejemplo ...

// Proporcionar métodos de acceso público para leer / modificar

// atributos privados del código del cliente.

// El código del cliente utilizará este método para leer ("obtener") el valor del

// atributo "nombre" de un objeto Student en particular.

public String getName () {

nombre de retorno;

}

// El código del cliente utilizará este método para modificar ("establecer") el valor del

// atributo "nombre" de un objeto Student en particular.

public void setName (String newName) {

nombre = nuevoNombre;

}

// El código del cliente utilizará este método para leer ("obtener") el valor del

// atributo facultyAdvisor de un objeto Student en particular.

profesor público getFacultyAdvisor () {

volver facultyAdvisor;

}

// El código del cliente utilizará este método para modificar ("establecer") el valor del

// atributo facultyAdvisor de un objeto Student en particular.

public void setFacultyAdvisor (Professor p) {

consejero de la facultad = p;

}

// etc.

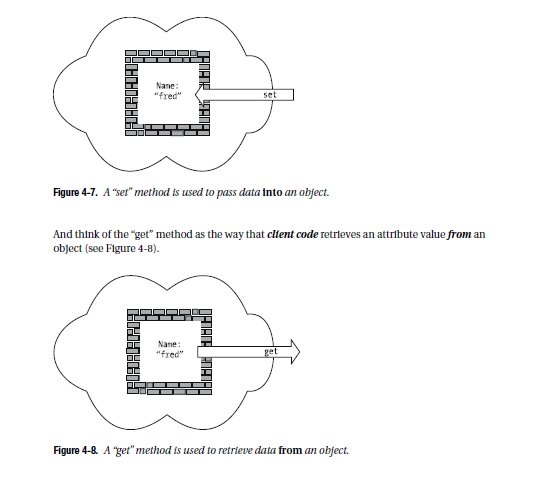
}

La nomenclatura "obtener" y "establecer" se establece desde el punto de vista del código de cliente: piense en un "conjunto" método como la forma en que el código del cliente introduce un valor en el atributo de un objeto (consulte la Figura 4-7).

Figura 4-7. Se utiliza un método de "conjunto" para pasar datos a un objeto.

Figura 4-8. Se utiliza un método "get" para recuperar datos de un objeto.

Y piense en el método "get" como la forma en que el código de cliente recupera un valor de atributo de un objeto (vea la Figura 4-8).



**Encabezados de método recomendados "Obtener" / "Establecer"**

Para una declaración de atributo del formulario accesibilidad \* tipo de atributo atributoName; \* típicamente privado por ejemplo,

private String majorField;

las reglas para formular encabezados de métodos de acceso convencionales son las siguientes. Para un método "get", la fórmula es la siguiente:

getAttributeName de tipo de atributo público ()

por ejemplo,

public String getMajorField ()

• El nombre del método se formula escribiendo en mayúscula la primera letra del atributo nombre en cuestión (por ejemplo, majorField) y pegar "get" al frente (por ejemplo, getMajorField).

• Tenga en cuenta que normalmente no pasamos ningún argumento a un método "get", porque todo lo que queremos un objeto a hacer es devolvernos el valor de uno de sus atributos; normalmente no necesitamos decirle al objeto algo especial para que sepa cómo hacer esto.

• Además, debido a que esperamos que un objeto devuelva el valor de un atributo específico, el tipo de retorno del método "get" debe coincidir con el tipo del atributo de interés. Si estamos "obteniendo" el valor de un atributo int, luego el tipo de retorno del correspondiente

El método "get" debe ser int; si estamos "obteniendo" el valor de un atributo String, entonces el retorno el tipo del método "get" correspondiente debe ser String; Etcétera.

• A continuación, se muestra un método típico "get" en su totalidad, que se muestra en el contexto de la clase del estudiante:

estudiante de clase pública {

private String majorField;

// Otros atributos omitidos en este ejemplo.

public String getMajorField {

// Devuelve el valor del atributo majorField.

return majorField;

}

// etc.

}

Para un método de "conjunto", la fórmula es la siguiente:

public void setAttributeName (attributeType parameterName)

por ejemplo,

public void setMajorField (String mayor)

• El nombre del método se formula escribiendo en mayúscula la primera letra del atributo nombre en cuestión (por ejemplo, majorField) y pegar "set" al frente (por ejemplo, setMajorField).

• En el caso de un método "set", debemos pasar el valor que queremos que use el objeto

al establecer su valor de atributo correspondiente, y el tipo de valor que estamos pasando

in debe coincidir con el tipo de atributo que se está estableciendo. Si estamos "configurando" el valor de un int atributo, entonces el argumento que se pasa al método "set" correspondiente debe ser un int; si estamos "configurando" el valor de un atributo String, entonces el argumento que se pasa en el método "set" correspondiente debe ser una cadena; Etcétera.

• Dado que normalmente se espera que los métodos simples "establecidos" realicen su misión en silencio, sin devolviendo un valor al cliente, normalmente declaramos métodos "establecidos" para tener un retorno tipo de vacío.

• Aquí estáun método típico de "conjunto" en su totalidad, que se muestra en el contexto de la clase Student:

estudiante de clase pública {

private String majorField;

// Otros atributos omitidos en este ejemplo.

public String getMajorField {

// Devuelve el valor del atributo majorField.

return majorField;

}

public void setMajorField (String major) {

// Asignar el valor pasado como argumento como el nuevo valor de

// el atributo majorField.

majorField = major;

}

}

Hay una excepción a la convención de nomenclatura del método "get": cuando un atributo es de tipo booleano, se recomienda nombrar el método "get" comenzando con el verbo es en lugar de con obtener. El método "set" para un atributo booleano seguiría la convención de nomenclatura estándar, sin embargo, por ejemplo:

estudiante de clase pública {

honorsStudent booleano privado;

// otros atributos omitidos de este ejemplo ...

// Obtener método. Para un booleano, el nombre del método comienza con "es" frente a "obtener".

public boolean isHonorsStudent () {

return honorsStudent;

}

// Establecer método.

public void setHonorsStudent (booleano x) {

honoresStudent = x;

}

// etc.

}

Todos los cuerpos de los métodos "get" / "set" que hemos visto hasta ahora son simples "frases ingeniosas": estamos ya sea devolviendo el valor del atributo de interés con una declaración de devolución simple en un "get" método, o copiando el valor del argumento pasado al atributo interno en un método "set" para almacenarlo. Esto no implica que todos los métodos "get" / "set" necesiten ser así de simples; de hecho, hay infinitas posibilidades para lo que realmente se codifica en los métodos de acceso, porque, como discutimos Antes, los métodos deben implementar reglas de negocio, no solo sobre cómo se comporta un objeto, sino también qué estados válidos pueden asumir sus datos.

Como ejemplo simple, digamos que siempre queremos almacenar el nombre de un estudiante como "Steve Barker ”en el formato“ S. BARKER ”, donde abreviamos el nombre a una sola letra y representar el nombre completo en mayúsculas. Por lo tanto, es posible que deseemos escribir el método setName de la clase de estudiantes de la siguiente manera:

public void setName (String newName) {

// Primero, formatee el newName, según sea necesario ...

// Pseudocódigo.

if (newName contiene el nombre completo) {

newName = newName con el nombre convertido a un solo carácter

seguido de un punto;

}

// A continuación, convierta newName a todo en mayúsculas.

// Pseudocódigo.

newName = versión en mayúsculas de newName;

// Solo entonces actualizamos el atributo de nombre con el valor (modificado).

nombre = nuevoNombre;

}

**La "persistencia" de los valores de los atributos**

Debido a que no lo he dicho explícitamente antes, y porque puede que no sea obvio para todos, quisiera llamar la atención ahora sobre el hecho de que los valores de los atributos de un objeto persisten mientras el objeto persiste en la memoria. Es decir, una vez que instanciamos un objeto Student en nuestra aplicación

Estudiante s = nuevo Estudiante ();

luego cualquier valor que asignemos a los atributos de s

s.setName ("Mel");

persistirá hasta el momento en que el valor se cambie explícitamente

// Cambio de nombre de Student s.

s.setName ("Klemmie");

o el objeto como un todo es basura recolectada por la Máquina Virtual Java (JVM), un proceso que discutimos en el capítulo 3. Entonces, para volver a nuestra analogía de objetos como globos de helio de Capítulo 3, siempre que el "globo de helio" que representa al estudiante permanezca "inflado", siempre que pregunte s por su nombre, "recordará" cualquier valor que hayamos asignado por última vez a su atributo de nombre.

**Uso de métodos de acceso desde el código de cliente**

Ya sabemos cómo usar la notación de puntos para invocar métodos en objetos desde el código del cliente, y así que haremos lo mismo cuando invocamos métodos de acceso en referencias de objetos:

Estudiante s = nuevo Estudiante ();

// Modifica ("establece") el valor del atributo.

s.setName ("Joe");

// Leer ("obtener") el valor del atributo.

System.out.println ("Nombre:" + s.getName ());

Prometí anteriormente en este capítulo discutir cómo se puede informar a un estudiante determinado sobre qué profesor particular es su facultad consejero; ahora que conoce los métodos "establecidos", hacerlo es ¡Una toma! Suponiendo que (un) asesor de profesores es un atributo de la clase de estudiantes declarado type Professor, y (b) hemos escrito un método "establecido" para este atributo con el encabezado "estándar"

public void setFacultyAdvisor (Professor p), aquí está el código de cliente para "familiarizar" a los estudiantes con sus asesores:

Estudiante s1 = nuevo Estudiante ();

Estudiante s2 = nuevo Estudiante ();

Estudiante s3 = nuevo Estudiante ();

Estudiante s4 = nuevo Estudiante ();

// etc.

Profesor p1 = nuevo profesor ();

Profesor p2 = nuevo profesor ();

// etc.

// Detalles omitidos ...

s1.setFacultyAdvisor (p1);

s2.setfacultyAdvisor (p1);

s3.setFacultyAdvisor (p2);

s4.setFacultyAdvisor (p2);

// etc.

**El poder de la encapsulación más información Ocultación**

Aprendió anteriormente que la encapsulación es el mecanismo que agrupa el estado (atributo valores) y comportamiento (métodos) de un objeto. Ahora que ha obtenido información sobre el público / accesibilidad privada, la encapsulación justifica una discusión más profunda.

Es útil pensar en una o objeto como una "fortaleza" que "guarda" sus datos, es decir, los valores de todos sus atributos. En lugar de intentar marchar directamente a través de los muros de una fortaleza, que típicamente resulta en muerte y destrucción (!), idealmente nos acercaríamos al guardia en la puerta pedir permiso para entrar. En términos generales, lo mismo ocurre con los objetos: no podemos acceder a los valores de los atributos declarados de forma privada de un objeto sin el permiso de un objeto y conocimiento, es decir, sin utilizar uno de los métodos de acceso público de un objeto para acceder al valor del atributo.

Suponga que acaba de conocer a alguien por primera vez y desea saber su nombre. Uno La manera de determinar su nombre sería meter la mano en el bolsillo, sacar su billetera y mirar su licencia de conducir, esencialmente, acceder a sus valores de atributos privados sin su permiso.

La forma más socialmente aceptable sería simplemente preguntarle su nombre, similar a usar su getName, y permitirle responder en consecuencia. Puede responder con su formal un nombre, un apodo o un alias, o puede decir: "¡No es asunto tuyo!", pero lo importante El punto es que le está dando a la persona (objeto) control sobre su respuesta.

Al restringir el acceso a los atributos privados de un objeto a través de accesos públicos, obtenemos tres importantes beneficios:

• Evitar el acceso no autorizado a datos encapsulados

• Ayudando a garantizar la integridad de los datos

• Limitar los "efectos dominó" que de otro modo pueden ocurrir en una aplicación cuando el

los detalles de implementación privada de una clase deben cambiar Analicemos cada uno de estos beneficios en detalle.

**Prevención del acceso no autorizado a datos encapsulados**

Parte de la información que un objeto Student mantiene sobre sí mismo, por ejemplo, la identificación del estudiante número: puede ser muy confidencial. Un objeto Student puede optar por aprobar selectivamente junto con esta información cuando sea necesario, por ejemplo, al registrarse en un curso, pero No deseo entregar esta información a ningún objeto que por casualidad la solicite.

Simplemente haciendo que el atributo sea privado y omitiendo intencionalmente un método público "get" con el que solicitar el valor del atributo, no habría forma de que otro objeto solicite el Número de identificación del objeto de estudiante.

**Ayudando a garantizar la integridad de los datos**

Como se mencionó anteriormente, uno de los argumentos en contra de declarar atributos públicos es que el objeto pierde el control sobre sus datos, ya que, como vimos anteriormente, el valor de un atributo público se puede cambiar mediante el código del cliente sin tener en cuenta las reglas comerciales que la clase del objeto desee imponer.

Por otro lado, cuando se utiliza un método de acceso para cambiar el valor de un atributo privado, La verificación de valor se puede integrar en el método "set" para garantizar que el valor del atributo no se establezca a un valor "inadecuado".

Como ejemplo, digamos que declaramos un atributo de estudiante de la siguiente manera:

private String birthDate;

Nuestra intención es registrar las fechas de nacimiento en el formato “mm / dd / aaaa”. Al requerir que ese cliente código invocar métodos para manipular el atributo de fecha de nacimiento (en lugar de permitir el acceso público directo acceso al atributo), podemos proporcionar lógica dentro de esos métodos para validar el formato de cualquier fecha de nacimiento propuesta recientemente y rechace las que no sean válidas. Ilustraremos este concepto con declarando un método updateBirthDate para la clase Student como se muestra en el siguiente código:

estudiante de clase pública {

private String birthDate;

// otros atributos omitidos de este ejemplo ...

public boolean updateBirthDate (String newBirthDate) {

boolean newDateApproved;

// Realice las validaciones adecuadas.

// Recuerda, las cursivas representan pseudocódigo ...

if (la fecha no tiene el formato mm / dd / aaaa) {

newDateApproved = falso;

}

else if (mm no en el rango de 01 a 12) {

newDateApproved = falso;

}

de lo contrario, si (el número de día no es válido para el mes seleccionado) {

newDateApproved = falso;

}

else if (el año NO es bisiesto, pero se especificó 2/29) {

newDateApproved = falso;

}

// etc. para otras pruebas de validación.

else {

// Si hemos llegado tan lejos en el código, todo está bien con lo que

// pasado como un valor a este método, por lo que podemos seguir adelante y

// actualiza el valor del atributo birthDate con este valor.

BirthDate = newBirthDate;

// ¡Coloca nuestra bandera para indicar éxito!

newDateApproved = true;

}

return newDateApproved;

}

// etc.

}

Si se intenta pasar una fecha de nacimiento con formato incorrecto al método de código de cliente, como en s.updateBirthDate ("foo"); el cambio será rechazado y el valor del atributo birthDate de s no se modificará. De hecho, probablemente insertemos el intento de actualizar la fecha de nacimiento dentro de una declaración "si" para que podría detectar y reaccionar ante tal rechazo:

// En algún punto de la línea, la variable newDate toma un valor no válido.

String newDate = "foo";

// Más adelante en la aplicación ...

if (! (s.updateBirthDate (newDate)) {

// Pseudocódigo.

hacer lo que sea necesario si se rechaza el valor ...

}

Por otro lado, si la fecha de nacimientohabía sido declarado como un atributo público del Estudiante class, luego el compilador permitiría establecer el atributo directamente de la siguiente manera:

s.birthDate = "foo";

y, por lo tanto, sería posible corromper el valor del atributo omitiendo la comprobación de errores, basado en reglas comerciales, que un método "establecido" normalmente funcionaría para nosotros.

**Limitar los “efectos dominó” cuando cambian las funciones privadas**

A pesar de nuestros mejores intentos por evitar tales situaciones, a menudo tenemos la necesidad de volver atrás y modificar el diseño de una aplicación después de su implementación, ya sea cuando un cambio inevitable en requisitos, o si, por desgracia, descubrimos un defecto de diseño que necesita atención. Desafortunadamente, en una aplicación que no sea OO (o OO mal diseñada), esto puede abrirnos a "efectos dominó", donde decenas, cientos o miles de líneas de código en una aplicación tienen que cambiarse, volver a probarse, etc.

Uno de los ejemplos más dramáticos del impacto negativo de un cambio de diseño fue el notorio problema del año 2000. Cuando la necesidad de cambiar los formatos de fecha para acomodar unos cuatro dígitos año surgió a medida que se acercaba el año 2000, la carga de buscar a través de miles de millones de líneas de código en millones de aplicaciones en todo el mundo para encontrar todos estos casos y solucionarlos sin querer romper cualquier otra cosa, era alucinante. Mucha gente estaba convencida en ese momento de que el mundo se derretiría como resultado y, de hecho, ¡es bastante sorprendente que no fuera así!

Quizás el beneficio más dramático de la encapsulación combinado con el ocultamiento de información, por lo tanto, es que los detalles de implementación ocultos de una clase, es decir, su estructura de datos privados y / o su código de acceso (efectivamente privado), puede cambiar sin afectar la forma en que el cliente el código interactúa con los objetos que pertenecen a esa clase. Para ilustrar este principio, crearemos un ejemplo.

Supongamos que un atributo se declara en la clase Student de la siguiente manera:

int edad privada;

y que declaramos un método getAge () correspondiente de la siguiente manera:

public int getAge () {

edad de regreso;

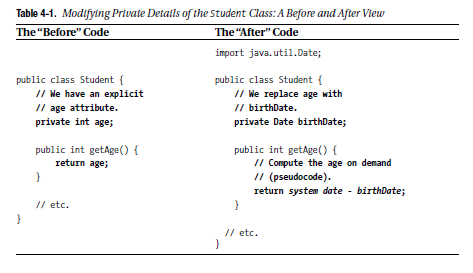
}

(Elegimos no declarar un método setAge porque decidimos que queremos que la edad sea un atributo de solo lectura). Luego procedemos a utilizar nuestra clase de Estudiantes en innumerables aplicaciones; entonces, literalmente en miles de lugares dentro del código de cliente de estas aplicaciones, escribimos declaraciones como el siguiendo, confiando en el método "get" para proporcionarnos la edad de un estudiante como un valor int, para ejemplo:

int currentAge = s.getAge ();

Unos años más tarde, decidimos modificar la estructura de datos de la clase Student para que, en lugar de mantener un atributo de edad de forma explícita, usamos el atributo de fecha de nacimiento del estudiante para calcular la edad de un estudiante cuando sea necesario. De esta manera modificamos nuestro código de clase Student como se muestra en la Tabla 4-1.

Tabla 4-1. Modificación de detalles privados de la clase del estudiante: una vista de antes y después



En la versión "después" de Student, calculamos la edad del estudiante restando su nacimiento fecha (almacenada como un valor de atributo) a partir de la fecha de hoy. Este es un ejemplo de lo que puede ser informalmente denominado pseudoatributo: para el código del cliente, la presencia de un método getAge () implica que hay un atributo con el nombre de edad, cuando en realidad puede que no lo haya.

¡La belleza es que no nos importa que los detalles privados del diseño de la clase del estudiante tengan cambiado! En todos los miles de lugares dentro del código de cliente de innumerables aplicaciones donde hemos utilizado código como

int currentAge = s.getAge ();

para recuperar la edad de un estudiante como un valor int, este código seguirá funcionando como está, sin ningún siendo necesarios cambios en el código del cliente, porque la expresión

s.getAge ()

todavía se evalúa como un valor int que representa la edad del estudiante. Por lo tanto, hemos evitado "temido" efectos dominó, y han reducido drásticamente la cantidad de esfuerzo necesario para adaptarse un cambio de diseño. Se dice que tales cambios están encapsulados o limitados al código interno de la clase de estudiantes solamente.

Por supuesto, todas las apuestas están canceladas si el desarrollador de una clase cambia una de sus funciones públicas:

la mayoría de las veces, un encabezado de método público, porque luego todo el código de cliente que pasa mensajes a objetos de este tipo que utilicen este método potencialmente tendrá que cambiar. Por ejemplo, si nosotros iban a cambiar el diseño de la clase Student para que el método getAge () ahora se declare para devolver un valor doble, como sigue:

estudiante de clase pública {

// Hemos cambiado el tipo del atributo age de int a double ...

doble edad privada;

// ... y el tipo de retorno del método getAge () en consecuencia.

public double getAge () {

edad de regreso;

}

// etc.

entonces gran parte de nuestro cliEl código ent se podría "romper" potencialmente, como en el siguiente ejemplo:

// ¡Esto ya no se compilará!

int currentAge = s.getAge ();

Este código de cliente en particular se “romperá” porque ahora tenemos una falta de coincidencia de tipos. Estamos volviendo un valor doble, pero estamos tratando de asignarlo a una variable int, que como aprendimos en el Capítulo 2

generar un error del compilador de la siguiente manera:

posible pérdida de precisión

encontrado: doble

requerido: int

Tendríamos que buscar todas las innumerables instancias en muchas aplicaciones potenciales donde llamamos al método getAge () en una referencia de estudiante, y modificamos cada una de esas líneas de código para hacer una conversión explícita de doble a int, de la siguiente manera:

// Ahora estamos usando un yeso.

int currentAge = (int) s.getAge ();

y, por lo tanto, podríamos generar un efecto dominó significativo. Pero de nuevo, este efecto dominó se debe a el hecho de que cambiamos una característica pública de nuestra clase, un encabezado de método público, para ser precisos.

Siempre que restrinjamos nuestros cambios a las funciones privadas de una clase, los efectos dominó no son un problema; cualquier código de cliente que se haya escrito previamente para usar métodos públicos de Estudiantes continuará para trabajar según lo previsto.

**Uso de métodos de acceso desde dentro de los métodos propios de una clase**

Anteriormente en el capítulo, discutimos el hecho de que una clase puede acceder directamente a su propia atributos por nombre, como en el siguiente método printStudentInfo:

estudiante de clase pública {

nombre de cadena privada;

Private String ssn;

// etc.

// Detalles omitidos.

public void printStudentInfo () {

// Estamos accediendo a nuestros propios atributos directamente.

System.out.println ("Nombre:" + este.nombre);

System.out.println ("ID de estudiante:" + this.ssn);

// etc.

}

// etc.

Sin embargo, se considera una práctica recomendada que una clase utilice sus propios métodos "get" / "set"

siempre que necesite acceder a uno de sus propios valores de atributo. Revisemos printStudentInfo

método para ilustrar esta mejor práctica:

estudiante de clase pública {

nombre de cadena privada;

Private String ssn;

// etc.

// Métodos de acceso "Variedad de jardín".

public String getName () {

nombre de retorno;

}

public void setName (String n) {

nombre = n;

}

public String getSsn () {

return ssn;

}

public void setSsn (String s) {

ssn = s;

}

public void printStudentInfo () {

// Ahora estamos usando nuestros propios métodos "get" para acceder a los nuestros

// valores de atributo.

System.out.println ("Nombre:" + this.getName ());

System.out.println ("ID de estudiante:" + this.getSsn ());

// etc.

}

// etc.

¿Por qué es importante utilizar los métodos "get" / "set" propios de una clase en lugar de acceder a los atributos? ¿directamente? Digamos que, en una fecha futura, los métodos getName y getSsn de la clase Student

se modifican de la siguiente manera:

public String getName () {

// ¡Las reglas comerciales han cambiado! Ahora queremos reformatear el nombre

// como se almacena dentro de un objeto Student antes de devolverlo a

// codigo del cliente.

// (Pseudocódigo.)

String reformattedName = nombre reformateado en el formulario

"Apellido nombre";

return reformattedName;

}

public String getSsn () {

// ¡Las reglas comerciales han cambiado! Ahora queremos reformatear el

// ssn almacenado dentro de un objeto Student para insertar guiones

// antes de devolverlo al código del cliente.

// (Pseudocódigo.)

String reformattedSsn = ssn reformateado con el formato "xxx-xx-xxxx";

return reformattedSsn;

}

Porque hemos rediseñado el método printStudentInfo para invocar this.getName () y this.

getSsn (), nos beneficiaremos automáticamente de los cambios en la lógica empresarial dentro de getName y

métodos getSsn:

// Codigo del cliente.

Estudiante s = nuevo Estudiante ();

s.setName ("Susan Yamate");

s.setSsn ("123456789");

s.printStudentInfo ();

Aquí está el resultado:

Nombre: Yamate, Susan

Identificación del estudiante: 123-45-6789

Por otro lado, si hubiéramos accedido a los atributos name y ssn directamente desde dentro del método printStudentInfo

public void printStudentInfo () {

// Estamos accediendo a atributos privados directamente por nombre en lugar de

// usando los métodos de obtención correspondientes.

System.out.println ("Nombre:" + nombre);

System.out.println ("ID de estudiante:" + ssn);

// etc.

}

no nos beneficiaríamos de cambios en la lógica empresarial:

// Codigo del cliente.

Estudiante s = nuevo Estudiante ();

s.setName ("Susan Yamate");

s.setSsn ("123456789");

s.printAllAttributes ();

Aquí está la salida (formateada incorrectamente):

Nombre: Susan Yamate

Identificación de estudiante: 123456789

Lo mismo ocurre con el uso de métodos de "conjunto" propios de una clase al actualizar el valor de un atributo desde dentro de otro método, por ejemplo, esta versión del método assignMajor de Estudiante:

estudiante de clase pública {

nombre de cadena privada;

Private String ssn;

principal de cadena privada;

profesor asesor privado;

// etc.

// Establecer / obtener métodos proporcionados; detalles omitidos.

public void assignMajor (String m, Professor p) {

// Preferido.

this.setMajor (m);

this.setAdvisor (p);

}

// etc.

se prefiere a esta versión:

estudiante de clase pública {

nombre de cadena privada;

Private String ssn;

principal de cadena privada;

profesor asesor privado;

// etc.

// Establecer / obtener métodos proporcionados; detalles omitidos.

public void assignMajor (String m, profesor p) {

// No tan deseable.

this.major = m;

this.advisor = p;

}

// etc.

porque los métodos "establecidos" de la clase para estudiantes pueden ser simples "frases ingeniosas" en la actualidad

public void setMajor (String m) {

mayor = m;

}

pero puede mejorarse para reflejar una lógica empresarial más sofisticada en algún momento en el futuro:

public void setMajor (String m) {

// Pseudocódigo.

buscar m en una base de datos para verificar que es una designación principal "aprobada" antes de actualizar el atributo principal

if (m es válido) {

mayor = m;

}

}

Por supuesto, el único lugar donde no podemos invocar los métodos "get" / "set" de una clase es dentro los métodos "get" / "set" en sí mismos. Hacerlo resultaría en un método infinitamente recursivo:

// ¡Este método es recursivo!

public void setName (String n) {

this.setName (n);

}

que se compilará correctamente, pero producirá un error de tiempo de ejecución cuando se invoca, de la siguiente manera:

// Codigo del cliente.

Estudiante s = nuevo Estudiante ();

s.setName ("Fred Schnurd");Aquí está el error:

Excepción en el hilo "principal" java.lang.StackOverflowError

en Student.setName (Student.java:8)

en Student.setName (Student.java:8)

en Student.setName (Student.java:8)

en Student.setName (Student.java:8)

en Student.setName (Student.java:8)

(¡repetido 1024 veces!)

**Excepciones a la regla pública / privada**

Aunque a menudo es el caso que

• Los atributos se declaran privados

• Los métodos se declaran públicos

• Se accede a los atributos privados a través de métodos públicos

Existen numerosas excepciones a esta regla, como se explica en las secciones siguientes.

**Excepción n. ° 1: Atributos de limpieza internos**

Una clase puede utilizar un atributo estrictamente para fines de limpieza interna. (Como el lavado de platos detergente que guardas debajo del fregadero, ¡los invitados no necesitan saberlo!) Para tales atributos, no necesitamos molestarnos en proporcionar accesos públicos.

Un ejemplo para la clase Student podría ser un atributo int countOfDsAndFs, usado para realizar un seguimiento de cuántas malas calificaciones ha recibido un estudiante para determinar si no el estudiante está en período de prueba académica. A su vez, podemos proporcionar un método de clase para estudiantes onAcademicProbation de la siguiente manera:

estudiante de clase pública {

private int countOfDsAndFs;

// otros atributos omitidos de este ejemplo ...

public boolean onAcademicProbation () {

booleano onProbation = falso;

// Si el estudiante recibió más de tres calificaciones deficientes,

// se le pondrá en período de prueba académica.

if (countOfDsAndFs> 3) {

onProbation = verdadero;

}

return onProbation;

}

// otros métodos omitidos en este ejemplo ...

}

El método onAcademicProbation utiliza el valor del atributo privado countOfDsAndFs para determinar si un estudiante está en período de prueba académica, pero ningún código de cliente necesita saber que existe un atributo como countOfDsAndFs, por lo que no hay métodos de acceso público explícitos proporcionado para este atributo. En cambio, tales atributos se establecen como un efecto secundario de realizar algunos otro método, como en el siguiente ejemplo, también tomado de la clase Student:

public void completeCourse (String courseName, int creditHours, char grade) {

// La actualización de este atributo privado se considera una

// "efecto secundario" de completar un curso.

if (grado == 'D' || grado == 'F') countOfDsAndFs ++;

// Otros detalles de procesamiento omitidos en este ejemplo ...

}

**Excepción n. ° 2: métodos de limpieza internos**

Algunos métodos también se pueden utilizar estrictamente para fines de limpieza interna, en cuyo caso estos también pueden declararse privados en lugar de públicos. (¡Un vecino no necesita saber que tenemos una criada que viene a limpiar cada dos semanas!) Un ejemplo de dicho método Student podría ser updateGpa, que vuelve a calcular el valor del atributo gpa cada vez que un estudiante completa otro curso y recibe una calificación. Lo único El momento en que este método puede necesitar ser llamado es quizás desde dentro de otro método de Student, por ejemplo, el método public completeCourse, de la siguiente manera:

estudiante de clase pública {

gpa doble privada;

privado int totalCoursesTaken;

private int totalQualityPointsEarned;

private int countOfDsAndFs;

// otros detalles omitidos ...

public void completeCourse (String courseName,

int creditHours, char grade) {

if (grado == 'D' || grado == 'F') {

countOfDsAndFs ++;

}

// Registrar la calificación en la transcripción.

// detalles omitidos ...

// Actualizar un atributo ...

totalCoursesTaken = totalCoursesTaken + 1;

// ... y llama a un método de limpieza PRIVADO desde dentro de este

// método público para ajustar el GPA del estudiante en consecuencia.

updateGpa (creditHours, grade);

}

// Los detalles de CÓMO se actualiza el GPA son profundos y oscuros

// ¡secreto! Incluso la EXISTENCIA de este próximo método está oculta a

// el "mundo exterior" (es decir, inaccesible desde el código del cliente) por

// en virtud de haber sido declarado PRIVADO.

private void updateGpa (int creditHours, char grade) {

int letterGradeValue = 0;

if (grado == 'A') letterGradeValue = 4;

if (grado == 'B') letterGradeValue = 3;

if (grado == 'C') letterGradeValue = 2;

if (grado == 'D') letterGradeValue = 1;

// Para una 'F', permanece 0. int qualityPoints = creditHours \* letterGradeValue;

// Actualiza dos atributos.

totalQualityPointsEarned =

totalQualityPointsEarned + qualityPoints;

gpa = totalQualityPointsEarned / totalCoursesTaken;

}

}

El código de cliente no debería poder hacer que se actualice directamente el gpa de un objeto Student; esta solo debería ocurrir como un efecto secundario de completar un curso. Haciendo el método updateGpa privado, hemos evitado que cualquier código de cliente invoque explícitamente este método para manipular el valor de este atributo fuera de contexto.

**Excepción n. ° 3: Atributos de "solo lectura"**

Si solo proporcionamos un método "get" para un atributo, pero no un método "set", entonces ese atributo es renderizado de forma eficaz como de solo lectura desde la perspectiva del código del cliente. Podríamos hacerlo, por ejemplo, con el número de identificación del estudiante, que una vez configurado, no debe modificarse.

estudiante de clase pública {

private String studentId;

// detalles omitidos

// Representamos studentId como un atributo de solo lectura escribiendo solo un método get

// para ello.

public String getStudentId () {

return studentId;

}

// El método set se omite intencionalmente de la clase.

}

¿Cómo establecemos inicialmente el valor de tal atributo? Ya hemos visto que algunos atributos " los valores se modifican como efecto secundario de realizar un método (como con el método countOfDsAndFs

atributo que discutimos anteriormente). También veremos cómo inicializar explícitamente un atributo de solo lectura.

un poco más adelante en este capítulo, cuando hablamos de constructores.

Del mismo modo, podríamos optar por proporcionar solo un método de "conjunto" para un atributo, en el que caso de que el atributo sea de solo escritura. Y, si no proporcionamos un método "get" ni "set", hemos representaba efectivamente el atributo como un elemento de datos privado de "mantenimiento", como se discutió anteriormente.

**Excepción n. ° 4: atributos públicos**

En raras ocasiones, una clase puede declarar los atributos seleccionados como públicos para facilitar el acceso; esto es sólo se hace cuando no existe una lógica empresarial que gobierne los atributos per se. Un ejemplo es la clase Java Point predefinida, que se utiliza para definir un (x, y) coordinar en espacio bidimensional; sus atributos se declaran simplemente como

Punto de clase pública {

// Ambos atributos son públicos:

público doble x;

público doble y;

// etc.

}

para que, en el código del cliente, podamos asignar valores fácilmente de la siguiente manera:

Punto p = nuevo Punto ();

p.x = 3,7;

p.y = -4,8;

Dicho esto, resista la tentación de declarar atributos con accesibilidad pública simplemente como ¡una forma perezosa de evitar tener que escribir métodos "get" / "set"! Hemos visto los muchos beneficios que Los métodos "get" / "set" proporcionan en términos de hacer cumplir la lógica empresarial cuando sea apropiado. Mientras gira , la gran mayoría de los atributos deberán regirse por dicha lógica empresarial.

**Constructores**

Cuando hablamos sobre la creación de instancias de objetos en el capítulo anterior, es posible que haya tenido curiosidad sobre la interesante sintaxis involucrada con la nueva palabra clave:

Estudiante x = nuevo Estudiante ();

En particular, es posible que se haya preguntado por qué se añadieron paréntesis al final de la declaración.

Resulta que cuando creamos una instancia de un objeto a través de la nueva palabra clave, en realidad estamos invocando un tipo especial de procedimiento llamado constructor. La invocación de un constructor sirve como una solicitud al JVM para construir (instanciar) un objeto nuevo en tiempo de ejecución asignando suficiente programa memoria para albergar los atributos del objeto. Volviendo a nuestra analogía del "objeto como globo de helio", estamos pidiendo a la JVM que infle un nuevo globo de helio de un tipo en particular.

**Constructores predeterminados**

Si no declaramos explícitamente ningún constructor para una clase, Java proporciona automáticamente un constructor predeterminado para esa clase. El constructor predeterminado no tiene parámetros, es decir, no toma argumentos.

y hace el "mínimo" requerido para inicializar un nuevo objeto: es decir, configurar todos los atributos a sus valores predeterminados equivalentes a cero.

Por lo tanto, aunque hayamos diseñado una clase sin ningún constructor explícito, como en la siguiente clase de estudiantes:

estudiante de clase pública {

// Atributos.

nombre de cadena privada;

// otros detalles omitidos ...

// Hemos declarado métodos, pero NO CONSTRUCTORES EXPLÍCITOS.

public String getName () {

nombre de retorno;

}

public void setName (String newName) {

nombre = nuevoNombre;

}

// etc.

}

todavía podemos escribir código de cliente para crear una instancia de un objeto Student "básico" de la siguiente manera:

Estudiante s1 = nuevo Estudiante ();

porque la JVM usa el constructor predeterminado para la clase Student.

**Escribiendo nuestros propios constructores explícitos**

No necesitamos depender de Java para proporcionar un constructor predeterminado para cada una de nuestras clases; en su lugar podemos escribir constructores de nuestro propio diseño para una clase en particular si deseamos hacer algo más

"Interesante" para inicializar un objeto cuando se instancia por primera vez.

Tenga en cuenta que la sintaxis del encabezado de un constructor es un poco diferente a la de un método:

Público \_\_\_\_\_\_\_\_\_\_\_\_\_ Estudiante ()

acceso NO tipo de retorno! el nombre del constructor debe coincidirn nombre de la clase del modificador, seguido de lista de formales separados por comas parámetros encerrados entre ()

• El nombre de un constructor debe be exactamente el mismo que el nombre de la clase para la que estamos escribir el constructor, no tenemos elección en el asunto.

• Se proporciona una lista de parámetros, entre paréntesis, para un encabezado de constructor como con encabezados de método. Y, al igual que con los encabezados de métodos, la lista de parámetros puede dejarse vacía si apropiado.

• No podemos especificar un tipo de retorno para un constructor; por definición, un constructor devuelve una referencia a un objeto recién creado del tipo representado por la clase a la que pertenece el constructor. Es decir, un constructor de la forma

// Nota: ¡sin tipo de retorno!

Estudiante público () {...}

devuelve una referencia de objeto Student recién instanciada. Un constructor de la forma

// Nota: ¡sin tipo de retorno!

Profesor público () {...}

devuelve una referencia de objeto Professor recién instanciada, y así sucesivamente.

Otra disparidad con respecto a la sintaxis del constructor en comparación con la de los métodos es que invocar un constructor no implica notación de puntos:

Profesor p = nuevo profesor ();

Esto se debe a que no estamos solicitando un servicio de un objeto en particular; más bien, estamos solicitando que la JVM cree un objeto completamente nuevo.

**Pasar argumentos a constructores**

Una de las motivaciones más comunes para declarar un constructor explícito para una clase es proporcionar una forma conveniente de pasar valores iniciales para los atributos de un objeto en el momento de la instanciación.

Si usamos un constructor predeterminado para instanciar un objeto básico, entonces debemos invocar el los métodos "set" del objeto uno por uno para inicializar sus valores de atributo, como se ilustra en el siguiente fragmento:

// Crea un objeto Student básico.

Estudiante s = nuevo Estudiante ();

// Inicializa los atributos uno por uno.

s.setName ("Fred Schnurd");

s.setSsn ("123-45-6789");

s.setMajor ("MATEMÁTICAS");

// etc.

Esto puede resultar bastante tedioso si hay muchos atributos para inicializar.

Alternativamente, si diseñamos un constructor que acepta argumentos, podemos simultáneamente instanciar un objeto y proporcionar valores de atributo iniciales significativos en una sola línea de código, para ejemplo:

// Esta única línea de código reemplaza las cuatro líneas anteriores. Estudiante s = nuevo estudiante ("Fred Schnurd", "123-45-6789", "MATH"); Para lograr esto, por supuesto, tendríamos que declarar un constructor de clase Student con un encabezado apropiado, como se muestra aquí:

estudiante de clase pública {

// Atributos.

nombre de cadena privada;

Private String ssn;

principal de cadena privada;

// etc.

// Hemos declarado un constructor que acepta tres argumentos, para acomodar

// pasando tres valores de atributo.

Public Student (String s, String n, String m) {

this.setName (n);

this.setSsn (s);

this.setMajor (m);

}

// etc.

Tenga en cuenta que estamos invocando los métodos setName, setSsn y setMajor en nuestro constructor para establecer los valores del nombre asociado, ssn y atributos principales en lugar de acceder a estos atributos directamente, una mejor práctica que se discutió anteriormente en el capítulo.

Los argumentos del constructor también se pueden usar como indicadores de control para influir en cómo un constructor se comporta, como se ilustra en el siguiente constructor de ejemplo:

Estudiante público (nombre de cadena, valor predeterminado asignado booleano) {

setName (n);

si (asignar valores predeterminados) {

this.setSsn ("?");

this.setMajor ("UNDECLARED");

}

}

El código de cliente para lo anterior podría tener el siguiente aspecto:

// QUEREMOS asignar valores predeterminados a otros atributos.

Estudiante s = estudiante nuevo ("Cynthia Coleman", verdadero);

**Reemplazo del constructor sin parámetros predeterminado**

Si lo deseamos, podemos programar explícitamente un constructor sin parámetros para que nuestras clases hagan algo más interesante que simplemente crear una instancia de un objeto básico, reemplazando así el predeterminado constructor sin parámetros con uno de nuestro propio diseño. Esto se ilustra con la siguiente clase:

estudiante de clase pública {

// Atributos.

nombre de cadena privada;

principal de cadena privada;

// etc.

// Hemos programado explícitamente un constructor sin parámetros, reemplazando así

// la versión predeterminada.

Estudiante público () {

// Quizás deseamos inicializar los valores de los atributos a algo que no sea

// sus equivalentes cero.

this.setName ("?");

this.setMajor ("UNDECLARED");

// etc.

}

// Otros métodos omitidos en este ejemplo.

}

**Constructores más elaborados**

Podemos programar un constructor para que haga lo que tenga sentido al construir un nuevo estudiante.

• Es posible que deseemos crear instancias de objetos adicionales relacionados con el objeto Student:

Estudiante de clase pública () {

// Cada estudiante controla su propia transcripción individual

// objeto.

transcripción privada de la transcripción;

Estudiante público () {

// Cree un nuevo objeto Transcript para este nuevo alumno.

transcripción = nueva Transcripción ();

// etc.

}

// etc.

}

• Es posible que deseemos acceder a una base de datos relacional para leer los datos necesarios para inicializar el Atributos del alumno:

estudiante de clase pública {

// Atributos.

String studentId;

Nombre de cadena;

doble gpa;

// etc.

// Constructor.

Estudiante público (ID de cadena) {

studentId = id;

// Pseudocódigo.

use studentId como clave principal para recuperar datos de la tabla Student de un base de datos relacional;

Si(studentId encontrado en la tabla Student) {

recuperar todos los datos en el registro del estudiante;

nombre = nombre recuperado de la base de datos;

gpa = valor recuperado de la base de datos;

// etc.

}

}

// etc.

}

• Es posible que deseemos comunicarnos con otros objetos ya existentes para anunciar un nuevo Existencia del estudiante:

estudiante de clase pública {

// Detalles omitidos.

// Constructor.

Public Student (String major) {

// Avisar al departamento principal designado del estudiante que un nuevo estudiante

// se unió a la universidad.

// Pseudocódigo.

majorDept.notify (sobre este estudiante ...);

// etc.

}

// etc.

}

etc., lo que se requiera de nuestra aplicación. Veremos ejemplos de tales constructores más adelante en el libro, cuando creamos el SRS.

**Constructores de sobrecarga**

Así como se nos permite sobrecargar métodos en Java, se nos permite sobrecargar constructores.

Es decir, podemos escribir tantos constructores diferentes para una clase dada como queramos, siempre que tienen diferentes firmas de argumentos.

A continuación, se muestra un ejemplo de una clase de estudiantes que declara tres constructores diferentes:

estudiante de clase pública {

nombre de cadena privada;

Private String ssn;

int edad privada;

// etc.

// Constructor # 1: no acepta argumentos; reemplaza al constructor predeterminado.

Estudiante público () {

// Asignar valores predeterminados a los atributos seleccionados, si lo desea.

this.setSsn ("?");

// Aquellos que no se inicializan explícitamente en el constructor

// asume automáticamente el valor equivalente a cero para su tipo respectivo.

}

// Constructor # 2: toma un solo argumento de cadena.

Public Student (String s) {

this.setSsn (s);

}

// Constructor # 3: toma dos Strings y un int como argumentos.

Public Student (String s, String n, int i) {

this.setSsn (s);

this.setName (n);

this.setAge (i);

}

// Otros métodos omitidos en este ejemplo.

}

Al sobrecargar el constructor de una clase, hacemos que la clase sea más versátil al dar código de cliente una variedad de constructores para elegir, dependiendo de las circunstancias. Aquí está un ejemplo de código de cliente que ilustra el uso de las tres formas del constructor Student:

// No sabemos NADA sobre nuestro primer alumno, así que usamos el

// constructor sin parámetros para instanciar s1.

Estudiante s1 = nuevo Estudiante ();

// Conocemos el ssn (solo) para nuestro segundo estudiante, por lo que usamos el segundo

// forma de constructor para instanciar s2.

Estudiante s2 = nuevo estudiante ("123-45-6789");

// Conocemos el ssn, el nombre y la edad de nuestro tercer alumno, por lo que usamos

// la tercera forma de constructor para instanciar s3.

Estudiante s3 = nuevo estudiante ("987-65-4321", "John Smith", 21);

Al igual que con los métodos sobrecargados, el compilador puede hacer coincidir sin ambigüedades qué versión de constructor se invoca en cada caso en función de las firmas de los argumentos:

• (): Ningún argumento le dice al compilador que estamos invocando al constructor # 1.

• ("123-45-6789"): un argumento de cadena le dice al compilador que estamos invocando

constructor # 2.

• ("987-65-4321", "John Smith", 21): dos cadenas y un int como argumentos le dicen al

compilador que estamos invocando al constructor # 3.

Este ejemplo también refuerza por qué no puede haber dos constructores con la misma firma de argumento.

Si se nos permitiera introducir un cuarto constructor cuya firma de argumento se duplicara el del constructor # 2, por ejemplo:

// Constructor # 4: toma un solo argumento String, duplicando así el

// firma de argumento del constructor # 2.

Public Student (String n) {// ¡¡¡ESTO NO SE COMPILARÁ !!!

this.setName (n);

}

entonces el compilador no sabría qué constructor - # 2 o # 4 - estamos tratando de invocar en el

siguiente código de cliente:

// Pseudocódigo.

Estudiante x = nuevo Estudiante (aStringExpression);

Entonces, para evitar una situación tan ambigua, el compilador genera un mensaje de error en el declaración del constructor # 4, como sigue:

Student (java.lang.String) ya está definido en Student

Estudiante público (Cadena n) {}

^

**Una advertencia importante sobre el constructor predeterminado**

Hay una advertencia muy importante sobre los constructores predeterminados en Java: si declaramos alguno de los nuestros constructores para una clase, con cualquier firma de argumento, luego el constructor sin parámetros predeterminado no se proporciona automáticamente. Esto es por diseño, porque se supone que si hemos ido a el problema de programar cualquier constructor para una clase, entonces debemos tener algunos requisitos de inicialización para esa clase que el constructor predeterminado posiblemente no podría anticipar.

La implicación de esta característica del lenguaje es la siguiente: si queremos o necesitamos un constructor que no acepta argumentos para una clase en particular junto con otras versiones de constructores que toman argumentos, debemos programar explícitamente un constructor sin parámetros. Para ilustrar este punto, consideremos una clase Student que solo declara un constructor explícito:

estudiante de clase pública {

// Detalles omitidos.

// Solo se declara explícitamente un constructor, y que toma un

// argumento de cadena única.

Public Student (String s) {

this.setSsn (s);

}

// etc.

}en esta clase de la siguiente manera:

Estudiante s = nuevo estudiante ("123-45-6789");

pero si intentamos usar el constructor predeterminado (que ahora es inexistente)

Estudiante s = nuevo Estudiante ();

obtendremos el siguiente error de compilación:

no puedo encontrar el símbolo

símbolo: constructor Student ()

ubicación: clase Estudiante

Estudiante s = nuevo Estudiante ();

^

En términos generales, se considera una buena práctica proporcionar siempre explícitamente un constructor (para reemplazar el valor predeterminado perdido) si proporcionamos algún constructor para una clase. Bien Revise la importancia de esta práctica cuando analicemos la herencia en el Capítulo 5.

Un error común que cometen los programadores principiantes de Java es declarar accidentalmente un tipo de retorno en un encabezado de constructor, por ejemplo:

estudiante vacío público () {...}

Este es un error particularmente difícil de rastrear, porque si bien tales declaraciones de encabezado compilar, son vistos por el compilador como métodos y no como constructores, y no pueden ser utilizado como tal. Lo que es peor, los desarrolladores pensarán que han programado un constructor cuando en realidad no lo han hecho; cualquier intento de utilizar dicho constructor en su aplicación se encontrará con el siguiente mensaje de error de compilación aparentemente críptico:

Estudiante s = nuevo Estudiante ();

Este es el error del compilador:

no puedo encontrar el símbolo

símbolo: constructor Student ()

ubicación: clase Estudiante

Estudiante s = nuevo Estudiante ();

^

**Uso de la palabra clave "this"**

para facilitar la reutilización de constructores Anteriormente en este capítulo, cubrimos la palabra clave this e ilustramos cómo se puede usar para calificar características de una clase cuando se accede desde métodos de la misma clase, como en

estudiante de clase pública {

// Detalles omitidos.

public void printAllAttributes () {

System.out.println ("Nombre:" + this.getName ());

System.out.println ("ID de estudiante:" + this.getSsn ());

// etc.

}

}

Ahora vamos a explorar un segundo uso alternativo de esta palabra clave, relacionado con la reutilización del código. de un constructor a otro dentro de la misma clase.

Es concebible que si hemos sobrecargado el constructor de una clase, habrá algunos pasos de inicialización comunes requeridos para todas las versiones. Por ejemplo, digamos que, para todas las novedades estudiantes, debemos

• Avise a la oficina de registro de la existencia de este estudiante.

• Cree una transcripción para este estudiante.

Si tuviéramos que declarar tres constructores para la clase Student, sería tedioso duplicar la misma lógica en los tres (consulte las líneas de código en negrita):

estudiante de clase pública {

// Detalles de atributos omitidos.

// Constructor # 1.

Estudiante público () {

// Asignar valores predeterminados a los atributos seleccionados ... se omiten los detalles.

// Pseudocódigo.

alertar a la oficina de registro de la existencia de este estudiante

// Cree una transcripción para este estudiante.

transcripción = nueva Transcripción ();

}

// Constructor # 2.

Public Student (String s) {

this.setSsn (s);

// ¡Este código está duplicado desde arriba!

// Pseudocódigo.

alertar a la oficina de registro de la existencia de este estudiante

// Cree una transcripción para este estudiante.

transcripción = nueva Transcripción ();

// fin de la duplicación de código

}

// Constructor # 3.

Public Student (String s, String n, int i) {

this.setSsn (s);

this.setName (n);

this.setAge (i);

// ¡¡¡DUPLICACIÓN TODAVÍA OTRA VEZ !!!

// Pseudocódigo.

alertar a la oficina de registro de la existencia de este estudiante

// Cree una transcripción para este estudiante.

transcripción = nueva Transcripción ();

// fin de la duplicación de código

}

// etc.

}

Peor aún, si la lógica tuviera que cambiar, ¡tendríamos que cambiarla en los tres constructores! Por suerte, la palabra clave this viene a nuestro rescate. Desde dentro de cualquier constructor de una clase X, podemos invocar cualquier otro constructor de la misma clase X a través de la siguiente sintaxis: esto (argumentos opcionales); Reescribamos nuestros tres constructores de Student anteriores para que el constructor n. ° 2 aproveche de la lógica del n. ° 1, y el n. ° 3 se aprovecha del n. ° 2:

estudiante de clase pública {

// Detalles de atributos omitidos.

// Constructor # 1.

Estudiante público () {

// Asignar valores predeterminados a los atributos seleccionados ... se omiten los detalles.

// Haz las cosas comunes a los tres constructores en este primero

// constructor ...

// Pseudocódigo.

alertar a la oficina de registro de la existencia de este estudiante

// Cree una transcripción para este estudiante.

transcripción = nueva Transcripción ();

}

// Constructor # 2.

Public Student (String s) {

// ... entonces, ¡REUTILIZA el código del primer constructor dentro del segundo!

esta();

// Luego, haga lo que sea necesario para el constructor # 2.

this.setSsn (s);

}

// Constructor # 3.

Public Student (String s, String n, int i) {

// ... y REUTILIZA el código del primer constructor dentro del tercero!

esta();

// Luego, haga todo lo que sea necesario para el constructor # 3.

this.setSsn (s);

this.setName (n);

this.setAge (i);

}

// etc.

}

Al invocar this (); desde dentro de los constructores n. ° 2 y n. ° 3, pudimos eliminar toda duplicación de código. Codificamos la lógica compartida una vez, en el constructor # 1 (el constructor sin parámetros), y luego invocó el constructor sin parámetros desde dentro de los otros dos.

Si queremos En el código del cliente, podemos crear una instancia de un estudianteted para invocar el segundo constructor desde dentro del tercero en lugar de invocar el primero desde el tercero, simplemente modificaríamos nuestro uso de esto de la siguiente manera:

// Constructor # 3.

Public Student (String s, String n, int i) {

// Aquí, estamos reutilizando el código del SEGUNDO constructor dentro del tercer

// simplemente cambiando la firma del argumento usada con esto (...).

esto es);

// Luego, haga lo que sea necesario para el constructor # 3;

// detalles omitidos.

}

Porque hemos modificado la sintaxis de this (...); declaración en el constructor # 3 para suministrar un solo argumento de cadena esto es) el compilador sabe que es el segundo constructor, que toma un solo argumento String, que deseamos reutilizar el código de. Al usar este (...); sintaxis para reutilizar código de un constructor a otro, tenga en cuenta que la declaración debe ser la primera declaración del constructor; es decir, el siguiente código no compilará:

// Constructor # 3.

Public Student (String s, String n, int i) {

// Haga lo que sea necesario para el constructor # 3;

// detalles omitidos.

...

// Luego, intente reutilizar el código del constructor # 2;

// ¡ESTA SIGUIENTE LÍNEA NO SE COMPILARÁ!

esto es);

}

Aquí está el error:

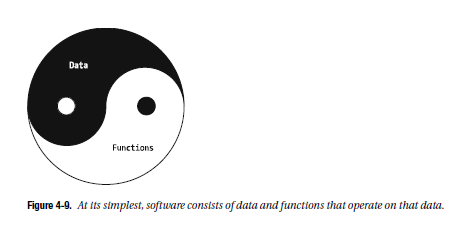
la llamada a esto debe ser la primera declaración en el constructor esto es);

^

Revisaremos esta palabra clave una vez más en el Capítulo 13, para analizar un tercer contexto en el que se puede usar.

**Software en su forma más simple, revisada**

Como discutimos en el Capítulo 3, el software en su forma más simple consta de dos componentes principales: datos, y funciones que operan en esos datos (vea la Figura 4-9).



También comparamos el enfoque de descomposición funcional del diseño de software con el enfoque orientado a objetos Acercarse. A modo de revisión, Con el enfoque de descomposición funcional para el desarrollo de software, nuestro principal la atención se centró en las funciones que debía realizar una aplicación; los datos eran un idea tardía. Es decir,

• Los datos se pasaron de una función a la siguiente.

• Por lo tanto, la estructura de datos tuvo que entenderse en muchos lugares, es decir, por muchas funciones. a lo largo de una aplicación.

• Si la estructura de datos de una aplicación tuvo que cambiar después de la implementación de la aplicación, no es trivial Los efectos dominó a menudo surgieron durante la aplicación.

• Si surgieron errores de integridad de los datos como resultado de una lógica defectuosa después de que una aplicación se haya completado integrado, a menudo era muy difícil señalar con precisión dónde, es decir, en qué función (es): es posible que se haya producido el error.

Ahora sabemos que al aprovechar los mecanismos de encapsulación más información ocultación, el enfoque orientado a objetos para el desarrollo de software soluciona la gran mayoría de estas deficiencias:

• Los datos se encapsulan dentro de los objetos como atributos y, si declaramos estos atributos tiene accesibilidad privada, entonces la estructura de datos debe ser entendida solo por el objeto / clase a la que pertenecen los datos.

• Si las declaraciones de atributos (privados) de una clase tienen que cambiar después de que una aplicación implementado, como fue el caso cuando modificamos la estructura de datos de la clase Student, reemplazar un atributo int age con un atributo Date birthDate; prácticamente no hay efectos dominó: solo debe cambiar la lógica interna de los métodos de la clase afectada.

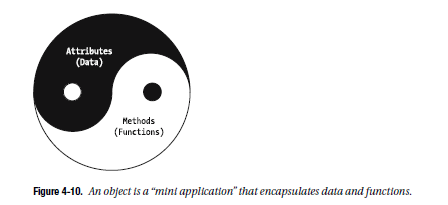
Figura 4-9. En su forma más simple, el software consta de datos y funciones que operan sobre esos datos.

(Recuerde que modificamos el funcionamiento interno del método getAge en uno de nuestros ejemplos de clase, pero que ninguno de los códigos de cliente que llamó a getAge tuvo que cambiar, porque no habíamos alterado la firma del método público del método).

• Cada clase es responsable de garantizar la integridad de los datos de su objeto. Por tanto, si la integridad de los datos surgen errores dentro de los datos de un objeto dado, podemos suponer que era la clase al que pertenece el objeto cuya lógica de método es defectuosa. (Recuerde el método updateBirthdate de un ejemplo anterior de la clase Student. Este método contenía todo tipo de comprobaciones de validez para garantizar que la cadena que se pasa como un El argumento representaba una fecha de nacimiento válida. Si de alguna manera se hubiera introducido una fecha de nacimiento no válida, saber que había algo defectuoso en la lógica de validación de la actualización Fecha de nacimiento método en particular.)

Si cada aplicación de software consta de datos y funciones que operan en esos datos, entonces un objeto puede considerarse como una especie de "mini aplicación" cuyos métodos (funciones) operan en sus atributos (datos), como se muestra en la Figura 4-10. Aprenderá en el Capítulo 5 cómo estos objetos "se unen obliga ”a colaborar en el cumplimiento de la misión general de una aplicación.

Figura 4-10. Un objeto es una "mini aplicación" que encapsula datos y funciones.



**Resumen**

En este capítulo, ha aprendido

• Cómo especificar formalmente los encabezados de los métodos, el "idioma" con el que se pueden utilizar los servicios. solicitado de un objeto, y cómo formular mensajes, utilizando notación de puntos, para obtener un objeto para realizar dichos servicios

• Que múltiples objetos a menudo tienen que colaborar en la realización de unafunción particular del sistema, como inscribir a un estudiante en un curso

• Que un objeto A solo puede comunicarse con otro objeto B si A tiene un asa en B, y las diversas formas en que se puede obtener dicho identificador / referencia

• Cómo las clases designan la accesibilidad pública / privada de sus características (atributos, métodos) a través de un mecanismo conocido como ocultación de información

• Qué tan poderoso es el ocultamiento de información de una característica del idioma, tanto en términos de protección integridad de los datos de un objeto y en la prevención de efectos dominó en el código del cliente cuando es privado los detalles de implementación de una aplicación cambian inevitablemente

• Cómo declarar y utilizar métodos de acceso ("get" / "set") para acceder con elegancia a los atributos de un objeto del código del cliente

• Cómo se especifica y se utiliza un tipo especial de procedimiento llamado constructor para controlar qué va a ocurrir cuando creamos nuevos objetos

• Cómo la sobrecarga permite que una clase tenga varios métodos con el mismo nombre and/or múltiples constructores tan largos como sus firmas de argumentos sean diferentes.