4.1 Events Drive Object Collaboration - 4.2.4 Method Return Types

Events Drive Object Collaboration:

El capítulo comienza mostrando 4 reglas básicas para el proceso de desarrollo de software de POOB:

1. Establecer correctamente los requisitos funcionales y la misión general de una aplicación.
2. Diseñar las clases adecuadas -sus estructuras de datos, comportamientos y relaciones entre sí-necesarias para cumplir estos requisitos y misión.
3. Instanciar estas clases para crear los tipos y el número de instancias de objetos adecuados
4. Puesta en marcha de estos objetos mediante eventos desencadenantes externos.

Después de esto el libro referencia la programación orientada a objetos de 2 formas:
-La primera es un hormiguero con hormigas que están de forma inactiva, si les cae un dulce cerca todas se alarman y comienzan a trabajar; dando referencia a las clases y objetos donde están estáticos y al momento de empezar algún método se empiezan a activar 1 o varias para completar su trabajo

-La segunda es un juego de billar, donde nosotros como usuarios al empezar un método le pegamos a una bola como objeto, donde este choca con mas objetos para hacerlos interactuar hasta que todos terminan de hacer sus trabajos y quedan estáticas como antes.

Declaring Methods:

Acá muestran cómo debería ser la interacción entre 2 objetos, colocándolos como “A” y “B”, donde A debe conocer los métodos de B para poder ordenarle que hacer (Además que colocan de ejemplo a un humano dándole órdenes a un perro):

- *El objeto A necesita tener claro exactamente cuál de los métodos/servicios quiere que B ejecute.*

> Piense en usted como el objeto A y en un perro como el objeto B. ¿Quiere que su perro se siente? ¿Se quede quieto?

- *En función del servicio solicitado, es posible que el objeto A deba proporcionar a B información adicional para que éste sepa exactamente cómo proceder.*

> Si le dice a su perro que busque, el perro necesita saber qué tiene que buscar: ¿una pelota? ¿Un palo? ¿El gato del vecino?

- *A su vez, el objeto B necesita saber si el objeto A espera que B le comunique el resultado de lo que se le ha pedido que haga.*

> de lo que se le ha pedido que haga. En el caso de una orden de ir a buscar algo, su perro le traerá el objeto solicitado como resultado. Sin embargo, si su perro está

en otra habitación y usted le ordena "¡Siéntate!", no verá el resultado de su orden; tendrá que confiar en que el perro ha hecho lo que usted le ha pedido.

Method Headers:

Documentación formal que se usa para los métodos en la cabecera donde se especifica como debe llamarse, los atributos que necesita de ingresar y si es necesario que retorna.

Ej del libro:

```
   boolean  registerForCourse(String courseID, int secNo)
return type  method name    comma-separated list of formal parameters,
                                    enclosed in parentheses
                                (parentheses may be left empty)
```

Method Naming Conventions:

Al igual que las variables en java, los métodos también se escriben de la forma Camel Casing:

La primera letra en minúscula y las siguientes letras de inicio de otra palabra en mayúsculas.

EJ: elejirAsesor (no es válido: ElejirAsesor, elejirasesor)

Passing Arguments to Methods:

Esta sección habla de los argumentos que debe recibir un método, indicando lo esencial o necesario que debe ser mandarle un argumento y si es necesario que ese método lo pida, o en cambio que el método no pida ningún argumento y solamente sea una función iterativa.

Method Return Types:

Esta sección habla de los tipos que puede retornar un método, empezando que, si uno no desea que retorne nada, directamente podemos colocar *Void* donde solo hará el llamado de función sin retornar, además podemos hacer un retorno de Booleans *true o false* donde indica si el procedimiento fue completado con éxito o hubo alguna discordancia en el camino; por ultimo se retornan los valores diferentes como puede ser un entero, carácter o cadena.

4.2.5 An Analogy - 4.3 Methods Implement Business Rules

Una analogía

Un método es una función - una función que es realizada por un objeto específico, pero una función al fin y al cabo.

Cuerpo de los métodos

Cuando diseñamos y programamos los métodos de una clase, no basta con declarar las cabeceras de los métodos: también debemos programar los detalles internos de cómo debe comportarse cada método cuando es invocado.

Las funciones pueden declararse en cualquier orden

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 dentro del método B aunque la declaración de la característica A venga después de la declaración del método B.

Declaración de retorno

```
void doSomething() {
  // Pseudocode.
  do whatever is required by this method ...

    return;
}
```

El cuerpo de un método puede incluir más de una sentencia return. Sin embargo de programación, sin embargo, es tener sólo una sentencia return en un método, al final.

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

Los cuerpos de los métodos con un tipo de retorno no vacío, por otro lado, deben incluir al menos una sentencia return explícita. En este caso, la palabra clave return debe ir seguida de una expresión que se evalúe a un valor compatible con el tipo de retorno declarado del método.

Los métodos implementan reglas de negocio

La lógica contenida dentro del cuerpo de un método define la lógica de negocio, también conocida como reglas de negocio, para una abstracción.

4.4 Objects As the Context for Method Invocation

Los objetos como contexto para invocación de métodos

Los métodos en una OOPL difieren de las funciones en una no-OOPL en que:

Las funciones son ejecutadas por el entorno de programación en su conjunto

Los métodos son ejecutados por objetos específicos, es decir, podemos invocar una función C “en el vacio” de la siguiente manera:

```
// A C program.
void main() {
    doSomething(42.0);  // invoke the doSomething function ...
    // etc.
}
```

Mientras que en un OOPL como Java, normalmente se debe calificar la llamada al método anteponiendo el nombre de la variable de referencia que representa al objeto que va a ejecutar el método, seguido del nombre de la variable de referencia. El nombre de la variable de referencia que representa el objeto que va a ejecutar el método, seguido de un punto

Esto se ve en el método registerForCourse:

```
// Instantiate two Student objects.
Student x = new Student();
Student y = new Student();

// Invoke the registerForCourse method on Student object x, asking it to
// register for course MATH 101, section 10; Student y is unaffected.
x.registerForCourse("MATH 101", 10);
```

Se refiere a una expresión de la forma referencia.Variable.NombreMetodo(args) como un mensaje:

“x.registerForCourse("MATH 101", 10);”

Puede interpretarse como “invocar un método en el objeto x” o “enviar un método al objeto x”.

Este código se debe considerar como una solicitud al objeto x para que realice un método como servicio, en nombre de la aplicación a la que pertenece el objeto.

Ya que utilizan un “punto” para añadir una llamada a un método a una variable de referencia concreta, se refiere a la notación referenceVariable.methodName(args) como notación de punto.

Ejemplo de tareas domesticas:

Una persona es capaz de:

-Sacar la basura

-Cortar el césped

-Lavar el coche

Expresión en java:

```
public class Person {
    // Attributes omitted from this snippet ...

    // Methods.
    void takeOutTheTrash() {  ... }
    boolean mowTheLawn() { ... }
    void washTheCar(Car c)  { ... }
}
```

Se decide que los hijos adolescentes Larry,Moe y Curly hagan cada uno una de las tres tareas mencionadas, ¿como se las pedimos? Si dijéramos simplemente:

“Por favor, lava el Camry”

“Por favor, saca la basura”

“Por favor, corta el césped, y avísame si vez alguna hierba cangreja”

Lo mas probable es que ninguna de estas tareas se haga, ya que no hemos encargado a los hijos a hacer una tarea en concreto. Larry,Moe y Curly probablemente se quedaran pegados al televisor, porque ninguno de ellos reconoce las peticiones que se les ha hecho

Ah, pero si ahora decimos esto:

- "Larry, por favor, lava el Camry."

- "Moe, por favor, saca la basura".

- "Curly, por favor, corta el césped, y avísame si ves alguna hierba cangreja".

Estaríamos dirigiendo cada solicitud a un hijo especifico, utilizando la sintaxis de java, esto se puede expresar de la siguiente manera

```
// We declare and instantiate three Person objects:
Person larry = new Person();
Person moe = new Person();
Person curly = new Person();

// And, while we're at it, a Car object, as well!
Car camry = new Car();

// We send a message to each son, indicating the service that we wish
// each of them to perform:
larry.washTheCar(camry);
moe.takeOutTheTrash();
boolean crabgrassFound = curly.mowTheLawn();

if (crabgrassFound) {
    // Pseudocode.
    handle the crabgrass ...
}
```

Al aplicar cada llamada a un método “hijo” especifico(referencia de objeto Persona), no hay ambigüedad en cuanto a que se le pide que realicen.

Asumiendo que takeOutTheTrash es un método definido para la clase Person como se ha ilustrado anteriormente, este código no compilara ya que le falta la notación de punto

```
public class BadCode {
  public static void main(String[] args) {
    // This next line won't compile -- where's the "dot"?  That is, which object
    // are we talking to???
    takeOutTheTrash();
  }
}
```

Teniendo el siguiente error:

```
cannot find symbol
symbol  :    method takeOutTheTrash()
location:  class BadCode
```

Sin embargo, en un lenguaje que no sea OOPL, como C, no existe la noción de objetos o clases, por lo que las funciones en estos lenguajes siempre se invocan “en el vacio”(entorno de programación en su conjunto):

```
// A C program.
void main() {
    doSomething(42.0);
    // etc.
}
```

Expresiones Java, revisadas

En el cap 2 de expresiones de java, una forma de expresión se omitió de la lista, a saber, los mensajes, porque aun no habíamos hablado de los objetos. Estas son las expresiones de java, añadiendo las expresiones de mensajes

- A constant: `7`, `false`
- A char(acter) literal: `'A'`, `'&'`
- A `String` literal: `"foo"`
- The name of any variable declared to be of one of the predefined types that we've seen so far: `myString`, `x`
- Any one of the preceding that is modified by one of the Java unary operators: `i++`
- ***A method invocation (“message”)***: `z.length()`
- Any two of the preceding that are combined with one of the Java binary operators: `z.length() + 2`
- Any of the preceding simple expressions enclosed in parentheses: `(z.length() + 2)`

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

Capturar el valor devuelto por un método

Siempre que se invoque un metodo con retorno de tipo no vacio, se puede elegir entre ignorar o reaccionar ante el valor que devuelve el metodo. En un ejemplo anterior, declaramos que el metodo registerForCourse de la clase Student con un tipo de retorno bool:

```
boolean registerForCourse(String courseId, int sectionNumber)
```

Pero no prestamos atención a que bool devuelve cuando se invoca el metodo:

```
x.registerForCourse("MATH 101", 10);
```

Si deseamos reaccionar ante el valor devuelto por un metodo no vacio, podemos optar por capturar el valor en una variable declarada del tipo apropiado, como este ejemplo:

```
boolean successfullyRegistered = x.registerForCourse("MATH 101", 10);

if (!successfullyRegistered) { // or:  if (successfullyRegistered == false)
    // Pseudocode.
    action to be taken if registration failed ...
}
```

Si solo pensamos utilizar el valor devuelto por un metodo en nuestro código, entonces la molestia de declarar la variable explicita como successfullyRegistered para capturar el resultado es una exageración, en vez de eso, podemos reaccionar al resultado simplemente anidando una expresión de mensaje dentro de una sentencia mas compleja.

Por ejemplo, podemos reeescribir el código eliminando successfullyRegistered:

```
if (!(x.registerForCourse("MATH 101", 10))) {
    // Pseudocode.
    action to be taken if registration failed ...
}
```

Ya que este metodo devuelve un valor bool, el mensaje x.registerForCourse(...) es una expresión booleana y puede utilizarse dentro del if de una sentencia.

A menudo combinamos llamadas a métodos con otros tipos de sentencias al desarrollar aplicaciones orientadas a objetos, ejemplo, devolver valores.

```
public class Student {
    Professor advisor;
    // Details omitted.

    public String getAdvisorsDepartment() {
        return advisor.getDepartment();  // a String expression
    }

    // etc.
}
```

O al imprimir la venta de comandos

```
Student s = new Student();
// Details omitted.
System.out.println("The student named " + s.getName() +
    " has a GPA of " + s.getGPA());
```

etc.

Firmas de métodos

La cabecera de un metodo consiste en como mínimo, en el tipo de retorno del método, el nombre y la lista formal de parámetros:

```
void switchMajor(String newDepartment, Professor newAdvisor)
```

Pero, 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 evidentes a primera vista

```
Student s = new Student();
Professor p = new Professor();
// Details omitted ...
s.chooseMajor("MATH", p);
```

- chooseMajor el nombre de un metodo definido para la clase Student; de lo contrario el compilador rechazaría esta línea
- el metodo Choosemajor declara dos parámetros de tipo string y profesor, respectivamente, porque esos son los tipos de argumentos que se usan en la clase student.

La firma de un metodo se refiere como aquellos aspectos en la cabecera de un metodo que son “descubribles” inspeccionando el código utilizado para invocar el metodo, a saber:

- El nombre del método

- El orden, los tipos y el número de parámetros declarados por el método

pero excluyendo

- Los nombres de los parámetros

- El tipo de retorno del método

Ejemplos de cabeceras de metodo y sus respectivas firmas de metodo:

- 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: `()`

Elegir nombres de métodos descriptivos

Asignar nombres descriptivos e intuitivos a los métodos ayuda a que el código sea autodocumentable, ejemplo:

```
public class IntuitiveNames {
    public static void main(String[] args) {
        Student student;
        Professor professor;
        Course course1;
        Course course2;
        Course course3;

        // Later in the program ...

        // This code is fairly straightforward to understand!
        // A student chooses a professor as its advisor ...
        student.chooseAdvisor(professor);
        // ... and registers for the first of three courses.
        student.registerForCourse(course1);

        // etc.
```

Compara este código con el siguiente:

```
public class FuzzyNames {
    public static void main(String[] args) {
        Student s;
        Professor p;
        Course c1;
        Course c2;
        Course c3;

        // Later in the program ...

        // Without comments, this next bit of code is not nearly as intuitive.
        s.choose(p);
        s.reg(c1);

        // etc.
```

4.5 Method Overloading - 4.8 Obtaining Handles on Objects

Method overloading

Es la capacidad de una clase de tener metodos con nombres iguales, pero diferentes parametros, esto significa que los programadores pueden usar mismos nombres para diferentes metodos y asi aplicarlos de forma sencilla, sin embargo para que esto se pueda hacer, los parametros de cada metodo con nombre igual deben tener diferentes tipos.

Esto permite la modificacion por un proceso de iteraciones a traves de objetos del mismo tipo o de objetos modificados de la misma forma.

Delegation

Consiste basicamente como la delegacion entre personas, en las cuales alguien le pide a otro que haga algo y esa persona puede decirle a otro lo mismo y asi, pasa lo mismo entre objetos, si para que un objeto haga algo en especifico necesita de la accion de otro y su resultado a la vez.

Obtaining Handles on Objects

Aqui se habla de como podemos dentro de los metodos de las clases, llamar a objetos ya definidosy ademas otros metodos dentro de estos objetos, asi podemos usar la delegacion para usar varias propiedades de los mismos, lo que significa que podemos obtener referencias o identificadores a objetos y asi operarlos, lo cual se logra.

Esto se logra creando e instanciando un nuevo objeto y accediento a sus propiedades dentro de una nueva clase o metodo y asi usarlo.

4.9 Objects As Clients and Suppliers - 4.10.4 Method Headers, Revisited

En la programación orientada a objetos, los objetos interactúan entre sí mediante el paso de mensajes, donde un objeto solicita un servicio a otro invocando sus métodos. Esta interacción establece roles de clientes y proveedores entre objetos durante un evento específico de paso de mensajes.

- Cliente: Un objeto que solicita un servicio a otro invocando sus métodos. En el ejemplo dado, el objeto Course se considera un cliente cuando llama al método register en el objeto Student.
- Proveedor: Un objeto que proporciona el servicio solicitado por el cliente. En el ejemplo, el objeto Student actúa como proveedor cuando proporciona su objeto Transcript al objeto Course a través del método getTranscript, y el objeto Transcript actúa como proveedor cuando verifica la finalización exitosa de un curso.

Los roles de cliente y proveedor no son fijos; dependen del evento específico de paso de mensajes. Por ejemplo, en un escenario donde un objeto solicita un servicio y otro lo proporciona, el solicitante es el cliente y el proveedor es el proveedor. Sin embargo, en un evento posterior donde los roles se invierten, el solicitante se convierte en el proveedor y viceversa.

En la programación orientada a objetos, podemos acceder a los atributos de un objeto utilizando la notación de punto. Sin embargo, aunque esta capacidad existe, no siempre es recomendable acceder directamente a los atributos de un objeto. Existen razones para restringir el acceso a los datos de un objeto, y hay mecanismos para hacer que el compilador de Java nos ayude a hacer cumplir esas restricciones.

En la práctica, los objetos a menudo restringen el acceso a algunas de sus características (atributos o métodos). Esta restricción se conoce como ocultamiento de información. En una aplicación orientada a objetos bien diseñada, una clase generalmente hace públicos los servicios que pueden realizar sus objetos, pero oculta los detalles internos de cómo realizan estos servicios, así como los datos que mantienen internamente para soportar estos servicios.

Por ejemplo, piensa en un anuncio de Páginas Amarillas para una tintorería. El anuncio promoverá los servicios que la tintorería ofrece, como la limpieza de ropa formal o alfombras, pero no revelará los detalles de cómo realizan la limpieza. De manera similar, en programación orientada a objetos, los objetos exponen los servicios que pueden proporcionar pero ocultan los detalles de implementación y los datos internos.

La accesibilidad de una característica de un objeto se establece mediante modificadores de acceso colocados al principio de su declaración. Java define varios modificadores de acceso, siendo los dos principales `private` y `public`.

- `private`: Los miembros marcados como `private` solo son accesibles dentro de la clase en la que están declarados. No pueden ser accedidos desde fuera de la clase, ni siquiera a través de la herencia.

- `public`: Los miembros marcados como `public` son accesibles desde cualquier otra clase. Pueden ser accedidos desde cualquier lugar donde se tenga una referencia al objeto que los contiene.

El uso adecuado de estos modificadores de acceso es fundamental para garantizar un diseño de clase seguro y coherente, donde la información se oculte adecuadamente y solo se expongan los servicios necesarios para interactuar con los objetos

Cuando una característica se declara como pública, es accesible libremente desde el código cliente utilizando la notación de punto. Por ejemplo, si declaramos el atributo name de la clase Student como público, podemos acceder directamente a él desde el código cliente:

```
public class Student {
   public String name;
   // etc.
}
```

De manera similar, si declaramos el método isHonorsStudent como público, podemos invocarlo desde el código cliente:

```
public class Student {
   // Atributos omitidos.
   // Métodos.
   public boolean isHonorsStudent() { ... }
   // etc.
}
```

Por otro lado, cuando una característica se declara como privada, no es accesible desde fuera de la clase en la que se declara. Por ejemplo, si declaramos el atributo ssn de la clase Student como privado:

```
public class Student {
   public String name;
   private String ssn;
   // etc.
}
```

Entonces no podemos acceder directamente a ssn desde el código cliente. Lo mismo es cierto para los métodos privados. Si declaramos un método como privado, no podemos invocarlo desde el código cliente:

```
public class Student {
   // Atributos omitidos.
   // Métodos.
   public boolean isHonorsStudent() { ... }
   private void printInfo() { ... }
   // etc.
}
```

En la programación orientada a objetos, los métodos de una clase suelen declararse como públicos porque un objeto necesita publicitar sus servicios para que el código cliente pueda solicitar estos servicios, como en la analogía de la publicidad en las Páginas Amarillas. Por otro lado, la mayoría de los atributos suelen declararse como privados (y efectivamente "ocultos"), para que un objeto pueda mantener un control completo sobre sus datos.
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 solicita a otro objeto B que realice 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 simplemente necesita confiar en que el objeto B realizará el servicio "publicitado". Esto se representa conceptualmente como una pared de ladrillos impenetrable que separa las partes privadas de una clase/objeto del código cliente.

4.10.5 Accessing the Features of a Class from Within Its Own Methods

(Acceso a las características de una clase por medio de sus propios métodos)

En esta sección se discute acerca de la accesibilidad de las caracteristicas de una clase. Se destaca que, independientemente de la accesibilidad (pública o privada) de las características de una clase, se puede acceder a todas por medio de los métodos de la propia clase. Además, se muestra que no es necesario utilizar la notación de punto para acceder a las características de la clase desde sus propios métodos; el compilador comprende automáticamente que se está accediendo a las características de la propia clase. Lo anterior se ilustra mediante un ejemplo de una clase Student:

```
public class Student {
    // A few private attributes.
    private String name;
    private String ssn;
    private double totalLoans;
    private double tuitionOwed;

    // Get/set methods would be provided for all of these attributes;
    // details omitted ...

    public void printStudentInfo() {
        // Accessing attributes of the Student class.
        System.out.println("Name:  " + name);
        System.out.println("Student ID:  " + ssn);
        // etc.
    }

    public boolean allBillsPaid() {
        boolean answer = false;
        // Accessing another method of the Student class.
        double amt = moneyOwed();

        if (amt == 0.0) {
          answer = true;
        }
        else {
          answer = false;
        }

        return answer;
    }
```

En el ejemplo anterior podemos ver como la función printStudentInfo() está accediendo al atributo name sin la necesidad de la notación de punto. De la misma manera, la función allBillsPaid() accede a la función moneyOwned() definida posteriormente sin la notación de punto.

Se introduce el uso de la palabra clave this en notación de punto (this.featureName) para enfatizar que se está accediendo a otra característica de la misma clase. Veamos cómo se usa la palabra clave con el ejemplo anterior.

```
    public void printStudentInfo() {
        // We've added the prefix "this.".
        System.out.println("Name:  " + this.name);
        System.out.println("Student ID:  " + this.ssn);
        // etc.
    }

    public boolean allBillsPaid() {
        boolean answer = false;
        // We've added the prefix "this.".
        double amt = this.moneyOwed();

        if (amt == 0.0) {
          answer = true;
        }
        else {
          answer = false;
        }
```

Se discute que ambas prácticas, tanto usar this.featureName como omitir el prefijo this., son aceptables, y la práctica común es no usar el prefijo this. a menos que sea necesario

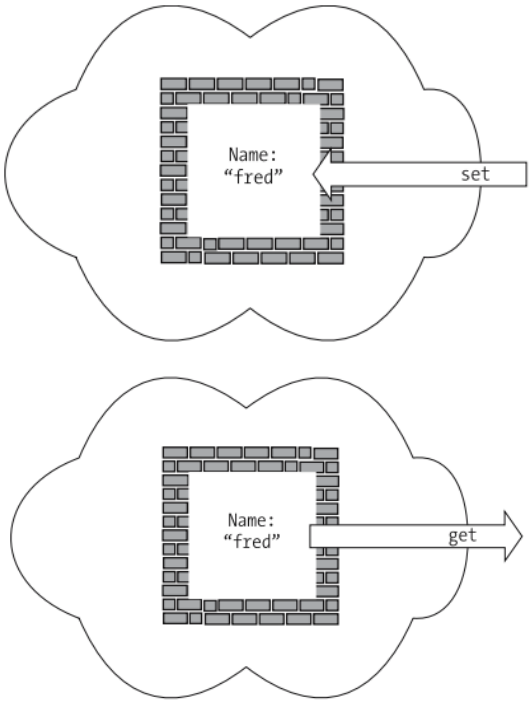
para evitar ambigüedades. También se menciona que se puede utilizar this para desambiguar entre un parámetro de método y un atributo de la clase si tienen el mismo nombre. Finalmente, indican que la palabra clave this se utiliza también en otros contextos, como en la reutilización de código y la auto-referencia de objetos.

4.11 Accessing Private Features from Client Code

(Acceder a caracteristicas privadas desde el código del cliente)

En esta sección se aborda la forma en que se accede a las características privadas de una clase desde el código del cliente, a través de métodos de acceso públicos (conocidos como "get" y "set"). Se destaca la práctica recomendada de proporcionar métodos de acceso públicos para permitir que el código cliente lea o modifique valores de atributos privados de un objeto, manteniendo así el principio de encapsulación.

Se explica la convención de nomenclatura para estos métodos, donde los métodos "get" se utilizan para obtener valores y los métodos "set" para establecer valores.



En el siguiente ejemplo, se usan atributos privados (name, facultyAdvisor) y se muestran los métodos de acceso públicos correspondientes (getName, setName, getFacultyAdvisor, setFacultyAdvisor).

```
public class Student {
   private String name;
   private Professor facultyAdvisor;

   public String getName() {
      return name;
   }

   public void setName(String newName) {
      name = newName;
   }

   public Professor getFacultyAdvisor() {
      return facultyAdvisor;
   }

   public void setFacultyAdvisor(Professor p) {
      facultyAdvisor = p;
   }
   // Other attributes and methods may be present (omitted in this example).
}
```

Hay una excepción en la convención de nombres: cuando un atributo es de tipo booleano, se recomienda comenzar el nombre del método "get" con "is" en lugar de "get".

Finalmente, se destaca que el valor de los atributos persiste mientras el objeto esté en memoria, y la importancia de utilizar métodos de acceso desde el código cliente para interactuar con estos atributos.

4.12 El Poder de la Encapsulación Más el Ocultamiento de Información

Es una técnica que permite limitar el acceso a los atributos y métodos de un objeto, evitando que se modifiquen o accedan a ellos desde fuera de la clase que al hacer esto se obtienen los tres siguientes beneficios

- Prevenir el acceso no autorizado a los datos encapsulados.
- Ayudar a garantizar la integridad de los datos.
- Limitar los "efectos de onda" que de otra manera pueden ocurrir en toda una aplicación.

Previniendo el Acceso No Autorizado a Datos Encapsulados

Hay atributos de clases que tienen un alto nivel de confidencialidad, por ejemplo el id de un estudiante o el número de una cuenta bancaria. Al hacer al atributo primado, se evita que cualquier objeto tenga acceso a este

Ayudando a Garantizar la Integridad de los Datos

Al establecer atributos como privados, se evita que cualquier objeto que lo desee pueda cambiar el valor de este. Además, estableciendo un atributo como privado y creando un método que pueda modificar al atributo, se podrán poner ciertas restricciones que ayudarán a salvaguardar la integridad de la información que se quiera guardar, un ejemplo de esto se muestra en las siguientes líneas de código:

```
public class Student {
    private String birthDate;
    // other attributes omitted from this example ...

    public boolean updateBirthDate(String newBirthDate) {
        boolean newDateApproved;

        // Perform appropriate validations.
        // Remember, italics represent pseudocode ...
        if (date is not in the format mm/dd/yyyy) {
            newDateApproved = false;
        }
        else if (mm not in the range 01 to 12) {
            newDateApproved = false;
        }
        else if (the day number isn't valid for the selected month) {
            newDateApproved = false;
        }
        else if (the year is NOT a leap year, but 2/29 was specified) {
            newDateApproved = false;
        }
        // etc. for other validation tests.


          else {
              // If we've gotten this far in the code, all is well with what was
              // passed in as a value to this method, and so we can go ahead and
              // update the value of the birthDate attribute with this value.
              birthDate = newBirthDate;

              // Set our flag to indicate success!
              newDateApproved = true;
          }

          return newDateApproved;
      }

      // etc.
  }
```

Limitando los "Efectos de Onda" Cuando las Características Privadas Cambian

La encapsulación y el ocultamiento de información permiten controlar el acceso a los datos y proteger la complejidad interna de una clase, sin afectar su uso externo evitando "Efectos de onda" no deseados. un ejemplo de esto se da en las siguientes líneas de código

The "Before" Code	The "After" Code

`public class Student {`
`  // We have an explicit`
`  // age attribute.`
`  private int age;`
 	
`  public int getAge() {`
`  return age;`
`  }`
 	
`  // etc.`
`}`
	`import java.util.Date;`
	
	`public class Student {`
	`  // We replace age with`
	`  // birthDate.`
	`  private Date birthDate;`
	
	`  public int getAge() {`
	`  // Compute the age on demand`
	`  // (pseudocode).`
	`  return system date - birthDate;`
	`  }`
	
	`  // etc.`
	`}`

A pesar de haber cambiado el código, este va a seguir funcionando de manera correcta, y sin que el cliente se haya dado cuenta del cambio realizado, las funcionalidades seguirán intactas.

Mientras restrinjamos nuestros cambios a las características privadas de una clase, los efectos de onda no son un problema; cualquier código cliente que haya sido escrito previamente para usar métodos públicos de Student continuará funcionando según lo previsto.

un ejemplo de esto son las siguientes líneas de código, en las cuales se modificó el tipo del atributo age, en la mayoría de los casos después de la implementación de código se esperaría que el valor recibido fuera de tipo entero, pero como ahora es double generaría un error de compilación

```
public class Student {
    // We've changed the type of the age attribute from int to double ...
    private double age;

    // ... and the return type of the getAge() method accordingly.
    public double getAge() {
        return age;
    }
```

Utilizando Métodos de Acceso desde Dentro de los Propios Métodos de una Clase

Es importante utilizar los propios métodos "get"/"set" de una clase en lugar de acceder directamente a los atributos por varias razones:

Encapsulamiento y Ocultamiento de Información: Al acceder a los atributos a través de sus métodos "get"/"set", se garantiza que cualquier lógica de negocio asociada con esos atributos se aplique correctamente. Esto ayuda a mantener la integridad de los datos y a prevenir cambios no deseados en los mismos.

Adaptabilidad y Mantenibilidad: Si en el futuro se realizan cambios en la lógica de negocio de los métodos "get"/"set", todas las partes del código que los utilizan se beneficiarán automáticamente de esos cambios. Esto significa que no es necesario modificar manualmente todas las instancias en el código donde se accede a esos atributos.

Facilidad de Depuración: Utilizar los métodos "get"/"set" proporciona una capa adicional de abstracción, lo que facilita la identificación y corrección de errores en la lógica de acceso a los atributos.

4.13 Exceptions to the Public/Private Rule

Aunque a menudo

- Los atributos se declaran privados.
- Los métodos se declaran públicos.
- Se accede a los atributos privados a través de métodos públicos.

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

Exception #1: Internal Housekeeping Attributes (Atributos de gestión interna)

Un atributo puede ser utilizado por una clase con fines estrictamente internos. Para tales atributos, no necesitamos molestarnos en proporcionar accesores públicos.

Exception #2: Internal Housekeeping Methods (Métodos de gestión interna)

Algunos métodos también pueden utilizarse estrictamente para fines internos, en cuyo caso estos también pueden ser declarados privados en lugar de públicos.

Exception #3: “Read-Only” Attributes (Atributos de “sólo lectura”)

Si sólo proporcionamos un método "get" para un atributo, pero no un método "set", entonces ese atributo es de sólo lectura desde la perspectiva del código cliente.

Exception #4: Public Attributes (Atributos públicos)

En raras ocasiones, una clase puede declarar atributos seleccionados como públicos para facilitar el acceso.

Esto sólo se hace cuando no hay una lógica de negocio que gobierne los atributos en sí.

Constructores

Cuando instanciamos un objeto mediante la palabra clave new, en realidad estamos invocando un tipo especial de procedimiento llamado constructor. Invocar un constructor sirve como una petición a la JVM para construir (instanciar) un nuevo objeto en tiempo de ejecución mediante la asignación de programa suficiente para albergar los atributos del objeto.

```
Student x = new Student();
```

Constructores por defecto

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

y hace lo "mínimo" necesario para inicializar un nuevo objeto: es decir, establece todos los atributos a sus valores por defecto equivalentes a cero.

Escribir nuestros propios constructores explícitos

Podemos crear constructores de nuestro propio diseño para una clase en particular si deseamos hacer algo más "interesante" para inicializar un objeto cuando se instancie por primera vez.

```
public          _____________       Student()
access       NO return type!     constructor name must match
modifier                         class name, followed by
                                 comma-separated list of formal
                                 parameters enclosed in ()
```

- El nombre de un constructor debe ser exactamente el mismo que el de la clase para la que estamos escribiendo.

- Se proporciona una lista de parámetros, encerrada entre paréntesis, la lista de parámetros puede dejarse vacía si es apropiado.

- No podemos especificar un tipo de retorno para un constructor; por definición, un constructor devuelve un objeto recién creado del tipo representado por la clase a la que pertenece el constructor.

Pasar argumentos a los constructores

Si utilizamos un constructor por defecto para instanciar un objeto básico, debemos invocar los métodos "set" del objeto una vez.

```
// Create a bare-bones Student object.
Student s = new Student();

// Initialize the attributes one by one.
s.setName("Fred Schnurd");
s.setSsn("123-45-6789");
s.setMajor("MATH");
// etc.
```

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

```
// This single line of code replaces the previous four lines.
Student s = new Student("Fred Schnurd", "123-45-6789", "MATH");
```

```
public class Student {
  // Attributes.
  private String name;
  private String ssn;
  private String major;
  // etc.

// We've declared a constructor that accepts three arguments, to accommodate
// passing in three attribute values.
public Student(String s, String n, String m) {
  this.setName(n);
  this.setSsn(s);
  this.setMajor(m);
}

// etc.
```

Sustitución del constructor sin parámetros por defecto

Si lo deseamos, podemos programar explícitamente un constructor sin parámetros para nuestras clases para hacer algo más interesante que la mera instanciación de un objeto sin parámetros, sustituyendo así el por defecto por uno diseñado por nosotros.

```
public class Student {
    // Attributes.
    private String name;
    private String major;
    // etc.

    // We've explicitly programmed a parameterless constructor, thus replacing
    // the default version.
    public Student() {
        // Perhaps we wish to initialize attribute values to something other than
        // their zero equivalents.

         this.setName("?");
         this.setMajor("UNDECLARED");
         // etc.
     }

     // Other methods omitted from this example.
 }
```

Constructores más elaborados

Podemos realizar varias funciones dentro de la clase constructor como:

- Podemos desear instanciar objetos adicionales relacionados con el objeto Estudiante.
- Podemos acceder a una base de datos relacional para leer los datos necesarios para inicializar los atributos del Estudiante.
- Es posible que deseemos comunicarnos con otros objetos ya existentes para anunciar la existencia de un nuevo Estudiante.

(Hay ejemplos en el libro detallando cada uno)

4.14.6 Overloading Constructors (Sobrecarga de constructores)

Se permite sobrecargar constructores, es decir, es posible escribir tantos constructores diferentes para una clase determinada como se desee, siempre que tengan firmas de argumentos diferentes. Al sobrecargar el constructor de una clase, la clase es más versátil al darle al código del cliente una variedad de constructores para elegir. Al igual que con los métodos sobrecargados, el compilador puede hacer coincidir sin ambigüedades qué versión del constructor se invoca en cada caso en función de las firmas de los argumentos.

4.14.7 An Important Caveat Regarding the Default Constructor (Una advertencia importante sobre el constructor predeterminado)

Si declaramos cualquiera de nuestros propios constructores para una clase, con cualquier firma de argumento, entonces el constructor predeterminado sin parámetros no se

proporciona automáticamente. La implicación de esta característica del lenguaje es la siguiente: si queremos o necesitamos un constructor que no acepte argumentos para una clase particular junto con otras versiones de constructores que sí toman argumentos, debemos programar explícitamente un constructor sin parámetros.

Se considera una buena práctica proporcionar siempre explícitamente un constructor sin parámetros si proporcionamos algún constructor para una clase. Un error común que cometen los programadores principiantes de Java es declarar accidentalmente un tipo de retorno en el encabezado de un constructor, por ejemplo:

```
public void Student() { ... }
```

Esto hará que el programa considere al constructor como un método y no como un constructor.

4.14.8 Using the "this"Keyword to Facilitate Constructor Reuse (Uso de la palabra clave "this" para facilitar la reutilización del constructor)

La palabra clave "this" se puede usar para calificar opcionalmente características de una clase cuando se accede desde dentro de métodos de la misma clase.
Un segundo uso alternativo de la palabra clave "this", relacionado con la reutilización de código de un constructor por otro dentro de la misma clase. Es posible que si hemos sobrecargado el constructor de una clase, se requieran algunos pasos de inicialización comunes para todas las versiones. Si fuera necesario cambiar la lógica, ¡tendríamos que cambiarla en los tres constructores! Afortunadamente, 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 con la siguiente sintaxis:

```
this(optional arguments);
```

4.15 Software at Its Simplest, Revisited (Software en su forma más simple, revisado)

El software en su forma más simple consta de dos componentes principales: datos y funciones que operan sobre esos datos.
Con el enfoque de descomposición funcional para el desarrollo de software, nuestro enfoque principal estaba en las funciones que debía realizar una aplicación; los datos fueron una ocurrencia tardía.

Eso es,
• Los datos se pasaban de una función a la siguiente.
• Por lo tanto, la estructura de datos tenía que entenderse mediante muchas funciones a lo largo de una aplicación.

• Si la estructura de datos de una aplicación tenía que cambiar después de su implementación, a menudo surgían efectos dominó no triviales en toda la aplicación.
• Si surgían errores de integridad de datos como resultado de una lógica defectuosa a menudo era muy difícil determinar con precisión en qué funciones específicas podría haber ocurrido el error.

El enfoque orientado a objetos para el desarrollo de software soluciona la gran mayoría de estas deficiencias:
• Los datos se encapsulan dentro de objetos como atributos y, si declaramos que estos atributos tienen accesibilidad privada, entonces la estructura de datos debe ser entendida sólo 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 se haya implementado una aplicación: sólo se debe cambiar la lógica interna de los métodos de la clase afectada.
• Cada clase es responsable de garantizar la integridad de los datos de su objeto. Por lo tanto, si surgen errores de integridad de datos dentro de los datos de un objeto determinado, podemos asumir que fue la clase a la que pertenece el objeto cuya lógica del método es defectuosa.
Si cada aplicación de software consta de datos y funciones que operan sobre esos datos, entonces un objeto puede considerarse como una especie de “mini aplicación” cuyos métodos (funciones) operan sobre sus atributos (datos).