5.D. Encapsulación, control de acceso y visibilidad.

Sitio: Formación Profesional a Distancia
Curso: Programación
Libro: 5.D. Encapsulación, control de acceso y visibilidad.
Imprimido por: Iván Jiménez Utiel
Día: martes, 7 de enero de 2020, 22:18

Tabla de contenidos

1. Encapsulación, control de acceso y visibilidad.

Dentro de la Programación **Orientada a Objetos** ya has visto que es muy importante el concepto de **ocultación**, la cual ha sido lograda gracias a la **encapsulación** de la información dentro de las clases. De esta manera una clase puede ocultar parte de su contenido o restringir el acceso a él para evitar que sea manipulado de manera inadecuada. Los **modificadores de acceso** en Java permiten especificar el **ámbito de visibilidad** de los miembros de una clase, proporcionando así un mecanismo de accesibilidad a varios niveles.

Acabas de estudiar que cuando se definen los miembros de una clase (atributos o métodos), e incluso la propia clase, se indica (aunque sea por omisión) un modificador de acceso. En función de la visibilidad que se desee que tengan los objetos o los miembros de esos objetos se elegirá alguno de los modificadores de acceso que has estudiado. Ahora que ya sabes cómo escribir una clase completa (declaración de la clase, declaración de sus atributos y declaración de sus métodos), vamos a hacer un repaso general de las opciones de **visibilidad** (**control de acceso**) que has estudiado.

Los modificadores de acceso determinan si una clase puede utilizar determinados miembros (acceder a atributos o invocar miembros) de otra clase. Existen dos niveles de control de acceso:

1. **A nivel general (nivel de clase)**: visibilidad de la propia clase.

2. **A nivel de miembros**: especificación, miembro por miembro, de su nivel de visibilidad.

En el caso de la clase, ya estudiaste que los niveles de visibilidad podían ser:

- **Público** (modificador public), en cuyo caso la clase era visible a cualquier otra clase (cualquier otro fragmento de código del programa).

- **Privada al paquete** (sin modificador o modificador "por omisión"). En este caso, la clase sólo será visible a las demás clases del mismo paquete, pero no al resto del código del programa (otros paquetes).

En el caso de los miembros, disponías de otras dos posibilidades más de niveles de accesibilidad, teniendo un total de cuatro opciones a la hora de definir el control de acceso al miembro:

- **Público** (modificador `public`), igual que en el caso global de la clase y con el mismo significado (miembro visible desde cualquier parte del código).

- **Privado al paquete** (sin modificador), también con el mismo significado que en el caso de la clase (miembro visible sólo desde clases del mismo paquete, ni siquiera será visible desde una subclase salvo si ésta está en el mismo paquete).

- **Privado** (modificador `private`), donde sólo la propia clase tiene acceso al miembro.

- **Protegido** (modificador `protected`)

Para saber más

Puedes echar un vistazo al artículo sobre el control de acceso a los miembros de una clase Java en los manuales de Oracle (en inglés):

Controlling Access to Members of a Class.

Autoevaluación

Si queremos que un atributo de una clase sea accesible solamente desde el código de la propia clase o de aquellas clases que hereden de ella, ¿qué modificador de acceso deberíamos utilizar?

- private.
- protected.
- public.
- Ninguno de los anteriores.

1.1. Ocultación de atributos. Métodos de acceso.

Los atributos de una clase suelen ser declarados como privados a la clase o, como mucho, `protected` (accesibles también por clases heredadas), pero no como `public`. De esta manera puedes evitar que sean manipulados inadecuadamente (por ejemplos modificarlos sin ningún tipo de control) desde el exterior del objeto.

En estos casos lo que se suele hacer es declarar esos atributos como privados o protegidos y crear métodos públicos que permitan acceder a esos atributos. Si se trata de un atributo cuyo contenido puede ser observado pero no modificado directamente, puede implementarse un método de "obtención" del atributo (en inglés se les suele llamar método de tipo **get**) y si el atributo puede ser modificado, puedes también implementar otro método para la modificación o "establecimiento" del valor del atributo (en inglés se le suele llamar método de tipo **set**). Esto ya lo has visto en apartados anteriores.

Si recuerdas la clase `Punto` que hemos utilizado como ejemplo, ya hiciste algo así con los métodos de obtención y establecimiento de las coordenadas:

```
private int x, y;
```

```
// Métodos get
```

```
public int obtenerX () { return x; }
```

```
public int obtenerY () { return y; }
```

```
// Métodos set
```

```
public void establecerX (int x) { this.x= x; }
```

```
public void establecerY (int y) { this.y= y; }
```

Así, para poder obtener el valor del atributo `x` de un objeto de tipo `Punto` será necesario utilizar el método `obtenerX()` y no se podrá acceder directamente al atributo `x` del objeto.

En algunos casos los programadores directamente utilizan nombres en inglés para nombrar a estos métodos: `getX ,getY (), setX, setY, getNombre, setNombre, getColor`, etc.

También pueden darse casos en los que no interesa que pueda observarse directamente el valor de un atributo, sino un determinado procesamiento o cálculo que se haga con el atributo (pero no el valor original). Por ejemplo podrías tener un atributo **DNI** que almacene los 8 dígitos del DNI pero no la letra del **NIF** (pues se puede calcular a partir de los dígitos). El método de acceso para el DNI (método `getDNI`) podría proporcionar el DNI completo (es decir, el NIF, incluyendo la letra), mientras que la letra no es almacenada realmente en el atributo del objeto. Algo similar podría suceder con el **dígito de control de una cuenta bancaria**, que puede no ser almacenado en el objeto, pero sí calculado y devuelto cuando se nos pide el número de cuenta completo.

En otros casos puede interesar disponer de métodos de modificación de un atributo pero a través de un determinado procesamiento previo para por ejemplo poder controlar errores o valores inadecuados. Volviendo al ejemplo del NIF, un método para modificar un DNI (método `setDNI`) podría incluir la letra (NIF completo), de manera que así podría comprobarse si el número de DNI y la letra coinciden (es un NIF válido). En tal caso se almacenará el DNI y en caso contrario se producirá un error de validación (por ejemplo lanzando una excepción). En cualquier caso, el DNI que se almacenara sería solamente el número y no la letra (pues la letra es calculable a partir del número de DNI).

Autoevaluación

Los atributos de una clase suelen ser declarados como public **para facilitar el acceso y la visibilidad de los miembros de la clase. ¿Verdadero o falso?**

- Verdadero.
- Falso.

1.2. Getters y Setters

- Es muy común utilizar la nomenclatura de **getters** y **setters** en lugar de **obtener** y **establecer** para establecer los métodos de acceso a los atributos de las clases.
- La mayoría de los IDEs modernos, tienen desarrolladas implementaciones para generarlos automáticamente y será con los nombres **get y set**.
  - Vease el caso de NetBeans : http://www.fernandezsansalvador.es/blog/netbeans-generar-setters-y-getters-automaticamente
  - Video para NetBeans en YouTube :
  - Eclipse : https://www.fernandezsansalvador.es/blog/eclipse-generar-setters-y-getters-automaticamente
  - NetBeans : https://stackoverflow.com/questions/21074402/generate-getters-and-setters-in-netbeans
- Buenas Prácticas :
  - El nombre del método depende de cada programador, pero para tomar un buen estilo de programación es recomendable anteponer la palabra get o set y el nombre del atributo con la primera letra en mayúscula.
  - Ejemplo de métodos para los atributos **edad** y **nombre** de una clase :
    - **getEdad(), setEdad()**
    - **getNombre(), setNombre()**
  - Hay ocasiones que en los apuntes se utilizan las denominaciones **establecer** y **obtener** en lugar de **set** y **get**, y desde luego no es erroneo ni mucho menos, pero siempre tendremos un problema de internacionalización del código que desarrollemos.
    - En general es interesante desarrollar código para el mayor número posible de personas en todo el mundo, no solo para hispanohablantes.

1.3. Ocultación de métodos.

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

En el ejemplo anterior de objetos que contienen un DNI, será necesario calcular la letra correspondiente a un determinado número de DNI o comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos como métodos protegidos).

Autoevaluación

Dado que los métodos de una clase forman la interfaz de comunicación de esa clase con otras clases, todos los elementos de una clase deben ser siempre declarados como públicos. ¿Verdadero o falso?

- Verdadero.
- Falso.

1.4. Ejercicio resuelto.

Vamos a intentar implementar una clase que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un **DNI español** y que tenga las siguientes características:

- La clase almacenará el número de DNI en un `int`, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: `private int numDNI`.
- Para acceder al DNI se dispondrá de dos métodos **obtener** (`get`), uno que proporcionará el número de DNI (sólo las cifras numéricas) y otro que devolverá el NIF completo (incluida la letra). El formato del método será:

```
* public int obtenerDNI ().
```

```
* public String obtenerNIF ().
```

- Para modificar el DNI se dispondrá de dos métodos establecer (`set`), que permitirán modificar el DNI. Uno en el que habrá que proporcionar el NIF completo (número y letra). Y otro en el que únicamente será necesario proporcionar el DNI (las siete u ocho cifras). Si el DNI/NIF es incorrecto se debería lanzar algún tipo de **excepción**. El formato de los métodos (**sobrecargados**) será:

```
* public void establecer (String nif) throws ...
```

```
* public void establecer (int dni) throws ...
```

- La clase dispondrá de algunos métodos internos privados para calcular la letra de un número de DNI cualquiera, para comprobar si un DNI con su letra es válido, para extraer la letra de un NIF, etc. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos (pertenecientes a la clase). Formato de los métodos:

```
* private static char calcularLetraNIF (int dni).
```

```
* private boolean validarNIF (String nif).
```

```
* private static char extraerLetraNIF (String nif).
```

```
* private static int extraerNumeroNIF (String nif).
```

Para calcular la letra NIF correspondiente a un número de DNI puedes consultar el artículo sobre el NIF de la Wikipedia:

Artículo en la Wikipedia sobre el Número de Identificación Fiscal (NIF).

Solución

La clase tendrá un único atributo de objeto: **el número de DNI**.

private int numDNI;

Está claro que para poder trabajar con los DNI/NIF vas a necesitar implementar el algoritmo para calcular la letra de un número de DNI. Para ello puedes crear un método (que en principio podría ser privado) que realice ese cálculo. Para facilitar la implementación de ese método, crearemos un arrray estático y constante (final) con las letras posibles que puede tener un NIF y en el orden adecuado para la aplicación del algoritmo de cálculo de la letra (algoritmo conocido como **módulo 23**):

private static final String LETRAS_DNI= "TRWAGMYFPDXBNJZSQVHLCKE";

Con esta cadena disponible, es muy sencillo implementar el algoritmo del **módulo 23**:

```
private static char calcularLetraNIF (int dni) {
  char letra;

  // Cálculo de la letra NIF
  letra= LETRAS_DNI.charAt(dni % 23);

   // Devolución de la letra NIF
   return letra;
}
```

Este método estático ha sido definido como privado, aunque también podría haber sido definido como público para que otros objetos pudieran hacer uso de él (típico ejemplo de uso de un método estático).

Para poder manipular adecuadamente la cadena NIF, podemos crear un par de métodos para extraer el número de DNI o la letra a partir de una cadena NIF. Ambos métodos pueden declararse estáticos y privados (aunque no es la única posibilidad):

```
private static char extraerLetraNIF (String nif) {
    char letra=   nif.charAt(nif.length()-1);
    return letra;
}
private static int extraerNumeroNIF (String nif) {
    int numero= Integer.parseInt(nif.substring(0, nif.length()-1));
    return numero;
}
```

Una vez que disponemos de todos estos métodos es bastante sencillo escribir un método de comprobación de la validez de un NIF:

- Extracción del número.
- Extracción de la letra.
- Cálculo de la letra a partir del número.
- Comparación de la letra extraída con la letra calculada.

De manera que el método nos podría quedar:

```
private static boolean validarNIF (String nif) {
    boolean valido= true;   // Suponemos el NIF válido mientras no se encuentre algún fallo
```

```
    char letra_calculada;
    char letra_leida;
    int  dni_leido;
    if (nif == null) {  // El parámetro debe ser un objeto no vacío
        valido= false;
    }
    else if (nif.length()<8 || nif.length()>9) {    // La cadena debe estar entre 8(7+1) y 9(8+1) caracteres
        valido= false;
    }
    else {
        letra_leida= DNI.extraerLetraNIF (nif);    // Extraemos la letra de NIF (letra)
        dni_leido= DNI.extraerNumeroNIF (nif);  // Extraemos el número de DNI (int)
        letra_calculada= DNI.calcularLetraNIF(dni_leido);  // Calculamos la letra de NIF a partir del número extraído
        if (letra_leida == letra_calculada) {   // Comparamos la letra extraída con la calculada
            // Todas las comprobaciones han resultado válidas. El NIF es válido.
            valido= true;
        }
        else {
                valido= false;
        }
    }
        return valido;
}
```

En el código de este método puedes comprobar que se hace uso de los métodos estáticos colocando explícitamente el nombre de la clase:

* DNI.extraerLetraNIF.

* DNI.extraerNumeroNIF.

* DNI.calcularLetraNIF.

En realidad en este caso no habría sido necesario pues estamos en el interior de la clase, pero si finalmente hubiéramos decidido hacer públicos estos métodos, así es como habría que llamarlos desde fuera (usando el nombre de la clase y no el de una instancia).

Y por último tan solo quedarían por implementar los métodos públicos (la interfaz):

- Los dos métodos **obtener** (get). Obtener el NIF (String) u obtener el DNI (int).
- Los dos métodos **establecer** (set). A partir de un int y a partir de un String.

En el primer caso habrá que devolver información añadiéndole (si es necesario) información adicional calculada, y en el segundo habrá que realizar una serie de comprobaciones antes de proceder a almacenar el nuevo valor de DNI/NIF.

El código de los métodos **obtener** podría quedar así;

```
 public String obtenerNIF () {
   // Variables locales
    String cadenaNIF;   // NIF con letra para devolver
    char letraNIF;      // Letra del número de NIF calculado
    // Cálculo de la letra del NIF
    letraNIF= DNI.calcularLetraNIF (numDNI);
    // Construcción de la cadena del DNI: número + letra
    cadenaNIF= Integer.toString(numDNI) + String.valueOf(letraNIF);
    // Devolución del resultado
    return cadenaNIF;
}
public int obtenerDNI () {
    return numDNI;
}
```

En el caso de los métodos establecer (**método establecer sobrecargado**) podemos lanzar una excepción básica con un mensaje de error de "NIF/DNI inválido" para que la reciba el objeto que utilice este método. De esta manera podría controlarse el error de un posible establecimiento de valores de NIF/DNI inválido.

El código de los métodos **establecer** podría quedar así;

```
public void establecer (String nif) throws Exception {
    if (validarNIF (nif)) { // Valor válido: lo almacenamos
        this.numDNI= DNI.extraerNumeroNIF(nif);
    }
    else { // Valor inválido: lanzamos una excepción
        throw new Exception ("NIF inválido: " + nif);
    }
```

```
}
public void establecer (int dni) throws Exception {
   // Comprobación de rangos
   if (dni>999999 && dni<99999999) {
      this.numDNI= dni; // Valor válido: lo almacenamos
   }
   else { // Valor inválido: lanzamos una excepción
      throw new Exception ("DNI inválido: " + String.valueOf(dni));
   }
}
```

El código completo de la clase DNI podría ser:

```
/**---------------------------------------------------------------
 * Clase DNI
 ---------------------------------------------------------------*/

   public class DNI {
      // Atributos estáticos

      // Cadena con las letras posibles del DNI ordenados para el cálculo de DNI
      private static final String LETRAS_DNI= "TRWAGMYFPDXBNJZSQVHLCKE";

      // Atributos de objeto
      private int numDNI;

      // Métodos

      public String obtenerNIF () {
```

```
    // Variables locales
    String cadenaNIF;   // NIF con letra para devolver
    char letraNIF;      // Letra del número de NIF calculado

    // Cálculo de la letra del NIF
    letraNIF= calcularLetraNIF (numDNI);

    // Construcción de la cadena del DNI: número + letra
    cadenaNIF= Integer.toString(numDNI) + String.valueOf(letraNIF);

    // Devolución del resultado
    return cadenaNIF;
}

public int obtenerDNI () {
    return numDNI;
}

public void establecer (String nif) throws Exception {
    if (DNI.validarNIF (nif)) { // Valor válido: lo almacenamos
        this.numDNI= DNI.extraerNumeroNIF(nif);
    }
    else { // Valor inválido: lanzamos una excepción
        throw new Exception ("NIF inválido: " + nif);
    }

}

public void establecer (int dni) throws Exception {

    // Comprobación de rangos
```

```
    if (dni>999999 && dni<99999999) {
        this.numDNI= dni; // Valor válido: lo almacenamos
    }
    else { // Valor inválido: lanzamos una excepción
        throw new Exception ("DNI inválido: " + String.valueOf(dni));
    }
}

private static char calcularLetraNIF (int dni) {
    char letra;

    // Cálculo de la letra NIF
    letra= LETRAS_DNI.charAt(dni % 23);

    // Devolución de la letra NIF
    return letra;
}

private static char extraerLetraNIF (String nif) {
    char letra=   nif.charAt(nif.length()-1);
    return letra;
}

private static int extraerNumeroNIF (String nif) {
    int numero= Integer.parseInt(nif.substring(0, nif.length()-1));
    return numero;
}

private static boolean validarNIF (String nif) {
    boolean valido= true;   // Suponemos el NIF válido mientras no se encuentre algún fallo
    char letra_calculada;
```

```
        char letra_leida;
        int  dni_leido;

        if (nif == null) {  // El parámetro debe ser un objeto no vacío
            valido= false;
        }
        else if (nif.length()<8 || nif.length()>9) {    // La cadena debe estar entre 8(7+1) y 9(8+1) caracteres
            valido= false;
        }
        else {
            letra_leida= DNI.extraerLetraNIF (nif);    // Extraemos la letra de NIF (letra)
            dni_leido= DNI.extraerNumeroNIF (nif);  // Extraemos el número de DNI (int)
            letra_calculada= DNI.calcularLetraNIF(dni_leido);  // Calculamos la letra de NIF a partir del
número extraído
            if (letra_leida == letra_calculada) {   // Comparamos la letra extraída con la calculada
                // Todas las comprobaciones han resultado válidas. El NIF es válido.
                valido= true;
            }
            else {
                    valido= false;
            }
        }

        return valido;
    }
}
```