Encapsulamiento

vs. ocultamiento de información

Encapsulamiento

Encapsular: acción de poner juntas ciertas cosas dado que hay una razón para ello.

En la **POO** aquellas cosas serán los datos y los métodos que operan sobre esos datos.

Mediante el encapsulamiento es que creamos las entidades que deseamos manejar en nuestros sistemas.

El encapsulamiento tiene dos sentidos:

- *Especialización*, ya que el propio objeto es aquél que sabrá cómo manejar los datos que contiene. Es por eso que el mejor lugar para hacerlo, es dentro del mismo objeto (con los métodos asociados a los datos dados).
- *Completitud*, ya que nos permite descansar en que la abstracción construida representa a la entidad, y a aquella responsabilidad que tendrá asignada dentro de nuestro sistema.

Sin embargo, nos interesa mucho más la interfaz pública de los objetos que sus representaciones internas y datos asociados.

Se define como **interfaz pública** de una clase al conjunto de responsabilidades que los objetos de esa clase estarán brindando, desde el punto de vista externo a la misma.

Encapsulamiento y ocultamiento de información

El encapsulamiento nos permite armar nuestros objetos, juntando aquellas responsabilidades que necesitan dentro del sistema con los datos para poder hacerlo.

Ocultar la información es utilizar las técnicas que nos brinda el lenguaje para abstraer a nuestros módulos cliente de los detalles de implementación.

El encapsulamiento nos habla de límites: "esta responsabilidad es mía, la llevaré a cabo con estos datos".

El ocultamiento de información nos indica buenas prácticas de programación: "necesito de este otro objeto, pero no me importa cómo resuelva sus responsabilidades mientras lo haga por mí".

Un concepto no implica el otro, ni viceversa.

Reglas

Encapsulamiento

1. Ubicar los datos y las operaciones que trabajan sobre esos datos en la misma clase.
2. Utilizar diseño guiado por las responsabilidades para determinar la agrupación de datos y operaciones dentro de clases.

Ocultamiento de información

1. No exponer atributos.
2. No exponer diferencia entre atributos propiamente dichos y atributos calculados.
3. No exponer la estructura interna de una clase.
4. No exponer detalles de implementación de una clase.

Getters y Setters

Dado que bajo ciertas circunstancias es necesario instruir a un objeto la necesidad de cambiar algún valor interno del mismo, es importante proporcionar un mecanismo para poder hacerlo sin romper el encapsulamiento (en términos rudimentarios, sin hacer públicos algunos -o todos- sus miembros).

Es por ello que existen dos tipos de métodos muy simples que se denominan **accesores**, y serán los que nos permitan acceder a esos miembros privados *de una manera controlada por el diseñador de la clase*.

Los getters nos servirán para obtener el valor de un miembro, y los setters para establecerlo. Su raíz en el idioma inglés hace juego con las palabras "get" y "set" (obtener y establecer).

Getters y Setters

Típicamente, un getter tiene esta estructura:

```
public Integer getEdad() {
  return this.edad;
}
```

Un setter, en cambio, responde a la siguiente estructura:

```
public void setEdad(Integer edad) {
  this.edad = edad;
}
```

Cuando estamos a cargo de la definición de una clase, definimos nosotros mismos el nivel de complejidad que estos métodos encapsulan, y qué tan directamente permitimos que un agente externo acceda a los miembros de la misma.

> **Nota:** Para utilizar algunas tecnologías Java, es necesario escribir setters y getters de todos los atributos *persistentes* de una clase. Ya llegaremos a eso, pero por el momento es interesante saberlo.

Ejemplo: ocultamiento

Suponga que se tiene una clase MyDate que incluye los atributos *day, month, year.*
Una implementación simple permite el acceso directo a los atributos de los datos, por ejemplo:

```
public class MyDate {
     public int day;
     public int month;
     public int year;
}
```

El código del cliente accede a los atributos y comete errores:

```
d.day=32;  // día inválido

d.month = 2;
d.day=30; // mal

d.day = d.day + 1;   //no se realiza la validación para pasar al siguiente día.
```

Ejemplo: ocultamiento

Para resolver este problema se ocultan los atributos de los datos utilizando el modificador de acceso **`private`** y proveyendo métodos de asignación y recuperación (setters y getters).

```
public class MyDate{
      private int day;
      private int month;
      private int year;

      public boolean setDate(int d){ …}
      public boolean setMonth(int m){ … }
      public boolean setYear(int y){ … }
      public int getDay(){ ... }
      public int getMonth() { … }
      public int getYear() { … }
}

MyDate fecha = new MyDate();
fecha.setDay(32);  // devolverá false.
fecha.setMonth(2);
fecha.setDay(30);  //devolverá falso.
fecha.setDay(fecha.getDay + 1);  // devolverá falso o  incrementará el día y el mes en caso de que el día ya se encuentre en el valor máximo para ese mes.
```

Ejemplo: encapsulamiento

El programador de la clase MyDate ha decidido cambiar la implementación interna, reemplazando los atributos ***day, month, year*** por la cantidad de días que transcurren desde el comienzo de una época, logrando de esta forma simplificar las comparaciones y operaciones entre intervalos de tiempo.

```
public class MyDate{

      private long date;

      public boolean setDate(int d){ …}
      public boolean setMonth(int m){ … }
      public boolean setYear(int y){ … }
      public int getDay(){ ... }
      public int getMonth() { … }
      public int getYear() { … }
}
```

Dado que el programador encapsuló los atributos detrás de una interfaz pública, puede hacer estos cambios sin afectar el código del cliente.