Que es delegación en POO

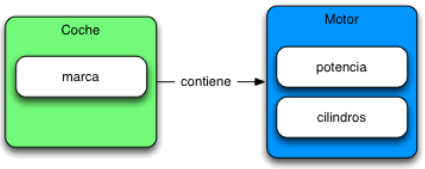
El concepto de delegación (Delegation) es más amplio que lo que aquí se describe pero basta como introducción. Profundizaremos en este concepto en otro boletín.

Se trata de una técnica en la que un objeto permite mostrar cierto método al exterior, pero internamente la implementación de dicho método consiste en delegar en otro método de un objeto de otra clase. Lo entenderemos enseguida con un ejemplo.

Como veremos, la necesidad de delegación aparece cuando en una clase *A,* hay composición de clases y se quiere ocultar la existencia de la composición a los usuarios de la clase A. A menudo, se quiere ocultar la composición para hacer más sencillo el manejo de una clase y para hacerla más resistente a los cambios

Para explicar la delegación vamos a seguir este artículo, que por su sencillez nos vale como introducción.
https://www.arquitecturajava.com/eclipse-y-el-concepto-de-delegacion/

Ejemplo: Tenemos dos clases: *Coche* y *Motor*. Hay composición entre Coche y Motor. El usuario de la clase Coche es la clase Unidad5 y también se ve obligado a manejar la clase Motor




```
class Coche {

    private String marca;
    private Motor motor;

    public Motor getMotor() {
        return motor;
    }

    public void setMotor(Motor motor) {
        this.motor = motor;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}

class Motor {

    private int potencia;
    private int cilindros;

    public int getPotencia() {
        return potencia;
    }

    public void setPotencia(int potencia) {
        this.potencia = potencia;
    }
```

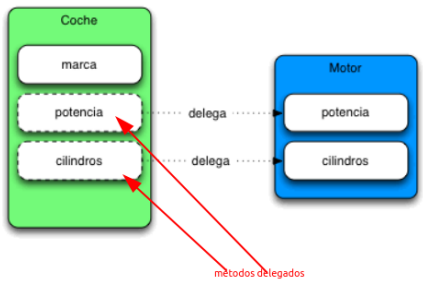
```
    public int getCilindros() {
        return cilindros;
    }

    public void setCilindros(int cilindros) {
        this.cilindros = cilindros;
    }
}

class Unidad5 {

    public static void main(String[] args) {
        Coche c = new Coche();
        c.setMarca("toyota");
        Motor m = new Motor();
        m.setCilindros(6);
        m.setPotencia(100);
        c.setMotor(m);

        System.out.println(c.getMotor().getPotencia());

    }

}
```

El programador de *Unidad5* tiene que saber que hay una clase motor que forma parte de la clase Coche, y por tanto, debe conocer su constructor y sus métodos. ¿Se podría evitar esto?

Observa que *Unidad5* tiene que crear objetos de la clase motor y lo engorroso que le resulta a *Unidad5* acceder a la potencia del motor.

```
c.getMotor().getPotencia()
```

Una forma de frenar esta complicación es crear métodos intermediarios o representantes en la clase Coche. Estos métodos luego no hacen nada directamente, simplemente "invocan" a los de la clase Motor, pero conseguimos simplificar la vida al programador de *Unidad5*. A estos métodos representantes de otros de la clase Coche les llamamos *métodos delegados* porque simplemente representan a otros métodos que realmente hacen el trabajo. Y como buenos "jefes" o "delegados" delegan el trabajo duro en terceros.



Añadimos los métodos delegados a mano o con IDE. Para que aparezca la opción delegate methods hay que estar situado en la clase Coche, que es donde tiene sentido añadir métodos delegados.
Y simplificamos el acceso a la potencia

```
class Coche {

    private String marca;
    private Motor motor;

    public int getPotencia() {//muchas veces coincide el nombre entre delegado y representado
        return motor.getPotencia();
```

```
    }

    public void setPotencia(int potencia) {
        motor.setPotencia(potencia);
    }

    public int getCilindros() {
        return motor.getCilindros();
    }

    public void setCilindros(int cilindros) {
        motor.setCilindros(cilindros);
    }

    public Motor getMotor() {
        return motor;
    }

    public void setMotor(Motor motor) {
        this.motor = motor;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}

class Motor {

    private int potencia;
    private int cilindros;

    public int getPotencia() {
        return potencia;
    }

    public void setPotencia(int potencia) {
        this.potencia = potencia;
    }

    public int getCilindros() {
        return cilindros;
    }

    public void setCilindros(int cilindros) {
        this.cilindros = cilindros;
    }
}

class Unidad5 {

    public static void main(String[] args) {
        Coche c = new Coche();
        c.setMarca("toyota");
        Motor m = new Motor();
        m.setCilindros(6);
        m.setPotencia(100);
        c.setMotor(m);

        System.out.println(c.getPotencia());//ahora más fácil

    }

}
```

Ejercicio U5_B2_E1: mejora otro poco el código anterior de forma que el programador no tenga que ser consciente de la clase motor de forma que sea la propia clase Coche la que cree el objeto motor de forma que Unidad5 quede así

```
class Unidad5 {

    public static void main(String[] args) {
```

```
        Coche c = new Coche("toyota",100,6);
        System.out.println(c.getPotencia());

    }

}
```

Reflexion: ¿Usaste composición y delegación en la clase SieteYMedia en la tarea anterior?: ¡Sí!. La clase interfaceConsola no sabe que hay una clase Baraja. La clase Baraja compone una parte de la clase SieteYMedia ya que tiene un array de tipo Baraja. Por otro lado se usa delegación , ya que cuando interfaceConsola pide una carta a la clase SieteYMedia, realmente SieteYMedia delega en la clase Baraja para obtener una nueva carta del mazo.

Acoplamiento de clases
Uno de los objetivos más importantes del diseño orientado a objetos es conseguir entre las clases un bajo acoplamiento.

El acoplamiento entre clases es una medida de la interconexión o dependencia entre esas clases. Un acoplamiento fuerte significa que las clases relacionadas necesitan saber detalles internos unas de otras, los cambios se propagan por el sistema y el sistema es posiblemente más difícil de entender. Por ello deberemos siempre intentar que nuestras clases tengan entre sí un acoplamiento bajo. Cuantas menos cosas conozca la clase A sobre la clase B, menor será su acoplamiento.

Porqué es importante un bajo acoplamiento
Un bajo acoplamiento permite:

- Entender una clase sin leer otras.
- Cambiar una clase sin afectar a otras
- Mejora la mantenibilidad del código

Como conseguir bajo acoplamiento
Hay una serie de situaciones y soluciones pero ahora nos centramos en dos:

1. Los atributos de una clase deberán ser privados y la única forma de acceder a ellos debe ser a través de los métodos getter y setter. Es decir el principio de ocultación favorece el bajo acoplamiento
2. Utilizar delegación en las clases compuestas

La delegación y acoplamiento

Una clase Compuesta es más compleja y por tanto también su uso ya que el usuario tiene que ser consciente de esta estructura de composición, recuerda en el ejercicio del paquete ordenador la siguiente instrucción de miAplicacion.Principal

```
System.out.println(o.getNSerie() + ", memoria " + o.getMemoria().getCapacidad() + "GB, disco " + o.getDisco().getTipo());
```

esto delata que el programador de *miAplicacion.Principal* tiene que ser consciente de que existe una clase Memoria ya que getMemoria() devuelve un objeto Memoria, y debemos conocer detalles sobre esta clase como que tiene un método getCapacidad()

La delegación puede disminuir el acoplamiento entre una clase compuesta como Ordenador y otra clase que la usa como Principal ya que los métodos delegados generados aíslan a Principal de detalles internos de la composición de la clase Ordenador.

Ejercicio U5_B2_E2: Mejorar el ejemplo de Ordenadores usando principio de ocultación y disminuyendo acoplamiento con métodos delegados.

Con la solución del ejercicio de ordenadores del boletín pasado que se incluye aquí más abajo, se pide mejorar este código con 3 cuestiones:

1. Usar import para hacer el código más legible, aunque como ya indicamos lo hicimos a propósito por motivos didácticos para ser muy conscientes de que las clases pertenecen a paquetes y para ver por otro lado que import es "útil".
2. Añade a los objetos Memoria y Disco una versión de constructor para que haga un objeto copia. Similar en efecto al que tiene la clase Punto y usar principio de ocultación
3. genera los métodos delegados necesarios para que la clase Principal no se entere de la composición de clases en el paquete ordenador y disminuir acoplamiento

Código a cambiar: en la carpeta de soluciones hay un OrdenadoresSinDelegacion.zip o más incómodo también puedes coger el código de la solución del boletín 02A
también se incluye a continuación el código si prefieres copiar y pegar

```
//Disco.java
package ordenador;
public class Disco {
    private float capacidad;
    private String tipo;

    Disco(float capacidad, String tipo) {
        this.capacidad = capacidad;
        this.tipo = tipo;
    }
     //sólo hace falta public este método porque lo usa principal desde otro paquete
    public String getTipo() {
        return tipo;
    }

   float getCapacidad() {
        return capacidad;
    }

}


//Memoria.java
package ordenador;
public class Memoria {
    private int capacidad;
    private String tipo;
    private int velocidad;

    Memoria(int capacidad, String tipo, int velocidad) {

        this.capacidad = capacidad;
        this.tipo = tipo;
        this.velocidad = velocidad;


    }
    //sólo hace falta public este porque lo usa principal desde otro paquete
    public int getCapacidad(){
        return this.capacidad;
    }
    String getTipo(){
        return tipo;
```

```
    }
    int getVelocidad(){
        return velocidad;
    }


}


//Procesador.java
package ordenador;
class Procesador{
    private String modelo;
    private float velocidad;

    Procesador(String modelo, float velocidad) {
        this.modelo = modelo;
        this.velocidad = velocidad;
    }

}



//Ordenador.java
package ordenador;

public class Ordenador {
    private String nSerie;
    private Procesador p;
    private Memoria m;
    private Disco d;
    private int pvp;

    public Ordenador(String nSerie,int capacidadMemoria,String tipMemoria,int velocidadMemoria,float capacidadDisco,String tipoDisco,
String tipoProcesador, float velocidadProcesador,int precio) {
        this.nSerie=nSerie;
        this.m= new Memoria(capacidadMemoria,tipMemoria,velocidadMemoria);
        this.d=new Disco(capacidadDisco, tipoDisco);
        this.p= new Procesador(tipoProcesador,velocidadProcesador);
    }
    public String getNSerie(){
        //los Strings son inmutables, No hay problemas con principio de ocultación si el atributo es private
        return nSerie;
    }
    public Memoria getMemoria(){
        Memoria copia= new Memoria(m.getCapacidad(),m.getTipo(),m.getVelocidad());
        return copia;
    }
    public Disco getDisco(){
        Disco copia=new Disco(d.getCapacidad(),d.getTipo());
        return copia;
    }


}
```