# Guía práctica de estudio 05: Abstracción y Encapsulamiento


***Elaborado por:***
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

***Autorizado por:***
M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 05:

# Abstracción y Encapsulamiento

## Objetivo:

Aplicar el concepto de abstracción para el diseño de clases que integran una solución, utilizando el encapsulamiento para proteger la información y ocultar la implementación.

## Introducción

La **abstracción** es la habilidad de ignorar los detalles de las partes para enfocar la atención en un nivel más alto de un problema.

El **encapsulamiento** sucede cuando algo es envuelto en una capa protectora. Cuando el **encapsulamiento** se aplica a los objetos, significa que los datos del objeto están protegidos, **"ocultos"** dentro del objeto. Con los datos ocultos, ¿cómo puede el resto del programa acceder a ellos? (El acceso a los datos de un objeto se refiere a leerlos o modificarlos.) El resto del programa no puede acceder de manera directa a los datos de un objeto; lo tiene que hacer con ayuda de los métodos del objeto.

En el supuesto de que los métodos de un objeto estén bien escritos, los métodos aseguran que se pueda acceder a los datos de manera adecuada. Al hecho de empaquetar o proteger los datos o atributos con los métodos se denomina **encapsulamiento**.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Abstracción

La **abstracción** es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. La **abstracción** posee diversos grados o **niveles de abstracción**, los cuales ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real.

En el análisis hay que concentrarse en **¿Qué hace?** y no en *¿Cómo lo hace?*

El principio de **abstracción** es más fácil de entender con una analogía del mundo real, por ejemplo, la televisión. Todos estamos familiarizados con sus características y su manejo manual o con el control remoto (encender, apagar, subir volumen, etc.) incluso sabemos cómo conectarlo con otros aparatos externos como altavoces o reproductores, o al internet. Sin embargo, no todos sabemos cómo funciona internamente, es decir, como recibe la señal, como la traduce y la visualiza en la pantalla, etc.

Normalmente sucederá que no sepamos cómo funciona el aparato de televisión pero si sabemos cómo utilizarlo. Esta característica se debe a que la televisión separa claramente su implementación interna de su **interfaz** externa. Interactuamos con la televisión a través de su **interfaz**: los botones de encendido, cambio de canal, volumen, etc. no conocemos el tipo de tecnología que utiliza, el método de generar la imagen en la pantalla o como funciona internamente, es decir, su implementación, ya que ello no afecta a su **interfaz**.

## Encapsulamiento

La **encapsulación** o **encapsulamiento** significa reunir en una cierta estructura a todos los elementos que a un cierto **nivel de abstracción** se pueden considerar pertenecientes a una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.

El **encapsulamiento** oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior por lo que se denomina también **ocultación de datos**. Un objeto tiene que presentar **"una cara"** al mundo exterior de modo que se puedan iniciar sus operaciones.

Por ejemplo, la televisión tiene un conjunto de botones para encender, apagar y/o cambiar canal. Una lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura, el nivel de agua, etc. Los botones de la televisión y la lavadora constituyen la comunicación con el mundo interior, es decir son las **interfaces**.

La **interfaz** de una clase representa un **"contrato"** de prestación de servicios entre ella y los demás componentes del sistema. De este modo, los clientes solo necesitan conocer los servicios que este ofrece y no como están implementados internamente.

Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella.

## Reglas de visibilidad

Las reglas de visibilidad complementan o refinan el concepto de **encapsulamiento**. Los diferentes niveles de visibilidad dependen del lenguaje de programación con el que se trabaje, pero en general siguen el modelo de C++, estos niveles de visibilidad son:

- El nivel más fuerte se denomina nivel **"privado"**; la sección privada de una clase es totalmente invisible para otras clases, solo miembros de la misma clase pueden acceder a atributos localizados en la sección privada.
- Es posible aliviar el nivel de ocultamiento situando algunos atributos en la sección **"protegida"** de la clase. Estos atributos son visibles tanto para la misma clase como para las clases derivadas de la clase. Para las restantes clases permanecen invisibles.
- El nivel más débil se obtiene situando los atributos en la sección **"pública"** de la clase con lo cual se hacen visibles a todas las clases.

Los **atributos privados** están contenidos en el interior de la clase ocultos a cualquier otra clase. Ya que los atributos están **encapsulados** dentro de una clase, se necesitará definir cuáles son las clases que tienen acceso a visualizar y cambiar los atributos. Esta característica se conoce como **visibilidad de los atributos**. En general se recomienda visibilidad **privada** o **protegida** para los atributos.

## Modificadores de acceso

Los modificadores de acceso se utilizan para definir la visibilidad de los miembros de una clase (atributos y métodos) y de la propia clase. En Java existen tres modificadores de acceso:

- **public**
- **protected**
- **private**

Sin embargo, existen cuatro niveles de acceso. Cuando no se especifica ninguno de los tres modificadores anteriores se tiene el nivel de acceso por defecto, que es el nivel de paquete.

La sintaxis para los modificadores de acceso es simplemente anteponerlos a la declaración de atributos y métodos.

***Modificador*** *tipoDato nombreVariable;*

***Modificador*** *tipoDato nombreMetodo(parámetros…)*

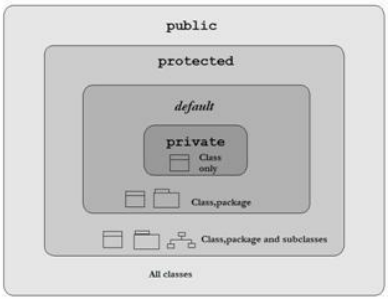
Ejemplo:

***private*** *String nombre;*

***public*** *int operacion(int a, int b){*

A continuación se muestra el acceso permitido para cada modificador.

| Visibilidad | Public | Protected | Default | Private |
|---|---|---|---|---|
| Desde la misma Clase | SI | SI | SI | SI |
| Desde cualquier Clase del mismo Paquete | SI | SI | SI | NO |
| Desde una SubClase del mismo Paquete | SI | SI | SI | NO |
| Desde una SubClase fuera del mismo Paquete | SI | SI, a través de la herencia | NO | NO |
| Desde cualquier Clase fuera del Paquete | SI | NO | NO | NO |




Un principio fundamental en la programación orientada a objetos es la **ocultación de la información**, que significa que determinados datos del interior de una clase no pueden ser accedidos por funciones externas a la clase, esto sugiere que solamente la información sobre lo que puede hacer una clase debe estar visible desde el exterior, pero no cómo lo hace. Esto tiene una gran ventaja: si ninguna otra clase conoce cómo está almacenada la información entonces se puede cambiar fácilmente la forma de almacenarla sin afectar otras clases.

## Acceso a miembros

Podemos reforzar la separación del qué hacer del cómo hacerlo, declarando los campos como **privados** y usando un **método de acceso** para acceder a ellos.

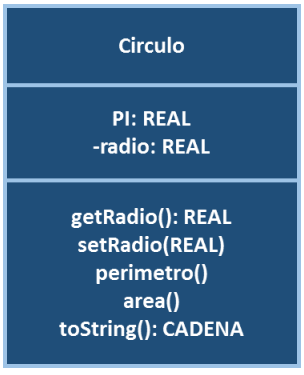
Los **métodos de acceso** son el medio de acceder a los atributos privados del objeto. Son métodos públicos del objeto y pueden ser:

- **Métodos modificadores**: llamamos métodos modificadores a aquellos métodos que dan lugar a un cambio en el valor de uno o varios de los atributos del objeto.
- **Métodos consultores u observadores**: son métodos que devuelven información sobre el contenido de los atributos del objeto sin modificar los valores de estos atributos.

Cuando se crea una clase es frecuente que lo primero que se haga sea establecer métodos para consultar sus atributos y estos métodos suelen ir precedidos del prefijo **get** (getNombre, getValor, etc.) por lo que muchas veces se alude coloquialmente a ellos como **"métodos get"** o **"getters"**. Los **métodos get** son un tipo de **métodos consultores**, porque solo consultan y devuelven el valor de los atributos de un objeto.

Se suele proceder de igual forma con métodos que permitan establecer los valores de los atributos. Estos métodos suelen ir precedidos del prefijo **set** (setNombre, setValor, etc.) por lo que muchas veces se alude coloquialmente a ellos como **"métodos set"** o **"setters"**. Los **métodos set** son un tipo de **métodos modificadores**, porque cambian el valor de los atributos de un objeto.

Ejemplo:



```java
public class Circulo {

    static float PI = 3.14159f;
    private float radio;

    public float getRadio() {
        return radio;
    }
    public void setRadio(float radio) {
        this.radio = radio;
    }
    public float perimetro(){
        return 2 * PI * radio;
    }
    public float area(){
        return PI * radio * radio;
    }
    public String toString() {
        return "Circulo [radio=" + radio + "]";
    }
}
```

```java
public class PruebaFiguras {

    public static void main(String[] args) {
        Circulo cir=new Circulo();
        cir.setRadio(7.2f);
        System.out.println("El area es " + cir.area());
    }

}
```

Parece ser que proporcionar herramientas para establecer y obtener es esencialmente lo mismo que hacer las variables de instancia *public*. Si una variable de instancia se declara como *public*, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como *private*, un método set *public* evidentemente permite a otros métodos el acceso a la variable, pero el método set puede controlar la manera en que el cliente puede tener acceso a la variable.

Para el mismo ejemplo, se puede validar que el radio que se le asigna a un círculo nunca sea negativo modificando su setter:

```java
public void setRadio(float radio) {
    if(radio < 0){
        radio = 0;
    }
    this.radio = radio;
}
```

Entonces, aunque los métodos **set** y **get** proporcionan acceso a los datos privados, el programador restringe su acceso mediante la implementación de los métodos.

## Composición

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como **composición** y algunas veces como relación **"tiene un".**

El concepto de **composición** no fue creado para la programación; suele usarse a menudo para objetos complejos en el mundo real. Toda criatura viviente y la mayor parte de los

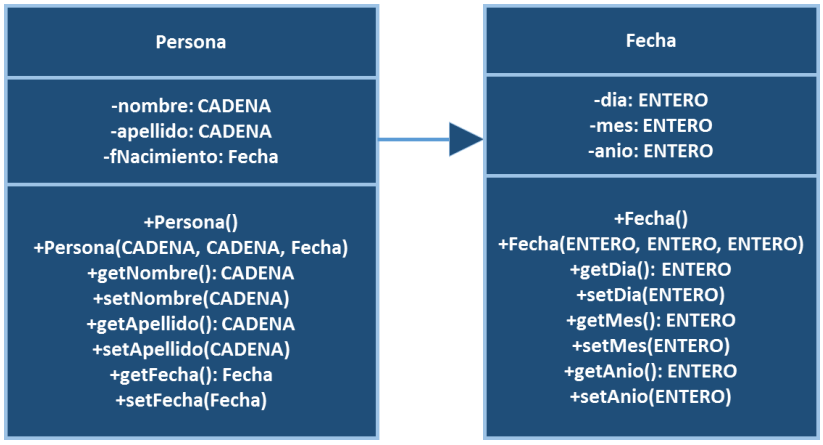
productos manufacturados están constituidos por partes. A menudo, cada parte es un subsistema que está integrado por su propio conjunto de sub-partes. Junto, todo el sistema forma una jerarquía de composición.

Por ejemplo, el cuerpo humano está compuesto por varios órganos: cerebro, corazón, estómago, huesos, músculos, etc. A su vez, cada uno de estos órganos está compuesto por muchas células, y cada una de estas células está compuesta por muchos orgánulos, como el núcleo (el "cerebro" de una célula) y las mitocondrias (los "músculos" de una célula). Cada orgánulo está compuesto por muchas moléculas. Y finalmente, cada molécula orgánica está compuesta por muchos átomos.

En una jerarquía de **composición** (así como en una jerarquía de agregación), la relación entre una clase contenedora y una de sus clases parte se denomina relación tiene-un. Por ejemplo, cada cuerpo humano tiene un cerebro y tiene un corazón.

La **composición** es una forma de **reutilización de software**, en donde una clase tiene como miembros referencias a objetos de otras clases.

Ejemplo:

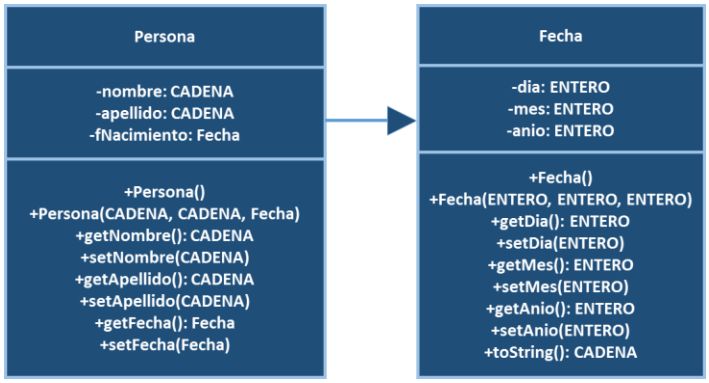


```
 class Fecha {
ivate int dia;
ivate int mes;
ivate int anio;

blic Fecha() {


blic Fecha(int dia, int mes, int anio) {
  setDia(dia);
  setMes(mes);
  setAnio(anio);

blic int getDia() {
  return dia;

blic void setDia(int dia) {
  if(dia > 0 && dia < 32){
      this.dia = dia;
  } else {
      System.out.println("Día no valido");
  }

blic int getMes() {
  return mes;

blic void setMes(int mes) {
  if(mes>0 && mes < 13){
      this.mes = mes;
  } else {
      System.out.println("Mes no valido");
  }

blic int getAnio() {
  return anio;

blic void setAnio(int anio) {
  if(anio>0){
      this.anio = anio;
  } else {
      System.out.println("El año no puede ser negativo");
  }
```

```
public class Persona {
    private String nombre;
    private String apellido;
    private Fecha fNacimiento;

    public Persona() {
    }
    public Persona(String nombre, String apellido, Fecha fNaci
        this.nombre = nombre;
        this.apellido = apellido;
        this.fNacimiento = fNacimiento;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public Fecha getFNacimiento() {
        return fNacimiento;
    }
    public void setFNacimiento(Fecha fNacimiento) {
        this.fNacimiento = fNacimiento;
    }
}
```

```
public class PruebaPersona {

    public static void main(String[] args) {
        Persona per1 = new Persona();
        Fecha nac = new Fecha();
        per1.setNombre("Juan");
        per1.setApellido("Perez");
        nac.setDia(15);
        nac.setMes(8);
        nac.setAnio(1950);
        per1.setFNacimiento(nac);
        System.out.println("Nombre : " + per1.getNombre());
        System.out.println("Apellido : " + per1.getApellido());
        System.out.println("Fecha Nacimiento : " + per1.getFNacimiento().getDia() +
                "/"+per1.getFNacimiento().getMes()+"/"+per1.getFNacimiento().getAnio());
    }

}
```

Sin embargo, se pueden modificar las clases para que no tenga que instanciarse la clase Fecha fuera de la clase Persona, quedando así un mejor diseño de clases.




```java
public class Fecha {
    private int dia;
    private int mes;
    private int anio;

    public Fecha() {

    }
    public Fecha(int dia, int mes, int anio) {
        setDia(dia);
        setMes(mes);
        setAnio(anio);
    }
    public int getDia() {
        return dia;
    }
    public void setDia(int dia) {
        if(dia > 0 && dia < 32){
            this.dia = dia;
        } else {
            System.out.println("Día no valido");
        }
    }
    public int getMes() {
        return mes;
    }
    public void setMes(int mes) {
        if(mes>0 && mes < 13){
            this.mes = mes;
        } else {
            System.out.println("Mes no valido");
        }
    }
    public int getAnio() {
        return anio;
    }
    public void setAnio(int anio) {
        if(anio>0){
            this.anio = anio;
        } else {
            System.out.println("El año no puede ser negativo");
        }
    }
    public String toString() {
        return dia + "/" + mes + "/" + anio;
    }

}
```

```java
public class Persona {
    private String nombre;
    private String apellido;
    private Fecha fNacimiento;

    public Persona() {
        fNacimiento = new Fecha();
    }
    public Persona(String nombre, String apellido, int fDia, int fMes, int fAnio) {
        this.nombre = nombre;
        this.apellido = apellido;
        fNacimiento.setDia(fDia);
        fNacimiento.setMes(fMes);
        fNacimiento.setAnio(fAnio);
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public void setFNacimiento(int dia, int mes, int anio){
        fNacimiento.setDia(dia);
        fNacimiento.setMes(mes);
        fNacimiento.setAnio(anio);
    }
    public Fecha getFNacimiento(){
        return fNacimiento;
    }
}
```

```java
public class PruebaPersona {

    public static void main(String[] args) {
        Persona per1 = new Persona();
        per1.setNombre("Juan");
        per1.setApellido("Perez");
        per1.setFNacimiento(15, 8, 1950);
        System.out.println("Nombre : " + per1.getNombre());
        System.out.println("Apellido : " + per1.getApellido());
        System.out.println("Fecha Nacimiento : " + per1.getFNacimiento());
    }

}
```

## Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Joyanes, Luis*
***Fundamentos de programación. Algoritmos, estructuras de datos y objetos.***
*Cuarta Edición*
*México*
*Mc Graw Hill, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*