**4. Encapsulamiento**

**4.1 Introducción**

El encapsulamiento es uno de los fundamentos de POO. Se refiere a la agrupación de datos con los métodos que operan sobre esos datos. El encapsulamiento se usa para ocultar las variables o el estado de un objeto de datos estructurados dentro de una clase, evitando el acceso directo de las partes no autorizadas a ellos.

Los métodos públicamente accesibles generalmente se proporcionan en la clase (los denominados getters y setters) para acceder a las variables, y otras clases de clientes llaman a estos métodos para recuperar y modificar los valores dentro del objeto.

Se dice que el objeto "publica sus interfaces". Otros objetos se adhieren a estas interfaces para usar el objeto sin tener que preocuparse por la forma en que el objeto lo logra. La idea es "no me digas cómo lo haces, solo hazlo".

**4.2 Ventajas**

La principal ventaja del encapsulamiento es la seguridad de los datos.

 Los beneficios del encapsulamiento incluyen:

* Proteger un objeto del acceso no deseado por parte de los clientes.
* Permitir el acceso a un nivel sin revelar los detalles complejos por debajo de ese nivel.
* Reducir los errores humanos.
* Simplificar el mantenimiento de la aplicación.
* Hacer que la aplicación sea más fácil de entender.

​Para un mejor encapsulamiento, los datos de los objetos casi siempre deben estar restringidos a privados o protegidos. Si elige establecer el nivel de acceso en público, asegúrese de comprender las ramificaciones de la elección.

**4.3 Modificadores de nivel de acceso**

Los modificadores de nivel de acceso determinan si otras clases pueden usar un objeto particular o invocar un método particular. Hay dos niveles de control de acceso:

* En el nivel superior: ***public*** (*público*) o ***package-private*** (*sin modificador explícito*).
* En el nivel de miembro: ***public*** (*público*), ***private*** (*privado*), ***protected*** (*protegido*) o ***package-private*** (*sin modificador explícito*).

**4.4 Visibilidad**

* ***public*** *(público):* accesible por cualquier clase.
* ***protected*** *(protegido):* accesible por la clase, todas las clases heredadas y las clases del paquete actual (editado).
* ***private*** *(privado):* accesible solo por la clase.
* ***Sin modificador:*** accesible para todas las clases del paquete actual.

**4.5 Implementación**

public class Encapsulamiento

{

​   public char variablePublica;

   protected int variableProtegida;

   private String variablePrivada;

​    public char getVariablePublica()

{

    return variablePublica;

  }

   public void setVariablePublica(char variablePublica)

{

    this.variablePublica = variablePublica;

  }

   public char getVariablePublica()

{

    return variablePublica;

  }

   public void setVariablePublica(char variablePublica)

{

    this.variablePublica = variablePublica;

  }

   public char getVariablePublica()

{

    return variablePublica;

  }

   public void setVariablePublica(char variablePublica)

{

    this.variablePublica = variablePublica;

  }

   public void metodoPublico()

{

  }

   protected void metodoProtegido()

{

  }

​    private void metodoPrivado()

{

  }

}

**5. Abstracción**

**5.1 Introducción**

La abstracción es uno de los conceptos clave en POO. Su objetivo principal es manejar la complejidad al ocultar detalles innecesarios, separando la interfaz e implementación. Eso permite implementar una lógica más compleja sobre la abstracción proporcionada sin comprender o incluso pensar en toda la complejidad oculta.

La abstracción intenta minimizar los detalles para que el programador pueda enfocarse en unos pocos conceptos a la vez. La abstracción es la base del desarrollo de software.

**5.2 Ventajas**

* Se evita la duplicación de código y, por lo tanto, el programador no tiene que repetir tareas bastante comunes cada vez que se realiza una operación similar.
* Permite modificar los detalles de implementación interna sin afectar a los usuarios de la abstracción.
* Ayuda a definir las características de subclases de manera eficiente.

**6. Modularidad y reutilización**

En la presente sección aprenderemos los conceptos y técnicas de desarrollo relacionados con la modularidad y la reutilización de código. Los actuales proyectos de desarrollo, dependiendo de su objetivo final se vuelven tediosos de manejar, pero con la correcta utilización de las características del paradigma orientado a objetos, específicamente la modularidad, nos ayudará a que proyectos complejos se vuelvan fáciles de interpretar y manejar, que nuestro código esté ordenado y que podemos reutilizar bloques para hacer eficiente los trabajos y los tiempos de entrega.

**6.1 Modularidad:**

En el anterior contenido del parcial 1 del módulo No. 2 del presente curso hemos hecho una introducción acerca del término, discutiremos entonces algunas de las características y beneficios principales de modular el código de nuestras aplicaciones software, algunas generalidades y las características que giran alrededor del término.

**Booch** nos deja dos objetivos primordiales, primero que los módulos deben ser cohesivos, con estructuras de datos compartidos, algunas de las situaciones que pueden motivar la utilización de esta característica pueden ser las siguientes:

1. Modificaciones repetitivas en diferentes secciones del código
2. Escritura de código similar en diferentes partes para resolver una tarea común
3. Hacer archivos que sean de fácil lectura e interpretación.
4. Separar las diferentes lógicas que puedan actuar al mismo tiempo como piezas independientes usables en otras partes de nuestra aplicación.