TEMA 2. Encapsulación

1. En Programación Orientada a Objetos (POO), ¿Qué buscan la **encapsulación** y **la ocultación** de información? Enumera brevemente algunas ventajas de la ocultación de información.

Respuesta

La **encapsulación en Programación Orientada a Objetos** consiste en agrupar en una misma unidad los datos y los métodos que operan sobre ellos. De esta forma, los objetos contienen tanto su **estado (atributos)** como su **comportamiento (métodos)**. La ocultación de información es una parte fundamental de la encapsulación y consiste en restringir el acceso directo a los detalles internos de un objeto, permitiendo únicamente el acceso a través de una interfaz controlada.

El objetivo principal de estas técnicas es proteger el estado interno del objeto para evitar modificaciones indebidas y garantizar que los datos siempre mantengan valores válidos. En lugar de permitir el acceso directo a los atributos, se obliga a utilizar métodos públicos que controlen cómo se accede o modifica la información.

Entre las ventajas de la ocultación de información destacan la reducción de errores, ya que se evita el uso incorrecto de los datos internos; el aumento de la mantenibilidad del código, porque los cambios en la implementación interna no afectan al resto del programa; y la mejora de la reutilización, ya que la clase puede utilizarse en distintos contextos sin depender de su implementación interna.

2. ¿Qué se entiende por la **interfaz pública** de un objeto o clase en POO? Describe brevemente cómo se relaciona con la ocultación de información.

Respuesta

La **interfaz pública de una clase** es el conjunto de métodos y miembros que pueden ser utilizados desde el exterior de la clase. Representa la forma en la que otros objetos interactúan con esa clase, definiendo qué operaciones están permitidas y cómo deben utilizarse. En Java, esta interfaz suele estar formada por los métodos declarados con el modificador public.

La interfaz pública actúa como un contrato que define el comportamiento accesible del objeto sin revelar cómo está implementado internamente. Esto significa que el usuario de la clase puede utilizar sus métodos sin necesidad de conocer los detalles de su funcionamiento interno.

La relación con la ocultación de información radica en que la interfaz pública permite exponer únicamente lo necesario para utilizar la clase, mientras que el resto de los detalles se mantienen ocultos. Gracias a esto, la implementación interna puede modificarse sin afectar al código que utiliza la clase.

3. Brevemente: ¿Por qué hay que ser conscientes y diseñar con cuidado la **interfaz pública** de una clase? ¿Es fácil cambiarla?

Respuesta

La **interfaz pública de una clase** debe diseñarse cuidadosamente porque representa la forma en que otros componentes del programa interactúan con ella. Si se diseña de forma incorrecta o incompleta, puede provocar dependencias innecesarias o permitir usos indebidos del objeto. Una interfaz mal diseñada puede obligar a modificar múltiples partes del programa cuando se necesite realizar algún cambio.

Cambiar la interfaz pública no suele ser sencillo, ya que puede afectar a todo el código que utiliza esa clase. Si se modifica la firma de un método público o se elimina, el código cliente puede dejar de funcionar o requerir modificaciones importantes. Por esta razón, es habitual intentar mantener la interfaz pública estable una vez que la clase se encuentra en uso.

Diseñar correctamente la interfaz pública favorece la robustez del programa y facilita el mantenimiento. Una interfaz clara, sencilla y bien definida permite que otros desarrolladores utilicen la clase sin necesidad de conocer su implementación interna.

4. ¿Qué son las **invariantes de clase** y por qué la ocultación de información nos ayuda?

Respuesta

Las invariantes de clase son condiciones o reglas que deben cumplirse siempre para que un objeto se considere válido. Estas reglas definen el estado correcto del objeto y deben mantenerse antes y después de ejecutar cualquier método público. Por ejemplo, una clase que represente una cuenta bancaria podría tener como invariante que el saldo nunca sea negativo.

La ocultación de información ayuda a mantener estas invariantes porque evita que el estado interno del objeto sea modificado directamente desde el exterior. En su lugar, se obliga a utilizar métodos que pueden comprobar si las modificaciones cumplen las reglas establecidas.

De esta forma, los métodos públicos pueden incluir validaciones que garanticen que los datos del objeto permanecen en un estado válido. Sin ocultación de información, cualquier parte del programa podría modificar los atributos directamente, lo que aumentaría el riesgo de inconsistencias.

5. Pon un ejemplo de una clase `Punto` en `Java`, con dos coordenadas, `x` e `y`, de tipo `double`, con un método `calcularDistanciaAOrigen`, y que haga uso de la ocultación de información. ¿Cuál es la interfaz pública de la clase `Punto`? ¿Qué significa `public` y `private`?

Respuesta

```
public class Punto {

    private double x;
    private double y;

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }
```

```
    public double calcularDistanciaAOrigen() {
        return Math.sqrt(x * x + y * y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

En esta clase, los atributos **x** e **y** están declarados como privados, lo que impide que otras clases puedan acceder directamente a ellos. El acceso a estos valores se realiza mediante métodos públicos, como el constructor y los métodos **getX** y **getY**. Esto permite controlar cómo se utiliza la información interna del objeto.

La interfaz pública de la clase está formada por el constructor, el método **calcularDistanciaAOrigen** y los métodos **getX** y **getY**. Estos son los elementos que pueden utilizar otras clases para interactuar con el objeto **Punto**.

El modificador **publi**c indica que un miembro puede ser utilizado desde cualquier otra clase. El modificador **private** indica que el miembro sólo puede utilizarse dentro de la propia clase, ocultando así los detalles de implementación.

6. En Java, ¿A quiénes se pueden aplicar los modificadores `public` o `private`?

Respuesta

En Java, los modificadores de acceso **public** y **private** pueden aplicarse a distintos elementos del lenguaje. Principalmente, se utilizan en **clases**, **atributos**, **métodos** y **constructores** para controlar su visibilidad desde otras partes del programa.

Cuando se aplica a una clase, el modificador determina si la clase puede ser utilizada desde otros paquetes. Cuando se aplica a atributos o métodos, determina si pueden ser accedidos o invocados desde otras clases. En el caso de los constructores, el modificador define si se permite crear instancias de la clase desde el exterior.

El uso adecuado de estos modificadores permite controlar el acceso a los elementos internos de una clase, reforzando la encapsulación y ayudando a mantener la integridad del objeto.

7. En POO, la visibilidad puede ser pública o privada, pero ¿existen más tipos de visibilidad? ¿Qué ocurre en Java? ¿Y en otros lenguajes?

Respuesta

En Programación Orientada a Objetos existen varios niveles de visibilidad que permiten controlar el acceso a los miembros de una clase. Además de la visibilidad pública y privada, algunos lenguajes incluyen niveles

intermedios que permiten compartir información dentro de ciertos contextos específicos.

En Java, existen **cuatro niveles de visibilidad: public, private, protected y visibilidad por defecto (también llamada package-private)**. La visibilidad protected permite el acceso desde clases del mismo paquete y desde subclases, mientras que la visibilidad por defecto permite el acceso sólo dentro del mismo paquete.

En otros lenguajes orientados a objetos pueden existir niveles adicionales o variaciones en su comportamiento. Por ejemplo, en C++ existe el modificador protected con un significado similar, y también se pueden utilizar mecanismos como las clases amigas para permitir accesos controlados entre clases.

8. Responde: Los miembros de instancia privados de un objeto están ocultos para (a) otras clases o (b) otras instancias, aunque sean de la misma clase. Pon un ejemplo añadiendo un método `calcularDistanciaAPunto(Punto otro)` y explica la respuesta.

Respuesta

Los miembros privados de una clase están ocultos para otras clases, pero no para otras instancias de la misma clase. Esto significa que un objeto puede acceder a los atributos privados de otro objeto si ambos pertenecen a la misma clase.

```
public double calcularDistanciaAPunto(Punto otro) {
    double dx = this.x - otro.x;
    double dy = this.y - otro.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

En este ejemplo, el método recibe otro objeto de tipo **Punto** y accede directamente a sus atributos privados **x** e **y**. Esto es posible porque el acceso se realiza desde dentro de la misma clase.

Por lo tanto, los miembros privados están ocultos únicamente para otras clases. Las distintas instancias de la misma clase pueden acceder entre sí a sus miembros privados sin restricciones.

9. ¿Qué son los métodos "getter" y "setter" en los lenguajes orientados a objetos?

Respuesta

Los métodos getter y setter son métodos utilizados para acceder y modificar los atributos privados de una clase. Un método getter permite obtener el valor de un atributo, mientras que un método setter permite modificar su valor de forma controlada.

Estos métodos forman parte de la interfaz pública de la clase y permiten mantener la encapsulación. En lugar de permitir el acceso directo a los atributos, se utilizan estos métodos para controlar cómo se leen o modifican los datos.

Además, los setters pueden incluir validaciones para asegurar que los nuevos valores cumplen ciertas condiciones, lo que ayuda a mantener las invariantes de clase y evita estados inconsistentes en los objetos.

10. Cuando nos referimos a que la ocultación de información mejora la "seguridad" del programa, ¿nos referimos a que no pueda ser "hackeado"?

Respuesta

Cuando se afirma que la ocultación de información mejora la seguridad del programa, no se está haciendo referencia a la protección frente a ataques informáticos externos. En este contexto, la seguridad se refiere a la fiabilidad y consistencia del programa frente a errores de programación.

La ocultación de información evita que otras partes del programa modifiquen directamente los datos internos de un objeto. Esto reduce la probabilidad de que el objeto entre en un estado inválido o inconsistente debido a modificaciones incorrectas.

Por tanto, la seguridad en este contexto está relacionada con la robustez del software y la prevención de errores, no con la protección frente a accesos malintencionados o ataques de seguridad informática.

11. ¿Qué diferencia hay entre **miembro de instancia** y **miembro de clase**? ¿Los miembros de clase también se pueden ocultar?

Respuesta

Un miembro de instancia es aquel que pertenece a cada objeto individual creado a partir de una clase. Cada instancia tiene su propia copia de estos miembros. En cambio, un miembro de clase pertenece a la clase en sí misma y es compartido por todas las instancias de esa clase. En Java, los miembros de clase se declaran utilizando la palabra clave **static**.

Los miembros de clase también pueden ocultarse utilizando modificadores de acceso como **private**. Esto permite controlar cómo se accede a la información compartida entre todas las instancias.

La encapsulación se aplica tanto a miembros de instancia como a miembros de clase, permitiendo proteger la información interna y controlar su acceso mediante métodos públicos.

12. Brevemente: ¿Tiene sentido que los constructores sean privados?

Respuesta

Sí, en algunos casos puede tener sentido que los constructores sean privados. Esto impide que otras clases creen instancias directamente, permitiendo que la propia clase controle cómo y cuándo se crean los objetos.

Este enfoque se utiliza en patrones de diseño como el patrón Singleton, donde se quiere garantizar que sólo exista una instancia de la clase. También se utiliza cuando se desea crear objetos mediante métodos factoría que realizan validaciones o transformaciones antes de crear el objeto.

Los constructores privados permiten un mayor control sobre la creación de instancias y refuerzan la encapsulación de la clase.

13. ¿Cómo se indican los **miembros de clase** en Java? Pon un ejemplo, en la clase `Punto` definida anteriormente, para que incluya miembros de clase que permitan saber cuáles son los valores `x` e `y` máximos que se

han establecido en todos los puntos que se hayan creado hasta el momento.

Respuesta

Los miembros de clase en Java se indican utilizando la palabra clave static. Estos miembros pertenecen a la clase y son compartidos por todas las instancias.

```
public class Punto {

    private double x;
    private double y;

    private static double maxX = Double.MIN_VALUE;
    private static double maxY = Double.MIN_VALUE;

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;

        if (x > maxX) maxX = x;
        if (y > maxY) maxY = y;
    }

    public static double getMaxX() {
        return maxX;
    }

    public static double getMaxY() {
        return maxY;
    }
}
```

En este ejemplo, **maxX** y **maxY** son atributos compartidos por todas las instancias. Cada vez que se crea un nuevo objeto, se actualizan si el nuevo punto tiene coordenadas mayores.

14. Como sería un método factoría dentro de la clase `Punto` para construir un `Punto` a partir de dos coordenadas, pero que las redondee al entero más cercano. Escribe sólo el código del método, no toda la clase ¿Has usado `static`?

Respuesta

```
public static Punto crearPuntoRedondeado(double x, double y) {
    return new Punto(Math.round(x), Math.round(y));
}
```

Se ha utilizado **static** porque el método pertenece a la clase y no a una instancia concreta. Esto permite crear objetos sin necesidad de tener previamente un objeto de la clase.

15. Cambia la implementación de `Punto`. En vez de dos `double`, emplea un array interno de dos posiciones, intentando no modificar la interfaz pública de la clase.

Respuesta

```
public class Punto {

    private double[] coordenadas = new double[2];

    public Punto(double x, double y) {
        coordenadas[0] = x;
        coordenadas[1] = y;
    }

    public double getX() {
        return coordenadas[0];
    }

    public double getY() {
        return coordenadas[1];
    }

    public double calcularDistanciaAOrigen() {
        return Math.sqrt(coordenadas[0] * coordenadas[0] +
                         coordenadas[1] * coordenadas[1]);
    }
}
```

En este caso se ha cambiado la implementación interna utilizando un array, pero la interfaz pública sigue siendo la misma. Esto demuestra una de las ventajas de la encapsulación, ya que el cambio interno no afecta al código que utiliza la clase.

16. Si un atributo va a tener un método "getter" y "setter" públicos, ¿no es mejor declararlo público? ¿Cuál es la convención más habitual sobre los atributos, que sean públicos o privados? ¿Tiene esto algo que ver con las "invariantes de clase"?

Respuesta

Aunque un atributo tenga métodos getter y setter públicos, no es recomendable declararlo público. Utilizar métodos permite incluir validaciones, cálculos adicionales o cambios en la implementación sin afectar al código cliente.

La convención más habitual en Programación Orientada a Objetos es declarar los atributos como privados y proporcionar métodos públicos sólo cuando sea necesario. Esto permite mantener el control sobre el acceso y

modificación de los datos.

Esta práctica está directamente relacionada con las invariantes de clase, ya que los setters pueden comprobar que los valores asignados cumplen las reglas establecidas, garantizando que el objeto permanezca en un estado válido.

17. ¿Qué significa que una clase sea **inmutable**? ¿qué es un método modificador? ¿Un método modificador es siempre un "setter"? ¿Tiene ventajas que una clase sea inmutable?

Respuesta

Una clase es inmutable cuando el estado de sus objetos no puede modificarse después de su creación. Esto implica que todos sus atributos se establecen en el constructor y no existen métodos que permitan cambiarlos posteriormente.

Un método modificador es cualquier método que altera el estado interno de un objeto. Un setter es un tipo concreto de método modificador, pero no todos los métodos modificadores son setters, ya que un método puede modificar el estado realizando cálculos o cambios más complejos.

Las clases inmutables tienen ventajas importantes, como mayor seguridad frente a errores, facilidad para trabajar en entornos concurrentes y simplificación del razonamiento sobre el comportamiento del programa.

18. ¿Es recomendable incluir métodos "setter" siempre y como convención?

Respuesta

No es recomendable incluir métodos setter siempre como norma general. Los setters deben utilizarse únicamente cuando sea necesario permitir la modificación del estado de un objeto.

Incluir setters sin necesidad puede romper la encapsulación y facilitar que el objeto entre en estados inconsistentes. Además, puede dificultar el mantenimiento del programa si se permite modificar atributos que deberían permanecer constantes.

En muchos casos, es preferible crear objetos inmutables o proporcionar métodos específicos que realicen cambios controlados en el estado del objeto.

19. ¿La clase `String` en Java es mutable o inmutable? ¿Qué ocurre al concatenar dos cadenas? ¿Qué debemos hacer si vamos a hacer una operación que implique concatenar muchas veces para construir paso a paso una cadena muy larga?

Respuesta

La clase **String** en Java es inmutable, lo que significa que su contenido no puede modificarse después de su creación. Cada vez que se realiza una operación que aparentemente modifica una cadena, en realidad se crea un nuevo objeto.

Al concatenar dos cadenas, se genera una nueva instancia de String que contiene el resultado de la concatenación, mientras que las cadenas originales permanecen sin cambios.

Cuando se necesita realizar muchas concatenaciones, es recomendable utilizar la clase **StringBuilder**, ya que permite modificar el contenido de forma eficiente sin crear múltiples objetos intermedios.

20. En POO ¿Cómo se comparan objetos de una misma clase? ¿Por su contenido o por su identidad? ¿Qué es el método equals en Java? ¿Qué hace por defecto? ¿Cómo se deben comparar dos cadenas en Java?

Respuesta

En Programación Orientada a Objetos, los objetos pueden compararse por su identidad o por su contenido. La identidad determina si dos referencias apuntan al mismo objeto en memoria, mientras que la comparación por contenido evalúa si los objetos tienen valores equivalentes.

En Java, el método **equals** se utiliza para comparar el contenido de los objetos. Por defecto, el método equals heredado de la clase **Object** compara la identidad, es decir, si ambas referencias apuntan al mismo objeto.

Para comparar cadenas en Java se debe utilizar el método equals, ya que el operador == sólo compara referencias. Utilizar equals permite comparar el contenido real de las cadenas.

21. ¿Qué son las clases "wrapper" en un lenguaje de programación orientado a objetos? ¿Cómo se hace? ¿Es un proceso automático? ¿Qué ventajas tienen? ¿Todos los lenguajes orientados a objetos tienen tipos primitivos y necesitan wrappers?

Respuesta

Las clases wrapper son clases que encapsulan tipos primitivos dentro de objetos. En Java, existen clases como **Integer**, **Double** o **Boolean** que permiten tratar valores primitivos como objetos.

En Java, la conversión entre tipos primitivos y wrappers puede realizarse de forma automática mediante un mecanismo llamado autoboxing y unboxing. Esto facilita el uso de tipos primitivos en colecciones y estructuras que requieren objetos.

Las clases wrapper permiten utilizar métodos adicionales, trabajar con colecciones genéricas y mejorar la integración con el modelo orientado a objetos. No todos los lenguajes orientados a objetos tienen tipos primitivos separados, por lo que no todos necesitan wrappers.

22. ¿En POO qué es un **tipo de dato enumerado**? ¿En Java, un tipo de dato enumerado es una clase? ¿Qué ventajas tienen en términos de encapsulación los enumerados en Java?

Respuesta

Un tipo enumerado es un tipo de dato que define un conjunto limitado y fijo de valores posibles. Se utiliza cuando un valor sólo puede tomar una serie concreta de opciones predefinidas.

En Java, un tipo enumerado es realmente una clase especial que puede contener atributos, métodos y constructores. Cada valor del enumerado es una instancia única de esa clase.

Los enumerados en Java mejoran la encapsulación porque permiten agrupar comportamiento y datos relacionados con cada valor. Además, evitan el uso de constantes sueltas y reducen errores al garantizar que sólo se utilicen valores válidos.

23. Crea un tipo enumerado en Java que se llame `Mes`, con doce posibles instancias y que además proporcione métodos para obtener cuántos días tiene ese mes, el ordinal de ese mes en el año (1-12), empleando atributos privados y constructores del tipo enumerado. Añade además cuatro métodos para devolver si ese mes tiene algunos días de invierno, primavera, verano u otoño, indicando con un booleano el hemisferio (norte o sur, parámetro `enHemisferioNorte`). Es decir: `esDePrimavera(boolean esHemisferioNorte)`, `esDeVerano(boolean esHemisferioNorte)`, `esDeOtoño(boolean esHemisferioNorte)`, `esDeInvierno(boolean esHemisferioNorte)`

Respuesta

```
public enum Mes {

    ENERO(31, 1), FEBRERO(28, 2), MARZO(31, 3),
    ABRIL(30, 4), MAYO(31, 5), JUNIO(30, 6),
    JULIO(31, 7), AGOSTO(31, 8), SEPTIEMBRE(30, 9),
    OCTUBRE(31, 10), NOVIEMBRE(30, 11), DICIEMBRE(31, 12);

    private int dias;
    private int ordinal;

    Mes(int dias, int ordinal) {
        this.dias = dias;
        this.ordinal = ordinal;
    }

    public int getDias() {
        return dias;
    }

    public int getOrdinal() {
        return ordinal;
    }

    public boolean esDePrimavera(boolean enHemisferioNorte) {
        if (enHemisferioNorte)
            return ordinal >= 3 && ordinal <= 5;
        else
            return ordinal >= 9 && ordinal <= 11;
    }
```

```
    public boolean esDeVerano(boolean enHemisferioNorte) {
        if (enHemisferioNorte)
            return ordinal >= 6 && ordinal <= 8;
        else
            return ordinal == 12 || ordinal <= 2;
    }

    public boolean esDeOtoño(boolean enHemisferioNorte) {
        if (enHemisferioNorte)
            return ordinal >= 9 && ordinal <= 11;
        else
            return ordinal >= 3 && ordinal <= 5;
    }

    public boolean esDeInvierno(boolean enHemisferioNorte) {
        if (enHemisferioNorte)
            return ordinal == 12 || ordinal <= 2;
        else
            return ordinal >= 6 && ordinal <= 8;
    }
}
```