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 la Programación Orientada a Objetos (POO) busca agrupar datos y métodos que operan sobre esos datos dentro de una misma unidad, llamada clase. Este concepto está estrechamente relacionado con la ocultación de información, que consiste en restringir el acceso directo a los detalles internos de un objeto, exponiendo únicamente lo necesario a través de una interfaz pública. Esto permite que los detalles de implementación puedan cambiar sin afectar a los usuarios del objeto.

Entre las ventajas de la ocultación de información se encuentran: (1) mejora la seguridad del programa al evitar accesos indebidos o modificaciones no controladas de los datos internos; (2) facilita el mantenimiento y la evolución del código, ya que los cambios internos no afectan a otras partes del programa; y (3) promueve la reutilización del código al proporcionar interfaces claras y estables.

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 un objeto o clase en POO se refiere al conjunto de métodos y atributos que están disponibles para ser utilizados por otras partes del programa. Esta interfaz define cómo interactuar con el objeto, ocultando los detalles internos de su implementación. Por ejemplo, en una clase `CuentaBancaria`, la interfaz pública podría incluir métodos como `depositar` o `retirar`, mientras que los detalles sobre cómo se almacenan los saldos quedarían ocultos.

La relación con la ocultación de información es directa, ya que la interfaz pública actúa como un contrato que permite a los usuarios del objeto interactuar con él sin necesidad de conocer su

funcionamiento interno. Esto reduce la complejidad y minimiza el riesgo de errores al modificar el código interno.

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

Respuesta

Diseñar con cuidado la interfaz pública de una clase es crucial porque esta define cómo otros componentes del programa interactúan con la clase. Una interfaz mal diseñada puede llevar a un uso incorrecto del objeto o dificultar su mantenimiento. Además, una interfaz pública actúa como un contrato, y cualquier cambio en ella puede romper la compatibilidad con el código que depende de esa clase.

Cambiar la interfaz pública no es fácil, especialmente en sistemas grandes o distribuidos, ya que puede requerir modificaciones en múltiples partes del programa. Por ello, es importante planificar y diseñar la interfaz pública con previsión, asegurándose de que sea clara, coherente y lo más estable posible.

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 en todo momento para garantizar que un objeto se encuentra en un estado válido. Por ejemplo, en una clase `CuentaBancaria`, una invariante podría ser que el saldo nunca sea negativo. Estas reglas aseguran la consistencia y la corrección del comportamiento del objeto.

La ocultación de información ayuda a mantener las invariantes de clase al restringir el acceso directo a los atributos internos. Al obligar a los usuarios a interactuar con el objeto a través de métodos controlados, se pueden validar las operaciones y garantizar que las invariantes no se violen. Esto refuerza la robustez y la fiabilidad del programa.

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;
    }
}
```

La interfaz pública de la clase `Punto` incluye el constructor `Punto(double x, double y)` , el método `calcularDistanciaAOrigen` y los métodos `getX` y `getY` . Estos métodos permiten interactuar con los objetos de la clase sin exponer directamente los atributos internos `x` e `y` .

El modificador `public` indica que un método o atributo es accesible desde cualquier parte del programa, mientras que `private` restringe el acceso únicamente a la propia clase. Esto permite controlar cómo se accede y modifica el estado interno del objeto.

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

Respuesta

En Java, los modificadores `public` y `private` se pueden aplicar a clases, métodos y atributos. Un miembro de clase declarado como `public` es accesible desde cualquier otra clase en cualquier paquete, mientras que un miembro declarado como `private` solo es accesible dentro de la propia clase en la que se declara.

Además de `public` y `private` , Java también ofrece otros modificadores de acceso como `protected` y el modificador de paquete (sin especificador), que ofrecen diferentes niveles de visibilidad y acceso.

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

Además de la visibilidad pública y privada, en POO también puede existir la visibilidad protegida (protected) y la visibilidad por defecto (package-private en Java). La visibilidad protegida permite el acceso a clases en el mismo paquete y a subclases, mientras que la visibilidad por defecto (sin modificador) restringe el acceso a clases en el mismo paquete.

En Java, estos cuatro niveles de visibilidad son: `public` , `protected` , (sin modificador) y `private` . Otros lenguajes de programación orientada a objetos pueden tener conceptos similares pero con diferentes palabras clave o reglas. Por ejemplo, en C++, además de `public` y `private` , existe el modificador `protected` y el modificador de clase base, que permite controlar el acceso a los miembros de una clase base desde clases derivadas.

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 de instancia privados de un objeto están ocultos para (a) otras clases. Esto significa que no se puede acceder directamente a ellos desde fuera de la clase en la que están definidos. Por ejemplo, si tenemos dos objetos de la clase `Punto`, cada uno tiene sus propias coordenadas `x` e `y`, y estas están encapsuladas y son inaccesibles directamente desde fuera de la clase.

Sin embargo, otras instancias de la misma clase pueden acceder a estos miembros privados. Por ejemplo, podríamos tener un método `calcularDistanciaAPunto(Punto otro)` que calcule la distancia entre el punto actual y otro punto pasado como parámetro. Este método podría acceder a los miembros privados `x` e `y` de ambos puntos para realizar el cálculo.

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

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

Respuesta

Los métodos "getter" y "setter" son métodos especiales en los lenguajes orientados a objetos que se utilizan para acceder y modificar los valores de los atributos privados de una clase. Un método "getter" devuelve el valor de un atributo, mientras que un método "setter" establece o actualiza el valor de ese atributo.

Por ejemplo, si tenemos un atributo privado `nombre` en una clase, podríamos tener un método "getter" llamado `getNombre()` que devuelva el valor de `nombre`, y un método "setter" llamado `setNombre(String nuevoNombre)` que establezca un nuevo valor para `nombre`.

Estos métodos son una parte importante de la encapsulación, ya que permiten controlar el acceso y la modificación de los atributos de una clase, asegurando que se mantengan las invariantes de clase y se realicen las validaciones necesarias.

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 hablamos de que la ocultación de información mejora la "seguridad" del programa, nos referimos principalmente a la protección de los datos internos de un objeto contra accesos o modificaciones no autorizadas. Al ocultar la información y exponer solo lo necesario a través de una interfaz pública, se reduce el riesgo de que otras partes del programa (o usuarios malintencionados) puedan interferir con el funcionamiento interno del objeto de manera inesperada.

Sin embargo, esto no significa que el programa no pueda ser "hackeado". La seguridad en términos de protección contra ataques externos implica muchas otras consideraciones, como la validación de entradas, la gestión de permisos y la protección contra vulnerabilidades conocidas. La ocultación de información es solo una parte de un enfoque de seguridad más amplio.

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

Respuesta

La diferencia entre **miembro de instancia** y **miembro de clase** radica en cómo se asocian con los objetos de una clase. Un **miembro de instancia** es un atributo o método que pertenece a una instancia específica de una clase (es decir, a un objeto), mientras que un **miembro de clase** es un atributo o método que pertenece a la clase en sí misma y es compartido por todas las instancias de esa clase.

Sí, los miembros de clase también se pueden ocultar. Al igual que con los miembros de instancia, se pueden utilizar modificadores de acceso como `private` para restringir el acceso a los miembros de clase. Por ejemplo, en una clase `Contador`, podríamos tener un miembro de clase `totalContadores` que sea privado y solo accesible a través de métodos de la propia clase.

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

Respuesta

Sí, tiene sentido que los constructores sean privados en ciertos casos. Un constructor privado impide que se creen instancias de la clase desde fuera de la propia clase, lo que puede ser útil para controlar la creación de objetos y garantizar que se cumplan ciertas condiciones o invariantes.

Por ejemplo, en una clase que implementa el patrón Singleton, el constructor es privado para evitar que se creen múltiples instancias de la clase. En su lugar, se proporciona un método público estático que devuelve la única instancia 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

En Java, los miembros de clase se indican utilizando el modificador `static`. Un miembro de clase es compartido por todas las instancias de la clase y se puede acceder a él sin necesidad de crear un objeto de la clase.

En la clase `Punto`, podríamos tener miembros de clase `maxX` y `maxY` que se actualizan cada vez que se crea un nuevo punto. Estos miembros de clase almacenarían los valores máximos de `x` e `y` establecidos hasta el momento.

```java
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;
    }
}
```

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

```java
public static Punto crearDesdeCoordenadas(double x, double y) {
    int xRedondeado = (int) Math.round(x);
    int yRedondeado = (int) Math.round(y);
    return new Punto(xRedondeado, yRedondeado);
}
```

Sí, he usado `static` . El método `crearDesdeCoordenadas` es un método de clase (método estático) que se puede llamar sin necesidad de crear una instancia de la clase `Punto` . Este método redondea las coordenadas al entero más cercano y luego llama al constructor de la clase `Punto` para crear un nuevo objeto `Punto` con esos valores redondeados.

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

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

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

La implementación de la clase `Punto` ha cambiado para utilizar un array interno `coordenadas` de dos posiciones en lugar de dos atributos `double` separados. Sin embargo, la interfaz pública de la clase se ha mantenido igual, por lo que el código que depende de la clase `Punto` no necesita conocer los detalles de esta implementación.

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

No, no es mejor declarar un atributo como público incluso si va a tener un método "getter" y "setter" públicos. Declarar un atributo como público expone directamente su valor y permite que sea modificado sin restricciones, lo que puede violar las invariantes de clase y hacer que el objeto quede en un estado inconsistente.

La convención más habitual es declarar los atributos como privados y proporcionar métodos "getter" y "setter" públicos para acceder y modificar sus valores. Esto permite controlar el acceso y la modificación de los atributos, asegurando que se mantengan las invariantes de clase y se realicen las validaciones necesarias.

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** si su estado no puede ser modificado una vez que ha sido creada. Esto significa que todos sus atributos son finales (final) y no pueden ser cambiados, y que no tiene métodos "setter" o modificadores que alteren su estado.

Un método modificador es un método que cambia o modifica el estado de un objeto. No todos los métodos modificadores son "setters", pero todos los "setters" son métodos modificadores. Un método modificador puede realizar otras acciones además de simplemente establecer un valor, como validar o transformar el valor antes de asignarlo.

Tener clases inmutables tiene varias ventajas: (1) son más seguras en entornos concurrentes, ya que su estado no puede ser cambiado por múltiples hilos al mismo tiempo; (2) son más fáciles de razonar y entender, ya que su comportamiento es predecible y no depende de su estado interno; y (3) pueden ser utilizadas como claves en estructuras de datos como mapas o conjuntos, ya que su estado no cambia y por lo tanto su hashcode es constante.

18. ¿Es recomendable incluir métodos "setter"

siempre y como convención?

Respuesta

No, no es recomendable incluir métodos "setter" siempre. Incluir un método "setter" para un atributo implica que el valor de ese atributo puede ser cambiado en cualquier momento, lo que puede violar las invariantes de clase y hacer que el objeto quede en un estado inconsistente.

Los métodos "setter" deben ser incluidos solo cuando sea necesario y tenga sentido desde el punto de vista del diseño de la clase. Por ejemplo, si un atributo debe ser configurable o actualizado después de la creación del objeto, entonces puede tener sentido proporcionar un "setter" para ese atributo. Sin embargo, si un atributo es fundamental para el estado del objeto y no debe cambiar una vez establecido, entonces no debe tener un "setter".

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. Esto significa que una vez que se crea un objeto `String`, su valor no puede ser cambiado. Cuando se concatena una cadena a otra, en realidad se está creando un nuevo objeto `String` que contiene el valor de ambas cadenas.

Si se va a realizar una operación que implique concatenar muchas veces para construir paso a paso una cadena muy larga, es recomendable utilizar la clase `StringBuilder` o `StringBuffer` en su lugar. Estas clases representan cadenas mutables y permiten modificar el contenido de la cadena sin crear nuevos objetos. Por ejemplo:

```
StringBuilder sb = new StringBuilder();
for (String s : listaDeCadenas) {
    sb.append(s);
}
String resultado = sb.toString();
```

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 POO, los objetos de una misma clase se pueden comparar por su contenido o por su identidad, dependiendo de cómo se haya implementado la comparación. La comparación por contenido verifica si los atributos relevantes de los objetos son iguales, mientras que la comparación por identidad verifica si los objetos son la misma instancia en memoria.

En Java, el método `equals` se utiliza para comparar objetos por su contenido. Por defecto, el método `equals` de la clase `Object` compara la identidad de los objetos, pero muchas clases (incluyendo `String`) sobrescriben este método para proporcionar una comparación por contenido.

Para comparar dos cadenas en Java, se debe utilizar el método `equals` de la clase `String`, que compara el contenido de las cadenas. Por ejemplo:

```
String cadena1 = "hola";
String cadena2 = "hola";
boolean sonIguales = cadena1.equals(cadena2); // true
```

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 envuelven o encapsulan un tipo de dato primitivo en un objeto. Proporcionan una forma de utilizar tipos primitivos como objetos, lo que puede ser útil en situaciones donde se requieren objetos en lugar de valores primitivos (por ejemplo, en colecciones como `ArrayList`).

En Java, por ejemplo, existen clases wrapper como `Integer` para el tipo `int`, `Double` para el tipo `double`, y `Boolean` para el tipo `boolean`. Estas clases proporcionan métodos para convertir entre el tipo primitivo y el objeto wrapper, así como otros métodos útiles.

El proceso de "boxing" y "unboxing" (convertir entre un tipo primitivo y su correspondiente wrapper) no es automático en todos los lenguajes orientados a objetos. En Java, por ejemplo, el proceso es automático a partir de Java 5, gracias a la característica de autoboxing. Sin embargo, en otros lenguajes puede ser necesario realizar la conversión manualmente.

No todos los lenguajes orientados a objetos tienen tipos primitivos y necesitan wrappers. Algunos lenguajes, como Smalltalk o Ruby, no hacen distinción entre tipos primitivos y objetos, y todos los valores son tratados como objetos.

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 de dato enumerado** es un tipo de dato que consiste en un conjunto limitado y fijo de valores posibles, que representan las distintas opciones o categorías que puede tomar una variable de ese tipo. Por ejemplo, un tipo de dato enumerado `DiasDeLaSemana` podría tener como valores posibles `LUNES`, `MARTES`, `MIERCOLES`, etc.

En Java, un tipo de dato enumerado es una clase especial que extiende la clase `java.lang.Enum`. Cada valor posible del enumerado es una instancia de esta clase. Por ejemplo, el enumerado `DiasDeLaSemana` sería una clase con un conjunto fijo de instancias, una para cada día de la semana.

Los enumerados en Java tienen varias ventajas en términos de encapsulación: (1) los valores posibles están encapsulados dentro de la clase del enumerado, lo que impide que se creen valores no válidos; (2) se pueden definir métodos y atributos en el enumerado, lo que permite asociar comportamiento y datos adicionales a cada valor; y (3) se puede controlar la serialización y deserialización de los valores del enumerado, lo que garantiza que se mantenga la integridad de los datos.

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 final int dias;
    private final int ordinal;

    private 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) {
        return (this == MARZO || this == ABRIL || this == MAYO) ^ !enHemisferioNorte;
    }

    public boolean esDeVerano(boolean enHemisferioNorte) {
```

```
        return (this == JUNIO || this == JULIO || this == AGOSTO) ^ !enHemisferioNorte;
    }

    public boolean esDeOtoño(boolean enHemisferioNorte) {
        return (this == SEPTIEMBRE || this == OCTUBRE || this == NOVIEMBRE) ^ !enHemisferioNor
    }

    public boolean esDeInvierno(boolean enHemisferioNorte) {
        return (this == DICIEMBRE || this == ENERO || this == FEBRERO) ^ !enHemisferioNorte;
    }
}
```