

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 y la ocultación de información son principios fundamentales en la Programación Orientada a Objetos (POO) que buscan proteger los datos y garantizar la integridad de los objetos. La encapsulación consiste en agrupar datos y métodos que operan sobre esos datos dentro de una misma unidad, llamada clase. Por otro lado, la ocultación de información se refiere a restringir el acceso directo a los datos internos de un objeto, permitiendo que solo se interactúe con ellos a través de métodos específicos.

Entre las ventajas de la ocultación de información se encuentran las siguientes:

1. **Protección de los datos:** Al restringir el acceso directo a los atributos de una clase, se evita que estos sean modificados de manera indebida o accidental, lo que ayuda a mantener la consistencia y la validez de los datos.
2. **Facilidad de mantenimiento:** Al encapsular los datos y exponer solo lo necesario mediante métodos públicos, se reduce el riesgo de que cambios internos en la implementación afecten a otras partes del programa.
3. **Mayor control:** Los métodos que permiten acceder o modificar los datos pueden incluir validaciones o restricciones, lo que asegura que los datos siempre cumplan con ciertas condiciones.
4. **Modularidad:** La encapsulación facilita la división del código en módulos independientes, lo que mejora la organización y la reutilización del código.

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 Programación Orientada a Objetos (POO) se refiere al conjunto de métodos y atributos que están disponibles para ser utilizados por otras clases o partes del programa. Es la forma en que un objeto expone su funcionalidad al mundo exterior, permitiendo que otras partes del programa interactúen con él sin necesidad de conocer los detalles internos de su implementación.

La relación entre la interfaz pública y la ocultación de información es fundamental. La ocultación de información implica que los detalles internos de una clase, como sus atributos o métodos privados, no son accesibles directamente desde fuera de la clase. En cambio, la interfaz pública actúa como un intermediario, proporcionando acceso controlado a las funcionalidades necesarias mientras protege los datos internos. Esto permite mantener la integridad de los datos y facilita la evolución del código, ya que los cambios internos en la implementación no afectan a las partes externas que interactúan con la clase.

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 otras partes del programa interactuarán con la clase. Una interfaz pública mal diseñada puede llevar a un uso incorrecto de la clase, dificultar su mantenimiento y limitar su reutilización. Además, una interfaz pública bien definida ayuda a que el código sea más comprensible y fácil de usar por otros desarrolladores.

Cambiar la interfaz pública de una clase no es una tarea sencilla, especialmente si la clase ya está siendo utilizada en múltiples partes del programa. Modificarla puede requerir cambios en todas las dependencias que interactúan con ella, lo que puede ser costoso y propenso a errores. Por esta razón, es importante planificar cuidadosamente la interfaz pública desde el principio, considerando tanto las necesidades actuales como las futuras del sistema.

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 encuentre en un estado válido. Estas condiciones suelen estar relacionadas con los valores de los atributos de la clase y las relaciones entre ellos. Por ejemplo, en una clase que representa un intervalo de números, una invariante podría ser que el valor del límite inferior siempre sea menor o igual al del límite superior.

La ocultación de información contribuye a mantener las invariantes de clase al restringir el acceso directo a los atributos internos. Al obligar a los usuarios de la clase a interactuar con ella a través de métodos públicos, se puede garantizar que cualquier modificación de los datos internos pase por validaciones que aseguren el cumplimiento de las invariantes. Esto reduce el riesgo de errores y mejora la robustez 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 getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    public void setX(double x) {  
        this.x = x;  
    }  
  
    public void setY(double y) {  
        this.y = y;  
    }  
  
    public double calcularDistanciaAOrigen() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

La interfaz pública de la clase `Punto` está compuesta por los métodos `getX`, `getY`, `setX`, `setY` y `calcularDistanciaAOrigen`. Estos métodos permiten interactuar con los atributos `x` e `y` de manera controlada, sin exponerlos directamente.

El modificador `public` indica que un método o atributo es accesible desde cualquier otra clase. Por otro lado, el modificador `private` restringe el acceso, permitiendo que solo los métodos de la misma

clase puedan acceder o modificar los atributos privados. Esto asegura que los datos internos estén protegidos y que solo puedan ser manipulados de manera controlada.

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:

1. **Clases:** Una clase puede ser declarada como `public` si se desea que sea accesible desde cualquier otro paquete. Si no se especifica ningún modificador, la clase tendrá visibilidad por defecto (`package-private`), lo que significa que solo será accesible desde otras clases dentro del mismo paquete.
2. **Atributos:** Los atributos de una clase pueden ser `public` para que sean accesibles desde cualquier lugar, o `private` para que solo sean accesibles dentro de la misma clase. Esto permite controlar el acceso a los datos y proteger la integridad de los mismos.
3. **Métodos:** Los métodos pueden ser `public` para que puedan ser invocados desde cualquier otra clase, o `private` para que solo puedan ser utilizados dentro de la clase en la que están definidos. Esto es útil para encapsular la lógica interna de la clase.
4. **Constructores:** Los constructores también pueden ser `public` o `private`. Un constructor `public` permite crear instancias de la clase desde cualquier lugar, mientras que un constructor `private` restringe la creación de instancias, lo que puede ser útil en patrones de diseño como el Singleton.

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 Java existen otros dos niveles de visibilidad: `protected` y el nivel de visibilidad por defecto (también conocido como `package-private`).

1. **`protected` :** Los miembros con este modificador son accesibles desde la misma clase, desde las clases del mismo paquete y desde las subclases, incluso si estas se encuentran en un paquete

diferente. Este nivel de visibilidad es útil cuando se desea permitir que las subclases accedan a ciertos miembros, pero no otras clases externas.

2. **Visibilidad por defecto (package-private):** Si no se especifica ningún modificador de acceso, los miembros de la clase son accesibles únicamente desde otras clases dentro del mismo paquete. Este nivel de visibilidad es útil para organizar el código en paquetes y limitar el acceso a los miembros solo a las clases relacionadas.

En otros lenguajes de programación orientados a objetos, como C++, existen conceptos similares, aunque con algunas diferencias. Por ejemplo, en C++ también se utiliza `public`, `private` y `protected`, pero además se puede especificar la visibilidad de los miembros en bloques dentro de la definición de la clase. Otros lenguajes, como Python, no tienen modificadores de acceso explícitos, pero utilizan convenciones como el uso de guiones bajos para indicar que un atributo o método es privado.

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 otras clases, pero no para otras instancias de la misma clase. Esto significa que, aunque un atributo privado no puede ser accedido directamente desde una clase externa, sí puede ser accedido por métodos de la misma clase, incluso si estos métodos operan sobre otra instancia de la misma clase.

Por ejemplo, en la clase `Punto`, se puede implementar un método `calcularDistanciaAPunto` que calcule la distancia entre el punto actual y otro punto pasado como parámetro. Aunque los atributos `x` y `y` son privados, el método puede acceder a ellos porque ambos puntos son instancias de la misma clase `Punto`.

```

public class Punto {
    private double x;
    private double y;

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

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

```

En este ejemplo, el método `calcularDistanciaAPunto` accede directamente a los atributos privados `x` e `y` del objeto `otro`, ya que ambos son instancias de la misma clase `Punto`. Esto demuestra que los miembros privados no están ocultos entre instancias de la misma clase, pero sí lo están para otras clases.

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

Respuesta

Los métodos "getter" y "setter" son funciones utilizadas en los lenguajes orientados a objetos para acceder y modificar los atributos privados de una clase. Estos métodos permiten implementar la encapsulación, ya que proporcionan un control sobre cómo se accede y se modifica el estado interno de un objeto.

Un método "getter" se utiliza para obtener el valor de un atributo privado. Por lo general, su nombre comienza con la palabra `get` seguida del nombre del atributo con la primera letra en mayúscula. Por ejemplo, si un atributo se llama `nombre`, el método getter correspondiente sería `getNombre`.

Por otro lado, un método "setter" se utiliza para modificar el valor de un atributo privado. Su nombre suele comenzar con la palabra `set` seguida del nombre del atributo con la primera letra en mayúscula. Por ejemplo, para el atributo `nombre`, el método setter sería `setNombre`. Este método

puede incluir validaciones para asegurarse de que el nuevo valor sea válido antes de asignarlo al atributo.

El uso de getters y setters es una práctica común en POO porque permite mantener el principio de ocultación de información, al mismo tiempo que proporciona una forma controlada y segura de interactuar con los datos internos de un objeto.

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 dice que la ocultación de información mejora la "seguridad" del programa, no se refiere a la protección contra ataques externos o ciberseguridad, sino a la capacidad de proteger la integridad de los datos y el correcto funcionamiento del programa. La ocultación de información permite que los datos internos de una clase estén protegidos contra accesos o modificaciones indebidas, lo que reduce la posibilidad de errores y comportamientos inesperados.

Este concepto está relacionado con la idea de que los datos sensibles o críticos de un objeto no deben ser accesibles directamente desde fuera de la clase. En su lugar, se deben proporcionar métodos controlados (como getters y setters) que permitan acceder o modificar los datos de manera segura, aplicando validaciones o restricciones si es necesario. Esto asegura que el objeto siempre se mantenga en un estado válido y predecible.

Por lo tanto, aunque la ocultación de información no protege un programa contra ataques externos, sí contribuye a la robustez y confiabilidad del código, lo que puede considerarse una forma de "seguridad" en el contexto del diseño de software.

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 una variable o método que pertenece a una instancia específica de una clase. Esto significa que cada objeto creado a partir de la clase tiene su propia copia de los miembros

de instancia, y los cambios realizados en un objeto no afectan a los demás. Por ejemplo, si se tiene una clase `Coche` con un atributo de instancia `color`, cada objeto de tipo `Coche` puede tener un color diferente.

Por otro lado, un miembro de clase es compartido por todas las instancias de la clase. En Java, los miembros de clase se definen utilizando el modificador `static`. Esto significa que no es necesario crear una instancia de la clase para acceder a estos miembros; se pueden acceder directamente a través del nombre de la clase. Por ejemplo, si se tiene un atributo estático `numeroDeCoches` en la clase `Coche`, este atributo será común para todos los objetos de tipo `Coche` y se puede acceder a él como `Coche.numeroDeCoches`.

Los miembros de clase también se pueden ocultar utilizando el modificador de acceso `private`. Esto asegura que solo los métodos de la misma clase puedan acceder a ellos, lo que permite mantener la encapsulación y proteger la integridad de los datos compartidos entre las instancias de la clase.

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

Respuesta

Sí, tiene sentido que los constructores sean privados en ciertos casos específicos. Un constructor privado se utiliza principalmente para restringir la creación de instancias de una clase desde fuera de la misma. Esto es útil en patrones de diseño como el Singleton, donde se necesita garantizar que solo exista una única instancia de la clase en todo el programa.

Además, los constructores privados también se emplean en clases que contienen únicamente métodos estáticos, como las clases de utilidades. En estos casos, el constructor privado evita que se creen instancias innecesarias de la clase, ya que su propósito no es representar objetos, sino proporcionar un conjunto de funciones relacionadas.

Por lo tanto, aunque no es común que los constructores sean privados, su uso es adecuado en situaciones donde se desea controlar estrictamente la creación de instancias o evitarla por completo.

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`. Esto significa que el miembro pertenece a la clase en sí, en lugar de a una instancia específica. Los miembros estáticos se comparten entre todas las instancias de la clase y se pueden acceder directamente a través del nombre de la clase.

A continuación, se muestra un ejemplo de cómo modificar la clase `Punto` para incluir miembros de clase que registren los valores máximos de `x` e `y` establecidos en todos los puntos creados:

```

public class Punto {
    private double x;
    private double y;
    private static double maxX = Double.NEGATIVE_INFINITY;
    private static double maxY = Double.NEGATIVE_INFINITY;

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

    private static void actualizarMaximos(double x, double y) {
        if (x > maxX) {
            maxX = x;
        }
        if (y > maxY) {
            maxY = y;
        }
    }

    public static double getMaxX() {
        return maxX;
    }

    public static double getMaxY() {
        return maxY;
    }

    // Métodos existentes...
}

```

En este ejemplo, los atributos `maxX` y `maxY` son miembros de clase, ya que están declarados como `static`. Esto significa que son compartidos por todas las instancias de la clase `Punto`. El método privado `actualizarMaximos` se asegura de que estos valores se actualicen cada vez que se crea un nuevo objeto `Punto` con coordenadas mayores que las actuales.

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

Un método factoría es un método estático que se utiliza para crear instancias de una clase. En este caso, el método factoría redondeará las coordenadas al entero más cercano antes de crear un nuevo objeto `Punto` . A continuación se muestra el código del método:

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

En este ejemplo, el método `crearPuntoRedondeado` es estático (`static`), lo que significa que se puede llamar directamente desde la clase `Punto` sin necesidad de crear una instancia. Este método toma dos coordenadas de tipo `double` , las redondea al entero más cercano utilizando el método `Math.round` y luego crea y devuelve una nueva instancia de `Punto` con las coordenadas redondeadas.

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

A continuación se muestra cómo se puede modificar la implementación de la clase `Punto` para utilizar un array interno de dos posiciones en lugar de dos atributos `double` , manteniendo la misma interfaz pública:

```

public class Punto {
    private double[] coordenadas;

    public Punto(double x, double y) {
        this.coordenadas = new double[2];
        this.coordenadas[0] = x;
        this.coordenadas[1] = y;
    }

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

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

    public void setX(double x) {
        this.coordenadas[0] = x;
    }

    public void setY(double y) {
        this.coordenadas[1] = y;
    }

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

```

En esta implementación, los atributos `x` e `y` se han reemplazado por un array privado `coordenadas` de dos posiciones. La interfaz pública de la clase no se ha modificado, ya que los métodos `getX`, `getY`, `setX`, `setY` y `calcularDistanciaAOrigen` siguen funcionando de la misma manera, pero ahora operan sobre el array interno.

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. La razón principal es que los métodos "getter" y "setter" permiten controlar el acceso y la modificación del atributo, lo que no es posible si el atributo es público. Por ejemplo, un método "setter" puede incluir validaciones para asegurarse de que el nuevo valor sea válido antes de asignarlo al atributo.

La convención más habitual en la Programación Orientada a Objetos es declarar los atributos como privados y proporcionar métodos "getter" y "setter" públicos solo cuando sea necesario. Esto sigue el principio de ocultación de información, que busca proteger los datos internos de una clase y garantizar que solo puedan ser modificados de manera controlada.

Este enfoque está relacionado con las invariantes de clase, ya que al restringir el acceso directo a los atributos, se puede garantizar que las invariantes se mantengan. Los métodos "setter" pueden incluir lógica para verificar que cualquier cambio en los atributos no viole las reglas o condiciones que definen un estado válido para la clase.

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 su estado no puede cambiar después de ser creada. Esto significa que todos los atributos de la clase son finales (es decir, no pueden ser modificados) y no se proporcionan métodos que permitan alterar su estado interno. Un ejemplo típico de una clase inmutable en Java es la clase `String`.

Un método modificador es cualquier método que cambia el estado interno de un objeto. Aunque los métodos "setter" son un tipo común de método modificador, no todos los métodos modificadores son "setters". Por ejemplo, un método que agrega un elemento a una lista interna o que incrementa un contador también sería un método modificador.

Las clases inmutables tienen varias ventajas. Son más fáciles de razonar, ya que su estado no cambia, lo que reduce la posibilidad de errores relacionados con modificaciones inesperadas. También son inherentemente seguras para su uso en entornos concurrentes, ya que múltiples hilos pueden acceder a ellas sin riesgo de interferencias. Sin embargo, su principal desventaja es que pueden generar más objetos, lo que podría afectar el rendimiento en ciertos casos.

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

Respuesta

No es recomendable incluir métodos "setter" siempre y como convención. Los métodos "setter" solo deben incluirse cuando sea necesario permitir que el estado de un objeto cambie después de ser creado. En muchos casos, es preferible diseñar clases inmutables o limitar los cambios al estado interno de un objeto para garantizar su consistencia y proteger las invariantes de clase.

Incluir métodos "setter" indiscriminadamente puede llevar a un diseño menos robusto, ya que aumenta el riesgo de que el estado del objeto sea modificado de manera incorrecta o inesperada. En su lugar, se recomienda analizar cuidadosamente los requisitos de la clase y proporcionar métodos "setter" solo cuando sea imprescindible.

La decisión de incluir métodos "setter" debe estar alineada con los principios de encapsulación y ocultación de información, que buscan proteger los datos internos de una clase y garantizar que cualquier modificación se realice de manera controlada.

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 estado no puede cambiar después de ser creado. Cada vez que se realiza una operación que parece modificar una cadena, como la concatenación, en realidad se crea un nuevo objeto `String` con el resultado de la operación.

Por ejemplo, al concatenar dos cadenas, se genera un nuevo objeto `String` que contiene el resultado de la concatenación, mientras que los objetos originales permanecen sin cambios. Esto puede ser ineficiente si se realizan muchas concatenaciones, ya que se crean múltiples objetos intermedios.

Para operaciones que implican concatenar muchas cadenas, se recomienda utilizar la clase `StringBuilder` o `StringBuffer`. Estas clases son mutables y están diseñadas específicamente para manejar modificaciones frecuentes de cadenas de manera eficiente. Por ejemplo:

```
StringBuilder sb = new StringBuilder();
sb.append("Hola");
sb.append(" ");
sb.append("Mundo");
String resultado = sb.toString();
```

En este caso, se evita la creación de múltiples objetos intermedios, lo que mejora el rendimiento.

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 de una misma clase se pueden comparar por su contenido o por su identidad, dependiendo del contexto y de cómo se implemente la comparación. La identidad se refiere a si dos referencias apuntan al mismo objeto en memoria, mientras que el contenido se refiere a si los valores de los atributos de los objetos son iguales.

En Java, el método `equals` se utiliza para comparar el contenido de dos objetos. Por defecto, el método `equals` heredado de la clase `Object` compara la identidad de los objetos, es decir, si las referencias apuntan al mismo objeto. Para comparar el contenido, es necesario sobrescribir el método `equals` en la clase y definir la lógica de comparación adecuada.

Por ejemplo, para comparar dos cadenas en Java, se debe utilizar el método `equals` en lugar del operador `==`, ya que este último compara la identidad y no el contenido:

```
String cadena1 = "Hola";
String cadena2 = new String("Hola");

if (cadena1.equals(cadena2)) {
    System.out.println("Las cadenas son iguales por contenido.");
}
```

En este ejemplo, `cadena1` y `cadena2` tienen el mismo contenido, por lo que el método `equals` devuelve `true`, aunque sean objetos diferentes en memoria.

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" en un lenguaje de programación orientado a objetos son clases que encapsulan un tipo de dato primitivo, proporcionando métodos y funcionalidades adicionales. En Java, las clases "wrapper" incluyen `Integer`, `Double`, `Boolean`, entre otras, que corresponden a los tipos primitivos `int`, `double`, `boolean`, etc.

El proceso de convertir un tipo primitivo en su clase "wrapper" se llama "autoboxing", y el proceso inverso se llama "unboxing". Ambos procesos son automáticos en Java desde la versión 5. Por ejemplo:

```
int numero = 10;
Integer numeroWrapper = numero; // Autoboxing
int numeroPrimitivo = numeroWrapper; // Unboxing
```

Las clases "wrapper" son útiles porque permiten tratar los tipos primitivos como objetos, lo que es necesario en ciertas situaciones, como al trabajar con colecciones genéricas (`List`, `Set`, etc.) que solo aceptan objetos.

No todos los lenguajes orientados a objetos tienen tipos primitivos y necesitan wrappers. Por ejemplo, en Python, todos los datos son objetos, por lo que no existe una distinción entre tipos primitivos y

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 conjunto de valores constantes predefinidos que representan un grupo de opciones relacionadas. En Java, los tipos enumerados se definen utilizando la palabra clave `enum` y, técnicamente, son clases especiales que pueden contener atributos, métodos y constructores.

Los enumerados en Java tienen varias ventajas en términos de encapsulación. Al ser clases, permiten agrupar datos y comportamientos relacionados en una sola unidad. Por ejemplo, se pueden definir métodos dentro del enumerado para realizar operaciones específicas relacionadas con los valores del mismo. Además, los enumerados garantizan que solo se puedan utilizar los valores predefinidos, lo que reduce el riesgo de errores y mejora la legibilidad del código.

Por ejemplo:

```
enum Dia {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;  
}
```

En este caso, `Dia` es un tipo enumerado que solo puede tomar uno de los valores definidos.

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

```
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;  
  
    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 enHemisferioNorte ? (ordinal >= 3 && ordinal <= 5) : (ordinal >= 9 && ordinal <= 11)  
    }  
  
    public boolean esDeVerano(boolean enHemisferioNorte) {  
        return enHemisferioNorte ? (ordinal >= 6 && ordinal <= 8) : (ordinal >= 12 || ordinal <= 2)  
    }  
  
    public boolean esDeOtoño(boolean enHemisferioNorte) {  
        return enHemisferioNorte ? (ordinal >= 9 && ordinal <= 11) : (ordinal >= 3 && ordinal <= 5)  
    }  
  
    public boolean esDeInvierno(boolean enHemisferioNorte) {  
        return enHemisferioNorte ? (ordinal >= 12 || ordinal <= 2) : (ordinal >= 6 && ordinal <= 8)  
    }  
}
```

```
        return enHemisferioNorte ? (ordinal >= 9 && ordinal <= 11) : (ordinal >= 3 && ordinal <= 5)
    }

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