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** busca agrupar los datos (atributos) y el código (métodos) que actúan sobre ellos en una unidad única llamada clase. La **ocultación de información** complementa esto haciendo que ciertos detalles internos no sean accesibles directamente desde fuera de la clase. 🔒

Ventajas principales de la ocultación de información:

- 🛡️ **Protección de invariantes:** Los atributos se modifican únicamente a través de métodos que validan los cambios
- 🔧 **Flexibilidad en cambios internos:** Se puede modificar la implementación interna sin afectar al código externo
- 🏷️ **Control de acceso:** Se decide explícitamente qué se expone y qué se oculta
- 🎯 **Interfaz clara:** Los usuarios de la clase solo ven lo necesario, simplificando su uso

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** es el conjunto de métodos y atributos que se han marcado como `public` en una clase, es decir, lo que el mundo exterior puede ver y utilizar. Es el "contrato" que la clase ofrece a otros desarrolladores. 📢

La interfaz pública está intimamente ligada a la ocultación de información: mientras que la interfaz pública expone **qué** hace la clase, los detalles privados ocultan **cómo** lo hace. Los atributos privados y los métodos privados forman la **interfaz privada**, que es interna y no debe ser accedida desde fuera. De esta forma, la clase mantiene el control sobre su estado mientras proporciona una forma segura de interacción. 🔐

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 es como un "contrato" con quien la usa. 📝 Una vez publicada, cualquier código externo puede depender de ella, por lo que cambiarla posteriormente puede **romper el código que la utiliza**. Si se modifica un método público o se elimina un atributo público, todos los clientes de esa clase pueden verse afectados.

No es fácil cambiarla. ✖ Una vez que se expone públicamente, se asume estabilidad. Por eso es fundamental diseñar cuidadosamente qué se expone en la interfaz pública desde el principio, considerando

solo lo esencial. Cambios en la interfaz requieren sincronizar cambios en todo el código que dependa de ella, lo que puede ser una pesadilla en proyectos grandes. 😰

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

Respuesta:

Las **invariantes de clase** son condiciones que deben mantenerse siempre como verdaderas durante toda la vida del objeto. 📌 Por ejemplo, en una clase `Cuenta`, la invariante podría ser que el saldo nunca sea negativo, o en un `Círculo`, que el radio siempre sea mayor que cero. Son las "reglas" que garantizan que el objeto siempre este en un estado válido y coherente.

La ocultación de información protege estas invariantes de forma crucial. 🛡️ Si los atributos fueran públicos, el código externo podría modificarlos directamente, violando las invariantes (por ejemplo, asignando un radio negativo). Al mantener los atributos como privados, la clase controla **todas** las modificaciones a través de métodos que pueden validar y asegurar que las invariantes se cumplan siempre. Esto es un superpoder de la encapsulación: garantizar la integridad del objeto. ✅

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:

```java
public class Punto {
    // Atributos privados (ocultos) 🔒
    private double x;
    private double y;

    // Constructor público (interfaz pública)
    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Método público (interfaz pública)
    public double calcularDistanciaAOrigen() {
        return Math.sqrt(x * x + y * y);
    }
}
```

`public` 📢 significa que el método o atributo es accesible desde cualquier otra clase. En este ejemplo, el constructor y `calcularDistanciaAOrigen()` son públicos, por lo que cualquiera puede crear un `Punto` y llamar al método.

`private` 🔒 significa que el atributo o método solo es accesible dentro de la propia clase. Los atributos `x` e `y` son privados, por lo que solo métodos dentro de `Punto` pueden acceder o modificarlos directamente.

Interfaz pública de `Punto`: El constructor `Punto(double, double)` y el método `double calcularDistanciaAOrigen()`. Solo estos son visibles y utilizables desde fuera de la clase.

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

Respuesta:

Los modificadores `public` y `private` en Java se pueden aplicar a:

- 🔹 **Atributos de instancia:** `private double x;` o `public int contador;`
- 🔹 **Métodos:** `public void saludar()` o `private void validar()`
- 🔹 **Constructores:** `public Punto(...)` o `private Punto(...)`
- 🔹 **Clases internas (inner classes):** Una clase anidada dentro de otra

No se pueden aplicar directamente a variables locales (variables dentro de métodos), ya que estas tienen un alcance limitado al método en el que se declaran.

Es importante notar que los modificadores solo controlan la **visibilidad** desde fuera de la clase, pero dentro de la misma clase, todos los miembros (públicos y privados) son siempre accesibles. 🎯

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:

Aunque pública y privada son los extremos más comunes, existen **niveles intermedios de visibilidad** en la mayoría de lenguajes orientados a objetos. 📊

En **Java** existen **4 niveles de visibilidad:**

- 🌍 **`public`:** Accesible desde cualquier clase en cualquier paquete
- 🏛️ **`protected`:** Accesible solo dentro del mismo paquete y por subclases (herencia)
- ⚪ **Sin modificador (package-private):** Accesible solo dentro del mismo paquete
- 🔒 **`private`:** Accesible solo dentro de la misma clase

En **otros lenguajes:** Varían según el diseño. Por ejemplo, C++ tiene `public`, `private` y `protected`, mientras que Python confía más en convenciones de nombres (como prefijo `_` para indicar "privado") que en restricciones del lenguaje. 🐍 Algunos lenguajes como C# ofrecen incluso más opciones como `internal` o `protected internal`.

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:

Respuesta: (a) Otras clases. 🎯 Los miembros privados están ocultos para otras clases, pero **no** para otras instancias de la misma clase. Una instancia de `Punto` puede acceder a los atributos privados de otra

instancia de `Punto`.

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

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

    // Este método puede acceder a los atributos privados de otro Punto ✅
    public double calcularDistanciaAPunto(Punto otro) {
        double dx = this.x - otro.x;  // Acceso a x privada de otra
instancia
        double dy = this.y - otro.y;  // Acceso a y privada de otra
instancia
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

La visibilidad `private` se define a nivel de **clase**, no a nivel de **instancia**. Por lo tanto, dentro de la clase, cualquier instancia puede acceder a los miembros privados de cualquier otra instancia de la misma clase. 🔓 Esto tiene sentido porque son todas "del mismo tipo" y se presume confianza entre ellas. Solo las clases externas son las que no pueden acceder.

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

Respuesta:

Los **getters** y **setters** son métodos públicos que permiten leer y modificar, respectivamente, los atributos privados de una clase de forma controlada. 🧭 Un getter devuelve el valor de un atributo, mientras que un setter permite asignarlo tras aplicar validaciones o lógica necesaria.

Su uso mantiene la ocultación de información: aunque el atributo esté privado, la clase decide cómo exponerlo y qué reglas aplicar al modificarlo. ✅ Esto facilita mantener invariantes y agregar lógica adicional (por ejemplo, comprobaciones o eventos) sin cambiar la interfaz pública.

Ejemplo breve:

```
public double getX() { return x; }
public void setX(double x) { if (x >= 0) this.x = x; }
```

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

Respuesta:

No exactamente. Cuando se dice que la ocultación mejora la "seguridad" del programa, se refiere principalmente a la **robustez** y **integridad** del estado interno, no a la seguridad frente a ataques externos. 🛡️

Ocultar atributos y validar cambios reduce errores, evita estados inconsistentes y facilita mantener invariantes; eso ayuda a prevenir fallos accidentales o abusos desde código legítimo. Para evitar ataques maliciosos (hacking) se requieren medidas adicionales: control de acceso, autenticación, autorización, cifrado y prácticas de seguridad específicas. 🔐

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 un atributo o método que pertenece a una instancia concreta de la clase; cada objeto tiene su propia copia del miembro. En cambio, un **miembro de clase** (o `static` en Java) pertenece a la clase en sí y se comparte entre todas las instancias. 🧩

Los miembros de clase también se pueden ocultar usando modificadores de visibilidad (`private`, `public`, etc.). Por ejemplo, `private static int contador;` mantiene el control centralizado sobre datos compartidos mientras evita accesos directos desde el exterior. Esto permite aplicar las mismas garantías de encapsulación y control de invariantes a los datos compartidos. 🔁

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

Respuesta:

Sí, tiene sentido en ciertos patrones de diseño. Un constructor `private` evita que el usuario cree instancias directamente; esto es útil para implementar **singleton**, **factorías** controladas o constructores con validación centralizada. 🔒

Al declarar constructores privados se obliga a utilizar métodos estáticos (factorías) o constructores internos que gestionen la creación, permitiendo controlar la creación de objetos, reusar instancias o aplicar validaciones antes de construir. ⚙️

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 marcan con la palabra clave `static`. Estos miembros son compartidos por todas las instancias de la clase. A continuación se muestra un ejemplo que añade contadores máximos a la clase `Punto` y los actualiza en el constructor:

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

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

Los atributos `maxX` y `maxY` son `static` y privados; se exponen mediante getters estáticos para mantener la encapsulación. 📈

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 crearRedondeado(double x, double y) {
    double rx = Math.rint(x); // redondeo al entero más cercano
    double ry = Math.rint(y);
    return new Punto(rx, ry);
}
```

Sí: se ha usado `static` porque la factoría pertenece a la clase y no a una instancia concreta. 🏭

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:

Se puede mantener la interfaz pública mientras la representación interna cambia a un array de dos `double`. A continuación se muestra una implementación compatible con los constructores y métodos anteriores:

```
public class Punto {
    private double[] coord = new double[2]; // coord[0] = x, coord[1] = y

    public Punto(double x, double y) {
```

```
        this.coord[0] = x;
        this.coord[1] = y;
    }

    public double calcularDistanciaAOrigen() {
        double x = coord[0];
        double y = coord[1];
        return Math.sqrt(x*x + y*y);
    }

    public double getX() { return coord[0]; }
    public double getY() { return coord[1]; }

    // Si existía calcularDistanciaAPunto, seguiría funcionando igual
    public double calcularDistanciaAPunto(Punto otro) {
        double dx = this.coord[0] - otro.coord[0];
        double dy = this.coord[1] - otro.coord[1];
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

La interfaz pública (constructores y métodos `getX`, `getY`, `calcularDistanciaAOrigen`, etc.) no cambia; solo cambia la representación interna. 🔁

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 pueda parecer redundante, no es recomendable declarar el atributo público si ya se va a proporcionar `getter` y `setter`. La convención habitual es declarar los atributos como `private` y exponer acceso mediante métodos. 🧭

Esto permite validar y controlar las modificaciones (por ejemplo, mantener invariantes) y cambiar la implementación interna sin romper clientes. Por tanto, la práctica protege las invariantes de clase y favorece la evolución del código. 💡

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 los objetos no pueden cambiar su estado una vez creados. 🔐 Es decir, no hay forma de modificar los atributos después de la construcción. Un método modificador es cualquier método que altera el estado interno del objeto; no es necesariamente un setter.

Un setter es una forma de método modificador, pero no todos los métodos modificadores son setters. Por ejemplo, un método `incrementarContador()` es un modificador pero no es un setter. ✏

Ventajas de la inmutabilidad: 🎯

- Seguridad en concurrencia: varios hilos pueden compartir el objeto sin sincronización
- Facilidad de razonamiento: el estado no cambia sorpresivamente
- Uso como claves en colecciones (HashMap, HashSet)
- La clase `String` en Java es inmutable por razones de seguridad y rendimiento

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

Respuesta:

No, no es recomendable incluir setters por defecto en todas las clases. 🚫 Solo deben añadirse cuando tenga sentido permitir la modificación de un atributo. Muchas clases se benefician de ser completamente inmutables (sin setters) o de tener setters altamente selectivos.

La convención moderna es: incluir solo los setters que sean necesarios para el negocio, con validaciones fuertes. Esto protege las invariantes y permite evolucionar la clase sin exponer detalles innecesarios. ⚡

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**. 🔒 Cuando se concatenan dos cadenas usando el operador `+` o el método `concat()`, en lugar de modificar la cadena original, se crea un **nuevo objeto `String`** con el resultado. Esto es costoso en términos de memoria y rendimiento.

Si se va a realizar concatenación múltiple en un bucle o proceso iterativo, se debe usar la clase **`StringBuilder`**, que es mutable y diseñada para acumular cambios eficientemente. 📈

```
// ✖ Ineficiente: crea muchas cadenas intermedias
String resultado = "";
for (int i = 0; i < 1000; i++) {
    resultado += "línea " + i + "\n";
}

// ✔ Eficiente: usa StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append("línea ").append(i).append("\n");
}
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, por defecto, los objetos se comparan por **identidad** (si son el mismo objeto en memoria), usando el operador `==`. Sin embargo, muchas veces interesa comparar por **contenido** (si tienen los mismos valores). 🎯

El método **`equals()`** permite comparar objetos por contenido. Por defecto (en la clase `Object`), `equals()` hace lo mismo que `==`: compara identidades. Sin embargo, muchas clases lo sobrescriben para comparar contenido. Por ejemplo, `String` lo sobrescribe para comparar carácter por carácter. 📊

```
String a = "hola";
String b = new String("hola");

a == b            // false: diferentes objetos en memoria
a.equals(b)       // true: mismo contenido
```

Para comparar dos cadenas en Java, siempre se debe usar **`equals()`** (o `equalsIgnoreCase()` para ignorar mayúsculas), nunca `==`. ✅

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:

Clases **wrapper** (envoltorio) son clases que envuelven tipos primitivos para permitir tratarlos como objetos. 📦 Por ejemplo, `Integer` envuelve `int`, `Double` envuelve `double`, etc. Esto permite usar primitivos en contextos que requieren objetos (colecciones genéricas, métodos que aceptan `Object`, etc.).

En Java, la conversión entre primitivos y wrappers es **automática**: el compilador la hace mediante *autoboxing* (primitivo → wrapper) y *unboxing* (wrapper → primitivo). 🔄

Ventajas:

- Permite usar tipos primitivos en colecciones genéricas (`List<Integer>` en vez de `List<Object>`)
- Proporciona métodos útiles (`Integer.parseInt()`, `Double.MAX_VALUE`, etc.)
- Facilita el polimorfismo con tipos numéricos

No todos los lenguajes OO poseen tipos primitivos distintos de objetos. Por ejemplo, Python y Ruby tratan todo como objetos, por lo que no necesitan wrappers explícitos. 🌱

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** (enum) es un tipo que define un conjunto finito y bien conocido de valores constantes. 🏷️ Por ejemplo, en lugar de usar `int` para representar un mes (riesgo de valores inválidos como 13), se define un enum `Mes` con doce valores: `ENERO`, `FEBRERO`, ..., `DICIEMBRE`.

En Java, un enum **es una clase especial** que hereda de `java.lang.Enum`. 🎯 Los enumerados ofrecen grandes ventajas en términos de encapsulación:

- ✅ **Seguridad de tipos:** Solo se pueden asignar valores válidos; el compilador evita errores
- 🔒 **Encapsulación fuerte:** Los valores están predefinidos y no pueden cambiar
- 📊 **Métodos asociados:** Se pueden añadir métodos y atributos específicos a cada instancia
- 🛡️ **Protección:** Las enumeraciones son implícitamente inmutables y thread-safe

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.

Respuesta:

24. Añade a la clase `Mes` del ejercicio anterior 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(1, 31),
    FEBRERO(2, 28),
    MARZO(3, 31),
    ABRIL(4, 30),
    MAYO(5, 31),
    JUNIO(6, 30),
    JULIO(7, 31),
    AGOSTO(8, 31),
    SEPTIEMBRE(9, 30),
    OCTUBRE(10, 31),
    NOVIEMBRE(11, 30),
    DICIEMBRE(12, 31);

    private final int ordinal;  // 1-12
    private final int dias;

    // Constructor privado
```

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

    // Métodos públicos
    public int getOrdinal() {
        return ordinal;
    }

    public int getDias() {
        return dias;
    }

    public boolean esDePrimavera(boolean enHemisferioNorte) {
        if (enHemisferioNorte) {
            return this == MARZO || this == ABRIL || this == MAYO;
        } else {
            return this == SEPTIEMBRE || this == OCTUBRE || this ==
NOVIEMBRE;
        }
    }

    public boolean esDeVerano(boolean enHemisferioNorte) {
        if (enHemisferioNorte) {
            return this == JUNIO || this == JULIO || this == AGOSTO;
        } else {
            return this == DICIEMBRE || this == ENERO || this == FEBRERO;
        }
    }

    public boolean esDeOtoño(boolean enHemisferioNorte) {
        if (enHemisferioNorte) {
            return this == SEPTIEMBRE || this == OCTUBRE || this ==
NOVIEMBRE;
        } else {
            return this == MARZO || this == ABRIL || this == MAYO;
        }
    }

    public boolean esDeInvierno(boolean enHemisferioNorte) {
        if (enHemisferioNorte) {
            return this == DICIEMBRE || this == ENERO || this == FEBRERO;
        } else {
            return this == JUNIO || this == JULIO || this == AGOSTO;
        }
    }
}
```

Explicación: El enum `Mes` utiliza un constructor privado para inicializar los atributos `ordinal` y `dias` de cada instancia. 🎭 Los métodos de estación devuelven `true` si el mes pertenece a esa estación en el

hemisferio indicado. La encapsulación es máxima: los valores están predefinidos y validados en tiempo de compilación. ✅