# 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.

## Propósito de la Encapsulación y la Ocultación

En el paradigma de la POO, la encapsulación y la ocultación de información buscan crear componentes de software autónomos y seguros. El objetivo principal es separar la interfaz (lo que el objeto hace) de la implementación (cómo lo hace). Mientras que en lenguajes procedimentales como C el acceso a los miembros de una struct es directo, aquí se busca que el estado interno del objeto sea inaccesible para el exterior, salvo por los canales estrictamente autorizados.

Este aislamiento garantiza que el objeto tenga el control total sobre sus propios datos. Al impedir el acceso directo, se evita que el código cliente (otras partes del programa) dependa de detalles internos que podrían cambiar en el futuro. Esto establece un "contrato" de uso: el objeto promete realizar ciertas acciones a través de sus métodos públicos, pero se reserva el derecho de gestionar su memoria y lógica interna de la forma más eficiente posible.

Ventajas de la Ocultación de Información
La restricción de visibilidad mediante modificadores como private aporta beneficios estructurales que mejoran la calidad del software a largo plazo:

Protección de la Integridad: Evita que los datos tomen valores inválidos o incoherentes, ya que cualquier modificación debe pasar por métodos de control (setters) que validan la entrada.

Independencia de Implementación: Permite sustituir la lógica interna o las estructuras de datos de una clase sin que los programas que la utilizan sufran errores o requieran cambios.

Reducción de la Complejidad: El programador que utiliza la clase no necesita comprender cientos de líneas de lógica interna; solo requiere conocer la firma de los métodos públicos disponibles.

Facilidad de Debugging: Al centralizar el acceso a las variables en puntos específicos, es mucho más sencillo rastrear en qué momento y por qué un atributo ha cambiado de valor.

---

## 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.

La **interfaz pública** de una clase o un objeto está formada por el conjunto de métodos y constantes que son accesibles desde el exterior (normalmente definidos con el modificador `public`). Es, en esencia, el "contrato" que ofrece la clase al resto del programa; define *qué* operaciones se pueden realizar con el objeto, pero no *cómo* se llevan a cabo internamente.

**Relación con la ocultación de información (Encapsulamiento)**

La interfaz pública es la cara visible que hace posible la ocultación de información. El estado interno del objeto (sus atributos o variables) y los detalles complejos de su funcionamiento se mantienen ocultos (marcados como `private` o `protected`). La única forma de interactuar con esos datos protegidos es a través de la interfaz pública, lo que garantiza que el objeto mantenga su integridad y que ninguna otra parte del código pueda alterar su estado de forma indebida o saltándose las reglas del negocio.

---

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

**Importancia del diseño de la interfaz pública:**

La interfaz pública es el "contrato" que tu clase firma con el resto del programa. Hay que diseñarla con cuidado porque es la única parte de la clase que los demás objetos pueden ver y utilizar. Un buen diseño asegura que el sistema sea fácil de entender y que las interacciones entre objetos sean predecibles.

**¿Es fácil cambiarla?**

No, no es fácil. Una vez que otros programadores o módulos de código empiezan a usar tu clase, cualquier cambio en la interfaz pública (como cambiar el nombre de un método o los parámetros que recibe) "rompe" todo el código que dependía de ella.

El principio de encapsulación: Por eso es vital exponer solo lo necesario. Lo que es privado se puede cambiar fácilmente sin afectar a nadie; lo que es público es casi "para siempre".
Efecto dominó: Un cambio pequeño en una interfaz pública mal diseñada puede obligar a refactorizar cientos de líneas de código en otras partes del proyecto.

---

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

Las **invariantes de clase** son condiciones lógicas o reglas sobre el estado de un objeto que deben mantenerse siempre verdaderas durante toda la vida útil de dicho objeto (desde que termina de ejecutarse su constructor hasta que es destruido). Es decir, son las reglas de negocio que garantizan que un objeto nunca se encuentre en un estado inválido, corrupto o inconsistente.

**¿Por qué la ocultación de información nos ayuda?**

La ocultación de información (encapsulamiento) es el mecanismo clave para proteger estas invariantes. Al ocultar los atributos internos (declarándolos como `private` o `protected`), evitamos que el código externo pueda modificar el estado del objeto libremente y saltarse las reglas.

Si los atributos fueran públicos, cualquier parte del programa podría alterarlos e introducir un valor inválido, rompiendo la invariante. Al forzar que todas las modificaciones pasen por los métodos de la interfaz pública (constructores, *setters* o métodos de acción), la clase tiene el control absoluto para validar los datos antes de aplicar cualquier cambio, asegurando que la invariante se cumpla siempre.

---

## 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`?

```
public class Punto {
    // Ocultación de información: atributos privados
    private double x;
    private double y;

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

    // Métodos de acceso (Getters y Setters)
    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    // Método de cálculo
    public double calcularDistanciaAOrigen() {
        // El origen es (0,0), usamos el Teorema de Pitágoras
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
    }
}
```

## ¿Cuál es la interfaz pública de la clase `Punto`?

La interfaz pública está formada por todos aquellos elementos a los que se puede acceder desde fuera de la clase. En este caso, la conforman:

- **El constructor:** `Punto(double x, double y)`
- **Los métodos consultores (getters):** `getX()`, `getY()`
- **Los métodos modificadores (setters):** `setX(double x)`, `setY(double y)`
- **El método de acción:** `calcularDistanciaAOrigen()`

Estos métodos definen qué se puede hacer con un objeto de tipo `Punto` sin exponer cómo están almacenados internamente sus datos.

## ¿Qué significa `public` y `private`?

Son modificadores de acceso que determinan la visibilidad de los componentes de una clase:

- `private` **(Privado):** Significa que el atributo o método solo es visible y accesible desde el interior de la propia clase donde ha sido definido. Es el pilar de la ocultación de información, ya que impide que el estado del objeto sea alterado directamente desde el exterior.
- `public` **(Público):** Significa que el atributo o método es visible y accesible desde cualquier otra clase del programa. Se utiliza para definir la interfaz pública y permitir la interacción controlada con el objeto.

---

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

---

En Java, los modificadores de acceso como `public` o `private` se utilizan para definir el nivel de visibilidad de los componentes de tu código. Se pueden aplicar a los siguientes elementos:

1. **Atributos (campos o variables de instancia):** Las variables que definen el estado de un objeto. Declararlos como `private` es la base del principio de ocultación de información.
2. **Métodos:** Las funciones que definen el comportamiento. Los métodos `public` forman la interfaz pública (lo que el mundo exterior puede usar), mientras que los `private` son funciones de apoyo de uso exclusivamente interno.
3. **Constructores:** Determinan desde dónde se puede crear una instancia de la clase. (Por ejemplo, un constructor `private` impide que otras clases usen la palabra clave `new` para instanciarla, lo cual es muy útil en patrones de diseño como *Singleton*).
4. **Clases e Interfaces:** * Las clases **externas** (de nivel superior o *top-level*) **solo** pueden ser `public` o tener visibilidad por defecto (sin modificador). No pueden ser `private`.
    - Las clases **internas** (anidadas o *nested classes*) **sí** pueden declararse como `private`, comportándose como un miembro interno más de la clase que las envuelve.

**Resumen de aplicación en Java:**

| Elemento a modificar | `public` | `private` | Observaciones |
|---|---|---|---|
| **Clase externa** (*Top-level*) | Sí | No | Si no es `public`, su acceso es *package-private* por defecto. |
| **Clase interna** (*Nested*) | Sí | Sí | Se gestiona con las mismas reglas que un atributo o método. |
| **Atributos / Variables** | Sí | Sí | Ocultarlos con `private` garantiza la integridad del estado. |
| **Métodos** | Sí | Sí | Exponer lo necesario con `public`, ocultar la lógica con `private`. |
| **Constructores** | Sí | Sí | Controla de forma estricta la creación de objetos. |

---

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

**Tipos de visibilidad más allá de `public` y `private`:**

Sí, existen otros niveles de visibilidad que permiten un control más fino sobre quién puede acceder a los miembros de una clase. Los más comunes son el acceso de **paquete** (package) y el acceso **protegido** (protected).

### ¿Qué ocurre en Java?

Java maneja cuatro niveles de visibilidad (modificadores de acceso):

1. `public`: El miembro es accesible desde cualquier clase en cualquier paquete.
2. `protected`: El miembro es accesible dentro del mismo paquete **y también por subclases** (herencia), incluso si están en paquetes diferentes.
3. `default` **(sin modificador)**: También llamado *package-private*. El miembro solo es accesible por clases que estén en el **mismo paquete**. No es accesible desde fuera del paquete, ni siquiera por subclases.
4. `private`: El miembro solo es accesible dentro de la propia clase.

| Modificador | Clase | Paquete | Subclase | Global |
|---|---|---|---|---|
| `public` | Sí | Sí | Sí | Sí |
| `protected` | Sí | Sí | Sí | No |
| `default` | Sí | Sí | No | No |
| `private` | Sí | No | No | No |

### ¿Y en otros lenguajes?

La gestión de la visibilidad varía significativamente según el lenguaje:

- **C++**: Tiene `public`, `protected` y `private`, pero añade el concepto de `friend` (clases o funciones amigas), que permite a elementos externos acceder a miembros privados de una clase específica.
- **Python**: No tiene modificadores de acceso reales (todo es técnicamente público). Sin embargo, se usa una convención: un guion bajo ( `_variable` ) indica que es para uso interno, y dos guiones bajos ( `__variable` ) activan el *name mangling* para evitar colisiones accidentales.
- **C#**: Incluye los niveles de Java pero añade `internal`, que limita el acceso al mismo "ensamblado" (archivo .dll o .exe), y `protected internal`.

- **Swift**: Utiliza niveles como `open`, `public`, `internal`, `fileprivate` y `private`, enfocándose mucho en la estructura de los módulos y archivos físicos.

---

## 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.

---

La respuesta correcta es la **(a) otras clases**.

En Java, los modificadores de acceso (como `private`) actúan a **nivel de clase**, no a nivel de objeto o instancia. Esto significa que un objeto puede acceder directamente a los atributos y métodos privados de *cualquier otro objeto*, siempre y cuando ambos pertenezcan a la **misma clase**.

**Ejemplo con la clase `Punto`**

Si añadimos el método `calcularDistanciaAPunto` a nuestra clase, podemos acceder directamente a las coordenadas `x` e `y` del objeto `otro`, a pesar de que están declaradas como `private`:

```
// Implementación en Java (JDK 25)
public class Punto {
    private double x;
    private double y;

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

    // Método que calcula la distancia entre la instancia actual (this) y otra (otro)
    public double calcularDistanciaAPunto(Punto otro) {
```

```
        // ¡Atención aquí! Podemos acceder a otro.x y otro.y directamente,
        // sin necesidad de usar getters (otro.getX()), porque estamos dentro de la clase
        double diferenciaX = this.x - otro.x;
        double diferenciaY = this.y - otro.y;

        return Math.sqrt(Math.pow(diferenciaX, 2) + Math.pow(diferenciaY, 2));
    }
}
```

**Explicación de la respuesta**

Cuando el compilador de Java evalúa si es legal acceder a otro.x, la única comprobación que hace es mirar en qué clase está escrito el código que intenta el acceso. Como el método calcularDistanciaAPunto forma parte del código de la clase Punto, tiene permisos absolutos para ver y modificar cualquier miembro private de cualquier instancia de la clase Punto que reciba.

La ocultación de información con *private* sirve para establecer un muro de protección frente a otras clases del sistema (opción a). Sin embargo, el lenguaje asume que una clase tiene un conocimiento íntimo de su propia estructura interna, por lo que permite que las instancias de una misma familia compartan sus "secretos" entre sí.

---

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

**Definición:**

Los métodos **getter** y **setter** son métodos públicos que se utilizan para acceder y modificar el valor de los atributos privados de una clase. Son la herramienta principal para implementar la **ocultación de información** y el **encapsulamiento**.

**1. El método Getter (Accesor):**

Su función es **recuperar** o leer el valor de un atributo privado.

- **Sintaxis**: Por convención, empiezan con la palabra `get` seguida del nombre del atributo (ej. `getEdad()`).

- **Retorno**: Siempre devuelven un valor del mismo tipo que el atributo.

**2. El método Setter (Mutador):**

Su función es **establecer** o modificar el valor de un atributo privado.

- **Sintaxis**: Empiezan con la palabra `set` seguida del nombre del atributo (ej. `setEdad(int nuevaEdad)`).
- **Validación**: Su mayor ventaja es que permiten añadir lógica de control. Por ejemplo, un `setEdad` puede verificar que el número no sea negativo antes de asignarlo.

**¿Por qué usarlos en lugar de hacer los atributos públicos?**

- **Control total**: Puedes hacer que un atributo sea de "solo lectura" si creas el getter pero no el setter.
- **Flexibilidad**: Si en el futuro decides cambiar cómo se almacena un dato internamente, solo tienes que cambiar el código dentro del getter o setter, y el resto del programa no se enterará.
- **Validación de datos**: Evitas que la clase entre en un estado inválido (como un objeto `Persona` con altura de -2 metros).

**Ejemplo rápido en Java:**

```
private int velocidad;

// Getter
public int getVelocidad() {
    return velocidad;
}

// Setter con validación
public void setVelocidad(int v) {
    if (v >= 0) {
        this.velocidad = v;
    } else {
        System.out.println("Error: La velocidad no puede ser negativa");
    }
}
```

---

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

**No, no se refiere a la "ciberseguridad" externa:**

Es un error común confundir la **seguridad de tipos y de diseño** con la seguridad contra ataques informáticos (como malware o inyecciones de código). En POO, la "seguridad" mediante la ocultación de información se refiere a la **integridad del estado del objeto**.

**¿A qué nos referimos realmente?**

Nos referimos a evitar que el programa falle debido a usos incorrectos o accidentales por parte del propio programador o de otros componentes del software:

- **Protección contra errores accidentales**: Evita que una clase externa modifique un atributo de forma que deje al objeto en un estado inconsistente o absurdo (ej. poner un saldo negativo en una cuenta bancaria que no lo permite).
- **Blindaje de la lógica interna**: Al hacer los datos privados, obligas a que cualquier cambio pase por los métodos ( `setters` ), donde puedes aplicar reglas de validación.
- **Reducción del acoplamiento**: Si el "mundo exterior" no conoce los detalles internos, es mucho más difícil que un cambio en una parte del código rompa otras partes de forma inesperada.

**Diferencia clave:**

1. **Seguridad en POO**: Es una "red de seguridad" para el desarrollo. Asegura que el objeto siempre se comporte como debe y que nadie "toque" sus engranajes internos sin permiso.
2. **Ciberseguridad**: Se encarga de proteger el sistema contra intrusos, robo de datos o ataques malintencionados. Aunque un código bien encapsulado es más robusto y fácil de auditar, la ocultación por sí sola no detiene a un hacker profesional.

> ***En resumen**: La ocultación de información hace que el programa sea **robusto y fiable**, no necesariamente "impenetrable" desde un punto de vista informático.*

---

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

**Diferencia fundamental:**

La distinción radica en a quién "pertenece" el dato o el comportamiento: si a cada objeto individual o a la plantilla general (la clase).

- **Miembro de instancia**:
  - Pertenece a cada objeto creado.
  - Cada objeto tiene su propia copia de los atributos. Si cambias el valor en el `Objeto A`, el `Objeto B` no se ve afectado.
  - Ejemplo: El nombre de una persona o las coordenadas `x` e `y` de un punto.
- **Miembro de clase (estático)**:
  - Se declara con la palabra clave `static` en Java.
  - Pertenece a la clase en sí, no a los objetos.
  - Existe una única copia compartida por todas las instancias. Si un objeto modifica un miembro estático, el cambio es visible para todos los demás.
  - Ejemplo: Un contador que lleve la cuenta de cuántos objetos `Punto` se han creado en total.

**¿Los miembros de clase también se pueden ocultar?**

**Sí, por supuesto.** Los miembros de clase (atributos y métodos `static`) pueden y suelen ocultarse utilizando los mismos modificadores de acceso que los miembros de instancia.

- **Atributos estáticos privados**: Se utilizan para guardar información global de la clase que no queremos que nadie externo modifique directamente (ej. una constante de configuración interna o un contador de instancias).
- **Métodos estáticos privados**: Son métodos de utilidad que la clase usa internamente para realizar cálculos compartidos, pero que no forman parte de la interfaz pública.

**Ejemplo comparativo en Java:**

```
public class Circulo {
    // Miembro de clase oculto (estático y privado)
```

```
    private static int numeroDeCirculos = 0;

    // Miembro de instancia oculto (privado)
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
        numeroDeCirculos++; // Incrementa el contador compartido
    }

    // Método de clase público para acceder al miembro oculto
    public static int getTotalCirculos() {
        return numeroDeCirculos;
    }
}
```

---

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

**¿Tiene sentido?**

**Sí, tiene mucho sentido.** Aunque a primera vista parezca contradictorio (ya que un constructor privado impide que uses `new Clase()` desde fuera), es una técnica fundamental en patrones de diseño avanzados y para el control estricto de la creación de objetos.

**Casos de uso principales:**

- **Patrón Singleton**: Se asegura de que **solo exista una única instancia** de la clase en todo el programa. La propia clase crea esa instancia internamente y la sirve a través de un método público.
- **Clases de utilidad**: Clases que solo contienen métodos estáticos (como la clase `Math` de Java). Al hacer el constructor privado, evitas que alguien malgaste memoria creando un objeto que no tiene sentido que exista.
- **Métodos de factoría (Factory Methods)**: Cuando quieres obligar al usuario a crear objetos a través de métodos específicos que pueden tener nombres más descriptivos que el propio constructor o que pueden reciclar objetos ya creados.

- **Clases de constantes**: Para evitar que se instancien clases que solo sirven para agrupar valores fijos ( `public static final` ).

**Ejemplo rápido:**

```
public class Configurador {
    // Constructor privado: nadie fuera de esta clase puede hacer "new Configurador()"
    private Configurador() { }

    public static void cargarConfiguracion() {
        // Lógica estática...
    }
}
```

---

## 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.

En Java, los **miembros de clase** (tanto atributos como métodos) se indican utilizando la palabra reservada `static` .

A diferencia de los miembros de instancia (que pertenecen a cada objeto individual), los miembros estáticos pertenecen a la clase en sí. Esto significa que **todas las instancias de esa clase comparten la misma variable en memoria**. Si un objeto modifica una variable o atributo `static` , el cambio será visible para todos los demás objetos de esa misma clase.

### Ejemplo en la clase `Punto`

Para rastrear los valores máximos históricos de `x` e `y` en todos los puntos creados, definimos dos atributos `static` . También es buena práctica crear métodos `static` para consultarlos sin necesidad de tener un objeto instanciado.

```java
// Implementación en Java (JDK 25)
public class Punto {
    // Miembros de instancia (cada punto tiene los suyos de forma independiente)
    private double x;
    private double y;

    // Miembros de clase (compartidos por absolutamente todos los puntos)
    // Inicializamos con el valor mínimo posible para que el primer punto lo sobreescriba
    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);
    }

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

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

    // Método de clase privado para actualizar los valores máximos compartidos
    private static void actualizarMaximos(double nuevaX, double nuevaY) {
        if (nuevaX > maxX) {
            maxX = nuevaX;
        }
        if (nuevaY > maxY) {
            maxY = nuevaY;
        }
    }

    // Métodos de clase públicos para consultar los máximos desde el exterior
    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`?

Aquí tienes únicamente el código del método solicitado:

```
// Método factoría en Java (JDK 25)
public static Punto crearPuntoRedondeado(double x, double y) {
    // Math.round devuelve un 'long' que se convierte implícitamente al 'double' que esper
    return new Punto(Math.round(x), Math.round(y));
}
```

##¿Has usado static?
Sí, es estrictamente necesario usar static.

Un método factoría (o factory method) tiene como objetivo principal construir y devolver una nueva instancia de la clase. Si el método no fuera estático (es decir, si fuera un método de instancia), necesitarías tener un objeto Punto ya creado previamente en memoria para poder invocarlo. Al declararlo como static, el método pertenece a la clase en sí y puedes llamarlo directamente sin instanciarla previamente, usando la sintaxis: Punto miPunto = Punto.crearPuntoRedondeado(3.8, 5.1);.

---

## 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.

---

Al utilizar el encapsulamiento y la ocultación de información, podemos cambiar por completo la estructura interna de los datos sin que el código externo que usa nuestra clase se vea afectado, ya que la interfaz pública (constructores y métodos) se mantiene exactamente igual.

Aquí tienes la nueva implementación utilizando un array:

```
// Implementación en Java (JDK 25)
public class Punto {
    // El estado interno cambia: ahora usamos un array en lugar de dos variables sueltas
    private double[] coordenadas;

    // La interfaz pública se mantiene INTACTA

    // El constructor sigue pidiendo dos double independientes
    public Punto(double x, double y) {
        // Inicializamos el array con los valores recibidos
        this.coordenadas = new double[]{x, y};
    }

    // Los getters devuelven double, extrayéndolos del array internamente
    public double getX() {
        return this.coordenadas[0];
    }

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

    // Los setters reciben un double y modifican la posición correspondiente del array
    public void setX(double x) {
        this.coordenadas[0] = x;
    }

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

```
    }

    // El método de cálculo sigue funcionando sin cambios en su firma
    public double calcularDistanciaAOrigen() {
        return Math.sqrt(Math.pow(this.coordenadas[0], 2) + Math.pow(this.coordenadas[1], 
    }
}
```

---

## 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"?

**¿No es mejor declararlo público si tiene getter/setter?**

**No.** Aunque a corto plazo parezca que el resultado es el mismo, declarar un atributo como público elimina el **control** sobre el dato. Al usar getters y setters mantienes una capa de abstracción:

- Puedes cambiar el tipo de dato interno o el nombre del atributo sin afectar a quienes usan la clase.
- Puedes añadir lógica (como logs o disparadores de eventos) cada vez que alguien consulte o modifique el valor.

**Convención más habitual:**

La convención universal en POO (y especialmente en Java) es que los **atributos sean siempre** `private`.

- El estado interno de un objeto debe estar protegido.
- Si el mundo exterior necesita interactuar con esos datos, se le proporcionan métodos públicos controlados.
- Excepción: Solo se suelen ver atributos públicos en constantes ( `public static final` ) o en estructuras de datos muy simples (como un `Record` o un `DTO` básico), aunque

incluso ahí es preferible el acceso mediante métodos.

**Relación con las "invariantes de clase":**

Este es el punto más importante. Una **invariante de clase** es una condición o regla que debe cumplirse siempre para que un objeto sea considerado "válido".

- **Ejemplo de invariante**: En una clase `Fecha`, el atributo `dia` no puede ser mayor a 31.
- **El peligro de lo público**: Si el atributo `dia` es público, cualquier parte del programa podría poner `fecha.dia = 99;`, rompiendo la invariante y dejando el objeto en un estado corrupto.
- **La seguridad del Setter**: Al usar un setter privado/protegido con lógica, puedes validar la entrada:

```
public void setDia(int nuevoDia) {
    if (nuevoDia >= 1 && nuevoDia <= 31) {
        this.dia = nuevoDia;
    } else {
        // Lanzar error o ignorar: la invariante se mantiene a salvo
    }
}
```

---

# 17. Clases inmutables y métodos modificadores

**¿Qué significa que una clase sea inmutable?**

Una clase es **inmutable** cuando su estado (el valor de sus atributos) no puede ser modificado una vez que el objeto ha sido creado.

- Para lograr esto, los atributos suelen declararse como `private` y `final`.
- La clase no proporciona métodos que alteren sus datos internos.
- Un ejemplo clásico en Java es la clase `String`. Cuando "modificas" un String, en realidad se crea uno nuevo en memoria.

**¿Qué es un método modificador?**

Un **método modificador** (también llamado *mutador*) es cualquier método que cambia el estado interno de un objeto, es decir, que altera el valor de uno o más de sus atributos.

**¿Un método modificador es siempre un "setter"?**

**No.** Aunque todos los *setters* son métodos modificadores, no todos los modificadores tienen que ser *setters*.

- Un **Setter** tiene la forma estándar `setAtributo(valor)`.
- Un **Modificador** puede tener cualquier nombre y lógica. Por ejemplo, en una clase `CuentaBancaria`, el método `retirarEfectivo(double cantidad)` es un método modificador porque altera el atributo `saldo`, pero no es un simple *setter*.

**Ventajas de que una clase sea inmutable:**

1. **Seguridad en hilos (Thread-safe)**: Al no poder cambiar, varios procesos pueden acceder al objeto al mismo tiempo sin riesgo de que los datos se corrompan.
2. **Simplicidad**: El estado del objeto es predecible durante todo su ciclo de vida; no hay que preocuparse por cambios inesperados.
3. **Uso como llaves**: Son ideales para usarse en colecciones como `HashMap` o `HashSet`, ya que su valor (y por tanto su *hash*) no cambiará nunca.
4. **Prevención de efectos secundarios**: Puedes pasar el objeto a cualquier función con la total seguridad de que la función no modificará tus datos originales.

---

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

**No, no es recomendable incluirlos por defecto:**

Existe la falsa creencia de que cada atributo privado debe tener automáticamente su propio `getter` y `setter`. Sin embargo, esto puede ser una mala práctica de diseño que rompe el principio de **encapsulamiento**.

**Razones para evitar los "setters" innecesarios:**

- **Pérdida de inmutabilidad**: Un objeto con muchos setters es, por definición, mutable. Esto lo hace más difícil de depurar y menos seguro en entornos con múltiples hilos (threads).

- **Violación de la lógica de negocio**: Muchos atributos no deberían cambiar de forma independiente. Por ejemplo, en una clase `Pedido`, el `precioTotal` no debería tener un setter manual, sino calcularse automáticamente al añadir productos.
- **Exposición de la implementación interna**: Si una clase tiene setters para todos sus campos, básicamente estás exponiendo su estructura interna, lo que genera un "acoplamiento fuerte" con otras partes del programa.

**Cuándo SÍ y cuándo NO usarlos:**

- **SÍ usarlos cuando**:
  - El atributo es una propiedad que legítimamente puede cambiar durante la vida del objeto (ej. el `volumen` de un televisor).
  - Necesitas realizar **validaciones** antes de cambiar el valor (ej. comprobar que la edad no sea negativa).
- **NO usarlos cuando**:
  - El valor se establece en el constructor y no debe cambiar (atributos `final`).
  - El cambio del valor depende de un cálculo complejo o de la actualización de otros atributos (es mejor usar un método con un nombre descriptivo de la acción).
  - Quieres crear una clase **inmutable**.

**La convención moderna:**

La tendencia actual en el diseño de software es **favorecer la inmutabilidad**. Es mejor no incluir setters a menos que sea estrictamente necesario. Si necesitas cambiar un valor, a menudo es preferible crear un nuevo objeto con el valor actualizado en lugar de modificar el existente.

---

## 19. ¿La clase `String` en Java es mutable o inmutable? ¿Qué ocurre al concatenar dos cadenas?

**Naturaleza de la clase `String`:**

La clase `String` en Java es **estrictamente inmutable**. Una vez que un objeto `String` ha sido creado en la memoria (específicamente en el *String Pool*), su contenido no puede ser modificado.

### ¿Qué ocurre al concatenar dos cadenas?

Cuando realizas una operación de concatenación (por ejemplo, usando el operador `+`), **no se modifica** ninguna de las cadenas originales. En su lugar:

1. Java reserva un nuevo espacio en memoria.
2. Crea un **nuevo objeto** `String` que contiene la combinación de ambas.
3. Las cadenas originales permanecen intactas en la memoria hasta que el *Garbage Collector* las elimine (si ya no tienen referencias).

### ¿Qué debemos hacer para concatenar muchas veces?

Si vas a construir una cadena muy larga mediante un bucle o muchas operaciones sucesivas, usar el operador `+` es muy ineficiente porque genera miles de objetos temporales intermedios, lo que consume mucha memoria y tiempo de CPU.

En estos casos, lo correcto es utilizar la clase `StringBuilder`:

- `StringBuilder`: Es una clase **mutable** diseñada específicamente para modificar contenido sin crear objetos nuevos constantemente.
- **Método** `append()`: Permite añadir texto al final de la estructura existente de forma eficiente.
- `toString()`: Una vez que hayas terminado de construir la cadena, conviertes el `StringBuilder` en un `String` final.

**Ejemplo comparativo:**

```
// FORMA INCORRECTA (Ineficiente)
String resultado = "";
for (int i = 0; i < 1000; i++) {
    resultado += i; // Crea 1000 objetos String diferentes
}

// FORMA CORRECTA (Eficiente)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i); // Modifica el mismo objeto internamente
}
String resultadoFinal = 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?

### ¿Por contenido o por identidad?

En Programación Orientada a Objetos, existen dos formas de entender la igualdad:

- **Igualdad por Identidad (Referencia)**: Se comprueba si dos variables apuntan exactamente al mismo objeto en la memoria (la misma dirección). Para esto se utiliza el operador `==`.
- **Igualdad por Contenido (Estado)**: Se comprueba si dos objetos distintos tienen los mismos valores en sus atributos.

### ¿Qué es el método `equals` en Java?

El método `equals` es un método definido en la clase `Object` (la raíz de todas las clases en Java) diseñado específicamente para realizar la **comparación por contenido**.

### ¿Qué hace por defecto?

Aquí está el detalle crítico: por defecto, la implementación de `equals` en la clase `Object` hace exactamente lo mismo que el operador `==`, es decir, **compara identidades**.

- Si quieres que tu clase (por ejemplo, `Punto`) se compare por sus coordenadas `x` e `y`, **debes sobrescribir (override)** el método `equals` para definir qué significa que dos puntos sean "iguales" para ti.

### ¿Cómo se deben comparar dos cadenas en Java?

Nunca debes comparar el contenido de dos cadenas con `==`.

- El operador `==` comparará si son el mismo objeto en memoria (lo cual puede dar resultados confusos debido al *String Pool*).
- Para comparar el texto que contienen, **siempre debes usar el método `.equals()`**.

```
String s1 = new String("hola");
String s2 = new String("hola");
```

```
if (s1 == s2)      // FALSO: son objetos distintos en memoria
if (s1.equals(s2)) // VERDADERO: contienen el mismo texto
```

---

## 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?

---

Las clases *wrapper* (o envoltorio) son clases diseñadas específicamente para "envolver" o encapsular un tipo de dato primitivo (como `int`, `double` o `boolean`) dentro de un objeto real. En Java, por ejemplo, el primitivo `int` tiene su *wrapper* `Integer`, y `char` tiene `Character`.

**¿Cómo se hace y es un proceso automático?**

Se hace asignando el valor primitivo a una variable del tipo de la clase *wrapper*. En las versiones modernas de Java, este proceso es **completamente automático**.

- El paso automático de un tipo primitivo a su objeto correspondiente se llama **Autoboxing**.
- El paso automático inverso (de objeto a primitivo) se llama **Unboxing**.

```
// Ejemplo en Java (JDK 25)
public void ejemploWrappers() {
    // Autoboxing: el compilador convierte automáticamente el primitivo 5 en un objeto Int
    Integer numeroObjeto = 5;

    // Unboxing: el objeto Integer se convierte automáticamente al primitivo int para hace
    int primitivo = numeroObjeto + 10;
}
```

---

# 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?

---

En la Programación Orientada a Objetos (POO), un **tipo de dato enumerado** (comúnmente llamado *enum*) es un tipo de dato especial que permite definir un conjunto fijo de constantes predefinidas. Su propósito principal es restringir los valores que puede tomar una variable a una lista cerrada y segura, lo que hace que el código sea mucho más legible y menos propenso a errores (por ejemplo, definir los estados de un pedido: `PENDIENTE`, `ENVIADO`, `ENTREGADO`).

### ¿En Java, un tipo de dato enumerado es una clase?

**Sí, absolutamente.** A diferencia de otros lenguajes (como C) donde los enumerados son simplemente números enteros con nombre, en Java un `enum` es un tipo especial de clase. Cuando el compilador procesa un `enum`, internamente crea una clase que hereda de la clase base `java.lang.Enum`. Esto significa que las constantes enumeradas son, de hecho, objetos reales, y el `enum` puede contener atributos, métodos e incluso implementar interfaces.

### ¿Qué ventajas tienen en términos de encapsulación?

La ventaja fundamental es que **permiten encapsular estado y comportamiento dentro de las propias constantes**.

En lugar de tener la información asociada a una constante dispersa por el código (por ejemplo, usando condicionales `switch` para saber qué descripción tiene un estado), el propio `enum` puede almacenar esos datos internamente. Al usar atributos marcados como `private` y un constructor (que en los enums es privado por defecto), garantizas que nadie pueda alterar las propiedades de las constantes una vez creadas, exponiendo solo lo necesario a través de su interfaz pública (getters).

```
// Ejemplo en Java (JDK 25)
public enum CategoriaCliente {
    // Las constantes son instancias de la clase, inicializadas con datos específicos
    BRONCE(5.0, "Acceso estándar"),
    PLATA(10.0, "Acceso prioritario"),
```

```
    ORO(20.0, "Acceso total VIP");

    // Ocultación de información: estado interno fuertemente encapsulado
    private final double porcentajeDescuento;
    private final String nivelAcceso;

    // Constructor (siempre es privado en los enums, impidiendo la creación desde fuera)
    CategoriaCliente(double porcentajeDescuento, String nivelAcceso) {
        this.porcentajeDescuento = porcentajeDescuento;
        this.nivelAcceso = nivelAcceso;
    }

    // Interfaz pública (Getters)
    public double getPorcentajeDescuento() {
        return this.porcentajeDescuento;
    }

    public String getNivelAcceso() {
        return this.nivelAcceso;
    }
}
```

---

## 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)`

Aquí tienes la implementación completa del enumerado `Mes` cumpliendo con todos los requisitos de encapsulación y lógica solicitados.

Ten en cuenta que astronómicamente las estaciones cambian alrededor de los días 21 de los meses de marzo, junio, septiembre y diciembre. Por tanto, esos meses de transición tienen "algunos días" de dos estaciones distintas, lo cual se refleja en la lógica de los métodos.

```
// Implementación en Java (JDK 25)
public enum Mes {
    ENERO(31, 1),
    FEBRERO(28, 2), // Asumimos año no bisiesto por simplicidad
    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);

    // Atributos privados para mantener la encapsulación
    private final int dias;
    private final int ordinal;

    // Constructor privado del enumerado
    Mes(int dias, int ordinal) {
        this.dias = dias;
        this.ordinal = ordinal;
    }

    // Métodos consultores (Getters)
    public int getDias() {
        return this.dias;
    }

    public int getOrdinal() {
```

```
        return this.ordinal;
    }

    // --- Métodos de estaciones ---
    // La primavera (Norte) va de finales de marzo a finales de junio.
    public boolean esDePrimavera(boolean enHemisferioNorte) {
        return enHemisferioNorte
            ? (this.ordinal >= 3 && this.ordinal <= 6)  // Norte: Marzo - Junio
            : (this.ordinal >= 9 && this.ordinal <= 12); // Sur: Septiembre - Diciembre
    }

    // El verano (Norte) va de finales de junio a finales de septiembre.
    public boolean esDeVerano(boolean enHemisferioNorte) {
        return enHemisferioNorte
            ? (this.ordinal >= 6 && this.ordinal <= 9)   // Norte: Junio - Septiembre
            : (this.ordinal == 12 || this.ordinal <= 3); // Sur: Diciembre - Marzo
    }

    // El otoño (Norte) va de finales de septiembre a finales de diciembre.
    public boolean esDeOtono(boolean enHemisferioNorte) {
        return enHemisferioNorte
            ? (this.ordinal >= 9 && this.ordinal <= 12)  // Norte: Septiembre - Diciembre
            : (this.ordinal >= 3 && this.ordinal <= 6);  // Sur: Marzo - Junio
    }

    // El invierno (Norte) va de finales de diciembre a finales de marzo.
    public boolean esDeInvierno(boolean enHemisferioNorte) {
        return enHemisferioNorte
            ? (this.ordinal == 12 || this.ordinal <= 3)  // Norte: Diciembre - Marzo
            : (this.ordinal >= 6 && this.ordinal <= 9);  // Sur: Junio - Septiembre
    }
}
```

---

## 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

**Implementación de los métodos de estación:**

Para resolver esto, debemos recordar que las estaciones están invertidas entre el hemisferio norte y el sur. Utilizaremos una estructura de control (como un `switch` o `if-else`) basada en el número del mes para determinar a qué estación pertenece.

**Ejemplo de código lógico:**

```
public boolean esDePrimavera(boolean esHemisferioNorte) {
    if (esHemisferioNorte) {
        // Marzo (3), Abril (4), Mayo (5), Junio (6)
        return (mes >= 3 && mes <= 6);
    } else {
        // Septiembre (9), Octubre (10), Noviembre (11), Diciembre (12)
        return (mes >= 9 && mes <= 12);
    }
}

public boolean esDeVerano(boolean esHemisferioNorte) {
    if (esHemisferioNorte) {
        // Junio (6), Julio (7), Agosto (8), Septiembre (9)
        return (mes >= 6 && mes <= 9);
    } else {
        // Diciembre (12), Enero (1), Febrero (2), Marzo (3)
        return (mes == 12 || mes == 1 || mes == 2 || mes == 3);
    }
}

public boolean esDeOtoño(boolean esHemisferioNorte) {
    if (esHemisferioNorte) {
        // Septiembre (9), Octubre (10), Noviembre (11), Diciembre (12)
        return (mes >= 9 && mes <= 12);
    } else {
        // Marzo (3), Abril (4), Mayo (5), Junio (6)
        return (mes >= 3 && mes <= 6);
    }
}

public boolean esDeInvierno(boolean esHemisferioNorte) {
    if (esHemisferioNorte) {
        // Diciembre (12), Enero (1), Febrero (2), Marzo (3)
        return (mes == 12 || mes == 1 || mes == 2 || mes == 3);
    } else {
        // Junio (6), Julio (7), Agosto (8), Septiembre (9)
        return (mes >= 6 && mes <= 9);
```

```
    }
}
```

---