TEMA 1. Clases y objetos

1. ¿Cuáles son las cuatro características básicas de la programación orientada a objetos? Describe brevemente cada una

Las cuatro características básicas de la programación orientada a objetos son **encapsulamiento**, **abstracción**, **herencia** y **polimorfismo**. Estas propiedades definen cómo se organiza y estructura el código cuando se trabaja con clases y objetos, permitiendo crear programas más modulares, entendibles y fáciles de mantener.

El **encapsulamiento** consiste en agrupar datos (atributos) y funciones que operan sobre esos datos (métodos) dentro de una misma unidad llamada clase. Además, permite ocultar la información interna para que solo se pueda acceder a ella mediante métodos públicos controlados. Desde la perspectiva de alguien acostumbrado a C, sería similar a combinar una `struct` y las funciones que la manipulan, pero con protección frente a accesos indebidos.

La **abstracción** se centra en mostrar únicamente los detalles esenciales de un objeto y ocultar lo que no es relevante para el uso externo. En otras palabras, la clase expone una interfaz clara y sencilla, mientras que la complejidad interna queda oculta. Esto facilita trabajar con conceptos más cercanos al mundo real, sin necesidad de preocuparse por cómo están implementados internamente.

La **herencia** permite crear nuevas clases basadas en otras ya existentes, reutilizando su código y ampliándolo o modificándolo cuando sea necesario. Es como crear una variación de algo ya definido, evitando escribir todo desde cero. Por último, el **polimorfismo** ofrece la capacidad de usar una misma operación de distintas formas, según el tipo concreto del objeto que la implemente. Esto hace posible que distintas clases respondan de manera diferente a un mismo método, siempre manteniendo una interfaz común que facilita el diseño del programa.

Si quieres, puedo desarrollarlas con ejemplos sencillos en Java adaptados a tu nivel actual.

2. Cita cuatro lenguajes populares que permitan la programación orientada a objetos

Cuatro lenguajes de programación populares que permiten trabajar con orientación a objetos son **Java**, **C++**, **Python** y **C#**. Todos ellos incorporan mecanismos como clases, objetos, herencia y polimorfismo, aunque cada uno lo hace con un estilo y unas reglas propias. Esta variedad permite escoger el lenguaje que mejor se adapte a las necesidades del proyecto o al nivel de abstracción requerido.

Java es uno de los lenguajes orientados a objetos más conocidos y se caracteriza por aplicar este paradigma de forma muy estricta, ya que prácticamente todo gira alrededor de clases y objetos. **C++**, aunque proviene de C, añade orientación a objetos sin abandonar su naturaleza de lenguaje de bajo nivel, lo que permite combinar programación estructurada con técnicas de diseño más avanzadas. Esta flexibilidad lo ha mantenido vigente en ámbitos como sistemas, videojuegos o desarrollo de motores gráficos.

Por su parte, **Python** ofrece una orientación a objetos muy sencilla y accesible, lo que facilita aprender el paradigma sin necesidad de enfrentarse a una sintaxis compleja. Aunque permite otros estilos de

programación, sus clases y objetos se utilizan habitualmente para modelar problemas de forma clara. Finalmente, **C#** es un lenguaje moderno diseñado por Microsoft que incorpora una orientación a objetos robusta y coherente, ampliamente utilizado en aplicaciones de escritorio, servicios web y desarrollo con Unity.

Si lo deseas, puedo explicarte cómo se vería una clase muy básica en cada uno de estos lenguajes para comparar estilos y ayudarte a relacionarlo con tus conocimientos previos de C/C++.

3. Los paradigmas anteriores a la POO, ¿Qué es la **programación estructurada**? y, todavía mejor, ¿Qué es la **programación modular**?

La **programación estructurada** es un paradigma que organiza el código siguiendo un conjunto reducido de estructuras de control: secuencia, selección (*if/else*, *switch*) e iteración (*for*, *while*). Su objetivo es evitar el uso de saltos incontrolados como `goto`, que complican el flujo lógico del programa. Este enfoque fomenta que el código se lea de arriba abajo de forma clara, lineal y predecible, lo que facilita el razonamiento sobre su comportamiento. En lenguajes como C, esta manera de programar es la base del diseño y sigue siendo fundamental incluso en contextos modernos.

La programación estructurada también promueve dividir el código en pequeñas unidades funcionales, como funciones bien definidas, cada una con una responsabilidad concreta. Esto mejora la claridad y facilita las pruebas y la depuración, porque cada parte del programa se puede analizar de manera aislada. Para alguien que viene de C, este paradigma resulta completamente familiar, ya que constituye la base del estilo tradicional que se utiliza para resolver problemas sin recurrir a conceptos de objetos o clases.

Por otra parte, la **programación modular** lleva la idea de la organización un paso más allá, proponiendo dividir un programa en módulos independientes que agrupan funciones y datos relacionados. Cada módulo actúa como una pieza autocontenida del sistema, con una interfaz bien definida que especifica qué ofrece al resto del programa. Esta organización permite trabajar en partes del software sin afectar a las demás, lo que facilita la escalabilidad y el mantenimiento a medida que el proyecto crece.

Mientras que la programación estructurada se centra sobre todo en controlar la lógica interna del código, la programación modular se enfoca en su arquitectura global, estableciendo cómo se relacionan entre sí los distintos componentes. En C, esto se aproxima al uso de ficheros `.h` y `.c` para separar declaraciones e implementaciones, permitiendo que cada módulo o biblioteca mantenga su propio espacio de datos y funciones. Este enfoque sirve como transición natural hacia la programación orientada a objetos, donde las clases actúan como módulos aún más encapsulados y especializados.

4. ¿Qué tres elementos definen a un objeto en programación orientada a objetos?

En programación orientada a objetos, un objeto se define mediante **tres elementos fundamentales: atributos, métodos e identidad**. Estos componentes permiten representar entidades del mundo real de forma estructurada, combinando datos y comportamientos dentro de una unidad coherente. A diferencia de la programación estructurada, aquí se trabaja con entidades que “saben” cosas y pueden “hacer” cosas en función de su estado.

Los **atributos** representan el estado interno del objeto. Son variables que almacenan información relevante, como por ejemplo el nombre, la edad o el precio asociado a una instancia concreta. Desde la perspectiva de alguien habituado a C, los atributos pueden

recordarse a los campos de una `struct`, pero integrados dentro de un sistema que controla quién puede acceder a ellos y quién no.

Los **métodos** definen el comportamiento del objeto. Son funciones asociadas directamente a la clase, utilizadas para operar sobre los atributos o para realizar acciones específicas. Esta idea se aproxima a las funciones que manipulan una `struct` en C, pero aquí se encuentran ligadas directamente a los datos que manejan, lo que facilita que cada objeto controle su propio funcionamiento.

Por último, la **identidad** distingue a un objeto de cualquier otro, incluso aunque compartan exactamente los mismos valores en sus atributos. Cada instancia creada ocupa un lugar propio en memoria y se considera única. Esta noción no aparece explícitamente en la programación estructurada, pero resulta esencial en la POO porque permite gestionar múltiples objetos similares sin perder su individualidad dentro del programa.

5. ¿Qué es una clase? ¿Es lo mismo que un objeto? ¿Qué es una instancia? ¿Todos los lenguajes orientados a objetos manejan el concepto de clase?

Una **clase** es un modelo o plantilla que descrEn programación orientada a objetos, un objeto se define mediante **tres elementos fundamentales: atributos, métodos e identidad**. Estos componentes permiten representar entidades del mundo real de forma estructurada, combinando datos y comportamientos dentro de una unidad coherente. A diferencia de la programación estructurada, aquí se trabaja con entidades que “saben” cosas y pueden “hacer” cosas en función de su estado.

Los **atributos** representan el estado interno del objeto. Son variables que almacenan información relevante, como por ejemplo el nombre, la edad o el precio asociado a una instancia concreta. Desde la perspectiva de alguien habituado a C, los atributos pueden recordarse a los campos de una `struct`, pero integrados dentro de un sistema que controla quién puede acceder a ellos y quién no.

Los **métodos** definen el comportamiento del objeto. Son funciones asociadas directamente a la clase, utilizadas para operar sobre los atributos o para realizar acciones específicas. Esta idea se aproxima a las funciones que manipulan una `struct` en C, pero aquí se encuentran ligadas directamente a los datos que manejan, lo que facilita que cada objeto controle su propio funcionamiento.

Por último, la **identidad** distingue a un objeto de cualquier otro, incluso aunque compartan exactamente los mismos valores en sus atributos. Cada instancia creada ocupa un lugar propio en memoria y se considera única. Esta noción no aparece explícitamente en la programación estructurada, pero resulta esencial en la POO porque permite gestionar múltiples objetos similares sin perder su individualidad dentro del programa. ibe cómo serán los objetos que se creen a partir de ella. Define qué **atributos** tendrán esos objetos y qué **métodos** podrán ejecutar. En otras palabras, especifica la estructura y el comportamiento comunes que compartirán todas sus instancias. Desde la perspectiva de alguien que viene de C, una clase podría verse como una evolución de una `struct`, pero con funciones integradas y con mecanismos de protección y organización mucho más avanzados.

Un **objeto** no es lo mismo que una clase. La clase actúa como un plano teórico, mientras que el objeto es una entidad concreta creada siguiendo ese plano. Es similar a la diferencia entre el diseño de un coche y un coche

real que se puede conducir. Cada objeto posee sus propios valores para los atributos definidos en la clase y puede ejecutar los métodos que dicha clase ofrece.

Una **instancia** es simplemente otro nombre para un objeto creado a partir de una clase. Cuando se dice que "se instancia una clase", significa que se está creando un objeto real basado en su definición. Cada instancia tiene su identidad propia y mantiene su propio estado, de forma independiente del resto de objetos generados a partir de la misma clase.

No todos los lenguajes orientados a objetos manejan necesariamente el concepto de clase. Lenguajes como **Java**, **C++** o **C#** sí se basan en clases para organizar los objetos, pero otros, como **JavaScript**, utilizan un modelo de objetos basado en prototipos, donde no existen clases en el sentido tradicional y los objetos se crean a partir de otros objetos. Aun así, la mayoría de los lenguajes modernos con orientación a objetos incorporan el concepto de clase porque facilita la organización y el diseño del software.

6. ¿Dónde se almacenan en memoria los objetos? ¿Es igual en todos los lenguajes? ¿Qué es la **recolección de basura**?

El lugar donde se almacenan los **objetos en memoria** depende del lenguaje y del modo en que estos se crean. En lenguajes como Java, los objetos siempre se almacenan en una zona llamada *heap*, que es un espacio de memoria destinado a datos dinámicos. Solo las referencias a esos objetos se guardan en la *stack*. En C++, en cambio, un objeto puede almacenarse en la *stack* si se declara como variable local, o en el *heap* si se crea con `new`. Esto hace que la gestión de memoria sea más flexible, pero también más compleja, porque requiere liberar manualmente los recursos.

No todos los lenguajes gestionan la memoria de la misma manera. Java y Python, por ejemplo, utilizan *heap* casi exclusivamente para los objetos y delegan la limpieza de memoria en un sistema automático. C++ permite usar tanto *stack* como *heap*, exigiendo al programador controlar la vida útil de los objetos creados dinámicamente. Esta diferencia afecta al rendimiento, a la facilidad de depuración y al riesgo de errores como fugas de memoria o uso de punteros inválidos.

La **recolección de basura** (*garbage collection*) es un mecanismo automático que libera memoria eliminando objetos que ya no están siendo utilizados por el programa. En lenguajes como Java o Python, el programador no necesita preocuparse por liberar memoria manualmente, porque el sistema detecta cuándo un objeto ha dejado de ser accesible y lo elimina. Esto reduce significativamente errores comunes, como olvidarse de liberar memoria o intentar acceder a una zona ya liberada.

Sin embargo, la recolección de basura no es universal. Lenguajes como C y C++ no disponen de este sistema integrado, por lo que la gestión de memoria depende totalmente del programador. Esto ofrece más control y puede mejorar el rendimiento, pero también aumenta el riesgo de errores graves. Por tanto, el concepto de dónde se almacenan los objetos y cómo se gestionan sus ciclos de vida varía considerablemente entre distintos lenguajes orientados a objetos.

7. ¿Qué es un método? ¿Qué es la **sobrecarga de métodos**?

Un **método** es una función asociada a una clase que define un comportamiento que los objetos de esa clase pueden ejecutar. En términos prácticos, un método opera sobre el estado del objeto (sus atributos) y/o produce un resultado relacionado con su responsabilidad. A diferencia de una función libre en C, el método está "ligado" a un tipo (la clase) y dispone de

un parámetro implícito —el objeto receptor—, aunque en lenguajes como Java ese parámetro no se escribe de forma explícita. Existen métodos de instancia (actúan sobre un objeto concreto) y métodos estáticos (pertenecen a la clase y no a una instancia).

La **sobrecarga de métodos** consiste en definir varios métodos con el **mismo nombre** dentro de la **misma clase**, diferenciándose por su **lista de parámetros** (número, tipos y/o orden). El compilador selecciona cuál invocar en **tiempo de compilación** según la firma más compatible con los argumentos usados. La sobrecarga no debe confundirse con la **sobrescritura (override)**, que ocurre cuando una subclase redefine un método heredado con la misma firma para cambiar su comportamiento en **tiempo de ejecución** (polimorfismo). En lenguajes como Java, el tipo de retorno no participa por sí solo en la sobrecarga; debe cambiar la lista de parámetros.

Ejemplo sencillo en Java (sobrecarga):

```
class Calculadora {
    // Método de instancia
    int sumar(int a, int b) { return a + b; }

    // Sobrecarga por número de parámetros
    int sumar(int a, int b, int c) { return a + b + c; }

    // Sobrecarga por tipos
    double sumar(double a, double b) { return a + b; }

    // Método estático (también puede sobrecargarse)
    static int abs(int x) { return x < 0 ? -x : x; }
    static double abs(double x) { return x < 0 ? -x : x; }
}
```

Contraste rápido con sobrescritura (no es sobrecarga):

```
class Figura {
    double area() { return 0; }
}
class Circulo extends Figura {
    // Sobrescritura: misma firma, comportamiento diferente en subclase
    double area() { return Math.PI * 1 * 1; }
}
```

8. Ejemplo mínimo de clase en Java, que se llame Punto, con dos atributos, x e y, con un método que se llame `calculaDistanciaAOrigen`, que calcule la distancia a la posición 0,0. Por sencillez, los atributos deben tener visibilidad por defecto. Crea además un ejemplo de uso con una instancia y uso del método

A continuación se muestra un ejemplo mínimo de clase en Java llamada `Punto`, con dos atributos `x` e `y` con **visibilidad por defecto** (sin modificador) y un método

`calculaDistanciaAOrigen()` que devuelve la distancia al origen `(0,0)`. El método usa la fórmula de la distancia euclidiana, que en este caso se reduce a `sqrt(x^2 + y^2)`. Se omiten constructores explícitos para mantener el ejemplo lo más simple posible; Java proporcionará el constructor por defecto sin argumentos.

También se incluye un ejemplo de uso en un `main` donde se crea una **instancia** de `Punto`, se asignan valores a sus atributos y se llama al método. Este enfoque permite ver de forma directa cómo acceder a atributos con visibilidad por defecto dentro del mismo paquete y cómo invocar el comportamiento definido en la clase.

```
// Archivo: Punto.java
class Punto {
    // Atributos con visibilidad por defecto (package-private)
    double x;
    double y;

    // Método que calcula la distancia al origen (0,0)
    double calculaDistanciaAOrigen() {
        return Math.sqrt(x * x + y * y);
    }
}
```

```
// Archivo: EjemploUso.java
class EjemploUso {
    public static void main(String[] args) {
        Punto p = new Punto();  // instancia usando el constructor por defecto
        p.x = 3.0;
        p.y = 4.0;

        double d = p.calculaDistanciaAOrigen();
        System.out.println("Distancia al origen: " + d); // Imprime 5.0
    }
}
```

9. ¿Cuál es el punto de entrada en un programa en Java? ¿Qué es `static` y para qué vale? ¿Sólo se emplea para ese método `main`? ¿Para qué se combina con `final`?

El **punto de entrada** de un programa en Java es el método `public static void main(String[] args)` contenido en alguna clase cargada por la JVM al iniciar la aplicación. Debe ser `public` para que la JVM pueda invocarlo desde fuera de la clase, `static` para poder llamarlo **sin crear un objeto previo**, y devolver `void` porque no retorna resultado al sistema (la finalización y el código de salida se gestionan con `System.exit` si es necesario). También se admite la variante con varargs `main(String... args)`. El cargador de clases localizará ese `main` en la clase indicada al ejecutar `java NombreDeLaClase` (o la definida en el `Main-Class` del *manifest* si se lanza un JAR).

La palabra clave `static` indica que un miembro **pertenece a la clase**, no a las instancias. Un **método estático** puede llamarse sin crear objetos y no puede acceder directamente a atributos o métodos de instancia (porque no tiene un receptor implícito). Un **campo estático** es compartido por todas las instancias; se inicializa una sola vez por clase. Además, existen **bloques estáticos** para inicialización avanzada y **clases anidadas estáticas** (análogas a clases "normales" pero agrupadas por organización). Por tanto, `static` no se usa solo en `main`: se emplea mucho en utilidades (por ejemplo, `Math.sqrt`), *factory methods*, contadores globales, *singletons* controlados y constantes.

La combinación `static` + `final` se usa típicamente para **constantes**: valores inmutables asociados a la clase, generalmente con modificador `public` y nombre en mayúsculas con guiones bajos. Aquí, `static` evita tener que crear objetos para acceder al valor y `final` impide que se reasigne. Más allá de campos, `final` también bloquea la sobrescritura de métodos y la herencia de clases, pero lo más común junto con `static` es declarar constantes simbólicas.

Ejemplos rápidos:

```
// Punto de entrada
public class App {
    public static void main(String[] args) {
        System.out.println("Hola, Java");
    }
}
```

```
// static en utilidades y constantes
public class Matematicas {
    public static final double PI_APROX = 3.141592653589793;

    public static int doble(int x) {        // método de clase
        return 2 * x;
    }

    static {                                // bloque estático (se ejecuta una
vez)
        // Inicialización compleja si hiciera falta
    }
}
```

```
// Campo estático compartido
class Contador {
    private static int totalCreados = 0;

    public Contador() { totalCreados++; }

    public static int getTotalCreados() { return totalCreados; }
}
```

10. Intenta ejecutar un poco de Java de forma básica, con los comandos `javac` y `java`. ¿Cómo podemos compilar el programa y ejecutarlo desde linea de comandos? ¿Java es compilado? ¿Qué es la **máquina virtual**? ¿Qué es el *byte-code* y los ficheros `.class`?

Para **compilar y ejecutar** desde línea de comandos se suele trabajar con dos herramientas: `javac` (compilador) y `java` (intérprete/ejecutor de bytecode en la JVM). En un directorio sin paquetes, el flujo básico consiste en guardar el código fuente en un archivo cuyo nombre coincida con la clase pública y con extensión `.java`, compilar para generar uno o varios `.class`, y ejecutar indicando el **nombre de la clase** (no el fichero). Si hubiese paquetes (`package ...;`), la ejecución debe realizarse desde el directorio raíz del paquete (o ajustando el `-cp` / `CLASSPATH`).

Ejemplo mínimo:

```java
// Archivo: App.java
public class App {
    public static void main(String[] args) {
        System.out.println("Hola, Java");
    }
}
```

```
# Compilación
javac App.java

# Genera: App.class (bytecode)
# Ejecución (usar el nombre de la clase, sin .class ni ruta)
java App
```

```
# Con paquetes (p. ej., src/com/ejemplo/App.java con "package com.ejemplo;")
javac -d out src/com/ejemplo/App.java
java -cp out com.ejemplo.App
```

Java es **compilado a bytecode** y luego **interpretado/ejecutado** por la Máquina Virtual de Java (JVM), con optimizaciones en tiempo de ejecución (*JIT compilation*). En otras palabras, no se compila directamente a código máquina específico del sistema; primero se traduce a un formato intermedio portátil. Este enfoque permite escribir el programa una vez y ejecutarlo en múltiples plataformas donde exista una JVM compatible, manteniendo portabilidad sin renunciar a optimizaciones dinámicas que mejoran el rendimiento en ejecución.

La **Máquina Virtual de Java (JVM)** es el entorno de ejecución que carga, verifica y ejecuta el bytecode. Proporciona servicios como gestión de memoria (incluida la recolección de basura), seguridad, *class loading* y compilación JIT. El **bytecode** es el conjunto de instrucciones intermedias que genera `javac` y que se almacena en los ficheros `.class`; dichos ficheros contienen el código portable que la JVM entiende y

transforma a código nativo optimizado durante la ejecución. Por ello, la secuencia típica es: código fuente `.java` → compilación con `javac` → bytecode `.class` → ejecución con `java` sobre la JVM.

11. En el código anterior de la clase `Punto` ¿Qué es `new`? ¿Qué es un **constructor**? Pon un ejemplo de constructor en una clase `Empleado` que tenga DNI, nombre y apellidos

En Java, `new` es el operador que **reserva memoria en el *heap* y crea un objeto**, devolviendo una **referencia** a ese objeto. Además, invoca el **constructor** correspondiente para inicializar el estado inicial. En el ejemplo de `Punto`, `new Punto()` crea una instancia con los atributos `x` e `y` en sus valores por defecto (cero para `double`) al no haberse definido un constructor propio.

Un **constructor** es un **método especial** cuya misión es **inicializar** una nueva instancia. Se caracteriza porque **tiene el mismo nombre que la clase**, **no declara tipo de retorno** (ni siquiera `void`), y se ejecuta automáticamente al llamar a `new`. Si no se define ninguno, el compilador proporciona un **constructor por defecto** sin parámetros. Es habitual **sobrecargar** constructores para ofrecer distintas formas de crear el objeto (por ejemplo, con o sin datos iniciales).

Ejemplo de clase `Empleado` con constructores (DNI, nombre, apellidos) y uso:

```
public class Empleado {
    // Atributos
    private final String dni;   // Se asume obligatorio e inmutable
    private String nombre;
    private String apellidos;

    // Constructor principal (obligando a proporcionar todos los datos)
    public Empleado(String dni, String nombre, String apellidos) {
        if (dni == null || dni.isBlank()) {
            throw new IllegalArgumentException("El DNI no puede ser nulo ni
vacío");
        }
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    // Constructor alternativo (mínimo imprescindible)
    public Empleado(String dni) {
        this(dni, "", "");
    }

    // Getters y setters necesarios
    public String getDni() { return dni; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public String getApellidos() { return apellidos; }
    public void setApellidos(String apellidos) { this.apellidos = apellidos; }
}
```

```java
// Ejemplo de uso
public class Demo {
    public static void main(String[] args) {
        Empleado e1 = new Empleado("12345678A", "Ana", "Pérez Gómez");
        Empleado e2 = new Empleado("87654321B"); // nombre/apellidos por defecto

        System.out.println(e1.getDni() + " - " + e1.getNombre());
        e2.setNombre("Luis");
        e2.setApellidos("López");
        System.out.println(e2.getDni() + " - " + e2.getNombre() + " " +
e2.getApellidos());
    }
}
```

12. ¿Qué es la referencia `this`? ¿Se llama igual en todos los lenguajes? Pon un ejemplo del uso de `this` en la clase `Punto`

La referencia **`this`** es un identificador especial disponible dentro de los métodos y constructores de una clase que **apunta a la instancia actual**: el objeto que está ejecutando ese código. Se utiliza para **desambiguar** entre atributos de instancia y parámetros locales con el mismo nombre, para **encadenar constructores** (con `this(...)`) y para **pasar la referencia del propio objeto** a otros métodos. En términos de punteros, puede pensarse como "la dirección del objeto actual", pero en Java se presenta como una referencia segura y sin aritmética de punteros.

No todos los lenguajes lo llaman igual ni se comporta exactamente del mismo modo. En **Java** y **C#** se llama `this`; en **C++** es también `this`, pero es un **puntero** (`T*`) que se desreferencia implícitamente; en **JavaScript**, `this` existe pero **su valor depende del contexto de llamada**, no de la clase; y en **Python** no hay palabra clave equivalente: por convención se usa el primer parámetro `self` en métodos de instancia, que se pasa de manera explícita.

Ejemplo de uso en `Punto` (desambiguación en el constructor, encadenamiento y método de instancia):

```java
class Punto {
    double x;
    double y;

    // Constructor principal
    Punto(double x, double y) {
        this.x = x;    // 'this.x' es el atributo; 'x' es el parámetro
        this.y = y;
    }

    // Constructor por defecto que encadena al principal
    Punto() {
        this(0.0, 0.0); // llamada a otro constructor de la misma clase
    }
```

```
    // Método de instancia que usa el estado del objeto actual
    double calculaDistanciaAOrigen() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }

    // Ejemplo de pasar 'this' como referencia
    boolean esIgualA(Punto otro) {
        return this.x == otro.x && this.y == otro.y;
    }
}
```

13. Añade ahora otro nuevo método que se llame `distanciaA`, que reciba un `Punto` como parámetro y calcule la distancia entre `this` y el punto proporcionado

Para implementar `distanciaA` en Java, se define un método de instancia que reciba otro `Punto` y calcule la distancia euclídea entre ambos: $$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$. En Java, `this` referencia al objeto actual (similar al uso implícito de `this` en C++), por lo que `this.x` y `this.y` representan las coordenadas del punto desde el que se invoca el método. Se recomienda validar que el parámetro no sea `null` para evitar una excepción de referencia nula.

A nivel de implementación, puede usarse `Math.hypot(dx, dy)`, que resulta más estable numéricamente que calcular manualmente `Math.sqrt(dx*dx + dy*dy)` en casos extremos de valores muy grandes o muy pequeños. El método devuelve un `double`. Si la clase `Punto` es inmutable (campos `final`), la operación no modifica estado; si no lo es, igualmente `distanciaA` solo lee los campos.

A continuación se muestra un ejemplo completo y minimalista de la clase con el método solicitado:

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

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

    public double getX() { return x; }
    public double getY() { return y; }

    // Método solicitado: calcula la distancia entre this y el punto
proporcionado.
    public double distanciaA(Punto otro) {
        if (otro == null) {
            throw new IllegalArgumentException("El punto proporcionado no puede
ser null");
        }
        double dx = otro.x - this.x;
        double dy = otro.y - this.y;
```

```
        return Math.hypot(dx, dy); // equivalente a Math.sqrt(dx*dx + dy*dy)
    }
}
```

En uso, la llamada sería `p1.distanciaA(p2)`, donde `p1` y `p2` son instancias de `Punto`. Si se prefiriera evitar `Math.hypot`, puede reemplazarse la línea de retorno por `return Math.sqrt(dx * dx + dy * dy);` con el mismo resultado práctico en rangos normales de valores.

14. El paso del `Punto` como parámetro a un método, es **por copia** o **por referencia**, es decir, si se cambia el valor de algún atributo del punto pasado como parámetro, dichos cambios afectan al objeto fuera del método? ¿Qué ocurre si en vez de un `Punto`, se recibiese un entero (`int`) y dicho entero se modificase dentro de la función?

En Java **todo** se pasa **por valor**. En el caso de objetos (como `Punto`), lo que se pasa **por valor** es la **referencia** al objeto. Esto significa que dentro del método se recibe **una copia de la referencia**, pero ambas referencias apuntan al **mismo objeto** en memoria. Por ello, si dentro del método se **modifica el estado** del objeto (por ejemplo, cambiando sus atributos mediante setters), esos cambios **sí se observan fuera** del método. En cambio, si dentro del método se **reasigna** el parámetro para que apunte a otro objeto, esa reasignación **no afecta** a la referencia que tiene el llamador.

Ejemplo ilustrativo con un objeto mutable:

```
void moverX(Punto p) {       // p es una copia de la referencia
    p.setX(p.getX() + 1);    // cambia el estado del objeto -> se ve fuera
    p = new Punto(0, 0);     // solo cambia la referencia local -> NO se ve fuera
}
```

Con los **tipos primitivos** (como `int`), también se pasa por **valor**, pero aquí lo que se copia es el **valor primitivo** en sí. Por tanto, cualquier modificación del parámetro dentro del método afecta **solo a la copia local** y **no** al valor original del llamador. Esto contrasta con C++ cuando se pasa "por referencia" (`int&`), donde sí se modificaría el original; en Java, no existe paso por referencia para parámetros.

Ejemplo con primitivo:

```
void incrementa(int n) { // n es una copia del valor
    n++;                 // solo cambia la copia local
}
```

15. ¿Qué es el método `toString()` en Java? ¿Existe en otros lenguajes? Pon un ejemplo de `toString()` en la clase `Punto` en Java

`toString()` es un método definido en `java.lang.Object` que devuelve una representación textual del objeto. Al sobrescribirlo en una clase, se controla cómo se mostrará el objeto al

imprimirlo, al concatenarlo con cadenas o al depurarlo. Es una práctica habitual para facilitar el logging y la lectura del estado interno de las instancias; si no se sobrescribe, la implementación por defecto muestra el nombre de la clase y un identificador basado en el hash del objeto, lo cual suele ser poco informativo.

Este concepto existe en otros lenguajes con nombres o mecanismos análogos. En C# se usa `ToString()` en `System.Object`. En Python, la función equivalente es `__str__` (y `__repr__` para representaciones más técnicas). En C++ no existe un `toString()` universal, pero se suele lograr algo similar sobrecargando el operador de inserción en flujo `operator<<` para `std::ostream`, o usando funciones como `std::to_string` para tipos primitivos. En todos los casos, la idea es proporcionar una representación legible del estado del objeto.

Ejemplo en Java para una clase `Punto` sobrescribiendo `toString()`:

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

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

    public double getX() { return x; }
    public double getY() { return y; }

    @Override
    public String toString() {
        // Opción 1: formato explícito
        return "Punto(x=" + x + ", y=" + y + ")";
        // O bien, con formato:
        // return String.format("Punto(x=%.3f, y=%.3f)", x, y);
    }
}
```

Con esta implementación, al ejecutar `System.out.println(new Punto(3, 4));` se imprimirá `Punto(x=3.0, y=4.0)`. Esta representación facilita la lectura en logs y durante la depuración sin necesidad de inspeccionar manualmente cada atributo.

16. Reflexiona: ¿una clase es como un `struct` en C? ¿Qué le falta al `struct` para ser como una clase y las variables de ese tipo ser instancias?

Desde un punto de vista conceptual, una clase en Java se parece a un `struct` de C en el sentido de que ambos permiten agrupar datos bajo un mismo tipo. En los dos casos, se pueden crear variables cuyo tipo es dicho agrupamiento, y esas variables contienen o referencian valores asociados a cada campo definido. Sin embargo, la similitud termina prácticamente ahí, porque un `struct` en C es únicamente una **agrupación de datos**, mientras que una clase en Java combina datos **y comportamiento**.

Para que un `struct` fuese equivalente a una clase, necesitaría principalmente la capacidad de **definir métodos asociados al tipo**, algo que en C no existe de forma directa. En una clase, los métodos pueden operar sobre los atributos del propio objeto, accediendo a ellos mediante `this`, y permitiendo encapsulación, validación y lógica interna. Además, faltaría el concepto de **encapsulamiento**, es decir, la posibilidad de declarar atributos públicos, privados o protegidos, mientras que en C todos los campos de un `struct` son públicos por definición.

Otra diferencia esencial es la capacidad de crear **instancias reales** con identidad propia, lo que implica características adicionales como constructores, destructores gestionados automáticamente por el lenguaje y la posibilidad de controlar cómo se inicializa el objeto. En C, un `struct` no tiene constructores ni inicialización automática más allá de lo que permita la sintaxis. Las clases también incorporan soporte para características más avanzadas como herencia, polimorfismo y métodos virtuales, elementos ausentes en el modelo de tipos del lenguaje C.

En conjunto, puede afirmarse que un `struct` proporciona solo la parte "dato" de una clase, pero carece de todo el "comportamiento" asociado, además de los mecanismos de encapsulación y la orientación a objetos que permiten que cada instancia sea más que un simple grupo de variables. Por eso, aunque estructuralmente se parezcan, su propósito y posibilidades dentro de sus respectivos lenguajes son significativamente distintos.

17. Quitemos un poco de magia a todo esto: ¿Como se podría "emular", con `struct` en C, la clase `Punto`, con su función para calcular la distancia al origen? ¿Qué ha pasado con `this`?

Se puede "emular" una clase `Punto` en C usando un `struct` para los datos y funciones libres que operan sobre él. No habrá métodos dentro del `struct`, pero puede definirse una convención: las funciones que actúan "como métodos" reciben un puntero al `struct` y operan sobre sus campos. En este esquema, `this` se vuelve **un parámetro explícito**, típicamente llamado `p` o `self`. Además, puede añadirse una función de "inicialización" que haga el papel de constructor (no hay constructores en C).

```c
#include <math.h>
#include <stdio.h>

typedef struct {
    double x;
    double y;
} Punto;

// "Constructor" (inicializador)
void Punto_init(Punto* p, double x, double y) {
    p->x = x;
    p->y = y;
}

// "Método" distancia al origen: sqrt(x^2 + y^2)
double Punto_distancia_al_origen(const Punto* p) {
    double dx = p->x;
    double dy = p->y;
    // hipot equivalente a sqrt(dx*dx + dy*dy), cuando esté disponible
```

```
    return hypot(dx, dy);
}

int main(void) {
    Punto a;
    Punto_init(&a, 3.0, 4.0);
    printf("distancia al origen = %.2f\n", Punto_distancia_al_origen(&a)); // 5.00
    return 0;
}
```

En este patrón, `this` **no existe como palabra clave**; lo que antes era `this` en Java se pasa explícitamente como `Punto* p` (o `const Punto* p` si no se va a modificar). La llamada también refleja esta diferencia: en Java se usaría `a.distanciaAlOrigen()`, mientras que en C se invoca `Punto_distancia_al_origen(&a)`. Si se desea acercarse aún más a la "sensación" de métodos, puede definirse una "tabla de métodos" mediante punteros a función, aunque esto es opcional y añade complejidad.

```
typedef struct Punto Punto;

typedef struct {
    double (*distancia_al_origen)(const Punto*);
} PuntoVTable;

struct Punto {
    double x;
    double y;
    const PuntoVTable* vptr; // "tabla" de métodos
};

double Punto_dist_origen_impl(const Punto* p) { return hypot(p->x, p->y); }

const PuntoVTable PUNTO_VTBL = {
    .distancia_al_origen = Punto_dist_origen_impl
};

void Punto_init(Punto* p, double x, double y) {
    p->x = x; p->y = y; p->vptr = &PUNTO_VTBL;
}

// Uso: a.vptr->distancia_al_origen(&a);
```

Este último enfoque muestra cómo `this` se modela siempre como **puntero explícito** pasado a las funciones (ya sean funciones libres o a través de una "vtable"). Aun así, siguen faltando rasgos propios de clases: encapsulación real (los campos son públicos), constructores/destructores automáticos, herencia y comprobaciones del compilador orientadas a objetos. Pero para el caso práctico de `Punto` y su distancia al origen, el patrón con `struct` + funciones libres es suficiente y claro.