TEMA 1. Clases y objetos

1. Las cuatro características básicas de la Programación Orientada a Objetos (POO)

1. **Encapsulamiento**
    - Consiste en **ocultar los detalles internos** de un objeto y exponer solo lo necesario.
    - Protege los datos y evita modificaciones indebidas, normalmente mediante **atributos privados** y **métodos públicos**.
2. **Abstracción**
    - Permite **simplificar la complejidad** mostrando solo las características esenciales de un objeto.
    - Se centra en *qué hace* un objeto, no en *cómo lo hace*.
3. **Herencia**
    - Permite que una clase (subclase) **herede atributos y métodos** de otra clase (superclase).
    - Facilita la reutilización de código y la creación de jerarquías lógicas.
4. **Polimorfismo**
    - Permite que **un mismo método** tenga **distintos comportamientos** según el objeto que lo utilice.
    - Se aplica mediante *sobrecarga* (mismo nombre, distintos parámetros) y *sobrescritura* (la subclase redefine el método de la superclase).

---

2. Lenguajes populares que permitan la programación orientada a objetos

Cuatro lenguajes populares que permiten la programación orientada a objetos:

1. **C++**
   Ampliamente usado en sistemas, videojuegos y aplicaciones de alto rendimiento. Soporta POO de forma completa.
2. **Java**
   Uno de los lenguajes más extendidos en empresas. Su diseño está fuertemente basado en la POO.
3. **Python**
   Aunque es multiparadigma, ofrece un modelo de objetos muy flexible y sencillo de usar.
4. **C#**
   Desarrollado por Microsoft, usado en aplicaciones de escritorio, web, móviles y videojuegos (Unity).

Totalmente orientado a objetos.

---

3. Paradigmas anteriores a la POO

¿Qué es la **programación estructurada**?

La **programación estructurada** es un paradigma que surgió para mejorar el desorden y la complejidad del código "espagueti" típico de los primeros programas llenos de saltos `goto`.
Su idea principal es **organizar el programa en estructuras de control bien definidas**, sin saltos arbitrarios.

Se basa en **tres tipos básicos de estructuras**:

1. **Secuencia**: instrucciones ejecutadas una detrás de otra.
2. **Selección**: decidir entre varias alternativas (por ejemplo, `if`, `switch`).
3. **Iteración**: repetir acciones (`for`, `while`).

Con estas tres estructuras se puede escribir cualquier programa sin necesidad de usar `goto`, lo que hace el código:

- Más **legible**
- Más **fácil de depurar**
- Más **mantenible**

¿Qué es la **programación modular**?

La **programación modular** es una evolución natural de la programación estructurada.
Su objetivo es **dividir un programa grande en partes más pequeñas llamadas módulos**, cada una con una función muy concreta.

Un **módulo** es un bloque de código que:

- Realiza una tarea específica
- Es independiente del resto lo máximo posible
- Tiene una **interfaz bien definida** (p. ej., funciones públicas)
- Oculta sus detalles internos (idea que luego heredará la POO como *encapsulamiento*)

Ventajas clave:

- Facilita el **mantenimiento** de grandes programas
- Permite **reutilizar** módulos en otros proyectos
- Divide el trabajo entre varios programadores
- Reduce los errores al trabajar con piezas más pequeñas y controladas

---

Relación entre ambas

- La **programación estructurada** organiza la lógica interna del código.

- La **programación modular** organiza el **proyecto completo** en bloques grandes y bien definidos.

Juntas crearon la base conceptual de lo que luego evolucionó en la **Programación Orientada a Objetos**, donde los módulos ya no son simples funciones o archivos, sino **objetos**.

---

4. Elementos que definen a un objeto en programación orientada a objetos

En programación orientada a objetos, **un objeto** se define por **tres elementos fundamentales**:

1. **Atributos (o propiedades)**

Son los **datos** que describen el estado del objeto.
Representan *características* del objeto en la vida real.

Ejemplo:
Un objeto `Coche` puede tener atributos como:

- `color`
- `marca`
- `velocidad`

2. **Métodos (o comportamientos)**

Son las **acciones** que el objeto puede realizar o las operaciones que puede ejecutar.
Definen cómo interactúa el objeto con el exterior y consigo mismo.

Ejemplo:
El objeto `Coche` puede tener métodos como:

- `acelerar()`
- `frenar()`
- `tocarClaxon()`

3. **Identidad**

Es lo que **distingue un objeto de otro**, aunque tengan los mismos atributos.
Dos objetos pueden tener propiedades idénticas, pero siguen siendo instancias distintas.

Ejemplo:
Dos coches rojos de la misma marca, modelo y velocidad siguen siendo **dos coches diferentes**, no el mismo objeto.

---

5. **Clase, objeto e instancia: diferencias y aclaraciones**

¿Qué es una *clase*?

Una **clase** es un *modelo* o *plantilla* que define qué atributos y métodos tendrán sus objetos.
Es una descripción general, **no ocupa memoria** hasta que se crean objetos.

Ejemplo sencillo en C++:

```
class Coche {
public:
    string color;
    void arrancar() { /* ... */ }
};
```

La clase dice *qué es* un coche y *qué puede hacer*.

¿Es lo mismo una clase que un objeto?

No. Son cosas distintas:

- La **clase** es el *concepto*, el plano, la definición.
- El **objeto** es el *resultado real* creado a partir de esa clase.

Analogía:

- Clase = plano de una casa
- Objeto = casa construida a partir del plano

En programación:

```
Coche miCoche;  // ← Esto es un objeto
```

¿Qué es una instancia?

Una **instancia** es simplemente **un objeto creado a partir de una clase**.

Así que:

- "Objeto" e "instancia" suelen referirse a *lo mismo*.
- Se dice que un objeto es *una instancia de la clase X*.

Ejemplo:

```
Coche coche1;  // coche1 es una instancia de Coche
Coche coche2;  // coche2 es otra instancia distinta
```

¿Todos los lenguajes orientados a objetos manejan el concepto de clase?

No.
Muchos sí, pero **no todos los lenguajes orientados a objetos usan clases**.

✔ Lenguajes con clases (clásicos)

- C++
- Java
- C#
- Python (sí tiene clases, aunque sea multiparadigma)

✔ Lenguajes orientados a objetos *sin* clases (basados en prototipos)

- **JavaScript**
- **Lua**

Estos lenguajes están basados en **objetos prototípicos**, donde no hay clases formales:
los objetos se crean copiando otros objetos (los prototipos).

JavaScript, por ejemplo, tiene hoy una sintaxis de "clase", pero *por debajo* sigue funcionando con prototipos.

Resumen rápido

Concepto	Qué es
Clase	Modelo o plantilla que define atributos y métodos.
Objeto	Entidad real creada a partir de una clase; tiene estado propio.
Instancia	Sinónimo práctico de objeto; objeto creado a partir de una clase.
¿Todos los lenguajes usan clases?	No. Algunos usan prototipos (JavaScript, Lua).

6.1. ¿Dónde se almacenan en memoria los objetos?

La respuesta depende del **lenguaje de programación**, pero en general:

✔ En la mayoría de lenguajes orientados a objetos (como **Java**):

Los **objetos se almacenan en el *heap***.

🧠 ¿Por qué en el *heap*?

- El *heap* es la zona de memoria destinada a datos que **pueden cambiar de tamaño** y **viven más tiempo**.
- Los objetos se crean en tiempo de ejecución con `new`, así que necesitan esta flexibilidad.

Ejemplo en Java:

```
Persona p = new Persona(); // el objeto Persona está en el heap
```

La **referencia** p se guarda en la **pila (stack)**, pero
el objeto real está en el *heap*.

✔ ¿Es igual en todos los lenguajes?

No. Cambia según el lenguaje.

◇ Java, C#, Python

→ **Los objetos viven en el *heap***.
La gestión de memoria se realiza automáticamente mediante **recolección de basura**.

◇ C++

En C++ los objetos pueden almacenarse en varios lugares:

- En el **stack**

```
Coche c;   // objeto en stack
```

- En el **heap** (si usas new)

```
Coche* c = new Coche();  // objeto en heap
```

- Incluso en zonas especiales como **memoria estática**.

6.2. ¿Qué es la recolección de basura (garbage collection)?

La **recolección de basura** es un proceso automático que:

✔ Libera memoria ocupada por objetos que **ya no se están usando**

Es decir, por objetos que **no tienen referencias** apuntando hacia ellos.

✔ ¿Qué ventajas tiene?

- Evita fugas de memoria (*memory leaks*)
- Facilita el trabajo del programador
- Hace el programa más seguro y estable

✔ ¿Qué lenguajes la usan?

- Java
- C#
- Python
- Kotlin

✔ ¿Qué lenguajes *no la usan*?

- **C y C++** → El programador debe liberar la memoria manualmente con `delete` o `free`.

▧ Resumen claro

Concepto	Explicación
Dónde están los objetos (Java)	En el *heap*
Dónde está la referencia	En el *stack*
¿Es igual en todos los lenguajes?	No; depende del lenguaje
Recolección de basura	Sistema automático que libera memoria de objetos sin usar

---

7.1. ¿Qué es un método?

Un **método** es una función que pertenece a una **clase** y define una **acción** o **comportamiento** que los objetos de esa clase pueden realizar.

En otras palabras:
➡ **Un método es lo que un objeto *puede hacer*.**

Ejemplo en Java:

```
class Coche {
    void arrancar() {
        System.out.println("El coche está arrancando...");
    }
}
```

`arrancar()` es un **método** de la clase `Coche`.

7.2. ¿Qué es la *sobrecarga de métodos*?

La **sobrecarga de métodos (method overloading)** ocurre cuando **varios métodos tienen el mismo nombre**, pero **diferentes parámetros**.

Los métodos sobrecargados se diferencian por:

- número de parámetros
- tipo de parámetros
- orden de parámetros

👉 **No se diferencian por el tipo de retorno.**

Ejemplo en Java de sobrecarga:

```
class Calculadora {
    int sumar(int a, int b) {
        return a + b;
    }

    double sumar(double a, double b) {
        return a + b;
    }

    int sumar(int a, int b, int c) {
        return a + b + c;
    }
}
```

Aquí tenemos **tres métodos llamados** `sumar`, pero cada uno funciona con parámetros distintos.
Esto es **sobrecarga**.

Resumen rápido

Concepto	Explicación
Método	Acción o comportamiento de un objeto.
Sobrecarga de métodos	Varios métodos con el mismo nombre pero distinta lista de parámetros.

---

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

Clase `Punto` y ejemplo de uso

✔ Requisitos cumplidos:

- Nombre de la clase: **Punto**
- Dos atributos: **x** e **y**
- Visibilidad **por defecto** (es decir, *package-private*)
- Método `calculaDistanciaAOrigen()`
- Ejemplo de creación de un objeto y llamada al método

📌 Código de la clase `Punto`

```
class Punto {
    int x;  // visibilidad por defecto
    int y;  // visibilidad por defecto

    double calculaDistanciaAOrigen() {
        return Math.sqrt(x * x + y * y);
    }
}
```

📌 Ejemplo de uso

```
public class Main {
    public static void main(String[] args) {
        Punto p = new Punto();

        p.x = 3;
        p.y = 4;

        double distancia = p.calculaDistanciaAOrigen();
        System.out.println("Distancia al origen: " + distancia);
    }
}
```

✔ ¿Qué imprimirá?

Como `(3,4)` forma el típico triángulo pitagórico:

```
Distancia al origen: 5.0
```

---

9. Punto de entrada, `static` y `final` en Java

✔ ¿Cuál es el **punto de entrada** de un programa en Java?

En Java, **todo programa comienza ejecutando el método** `main` con esta firma exacta:

```
public static void main(String[] args)
```

Este método es el punto de entrada porque la **JVM** lo busca explícitamente al iniciar el programa.

✔ ¿Qué es `static`?

La palabra clave `static` significa que un atributo o método **pertenece a la clase**, no a las instancias (objetos).

En otras palabras:
➡ Algo `static` existe incluso **sin crear objetos**.

Ejemplo:

```
class Contador {
    static int total = 0;  // pertenece a la clase, no a objetos
}
```

Puedes acceder así:

```
System.out.println(Contador.total);
```

✔ ¿Para qué se usa `static` en el método `main`?

El método `main` debe ser `static` porque:

- La JVM necesita ejecutarlo **sin crear ningún objeto previamente**
- Si fuese no estático, tendrías que instanciar una clase antes de poder llamar a `main`, lo cual es imposible sin que ya exista un punto de entrada

Por eso:

```
public static void main(String[] args)
```

es obligatorio tal cual.

✔ ¿Sólo se usa `static` para el `main`?

No. `static` se usa para otras cosas, por ejemplo:

1 Atributos de clase (variables estáticas)

Compartidos por **todas** las instancias.

```
static int contadorObjetos;
```

2 Métodos de clase

Métodos que se pueden llamar sin crear objetos.

```
static void mostrarMensaje() {
    System.out.println("Hola");
}
```

3 Bloques estáticos

Se ejecutan una vez cuando se carga la clase.

```
static {
    System.out.println("Cargando clase...");
}
```

4 Clases internas estáticas

```
static class Nodo { }
```

✔ ¿Para qué se combina `static` con `final`?

La combinación `static final` se usa para crear **constantes**, es decir:

- Son de la clase (no de los objetos)
- No pueden cambiarse una vez asignadas

Ejemplo típico:

```
static final double PI = 3.14159;
```

Esto sirve para valores constantes que deben ser accesibles desde cualquier parte del programa, y que no deben modificarse.

▧ Resumen claro

Concepto	Explicación
Punto de entrada de Java	`public static void main(String[] args)`
static	Pertenecen a la clase, no a los objetos
¿Solo para `main`?	No, también para atributos, métodos y bloques de clase
static + final	Constantes de clase (no cambian jamás)

---

10. Compilar y ejecutar Java desde terminal

Vamos a usar un ejemplo muy simple:

`Main.java`:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hola desde Java!");
    }
}
```

✔ ¿Cómo *compilar* un programa Java?

Para compilar un archivo `.java` usamos **`javac`**:

```
javac Main.java
```

Si no hay errores, el compilador genera un archivo:

```
Main.class
```

Ese archivo contiene **bytecode**, no código máquina.

✔ ¿Cómo *ejecutar* un programa Java?

Para ejecutarlo usamos `java`:

```
java Main
```

Ojo:
➡ **No se pone** `.class`
➡ Solo el nombre de la clase que contiene `main`.

La salida será:

```
Hola desde Java!
```

¿Java es compilado?

✔ Java es **compilado** *y* **interpretado** (en realidad, ejecutado por la JVM).

Proceso:

1. El código fuente `.java`
   ⬇ es **compilado por** `javac`
2. Se genera `.class` con **bytecode**
   ⬇
3. Ese bytecode es **ejecutado por la JVM** (que lo interpreta y/o lo compila en caliente con JIT)

Por eso Java se considera:

- **Compilado** → hacia bytecode
- **Interpretado/ejecutado por máquina virtual** → la JVM

¿Qué es la **máquina virtual Java (JVM)**?

La **JVM** es un programa que:

- Lee y ejecuta el bytecode
- Hace tu programa independiente del sistema operativo
- Gestiona memoria
- Controla el *garbage collector*
- Optimiza la ejecución (JIT compiler)

El lema famoso de Java:
➡ *"Write once, run anywhere"*
es posible gracias a la JVM.

¿Qué es el *bytecode*?

El **bytecode** es un conjunto de instrucciones **intermedias**, no específicas de ningún procesador real.

- No es código máquina
- No lo entiende la CPU
- Sí lo entiende la JVM

Es lo que aparece dentro de los ficheros `.class`.

¿Qué son los ficheros `.class`?

Son archivos generados por `javac` que contienen **bytecode**.
Cada clase pública genera su propio archivo:

Ejemplo:

Si tu archivo `Main.java` contiene 2 clases:

```
public class Main { ... }
class Auxiliar { ... }
```

Al compilar obtienes:

```
Main.class
Auxiliar.class
```

▧ Resumen claro

Tema	Explicación
Compilar Java	`javac Nombre.java`
Ejecutar Java	`java NombreDeLaClase`
Java es compilado?	Sí → a bytecode; luego ejecutado por la JVM
JVM	La máquina virtual que interpreta/ejecuta el bytecode
Bytecode	Representación intermedia generada por el compilador
.class	Ficheros que contienen el bytecode

11. `new`, los constructores y un ejemplo en Java

✔ ¿Qué es `new` en Java?

`new` es un **operador** que sirve para **crear un objeto en memoria** a partir de una clase.

Cuando escribes:

```
Punto p = new Punto();
```

Estás haciendo **tres cosas**:

1. **Reservar memoria** en el *heap* para un objeto `Punto`.
2. **Ejecutar el constructor** de la clase `Punto`.
3. **Devolver una referencia** al objeto recién creado.

✔ ¿Qué es un *constructor*?

Un **constructor** es un método especial que:

- Tiene **el mismo nombre que la clase**
- **No tiene tipo de retorno** (ni siquiera `void`)
- Se ejecuta automáticamente cuando haces `new`
- Sirve para **inicializar los atributos** del objeto

Si no defines ninguno, Java crea uno por defecto vacío.
Pero normalmente definimos nuestros propios constructores.

✔ Ejemplo: Clase `Empleado` con un constructor

Requisitos del enunciado:

- Clase `Empleado`
- Atributos: DNI, nombre, apellidos
- Un constructor que inicialice esos datos

Aquí tienes el ejemplo en Java:

```
class Empleado {
    String dni;
    String nombre;
    String apellidos;

    // Constructor
    Empleado(String dni, String nombre, String apellidos) {
```

```
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

✔ Ejemplo de uso del constructor

```
public class Main {
    public static void main(String[] args) {
        Empleado e = new Empleado("12345678A", "Carlos", "Pérez Gómez");

        System.out.println("DNI: " + e.dni);
        System.out.println("Nombre: " + e.nombre);
        System.out.println("Apellidos: " + e.apellidos);
    }
}
```

Resumen muy claro

Concepto	Explicación
`new`	Crea un objeto en memoria y llama al constructor
Constructor	Método especial que inicializa los atributos de un objeto
Nombre del constructor	Debe coincidir con el nombre de la clase
Tipo de retorno	Ninguno (ni siquiera `void`)

---

12. ¿Qué es la referencia `this`?

En **Java**, `this` es una **referencia al objeto actual**, es decir, a **la instancia** sobre la que se está ejecutando el método o el constructor. Sirve para:

1. **Diferenciar** entre atributos del objeto y **parámetros con el mismo nombre**.
2. **Pasar la propia instancia** como argumento a otros métodos.
3. **Encadenar constructores** dentro de la misma clase: `this(...)`.
4. Acceder a **métodos** o **atributos** de la instancia de forma explícita (cuando hay ambigüedad o por claridad).

> Importante: `this` **no puede usarse en contextos** `static`, porque en un método estático **no existe** una instancia.

12.1 ¿Se llama igual en todos los lenguajes?

- **Java, C++, C#, JavaScript** → usan `this`.
- **Python** → usa `self` (es un parámetro explícito en los métodos de instancia).
- **Kotlin** → usa `this` (y permite `this@NombreClase` para desambiguar en clases anidadas).
- **Ruby** → usa `self`.

El concepto es equivalente (referirse al objeto actual), pero **el nombre y algunos detalles** de uso pueden variar según el lenguaje.

Ejemplos en Java con la clase `Punto`

Partimos de tu clase mínima y añadimos varios usos útiles de `this`:

1) **Diferenciar atributos y parámetros en el constructor**

```
class Punto {
    int x; // visibilidad por defecto
    int y; // visibilidad por defecto

    // Constructor: los parámetros se llaman igual que los atributos
    Punto(int x, int y) {
        this.x = x; // 'this.x' → atributo; 'x' → parámetro
        this.y = y;
    }

    double calculaDistanciaAOrigen() {
        // 'this' es opcional aquí, pero puede usarse por claridad
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
```

2) **Encadenar constructores con `this(...)`**

```
class Punto {
    int x;
    int y;

    Punto() {
        // Llama al otro constructor de la misma clase
        this(0, 0);
    }

    Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

> Nota: la llamada a `this(...)` debe ser **la primera línea** del constructor.

3) **Devolver `this` para permitir encadenamiento (fluent API)**

```
class Punto {
    int x;
    int y;

    Punto setX(int x) {
        this.x = x;
        return this; // devuelve la instancia actual
    }

    Punto setY(int y) {
        this.y = y;
        return this;
    }

    double calculaDistanciaAOrigen() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}

class Main {
    public static void main(String[] args) {
        Punto p = new Punto().setX(3).setY(4);
        System.out.println(p.calculaDistanciaAOrigen()); // 5.0
    }
}
```

4) **Pasar la instancia actual como argumento**

```
class Utilidades {
    static double distanciaEntre(Punto a, Punto b) {
        int dx = a.x - b.x;
        int dy = a.y - b.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}

class Punto {
    int x;
    int y;

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

    double distanciaA(Punto otro) {
        // Pasamos la instancia actual como 'this'
        return Utilidades.distanciaEntre(this, otro);
```

```
    }
}
```

Resumen rápido

- `this` → referencia a **la instancia actual**.
- Útil para: desambiguar atributos/parámetros, encadenar constructores, patrones de métodos encadenados, y pasar la propia instancia.
- **No** se puede usar en métodos `static`.
- El nombre cambia en algunos lenguajes (p. ej., `self` en Python), pero la **idea es la misma**.

✔ 13. Clase `Punto` con el nuevo método `distanciaA`

```
class Punto {
    int x;  // visibilidad por defecto
    int y;  // visibilidad por defecto

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

    double calculaDistanciaAOrigen() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }

    // Nuevo método solicitado
    double distanciaA(Punto otro) {
        int dx = this.x - otro.x;
        int dy = this.y - otro.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

✔ Ejemplo de uso

```
public class Main {
    public static void main(String[] args) {
        Punto p1 = new Punto(3, 4);
        Punto p2 = new Punto(0, 0);

        double d = p1.distanciaA(p2);
```

```
        System.out.println("Distancia entre p1 y p2: " + d);
    }
}
```

Resultado esperado:

```
Distancia entre p1 y p2: 5.0
```

✔ Explicación breve

- `this.x` → coordenada del punto actual.
- `otro.x` → coordenada del punto pasado como parámetro.
- `dx` y `dy` son las diferencias en cada eje.
- La distancia euclidiana se calcula con: $$ \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} $$

---

14. ¿El **paso de parámetros** en Java es **por copia o por referencia**?

✔ En Java TODO se pasa SIEMPRE por valor (por copia).

Pero cuidado:

- Cuando pasas un **tipo primitivo** → se copia el **valor**.
- Cuando pasas un **objeto** (como `Punto`) → se copia **la referencia** al objeto.

Esto genera dos comportamientos distintos.

✔ Caso 1: Pasar un objeto `Punto`

El método recibe **una copia de la referencia**, pero AMBAS referencias apuntan al **mismo objeto** del *heap*.

Por eso:

➡ **Si modificas atributos del objeto dentro del método, los cambios afectan fuera.**

Ejemplo:

```
void mover(Punto p) {
    p.x = 100;   // modifica el objeto real
}
```

Uso:

```
Punto p1 = new Punto(3, 4);
mover(p1);
System.out.println(p1.x); // 100 → CAMBIÓ
```

✔ ¿Por qué ocurre?

Porque tanto `p1` como el parámetro `p` apuntan al **mismo objeto** en memoria.

Lo que se copia es **la dirección**, no el objeto.

✔ Caso 2: Pasar un entero (`int`)

Los tipos primitivos en Java (*int, double, boolean, char...*) se pasan **por copia del valor**.

Eso significa que si modificas el parámetro dentro del método, NO afecta fuera.

Ejemplo:

```
void cambiarEntero(int n) {
    n = 999;  // solo cambia la copia
}
```

Uso:

```
int a = 5;
cambiarEntero(a);
System.out.println(a); // 5 → NO CAMBIÓ
```

✔ ¿Por qué?

Porque `a` y `n` son **copias independientes** del valor 5.

✔ ¿Entonces Java tiene "paso por referencia"?

✘ No.

Java **no** tiene paso por referencia auténtico (como C++ cuando pasas con `&`).

Lo que tiene es:

- **Paso por valor de primitivos**, y
- **Paso por valor de referencias a objetos**.

Ese matiz es MUY importante.

▧ Resumen final

Tipo pasado a un método	¿Qué se copia?	¿Afectan las modificaciones fuera del método?
Objeto (Punto)	Se **copia la referencia**, no el objeto	✔ Sí, los cambios afectan al objeto real
Primitivo (int)	Se copia **el valor**	❌ No afectan fuera del método

---

15. ¿Qué es el método `toString()` en Java?

En Java, `toString()` es un método que tienen **todas las clases**, porque está definido en la clase base `java.lang.Object`, de la cual **heredan todas las clases**.

✔ ¿Para qué sirve?

`toString()` devuelve una **representación en forma de texto** del objeto.

Su objetivo es:

- Mostrar la información del objeto de forma legible
- Facilitar depuración, logs y mensajes por consola
- Convertir un objeto a cadena automáticamente cuando se usa con `System.out.println()` o concatenaciones

Ejemplo:

```
System.out.println(miPunto);
```

Java internamente hace:

```
miPunto.toString();
```

✔ ¿Existe en otros lenguajes?

Sí, pero con *nombres distintos*:

- **JavaScript** → `toString()` también
- **C#** → `ToString()` con mayúscula
- **Python** → usa `__str__()` para la cadena legible y `__repr__()` para representación oficial
- **C++** → no existe un "toString universal"; se suele sobrecargar `operator<<` o crear un método propio

Es un concepto muy común: "obtener una cadena que describa al objeto".

✔ Ejemplo de `toString()` en la clase `Punto`

```
class Punto {
    int x;  // visibilidad por defecto
    int y;  // visibilidad por defecto

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

    double calculaDistanciaAOrigen() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }

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

    @Override
    public String toString() {
        return "Punto(" + x + ", " + y + ")";
    }
}
```

✔ Ejemplo de uso

```
public class Main {
    public static void main(String[] args) {
        Punto p = new Punto(3, 4);
        System.out.println(p);          // Llama automáticamente a p.toString()
        System.out.println(p.toString()); // Equivalente
    }
}
```

Salida:

```
Punto(3, 4)
Punto(3, 4)
```

Resumen rápido

Concepto	Explicación
`toString()`	Representación textual legible del objeto
Dónde se define	En `Object`, la superclase de todas
Cuándo se llama	En `System.out.println(objeto)` o al concatenar cadenas
¿Existe en otros lenguajes?	Sí, pero con nombres distintos (`__str__`, `ToString`, etc.)

---

16. ¿Una clase es como un `struct` en C?

✔ Respuesta corta:

NO, un `struct` en C *no es* una clase...
Pero **sí se parece a la parte de "atributos" de una clase**.

Un `struct` en C es simplemente un **contenedor de datos**.
No tiene **métodos**, ni **constructores**, ni **encapsulamiento**, ni **herencia**, ni **polimorfismo**, ni nada que forme parte de la Programación Orientada a Objetos.

Ejemplo de `struct` en C:

```c
struct Punto {
    int x;
    int y;
};
```

Esto solo define dos campos. No puede tener comportamiento asociado.

✔ ¿Qué le falta al `struct` para ser como una clase?

Para que un `struct` en C fuese equivalente a una **clase** en Java, le faltarían **varias características esenciales**:

✔ 1. **Métodos dentro del propio tipo**

Una clase tiene métodos:

```
class Punto {
    int x, y;

    double distanciaAOrigen() { ... }
}
```

Un `struct` en C **no puede** tener funciones dentro.
Solo se puede simular haciendo funciones externas que reciban un puntero al struct.

✔ 2. **Constructores**

Las clases permiten inicializar automáticamente el objeto:

```
Punto p = new Punto(3, 4);
```

En C no hay constructores. Hay que asignar "a mano":

```
struct Punto p = {3, 4};
```

✔ 3. **Encapsulamiento**

En Java puedes controlar el acceso:

```
private int x;
public int getX() { return x; }
```

En C, un `struct` **no puede tener visibilidad privada**.
Todo es público.

✔ 4. **Herencia**

Una clase puede extender a otra:

```
class Empleado extends Persona { }
```

En C, los `struct` **no pueden heredar** (solo puedes "simular" algo parecido copiando campos, pero no es herencia real).

✔ 5. **Polimorfismo**

En Java puedes sobrescribir métodos y usar referencias polimórficas:

```
Persona p = new Empleado();
p.saludar(); // polimorfismo
```

En C, imposible con un struct (salvo con técnicas manuales complejas como "tablas de función", pero no forman parte del lenguaje).

✔ 6. `this` / referencia implícita al objeto

En una clase, los métodos tienen una referencia automática al objeto (`this`).

En C, las funciones no saben a qué struct pertenecen.

✔ 7. **Instancias reales**

En Java:

```
Punto p = new Punto();
```

`p` es una instancia.

En C:

```
struct Punto p;
```

Esto **no es una instancia orientada a objetos**, es solo una región de memoria con datos.

▧ Resumen final

Característica	Struct en C	Clase en Java
Atributos	✔ Sí	✔ Sí
Métodos	✘ No	✔ Sí
Constructores	✘ No	✔ Sí

Característica	Struct en C	Clase en Java
Encapsulamiento	❌ No	✔ Sí
Herencia	❌ No	✔ Sí
Polimorfismo	❌ No	✔ Sí
`this`	❌ No	✔ Sí
Instancias	⚠ Solo datos	✔ Objetos reales

✔ Conclusión clara

Un `struct` en C es equivalente **solo a la parte de datos** de una clase.
Para convertirse en una "clase real", necesitaría:

👉 **métodos**,
👉 **constructores**,
👉 **encapsulamiento**,
👉 **herencia**,
👉 **polimorfismo**,
👉 **comportamiento asociado**,
👉 **y una referencia implícita al objeto (`this`)**.

Es decir, le falta casi *todo* lo que define la POO.

---

17. ¿Cómo emular una clase `Punto` de Java usando `struct` en C?

En C **no existen clases**, **ni métodos**, **ni this**, **ni constructores**.
Pero podemos **simular** la idea con:

- Un `struct` → para almacenar los datos
- Una función externa → para operar sobre estos datos
- Pasar un puntero al struct → para que la función sepa *qué objeto* estamos usando
  (esto es lo que sustituye a `this`)

✔ Versión en C que imita la clase `Punto`

1) Definición del "objeto" usando `struct`:

```
struct Punto {
    int x;
```

```
    int y;
};
```

Esto solo contiene *datos*. No puede contener funciones.

---

2) "Constructor" simulado: una función que inicializa el struct

```
void Punto_init(struct Punto* p, int x, int y) {
    p->x = x;
    p->y = y;
}
```

📌 En Java escribirías:

```
this.x = x;
```

Pero en C:

```
p->x = x;
```

porque `p` es un puntero al struct.

---

3) Función que imita a `calculaDistanciaAOrigen()`

```
#include <math.h>

double Punto_calculaDistanciaAOrigen(struct Punto* p) {
    return sqrt(p->x * p->x + p->y * p->y);
}
```

Esta función sería un **método** en Java, pero en C está *fuera* del struct.

---

4) Ejemplo de uso

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

struct Punto {
    int x;
```

```
    int y;
};

void Punto_init(struct Punto* p, int x, int y) {
    p->x = x;
    p->y = y;
}

double Punto_calculaDistanciaAOrigen(struct Punto* p) {
    return sqrt(p->x * p->x + p->y * p->y);
}

int main() {
    struct Punto p;
    Punto_init(&p, 3, 4);

    double d = Punto_calculaDistanciaAOrigen(&p);
    printf("Distancia al origen: %.2f\n", d);
    return 0;
}
```

Salida:

```
Distancia al origen: 5.00
```

✔ ¿Qué ha pasado con `this`?

En Java:

- `this` es una **referencia implícita** al objeto actual.
- No tienes que pasarla como parámetro: está *automáticamente disponible*.

Ejemplo en Java:

```
double calculaDistanciaAOrigen() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

En C:

- No existe `this`.
- Si quieres saber *a qué struct* te refieres, **tienes que pasarlo como parámetro**.

Ejemplo equivalente en C:

```c
double Punto_calculaDistanciaAOrigen(struct Punto* p) {
    return sqrt(p->x * p->x + p->y * p->y);
}
```

Aquí, `p` **es el** `this` **manual**.

✔ Resumen claro

Concepto	Java (POO)	C (struct + funciones)
Datos	atributos dentro de la clase	campos del struct
Métodos	dentro de la clase	funciones externas
this	implícito	hay que pasarlo como puntero
Constructor	método especial	función normal (`Punto_init`)
Invocación	`p.calculo()`	`Punto_calculo(&p)`

Conclusión:

👉 En C **tú mismo gestionas** lo que Java hace automáticamente:

- pasar `this`
- inicializar el objeto
- separar datos y funciones
- gestionar visibilidad
- evitar errores de punteros

👉 Java aporta **abstracción, encapsulamiento, y organización**, mientras que C te da control total pero a mano.
