POO en otros Lenguajes

Programación Orientada a Objetos (CC2008) - 2026

POO en otros lenguajes

Semestre 02, 2026

Java no inventó la POO

La Programación Orientada a Objetos es un paradigma, no un lenguaje. Java es una de sus implementaciones más populares, pero está lejos de ser la única.

Simula 67 introdujo las clases y los objetos casi treinta años antes que Java. Smalltalk formalizó el envío de mensajes. C++ llevó la POO al mundo de los sistemas.

Aprender cómo otros lenguajes resuelven los mismos problemas sirve para separar el concepto de la sintaxis. Cuando alguien dice "una clase", debería poder escribirla en cualquier lenguaje.

Los pilares son portátiles

Los cuatro pilares que ya conocen aparecen en todos los lenguajes de esta presentación, aunque con nombres y reglas distintas.

- Abstracción: modelar lo esencial y esconder el resto.

- Encapsulamiento: controlar quién toca los datos.
- Herencia: reutilizar y especializar comportamiento.
- Polimorfismo: un mismo contrato, muchas implementaciones.

Lo que cambia entre lenguajes es qué tan estricto es el compilador, cuánta ceremonia exige y qué mecanismo usa para el polimorfismo.

Mapa rápido

- C#: el pariente cercano de Java, con azúcar sintáctica que elimina mucho código repetitivo.
- Python: dinámico, permisivo y con herencia múltiple real.
- Rust: sin herencia, con traits y composición.
- PHP: herencia simple con traits para reutilizar sin heredar.

C# — el pariente cercano

Creado por Microsoft en 2000 y diseñado por Anders Hejlsberg, el mismo autor de Turbo Pascal y Delphi. Corre sobre .NET.

La sintaxis es tan parecida a Java que se puede leer sin traducción. Las diferencias interesantes están en lo que C# agregó después.

```
public class Estudiante
{
    public string Nombre { get; set; }
    public int Carne { get; init; }

    public Estudiante(string nombre, int carne)
    {
```

```
        Nombre = nombre;
        Carne = carne;
    }

    public override string ToString() => $"{Carne} - {Nombre}";
}
```

C# — propiedades

En Java, exponer un atributo de forma controlada obliga a escribir un getter y un setter por cada campo. En C# eso se declara en una línea.

```
// Java
private String nombre;
public String getNombre() { return nombre; }
public void setNombre(String n) { this.nombre = n; }

// C#
public string Nombre { get; set; }
```

La propiedad se usa como si fuera un atributo público, pero por debajo sigue siendo una llamada a método. Se puede cambiar la implementación después sin romper a quien la usa.

El modificador `init` permite asignar el valor solo durante la construcción del objeto, lo que da inmutabilidad sin escribir un constructor defensivo.

C# — virtual y override explícitos

En Java todo método de instancia es sobrescribible por defecto. En C# hay que pedirlo con `virtual` y aceptarlo con `override`.

```
public class Animal
{
    public virtual void HacerSonido() => Console.WriteLine("...");
}

public class Perro : Animal
{
    public override void HacerSonido() => Console.WriteLine("Guau");
}
```

La ventaja es que el diseño de la clase base queda documentado en el código. Un método sin `virtual` es un contrato cerrado y el compilador lo defiende.

C# — records

Para clases que solo transportan datos, C# genera constructor, comparación por valor, `ToString` y descomposición automáticamente.

```
public record Punto(double X, double Y);

var p1 = new Punto(1, 2);
var p2 = new Punto(1, 2);

Console.WriteLine(p1 == p2);  // True, compara por valor
Console.WriteLine(p1);        // Punto { X = 1, Y = 2 }
```

Java tiene `record` desde la versión 16, así que esta idea ya viajó de regreso.

C# — herencia simple, igual que Java

Una aclaración importante, C# tampoco tiene herencia múltiple de clases. La documentación oficial es explícita en que una clase derivada puede tener una sola clase base directa.

Lo que sí ofrece es implementar varias interfaces y, desde C# 8, interfaces con implementación por defecto.

```
public interface IFigura
{
    double Area();
    void Mostrar() => Console.WriteLine($"Area: {Area()}");
}
```

El lenguaje con herencia múltiple real de esta presentación es Python. C++ también la tiene, y de ahí viene el problema del diamante que Java decidió evitar.

C# — ventajas

- LINQ para consultar colecciones con una sintaxis declarativa.
- `async` y `await` integrados en el lenguaje.
- Structs para tipos por valor, sin pasar por el heap.
- Sobrecarga de operadores.
- Métodos de extensión para agregar comportamiento a clases ajenas.
- Tipos de referencia anulables verificados por el compilador.

El problema del diamante

```
classDiagram
    direction TB
    class Dispositivo {
        +encender()
    }
    class Camara {
        +encender()
    }
```

```
    class Microfono {
        +encender()
    }
    Camara --|> Dispositivo
    Microfono --|> Dispositivo
    Celular --|> Camara
    Celular --|> Microfono
```

Una clase hereda de dos clases que a su vez heredan de una misma tercera, y las dos sobrescriben el mismo método. Celular no define `encender`, lo hereda, y Dispositivo le llega por dos caminos distintos. Esa forma de rombo es la que le da nombre al problema.

El diamante — lo que el lenguaje tiene que decidir

- Precedencia: Camara y Microfono definen `encender`. Al llamarlo sobre un Celular hay que elegir uno de los dos, y esa elección no está escrita en ninguna de las cuatro clases.
- Repetición: los dos padres delegan en la clase de arriba. Si cada rama sube por su cuenta hasta la raíz, el `encender` de Dispositivo termina ejecutándose dos veces.
- Estado: si Dispositivo lleva un contador o abre un recurso, esa segunda ejecución deja el objeto en un estado que nadie escribió y que nadie revisó.

El diamante — la ejecución doble

Dos recorridos posibles sobre el mismo diamante, con resultados distintos.

- En profundidad: Celular, Camara, Dispositivo, Microfono, Dispositivo. Cada rama sube completa antes de pasar a la siguiente, así que la raíz aparece dos veces. Es lo que hacían las clases antiguas de Python, antes de la versión 2.3.
- Linearización: Celular, Camara, Microfono, Dispositivo. La raíz se pospone hasta que ya pasaron todas las clases que dependen de ella, así que se ejecuta una sola vez.

La diferencia no es estética. Con el recorrido en profundidad el contador de Dispositivo queda en dos y el método aparece dos veces en la salida, sin que ninguna de las cuatro clases tenga un error.

El diamante — los lenguajes expuestos

- C++ permite herencia múltiple de clases sin restricciones. Es el caso original.
- Python permite herencia múltiple y fija el orden con una linearización.
- Eiffel la llama herencia repetida y obliga a desambiguar con `rename` y `select`.
- Perl permite herencia múltiple y desde la versión 5.10 ofrece C3.
- Scala mezcla muchos traits y ordena el resultado con una linearización.
- PHP mezcla muchos traits y obliga a elegir a mano cuando chocan.

El diamante — dos formas del mismo conflicto

- En C++, Python, Eiffel y Perl el conflicto nace de un rombo, una misma raíz que llega por varios caminos.
- En Scala y PHP no hay rombo, solo bloques planos que traen el mismo nombre, así que la resolución es más simple.

El diamante — seis respuestas distintas

Cada lenguaje tomó una decisión distinta frente a este problema.

- Java lo prohíbe y permite una sola clase base con varias interfaces.
- C# hace lo mismo y desde la versión 8 exige resolver a mano los conflictos entre interfaces con implementación por defecto.
- Python lo permite y lo resuelve con el MRO.
- Rust lo elimina de raíz al no tener herencia.
- PHP lo prohíbe entre clases, y cuando aparece entre traits obliga a elegir con `insteadof`.
- C++ lo permite y ofrece herencia virtual para desambiguar.

Ninguna decisión es la correcta. Cada una intercambia flexibilidad por predictibilidad.

Python — dinámico y permisivo

Python es multiparadigma. Se puede escribir un script sin una sola clase, o un sistema completo orientado a objetos.

La diferencia más visible frente a Java es la ausencia de tipos declarados y de modificadores de acceso.

```
class Estudiante:
    def __init__(self, nombre, carne):
        self.nombre = nombre
        self.carne = carne

    def __str__(self):
        return f"{self.carne} - {self.nombre}"
```

```
e = Estudiante("Ana", 24001)
print(e)
```

No hay `new`, no hay `public`, no hay tipo de retorno. El parámetro `self` es explícito y equivale al `this` de Java.

Python — encapsulamiento por convención

La documentación oficial lo dice sin rodeos, las variables de instancia privadas que no se puedan acceder desde fuera no existen en Python.

- Un nombre con un guion bajo `_saldo` es una señal de "no lo toques", pero nada lo impide.
- Un nombre con dos guiones bajos `__saldo` activa el name mangling y se renombra a `_Clase__saldo`.

```
class Cuenta:
    def __init__(self):
        self.__saldo = 0

c = Cuenta()
print(c.__saldo)          # AttributeError
print(c._Cuenta__saldo)   # 0, sigue ahí
```

El name mangling está diseñado para evitar accidentes entre clases padre e hija, no para imponer seguridad.

Python — propiedades

El decorador `@property` cumple el mismo papel que las propiedades de C#. Permite empezar con un atributo público y agregarle validación después sin

cambiar el código que lo usa.

```
class Circulo:
    def __init__(self, radio):
        self._radio = radio

    @property
    def area(self):
        return 3.14159 * self._radio ** 2

c = Circulo(2)
print(c.area)   # se lee como atributo, se ejecuta como metodo
```

Python — herencia múltiple real

Aquí está la diferencia grande con Java. Una clase de Python puede heredar de varias clases a la vez.

```
class Dispositivo:
    def encender(self):
        print("Encendiendo dispositivo")

class Camara(Dispositivo):
    def encender(self):
        print("Encendiendo camara")
        super().encender()

class Microfono(Dispositivo):
    def encender(self):
        print("Encendiendo microfono")
        super().encender()

class Celular(Camara, Microfono):
    pass

Celular().encender()
```

La salida es camara, microfono, dispositivo. Cada clase se ejecuta una sola vez, aunque `Dispositivo` aparezca por dos caminos distintos.

Python — el MRO fija el orden

```
print(Celular.__mro__)
# (Celular, Camara, Microfono, Dispositivo, object)
```

La documentación de Python describe tres garantías del algoritmo. Preserva el orden izquierda a derecha de cada clase, coloca a cada clase antes que a todos sus padres y es monótono, es decir que subclasificar no altera la precedencia ya establecida.

La llamada a `super` no significa clase padre, significa la siguiente clase en el MRO del objeto real. Por eso el `super` de Camara lleva a Microfono y no a Dispositivo, aunque Camara no sepa que Microfono existe.

Python — cuando no existe un orden válido

```
class A: pass
class B: pass

class C(A, B): pass
class D(B, A): pass

class E(C, D): pass
# TypeError: Cannot create a consistent
# method resolution order (MRO) for bases A, B
```

C exige que A vaya antes que B y D exige exactamente lo contrario. No hay ningún orden que respete a las dos, así que no es un caso difícil, es un caso

imposible.

Python rechaza la clase al definirla y no al llamar al método. El costo de la herencia múltiple es real, pero se paga temprano y con un mensaje que dice cuáles son las bases en conflicto.

Python — duck typing

Python no exige que una clase declare que implementa una interfaz. Si el objeto tiene el método, sirve.

```
class Pato:
    def hablar(self):
        return "Cuac"

class Robot:
    def hablar(self):
        return "Beep"

for x in [Pato(), Robot()]:
    print(x.hablar())
```

Ni `Pato` ni `Robot` comparten una clase base. El polimorfismo funciona por la forma del objeto, no por su tipo declarado. De ahí el nombre, si camina como pato y suena como pato, es un pato.

El costo es que el error aparece en tiempo de ejecución y no en compilación.

Python — ventajas

- Sintaxis mínima, mucho menos código para el mismo modelo.
- Herencia múltiple con un orden de resolución bien definido.

- Métodos especiales que integran las clases propias con el lenguaje, como `__len__`, `__eq__` y `__add__`.
- Ecosistema enorme en ciencia de datos y automatización.
- Ideal para prototipar un diseño antes de llevarlo a un lenguaje estricto.

Rust — objetos sin herencia

Rust es un lenguaje de sistemas enfocado en seguridad de memoria sin recolector de basura.

El Rust Book responde directamente la pregunta de si Rust es orientado a objetos. Tiene objetos, porque los structs y los enums guardan datos y los bloques `impl` les dan métodos. Tiene encapsulamiento, porque todo es privado por defecto y se expone con `pub`.

Pero también dice lo contrario sobre el tercer pilar: si un lenguaje necesita herencia para ser orientado a objetos, entonces Rust no lo es. No hay forma de definir un struct que herede los campos y los métodos de otro.

Rust — traits

Un trait declara qué método debe tener un tipo y con qué firma, sin decir cómo lo resuelve. Es lo más parecido a una interfaz de Java.

```
trait RayIntersect {
    fn ray_intersect(&self, origen: &Vec3, direccion: &Vec3) -> bool;
}
```

Rust — cada tipo trae su propia implementación

```
impl RayIntersect for Sphere {
    fn ray_intersect(&self, origen: &Vec3, direccion: &Vec3) -> bool {
        // la matemática propia de la esfera
    }
}

impl RayIntersect for Cube {
    fn ray_intersect(&self, origen: &Vec3, direccion: &Vec3) -> bool {
        // la matemática propia del cubo
    }
}
```

Cada tipo resuelve el método a su manera dentro de su propio bloque `impl`. Sphere y Cube no heredan de nada ni comparten una clase base, responden al mismo mensaje solo porque los dos implementan el mismo trait.

Rust — composición sobre herencia

Cuando una clase de Java heredaría, Rust incrusta el otro tipo como campo y delega.

```
struct Motor {
    caballos: u32,
}

struct Carro {
    motor: Motor,
    placa: String,
}

impl Carro {
    fn potencia(&self) -> u32 {
        self.motor.caballos
    }
}
```

Es la relación "tiene un" que ya vieron en el tema de relaciones entre clases, elevada a única opción del lenguaje.

Rust — ventajas

- Sin recolector de basura y sin punteros nulos.
- El compilador garantiza que no haya condiciones de carrera entre hilos.
- Los traits se pueden implementar sobre tipos ajenos.
- Enums con datos asociados que hacen innecesarias muchas jerarquías de clases.
- Obliga a diseñar por composición, evitando jerarquías frágiles.

PHP — objetos para la web

Rasmus Lerdorf creó PHP en 1994 como un conjunto de scripts para su página personal. El modelo de objetos que se usa hoy llegó con PHP 5 en 2004 y se reescribió por completo para PHP 7.

Hoy tiene tipos declarados, visibilidad verificada por el motor, clases abstractas, interfaces, enums y traits.

La mayoría de los sitios web del mundo corre sobre PHP, casi siempre a través de frameworks orientados a objetos como Laravel o Symfony.

Todo el código de esta sección vive dentro de un archivo con extensión php, después de la etiqueta de apertura del lenguaje. Se omite aquí para dejar el modelo de objetos a la vista.

PHP — una clase

```
class Estudiante
{
    public function __construct(
        public string $nombre,
        public readonly int $carne
    ) {}

    public function __toString(): string
    {
        return "{$this->carne} - {$this->nombre}";
    }
}

$e = new Estudiante("Ana", 24001);
echo $e;
```

El constructor siempre se llama `__construct`, sin importar el nombre de la clase. Escribir la visibilidad delante de cada parámetro declara el atributo y lo asigna en el mismo lugar, así que el cuerpo queda vacío. Esta forma abreviada se llama promoción de propiedades y existe desde PHP 8.

El modificador `readonly` permite asignar el atributo una sola vez y solo desde dentro de la clase. Cumple el papel del `final` de Java sobre un campo y del `init` de C#.

`__toString` es un método mágico y equivale a sobrescribir `toString` en Java. Los nombres con doble guion bajo conectan la clase con el lenguaje, igual que en Python.

Los cambios de superficie son que toda variable empieza con signo de dólar y que el acceso a miembros usa flecha en lugar de punto.

PHP — visibilidad verificada

Aquí está la diferencia grande con Python. El encapsulamiento de PHP no es una convención, lo hace cumplir el motor.

```
class Cuenta
{
    private float $saldo = 0;

    public function depositar(float $monto): void
    {
        $this->saldo += $monto;
    }
}

$c = new Cuenta();
$c->saldo = 1000;   // Error: Cannot access private property
```

Las palabras `public`, `protected` y `private` significan exactamente lo mismo que en Java. El error aparece en tiempo de ejecución porque PHP no tiene una fase de compilación separada.

PHP — herencia simple y clases abstractas

```
abstract class Animal
{
    public function __construct(protected string $nombre) {}

    abstract public function hacerSonido(): string;

    public function presentarse(): string
    {
        return "{$this->nombre} dice {$this->hacerSonido()}";
    }
}

final class Perro extends Animal
```

```
{
    public function hacerSonido(): string { return "Guau"; }
}
```

Una clase extiende a lo sumo una clase padre, igual que en Java y en C#. Las palabras `abstract` y `final` tienen el mismo significado, y `parent` sustituye a `super` para llamar al método de la clase base.

PHP — interfaces múltiples

```
interface Figura { public function area(): float; }

interface Dibujable { public function dibujar(): void; }

class Circulo implements Figura, Dibujable
{
    public function __construct(private float $radio) {}

    public function area(): float
    {
        return M_PI * $this->radio ** 2;
    }

    public function dibujar(): void
    {
        echo "Circulo de area {$this->area()}";
    }
}
```

Una clase implementa tantas interfaces como necesite, y una interfaz puede extender a varias interfaces a la vez. Es la misma salida que eligió Java para esquivar el problema del diamante.

PHP — traits

La documentación oficial define el trait como un mecanismo para reducir las limitaciones de la herencia simple, reutilizando conjuntos de métodos en clases independientes que viven en jerarquías distintas.

```
trait Registrable
{
    public function registrar(string $mensaje): void
    {
        echo date("H:i:s") . " " . $mensaje;
    }
}

trait Exportable
{
    public function aJson(): string
    {
        return json_encode(get_object_vars($this));
    }
}

class Pedido
{
    use Registrable, Exportable;
}
```

Inyecta sus métodos y atributos en la clase, como si estuvieran escritos ahí. Una clase puede usar todos los traits que quiera.

Lo que no hace es crear una relación es un. `Pedido` no es un `Registrable` y el trait no sirve como tipo, así que no reemplaza a las interfaces. Lo habitual es una interfaz para el contrato y un trait para el código.

PHP — conflictos entre traits

```
trait Ave { public function moverse(): string { return "Vuela"; } }
trait Pez { public function moverse(): string { return "Nada"; } }

class Pinguino
{
    use Ave, Pez {
        Pez::moverse insteadof Ave;
        Ave::moverse as intentarVolar;
    }
}
```

Cuando dos traits traen el mismo método, PHP obliga a elegir con `insteadof` y deja conservar el otro con `as`.

Sin conflicto el orden es la clase, luego los traits y al final el padre.

PHP — enums con comportamiento

```
enum Estado: string
{
    case Pendiente = "pendiente";
    case Enviado   = "enviado";
    case Entregado = "entregado";

    public function etiqueta(): string
    {
        return match($this) {
            Estado::Pendiente => "En espera",
            Estado::Enviado   => "En camino",
            Estado::Entregado => "Recibido",
        };
    }
}
```

```
echo Estado::Enviado->etiqueta();
```

Desde PHP 8.1 un enum es un tipo con métodos propios que incluso puede implementar interfaces. Reemplaza jerarquías de clases enteras que solo existían para representar un conjunto cerrado de valores.

PHP — ventajas

- Diseñado para la web. Cada petición arranca y termina limpia.
- Traits para reutilizar comportamiento entre clases sin construir una jerarquía.
- Métodos mágicos que integran las clases propias con el lenguaje.
- Frameworks maduros como Laravel y Symfony, construidos por completo sobre POO.
- Tipos declarados opcionales, con herramientas externas que los verifican antes de ejecutar.
- Arrastra una biblioteca estándar con nombres y parámetros inconsistentes.

Comparación

| Concepto | Java | C# | Python | Rust | PHP | | --- | --- | --- | --- | --- | --- | | Tipado | Estático | Estático | Dinámico | Estático | Gradual | | Herencia de clases | Simple | Simple | Múltiple | No existe | Simple | | Interfaces | Sí | Sí | Protocolos | Traits | Sí | | Reuso horizontal | Interfaces por defecto | Interfaces por defecto | Herencia múltiple | Traits | Traits | | Sobrescritura | Por defecto | Explícita | Por defecto | No aplica | Por defecto | | Privacidad | Compilador | Compilador | Convención | Compilador | Motor en ejecución | | Memoria | Recolector | Recolector | Recolector | Ownership | Recolector | | Polimorfismo | Interfaces | Interfaces | Duck typing |

Traits | Interfaces |

Qué llevarse

- Los pilares de la POO son independientes del lenguaje. La sintaxis es lo que se olvida al cambiar de lenguaje, el diseño es lo que se transfiere.
- La herencia múltiple no es un lujo que a Java le falte, es una decisión de diseño con un costo asociado.
- La composición funciona en todos los lenguajes. Rust demuestra que se puede construir un sistema completo sin herencia.
- Los traits de PHP y de Rust muestran que se puede compartir comportamiento entre clases sin construir una jerarquía.
- Aprender un segundo lenguaje orientado a objetos es mucho más barato que aprender el primero.

Para pensar

Tomen el último proyecto de este curso y respondan estas preguntas.

- ¿Cuántas de sus interfaces desaparecerían si el lenguaje tuviera duck typing?
- ¿Cuántas de sus jerarquías de herencia se podrían reescribir como composición?
- ¿Qué parte del código se volvería más segura con un compilador que verifique la propiedad de la memoria?