Introducción a POO

Programación Orientada a Objetos (CC2008) - 2026

Introducción a POO

Semestre 02, 2025

Contexto

- El software moderno es cada vez más complejo.
- Los programas requieren modelar entidades del mundo real.
- El paradigma estructurado resulta limitado al manejar grandes volúmenes de datos y relaciones.
- Necesitamos una forma de organizar el código de manera más cercana a cómo pensamos y entendemos el mundo.

Paradigma estructurado (procedural)

- Basado en funciones y procedimientos que operan sobre datos.
- El flujo de control es difícil de seguir en sistemas grandes.

- Baja reutilización de código.
- Difícil mantenimiento y escalabilidad.
- Poca correspondencia entre el modelo del problema y la solución en código.

Evolución del software

- En el mundo real interactuamos con "cosas" que tienen **estado** y **comportamiento**.
- Queremos representar entidades como: clientes, productos, autos, sensores, etc.
- La POO propone una solución basada en **objetos**: unidades que encapsulan datos + lógica.
- Permite modelar sistemas complejos de forma más natural.

Fundamentos

¿Qué es la Programación Orientada a Objetos?

- Paradigma de programación basado en el concepto de **objetos**.
- Un objeto tiene:
    - **Atributos**: representan su estado.

    - **Métodos**: representan su comportamiento.

- Los objetos se crean a partir de **estructuras llamadas clases**.

- La idea es encapsular datos y lógica en una unidad coherente.

Comparación con otros paradigmas

| | Estructurada | POO | |---|---|---| | Organización | Funciones y datos separados | Entidades combinadas (objetos) | | Reutilización | Limitada | Alta (herencia, composición) | | Escalabilidad | Baja | Alta | | Mantenimiento | Difícil | Más sencillo y modular |

Ventajas

- Mejor organización del código
- Mayor modularidad y reutilización
- Facilita el mantenimiento y escalabilidad
- Más cercano al modelo mental humano

Desventajas

- Mayor curva de aprendizaje
- Puede haber sobreingeniería
- No siempre es necesario para problemas simples

Aplicaciones

- Desarrollo de software a gran escala
- Aplicaciones móviles (Android usa Java/Kotlin)
- Videojuegos y simulaciones
- Sistemas empresariales y backend
- Interfaces gráficas de usuario (GUI)
- Videojuego:
  - `Jugador`, `Enemigo`, `Arma`
- Sistema bancario:
  - `Cuenta`, `Cliente`, `Transacción`
- Aplicación de mensajería:
  - `Usuario`, `Mensaje`, `Chat`

¿Qué es un compilador?

Un **compilador** es un programa que traduce el código fuente escrito en un lenguaje de programación de alto nivel a código máquina (binario) o a un formato intermedio que puede ser ejecutado.

Etapas de compilación

1. **Análisis léxico (tokenización)**
    - El código fuente se divide en unidades mínimas llamadas *tokens* (palabras clave, identificadores, operadores).

2. **Análisis sintáctico (parsing)**
    - Verifica la estructura gramatical del código y construye un árbol sintáctico.

3. **Análisis semántico**
    - Comprueba tipos de datos, declaraciones y el significado de las instrucciones.

4. **Generación de código intermedio**
    - Traduce el código fuente a un lenguaje intermedio independiente del hardware.

5. **Optimización**
    - Mejora el código intermedio para mayor eficiencia.

6. **Generación de código máquina**
    - Convierte el código optimizado en binario ejecutable para el procesador.

Compiladores vs. Intérpretes

| Característica | Compilador | Intérprete | |---|---|---| | Traducción | Traduce todo el código a ejecutable | Traduce y ejecuta línea a línea| | Tiempo de ejecución | Más rápido | Más lento |

| Característica | Compilador | Intérprete | |---|---|---| | Depuración | Menos flexible | Más fácil (error en tiempo real)| | Ejemplo de lenguajes | C, C++, Rust, Go | Python, Ruby, JavaScript | | Salida | Archivo binario (*.exe, *.out) | Ningún archivo binario |

| Característica | Compilador | Intérprete | |---|---|---| | Ejemplo de lenguajes | C, C++, Rust, Go | Python, Ruby, JavaScript | | Salida | Archivo binario (*.exe, *.out) | Ningún archivo binario |

Java

Java utiliza un enfoque **híbrido**:

1. El compilador `javac` traduce el código fuente (*.java*) a **bytecode** (*.class*).
2. Este bytecode no es código máquina; es ejecutado por la **Máquina Virtual de Java (JVM)**.
3. La JVM interpreta o compila el bytecode en tiempo de ejecución usando técnicas como **Just-In-Time (JIT) compilation**.

Ventajas

- **Portabilidad:** El bytecode puede ejecutarse en cualquier plataforma con JVM.
- **Seguridad:** La JVM proporciona un sandbox para el código.
- **Optimización en tiempo real:** Gracias al compilador JIT.

Flujo en Java

```
Código fuente (.java)
         ↓
Compilador Java (javac)
         ↓
Bytecode (.class)
         ↓
Máquina Virtual de Java (JVM)
         ↓
Código máquina (ejecución en tiempo real)
```