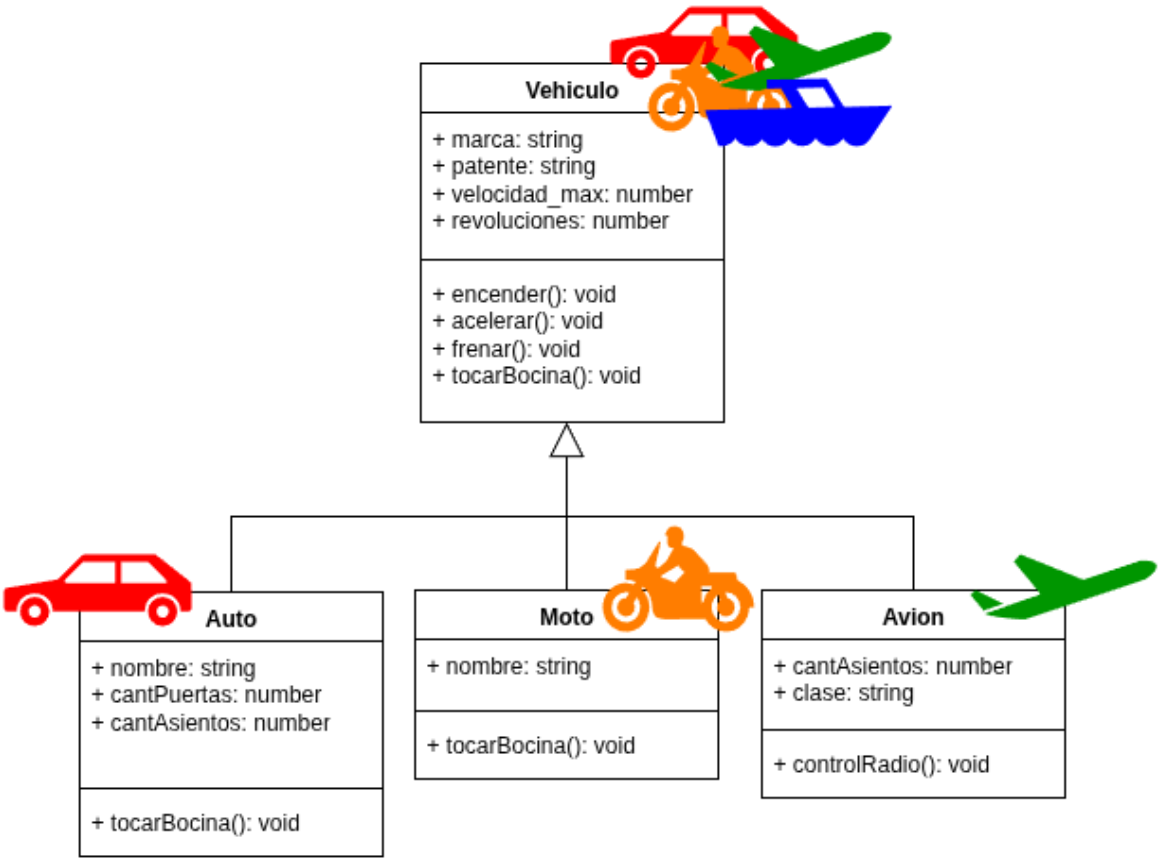
Programación orientada a objetos

La **programación orientada a objetos (POO)** es un paradigma que organiza el código en *objetos* que encapsulan **estado** (datos o atributos) y **comportamiento** (métodos o funciones). En POO se modelan entidades del mundo real con sus propiedades y acciones. Este enfoque enfatiza el encapsulamiento y la modularidad; por ejemplo, en Java se definen **clases** (plantillas) e instancias de esas clases. Características clave de la POO incluyen la **herencia** (clases derivadas que heredan atributos y métodos), el **polimorfismo** (objetos de distintas clases compartiendo interfaz) y el **encapsulamiento** (ocultar datos internos en los objetos). En resumen, los programas orientados a objetos definen primero las clases y luego trabajan con sus instancias para ejecutar métodos específicos.

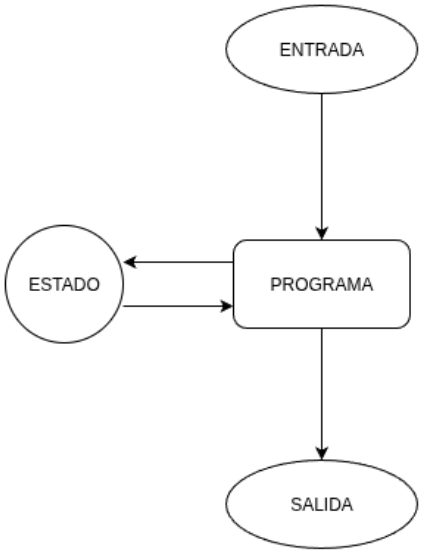
- **Objetos**: Entidades con atributos (datos) y métodos (funciones) que representan conceptos o entidades concretas.
- **Clases e instancias**: Una *clase* define la estructura (campos y métodos) y se crean *instancias* de ella. En Java se usan clases explícitas.
- **Encapsulación**: Cada objeto es responsable de sus datos; se interactúa con él solo mediante sus métodos públicos.
- **Herencia y Polimorfismo**: Las clases pueden extender otras (herencia) y objetos pueden comportarse según su tipo real (polimorfismo), permitiendo reutilizar y extender código.

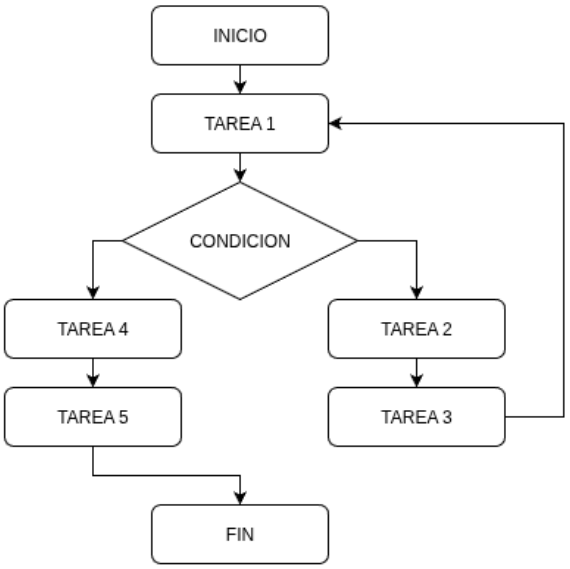


Programación imperativa

La **programación imperativa** es un paradigma clásico basado en instrucciones secuenciales que alteran el *estado* del programa. Un programa imperativo se ve como "una serie de órdenes" paso a paso para el ordenador. En este estilo se usan estructuras de control para definir el flujo: **condicionales** (`if`, `switch`) para tomar decisiones y **bucles** (`for`, `while`) para repetir tareas hasta cumplir una condición. Este paradigma es el más antiguo y subyace en lenguajes como C; incluso en Java, dentro de los métodos, el código suele ser imperativo (se actualizan variables, se ejecutan bucles, etc.). Es fácil de entender porque sigue el orden de ejecución literal, aunque puede requerir mucho código para tareas complejas.

- **Secuencialidad**: Se describe el algoritmo indicando explícitamente qué hacer en cada paso.
- **Cambio de estado**: Las instrucciones modifican valores de variables a medida que avanza la ejecución.
- **Estructuras de control**: Se emplean `if/else`, `switch`, `for`, `while`, `do/while` para controlar el flujo del programa.
- **Tradicional**: Paradigma base de la programación procedimental; no concentra datos y funciones en unidades, sino que describe los pasos para procesar datos de entrada a salida.





Programación orientada a eventos

En la **programación orientada a eventos** el flujo de ejecución del programa está dictado por *eventos* externos o internos. Un evento puede ser, por ejemplo, una acción del usuario (clic, pulsación de tecla), señales del sistema, o mensajes de red. En este paradigma, en lugar de seguir un orden fijo, el programa “escucha” eventos y ejecuta un manejador o *callback* asociado cuando ocurre cada evento. Al iniciar, el programa realiza inicializaciones y luego queda en espera; cuando llega un evento, se activa el código correspondiente. Este enfoque es típico en interfaces gráficas y aplicaciones web interactivas (donde el usuario define cuándo suceden las cosas), así como en sistemas de comunicación entre componentes.

- **Eventos y manejadores**: Se definen eventos (por ejemplo, clic en un botón, llegada de datos) y funciones que se ejecutan al producirse cada evento.
- **Flujo reactivo**: El programa no avanza linealmente; inicializa y luego espera eventos. Cuando ocurre uno esperado, salta al manejador correspondiente.
- **Uso común**: Fundamental en GUIs, juegos, aplicaciones web modernas, o arquitecturas “pub/sub” donde el usuario o el entorno disparan las acciones. El control del programa lo ejerce quien genera los eventos (por ejemplo, el usuario), no el programador directamente.