Inheritance

Inheritance allows one class (called the **subclass** or **child class**) to **inherit the properties and behaviors** of another class (called the **superclass** or **parent class**). In simpler terms, **inheritance enables code reuse** by letting you define common logic in a base class and then extend or specialize it in multiple derived classes.

Real-World Analogy

Think of a **User** system in a web application:

- The base `User` class holds common attributes like `username`, `email`, and methods like `login()` or `logout()`.
- Specialized roles like `Admin`, `Customer`, and `Vendor` inherit from `User` but add role-specific behavior.

This mirrors the real world: all admins, customers, and vendors are users—but with different privileges or responsibilities.

Why Inheritance Matters

Inheritance offers several compelling advantages in software design:

- **Code Reusability**: Common logic is written once in the parent class and inherited by all child classes, reducing duplication.
- **Logical Hierarchy**: Inheritance models real-world "is-a" relationships, such as `ElectricCar is a Car` or `Admin is a User`.
- **Ease of Maintenance**: Changes to shared behavior only need to be made in one place (the superclass) and all subclasses benefit automatically.
- **Improved Readability**: It encourages DRY (Don't Repeat Yourself) code and helps organize classes in a clean, understandable structure.

How Inheritance Works

When a class inherits from another:

- The subclass **inherits all non-private fields and methods** of the superclass.
- The subclass can **override** inherited methods to provide a different implementation.
- The subclass can also **extend** the superclass by adding new fields and methods.

This allows for both **reuse** and **customization**.

Code Example: Car Hierarchy

Let's model a simple **Car** system.

Java

```
class Car {
    protected String make;
    protected String model;

    public void startEngine() {
        System.out.println("Engine started");
    }

    public void stopEngine() {
        System.out.println("Engine stopped");
    }
}
```

Python

```
class Car:
    def __init__(self):
        self.make = None
        self.model = None

    def start_engine(self):
        print("Engine started")

    def stop_engine(self):
        print("Engine stopped")
```

Now you can create specialized types of cars:

Java

```
class ElectricCar extends Car {
    public void chargeBattery() {
        System.out.println("Battery charging");
    }
}
```

```
class GasCar extends Car {
    public void fillTank() {
        System.out.println("Filling gas tank");
    }
}
```

Python

```
class ElectricCar(Car):
    def charge_battery(self):
        print("Battery charging")

class GasCar(Car):
    def fill_tank(self):
        print("Filling gas tank")
```

In this example:

Both ElectricCar and GasCar inherit the make, model, startEngine(), and stopEngine() methods from the Car class.
Each subclass adds behavior specific to its type.
This structure mirrors the real-world relationship: an electric car is a car, and so is a gas car.
When to Use Inheritance
Use inheritance when:

There is a clear "is-a" relationship
The parent class defines common behavior that should be shared
The child class does not violate the behavior expected from the parent
Avoid inheritance when:

The relationship is more of a "has-a" or "uses-a" (prefer composition)
You want to combine behaviors dynamically (use interfaces or strategy pattern)
You need flexibility or runtime switching between behaviors

Use Inheritance with Caution

While inheritance is powerful, overusing it or applying it incorrectly can lead to tight coupling and fragile hierarchies.

Common Pitfalls:

Misusing inheritance for code reuse: Inheriting from a class just to reuse methods, without a true “is-a” relationship, leads to poor design.
Deep inheritance chains: Long hierarchies become hard to understand, modify, or test.
Tight coupling: Subclasses are tightly coupled to the internal implementation of the superclass, making changes risky.
This is why many modern OOP designs favor composition over inheritance .

Inheritance vs. Composition

Prefer composition over inheritance when:

You need flexibility and runtime behavior changes
The relationship is "has-a" rather than "is-a"
You want to avoid coupling to a class hierarchy
Example:
Instead of this:

Java

```
class Printer extends Logger {
    // bad inheritance just to reuse log()
}
```

Do this:

Java

```
class Printer {
    private Logger logger;
```

```
    public Printer(Logger logger) {
        this.logger = logger;
    }

    public void print(String message) {
        logger.log("Printing: " + message);
    }
}
```

Python

```
class Printer:
    def __init__(self, logger):
        self.logger = logger

    def print_(self, message):
        self.logger.log(f"Printing: {message}")
```

Composition gives more control, better testability, and looser coupling.