scaler.com

# Composition in Java with Examples - Scaler Topics

*Sushant Gaurav*

8–10 minutes

---

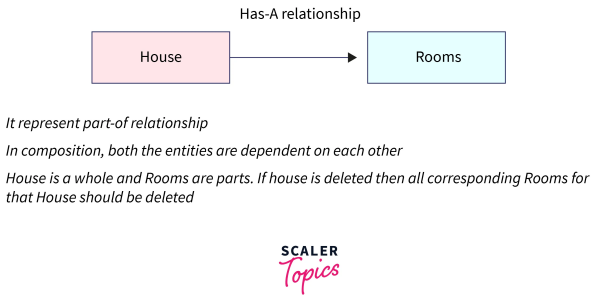
## Overview

Association, aggregation, and composition are three significant concepts to learn in Java. By constructing classes using composition, developers may build complex systems while avoiding the drawbacks of deep class hierarchies. Thus, mastering composition enables Java programmers to create efficient and adaptive solutions, improving software quality and design. In this article, we will focus only on composition. To learn the difference among the above three, go to Association-Aggregation-Composition In Java

## What is Composition in Java?

Composition is when a class contains instances of other classes to use their abilities. It is constructing large classes by merging smaller, reusable components. This helps in reusing code and organizing it better. Unlike inheritance, where a class inherits traits from a parent class, composition focuses on how things work together.

**Example**:




Imagine making a Car class. Instead of inheriting from the Engine, Wheels, and Transmission classes, you can put these instances inside the Car class. This has benefits: it keeps things simple, lets you easily swap or add parts, and makes finding and fixing issues

easier.

In Java programming, there are two concepts: association and composition. The association is a connection between two groups, like using something. Composition is more robust, where one class is made of another, and the inner class can't exist alone. Knowing these differences helps make strong, organized Java programs.

To learn about the association in Java, click Association Composition and Aggregation in Java.

**Benefits of Composition in Java**

Here are the key benefits of using composition in Java:

- **Modularity**:

  Composition encourages modularity by allowing you to construct software as a collection of independent and interchangeable components. Because each element tackles a distinct task, this improves code readability and reusability.

- **Code Reusability**:

  You may reuse existing classes by merging them into new ones via composition. This minimizes duplication and development time while also encouraging more efficient coding.

- **Flexibility**:

  Composition allows for dynamic changes to class behaviour by changing the composition of objects. This allows for adaptability to new requirements without affecting the present software.

- **Maintenance**:

  When using composition, changes or updates to one class may or may not affect other classes. This separation makes debugging, maintenance, and troubleshooting easier.

- **Loose Coupling**:

  Composition encourages loose coupling among classes, which reduces interdependence. This results in simpler code to comprehend, alter, and test.

- **Abstraction**:

  Composed classes can describe complex systems at a higher degree of abstraction. This abstract perspective assists developers in immediately comprehending the system's design.

- **Scalability**:

  As software needs to expand, composition enables inserting new components without affecting the existing system, assuring scalability.

- **Design Clarity**:

  Composition promotes a more natural and straightforward design approach, mirroring real-world item connections.

## Real Life Example

Consider the case of constructing an automobile out of composition. The engine, wheels, gearbox, and other components make up an automobile. Rather than developing a single class to represent the entire automobile, we will construct distinct classes for each component and combine them to produce a complete car object.

**Engine Class**:

Begin by creating an Engine class that contains engine behaviour. This class might have methods for starting, stopping, and revving the engine.

**Wheels Class**:

Next, create a Wheels class to represent the car's wheels. This class may provide methods for rotating the wheels and calculating speed.

**Car Class (Using Composition)**:

Now comes the compositional magic. Make a Car class using instances of the Engine and Wheels classes as members. A Car object may now delegate engine-related duties to the Engine object and wheel-related responsibilities to the Wheel object via composition.

Composition results in a modular design where each component (engine and wheels) may be designed and tested individually. Any upgrades or enhancements to the engine or wheels will not demand a change in the overall automobile design.

The potential of this object-oriented paradigm in Java is demonstrated by the example of making an automobile through composition. Composition increases code reusability, maintainability, and scalability by allowing you to build complicated systems from smaller, well-defined classes. By separating and composing responsibilities across separate classes, you make a more flexible and understandable codebase, a characteristic of good software development.

## Implementation of Composition in Java

Composition is a basic object-oriented programming technique that allows big classes to be created by merging smaller, more focused ones. Composition in Java is accomplished by generating objects of one class within another, allowing for the creation of complicated interactions between objects.

**Example**

Let's look at a real-world example: a Car class that is made up of Engine and Tyre classes. This composition relationship emphasizes the concept that a car has an engine and has tires, emphasizing the importance of the link.

```
class Engine {
    // Engine implementation
}

class Tire {
    // Tire implementation
}

class Car {
    private Engine car engine;
    private Tire[] tires;

    public Car() {
        carEngine = new Engine();
        tires = new Tire[4];
        for (int i = 0; i < 4; i++) {
            tires[i] = new Tire();
        }
    }
    // Other Car methods and functionalities
}
```

**Explanation**

In this example, the Car class contains an Engine object and an array of Tyre objects. When you create a new Car instance, it immediately produces an instance of Engine and four cases of Tyre. This structure ensures the Car class has all its necessary components.

Composition's appeal rests in its ability to generate modular and manageable code. Changes to the Engine or Tyre classes do not affect the Car class as long as the external interface stays constant. This improves code flexibility and reuse.

Understanding the result of this example does not entail direct output on the console; instead, it demonstrates how composition may efficiently organize and structure code. Using composition, Java programmers may design complex systems while keeping a clear object hierarchy.

## FAQs

Q:**What is a composition in Java?**

A: In Java, composition involves a class containing objects from

other classes, creating links for better abstraction and reusability.

Q:**What distinguishes composition from inheritance?**

A: Composition and inheritance are design concepts in object-oriented programming. Inheritance lets a class inherit traits from a parent class, while composition involves using instances of other classes within a class for a has-a relationship.

Q: **What are the advantages of composing?**

A: In Java programming, composition has various advantages. Because classes may be written and tested individually, it promotes code reuse and modular design. It also provides greater flexibility and dynamic behaviour since objects may be changed or updated at runtime, improving maintenance and scalability.

Q:**How can I do composition in Java?**

A: Composition involves making instances of other classes inside your class, creating a part-of relationship that separates behaviour and reduces class dependencies. Employ interfaces or abstract classes for effective composition to maintain flexibility and adhere to principles.

## Conclusion

- Composition enables developers to create modular and reusable code by constructing items from other elements. This method improves the codebase's maintainability and enables more efficient updates and additions.
- Composition increases code reuse by creating a has-a link between classes. This implies that a class may use the features of other classes without inheriting their whole behaviour, resulting in a more flexible and adaptable code structure.
- The strong connection between classes that might arise with inheritance is reduced via composition. This loose coupling is essential for lowering dependencies and making the codebase more adaptable to changes.
- Classes may be built to replicate real-world items through composition more closely. Each class focuses on a different part of the functionality, resulting in cleaner and more understandable code.
- The dynamic character of the composition adds to its charm. At runtime, objects may be assembled and recomposed, allowing for dynamic changes and fine-tuning of an application's behaviour.