# Composition and Decoupling in OOP

## 1. Composition

Composition is an object-oriented programming (OOP) concept where one class contains an object of another class to reuse code.   
Instead of inheriting from another class, you embed an instance of it.  
  
Use composition when:  
- You only need specific features from another class.  
- You want to keep classes independent and reusable.  
  
Example:  
class Engine:  
 def start(self):  
 print("Engine started")  
  
class Car:  
 def \_\_init\_\_(self):  
 self.engine = Engine() # Composition  
  
 def start(self):  
 self.engine.start()  
  
car = Car()  
car.start()  
  
Here, Car contains an Engine and uses its functionality.

## 2. Decoupling

Decoupling means reducing the direct dependency between two parts of a system so that changing one part doesn’t break or affect the other.  
  
Without Decoupling (Tightly Coupled):  
class Car:  
 def start(self):  
 engine = Engine() # Hardcoded  
 engine.start()  
  
With Decoupling (Using Composition):  
class Car:  
 def \_\_init\_\_(self, engine):  
 self.engine = engine  
  
 def start(self):  
 self.engine.start()  
  
car1 = Car(PetrolEngine())  
car2 = Car(ElectricEngine())  
  
Benefits of Decoupling:  
- Easier to test using mock objects  
- More flexible and maintainable code  
- Promotes high reusability  
  
In real-world backend systems, decoupling is done using dependency injection, where services like logging, database, or authentication are passed into other classes to reduce tight dependency.

**3**So, use composition when u need only one specific feature from another class, but don’t u think if we do engine() in \_\_init\_\_(self) we can still access all other classes too?

**✅ Yes, with composition:**

When you do:

python

CopyEdit

self.engine = Engine()

You **do get access to all public methods and attributes** of the Engine class.

So yes — self.engine.start() is available, and if Engine had stop() or status() methods, you could call those too:

python

CopyEdit

self.engine.stop()

self.engine.status()

**🔍 So what's the *real* difference then?**

Composition is **not about limiting access**, but about **how the relationship is structured** and **how much control you want**.

| **Concept** | **Composition** | **Inheritance** |
| --- | --- | --- |
| Design Philosophy | One object *has* another | One object *is* another |
| Access | Can access all public methods of included class | Inherits all public/protected members |
| Flexibility | You can **replace** or **change** components | Harder to modify inherited behaviors |
| Use case | Reuse functionality, plug-and-play | Extend functionality, polymorphism |

**✅ Why still use composition then?**

Even if it gives full access, **composition gives you control**:

* You decide **what to expose**. Maybe you only call engine.start() and never expose engine.status().
* You can **replace Engine with another implementation** (like ElectricEngine) without changing Car.

Example:

python

CopyEdit

class ElectricEngine:

def start(self):

print("Electric engine humming...")

class Car:

def \_\_init\_\_(self, engine):

self.engine = engine

def drive(self):

self.engine.start()

print("Car is moving")

So now:

python

CopyEdit

engine = ElectricEngine()

my\_car = Car(engine)

my\_car.drive()

No change in Car logic, just a different engine passed in.

**💡 TL;DR:**

* ✅ Yes, self.engine = Engine() gives full access.
* 🧠 But composition is chosen for **flexibility**, **decoupling**, and **clear design**, not just for limiting access.
* 🧩 You can plug-and-play different behaviours without inheritance hell.

**4🧠 Summary for Decoupling**

| **Tight Coupling** | **|Loose Coupling (Decoupling)** |
| --- | --- |
| Hard to test | |Easy to test with mock objects |
| Rigid — hard to modify later | |Flexible, easier to extend/maintain |
| Reuse is low | |Reuse is high |