Dependency

What happens when a class needs to use another class for a brief moment to get a job done, without needing to hold onto it forever?

This is the world of Dependency.

Unlike association, aggregation, or composition, dependency is not structural. It does not imply a long-term relationship or shared lifecycle. Instead, it reflects a one-time interaction, often through method parameters, local variables, or return types.

If class A uses class B just to perform a task—not to store or own—that's a dependency.

1. What is Dependency?
A Dependency exists when one class relies on another to fulfill a responsibility, but does so without retaining a permanent reference to it.

This typically happens when:

A class accepts another class as a method parameter.
A class instantiates or uses another class inside a method.
A class returns an object of another class from a method.
Key Characteristics of Dependency
Short-lived: The relationship exists only during method execution.
No ownership: The dependent class does not store the other as a field.
"Uses-a" relationship: The class uses another to accomplish a task, but does not retain it.
Real-World Analogy
Imagine a Chef preparing a meal.

The chef picks up a Knife to chop vegetables.
Once the chopping is done, the knife is put away or reused elsewhere.
The chef doesn't necessarily own the knife or keep it stored long-term.
This represents a dependency. The chef depends on the knife only during the cooking process.

2. Code Example
Let's model a simple Printer that depends on a Document to print.

Document Class

Python

```
class Document:
    def __init__(self, content):
        self.content = content
```

```
    def get_content(self):
        return self.content
```

Java

```
class Document {
    private String content;

    public Document(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

Printer Class

Python

```
class Printer:
    def print(self, document):  # dependency: Document
        print("Printing:", document.get_content())
```

Java

```
class Printer {
    public void print(Document document) { // dependency: Document
        System.out.println("Printing: " + document.getContent());
    }
}
```

Here:

Printer depends on Document to complete its work.
But Printer does not store or own the Document—it simply uses it during method execution.
3. UML Representation
In UML class diagrams, dependency is shown using a dashed arrow pointing from the dependent class to the class it depends on.

Printer ⟶ Document

This indicates that Printer temporarily uses Document, but does not own or associate with it in a structural sense.

4. Dependency Injection
Dependency Injection is a technique where you provide dependencies from outside, rather than letting the class create or control them internally.

This allows:

Better testability (you can inject mocks)
Greater modularity (swap implementations)
Loose coupling
Example:

Python

```
class NotificationService:
    def __init__(self, sender):  # sender is an interface-like dependency
        self.sender = sender  # Injected dependency

    def notify_user(self, message):
        self.sender.send(message)  # Uses sender temporarily
```

Java

```
interface Sender {
    void send(String message);
}

public class NotificationService {
    private final Sender sender; // Interface

    public NotificationService(Sender sender) {
        this.sender = sender; // Injected dependency
    }

    public void notifyUser(String message) {
        sender.send(message); // Uses sender temporarily
    }
}
```

The dependency (Sender) is provided externally, not created internally.

NotificationService does not care how messages are sent—it just depends on a Sender interface.
This promotes loose coupling, testability, and open/closed design.