Dependency Injection (DI) – Beginner Friendly Notes with Simple Examples

This note explains **Dependency Injection (DI)** in the simplest possible way using **real-world examples + crystal-clear Java code**.

⭐1. What is Dependency Injection?

Dependency Injection (DI) simply means:

> **A class should NOT create the objects it depends on. Someone else should give those objects to it.**

🦡Without DI (bad)

A class creates the object itself.

```java
class Student {
    Laptop laptop = new Laptop(); // Student controls the object creation

    void study() {
        laptop.compile();
    }
}
```

Problems: - ❌Hard to replace Laptop with a new type (e.g., SuperLaptop) - ❌Hard to test (cannot use mock object) - ❌Class becomes tightly coupled

---

🦝With DI (good)

```java
class Student {
    private Laptop laptop;

    public Student(Laptop laptop) {  // Laptop injected
        this.laptop = laptop;
    }

    void study() {
        laptop.compile();
    }
}
```

Benefits: - ✔️Loose coupling - ✔️Easy to change Laptop → GamingLaptop - ✔️Easy for testing - ✔️ Clean architecture

---

⭐2. Why is DI Used?

DI helps you write: - Cleaner code - Maintainable code - Replaceable components - Testable code

Everything becomes plug-and-play.

Example:

```
Laptop lap = new Laptop();
Student s = new Student(lap); // Injecting dependency
```

⭐3. Types of Dependency Injection

There are **3 types** of DI. We'll explain them with simple examples.

🦜3.1 Constructor Injection (BEST)

You pass the dependency through the constructor.

```
class Student {
    private Laptop laptop;

    public Student(Laptop laptop) {  // Inject here
        this.laptop = laptop;
    }

    void study() {
        laptop.compile();
    }
}
```

Usage:

```
Laptop lap = new Laptop();
Student s = new Student(lap);
```

🦝 Advantages

- Guarantees dependency is always provided
- Makes class **immutable**
- Easy to test
- Widely recommended

🦜 3.2 Setter Injection

Dependency is provided through a setter method.

```
class Student {
    private Laptop laptop;

    public void setLaptop(Laptop laptop) {  // Inject here
        this.laptop = laptop;
    }

    void study() {
        laptop.compile();
    }
}
```

Usage:

```
Student s = new Student();
s.setLaptop(new Laptop());
```

🦝 Useful when

- Dependency is optional
- You want to change dependency later

⚠ 3.3 Field Injection (Not recommended for pure Java)

```
class Student {
    @Autowired
    Laptop laptop; // Inject directly
}
```

Problems: - Not testable - No control - Exists only in frameworks like Spring

---

⭐4. What is a Dependency?

A **dependency** is simply an object your class needs.

Example: Student needs Laptop.

```
class Laptop {
    void compile() {
        System.out.println("Compiling...");
    }
}
```

```
class Student {
    Laptop laptop; // dependency
}
```

⭐5. DI + Polymorphism (Very Important)

If a class expects a parent type, you can inject ANY child type.

Example:

```
class Car {
    void start() { System.out.println("Car started"); }
}

class ElectricCar extends Car {
    void start() { System.out.println("Electric Car started silently"); }
}

class Driver {
    private Car car;

    public Driver(Car car) {  // Accepts parent type
        this.car = car;
    }

    void drive() {
        car.start();
    }
}
```

Usage:

```
Driver d1 = new Driver(new Car());
Driver d2 = new Driver(new ElectricCar());
```

✔️DI supports **upcasting** (child → parent) ✔️Allows switching implementations easily

---

⭐6. Why DI Is Better Than Creating Objects Manually

🦡Manual creation

```
Driver d = new Driver(new Car());
```

You decide the dependency → tight coupling.

🦝DI approach

```
public Driver(Car car) {}
```

Dependency supplied from outside → flexible.

⭐7. DI Makes Testing Super Easy

Using constructor DI:

```
Laptop mockLaptop = Mockito.mock(Laptop.class);
Student s = new Student(mockLaptop);
```

Now you can test Student *without creating a real Laptop*.

⭐8. Multiple Constructors Issue

If your class has **multiple constructors**, DI frameworks cannot decide which one to use.

```
class Student {
    Student() {}
    Student(Laptop laptop) {}
}
```

Ambiguity → You must specify which constructor to use.

In frameworks (like Spring):

```
@Autowired
public Student(Laptop laptop) {}
```

⭐9. Upcasting & DI

One of the biggest powers of DI is **upcasting**.

You write:

```
class Driver {
    Car car;
    Driver(Car car) { this.car = car; }
}
```

Now you can pass:

```
new Driver(new ElectricCar());
```

This allows DI containers to plug different implementations.

⭐10. DI Scopes (Simple Explanation)

DI frameworks (Example: Spring) give object scopes: - **Singleton** → One object shared everywhere - **Prototype** → New object each time - **Request/Session** → Web-specific scopes

Pure Java DI only creates new objects when YOU want.

⭐11. Circular Dependency Problem (Beginner-friendly)

Example:

```java
class A { B b; }
class B { A a; }
```

Both depend on each other → DI can get confused.

✔️Setter injection solves it. ✔️Constructor injection cannot.

⭐12. Summary Table To Remember

Concept	Meaning
Dependency	An object another class needs
DI	Giving dependency from outside
Constructor DI	Best, safe, testable
Setter DI	Optional
Field DI	Avoid in pure Java
Upcasting	Child object used as parent
Multiple constructors	Must choose one
DI benefits	Loose coupling, testability

🐻‍❄️Final Beginner-Friendly Example (Full Flow)

```java
// Dependency (what Student needs)
class Laptop {
    void compile() {
        System.out.println("Compiling...");
    }
}

// Student depends on Laptop
class Student {
    private Laptop laptop;

    // Constructor Injection
    public Student(Laptop laptop) {
        this.laptop = laptop;
    }

    void study() {
        System.out.println("Studying...");
        laptop.compile();
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Laptop lap = new Laptop();      // Create dependency
        Student s = new Student(lap);   // Inject dependency

        s.study();
    }
}
```

🦝**Clear**

🦝**Beginner-friendly**

🦝**Shows object creation**

🦝**Shows DI in action**

If you'd like, I can also prepare: - A **diagram version** - A **PDF version** - A **Q&A version for interviews**