Dependency Injection

إيه هو Dependency Injection؟

لو اشتغلت على برمجة أوبجكت أوريانتيد قبل كده، أكيد حسيت إن الموضوع بيبقى معقد لما الكلاسات بتعتمد على بعضها بشكل شديد. بس الحقيقة إن Dependency Injection (DI) هو سر فصل الاعتمادات دي وفهم التصميم صح.

يعني إيه Dependency Injection؟

الـ DI ببساطة هي طريقة بنفصل بيها إنشاء الـ dependencies (الكائنات اللي الكلاس بيعتمد عليها) عن الكود الرئيسي. بدل ما الكلاس ينشئ الـ dependency بنفسه، بنحقنها من خارج، وده بيجي تحت أنماط التصميم الـ Creational اللي بتركز على إنشاء الكائنات بطريقة مرنة.

في سياق الـ Inversion of Control، الـ DI بتفصل بناء الكائن وتوفير الـ dependencies عن اللوجيك الرئيسي، عشان نقلل الـ tight coupling ونحسن الاختبار.

ليه بنستخدم DI؟

علشان التصميم بيشتغل بمبدأ Loose Coupling، فبنوزع الاعتمادات على واجهات (Interfaces) بدل تنفيذات محددة، وده بيخلينا نغير المكونات بسهولة من غير ما نعدل في أماكن كتير. كمان، الـ DI جزء من Creational Patterns اللي بتخفي منطق الإنشاء عن الكود.

1. إيه هي Creational Design Patterns؟

الـ Creational Patterns هي العملية اللي بيتم فيها إنشاء الكائنات بطريقة تخفي اللوجيك عن الكلاينت، وده بيحقق استقلالية وإعادة استخدام. أمثلة شهيرة:

- **Singleton:** يضمن إن الكلاس له instance واحد بس، ويوفر نقطة وصول عامة.
- **Factory Method:** يحدد interface لإنشاء كائن، بس يسيب للـ subclasses تقرر النوع.
- **Abstract Factory:** يوفر interface لإنشاء عائلات من الكائنات المتعلقة بدون تحديد الكلاسات الفعلية.
- **Builder:** يفصل بناء كائن معقد عن تمثيله، عشان يقدر يبني تمثيلات مختلفة بنفس العملية.
- **Prototype:** ينشئ كائنات جديدة بنسخ كائن موجود (prototype).

طيب، هل DI نمط Creational؟ أيوة، لأنه بيسيطر على إنشاء الـ dependencies خارجيًا، بيحقنها في الكائنات، بيسمح بتكوين مرن بدون تغيير الكود، وبيبسط صيانة التطبيقات الكبيرة.

2. إزاي كانت الـ Object Creation قبل DI؟

قبل انتشار DI (في أيام البرمجة التقليدية زي في بعض الأنظمة القديمة)، كان الوضع مختلف تمامًا. خلينا نأخد أمثلة زي في C++ أو Java القديمة:

- الكلاس كان بينشئ dependencies بنفسه داخل الكونستراكتور أو الدوال.
- النتيجة كانت كود سريع بس مرتبط جدًا (tight coupling)، فلو غيرت dependency، كنت لازم تغير الكود في كل مكان.
- العيوب:
  - صعوبة الاختبار (مش هتقدر تستبدل dependencies بـ mocks).
  - إدارة الـ lifetimes كانت معقدة وبتعتمد على المبرمج.
  - مكتبات مختلفة لكل جزء، فكان صعب إنك توسع التطبيق.

3. إيه اللي تحسن مع Dependency Injection؟

الـ DI جاب ثورة في إنشاء الكائنات وإدارة الاعتمادات. أهم التحسينات:

- **Loose Coupling:** الكلاسات بتعتمد على abstractions او interfaces مش concrete implementations، فسهل تعديل أو استبدال المكونات بدون تأثير.
- **تحسين الاختبار (Improved Testability):** الـ dependencies بتتحقن، ففي الـ unit tests، نقدر نستبدلها بـ mocks أو stubs عشان نختبر الوحدات بشكل منفصل.
- **صيانة أفضل (Better Maintainability):** سيطرة مركزية على الإنشاء عشان إدارة التعقيد والتكوين.
- **مرونة (Adaptability):** تدعم تنفيذات متعددة وتغييرات ديناميكية في الـ runtime.

مقارنة بالطرق القديمة، الـ DI أسرع في الصيانة لأنه بيفصل اللوجيك، وبيخزن الـ configurations خارجيًا.

مثال توضيحي:

تخيل عندك كلاس Service يعتمد على Logger:

بدون DI: الـ Service بينشئ Logger بنفسه.

مع DI: نحقن الـ Logger من خارج.

مثال في C#:

```
public interface ILogger {
    void Log(string message);
}
```

```
public class ConsoleLogger : ILogger {
    public void Log(string message) {
        Console.WriteLine(message);
    }
}

public class MyService {
    private readonly ILogger _logger;

    public MyService(ILogger logger) {  // Injection هنا
        _logger = logger;
    }

    public void DoWork() {
        _logger.Log(";("العمل تم!"
{
{

استخدام //: var service = new MyService(new ConsoleLogger());
```

هنا، MyService مش بينشئ Logger، ده بيحقن، فلو عايز تغير Logger لـ FileLogger، تغير في الإنشاء بس.

طيب لو عايز تشوف كل الـ dependencies حتى لو مفيش، ده زي FULL JOIN في SQL، بس هنا بتستخدم DI Container زي في .NET لإدارة الكل.

🧠 طيب الناس بتتلخبط في إيه؟

- **نسيان استخدام Interfaces:** كتير بنكتب DI من غير abstractions، وده بيفقد الفائدة.
- **خلط بين أنواع الـ Injection:** ناس مش عارفة الفرق بين Constructor و Setter Injection، وده بيأثر على التصميم.
- **Circular Dependencies:** اعتمادات دائرية بتسبب أخطاء runtime.

🔸 **خلط بين DI و Inversion of Control:** ناس بتعتقد إن DI هو نفسه IoC، بس IoC مفهوم أوسع.

⚙️ Tips مهمة للتعامل مع DI:

✔️ دايمًا حدد الـ Interfaces بوضوح عشان loose coupling.

✔️ راجع الـ lifetimes (Singleton, Scoped ,Transient) في الـ Container.

✔️ اختبر الـ DI configuration في الـ program.cs عشان تكتشف الأخطاء بدري.