IOC -

در برنامه نویسی سنتی، هر کلاس معمولاً مسئول ایجاد و مدیریت وابستگی‌های خود است، که باعث افزایش couplingبین اجزا می‌شود IOC این روند را معکوس می‌کند، به این معنا که کنترل ایجاد و مدیریت وابستگی‌ها به یک مکانیزم خارجی واگذار می‌شود. این تغییر موجب افزایش قابلیت تست، نگهداری و توسعه نرم‌افزار به شیوه‌ای ماژولار و انعطاف‌پذیر می‌شود.

DIP -

ماژولهای سطح بالا (High-Level Modules) (مثال: منطق کسبوکار) نباید به ماژولهای سطح پایین (Low-Level Modules) (مثال: دسترسی به دیتابیس یا سرویسهای خارجی) وابسته باشند. هر دو باید به انتزاعها (Abstractions) وابسته باشند.

انتزاع ها نباید به جزئیات (Details) وابسته باشند، بلکه جزئیات باید به انتزاع ها وابسته باشند.

IOC Container -

یک فریم ورک است که ایجاد، مدیریت و تزریق وابستگی‌ها (Dependencies) بین کلاس‌ها را خودکار می‌کند. این ابزار برای پیاده‌سازی الگوی طراحی تزریق وابستگی استفاده می‌شود تا وابستگی بین اجزای برنامه را کاهش دهد و کد را قابل‌تست، قابل‌نگهداری و انعطاف‌پذیر کند.

Factory as IoC Container -

در برخی موارد، می‌توان از **الگوی فکتوری** برای شبیه‌سازی رفتار یک کانتینر IoC ساده استفاده کرد. این روش بیشتر در پروژه‌های کوچک یا مواردی که نمی‌خواهید از کتابخانه‌های پیچیده DI استفاده کنید، کاربرد دارد. در این حالت، فکتوری مسئولیت ایجاد اشیا و مدیریت وابستگی‌ها را برعهده می‌گیرد.

فکتوری تمام منطق ساخت اشیا و وابستگی‌های آن‌ها را متمرکز می‌کند.

استفاده از فکتوری به عنوان کانتینر IoC یک راه‌حل **ساده و سبک** برای مدیریت وابستگی‌ها است، اما در پروژه‌های بزرگ یا پیچیده، **کانتینرهای حرفه‌ای** مانندMicrosoft.Extensions.DependencyInjection یا Autofac گزینه بهتری هستند. این روش بیشتر شبیه **نیمه‌خودکارسازی**Dependency Injection است تا یک جایگزین کامل برای IoC Container

تزریق وابستگی (DI) یکی از **الگوهای طراحی نرم‌افزاری** است که بر پایه اصل **واژگونی کنترل** و **اصل وارونگی وابستگی** از اصول **SOLID** استوار است. این الگو با هدف **کاهش وابستگی (Coupling)** بین اجزای نرم‌افزار و افزایش **انسجام (Cohesion)** طراحی شده است.

DI اصل DIP را با تزریق وابستگی‌ها از طریق **اینترفیس‌ها** (به جای کلاس‌های مشخص) پیاده‌سازی می‌کند.

واژگونی کنترل:(IoC)
«کنترل ایجاد و مدیریت اشیا از کلاس‌ها به یک فریمورک یا کانتینر خارجی سپرده می‌شود».

DIنمونه‌ای از IoC است که در آن، وابستگی‌ها از خارج به کلاس تزریق می‌شوند.

تزریق(Injection)

فرآیند **انتقال کنترل** ایجاد و مدیریت وابستگی‌ها به یک موجودیت خارجی) مانند کانتینر.(IoC

این انتقال باعث می‌شود کلاس‌ها تنها روی **مصرف** وابستگی‌ها تمرکز کنند، نه ایجاد آن‌ها.

Service LifeTimes in .Net -

در.NET ، **Service Lifetimes** تعیین می‌کنند که یک نمونه (Instance) از سرویس چگونه و در چه بازه‌ای ایجاد و مدیریت شود. این مفهوم بخش کلیدی **Dependency Injection (DI)** است و سه حالت اصلی دارد **Transient** ، **Scoped** ، **Singleton** .انتخاب Lifetime مناسب برای جلوگیری از مشکلاتی مانند **نشت حافظه (Memory Leaks)** یا **رفتار ناخواسته** حیاتی است.

---

1. **Transient Lifetime**

- **ویژگی** :هر بار که سرویس درخواست می‌شود، یک **نمونه جدید** ایجاد می‌شود.
- **موارد استفاده**:
  - سرویس‌های **بدون حالت**.**(Stateless)**
  - سرویس‌های سبک که ایجاد آن‌ها هزینه کمی دارد.
  - مواردی که هر مصرف‌کننده باید نمونه مستقل خود را داشته باشد.

2. **Scoped Lifetime**

- **ویژگی** :یک نمونه **به ازای هر Scope** ایجاد می‌شود.
  - در برنامه‌های وب، هر **درخواست HTTP** یک Scope است.
  - در برنامه‌های غیروب) مثل(Console ، باید Scope را دستی ایجاد کرد.
- **موارد استفاده**:
  - سرویس‌هایی که در طول یک درخواست باید حالت (State) حفظ کنند.
  - اتصالات به دیتابیس مثل DbContext درEF Core

3. **Singleton Lifetime**

- **ویژگی** :یک نمونه **برای کل چرخه حیات برنامه** ایجاد می‌شود.
- **موارد استفاده**:
  - سرویس‌های **پرهزینه** مثل کش یا پیکربندی.
  - سرویس‌های **حالت‌دار (Stateful)** که نیاز به اشتراک‌گذاری دارند.
  - **توجه** Singleton :ها باید **Thread-Safe** باشند!

جمع‌بندی

- :**Transient**برای سرویس‌های سبک و بدون حالت.
- :**Scoped**برای سرویس‌های وابسته به یک واحد کاری (مثل درخواست وب).
- :**Singleton**برای سرویس‌های پرهزینه یا حالت‌دار در سطح برنامه.