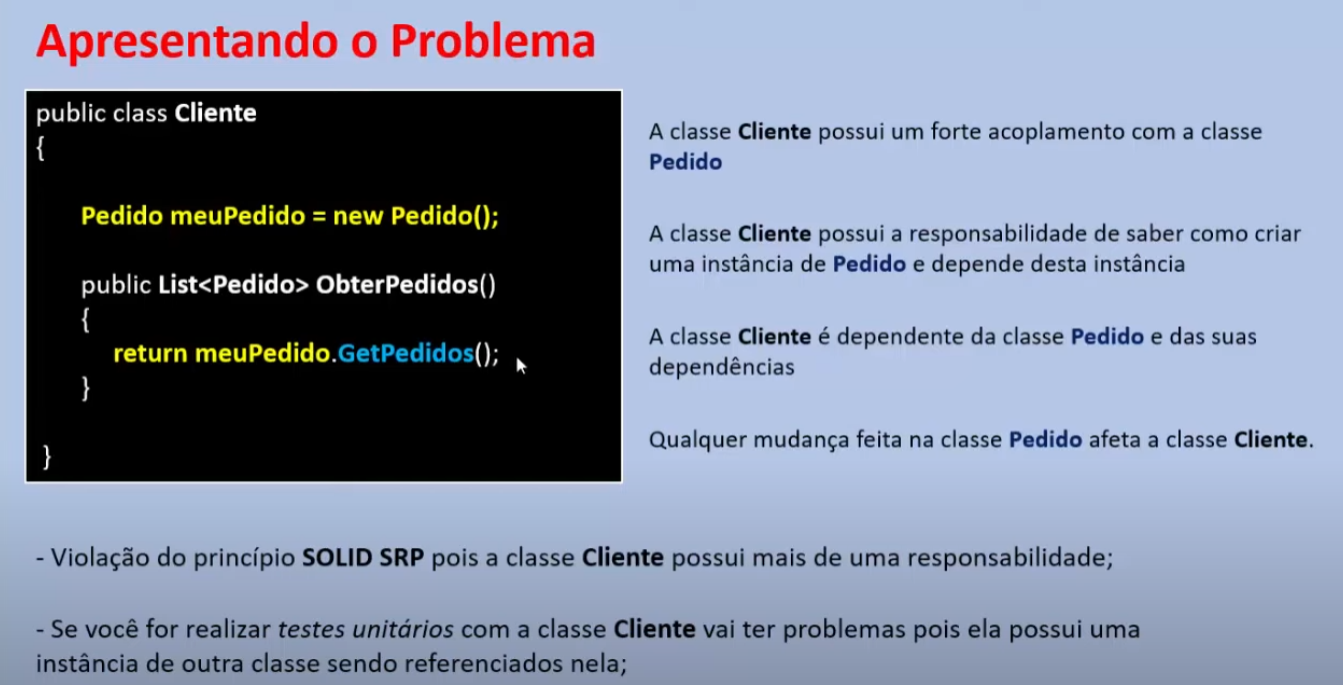
**Injeção de dependência**

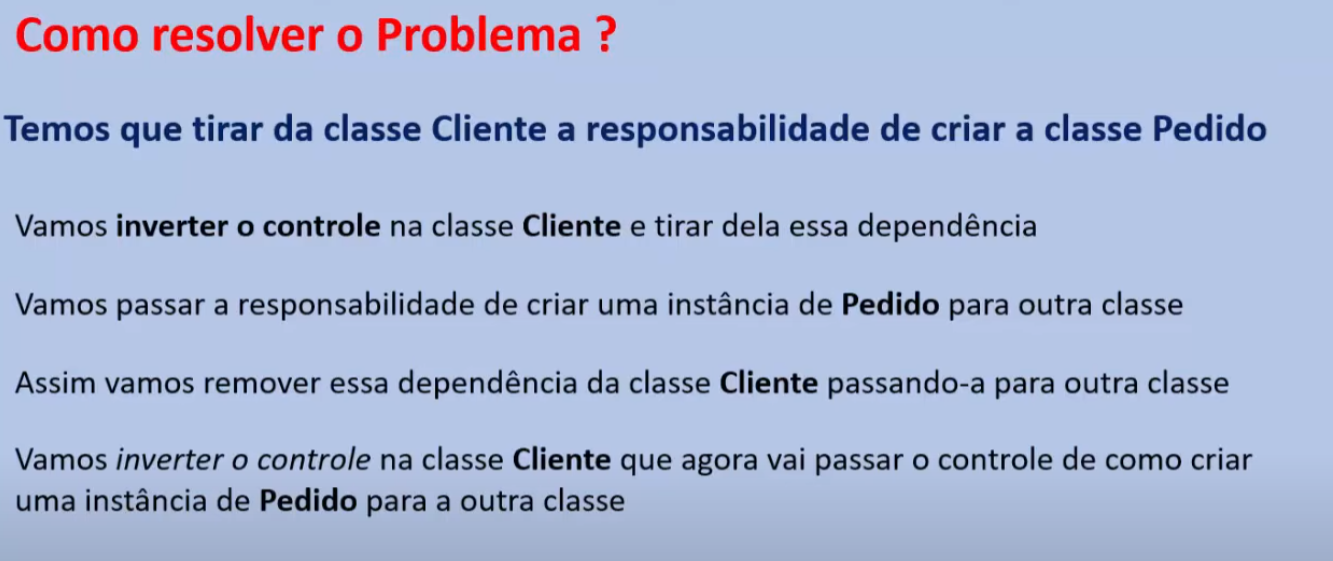
É uma técnica de programação usada para tornar uma classe independente de suas **dependências**.

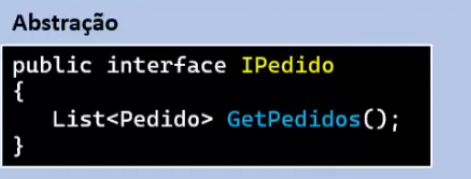
A injeção de dependência (DI) é um padrão usado para implementar a **inversão de Controle (IoC)** e assim reduzir o acoplamento entre os objetos.

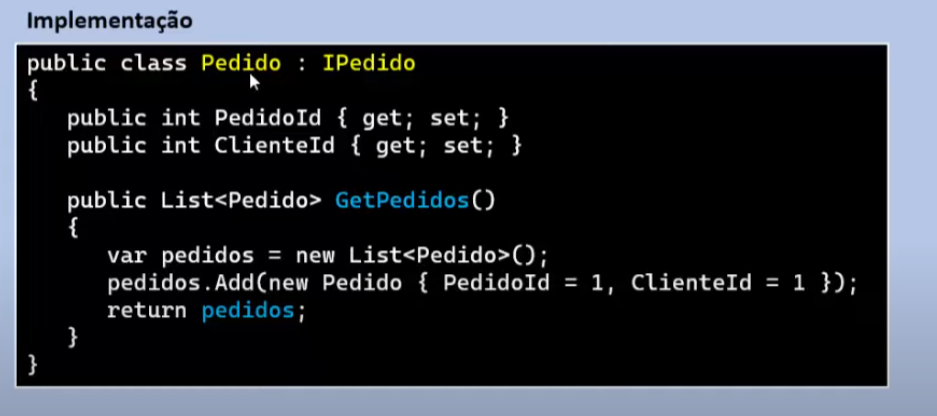
Ao aplicar a injeção de dependência fazemos com que um objeto forneça as **dependências** de outro objeto.

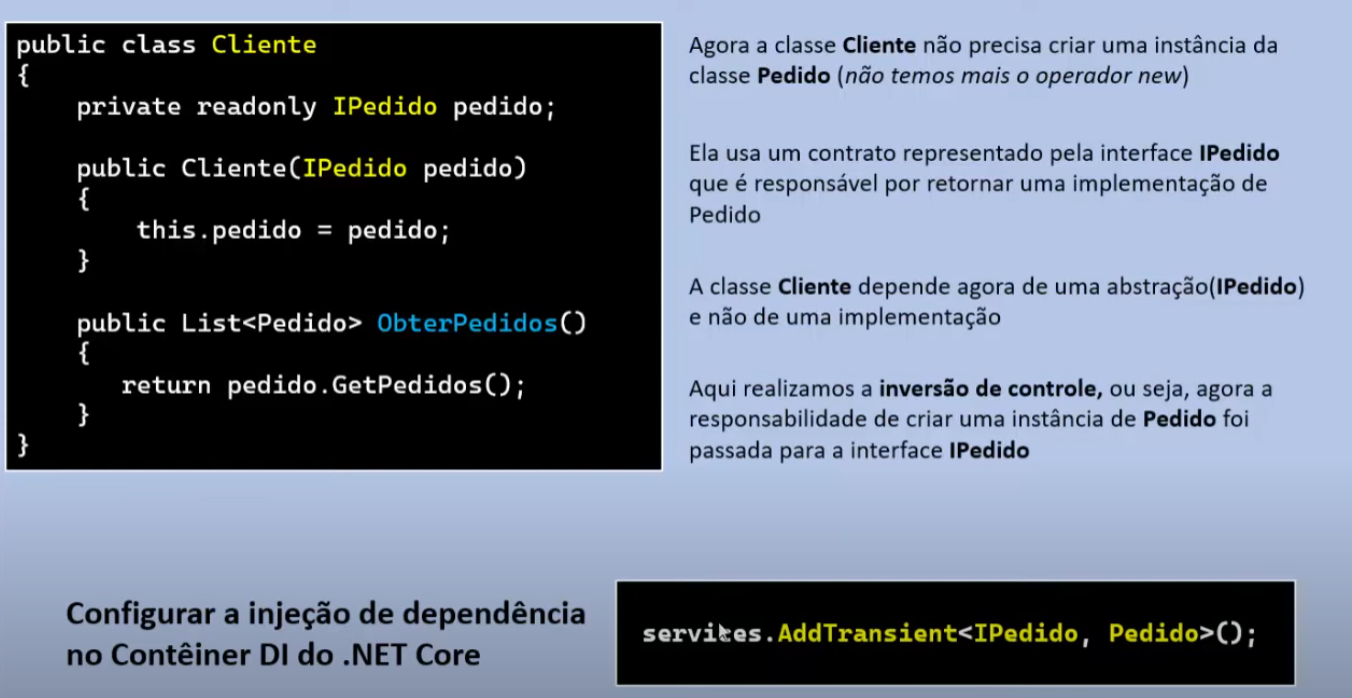
1. **Resolvendo o problema:**











1. **Container de Injeção de dependência:**

A plataforma .NET possui um Contêiner de Injeção de Dependência nativo definido em **Microsoft.Extensions.DependencyInjection**.

Mas existemm outros contêineres para a plataforma .NET que podemos usar como: **Simple Injector, Autofac, Ninject, Spring.NET, Unity, Castle Windsor, etc.**

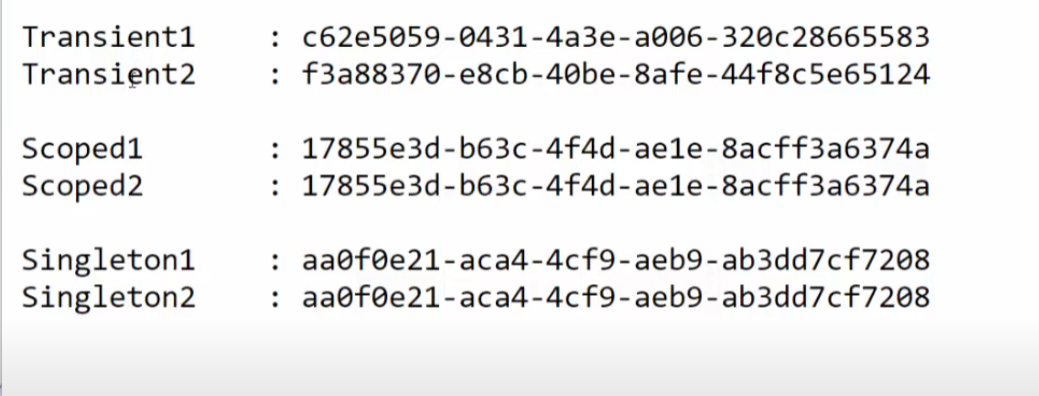
1. **Tempo de vida útil do serviço:**

Quando registramos serviços em um container, precisamos definir o tempo de vida que queremos usar para esta serviço.

O tempo de vida do serviço controla por quanto tempo um objeto vai existir após ter sido criado pelo contêiner.

O tempo de vida pode ser definido usando o método de extensão apropriado no **IserviceColletion** ao registrar o serviço.

* 1. **Tipos de tempo de vida:**
     1. **Transient**: São criados cada vez que são solicitados. Cada vez que você injetar o serviço em uma classe, será criada uma nova instância do serviço. É indicado para serviços leves e sem estado. São registrados usando o mmétodo **AddTransient**.
     2. **Scoped**: São criados em cada solicitação (uma vez por solicitação do cliente). É indicado para aplicações WEB. Se durante um request você usar a mesma injeção de dependência em muitos lugares, você vai usar a mesma instância de objetos, e ele fará referência à mesma alocação de memória. São registrados usando o método **AddScoped**.
     3. **Singleton**: São criados uma vez durante a vida útil do aplicativo que usa a mesma instância para todo o aplicativo. São registrados usando o método **AddSingleton.**
     4. **Exemplo:**



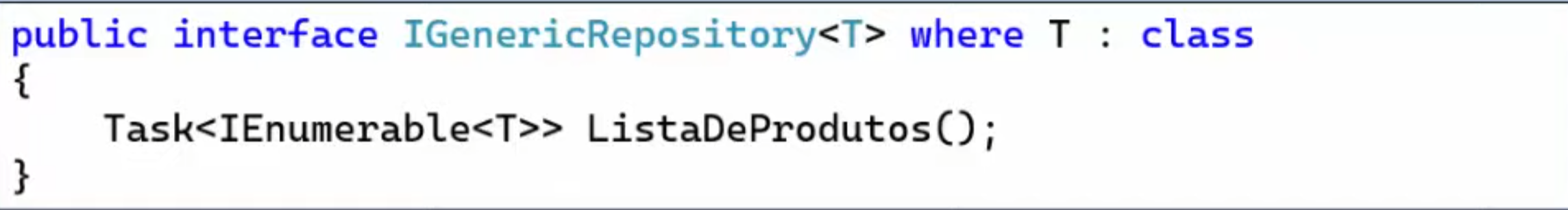
Nesse exemplo percebemos a implementação de um uuid em uma variavél, percebemos que o **Transient** possui valores diferentes em cada request, o **Scoped** possui um valor único para cada usuario porém toda vez que o usuario entrar tera um novo valor, o **Singleton** possui o mesma valor e nunca ira alterar enquanto a aplicação estiver executando.

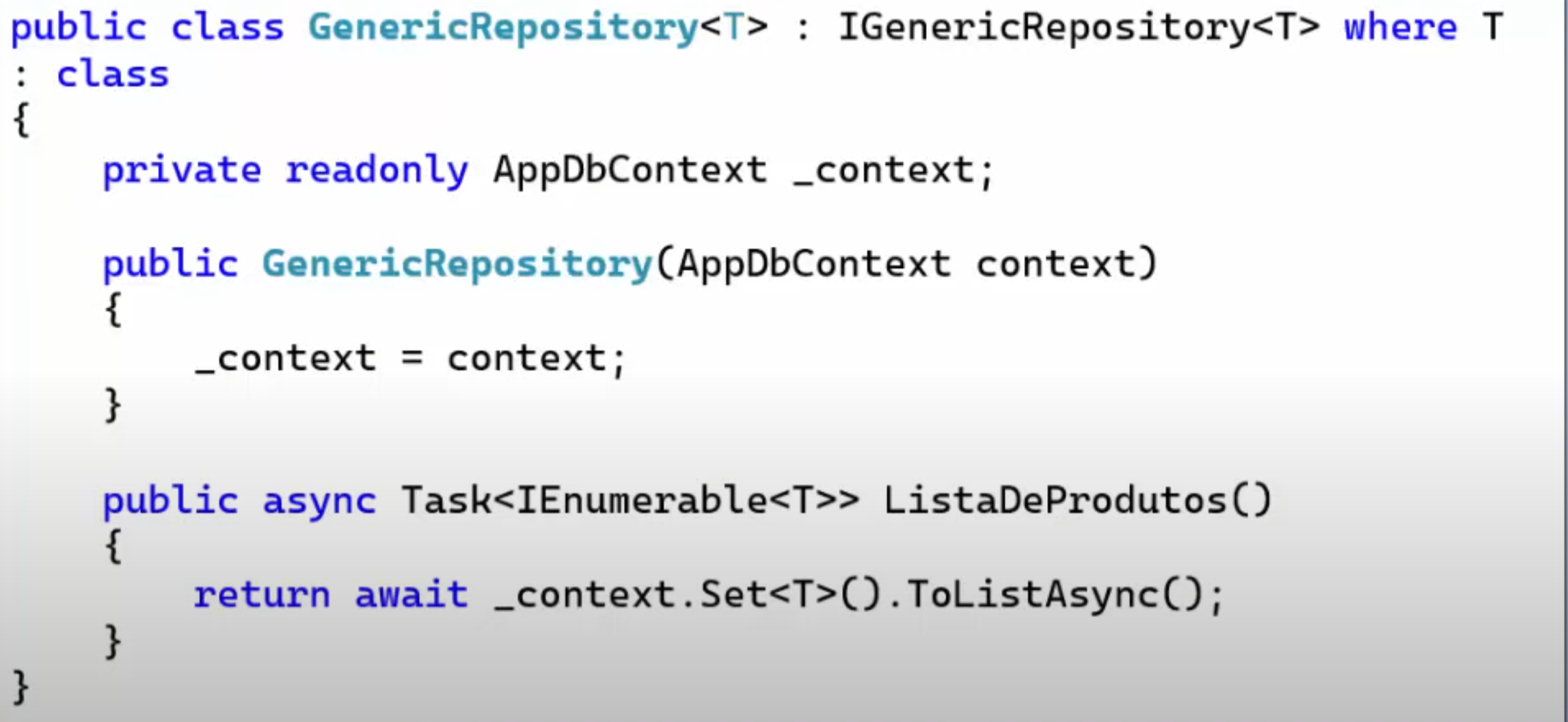
1. **Injeção de dependências em objetos genéricos:**

Como já vimos existe o padrão **Repository Especifico** que define um comportamento para apenas uma entidade, mas e se tivermos um **Repository Generico** ? Existe uma corrente que diz que isso é um anti padrão e que não deve ser aplicado, porem vamos entender como isso funciona:

* 1. **Repository generic:**

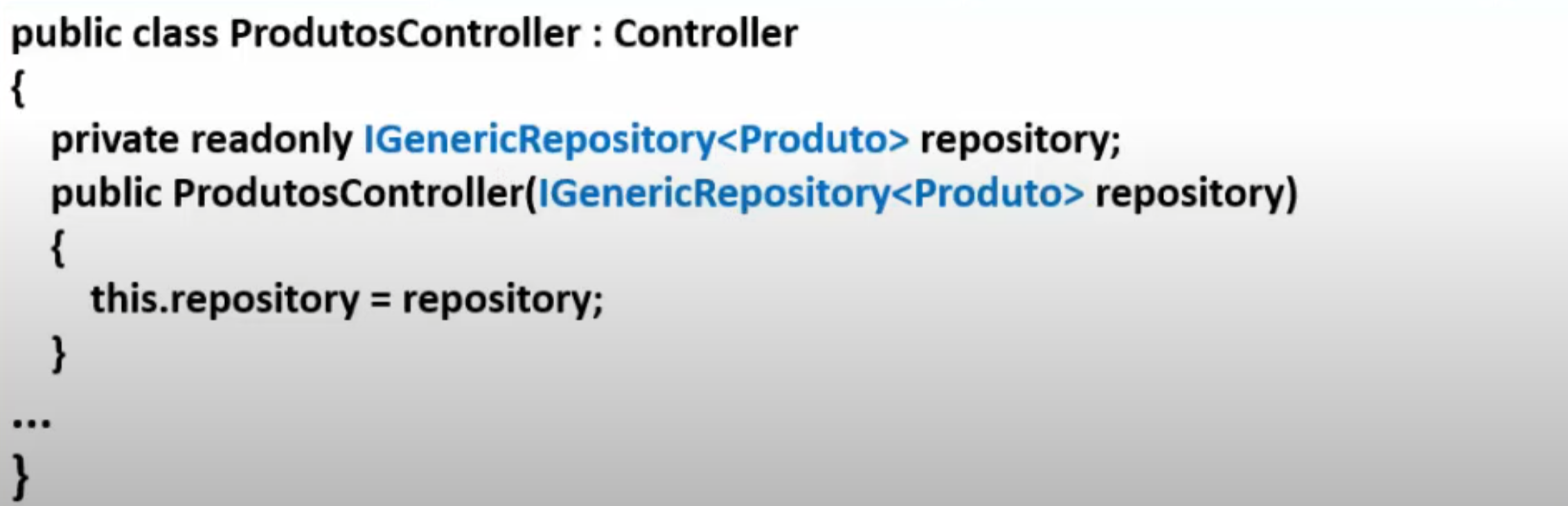
Define métodos genéricos para os tipos mais comuns das operações, como atualização, inclusão, busca e exclusão.













1. **Tipos de Injeção de dependência:**
   1. **Construtor:**

- É o processo que utiliza o construtor para passar as dependências de umma classe.

- As dependências são declaradas como parâmetros do construtor.

- Não podemos criar uma nova instância da classe sem passar uma variável do tipo exigido pelo construtor.

**- Quando usar:** Quando a classe tiver uma dependência sem a qual ela não vai funcionar corretamente.

**- Vantagens:**

- Cria um forte contrato de dependência.

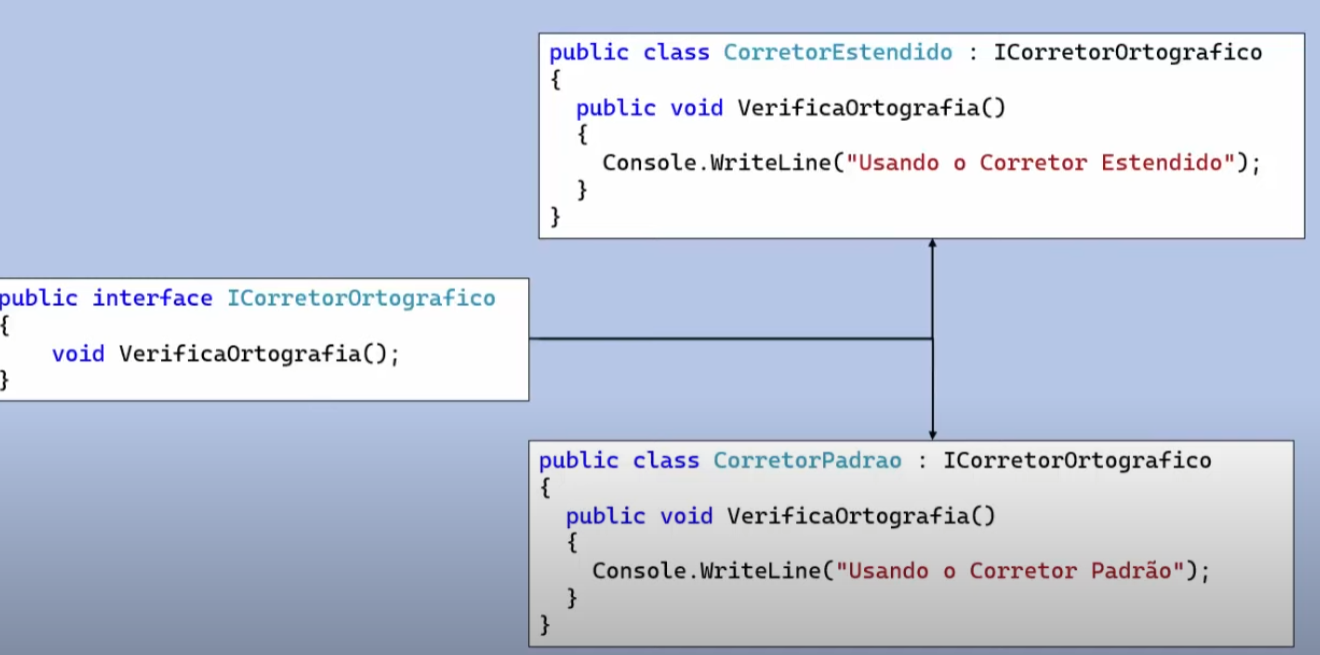
- Facilita os testes unitários à medida que as dependências são passadas pelo construtor.

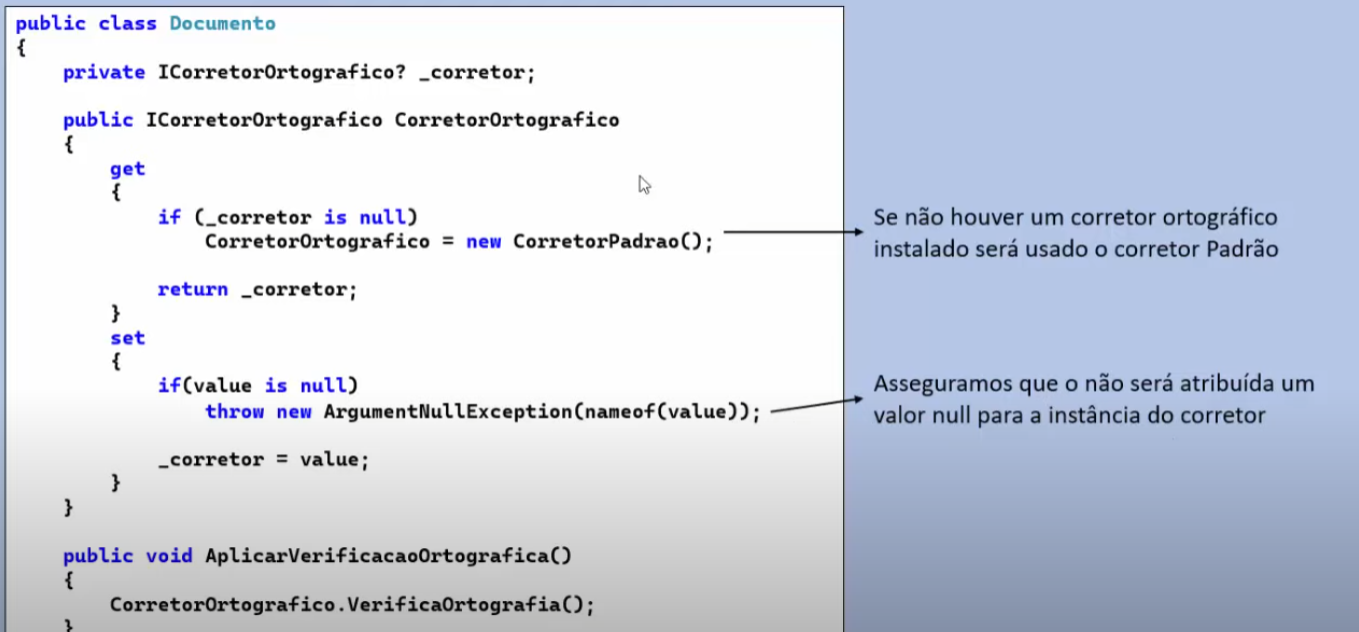
- Facilita a manutenção do codigo.

* 1. **Propriedade:**

- Fornecemos as dependências usadas por meio de uma propriedade pública da classe.

**- Quando usar:** Quando não podemos usar o construtor da classe, quando as dependências são opcionais para a classe funcionar ou quando as dependências podem ser alteradas depois da classe ser instanciada.

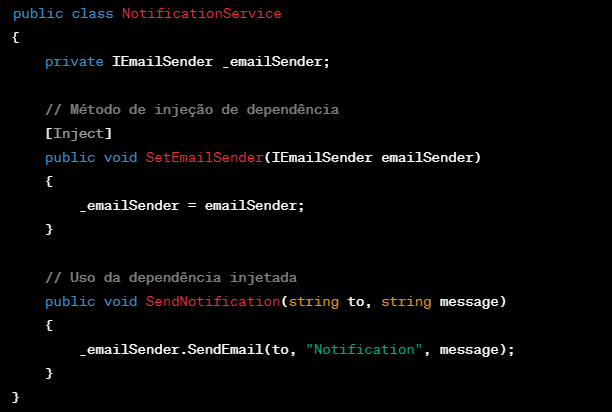
****

****

* 1. **Método:**

- Funciona basicamente através dos atributos **[Dependency]**  ou **[Inject]** que alguns frameworks trazem.

- oferece flexibilidade na definição do momento em que a dependência é injetada, pois você pode chamar o método de injeção de dependência em qualquer momento após a criação do objeto, e também permite a injeção de múltiplas instâncias de dependências diferentes em um único objeto, caso necessário.





1. **Service Locator:**

É um padrão de projeto que permite desacoplar clientes de serviços (descritos por uma interface pública) da classe concreta que implementa esses serviços.

Usa um registro central conhecido como **localizador de serviço** que, mediante solicitação, retorna as informações necessárias para executar uma determinada tarefa.

**Localizar** e **obter** por conta própria um serviço no contêiner DI nativo.

De forma simplificada o ServiceLocator é uma forma de armazenar os serviços sem depender do contexto geral da aplicação.



Nesse exemplo usamos o ServiceLocator para localizar os services disponiveis.

Para isso precisamos adicionar os services no ServiceLocator:



Utilizando esse padrão tambem podemos ter **multiplas implementações de uma interface**, que nesse caso é a IcustomLogger.

1. **Container DI nativo da plataforma .NET:**

A plataforma .NET possui o seu próprio container DI nativo que podemos usar e que funciona bem para muitas das necessidades de inicialização das aplicações.

No entanto ele apresenta algumas limitações como não possuir suporte de DI via Propriedade e existem outros contêineres DI com mais recursos que podemos usar.

1. **Autofac:**

O Autofac é o contêiner DI baseado na plataforma .NET mais usado do para ASP.NET e também é totalmente compatível com .NET Core.

Ele possui vários recursos e é mais flexível do que o contêiner DI nativo em casos específicos, tornando-se uma boa alternativa de uso em aplicações na plataforma .NET.

* 1. **Recursos especiais do Autofac:**

- Escopos de tempo de vida marcados e serviços de escopo para essas tags;

- Resolução de serviço com metadados associados;

- Definição de variantes com nome/chave de um serviço;

- Resolução de função de fábrica que você pode usar sempre que quiser;

- Instanciação Lazy.

* 1. **Aplicando o Autofac a um projeto:**

