**1.Рефлексия**

Рефлексия в C# — это мощный механизм, позволяющий получить информацию о типах данных, их свойствах, методах и событиях во время выполнения программы. С помощью рефлексии вы можете динамически создавать экземпляры объектов, вызывать методы и получать доступ к полям и свойствам, не зная о них на этапе компиляции.

### Основные аспекты рефлексии в C#

1. Получение информации о типах: Вы можете получить метаданные о типе, используя классы из пространства имен System.Reflection.
2. Динамическое создание объектов: С помощью рефлексии можно создавать экземпляры типов в рантайме.
3. Вызов методов: Вы можете вызывать методы объектов, не имея их имен на этапе компиляции.
4. Доступ к полям и свойствам: Позволяет изменять и получать значения полей и свойств объектов.

### Применение рефлексии

* Инструменты и библиотеки: Рефлексия часто используется в библиотеках, таких как Entity Framework, для работы с моделями данных.
* Сериализация и десериализация: Рефлексия может использоваться для создания механизмов сериализации (например, JSON или XML).
* Тестирование: В юнит-тестах рефлексия может использоваться для проверки приватных членов классов.

### Примечания

* Производительность: Рефлексия может быть медленнее, чем прямой код, из-за необходимости поиска метаданных и динамического вызова.
* Безопасность: Использование рефлексии может привести к нарушению инкапсуляции и безопасности, поэтому следует быть осторожным с доступом к приватным членам.

Существует несколько способов получения типа (или метаданных о типе) с использованием рефлексии:

### 1. Получение типа через имя класса

Вы можете получить тип, используя полное имя класса с помощью метода Type.GetType().

### 2. Получение типа с помощью оператора typeof

Оператор typeof является самым простым и распространенным способом получения типа. Он используется для получения статического типа.

### 3. Получение типа из экземпляра объекта

Вы можете получить тип объекта в рантайме, используя свойство GetType().

### 4. Получение типа через массивы

Если вы работаете с массивами, вы можете получить тип элементов массива с помощью свойства GetType().

### 5. Получение типа через Assembly

Вы можете получить типы из сборки (Assembly) с помощью методов GetTypes() или GetType().

### 6. Получение типа через AppDomain

Вы можете получить типы, загруженные в текущий домен приложения, с помощью метода AppDomain.CurrentDomain.GetAssemblies().

**2. Dependency injection(DI)**

Dependency Injection (DI) — это паттерн проектирования, который используется для управления зависимостями в приложениях. Он позволяет инвертировать управление (Inversion of Control, IoC) и улучшает тестируемость, модульность и гибкость кода. DI позволяет классам получать свои зависимости извне, а не создавать их самостоятельно, что способствует лучшей организованности кода и облегчает его поддержку.

### Основные принципы Dependency Injection

1. Инверсия управления: В классах, использующих DI, управление зависимостями передано внешнему контейнеру, а не самим классам.
2. Отделение интерфейса от реализации: Классы зависят от абстракций (интерфейсов), а не от конкретных реализаций.
3. Упрощение тестирования: Позволяет легко подменять зависимости на моки или стабы, что упрощает юнит-тестирование.

### Способы внедрения зависимостей

Существует несколько способов внедрения зависимостей:

1. Внедрение через конструктор: Зависимости передаются в класс через его конструктор.

* public interface IService

{

void Serve();

}

public class Service : IService

{

public void Serve()

{

Console.WriteLine("Service Called");

}

}

public class Consumer

{

private readonly IService \_service;

public Consumer(IService service)

{

\_service = service;

}

public void Start()

{

\_service.Serve();

}

}

2.Внедрение через свойства: Зависимости устанавливаются через публичные свойства класса.

public class Consumer

{

public IService Service { get; set; }

public void Start()

{

Service?.Serve();

}

}

Внедрение через интерфейсы: Класс реализует интерфейс, который содержит методы для установки зависимостей.

public interface IConsumer

{

void SetService(IService service);

}

public class Consumer : IConsumer

{

private IService \_service;

public void SetService(IService service)

{

\_service = service;

}

public void Start()

{

\_service?.Serve();

}

}

### Жизненные циклы

Transient: новый экземпляр объекта создаётся каждый раз, когда он запрашивается.

services.AddTransient<IService, Service>();

Scoped: экземпляр объекта создаётся один раз на каждый запрос. В основном используется в веб-приложениях.

services.AddScoped<IService, Service>();

Singleton: один и тот же экземпляр объекта используется на протяжении всего времени жизни приложения.

services.AddSingleton<IService, Service>(

**3.Контроль инверсий**

Контроль инверсий (Inversion of Control, IoC) — это принцип проектирования, который позволяет разрабатывать более гибкие и легко расширяемые приложения. В контексте C# и .NET, IoC означает передачу контроля за созданием объектов и управлением их зависимостями внешним компонентам, а не самим классам. Это улучшает модульность, тестируемость и поддержку кода.

### Основные концепции контроля инверсий

1. Определение IoC:
   * Контроль инверсий — это принцип, при котором объекты не создают свои зависимости самостоятельно, а получают их извне. Это позволяет классам оставаться независимыми от конкретных реализаций, что в свою очередь облегчает замену и тестирование.
2. Отделение интерфейса от реализации:
   * Классы зависят от абстракций (интерфейсов), что позволяет легко заменять реализации без изменения кода потребителя.
3. Упрощение тестирования:
   * С помощью IoC можно легко подменять зависимости на моки или стабы, что упрощает юнит-тестирование.

### Способы реализации IoC в C#

#### 1. Внедрение зависимостей (Dependency Injection)

Это наиболее распространенный способ реализации IoC. Внедрение зависимостей может осуществляться несколькими способами:

* Внедрение через конструктор:
* public interface IService

{

void Execute();

}

public class Service : IService

{

public void Execute()

{

Console.WriteLine("Service Executed");

}

}

public class Consumer

{

private readonly IService \_service;

public Consumer(IService service)

{

\_service = service;

}

public void Start()

{

\_service.Execute();

}

}

* Внедрение через свойства:
* public class Consumer

{

public IService Service { get; set; }

public void Start()

{

Service?.Execute();

}

}

* Внедрение через методы:
* public class Consumer

{

private IService \_service;

public void SetService(IService service)

{

\_service = service;

}

public void Start()

{

\_service?.Execute();

}

}

### Контейнеры IoC

Контейнеры IoC помогают управлять жизненным циклом объектов и их зависимостями. Они позволяют легко регистрировать и разрешать зависимости.

#### Пример использования контейнера IoC (например, Autofac)

using Autofac;

public interface IService

{

void Execute();

}

public class Service : IService

{

public void Execute()

{

Console.WriteLine("Service Executed");

}

}

public class Consumer

{

private readonly IService \_service;

public Consumer(IService service)

{

\_service = service;

}

public void Start()

{

\_service.Execute();

}

}

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

builder.RegisterType<Service>().As<IService>();

builder.RegisterType<Consumer>();

var container = builder.Build();

var consumer = container.Resolve<Consumer>();

consumer.Start(); // Output: Service Executed

}

}

### Service Locator

Service Locator — это паттерн, который позволяет запрашивать зависимости из центрального хранилища. Хотя это не является чистым IoC, этот подход также используется для управления зависимостями.

public class ServiceLocator

{

private static IService \_service;

public static void Register(IService service)

{

\_service = service;

}

public static IService GetService()

{

return \_service;

}

}

class Program

{

static void Main(string[] args)

{

ServiceLocator.Register(new Service());

var consumer = new Consumer(ServiceLocator.GetService());

consumer.Start(); // Output: Service Executed

}

}

### Преимущества контроля инверсий

1. Упрощение тестирования:
   * Легкость в создании юнит-тестов с подменой зависимостей.
2. Гибкость:
   * Легкость в изменении и замене реализаций.
3. Повышение модульности:
   * Классы меньше зависят друг от друга, что упрощает рефакторинг и сопровождение.

Тут наглядный пример как понимать и работать с инверсией зависимостей.

### https://habr.com/ru/articles/321344/