Day 04. Reflection

## Общие требования

* Убедитесь, что на вашем компьютере установлен [SDK для разработки на .NET 5](https://dotnet.microsoft.com/download) и вы используете именно его.
* Помните, ваш код будут читать! Обратите особое внимание на оформление вашего кода и именование переменных. Обязательно придерживайтесь общепринятых стандартов [C# Coding Conventions](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions).
* Самостоятельно выберите удобную для себя IDE.
* Программа должна иметь возможность запуска через командную строку dotnet.
* В каждом из заданий указаны примеры ввода и вывода. Решение должно использовать их как верный формат.
* В начале каждого задания приведен список рекомендованных языковых конструкций.
* Если затрудняетесь в решении задачи, обратитесь с вопросами к другим участникам бассейна, интернету, Google, посмотрите на StackOverflow.
* С основными возможностями языка C# можно ознакомиться в [официальной спецификации](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/language-specification/introduction).
* Избегайте **хардкода** и “**магических чисел**”.
* Вы демонстрируете все решение, верный результат работы программы – лишь один из способов проверки ее корректной работы. Поэтому когда необходимо получить определенный вывод в результате работы ваших программ, запрещено показывать пред рассчитанный результат.
* Обращайте особое внимание на термины, выделенные **bold** шрифтом: их лучше сразу погуглить, их изучение пригодится вам как в выполнении текущего задания, так и в вашей дальнейшей карьере .NET разработчика.
* Have fun :)

## Требования к заданиям дня

* Каждому из заданий должно соответствовать отдельное решение, точкой входа в которое является консольное приложение, созданное на основе стандартного шаблона .NET SDK.
* Используйте **var**.
* Название решения (и его отдельного каталога) должно выглядеть как d{*xx*}, где *xx* - цифры текущего дня. Названия проектов указаны в задании.
* Для форматирования выходных данных используйте **культуру** en-GB: N2 для вывода денежных сумм, d для дат.

## Что нового

* Рефлексия
* Метапрограммирование
* Атрибуты
* Runtime
* Открытое программное обеспечение
* Фабрика

## Рекомендованные языковые конструкции

* Activator
* params, typeof
* Type
* Attribute

# Интро

“Since everything is a reflection of our minds, everything can be changed by our minds.”

**Gautama Buddha**

Вы уже освоились в C# настолько, что решили немного порефлексировать. Но нет, речь не о том, чтобы размышлять о своем психическом состоянии или переживаниях. Вы решили открыть для себя темную сторону .NET - Reflection. Reflection или отражение, вид метапрограммирования, которое позволяет исследовать программу в **рантайме**. Например получать сведения о типах, составе и свойствах полей. И даже немного модифицировать их поведение.

Мы живем в удивительную эпоху, когда вокруг нас много информации в открытом доступе. Тоже касается и программного обеспечения. Есть прекрасный фильм об истории развития движения за **открытое программное обеспечение** - [Revolution OS](https://en.wikipedia.org/wiki/Revolution_OS). Посмотрите его на досуге.

Пока же порадуемся тому, что Microsoft тоже присоединилась к этому движению и весь исходный код платформы .NET доступен на [github](https://github.com/dotnet). Тому, что для любой библиотеки, которой мы используем в повседневной разработке, мы можем не только изучить исходники на гитхабе, но и получить информацию о ней прямо в коде в процессе работы нашего приложения.

Рассмотрим некоторые из них.

Примечание. Для лучшего понимания задания лучше выполнять в указанном порядке.

# Задание 00. Документация кода

## Разрешенные языковые конструкции

## Структура проекта

d04\_ex00

Program.cs

## Задание

Хорошо документированный код - код, который куда проще поддерживать. Это знают все. Как знают и то, что иногда это забывается или времени на хорошую документацию не хватает.

Что если можно попытаться делать это автоматически?

Для этого уже есть немало идей и их исполнений, но здесь и сейчас давайте посмотрим на основы. Возьмем реально существующий класс [DefaultHttpContext](https://github.com/dotnet/aspnetcore/blob/c660e9b1e3d6918e327499d340cbc38065e34436/src/Http/Http/src/DefaultHttpContext.cs). Он содержится в nuget-пакете Microsoft.AspNetCore.Http и активно используется в ASP.NET Core для хранения и управления информацией об http-запросах и ответах.

Рефлексия позволит нам получить информацию о типах, свойствах, методах этого класса. Почему бы не воспользоваться ей и не вывести на экран описание их всех?

Здесь нам помогут **класс Type**, который в сути своей является корнем **System.Reflection** и основным способом доступа к **метаданным**, и **его методы**.

Объект класса Type для DefaultHttpContext можно получить с помощью **typeof**.

Начнем с общей информации: давайте выведем полное наименование типа, описание его сборки (**Assembly**) и имя его базового типа.

Далее получим для класса DefaultHttpContext описание его [полей](https://docs.microsoft.com/en-us/dotnet/api/system.type.getfields?view=net-5.0) - давайте выведем для списка FieldInfo наименование и имя типа для каждого поля. То же сделаем для [свойств](https://docs.microsoft.com/en-us/dotnet/api/system.type.getproperties?view=net-5.0) и списка PropertyInfo.

Подумайте о том, [чем они отличаются](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-properties).

Еще у DefaultHttpContext наверняка есть [методы](https://docs.microsoft.com/en-us/dotnet/api/system.type.getmethods?view=net-5.0). Их список - список объектов MethodInfo - тоже нужно вывести, но для каждого из них указать название, возвращаемый методом тип и параметры метода.

При получении этих списков обратите внимание на **BindingFlags**. В выводе нас интересуют публичные и непубличные поля и только публичные методы и свойства. Учитывать нужно как **статичные**, так и не статичные.

Ни один из списков выше не требует сортировки при выводе. Вы ведь не пишете код в алфавитном порядке, не так ли?

### Формат ответа на выходе

Type: {FullName}

Assembly: {AssemblyFullName}

Based on: {BaseType}

Fields:

{field.TypeName} {field.Name}

...

Properties:

{property.TypeName} {property.Name}

…

Methods:

{method.ReturnTypeName} {method.Name} ({method.Parameters[]})

...

### Пример запуска приложения из папки проекта и вывода

$ dotnet run

Type: Microsoft.AspNetCore.Http.DefaultHttpContext

Assembly: Microsoft.AspNetCore.Http, Version=2.2.2.0, Culture=neutral, PublicKeyToken=adb9793829ddae60

Based on: Microsoft.AspNetCore.Http.HttpContext

Fields:

Microsoft.AspNetCore.Http.Features.FeatureReferences`1[Microsoft.AspNetCore.Http.DefaultHttpContext+FeatureInterfaces] \_features

Microsoft.AspNetCore.Http.HttpRequest \_request

Microsoft.AspNetCore.Http.HttpResponse \_response

Microsoft.AspNetCore.Http.Authentication.AuthenticationManager \_authenticationManager

Microsoft.AspNetCore.Http.ConnectionInfo \_connection

Microsoft.AspNetCore.Http.WebSocketManager \_websockets

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.IItemsFeature] \_newItemsFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.IServiceProvidersFeature] \_newServiceProvidersFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.Authentication.IHttpAuthenticationFeature] \_newHttpAuthenticationFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.IHttpRequestLifetimeFeature] \_newHttpRequestLifetimeFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.ISessionFeature] \_newSessionFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.ISessionFeature] \_nullSessionFeature

System.Func`2[Microsoft.AspNetCore.Http.Features.IFeatureCollection,Microsoft.AspNetCore.Http.Features.IHttpRequestIdentifierFeature] \_newHttpRequestIdentifierFeature

Properties:

Microsoft.AspNetCore.Http.Features.IFeatureCollection Features

Microsoft.AspNetCore.Http.HttpRequest Request

Microsoft.AspNetCore.Http.HttpResponse Response

Microsoft.AspNetCore.Http.ConnectionInfo Connection

Microsoft.AspNetCore.Http.Authentication.AuthenticationManager Authentication

Microsoft.AspNetCore.Http.WebSocketManager WebSockets

System.Security.Claims.ClaimsPrincipal User

System.Collections.Generic.IDictionary`2[System.Object,System.Object] Items

System.IServiceProvider RequestServices

System.Threading.CancellationToken RequestAborted

System.String TraceIdentifier

Microsoft.AspNetCore.Http.ISession Session

Methods:

IFeatureCollection get\_Features ()

HttpRequest get\_Request ()

HttpResponse get\_Response ()

ConnectionInfo get\_Connection ()

AuthenticationManager get\_Authentication ()

WebSocketManager get\_WebSockets ()

ClaimsPrincipal get\_User ()

Void set\_User (ClaimsPrincipal value)

IDictionary`2 get\_Items ()

Void set\_Items (IDictionary`2 value)

IServiceProvider get\_RequestServices ()

Void set\_RequestServices (IServiceProvider value)

CancellationToken get\_RequestAborted ()

Void set\_RequestAborted (CancellationToken value)

String get\_TraceIdentifier ()

Void set\_TraceIdentifier (String value)

ISession get\_Session ()

Void set\_Session (ISession value)

Void Initialize (IFeatureCollection features)

Void Uninitialize ()

Void Abort ()

Type GetType ()

String ToString ()

Boolean Equals (Object obj)

Int32 GetHashCode ()

# Задание 01. Нарушение правил

## Структура проекта

d04\_ex01

Program.cs

## Задание

Приватные поля изменять нельзя. Точка. Это невозможно по задумке, плохо и в большинстве стран карается законом.

Но любые знания дают нам силу. В том числе знания о том, как нарушать некоторые правила. И так мы узнаем, что значение приватного поля можно изменить в рантайме, используя рефлексию.

Давайте снова возьмем реально существующий класс [DefaultHttpContext](https://github.com/dotnet/aspnetcore/blob/c660e9b1e3d6918e327499d340cbc38065e34436/src/Http/Http/src/DefaultHttpContext.cs). Он содержится в nuget-пакете Microsoft.AspNetCore.Http и активно используется в ASP.NET Core для хранения и управления информацией об http-запросах и ответах.

Для начала создайте экземпляр класса DefaultHttpContext. Выведите значение его свойства Response в консоль. Вы увидите полное название типа DefaultHttpResponse, так сработал метод ToString() [по умолчанию](https://docs.microsoft.com/en-us/dotnet/api/system.object.tostring?view=net-5.0#the-default-objecttostring-method).

Свойство Response ничего в себе не хранит, оно **инкапсулирует** приватное поле \_response, значение которого не доступно напрямую (в этом и смысл приватности).

Когда создается экземпляр DefaultHttpResponse, в поле \_response создается и сохраняется объект DefaultHttpResponse. Его мы и вывели выше.

Давайте изменим его значение.

Получите объект типа FieldInfo для поля \_response. Не забудьте о флагах и том, что поле приватное и не статическое. В этом вам помогут знания полученные в задании выше.

Теперь воспользуйтесь [методом установки значения](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.fieldinfo.setvalue?view=net-5.0#System_Reflection_FieldInfo_SetValue_System_Object_System_Object_) и укажите новое значение - null.

Выведите повторно значение свойства Response в консоль. Будет выведена пустая строка.

Итак, мы изменили значение приватного поля. Имейте ввиду, скорее всего данная модификация сломает поведение класса DefaultHttpContext*.* Поэтому используйте данное знание с осторожностью.

### Формат ответа на выходе

Old Response value: {value}

New Response value: {value}

### Примеры запуска приложения из папки проекта и вывода

$ dotnet run

Old Response value: Microsoft.AspNetCore.Http.Internal.DefaultHttpResponse

New Response value:

# Задание 02. Автозаполнение

## Структура проекта

d04\_ex02/

Attributes/

NoDisplayAttribute.cs

ConsoleSetter/

ConsoleSetter.cs

Models/

IdentityUser.cs

IdentityRole.cs

Program.cs

## Задание

Что если мы хотим дать пользователю заполнить какой-то объект? Что если нам не обязательно знать, что это за объект, а с помощью рефлексии ваш код сам может пробежаться по свойствам класса и заполнить нужное?

Давайте попробуем.

Возьмем класс IdentityUser. Поместите его в проект:

public class IdentityUser

{

public IdentityUser()

{

}

public IdentityUser(string userName) : this()

{

UserName = userName;

}

public virtual string UserName { get; set; }

public virtual string NormalizedUserName { get; set; }

public virtual string Email { get; set; }

public virtual string NormalizedEmail { get; set; }

public virtual bool EmailConfirmed { get; set; }

public virtual string PasswordHash { get; set; }

public virtual string SecurityStamp { get; set; }

public virtual string ConcurrencyStamp() => Guid.NewGuid().ToString();

public virtual string PhoneNumber { get; set; }

public virtual bool PhoneNumberConfirmed { get; set; }

public virtual bool TwoFactorEnabled { get; set; }

public virtual DateTimeOffset? LockoutEnd { get; set; }

public virtual bool LockoutEnabled { get; set; }

public override string ToString()

=> $"User: {UserName}, {Email}, {PhoneNumber}";

}

За основу этого класса взят тип [IdentityUser](https://github.com/aspnet/AspNetIdentity/blob/main/src/Microsoft.AspNet.Identity.EntityFramework/IdentityUser.cs) из ASP.NET Identity. Данный класс используется для авторизации и аутентификации пользователей в Web-приложениях.

Давайте реализуем приложение, которое при запуске будет запрашивать у пользователя данные для заполнения этого класса. Разберемся подробнее.

В первом задании мы научились получать список свойств типа, во втором - устанавливать значения этих свойств. Создайте класс ConsoleSetter, он будет отвечать за заполнение данных через консоль в методе SetValues<T>(T input). Обратите внимание, методу должно быть все равно, какой именно объект его просят заполнить, поэтому мы делаем тип T **обобщенным**. Однако далее нам будет важно, чтобы этот тип являлся классом, давайте добавим [ограничение](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint).

Реализуйте метод *SetValues*. Пусть он для начала выведет в консоль приветствие “Let's set {typeName}!”, обозначая, какой именно класс мы сейчас будем заполнять. Теперь для каждого публичного и не статического свойства выведите в консоль просьбу “Set {propertyName}:” и установите значение, введенное на эту просьбу пользователем, в это свойство.

Получили заполнение класса без необходимости явно прописывать каждый вывод. Давайте настроим его.

Вводить большую часть значений неудобно и не нужно. Давайте определим, что для заполнения класса нам нужно попросить у пользователя заполнить только свойства *UserName, Email* и *PhoneNumber.*

Пометить свойства, не обязательные для заполнения, нам помогут **атрибуты**.

Атрибуты в C# позволяют добавить к типу различные **метаданные**. В .NET атрибуты используются повсеместно. Например для валидации моделей используются атрибуты из пространства имен **System.ComponentModel.DataAnnotations**. **Xunit** активно использует атрибуты в качестве маркеров, например для определения какой метод является тестом, а какой тестом с параметрами.

Помимо этого, атрибуты часто используются для динамической идентификации типов во время выполнения программы и для форматированного вывода. Это нам и пригодится.

Объявим [свой тип атрибутов](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/creating-custom-attributes) *NoDisplayAttribute*, унаследовав его от **System.Attribute**. Укажем также [условия его использования](https://docs.microsoft.com/en-us/dotnet/standard/attributes/writing-custom-attributes): он должен быть доступен к использованию только со свойствами, один и тот же атрибут можно указать у свойства только один раз. Пометьте этим атрибутом все свойства класса IdentityUser, кроме *UserName, Email* и *PhoneNumber.*

Теперь добавьте в *SetValues* фильтрацию: вывод и ввод нам нужен только для свойств, не отмеченных атрибутом *NoDisplayAttribute.* Сделать это можно, обратившись к [коллекции атрибутов](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/accessing-attributes-by-using-reflection) каждого свойства. Проверьте себя: для *IdentityUser* должны запрашиваться только три свойства.

Ну а теперь наведем красоту. Воспользуйтесь уже существующим атрибутами **DescriptionAttribute** и **DefaultValueAttribute**, укажите их для свойств *UserName (User name, “Me”), Email (Email address, “test@test”)* и *PhoneNumber (Phone number, “1234567890”)* класса *IdentityUser.* Теперь доработайте *SetValues* так, чтобы при запросе данных выводилась просьба с содержимым **DescriptionAttribute**, а в случае, если пользователь данные не ввел, свойство заполнялось значением по умолчанию, указанным в **DefaultValueAttribute**.

Осталось добавить в консоль сообщение об успешном заполнении и вывести данные методом ToString().

Проверьте себя: передайте в *SetValue* вместо экземпляра *IdentityUser* экземпляр другого класса:

public class IdentityRole

{

public IdentityRole()

{

}

public string Name { get; set; }

public string Description { get; set; }

public override string ToString()

=> $"{Name}, {Description}";

}

### Формат ответа на выходе

Let's set {typeName}

Set {propertyDescription}:

Set {propertyDescription}:

Set {propertyDescription}:

...

We've set our instance!

#### Примеры запуска приложения из папки проекта и вывода

$ dotnet run

Let's set IdentityUser!

Set User name:

Ann

Set Email address:

ann@test.ru

Set Phone number:

123000123

We've set our instance!

User: Ann, ann@test.ru, 123000123

$ dotnet run

Let's set IdentityUser!

Set User name:

Set Email address:

Set Phone number:

We've set our instance!

User: Me, test@test, 1234567890

$ dotnet run

Let's set IdentityRole!

Set Name:

Moderator

Set Description:

A role for moderation

We've set our instance!

Moderator, A role for moderation

# Задание 03. Создание объектов

## Структура проекта

d04\_ex03

Models/

IdentityUser.cs

IdentityRole.cs

Program.cs

TypeFactory.cs

## Задание

Иногда при использовании рефлексии бывает необходимо создать экземпляр определенного класса. Например, при реализации собственного **DI контейнера**, когда вы регистрируете в нем свои типы, которые хотите использовать для внедрения зависимостей. Способов использования может быть масса. Но так как действие происходит в рантайме, и тип, который нужно создать, также известен только в рантайме, у вас нет возможности использовать **ключевое слово new()**.

Как создать экземпляр объекта в рантайме?

Давайте реализуем класс **фабрики** *TypeFactory*, который будет предоставлять набор **статических** и **обобщенных** методов:

* CreateWithConstructor<T>
* CreateWithActivator<T>

В каждом из них мы будем использовать отдельный способ создания экземпляра класса T. Кроме того нам будет важно, чтобы T являлся классом, давайте добавим [ограничение](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint).

*CreateWithConstructor* позволит создать объект типа, используя конструктор объекта. Для реализации необходимо, используя рефлексию, [получить объект типа ConstructorInfo](https://docs.microsoft.com/en-us/dotnet/api/system.type.getconstructor?view=net-5.0). Для простоты воспользуемся конструктором без параметров, этот вариант описан в приведенной статье. Далее воспользуйтесь методом **Invoke** - это позволит вызвать конструктор класса и получить созданный объект.

*CreateWithActivator* позволит создать объект типа используя [методы](https://docs.microsoft.com/en-us/dotnet/api/system.activator?view=net-5.0) статического класса **Activator**.

Скопируйте в проект классы *IdentityUser* и *IdentityRole* из Задания 02.

Создайте с помощью *TypeFactory* два объекта типа *IdentityUser*. Напомним, что для простоты при создании мы используем конструктор без параметров, но *ConstructorInfo* и *Activator* также позволяют создавать объекты, используя конструктор с параметрами.

Выведите "user1 == user2" в случае равенства объектов и "user1 != user2" в обратном случае.

Сделайте то же для класса *IdentityRole.*

Подумайте, какой вывод должен быть здесь верным и [почему](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/equality-operators#reference-types-equality)?

### Бонусное задание

Добавьте в класс *TypeFactory* метод *CreateWithParameters<T>*, чтобы он принимал массив объектов (*object*) и, используя метод класса *Activator,* который позволяет передать параметры в конструктор, создавал экземпляр класса с помощью параметризованного конструктора.

Создайте объект типа *IdentityUser* с заданным UserName и выведите его имя.

### Формат ответа на выходе

$ dotnet run

d04\_ex03.Models.IdentityUser

user1 != user2

d04\_ex03.Models.IdentityRole

role1 != role2

### Формат ответа на выходе (бонусное задание)

$ dotnet run

d04\_ex03.Models.IdentityUser

user1 != user2

d04\_ex03.Models.IdentityRole

role1 != role2

d04\_ex03.Models.IdentityUser

Set name:

Activated user

Username set: Activated user