**2. Faker**

Необходимо реализовать генератор объектов со случайными тестовыми данными:  
class User

{

public string Name { get; set; }

public int Age { get; set; }

}

var faker = new Faker();

int i = faker.Create<int>(); // 542

double d = faker.Create<double>(); // 12.458

User user = faker.Create<User>(); // User { Name: "asdwerpasdf", Age: 987 }

// Далее в примерах List носит иллюстративный характер.

// Вместо него может быть выбрана любая коллекция из условия.

List<User> users = faker.Create<List<User>>();

List<List<User>> lists = faker.Create<List<List<User>>>();

List<int> ints = faker.Create<List<int>>();

При создании объекта следует использовать **конструктор,** а также заполнять **публичные поля и свойства с публичными сеттерами, которые не были заполнены в конструкторе**. Следует учитывать сценарии, когда у класса только приватный конструктор, несколько конструкторов, конструктор с параметрами и публичные поля/свойства.

При наличии нескольких конструкторов следует отдавать предпочтение конструктору с б***о***льшим числом параметров, однако если при попытке его использования возникло исключение, следует пытаться использовать остальные.

Обратите внимание, что у пользовательских типов-значений (value types), которыми являются структуры, объявляемые ключевым словом struct, **всегда есть конструктор без параметров,** однако в дополнение к нему может быть объявлен и пользовательский конструктор (который следует пытаться использовать первым, руководствуясь логикой предпочтения конструктора с б***о***льшим числом параметров).

Заполнение должно быть **рекурсивным** (если полем является другой объект, то он также должен быть создан с помощью Faker).

Реализовать генераторы случайных значений для базовых типов-значений (int, long, double, float, etc), строк, одного любого системного класса для представления определенного типа данных на выбор (дата/время, url, etc), коллекций объектов всех типов, которые могут быть сгенерированы Faker (поддержка разновидностей IEnumerable<T>, List<T>, IList<T>, ICollection<T>, T[] на усмотрение автора, минимум один вариант из приведенных).

**Создание коллекций должно выполняться аналогично созданию других типов, для которых есть генераторы.** Внутри кода Faker не должно быть проверок if/switch на коллекцию и какой-то особой обработки. Один из вариантов интерфейса IValueGenerator для такой обработки есть в [FAQ](https://bsuir.ishimko.me/mpp-dotnet/2-faker/faq#igenerator). Допустим и любой другой способ, согласующийся с требованием об отсутствии специальных проверок для коллекций.

Предусмотреть обработку **циклических зависимостей:**

class A

{

public B { get; set; }

}

class B

{

public C { get; set; }

}

class C

{

public A { get; set; } // циклическая зависимость,

// может быть на любом уровне вложенности

}

**Работа Faker'a должны быть полностью проверена с помощью модульных тестов. Использование для этого вспомогательной консольной программы запрещается.**

### Дополнительное задание

Настройка генерируемых случайных значений для конкретного поля путем передачи собственного генератора для конкретного поля/свойства:

var config = new FakerConfig();

// настройка может иметь и другой API на усмотрение автора

// (см. ограничение далее по заданию)

config.Add<Foo, string, CityGenerator>(foo => foo.City);

var faker = new Faker(config);

Foo foo = faker.Create<Foo>(); // при заполнении свойства City должен использоваться CityGenerator

**Ограничение:** задавать имя свойства/поля в виде строки (явно или с помощью оператора nameof) запрещено, следует использовать деревья выражений: [Expression Trees (C#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/).

Настройка должна работать и для неизменяемых объектов, свойства которых не имеют публичного доступа для записи, а создание выполняется через конструктор:

public class Person

{

public string Name { get; }

public Person(string name)

{

Name = name;

}

}

Если для такого объекта вы задаете собственный генератор для поля Name config.Add<Person, string, NameGenerator>(p => p.Name), то он должен использоваться при при генерации параметра name для конструктора. При обработке такой ситуации достаточно анализировать имена и типы параметров конструктора, предполагая, что его реализация тривиальна (параметры присваиваются свойствам с соответствующими именами).