Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Ambientes Virtuais de Execução – Semestre de Verão – 2015/2016

Terceira série de exercícios

Objectivos: Prática com Delegates e Genéricos
Data limite de entrega: 30 de Maio de 2016

Parte 1

Implemente a biblioteca `AutoMapper` que permite realizar o mapeamento entre as propriedades de objectos de tipos diferentes (**tipos valor** ou **referência**). Por exemplo, dada uma instância do tipo `Student` é possível obter uma instância do tipo `Person` de acordo com o teste unitário apresentado na Figura 1.

```
class Student {
  public string Name{ get; set; }
  public int Nr { get; set; }
}
```

```
class Person {
  public string Name{ get; set; }
  public int Nr { get; set; }
}
```

```
Mapper<Student, Person> m = AutoMapper
    .Build<Student, Person>()
    .CreateMapper();
Student s = new Student { nr = 27721, name = "Ze Manel" };
Person p = m.Map(s);
Assert.AreEqual(s.Name, p.Name);
Assert.AreEqual(s.Nr, p.Nr);
```

Figura 1

A instância de `Mapper` obtida na Figura 1 também deve permitir o mapeamento entre sequências de elementos tal como exemplificado na Figura 2.

```
Student[] stds = {
        new Student{ Nr = 27721, Name = "Ze Manel"},
        new Student{ Nr = 15642, Name = "Maria Papoila"}};
Person[] expected = {
        new Person{ Nr = 27721, Name = "Ze Manel"},
        new Person{ Nr = 15642, Name = "Maria Papoila"}};
Mapper<Student, Person> m = AutoMapper
    .Build<Student, Person>()
    .CreateMapper();

List<Person> ps = m.Map<List<Person>>(stds);

CollectionAssert.AreEqual(expected, ps.ToArray());
```

Figura 2

O `AutoMapper` só suporta o mapeamento para propriedades com o **mesmo nome** e **tipo compatível**. Além disso, deve suportar casos em que o tipo destino seja um tipo referência e **não** tenha um construtor sem parâmetros (implemente a solução que entender mais adequado para este caso).
O método `CreateMapper` retorna uma instância de `Mapper` com a interface apresentada na Figura 3.

```
interface Mapper<TSrc, TDest>
{
    TDest Map(TSrc src);
    TColDest Map<TColDest>(IEnumerable<TSrc> src) where TColDest : ICollection<TDest>;
}
```

Figura 3

Implemente os testes unitários necessários para validar o correcto funcionamento de todas as funcionalidades pedidas, incluindo para **tipos valor** e **tipos referência** (**com** e **sem** construtor sem parâmetros).

Parte 2

Adicione a possibilidade de mapear sequências de elementos para Arrays ou `IEnumerable<T>`. Para tal, adicione a `Mapper` os métodos `MapToArray` e `MapLazy` apresentados na Figura 4.

```
interface Mapper<TSrc, TDest>
{
    TDest Map(TSrc src);
    TColDest Map<TColDest>(IEnumerable<TSrc> src) where TColDest : ICollection<TDest>;
    TDest[] MapToArray(IEnumerable<TSrc> src);
    IEnumerable<TDest> MapLazy(IEnumerable<TSrc> src);
}
```

Figura 4

Parte 3

Por omissão o `AutoMapper` faz a correspondência automática entre propriedades com o mesmo nome e tipo. Pretende-se dar ao utilizador a possibilidade de indicar quais as propriedades que devem ser ignoradas pelo `AutoMapper`. Para tal o utilizador pode especificar:

- o nome da propriedade que não deve ser afectada pelo `Mapper` conforme exemplo da Figura 5.
- o tipo do *custtom attribute* anotado nas propriedades que devem ser ignoradas, conforme exemplo da Figura 6.

```
Mapper<Student, Person> m = AutoMapper
                .Build<Student, Person>()
                .IgnoreMember("Name")
                .CreateMapper();
```

Figura 5

```
Mapper<Student, Person> m = AutoMapper
                .Build<Student, Person>()
                .IgnoreMember<AvoidMapping>()
                .CreateMapper();
```

Figura 6

Parte 4

No exemplo da Figura 7 a propriedade `Id` de `Person` não tem correspondência automática com nenhuma propriedade de `Student` de acordo com a convenção por omissão do `AutoMapper`.

```
class Student {
  public string Name{ get; set; }
  public int Nr { get; set; }
}
```

```
class Person {
  public string Name{ get; set; }
  public string Id { get; set; }
}
```

Figura 7

Contudo, o `AutoMapper` deve permitir a adaptação de mapeamentos através do método `ForMember` de acordo com o exemplo da Figura 8. O método `ForMember` recebe o nome da propriedade destino e uma função que extrai um valor do objecto fonte. O tipo do valor extraído do objecto fonte e o tipo da propriedade destino têm que ser compatíveis.

```
Mapper<Student, Person> m = AutoMapper
     .Build<Student, Person>()
     .ForMember("Id", src => src.Nr.ToString())
     .CreateMapper();
Student s = new Student { nr = 27721, name = "Ze Manel" };
Person p = m.Map(s);
Assert.AreEqual(s.Name, p.Name);
Assert.AreEqual(s.Nr.ToString(), p.Id);
```

Figura 8

Observações finais:

Note que o `AutoMapper` oferece uma API fluente, ou seja, podem ser encadeadas chamadas consecutivas aos métodos `IgnoreMember()` e `ForMember()`, sobre o objecto resultante do `Build()`.