

1. [6] Pretende-se desenvolver uma biblioteca para definição e validação de regras sobre propriedades de objectos.

Uma regra é representada pela interface `IValidation { bool Validate(object obj); }` (exemplos de implementações desta interface: `Above18`, `NotNull`).

Uma instância de `Validator<T>` representa um validador para uma entidade de domínio de tipo `T`, aos quais é possível acrescentar regras de validação para determinadas propriedade (identificada pelo seu nome), tal como apresentado no exemplo seguinte:

<pre>struct Student { public int Age; public String Name; }</pre>	<pre>Student s = new Student(); s.Age = 20; s.Name = "Anacleto"; Validator<Student> validator = new Validator<Student>() .AddValidation("Age", new Above18()) .AddValidation("Name", new NotNull()); validator.Validate(s);</pre>
---------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- a) [1] Implemente a classe `Above18`, que retorna `true` se o valor recebido por parâmetro for superior a 18, ou `false` caso contrário.
- b) [2] Implemente a classe `Validator<T>` tendo em conta que o método `Validate` lança a exceção `ValidationException`, se falhar alguma das regras.
- c) [2] Sem alterar o código escrito anteriormente, acrescente o necessário para que seja suportada a adição de regras na forma de delegates do tipo `Func<W, bool>`, conforme demonstra o exemplo seguinte. Se a propriedade indicada não for do tipo `W`, então é lançada a exceção `TypeMismatchException`.

<pre>Validator<Student> validator = new Validator<Student>() .AddValidation<String>("Name", UtilMethods.Max50Chars);</pre>	<pre>class UtilMethods { public static bool Max50Chars(String s) { /* ... */ } }</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------

- d) [2] Adicione o necessário e modifique apenas o método `AddValidation(string name, IValidation val)` de modo a que sobre a mesma propriedade possam ser adicionados vários validadores.

2. Reflexão, Interfaces, Delegates

Pretende-se implementar uma forma de construir validações para métodos que retornam `void` e recebem um único parâmetro.

O caso de utilização seguinte demonstra um exemplo de construção de uma instância de `Action<T>` que executa o método `AddRate` sobre o objecto `apple`, caso todos os predicados especificados com `With` retornem `true`.

Lança a exceção `ArgumentException` caso contrário. O método `With` pode ser chamado zero ou mais vezes.

```
Action<double> addRate = Properties  
    .ValidateMethod<double>(apple.AddRate)  
    .With(val => val >= 0)  
    .With(val => isLegalTax(val))  
    .Create();  
addRate(7.3); // Add 7.3 rate to apple if tax is positive and legal
```