Ambientes virtuais de Execução – 1.º Teste de Época Normal – 18 de janeiro de 2019
2018/2019 Semestre de Inverno - Duração 2h30

Número: _____ Nome: _____

Nas questões 1 a 3, marque cada alternativa como verdadeira (V) ou falsa (F). Uma alternativa assinalada corretamente conta 0,5 valores, incorretamente desconta 0,25 valores ao total da respectiva questão.

1. O método genérico `Print` abaixo pode ser invocado sobre tipos valor?

```
class Program { public static void Print<T>(T t) { Console.WriteLine(t.ToString()); } }
```

   (a) ____ Não, porque falta indicar a restrição `where T : struct` no parâmetro tipo `T`.

   (b) ____ Sim, mas só se o tipo valor for um tipo anulável (`Nullable`).

   (c) ____ Sim, mas poderá ocorrer uma operação de *box* dentro do código do método `Print`.

   (d) ____ Sim, mas poderá ocorrer uma operação de *box* ou ser lançada uma exceção `System.NullReferenceException` dentro do código do método `Print`.

2. Com as classes do espaço de nomes `Reflection.Emit`...

   (a) ____ é possível acrescentar métodos a classes já existentes.

   (b) ____ é possível acrescentar novos tipos num *assembly* existente.

   (c) ____ é possível gerar mais do que uma definição de um tipo no presente *assembly* dinâmico.

   (d) ____ é possível usar um tipo gerado num *assembly* dinâmico mesmo que não se guarde a referência para a instância de `Type` na altura da geração.

3. Considere as definições:

```
public delegate void WithParamC(C arg);
public delegate C WithReturnC();
public class C
{
  public void F1(object obj) { }
  public void F2(Derived obj) { }
  public object F3() {
    return new Derived();
  }
  public Derived F4() {
    return new Derived();
  }
}
public class Derived : C {}
```

   Dada a variável `c` definida pela expressão: `C c = new C();` e tendo em conta as regras de co-variância e contra-variância dos *delegates* ...

   (a) ____ o código `WithReturnC del3 = c.F3; Derived obj = (Derived)del3();` **não é aceite** pelo compilador.

   (b) ____ o código `WithReturnC del4 = c.F4; object obj1 = del4();` **é aceite** pelo compilador.

   (c) ____ o código `WithParamC del1 = c.F1; del1(new Derived());` **não é aceite** pelo compilador.

   (d) ____ o código `WithParamC del2 = c.F2; del2(new Derived());` **é aceite** pelo compilador.

4. [2,5] Escreva em IL o código do construtor e do método `InvokeAll` da classe `B`.

```
delegate object Func(int p);
abstract class A {
  protected Func Handlers { get; set; }
  protected int MaxNumHandlers { get; set; }
  protected A(int maxNumHandlers) {
    this.MaxNumHandlers = maxNumHandlers;
  }
}
```

```
class B : A
{
  public B() : base(10) { Handlers = Bundle; }
  private object Bundle(int p) { return p; }
  public object InvokeAll(object parameter) {
    return Handlers((int)parameter);
  }
}
```

5. [2,5] Acrescente à interface `IEnumerable<T>` suporte para a operação *lazy* genérica `Chunk`, que recebe uma sequência de `T` e produz uma nova sequência de sequências de `T` (i.e. `IEnumerable<IEnumerable<T>>`) obtida separando a sequência fonte de acordo com o predicado passado a `Chunk` (e.g. `v => v % 2 == 0`). Ou seja, o último elemento de cada subsequência satisfaz o predicado, sendo que os anteriores elementos não. As sequências deverão ser *lazy*.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
IEnumerable<IEnumerable<int>> chunked = numbers.
    Chunk(v => v % 2 == 0);
foreach (IEnumerable<int> s in chunked) {
  Console.WriteLine(String.Join(",", s));
}
```

```
Output:
1,2
3,4
5,6
7,8
9,10
```

6. [9]

```
public interface IFixture
{
  object New();
}
```

```
public interface IPropertyFixture {
  void SetNewValue(object target);
  PropertyInfo GetProperty();
}
```

```
public class Fixture : IFixture {
  public static int MAX = 10;
  public static Random random = new Random();
  private Type targetType;
  private static Dictionary<Type, IFixture> fixtures = new Dictionary<Type, IFixture>();
  private List<IPropertyFixture> propertyFixtures = new List<IPropertyFixture>();
  private static IFixture GetOrCreateFixture(Type type) {
    IFixture fix = null;
    if (fixtures.ContainsKey(type)) {
      fix = fixtures[type];
    }
    else {
      fix = new Fixture(type);
      fixtures.Add(type, fix);
    }
    return fix;
  }
  static Fixture() {
    // Fixtures de tipos primitivos suportados
    fixtures.Add(typeof(int), new FixtureInt());
  }
  ...
}
```

Uma instância da classe `Fixture` permite gerar uma instância aleatória, do tipo cujo `Type` é passado no parâmetro do construtor de `Fixture`. A interface `IFixture` define o método `New` usado para gerar a instância aleatória.

```
public class Student {
  public Student(School school) {
    this.School = school;
  }
  public Student(int nr, School school) {
    this.Nr = nr; this.School = school;
  }
  public int Nr { get; set; }
  public School School { get; set; }
  public override string ToString() {
    return string.Format("[Student: Nr = {0},
        School = {1}]", Nr, School);
  }
}
```

```
public class School
{
  public int Id { get; set; }
  public School(int id)
  {
    this.Id = id;
  }
  public override string ToString()
  {
    return string.Format("[School: Id={0}]",
        Id);
  }
}
```

```
Fixture fixture = new Fixture(typeof(Student));
var autoGeneratedClass = fixture.New();
Console.WriteLine(autoGeneratedClass);
```

```
Output:
[Student: Nr = 1, School = [School: Id=5]]
```

A geração da instância aleatória é feita de duas formas: 1) escolhendo um construtor aleatoriamente, e escolhendo valores aleatórios para os seus parâmetros, **e** 2) inicializando as propriedades do objecto também de forma aleatória. Admita que o `Type` passado no parâmetro do construtor representa um **tipo referência** (i.e. não são suportadas instâncias da classe `Fixture` que representem tipos-valor). Contudo, é possível obter um objecto `IFixture` para tipos-valor que foram adicionados à *cache* `fixtures` usando o método estático `GetOrCreateFixture`.

A **eficiência** da solução implementada é contabilizada na avaliação das questões.

NÃO poderá acrescentar outros campos à classe `Fixture` além dos definidos na listagem apresentada. **Poderá implementar outras classes auxiliares.**

(a) [2] Defina o construtor de `Fixture` que popula a lista `propertyFixtures` com objectos que serão usados pelo método `New` para iniciar cada uma das propriedades do objecto de forma aleatória. Os objectos armazenados em `propertyFixtures` implementam a interface `IPropertyFixture` (ver listagem acima).

(b) [2] Implemente o método `New` que cria uma instância aleatória do tipo `targetType` escolhendo 1) um construtor aleatoriamente, e escolhendo valores aleatórios para os seus parâmetros, e 2) inicializando as propriedades do objecto também de forma aleatória.

(c) [1] Sem escrever código nem dar detalhes de implementação enumere apenas os campos e/ou métodos da classe `Fixture` e interface `IFixture` que teria de modificar ou adicionar para suportar propriedades do tipo `IEnumerable` genéricas simples (não considere enumeráveis de enumeráveis).

(d) [2] Pretende-se dar suporte à definição de um gerador específico para uma dada propriedade através de um método `SetSupplier` que recebe uma instância de um *delegate* como no seguinte exemplo: `fixture.SetSupplier<int>("Nr", () => 10);`

O novo gerador indicado através deste método **deve substituir** o gerador existente criado no construtor de `Fixture`. Implemente o método `SetSupplier` se necessário criando nova(s) classe(s) auxiliar(es), mas sem alterar **NADA** das alíneas anteriores.

(e) [2] No caso de propriedades de tipo primitivo pretende-se que estas possam ser marcadas com uma anotação que especifica a validação a realizar ao valor aleatório antes de afectar essa propriedade. Considere, por exemplo, que para a propriedade `Nr` queria que o valor fosse menor que 50000.

i. Implemente o *Custom Attribute* e demonstre como é usado no exemplo dado para a propriedade `Nr` de `Student`.

ii. Implemente a solução adicionando código **apenas** no construtor da classe `Fixture` e se necessário criando nova(s) classe(s) auxiliar(es).