Ambientes Virtuais de Execução
(1ºS 2010/2011)

Lista de Exercícios de Preparação para a 2ª Ficha

I Parte

A. Estrutura de Tipos *(Passagem de parâmetros por valor e referência. Propriedades. Interfaces, Arrays e Enumerados.)*

1. Diga quais as restrições que se aplicam às constantes enquanto membros de um tipo. Quais as diferenças que existem entre usar uma constante ou um membro estático *read-only* quando se pretende definir um valor constante.
2. O C# usa as palavras chave out e ref para especificar a passagem de parâmetros por referência. Qual a diferença de semântica entre estas duas palavras chave? Porque razão o C# não permite o overload de métodos que se distinguem apenas pela utilização de out em vez de ref ou vice-versa?
3. O que são *indexers*? Quais as restrições na declaração de um *indexer*?
4. Qual a responsabilidade do construtor de tipo no CLR? Diga quais são as duas semânticas de invocação do construtor de tipo.
5. De que forma um contentor pode implementar ambas as interfaces genérica `IEnumerator<T>` e não genérica `IEnumerator`? Realize um troço de código demonstrativo.
6. Numa interface em C# podem ser declarados métodos e eventos, mas não podem ser declarados *delegates*. Porquê?
7. O que entende por implementação explícita de membros de uma interface em C#? Indique duas situações práticas onde é essencial usar esta técnica.

B. Delegates e Eventos

1. A declaração dum evento inclui o nome dum tipo *delegate*. Qual a necessidade de se especificar este tipo?
2. Enumere duas razões que justifiquem a escrita de código específico para controlar a subscrição/revogação de eventos. Como é que isso pode ser feito em C#?
3. Considere a definição do delegate: public delegate `int MyDelegate(int x, string y);`

   a) Explique o código gerado pelo compliador de C# dada esta definição.

Nota: Na descrição omita a explicação dos métodos `BeginInvoke` e `EndInvoke`.

   b) Qual o código gerado pelo compilador para a seguinte instrução, assumindo que `myDel` é uma referência para uma instânciade MyDelegate: `myDel(2, "SLB");`

4. Num *multicast delegate*, um ou mais métodos presentes na cadeia de invocação podem lançar uma excepção.

   c) Quais os cuidados a ter neste caso? Desenvolva um código ilustrativo deste problema.

   d) Altere o código desenvolvido na alínea anterior, de modo que seja retornada uma excepção (dum tipo `Exception` a criar) que inclua todas as excepções lançadas na cadeia, bem como o `delegate` onde este ocorreu. A excepção retornada tem suporte para enumeração de objectos que incluem a excepção lançada e bem como o nome do método associado ao `delegate` e tipo onde o método está definido.

5. Qual o problema do código seguinte:

```
class TargetMethods {
    delegate void Action(int x);
    delegate void OtherAction(int x);
    public static void DoAction(Action a, int i) { a(i); }
    public static void SomeAction(int i) { Console.WriteLine(i);}

    public static void Test() {
        OtherAction o = SomeAction;
        DoAction(SomeAction, 10);
        DoAction(o, 10);
    }
}
```

C. Genéricos e Tipos anuláveis

1. O que entende por um tipo aberto?

2. O tipo parâmetro da classe G pode ser um tipo da categoria valor ou referência. A chamada ao método virtual ToString terá de ser feita de forma diferente em função da categoria do tipo T. Explique sucintamente a solução existente na plataforma .NET para que o compilador C# consiga traduzir este código para IL.

```
class G<T> {
   private T t;
   public String m() { return t.ToString(); }
}
```

3. Considere o seguinte troço de código C#. Qual o output resultante da invocação do método f?

```
class GenericClass<T> {
     public static int aField=1;
     public static void f() {
            GenericClass<int>.aField = 10;
            Console.WriteLine(GenericClass<int>.aField + GenericClass<char>.aField);
     }
}
```

4. O tipo `System.Nullable` necessita de algum tratamento especial pela VES (máquina virtual de execução)? Justifique dando exemplos.

5. Porque razão foi definida a interface genérica `IEquatable<T>` se já existe o método `Equals` em `Object`?

D. Gestão de Memória (Libertação automática de memória. Finalização de objectos; finalização determinística; Destrutores em C#. Pattern Dispose.)

1. O que se consideram "raízes" no contexto da *garbage collection*? Identifique os tipos de raízes existentes no CLR.

2. Compare a eficiência do *managed heap* do CLR com a de um *heap* clássico (C/C++) no que se refere à operação de alocação de memória.

3. O processo de recolha de memória (*garbage collection*) utiliza informação que lhe permite determinar as raízes existentes na aplicação para um dado ciclo de recolha. De que informação se trata e quem a produz?

4. Explique sucintamente mas através de casos concretos as vantagens da existência de gerações nos algoritmos de recolha automática de memória. Haverá desvantagens?

5. Qual a vantagem da utilização do padrão *Dispose* em relação à utilização simples do *finalizer*? Fará sentido que um tipo implemente o padrão *Dispose* sem redefinir o método `Finalize`?

6. É possível que um objecto que implemente o método Finalize() seja destruído (o seu espaço reclamado) na geração 0? Justifique.

7. Quais as consequências que decorrem do facto do método **`Finalize`** bloquear a *thread* que o executa? (Escreva um programa em que isso aconteça e analise o respectivo comportamento.)

II Parte

1. Considere as seguintes definições:

```
public interface ICommon
{
   void DoIt();
}
```

```
public class Base : ICommon {
   void ICommon.DoIt() { }
   public virtual void DoIt() { }
   public delegate void Del();
}

public class Derived : Base, ICommon {
   void ICommon.DoIt() { }
   public new virtual void DoIt() { Console.WriteLine}
   public void m() { }
   public event Action<int> someIntAction;
}
```

a) Represente sucintamente a tabela de métodos dos tipos `Base` e `Derived`, incluindo a tabela de interfaces. Identifique claramente os métodos cuja chamada só é decidida em tempo de execução.

b) Quais os métodos realmente chamados aquando da execução do seguinte código?

```
static void Main() {
   Derived r1 = new Derived();
   Base    r2 = r1;
   ICommon r3 = r1;
   r2.DoIt();
   r3.DoIt();
}
```

2. Considere o seguinte troço de código.

```
class TestDelegate {
    public delegate int D(int x);
    int M1(int x)
    {
        x++;
        Console.Write("M1/{0} ", x);
        return x;
    }
    static int M2(int x)
    {
        x++;
        Console.Write("M2/{0} ", x);
        return x;
    }
```

```
    static void Main(string[] args)
    {
        TestDelegate o = new TestDelegate();
        D dlg1 = new D(o.M1), dlg2 = new D(M2),
          dlg3 = dlg1 + dlg2;
        dlg3 += dlg3;
        int y = 0;
        Console.WriteLine(dlg3(y));
        dlg3 -= new D(o.M1);
        Console.WriteLine(dlg3(y));

        Console.ReadKey();
    }
}
```

a) Qual a saída produzida pelo método `Main`? Justifique.

b) Se o *delegate* `D` e os métodos `M1` e `M2` recebessem o parâmetro por referência (aplicando o atributo `ref`), qual seria a saída? Porquê? Qual a consequência de usar, em alternativa, o atributo `out`?

3. O método `Zip` junta cada um dos elementos da primeira sequência com o elemento de mesmo índice da segunda sequência, Se as sequências não têm o mesmo número de elementos, o método junta as sequências até uma delas terminar. Este método recebe uma função que define como a junção é realizada.

e) Implemente o método `Zip` como um método de extensão para `IEnumerable<T>`. O método tem a seguinte assinatura:

```
IEnumerable<TResult> Zip<TFirst, TSecond, TResult>(
      this IEnumerable<TFirst> first,
      IEnumerable<TSecond> second,
      Func<TFirst, TSecond, TResult> resultSelector
);
```

f) Utilizando o *extension method* Zip, desenvolvido na alínea anterior, implemente o método `IEnumerable<string> GetFileLines(string fileName)` que retorna um enumerado de strings, cada um delas contendo o número de linha e o conteúdo dessa linha do ficheiro de texto cuja path é `fileName`.

4. O método `IEnumerable<FileInfo> GetFiles(string baseDir)` retorna a lista recursiva de ficheiros existentes abaixo de baseDir.

g) Implemente o método `GetFiles`.

h) Utilizando o método GetFiles desenvolvido na alínea anterior e o *extension method* `IEnumerable<T> Where<T>(this IEnumerable<T>col, Func<T, bool>predicate)` presente no *namespace* `System.Linq`, crie um programa exemplo que liste recursivamente na consola todos os ficheiros a partir de uma pasta raíz que tenham a extensão ".cs" e todos os ficheiros maiores que uma dada dimensão em `KBytes`.

5. Considere o código apresentado à frente. Na resposta às questões seguintes assuma que o código é compilado em modo *release* e que apenas ocorre o ciclo de recolha explícito no código.

```
/*
  Fábrica específica para optimizar
  a criação de objectos em cenários em que o tempo
  de criação é significativo e existe
  muita dinâmica de criação destruição
*/
class PoolFactory<T>  where T:new() {
  WeakReference[] pool;
  int count;

  public static PoolFactory<T>
  Create(int size) {
    return new PoolFactory<T>(size);
  }

  public PoolFactory(int size) {
     pool = new WeakReference[size];
     count=0;
  }

  public T get() {
    while (count != 0) {
       WeakReference w =  pool[--count];
       if (w.Target != null) {
          pool[count]=null;
          return (T) w.Target;
       }
    }
    return new T();
  }

  public void put(T t) {
     if (count == pool.Length)
       return;
     pool[count++] = new WeakReference(t);
  }
}
```

```
public class A { double d;  }

class Program {
  static PoolFactory<A> factory;
  const int TABLESIZE=1024*32;

  static void showMem(string msg) {
     System.Console.WriteLine("Memory on {1}={0}",
              GC.GetTotalMemory(false), msg);
  }

  static void Main(string[] args) {
      factory = PoolFactory<A>.Create(TABLESIZE);
      A[] As = new A[TABLESIZE];

      showMem("on start");
      for (int i = 0; i < TABLESIZE; ++i)
         As[i] = factory.get();

       showMem("after creation 1");
       for (int i = 0; i < TABLESIZE; ++i) {
          factory.put( As[i]);
          As[i] = null;
       }

       showMem("after delection 1");

       for (int i = 0; i < TABLESIZE; ++i) {
          As[i] = factory.get();
          As[i] = null;
       }
       showMem ("after creation 2");
       GC.Collect();

       showMem("after delection 2");
       GC.Collect();
       GC.WaitForPendingFinalizers();
       showMem("after delection 3");

       GC.Collect();
       showMem("after delection 4");
   }
  }
```

a) Justifique output resultante da execução do programa, quer do ponto de vista da relação entre os resultados das vária chamadas à função `showMem` quer, na medida do possível, o valor apresentado por cada chamada.

b) Qual a consequência no *output* apresentado da remoção da linha marcada a ***bold*** na classe genérica `PoolFactory<T>`?

6. Em certos cenários era interessante ter *delegates* que não forcem o tempo de vida dos *targets* associados, que se podem designar como *weak delegates*. Quando o *delegate*(*weak*) tiver a única referência para o objecto *target*, o *target* pode ser reclamado pelo GC, deixando, a partir desse momento, de ser realizado o *invoke* do delegate. Como os *delegates* na implementação actual do CLR guardam sempre *strong references* para o respectivo *targets*, este modo de funcionamento não é suportado. O código seguinte apresenta uma proposta de uma classe genérica(`WeakDelegate<T>`) para representar *weak delegates* e uma especialização (`WeakEventHandler`) para *weak delegates* do tipo `EventHandler` .

```
public abstract class WeakDelegate<T>   where T: class {
    WeakReference r;

    public WeakDelegate(T d)  {
        if (!(d is Delegate)) throw new ArgumentException("Not a delegate type!");
        if (d.GetInvocationList().Length > 1)
          throw new ArgumentException("Can´t be a chained delegate!");
        r = new WeakReference(d);
    }
    public abstract T Delegate { get; }
    protected T RealDelegate { get { return (T)r.Target; } }

}

public class WeakEventHandler : WeakDelegate<EventHandler> {
    public WeakEventHandler(EventHandler d) : base(d) { }

    public override EventHandler Delegate { get { return Invoke; } }

    private void Invoke(object sender, EventArgs args) {
        EventHandler e = RealDelegate;
        if (e != null) e(sender, args);
    }
}
```

Considere a seguinte utilização da classe `WeakEventHandler`. Na resposta às questões seguintes assuma que o código é compilado em modo *release* e que apenas ocorre o ciclo de recolha explícito no código.

```
public class A {
    public  void callback(object sender, EventArgs args)  { Console.WriteLine("callback called!"); }
}

public class Program {
    public static void Main() {
        A a = new A();
        EventHandler e = a.callback;
        WeakEventHandler2 d = new WeakEventHandler2(e);
        d.Delegate(null, EventArgs.Empty);
        GC.Collect();
        d.Delegate(null, EventArgs.Empty);
        GC.KeepAlive(a);
    }
}
```

a) Explique o código das classes `WeakDelegate<T>` e `WeakEventHandler` e justifique o *output* resultante da execução do programa, identificando os problemas da pseudo solução apresentada.

b) Modifique a classe genérica para lançar uma excepção no caso de ser passado no construtor um *delegate* associado a um método estático.

c) Faça as alterações necessárias para obter o comportamento sugerido na introdução à questão.

Poderá ser útil considerar na sua resposta o método estático da classe `Delegate`,

```
Delegate CreateDelegate(Type delegateType, Object target, MethodInfo method)
```

que cria e retorna um *delegate* de acordo com os argumentos.

d) Altere a solução proposta para utilizar `GCHandles` em vez de *weak references*. Existe alguma potencial vantagem nesta alternativa? E desvantagem?

e) Proponha e implemente uma solução para remover um *weak delegate* de uma cadeia de *weak delegates* quando o *target* associado tiver sido reclamado pelo GC.