PADI 2012/13

Aula 2
Tópicos Adicionais de C#

Sumário

1. Propriedades
2. Excepções
3. Delegates e eventos
4. Generics
5. Threads e sincronização

1. Properties

Get/Set

Properties

- Maneira simples de controlar o acesso a campos privados de classes, *structs* e interfaces

```
public class X {
    private string nome;
    private int idade;

    public string Nome {
          get { return nome; }
          set { nome = value; }
    }

                    (...)

x.Nome = “Meireles”;
Console.WriteLine(“Chamo-me {0}", x.Nome);
```

- Formaliza o conceito dos métodos Get/Set
- Código mais legível

2. Excepções

Sintaxe

Lançamento

Intercepção

Excepções

- Causadas por uma situação **excepcional**
- São lançadas pelo sistema ou usando *throw*
- Usa-se colocando o código dentro de um bloco:

```
try {
   //código susceptível de gerar excepção
} catch (<TipoDaExcepção> e) {
   // tratamento da excepção e
} finally {
// é sempre executado
// limpeza de recursos alocados no try
}
```

- Podem-se criar novas excepções derivando de System.ApplicationException

Lançamento de uma Excepção

```
class MinhaExcepcao: ApplicationException {
     // métodos e atributos arbitrário
   }

   ...

   if (<condição de erro>) {
     throw new MinhaExcepcao();
   }
```

Intercepção de Excepções

- É procurado, pela pilha acima, um bloco `try`
- Procura-se um `catch` para a excepção
- Se não houver, executa-se o `finally`
- e sobe-se pelo pilha acima, executando os `finally` que houver, até a excepção ser apanhada ou o programa ser terminado ☹

Excepções Típicas

- System.ArithmeticException
- System.ArrayTypeMismatchException
- System.DivideByZeroException
- System.IndexOutOfRangeException.
- System.InvalidCastException
- System.MulticastNotSupportedException
- System.NullReferenceException
- System.OutOfMemoryException
- System.OverflowException
- System.StackOverflowException
- System.TypeInitializationException

Dicas para lidar com Excepções

- Não lançem excepções apenas para controlo de fluxo do programa.
- Não apanhem excepções que vocês não vão tratar.
- Truque: quando apanham uma excepção mas precisam de a atirar de novo, usem a cláusula *throw* isolada.
  - Evita que o *stacktrace* da excepção seja reescrito

```
try {
   //código susceptível de gerar excepção
} catch (<TipoDaExcepção> e) {
   // tratamento da excepção e
   throw; // em vez de “throw e;”
}
```

3. Delegates e Eventos

Delegates

- Semelhante a apontadores para funções:
  ```
  bool (*minhaFunc) (int) /* em C */
  ```

- Apontadores para métodos de objectos ou de classes:

  ```
  delegate bool MeuDelegate(int x);
  MeuDelegate md = new MeuDelegate(método);
  ```

- *Delegates* guardam uma "lista" de métodos.

- Podem ser manipulados com operações aritméticas: Combine (+), Remove (-)

- Um *delegate* vazio é igual a null.

Delegates (cont.)

```
delegate void MyDelegate(string s);

class MyClass {
public static void Hello(string s) {
  Console.WriteLine(" Hello, {0}!", s);
}

public static void Goodbye(string s) {
  Console.WriteLine(" Goodbye, {0}!", s);
}
```

```
public static void Main() {
  MyDelegate a, b, c;
  a = new MyDelegate(Hello);
  b = new MyDelegate(Goodbye);
  c = a + b;
  a("A");
  b("B");
  c("C");
}
}
```

```
Hello, A!
GoodBye, B!
Hello, C!
GoodBye, C!
```

Eventos

- Publicação – Subscrição
  - Classe ***Editora*** : gera um evento para avisar os objectos interessados (os subscritores);
  - Classe ***Subscritora*** : fornece um método que é invocado quando é gerado um evento

- A rotina invocada por um evento é um delegate:

```
public delegate void MeuDelegate( );
public event MeuDelegate evt;
```

- Têm permissões diferentes consoante quem manipula:
  - Classes subscritoras só podem usar += e -=

Convenção para Eventos

- Convenção para um delegate subscritor:
  - Devolvem **void**
  - Recebem dois argumentos:
    - 1º: o objecto que gerou o evento
    - 2º: uma instância de uma subclasse de EventArgs

```
public class MeusEventArgs: EventArgs {
   private int a;
   public MeusEventArgs(int a) {
       this. a = a;
   }
   public int A { get {return a;} }
}
public delegate void MeuSubs(object sender, MeusEventArgs a);
public event MeuSubs E;
```

Convenção para Eventos (2)

- Pode-se modificar *como* adicionar ou remover subscritores do evento
  - Tal como as Properties têm o get/set, os eventos têm o **add/remove**
    - Requer a utilização explícita de um delegate para guardar os subscritores
  - Permite fazer validações sobre os subscritores a adicionar/remover
    - E.g., para evitar que um subscritor seja adicionado 2 vezes

```
public delegate void MeuSubs(object sender, MeusEventArgs a);

private MeuSubs _e; // delegate com lista de subscritores
public event MeuSubs E
{
   add { _e += value; }
   remove { if(_e != null) { _e -= value; } }
}
```

Subscrever um Evento

```
public class MyClass {
   public void Callback(object sender, MeusEventArgs e) {
        Console.Writeline("Fired {0}", e.A);
   }
}
...
MyClass c = new MyClass();
E += new MeuSubs(c.Callback);
```

Notificar Subscritores

```
public void DispararEvento( ) {
   if (E != null)
      E(this, new MeusEventArgs(0));
}
```

Nota: *É verificado se existe pelo menos um delegado subscritor do evento. Se for gerado um evento para o qual não haja subscritores é gerada uma excepção.*

4. Generics (C# 2.0)

Colecções

Classes

Métodos

Generics (C# 2.0)

- Permitem a definição de estruturas **fortemente tipificadas**

- Em vez de

```
ArrayList list = new ArrayList();
list.Add(1);
int i = (int) list[0];
```

podemos ter

```
List<int> list = new List<int>();
list.Add(1);
int i = list[0]; // não é preciso fazer cast
```

Generics (classes)

- O programador pode criar uma classe genérica:

```
public class MinhaLista<T> { // T é um qualquer tipo
   T[] m_Items;

   public MinhaLista ():this(100) {}
   public MinhaLista (int size) { m_Items = new T[m_Size]; }
   public void Adiciona(T item) { … }
   public T Remove(int index) { … }
   public T Obtem(int index) { … }
}
```

- E utilizá-la:

```
MinhaLista<char> caracteres = new MinhaLista<char>(10);
caracteres.Adiciona('c');
char caracter = caracteres.Obtem(0); // não é preciso cast
```

Generics (métodos)

- O programador pode também definir métodos com Generics:

```
public class UmaClasse {
   public T Add<T>(T item) { … }
}
```

- O tipo usado é inferido aquando da compilação do código que invoca o método
- Utilização desses métodos:

```
UmaClasse classe = new UmaClasse();
classe.Add('c'); // Add(char)
Classe.Add(6); // Add(int)
```

5. Threads e Monitores

Threads

- Quando se usam threads:
  - Várias tarefas simultâneas
  - Suportam partilha de dados

- Construção:

```
//ThreadStart é um public delegate void ThreadStart();
ThreadStart ts = new ThreadStart(y.xpto);
Thread t = new Thread(ts);
t.Start(); // inicia execução
t.Join(); //  espera terminação
```

Threads (cont.)

- Outros métodos: Abort, Sleep, Join

```
using System;
using System.Threading;
public class Alpha
{
   public void Beta()
   {
      while (true)
      {
         Console.WriteLine("A.B is running in its own thread.");
      }
   }
};
```

Threads (cont.)

```
public class Simple
{
   public static int Main()
   {
      Alpha oAlpha = new Alpha();
      Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));
      oThread.Start();

  // Spin for a while waiting for the started thread to become alive:
      while (!oThread.IsAlive);

  // Put the Main thread to sleep for 1 ms to allow oThread to work:
      Thread.Sleep(1);

  // Request that oThread be stopped
      oThread.Abort();
   }
}
```

Sincronização

- Concorrência (*threads*) implica sincronização.
- `lock` permite exclusão mútua num bloco.
- Duas variantes:
- `lock(this)` exclusão mútua para aceder a um objecto
- `lock(typeof(this))`: exclusão mútua em relação aos métodos estáticos de uma classe

Sincronização (cont.)

- Monitores:
  - Monitor.Enter(this) [Equivalente a lock(this)]
    - Obtém o lock exclusivo sobre o objecto actual (*this*)
  - Monitor.Wait(this)
    - Liberta o lock sobre o objecto actual (*this*) e fica à espera de obter novo lock sobre esse objecto actual
  - Monitor.Pulse(this)
    - Notifica a próxima thread que está à espera de obter um lock sobre o objecto actual (*this*)
  - Monitor.PulseAll(this)
    - Notifica *todas* as threads que estão à espera de obter um lock sobre o objecto actual (*this*)

- Leitura recomendada:
  MSDN (http://msdn2.microsoft.com/en-au/library/system.threading.monitor.pulse.aspx)

Sincronização (WinForms)

- É preciso ter em atenção que o modelo de muitos sistemas de janelas (incluíndo o WinForms) não permite que controlos de UI sejam directamente manipulados por uma thread que não seja a que criou o controlo (típicamente a *UI thread*)
  - Irrelevante quando a aplicação só usa 1 thread
  - Quando a aplicação usa mais que 1 thread, devem usar:
    - Control.InvokeRequired
      - Devolve um booleano que indica se a thread actual pode manipular directamente o controlo
    - Control.Invoke(Delegate)
      - A thread em que o controlo foi creado irá invocar (de modo *síncrono*) o delegate passado como argumento
    - Control.BeginInvoke(Delegate)
      - A thread em que o controlo foi creado irá invocar (de modo *assíncrono*) o delegate passado como argumento

- Leitura recomendada:

  http://www.codeproject.com/csharp/begininvoke.asp