Curso de Extensão

Tecnologias Microsoft

UNICAMP

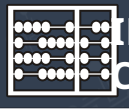
INF-0990

Programação em C#
Aula 3

Prof. Dr. Ricardo Ribeiro Gudwin

gudwin@unicamp.br

10 de Setembro de 2022



INSTITUTO DE COMPUTAÇÃO

Recursos Avançados - C#

Generics

- Possibilidades de Reuso no C#
  - Herança: Reuso baseado em tipo
  - Generics: Reuso baseado em *template*
- Tipos Genéricos
  - Um tipo (classe) genérico(a) é descrito utilizando-se placeholders <T> para serem substituidos por um outro tipo genérico, criando assim, diversas classes simultaneamente, onde a única diferença entre elas é esse tipo <T>
  - Exemplo:

```
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    public void Push (T obj) => data[position++] = obj;
    public T Pop() => data[--position];
}
```

  - T pode ser substituído por qualquer outro tipo: int, double, string, OutraClasse, gerando assim, uma coleção de tipos correlatos:
    - Stack<int>, Stack<double>, Stack<string>, Stack<OutraClasse>,...
    - Tipicamente utilizado com arrays
    - Podem haver mais de um placeholders <T,S>

Generics

- Métodos Genéricos
  - Além de classes genéricas, podem existir também **métodos** genéricos:

```
static void Swap<T> (ref T a, ref T b)
{ T temp = a;
   a = b;
   b = temp;
}
```

  - Para chamar um método genérico, não há a necessidade de indicar o placeholder, caso o tipo esteja definido:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Delegates

- Delegates
  - No C#, métodos podem ser passados como parâmetros para outros métodos
    - O **tipo** de um parâmetro como esse é um **delegate**
- Assinatura do Delegate
  - Descreve os **tipos dos parâmetros** que o método a ser passado como parâmetro deve ter e retornar
  - Declara um **nome** para o delegate, a ser usado como tipo, para parâmetros desse tipo

```
delegate int Transformer (int x);
```

  - Uma vez declarado, um delegate pode ser usado como tipo para um método que será passado como parâmetro
  - Exemplo:

```
int Square (int x) => x * x;
int Cube (int x) => x * x * x;
Transformer t = Square;
Transformer u = Cube;
int s = t(3);
int c = u(3);
```

Delegates

- Concatenação de Delegates
  - Delegates podem ser concatenados em cadeias de delegates, usando-se os operadores + e +=

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

  - Os operadores – e -= podem ser usados para “desconcatenar” um delegate concatenado

```
d -= SomeMethod1;
```

- Delegates Genéricos

```
public delegate T Transformer<T> (T arg);
```

  - Com delegates genéricos, é possível criar delegates que podem mapear em virtualmente qualquer método
  - Os tipos `Func` e `Action` do pacote `System`
    - São delegates genéricos

Eventos

- Eventos
  - Um *design-pattern* muito comum em engenharia de software é o *publish-subscribe*
    - Nesse *pattern*, alguns objetos publicam eventos, que podem ser de interesse de outros objetos, que então subscrevem-se para recebê-lo, quando o mesmo ocorrer
  - Esse *pattern* pode ser implementado em C# usando-se *delegates*
    - Entretanto, o uso de *delegates* permitiria que os *subscribers* fizessem modificações nos *delegates*, que poderiam comprometer o mecanismo
    - Por esse motivo, o C# criou o tipo `event`, que funciona como um *delegate*, mas possui proteção contra essa interferência inadequada
  - Eventos podem simplesmente indicar que alguma coisa aconteceu, ou podem incluir argumentos, que podem ser passados como parâmetros
  - Classes que publicam eventos devem:
    - Declarar os eventos que publicam usando o tipo `event`
    - Criar métodos que sejam `protected` e `virtual` com tipicamente um nome `onNomeEvento`, que invocam o evento, quando necessário, durante seu funcionamento
  - Classes que desejam se subscrever a um evento podem fazê-lo usando o mecanismo de concatenção de delegates: `+=`
    - Para deixar de receber esses eventos, é só usar o `-=`

Eventos

- A Classe `System.EventArgs`
  - Classe padrão que encapsula os parâmetros passados para um evento
  - Quando desejamos passar parâmetros para um evento, devemos criar uma classe que estende a classe `EventArgs`
- O delegate `System.EventHandler<TEventArgs>`
  - É o `event` padronizado para indicar que uma classe publica eventos
    - O genérico `TEventArgs` deve ser uma subclasse de `EventArgs`
- O *publisher* do evento, deve definir o evento e um método que o invoca:

```
public event EventHandler<TEventArgs> nomeEvento;

protected virtual void onNomeEvento(TEventArgs e)
{
    EventHandler<TEventArgs> handler = nomeEvento;
    handler?.Invoke(this, e);
}
```

  - Deve ainda chamar `onNomeEvento()` quando necessário em seu funcionamento
- O *subscriber* do evento, deve criar um método de *callback* para atribuir ao delegate da classe que o publica, e que será executado quando o evento for publicado
  - Esse método de callback deve ter a assinatura:
    - `void nomeCallback(object sender, TEventArgs e)`
  - E deve subscrever-se ao evento, concatenando o delegate `nomeEvento` com esse método de callback
    - `ObjetoQuePublica.NomeEvento += nomeCallback;`

Expressões Lambda

- Expressões Lambda
  - São um modo alternativo de definir uma função, por meio de uma expressão que utiliza o operador Lambda (=>)
  - Podem ser utilizadas para definir, de maneira sintética, métodos e delegates de funções simples
- Operador Lambda
  - Separa uma função dentre:
    - Parâmetros de entrada
    - Corpo da Função
- Sintaxe
  - `(input-parameters) => expression`
  - `(input-parameters) => { <sequence-of-statements> }`
  - Se houver somente um único parâmetro, os parêntesis dos parâmetros de entrada podem ser omitidos

Expressões Lambda

- Uso Típico
  - Considera os delegates genéricos `Func` e `Action` do pacote System
  - Exemplos:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5)); // Output: 25
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World"); // Output: Hello World!
```

  - Normalmente, o compilador consegue inferir o tipo dos parâmetros em uma expressão Lambda. Quando isso não for possível, é necessário especificar os tipos de cada parâmetro explicitamente
    - Exemplo:

```
var sqr = int (int x) => x;
```

Expressões Lambda

- Capturando Variáveis Externas
  - É possível utilizar variáveis definidas externamente a uma expressão lambda
  - Exemplo:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3));  // 6
```

- Lambdas com Variáveis Estáticas
  - Em alguns casos, para prevenir uma captura indesejável de uma variável externa, pode-se usar a keyword `static` para forçar a criação de variáveis estáticas internas à expressão
  - Exemplo:

```
Func<double, double> square = static x => x * x;
```

  - Quando o modificador `static` for utilizado, a expressão lambda não conseguirá capturar variáveis externas

Exceções

- Exceções
  - Código C# pode gerar exceções durante seu funcionamento
  - O tratamento dessas exceções pode ser realizado por meio de blocos do tipo **try-catch**

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

  - Todas as exceções são subclasses de `System.Exception`
    - Com isso, tanto pode-se tratar especificamente cada tipo de exceção, quanto se ter um único bloco catch, para a exceção `Exception`
  - O bloco `finally` SEMPRE executa, independentemente de ter havido ou não uma exceção
  - Para explicitamente se gerar uma exceção, utiliza-se o comando **throw**

```
throw new MyException();
```

Métodos de Extensão

- Métodos de Extensão
  - São um mecanismo do C# que permite que um tipo pré-existente seja estendido com novos métodos, sem ser necessária a alteração do tipo original
  - Corresponde a métodos estáticos encapsulados dentro de uma classe estática, que é usada para estender um tipo pré-existente
    - o modificador this é aplicado ao primeiro parâmetro de cada um desses métodos
    - O tipo desse primeiro parâmetro é o tipo que é estendido
    - O compilador executa a extensão automaticamente
  - Exemplo:

```
public static class StringHelper
{
    public static bool IsCapitalized (this string s)
    {
     if (string.IsNullOrEmpty(s)) return false;
     return char.IsUpper (s[0]);
    }
}
Console.WriteLine ("Perth".IsCapitalized());
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

Tuplas

- Tuplas
  - São uma maneira de agregar várias variáveis em uma só, principalmente para poder realizar um retorno múltiplo de um método, sem que seja necessário criar-se um tipo específico para essa finalidade
  - Cada elemento de uma tupla pode ser acessado individualmente, utilizando-se o operador `.ItemN`
  - Exemplos

```
var bob = ("Bob", 23);
Console.WriteLine (bob.Item1);  // Bob
Console.WriteLine (bob.Item2);  // 23

(string,int) GetPerson() => ("Bob", 23);

(string,int) person = GetPerson();
Console.WriteLine (person.Item1);
Console.WriteLine (person.Item2);

var tuple = (name:"Bob", age:23); // Elementos nomeados
Console.WriteLine (tuple.name); // Bob
Console.WriteLine (tuple.age);  // 23
```

Pattern

- Patterns
  - Em muitas situações, pode ser conveniente testar se um objeto segue ou não um determinado padrão
  - O C# possui o operador is, que verifica se um objeto ou variável segue ou não um padrão
    - Tipos de padrões
      - Constantes: `if (obj is 3)`
      - Padrões de Tipo: `if (obj is string)`
      - Padrões Relacionais: `if (x is > 100)`
      - Padrões de Propriedades: `if (obj is string { Length:4 })`
  - Outra possibilidade é testar padrões usando:
    - Expressões switch
    - Statements switch

Atributos

- Atributos
  - Também chamados de anotações, são um mecanismo de extensão para adicionar informação customizada a elementos de código (assemblies, tipos, membros, valores de retorno, parâmetros e tipos genéricos de parâmetros)
  - Essa extensibilidade é útil para serviços que se integram ao código, sem que seja necessário keywords especializadas na linguagem C#
    - O uso de atributos pode ser particularmente útil quando se utiliza frameworks, tais como e.g. o ASP.NET, que espera que certos atributos sejam definidos, de acordo com as especificações do framework
  - Atributos são classes que estendem a classe `Attribute`, e devem adotar um nome padronizado `NNNAttribute`, onde `NNN` pode ser qualquer coisa
  - Atributos podem ser adicionados, utilizando-se a notação `[Atributo]`, imediatamente antes do elemento ao qual se deseja anotar
    - E.g. o atributo `Obsolete` é definido pela classe `ObsoleteAttribute`, que estende a classe `Attribute`, e pode ser anotado, por exemplo, com o seguinte código:

```
[Obsolete] public class Foo {...}
```

Atributos

- Atributos
  - A Base Class Library do .NET pré-define um grande número de atributos
    - Novos atributos podem ser criados
  - Vários atributos podem ser anotados simultaneamente

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

  - Podem ter parâmetros
    - Podem ser posicionais ou nomeados

```
[XmlType ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }

[assembly: AssemblyFileVersion ("1.2.3.4")]
```

Overload de Operadores

- Da mesma forma que métodos
  - Em C# operadores como `+ (unary), - (unary),!,~,++,--,+, -,*,/,%,&, |,^,<<,>>,==,!=,>, <,>=,<=` podem sofrer overload, e serem redefinidos para funcionarem com tipos definidos pelo programador
  - Exemplos

```
public static Note operator + (Note x, int semitones){ … }
public static Note operator + (Note x, Note y){ … }
```