PEF – 3528 – Ferramentas Computacionais na Mecânica das Estruturas: Criação e Concepção

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Introdução ao C#

Parte III – Propriedades

Propriedades

- Considere o seguinte exemplo:

```
struct ScreenPosition
{
    public int X; public int Y;
    public ScreenPosition(int x, int y)
    {
        this.X = rangeCheckedX(x);
        this.Y = rangeCheckedY(y);
    }
    private static int rangeCheckedX(int x)
    {
        if (x < 0 || x > 1280)
        {
            throw new ArgumentOutOfRangeException("X");
        }
        return x;
    }
    private static int rangeCheckedY(int y)
    {
        if (y < 0 || y > 1024)
        {
            throw new ArgumentOutOfRangeException("Y");
        }
        return y;
    }
}
```

Propriedades

- Como pode se observar, as propriedades não estão encapsuladas (*private*) dentro da classe. Isso é um problema pois não se tem o controle de como e onde as variáveis serão alteradas:

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // oops
```

Properties

- Como prática, implementa-se funções *get* e *set*, nas quais se modifica a variável (dentro da classe) com as variáveis *private:*

```
struct ScreenPosition
{ …
public int GetX() { return this.x; }
public void SetX(int newX) { this.x = rangeCheckedX(newX); }
…
private static int rangeCheckedX(int x) { … }
private static int rangeCheckedY(int y) { … }
private int x, y;
}
```

Properties

- Assim, o código fica:

```
int xpos = origin.GetX();
origin.SetX(xpos + 10);
```

- Ao invés de:

```
origin.X += 10;
```

Properties

- Um propriedade no C# é um híbrido entre método e variável: parece uma variável mas funciona como um método.
- Ela implementa operadores de acesso à variável (*get* e *set*).
- A sintaxe é:

```
AccessModifier Type PropertyName
{
    get
    {
        // read accessor code
    }
    set
    {
        // write accessor code
    }
}
```

Properties

```
struct ScreenPosition
{
        private int _x, _y;
        public ScreenPosition(int X, int Y)
        {
                this._x = rangeCheckedX(X);
                this._y = rangeCheckedY(Y);
        }
        public int X
        {
                get { return this._x; }
                set { this._x = rangeCheckedX(value); }
        }
        public int Y
        {
                get { return this._y; }
                set { this._y = rangeCheckedY(value); }
        }
        private static int rangeCheckedX(int x) { … }
        private static int rangeCheckedY(int y) { … }
}
```

Usando as propriedades

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X;      // calls origin.X.get
int ypos = origin.Y;      // calls origin.Y.get

origin.X = 40;            // calls origin.X.set, with value set to 40
origin.Y = 100;           // calls origin.Y.Set, with value set to 100

origin.X += 10;
```

Propriedades

- Apenas leitura:

```
struct ScreenPosition
{
        private int _x;
        ...
        public int X
        {
                get { return this._x; }
        }
}
```

- Apenas gravação:

```
struct ScreenPosition
{
        private int _x;
        ...
        public int X
        {
                set { this._x = rangeCheckedX(value); }
        }
}
```

Acesso nas propriedades

```
struct ScreenPosition
{
        private int _x, _y;
        ...
        public int X
        {
                get { return this._x; }
                private set { this._x = rangeCheckedX(value); }
        }
        public int Y
        {
                get { return this._y; }
                private set { this._y = rangeCheckedY(value); }
        }
        ...
}
```

Gerando propriedades automáticas

```
class Circle
{
    public int Radius{ get; set; }
    ...
}
```

- Equivalente a:

```
class Circle
{
    private int _radius;
    public int Radius
    {
        get { return this._radius; }
        set {this._radius = value; }
    }
    ...
}
```

Indexadores (*Indexers*)

- Indexadores são um tipo especial de propriedade que acessam um conjunto de valores e retorna / seta um deles.

- Imagine um vetor de inteiros:

```
int[] bits = new int[5];
int[3] = 10;
```

Indexadores

- É possível encapsular um *array* desses em uma estrutura:

```
struct IntVec
{
        private int[] intVector;
        public IntVec(int[] initialValue) { intVector = Array.CopyTo(initialBitValue,0); }
        // indexer to be written here
}
```

Indexadores

- O indexador para tal classe pode ser escrito como:

```
struct IntVec
{ ...
        // indexer to be written here
        public int this [index]
        {
                get { return intVector[index]; }
                set { intVector[index] = value; }
        }
}
```

Introdução ao C#

Parte III – *Generics* e Coleções

Generics

- *Generics* é a solução para o seguinte problema:
  - Imagine que deseja se construir uma lista ligada ou uma fila, por exemplo, que armazene inteiros.
  - Agora, imagine a mesma estrutura armazenando qualquer coisa (pode-se usar *object* com esse intuito).
  - Se se desejar armazenar inteiros nessa nova lista será possível mas, para uso, será necessário, em todas as vezes que se requisitar um resultado, fazer o *unboxing* da variável.

Generics

- Sendo assim, encontrou-se a solução dos *Generics*:
  - Melhoram a segurança
  - Eliminam o *casting* (*boxing* e *unboxing*)
  - Facilitam a criação de classes genéricas

Generics

- A maneira de definir uma classe genérica é colocando um parâmetro genérico entre < e > :

```
class Queue<T>
{
    ...
}
```

Generics

- A implementação da classe fica:

```
class Queue<T>
{...
        private T[] data; // array is of type 'T' where 'T' is the type parameter
        ...
        public Queue()
        {
                this.data = new T[DEFAULTQUEUESIZE]; // use 'T' as the data
type
        }
        public Queue(int size)
        {...
                this.data = new T[size];
        ...}

public void Enqueue(T item) // use 'T' as the type of the method parameter
        {...}
        public T Dequeue() // use 'T' as the type of the return value
        {...
                T queueItem = this.data[this.tail]; // the data in the array is of type
'T'
                ...
                return queueItem;
        }
}
```

Generics

- Assim, se se desejar criar filas com tipos diferentes, utiliza-se:

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();
```

Generics

- Os tipos podem ser outros tipos que tenham tipos genéricos:

```
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

Restrições

- Na definição de uma classe pode-se restringir os tipos que podem ser utilizados na classe, por exemplo:

public class PrintableCollection<T> where T : IPrintable

Métodos Genéricos

- Além de classes genéricas, pode-se criar métodos genéricos:

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

Métodos Genéricos

- Seu uso fica:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```

Coleções

Collection	Description
List<T>	A list of objects that can be accessed by index, like an array, but with additional methods to search the list and sort the contents of the list.
Queue<T>	A first-in, first-out data structure, with methods to add an item to one end of the queue, remove an item from the other end, and examine an item without removing it.
Stack<T>	A first-in, last-out data structure with methods to push an item onto the top of the stack, pop an item from the top of the stack, and examine the item at the top of the stack without removing it.
LinkedList<T>	A double-ended ordered list, optimized to support insertion and removal at either end. This collection can act like a queue or a stack, but it also supports random access like a list.
HashSet<T>	An unordered set of values that is optimized for fast retrieval of data. It provides set-oriented methods for determining whether the items it holds are a subset of those in another *HashSet<T>* object as well as computing the intersection and union of *HashSet<T>* objects.
Dictionary<TKey, TValue>	A collection of values that can be identified and retrieved by using keys rather than indexes.
SortedList<TKey, TValue>	A sorted list of key/value pairs. The keys must implement the *IComparable<T>* interface.

Coleções

- Há uma forma simples de inicializar a coleção sem adicionar elementos através dos métodos (*Add*, *Append*):

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

```
Dictionary<string, int> ages = new Dictionary<string, int>(){{"John", 44},
{"Diana", 45}, {"James", 17}, {"Francesca", 15}};
```

Método de busca

- Coleções permitem que sejam realizadas buscas utilizando a função *Find*.
- Para utilizar tal função é necessário o uso de um recurso denominado *Lambda Expressions*.

Método de busca

- Essas expressões são expressões simples que retornam *true* ou *false* e que, em caso verdadeiro, setam o resultado da busca.

- Veja o exemplo a seguir:

Método de busca

```
struct Person
{
        public int ID { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
}
...
// Create and populate the personnel list
List<Person> personnel = new List<Person>()
{
        new Person() { ID = 1, Name = "John", Age = 47 },
        new Person() { ID = 2, Name = "Sid", Age = 28 },
        new Person() { ID = 3, Name = "Fred", Age = 34 },
        new Person() { ID = 4, Name = "Paul", Age = 22 },
};
// Find the member of the list that has an ID of 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });
Console.WriteLine("ID: {0}\nName: {1}\nAge: {2}", match.ID, match.Name, match.Age);
```

Método de busca

- A saída desse programa é:

ID: 3

Name: Fred

Age: 34

Introdução ao C#

Parte III – Eventos

Delegates

- Um *delegate* é um tipo referência porém aponta para métodos e não variáveis.

- Isso é o uso de um método:

```
Processor p = new Processor();
p.performCalculation();
```

Delegates

- Agora, observe o uso de um *delegate*:

```
Processor p = new Processor();
delegate ... performCalculationDelegate ...;
performCalculationDelegate = p.performCalculation;
```

- E para invocar a função:

```
performCalculationDelegate();
```

Delegates

- Considere a seguinte classe:

```
class Controller
{
        // Fields representing the different machines
        private FoldingMachine folder;
        private WeldingMachine welder; private
PaintingMachine painter;
        …
        public void ShutDown()
        {
                folder.StopFolding();
                welder.FinishWelding();
                painter.PaintOff();
        }
        …
}
```

Delegates

- Pode-se implementar a mesma classe usando *delegate*:

```
class Controller
{
        delegate void stopMachineryDelegate(); // the delegate type
        private stopMachineryDelegate stopMachinery; // an instance of the delegate
        …
        public Controller()
        {
                this.stopMachinery += folder.StopFolding;
        }
        …
}
```

Delegates

- Para invocar o método utiliza-se:

```
public void ShutDown()
{
    this.stopMachinery();
    …
}
```

Delegates

- É importante notar o operador += que permite a adição de mais de um método ao *delegate*:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Delegates

- Para remover um método do *delegate* a sintaxe é simples:

this.stopMachinery -= folder.StopFolding;

Delegates

- É importante observar que a assinatura do *delegate* e da função devem ser as mesmas. Mas, e se se desejar adicionar algo com assinatura diferente?

```
this.stopMachinery += folder.StopFolding;
void StopFolding(int shutDownTime); // Shut down in the specified number of seconds
```

Delegates

- Uma solução é criar uma função que adapta o problema:

```
void FinishFolding()
{
    folder.StopFolding(o); // Shut down immediately
}
```

Delegates

- Uma segunda solução é usar *lambda expressions*:

this.stopMachinery += (() => folder.StopFolding(o));

Lambda Expressions

- São formas de usar *lambda expressions:*

```
x => x * x       // A simple expression that returns the square of its parameter
                 //The type of parameter x is inferred from the context.

x => { return x * x ; }    // Semantically the same as the preceding
                           // expression, but using a C# statement block as
                           // a body rather than a simple expression
```

Lambda Expressions

```
(int x) => x / 2        // A simple expression that returns the value of the
                        // parameter divided by 2
                        //The type of parameter x is stated explicitly.

() => folder.StopFolding(o) // Calling a method
                            //The expression takes no parameters.
                            //The expression might or might not
                            // return a value.
```

Lambda Expressions

```
(x, y) => { x++; return x / y; }  // Multiple parameters; the compiler infers the parameter types.
                                  //The parameter x is passed by value, so the effect of the ++
                                  // operation is local to the expression.

(ref int x, int y) => { x++; return x / y; }      // Multiple parameters with explicit types
                                                  // Parameter x is passed by reference,
                                                  // so the effect of the ++ operation is permanent.
```

Eventos

- Eventos são responsáveis por disparar *delegates* quando algum fato acontece durante a execução do programa.
- O funcionamento é simples: algo dispara o evento e, nesse caso, o evento chama todas as funções associadas a ele.
- Isso é muito utilizado em aplicações gráficas (como o WPF que será visto).

Eventos

- Declara-se o evento da seguinte maneira:

event delegateTypeName eventName

- Assim é possível escrever:

```
class TemperatureMonitor
{
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        …
}
```

Eventos

```
class TemperatureMonitor
{
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        ...
}
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += (() => { folder.StopFolding(o); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

Eventos

- Da mesma forma que *delegates*, subscreve-se para um evento usando o operador += e se cancela a subscrição com o operador -=

Invocando um evento (*raising an event*)

```
class TemperatureMonitor
{
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        …
        private void Notify()
        {
                if (this.MachineOverheating != null) { this.MachineOverheating(); }
        }
        …
}
```

Introdução ao C#

Parte III – LINQ

Language INtegrated Query (LINQ)

- LINQ é uma maneira de realizar pesquisas (*queries*) em entidades na memória (como vetores, listas, dicionários, etc).

- Considere por exemplo os seguintes dados:

Language INtegrated Query (LINQ)

Customer Information

CustomerID	FirstName	LastName	CompanyName
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers

Address Information

CompanyName	City	Country
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

LINQ

```
var customers = new[] {
        new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",
              CompanyName = "Alpine Ski House" },
        new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",
              CompanyName = "Coho Winery" },
        new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",
              CompanyName = "Alpine Ski House" },
        new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",
              CompanyName = "Trey Research" },
        new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",
              CompanyName = "Wingtip Toys" },
        new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",
              CompanyName = "Coho Winery" },
        new { CustomerID = 7, FirstName = "John", LastName = "Kane",
              CompanyName = "Wingtip Toys" },
        new { CustomerID = 8, FirstName = "David", LastName = "Simpson",
              CompanyName = "Trey Research" },
        new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",
              CompanyName = "Wingtip Toys" },
        new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",
              CompanyName = "Wide World Importers" }
    };
```

LINQ

```
var addresses = new[] {
        new { CompanyName = "Alpine Ski House", City = "Berne", Country = "Switzerland"},
        new { CompanyName = "Coho Winery", City = "San Francisco", Country = "United States"},
        new { CompanyName = "Trey Research", City = "New York", Country = "United States"},
        new { CompanyName = "Wingtip Toys", City = "London", Country = "United Kingdom"},
        new { CompanyName = "Wide World Importers", City = "Tetbury", Country = "United Kingdom"}
};
```

LINQ

- Se deseja-se imprimir na tela o primeiro nome do consumidor pode-se fazer:

```
IEnumerable<string> customerFirstNames = customers.Select(cust => cust.FirstName);
foreach (string name in customerFirstNames)
{
    Console.WriteLine(name);
}
```

LINQ

- A saída fica:

Kim

Jeff

Charlie

Chris

Dave

Ann

John

David

Greg

Tim

- Observe o comando usado (*Select*)

LINQ

- E se desejar-se saber o nome e sobrenome dos clientes?

```
IEnumerable<string> customerNames =
customers.Select(cust => String.Format("{0} {1}", cust.FirstName,
cust.LastName));
```

LINQ

- Ainda é possível criar uma classe e alimentar esses dados:

```
class FullName
{
        public string FirstName{ get; set; }
        public string LastName{ get; set; }
}
...
IEnumerable<FullName> customerNames = customers.Select(cust => new FullName { FirstName = cust.FirstName, LastName = cust.LastName } );
```

LINQ

- Com o LINQ é possível filtrar dados:

```
IEnumerable<string> usCompanies = addresses.Where(addr => String.Equals(addr.Country,
"United States")).Select(usComp => usComp.CompanyName);

foreach (string name in usCompanies)

{

    Console.WriteLine(name);

}
```

LINQ

- Ainda é possível ordernar, agrupar e agregar dados.

```
IEnumerable<string> companyNames = addresses.OrderBy(addr =>
addr.CompanyName).Select(comp => comp.CompanyName);

foreach (string name in companyNames)

{

    Console.WriteLine(name);

}
```

LINQ

```
var companiesGroupedByCountry = addresses.GroupBy(addrs => addrs.Country);
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
        Console.WriteLine("Country: {0}\t{1} companies", companiesPerCountry.Key, companiesPerCountry.Count());
        foreach (var companies in companiesPerCountry)
        {
                Console.WriteLine("\t{0}", companies.CompanyName);
        }
}
```

LINQ

```
var companiesGroupedByCountry = addresses.GroupBy(addrs => addrs.Country);
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
        Console.WriteLine("Country: {0}\t{1} companies", companiesPerCountry.Key, companiesPerCountry.Count());
        foreach (var companies in companiesPerCountry)
        {
                Console.WriteLine("\t{0}", companies.CompanyName);
        }
}
```

LINQ

```
var companiesAndCustomers = customers .Select(c => new { c.FirstName, c.LastName,
c.CompanyName }) .Join(addresses, custs => custs.CompanyName, addrs =>
addrs.CompanyName, (custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country
});


foreach (var row in companiesAndCustomers)
{
        Console.WriteLine(row);
}
```

LINQ

{ FirstName = Kim, LastName = Abercrombie, Country = Switzerland }

{ FirstName = Jeff, LastName = Hay, Country = United States }

{ FirstName = Charlie, LastName = Herb, Country = Switzerland }

{ FirstName = Chris, LastName = Preston, Country = United States }

{ FirstName = Dave, LastName = Barnett, Country = United Kingdom }

{ FirstName = Ann, LastName = Beebe, Country = United States }

{ FirstName = John, LastName = Kane, Country = United Kingdom }

{ FirstName = David, LastName = Simpson, Country = United States }

{ FirstName = Greg, LastName = Chapman, Country = United Kingdom }

{ FirstName = Tim, LastName = Litton, Country = United Kingdom }

Operadores de pesquisa

- Uma alternativa ao uso de *lambda expressions* é o uso dos operadores de pesquisa.
- Cabe ressaltar que certas operações mais complexas só são possíveis de serem executadas usando o que foi visto anteriormente.

Operadores de pesquisa

var customerFirstNames = from cust in customers select cust.FirstName;

var customerNames = from cust in customers select new { cust.FirstName, cust.LastName };

var usCompanies = from a in addresses where String.Equals(a.Country, "United States") select a.CompanyName;

Operadores de pesquisa

```
var companyNames = from a in addresses orderby a.CompanyName select a.CompanyName;

var companiesGroupedByCountry = from a in addresses group a by a.Country;

var countriesAndCustomers = from a in addresses
                            join c in customers
                            on a.CompanyName equals c.CompanyName
                            select new { c.FirstName, c.LastName, a.Country };
```

Introdução ao C#

Parte III – *Operator Overloading*

Operator Overloading

- É possível definir para objetos operadores.
- Isso é denominado *operator overloading*.

Operator Overloading

- Por exemplo, pode-se definir o seguinte conjunto de operações para um número complexo:

Operation	Calculation
(a + bi) + (c + di)	((a + c) + (b + d)i)
(a + bi) – (c + di)	((a – c) + (b – d)i)
(a + bi) * (c + di)	((a * c – b * d) + (b * c + a * d)i)
(a + bi) / (c + di)	(((a * c + b * d) / (c * c + d * d)) + ((b * c - a * d) / (c * c + d * d))i)

Operator Overloading

- Define-se um operador de soma como:

```
class Complex
{
        ...
        public static Complex operator +(Complex lhs, Complex rhs)
        {
                return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
        }
}
```

Operator Overloading

```
public static Complex operator -(Complex lhs, Complex rhs)
{
        return new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
}

public static Complex operator *(Complex lhs, Complex rhs)
{
        return new Complex(lhs.Real * rhs.Real - lhs.Imaginary * rhs.Imaginary, lhs.Imaginary * rhs.Real + lhs.Real * rhs.Imaginary);
}
public static Complex operator /(Complex lhs, Complex rhs)
{
        int realElement = (lhs.Real * rhs.Real + lhs.Imaginary * rhs.Imaginary) / (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);
        int imaginaryElement = (lhs.Imaginary * rhs.Real - lhs.Real * rhs.Imaginary) / (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);
        return new Complex(realElement, imaginaryElement);
}
```