FACULDADE DOM BOSCO de Porto Alegre

Paradigmas de Linguagem de Programação

Prof. Filipo Mór

2019/II - www.filipomor.com

Conceitos de Programação Orientada a Objetos

INTRODUÇÃO AO .NET FRAMEWORK E CLR

O que é o .NET Framework?

- O .NET Framework é um modelo de programação de código gerenciado da Microsoft para criar aplicativos em clientes, servidores e dispositivos móveis ou incorporados ao Windows.

- Conjunto rico de bibliotecas com os mais variados usos;
- Controle de versão: fim do “DLL Hell”;
- Facilidade de desenvolvimento de aplicações desde as mais simples até as mais complexas;
- Facilidade na instalação e na distribuição de aplicações;
- Alta escalabilidade para ambientes de missão crítica;
- Interoperabilidade entre plataformas e componentes desenvolvidos em outras linguagens .NET;
- Sintonizado com as últimas tecnologias;
- Orientado a objetos;
- Tecnologia baseada em máquina virtual;

O que posso fazer com o .NET?

- O .NET permite desenvolver soluções como:
  - Aplicativos Web
  - Aplicativos para Servidores
  - Aplicativos para Windows
  - Aplicativos para Windows Phone
  - Aplicativos de Banco de Dados
  - Serviços Windows
  - Web Services
  - e muito mais

O que é CLR?

- O Common Language Runtime (CLR) é o componente encarregado de gerenciar aplicações desenvolvidas em .NET.
- O compilador de cada linguagem segue uma série de especificações, conhecidas como Common Language Infrastructure (CLI).
- Estas especificações são abertas (ECMA-335, ISO/IEC 23271), assim permitem a interoperabilidade entre outras linguagens e plataformas.
  - Ex. Mono é um CLR que opera em Linux, BSD, UNIX, Mac OS X e Solaris.

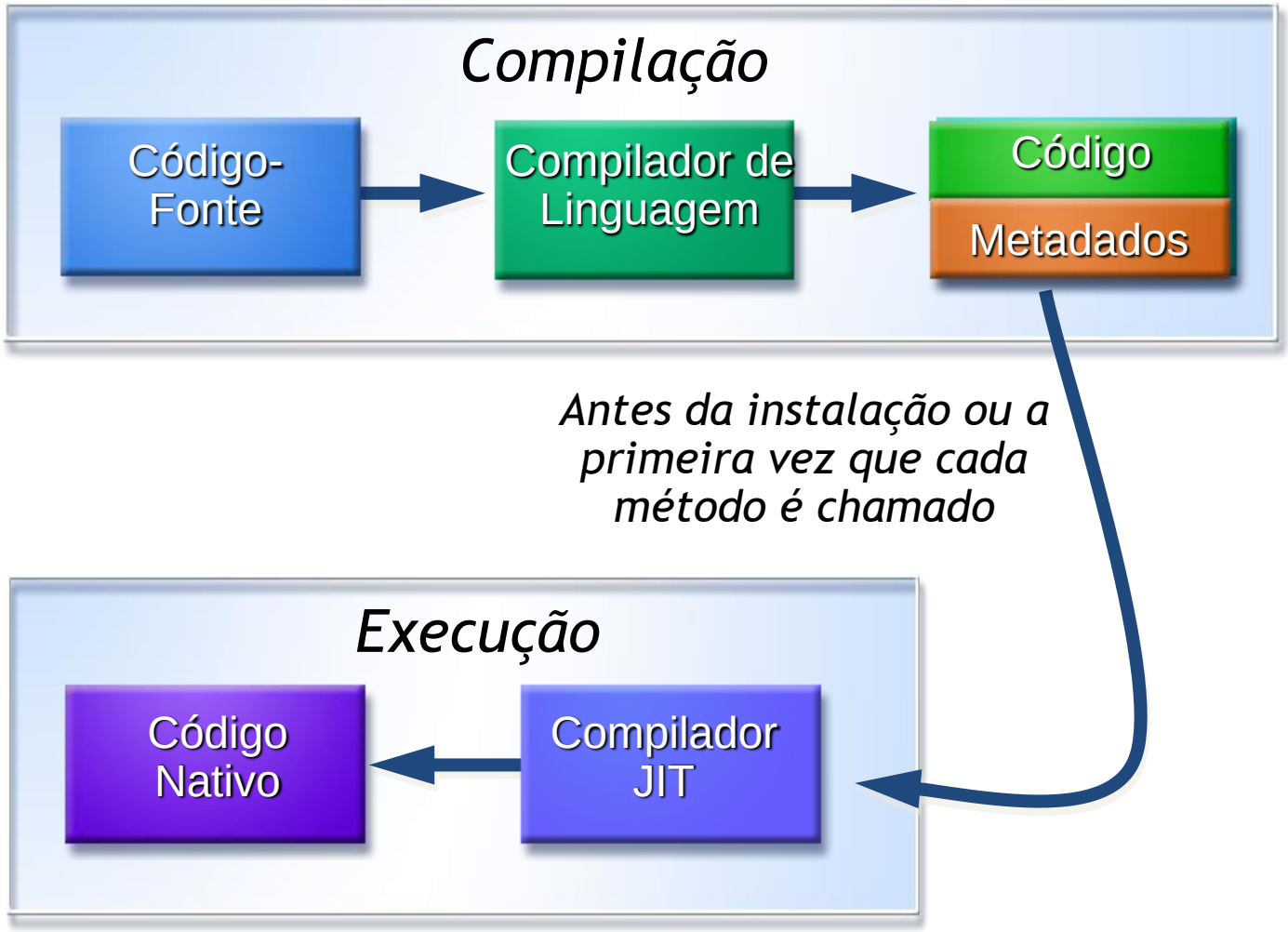
Versões

Versão do .NET Framework	Inclui versão do CLR
1.0	1.0
1.1	1.1
2.0	2.0
3.0	2.0
3.5	2.0
4	4
4.5	4

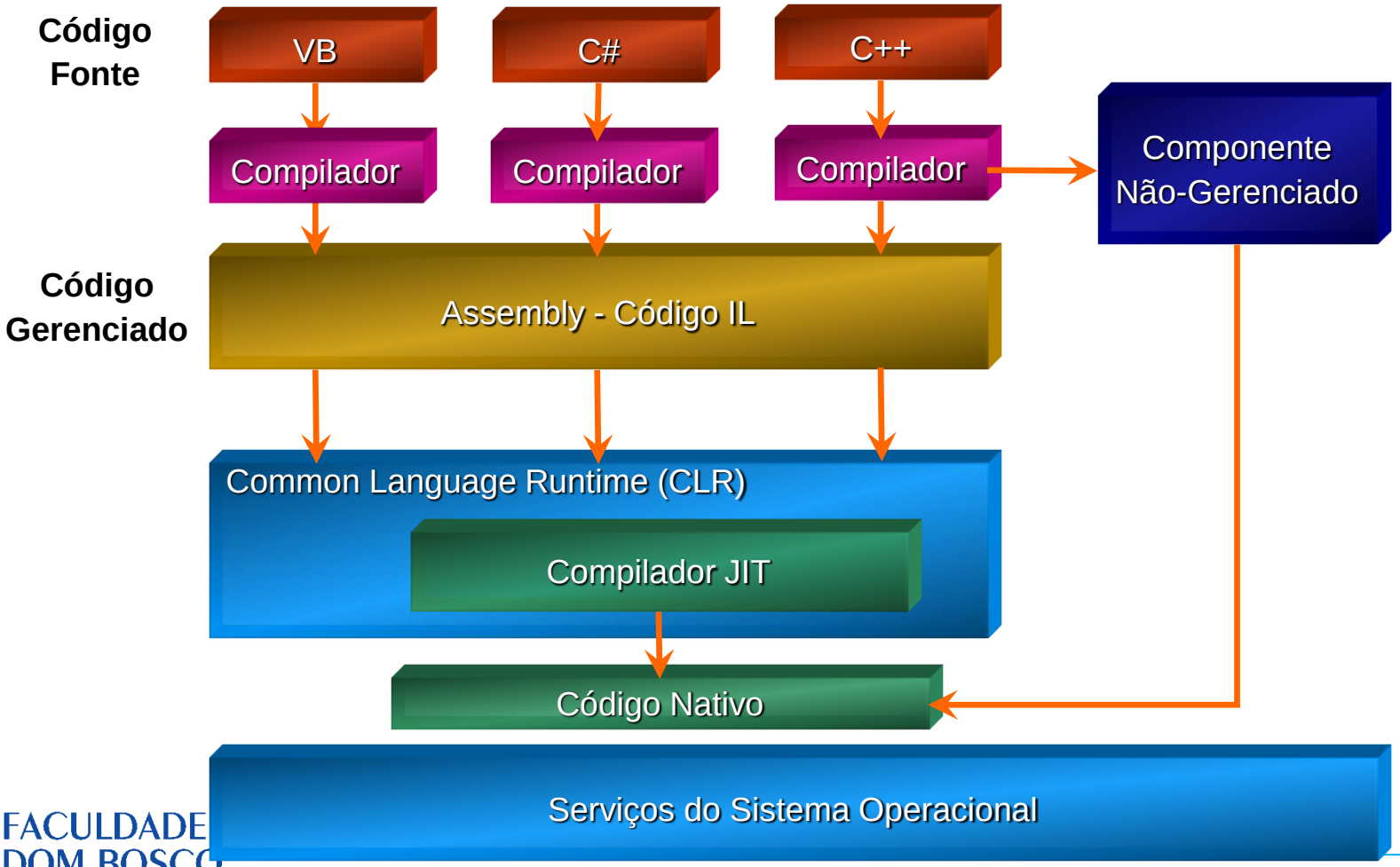
Serviços presentes no CLR

- Gerenciamento de memória
- Tratamento de exceções
- Compilação
- Segurança
- Outros

Processo de compilação



Processo de compilação

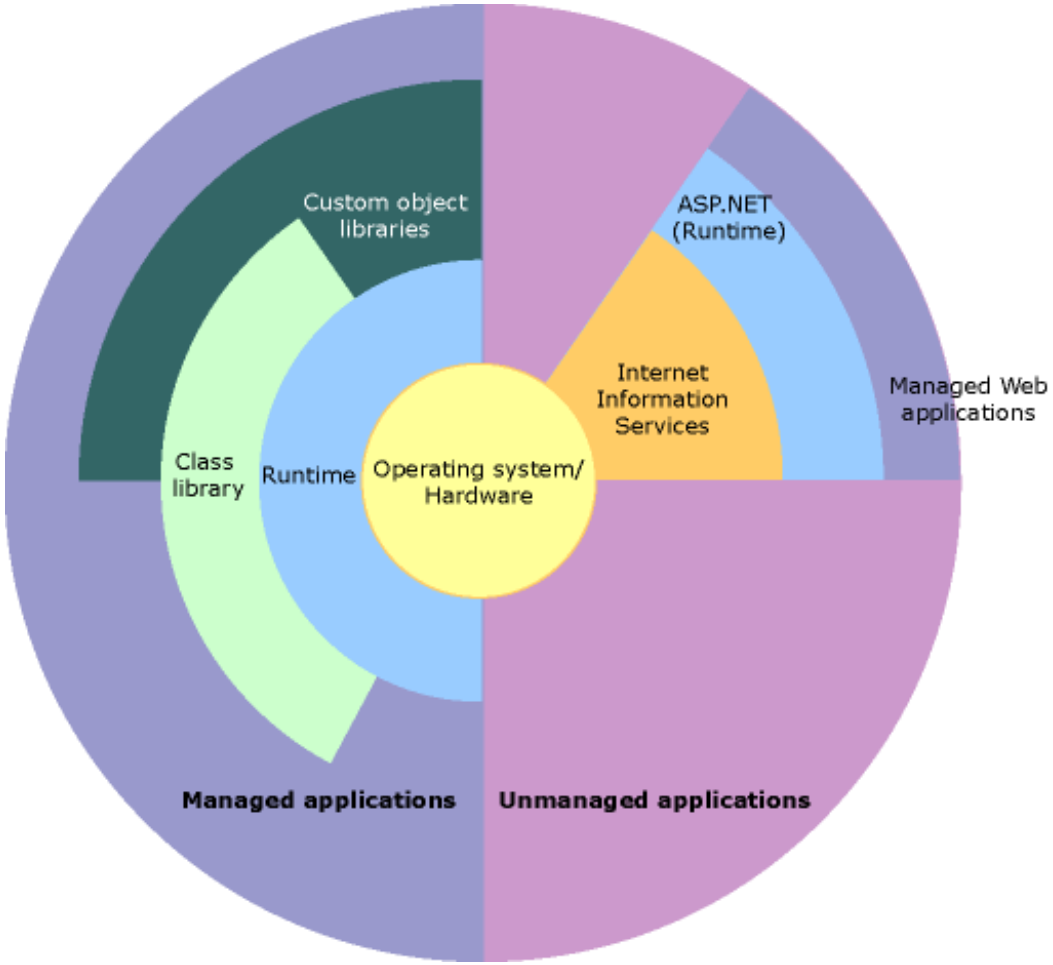


Código managed X unmanaged

- Managed
  - Código Fonte
  - Compilador
  - Código Intermediário (IL): .DLL ou .EXE
  - Requer o ambiente CLR para executar
  - Código de Máquina (Binário)

- Unmanaged
  - Código Fonte
  - Compilador
  - Código de Máquina (Binário)
  - NÃO requer o ambiente CLR para executar

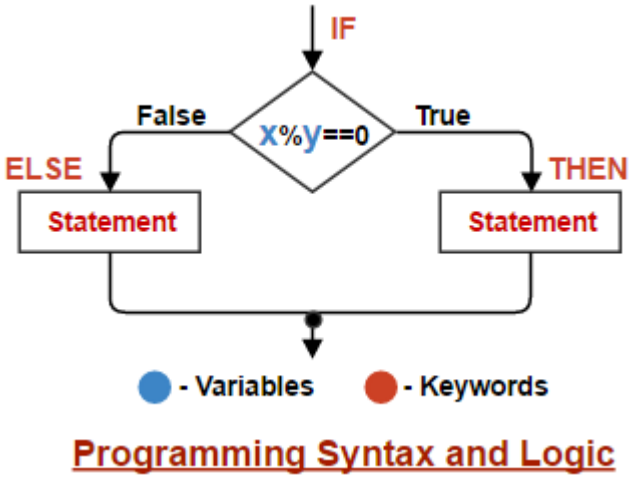
Código managed X unmanaged



Linguagens .NET

- Microsoft Visual Basic
- Microsoft C#
- Microsoft C++ (Gerenciado/Não-Gerenciado)
- Outras:

https://bitbucket.org/brianritchie/wiki/wiki/.NET%20Languages



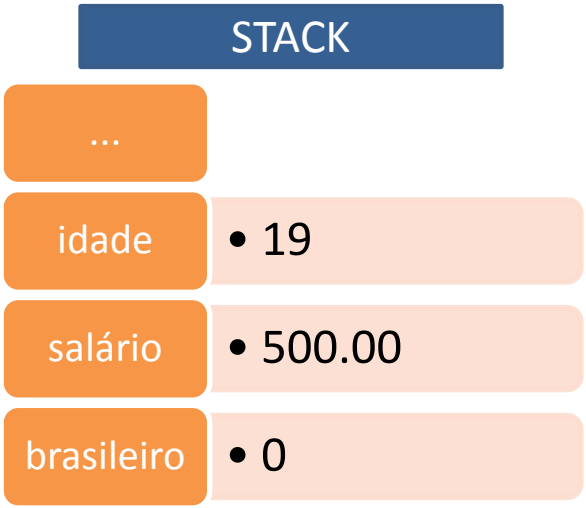
LÓGICA DE PROGRAMAÇÃO

Tipos de Dados

byte	• Inteiro de 8 bits sem sinal
sbyte	• Inteiro de 8 bits com sinal
int	• Inteiro de 32 bits com sinal
uint	• Inteiro de 32 bits sem sinal
long	• Inteiro de 64 bits com sinal
ulong	• Inteiro de 64 bits sem sinal
short	• Inteiro de 16 bits com sinal
ushort	• Inteiro de 16 bits sem sinal
decimal	• Ponto flutuante decimal de 128 bits. Este tipo tem uma precisão de 28 casas decimais.
double	• Ponto flutuante precisão dupla de 64 bits. Este tipo tem uma precisão de 15 casas decimais.
float	• Ponto flutuante precisão simples de 32 bits. Este tipo tem uma precisão de 7 casas decimais.
bool	• Tipo de dados booleano. Pode ser apenas true ou false.
char	• Um único caractere unicode de 16 bits.
string	• Texto em Unicode com até 1 gigabyte.

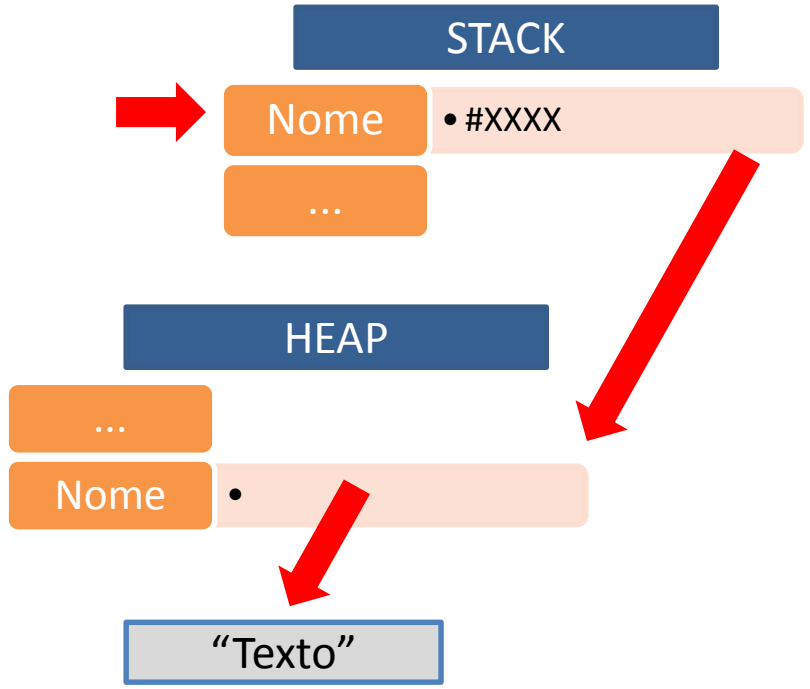
Tipo por valor (value type)

- Armazenado na memória Stack.
- Trabalha com dados diretamente.
- Não pode ser nulo.
- Exemplo:
  - Inteiros
  - Decimais
  - Booleanos
  - Estruturas
  - Enumerações



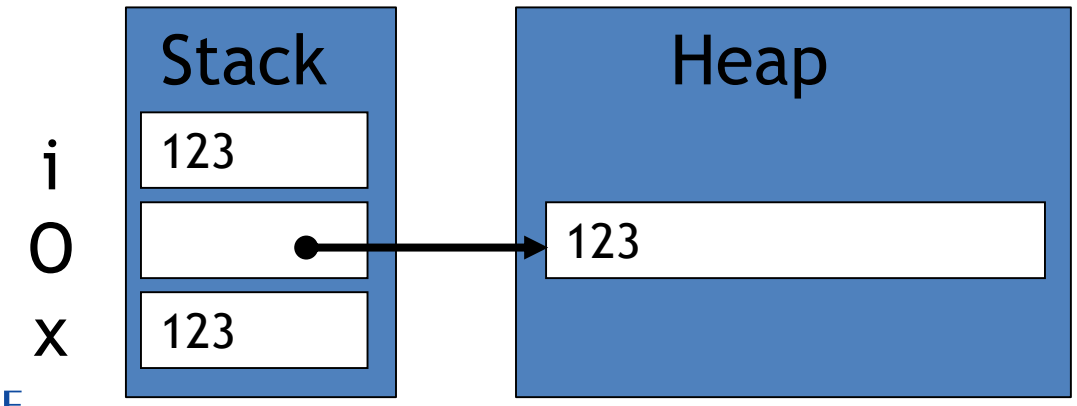
Tipo por referência (reference type)

- Contém uma referência a um ponteiro na memória Heap.
- Pode ser nulo
- Exemplo:
  - Array
  - String
  - Instâncias de classes



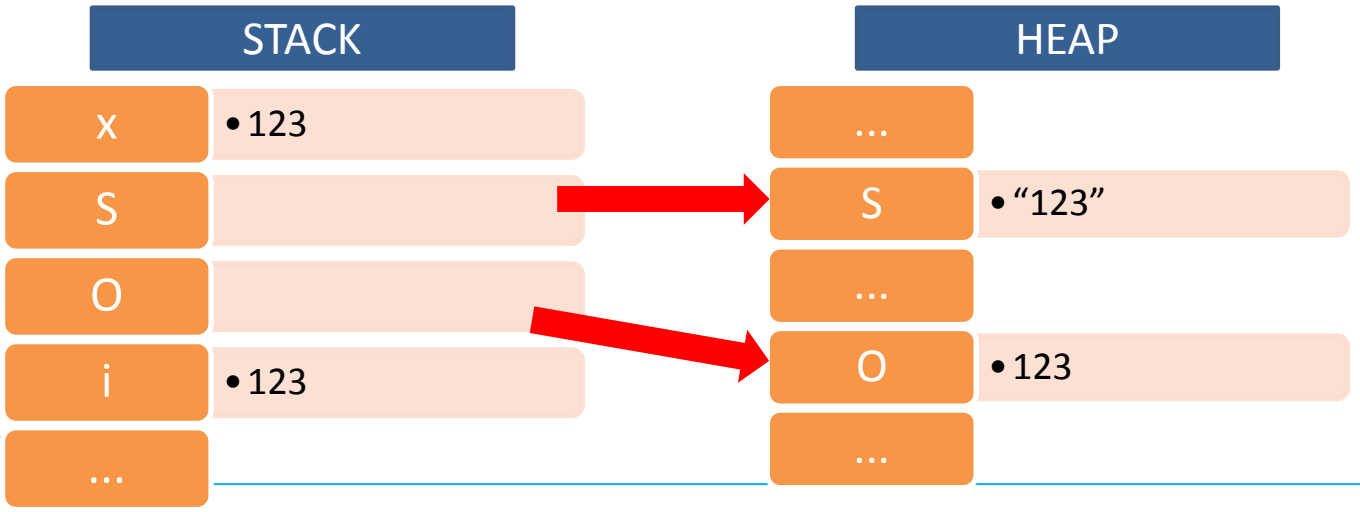
Boxing e unboxing

```
int i = 123;                    // Tipo por valor
object O;                       // Tipo por referência
O = i;                          // Causa "boxing"
string S;
S = O.ToString()                // Chamada método
int x;
x = (int) O;                    // Faz "unboxing"
```



Boxing e unboxing

```
int i = 123;                    // Tipo por valor
object O;
O = i;                          // Causa “boxing”
string S;                       // Tipo por referência
S = O.ToString();          // Chamada método
int x;
x = (int) O;                   // Faz “unboxing”
```



Operadores

Aritméticos	• +, -, *, /, %
Atribuição	• =, +=, -=, *=, /=, <<=, >>=, &=, ^=, |=
Concatenação	• +
Criação de Objetos	• new
Igualdade e Diferença	• ==, !=
Incremento e Decremento	• ++, --
Lógicos e Bit a bit	• &&, ||, !, &, |, ^, ~
Primários	• typeof, sizeof, checked, unchecked
Relacionais	• <, >, <=, >=, is

Estruturas de controle de fluxo

- if ... else
  - Comando condicional: executa um bloco de comandos se uma condição for verdadeira.
  - A cláusula else (condição falsa) é opcional.

```
if (idade >= 18)
{
    Response.Write("Autorizado.");
    Response.Write("Sua idade é: " + idade);
}
else if (idade > 15 && idade < 18)
{
    Response.Write("Somente com os pais.");
    Response.Write("Menor de 18 anos.");
}
else
{
    Response.Write("Não autorizado.");
    Response.Write("Menor de 15 anos.");
}
```

Estruturas de controle de fluxo

- switch ... case
  - Estrutura de decisão que seleciona um comando com base no valor de uma variável.
  - A cláusula default é opcional.

```
switch (sexo)
{
    case "masculino":
        Response.Write("Homem");
        break;
    case "feminino":
        Response.Write("Mulher");
        break;
    default:
        Response.Write("Não informado");
        break;
}
```

Estruturas de repetição

- for
  - Estrutura de repetição composta por três expressões:
    - Inicialização.
    - Condição de parada.
    - Atualização.

```
for (int i = 0; i < 10; i++)
{
    Response.Write(i);
    Response.Write("<br />");
}
```

Estruturas de repetição

- while
  - Estrutura de repetição que realiza as operações indicadas enquanto a condição especificada for verdadeira.

```
int i = 0;
while(i < 10)
{
    Response.Write(i);
    Response.Write("<br />");
    i++;
}
```

Estruturas de repetição

- do ... while
  - Estrutura de repetição semelhante à anterior, porém as condições são verificadas ao final da execução.
  - As operações especificadas são executadas pelo menos uma vez.
  - Necessita do caractere “;” ao final da estrutura.

```
int i = 0;
do
{
    Response.Write(i);
    Response.Write("<br />");
    i++;
}
while (i < 10);
```

Conversão de tipos

```
// Exemplo I
string s = "123";
int i = s;
Response.Write(i);
```

```
// Exemplo II
string s = "123";
int i = Convert.ToInt32(s);
Response.Write(i);
```

```
// Exemplo III
string s = "123";
int i = int.Parse(s);
Response.Write(i);
```

```
// Exemplo IV
string s = "valor";
int i = Convert.ToInt32(s);
Response.Write(i);
```

```
// Exemplo V
string s = "valor";
int i = 0;
if (int.TryParse(s, out i))
{
    Response.Write(i);
}
else
{
    Response.Write("inválido");
}
```

LABORATÓRIO 01

Array

- Array é um tipo que permite o armazenamento de uma coleção de valores de um mesmo tipo.
- Arrays são indexados a partir de zero (0).
- Não podem ter seu tamanho alterado depois de instanciados.

Array

- Para declarar um Array, basta adicionar um par de colchetes logo após a declaração do tipo dos elementos individuais

```
int[] meuVetorDeInteiros;
string[] meuVetorDeStrings;
```

Array

- Instanciando arrays

```
int[] codigos = new int[5];
string[] nomes = new string[100];
object[] produtos = new object[50];
int[] pedidos = {1, 4, 6, 8, 10, 68, 90, 98, 182, 500};
```

Array

- Preenchendo um array

```
nomes[0] = "José";
nomes[1] = "João";
```

Array

- Arrays podem ser:
  - Unidimensionais
  - Bidimensionais
  - Jagged

Array

- Unidimensionais

```
int[] codigos = new int[5];
codigos[0] = 1;
```

```
int[] codigos = {1,3,6,7,8};
```

Array

- Bidimensionais

```
int[,] codigos = new int[2,2];
codigos[0,0] = 11;
```

```
int[,] codigos =
   {{11,42},{35,44}};
```

Array

- Jagged

```
int[][] codigos = new int[2][];
codigos[0] = new int[2];
codigos[0][0] = 11;
```

```
int[][] codigos = { new int[]{11,42}, new int[]{35,44} };
```

Estruturas de repetição

- foreach
  - Esta estrutura de repetição é uma variação do for.
  - Especifica uma variável auxiliar e a coleção ou array cujos elementos serão percorridos.

```
int[] i = { 1, 3, 5, 7, 9 };
foreach (int j in i)
{
    Response.Write(j);
    Response.Write("<br />");
}
```

Enumerações

- Definindo tipos enumerados

```
// Declarando
enum DiasUteis
{
   Segunda, Terca, Quarta, Quinta, Sexta
}

...
// Instanciando
DiasUteis du = DiasUteis.Sexta;

// Imprime "Sexta"
Response.Write(du);
```

Namespaces

- Declarando um namespace

```
namespace Cadastro {
   public class Cliente {}
}
```

- Namespaces em cadeia

```
namespace Cadastro.Telas {
   public class TelaCliente {}
}
```

- Instrução Using

```
using System;
using System.Data;
using Cadastro.Telas;
using Pessoa = Cadastro.Cliente;
```

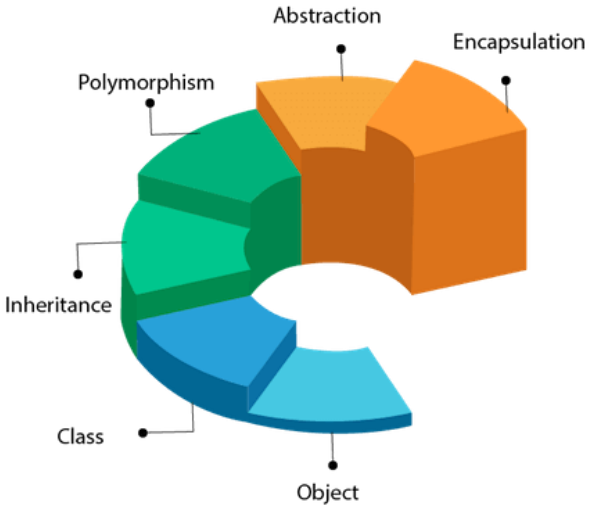
Comentários e regiões

- Criando comentários e regiões:

```
// Comentário de uma linha
/*
   Comentário com
   mais de uma linha
*/
/// <summary>
/// Documentação XML
/// </summary>
private int Atributo;
#region Região
   private int Atributo1;
   private int Atributo2;
#endregion
```

LABORATÓRIO 02

OOPs (Object-Oriented Programming System)



ORIENTAÇÃO A OBJETO
PARTE I

Conceitos de orientação a objeto

Classes

- Uma classe é uma “fábrica” para produzir objetos
- Determina um conjunto de objetos com:
  - propriedades semelhantes
  - comportamentos semelhantes
  - relacionamentos comuns com outros objetos

Classes

- Membros das classes
  - Constantes, atributos, métodos, propriedades, indexadores, eventos, operadores, construtores, destrutores
  - Membros “de instância” e “de classe”
  - Tipos aninhados

Modificadores de acesso

- Modificadores de acesso são utilizados para definir níveis de acesso a membros das classes

Declaração	Definição
public	Acesso ilimitado
private	Acesso limitado à classe e seus membros
internal	Acesso limitado ao programa (assembly)
protected	Acesso limitado à classe, seus membros e a tipos derivados da mesma
protected internal	Acesso limitado à classe, classes derivadas ou membros deste programa (assembly)

Instâncias

- Objetos são gerados a partir de classes
- Uma classe define as propriedades e o comportamento dos objetos gerados por ela
- Todo objeto é uma instância de uma classe

Classes - Atributos

- Definindo uma classe e seus atributos

```
public class Cliente {
  private string nome;
  private decimal limiteCredito;
  private uint   clienteID;
}
```

- Instanciando uma classe

```
Cliente proximoCliente = new Cliente();
```

Classes - Métodos

- Métodos representam as operações associadas à classe

```
public void AumentarLimite(decimal val) {
     limiteCredito += val;
}
```

- Chamando um método

```
proximoCliente.AumentarLimite(100M);
```

Classes - Métodos

- Os parâmetros de um método podem receber um modificador que indica a direção do fluxo de dados
  - Entrada
  - Saída
  - Entrada/saída

[nenhum]	Se não existe modificador, assume que é parâmetro de entrada passado por valor.
out	Parâmetro de saída. Seu valor será atribuído pelo próprio método, não necessitando de inicialização prévia.
ref	Parâmetro de entrada/saída. Seu valor pode ser atribuído antes da chamada do método ou sofrer alteração pelo método.
params	Permite receber um número variável de parâmetros através de um array.

Classes - Métodos

- Parâmetro de saída

```
public void Adicionar(int x, int y, out int r) {…}
Adicionar(1, 2, out resultado);
```

- Parâmetro de entrada/saída

```
public void ParaMaiuscula(ref string s) {…}
ParaMaiuscula(ref frase);
```

- Parâmetros variáveis

```
public void MostrarLista(params int[] lista) {…}
int[] array = new int[3] {1,2,3};
MostrarLista(array);
MostrarLista(1,2,3);
MosttarLista(1,2,3,4,5);
```

Classes - Propriedades

- Propriedades são métodos que protegem o acesso aos membros da classe

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

- Acessando propriedades

```
proximoCliente.Nome = "Microsoft";
```

Classes - Propriedades

- Get e Set auto-implementados:

```
public string Nome {
        get;set;
}
```

- Acessando propriedades

```
proximoCliente.Nome = "Microsoft";
```

Classes - Construtores

- Construtores são métodos especiais que implementam as ações necessárias para inicializar um objeto
  - Tem o mesmo nome da classe
  - Não possuem tipo de retorno (nem *void*)
  - Parâmetros são opcionais

```
public Cliente(string n, uint i) {
      nome = n;
      clienteID = i;
}
```

Classes - Sobrecarga

- Chama-se de sobrecarga de métodos (overloading) o ato de criar diversos métodos com o mesmo nome que se diferenciam pela lista de argumentos (parâmetros)
- Métodos com mesmo nome, mas com tipo, quantidade ou ordenação de parâmetros diferentes, são considerados métodos diferentes

Classes - Sobrecarga

- Exemplo: sobrecarga de construtor

```
public class Data
{
  private int dia, mes, ano;
  public Data(int d, int m, int a) {
    dia = d;
    mes = m;
    ano = a;
  }
  public Data(Data d) : this(d.dia, d.mes, d.ano){
  }
}
```

LABORATÓRIO 03

Herança

- Herança é uma relação de especialização entre classes
- A idéia central de herança é que novas classes são criadas a partir de classes já existentes
  - Subclasse herda de uma Superclasse
  - Subclasse é mais específica que a Superclasse
- Herança origina uma estrutura em árvore

Herança

- Para definir a herança de classes em C# utiliza-se um “:” seguido do nome da superclasse
- C# suporta herança simples de classes

```
public class Classe : SuperClasse {
      …
}
```

Herança

- Ao definir os construtores de uma subclasse:
  - O construtor deve obrigatoriamente chamar o construtor da classe base para inicializar os atributos herdados
  - Caso um construtor não referencie o construtor da classe base, C# automaticamente referencia o construtor vazio da classe base
  - O construtor pode referenciar explicitamente um construtor da classe base via a palavra-chave base após a assinatura do construtor da subclasse e da marca “:”

Sobrescrita de métodos

- Uma subclasse pode sobrescrever (do inglês *override*) métodos da superclasse
  - Sobrescrita permite completar ou modificar um comportamento herdado
  - Quando um método é referenciado em uma subclasse, a versão escrita para a subclasse é utilizada, ao invés do método na superclasse
  - Em C#, um método que sobrescreve um método herdado é marcado pela palavra-chave override

Sobrescrita de métodos

- Um método de uma classe, que pode ser sobrescrito em uma subclasse, deve ser marcado pela palavra-chave virtual
- O método herdado pode ser referenciado através da construção base.nome_método

```
public class SuperClasse {
  public virtual void Metodo(){…}
}
public class Classe : SuperClasse {
  public override void Metodo() {…}
}
```

Polimorfismo

- Polimorfismo é a capacidade de assumir formas diferentes

- C# permite a utilização de variáveis polimórficas
  - Uma mesma variável permite referência a objetos de tipos diferentes
  - Os tipos permitidos são de uma determinada classe e todas as suas subclasses

Polimorfismo

- Uma variável do tipo da superclasse pode armazenar uma referência da própria superclasse ou de qualquer uma de suas subclasses

```
public class Classe : SuperClasse {
      …
}

SuperClasse obj;
obj = new Classe();
```

Operadores de polimorfismo

- **IS** e **AS**

```
if (computador is Produto)
{
   // ações
}
```

```
Produto produto = computador as Produto;

if (produto != null)
{
   Fornecedor fornecedor = produto.Fornecedor;
}
```

Polimorfismo

- Em C# podemos utilizar métodos polimórficos
  - Uma mesma operação pode ser definida em diversas classes de uma hierarquia.
    - cada classe oferece sua própria implementação utilizando o mecanismo de sobrescrita de métodos

Classes abstratas

- Em uma hierarquia de classe, quanto mais alta a classe, mais abstrata é sua definição
  - Uma classe no topo da hierarquia pode definir apenas o comportamento e atributos que são comuns a todas as classes
  - Em alguns casos, a classe nem precisa ser instanciada diretamente e cumpre apenas o papel de ser um repositório de comportamentos e atributos em comum
- É possível definir classes, métodos e propriedades abstratas em C#

Classes abstratas

- Marca-se a classe com a palavra-chave *abstract*

```
public abstract class Funcionario()
{
  public abstract decimal CalcularSalario();
  public abstract string Codigo {get; set;}
}
```

Herança – palavras-chave

ABSTRACT	• Indica uma classe, método ou propriedade que não admite instâncias diretamente.
OVERRIDE	• Indica uma redefinição em uma classe derivada.
VIRTUAL	• Indica um elemento da classe base que pode ser redefinido.
THIS	• Indica um elemento da própria classe.
BASE	• Indica um elemento da classe base.
SEALED	• Indica uma classe que não admite derivadas.

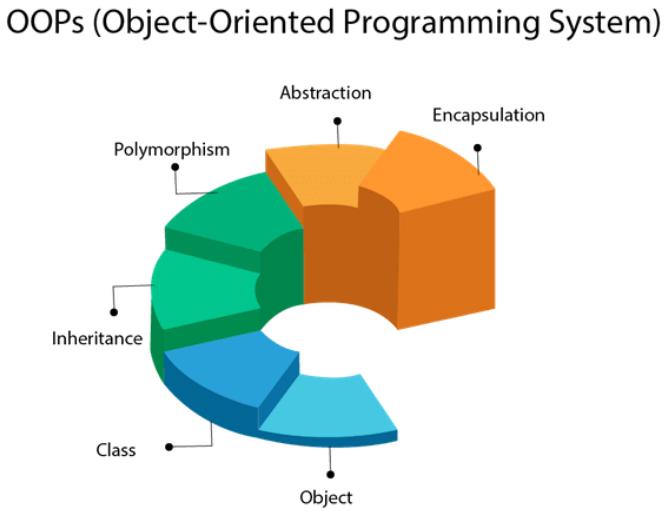
Modificadores de classes

- Public: permite que a classe seja acessada por qualquer assembly.
- Sealed: não permite que a classe seja herdada.
- Partial: permite que a classe tenha seu escopo dividido em dois arquivos.
- Static: especifica que a classe somente tem membros estáticos. Não pode ser instanciada.
- Abstract: define moldes para classes filhas. Não pode ser instanciada.

Modificadores de membros

- Public: permite que os membros das classes sejam acessados por qualquer outro escopo.
- Private: acesso restrito ao escopo da classe.
- Protected: acesso restrito a classe e as derivadas.
- Internal: permite acesso somente por classes do mesmo assembly.
- Static: permite acesso, sem necessidade do objeto ser instanciado.
- Abstract: são métodos de classes Abstract que não possuem implementação.
- Virtual: permite que os métodos sejam sobrescritos por classes filhas.
- Readonly: limita acesso a somente leitura aos atributos da classe.

LABORATÓRIO 04



ORIENTAÇÃO A OBJETO
PARTE II

Interfaces

- Interfaces podem ser utilizadas para separar a especificação do comportamento de um objeto de sua implementação concreta
- Dessa forma a interface age como um contrato, o qual define explicitamente quais métodos uma classe deve obrigatoriamente implementar
  - Por exemplo, suponha a necessidade de implementação da estrutura de dados Pilha
    - Toda pilha deve possuir as operações empilha(), desempilha(), estaVazia()
    - Mas a pilha pode ser implementada com array, lista encadeada, etc

Interfaces

- Existem dois motivos básicos para fazer uso de interfaces:
  - Uma interface é como um contrato que determina o que deve fazer parte de suas classes derivadas;
  - Bibliotecas padronizadas de interfaces uniformizam a construção de projetos.
- Uma interface informa apenas quais são o nome, tipo de retorno e os parâmetros dos métodos.
  - A forma como os métodos são implementados não é preocupação da interface.
  - A interface representa o modo como você quer que um objeto seja usado.

Interfaces

- Declarando interfaces:
  - Uma interface é declarada de forma semelhante a uma classe
  - Utiliza-se a palavra-chave interface ao invés de class
  - Em C#, interfaces podem conter métodos, propriedades, indexadores e eventos
  - Não é possível fornecer modificadores para os membros da interface
    - São implicitamente públicos e abstratos

Interfaces

- Restrições importantes:
  - Uma interface não permite a presença de atributos
  - Uma interface não permite construtores
    - Não é possível instanciar uma interface
  - Não é possível fornecer modificadores para os membros da interface
  - Não é possível aninhar declaração de tipos dentro de uma interface
  - Interfaces somente podem herdar de outras interfaces

Interfaces

- Declarando uma interface:

```
interface IPilha {
  void Empilhar(object obj);
  object Desempilhar();
  object Topo{get;}
}
```

Interfaces

- Implementando interfaces:
  - Como interfaces são compostas de métodos abstratos, esses métodos deverão ser implementados por alguma classe concreta
  - Logo, dizemos que uma interface é implementada por uma classe
  - Utiliza-se a mesma notação de herança
  - A classe deverá implementar todos os métodos listados na interface
    - A implementação deve ser pública, não estática e possuir a mesma assinatura
  - Uma classe pode implementar diversas interfaces

Interfaces

- Implementando uma interface:

```
public class PilhaArray : IPilha {
  private object[] elementos;
  public void Empilhar(object obj){...}
  public object Desempilhar(){...}
  public object Topo{
      get {...}
  }
  ...
}
```

Interfaces

- Implementação explícita de interfaces:
  - Se uma classe implementa duas interfaces que contêm um membro com a mesma assinatura, a mesma implementação será utilizada para as duas interfaces
    - Esta característica pode tornar o código inconsistente
  - C# permite implementar explicitamente um método de uma interface agregando o nome da interface antes do nome do método
    - Como consequência, os métodos somente poderão ser chamados via uma variável do tipo da interface adequada

Interfaces

- Implementando explicitamente uma interface:

```
interface IUmaInterface {
   void metodo();
}
interface IOutraInterface {
   void metodo();
}
public class MinhaClasse : IUmaInterface, IOutraInterface
{
   public void IUmaInterface.metodo(){...}
   public void IOutraInterface.metodo(){...}
}
```

Polimorfismo

- Quando declaramos uma variável como sendo do tipo de uma interface, essa variável irá aceitar qualquer objeto de uma classe que implemente essa interface
- Dessa maneira, temos acessos aos métodos definidos na interface de forma independente do tipo de objeto que estamos utilizando

Polimorfismo

```
interface IMinhaInterface {
      …
}

public class Classe : IMinhaInterface {
      …
}

MinhaInterface obj;
obj = new Classe();
```

LABORATÓRIO 05

Interfaces do Framework

- No ambiente .NET temos uma grande quantidade de interfaces pré-definidas. Por exemplo:
  - IComparable e IComparer para a ordenação de objetos.
  - IEnumerable e IEnumerator para implementar a operação foreach.
  - ICloneable para permitir a criação de cópia de objetos
  - IFormattable para definir cadeias de caracteres formatadas.
  - IDataErrorInfo para associar mensagens de erros a uma classe.

Interfaces do Framework

- IComparable:
  - Interface para comparação de valores segundo alguma ordem parcial
  - Define o método CompareTo() que deve retornar um valor inteiro com o resultado da comparação
    - Menor que zero – se a instância atual é menor que o valor do parâmetro
    - Zero – se a instância atual é igual ao valor do parâmetro
    - Maior que zero – se a instância atual é maior que o valor do parâmetro

Interfaces do Framework

- IComparer:
  - Permite diferentes algoritmos de comparação de valores
  - Define o método Compare() que recebe dois objetos e deve retornar um valor inteiro com o resultado da comparação
    - Menor que zero – se o primeiro objeto for menor que o segundo objeto, de acordo com o algoritmo implementado
    - Zero – se os objetos forem iguais
    - Maior que zero – se o primeiro objeto for maior que o segundo objeto

LABORATÓRIO 06

Estruturas

- Estruturas são tipos por valor, que podem conter:
  - Um construtor
  - Constantes
  - Atributos
  - Métodos
  - Propriedades
- Uso recomendado para representar objetos leves e/ou que eventualmente podem constituir arrays de grande dimensão.
- Não podem ser herdados, porem podem implementar Interfaces.

Estruturas

- Exemplo de uma estrutura:

```
struct Circulo {
    private int _raio;          // Atributo
    public double Circunferencia      // Propriedade
    { get { return 2 * _raio * Math.PI; } }
                // Regra específica para retornar  um valor.
    public Circulo(int raio)  // Construtor com um argumento
    { this._raio = raio; }    // Atribuição do valor do argumento
}                             // para o atributo do objeto.
// Instancia de uma estrutura.
Circulo meuCirculo = new Circulo(10);
// Imprime o valor de uma propriedade
Response.Write(meuCirculo.Circunferencia);
}
```

Estruturas

- Exemplo de sobrecarga de métodos:

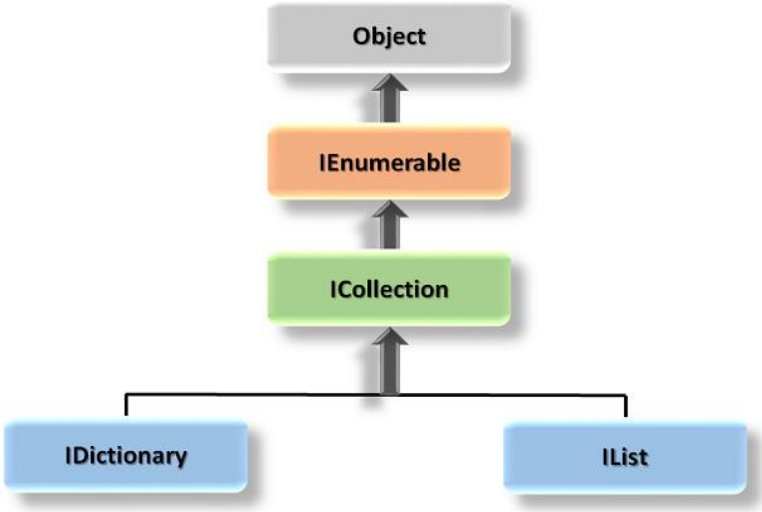
```
... Dentro da mesma estrutura do exemplo anterior:

public void DiminuirRaio(){    // Método simples sem argumentos
    if (_raio > 1)
        _raio--;
}

public void DiminuirRaio(int valor){// Overload do método anterior
    if (_raio - valor > 1)             // com um argumento
        _raio -= valor;
}

... Dentro do evento Page_Load

meuCirculo.DiminuirRaio();     // Chamando o método sem argumentos
meuCirculo.DiminuirRaio(2);    // Chamando o overload
                               // do método anterior
```



COLEÇÕES

Coleções

- Uma ferramenta básica que deve estar presente na caixa de ferramenta de qualquer desenvolvedor
- São classes usadas para agrupar e gerenciar objetos relacionados e que permitem armazenar, buscar e interagir com estes objetos
- As Collections possuem mais funcionalidades do que um array, facilitando sua utilização

Coleções

- O namespace System.Collections contém diversos tipos de collections. Estas collections são responsáveis por agrupar e organizar grandes quantidades de dados

Nome	Descrição
ArrayList	Uma simples coleção de objetos redimensionável e baseada em index.
SortedList	Uma coleção de pares nome/valor ordenada por chave.
Queue	Uma coleção de objetos First-in, First-out.
Stack	Uma coleção de objetos Last-in, First-out.
...	...

Tipos genéricos

- Genéricos são construções do sistema de tipos do .NET Framework que permitem a construção de novos tipos com flexibilidade de tipagem
- Introduzem o conceito de tipos parametrizados

Tipos genéricos

- Pode ser aplicado em:
  - Classes
  - Interfaces
  - Métodos
  - Structs
  - Delegates

Tipos genéricos

- Vantagens:
  - Diminui a necessidade do uso de sobrecarga
  - Permitem criar estruturas de dados sem se restringir a um tipo específico
    - É o exemplo de uso mais utilizado
    - Evita erros de conversão em tempo de run-time de e para Object
  - Maior desempenho
    - Evita boxing / unboxing
  - Verifica o tipo em tempo de compilação

Coleções genéricas

- Disponibilizadas no namespace System.Collections.Generic
  - É o tipo de coleções mais recomendado
- Principais coleções:
  - List, LinkedList, SortedList
  - Dictionary, SortedDictionary
  - KeyedCollection
  - Queue
  - Stack

Coleções genéricas

System.Collections	System.Collections.Generics
ArrayList	List<>
Queue	Queue<>
Stack	Stack<>
Hashtable	Dictionary<>
SortedList	SortedList<>
ListDictionary	Dictionary<>
HybridDictionary	Dictionary<>
OrderedDictionary	Dictionary<>
SortedDictionary	SortedDictionary<>
NameValueCollection	Dictionary<>

Coleções genéricas

System.Collections	System.Collections.Generics
StringCollection	List<String>
StringDictionary	Dictionary<String>
N/A	LinkedList<>

- Todas as classes apresentadas acima possuem funcionalidades e métodos semelhantes a sua correspondente no namespace System.Collections, exceto a classe LinkedList<>, que é exclusiva do namespace System.Collections.Generic.

List

- List é uma coleção sem tamanho fixo, não ordenada e que aumenta conforme a necessidade do programador
- É possível criar Lists capazes de armazenar qualquer tipo de dados: int, string, ou até objetos de classes que você mesmo tenha construído

List

- Criar um List é muito simples. Ele é instanciado como um objeto qualquer, mas deve-se declarar o tipo a ser utilizado:

```
List<int> lista = new List<int>();
```

- Existem duas maneiras de se adicionar itens em um List:
  - A primeira é utilizando os métodos Add, para adicionar apenas um valor, e AddRange, para adicionar vários itens, que normalmente vêm de um array ou de outra collection.
  - A segunda é adicionar diretamente em um determinada posição via os métodos Insert e InsertRange.

List

- Exemplos:
  - Add():

```
List<int> list = new List<int>();
list.Add(12);
list.Add(32);
list.Add(50);
list.Add(“teste”);  // Erro! Tipo inválido
```

  - AddRange():

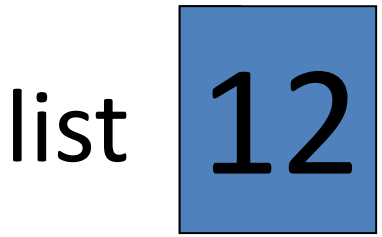
```
int[] outralista = new int[] {1, 2, 3, 4};
list.AddRange(outralista);
```

Método Add()

list 

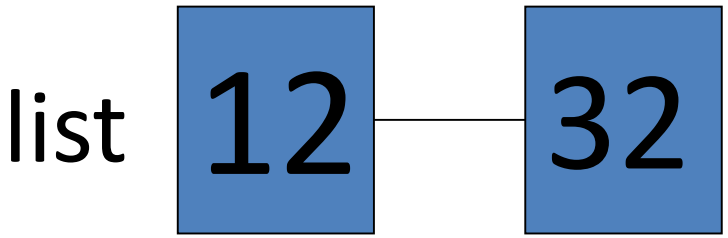
- List<int> list = new List<int>();
- list.Add(12);
- list.Add(32);
- list.Add(25);
- list.AddRange(new int[]{2, 4, 6});

Método Add()



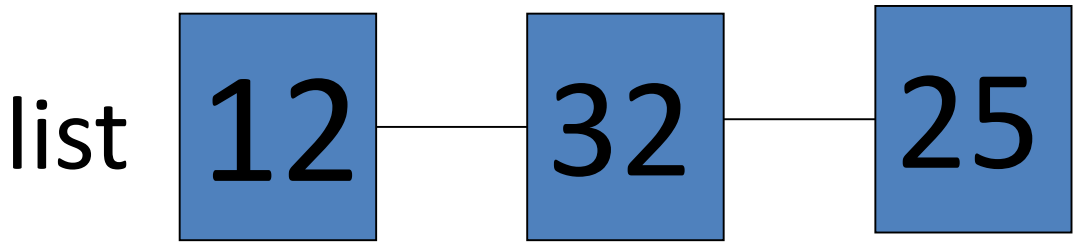
- List<int> list = new List<int>();
- list.Add(12);
- list.Add(32);
- list.Add(25);
- list.AddRange(new int[]{2, 4, 6});

Método Add()



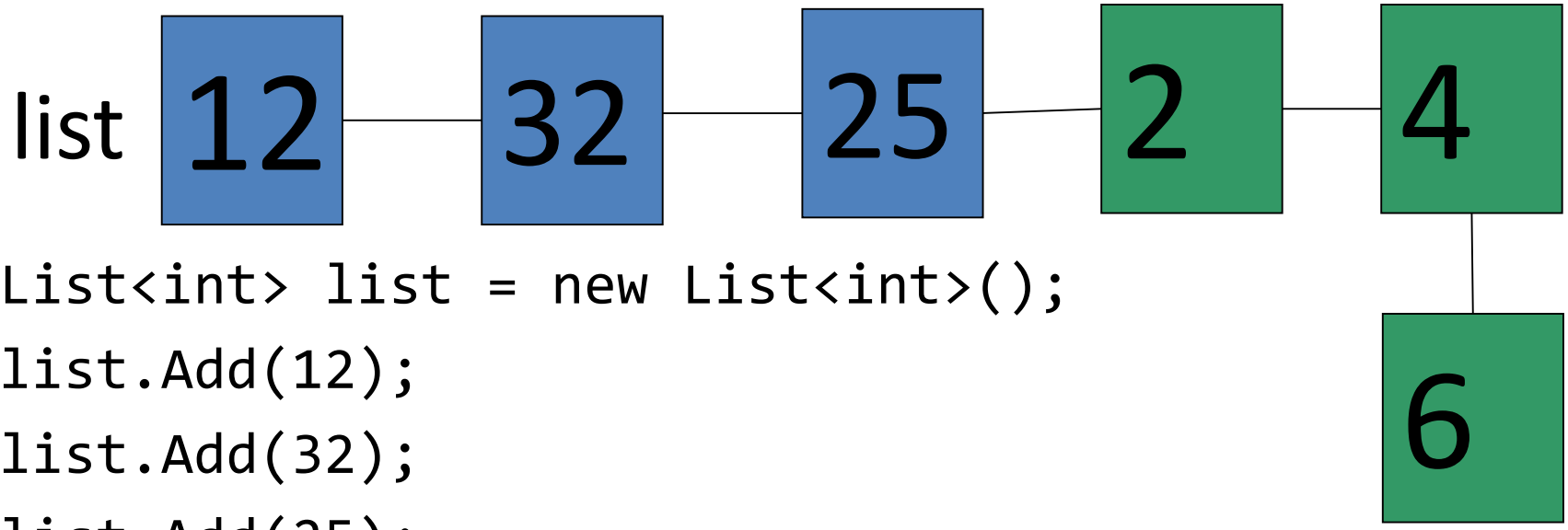

```
List<int> list = new List<int>();
list.Add(12);
list.Add(32);
list.Add(25);
list.AddRange(new int[]{2, 4, 6});
```

Método Add()




```
List<int> list = new List<int>();
list.Add(12);
list.Add(32);
list.Add(25);
list.AddRange(new int[]{2, 4, 6});
```

Método AddRange()




```
List<int> list = new List<int>();
list.Add(12);
list.Add(32);
list.Add(25);
list.AddRange(new int[]{2, 4, 6});
```

List

- Os métodos Add() e AddRange() adicionam itens na última posição do List
- Para adicionar um ou vários objetos em uma posição específica do List utilize o método Insert
- Para adicionar para vários itens utilize InsertRange

List

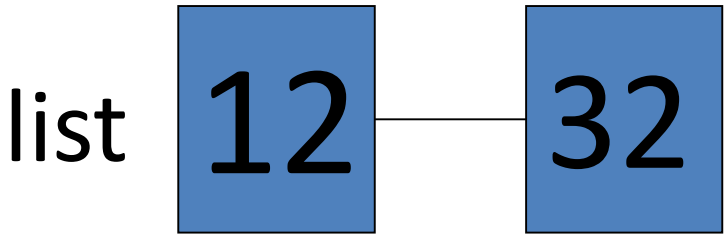
- Exemplos:
  - Insert():

```
List<int> list = new List<int>();
list.Insert(0,12);
list.Insert(1,32);
list.Insert(1,50);
list.Insert(3,44);
```

  - InsertRange():

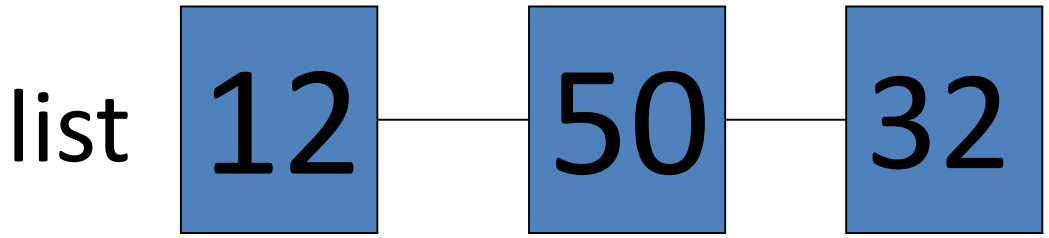
```
int[] outralist = new int[] {2, 4, 6};
list.InsertRange(2,outralist);
```

Método Insert()



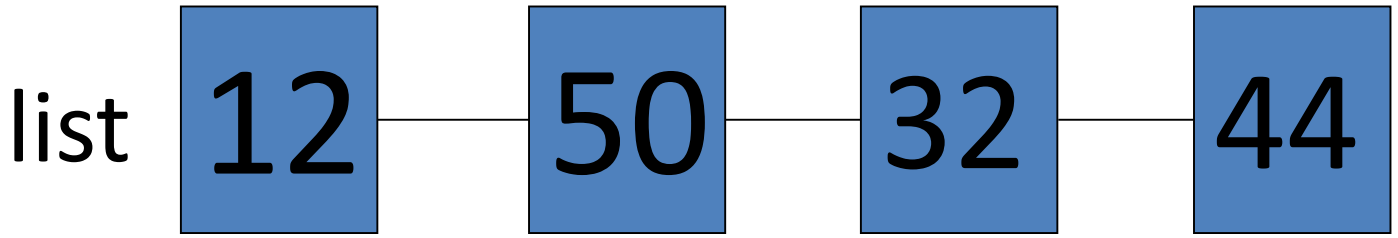
- list.Insert(0, 12);
- list.Insert(1, 32);
- list.Insert(1, 50);
- list.Insert(3, 44);
- list.Insert(10,100); // Funciona ?

Método Insert()



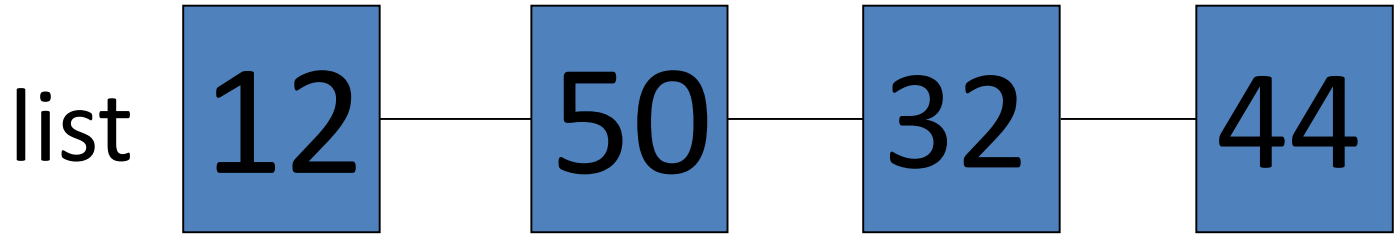

```
list.Insert(0, 12);
list.Insert(1, 32);
list.Insert(1, 50);
list.Insert(3, 44);
list.Insert(10,100);  // Funciona ?
```

Método Insert()

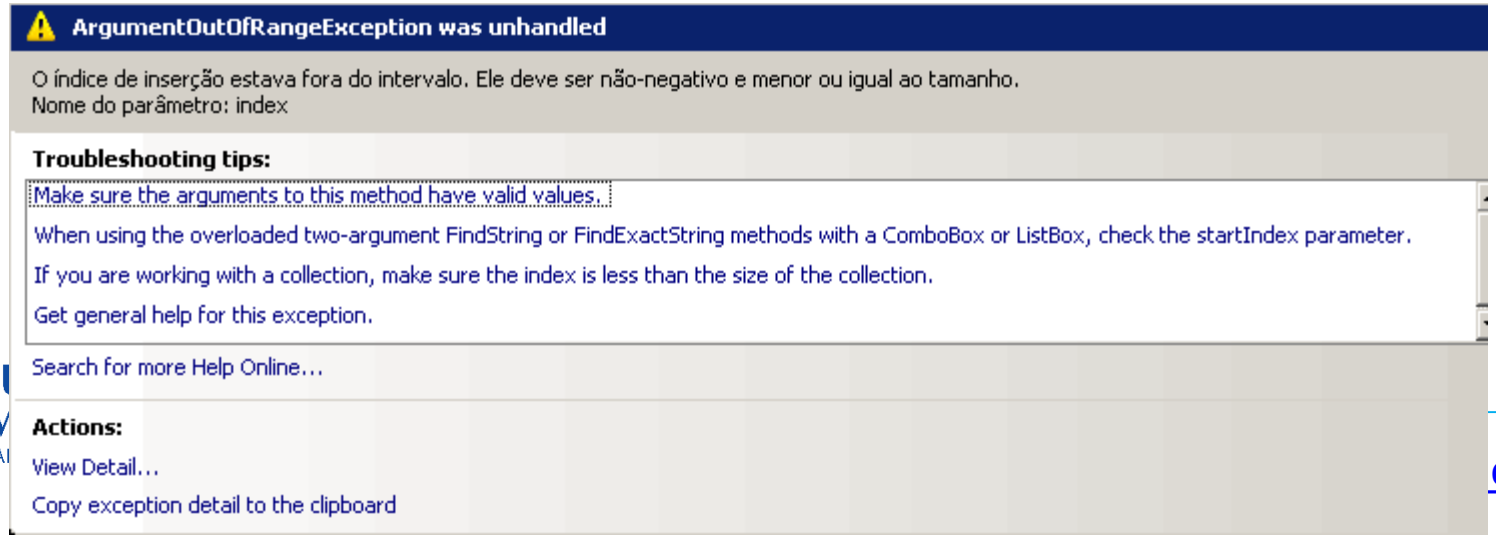



```
list.Insert(0, 12);
list.Insert(1, 32);
list.Insert(1, 50);
list.Insert(3, 44);
list.Insert(10,100);  // Funciona ?
```

Método Insert()




```
list.Insert(0, 12);
list.Insert(1, 32);
list.Insert(1, 50);
list.Insert(3, 44);
list.Insert(10,100);  // Funciona ?
```



List

- Ou utilize a maneira mais simples, via indexadores:

```
List<int> list = new List<int>();
list[3] = 17;
```

- Para a remoção de itens existem três métodos, o Remove(int item), o RemoveAt(int index) e o RemoveRange(int ini, int fim).

List

- Existem ainda outros métodos que podem ser úteis:
  - IndexOf(object item), que retorna o índice do objeto passado como parâmetro;
  - Contains(object item), que verifica se o objeto existe na lista. Se existir, retorna True;
  - Clear(), que apaga todos os itens da lista;
  - Sort(), que ordena a lista;
  - Count(), que retorna o número de itens na lista.

Dictionary<K, V>

- Semelhante a List<T>, porem permite o uso de uma chave ‘K’ de um tipo predefinido, para referenciar um valor do tipo ‘V’.

```
Dictionary<string, string> dic = new Dictionary<string, string>();
        //Instancia um dicionário string-string

dic.Add("Name", "Nome");
dic.Add("Day", "Dia");

dic.Remove("Day");

Response.Write(dic["Name"]);
```

LABORATÓRIO 07

Questões?

FACULDADE DOM BOSCO de Porto Alegre

www.filipomor.com