C#

Seminário de LP
Apresentação da Linguagem

Breno Zupeli
Luiz Meirelles
Pedro Paternostro

Agenda.

1. Histórico.
2. Instalação.
3. Características.
4. Paradigma.
5. Interoperabilidade.
6. Portabilidade.
7. Confiabilidade.
8. Avaliação e Comparação com outras LP’s.

Histórico

Criada pela Microsoft em 2000 visando uma linguagem simples, moderna, de propósito geral e orientada a objetos, C Sharp também deu grande importância à portabilidade, robustez e durabilidade do código assim como à produtividade do programador e ao suporte à internacionalização e localização.

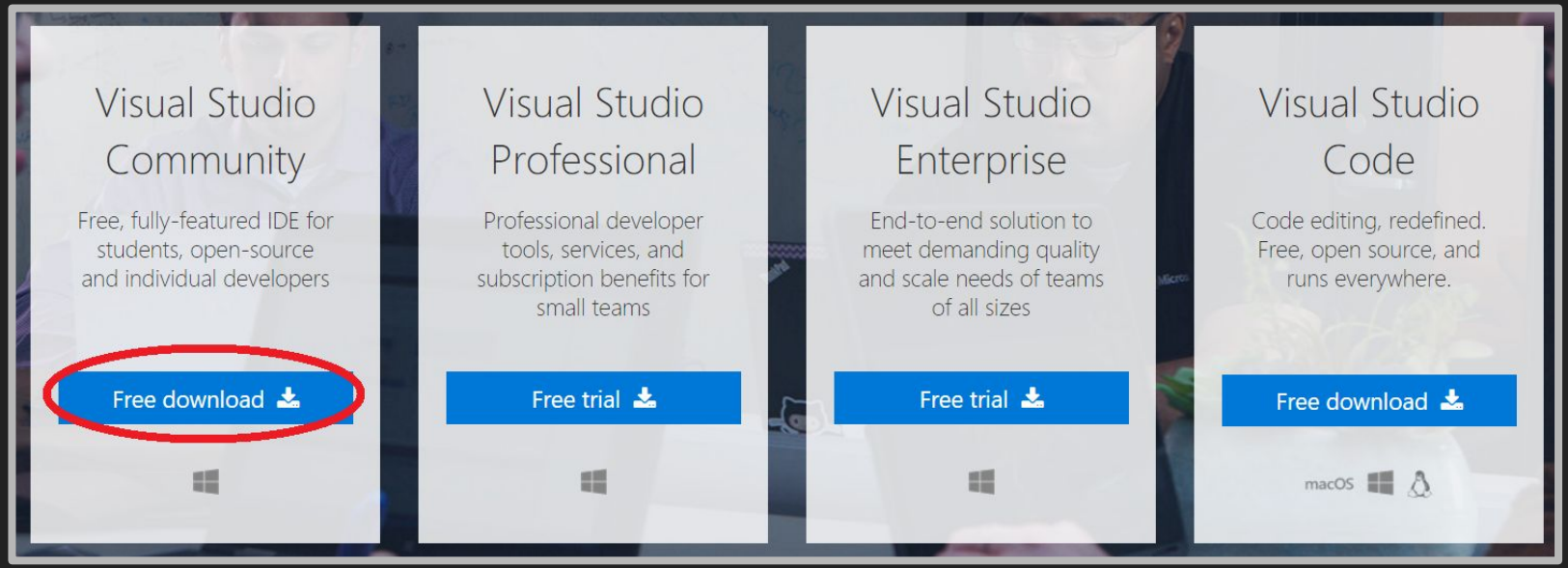
Foi projetada para a possibilidade de ser usada em sistemas de propósito geral e também em sistemas embarcados por ser bastante econômica com relação ao uso de memória e poder de processamento. Foi padronizada em 2001 como ECMA-334 e em 2003 se tornou padrão ISO/IEC 23270.

Anders Hejlsberg
Arquiteto Principal

Instalação

Windows:

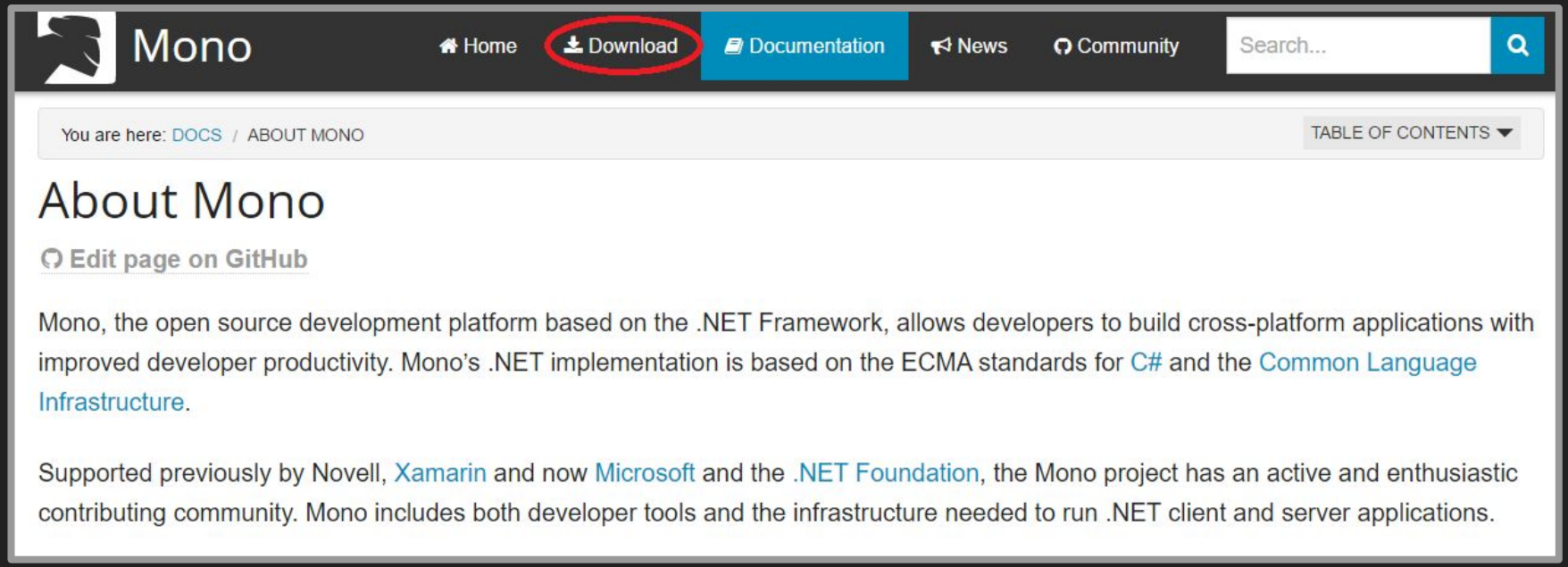
https://www.visualstudio.com/downloads/



Instalação

Linux ou Mac:

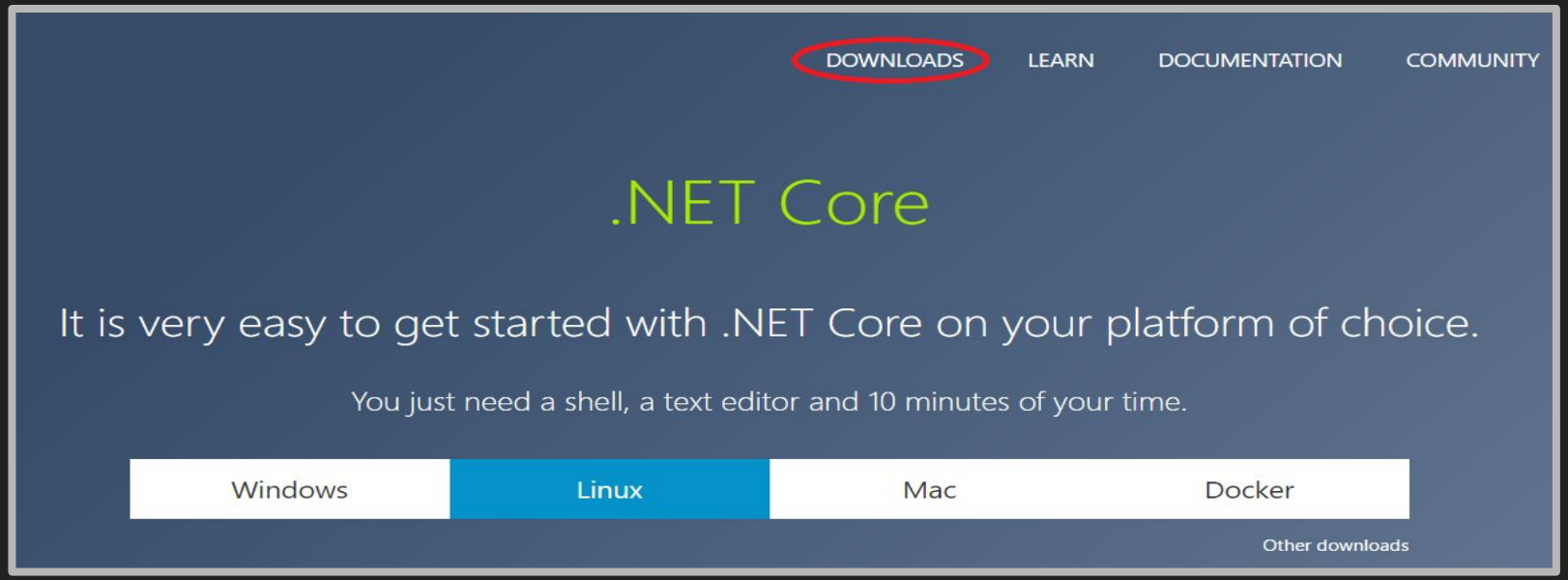
http://www.mono-project.com/



Instalação

Linux ou Mac:

https://www.microsoft.com/**net/core**



Hello World!

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Projeto_Teste {
8        class Program {
9              static void Main(string[] args) {
10                   Console.WriteLine(“Hello World.”);
11                   Console.ReadKey();
12             }
13     }
14 }
```

Palavras Reservadas

abstract	as	base	bool	namespace	new	null	object
break	byte	case	catch	operator	out	out (generic modifier)	override
char	checked	class	const	params	private	protected	public
continue	decimal	default	delegate	readonly	ref	return	sbyte
do	double	else	enum	sealed	short	sizeof	stackalloc
event	explicit	extern	false	static	string	struct	switch
finally	fixed	float	for	this	throw	true	try
foreach	goto	if	implicit	typeof	uint	ulong	unchecked
in	in (generic modifier)	int	interface	unsafe	ushort	using	virtual
internal	is	lock	long	void	volatile	while	

Valores e Tipos de Dados

- Tipagem estática e forte! Erros de tipo detectados em compilação ou execução. Porém possui aritmética de ponteiros.
- Diferentemente de Java, que conta com classes wrapper, todos os tipos primitivos de C# são objetos no System namespace e tem apelidos designados, **int** para **System.Int32**, **double** para **System.Double**.
- Possui enum e tipo booleano restrito, que não pode ser convertido para int, prática comum em C e C++.
- Suporta declaração implícita (e de tipagem forte) de variáveis por meio da palavra chave **var.**

Permite o uso de Ponteiros, porém o código passa a ser não gerenciado, chamado “unsafe”.

```
1  class TestePonteiro {
2      unsafe static void Main() {
3          int[] numeros = {0,1,2,3,4};
4          fixed (int* p1 = numeros) {
5              for(int* p2=p1; p2<p1+numeros.Length; p2++) {
6                  System.Console.WriteLine("Value:{0} @ Address:{1}", *p2, (long)p2);
7              }
8          }
9      }
10 }
```

```
Value:0 @ Address:139878486508888
Value:1 @ Address:139878486508892
Value:2 @ Address:139878486508896
Value:3 @ Address:139878486508900
Value:4 @ Address:139878486508904
```

Delegates

- Um delegate é um tipo de referência que pode ser usado para encapsular um método.
- Delegates são semelhantes aos ponteiros de função em C e C++, entretanto, são fortemente tipados e seguros.

```
// Declare delegate -- defines required signature:
delegate double MathAction(double num);

class DelegateTest
{
    // Regular method that matches signature:
    static double Double(double input)
    {
        return input * 2;
    }

    static void Main()
    {
        // Instantiate delegate with named method:
        MathAction ma = Double;

        // Invoke delegate ma:
        double multByTwo = ma(4.5);
        Console.WriteLine("multByTwo: {0}", multByTwo);

        // Instantiate delegate with anonymous method:
        MathAction ma2 = delegate(double input)
        {
            return input * input;
        };

        double square = ma2(5);
        Console.WriteLine("square: {0}", square);

        // Instantiate delegate with lambda expression
        MathAction ma3 = s => s * s * s;
        double cube = ma3(4.375);

        Console.WriteLine("cube: {0}", cube);
    }
    // Output:
    // multByTwo: 9
    // square: 25
    // cube: 83.740234375
}
```

Tipo	Tamanho	Valores Possíveis
bool	1 byte	true e false
byte	1 byte	0 a 255
sbyte	1 byte	-128 a 127
short	2 bytes	-32768 a 32767
ushort	2 bytes	0 a 65535
int	4 bytes	-2147483648 a 2147483647
uint	4 bytes	0 to 4294967295
long	8 bytes	–9223372036854775808L to 9223372036854775807L
ulong	8 bytes	0 a 18446744073709551615
float	4 bytes	Números até 10 elevado a 38. Exemplo: 10.0f, 12.5f
double	8 bytes	Números até 10 elevado a 308. Exemplo: 10.0, 12.33
decimal	16 bytes	números com até 28 casas decimais. Exemplo 10.991m, 33.333m
char	2 bytes	Caracteres delimitados por aspas simples. Exemplo: 'a', 'ç', 'o'

Possui Collections!

- List<T>

```
List<Funcionario> funcList = new List<Funcionario>();
funcList.Add(new Funcionario(“Alberto”,1500));
Console.WriteLine(funcList[0]);   // ToString() de Alberto será chamado!
```

- Queue<T>

```
Queue<string> numbers = new Queue<string>();
numbers.Enqueue(“one”);
numbers.Enqueue(“two”);
string oldest = numbers.Dequeue(); // retira elemento mais antigo e retorna-o
```

- Dictionary<TKey, TValue>

```
Dictionary<string, string> openWith = new Dictionary<string, string>();
openWith.Add("txt", "notepad.exe");
Console.WriteLine(openWith[txt]);  // acessa valor e imprime
```

Structs

- Instâncias pequenas, imutáveis e que não precisem de encapsulamento;
- Estas estruturas têm seu conteúdo integralmente copiado quando precisa transportar de um local para outro. Portanto são tipos por valor (value types);
- Não permitem herança;
- Implicitamente herda da classe abstrata **ValueType**, que por sua vez é derivada de **Object**.
- Tais classes só possuem comportamento, e não estado.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Projeto_Teste {
8        struct Program {
9              static void Main(string[] args) {
10                   Console.WriteLine(“Hello World.”);
11                   Console.ReadKey();
12             }
13    }
14 }
```

Constantes

```
1  class Calendario {
2        const int meses = 12;
3        const int semanas = 52;
4        const int dias = 365;
5
6        public const double diasPorSemana = (double) dias / semanas;
7        public const double diasPorMes = (double) dias / meses;
8  }
9
10 int x = Calendario.diasPorMes;
```

Variáveis e Constantes

Suporte a tipos anônimos, o programador pode declarar objetos sem precisar definir explicitamente o tipo primeiro.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

```
1  // Example #2: var is required because
2  // the select clause specifies an anonymous type
3  var custQuery = from cust in customers
4                  where cust.City == "Phoenix"
5                  select new { cust.Name, cust.Phone };
```

Permite variáveis dinâmicas, de tipagem em tempo de execução.

```
dynamic a = 10;
dynamic b = “Hello World!”;
dynamic c = 25.5m;

c = a;  // muda para int em tempo de execução
c = b;  // muda para string em tempo de execução
```

C# aborda o armazenamento com pilha e monte (Algol-Like) para aproveitar o melhor de cada estrutura, obtendo maior eficiência. Possui também suporte nativo a serialização.

```
[Serializable]
public class MyObject {
  public int n1 = 0;
  public int n2 = 0;
  public String str = null;
}
```

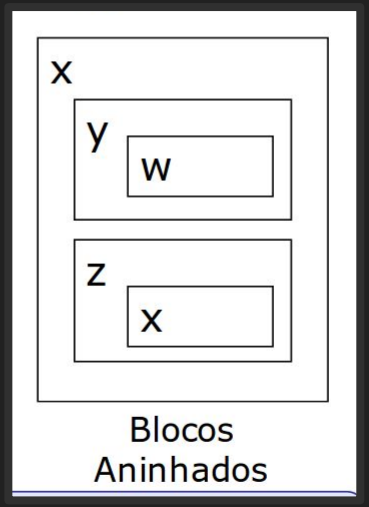
```
MyObject obj = new MyObject();
obj.n1 = 1;
obj.n2 = 24;
obj.str = "Some String";
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Create, FileAccess.Write, FileShare.None);
formatter.Serialize(stream, obj);
stream.Close();
```

```
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Open, FileAccess.Read, FileShare.Read);
MyObject obj = (MyObject) formatter.Deserialize(stream);
stream.Close();
```

Amarrações

- Escopo estático, blocos aninhados (Algol-Like)
- Sobrecarga de funções e de operadores!
- Definições e declarações irrestritas, em qualquer lugar.
- É case sensitive!

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```



Comandos

Atribuição:

```
area = 3.14 * (raio * raio);
area++;
area *= area;
```

Iterativos:

```
for (int i = 0; i < 5; i++) { … }
while (x < 5) { x++; }
do { x++; } while (x < 5);
foreach(int element in myCollection) {
        Console.WriteLine(element);
}
```

Condicionais:

```
if (5 > 2) { … } else if { ... } else { ... }

switch (idade) {
        case 18:
                Console.WriteLine(“Voto obrigatório!”);
                break;
        case 16:
                Console.WriteLine(“Voto facultativo!”);
                break;
        default: Console.WriteLine(“Não altera.”);
                break;
}
```

Desvio irrestrito:

```
for (int i = 0; i < 10; i++) {
        if (i == 5)
                goto Meio;
}
Meio:
        Console.WriteLine(“Saiu no meio.”);
```

```
IEnumerable<int> Power(int number, int
exponent) {
        int result = 1;

        for (int i = 0; i < exponent; i++) {
                result = result * number;
                yield return result;
        }
}

foreach (int i in Power(2, 8)) {
                Console.Write("{0} ", i);
        }
```

Expressões

Aritméticas

```
double a = 20 / 3;
double a = 20 / 3.0;
```

Binárias

```
a & b;
a | b;
a << b; // left shift
a ^ b;  // XOR
~a;
```

Relacionais

```
a > b
a >= b
a == b
```

Booleanas

```
a && b
a || b
!a
a ^ b
```

Agregações

```
int[] intArray = new int[] { 3 + i, 4, 5 };
```

Condicionais

```
int nota = (bomSeminario) ? 10 : 0;
```

Com efeitos colaterais

```
a++;
```

Referenciamento

```
*p = 22 + *p;
Funcionario func = new
        Funcionario(“Bob”, 2600);
```

Modularização

Assim como C++, possui namespaces. Eles podem ser aninhados dentro dos outros.

Quando os nomes são muito grandes recomenda-se o uso de aliases.

```
using SBC = Seminario.Banco.Contas;
```

```
namespace Seminario.Banco
{
    class UmaClasse { … }

    interface UmaInterface { … }

    namespace Contas {
        // o nome é Seminario.Banco.Contas
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

- Possibilita passagem de parâmetros posicional e por nome. Passagem de parâmetros se faz por cópia, com possibilidade de passagem por referência usando as palavras chave **ref** e **out**.

```
float CalcularIMC(int peso, int altura) { … }

CalcularIMC(80, 180);

CalcularIMC(altura: 180, peso: 80);
CalcularIMC(peso: 80, altura: 180);
```

Um pouco mais sobre out & ref

A palavra **out** tem função semelhante à **ref,** porém **ref** necessita que a variável tenha sido inicializada antes da passagem. Com **out** possibilitando a passagem de variáveis não inicializadas, faz-se necessário a atribuição de um valor a variável antes que seja retornada pelo método.

Apesar de **ref** e **out** causarem comportamentos distintos, não são consideradas distintas na assinatura do método, impossibilitando sobrecargas como a que segue:

```
public void MetodoExemplo(out int i) { }
public void MetodoExemplo(ref int i) { }
```

É possível definir valores default para parâmetros, utilizando-se disso e da passagem por nome, também suporta parâmetros opcionais!

```
void MetodoExemplo(int intObrigatoria, string StrOpcional = “C#”, int intOpcional = 47) { ... }
```

```
MetodoExemplo(4, ,3);       // Não compila!

MetodoExemplo(4, intOpcional: 3);      // Ok!
```

Possui alternativa para parâmetros variáveis pela palavra chave **params**

```
void Funcao(int x, params string[] values) { ... }

Funcao(10, "hello", "there");
Funcao(999, “produtos”, “de”, “limpeza”);
```

Polimorfismo

C# não suporta herança múltipla, possui interfaces e classes abstratas!

```
interface IForma {
    void Desenha();
}

abstract class Poligono : IForma {
    public void Desenha() { … }
    // public void Desenha();
    abstract public void GetVertices() { … }
}
```

```
class Quadrado : Poligono() {
    public void GetVertices() { … }
}
```

Uso da palavra chave **virtual** para sinalizar métodos para a amarração tardia e **base** para se referir a métodos e atributos herdados.

```
class Pai {
	protected virtual void UmMetodo() { … }
}

class Filho : Pai {
	protected override void UmMetodo() {
		base.UmMetodo();
		(…)
	}
}
```

Coerção

```
float a = 12;   // Conversão de ampliação é implícita!

int b = 12.0f;   // ERRO! Conversão de estreitamento deve ser explicitada.

int c = (int) 12.0f;   // Ok!
```

Suporta sobrecarga de funções

```
void Desenha(Forma forma, float tamanho) { ... }
                                                    // Ok!
void Desenha(Forma forma, int quantidade) { … }


void Funcao(int, float) { … }

void Funcao(float, int) { ... }

Funcao(5, 2);    // ERRO (Call is ambiguous)
Funcao((float) 5, 2)  // Ok!
```

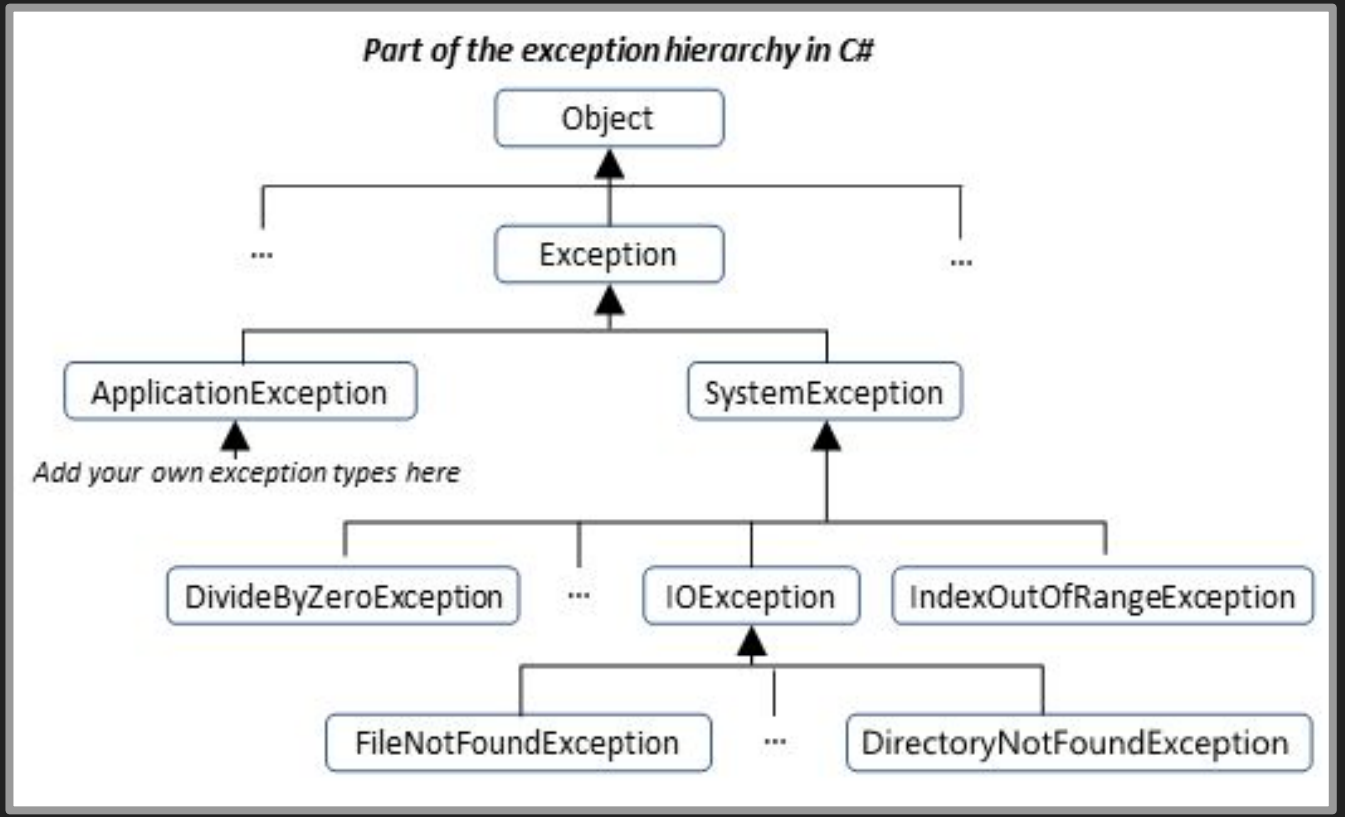
Polimorfismo paramétrico! (Tipos Genéricos)

```
1  public class Pilha<T> {
2      T[] m_Items;
3      public Stack():this(100) {}
4      public Stack(int size) {
5              m_Size = size;
6              m_Items = new T[m_Size];
7      }
8      public void Push(T item){...}
9      public T Pop() {...}
10 }
```

```
1  Pilha<int> pilha = new Stack<int>();
2  pilha.Push(1);
3  pilha.Push(2);
4  int number = pilha.Pop();
```

Exceções

C# possui mecanismo de tratamento de exceções, porém não possui Checked Exceptions, ou seja, o programador pode escolher não tratar qualquer exceção.



Curiosidade: http://www.artima.com/intv/handcuffs.html

Uma vez que uma exceção ocorre no bloco **try**, o fluxo do programa vai para o primeiro handler associado na pilha de chamada.

Se não for encontrado nenhum handler para determinada exceção, a execução do programa para com uma mensagem de erro.

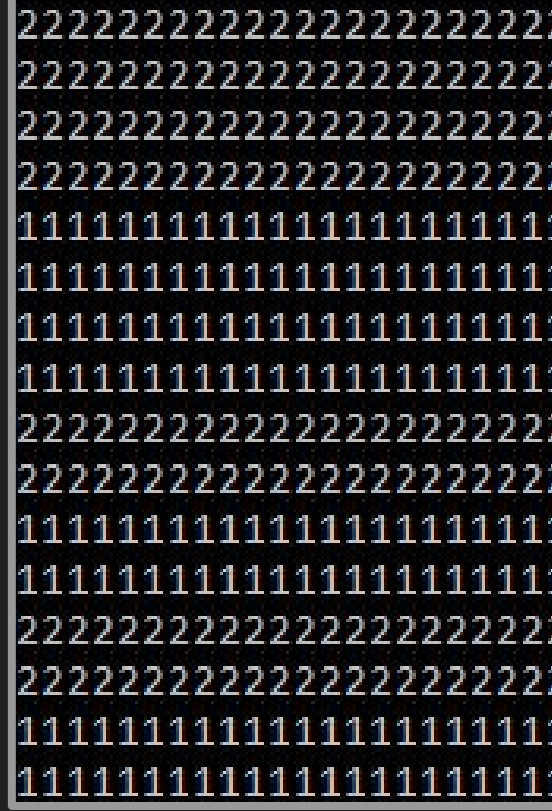
```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Saída: Attempted divide by zero.

Concorrência

Threads

```
1  using System;
2  using System.Threading;
3  namespace UsandoThreads {
4      class Program {
5          static void Main(string[] args) {
6              // dispara uma nova thread para executar
7              Thread t = new Thread(NovaThread);
8              t.Start();
9              // Simultaneamente, executa uma tarefa na thread principal
10             for (int i = 0; i < 10000; i++) Console.Write("1");
11         }
12         static void NovaThread() {
13             for (int i = 0; i < 10000; i++) Console.Write("2");
14         }
15     }
16 }
```



Saída

Semáforos

```
1  using System;
2  using System.Threading;
3  class Semaforos {
4        static Thread[] threads = new Thread[5];
5        static Semaphore sem = new Semaphore(3, 3);
6        static void Metodo() {
7                Console.WriteLine("{0} Esperando na fila...", Thread.CurrentThread.Name);
8                sem.WaitOne();
9                Console.WriteLine("{0} Entrou no Metodo!", Thread.CurrentThread.Name);
10               Thread.Sleep(300);
11               Console.WriteLine("{0} Saiu do Metodo!", Thread.CurrentThread.Name);
12               sem.Release();
13       }

          /*  Continua */
```

```
    /* Continuação */

14  static void Main(string[] args) {
15          for (int i = 0; i < 5; i++) {
16                  threads[i] = new Thread(Metodo);
17                  threads[i].Name = "thread_" + i;
18                  threads[i].Start();
19          }
20          Console.Read();
21  }
22 }
```

```
thread_1 Esperando na fila...
thread_1 Entrou no Metodo!
thread_2 Esperando na fila...
thread_2 Entrou no Metodo!
thread_3 Esperando na fila...
thread_4 Esperando na fila...
thread_0 Saiu do Metodo!
thread_3 Entrou no Metodo!
thread_1 Saiu do Metodo!
thread_4 Entrou no Metodo!
thread_2 Saiu do Metodo!
thread_3 Saiu do Metodo!
thread_4 Saiu do Metodo!
```

Saída

IO comum

```
Console.WriteLine("Digite o seu nome:");
string line = Console.ReadLine();
Console.WriteLine("Olá " + line + “, pressione Enter para terminar o programa.”);
Console.Read();
```

IO de arquivos

```
var text =File.ReadAllText
      (@"C:\words.txt");

File.WriteAllText
      (@"C:\words.txt", text + "DERP");
```

```
string path = @"c:\temp\MyTest.txt";
if (!File.Exists(path))
{
    // Create a file to write to.
    using (StreamWriter sw = File.CreateText(path))
    {
        sw.WriteLine("Hello");
        sw.WriteLine("And");
        sw.WriteLine("Welcome");
    }
}

// Open the file to read from.
using (StreamReader sr = File.OpenText(path))
{
    string s = "";
    while ((s = sr.ReadLine()) != null)
    {
        Console.WriteLine(s);
    }
}
```

Paradigma

C# dá certo suporte à programação funcional por possuir características como delegates, mas é majoritariamente imperativa e orientada a objetos, assim segundo a classificação de Tucker e Noonan, a linguagem se enquadra como Multiparadigma.

Portabilidade

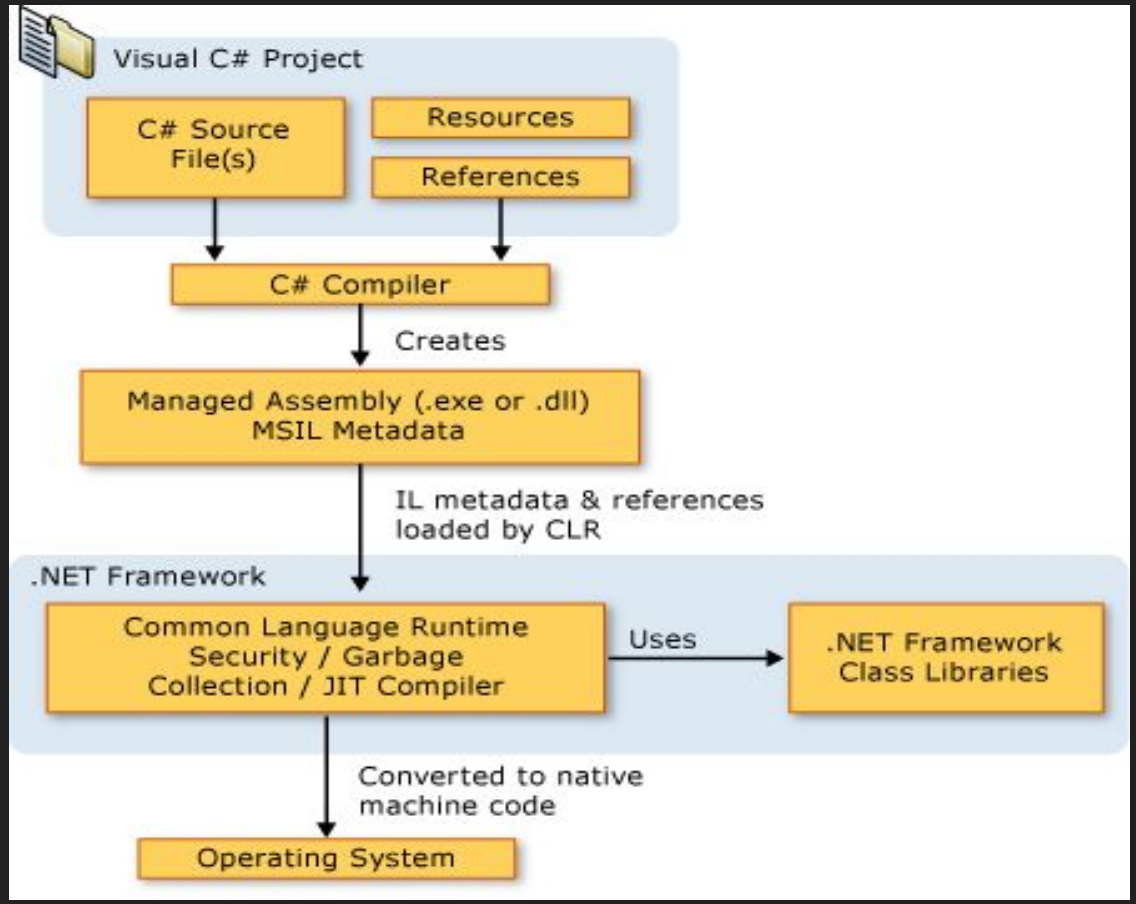
Compilação:

O código fonte é compilado para Common Intermediate Language (CIL) que é interpretado pela máquina virtual Common Language Runtime (CLR).

CIL é uma linguagem de programação de baixo nível que é assemblado em código chamado bytecode.

CLR é a “máquina virtual” da plataforma .NET.

Compilação:


Visual C# Project
C# Source File(s)
Resources
References
C# Compiler
Creates
Managed Assembly (.exe or .dll)
MSIL Metadata
IL metadata & references loaded by CLR
.NET Framework
Common Language Runtime Security / Garbage Collection / JIT Compiler
Uses
.NET Framework Class Libraries
Converted to native machine code
Operating System

Confiabilidade

Linguagem investe bastante em confiabilidade:

- É fortemente tipada!
- Faz checagem de índice de vetores
- Detecta tentativas de uso de variáveis não inicializadas
- Possui coletor de lixo automático
- Possui diversas IDEs com compilação em tempo real para a checagem de erros de programação.
- Não possui Checked Exceptions
- Ponteiros

Outras Propriedades

Ortogonalidade: Diferentemente de Java, a linguagem possui sistema de tipos unificado, onde todos os tipos derivam de um tipo raiz (System.Object), assim todos os tipos implementam o método ToString() por exemplo.

Legibilidade: Melhor legibilidade que C e C++, ainda possui ponteiros, porém seu uso é desestimulado, aparecendo em poucos programas.

Reusabilidade e Modificabilidade: Linguagem é orientada a objetos, possui namespaces e encoraja o encapsulamento, assim o bom uso da linguagem torna o código bastante reusável e sujeito a modificações sem que seja necessário alterar outras partes do programa.

Redigibilidade: Não possui tanto foco em redigibilidade, apesar de possuir alguns recursos que podem auxiliar na mesma, como ponteiros para variáveis e expressões Lambda.

Facilidade de Aprendizado: Apesar de possuir certo nível de complexidade, assim como Java a linguagem é de fácil iniciação para programadores inexperientes, e principalmente para programadores C e C++.

Eficiência: A linguagem tem bom nível de otimização, porém na maioria das vezes prioriza a confiabilidade e modificabilidade do código, assim como o aproveitamento do tempo do programador, que passa a não focar-se tanto na otimização do código.

“Otimização prematura é a raiz de todo o mal” - Donald Knuth

Critérios	C++	Java	C#
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Parcial
Aprendizado	Não	Parcial	Sim
Eficiência	Sim	Parcial	Parcial
Portabilidade	Não	Sim	Sim
Método de Projeto	OO e Estruturado	OO	OO
Evolutibilidade	Parcial	Sim	Sim
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Parcial
Custo	Depende	Depende	Depende

Critérios	C++	Java	C#
Escopo	Sim	Sim	SIm
Expr. e Comandos	Sim	Sim	Sim
Tipos Prim. e Comp.	Sim	Sim	Sim
Gerenc. de Memória	Programador	Sistema	Sistema
Perist. de Dados	Biblioteca	Bibl., JDBC, Seriali.	Bilblioteca, Serializ.
Pass. de Param.	Valor, referencia, lista variavel e default	Lista variavel, por valor e por cópía de ref.	Valor, referencia, lista variavel e default
Encapsulamento	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Parcial
Verif. de tipos	Estatica/Dinamica	Estatica/Dinamica	Estatica/Dinamica
Polimosfimo	Todos	Todos	Todos

Critérios	C++	Java	C#
Exceções	Parcial	Sim	Parcial
Concorrência	Biblioteca	Sim	Sim

Referências.

https://pt.wikipedia.org/wiki/C_Sharp

https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

https://msdn.microsoft.com/en-us/library/kx37x362.aspx

https://www.caelum.com.br/apostila-csharp-orientacao-objetos/

Extras

Mais sobre expressões Lambda
https://msdn.microsoft.com/pt-br/library/bb397687.aspx
Mais sobre ponteiros e código unsafe
https://msdn.microsoft.com/en-us/library/t2yzs44b.aspx
Mais sobre tipos genéricos
https://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx
Mais sobre threading
https://www.tutorialspoint.com/csharp/csharp_multithreading.htm
Delegates
https://msdn.microsoft.com/en-us/library/ms173171.aspx