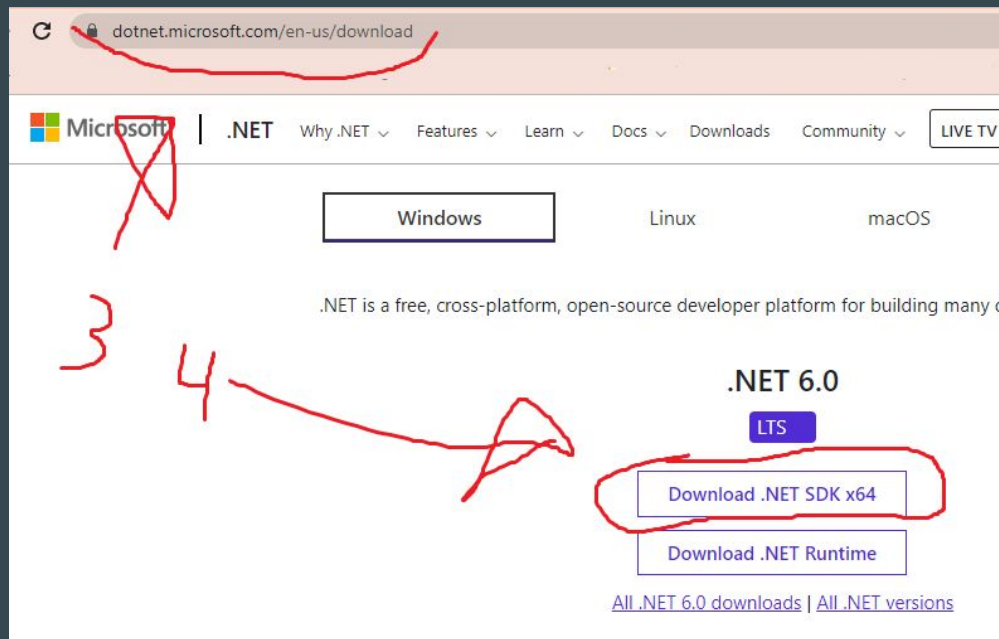
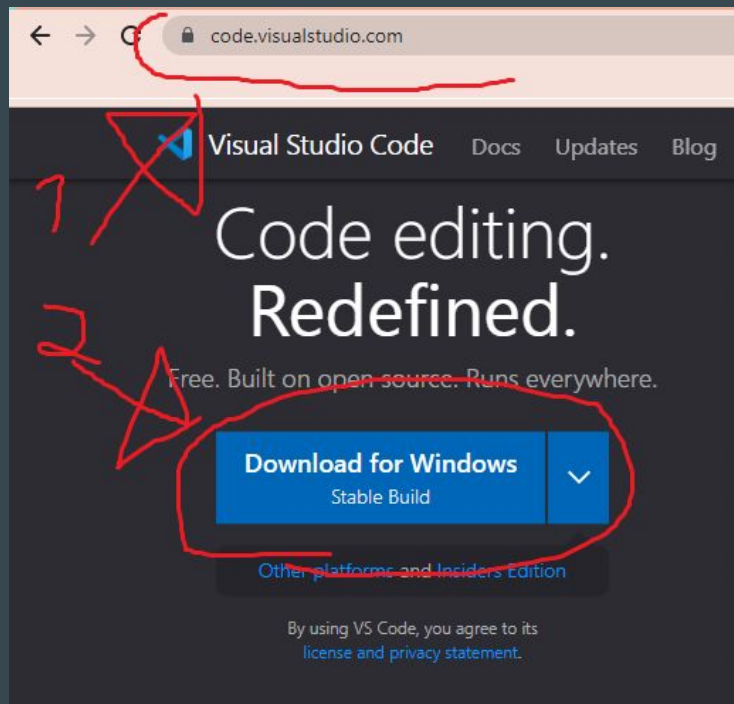


Análise Técnica C Hashtag

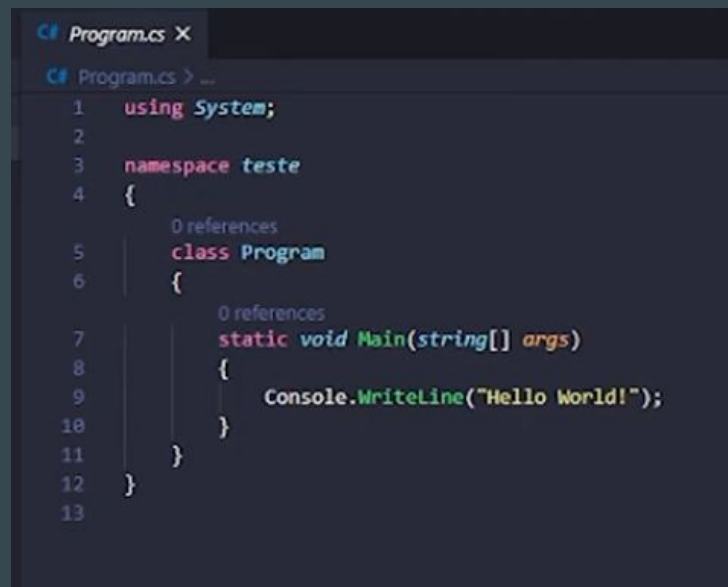
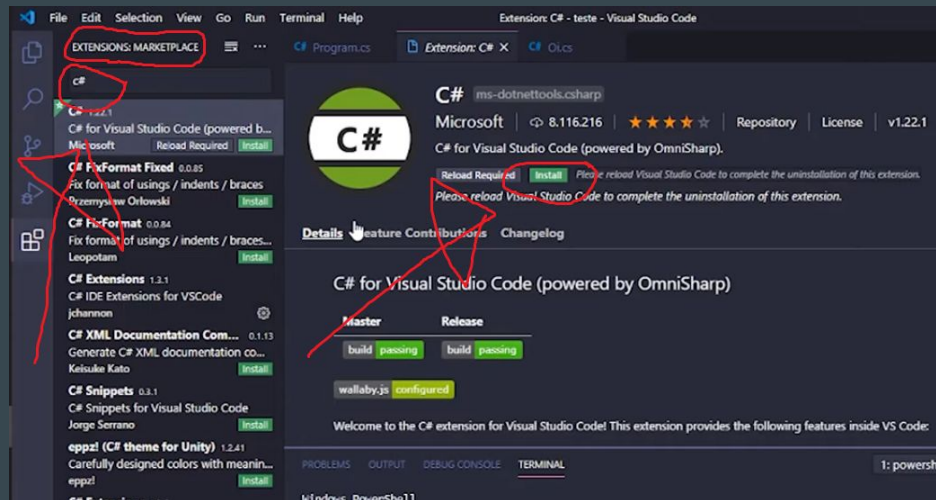
...

Alunos: Abraão Santos & Dionatas Brito

Instalando: Baixe e instale o VScode e o Dotnet



Configurando o **V**Scod**e**: instale a extensão de **C#** e agora é só rodar seu programa utilizando o comando **dotnet run**



Preliminares: Processos de tradução

Todas as linguagens do .NET são compiladas na mesma forma intermediária, Intermediate Language (IL). Diferentemente de Java, entretanto, a IL nunca é interpretada. Um compilador Just-in-Time é usado para converter IL em código de máquina antes de ela ser executada.



.NET

Preliminares: Paradigmas da linguagem

- Funcional
- Genérica
- Declarativa
- Orientada a objetos
- Imperativa



Nomes e Variáveis: Declarando uma variável

Assim como em C, em C# a regra para nomear uma variável é que o nome dela sempre comece por uma **letra** ou '_'. No meio do nome podem-se usar números, mas não se devem usar caracteres especiais e também não pode ser uma palavra reservada.

`int lnumero -> não pode`

`int numerol -> pode`

`int num_random -> pode`

Nomes e Variáveis: É *case-sensitive*?

Sim!!

```
int count;
```

```
int Count;
```

```
int cOunt;
```

São declarações de diferentes variáveis

Nomes e Variáveis: Tamanho dos nomes

- Em c#, não existe restrição de tamanho para nomes;



Nomes e Variáveis: Palavras especiais

- Palavras-chave x Palavras reservadas;

1	abstract	as	base	bool
2	break	byte	case	catch
3	char	checked	class	const
4	continue	decimal	default	delegate
5	do	double	else	enum
6	event	explicit	extern	false
7	finally	fixed	float	for
8	foreach	goto	if	implicit
9	in	int	interface	internal
10	is	lock	long	namespace
11	new	null	object	operator
12	out	override	params	private
13	protected	public	readonly	ref
14	return	sbyte	sealed	short
15	sizeof	stackalloc	static	string
16	struct	switch	this	throw
17	true	try	typeof	uint
18	ulong	unchecked	unsafe	ushort
19	using	virtual	void	volatile
20	while			

1	add	and	alias	ascending
2	async	await	by	descending
3	dynamic	equals	from	get
4	global	group	init	into
5	join	let	managed	nameof
6	nint	not	notnull	nuint
7	on	or	orderby	partial
8	record	remove	select	set
9	unmanaged	value	var	when
10	where	with	yield	

Nomes e Variáveis: Variáveis anônimas

- Em variáveis locais;
- Em instruções;

- Em uma instrução de inicialização `for`.

C#

```
for (var x = 1; x < 10; x++)
```

- Em uma instrução de inicialização `foreach`.

C#

```
foreach (var item in list) {...}
```

- Em uma instrução `using`.

C#

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

C#

```
// i is compiled as an int  
var i = 5;
```

```
// s is compiled as a string  
var s = "Hello";
```

```
// a is compiled as int[]  
var a = new[] { 0, 1, 2 };
```

```
// expr is compiled as IEnumerable<Customer>  
// or perhaps IQueryable<Customer>  
var expr =  
    from c in customers  
    where c.City == "London"  
    select c;
```

```
// anon is compiled as an anonymous type  
var anon = new { Name = "Terry", Age = 34 };
```

```
// list is compiled as List<int>  
var list = new List<int>();
```

Nomes e Variáveis: Categorias de Variáveis

- Variáveis estáticas;
- Variáveis dinâmicas da pilha;
- Variáveis dinâmicas do monte implícitas;

```
1 static void Main(string[] args)
2 {
3     //Declaração da lista
4     List<string> lista = new List<string>();
5
6     string opcao = "1";
7
8     while(opcao == "1")
9     {
10         Console.WriteLine("Digite um nome para inserir na lista:");
11         string nome = Console.ReadLine();
12         //Adiciona o item à lista
13         lista.Add(nome);
14     }
```

```
class Program
{
    public static void soma(int x, int y)
    {
        int soma;
        soma = x + y;
        Console.WriteLine("O resultado da soma é: " + soma);
    }
}
```

```
C#
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
0
5
99
*/
```

Nomes e Variáveis: ocultamento de nomes

Você pode ocultar uma variável sombreando-a , ou seja, redefinindo-a com uma variável de mesmo nome.

Sombreamento no escopo: Você pode sombreá-lo por meio do escopo, redeclarando-o dentro de uma sub-região da região que contém a variável que você deseja ocultar.

```
Module shadowByScope
    ' The following statement declares num as a module-level variable.
    Public num As Integer
    Sub show()
        ' The following statement declares num as a local variable.
        Dim num As Integer
        ' The following statement sets the value of the local variable.
        num = 2
        ' The following statement displays the module-level variable.
        MsgBox(CStr(shadowByScope.num))
    End Sub
    Sub useModuleLevelNum()
        ' The following statement sets the value of the module-level variable.
        num = 1
        show()
    End Sub
End Module
```

Nomes e Variáveis: ocultamento de nomes

Sombreamento por herança: você poderá sombreá-la por meio da herança, redeclarando-a com a palavra-chave **Shadows** em uma classe derivada.

```
Public Class shadowBaseClass
    Public shadowString As String = "This is the base class string."
End Class
Public Class shadowDerivedClass
    Inherits shadowBaseClass
    Public Shadows shadowString As String = "This is the derived class string."
    Public Sub showStrings()
        Dim s As String = "Unqualified shadowString: " & shadowString &
            vbCrLf & "MyBase.shadowString: " & MyBase.shadowString
        MsgBox(s)
    End Sub
End Class
```

Nomes e Variáveis: definição de constantes

Em C# o **#define não funciona**, e não tem algo equivalente, então para definir constantes, uma abordagem é agrupá-las em uma única classe estática de nome **Constants**. Isso exigirá que todas as referências às constantes sejam precedidas com o nome de classe.

```
using System;

static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

Tipos de dados: tipos oferecidos

C# Type

bool
byte
sbyte
char
decimal
double
float
int
uint
long
ulong
object
short
ushort
string

NET Framework type

System.Boolean
System.Byte
System.SByte
System.Char
System.Decimal
System.Double
System.Single
System.Int32
System.UInt32
System.Int64
System.UInt64
System.Object
System.Int16
System.UInt16
System.String

Exemplos:

```
bool flag;  
byte x;  
char c;  
float pi = 3,14;  
int y = 54;  
string str1 = "Ola mundo";
```

Tipos de dados: Vinculações (tipagem)

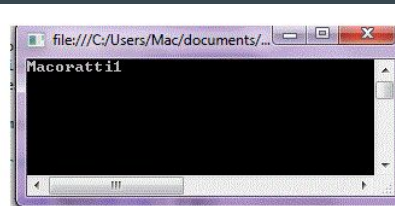
- C# usa tipagem Estática (tempo de compilação);
- Pode habilitar a tipagem dinâmica (tempo de execução);

```
// código C#  
static void Main(string[] args)  
{  
    object teste = "Macoratti";  
    teste += 1;  
    Console.WriteLine(teste);  
    Console.ReadKey();  
}
```

```
class Program  
{  
    static void Main(string[] args)  
    {  
        object teste = "Macoratti";  
        teste += 1;  
    }  
}
```

Operator '+' cannot be applied to operands of type 'object' and 'int'

```
// código C# 4.0  
static void Main(string[] args)  
{  
    dynamic teste = "Macoratti";  
    teste += 1;  
    Console.WriteLine(teste);  
    Console.ReadKey();  
}
```



Tipos de dados: Ponteiros e Referências

- ponteiros e referências;
- Aritmética de ponteiros;

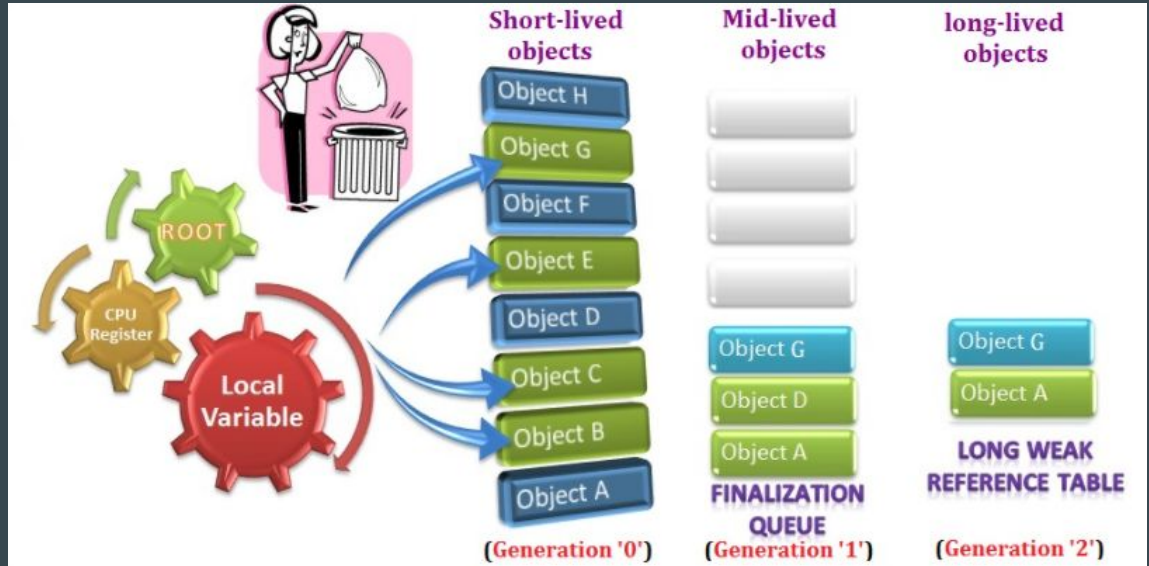
C#

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

Tipos de dados: Lixo de memória (garbage)

- GC - Garbage Collector;
- Finalise() e Dispose();
- 3 Gerações;



Tipos de dados: Verificação de Tipos

- C# é considerada quase fortemente tipada;

```
static void Main(string[] args)
{
    double a = 3;
    double b = 4;
    int c = a + b;
    Console.WriteLine(c);
    Console.ReadKey();
}
```

```
static void Main(string[] args)
{
    double a = 3;
    double b = 4;
    int c = (int)(a + b);
    Console.WriteLine(c);
    Console.ReadKey();
}
```

Tipos de dados: Func, Action, e Predicate

No C# você pode utilizar um delegate Func, Action ou Predicate para tipar uma função anônima.

Action é uma Func que não terá um retorno.

Predicate é uma Func que retorna um bool.

```
var operacoes = new Dictionary<string, Func<int, int, int>> {  
    {"+", (op1, op2) => op1 + op2 },  
    {"-", (op1, op2) => op1 - op2 },  
    {"*", (op1, op2) => op1 * op2 },  
    {"/", (op1, op2) => op1 / op2 }  
};  
Write(operacoes["+"](10, 20)); //imprime 30
```

```
var compareZero = new Dictionary<string, Predicate<int>> {  
    {">", (x) => x > 0 },  
    {"<", (x) => x < 0 },  
    {"=", (x) => x == 0 }  
};  
Write(compareZero["="](5)); //imprimirá False
```

```
var acoes = new Dictionary<string, Action<int>> {  
    {"Criar", (parametro) => Criar(parametro) },  
    {"Editar", (parametro) => Editar(parametro) },  
    {"Apagar", (parametro) => Apagar(parametro) },  
    {"Imprimir", (parametro) => Imprimir(parametro) }  
};  
acoes["Criar"](1); //executará o método Criar
```

Operadores: Atribuição como expressão

A expressão **condition** deve ser avaliada para true ou false. Se **condition** for avaliada como true, a expressão **consequent** será avaliada e seu resultado se tornará o resultado da operação. Se **condition** for avaliada como false, a expressão **alternative** será avaliada e seu resultado se tornará o resultado da operação.

```
condition ? consequent : alternative
```

```
var rand = new Random();  
var condition = rand.NextDouble() > 0.5;  
  
var x = condition ? 12 : (int?)null;
```

Operadores: Operadores Oferecidos

- Operador de Atribuição;
- Operadores Aritméticos de Atribuição Reduzida;
- Operadores Aritméticos;
- Operadores incrementais e decrementais;

```
int a = 10;  
int b = a;
```

```
int x = 5;  
x += 5; // é a mesma coisa que x = x + 5  
Console.WriteLine("Valor do x = " + x);  
Console.ReadKey();;
```

```
int a = 5, b = 10, c = 15, d = 20; // declaramos quatro variáveis do tipo int  
Console.WriteLine(a + d); // operação de soma  
Console.WriteLine(c - a); // operação de subtração  
Console.WriteLine(b * c); // operação de multiplicação  
Console.WriteLine(d / b); // operação de divisão  
Console.WriteLine(c % b); // operação de módulo (resto de divisão)  
Console.ReadKey();
```

```
Console.WriteLine("\n++x +20 = \n");  
Console.WriteLine(++x + 20 + "\n");
```

```
Console.WriteLine("\n--x +20 = \n");  
Console.WriteLine(--x + 20 + "\n");
```

Operadores: Operadores Oferecidos

- Operadores Relacionais;
- Operadores Lógicos;
- Operadores Ternários;

```
int a = 10, b = 25, c = 50, d = 100; // declaramos quatro variáveis de tipo int

Console.WriteLine(a == d); // avaliamos a igualdade entre a e d
Console.WriteLine(b != c); // avaliamos a desigualdade entre b e c
Console.WriteLine(a > b); // avaliamos se a é maior que b
Console.WriteLine(c < d); // avaliamos se c é menor que d
Console.WriteLine(c >= a); // avaliamos se c é maior ou igual que a
Console.WriteLine(d <= b); // avaliamos se d é menor ou igual que b

Console.ReadKey();
```

```
int a = 5, b = 10, c = 15, d = 20; // declaramos quatro variáveis do tipo int

Console.WriteLine(a == 5 && d == 10); // avaliamos se a é igual a 5 e se d é igual a 10
Console.WriteLine(c < b || d == 20); // avaliamos se c é menor que b ou se d é igual a 20
Console.WriteLine(!(b > a)); // negamos que b é maior que a

Console.ReadKey();
```

```
int x = 5, y = 10; // declaradas duas variáveis de tipo int

Console.WriteLine(x < y ? "sim" : "não"); // expressão x < y é avaliada
```

Operadores: Sobrecarga de operadores

- Operadores Unários (+, -, !, ~, ++, --);
- ❖ static - O método sobrecarregado do operador
- ❖ operator - é uma palavra-chave
- ❖ op - usa um símbolo especial do operador
- ❖ Type - o tipo

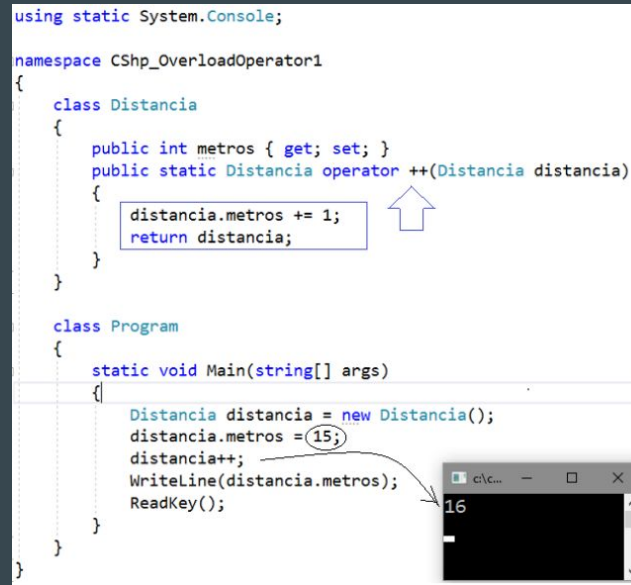
Sintaxe:

```
public static return_type operator op (Type t)
{
    // TODO:
}
```

```
using static System.Console;

namespace CShp_OverloadOperator1
{
    class Distancia
    {
        public int metros { get; set; }
        public static Distancia operator ++(Distancia distancia)
        {
            distancia.metros += 1;
            return distancia;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Distancia distancia = new Distancia();
            distancia.metros = 15;
            distancia++;
            WriteLine(distancia.metros);
            ReadKey();
        }
    }
}
```



The screenshot shows a C# program that demonstrates operator overloading. It defines a class named 'Distancia' with a public integer property 'metros'. An overloaded increment operator '++' is implemented as a static method that takes a 'Distancia' object and increments its 'metros' property by 1, then returns the object. In the 'Main' method, a 'Distancia' object is created, its 'metros' property is set to 15, and then the '++' operator is used to increment it. The output shows the value 16.

Operadores: Sobrecarga de operadores

- Operadores Binários (+, -, *, /, %, &, |, ^, <<, >>);

❖ Type1 - t1 : é o operador do lado esquerdo

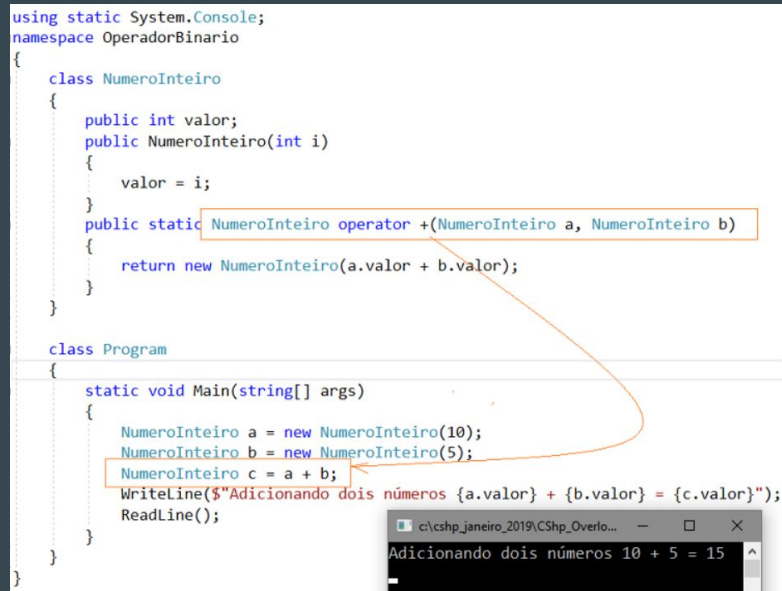
❖ Type2 - t2 : é o operador do lado direito

Sintaxe:

```
public static return_type operator op (Type1 t1, Type2 t2)
{
    // TODO:
}
```

```
using static System.Console;
namespace OperadorBinario
{
    class NumeroInteiro
    {
        public int valor;
        public NumeroInteiro(int i)
        {
            valor = i;
        }
        public static NumeroInteiro operator +(NumeroInteiro a, NumeroInteiro b)
        {
            return new NumeroInteiro(a.valor + b.valor);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            NumeroInteiro a = new NumeroInteiro(10);
            NumeroInteiro b = new NumeroInteiro(5);
            NumeroInteiro c = a + b;
            WriteLine($"Adicionando dois números {a.valor} + {b.valor} = {c.valor}");
            ReadLine();
        }
    }
}
```



Operadores: Sobrecarga de operadores

- Operadores de Comparação (==, !=, <, >, <=, >=);

Sintaxe:

```
public static bool operator op (Type1 t1, Type2 t2)
{
    // TODO:
}
```

```
public class Distancia
{
    public double metros { get; set; }

    public static bool operator <(Distancia d1, Distancia d2)
    {
        return (d1.metros < d2.metros);
    }
    public static bool operator >(Distancia d1, Distancia d2)
    {
        return (d1.metros > d2.metros);
    }
}
```

using static System.Console;

```
class Program
{
    static void Main(string[] args)
    {
        Distancia d1 = new Distancia { metros = 149.600 };
        Distancia d2 = new Distancia { metros = 227.940 };
        if (d1 < d2)
        {
            WriteLine($"{d1.metros} é menor que {d2.metros}");
        }
        else if (d1 > d2)
        {
            WriteLine($"{d2.metros} é menor que {d1.metros}");
        }
    }
}
```

Operadores: Curto Circuito

- Operadores AndAlso (&&) e OrElse (||).

Operador	Descrição
&& (E)	Se duas expressões condicionais forem verdadeiras o resultado é verdadeiro.
(OU)	Se qualquer expressão condicional for verdadeira o resultado é verdadeiro.

Tipo Abstrato de Dados (TAD): Suporte

- Encapsulamento via tad de classes;
- Usuário tem restrição a todo acesso;

Ex: Tad Pessoa ->

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  namespace P00
7  {
8      public class Pessoa
9      {
10         public Pessoa(string cabelo)//Valor obrigatório
11         {
12             Olhos = 2;//Valor default
13             Bracos = 2;
14             Pernas = 2;
15             CorCabelo = cabelo;
16         }
17         public Pessoa() {
18         }
19         public string Nome { get; set; }
20         public int Olhos { get; set; }
21         public string CorCabelo { get; set; }
22         public int Bracos { get; set; }
23         public int Pernas { get; set; }
24     }
25 }
```

Tipo Abstrato de Dados (TAD) - Genéricos/Parametrizados

- Coleções armazenam objetos de qualquer classe;
- Coleções genéricas acessadas por índices;
- Coleções genéricas predefinidas:
 - Array
 - List
 - Stack
 - Queue
 - Dictionary

```
C# Copie  
  
// Declare the generic class.  
public class GenericList<T>  
{  
    public void Add(T input) { }  
}  
class TestGenericList  
{  
    private class ExampleClass { }  
    static void Main()  
    {  
        // Declare a list of type int.  
        GenericList<int> list1 = new GenericList<int>();  
        list1.Add(1);  
  
        // Declare a list of type string.  
        GenericList<string> list2 = new GenericList<string>();  
        list2.Add("");  
  
        // Declare a list of type ExampleClass.  
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();  
        list3.Add(new ExampleClass());  
    }  
}
```

Tipo Abstrato de Dados (TAD) - Encapsulamento

- Public;
- Private;
- Protected;

C#

```
class Employee
{
    private int _i;
    double _d; // private access by default
}
```

C#

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

C#

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

Tipo Abstrato de Dados (TAD) - Encapsulamento

- Internal;
- Protected Internal;
- assembly = arquivos (.exe ou .dll);

C#

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

Polimorfismo: Tipos

- Ad-hoc
 - Sobrecarga;
 - Coerção (Ampliação e Estreitamento);

```
int val = 24;  
long testeLong = valor;  
  
float fl = 24.0f;  
double novoTesteDouble = fl;
```

```
double db = 23.9;  
float testeFloat = db; ✗  
  
float testeFloat = (float) db; ✓
```

```
static long Somar(long x, long y)  
{  
    return x + y;  
}
```

```
static int Somar(int x, int y)  
{  
    return x + y;  
}
```

```
static double Somar(double x, double y)  
{  
    return x + y;  
}
```

```
static void Main(string[] args)  
{  
    Console.WriteLine("***** Sobrecarga de Métodos *****\n");  
  
    Console.WriteLine(Somar(10, 10));  
    Console.WriteLine(Somar(3.5, 7.4));  
    Console.WriteLine(Somar(8000000000000, 7000000000000));  
}
```


Polimorfismo: Tipos

- Universal
 - Paramétrico (linguagem mais expressiva);
 - Inclusão (substituição de uma classe mãe x descendente);

```
// List<T>
List<int> numeros = new List<int>();

// Dictionary<TKey, TValue>
var palavras = new Dictionary<string, int>();
```

```
public class Conta {}
public class Poupança : Conta {}
```

```
List<Conta> contas = new
List<Conta>();
contas.Add(new Conta());
contas.Add(new Poupança());
```

```
var d = new Poupança();
var e = new Conta();
bool b;

b = d is Conta; // b = true;
b = d is Object; // b = true;
b = e is Poupança; // b = false;
```

Estruturas de Controle

- **Estruturas de seleção:**

- if if – else if – else – if
- switch – case

- **Estruturas de repetição:**

- while
- do – while
- for
- foreach

```
do{  
    bloco de comandos  
}while( expressão booleana );
```

```
while ( expressão booleana ){  
    bloco de comandos  
}
```

```
List<int> dados = new List<int>( );  
dados.Add(1);  
dados.Add(2);  
int valor = 0;  
foreach (int a in dados){  
    valor += a;  
}  
meuLabel.Text = valor.ToString( );
```

Sentenças de Desvio Incondicional: goto e break

```
private static int Recurso1_Goto()
{
    int conta = 0;
    for (int a = 0; a < 10; a++)
    {
        for (int y = 0; y < 10; y++)
        {
            for (int x = 0; x < 10; x++)
            {
                if (x == 5 && y == 5)
                {
                    goto Saida;
                }
            }
            conta++;
        }
        Saida:
        continue;
    }
    return conta;
}
```

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
```

Sentenças de Desvio Incondicional: continue e return

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
```

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
```

Passagem de parâmetros: por valor e Out

```
using System;
class TesteValue
{
    static void Metodo(int Param1)
    {
        Param1 = 100;
        // Aqui a saída será 100
        Console.WriteLine("Valor em Metodo = " + Param1);
    }

    static void Main()
    {
        int valor = 5;
        Metodo(valor);
        // Aqui a saída será 5, perceba que o valor não foi modificado
        Console.WriteLine("Valor na Main = " + valor);
        Console.Read();
    }
}
```

```
using System;
class TesteOut
{
    static void Metodo(out int Param1)
    {
        Param1 = 100;
        Console.WriteLine("Valor em Metodo = " + Param1);
    }

    static void Main()
    {
        int valor = 5;
        Metodo(out valor);
        Console.WriteLine("Valor na Main = " + valor);
        Console.Read();
    }
}
```

Passagem de parâmetros: ref e Params

```
using System;

class TesteRef
{
    static void Metodo(ref int Param1)
    {
        Param1 = Param1 + 100;
    }

    static void Main()
    {
        int valor = 5;
        Metodo(ref valor);
        Console.WriteLine("Valor na Main = " + valor);
        Console.Read();
    }
}
```

```
using System;

class TesteParams
{
    static int Soma(params int[] Param1)
    {
        int val = 0;
        foreach(int P in Param1)
        {
            val = val + P;
        }
        return val;
    }

    static void Main()
    {
        Console.WriteLine(Soma(10, 20, 30));
        Console.WriteLine(Soma(10, 20, 30, 40, 50));
        Console.Read();
    }
}
```

Sobrecarga de Subprogramas

O C# permite que nós tenhamos vários métodos com o mesmo nome dentro de uma mesma classe, mas desde que estes métodos tenham diferentes conjuntos de parâmetros, sejam eles números, simplesmente a ordem dos parâmetros ou então tipos diferentes.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        public static int Quadrado(int x)
        {
            return x * x;
        }

        public static double Quadrado(double y)
        {
            return y * y;
        }
    }
}
```

Subprogramas Genéricos

A forma de sintaxe para a declaração usa os caracteres <T> após o nome do método mas antes da lista de parâmetro formal.

```
static void Trocar<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```


Herança de Tipos: Simples, hierárquica, multinível

```
class A
{
    public void Exibir()
    {
        Console.WriteLine("Método da classe A");
    }
}
class B : A // A classe B deriva da classe A
{
    public void Mostrar()
    {
        Console.WriteLine("Método da classe B");
    }
}
class C : A // A classe C é derivada da classe A
{
    public void Apresentar()
    {
        Console.WriteLine("Método da Classe C");
    }
}
```

```
public class Conta //classe base
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }
}

public class ContaPoupanca : Conta
//classe derivada
{
    public int JurosMensais { get; set; }
}
```

```
class A
{
    public void Exibir()
    {
        Console.WriteLine("Método da classe A");
    }
}
class B : A // A classe B deriva da classe A
{
    public void Mostrar()
    {
        Console.WriteLine("Método da classe B");
    }
}
class C : B // A classe C é derivada da classe B
{
    public void Apresentar()
    {
        Console.WriteLine("Método da Classe C");
    }
}
```

Herança de Tipos: Múltipla

A linguagem C# não suporta herança múltipla de classes.

Para superar esse problema, podemos usar interfaces onde uma classe pode implementar mais de uma interface ou de uma classe e de uma ou mais de uma interface.

```
class Forma
{
    public void setLado(int s)
    {
        lado = s;
    }
    protected int lado;
}

interface ICusto
{
    int getCusto(int area);
}

class Quadrado : Forma, ICusto
{
    public int getArea()
    {
        return (lado * lado);
    }

    public int getCusto(int area)
    {
        return area * 10;
    }
}
```

Sobrescrita de subprogramas

Dado a classe garrafa e o método finalidade a seguir:

Para executar então a sobrescrita do método Finalidade que está no objeto pai, utilizamos o override.

Ainda podemos, caso necessário, chamar o método Finalidade do objeto pai

```
public class Garrafa
{
    public virtual void Finalidade()
    {
        Console.WriteLine("Garrafa genérica");
    }
}
```

```
public class GarrafaTermica : Garrafa
{
    public override void Finalidade()
    {
        Console.WriteLine("Manter a temperatura");
    }
}
```

```
public class GarrafaTermica : Garrafa
{
    public override void Finalidade()
    {
        base.Finalidade();
        Console.WriteLine("Manter a temperatura");
    }
}
```

Subprogramas Abstratos

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}

// Output: Area of the square = 144
```

Exceções: Mecanismo de controle de exceções

try-catch

finally

throw

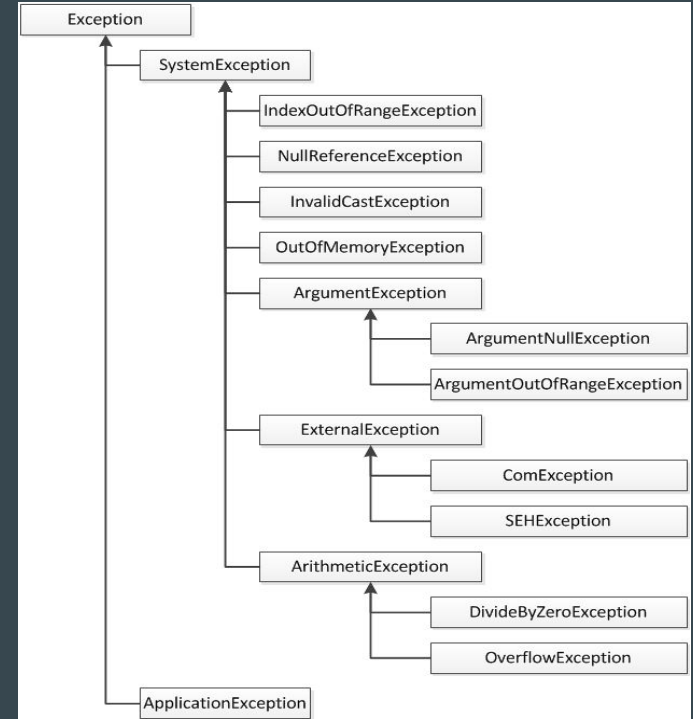
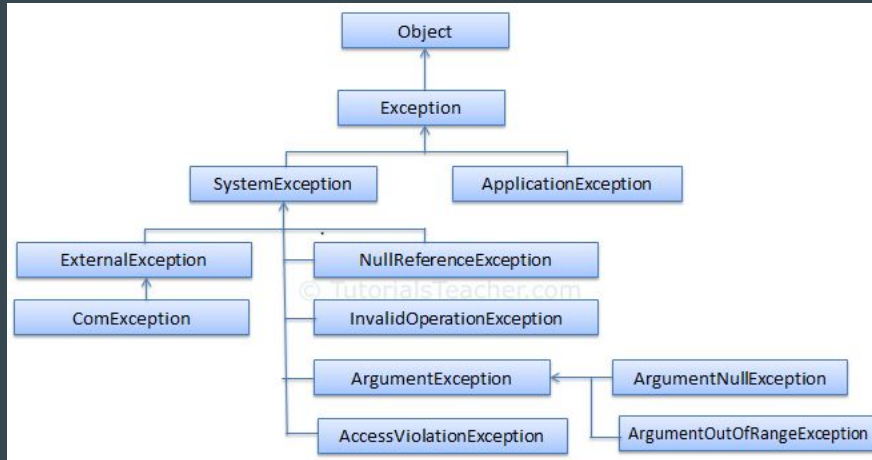
```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

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

Exceções: Hierarquia

- System.Exception;
- Raiz: Exception (herda de Object);



Exceções: Checadas e Não checadas;

- Na linguagem `c#` as exceções não são checadas;

O problema com exceções verificadas Uma conversa com Anders Hejlsberg, Parte II

por Bill Venners com Bruce Eckel
18 de agosto de 2003

Resumo

Anders Hejlsberg, o arquiteto líder de `C#`, conversa com Bruce Eckel e Bill Venners sobre questões de versão e escalabilidade com exceções verificadas.

Permanecendo Neutro em Exceções Verificadas

Bruce Eckel : `C#` não tem exceções verificadas. Como você decidiu colocar ou não exceções verificadas em `C#`?

Anders Hejlsberg : Vejo dois grandes problemas com exceções verificadas: escalabilidade e capacidade de versão. Eu sei que você também escreveu sobre exceções verificadas e tende a concordar com nossa linha de pensamento.

Exceções: Exceções lançadas

- Em c#, é obrigatório a declaração de exceções lançadas;

```
C#
class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "parameter can't be null.");
        }
    }

    public static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
   at TryFinallyTest.Main() Exception caught.
* */
```


Avaliação da linguagem (Sebesta): Legibilidade

- **Simplicidade geral:** Há mais de uma maneira para realizar uma mesma operação e o uso de sobrecarga de operadores;
- **Ortogonalidade:** Permite a utilização de ponteiros resultando em um leque maior de possibilidades de implementações;
- **Tipos de dados:** Possui uma grande possibilidade de tipos de dados (booleano, string..);
- **Projeto da sintaxe:** Com o uso de palavras especiais da linguagem (while, class, for..), C# garante legibilidade;

Avaliação da linguagem (Sebesta): Facilidade de escrita

- **Simplicidade e Ortogonalidade:** Possui um número menor de construções primitivas e um conjunto de regras consistente para combiná-las;
- **Expressividade:** Está correlacionada com operadores poderosos que permitem muitas computações com um programa muito pequeno (tipo ArrayList...), que contém um conjunto de operações complexas já implementadas, onde em outras linguagens seria necessário uma implementação

Avaliação da linguagem (Sebesta): Confiabilidade

- **Verificação de tipos:** em C# só é permitido erros de tipos explícitos no programa.
- **Tratamento de exceções:** o tratamento de exceções é sim algo muito presente em C#, contendo um suporte muito bom e amplo para o programador.
- **Apelidos:** Como C# existe referência e ponteiros, ela perde uns pontos nesse quesito.
- **Legibilidade e facilidade de escrita:** Como já citado anteriormente, legibilidade e facilidade de escrita é um ponto positivo para o C#.

Avaliação da linguagem (Sebesta): Custo

- **Treinar programadores:** Como `c#` é uma linguagem de fácil escrita, o custo do treino de novos programadores seria menor que boa parte das outras linguagens.
- **Escrever programas:** Claramente como `c#` é uma linguagem com paradigma orientado a objeto, seria muito menos custoso fazer um software voltado para esse tema.
- **Compilar programas:** Como já foi visto, o VScode é uma plataforma gratuita para montar projetos e compilar.

Avaliação da linguagem (Sebesta): Custo

- **Executar programas:** o conceito mais importante hoje é otimização, que é o nome dado para um conjunto de técnicas que os compiladores usam para diminuir o tamanho e/ou aumentar a velocidade do código que produzem.
- **Custo do sistema de implementação:** Como já dito antes, existe um software gratuito que permite a implementação em c#.
- **Custo de uma confiabilidade baixa:** como c# tem um certo nível de confiabilidade, é possível dizer que esse custo seria baixo.
- **Custo da manutenção de programas:** novamente dependente da legibilidade, que já foi visto que c# perde uns pontos nesse aspecto, levando a crer que o custo da manutenção pode ser um pouco mais elevado.

Avaliação da Linguagem (Varejão): Critérios Gerais

Critérios Gerais	C#
Aplicabilidade	Parcial
Confiabilidade	Parcial
Facilidade de Aprendizado	Parcial
Eficiência	Parcial
Portabilidade	Sim
Suporte ao Método de Projeto	Orientada a Objetos
Evolutibilidade	Sim
Reusabilidade	Sim
Integração	Sim
Custo	Dependente da Ferramenta de Desenvolvimento (No caso de VScode é grátis)

Avaliação da Linguagem (Varejão): Critérios Específicos

Critérios de Avaliação:	C#
Escopo	Sim
Expressões e Comandos	Sim
Tipos Primitivos e Compostos	Sim
Gerenciamento de Memória	Garbage Collector
Persistência de Dados	Biblioteca de Classes
Passagem de Parâmetros	Valor, referência, out e Params
Encapsulamento e Proteção	Sim
Sistema de Tipos	Sim
Verificação de Tipos	Estática e Dinâmica
Polimorfismo	Sim
Exceções	Sim
Concorrência	Parcial