LINGUAGEM

APRESENTAÇÃO

APRESENTAÇÃO

- Criada por Anders Hejlsberg e meados dos anos 90.
- Inicialmente se chamaria COOL
- Faz parte do Framework .NET
- Parecida com C++ e Java.

APRESENTAÇÃO

- Linguagem simples, robusta, orientada a objetos, fortemente tipada e altamente escalável.
- C# é padronizado seguindo o ***ECMA-334*** e a ***iso/IEC 23270*** .
- O código fonte é compilado para Common Intermediate Language (CIL), este é interpretado pela máquina virtual Common Language Runtime (CLR)

OBJETIVOS DA LINGUAGEM

- A linguagem destina-se a ser simples.
- A linguagem deve ser fortemente tipada.
- A portabilidade é muito importante.
- Deve ser adequada para escrever aplicações de todos os tipos.
- Não foi concebida para competir diretamente no desempenho e tamanho com C ou linguagem Assembly.

VALORES E TIPOS DE DADOS

TIPOS PRIMITIVOS

sbyte	inteiros com sinal de 8 bits com valores entre -128 e 127
byte	inteiros de 8 bits sem sinal com valores entre 0 e 255
short	inteiros de 16 bits com valores entre -32768 e 32767
ushort	inteiros de 16 bits sem sinal com valores entre 0 e 65535
int	inteiros de 32 bits com valores entre -2147483648 e 2147483647
uint	inteiros de 32 bits sem sinal com valores entre 0 e 4294967295

TIPOS PRIMITIVOS

long	inteiros de 64 bits com valores entre -9223372036854775808 e 9223372036854775807.
ulong	inteiros sem sinal de 64 bits com valores entre 0 e 18446744073709551615.
char	inteiros de 16 bits sem sinal com valores entre 0 e 65535.
float	valores que variam de aproximadamente 1.5 * 10^-45 para 3.4 * 10^38 com uma precisão de 7 dígitos.

TIPOS PRIMITIVOS

double	representar valores que variam de aproximadamente 5.0 * 10^-324 para 1.7 × 10^308 com uma precisão de 15-16 dígitos.
bool	representa quantidades lógicas booleanas. Os valores possíveis do tipo bool estão true e false.
decimal	pode representar valores variando 1.0 * 10^-28 para aproximadamente 7.9 * 10^28 com 28 a 29 dígitos significativos.

TIPOS COMPOSTOS - ENUMERADO

A palavra-chave utilizada para criar uma enumeração é enum.

```
enum Genero {
    Aventura,
    Drama,
    Romance,
    Suspense,
    Terror
};

enum Day {Sat, Sun, Mon, Tue,
Wed, Thu, Fri};
```

TIPOS COMPOSTOS - STRUCT

O tipo struct é um tipo de valor normalmente usado para encapsular pequenos grupos de variáveis relacionadas, tais como coordenadas de um retângulo ou as características de um item em um inventário.

```
public struct Livro{
    public decimal preco;
    public string titulo;
    public string autor;
}
```

TIPOS COMPOSTOS - MATRIZ

É possível armazenar diversas variáveis do mesmo tipo em uma matrizes.

```
int[] matriz = { 1, 2, 3, 4, 5, 6 };

int[,] multiDimensionalMatriz = { { 1, 2, 3 }, { 4, 5, 6 } };

int[][] matrizDenteada = new int[6][];
```

TIPOS DE REFERÊNCIA

- Há duas classificações para tipos em C#: tipos de referência e tipos de valor.
  - Variáveis de tipos de referência armazenam referências em seus dados (objetos).
  - Variáveis de tipos de valor contém diretamente seus dados.

TIPOS DE REFERÊNCIA - CLASS

- Declaradas diretamente dentro de um namespace.
- Os membros da classe, podem ser públicos, internos protegidos, protegidos, internos, privados ou protegidos privados.
- É possível declarar classes genéricas e abstratas.

```
class Fruta {
    private string nome;

    public Fruta(string nome){
        this.nome = nome;
    }
}
```

TIPOS DE REFERÊNCIA - INTERFACE

- A interface especifica os membros que devem ser fornecidos por estruturas que a implementam.
- Classes e structs podem implementar interfaces.

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

public class Quadrado:IForma{
    public void Desenhar(){}
}
```

AMARRAÇÕES

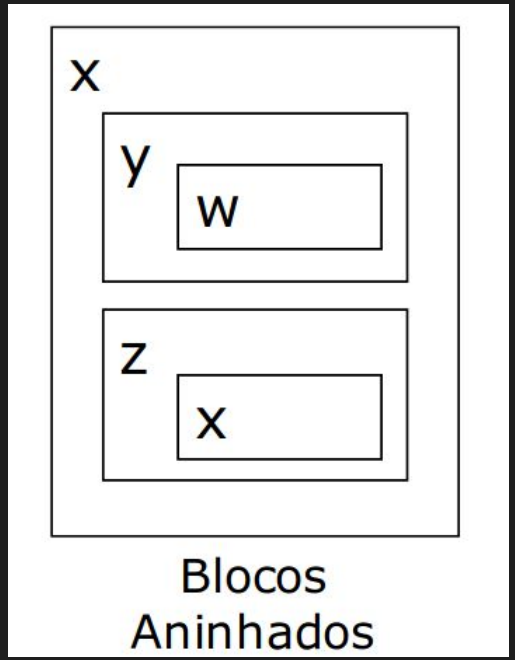
TIPAGEM

- Possui tipagem estática.
- Pode habilitar tipagem dinâmica com o uso da palavra chave dynamic.

```
dynamic x = 24;
x = "It's-a Me, Mario!";
x = false;
```

ESCOPO

- Escopo (Algol-Like): possui blocos aninhados.
- Dentro de um namespace só pode ser declarados structs, classes, enum, interface e delegates.
- Não possui variáveis globais. Para “burlar” isso deve ser criada uma classe estática.



MODIFICADORES DE ACESSO

- **public**: O acesso não é restrito.
- **protected**: O acesso é limitado à classe que os contém ou aos tipos derivados da classe que os contém.
- **internal**: O acesso é limitado ao assembly atual.
- **private**: O acesso é limitado ao tipo recipiente.

IDENTIFICADORES

- Os identificadores devem começar com uma letra ou _.
- Os identificadores podem conter caracteres Unicode.
- É possível declarar identificadores que correspondem às palavras-chave usando o prefixo @ no identificador.
- Case sensitive.

```
idade
_endereco
nomeCliente
Telefone2
@if
nomeAluno NomeAluno
```

IDENTIFICADORES - CONVENÇÃO DE NOMENCLATURA

- Os nomes de interface começam com I maiúsculo.
- Usar um nome de tipo singular para uma enumeração.
- Os identificadores não devem conter dois caracteres _ consecutivos.
- PascalCasing, usada para todos os identificadores e camelCasing usada para os nomes de parâmetro.
- Usar nomes de parâmetro descritivo.

IDENTIFICADORES - KEYWORDS

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	default	else	enum
event	explicit	extern	false	finally	fixed	float
for	foreach	goto	if	implicit	in	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte	sealed
short	sizeof	static	string	struct	switch	this

VARIÁVEIS E CONSTANTES

VARIÁVEIS

- C# usa especificação explícita assim como C, Java, etc...
- Suporta o uso de variáveis anônimas.
- O operador *new* indica uma alocação dinâmica, fora isso, tende a ser estática.

VARIÁVEIS

- Assim como em outras linguagens imperativas, C# também possui o conceito de variáveis.
- É possível definir uma variável indicando seu tipo seguido por um identificador.

```
int a;
bool b = t;
Pessoa p;
Livro l = new Livro();
```

VARIÁVEIS

C # 3.0 introduziu a tipagem implícita para variáveis locais *"var"*.

```
int i = 100; // tipagem explícita
var j = 100; // tipagem implícita
```

VARIÁVEIS

Variáveis de tipo implícito devem ser inicializadas no momento da declaração.

```
var i = 100; // Valido
var j; // oi.cs(32,13): error CS0818: An implicitly typed local
variable declarator must include an initializer

var a = "aa";
a = 1;//oi.cs(33,17): error CS0029: Cannot implicitly convert type
`int' to `string'
```

VARIÁVEIS

Em C#, não se pode criar variáveis globais “verdadeiras”.

```
public static class Globals{
    // Constante
    public const Int32 BUFFER_SIZE = 512;
    // Nao constante
    public static String FILE_NAME = "Output.txt";
    // Somente leitura
    public static readonly String CODE_PREFIX = "US-";
}
```

CONSTANTES

Existem “dois tipos” de constantes, os que são declarados com a *keyword const*, e outros com *readonly*.

```
public static class Globals{
    // Constante
    public const Int32 BUFFER_SIZE = 512;
    // Nao constante
    public static String FILE_NAME = "Output.txt";
    // Somente leitura
    public static readonly String CODE_PREFIX = "US-";
}
```

MANIPULAÇÃO DE PONTEIROS

Em C#, normalmente não é possível manipular diretamente o endereço de memória de variáveis.

Isso se torna possível através do modificador de bloco Unsafe, e com a passagem do parâmetro de compilação -unsafe ou com o bloco *<AllowUnsafeBlocks>* no MSBuild (Plataforma de compilação para Microsoft e Visual Studio).

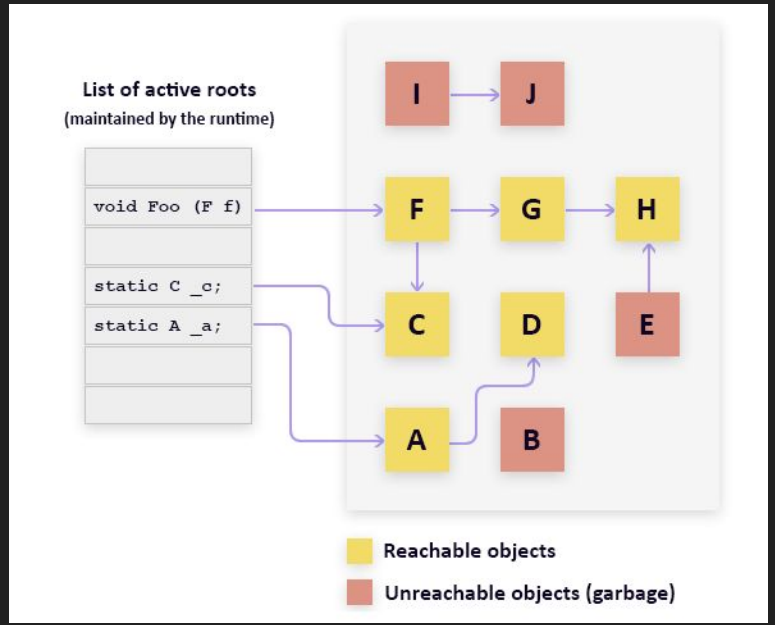
```
<PropertyGroup>
   <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

MANIPULAÇÃO DE PONTEIROS

```
unsafe {
    int x = 100;
    int *ptr = &x;
    Console.WriteLine((int)ptr); // Printa endereço de memória
    Console.WriteLine(*ptr);     // Printa o valor na memória
    Console.ReadLine();
}
```

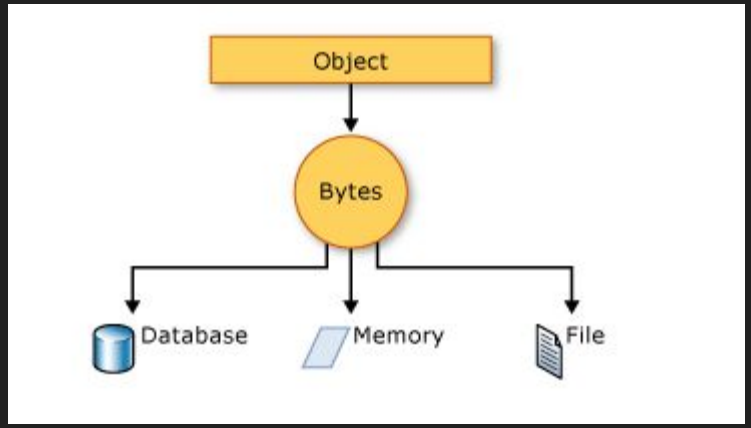
COLETA DE LIXO

- Quando o lixo é coletado, apenas objetos que não estão mais em uso pela aplicação são liberados.
- C#, assim como JAVA, utiliza a coleção de lixo geracional.



SERIALIZAÇÃO

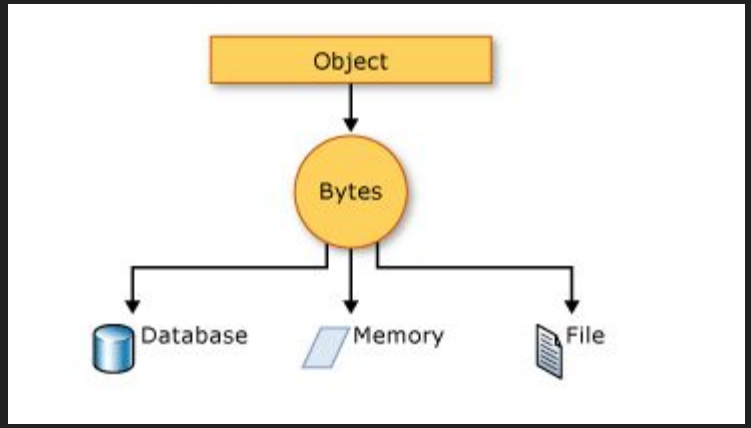
Muitas vezes, precisamos armazenar objetos em um armazenamento físico para que possam ser lidos novamente e convertidos em um objeto.



SERIALIZAÇÃO

Serialização binária vs XML

Performance vs flexibilidade/legibilidade



SERIALIZAÇÃO

```
// Definindo a classe a ser serializada
[Serializable]
public class MyObject {
    public int n1 = 0;
    public int n2 = 0;
    public String str = null;
}
```

SERIALIZAÇÃO

```
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();
```

SERIALIZAÇÃO

É possível definir atributos que não são serializáveis.

```
// Definindo um atributo não serializável.
[NonSerialized()]
public string member5;
```

EXPRESSÕES
&
COMANDOS

OPERADORES ARITMÉTICOS

```
int i = 3
i = 6 + 1; //i = 7
i = i - 1; //i = 6
i = 14 * 2; //i = 28
i = i / 2; //i = 14
i = 24 % 5; //i = 4
```

OPERADORES ARITMÉTICOS

```
int i = 3
i = 6 + 1; //i = 7
i = i - 1; //i = 6
i = 14 * 2; //i = 28
i = i / 2; //i = 14
i = 24 % 5; //i = 4
```

OPERADORES DE ATRIBUIÇÃO

```
int j = 3, k;
j += 1; //j = 4
j -= 1; //j = 3
j  *= 2; //j = 6
j /= 2; //j = 3
j %= 5; //j = 3
k = j = 32; //k e j são 32
```

OPERADORES BINÁRIOS

```
int k = 12;
k = ~ k; //k = -13
k = 8 & 2; //k = 0
k = 8 | 2; //k = 10
k = 10 ^ 2; //k = 8
k = 24 << 2; //k = 96
k = 12 >> 2; //k = 3
```

OPERADORES BINÁRIOS

```
int k = 12;
k = ~ k; //k = -13
k = 8 & 2; //k = 0
k = 8 | 2; //k = 10
k = 10 ^ 2; //k = 8
k = 24 << 2; //k = 96
k = 12 >> 2; //k = 3
```

OPERADORES DE IGUALDADE

```
bool b;
b = 12 == 16; //b = false
b = 12 != 16; //b = true
```

OPERADOR CONDICIONAL

```
int a = 4, b = 3;
int c = a < b ? a : b; //c = b
```

OPERADORES BOOLEANOS

```
bool a = true, b = false, c;
c = a && b; //c = false
c = a || b; //c = true
c = a ^ b; //c = true
c = ! a; //c = false
```

OPERADORES BOOLEANOS

```
int i = 3
i = 6 + 1; //i = 7
i = i - 1; //i = 6
i = 14 * 2; //i = 28
i = i / 2; //i = 14
i = 24 % 5; //i = 4
```

OPERADORES RELACIONAIS

```
bool d;
d = 10.0f < 12.24f; //d = true
d = 10.0f > 12.24f; //d = false
d = 24 <= 23; //d = false
d = 24 >= 23; //d = true
```

OPERADORES CONDICIONAIS NULOS

Testam a presença de valores nulos antes de acessar um membro ou índice.

```
Cpf cpf  = pessoa?.Cpf;
Pessoa p = listaPessoas?[24];
bool? val = cpf.Validar();
```

OPERADORES CONDICIONAIS NULOS

Testam a presença de valores nulos antes de acessar um membro ou índice.

```
Cpf cpf  = pessoa?.Cpf;
Pessoa p = listaPessoas?[24];
bool? val = cpf.Validar();
```

Tipos primitivos e structs não podem ter o valor null, exceto caso o tipo da variável tenha uma '?' logo após a declaração de tipo.

```
int x = null; // Exceção
int? x = null; // OK
```

OPERADOR DE COALESCÊNCIA NULA

Representado por “??”. Utiliza dois operandos, um a sua esquerda e outro a direita. Retorna o operando da esquerda caso não seja nulo, do contrário avalia operando da direita e retorna o resultado.

```
Gato g = gatos[58] ?? new Gato();
```

OPERADOR DE COALESCÊNCIA NULA

Representado por “??”. Utiliza dois operandos, um a sua esquerda e outro a direita. Retorna o operando da esquerda caso não seja nulo, do contrário avalia operando da direita e retorna o resultado.

```
Gato g = gatos[58] ?? new Gato();
```

OPERADORES DE INCREMENTO E DECREMENTO

```
int a = 3;


a++; //4
a--; //3
++a; //4
--a; //3
```

OPERADORES DE CONVERSÃO DE TIPO

```
Casa h = new Casa();

Conversão de tipo:
Sala c = (Sala) h; // Exceção!!

Conversão segura de tipo:
Sala c  = h as Sala; // b = null;
```

OPERADORES DE CONVERSÃO DE TIPO

```
Casa h = new Casa();

Conversão de tipo:
Sala c = (Sala) h; // Exceção!!

Conversão segura de tipo:
Sala c  = h as Sala; // b = null;
```

OPERADOR DE TESTE DE TIPO

Retorna true caso o operando da esquerda possa ser convertido para o tipo descrito pelo operando na direita e não seja null.

```
bool d =  h is Sala: //d = false
bool e = h is Casa; //e = true
```

CHAMADAS DE FUNÇÃO

A chamada de funções acontece assim como em linguagens como C, C++, Java.

```
Console.WriteLine("*__*");
Math.Sqrt(42);
```

CHAMADAS DE FUNÇÃO

A chamada de funções acontece assim como em linguagens como C, C++, Java.

```
Console.WriteLine("*__*");
Math.Sqrt(42);
```

FUNÇÕES LAMBDA

São funções anônimas que geralmente são usadas para executar tarefas simples.

```
(int a, int b) => a + b;

candidatos.Where(c => c.Eleito);
```

LINQ

LINQ (Language Integrated Query) é uma biblioteca desenvolvida visando tornar a tarefa de aprendizado de manipulação de banco de dados com C# mais fácil.

A sintaxe é baseada em SQL.

```
IEnumerable<Candidato> listaC;

listaC = from c in candidatos
            where c.Eleito
                orderby c.Nome
                    select c;

listaC.Where(c => c.Votos > 2000);
```

DELEGATE

- É tipo de dado parecido com os ponteiros em C, porém fortemente tipado e orientado a objetos.
- São definidos com o uso da palavra-chave delegate.

```
public delegate void Teste(float a);

public class DelTeste{
    public static void m1(int a){}
    public static void m2(float a){}

    public DelTeste(){
        //Teste del = m1;
        Teste del = m2;
    }
}
```

EXPRESSÕES DE REFERENCIAMENTO

- [] Acesso de elemento de vetor.
- * Acesso de variável apontada por ponteiro.
- . Acesso de membro de uma estrutura ou objeto.
- -> Acesso de membro de uma estrutura apontada por um ponteiro.
- & Retorno de referência do operando.

EXPRESSÕES DE REFERENCIAMENTO

- [] Acesso de elemento de vetor.
- * Acesso de variável apontada por ponteiro.
- . Acesso de membro de uma estrutura ou objeto.
- -> Acesso de membro de uma estrutura apontada por um ponteiro.
- & Retorno de referência do operando.

EXPRESSÕES DE AGREGAÇÃO

```
Pessoa p = new Pessoa()
                {Nome = "Evandro"} ;
```

A expressão acima é equivalente a:

```
Pessoa p = new Pessoa();
p.Nome = "Evandro";
```

EXPRESSÕES CONDICIONAIS

- As expressões condicionais disponíveis são if, else if, else e switch.

```
if (p.Cpf.Validar()) {
    /* Código*/
}
else if (p.Cpf.Validar()){
    /*Hello!!*/
}
else {
    /*Goodbye*/
}
```

```
switch( k ){
  case 1:
      // Pass
  case int i when i%2 == 0:
      // Pass
  default:
      // Pass
}
```

EXPRESSÕES ITERATIVAS

```
var candidatos = new Candidato[5];
```

```
while (candidatos.Count > 0){
    /* Nunca entra*/
}
```

```
do{
    Console.WriteLine("*__*");
}while(candidatos.Count > 0);
```

EXPRESSÕES ITERATIVAS

```
var candidatos = new Candidato[5];
```

```
for (int i = 0; i < 20; i++){
    Console.WriteLine("*__*");
}
```

```
foreach (var c in candidatos){
    Console.WriteLine(c.Nome);
}
```

DESVIOS INCONDICIONAIS - ESCAPE

Assim como em outras linguagens o break encerra um loop ou switch.

```
while (true){
    foreach (var c in candidatos){
        if (c.Eleito) break;
    }
    //Vem pra cá
}
```

DESVIOS INCONDICIONAIS - ESCAPE

Assim como em outras linguagens o continue encerra um passo da iteração.

```
foreach (Candidato c in candidatos){
    if (c.Eleito) continue;
}
```

DESVIOS INCONDICIONAIS - ESCAPE

```
public int RetornaDois()
{
    return 2;
}
```

```
public int Erro()
{
    throw new Exception("ERRO!!");
    return 2; //Código inalcançável
}
```

DESVIOS INCONDICIONAIS - IRRESTRITO

```
switch(caso){
    case 1:
        //Código
    case 2:
        goto case 1;
}
```

```
Print:
Console.WriteLine("Oi");
goto Print;
```

DESVIOS INCONDICIONAIS

```
public IEnumerable<int> AlgunsInteiros(){
    for (int i = 0; i < 5; i++){
        yield return i;
        //yield break;
    }
}
```

MODULARIZAÇÃO

SUBPROGRAMAS

- Abstração inferior (“pedaço”) de um programa com escopo local que pode ou não receber parâmetros, que por sua vez podem ou não ser modificados dentro da função
- Pode retornar valores de diversos tipos, inclusive não-primitivos
- Aumenta a legibilidade, facilita a depuração, manutenção, testes e o reuso de código

SUBPROGRAMAS - PARÂMETROS

- Em C#, similar ao Java e C++, admitindo quaisquer tipos de dados.
- Para constantes, palavras chave *const* e *readonly* além de outros modificadores como *out*, *ref* e *params*.
- **Bidirecional de Entrada e Saída.**
- Admite ponteiros explicitamente com o uso do modificador ***unsafe***, seja em blocos ou funções, para apontar para tipos primitivos (inclusive ponteiros) e structs definidas pelo programador que contenham apenas campos “unmanaged-typed” (tipos primitivos), mas nunca objetos.

SUBPROGRAMAS - PARÂMETROS

```
unsafe static void FastCopy ( byte* ps, byte* pd,  int count )
{
    //unsafe context: can use pointers here
}

public void Transfere (double valor, Conta destino) {
    // implementação da transferência
}
```

SUBPROGRAMAS - OUT

- Com o uso do modificador *out* no(s) parâmetros, apenas a referência para a variável é passada, de modo que é necessário inicializar a mesma na função e o valor atribuído é passado de volta à função chamadora

```
public void m1(out int val) {
    val = 100;
    Console.WriteLine(val);//100
}
```

```
static void Main() {
    int valor = 5;
    m1(out valor);
    Console.WriteLine(valor);//100
    Console.Read();
}
```

SUBPROGRAMAS - REF

- Com o uso do modificador *ref* no(s) parâmetros, é passada a referência para a variável, de modo que se possa acessar o valor, sendo um parâmetro de entrada e saída, similar a um ponteiro.

```
public void m2(ref int val) {
    val += 100; //105
}
```

```
public static void Main() {
    int valor = 5;
    m2(ref valor);
    //valor = 105
}
```

SUBPROGRAMAS - PARAMS

- Com o uso do modificador *params* no parâmetros, é passada uma lista variável de parâmetros por meio de um vetor unidimensional

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

```
static void Main() {
    int v1,v2;
    v1 = Soma(10, 20)); //30
    v2 = Soma(10, 20, 30); //60
}
```

NOVOS TIPOS DE DADOS

- **Tipos de dados anônimos**: struct para designar uma única variável.
- **Tipos de dados simples**: structs genéricas.
- **Tipos abstratos de dados**: structs ou classes. Está relacionada não só ao conjunto de valores do TAD, mas a todas as operações e procedimentos a ele associados.

```
var produto = new {
    id = 1,
    nome = "Monitor LED 32",
    preco = 1500
};
```

```
public struct Book{
   public decimal price;
   public string title;
   public string author;
}
```

PACOTES

- Limitações encontradas no reuso de código: conflitos de nomes de entidades devido às várias fontes utilizadas
- Solução: **PACOTES**
- Bibliotecas, Frameworks, Coleções, aplicações utilitárias, aplicações completas, etc.
- Em C# é utilizada a palavra-chave *namespace*

PACOTES

- Convenção de nomes (framework .NET)

```
NomeDaEmpresa.NomeDoProjeto.ModuloDoSistema
```

- Hierarquia de namespaces
- Aplicação mais legível e fim dos conflitos de nomes, desde que as classes conflitantes estejam em namespaces diferentes

PACOTES

- Para diferenciar as classes, é preciso referenciá-las pelo seu namespace:

```
var guilherme = new Caelum.Banco.Usuarios.Gerente();
```

- Palavra chave using para definir o namespace a ser utilizado a seguir
- Pode declarar mais de um, desde que não haja conflito de nomes
- Quando há conflito, é preciso referenciar uma (ou mais) das classes conflitantes pelo seu namespace completo
- Apelido (alias) para um namespace:

```
using Seguranca = Caelum.Banco.Seguranca;
```

POLIMORFISMO

INFORMAÇÕES GERAIS

- Fortemente tipada.
- Tipagem estática.
- Inferência de tipo com o uso da palavra-chave var.
- Uso da palavra-chave virtual para sinalizar amarração tardia.
- Uso da palavra-chave base para acessar métodos ou atributos herdados (para atributos é opcional).

INFORMAÇÕES GERAIS

- Fortemente tipada.
- Tipagem estática.
- Inferência de tipo com o uso da palavra-chave var.
- Uso da palavra-chave virtual para sinalizar amarração tardia.
- Uso da palavra-chave base para acessar métodos ou atributos herdados (para atributos é opcional).

- Apresenta os seguintes tipos de polimorfismo:
  - Ad-hoc:
    - Coerção
    - Sobrecarga
  - Universal:
    - Parámetrico
    - Inclusão

COERÇÃO

- Ampliação

```
int val = 24;
long testeLong = valor;

float fl = 24.0f;
double novoTesteDouble = fl;
```

COERÇÃO

- Ampliação

```
int val = 24;
long testeLong = valor;

float fl = 24.0f;
double novoTesteDouble = fl;
```

- Estreitamento

```
double db = 23.9;
float testeFloat = db;

float testeFloat = (float) db;
```

SOBRECARGA

```
int a = 13, b = 12, c;
c = a + b;

string d = "Olá";
d = d + " Mundo!";
```

SOBRECARGA

```
int a = 13, b = 12, c;
c = a + b;

string d = "Olá";
d = d + " Mundo!";
```

```
int somar (int a, int b){
    return a + b;
}

float somar (float a, float b){
    return a + b;
}
```

SOBRECARGA

```
public struct Medida{
    public int x, y;
    public static Medida operator + (Medida a, Medida b){
        Medida c;
        c.x = a.x + b.x;
        c.y = a.y + b.y;
        return c;
    }
}
```

PARAMÉTRICO

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

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

INCLUSÃO

- C# possui os conceitos de:
  - Herança de classes.
  - Classes Abstratas
  - Interfaces

  OBS: Herança múltipla não é permitida.

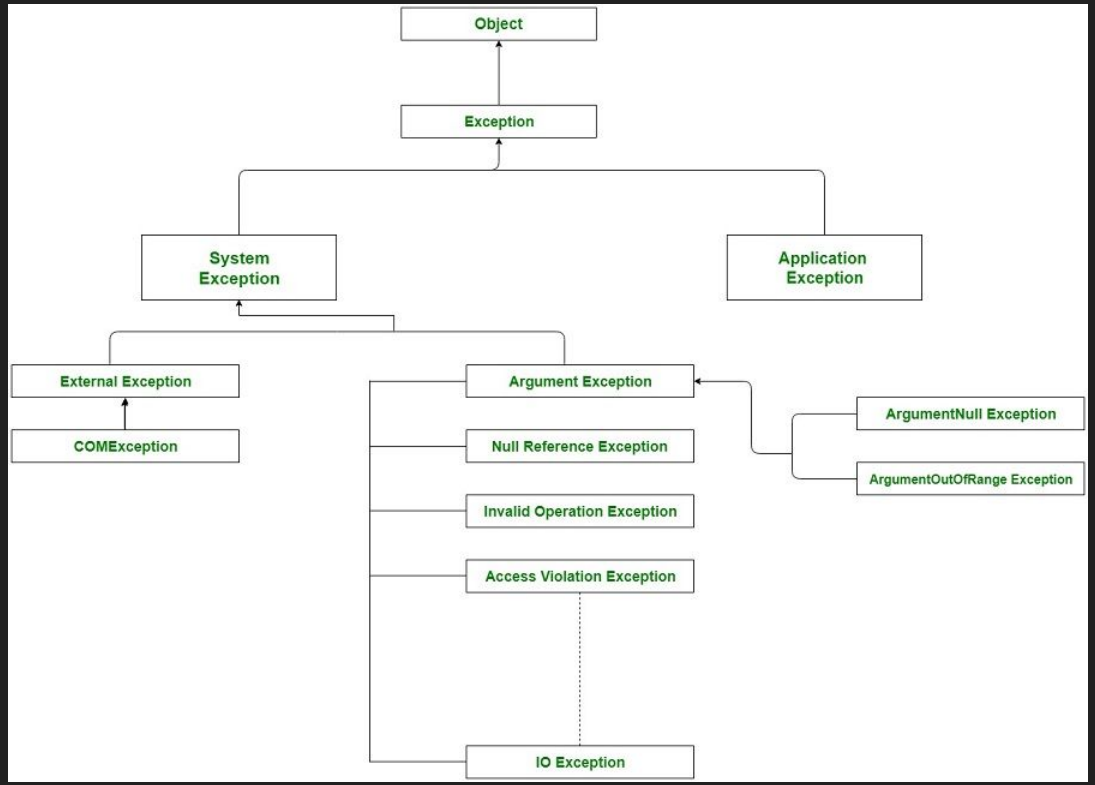
INCLUSÃO

```
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;
```

EXCEÇÕES

EXCEÇÕES


Object
Exception
System Exception
Application Exception
External Exception
COMException
Argument Exception
ArgumentNull Exception
ArgumentOutOfRange Exception
Null Reference Exception
Invalid Operation Exception
Access Violation Exception
IO Exception

EXCEÇÕES

- Exceções são tipos que derivam de System.Exception
- Uma “mistura” do tratamento de exceções de JAVA e C++

EXCEÇÕES - DEFININDO

```
using System;
using System.Runtime.Serialization;
[Serializable()]
public class TestException : Exception {
    public TestException() : base() { }
    public TestException(string message) : base(message) { }
    protected TestException(SerializationInfo info,
                            StreamingContext context) :
                                base(info, context) { }

}
```

EXCEÇÕES - WHEN

É possível utilizar a palavra-chave contextual *when* para especificar uma condição de filtro.

```
try{
    throw new TestException("Deu ruim!");
}
catch (TestException e) when (e.Message.Equals("Deu ruim!")){}
catch (Exception e) when (true){}
```

CONCORRÊNCIA

CONCORRÊNCIA

Em C #, a classe System.Threading.Thread é usada para trabalhar com threads. Permite criar e acessar threads individuais em um aplicativo multithread.

```
public void Teste(){}

public void Exec(){
    //Delegate utilizado pela Thread
    ThreadStart init = new ThreadStart(Teste);

    var th = new Thread(init)
}
```

CONCORRÊNCIA - ESTADOS

Os estados podem ser obtidos usando a enumeração ThreadState, que é uma propriedade da classe Thread.

CONCORRÊNCIA - ESTADOS

Os estados podem ser obtidos usando a enumeração ThreadState, que é uma propriedade da classe Thread.

- Aborted;
- AbortRequested;
- Background;
- Running;
- Stopped;
- Suspended;
- Unstarted;
- WaitSleepJoin;

CONCORRÊNCIA - SINCRONIZAÇÃO

Ações que podem ser tomadas por threads que acessam objetos sincronizados:

Enter/ TryEnter	Adquire o *lock* para o objeto. Geralmente marca o início da seção crítica.
Wait	Bloqueia a thread e libera o *lock* para outra thread acessar a seção crítica. Insere a thread em uma fila de threads bloqueadas.
Pulse/ PulseAll (signal)	Envia um sinal para as threads que estão na fila. A primeira thread a adquirir o *lock* irá entrar na seção críitica.
Exit	Libera o *lock* do objeto, e marca o fim da seção crítica.

CONCORRÊNCIA - SINCRONIZAÇÃO

```
// Definir o objeto sincronizado.
var obj = new Object();
// Definir a seção crítica.
Monitor.Enter(obj);
try {// Codigo da seçao critica.}
catch (Exception e){}
finally { Monitor.Exit(obj);}
```

- Podemos usar monitores para sincronizar objetos
- O uso de monitores é facilitado pelo bloco *try/catch/finally*.

CONCORRÊNCIA - SINCRONIZAÇÃO

O bloco try/catch/finally pode ser substituído pelo *lock* statement.

```
// Definir o objeto sincronizado.
var obj = new Object();

lock(obj);
```

CONCORRÊNCIA - SINCRONIZAÇÃO

Você pode definir métodos sincronizados também.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
void MetodoSincronizado()
{
  // Codigo
}
```

AVALIAÇÃO DE LINGUAGENS

MOTIVAÇÃO

- Resolver novos problemas e aprender novas LPs
- Analisar a viabilidade de um dado projeto em termos de LP
- Estimar o tempo e o custo de implementação
- Escolher a linguagem mais adequada para um projeto
- Necessidade de definir critérios de comparação, sejam gerais ou específicos, objetivos ou subjetivos
- Opiniões podem divergir!

CRITÉRIOS GERAIS - APLICABILIDADE

- *Linguagem oferece mecanismos para aplicações em geral ou aplicação em específico?*
- É uma linguagem de aplicação ampla, mas não é a mais adequada para lidar com recursos de mais “baixo nível” e controle de hardware, como seria o caso de C, por exemplo (mais permissiva);

CRITÉRIOS GERAIS - CONFIABILIDADE

- *Linguagem maximiza a segurança no uso dos recursos e detecta erros automaticamente (vazamento de memória e acessos indevidos, conflitos de tipos, etc.), além de preveni-los?*
- Sim. Herda modelos de Java, como no caso do coletor de lixo (GC), gera exceções e identifica erros detalhados por meio do compilador, tem grande controle da gerência de memória, grande variedade de classes de exceção, etc.

CRITÉRIOS GERAIS - FACILIDADE DE APRENDIZADO

- *Oferece quantidade regular de conceitos e pode ser facilmente aprendida por um programador?*
- Parcial
- Alguns conceitos e recursos são complexos.

CRITÉRIOS GERAIS - FACILIDADE DE APRENDIZADO

- Comparada com outras linguagens de paradigma e propósito similar, é mais facilmente aprendida por motivos como:
  - Grande número de inovações incorporadas à linguagem e que podem ser usadas de forma simples, na maioria dos casos
  - IDE amigável especializada na plataforma .NET com inúmeros recursos embarcados
  - Gerência de vários recursos feita pela própria LP.

CRITÉRIOS GERAIS - EFICIÊNCIA

- *Demanda de recursos de memória e processamento*
- Recurso na compilação e interpretação do código: código não é interpretado na compilação, mas em tempo de carga (JIT)
- Compilação tão rápida quanto as linguagens que não utilizam VM
- Comparação numérica entre LPs: **“The Computer Language Benchmarks Game”**
- Algoritmos de métodos matemáticos
- Conjunto de Mandelbrot, Pi, Árvore Binária, etc.

CRITÉRIOS GERAIS - EFICIÊNCIA

- Na comparação com Java, dos 10 algoritos testados, C# foi mais rápido em 8
- Em termos de memória, C# usou menos memória também em 8 algoritmos

Maior vantagem para C#:

regex-redux

source	secs	mem	gz	busy	cpu load
C# .NET Core	**2.17**	267,768	1869	4.26	36% 30% 84% 46%
Java	10.31	644,560	740	31.47	72% 72% 92% 70%

Maior vantagem para Java:

fannkuch-redux

source	secs	mem	gz	busy	cpu load
C# .NET Core	16.50	31,284	1189	65.39	99% 99% 99% 100%
Java	14.33	34,888	1282	56.56	99% 98% 99% 98%

CRITÉRIOS GERAIS - EFICIÊNCIA

- Na comparação com C++, dos 10 algoritmos testados, C# foi mais lento em todos
- Em termos de memória, C# usou mais memória em todos

Maior vantagem para C++:

mandelbrot

source	secs	mem	gz	busy	cpu load
C# .NET Core	5.60	65,296	816	21.78	96% 97% 99% 96%
C++ g++	1.51	25,828	1791	6.03	100% 100% 100% 100%

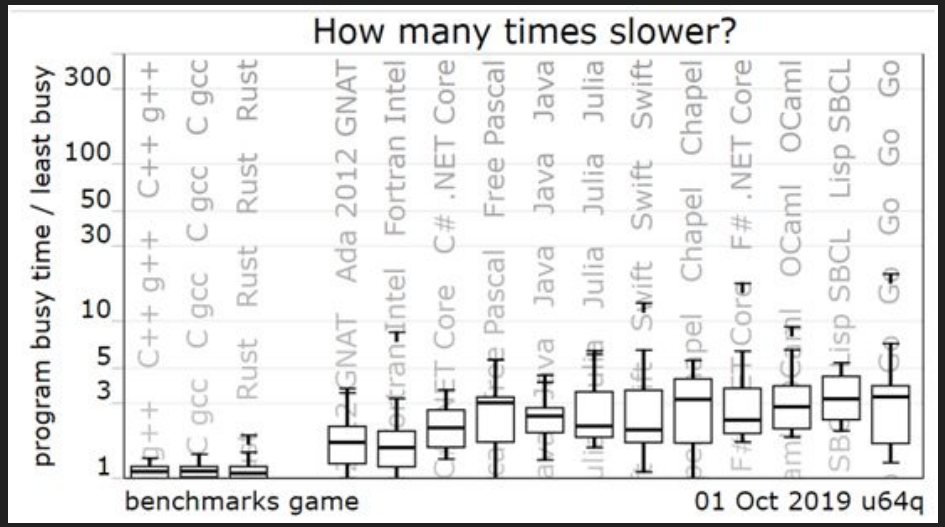
Menor vantagem para C++:

pidigits

source	secs	mem	gz	busy	cpu load
C# .NET Core	2.11	36,632	973	2.21	99% 2% 1% 3%
C++ g++	1.89	4,552	513	1.92	1% 100% 1% 0%

CRITÉRIOS GERAIS - EFICIÊNCIA

- Na comparação geral entre linguagens, temos o seguinte quadro:



Obs.: Melhor resultado da linguagem, comparado com o melhor resultado das outras linguagens, numa média. Quantas vezes mais lento (quanto menor, melhor)

CRITÉRIOS GERAIS - PORTABILIDADE

- *Facilidade de migrar os códigos fonte dos programas entre plataformas*
- C# possui ótima portabilidade
- Máquina virtual (similar à de Java) e código intermediário
- IDE Visual Studio para distribuições Linux
- Possibilidade de instalar apenas o ambiente de execução (máquina virtual e bibliotecas .NET)
- Também é possível executar códigos C# em sistemas não Windows usando implementações livres do Common Language Infrastructure como o Mono.

CRITÉRIOS GERAIS - MÉTODO DE PROJETO

- *A LP suporta o método de projeto a ser usado na aplicação?*
- C# tem o paradigma Orientado a Objeto (OO). Isso é uma escolha de projeto subjetiva, a depender do tipo de projeto. Entretanto, as linguagens OO têm ganhado cada vez mais força com o passar do tempo, sobretudo no mercado de TI

CRITÉRIOS GERAIS - EVOLUTIBILIDADE

- *A LP oferece estímulo para a criação de programas legíveis e facilmente atualizáveis?*
- Sim. Por ser uma linguagem OO, essa facilidade já se mostra presente na linguagem (conceito de classe, encapsulamento e abstração). Além disso, a própria sintaxe e os recursos da linguagem favorecem a legibilidade dos programas.

CRITÉRIOS GERAIS - REUSABILIDADE

- *Meios e facilidades para reutilização de código*
- Uma das mais completas em termos de recursos e reusabilidade
- Muitos recursos embarcados: bibliotecas, interfaces, conjuntos, listas, coleções, etc.
- Recursos da própria linguagem: diretiva using (transferência de função para a LP), separação em namespace, polimorfismo (redigibilidade e reusabilidade)
- Recursos embarcados na IDE (Ex.: Windows Form)

CRITÉRIOS GERAIS - INTEGRAÇÃO COM SOFTWARE

- Há ferramentas que fazem a integração de linguagens e tecnologias, como o IronPython, que permite rodar código Python no ambiente do C# e vice-versa
- C# suporta código legado, de linguagens não-gerenciadas (COM e DLL)
- Possui interoperabilidade com C++, componentes do ActiveX e API do Microsoft Windows
- Possui ferramentas para conexão com banco de dados

CRITÉRIOS GERAIS - CUSTO

- Custo em termos de implementação do projeto como um todo
- Popularidade da linguagem:
  - Bastante popular (grande variedade de aplicações e muito conteúdo disponível)
  - Um dos “carros-chefe” da Microsoft para a plataforma
  - Figura no top 10 das mais utilizadas (a frente de C e C++ e atrás de Java)
  - Relativamente fácil encontrar programadores e gerentes de projeto e empresas para implementação na linguagem
- Ampla variedade de recursos computacionais disponíveis (pagos e de graça)
- Custos relacionados à LP não são altos

Critérios Gerais	C	C++	Java	C#
Aplicabilidade	SIM	SIM	PARCIAL	PARCIAL
Confiabilidade	NÃO	NÃO	SIM	SIM
Aprendizado	NÃO	NÃO	NÃO	PARCIAL
Eficiência	SIM	SIM	PARCIAL	PARCIAL
Portabilidade	NÃO	NÃO	SIM	SIM
Método de Projeto	ESTRUTURADO	ESTRUTURADO E OO	OO	OO
Evolutibilidade	NÃO	PACIAL	SIM	SIM
Reusabilidade	PARCIAL	SIM	SIM	SIM
Integração	SIM	SIM	PARCIAL	SIM
Custo	DEPENDE	DEPENDE	DEPENDE	DEPENDE

Critérios Específicos	C	C++	Java	C#
Escopo	**SIM**	**SIM**	**SIM**	**SIM**
Expressões e comandos	SIM			SIM
Tipos primitivos e compostos	SIM	SIM	SIM	SIM
Gerenciamento de memória	PROGRAMADOR	PROGRAMADOR	SISTEMA	PROGRAMADOR/ SISTEMA
Persistência dos dados	Bibliotecas de classes e funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Bibliotecas de classes e funções e apoio da IDE
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por referência

C, C++, Java e C# requerem a definição explícita de entidades, associando-as a um escopo de visibilidade. Há pequenas diferenças.

Critérios Específicos	C	C++	Java	C#
Escopo	SIM	SIM	SIM	SIM
Expressões e comandos	**SIM**	**SIM**	**SIM**	**SIM**
Tipos primitivos e compostos	SIM	Todas oferecem uma ampla variedade de expressões e comandos.		SIM
Gerenciamento de memória	PROGRAMADOR	PROGRAMADOR	SISTEMA	PROGRAMADOR/ SISTEMA
Persistência dos dados	Bibliotecas de classes e funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Bibliotecas de classes e funções e apoio da IDE
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por referência

Critérios Específicos	C	C++	Java	C#
Escopo	SIM	SIM	SIM	SIM
Expressões e comandos	SIM	SIM	SIM	SIM
Tipos primitivos e compostos	**SIM**	**SIM**	**SIM**	**SIM**
Gerenciamento de memória	PROGRAMADO			ROGRAMADOR/ SISTEMA
Persistência dos dados	Bibliotecas de classes e funçõe			Bibliotecas de asses e funções e apoio da IDE
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por referência

Todas oferecem ampla variedade de tipos primitivos (mas C não oferece booleano) e compostos (mas nenhuma oferece conjunto potência). Recentemente, Java 8 incluiu o tipo função na linguagem (vide closures). C# possui tipo decimal.

Critérios Específicos	C	C++	Java	C#
Escopo	SIM	SIM	SIM	SIM
Expressões e comandos	SIM	SIM	SIM	SIM
Tipos primitivos e compostos	SIM	SIM	SIM	SIM
Gerenciamento de memória	**PROGRAMADOR**	**PROGRAMADOR**	**SISTEMA**	**PROGRAMADOR/ SISTEMA**
Persistência dos dados	Bibliotecas de classes e funçõ			Bibliotecas de asses e funções e apoio da IDE
Passagem de parâmetros	Lista variável e valor	default, por valor e por referência	valor e por cópia de referência	Lista variável, default, por valor e por referência

C/C++ deixam a cargo do programador. Java e C# utilizam coletor de lixo e gerenciam alocação de memória. C# permite que o programador faça alocação e manipulação de ponteiros

Critérios Específicos	C	C++	Java	C#
Escopo	SIM	SIM	SIM	SIM
Expressões e comandos	SIM			SIM
Tipos primitivos e compostos	SIM			SIM
Gerenciamento de memória	PROGRAMADO			OGRAMADOR/ SISTEMA
Persistência dos dados	**Bibliotecas de classes e funções**	**Biblioteca de classes e funções**	**JDBC, biblioteca de classes, serialização**	**LINQ, bibliotecas de classes e funções, serialização**
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por referência

C/C++ oferecem funções de I/O, mas deixam persistência a cargo do programador. Não existe padrão para interface com BD. Java possui serialização e padronizou interface com BD no JDBC, além de ter operações de I/O. C# possui o LINQ para facilitar a comunicação com BD e possui serialização

Critérios Específicos	C	C++	Java	C#
Escopo	SIM	SIM	SIM	SIM
Expressões e comandos	SIM	SIM	SIM	SIM
Tipos primitivos e compostos	SIM	SIM	SIM	SIM
Gerenciamento de memória	PROGRAMADO			PROGRAMADOR/ SISTEMA
Persistência dos dados	Bibliotecas de classes e funçõe			liotecas de es e funções poio da IDE
Passagem de parâmetros	**Lista variável e por valor**	**Lista variável, default, por valor e por referência**	**Lista variável, por valor e por cópia de referência**	**Lista variável, default, por valor e por referência**

C usa apenas passagem por valor, obrigando o uso de ponteiros em diversas ocasiões. C++ e C# oferecem o maior leque de opções. Java incluiu varargs na versão 5. C# possui modificadores para alterar a forma de passagem

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	**PARCIAL**	**SIM**	**SIM**	**SIM**
Sistema de tipos	NÃO			PARCIAL
Verificação de tipos	ESTÁTICA			STÁTICA / DINÂMICA
Polimorfismo	COERÇÃO E SOBRECARGA	TODOS	TODOS	TODOS
Exceções	NÃO	PARCIAL	SIM	SIM
Concorrência	NÃO (BIBLIOTECA DE FUNÇÕES)	NÃO (BIBLIOTECA DE FUNÇÕES)	SIM	SIM

C oferece apenas encapsulamento de dados. Versões recentes permitem ocultamento com declaração (.h) & definição (.c). Java, C++ e C# oferecem mecanismo de classes e pacotes com modificadores de acesso

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	PARCIAL	SIM	SIM	SIM
Sistema de tipos	**NÃO**	**PARCIAL**	**SIM**	**PARCIAL**
Verificação de tipos	ESTÁTICA			STÁTICA / INÂMICA
Polimorfismo	COERÇÃO E SOBRECARG			TODOS
Exceções	NÃO			SIM
Concorrência	NÃO (BIBLIOTECA DE FUNÇÕES)	NÃO (BIBLIOTECA DE FUNÇÕES)	SIM	SIM

Em C, diversos mecanismos (ex.: uniões livres, coerções e aritmética de ponteiros) permitem violação do sistema de tipos. C++ herda isso, mas possui sistema de tipos mais rigoroso se usado em sua forma OO. Java e C# possui um sistema de tipos bastante rigoroso apesar de este última permitir ponteiros num ambiente não-gerenciado

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	PARCIAL	SIM	SIM	SIM
Sistema de tipos	NÃO	PARCIAL	SIM	PARCIAL
Verificação de tipos	**ESTÁTICA**	**ESTÁTICA / DINÂMICA**	**ESTÁTICA / DINÂMICA**	**ESTÁTICA / DINÂMICA**
Polimorfismo	COERÇÃO E SOBRECARG			TODOS
Exceções	NÃO	PARCIAL	SIM	SIM
Concorrência	NÃO (BIBLIOTECA DE FUNÇÕES)	NÃO (BIBLIOTECA DE FUNÇÕES)	SIM	SIM

Todas as verificações de C são estáticas. C++, Java e C# fazem algumas verificações dinâmicas (ex.: amarração tardia, verificação de índice de vetor).

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	PARCIAL	SIM	SIM	SIM
Sistema de tipos	NÃO	PARCIAL	SIM	PARCIAL
Verificação de tipos	ESTÁTICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA
Polimorfismo	**COERÇÃO E SOBRECARGA**	**TODOS**	**TODOS**	**TODOS**
Exceções	NÃO			SIM
Concorrência	NÃO (BIBLIOTE DE FUNÇÕES)	DE FUNÇÕES)		SIM

C não possui polimorfismo paramétrico ou de inclusão. C++, Java e C# possuem todos, porém Java não permite sobrescrita de operadores.

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	PARCIAL	SIM	SIM	SIM
Sistema de tipos	NÃO	PARCIAL	SIM	PARCIAL
Verificação de tipos	ESTÁTICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA
Polimorfismo	COERÇÃO E SOBRECARG			TODOS
Exceções	**NÃO**	**PARCIAL**	**SIM**	**SIM**
Concorrência	NÃO (BIBLIOTECA DE FUNÇÕES)	NÃO (BIBLIOTECA DE FUNÇÕES)	SIM	SIM

C não oferece. C++ oferece, mas não obriga seu uso. Java e C# oferecem um sistema bastante rigoroso de tratamento de exceções.

Critérios Específicos	C	C++	Java	C#
Encapsulamento e proteção	PARCIAL	SIM	SIM	SIM
Sistema de tipos	NÃO	PARCIAL	SIM	PARCIAL
Verificação de tipos	ESTÁTICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA	ESTÁTICA / DINÂMICA
Polimorfismo	COERÇÃO E SOBRECARG	TODOS	TODOS	TODOS
Exceções	NÃO			SIM
Concorrência	**NÃO (BIBLIOTECA DE FUNÇÕES)**	**NÃO (BIBLIOTECA DE FUNÇÕES)**	**SIM**	**SIM**

Java e C# oferecem recursos nativos para exclusão mútua (sincronização) e oferecem threads em sua API básica.