

INF-0990 - Programação em C#

Painel ► Meus cursos ► INF-0990 ► Atividade 4: 10/09/2022 – 13:30-17:30 ► Aula Prática 4

Aula Prática 4

1) BCL: Base Class Library

1.1) Trabalhando com Strings e Textos

C# disponibiliza alguns métodos para manipulação de chars e strings, como colocá-las em maiúsculo, minúsculo, verificar se é uma letra, entre outras funcionalidades.

```
Console.WriteLine(char.ToUpper('c'));           // C
Console.WriteLine(char.IsWhiteSpace('\t'));      // True
Console.WriteLine(char.IsLetter('x'));           // True
Console.WriteLine(char.GetUnicodeCategory('x')); // LowercaseLetter
```

Para criar uma sequência de chars que se repetem, podemos usar o construtor da classe string. Também podemos usar esse construtor para construir uma string a partir de um vetor de chars. Por exemplo:

```
Console.WriteLine(new string('*', 10));          // *****

char[] ca = "Hello".ToCharArray();
string s = new string(ca);                      // Reconstrução de string a partir de vetor
de chars
Console.WriteLine(s);                          // "Hello"
```

Uma string vazia tem tamanho zero. Pelo fato de strings serem tipos de referência, elas podem ser nulas.

```

string empty = "";
Console.WriteLine(empty == "");           // True
Console.WriteLine(empty == string.Empty); // True
Console.WriteLine(empty.Length == 0);      // True

string nullString = null;
Console.WriteLine(nullString == null);    // True
Console.WriteLine(nullString == "");      // False
Console.WriteLine(string.IsNullOrEmpty (nullString)); // True
Console.WriteLine(nullString.Length == 0); // NullReferenceException

```

Podemos acessar elementos de uma string com a mesma indexação de um vetor. Como strings implementam enumeradores, podemos iterar sobre elas.

```

string str = "abcde";
char letter = str[1];           // 'b'

foreach (char c in "123")
    Console.Write (c + ",");    // 1,2,3,

```

C# permite que façamos buscas dentro de strings. Os principais métodos para isso são: *Contains*, *StartsWith*, e *EndsWith*. Veja os exemplos:

```

Console.WriteLine("quick brown fox".Contains("brown")); // True
Console.WriteLine("quick brown fox".EndsWith("fox"));   // True

// IndexOf retorna a primeira posição de um determinado char
Console.WriteLine("abcde".IndexOf("cd"));               // 2
Console.WriteLine("abcde".IndexOf("xx"));               // -1

// IndexOfAny retorna a primeira ocorrência de um dos chars determinados:
Console.WriteLine("ab,cd ef".IndexOfAny(new char[] { ' ', ',', '.' })); // 2
Console.WriteLine("pas5w0rd".IndexOfAny("0123456789".ToCharArray())); // 3

```

LastIndexOf é como *IndexOf*, mas funciona de trás pra frente em uma string. *LastIndexOfAny* faz a mesma coisa em ordem reversa.

Strings são tipos imutáveis, portanto, todos os métodos abaixo retornam uma nova string, deixando a string original inalterada.

```
// Substring extraem uma porção da string
string left3 = "12345".Substring(0, 3);      // "123";
string mid3  = "12345".Substring(1, 3);      // "234";

// Se omitirmos o comprimento, obtemos todo o restante da string
string end3  = "12345".Substring(2);         // "345";

// Insert e Remove inserem ou removem chars em uma determinada posição
string s1 = "helloworld".Insert(5, ", ");    // "hello, world"
string s2 = s1.Remove(5, 2);                 // "helloworld";

// PadLeft e PadRight estendem uma string para um determinado tamanho e
// com um determinado char (ou espaço, se não for especificado)
Console.WriteLine("12345".PadLeft(9, '*'));   // ****12345
Console.WriteLine("12345".PadLeft(9));        //      12345

// TrimStart, TrimEnd e Trim removem determinados chars
// (usualmente espaços em branco) do início, fim, ou em ambos os lugares
Console.WriteLine(" abc \t\r\n ".Trim().Length); // 3

// Replace substitui ocorrências de um determinado char ou substring
Console.WriteLine("to be done".Replace(" ", " | ")); // to | be | done
Console.WriteLine("to be done".Replace(" ", ""));    // tobedone
```

O método *Split* retorna um vetor de palavras, caso o delimitador seja espaços em branco. Por exemplo:

```
string[] words = "The quick brown fox".Split();

foreach(string s in words) Console.WriteLine(s);

// O método estático Join reverte o Split
string together = string.Join(" ", words);
Console.WriteLine(together);                // The quick brown fox

// O método estático Concat faz a concatenação de múltiplas strings.
// É equivalente ao operador +
string sentence = string.Concat("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";

Console.WriteLine(sameSentence);            // The quick brown fox
```

Quando chamamos o método *string.Format*, podemos fornecer um modelo de formatação, no qual determinamos as posições de futuras variáveis. Por exemplo:

```

string composite = "It's {0} degrees in {1} on this {2} morning";           // Futuras
    variáveis são especificadas com {}
string s = string.Format(composite, 35, "Perth", DateTime.Now.DayOfWeek); // Substitu
imos as posições com os valores das variáveis
Console.WriteLine(s);                                                     // It's 35
    degrees in Perth on this Thursday morning

// O uso de modelos de formatação nos permite determinarmos
// o comprimento dos valores, auxiliando no alinhamento
composite = "Name={0,-20} Credit Limit={1,15:C}";

Console.WriteLine(string.Format(composite, "Mary", 500));                 // Name=Mar
y                                Credit Limit=          $500.00
Console.WriteLine(string.Format(composite, "Elizabeth", 20000));          // Name=Eli
zabeth                          Credit Limit=        $20,000.00

// O equivalente ao format seria usarmos vários comandos de Pad
s = "Name=" + "Mary".PadRight(20) + " Credit Limit=" + 500.ToString("C").PadLeft(15);
Console.WriteLine(s);

```

Comparações de strings podem ser feitas de forma ordinal, sensíveis a cultura, sensíveis ou insensíveis a caixa.

```

Console.WriteLine(string.Equals("foo", "FOO", StringComparison.OrdinalIgnoreCase));
// True
Console.WriteLine("ü" == "Û");
// False

// O método de comparação CompareTo retorna um número positivo, negativo ou zero,
// dependendo se o primeiro valor vem antes, depois ou junto com o segundo valor
Console.WriteLine("Boston".CompareTo("Austin"));           // 1
Console.WriteLine("Boston".CompareTo("Boston"));           // 0
Console.WriteLine("Boston".CompareTo("Chicago"));          // -1
Console.WriteLine("ü".CompareTo("Û"));                     // 0
Console.WriteLine("foo".CompareTo("FOO"));                 // -1

// O próximo comando executa uma comparação sensíveis a cultura
Console.WriteLine(string.Compare("foo", "FOO", true));     // 0

```

Ao contrário de strings, o tipo *StringBuilder* é mutável. Quando precisamos concatenar várias strings, é mais eficiente usarmos a abordagem com *StringBuilder*.

```

System.Text.StringBuilder sb = new System.Text.StringBuilder();

for(int i = 0; i < 50; i++) sb.Append(i + ",");

// Para obter o resultado final, chame o método ToString()
Console.WriteLine(sb.ToString());

sb.Remove(0, 60);           // Remove os primeiros 50 caracteres
sb.Length = 10;            // Trunca para 10 caracteres
sb.Replace(",", "+");      // Substitui virgula pelo +

Console.WriteLine(sb.ToString()); // 3+24+25+26

sb.Length = 0;             // Limpa StringBuilder

```

1.2) Trabalhando com Data e Hora

Temos basicamente três formas de trabalhar com intervalo de horas, com o uso do objeto *TimeSpan*. Podemos criá-lo via um dos seus construtores, podemos chamar seu método estático, ou subtrair um objeto *DateTime* de outro. Veja os exemplos abaixo:

```

Console.WriteLine(new TimeSpan(2, 30, 0));           // 02:30:00
Console.WriteLine(TimeSpan.FromHours(2.5));        // 02:30:00
Console.WriteLine(TimeSpan.FromHours(-2.5));        // -02:30:00
Console.WriteLine(DateTime.MaxValue - DateTime.MinValue);

// TimeSpan sobrecarrega os operadores + e -
Console.WriteLine(TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30));
Console.WriteLine(TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1));

```

Também podemos acessar propriedades como dias, horas, minutos, dentre outras.

```

TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);

// Tipo int
Console.WriteLine(nearlyTenDays.Days);           // 9
Console.WriteLine(nearlyTenDays.Hours);          // 23
Console.WriteLine(nearlyTenDays.Minutes);        // 59
Console.WriteLine(nearlyTenDays.Seconds);        // 59
Console.WriteLine(nearlyTenDays.Milliseconds);  // 0

// A propriedade Total retorna valores com o tipo double
Console.WriteLine();
Console.WriteLine(nearlyTenDays.TotalDays);      // 9.99998842592593
Console.WriteLine(nearlyTenDays.TotalHours);     // 239.999722222222
Console.WriteLine(nearlyTenDays.TotalMinutes);   // 14399.9833333333
Console.WriteLine(nearlyTenDays.TotalSeconds);   // 863999
Console.WriteLine(nearlyTenDays.TotalMilliseconds); // 863999000

```

Quando desejamos trabalhar com data e hora, declaramos um tipo *DateTime*. Podemos também determinar o fuso horário, além de especificar a data e hora desejada.

```

DateTime d1 = new DateTime(2010, 1, 30);
Console.WriteLine(d1); // 1/30/2010 12:00:00 AM

DateTime d2 = new DateTime(2010, 1, 30, 12, 0, 0);
Console.WriteLine(d2); // 1/30/2010 12:00:00 PM
Console.WriteLine(d2.Kind); // Não especificado

DateTime d3 = new DateTime(2010, 1, 30, 12, 0, 0, DateTimeKind.Utc);
Console.WriteLine(d3); // 1/30/2010 12:00:00 AM -02:00
Console.WriteLine(d3.Kind); // Utc

DateTimeOffset d4 = d1; // Conversão implícita
Console.WriteLine(d4); // 1/30/2010 12:00:00 AM -02:00

DateTimeOffset d5 = new DateTimeOffset(d1, TimeSpan.FromHours(-8)); // -8 horas UTC
Console.WriteLine(d5); // 1/30/2010 12:00:00 AM -08:00

```

Em C#, podemos obter a data/hora do instante de tempo atual, com e sem o deslocamento do fuso horário UTC. Veja os exemplos:

```

Console.WriteLine(DateTime.Now); // Data/hora do instante atual
Console.WriteLine(DateTimeOffset.Now); // Data/hora do instante atual com deslocamento do fuso horário

Console.WriteLine(DateTime.Today); // Dia de hoje

Console.WriteLine(DateTime.UtcNow); // Data/hora do fuso UTC
Console.WriteLine(DateTimeOffset.UtcNow); // Data/hora do fuso UTC com deslocamento

```

1.3) Formatação e Conversão

O mecanismo de formatação mais simples é usar o método *ToString*, que retorna uma representação literal de um objeto. E o método *Parse*, que transforma uma representação literal em algum objeto determinado.

```

string s = true.ToString();
Console.WriteLine(s);                                // True

// Parse faz o trabalho inverso
bool b = bool.Parse(s);
Console.WriteLine(b);                                // True

// TryParse evita o lançamento de exceções em caso de erro de formatação
int i;
Console.WriteLine(int.TryParse("qwerty", out i));    // False
Console.WriteLine(int.TryParse("123", out i));        // True

if (int.TryParse("123", out int j))
{
    Console.WriteLine(j);                            // 123
}

bool validInt = int.TryParse("123", out int _);
Console.WriteLine(validInt);                          // True

```

Podemos também usar um provedor de formatação, como o *NumberFormatInfo*, que fornece instruções de como a formatação deve ser realizada.

```

System.Globalization.NumberFormatInfo f = new System.Globalization.NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine(3.ToString("C", f));                // $$ 3.00

// O provedor padrão é o CultureInfo.CurrentCulture
Console.WriteLine(10.3.ToString("C", null));          // $10.30

// Por conveniência, a maioria dos tipos são
// sobrecarregados para evitar o fornecimento do provedor
Console.WriteLine(10.3.ToString("C"));                // $10.30
Console.WriteLine(10.3.ToString("F4"));                // 10.3000

```

Para requisitar uma cultura em específico, podemos usar o método estático *GetCultureInfo* da classe *CultureInfo*.

```

using System.Globalization;

CultureInfo uk = CultureInfo.GetCultureInfo("en-GB");
Console.WriteLine(3.ToString("C", uk));           // £3.00

string composite = "Credit={0:C}";
Console.WriteLine(string.Format(composite, 500)); // Credit=$500.00

Console.WriteLine("Credit={0:C}", 500);           // Credit=$500.00

// E podemos definir uma cultura invariante
DateTime dt = new DateTime(2000, 1, 2);
System.Globalization.CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine(dt.ToString(iv));               // 01/02/2000 00:00:00
Console.WriteLine(dt.ToString("d", iv));          // 01/02/2000

object someObject = DateTime.Now;
string s = string.Format(CultureInfo.InvariantCulture, "{0}", someObject);
Console.WriteLine(s);

```

Para analisar (*parse*) uma formatação regional ou específica, podemos passar parâmetros de estilo para o método *Parse*. Por exemplo:

```

using System.Globalization;

try
{
    int error = int.Parse("(2)");                // Exceção lançada
}
catch(FormatException ex) { }

int minusTwo = int.Parse("(2)", NumberStyles.Integer | NumberStyles.AllowParentheses);
// OK
Console.WriteLine(minusTwo);
// -2

decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency, CultureInfo.GetC
ultureInfo ("en-GB"));
Console.WriteLine(fivePointTwo);
// 5.20

```

Outros exemplos:


```

using System.Globalization;

int thousand = int.Parse("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse("(2)", NumberStyles.Integer | NumberStyles.AllowParentheses);

Console.WriteLine(double.Parse("1,000,000", NumberStyles.Any));           // 1000000
Console.WriteLine(decimal.Parse("3e6", NumberStyles.Any));               // 3000000
Console.WriteLine(decimal.Parse("$5.20", NumberStyles.Currency));         // 5.20

NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
Console.WriteLine(double.Parse("€1 000 000", NumberStyles.Currency, ni)); // 1000000

```

Uma classe muito usada para conversão é a *Convert*. Essa classe fornece vários métodos estáticos como *ToInt32*, *ToBase64String*, dentre outros.

```

double d = 3.9;
int i = Convert.ToInt32(d);
Console.WriteLine(i); // 4

int thirty = Convert.ToInt32("1E", 16); // Parse de hexadecimal
uint five = Convert.ToUInt32("101", 2); // Parse de binary

Console.WriteLine(thirty); // 30
Console.WriteLine(five); // 5

// Conversões dinâmicas
Type targetType = typeof(int);
object source = "42";

object result = Convert.ChangeType(source, targetType);

Console.WriteLine(result); // 42
Console.WriteLine(result.GetType()); // System.Int32

// Conversões em Base-64
Console.WriteLine(Convert.ToBase64String(new byte[] { 123, 5, 33, 210 }));
Console.WriteLine(Convert.FromBase64String("ewUh0g==")); // ewUh0g==

```

Para trabalhar com conversões binárias, usamos a classe *BitConverter*.

```

foreach(byte b in BitConverter.GetBytes(3.5))
    Console.Write(b + " "); // 0 0 0 0 0 0 12 64

```

1.4) Trabalhando com Números

Tipos *BigInteger* suportam precisões numéricas arbitrárias, não tendo limites máximos ou mínimos. Cuidado para não ocupar toda a memória do seu PC.

[illegible]

Enquanto que tipos *Half* armazenam números de ponto flutuante que ocupam 16 bits. E podem representar valores entre -65500 e 65500.

```
Half h = (Half) 123.456;
Console.WriteLine(h);           // 123.44

Console.WriteLine(Half.MinValue); // -65500
Console.WriteLine(Half.MaxValue);  // 65500

Console.WriteLine((Half)65500);    // 65500
Console.WriteLine((Half)65490);    // 65500
Console.WriteLine((Half)65480);    // 65470
```

Podemos trabalhar com números complexos em C#, veja os exemplos abaixo:

```
using System.Numerics;

var c1 = new Complex(2, 3.5);
var c2 = new Complex(3, 0);

Console.WriteLine(c1);
Console.WriteLine(c2);

Console.WriteLine(c1.Real);      // 2
Console.WriteLine(c1.Imaginary); // 3.5
Console.WriteLine(c1.Phase);     // 1.05165021254837
Console.WriteLine(c1.Magnitude); // 4.03112887414927

Complex c3 = Complex.FromPolarCoordinates(1.3, 5);

// Os operadores aritméticos são sobrecarregados
// para usarmos com números complexos
Console.WriteLine(c1 + c2);      // (5, 3.5)
Console.WriteLine(c1 * c2);      // (6, 10.5)

Console.WriteLine(Complex.Atan(c1));
Console.WriteLine(Complex.Log10(c1));
Console.WriteLine(Complex.Conjugate(c1));
```

Gerar números aleatórios é uma tarefa comum em programação. Em C#, podemos fazer isso com a classe *Random*.

```

Random r1 = new Random(1); // Usamos semente = 1
Random r2 = new Random(1);
Console.WriteLine(r1.Next(100) + ", " + r1.Next(100)); // 24, 11
Console.WriteLine(r2.Next(100) + ", " + r2.Next(100)); // 24, 11

// Podemos usar o clock do computador como semente
Random r3 = new Random();
Random r4 = new Random();
Console.WriteLine(r3.Next(100) + ", " + r3.Next(100));
Console.WriteLine(r4.Next(100) + ", " + r4.Next(100));

// Podemos também usar Guid como semente
Random r5 = new Random(Guid.NewGuid().GetHashCode());
Random r6 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine(r5.Next (100) + ", " + r5.Next(100));
Console.WriteLine(r6.Next (100) + ", " + r6.Next(100));

// Random não é criptograficamente seguro,
// podemos resolver isso com o código abaixo:
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte[4];
rand.GetBytes(bytes);

Console.WriteLine(BitConverter.ToInt32(bytes, 0));

```

1.5) Guid e Utilidades

Identificadores únicos universais (Guid) podem ser usados por meio da classe *Guid*:

```

Guid g = Guid.NewGuid();
Console.WriteLine(g.ToString());

Guid g1 = new Guid("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine(g1 == g2); // True

byte[] bytes = g.ToByteArray();
Guid g3 = new Guid(bytes);
Console.WriteLine(g3);

```

Quando desejamos executar processos externos, usamos a classe utilitária *Process*, por exemplo:

```

using System.Diagnostics;

Process.Start("notepad.exe");

```

Assim como enviar e obter informações de processos.

```
using System.Diagnostics;

ProcessStartInfo psi = new ProcessStartInfo
{
    FileName = "cmd.exe",
    Arguments = "/c ipconfig /all",
    RedirectStandardOutput = true,
    UseShellExecute = false
};

Process p = Process.Start(psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine(result);
```

Para abrir um arquivo ou URL, podemos usar o exemplo abaixo:

```
using System.Diagnostics;

LaunchFileOrUrl("http://www.ic.unicamp.br");

void LaunchFileOrUrl(string url)
{
    Process.Start(new ProcessStartInfo(url) { UseShellExecute = true });
}
```

2) Coleções (Listas, Filas, Pilhas, Conjuntos, Dicionários)

C# fornece uma coleção de estruturas de dados como listas, pilhas, conjuntos e dicionários.

Para criar uma lista, criamos um objeto da classe *List*. Coleções suportam o uso de tipos de dados genéricos, portanto, podemos especificar se queremos que a lista armazene tipos inteiros, literais ou outros tipos. Por exemplo:

```

List<string> words = new List<string>();           // Cria uma lista do tipo string

words.Add("melon");                               // Adiciona um elemento no final d
a lista
words.Add("avocado");
words.AddRange(new[] { "banana", "plum" } );      // Adiciona uma sequencia de eleme
ntos no final da lista
words.Insert(0, "lemon");                         // Insere elemento no inicio da li
sta
words.InsertRange(0, new[] { "peach", "nashi" }); // Insere uma sequência de element
os no inicio da lista

words.Remove("melon");                           // Remove um elemento da lista
words.RemoveAt(3);                               // Remove o quarto elemento
words.RemoveRange(0, 2);                         // Remove os primeiros dois elemen
tos

words.RemoveAll(s => s.StartsWith("n"));          // Remove todas as strings começan
do com 'n':

Console.WriteLine(words[0]);                     // Primeira string
Console.WriteLine(words[words.Count - 1]);        // Última string
foreach(string s in words) Console.WriteLine(s);  // Todas as strings

List<string> subset = words.GetRange(1, 2);        // Cria uma sublista com o segundo
e terceiro elementos

string[] wordsArray = words.ToArray();            // Cria um vetor
do tipo string que recebe os elementos da lista

string[] existing = new string[1000];
words.CopyTo(0, existing, 998, 2);               // Copia os dois
primeiros elementos para o final de um vetor de strings

List<string> upperCaseWords = words.ConvertAll(s => s.ToUpper()); // Converte toda
s as strings para caixa alta
List<int> lengths = words.ConvertAll(s => s.Length); // Converte toda
s as strings para os seu comprimento

```

Outra estrutura de dados do tipo lista muito usada são as Listas Ligadas (*LinkedList*).

```

var tune = new LinkedList<string>();           // Cria uma lista ligada do tipo string
tune.AddFirst("do");                          // do
tune.AddLast("so");                          // do - so

tune.AddAfter(tune.First, "re");              // do - re - so
tune.AddAfter(tune.First.Next, "mi");         // do - re - mi - so
tune.AddBefore(tune.Last, "fa");              // do - re - mi - fa - so

tune.RemoveFirst();                          // re - mi - fa - so
tune.RemoveLast();                          // re - mi - fa

LinkedListNode<string> miNode = tune.Find("mi");
tune.Remove(miNode);                         // re - fa
tune.AddFirst(miNode);                       // mi - re - fa

```

Também podemos usar Filas (*Queue*) para armazenar nossos dados, por exemplo:

```

var q = new Queue<int>();                     // Cria uma fila do tipo int

q.Enqueue(10);                               // Adiciona a fila
q.Enqueue(20);

int[] data = q.ToArray();                    // Exporta para vetor

Console.WriteLine(q.Count);                  // 2
Console.WriteLine(q.Peek());                 // 10
Console.WriteLine(q.Dequeue());              // 10
Console.WriteLine(q.Dequeue());              // 20
Console.WriteLine(q.Dequeue());              // Lança exceção (fila vazia)

```

Pilhas (*Stack*) podem ser criadas de modo similar:

```

var s = new Stack<int>();                     // Cria uma pilha do tipo int

s.Push(1);                                   // 1
s.Push(2);                                   // 1,2
s.Push(3);                                   // 1,2,3

Console.WriteLine(s.Count);                  // 3
Console.WriteLine(s.Peek());                 // 3, Stack = 1,2,3
Console.WriteLine(s.Pop());                  // 3, Stack = 1,2
Console.WriteLine(s.Pop());                  // 2, Stack = 1
Console.WriteLine(s.Pop());                  // 1, Stack = vazia
Console.WriteLine(s.Pop());                  // Lança exceção

```

Conjuntos Hash (*HashSet*) são importantes para várias estruturas de dados comumente usadas, como dicionários. Esses conjuntos permitem armazenar apenas elementos únicos. Para criar um conjunto hash, podemos instanciar um objeto *HashSet*.

```
var letters = new HashSet<char>("the quick brown fox");

Console.WriteLine(letters.Contains('t'));    // true
Console.WriteLine(letters.Contains('j'));    // false

foreach(char c in letters) Console.Write(c); // the quickbrownfx
```

Conjuntos podem ser ordenados (*SortedSet*).

```
var letters = new SortedSet<char>("the quick brown fox");

foreach(char c in letters)
    Console.Write(c);                                // bcefhiknoqrtuwx

Console.WriteLine();

foreach (char c in letters.GetViewBetween('f', 'i')) // Iterador sobre uma sequência
    Console.Write(c);                                // fhi
```

Conjuntos disponibilizam vários operadores, como interseção, exceção, união, entre outros.

```
var letters = new HashSet("the quick brown fox");
letters.IntersectWith("aeiou");                    // Interseção
foreach(char c in letters) Console.Write(c);        // euio

Console.WriteLine();

letters = new HashSet("the quick brown fox");
letters.ExceptWith("aeiou");                        // Exceção
foreach(char c in letters) Console.Write(c);        // th qckbrwnfx

Console.WriteLine();

letters = new HashSet("the quick brown fox");
letters.SymmetricExceptWith("the lazy brown fox");// Interseção simétrica entre dois c
onjuntos
foreach(char c in letters) Console.Write(c);        // quicklazy
```

Dicionários (*Dictionary*) são conjuntos que permitem a atribuição de uma chave de identificação (*key*) a um valor. Ou seja, são estruturas de dados do tipo chave-valor.


```

var d = new Dictionary<string, int>();           // Cria dicionário com chave do tipo str
ing e valor do tipo int

d.Add("One", 1);
d["Two"] = 2;                                  // Adiciona elemento 2 ao dicionário com
a chave "Two"
d["Two"] = 22;                                 // Atualiza elemento, pois a chave "Two"
já está inserida
d["Three"] = 3;

Console.WriteLine(d["Two"]);                   // 22
Console.WriteLine(d.ContainsKey("One"));        // true (operação rápida)
Console.WriteLine(d.ContainsValue(3));          // true (operação lenta)

int val = 0;
if(!d.TryGetValue("one", out val))             // Tenta acessar o valor dado uma chave
    Console.WriteLine("No val");               // "No val"

// Dicionários podem ser enumerados de três formas diferentes
foreach(KeyValuePair<string, int> kv in d)
    Console.WriteLine(kv.Key + "; " + kv.Value); // One; 1 Two; 22 Three; 3

foreach(string s in d.Keys) Console.Write(s);   // OneTwoThree

foreach(int i in d.Values) Console.Write(i);    // 1223

```

Podemos ordenar uma lista a partir da implementação de um comparador, que irá determinar a forma como a comparação entre dois objetos deverá ser feita.

```

var wishList = new List<Wish>();
wishList.Add(new Wish("Peace", 2));
wishList.Add(new Wish("Wealth", 3));
wishList.Add(new Wish("Love", 2));
wishList.Add(new Wish("3 more wishes", 1));

wishList.Sort(new PriorityComparer());           // Ordena a lista, informando qual co
mparador será usado

foreach(Wish w in wishList) Console.WriteLine(w.Name);

class Wish
{
    public string Name;
    public int Priority;

    public Wish(string name, int priority)
    {
        Name = name;
        Priority = priority;
    }
}

class PriorityComparer : Comparer<Wish>           // Implementação de interface Compar
er
{
    public override int Compare(Wish x, Wish y)    // Sobrescrita do método de comparaç
ão, determinando como dois objetos são comparados
    {
        if(object.Equals(x, y)) return 0;
        return x.Priority.CompareTo(y.Priority);
    }
}

```

Vetores podem ser ordenados com expressão lambda. Analise o código abaixo. Qual será a ordem do vetor?

```

int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort(numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);

```

3) Utilizando Assemblies Externos

Para utilizar classes do BCL, basta referenciar seu *assembly* por meio da diretiva *using*, no início de nosso arquivo de código. Para referenciar *assemblies* externos, entretanto, é necessário, além disso, ter acesso a esses *assemblies* e referenciá-los em nosso projeto. Existem diferentes maneiras de fazer isso:

- Referenciando Pacotes Nuget
- Referenciando Projetos Externos
- Referenciando Assemblies em Arquivos

3.1) Referenciando Pacotes Nuget

Para referenciar pacotes Nuget, é necessário primeiro localizarmos a referência ao *assembly* no site do Nuget: <http://www.nuget.org>.

Por exemplo, como exercício, vamos utilizar o pacote Newtonsoft.Json, que é um pacote de classes bastante popular, para criar e manipular strings em JSON a partir de classes.]

Inicialmente, é necessário instalar o pacote em nosso projeto, utilizando o .NET CLI. Para isso, em nosso shell, nos posicionamos no diretório onde localiza-se o nosso projeto (o diretório onde está o arquivo .csproj) e executamos:

```
> dotnet add package Newtonsoft.Json --version 13.0.2-beta2
```

Se observarmos nosso arquivo de projetos, veremos que a seguinte diretiva foi inserida:

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.2-beta2" />
</ItemGroup>
```

Isso significa que, além de baixar e instalar o pacote Nuget, o CLI também inseriu uma referência para o *assembly* em nosso arquivo de projeto.

Uma vez com isso, já podemos usar as classes do *assembly* em nosso projeto. Para descobrir as classes disponíveis, precisamos procurar a documentação do pacote. No caso do Newtonsoft.Json, descobrimos, pelo Nuget que essa documentação fica em:

- <https://www.newtonsoft.com/json/help/html/Introduction.htm>

Podemos agora utilizar a classe *JsonConvert* para serializar e deserializar objetos em JSON. Para isso, basta inserir a diretiva *using Newtonsoft.Json;* no arquivo que utilizar classes do *assembly*.

Crie um novo arquivo *Serializer.cs* e adicione o seguinte código:

```

namespace proj;
using Newtonsoft.Json;
public class Serializer
{
    public Serializer() {
        Product product = new Product();
        product.Name = "Apple";
        product.ExpiryDate = new DateTime(2008, 12, 28);
        product.Price = 3.99M;
        product.Sizes = new string[] { "Small", "Medium", "Large" };
        string output = JsonConvert.SerializeObject(product, Formatting.Indented);
        Console.WriteLine(output);
        Product? deserializedProduct = JsonConvert.DeserializeObject<Product>(output);
        Console.WriteLine("Product: "+deserializedProduct);
    }
}

class Product {
    public string? Name;
    public DateTime ExpiryDate;
    public decimal Price;
    public string[]? Sizes;

    public override string ToString() {
        string outvar = "Name: "+Name+" Expiry Date: "+ExpiryDate+" Price: "+Price+" S
izes: ";
        foreach (string s in Sizes) {
            outvar += s + " ";
        }
        return(outvar);
    }
}

```

Agora, no código do programa principal, *Program.cs* copie o seguinte arquivo:

```

namespace proj;

using System;

class Program
{
    static void Main()
    {
        Serializer s = new Serializer();
    }
}

```

Compile o programa e execute. Você verá que usou com sucesso a biblioteca

3.2) Referenciando Projetos Externos

Para referenciar projetos externos, o processo é semelhante, mas usamos um comando do CLI diferente. Supondo que exista um projeto no diretório projeto, fazemos:

```
> dotnet add reference <NOME_PROJETO>
```

Em seguida, basta usar o comando *using* no arquivo para usar as classes do projeto.

3.3) Referenciando Assemblies em Arquivos

Para referenciar *assemblies* diretamente a partir de arquivos, o processo é ligeiramente diferente. Ao invés do usar o CLI, precisamos editar diretamente o arquivo .csproj:

Deve-se disponibilizar os arquivos .DLL dos *assemblies* em algum lugar conhecido (e.g.: diretório *lib*). Em seguida, acrescenta-se as seguintes diretivas no arquivo .csproj:

```
<ItemGroup>
  <Reference Include="ExternalLibrary">
    <HintPath>..\lib\ExternalLibrary.dll</HintPath>
  </Reference>
</ItemGroup>
```

Por fim, acrescenta-se as diretivas *using* necessárias no código em C#

4) Geração Automática de Documentação

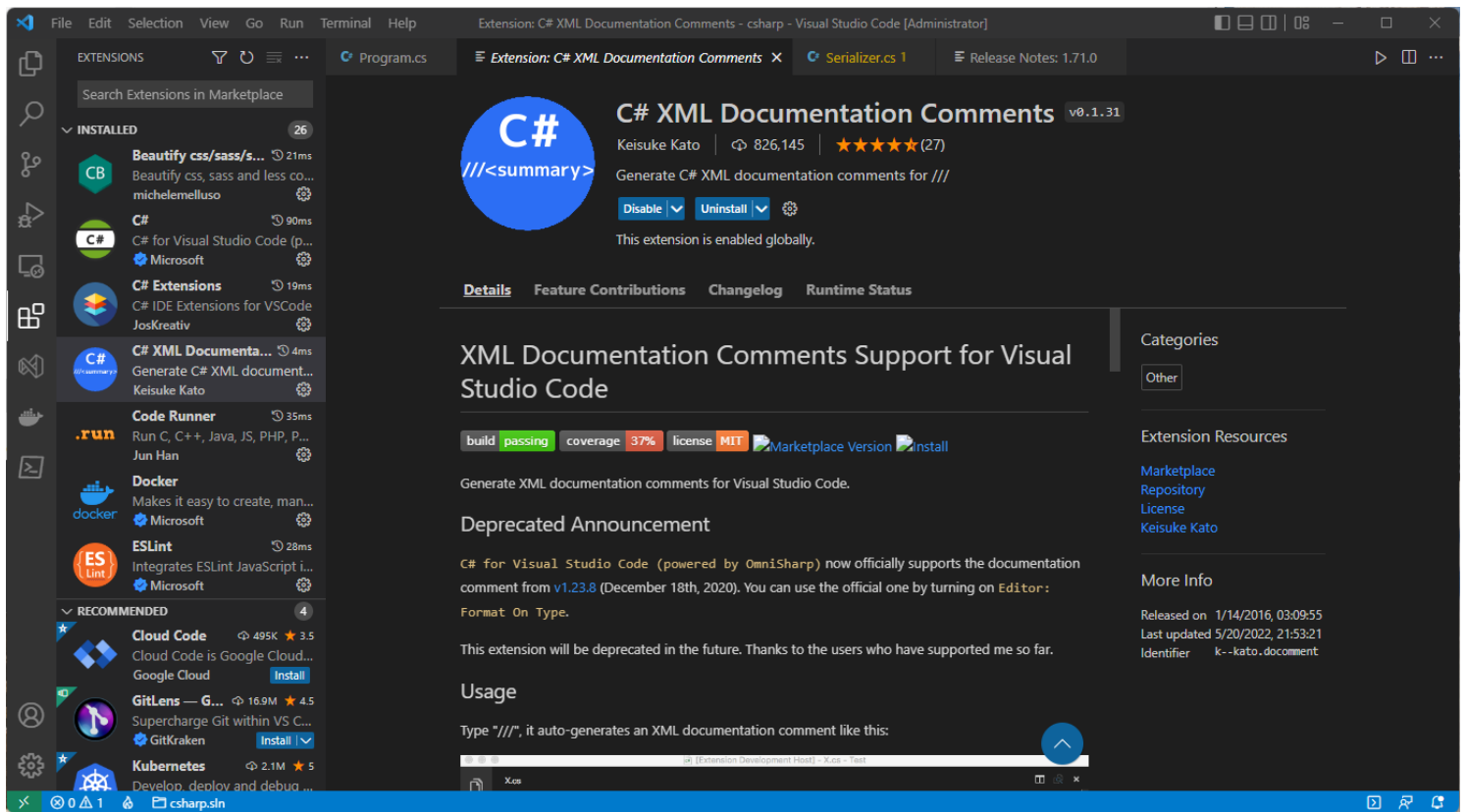
O C# possui um recurso para geração automática de documentação. Esse recurso permite que sejam inseridos comentários com código XML, que serão posteriormente processados por um gerador automático de documentação para gerar, tanto páginas HTML com a documentação de todas as classes, bem como documentos em PDF com essa documentação, se isso for desejado.

A documentação completa de todas as tags XML que podem ser utilizadas para diferentes níveis de documentação pode ser consultada aqui.

Um exemplo de possíveis comentários com essa documentação está colocado a seguir:

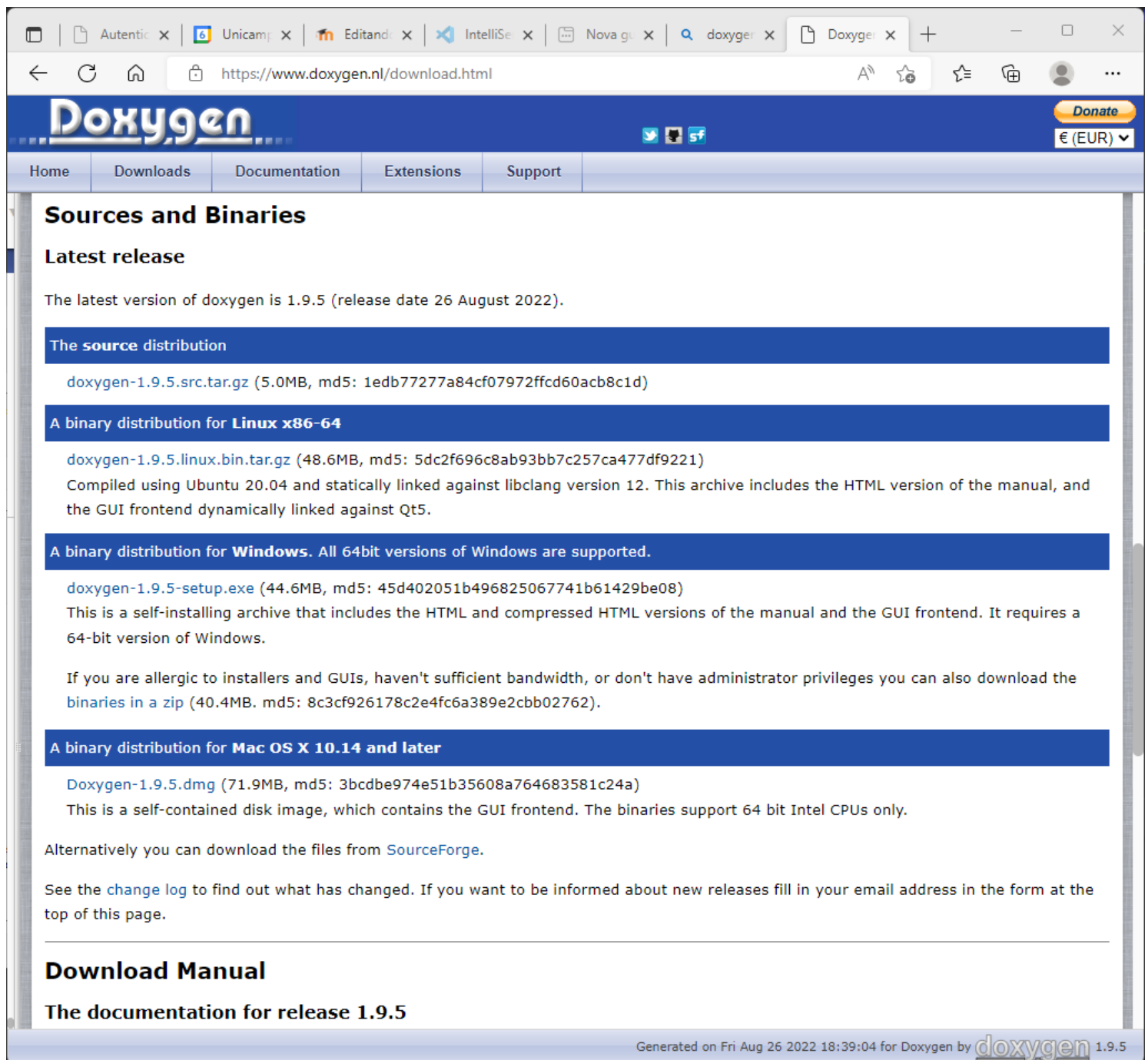
```
/// <summary>
/// Enter description for method someMethod.
/// ID string generated is "M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)".
/// </summary>
/// <param name="str">Describe parameter.</param>
/// <param name="num">Describe parameter.</param>
/// <param name="ptr">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public int someMethod(string str, ref int nm, void* ptr) { return 1; }
```

O Visual Studio Code possui diversos plugins para facilitar a geração desses comentários no seu programa. Um exemplo é a extensão C# XML Documentation Comments. Para começarmos nosso exercício de geração automática de documentação, instale esse plugin no seu VS Code.



Para iniciar um novo comentário, posicione o cursor sobre uma definição de classe ou de método, ou de variável, e digite `///`. Automaticamente, o VS code inserirá um template para o comentário pertinente ao elemento em questão, que você poderá complementar com informações sobre o código que está gerando.

Há vários programas que podem utilizar esses comentários e gerar automaticamente a documentação do seu projeto. Em nosso exercício, utilizaremos o Doxygen:

The image is a screenshot of a web browser displaying the Doxygen website's download page. The browser's address bar shows the URL 'https://www.doxygen.nl/download.html'. The website has a blue header with the 'Doxygen' logo on the left and a 'Donate' button with a currency selector set to '€ (EUR)' on the right. A navigation menu below the header includes links for 'Home', 'Downloads', 'Documentation', 'Extensions', and 'Support'. The main content area is titled 'Sources and Binaries' and features a 'Latest release' section. This section states that the latest version is 1.9.5, released on August 26, 2022. It then lists three distribution options: 1. 'The source distribution' with a link to 'doxygen-1.9.5.src.tar.gz' (5.0MB, md5: 1edb77277a84cf07972ffcd60acb8c1d). 2. 'A binary distribution for Linux x86-64' with a link to 'doxygen-1.9.5.linux.bin.tar.gz' (48.6MB, md5: 5dc2f696c8ab93bb7c257ca477df9221), noting it was compiled using Ubuntu 20.04 and statically linked against libclang version 12. 3. 'A binary distribution for Windows' stating that all 64-bit versions are supported, with a link to 'doxygen-1.9.5-setup.exe' (44.6MB, md5: 45d402051b496825067741b61429be08). It also provides a link to 'binaries in a zip' (40.4MB, md5: 8c3cf926178c2e4fc6a389e2cbb02762) for users who are allergic to installers or lack administrator privileges. Additionally, it mentions a 'Mac OS X 10.14 and later' distribution with a link to 'Doxygen-1.9.5.dmg' (71.9MB, md5: 3bcdbe974e51b35608a764683581c24a). At the bottom of the main content, it suggests downloading files from SourceForge and provides a link to the change log. A footer at the very bottom of the page indicates it was generated on Friday, August 26, 2022, at 18:39:04 for Doxygen by 'doxygen 1.9.5'.

Caso o Doxygen não esteja instalado na sua máquina, faça o download do Doxygen e instale na máquina que vc está usando. Em seguida, inicie o Doxygen GUI Frontend:

Doxygen GUI frontend

File Settings Help

Specify the working directory from which doxygen will run

Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation

Wizard Expert Run

Topics

- Project
- Mode
- Output
- Diagrams

Provide some information about the project you are documenting

Project name:

Project synopsis:

Project version or id:

Project logo: No Project logo selected.

Specify the directory to scan for source code

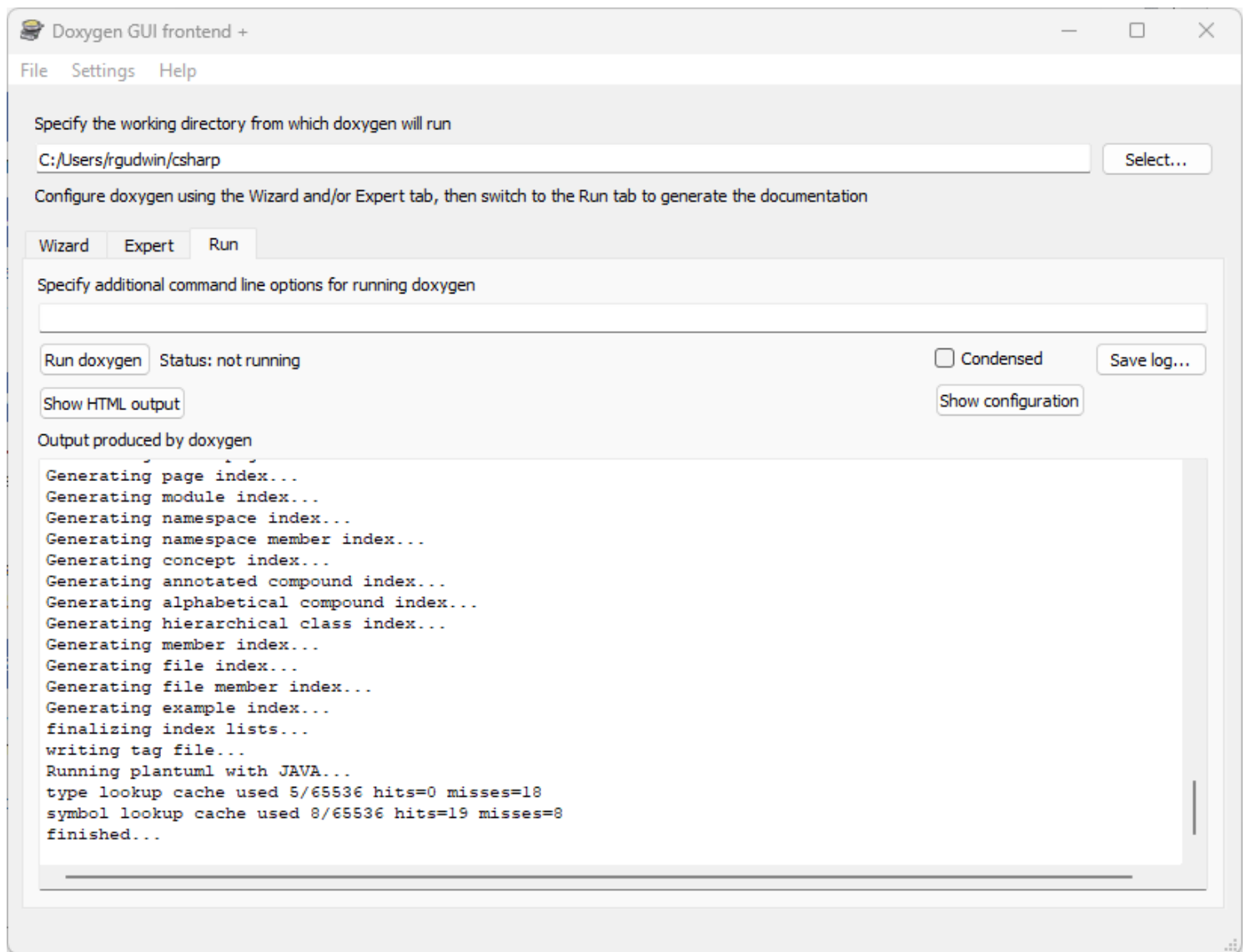
Source code directory:

☐ Scan recursively

Specify the directory where doxygen should put the generated documentation

Destination directory:

Para gerar a documentação automática, você precisa passar algumas informações para o Wizard. Quando a configuração da documentação estiver definida, basta ir para a aba Run e clicar em "Run doxygen". A documentação será gerada automaticamente.



Abra a documentação gerada no seu browser. Dependendo de quão completamente você documentou o seu código, você verá alguma coisa parecida com o seguinte:

My Project

Main Page Packages Classes Search

My Project

Packages

Classes

Class List

proj

Octopus

Panda

Product

Program

Serializer

Class Index

Class Members

proj.Panda Class Reference

Public Member Functions

Panda (string n)
Construtor da classe Panda More...

string getName ()
Esse método retorna o nome do Panda More...

Detailed Description

Essa é a classe Panda

Constructor & Destructor Documentation

Panda()

proj.Panda.Panda (string n)

Construtor da classe Panda

Parameters

n nome do Panda

Member Function Documentation

proj Panda

Generated by doxygen 1.9.5

Parabéns ! Você conseguiu gerar adequadamente a documentação automática do seu projeto.

5) Projeto Final - Parte 2

Desenvolva o minigame **Jewel Collector 2.0**, implementado previamente na aula 2. O objetivo dessa nova versão é melhorar o código anterior através da implementação dos novos conceitos e recursos aprendidos até o momento. Cada classe deve estar em um arquivo separado, com o nome *NomedaClasse.cs*. Particularmente, os seguintes recursos DEVEM NECESSARIAMENTE ser utilizados:

- Devem ser usados, tanto arrays como alguma instância de uma Collection (a seu critério)
- Mecanismo de Eventos para captura dos eventos de teclado e visualização do mapa no console
- Geração de Documentação Automática: Todas as classes, os métodos públicos das classes utilizadas, bem como os fields públicos devem ser comentados e incluídos na documentação gerada.

Implemente o mapa como uma matriz de itens (jewels, obstacles, demais elementos mostrados no mapa). Seu código **deverá** imprimir o mapa de forma simples, como no exemplo abaixo (não necessariamente dessa maneira):

```
void PrintMap() {  
  
    for (int i = 0; i < map.GetLength(0); i++) {  
        for (int j = 0; j < map.GetLength(1); j++) {  
            Console.Write(map[i, j]);  
        }  
        Console.WriteLine("\n");  
    }  
  
}
```

Note que o uso de polimorfismo se fará necessário, pois a variável *map* precisará armazenar os diversos tipos de objetos. Dica: Para escrever o objeto na tela, sobrescreva a método *ToString* em cada classe.

Lembrando que as joias serão do tipo **Red**, valor de 100 pontos, símbolo JR; **Green**, no valor de 50 pontos, símbolo JG; e **Blue**, no valor de 10 pontos, símbolo JB. Obstáculos serão do tipo **Water** com símbolo ##, ou **Tree** com símbolo \$\$\$. Espaços vazios com o símbolo --. Robô com o símbolo ME.

Nesta versão do jogo, o robô inicia com 5 pontos de energia e poderá se deslocar nas quatro direções. A cada deslocamento, ele perde 1 ponto de energia. Quando chegar a zero, o robô não poderá se mover mais; e o jogo termina.

O robô interage com o ambiente podendo **usar** os itens no mapa quando ele estiver em posições adjacentes a estes itens. O efeito do uso depende das características do item. Alguns poderão ser coletados (*collect*), sendo assim removidos do mapa e guardados na sacola do robô. Outros poderão ser usados pelo robô para recarregar (*recharge*) sua energia. Para usar (coletar/recarregar) um item, use a tecla **g**.

Os itens **Tree** e **Blue Jewel** fornecerão energia para o robô. Tree fornecerá 3 pontos de energia, enquanto que Blue Jewel fornecerá 5 pontos. Todas as joias serão coletadas após o uso. Utilize o conceito de interface para realizar essas ações.

Implemente também exceções para tratar os seguintes casos:

- robô tenta se deslocar para uma posição fora dos limites do mapa;
- robô tenta se deslocar para uma posição ocupada por outro item;
- outras situações que achar pertinente o uso de exceções.

As joias e os obstáculos são intransponíveis. Para cada comando executado pelo usuário, imprima o estado atual do mapa, a energia do robô, bem como o estado da sacola do robô.

Quando todas as joias forem coletadas, o jogo avança para a fase seguinte. No qual novas joias e obstáculos serão aleatoriamente posicionados no mapa. A cada nova fase, o mapa aumenta suas dimensões em 1 unidade, até o limite máximo de (30, 30) unidades. A quantidade de itens deverá aumentar proporcionalmente ao tamanho do mapa.

A partir da fase 2, teremos um elemento radioativo, símbolo !!, que retirará 10 pontos de energia, caso o robô passe em posições adjacentes. No entanto, este elemento será transponível. Caso o robô transponha, perderá no mínimo 30 pontos de energia e o elemento radioativo desaparecerá do mapa.

Observação: O que não estiver especificado, fica a critério do desenvolvedor.

Entrega

O Trabalho Final será entregue via Moodle até o final da disciplina de C#, data final para envio: **25/09 23:59**.

O código deverá ser disponibilizado em algum provedor de git, como o GitHub ou Gitlab, e o link para o repositório armazenando esse código deverá ser disponibilizado no Moodle.

Além do código fonte, compilável (com os arquivos de .sln e .csproj), deverá também ser gerada a documentação do projeto, utilizando-se o Doxygen. Essa documentação deverá estar em formato HTML e ser disponibilizada em um subdiretório de nome html.

A avaliação é INDIVIDUAL ! A nota mínima para aprovação na disciplina é 7.0.

Bom trabalho!

Última atualização: sábado, 10 Set 2022, 07:55

NAVEGAÇÃO



Painel

- Página inicial do site

Páginas do site

Meus cursos

INF-0990

Participantes

 Emblemas

 Competências

 Notas

Plano de Desenvolvimento da Disciplina

Atividade 1: 27/08/2022 08:30-12:30

Atividade 2: 03/09/2022 – 08:30-12:30

Atividade 3: 10/09/2022 – 08:30-12:30

Atividade 4: 10/09/2022 – 13:30-17:30

 Aula Teórica 4

 **Aula Prática 4**

 Avaliação Final

 Notas

INF-0991

INF-0992

INF-0993

INF-0994

INF-0995

INF-0996

INF-0997

INF-0998

INF-0999

ADMINISTRAÇÃO



Administração do curso

Você acessou como Victor Akira Hassuda Silva (Sair)
INF-0990