INF-0990 - Programação em C#

Aula Prática 3

1) Genéricos (*Generics*)

Genéricos (*Generics*) funcionam como um espaço reservado (*placeholder*) para que o desenvolvedor especifique o tipo de dado desejado. Portanto, genéricos existem para que possamos escrever código que possa ser reutilizável entre diferentes tipos. Com o uso de genéricos, escrever código de propósito geral se torna muito simples. Por exemplo, quando criamos uma pilha (*Stack*), informamos que deverá ser do tipo *int*:

```
var stack = new Stack<int>();      // Criação da pilha que armazenará inteiros

stack.Push(5);
stack.Push(10);

int x = stack.Pop();
int y = stack.Pop();

Console.WriteLine(x);              // 10
Console.WriteLine(y);              // 5

public class Stack<T>              // Na criação da classe, determinamos que haverá um
tipo genérico T
{
  int position;
  T[] data = new T[100];              // O tipo T é utilizado posteriormente como part
e do código
  public void Push(T obj) => data[position++] = obj;
  public T Pop() => data[--position];
}
```

Um tipo genérico pode ser introduzido na declaração de classes, estruturas e interfaces. E, ainda, pode ter múltiplos parâmetros genéricos. Além disso, podemos usar o conceito de sobrecarga com genéricos:

```
struct Nullable<T>
{
  public T Value { get; set; }
}

// Múltiplos parâmetros
class Dictionary<TKey, TValue> { }

// Sobrecarga
class A { }
class A<T> { }
class A<T1, T2> { }
```

Os tipos genéricos permitidos podem ser restringido com a palavra-chave *where*. Por exemplo:

```
class SomeClass { }
interface Interface1 { }

class GenericClass<T> where T : SomeClass, Interface1 { }  // Restrições
```

Genéricos também podem ser aplicados em herança.

```
class Stack<T> {  }
class SpecialStack<T> : Stack<T> {  }
```

Um tipo genérico pode se auto referenciar:

```
var b1 = new Balloon { Color = "Red", CC = 123 };
var b2 = new Balloon { Color = "Red", CC = 123 };

Console.WriteLine(b1.Equals(b2));            // True

public class Balloon : IEquatable<Balloon>   // Auto referência
{
  public string Color{ get; set; }
  public int CC{ get; set; }

  public bool Equals(Balloon b)
  {
    if (b == null) return false;
    return b.Color == Color && b.CC == CC;
  }
}
```

Variáveis estáticas são únicas para cada tipo genérico definido.

```
Console.WriteLine(++Bob<int>.Count);     // 1
Console.WriteLine(++Bob<int>.Count);     // 2
Console.WriteLine(++Bob<string>.Count);  // 1
Console.WriteLine(++Bob<object>.Count);  // 1

class Bob<T> { public static int Count; }
```

Classes genéricas não são covariantes, o que pode prejudicar a reusabilidade do código. Ser covariante significa que se A é convertível em B (por polimorfismo), então X<A> também seria convertível em X<B>.
Analise o código abaixo:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash(bears);               // Não irá compilar!

class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T>                 // Uma pilha simples
{
  int position;
  T[] data = new T[100];
  public void Push(T obj) => data[position++] = obj;
  public T Pop() => data[--position];
}

static class ZooCleaner
{
  public static void Wash(Stack<Animal> animals) { }   // Embora Bear seja um Animal,
 a falta de covariância impede a execução
}
```

Uma solução é transformar o método *Wash* em genérico e usar restrições de tipos.

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash(bears);              // Funciona!

class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj) => data[position++] = obj;
  public T Pop() => data[--position];

}

static class ZooCleaner
{
  public static void Wash<T>(Stack<T> animals) where T : Animal {  }      // Transform
ar o método em genérico e usar restrições de tipos
}
```

Interfaces são covariantes e podem fazer uso de genéricos, desde que marcado com o operador *out*.
Veja o exemplo:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;                          // Legal
bears.Push(new Bear());

public interface IPoppable<out T> { T Pop(); }            // Implementação de genéricos
public interface IPushable<in T> { void Push(T obj); }    // Implementação de genéricos

class Animal {}
class Bear : Animal {}
class Camel : Animal {}

// Note que Stack pode implementar ambas as interfaces
public class Stack<T> : IPoppable<T>, IPushable<T>
{
  int position;
  T[] data = new T[100];
  public void Push(T obj) => data[position++] = obj;
  public T Pop() => data[--position];
}
```

2) Delegates

Em C#, métodos podem ser passados como parâmetros para outros métodos. Uma declaração de um tipo delegate é similar a declaração de um método abstrato. Por exemplo:

```
Transformer t = Square;              // Criação de um tipo delegate
int result = t(3);                   // Invocação do delegate
Console.WriteLine(result);           // 9

int Square (int x) => x * x;

delegate int Transformer(int x);     // Declaração de um tipo delegate
```

Também podemos instanciar delegates com o operador *new*.

```
Transformer t = new Transformer(Square);   // Instanciação de um tipo delegate
int result = t(3);                         // Invocação do delegate
Console.WriteLine(result);                 // 9

int Square (int x) => x * x;

delegate int Transformer(int x);           // Declaração de um tipo delegate
```

Uma variável delegate pode ser determinada dinamicamente. Sendo útil para a escrita e reuso de métodos:

```
int[] values = { 1, 2, 3 };
Transform(values, Square);                          // Acoplar o método Square
foreach(int i in values) Console.Write(i + " ");  // 1 4 9

Console.WriteLine();

values = new int[] { 1, 2, 3 };
Transform(values, Cube);                             // Acoplar o método Cube
foreach(int i in values) Console.Write(i + " ");   // 1 8 27

void Transform(int[] values, Transformer t)        // Método recebe um delegate como p
arâmetro
{
  for (int i = 0; i < values.Length; i++)
    values[i] = t(values[i]);                      // Invocação do delegate
}

int Square(int x) => x * x;
int Cube(int x) => x * x * x;

delegate int Transformer(int x);                   // Declaração de um tipo delegate
```

Delegates também podem ser usados em conjunto com métodos estáticos, por exemplo:

```
Transformer t = Test.Square;
Console.WriteLine(t(10));                      // 100

class Test
{
  public static int Square(int x) => x * x;   // Método estático
}

delegate int Transformer(int x);               // Declaração de um tipo delegate
```

Todas as instâncias de *delegates* tem múltiplas capacidades de conversão (*multicast*). Os operadores + e += realizam a concatenação de um *delegate* com outro. *Delegates* concatenados, são executados em sequência, um após o outro. Os operadores - e -= promovem a remoção de um dos métodos do *delegate* concatenado.

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;              // Adição de segundo método

d();                           // SomeMethod1
                               // SomeMethod2

d -= SomeMethod1;              // Remoção do primeiro método
d();                           // SomeMethod2

void SomeMethod1 () => Console.WriteLine("SomeMethod1");
void SomeMethod2 () => Console.WriteLine("SomeMethod2");

delegate void SomeDelegate();   // Declaração de um tipo delegate
```

Um tipo delegate pode conter parâmetros genéricos. Com essa definição, podemos escrever um método *Transform* generalizado, que funciona para qualquer tipo.

```
int[] values = { 1, 2, 3 };
Util.Transform(values, Square);                    // Acoplamento dinâmico de método S
quare
foreach(int i in values) Console.Write(i + " ");   // 1 4 9

int Square (int x) => x * x;

public delegate T Transformer<T>(T arg);           // Declaração de um tipo delegate c
om uso de genéricos

public class Util
{
  public static void Transform<T>(T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t(values[i]);
  }
}
```

Tipos delegates são completamente incompatíveis entre si, mesmo quando suas assinaturas são idênticas:

```
D1 d1 = Method1;
D2 d2 = d1;            // Erro de compilação

static void Method1() { }

delegate void D1();
delegate void D2();
```

No entanto, podemos compor delegates, veja o exemplo:

```
D1 d1 = Method1;
D2 d2 = new D2(d1);      // Legal

void Method1() { }

delegate void D1();
delegate void D2();
```

Instâncias de delegates são consideradas idênticas se elas tem o mesmo método alvo.

```
D d1 = Method1;
D d2 = Method1;
Console.WriteLine(d1 == d2);         // True

static void Method1() { }

delegate void D();
```

Um delegate pode ter parâmetros mais específicos que o seu método alvo.

```
StringAction sa = new StringAction(ActOnObject);
sa("hello");

static void ActOnObject(object o) => Console.WriteLine(o);       // hello

delegate void StringAction(string s);
```

3) Eventos (*Events*)

Um *design-pattern* muito comum em engenharia de software é o *publish-subscribe*. Nesse *pattern*, alguns objetos publicam eventos, que podem ser de interesse de outros objetos, que então subscrevem-se para recebê-lo, quando o mesmo ocorrer. Esse *pattern* pode ser implementado em C# usando-se *delegates*. Entretanto, o uso de delegates permitiria que os subscribers fizessem modificações nos delegates, que poderiam comprometer o mecanismo. Por esse motivo, o C# criou o tipo event, que funciona como um delegate, mas possui proteção contra essa interferência inadequada. Eventos podem simplesmente indicar que alguma coisa aconteceu, ou podem incluir argumentos, que podem ser passados como parâmetros. Classes que publicam eventos devem declarar os eventos que publicam usando o tipo event, criar métodos que sejam protected e virtual com tipicamente um nome onNomeEvento, que invocam o evento, quando necessário, durante seu funcionamento. Classes que desejam se subscrever a um evento podem fazê-lo usando o mecanismo de concatenção de delegates: +=. Para deixar de receber esses eventos, é só usar o -=. Nesta primeira versão, implementamos o mecanismo de eventos com o uso de um delegate.

```
var stock = new Stock("MSFT");
stock.PriceChanged += ReportPriceChange;    // Define a função que será executada quan
do o evento ocorrer
stock.Price = 123;                          // Price changed from 0 to 123
stock.Price = 456;                          // Price changed from 123 to 456

void ReportPriceChange(decimal oldPrice, decimal newPrice)
{
  Console.WriteLine("Price changed from " + oldPrice + " to " + newPrice);
}

public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);

public class Stock
{
  string symbol;
  decimal price;

  public Stock(string symbol) { this.symbol = symbol; }

  public event PriceChangedHandler PriceChanged;        // Variável de evento

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      decimal oldPrice = price;
      price = value;
      if (PriceChanged != null)
        PriceChanged(oldPrice, price);   // Dispara o evento
    }
  }
}
```

Aqui temos uma implementação alternativa, em que não é necessário definir o *delegate*. Observe que utilizamos um método *protected* e *virtual*, que é chamado quando o evento é gerado. Para subscrever ao evento *PriceChanged*, associamos um *callback* a ele usando o mecanismo de concatenção de *delegates* (por meio do operador +=). Ao fazer isso, a função *stock_PriceChanged* vira uma função de callback quando o evento ocorre.

```
Stock stock = new Stock("THPW");
stock.Price = 27.10M;
stock.PriceChanged += stock_PriceChanged;
stock.Price = 31.59M;

void stock_PriceChanged(object sender, EventArgs e)
{
  Console.WriteLine("New price = " + ((Stock)sender).Price);
}

public class Stock
{
  string symbol;
  decimal price;

  public Stock(string symbol) { this.symbol = symbol; }

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged(EventArgs e)
  {
    PriceChanged?.Invoke(this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged(EventArgs.Empty);
    }
  }
}
```

4) Expressões Lambda

São um modo alternativo de definir uma função, por meio de uma expressão que utiliza o operador Lambda (=>). Podem ser utilizadas para definir, de maneira sintética, métodos e delegates de funções simples.

```
Transformer sqr = x => x * x;
Console.WriteLine(sqr(3));                      // 9

// Usando um bloco como corpo da expressão
Transformer sqrBlock = x => { return x * x; };
Console.WriteLine(sqrBlock(3));                 // 9

// Usando um delegate genérico
Func<int, int> sqrFunc = x => x * x;
Console.WriteLine(sqrFunc(3));                  // 9

// Sem argumentos
Func<string> greeter = () => "Hello, world";
Console.WriteLine(greeter());                   // Hello, world

// Com tipagem implícita
var greeter2 = () => "Hello, world";
Console.WriteLine(greeter2());                  // Hello, world

// Usando múltiplos argumentos
Func<string, string, int> totalLength = (s1, s2) => s1.Length + s2.Length;
int total = totalLength("hello", "world");
Console.WriteLine(total);                       // 10

// Explicitamente especificando os parâmetros
Func<int, int> sqrExplicit = (int x) => x * x;
Console.WriteLine(sqrExplicit(3));              // 9

delegate int Transformer(int i);
```

Uma expressão lambda pode referenciar variáveis locais e parâmetros do método em que ela foi definida.

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine(multiplier(3));           // 6
factor = 10;
Console.WriteLine(multiplier(3));           // 30

// Valores são lidos quando a expressão é executada, não quando ela foi declarada

// Expressões lambda podem alterar variáveis capturadas por elas mesmas
int seed = 0;
Func<int> natural = () => seed++;    // Incremento
Console.WriteLine(natural());           // 0
Console.WriteLine(natural());           // 1
Console.WriteLine(seed);                // 2
```

Variáveis capturadas tem sua vida útil correspondente ao do delegate.

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++;               // Retorna um closure
}

Func<int> natural = Natural();
Console.WriteLine(natural());      // 0
Console.WriteLine(natural());      // 1
```

Já uma variável local instanciada dentro de uma expressão lambda é unica por invocação do delegate.

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed++; };
}

Func<int> natural = Natural();
Console.WriteLine(natural());           // 0
Console.WriteLine(natural());           // 0
```

Expressões lambda podem ser definidas como estáticas (*static*).

```
Func<int, int> multiplier = static n => n * 2;       // Expressão lambda estática

Console.WriteLine(multiplier(123));

Foo();

void Foo()
{
  Console.WriteLine(Multiply(123));

  static int Multiply (int x) => x * 2;              // Método local estático
}
```

Quando capturamos a variável de iteração em um laço de repetição, o C# trata essa variável como se ela fosse declarada fora do laço. Isso significa que a mesma variável é capturada a cada iteração.

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions[i] = () => Console.Write(i);

foreach(Action a in actions) a();     // 333 (ao invés de 012)
```

Cada closure captura o mesmo valor 3 para a variável *i*. Veja outro exemplo:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.Write(i);
i = 1;
actions[1] = () => Console.Write(i);
i = 2;
actions[2] = () => Console.Write(i);
i = 3;
foreach(Action a in actions) a();    // 333
```

A solução nesse caso, se quisermos escrever o valor 012, é definir a variável de iteração para uma variável local, que esteja dentro do escopo:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions[i] = () => Console.Write(loopScopedi);
}

foreach(Action a in actions) a();              // 012
```

5) Exceções

Códigos C# podem gerar exceções durante seu funcionamento. O tratamento dessas exceções pode ser realizado por meio de blocos do tipo *try/catch*. No exemplo abaixo, temos uma exceção *DivideByZeroException* causada pela divisão por zero.

```
int y = Calc(0);        // Exceção causada pela divisão por zero
Console.WriteLine(y);

int Calc(int x) { return 10 / x; }
```

Podemos usar o bloco *try/catch* para capturar essa exceção e dar o devido tratamento.

```
try                                     // Bloco onde o código tentará ser executado
{
  int y = Calc(0);
  Console.WriteLine(y);
}
catch(DivideByZeroException ex)         // Caso ocorra alguma exceção, a capturamos
 e damos o devido tratamento
{
  Console.WriteLine("x cannot be zero");  // x cannot be zero
}

Console.WriteLine("program completed");

int Calc(int x) { return 10 / x; }
```

Podemos capturar diversos tipos de exceções com o uso de múltiplos blocos *catch*.

```
static void Main() { MainMain("one"); }

static void MainMain(params string[] args)
{
  try
  {
    byte b = byte.Parse(args[0]);
    Console.WriteLine(b);
  }
  catch(IndexOutOfRangeException)                                    // Capturamos exceçã
o IndexOutOfRangeException
  {
    Console.WriteLine("Please provide at least one argument");
  }
  catch(FormatException)                                             // Capturamos exceçã
o FormatException
  {
    Console.WriteLine("That's not a number!");
  }
  catch(OverflowException)                                            // Capturamos exceç
ão OverflowException
  {
    Console.WriteLine("You've given me more than a byte!");
  }
}

static int Calc(int x) { return 10 / x; }
```

Ainda podemos usar um bloco *finally*, que é executado sempre ao final do bloco *try* ou *catch*. Normalmente, se usa o bloco *finally* para executar alguma rotina de limpeza final. Por exemplo:

```
File.WriteAllText("file.txt", "test");         // Criar o arquivo "file.txt" com o con
teúdo "test"
ReadFile();

void ReadFile()
{
  StreamReader reader = null;
  try
  {
    reader = File.OpenText("file.txt");        // Abre o arquivo
    if (reader.EndOfStream) return;
    Console.WriteLine(reader.ReadToEnd());    // test
  }
  finally
  {
    if (reader != null) reader.Dispose();     // Fecha o arquivo
  }
}
```

Um forma mais simples de abrir um arquivo ou recurso e garantir o seu fechamento é usar a palavra-chave *using*. Assim, criamos um escopo no qual o recurso estará disponível.

```
File.WriteAllText("file.txt", "test");
ReadFile();

void ReadFile()
{
  using(StreamReader reader = File.OpenText("file.txt"))    // Arquivo aberto dentro d
o escopo do using
  {
    if(reader.EndOfStream) return;
    Console.WriteLine(reader.ReadToEnd());                 // test
  }

  // Arquivo já foi fechado
}
```

Podemos também empregar o comando *using var* para usar um recurso dentro do escopo atual.

```
if(File.Exists("file.txt"))
{
  using var reader = File.OpenText("file.txt");     // Abre o arquivo somente para o b
loco do if
  Console.WriteLine(reader.ReadLine());
}

// Arquivo já foi fechado
```

Exceções podem ser lançada durante o tempo de execução ou pelo próprio usuário. Quando se deseja lançar uma exceção, usamos a palavra-chave *throw*.

```
try
{
  Display(null);
}
catch(ArgumentNullException ex)
{
  Console.WriteLine("Caught the exception");            // Caught the exception
}

static void Display(string name)
{
  if(name == null)
    throw new ArgumentNullException(nameof(name));      // Exceção lançada pelo própri
o usuário

  Console.WriteLine(name);
}
```

6) Métodos de Extensão

São um mecanismo do C# que permite que um tipo pré-existente seja estendido com novos métodos, sem ser necessária a alteração do tipo original. O modificador *this* é aplicado ao primeiro parâmetro de cada um desses métodos.

```
Console.WriteLine("Perth".IsCapitalized());              // True

// Equivalnete a:
Console.WriteLine(StringHelper.IsCapitalized("Perth"));  // True

// Interfaces também podem ser estendidas
Console.WriteLine("Seattle".First());                    // S

public static class StringHelper
{
  public static bool IsCapitalized (this string s)       // Aplicação do modificador
 this
  {
    if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper(s[0]);
  }

  public static T First<T>(this IEnumerable<T> sequence)    // Estendendo interface IE
numerable
  {
    foreach (T element in sequence)
      return element;

    throw new InvalidOperationException("No elements!");
  }
}
```

Métodos de extensão, como métodos de instância, fornecem uma forma simples de encadear funções:

```
string x = "sausage".Pluralize().Capitalize();                  // Encadeamento de f
unções
Console.WriteLine(x);                                           // SAUSAGES

// Equivalente a:
string y = StringHelper.Capitalize(StringHelper.Pluralize("sausage"));
Console.WriteLine(y);                                           // SAUSAGES

public static class StringHelper
{
  public static string Pluralize(this string s) => s + "s";       // Implementação sim
ples para o plural

  public static string Capitalize(this string s) => s.ToUpper();  // Implementação par
a o capitalizar
}
```

7) Tuplas (*Tuples*)

São uma maneira de agregar várias variáveis em uma só, principalmente para poder realizar um retorno múltiplo de um método, sem que seja necessário criar-se um tipo específico para essa finalidade.
Tuplas são tipos mutáveis de valor.

```
var bob = ("Bob", 23);           // Compilador infere o tipo

Console.WriteLine(bob.Item1);    // Bob
Console.WriteLine(bob.Item2);    // 23

var joe = bob;                   // joe é um cópia de valor de bob
joe.Item1 = "Joe";               // Alterando atributo de joe

Console.WriteLine(bob);          // (Bob, 23)
Console.WriteLine(joe);          // (Joe, 23)
```

Embora o uso de *var* seja comum com o uso de tuplas, nós podemos especificar o tipo dos valores.

```
(string,int) bob  = ("Bob", 23);

Console.WriteLine(bob.Item1);     // Bob
Console.WriteLine(bob.Item2);     // 23
```

Métodos podem retornar tuplas, e as podemos receber em variáveis separadas, como na desconstrução.

```
(string, int) person = GetPerson();

Console.WriteLine(person.Item1);    // Bob
Console.WriteLine(person.Item2);    // 23

static (string,int) GetPerson() => ("Bob", 23);
```

Tuplas podem ser nomeadas:

```
var tuple = (Name:"Bob", Age:23);

Console.WriteLine(tuple.Name);     // Bob
Console.WriteLine(tuple.Age);      // 23
```

```
var person = GetPerson();

Console.WriteLine(person.Name);    // Bob
Console.WriteLine(person.Age);     // 23

static (string Name, int Age) GetPerson() => ("Bob", 23);
```

Tuplas podem ser desconstruídas em variáveis separadas. Por exemplo:

```
var bob = ("Bob", 23);

(string name, int age) = bob;   // Desconstrução da tupla bob em variáveis separadas

Console.WriteLine(name);        // Bob
Console.WriteLine(age);         // 23
```

E podemos construir tuplas com múltiplos valores:

```
var (name, age, sex) = GetBob();

Console.WriteLine(name);        // Bob
Console.WriteLine(age);         // 23
Console.WriteLine(sex);         // M

static (string, int, char) GetBob() => ("Bob", 23, 'M');
```

8) Padrões (*Patterns*)

Em muitas situações, pode ser conveniente testar se um objeto segue ou não um determinado padrão. O C# possui o operador *is*, que verifica se um objeto ou variável segue ou não um padrão.

```
Console.WriteLine(IsJanetOrJohn("Janet"));        // True
Console.WriteLine(IsJanetOrJohn("john"));         // True

bool IsJanetOrJohn (string name) =>
  name.ToUpper() is var upper && (upper == "JANET" || upper == "JOHN");
```

O uso do operador *is* nos habilita a usar comparações relacionais de modo mais simples, sem precisar repetir a variável avaliada.

```
Foo(3);                           // three

void Foo (object obj)
{
  if(obj is 3) Console.WriteLine("three");
}

int x = 150;

if (x is > 100) Console.WriteLine("x is greater than 100");                 // x is g
reater than 100

if (x is > 100 and < 200) Console.WriteLine("x is between 100 and 200");    // x is b
etween 100 and 200
```

O uso de comparações com o operador *switch* traz uma simplificação quando desejamos executar múltiplas avaliações, por exemplo:

```
Console.WriteLine(GetWeightCategory(15));                    // underweight
Console.WriteLine(GetWeightCategory(20));                    // normal
Console.WriteLine(GetWeightCategory(25));                    // overweight

string GetWeightCategory (decimal bmi) => bmi switch
{
  < 18.5m => "underweight",
  < 25m => "normal",
  < 30m => "overweight",
  _ => "obese"
};
```

```
Console.WriteLine(ShouldAllow(new Uri("http://www.linqpad.net")));        // True
Console.WriteLine(ShouldAllow(new Uri("ftp://ftp.microsoft.com")));       // False
Console.WriteLine(ShouldAllow(new Uri("tcp:foo.database.windows.net")));  // False

bool ShouldAllow(Uri uri) => uri switch
{
  { Scheme.Length: 4, Port: 80 } => true,
  _ => false
};
```

Podemos adicionar comparações de união e interseção com os operadores AND e OR.

```
Console.WriteLine(IsJanetOrJohn("Janet"));           // True
Console.WriteLine(IsVowel('e'));                     // True
Console.WriteLine(Between1And9(5));                  // True
Console.WriteLine(IsLetter('!'));                    // False

bool IsJanetOrJohn(string name) => name.ToUpper() is "JANET" or "JOHN";

bool IsVowel(char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';

bool Between1And9(int n) => n is >= 1 and <= 9;

bool IsLetter(char c) => c is >= 'a' and <= 'z'
                         or >= 'A' and <= 'Z';
```

Podemos verificar se o objeto é de um determinado tipo e valor, sem precisar realizar a conversão explicitamente.

```
object obj = 2m;                // decimal

Console.WriteLine(obj is < 3m);  // True
Console.WriteLine(obj is < 3);   // False
```

Com o uso do operador *is not* podemos verificar se um objeto não é de um determinado tipo.

```
object obj = true;                       // obj is not a string

if (obj is not string)
  Console.WriteLine("obj is not a string");
```

Ainda é possivel especificar atributos de um tipo para ser avaliado, por exemplo, o comprimento (*Length*) de uma string.

```
object obj = "test";                    // string with length of 4

if (obj is string { Length:4 })
  Console.WriteLine("string with length of 4");
```

9) Atributos

Também chamados de anotações, são um mecanismo de extensão para adicionar informação customizada a elementos de código (assemblies, tipos, membros, valores de retorno, parâmetros e tipos genéricos de parâmetros).

```
new Foo();          // Gerará um aviso de que a classe está obsoleta

[Obsolete]
public class Foo
{
}
```

Especialmente útil para adicionar informações de outras fontes, por exemplo, um tipo XML.

```
void Main()
{
}

[XmlType("Customer", Namespace = "http://oreilly.com")]
public class CustomerEntity
{
}
```

10) Sobrecarga de Operadores

Operadores podem ser sobrecarregados como qualquer método. Para isso, declaramos o operador como uma função. Veja os exemplos abaixo:

```
Note B = new Note(2);
Note CSharp = B + 2;

Console.WriteLine(CSharp.SemitonesFromA);     // 4

CSharp += 2;
Console.WriteLine(CSharp.SemitonesFromA);     // 6

public struct Note
{
  int value;
  public int SemitonesFromA => value;

  public Note (int semitonesFromA) { value = semitonesFromA; }  // Construtor

  public static Note operator + (Note x, int semitones)         // Sobrecarga do opera
dor de soma
  {
    return new Note (x.value + semitones);
  }

}
```

Operadores de conversões implícitas e explicitas também podem ser sobrecarregadas:

```
Note n = (Note)554.37;      // Conversão explicita
double x = n;               // Conversão implícita

Console.WriteLine(x);       // 554.3652

public struct Note
{
  int value;
  public int SemitonesFromA { get { return value; } }

  public Note (int semitonesFromA) { value = semitonesFromA; }

  // Conversão para hertz
  public static implicit operator double (Note x) =>          // Sobrecarga de operado
r de conversão implícita
    440 * Math.Pow (2, (double)x.value / 12);

  // Conversão de hertz
  public static explicit operator Note (double x) =>          // Sobrecarga de operado
r de conversão explicita
    new Note ((int)(0.5 + 12 * (Math.Log (x / 440) / Math.Log (2))));
}
```

Operadores de comparação (*true/false*) podem ser sobrecarregados, por exemplo:

```
SqlBoolean a = SqlBoolean.Null;

if(a)
  Console.WriteLine("True");
else if(!a)
  Console.WriteLine("False");
else
  Console.WriteLine("Null");

public struct SqlBoolean
{
  public static bool operator true(SqlBoolean x) => x.m_value == True.m_value;
// Sobrecarga operador true

  public static bool operator false(SqlBoolean x) => x.m_value == False.m_value;
// Sobrecarga operador false

  public static SqlBoolean operator ! (SqlBoolean x)
// Sobrecarga operador not
  {
    if (x.m_value == Null.m_value) return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }

  public static readonly SqlBoolean Null = new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True = new SqlBoolean(2);

  SqlBoolean (byte value) { m_value = value; }
// Construtor
  byte m_value;
}
```

Última atualização: quarta, 7 Set 2022, 23:48

Atividade 2: 03/09/2022 – 08:30-12:30
Atividade 3: 10/09/2022 – 08:30-12:30
Aula Teórica 3
Aula Prática 3
Atividade 4: 10/09/2022 – 13:30-17:30
INF-0991
INF-0992
INF-0993
INF-0994
INF-0995
INF-0996
INF-0997
INF-0998
INF-0999

ADMINISTRAÇÃO

Administração do curso
