INF-0990 - Programação em C#

Aula Prática 2

1) Classes

Uma classe em C# é declarada com a palavra-chave *class*. Para ser instanciada, ou seja, para se criar objetos a partir de uma determinada classe, usamos o operador *new*. Classes são estruturas de programação que armazenam variáveis e métodos, por exemplo:

```
var o = new Octopus();    // Criação de um objeto do tipo Octopus

Console.WriteLine(o.Age); // 10

class Octopus
{
  string name;            // Atributo do tipo string, sem valor
  public int Age = 10;    // Atributo do tipo int, com valor
}
```

Atributos do tipo leitura (*readonly*) permitem a criação de classes imutáveis. Isto é, uma vez que o objeto seja instanciado, os atributos com a propriedade de leitura não podem ser modificados.

```
var o = new Octopus("Jack");

Console.WriteLine(o.Name);  // Jack
o.Legs = 20;                // Erro de compilação

class Octopus
{
  public readonly string Name;
  public readonly int Legs = 8;

  public Octopus(string name)
  {
    Name = name;
  }
}
```

Constantes são fatoradas em tempo de compilação e suas chamadas são substituídas pelo valor definido. Quando a definição de uma constante envolve cálculos, estes são realizados em tempo de

compilação.

```
Console.WriteLine(Test.Message);    // Hello World

public class Test
{
  public const string Message = "Hello World";
  public const double twoPI  = 2 * System.Math.PI;   // Computação executada em tempo
 de compilação
}
```

Métodos podem ser declarados sem especificar o corpo, em um modo também chamado de "encorpados por expressão" (*expression-embodied*), como no exemplo em Foo2. Dessa forma, Foo1 e Foo2 são equivalentes:

```
Console.WriteLine(Foo1(10) == Foo2(10));    // True

int Foo1(int x) { return x * 2; }
int Foo2(int x) => x * 2;                   // Expression-embodied notation
```

Observe que a representação encorpada por expressão utiliza uma notação alternativa que é chamada de "Expressão Lambda", que veremos em maiores detalhes na próxima aula. Ela é apresentada aqui, para que gradativamente nos acostumemos a ela, pois é uma notação comum em algumas linguagens de programação mais modernas. Entretanto, é necessário observar que essa é uma notação alternativa, que pode não ser utilizada em muitos casos.

Também podemos declarar métodos locais, ou seja, métodos dentro de outros métodos.

```
WriteCubes();

void WriteCubes()
{
  Console.WriteLine(Cube(3));    // 27
  Console.WriteLine(Cube(4));    // 64
  Console.WriteLine(Cube(5));    // 125

  int Cube(int value) => value * value * value;   // Método local
}
```

Métodos de alto nível (*top-level*) são implicitamente locais, assim, podemos acessar variáveis declaradas externamente ao método.

```
int x = 3;
Foo();        // 3

// Foo é um método local
void Foo() => Console.WriteLine(x);    // Podemos acessar x
```

Em C#, a sobrecarga de métodos é realizada muito similar a outras linguagens de programação orientadas a objetos. Declaramos os métodos com o mesmo nome, porém alteramos os conjunto de parâmetros recebidos, isto é, a *assinatura* do método é alterada.

```
Octopus o = new Octopus();

o.Foo(123);         // int
o.Foo(123.0);       // double
o.Foo(123, 123F);   // int, float
o.Foo(123F, 123);   // float, int

class Octopus
{
    public void Foo(int x)         { Console.WriteLine("int"); }
    public void Foo(double x)      { Console.WriteLine("double"); }
    public void Foo(int x, float y) { Console.WriteLine("int, float"); }
    public void Foo(float x, int y) { Console.WriteLine("float, int"); }
}
```

No entanto, algumas sobrecargas são proibidas, como modificarmos apenas o retorno do método. A assinatura do método não engloba o tipo do retorno, assim, o compilador não consegue inferir qual método desejamos chamar:

```
class Octopus
{
  void Foo(int x) {}
  float Foo(int x) {}      // Erro de compilação
}
```

Métodos construtores são utilizados em programação orientada a objetos para inicializar atributos e executar demais funcionalidades do objeto no momento de sua instanciação. Desse modo, métodos construtores são declarados com o mesmo nome da classe, porém sem a definição do tipo de retorno.

```
Panda p = new Panda("Petey");   // Chamada de construtor

public class Panda
{
  string name;
  public Panda(string n)        // Definição do Construtor: observe que o método não t
em um tipo de retorno
  {
    name = n;                   // Código de inicialização da variável name
  }
}
```

Assim como os métodos tradicionais, os construtores podem ser sobrecarregados, por exemplo:

```
new Wine(78);
new Wine(78, 2001);

public class Wine
{
  public decimal Price;
  public int Year;

  public Wine(decimal price)    // Construtor
  {
    Price = price;
  }

  public Wine(decimal price, int year) : this (price)  // Construtor
  {
    Year = year;
  }
}
```

Note que, neste caso, a palavra-chave *this* foi empregada para se referir ao outro construtor, e passar o valor da variável *price*. Assim, um construtor é capaz de chamar os demais construtores da classe. Além disso, construtores podem ser declarados como privados, isto é, quando se deseja controlar a criação de instâncias de uma determinada classe.

```
Class1 c1 = Class1.Create();  // OK
Class1 c2 = new Class1();    // Erro: Não irá compilar

public class Class1
{
  Class1() { }        // Construtor privado

  public static Class1 Create()
  {
    // Execução de alguma lógica para configurar a instanciação da classe Class1
    return new Class1();
  }
}
```

Uma funcionalidade muito usada na linguagem C# é o uso de desconstrutores. Métodos desconstrutores são declarados na classe e determinam como os atributos podem ser separados quando a desconstrução é chamada. Por exemplo:

```
var rect = new Rectangle(3, 4);              // Criação de objeto do tipo Rectangle

(float width, float height) = rect;            // Desconstrução
Console.WriteLine(width + " " + height);    // 3 4

// O uso de tipagem implícita é usada com a palavra-chave var:
var (x, y) = rect;                           // Desconstrução
Console.WriteLine(x + " " + y);

// Se as variáveis já foram declaradas, podemos apenas chamar o desconstrutor
(x, y) = rect;
Console.WriteLine(x + " " + y);

// A partir do C#, é permitido o uso de sintaxe mista
double x1 = 0;
(x1, double y2) = rect;
Console.WriteLine(x + " " + y);

class Rectangle
{
  public readonly float Width, Height;

  public Rectangle(float width, float height)
  {
    Width = width;
    Height = height;
  }

  public void Deconstruct(out float width, out float height)    // Declaração do métod
o desconstrutor
  {
    width = Width;
    height = Height;
  }
}
```

Campos e propriedades também podem ser inicializados em uma declaração única em conjunto ou diretamente depois da construção:

```
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false }; // Inicializ
ação em conjunto, com a nomeação das propriedades e campos
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false }; // Inicializ
ação após a chamada do construtor

Console.WriteLine(b1.Name);   // Bo
Console.WriteLine(b2.Name);   // Bo

public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

C# também permite que parâmetros sejam opcionais, para isso, eles devem especificar um valor padrão.

```
Bunny b1 = new Bunny(name: "Bo");
Bunny b2 = new Bunny(name: "Bo", likesCarrots: true); // Sobrescrita de parâmetro opci
onal likesCarrots

Console.WriteLine(b1.LikesCarrots);   // False
Console.WriteLine(b2.LikesCarrots);   // True

public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny (string name, bool likesCarrots = false, bool likesHumans = false) // 1
parâmetro obrigatório e 2 opcionais
  {
    Name = name;
    LikesCarrots = likesCarrots;
    LikesHumans = likesHumans;
  }
}
```

A palavra-chave *this* é uma referência ao próprio objeto.

```
new Panda().Marry(new Panda());

public class Panda
{
  public Panda Mate;

  public void Marry(Panda partner)
  {
    Mate = partner;
    partner.Mate = this;        // Atribui o próprio objeto
  }
}
```

Propriedades (*property*) se parecem com atributos/campos, porém internamente contém lógica, como se fossem métodos. Properties são definidas com as palavras-chave *get* e *set* e implementam lógica de como definir e obter uma variável.

```
var stock = new Stock();

stock.CurrentPrice = 123.45M;
Console.WriteLine(stock.CurrentPrice);        // 123.45

var stock2 = new Stock { CurrentPrice = 83.12M };
Console.WriteLine(stock2.CurrentPrice);       // 83.12

public class Stock
{
  decimal currentPrice;           // Atributo privado

  public decimal CurrentPrice     // Propriedade pública
  {
    get { return currentPrice; } set { currentPrice = value; }
  }
}
```

Propriedades são úteis quando precisamos computar algum valor antes de retornar. Neste caso, *Worth* é uma propriedade calculada, não implementando formas de definir valores via *set*.

```
var stock = new Stock { CurrentPrice = 50, SharesOwned = 100 };

Console.WriteLine(stock.Worth);           // 5000

public class Stock
{
  decimal currentPrice;            // Atributo privado
  public decimal CurrentPrice      // Propriedade pública
  {
    get { return currentPrice; } set { currentPrice = value; }
  }

  decimal sharesOwned;             // Atributo privado
  public decimal SharesOwned       // Propriedade pública
  {
    get { return sharesOwned; } set { sharesOwned = value; }
  }

  public decimal Worth           // Propriedade calculada
  {
    get { return currentPrice * sharesOwned; }
  }
}
```

Propriedades também podem ser escritas sem especificar o corpo, ou seja, utilizamos uma sintaxe mais enxuta:

```
var stock = new Stock { CurrentPrice = 50, SharesOwned = 100 };

Console.WriteLine(stock.Worth);           // 5000

public class Stock
{
  decimal currentPrice;
  public decimal CurrentPrice
  {
    get { return currentPrice; } set { currentPrice = value; }
  }

  decimal sharesOwned;
  public decimal SharesOwned
  {
    get { return sharesOwned; } set { sharesOwned = value; }
  }

  public decimal Worth => currentPrice * sharesOwned;    // Propriedade sem corpo

  // A partir do C# 7, podemos também escrever get e set sem especificar o corpo
  public decimal Worth2
  {
    get => currentPrice * sharesOwned;
    set => sharesOwned = value / currentPrice;
  }

}
```

C# permite o uso de propriedades automáticas, sem a necessidade de declarar variáveis para armazenar os valores. Apenas declaramos os operadores de *get* e *set*.

```
var stock = new Stock { CurrentPrice = 50, SharesOwned = 100 };

Console.WriteLine(stock.Worth);           // 5000

public class Stock
{
  public decimal CurrentPrice { get; set; }   // Propriedade automática
  public decimal SharesOwned { get; set; }    // Propriedade automática

  public decimal Worth
  {
    get { return CurrentPrice * SharesOwned; }
  }
}
```

A inicialização de propriedades é realizada através da atribuição de valores com o operador de igualdade:

```
var stock = new Stock();

Console.WriteLine(stock.CurrentPrice);  // 123
Console.WriteLine(stock.Maximum);       // 999

public class Stock
{
  public decimal CurrentPrice { get; set; } = 123;   // Inicialização da propriedade
  public int Maximum { get; } = 999;                 // Inicialização da propriedade
}
```

Assim como métodos, propriedades podem ter seu acesso definido pelos modificadores *public* e *private*.

```
new Foo { X = 5 };    // Não irá compilar - X tem um modificador privado

public class Foo
{
  private decimal x;
  public decimal X
  {
    get         { return x;  }
    private set { x = Math.Round (value, 2); }    // set é um operador privado, não se
ndo possível definir seu valor fora do escopo da classe
  }

  public int Auto { get; private set; }  // Propriedade automática
}
```

Classes permitem o uso de indexadores(*Indexers*), assim como strings. A declaração do índice é feito por meio do operador de referência *this*. Veja o exemplo:

```
Sentence s = new Sentence();
Console.WriteLine(s[3]);       // fox

s[3] = "kangaroo";
Console.WriteLine(s[3]);       // kangaroo

// Faixas de indices também são permitidos
Console.WriteLine(s[^1]);               // fox
string[] firstTwoWords = s[..2];        // (The, quick)

class Sentence
{
  string[] words = "The quick brown fox".Split();     // Separa a string em um vetor d
e palavras

  public string this[int wordNum]      // Declaração do índice
  {
    get { return words[wordNum];  }
    set { words[wordNum] = value; }
  }

  public string this[Index index] => words[index];
  public string[] this[Range range] => words[range];

}
```

Um construtor estático(*static*) executa apenas uma vez por tipo, ao invés de uma vez por objeto.

```
new Test();
new Test();
new Test();      // Apenas um objeto é instanciado

class Test
{
  static Test()
  {
    Console.WriteLine("Type Initialized");
  }
}
```

Atributos estáticos inicializam logo antes do construtor ser chamado.

```
Console.WriteLine(Foo.X);  // 0
Console.WriteLine(Foo.Y);  // 3

class Foo
{
  public static int X = Y;
  public static int Y = 3;
}
```

Tipos parciais permitem que a definição de um tipo seja dividida em múltiplos arquivos.

```
new PaymentForm{ X = 3, Y = 4 };

partial class PaymentForm { public int X; }
partial class PaymentForm { public int Y; }
```

Um tipo parcial pode conter métodos parciais, por exemplo:

```
var paymentForm = new PaymentForm (50);

partial class PaymentForm
{
  public PaymentForm(decimal amount)
  {
    ValidatePayment(amount);
  }

  partial void ValidatePayment(decimal amount);
}

partial class PaymentForm
{
  partial void ValidatePayment(decimal amount)
  {
    if (amount < 100)
      throw new ArgumentOutOfRangeException("amount", "Amount too low!");
  }
}
```

O operador *nameof* permite obter o nome de variáveis:

```
int count = 123;
Console.WriteLine(nameof(count));     // count
```

Até este momento aprendemos a construir programas com instruções procedurais de alto nível (*top level*), ou seja, sem definir explicitamente um ponto de entrada para o programa. O que acontece é que, a partir do C# 10.0, quando não se define explicitamente um ponto de entrada para o programa, o compilador cria isso automaticamente. Veremos agora como declarar programas de acordo com o padrão de orientação a objetos, ou seja, definindo explicitamente um ponto de entrada. Para isso, devemos criar, em uma de nossas classes, explicitamente um método estático com o nome *Main*, o qual serve como ponto de entrada para a execução. Instruções procedurais somente serão permitidas dentro de métodos. Quando precisamos usar recursos implementados em outras classes, usamos a palavra-chave *using* para importar novos pacotes/classes. Observe que a construção a seguir é a que deve ser utilizada na grande maioria dos programas, pois define explicitamente o ponto de entrada do programa. Programas com instruções *top level* devem ser utilizadas somente em scripts simples. Qualquer programa mais complexo, deve utilizar o formato a seguir:

```
using System;

class Program
{
  static void Main()
  {
    Console.WriteLine(FeetToInches (30));      // 360
    Console.WriteLine(FeetToInches (100));     // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Para organizar nossas classes, temos a funcionalidade trazida pelos *namespaces*, que servem como pacotes. Namespaces podem ser aninhados, caso haja necessidade.

```
namespace Backend
{
  class Class1 {}
  class Class2 {}
}
```

Quando utilizamos o *using* no início de um arquivo, o que fazemos é nominar o namespace das classes que serão utilizadas pelo nosso programa. Dessa forma, todas as classes que precisemos devem estar devidamente declaradas por meio de uma diretiva do tipo *using*. Uma das funcionalidades introduzidas no C# 10, entretanto, foi a declaração automática de alguns namespaces mais utilizados, com o recurso "implicit usings", que encontra-se automaticamente habilitado no template *default* de projetos (veja o arquivo .csproj que foi gerado nos exemplos que você construiu para localizar essa configuração). Foi somente por esse motivo que não precisamos declarar o namespace *System* para poder usar a classe *Console*. Entretanto, é uma boa prática de engenharia de software sempre declarar todos os namespaces das classes utilizadas, para facilitar a um leitor do código que não tenha sido o autor do programa, uma melhor compreensão de onde estão localizadas as classes utilizadas, bem como facilitar a localização de sua documentação.

2) O Tipo *object*

O tipo *object* é a classe base para todos os outros tipos. Qualquer tipo de dado pode ser implicitamente convertido para objeto. Por exemplo, podemos usar esse conceito para escrever uma estrutura de dados de propósito geral.

```
Stack stack = new Stack();

stack.Push("sausage");             // Adição de dado tipo string
string s = (string)stack.Pop();    // Conversão explicita de object para string

Console.WriteLine(s);              // sausage

// Podemos adicionar tipos diferentes
stack.Push(3);                     // Adição de dado tipo int
int three = (int)stack.Pop();      // Conversão explicita de object para int

Console.WriteLine(three);          // 3

public class Stack
{
  int position;
  object[] data = new object[10];  // Criação de vetor do tipo objeto
  public void Push(object obj) { data[position++] = obj; }
  public object Pop() { return data[--position]; }
}
```

Boxing é a operação de converter um tipo de dado para o tipo mais ancestral, que é o tipo objeto. *Unboxing* é o processo inverso. Observe que quando usamos esse recurso, estamos utilizando a propriedade de polimorfismo da orientação a objetos.

```
int x = 9;
object obj = x;           // Box para int
int y = (int)obj;         // Unbox para int

Console.WriteLine(y);     // 9
```

Quando realizamos o *unboxing* dos dados, os tipos devem combinar, caso contrário uma exceção é lançada:

```
object obj = 9;           // 9 é inferido como tipo int
long x = (long) obj;      // InvalidCastException
```

Para consertar o exemplo acima, primeiro fizemos o *unbox* para o tipo int e então implicitamente convertemos para long:

```
object obj = 9;

// Unbox para int, com conversão implícita para long
long x = (int) obj;
Console.WriteLine(x);     // 9

// Também pode ser escrito assim:
object obj2 = 3.5;              // 3.5 é inferido como sendo do tipo double
int y = (int)(double) obj2;    // x agora vale 3
Console.WriteLine(y);     // 3
```

Existem duas formas básicas de obter o tipo de um objeto: a) Chamar o método *GetType* do objeto, b) Usar o operador *typeof* no tipo do objeto.

```
Point p = new Point();

Console.WriteLine(p.GetType().Name);             // Point
Console.WriteLine(typeof(Point).Name);           // Point
Console.WriteLine(p.GetType() == typeof(Point)); // True
Console.WriteLine(p.X.GetType().Name);           // Int32
Console.WriteLine(p.Y.GetType().FullName);       // System.Int32

public class Point { public int X, Y; }
```

Cada objeto implementa o método *ToString*, que permite o retorno de uma representação textual sobre o objeto. Podemos sobrescrever o método *ToString* para definir nossa própria representação.

```
int x = 1;
string s = x.ToString();             // s vale "1"

Panda p = new Panda { Name = "Petey" };
Console.WriteLine(p.ToString());     // Petey

public class Panda
{
  public string Name;
  public override string ToString() { return Name; }  // Sobrescrita do método ToStrin
g
}
```

A sobrescrita de um método deve ser feita com o uso da palavra-chave *override*.

3) Herança

Quando precisamos alterar o comportamento de uma classe já existente, usamos o conceito de herança. A alteração de comportamento pode implicar sobrescrever métodos para modificar o comportamento, como também adicionar novas funcionalidades. Em orientação a objetos, herança é implementada através da criação de uma classe filha que herda o comportamento de uma classe mãe, por exemplo:

```
Stock msft = new Stock { Name = "MSFT", SharesOwned = 1000 };

Console.WriteLine(msft.Name);         // MSFT
Console.WriteLine(msft.SharesOwned);  // 1000

House mansion = new House { Name = "Mansion", Mortgage = 250000 };

Console.WriteLine(mansion.Name);      // Mansion
Console.WriteLine(mansion.Mortgage);  // 250000

public class Asset           // Classe mãe
{
  public string Name;
}

public class Stock : Asset   // Classe filha que herda de Asset
{
  public long SharesOwned;
}

public class House : Asset   // Classe filha que herda de Asset
{
  public decimal Mortgage;
}
```

Note que, no exemplo acima, a propriedade *Name* é implementada somente na classe mãe, enquanto que as classes filhas *Stock* e *House* implementam novos atributos particularmente úteis para sua própria lógica. O acesso a propriedades da classes mãe é feita de modo transparente pelas classes filhas, evitando assim a repetição de código.

O uso de herança nos habilita a usar o conceito de polimorfismo. Com polimorfismo podemos escrever operações/métodos para a classe mãe e mesmo assim usá-los em classes filhas. Ou seja, a classe filha também é a classe mãe, apenas mais especializada.

```
Display(new Stock { Name="MSFT", SharesOwned=1000 });     // MSFT
Display(new House { Name="Mansion", Mortgage=100000 });   // Mansion

void Display(Asset asset)   // Uso de polimorfismo, isto é, tanto Stock como House são
Assets
{
  Console.WriteLine(asset.Name);
}

public class Asset           // Classe mãe
{
  public string Name;
}

public class Stock : Asset   // Classe filha que herda de Asset
{
  public long SharesOwned;
}

public class House : Asset   // Classe filha que herda de Asset
{
  public decimal Mortgage;
}
```

Podemos usar de conversões de referência quando trabalhamos com herança. Neste caso, uma operação de *upcast* converte um tipo filho em seu tipo mãe. Por exemplo:

```
Stock msft = new Stock();     // Tipo filho
Asset a = msft;               // Upcast para o tipo mãe

// Após o upcast, as duas variáveis ainda referenciam o mesmo objeto
Console.WriteLine(a == msft);  // True

public class Asset           // Classe mãe
{
  public string Name;
}

public class Stock : Asset   // Classe filha que herda de Asset
{
  public long SharesOwned;
}

public class House : Asset   // Classe filha que herda de Asset
{
  public decimal Mortgage;
}
```

De modo similar, podemos fazer uma operação de *downcasting*, ou seja, converter de um tipo mãe para um tipo filho.

```
Stock msft = new Stock();
Asset a = msft;                      // Upcast para o tipo mãe

Stock s = (Stock)a;                  // Downcast para tipo filho com conversão explíci
ta

Console.WriteLine(s.SharesOwned);
Console.WriteLine(s == a);          // True
Console.WriteLine(s == msft);       // True

House h = new House();
Asset a2 = h;               // Upcast sempre funciona
Stock s2 = (Stock)a2;       // ERROR: Downcast falhou: a não é uma Stock

public class Asset
{
  public string Name;
}

public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
}
```

Podemos testar uma conversão de referência com o operador *is*. Por exemplo:

```
Asset a = new Asset();

if (a is Stock)                                  // Verifica se a é do tipo Stock
  Console.WriteLine(((Stock)a).SharesOwned);   // Faz uma conversão (unbox) para Stock
para posterior acesso a propriedades

public class Asset
{
  public string Name;
}

public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
}
```

Juntamente com o teste do tipo via operador *is*, podemos fazer um *unbox* e atribuir uma referência para uso posterior. Esta construção simplifica o uso e é especialmetne útil em operações de controle de fluxo.

```
Asset a = new Stock { SharesOwned = 3 };

// Verificamos o tipo, se for Stock, atribuimos a referência s2, e testamos para mais
 condições
if (a is Stock s2 && s2.SharesOwned > 100000)
  Console.WriteLine($"Wealthy: {s2.SharesOwned}");
else
  s2 = new Stock();

Console.WriteLine (s2.SharesOwned);  // 0 - s2 ainda está no escopo

public class Asset
{
  public string Name;
}

public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
}
```

O operador *as* é utilizado para realizar a conversão para um tipo mais especializado, sem lançar exceções caso a conversão não seja possível. Por exemplo:

```
Asset a = new Asset();
Stock s = a as Stock;       // s é nulo; nenhuma exceção é lançada

// O operador as é similar ao operador is:
// expression is type ? (type)expression : (type)null

public class Asset
{
  public string Name;
}

public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
}
```

Por padrão, o C# não permite que métodos sejam sobrescritos nas classes filhas. No entanto, podemos permitir tal alteração com a palavra-chave *virtual* na definição do método na classe mãe. Assim, a classe filha pode implementar uma especialização deste método. Para isso, a classe mãe deve definir o método como *virtual* e a classe filha deve sobrescrever este método com a palavra-chave *override*.

```
House mansion = new House { Name="McMansion", Mortgage=250000 };
Console.WriteLine (mansion.Liability);             // 250000

public class Asset                                 // Classe mãe
{
  public string Name;
  public virtual decimal Liability => 0;           // Método definido como Virtual
}

public class House : Asset                         // Classe filha que herda de Asset
{
  public decimal Mortgage;
  public override decimal Liability => Mortgage;   // Método sobrescrito na classe fil
ha
}

public class Stock : Asset                         // Classe filha que herda de Asset
{
  public long SharesOwned;
  // Não iremos sobrescrever aqui, pois a implementação na classe mãe já nos é suficie
nte
}
```

A partir do C# 9, nós podemos sobrescrever um método de tal modo que ele retorne um tipo mais especifico do que o virtualmente determinado pela classe mãe.

```
House mansion1 = new House { Name = "McMansion", Mortgage = 250000 };
House mansion2 = mansion1.Clone();

public class Asset
// Classe mãe
{
  public string Name;
  public virtual Asset Clone() => new Asset { Name = Name };
// Método Clone retorna Asset
}

public class House : Asset
// Classe filha
{
  public decimal Mortgage;

  // Podemos retornar um tipo mais especializado
  public override House Clone() => new House { Name = Name, Mortgage = Mortgage };
// Método Clone retorna House
}
```

Uma classe declarada como abstrata nunca pode ser instanciada. Apenas as classes filhas podem fornecer uma implementação concreta e serem instanciadas. Classes abstratas podem conter somente membros abstratos.

```
Stock s = new Stock { SharesOwned = 200, CurrentPrice = 123.45M };

Console.WriteLine(s.NetValue);               // 24690.00

public abstract class Asset                  // Classe mãe definida como abstrata
{
  public abstract decimal NetValue { get; }  // Método definido como abstract, portant
o, sem implementação
}

public class Stock : Asset                   // Classe filha que traz a implementação
 concreta da classe mãe
{
  public long SharesOwned;
  public decimal CurrentPrice;

  // Método é sobrescrito e deve ser implementado pela classe filha
  public override decimal NetValue => CurrentPrice * SharesOwned;
}
```

No entanto, um método sobrescrito pode selar (*sealed*) sua própria implementação para prevenir que ela seja sobrescrita por outras classes herdadas.

```
House mansion = new House { Name="McMansion", Mortgage=250000 };

Console.WriteLine(mansion.Liability);      // 250000

public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;    // Método virtual
}

public class House : Asset
{
  public decimal Mortgage;
  public sealed override decimal Liability => Mortgage;   // Método sobrescrito e sela
do
}

// Podemos também selar uma classe inteira, implicitamente selando todos os seus métod
os virtuais
public sealed class Stock : Asset { /* ... */ }
```

Uma classe filha deve declarar seus próprios construtores. Desse modo, ela pode chamar qualquer um dos construtores que a classe mãe implemente, através do operador *base*.

```
new Subclass (123);

public class Baseclass                      // Classe mãe
{
  public int X;
  public Baseclass() { }                    // Construtor da classe mãe
  public Baseclass(int x) { this.X = x; }   // Construtor da classe mãe
}

public class Subclass : Baseclass           // Classe filha
{
  public Subclass(int x) : base(x) { }      // Construtor da classe filha, note que o
 construtor da classe mãe foi chamado com o operador base
}
```

Se o construtor de uma classe filha omita a palavra-chave *base*, ainda assim o construtor da classe mãe é implicitamente chamado.

```
new Subclass();

public class BaseClass
{
  public int X;
  public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (X); }    // 1
}
```

4) Modificadores de Acesso

Encapsulamento de atributos e funções é um conceito fundamental e amplamente utilizado em programação orientada a objetos. Esses conceitos são implementados através de modificadores de acesso em classes, métodos e atributos. Com eles, podemos determinar como o comportamento de nossos objetos são afetados por operações internas ao objeto, externas, e em classes filhas. Assim, evitamos que o objeto externe dados, operações e responsabilidades que não deveriam ser expostas.

Os modificadores de acesso podem ser públicos (*public*), privados (*private*), protegidos (*protected*) e internos (*internal*). No modo *protected*, o elemento pode ser acessado somente pelo código da mesma classe, ou uma classe filha e no modo *internal*, o elemento pode ser acessado por qualquer código no mesmo *assembly*, mas não em outro *assembly*.

```
class Class1 {}                 // Class1 é internal (padrão) - visível para outros ti
pos dentro da mesma biblioteca
public class Class2 {}          // Class2 é visível para todos, incluindo para outros
 tipos e em outras bibliotecas

class ClassA
{
  int x;                        // x é privado (padrão) - não pode ser acessado por ou
tros tipos
}

class ClassB
{
  internal int x;               // x pode ser acessado de outros tipos dentro da mesma
biblioteca
}

class BaseClass
{
  void Foo()           {}       // Foo é privado (padrão)
  protected void Bar() {}       // Foo é acessível para classes filhas
}

class Subclass : BaseClass
{
    void Test1() { Foo(); }     // Erro - não pode acessar Foo
    void Test2() { Bar(); }     // OK
}
```

Quando um método é sobrescrito, sua acessibilidade deve ser idêntica ao método virtual. Uma classe filha pode ser menos acessível que sua classe mãe, porém o contrário não é permitido.

5) Interfaces

Uma interface é uma estrutura disponível em orientação a objetos, para padronizar o modo de comunicação com um objeto. Uma interface especifica um conjunto de métodos que caracterizam um modo de interação com um objeto que a implemente. Dizemos que um objeto implementa uma interface, quando sua classe possui todos os métodos definidos na interface. De uma maneira informal, uma interface se parece muito com uma classe abstrata que não possui *fields*. Ou seja, os métodos são somente declarados, e nenhum *field* é declarado. As interfaces são utilizadas, na prática, como classes abstratas, que demandam que uma outra classe implemente seus métodos. Foram criadas, como uma solução para o problema da herança múltipla, pois uma classe pode implementar diversas interfaces. O mecanismo de herança é semelhante para interfaces. Podemos usar o polimorfismo de objetos, transformando-o em uma instância de qualquer uma das interfaces que implementa. Veja o exemplo a seguir.

```
IEnumerator e = new Countdown();    // Uso de polimorfismo

while (e.MoveNext())
  Console.Write(e.Current);         // 109876543210

public interface IEnumerator        // Declaração de uma interface com métodos prototi
pados
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}

class Countdown : IEnumerator       // Implementação de interface e seus métodos por u
ma classe
{
  int count = 11;
  public bool MoveNext () => count-- > 0;
  public object Current   => count;
  public void Reset()     { throw new NotSupportedException(); }
}
```

A classe Countdown implementa a interface IEnumerator. Dessa forma, podemos criar uma instância da classe Countdown com o operador new, e atribuí-la a uma variável que é do tipo IEnumerator (interface). Com relação ao uso, o acesso a essa variável se dá como se IEnumerator fosse uma classe e "e" uma instância dela.

Diferentes classes poderiam implementar a mesma interface, de diferentes maneiras. Dessa forma, podemos executar seus métodos, sem precisar saber como os métodos são implementados na classe. Com o uso de interface, podemos apenas usar os métodos e propriedades como declarados na interface.

Interfaces podem ser estendidas, de modo análogo a classes. Por exemplo:

```
IRedoable r = null;
IUndoable u = r;

public interface IUndoable { void Undo(); }
public interface IRedoable : IUndoable      // Interface estendida
{
  void Redo();
}
```

Implementar múltiplas interfaces pode muitas vezes resultar em colisões de assinaturas. Podemos resolver essas colisões através da implementação explicita de cada interface.

```
Widget w = new Widget();
w.Foo();                        // Widget's implementation of I1.Foo
((I1)w).Foo();                  // Widget's implementation of I1.Foo
((I2)w).Foo();                  // Widget's implementation of I2.Foo

interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2    // Implementação de múltiplas interfaces
{
  public void Foo ()
  {
    Console.WriteLine("Widget's implementation of I1.Foo");
  }

  int I2.Foo()                  // Implementação explicita de método da interface
  {
    Console.WriteLine("Widget's implementation of I2.Foo");
    return 42;
  }
}
```

Métodos definidos em interfaces são implicitamente selados (*sealed*). Eles deve ser marcados como virtuais, para que possam ser alterados em classes filhas.

```
RichTextBox r = new RichTextBox();

r.Undo();                           // RichTextBox.Undo
((IUndoable)r).Undo();              // RichTextBox.Undo
((TextBox)r).Undo();                // RichTextBox.Undo

public interface IUndoable { void Undo(); }

public class TextBox : IUndoable     // Classe mãe que implementa interface
{
  public virtual void Undo() => Console.WriteLine("TextBox.Undo");      // Alteração p
ara virtual na classe mãe
}

public class RichTextBox : TextBox   // Classe filha
{
  public override void Undo() => Console.WriteLine("RichTextBox.Undo"); // Sobrescrita
na classe filha
}
```

Reimplementação de interfaces em classes filhas pode gerar problemas de conflito sobre qual implementação será chamada. Uma boa prática é fazer a conversão do objeto para o tipo que implementa a interface que se deseja chamar.

```
RichTextBox r = new RichTextBox();

r.Undo();                 // RichTextBox.Undo
((IUndoable)r).Undo();    // RichTextBox.Undo
((TextBox)r).Undo();      // TextBox.Undo

public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public void Undo() => Console.WriteLine("TextBox.Undo");
}

public class RichTextBox : TextBox, IUndoable       // Reimplementação de interface
{
  public new void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```

Interfaces podem apresentar uma implementação padrão. Nestes casos, classes que a implementem não precisam necessariamente definir uma implementação.

```
var logger = new Logger();

((ILogger)logger).Log("message");     // message

interface ILogger
{
  void Log (string text) => Console.WriteLine(text);    // Implementação de método
}

class Logger : ILogger
{
  // Não precisamos implementar o método da interface
}
```

Interfaces também permitem o uso de propriedades estáticas. Por exemplo:

```
ILogger.Prefix = "File log: ";       // Definição de propriedade estática

var logger = new Logger();
((ILogger)logger).Log("message");   // File log: message

interface ILogger
{
  void Log (string text) => Console.WriteLine(Prefix + text);

  static string Prefix = "";         // Propriedade estática
}

class Logger : ILogger
{
  // Não precisamos implementar o método da interface
}
```

6) Enums

Enum são tipos especiais de valores que permitem a especificação de um grupo de constantes numéricas nomeadas. Por exemplo:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);

Console.WriteLine(isTop);       // True

// Definição de um enum com quatro constantes
public enum BorderSide { Left, Right, Top, Bottom }

// Podemos especificar um tipo alternativo de dado
public enum BorderSideByte : byte { Left, Right, Top, Bottom }

// Também podemos especificar valores explícitos para cada constante
public enum BorderSideExplicit : byte { Left=1, Right=2, Top=10, Bottom=11 }

public enum BorderSidePartiallyExplicit : byte { Left=1, Right, Top=10, Bottom }
```

Podemos explicitamente converter instâncias de enums para seu tipo integral, e de volta para seu tipo constante.

```
int i = (int)BorderSide.Left;
Console.WriteLine(i);                   // 0

BorderSide side = (BorderSide)i;
Console.WriteLine(side);                // Left

bool leftOrRight = (int)side <= 2;
Console.WriteLine(leftOrRight);         // True

HorizontalAlignment h = (HorizontalAlignment)BorderSide.Right;
Console.WriteLine(h);                   // Right

BorderSide b = 0;
Console.WriteLine(b);                   // Left

public enum BorderSide { Left, Right, Top, Bottom }

public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

Podemos combinar várias enums. Para evitar ambiguidades, devemos definir um valor para cada constante, tipicamente em potência de 2.

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;   // Enum combinados via
operador OR binário

if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine("Includes Left");                           // Includes Left

Console.WriteLine(leftRight.ToString());                        // 3 = "Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine(s == leftRight);                              // True

s ^= BorderSides.Right;                                         // Remove Right
Console.WriteLine(s);                                           // Left

public enum BorderSides { None = 0, Left = 1, Right = 2, Top = 4, Bottom = 8 }
```

Por conveniência, podemos incluir a combinação de enums dentro da declaração.

```
Console.WriteLine(BorderSides.All);                                 // All

Console.WriteLine(BorderSides.All ^ BorderSides.LeftRight);     // TopBottom

public enum BorderSides
{
  None = 0,
  Left = 1, Right = 2, Top = 4, Bottom = 8,
  LeftRight = Left | Right,
  TopBottom = Top | Bottom,
  All = LeftRight | TopBottom
}
```

7) Estruturas (struct)

Uma estrutura (*struct*) é similar a uma classe que tenha somente fields, ou seja, não tenha nenhum método. Além disso, no C#, uma estrutura é um tipo por valor (*value type*) ao invés de um tipo por referência (*reference type*).

```
Point p1 = new Point();       // p1.x e p1.y serão 1
Point p2 = new Point(2, 2);   // p2.x e p2.y serão 2

Console.WriteLine(p1);
Console.WriteLine(p2);

struct Point                  // Declaração de uma struct
{
  public int x, y;
  public Point()             { this.x = 1; this.y = 1; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Do mesmo modo que com classes, estruturas podem ser instanciadas por meio do operador *new.* Nestes casos, o construtor é executado. Ou podem ser instanciadas com o construtor padrão (*default*).

```
Point p1 = new Point();       // p1.x e p1.y serão 1
Point p2 = default;           // p2.x e p2.y serão 0 - Uso do construtor padrão
var points = new Point[10];   // Cada ponto no vetor terá o valor (0,0)

struct Point
{
  public int x = 1;
  public int y;
  public Point() => y = 1;
}
```

8) Tipos Aninhados

Um tipo aninhado é declarado dentro do escopo de outro tipo, por exemplo:

```
public class TopLevel
{
  public class Nested { }               // Classe aninhada
  public enum Color { Red, Blue, Tan }  // Enum aninhado
}

static void Main()
{
  TopLevel.Color color = TopLevel.Color.Red;
}
```

Tipos aninhados também devem respeitar os modificadores de acesso, como *private*.

```
public class TopLevel
{
  static int x;
  public class Nested
  {
    public static void Foo() { Console.WriteLine (TopLevel.x); }
  }
}

static void Main()
{
  TopLevel.Nested.Foo();
}
```

E *protected*:

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}

static void Main()
{
}
```

Projeto Final - Parte 1

Desenvolva o minigame chamado **Jewel Collector**. O objetivo desse jogo é que um robô, controlado pelo teclado, se desloque por um mapa 2D de modo a desviar dos obstáculos e coletar todas as joias. Para isso, as seguintes classes devem ser criadas:

- **Jewel.cs** - A classe Jewel deverá armazenar as informações da joia, como a posição (x, y) no mapa e o tipo, que poderá ser **Red**, no valor de 100 pontos; **Green**, no valor de 50 pontos; e **Blue**, no valor de 10 pontos.

- **Obstacle.cs** - A classe Obstacle deverá armazenar as informações do obstáculo, que será a posição (x, y) e o tipo. Cada obstáculo deverá possuir um tipo, que poderá ser **Water** ou **Tree**.
- **Robot.cs** - A classe Robot deverá ser responsável por armazenar as informações do robô, que será a posição (x, y) e uma sacola (**bag**), em que o robô colocará as joias coletadas no mapa. Além disso, a classe Robot deverá implementar os métodos para que o robô possa interagir com o mapa, isto é, deslocar-se nas quatro direções e coletar as joias. Além disso, implemente um método para imprimir na tela o total de joias armazenadas na sacola e o valor total.
- **Map.cs** - A classe Map deverá armazenar as informações do mapa 2D e implementar métodos para adição e remoção de joias e obstáculos. Além de um método para imprimir o mapa na tela. A impressão do mapa deverá seguir a seguinte regra: Robo será impresso como **ME**; Joias Red, como **JR**; Joias Green, como **JG**; Joias Blue, como **JB**; Obstáculos do tipo Tree, como **$$**; Obstáculos do tipo Water, como **##**; Espaços vazios, como **--**.
- **JewelCollector.cs** - A classe JewelCollector deverá ser responsável por implementar o método **Main()**, criar o mapa, inserir as joias, obstáculos, instanciar o robô e ler os comandos do teclado. Para que o usuário possa controlar o robô, os seguintes comandos deverão ser passados através das teclas **w**, **s**, **a**, **d**, **g**. Sendo que a tecla **w** desloca o robô para o norte, a tecla **s** desloca para o sul, a tecla **a** desloca para oeste e a tecla **d** para leste. Para coletar uma joia, use a tecla **g**.

Uma joia somente poderá ser coletada se o robô estiver em uma das posições adjacentes a ela. Todos os obstáculos são intransponíveis. Para cada comando executado pelo usuário, imprima o estado atual do mapa, bem como o estado da sacola do robô.

Observação: Caso julgue necessário para uma melhor estruturação do código, é permitido criar classes adicionais.

Fluxo de Execução

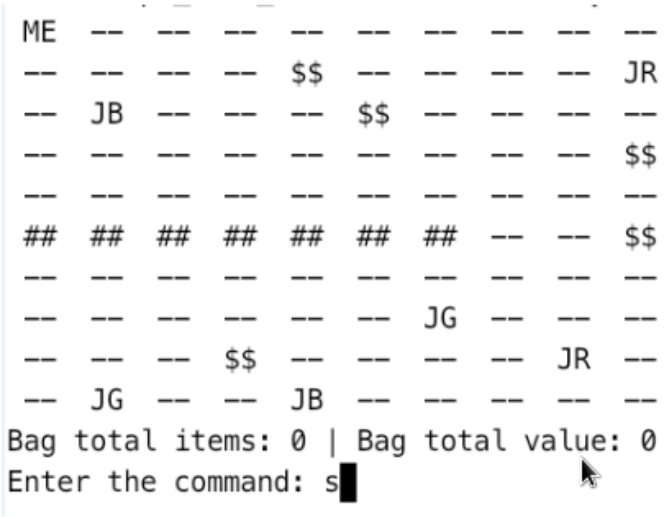
- Crie um mapa com dimensão 10x10
- Crie e insira as joias de acordo com o tipo e posição (x, y) abaixo:
  - Red - (1, 9)
  - Red - (8, 8)
  - Green - (9, 1)
  - Green - (7, 6)
  - Blue - (3, 4)
  - Blue - (2, 1)
- Crie e insira os obstáculos de acordo com o tipo e posição (x, y) abaixo:
  - Water - (5, 0)
  - Water - (5, 1)
  - Water - (5, 2)
  - Water - (5, 3)
  - Water - (5, 4)
  - Water - (5, 5)
  - Water - (5, 6)
  - Tree - (5, 9)
  - Tree - (3, 9)
  - Tree - (8, 3)
  - Tree - (2, 5)
  - Tree - (1, 4)
- Crie o robô na posição (x, y) = (0, 0).
- Inicie o jogo, isto é, leia o teclado e colete todas as joias e desvie dos obstáculos interativamente.

Para inicio do trabalho e leitura do teclado, poderá ser utilizado como base o código abaixo:

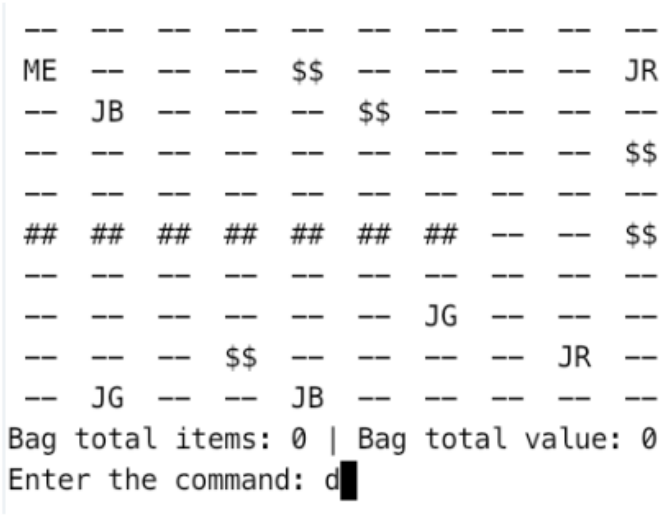
```
public class JewelCollector {

  public static void Main() {

      bool running = true;

      do {

          Console.WriteLine("Enter the command: ");
          string command = Console.ReadLine();

          if (command.Equals("quit")) {
              running = false;
          } else if (command.Equals("w")) {

          } else if (command.Equals("a")) {

          } else if (command.Equals("s")) {

          } else if (command.Equals("d")) {

          } else if (command.Equals("g")) {

          }
      } while (running);
  }
}
```

Exemplo de Execução

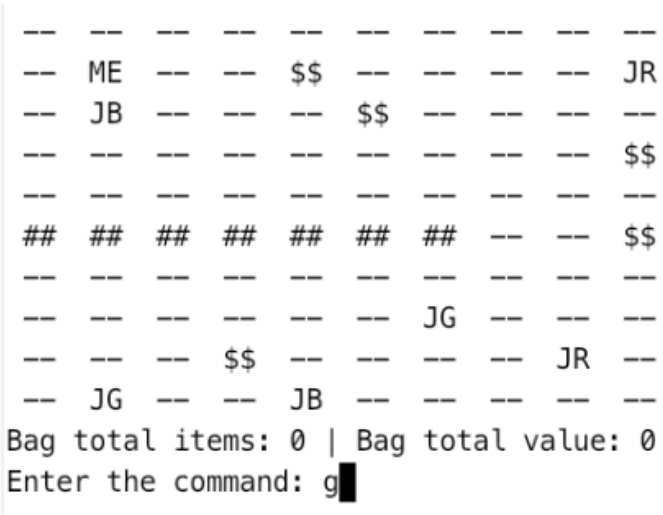
Um exemplo de execução é mostrado abaixo:



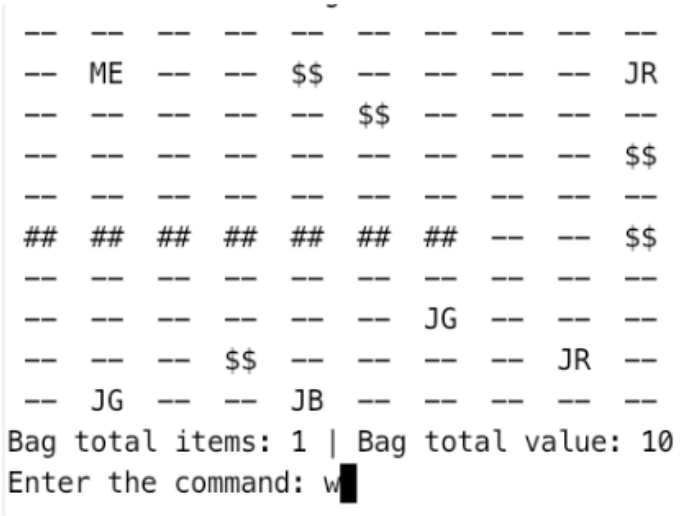
(a) Comando Move South



(b) Comando Move East



(c) Comando Get Jewel



(d) Comando Move North

Critérios de Avaliação

Em particular, o código será avaliado de acordo com os seguintes aspectos:

- Aplicação de princípios de Programação Orientada a Objetos (POO) em C# (60%).
- Funcionalidades implementadas (40%).

Entrega

O Trabalho Final será composto de 2 partes e será entregue ao final da disciplina de C#. A parte 2 será disponibilizada na próxima aula, por isso comecem desde agora a implementar a parte 1. **Bom trabalho!**

Última atualização: quinta, 1 Set 2022, 17:41

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

Aula Teórica 2

Aula Prática 2

Atividade 3: 10/09/2022 – 08:30-12:30

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
