Informática para internet

Fundamentos da programação orientada a objetos e sintaxe da linguagem

No material **Linguagem de *scripts***, da UC (unidade curricular) **Codificar aplicações *web***, exploramos a linguagem C# e começamos aprofundar a orientação a objetos. Neste conteúdo, vamos aprofundar ainda mais nossos conhecimentos sobre as funcionalidades e as peculiaridades deste paradigma de programação.

Herança

Herança é um dos pilares da orientação a objeto que permite definir uma classe filha que reutiliza (herda), estende ou modifica o comportamento de uma classe pai. Nesse sentido, a herança permite derivar uma nova classe mais especializada a partir de uma classe mais genérica.

Este tipo de recurso é utilizado para descrever a relação entre uma classe e outra, permitindo reutilizar o código. A classe cujos membros são herdados é chamada de classe base (superclasse); a classe que herda os membros da classe base, por sua vez, é chamada de classe derivada (subclasse).

Representação de herança utilizando UML:

Para entender melhor, vamos criar um exemplo utilizando ninjas. Para isso, pense no que os ninjas têm em comum.

Vamos definir que todos eles são lutadores com valores e ações. Um ninja precisa ter nome, vida, valores de ataque e de defesa. Além disso, ele deve ser capaz de correr, socar, chutar, pular e defender. Levando isso para um ambiente computacional, teríamos os atributos: nome, vida, ataque, defesa; e os métodos: correr, socar, chutar, pular, defender.

Entretanto, esses ninjas têm características próprias que os outros ninjas não devem ter em seus códigos. Logo, vamos definir que eles têm métodos (ações) específicos, como: lançar gelo, lançar fumaça, lançar ácido etc.

Vamos definir que a classe pai é a classe **Ninja**, que vai ter os atributos comuns de todos os ninjas. Também teremos as classes filhas, que serão **NinjaGelo**, **NinjaFumaca** e **NinjaAcido**, herdando tudo que a classe **Ninja** contém. Além disso, vamos adicionar ações específicas: a classe **NinjaGelo** vai lançar gelo, a **NinjaFumaca** vai lançar bomba de fumaça e **NinjaAcido** pode lançar ácido.

Nas classes filhas, é possível adicionar novos atributos e métodos. Entretanto, é importante notar que os métodos construtores não são herdados.

Para relembrar, construtor é um método que tem o mesmo nome da classe, que não retorna nenhum valor e que é chamado cada vez que um objeto da classe é criado.

A classe **Ninja** que imaginamos fica da seguinte forma:

```
class Ninja
{
  public int vida, forca, velocidade;
  public void Socar()
  {
    Console.WriteLine("Socando!");
  }
  public void Chutar()
  {
    Console.WriteLine("Chutando!");
  }
}
```

Quando criamos uma classe, para fazer a indicação de que esta classe herda a outra, colocamos o nome dela e, logo em seguida, dois pontos (:) e o nome da classe pai.

Lembrando que C# é uma linguagem *case sensitive*, por isso é muito importante tomar cuidado com a grafia da classe pai. Se o nome escrito na criação da classe filha não for idêntico ao nome da classe pai, não será possível fazer a herança.

A classe **NinjaGelo**, que herda a classe **Ninja**, fica da seguinte forma:

```
class NinjaGelo : Ninja //Classe NinjaGelo herda Ninja
{
  public void Gelo()
  {
    Console.WriteLine("Lançando gelo!");
  }
}
```

No *script* do *main* poderíamos instanciar um **NinjaGelo** desta maneira:

```
NinjaGelo ninjaGelo = new NinjaGelo ();
ninjaGelo.vida = 10; //Origem: classe base (ninja)
ninjaGelo.forca = 8; //Origem: classe base (ninja)
ninjaGelo.velocidade = 9; //Origem: classe base (ninja)
ninjaGelo.Socar(); //Origem: classe base (ninja)
ninjaGelo.Chutar(); //Origem: classe base (ninja)
ninjaGelo.Gelo(); // Origem: classe NinjaGelo
```

Se você se sentir confiante, vamos ao nosso primeiro desafio!

Crie três classes de ninjas. Para isso, defina como classe base a Ninja e determine atributos e métodos específicos de cada classe, por exemplo, no NinjaGelo, o poder de gelo é um método específico, assim como tempo de congelamento é um atributo específico.

Classes abstratas

Uma classe concreta é uma classe que tem atributos, métodos construtores e outros e pode ser instanciada, ou seja, permite a criação de objetos a partir dela. Classes concretas podem ser herdadas por outras classes.

As classes abstratas, por outro lado, servem para serem modeladas para suas classes derivadas e não podem ser instanciadas sozinhas. Uma maneira fácil de entender esse tipo de classe é pensar que ela é um rascunho de como as classes herdadas devem se comportar.

As classes derivadas, em geral, poderão sobrescrever os métodos para realizar sua implementação de maneira mais personalizada. Classes abstratas, por usa vez, são as que não permitem realizar qualquer

tipo de instância, pois são classes feitas especialmente para servirem de modelos para suas classes derivadas. Caso um ou mais métodos abstratos estejam presentes nessa classe abstrata, a classe filha estará obrigada a definir tais métodos para não se tornar uma abstrata também.

A funcionalidade dos métodos abstratos que são herdados pelas classes filhas é atribuída de acordo com o propósito dessas classes. Se não queremos atribuir uma funcionalidade para algum método abstrato, será preciso declarar os métodos (mesmo que vazios).

Abstract representa mais uma ideia do que um método concreto, como é o caso dos métodos comuns ou dos métodos virtuais.

```
public abstract class Veiculo
{

}
```

A classe *abstract* não é uma estrutura em si, por isso ela deve ser implementada.

```
Veiculo veiculo = new Veiculo(); //Não funciona, por ser abstrato
class Tanque : Veiculo
{

}
```

Uma classe não abstrata pode herdar uma classe abstrata, mas pode declarar métodos e atributos normalmente.

```
public abstract void Frear();
```

Métodos abstratos obrigatoriamente devem estar em classes abstratas. Estes métodos não têm escopo, apenas uma declaração. Nas classes que herdarem a classe abstrata, os métodos abstratos deverão ser obrigatoriamente implementados.

```
public abstract class Veiculo
{
  protected int resistencia, velocidade;

  public void Mover()
  {
    Console.WriteLine("movendo");
  }
  public abstract void Frear(); //método abstrato aqui
}
class Tanque : Veiculo
{
  public override void Frear()
  {
    Console.WriteLine("Parando o carro!");
  }
}
```

Se o método **Frear** não for implementado, o projeto não é compilado.

A palavra “veículo” representa apenas um conceito, uma forma generalizada de pensarmos nos objetos que dividem essa categoria. Por mais que veículo não exista, o objeto tanque – ou outro tipo de veículo – existe e depende do conceito de veículo.

Interface

Interface é uma estrutura que permite ao desenvolvedor especificar todos os métodos e as propriedades que ele deseja que sejam disponibilizadas pelas classes que a implementam.

Com uma interface é possível separar completamente a definição/assinatura de métodos de suas respectivas implementações, o que facilita muito a distribuição de tarefa entre os programadores, que trabalham em cima dos mesmos códigos.

Uma analogia interessante é pensar na interface como uma obra. A classe é como o pedreiro, que garante que vai construir (implementar) a casa (interface), desde que o arquiteto (interface) diga como é a estrutura da casa. Caso alguma das partes não cumpra com o combinado, a casa desaba (ou seja, um erro acontece).

A sintaxe da utilização de uma interface é colocar dois pontos após o nome da classe concreta que vai implementar a interface seguido do nome da interface.

```
Conta : IConta  //Conta é classe concreta e IConta a interface
```

Para definir uma interface, usamos a palavra-chave “interface”, conforme mostra o exemplo a seguir, em que definimos duas interfaces: **IAlimentacao** e **IRepousar**.

```
interface IAlimentacao
{
void Comer();
}
interface IRepousar
{
void Dormir();
}
```

Por convenção, ao declarar uma interface, usamos a letra **I** como letra inicial de seu nome.

Quando definimos um método dentro de uma interface, não são utilizados os modificadores de acesso **public**, **private**, ou **protected**, pois todos os elementos em uma interface são públicos. O corpo do método é substituído por ponto e vírgula, o que também ocorre nos métodos abstratos.

A implementação das interfaces definidas pode ser feita pela classe concreta **ClasseConcreta** da seguinte forma:

```
class ClasseConcreta : IAlimentacao, IRepousar
{
  public void Comer()
  {

  int numeroGarfadas;
  int numeroRepticoes;
  int resultado;

  Console.WriteLine("Escreva o número de garfadas");

  numeroGarfadas = int.Parse(Console.ReadLine());

  Console.WriteLine("Escreva o número de vezes que você repetiu a comida"
);

  numeroRepticoes = int.Parse(Console.ReadLine());

  resultado = numeroGarfadas * numeroRepticoes;

  Console.WriteLine("Você deu no total " + resultado + " garfadas.");

  }

  public void Dormir()
  {
    int horasDormidas;
    Console.WriteLine("Escreva o número de horas dormidas");
    horasDormidas = int.Parse(Console.ReadLine());

    if(horasDormidas < 4)
    {
      Console.WriteLine("Você dormiu muito pouco");
    }

    else if(horasDormidas >= 4 && horasDormidas < 7)
    {
      Console.WriteLine("Você dormiu pouco");
    }

    else if(horasDormidas >= 7 && horasDormidas < 10)
    {
      Console.WriteLine("Você dormiu muito bem");
    }

    else if(horasDormidas > 10)
    {
      Console.WriteLine("Você está em coma?");
```

```
    }
  }
}
```

A classes concreta implementa os métodos **Comer()** e **Dormir()**, definidos nas interfaces. Uma interface é como uma classe base abstrata que contém apenas membros abstratos. Qualquer classe ou *struct* que implementa a interface deve implementar todos os seus membros (caso isso não ocorra, o projeto gera erros).

O desenvolvedor não tem permissão para definir campos em uma interface, nem mesmo campos estáticos, pois campo é um detalhe intrínseco de implementação de classe ou estrutura. As interfaces podem conter propriedades, indexadores, métodos e eventos.

Interfaces podem herdar apenas outras interfaces, ou seja, uma interface não pode ser instanciada diretamente. Nesse sentido, seus membros são implementados por qualquer classe ou *struct* que implemente a interface.

Não confunda interface com interface do usuário. Em programação, interface não é um recurso visual, mas uma ferramenta de linguagens orientadas a objeto.

A interface possibilita separar o “o que” do “como”. A interface não se preocupa com a forma com a qual o método está sendo implementado, mas, sim, garante que este método esteja disponível a todos os objetos que a implementarem.

Por causa dessa característica, é possível criar sistemas flexíveis a mudanças. Isto é, ocorre porque, quando usamos interfaces, se precisar, podemos mudar a implementação para adaptar a aplicação a mudanças.

Modificadores de acesso

A herança proporciona um constante reúso de partes do nosso código. Novas classes são criadas a partir de outras já existentes, absorvendo atributos e comportamentos e adicionando os seus próprios.

Entretanto, nem todos os membros da classe base são obrigatoriamente acessíveis na classe derivada. Isto dependerá dos atributos de acesso do membro.

Vamos relembrar os atributos de acesso:

Public

Acesso permitido de qualquer classe em qualquer lugar.

private

Sem acesso de fora da classe.

protected

Acessível a partir de todas as classes no mesmo pacote e a partir de qualquer subclasse em qualquer lugar.

É usual restringir o acesso às informações entre classes. Entretanto, se usarmos o modificador *private* , nem mesmo as classes filhas podem acessar as informações da classe pai. Para não entrar em uma situação delicada, em vez disso, utilizamos o modificador *protected* .

Quando um atributo é *protected* , ele é visível para a classe que o contém, pode ser herdado, mas não pode ser acessado por uma instância da mesma classe, apenas por instâncias de classes que o herdam, como é possível ver no exemplo:

```
class Animais
{
  protected string nome = "Sem nome";
}

class Gatos : Animais
{
  void criarGatos()
  {
    Animais umAnimal = new Animais();

    umAnimal.nome = "Mio";
    //programa retorna um erro
  }
}
```

Para o programa funcionar, devemos acessar o método **nome** a partir de uma instância de **Gatos**, e não de **Animais**.

```
class Animais
{
  protected string nome = "Sem nome";
}

class Gatos : Animais
{
  void criarGatos()
  {
    Gatos umGato = new Gatos();

    umGato.nome = "Lunes";
    //o programa funciona normalmente
  }
}
```

Quando você está projetando classes com comportamento abstrato e fortemente reusáveis, esse modificador permite que você deixe os métodos que precisar disponíveis para quem vai herdar a classe e complementar o

comportamento dela.

Vamos ao nosso próximo desafio!

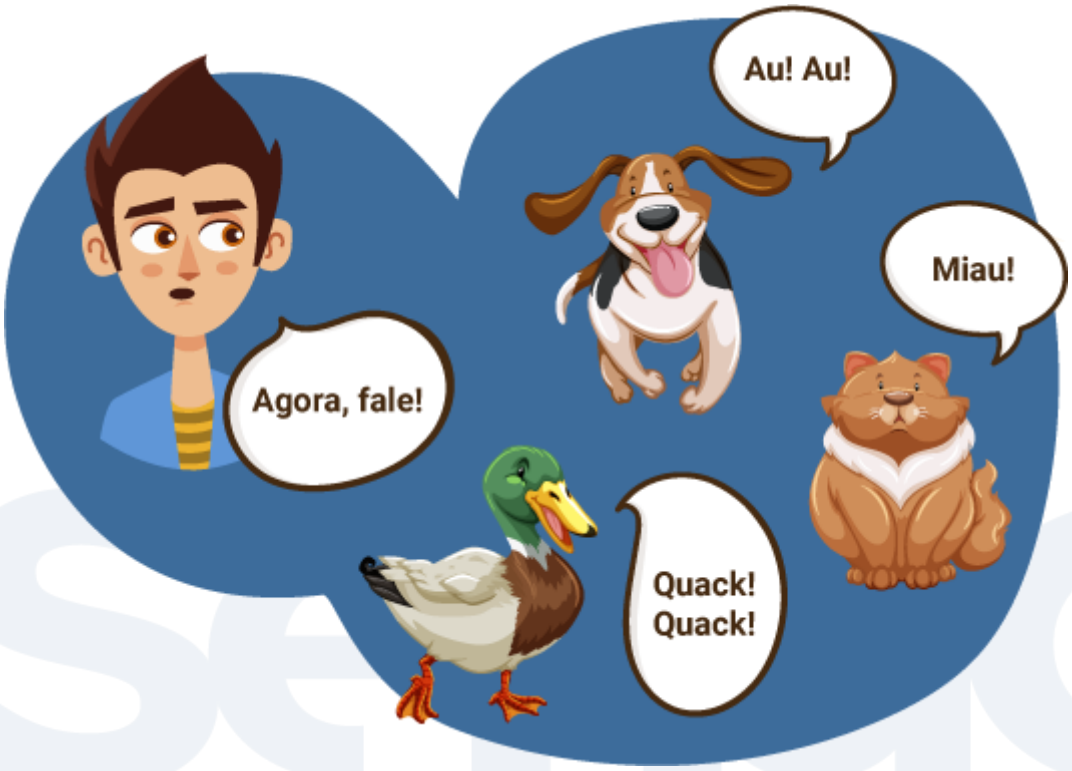
Para esse desafio, vamos definir três variáveis protected na classe Ninja. Também vamos definir essas três variáveis no construtor das classes filhas (por exemplo: classe NinjaGelo).

Polimorfismo

Polimorfismo significa muitas formas. Na orientação a objetos, você pode enviar o mesmo sinal para objetos distintos e fazê-los responder de maneiras diferentes.

Você pode enviar a mensagem “falar” para cada objeto que seja um animal, e cada um vai se comportar de maneira distinta para conseguir atender ao solicitado, pois eles são diferentes entre si e apresentam comportamentos distintos.

Quando uma mesma mensagem pode ser processada de diferentes formas, temos um polimorfismo.



Homem: Agora, fale!

Pato: Quack! Quack!

Cachorro: Au! Au!

Gato: Miau!

Polimorfismo é o conceito de que duas ou mais classes derivadas de uma mesma classe pai podem invocar métodos que têm a mesma identificação (assinatura, no exemplo: falar), que, no entanto, apresentam comportamentos completamente distintos, especializadas para cada classe derivada. Sendo assim, temos diferentes implementações de métodos que têm com o mesmo nome.

Existem dois tipos básicos de polimorfismo:

Polimorfismo em tempo de compilação (*overloading*/sobrecarga)

Tempo de compilação se refere ao que vem antes do processo de compilação. Isto é, o período necessário para o programa interpretar todas as informações e agrupá-las, de forma que a aplicação possa ser executada. Este é o momento em que costumam ocorrer erros de sintática, semânticos, de tipagem etc.

Polimorfismo em tempo de execução (*overriding*/sobrescrita)

Tempo de execução é tudo o que ocorre quando o código já está executando. É o tempo que o programa leva para receber alguma resposta específica. Em função de o código estar rodando, os dados inseridos direta ou indiretamente muitas vezes podem gerar algum tipo de erro ou quebra no código. Erros costumam paralisar a execução

Com a sobrecarga (*overload*) é possível definir diversas características, métodos ou procedimentos em uma classe com o mesmo nome, mas parâmetros diferentes.

A palavra-chave *override* é utilizada para modificar uma propriedade, um método virtual/abstrato, evento da classe base na classe derivada.

Polimorfismo é a característica única de linguagens orientadas a objetos que permite que diferentes objetos respondam a mesma mensagem, mas cada um a seu modo.

Em termos de programação, polimorfismo representa a capacidade de uma única referência invocar métodos diferentes, dependendo do seu conteúdo.

Devemos ter em mente que ambas as palavras-chaves são complementares. É necessário ter em mente que a propagação da palavra-chave virtual ocorre para descendentes. Um método virtual pode ser sobrescrito em descendentes e, ainda, em uma classe derivada.

Considere novamente a classe **Ninja**. Teremos nela um método **Update()**, que será chamado para realizar as ações do personagem em tempo de execução.

Métodos virtuais

```
public virtual void Update()
{
}
```

Métodos virtuais existentes em classes bases serão herdados nas classes filhas e o programador escolhe se quer substituí-los.

```
class Ninja
{
    protected int vida, forca, velocidade;
    protected void Socar()
    {
        Console.WriteLine("Socando!");
    }
    protected void Chutar()
    {
        Console.WriteLine("Chutando!");
    }
```

```
    public virtual void Update()
    {
        Console.WriteLine("=== UPDATE NINJA ===");
        Socar();
        Chutar();
    }
}
```

Utilizando o *override* em métodos virtuais nas classes filhas, os métodos das classes base são substituídos.

```
class NinjaGelo: Ninja
{
    public NinjaGelo()
    {
        Console.WriteLine("Iniciando NinjaGelo ");
        vida = 40;
        forca = 20;
        velocidade = 18;
    }
    public void Gelo()
```

```
    {
        Console.WriteLine("Lancando gelo!");
    }
    public override void Update()
    {
        Console.WriteLine("=== UPDATE NINJAGELO ===");
        Socar();
        Chutar();
        Gelo();
    }
}
```

Neste exemplo, o método **Update()** da classe **Ninja** não será chamado. Será chamado somente o **Update()** da classe **NinjaGelo**. Ainda é possível chamar o **Update()** da classe **Ninja** dentro da classe **NinjaGelo**. Para isso, basta utilizar o comando base, que indica acesso ao conteúdo da classe base.

```
//Update na classe NinjaGelo
public override void Update()
{
//Chama o update da classe Ninja
```

base.Update();

}

/ /Assim os dois Updates são “combinados”.

O uso do virtual em um método indica que ele pode ser sobreposto em uma classe derivada, opcionalmente. Se declararmos um método como *abstract*, ele deve, obrigatoriamente, ser sobreposto.

Referências

Consideramos duas referências de lutadores para um sistema de luta no contexto do nosso exemplo.

```
public Ninja lutador1, lutador2;
```

Então, teremos o seguinte se considerarmos que o lutador1 será um objeto do tipo **NinjaGelo**:

```
NinjaGelo ninjaGelo = new NinjaGelo();
lutador1 = ninjaGelo;
```

Estamos atribuindo um objeto do tipo **NinjaGelo** para uma referência do tipo **Ninja**. Isso é possível porque **NinjaGelo** tem como base a classe **Ninja**. Ao chamarmos o **Update** do lutador1, mesmo sendo do tipo **Ninja**, será o do **NinjaGelo**, pois o **Update** da classe base foi substituída pelo *override* .

Sabemos que a referência do lutador2 também terá uma chamada do **Update**. Se a classe do lutador2 utilizar o *override* no **Update()**, o chamado será do novo método.

Caso o *override* não seja utilizado, o chamado será da classe base, ou seja, da **Ninja**.

```
lutador2.Update();
```

Comentários

Anteriormente, explicamos o que é um comentário e como fazê-lo e aqui vamos retomar o assunto e expandir. Antes, vamos relembrar o que são comentários.

Comentários são textos que estão dentro do nosso código, mas, por algum motivo, gostaríamos que o compilador não adicionasse ao programa.

Trabalhar com comentários é uma prática frequentemente utilizada por desenvolvedores por diferentes motivos, como:

- Explicar rapidamente o que determinada parte do código faz
- Guardar uma parte de código que estava funcionando anteriormente
- Isolar partes do código para verificação de *bugs*
- Descrever qualquer outra informação necessária e relevante

É considerado uma boa prática fazer comentários ao longo do código para explicar o funcionamento e a relação entre objetos, funções etc. O objetivo disso é tornar a leitura para outros programadores – que não escreveram aquela determinada parte do código – mais rápida e simples.

Além disso, ajuda o programador a não se perder. Com o tempo, as aplicações começam a ficar muito grandes e complexas, tornando difícil para o autor do código lembrar exatamente o funcionamento de cada fragmento do código.

Por não ser compilado, comentários podem ser frases normais utilizadas em idiomas. Ele aceita palavras com acento, símbolos, maiúsculas e minúsculas, espaços etc.

```
//Função para solicitação do login e senha do usuário
```

Mesmo que uma parte de código funcional esteja dentro de um comentário, o compilador irá ignorá-lo enquanto ele estiver comentado.

Existem, basicamente, dois tipos diferentes de comentários, que são:

Comentários em linha usando o símbolo //

Nesse tipo de comentário, tudo o que vier depois do símbolo // é considerado comentário, e será ignorado pelo programa. A linha seguinte já é considerada código novamente.

Comentários multilinhas usando os sinais /* e */

Tudo o que estiver entre os sinais /* e */ é desconsiderado pelo compilador.

#REGION

O #region é um meio para facilitar a estruturação de tópicos no VS Code. Este recurso serve para diminuir visualmente o código e deixá-lo agrupado por tipo de execução. Em arquivos com código extensos, é útil ter

a possibilidade de encurtar ou ocultar partes do código para poder se concentrar na parte do arquivo em que se está trabalhando.

Na prática, quando colocado pelo programador, possibilita apenas esconder ou mostrar códigos, não afetando a forma com que o código se relaciona entre si ou é executado. É um recurso unicamente voltado à organização e à visualização do código.

Os critérios para definir a área da região, quais códigos entram em qual etc., são definidos pelos programadores e dependem das características de cada projeto.

```
#region exemploClasse
class Example
{
}
#endregion

class Program
{

  #region atributos
  int exemplo01;
  string exemplo02;
  #endregion

  static void Main()
  {
    #region corpoPrincipal
    Console.WriteLine("Hello world!");
    #endregion
  }
}
```

Toda região precisa ser finalizada com #endregion e é possível ter várias regiões diferentes dentro de um mesmo código.

Namespace

Enquanto a complexidade e o tamanho da nossa aplicação vão aumentando, alguns problemas vão surgindo também. Em especial, dificuldade na compreensão do código e de gerenciamento nomes (classes, métodos etc.).

É relativamente comum bibliotecas de terceiros gerarem conflitos entre os nomes e isso acaba criando problemas nos códigos. Fique sempre atento!

Para contornar esses problemas, podemos utilizar *namespaces* , que servem para evitar conflitos de nomes e organizar o código.

Sem esse recurso, para remediar o problema do conflito de nomes, utilizaríamos nomes mais complexos e significativos, criando regras para essa separação. Essa solução funciona, mas os nomes acabam ficando muito grandes e complexos, gerando confusão para os programadores que precisam ficar trabalhando em diversos locais diferentes do código.

Os *namespaces* fornecem uma alternativa mais viável para a solução desse problema. É criado uma espécie de sobrenome que funciona como identificador para outros elementos. Nesse cenário, é possível ter duas ou mais classes de mesmo nome, mas localizadas em *namespaces* diferentes.

```
namespace GerenciamentoAreas
{
  public class Terreno
  {
  }
}
```

Para acessar essa classe, vamos utilizar o caminho **GerenciamentoAreas.Terreno**. Esse passa a ser o nome completo da classe.

O comando **using** serve para especificar uma classe sem precisar dizer o *namespace* ao qual ela pertence. Caso seja se opte por somente utilizar o primeiro nome, **Terreno**, no exemplo, podemos declarar o **using** – no início do arquivo – com o nome do *namespace* e, depois disso, acessar o **Terreno** sem indicar todo o caminho até ele.

```
using GerenciamentoAreas;
```

Um exemplo para visualizar isso é a classe **Console**, que pertence ao *namespace***System**. Com o **using**, uma chamada de **WriteLine** fica assim:

```
using System;
  public class Program
  {
    public static void Main()
    {
      Console.WriteLine("Mensagem");
    }
  }
```

Sem o comando **using**, é necessário especificar o *namespace* do **Console**:

```
public class Program
  {
    public static void Main()
    {
      System.Console.WriteLine("Mensagem");
    }
  }
```

Quando um código está dentro de um *namespace* , ele pode fazer referência direta a tudo que está diretamente no mesmo *namespace* .

Escopo

O escopo de uma variável é a região da aplicação (programa) em que ela é vista/utilizável. Além dos modificadores de acesso, que já estudamos anteriormente neste conteúdo, é importante notar que as variáveis são enxergadas dentro do bloco na qual foram declaradas. Logo, definimos os escopos usando os operadores de chaves (“{“ e ”}”).

As variáveis declaradas dentro de **if()**, **while()** e **for()** são enxergadas dentro do bloco seguinte. Veja um exemplo:

```
using System;
public class Program
{
  public static void Main()
  {
    int numero = 0;
    //   definição do escopo
    if(numero == 0)
    {
      int numeroExemplo = 20;
      Console.WriteLine(numeroExemplo);
    }
    //   Aqui o numeroExemplo não existe mais
    Console.WriteLine(numero);
  }
}
```

Como a variável **numero** foi declarada dentro do **Main** e fora de qualquer verificação, ela pode ser acessada em qualquer lugar do código do **Main**. Entretanto, a variável **numeroExemplo** só pode ser vista, acessada e modificada dentro do bloco de **if**.

Passagem de parâmetro e *return*

Parâmetros são meios de se passar valores para um método. Argumentos podem ser passados para parâmetros por valor ou por referência. A passagem por referência permite que métodos, propriedades, indexadores, operadores, construtores e membros da função alterem o valor dos parâmetros e façam essa alteração persistir no ambiente de chamada.

Parâmetros são as informações de entrada que o método pode usar em seu processamento.

A sintaxe de se passar parâmetros no C# é:

```
[modificador] Tipo_de_dado Nome_do_parametro
```

Para passar um parâmetro por referência com a intenção de alterar o valor, use a palavra-chave **ref** ou **out**. Para passar por referência com a intenção de evitar a cópia, mas não alterar o valor, use o modificador **in**.

Valor

Primeiro, temos a passagem de parâmetro por valor, que é o padrão utilizado no C#. Ou seja, se nenhum modificador estiver declarado, automaticamente é um parâmetro *value* . Passar uma variável de tipo de valor para um método por valor significa passar uma cópia da variável para o método e qualquer mudança no valor do parâmetro é feita localmente no método, sem ser passada de volta para o código que chamou o método (não afetam os dados originais armazenados na variável de argumento).

Em suma, parâmetros por valor passam uma cópia da variável, portanto, são somente de entrada. Então, por mais que o valor do parâmetro tenha sido modificado dentro do método, essa mudança não irá acontecer no local que chamou o método.

Veja um exemplo:

```
class TesteValue
{
  static void MetodoExemplo(int numero)
  {
    numero = 42;
    Console.WriteLine("Número no Metodo: " + numero);
    // Aqui a saída será 42
  }
  static void Main()
  {
    int valor = 20;
    //observe que a atribuição foi feita antes de chamar o método
    MetodoExemplo (valor);
    //na linha acima estamos chamando o método e passando o valor como pa
râmetro
    Console.WriteLine("Número na Main: " + valor);
    // Aqui a saída será 20, perceba que o valor não foi modificado
  }
}
```

Após a execução do método **Main**, o valor da variável **valor** será 20. Independentemente de o valor do parâmetro **numero** ter sido modificado dentro do método **MetodoExemplo**, essa mudança não ocorre também no **Main** (que chamou o método).

Uma analogia para o funcionamento do valor é pensar em quando você copia um arquivo de uma pasta e cola em outro lugar em seu computador. O que foi criado foi uma cópia, são arquivos independentes, que não têm ligação ou que não modificam um ao outro.

Referência

Uma variável de um tipo de referência não contém seus dados diretamente, ela contém uma referência a seus dados.

Tipos de referência armazenam uma referência, que por sua vez indicam um caminho para os dados, disponíveis em algum outro lugar na memória do computador. Quando você atribui o valor de uma variável de referência para outra, o que é copiado é a referência. Seguindo no exemplo de pastas, vamos imaginar as referências como atalhos para arquivos. Se você tem um ou mais atalhos que apontam para um arquivo em particular, quando você fizer alterações no arquivo, elas vão afetar todos os arquivos que acessamos por meio dos atalhos.

Parâmetro ref

Parâmetros **ref** são de entrada e saída, o que significa que podem ser usados tanto para passar um valor para um método quanto para receber de volta esse valor de uma função. Parâmetros **ref** são criados precedendo um tipo de dado com o modificador **ref**. Sempre que um parâmetro **ref** é passado, é uma referência da variável que é passada para o método. Qualquer operação no parâmetro é feita no argumento.

Um argumento passado para um parâmetro **ref** precisa ser inicializado antes de ser passado.

Vejamos um exemplo:

```
class TesteRef

{

   static void MetodoExemplo(ref int numero)

   {

      numero = numero + 41;
```

```
            Console.WriteLine("Número no Metodo: " + numero);

            // Aqui a saída será 42

        }

        static void Main()

        {

            int valor = 20;

            //observe que a atribuição foi feita antes de chamar o método

            MetodoExemplo(ref valor);

            //na linha acima estamos chamando o método e passando o valor como parâmetro por referência

            Console.WriteLine("Valor na Main = " + valor);

            // Aqui a saída também será 42



        }

    }
```

A saída do exemplo acima será 42, o parâmetro **ref** atua tanto como parâmetro de entrada quanto de saída. Na hora da declaração, o modificador **ref** deve ser escrito antes do parâmetro a ser passado pelo código chamador da função.

Parâmetro *out*

Um parâmetro **out** indica que este valor será “retornado” para a função original, ou seja, **passados por referência. Em outras palavras, qualquer operação no parâmetro é feita no argumento. É como a ref, exceto pelo fato de que ref requer que a variável seja inicializada antes de ser passada.**

Out é um parâmetro de saída. O parâmetro **out** é escrito antes do tipo de dado com o modificador **out**. Sempre que um parâmetro **out** é passado, somente a referência da variável é passada para a função:

```
class TesteOut
{
    static void MetodoExemplo(out int numero)
    {
        numero = 42;
        Console.WriteLine("Número no Metodo: " + numero);
        // Aqui a saída será 42
    }

    static void Main()
    {
        int valor;
```

```
            //observe que não foi feita uma atribuição antes de chamar o método

            Metodo(out valor);

            //na linha acima estamos chamando o método e passando o valor como parâmetro por referência out

            Console.WriteLine("Valor na Main = " + valor);

            // Aqui a saída será 42

        }

    }
```

A saída no exemplo acima no método **Main** será 42, uma vez que o valor do parâmetro **out** é passado de volta ao código chamador.

Um parâmetro **out** pode ser passado para a função como **null** (como no exemplo acima), mas dentro da função esta variável tem que ter a atribuição de algum valor, caso contrário, o compilador vai exibir uma mensagem de erro.

Parâmetro *params*

O parâmetro **params** é muito útil quando o número de argumentos a ser passado é variável. O parâmetro **params** é somente de entrada.

Usando o **params**, você pode enviar uma lista separada por vírgulas dos argumentos do tipo especificado na declaração de parâmetros ou uma matriz de argumentos do tipo especificado. Você também pode não enviar

argumento. Se você não enviar argumento, o comprimento da lista **params** será zero.

Nenhum parâmetro adicional é permitido após a **params** em uma declaração de método e apenas uma palavra-chave **params** é permitida em uma declaração de método.

Vejamos um exemplo:

```
class TesteParams
{
   static int SomaExemplo(params int[] numeros)
   {
      int aux = 0;
      foreach(int P in numeros)
      {
         aux = aux + P;
      }
      return aux;
   }
   static void Main()
   {
      Console.WriteLine(SomaExemplo(42, 20));
```

```
            //Aqui a saída vai ser 62

            Console.WriteLine(SomaExemplo(42, 42, 42, 42));

            //Aqui a saída vai ser 168

        }

    }
```

A saída do programa acima será 62 e 168.

Note que parâmetros **ref** e **out** devem ser informados tanto no protótipo do método quanto na chamada.

Programação orientada a objetos *versus* programação estruturada

Neste conhecimento, nos aprofundamos mais ainda no mundo da programação orientada a objetos. É importante notar que a maioria dos recursos que aprendemos não estão presentes na programação estruturada, pois os recursos da linguagem orientada a objetos são fortemente ligados ao reaproveitamento de código. Sendo assim, um mesmo código pode ser chamado em diferentes locais do código, alterado em diversos pontos e executado com alterações feitas durante a execução.

A programação orientada a objetos não é melhor ou pior do que a estruturada, é apenas um modo diferente que atende a demandas diferentes da programação estruturada. A orientação a objetos é muito ligada aos problemas modernos que temos em programação, otimização de tempo, organização de código e reaproveitamento.

Utilizando os recursos aprendidos neste material, você agora é capaz de criar um código dinâmico, que é fortemente genérico para permitir uma extensão reutilização.