Aula 5 - Orientação a Objetos básica

Docupedia Export

Author:Balem Luis (CtP/ETS)
Date:31-Mar-2023 17:32

Table of Contents

1 Orientação a Objetos

Agora que já compreendemos o básico de C# e como programar no mesmo usando programação estruturada, podemos finalmente aprender como usar a Orientação a Objetos na linguagem, bem como o que é este paradigma de programação.
A Orientação a Objetos é basicamente a separação das implementações e dados de um programa em estruturas chamadas Objetos. Um Objeto é como se fosse qualquer dado que seu computador salva durante a execução de uma aplicação, ou seja, um espacinho na memória com bits de dados. Porém, um objeto pode conter uma grande quantidade de dados, como se fosse um conjunto de informações específicas sobre aquela instância. Outra característica de um objeto é que além de um estado, que é como os dados são apresentados dentro dele, ele também tem várias funções que desempenham uma funcionalidade diferente a depender do estado deste objeto.
Um exemplo poderia ser um cadastro de clientes. Você tem vários clientes, cada um com nome, endereço, cpf, entre outras informações. Em um único bloco de dados você guarda todas essas informações. Perceba que você tem vários objetos que tem a mesma estrutura, porém com dados diferentes. Ainda assim, você tem funções que quando executadas, tem um comportamento diferente a depender do objeto para qual são chamadas. Por exemplo, a função AtualizarEndereco vai mudar o endereço de um cliente específico, mas não de todos. Então o comportamento da função depende exclusivamente do objeto associado.
Isso é bem útil, porque sem esse tipo de ferramenta é difícil expressar a existência de vários objetos estruturalmente parecidos, mas com dados diferentes. Precisaríamos criar vários vetores com todos os dados de forma global (como era feito antigamente) para conseguir criar aplicações dessa natureza.
Para criar um objeto precisamos antes de um modelo, esse modelo especifica estruturalmente quais dados teremos bem como quais funções poderemos executar sobre esses objetos. A partir deste modelo iremos criar vários objetos. Este modelo se chama **Classe**. Abaixo, a forma de criar uma classe cliente como mencionada anteriormente em C#:

```
1  public class Cliente
2  {
3      public string Nome;
4      public string Endereco;
5      public long Cpf;
6
7      public void AtualizarEndereco(string novoEndereco)
8      {
9          Endereco = novoEndereco;
10     }
11 }
```

É importante perceber algumas coisinhas no código acima. Primeiramente a palavra reservada 'public' faz com que tudo que você está colocando seja visto e possa ser utilizado. Segundamente, o código acima não é executável: Você não deve usar dentro de outras funções, dentro de um For ou If. Ele não é executado em momento algum, é apenas uma declaração de estrutura, mas não um código executado linha a linha. As únicas coisas que eventualmente são executadas são as funções colocadas dentro destas classes.
O nome das variáveis que você vê dentro da classe chamam-se **Campos** em C#; já a função, que não segue mais o exemplo de função da matemática já que seu comportamento varia a depender do estado do objeto, recebe o nome de **Método**.
A utilização do código acima é bem simples:

```
using static System.Console;

Cliente cliente1 = new Cliente();
cliente1.Nome = "Gilmar";
cliente1.Endereco = "Rua do Gilmar";
cliente1.Cpf = "12345678-09"

Cliente cliente2 = new Cliente();
cliente2.Nome = "Pamella";
cliente2.Endereco = "Rua da Pamella";
cliente2.Cpf = "87654321-90"

cliente2.AtualizarEndereco("Avenida da Pamella");
```

Note que ao criarmos uma classe, estamos criando um tipo completamente novo. Assim, fazemos duas variáveis cliente1 e cliente2 e usamos a palavra reservada 'new' seguido do nome da classe e parênteses. Note que os dois objetos são diferentes e tem dados diferentes. Usamos a notação de ponto (variável.Campo/Método) para acessar os campos/métodos dentro da classe. Ao executar o AtualizarEndereco do cliente 2 apenas o endereço dele será atualizado.
A capacidade que a OO (Orientação a Objetos) nos da de representar um objeto real com suas características é um dos pilares da OO e chamamos ela de **Abstração**. A partir dessa característica construiremos aplicações bem modularizadas e poderosas.
Para criar uma classe, basta adicionar ao fim do arquivo Top Level, ela será automaticamente considerada fora da função Main.

2 Padrões de Nomenclatura

Agora que conhecemos a classe, vamos rapidamente entender o padrão de nomenclatura utilizada no C#:

- Para coisas públicas, utilizamos PascalCase: Escrevemos o nome onde cada palavra começa com letra maiúscula.
- Para coisas não-públicas, incluindo variáveis internas de métodos e parâmetros de funções públicas, usamos camelCase: A primeira palavra começa em minúsculo e as próximas em maiúsculo.

Verá que existem momentos em que podemos desrespeitar levemente essas regras, mas em geral, as siga.

3 Um exemplo OO usando Overload (Sobrecarga)

Overload é uma das capacidades da OO em que dentro de classes podemos ter várias funções com mesmo nome, desde que tenham parâmetros diferentes. Abaixo segue um exemplo legal onde podemos usar a OO para representar uma classe para horários e, adicionalmente, criamos um método Adicionar, que avança no tempo para podermos alterar o horário armazenado.
Importante: Não basta o nome dos parâmetros ser diferente, mas sim a quantidade ou os tipos devem diferir.

```
1  using static System.Console;
2
3  Horario h1 = new Horario();
4  Horario h2 = new Horario();
5
6  h1.Adicionar(20, 40, 30); //h1 = 20:40:30
7  h2.Adicionar(20, 30); //h2 = 00:20:30
8
9  h1.Adicionar(h2); //h1 = 21:01:00, h2 inalterado
10 h2.Adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado
11
12 WriteLine(h2.Formatar()); //21:21:30
13
14 public class Horario
15 {
16     // Valor inicial é 00:00:00
17     public int Hora = 0;
18     public int Minuto = 0;
19     public int Segundo = 0;
20
21     public void Adicionar(int segundos, int minutos, int horas)
22     {
23         Segundo += segundos;
24         if (Segundo > 59)
25         {
26             minutos += Segundo / 60;
27             Segundo = Segundo % 60;
28         }
29
30         Minuto += minutos;
31         if (Minuto > 59)
32         {
```

```
33              horas += Minuto / 60;
34              Minuto = Minuto % 60;
35          }
36
37          Hora += horas;
38          if (Hora > 23)
39          {
40              Hora = Hora % 24;
41          }
42      }
43
44      public void Adicionar(int segundos, int minutos)
45      {
46          Adicionar(segundos, minutos, 0);
47      }
48
49      public void Adicionar(int segundos)
50      {
51          Adicionar(segundos, 0, 0);
52      }
53
54      public void Adicionar(Horario horario)
55      {
56          Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
57      }
58
59      public string Formatar()
60      {
61          return $"{Hora}:{Minuto}:{Segundo}";
62      }
63  }
```

4 Construtores

Além disso, podemos fazer construtores, que são funções chamadas no momento que os objetos são criados a partir das classes. É possível ainda usar a sobrecarga de métodos para ter vários construtores diferentes. Esses construtores são usados, em geral, para inicializar os objetos e não costuma ter código muito pesado dentro deles. Observe o exemplo anterior, agora com construtores:

```
1   using static System.Console;
2
3   Horario h1 = new Horario(20, 40, 30); //h1 = 20:40:30
4   Horario h2 = new Horario(); //h2 = 00:00:00
5
6   h2.Adicionar(20, 30); //h2 = 00:20:30
7
8   h1.Adicionar(h2); //h1 = 21:01:00, h2 inalterado
9   h2.Adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado
10
11  WriteLine(h2.Formatar()); //21:21:30
12
13  public class Horario
14  {
15      // Valor inicial é 00:00:00
16      public int Hora = 0;
17      public int Minuto = 0;
18      public int Segundo = 0;
19
20      public Horario() // Construtores não tem retorno e o nome é o nome da classe
21      {
22          // Vazio não altera os dados
23      }
24
25      public Horario(int segundos, int minutos, int horas)
26      {
27          // Geralmente inicializamos 1 a 1, mas aqui preferimos usar a função Adicionar que já está pronta
28          // Hora = horas;
29          // Minuto = minutos;
30          // Segundo = segundos;
31          Adicionar(segundos, minutos, horas);
32      }
33
```

```
34      public void Adicionar(int segundos, int minutos, int horas)
35      {
36          Segundo += segundos;
37          if (Segundo > 59)
38          {
39              minutos += Segundo / 60;
40              Segundo = Segundo % 60;
41          }
42
43          Minuto += minutos;
44          if (Minuto > 59)
45          {
46              horas += Minuto / 60;
47              Minuto = Minuto % 60;
48          }
49
50          Hora += horas;
51          if (Hora > 23)
52          {
53              Hora = Hora % 24;
54          }
55      }
56
57      public void Adicionar(int segundos, int minutos)
58      {
59          Adicionar(segundos, minutos, 0);
60      }
61
62      public void Adicionar(int segundos)
63      {
64          Adicionar(segundos, 0, 0);
65      }
66
67      public void Adicionar(Horario horario)
68      {
69          Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
70      }
71
72      public string Formatar()
```

```
73      {
74          return $"{Hora}:{Minuto}:{Segundo}";
75      }
76  }
```

5 Encapsulamento

Observe o exemplo do horário e responda: Não é perigoso que um programador desavisado tente executar um código como o abaixo?

```
horario.Segundo += 1;
```

O código simples somaria um segundo no campo Segundo. O grande problema é que isso poderia fazer com que o valor de segundos chegasse a mais de 60 sem adicionar um valor no minuto. O programador não sabe ou esquece que o horário deve manter-se consistente e que deve utilizar o método Adicionar para fazer isso. E qual é o problema? Isso pode acarretar em Bugs. Este caso é simples, mas o mesmo problema pode escalar de formas astronômicas.
Outra situação: Imagine que você cria um sistema que utiliza uma classe da seguinte forma:

```
public class Cliente
{
    public string Login;
    public string Senha;
}
```

Depois de anos seu sistema é comprado e agora exige-se que a senha seja criptografada por questões de segurança. Isso faz com que você faça algo do tipo:

```
public class Cliente
{
    public string Login;
    public string Senha;

    public void DefinirSenha(string value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        Senha = senhaCriptografada;
    }
}
```

Ainda assim, você esquece de trocar algumas partes do software, as atribuições de senha pela função DefinirSenha e pior ainda, desavisados, os seus colegas acabam realizando implementações esquecendo-se de usar o DefinirSenha. Isso acarreta que agora várias senhas salvas estão sem criptografia e muito pior que isso: Você não sabe exatamente quais estão criptografadas e quais são só estranhas.
Isso pode causar muita dor de cabeça, e gerar muitos bugs. Graças a isso, a OO nos trás o pilar do **Encapsulamento** - o segundo pilar, depois da abstração, de um total de 4 pilares da OO. Para aplicá-lo usaremos a palavra reservada 'private' que esconde as estruturas de um código:

```
using static System.Console;

```

```
Cliente c = new Cliente();
c.senha = "Xispita"; //Erro pois senha é agora privada
WriteLine(c.ObterSenha());

public class Cliente
{
    public string Login;
    private string senha; // Letra minúscula (CamelCase), pois agora é privado

    public void DefinirSenha(string value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
    }

    public string ObterSenha()
    {
        return senha;
    }
}
```

Na verdade, nenhum campo deve ser público. Devemos usar essas funções de Definir e Obter, quando possível, para acessar os valores. Usamos o inglês Get e Set para as mesmas:

```
using static System.Console;

Cliente c = new Cliente();
c.SetLogin("Gilmar");
c.SetSenha("Xispita");

public class Cliente
{
    private string login;
    private string senha;

    public void SetSenha(string value)
    {
        string senhaCriptografada = "";
```

```
15          // Criptografa o value e guarda na variável senhaCriptografada
16          senha = senhaCriptografada;
17      }
18
19      public string GetSenha()
20      {
21          return senha;
22      }
23
24      public void SetLogin(string value)
25      {
26          login = value;
27      }
28
29      public string GetLogin()
30      {
31          return login;
32      }
33  }
```

Note que as funções GetLogin e SetLogin parecem inúteis, diferente da senha. Mas é justamente isso que é interessante. Em qualquer momento que precisemos alterar a forma de como o código conversa com os dados de login, basta alterar esses métodos. Ou seja, não precisamos reestruturar todo o código para que as coisas funcionem. Olhe como a vida seria mais fácil se usassemos Get/Set na senha desde o inicio:

```
1   // Antes da adição de criptografia, parecem métodos inúteis
2   private senha;
3   public void SetSenha(string value)
4   {
5       senha = value;
6   }
7
8   public string GetSenha()
9   {
10      return senha;
11  }
12
13  // Depois da adição da criptografia, pequena alteração e não precisa alterar mais nada no sistema
14  private senha;
15  public void SetSenha(string value)
```

```
16  {
17      string senhaCriptografada = "";
18      // Criptografa o value e guarda na variável senhaCriptografada
19      senha = senhaCriptografada;
20  }
21
22  public string GetSenha()
23  {
24      return senha;
25  }
```

Voltando ao nosso exemplo de Classe Horario, poderíamos reestruturá-la. Note que o Set pode ter uma implementação peculiar:

```
1   public class Horario
2   {
3       private int hora = 0;
4       private int minuto = 0;
5       private int segundo = 0;
6
7       public Horario() { }
8
9       public Horario(int segundos, int minutos, int horas)
10      {
11          Adicionar(segundos, minutos, horas);
12      }
13
14      public int GetHora()
15      {
16          return hora;
17      }
18
19      public int GetMinuto()
20      {
21          return minuto;
22      }
23
24      public int GetSegundo()
25      {
26          return segundo;
```

```
27      }
28
29      public void SetHora(int value)
30      {
31          hora = 0;
32          Adicionar(value, 0, 0);
33      }
34
35      public void SetMinuto(int value)
36      {
37          minuto = 0;
38          Adicionar(value, 0);
39      }
40
41      public void SetSegundo(int value)
42      {
43          segundo = 0;
44          Adicionar(value);
45      }
46
47      public void Adicionar(int segundos, int minutos, int horas)
48      {
49          segundo += segundos;
50          if (segundo > 59)
51          {
52              minutos += segundo / 60;
53              segundo = segundo % 60;
54          }
55
56          minuto += minutos;
57          if (minuto > 59)
58          {
59              horas += minuto / 60;
60              minuto = minuto % 60;
61          }
62
63          hora += horas;
64          if (hora > 23)
65          {
```

```
66                  hora = hora % 24;
67              }
68          }
69
70          public void Adicionar(int segundos, int minutos)
71          {
72              Adicionar(segundos, minutos, 0);
73          }
74
75          public void Adicionar(int segundos)
76          {
77              Adicionar(segundos, 0, 0);
78          }
79
80          public void Adicionar(Horario horario)
81          {
82              Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
83          }
84
85          public string Formatar()
86          {
87              return $"{hora}:{minuto}:{segundo}";
88          }
89      }
```

Infelizmente, é cansativo escrever um Get e Set. Se você fosse um programador Java, teria que se contentar com geradores automáticos que ainda deixam o código gigantesco. Felizmente, como desenvolvedor .NET, o C# possui algumas melhorias de sintaxe. Você pode escrever o Get/Set de login, por exemplo, da seguinte forma:

```
1   private string login;
2   public string Login
3   {
4       get
5       {
6           return login;
7       }
8       set
9       {
10          login = value;
```

```
    }
}
```

A palavra reservada contextual 'value' pode ser usada no set sem declarar. Além disso, ao usar este get/set você pode usar como se fosse uma variável. Vamos supor que você queira adicionar um "@" no início do login de um usuário. Veja a diferença de código usando o tradicional vs get/set desta forma:

```
// Tradicional
cliente.SetLogin("@" + cliente.GetLogin());

// Forma C#
cliente.Login = "@" + cliente.Login;
```

O nome desta forma mais simples chama-se propriedade.
Além disso você pode usar a forma autoimplementada. Ela cria o campo privada escondido por trás e implementa da forma mais simples possível o get/set:

```
public string Login { get; set; }
```

Você ainda pode deixar o set privado, veja uma possibilidade de implementação para a classe Horario usando esses recursos:

```
public class Horario
{
    //É possível ler Hora, Minuto e Segundo, mas alterá-la só com a função Adicionar, Construtor ou internalmente na classe
    public int Hora { get; private set; } = 0;
    public int Minuto { get; private set; } = 0;
    public int Segundo { get; private set; } = 0;

    public Horario() { }

    public Horario(int segundos, int minutos, int horas)
    {
        Adicionar(segundos, minutos, horas);
    }

    public void Adicionar(int segundos, int minutos, int horas)
    {
        Segundo += segundos;
        if (Segundo > 59)
        {
            minutos += Segundo / 60;
```

```
            Segundo = Segundo % 60;
        }

        Minuto += minutos;
        if (Minuto > 59)
        {
            horas += Minuto / 60;
            Minuto = Minuto % 60;
        }

        Hora += horas;
        if (Hora > 23)
        {
            Hora = Hora % 24;
        }
    }

    public void Adicionar(int segundos, int minutos)
    {
        Adicionar(segundos, minutos, 0);
    }

    public void Adicionar(int segundos)
    {
        Adicionar(segundos, 0, 0);
    }

    public void Adicionar(Horario horario)
    {
        Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
    }

    public string Formatar()
    {
        return $"{Hora}:{Minuto}:{Segundo}";
    }
}
```

Assim com o Encapsulamento podemos esconder dados e decidir como eles serão acessados/processados.

6 Dicas

Você pode, em implemenações de uma única linha, substituir as chaves por uma seta => para deixar o código mais enchuto. Também é recomendável que você utilize a palavra reservada 'this' - que se refere a própria classe - para melhorar a legibilidade quando você acessar algo dentro da propria classe que pertença a mesma:

```
1  public class Horario
2  {
3      public int Hora { get; private set; } = 0;
4      public int Minuto { get; private set; } = 0;
5      public int Segundo { get; private set; } = 0;
6
7      public Horario() { }
8
9      public Horario(int segundos, int minutos, int horas)
10         => Adicionar(segundos, minutos, horas);
11
12     public void Adicionar(int segundos, int minutos, int horas)
13     {
14         this.Segundo += segundos;
15         if (this.Segundo > 59)
16         {
17             minutos += this.Segundo / 60;
18             this.Segundo = this.Segundo % 60;
19         }
20
21         this.Minuto += minutos;
22         if (this.Minuto > 59)
23         {
24             horas += this.Minuto / 60;
25             this.Minuto = this.Minuto % 60;
26         }
27
28         this.Hora += horas;
29         if (this.Hora > 23)
30         {
31             this.Hora = this.Hora % 24;
32         }
33     }
34
```

```
35      public void Adicionar(int segundos, int minutos)
36          => Adicionar(segundos, minutos, 0);
37
38      public void Adicionar(int segundos)
39          => Adicionar(segundos, 0, 0);
40
41      public void Adicionar(Horario horario)
42          => Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
43
44      public string Formatar()
45          => $"{this.Hora}:{this.Minuto}:{this.Segundo}";
46  }
```

7 Exercício Proposto 2

Crie uma classe cliente com Nome, Endereço e Cpf. O Cpf deve possuir um campo privado no tipo long que guarda o Cpf como um número. A propriedade get/set do Cpf deve receber uma string na forma "123.456.789-90" e converte-lá num número, depois, no get, o contrário deve ocorrer retornando a string formatada.