INFORMÁTICA PARA INTERNET

C#: linguagem de *scripts* e fundamentos da programação

Continuando os estudos, agora é o momento de aprofundar ainda mais os conhecimentos sobre linguagem. Portanto, primeiramente, serão estudados os laços de repetição e, em seguida, o que é um paradigma de programação e como a tão falada orientação a objetos impacta a programação.

Por último – mas não menos importante –, serão abordadas a orientação a objetos em si, as suas funcionalidades e as suas peculiaridades.

Preparação para o estudo

O material apresenta exemplos práticos e desafios. Sendo assim, recomenda-se que você utilize o Visual Studio Code com a extensão C# instalada. A cada exemplo, crie um projeto do tipo console. Para tanto, siga estes passos:

1. Abra o Visual Studio Code.

2. Crie uma pasta na qual você deve armazenar o seu projeto e abra-a no Visual Studio Code.

3. Abra a aba de terminal usando o atalho **Ctrl + '** ou acessando o menu em **View > Terminal**.

4. Digite o comando `dotnet new console` e depois clique em **Enter**.

5. Abra o arquivo **Program.cs** gerado e faça a sua programação dentro do método **Main()**.

6. Após as alterações, utilize o comando dotnet run para rodar a aplicação no terminal do Visual Studio Code.

Caso necessário, reveja o conteúdo **Introdução à programação em C#**, em **Arquitetura *web* em camadas** e **Ambiente de desenvolvimento**, da unidade curricular **Monitorar projetos de aplicações *web***.

Resumo do que se sabe até agora

O nosso percurso de programação já passou pelas noções de algoritmos, pela organização das instruções e pela utilização de C#, nas quais foram criadas aplicações que conseguem manipular dados, realizar operações, receber entradas, gerar saídas e tomar decisões baseadas em comparações. Os principais conceitos que, neste momento, devem estar claros são:

Variáveis
Trata-se de porções de memória nomeadas que armazenam um valor. Em C#, as variáveis são fortemente tipadas (permitem apenas valor de um tipo, como número inteiro, decimal e texto).

Entradas
Trata-se de dado externo usado pelo algoritmo para o processamento. Em C#, em projetos do tipo console, usa-se o método `Console.ReadLine()` .

Saídas

Trata-se de resposta do processamento exposta ao usuário. Em C#, em projetos do tipo console, pode-se usar o método `Console.WriteLine()`.

Condições

Trata-se de comparações entre dois valores (que podem vir de variáveis), as quais geram a resposta “verdadeiro” ou “falso”. Em C#, têm-se operadores como **<**, **>**, **!=** e **==**, além de operações lógicas como **E (&&)** e **OU (||)**.

Decisões

Trata-se de desvios no fluxo do algoritmo que dependem de uma condição, ou seja, de estruturas de “se-então”. Em C#, usam-se `if-else` e `switch`.

Veja no código a seguir um exemplo que utiliza tais conceitos. Trata-se de um algoritmo que recebe o ano de nascimento de uma pessoa e calcula se ela é maior ou menor de idade.

Exemplo 1 – Programa em C# que verifica maioridade do usuário

```
                class Program {
                  static void Main(string[] args) {
                    /* definição de variáveis locais
                    do tipo inteiro */
                    int anoNascimento, idade;

                    anoNascimento = int.Parse(Console.ReadLine()); //leitura de
dados do usuário
                    idade = DateTime.Now.Year - anoNascimento; //ano atual - an
o informado

                    /* desvio condicional */
                    if (idade > 0)
                    {
                        if (idade >= 18) //if aninhado
                        {
                            Console.WriteLine("Maior de idade");
                        }
                        else
                        {
                            Console.WriteLine("Menor de idade");
                        }
                    }
                    else // idade é menor que zero
                    {
                        Console.WriteLine("Ano de nascimento inválido");
                    }
                  }
                }
```

Comentários em C#

Antes de avançar, note no exemplo anterior as frases descritivas no meio do código. Elas são os comentários, os trechos de texto livre ignorados pelo compilador que permitem um tipo de documentação do programa, ajudando a compreender a leitura do código por um desenvolvedor.

Há essencialmente dois tipos de comentários em C#:

1. Comentários em linha utilizando o símbolo **//**

O texto deve ocupar apenas uma linha. Tudo o que vier depois do símbolo **//** é considerado comentário e será ignorado pelo programa. A linha seguinte já é considerada código novamente.

2. Comentários multilinhas utilizando os sinais **/*** e ***/**

Com os sinais **/*** e ***/**, é possível escrever textos mais longos, que quebrem linhas. Tudo o que estiver entre os sinais **/*** e ***/** é desconsiderado pelo compilador. Além disso, os sinais podem ser utilizados também para um comentário aninhado ao código. Por exemplo, o trecho `if(idade >= 18 /*maior de idade*/)` seria considerado válido.

Para relembrar, reveja no exemplo 1 os comentários mencionados em uso.

Há ainda um comentário muito específico de ferramentas (tais como o Visual Studio Code) que consideram o sinal **///** como sendo um comentário de documentação. Como essas ferramentas permitem gerar documentação a partir do código, ficou estabelecido que esse sinal é um indicativo de que ali há informações sobre funcionalidades de um método ou uma classe, por exemplo.

Laços de repetição: conceito, sintaxe, identificadores, operadores e palavras reservadas

Imagine que você deseja construir um código que leia o salário de cinco funcionários e calcule a média salarial deles. Você poderia proceder da seguinte forma:

Exemplo 2 – Programa em C# que lê cinco valores de salários e calcula a média destes

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            float salario, soma=0,media;

            Console.WriteLine("Informe o valor do salário:");
            salario = float.Parse(Console.ReadLine());
            soma = soma + salario;

            Console.WriteLine("Informe o valor do salário:");
            salario = float.Parse(Console.ReadLine());
            soma = soma + salario;

            Console.WriteLine("Informe o valor do salário:");
            salario = float.Parse(Console.ReadLine());
            soma = soma + salario;

            Console.WriteLine("Informe o valor do salário:");
            salario = float.Parse(Console.ReadLine());
            soma = soma + salario;

            Console.WriteLine("Informe o valor do salário:");
            salario = float.Parse(Console.ReadLine());
            soma = soma + salario;

            media = soma / 5;
            Console.WriteLine("O valor médio dos salários é:
{0:N}",media);
        }
    }
}
```

O conjunto de linhas a seguir se repete cinco vezes para ler e somar os cinco salários:

```
Console.WriteLine("Informe o valor do salário:");
salario = float.Parse(Console.ReadLine());
soma = soma + salario;
```

Imagine agora que você quer fazer a média salarial de 100 funcionários e, em um outro caso, de 10.000.

Para evitar o número excessivo de linhas de código repetidas, são utilizados **laços de repetição**, uma das estruturas fundamentais de algoritmos e programação. Os laços de repetição permitem que um conjunto de comandos seja executado diversas vezes sem implicar um número maior de linhas de código.

Um laço sempre conta com uma condição para ser encerrado (caso contrário, os comandos que ele contém são reexecutados infinitamente). Assim, de maneira geral, um laço realiza uma comparação e, dependendo do resultado (como em um `if`), executa o conjunto de instruções ou encerra a repetição, partindo para a instrução imediatamente posterior.

Com relação ao momento em que a comparação acontece, podem-se classificar os laços como de “pré-teste” ou “pós-teste”.

Clique ou toque para visualizar o conteúdo.

(#modal-pre-teste) (#modal-pos-teste)

Laço de repetição com variável de controle

O laço de repetição com variável de controle conta com um número definido de repetições que serão executadas e controladas por uma variável.

Esse tipo de estrutura é utilizado quando se sabe exatamente o número de repetições a serem realizadas. Os parâmetros são uma variável com um valor inicial, uma condição de parada para tal variável e o incremento para esta.

Veja a sintaxe:

```
for(valor inicial; condição; incremento)
{
  <Lista de Comandos>;
}
```

O exemplo da média salarial dos funcionários poderia ser implementado também com a estrutura `for` . Veja o exemplo a seguir:

Exemplo 6 – Programa em C# que lê cinco valores de salários e calcula a média deles com *for*

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            float salario, soma=0,media;
            int cont;
            for(cont=1;cont<=5;cont++)
            {
              Console.WriteLine("Informe o valor do salário:");
              salario = float.Parse(Console.ReadLine());
              soma = soma + salario;
            }
            media = soma / cont;
            Console.WriteLine("O valor médio dos salários é:
{0:N}",media);
        }
    }
}
```

O laço `for` tem uma variável chamada **cont** que recebe 1 como valor inicial. Os comandos desse laço são repetidos enquanto **cont** for menor que 5. O incremento da variável **cont** é descrito no início do laço *for*.

Exemplo 7 – Somando os “n” primeiros números naturais

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
          int n = 5, soma = 0;
            for (int i=1; i<=n; i++)
            {
                soma += i;
            }
        }
    }
}
```

Veja que no exemplo anterior a variável `n` poderia ser inclusive preenchida com uma entrada de dados do usuário, deixando o laço mais flexível.

Nem sempre o incremento será uma operação `++`. Na verdade, ele pode ser qualquer operação. Veja no exemplo a seguir um trecho de código que mostra os `n` primeiros números pares. Nele, utilizou-se `i = i + 2`.

```
int n = 5;
for (int i = 0; i < n; i = 1 + 2)
{
      Console.WriteLine(i);
  }
```

Você está pronto para testar o que aprendeu até aqui sobre laços?

Para começar, tente fazer os exercícios propostos a seguir:

1. Desenvolva um código que leia um número inteiro e devolva o fatorial deste (por exemplo: fatorial de 5 é igual a 5 x 4 x 3 x 2 x 1).

2. Desenvolva um código que leia o nome e o tempo de dez cavalos e traga o nome e o tempo dos cavalos mais rápidos.

3. Desenvolva um código que imprima os números menores que 3.000 em ordem decrescente.

4. Desenvolva um código que leia um conjunto de números e imprima a quantidade de números pares e a quantidade de números ímpares lidos. A leitura deve acontecer consecutivamente até que o usuário digite o número 9999. Neste caso, o programa encerra mostrando as quantidades calculadas.

Laço de repetição condicional pré-teste

Por meio de um laço de repetição, é possível fazer com que comandos repetidos sejam executados várias vezes **enquanto** uma condição é satisfeita. Essa condição pode ser testada no início ou no final do laço.

Para o laço de repetição condicional pré-teste, antes que qualquer comando do laço seja executado, uma condição é testada. Enquanto ela for verdadeira, os comandos serão repetidos.

Veja a sintaxe:

```
while(condição)
{
  <Lista de Comandos>
}
```

Pode-se refazer o código do exemplo 2 utilizando a estrutura `while` :

Exemplo 3 – Programa em C# que lê cinco valores de salários e calcula a média deles com `while`

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            float salario, soma=0,media;
            int cont=0;
            while(cont<5)
            {
              Console.WriteLine("Informe o valor
do salário:");
              salario = float.Parse(Console.Read
Line());
              soma = soma + salario; //ACUMULADO
R
              cont = cont +1;        //CONTADOR
            }
            media = soma / cont;
            Console.WriteLine("O valor médio dos
salários é: {0:N}",media);
        }
    }
}
```

No código anterior, foi declarada e inicializada uma variável **cont** do tipo inteiro com o valor 0.

Tal variável é utilizada para contar o número de vezes que os comandos serão executados dentro do laço. A cada iteração (passagem dentro do laço), a variável “cont” é incrementada em 1 (cont = cont + 1;). Sempre que uma variável recebe ela mesma mais um valor constante, é chamada de contador.

```
variável = variável + contante numérica;
```

Quando uma variável recebe ela mesma mais uma segunda variável, é chamada de acumulador.

```
variável1 = variável1 + variável2;
```

Contadores e acumuladores são muito comuns em estruturas de repetição. O código anterior faz o mesmo que o código mencionado antes dele, mas é muito mais eficiente e contém menos linhas de código.

➔ O laço faz a comparação **cont < 5?**. Se o resultado for positivo, ele executa as instruções que estão no escopo do laço (ou seja, entre o par de chaves { e }) uma a uma. Se o resultado da comparação for falso, então avança para a instrução que vem logo após a chave que encerra o laço.

➔ Dentro do laço, são executados o WriteLine, o ReadLine, a soma e a contagem.

➔ Após, o laço volta ao seu topo, refazendo a comparação inicial. Nesse segundo momento, **cont** foi atualizado. Assim, se **cont < 5**, as instruções são executadas novamente; senão, o controle da aplicação sai do laço.

Dois fatos no exemplo anterior são importantes:

1. Se **cont** já iniciasse com valor maior ou igual a 5, o laço não executaria nenhuma vez.

2. Se dentro do *while* não houvesse a instrução `cont = cont +1`, o laço nunca pararia de executar, tendo em vista que, inalterado, o valor de **cont** nunca chegaria a ficar maior ou igual a 5.

Caso a intenção fosse calcular a média salarial de 1.000 funcionários, bastaria alterar a condição do *while* para **cont<1000**.

Veja a seguir um novo exemplo de utilização do laço *while*.

Uma progressão aritmética (PA) é uma sequência numérica em que cada termo, a partir do segundo, é igual à soma do termo anterior com uma constante. Essa constante é chamada de razão ou de diferença comum da progressão aritmética. O programa lerá o primeiro e o segundo números da progressão e, baseado em sua razão, calculará dez elementos dessa PA.

Observe este exemplo de execução:

```
Informe o primeiro número: (usuário digita) 4
Informe o segundo número:  (usuário digita) 10
Resultado: 4  10  16  22  28  34  40  46  52 58
```

Exemplo 4 – Programa em C# que calcula uma progressão aritmética

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            int num,num2,cont=0,razao;
            Console.WriteLine("Informe o primeir
o número:");

            num = int.Parse(Console.ReadLine());
            Console.WriteLine("Informe o segundo

número:");
            num2 = int.Parse(Console.ReadLine
());

            razao = num2 - num;
            while(cont<10)
            {
              Console.Write(" "+num);
              num = num + razao;
              cont++;
            }
        }
    }
}
```

Laço de repetição condicional pós-teste

Para o laço de repetição condicional pós-teste, toda a lista de comandos será executada pelo menos uma vez para depois a condição ser testada. Enquanto essa condição for verdadeira, os comandos serão repetidos.

Veja a sintaxe:

```
do{
  <Lista de Comandos>
}while(condição);
```

Pode-se refazer o código anterior utilizando a estrutura *while*:

Exemplo 5 – Programa em C# que lê cinco valores de salários e calcula a média deles com “do *while*”

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            float salario, soma=0,media;
            int cont=0;

            do{
              Console.WriteLine("Informe o valor do sal
ário:");

              salario = float.Parse(Console.ReadLine
());

              soma = soma + salario; //ACUMULADOR
              cont = cont +1;        //CONTADOR
            }while(cont<5);
            media = soma / cont;
            Console.WriteLine("O valor médio dos salári
os é: {0:N}",media);
        }
    }
}
```

Esse programa funciona semelhantemente ao outro. Contudo, os comandos internos do laço são todos executados para só depois a condição ser testada.

Teste de mesa

O teste de mesa é uma técnica de execução manual das linhas de um código com o objetivo de validar este último. Nesse processo, é construída uma tabela em que as colunas são compostas pelas variáveis do código. Também é possível acrescentar uma coluna com as impressões realizadas em tela.

Utilizando o código anterior, execute manualmente o código a partir da declaração de variáveis. À medida que as variáveis forem recebendo valores, estes devem ser repassados para a tabela demonstrada a seguir.

Quando o código trouxer entrada de dados (**Console.ReadLine**), escolha um valor para entrar no teste. No exemplo, será pedido que cinco valores de salários sejam informados.

salario	soma	media	cont	impressão
1200	0	3340	1	Inf. Salário:
2500	1200		2	Inf. Salário:
5000	3700		3	Inf. Salário:
3200	8700		4	Inf. Salário:
4800	11900		5	Inf. Salário:
	16700			Valor Médio: 3340

Faça o teste de mesa anterior acompanhando o código do exemplo 3 sem executá-lo em máquina. Anote os valores de entrada em um papel e confira se, executando mentalmente as instruções do código, você obtém os mesmos valores de `soma`, `media` e `cont`. Depois, teste rodando o programa e inclua **Console.WriteLine()** em cada iteração do laço para mostrar os valores das variáveis. Por fim, verifique se os valores obtidos são os mesmos que você descobriu nos testes de mesa

A partir de um número inteiro informado, diga se ele é um número primo ou não. Número primo é aquele que, em uma divisão de inteiros, resulta em resto zero quando dividido apenas por um e por ele mesmo.

Exemplo 8 – Número primo (ou não)

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            int num, cont,resto,contZero=0;
            Console.WriteLine("Informe um número inteiro:");
            num = int.Parse(Console.ReadLine());
           cont = num;
            while(cont<0)
            {
               resto = num % cont;
               if(resto==0)
                 contZero++;
               cont--;
            }
            if(contZero>2)
                Console.WriteLine("O número {0} não é primo."
,num);
            else
                Console.WriteLine("O número {0} é primo.",nu
m);
        }
    }
}
```

Como visto, o código lê um número e depois armazena o mesmo número em uma variável chamada **cont**. O número informado será dividido pela variável **cont**. Se o resto da divisão for igual a zero, um contador chamado **contZero** será incrementado. A cada iteração no laço *while*, a variável **cont** é decrementada. O objetivo é que ela seja atualizada pelos antecessores desse número.

Fazendo o teste de mesa do código anterior com “num” igual a 7, obtém-se o seguinte:

num	cont	resto	contZero	impressão
7	7	0	0	O número 7 é primo.
	6	1	1	
	5	2	2	
	4	3		
	3	1		
	2	1		
	1	0		

Agora, fazendo o teste de mesa do código anterior com “num” igual a 6, obtém-se o seguinte:

num	cont	resto	contZero	impressão
6	6	0	0	O número 6 não é primo.
	5	1	1	
	4	2	2	
	3	0	3	
	2	0	4	
	1	0		

Quer treinar mais um pouco?

Então, faça o teste de mesa para o código a seguir:

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            double i=5, n=5,n2=-1;

            while(n2!=5)
            {
               n2 = Math.Pow(n,i);
               Console.WriteLine(" {0} ",n2);
               if(n2<500)
                 Console.WriteLine(" X \n");
               else
                 Console.WriteLine(" Y \n");

               i--;
               n++;

               if(i==0)
                 n2 = 5;
            }
        }
    }
}
```

Estrutura de dados: *array* (sintaxe, declaração e uso)

Estruturas de dados são arranjos que armazenam e mantêm dados para facilitar o acesso a tais dados e a manipulação destes. Variáveis são legais e úteis, mas armazenam apenas um valor, o que, algumas vezes, pode não ser suficiente.

Imagine que você está desenvolvendo um programa em que precisa listar os contatos de uma agenda. Com os recursos estudados até agora, uma das soluções pode ser esta:

```
string contato1 = "Fulano da Silva";
string contato2 = "Cicrano de Souza";
string contato3 = "Beltrano Sá";
```

Não é difícil notar que a solução é extremamente inconveniente, basicamente por três motivos:

1. Só é possível utilizar três contatos no *software*.
2. Se o desejo for adicionar mais contatos, será necessário programar uma nova variável, ou seja, abrir o código apenas para incluir um contato novo.
3. Se o desejo for mostrar todos os contatos, necessariamente será preciso fazer algo como:

```
Console.WriteLine(contato1);
Console.WriteLine(contato2);
Console.WriteLine(contato3);
```

Sendo assim, impede-se o uso de um laço, já que a cada momento é utilizada uma variável diferente. O *software*, em suma, fica impraticável.

É hora de pensar em estruturas de dados. A mais básica de todas elas é o *array*, ou vetor.

O *array* é uma estrutura de dados que agrupa diferentes itens do mesmo tipo de dado. Em vez de declarar diversas variáveis para armazenar o nome de cada contato de uma turma, pode-se criar um *array* que armazene vários valores de uma vez e seja capaz de modificar cada um desses dados.

A sintaxe de declaração básica de um vetor é esta:

```
tipo[] identificador;
```

O sinal **[ ]** denota um *array* e será usado para acessar cada valor armazenado no vetor. Um exemplo de declaração completa de um vetor para armazenar os contatos do *software* ficaria assim:

```
string[] contatos = new string[3];
```

Como C# é uma linguagem orientada a objetos, vetores são considerados objetos. A palavra reservada “ `new` ” é utilizada para criar um *array* justamente por esse motivo. Com isso, também está sendo alocada (ou reservada) a memória necessária para o *array* (no exemplo, espaço para três valores do tipo *string*).

Dessa forma, está sendo declarado um vetor de três posições, e cada posição do vetor pode receber um valor do tipo `string`. Pode-se pensar o vetor como um conjunto de variáveis coladas, mas que são referenciadas por um único nome.

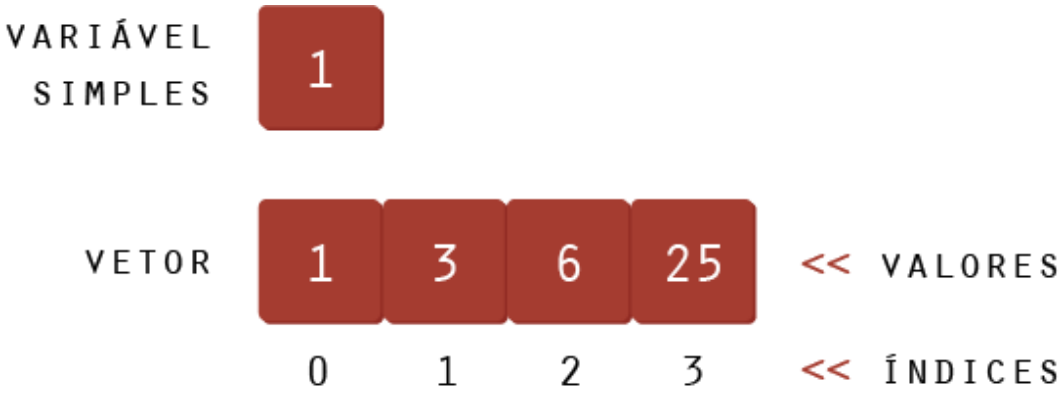

Figura 1 – Representação de uma variável em que apenas um valor pode ser armazenado por vez e de um vetor em que vários valores são armazenados (cada valor é referenciado por um índice)

Quanto ao tamanho, *arrays* são entidades estáticas, isto é, o valor informado quando eles são declarados indica uma quantidade de posições que permanecerá até o fim da aplicação. Entretanto, os valores de cada espaço podem ser modificados, trocados e excluídos livremente.

Na prática, um *array* é um grupo de posições de memória adjacentes que tem o mesmo identificador (nome) e o mesmo tipo. Para se referir a um elemento específico dentro do *array*, utilizam-se o nome dele (que é o mesmo para todos os elementos) e o número da posição (que é um valor que identifica uma das posições internas do *array*).

```
contatos[0] = "Fulano da Silva";
```

Lê-se que o vetor “contatos” na posição zero recebe o valor “Fulano da Silva”.

Em C#, como em várias outras linguagens de programação, o primeiro elemento de qualquer vetor é referenciado com o índice zero. Por isso, no exemplo anterior, atribuiu-se valor para `contatos[0]` em vez de para `contatos[1]`. O último elemento do *array* “contatos” declarado anteriormente (de três posições) seria referenciado como `contatos[3]`.

A inicialização das posições do vetor “contatos” ficaria assim:

```
contato[0] = "Fulano da Silva";
contato[1] = "Cicrano de Souza";
contato[2] = "Beltrano Sá";
```

A leitura de um vetor também segue o mesmo preceito de utilizar um índice para referenciar um dos valores armazenados:

```
Console.WriteLine("Nome do contato: {0}", contatos[1]);
```

A resposta do código citado seria “Nome do contato: Cicrano de Souza”.

Ainda há muito conteúdo a ser abordado, mas o que você acha de fazer um desafio agora? Caso se sinta preparado, faça o exercício a seguir:

Desenvolva um programa que declare o vetor “contatos”, que inicialize e que mostre os elementos na tela. Depois, siga lendo o conteúdo disponível neste material.

Uma das grandes vantagens dos vetores é a sua flexibilidade com relação aos índices, que podem ser números fixos ou estar em variáveis. Isso torna possível, inclusive, declarar um vetor com um valor informado pelo usuário, por exemplo.

```
int tamanhoArray = int.Parse(Console.ReadLine());
int meuVetor[] = new int[tamanhoArray];
```

Cabe ressaltar que o índice de um vetor sempre será um número inteiro. Para ler ou escrever valores em um vetor, pode-se usar o seguinte código:

```
int i = 0;
Console.WriteLine("Nome do contato: {0}", contatos[i]); //Ful
ano da Silva
i++; //i == 1
Console.WriteLine("Nome do contato: {0}", contatos[i]); //Cic
rano de Souza
contatos[i] = "Maria da Silva";
Console.WriteLine("Nome do contato: {0}", contatos[1]); //Mar
ia da Silva
```

A grande vantagem, portanto, é a capacidade de iterar, de usar laços para percorrer o vetor. Para tanto, o laço *for* é ideal, já que traz consigo uma variável de controle autoincrementada.

Exemplo 9 – Programa em C# que armazena nomes de contatos e depois mostra o nome de cada um dos contatos

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] contatos = new string[5];

            contatos[0] = "Maria";
            contatos[1] = "João";
            contatos[2] = "Arthur";
            contatos[3] = "Fernando";
            contatos[4] = "Júlia";

            for(int i = 0; i < contatos.Length; i++)
            {
                Console.WriteLine("Contato " + (i+1) + ": " +
contatos[i]);
            }
        }
    }
}
```

Veja que no exemplo anterior `contatos.Length` foi utilizado para referenciar o tamanho do vetor.

Preste muita atenção ao índice! Se você tentar acessar um valor de índice acima do tamanho definido para o vetor, a aplicação lançará um erro do tipo “IndexOutOfRangeException”, com uma mensagem informando que o índice

usado estava fora dos limites do *array*.

A instrução a seguir geraria esse erro:

```
                contatos[5] = "Outro nome"; //o tamanho de contatos é 5; o últi
mo índice é 4
```

O seguinte *for* também geraria esse erro por uma razão mais sutil: *for* acontece apenas porque o laço não para antes de **i** ficar igual a 5. Um simples **<=** pode causar problemas:

```
                for(int i = 0; i <= contatos.Length; i++)
                {
                    Console.WriteLine("Contato " + (i+1) + ": " + contatos
[i]);
                }
```

Exemplo 10 – Programa em C# em que o usuário informa nomes e notas de alunos de uma disciplina, calcula a média da classe e mostra quais foram as notas acima da média

O usuário, no início, informa quantas notas ele digitará:

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] notas;
            string[] nomes;
            double media;
            int quantidade;

            Console.Write("Informe quantos alunos: ");
            quantidade = int.Parse(Console.ReadLine());

            notas = new double[quantidade];
            nomes = new string[quantidade];
            media = 0;

            for(int i = 0; i < quantidade; i++){
                Console.Write("Digite o nome do aluno: ");
                nomes[i] = Console.ReadLine();
                Console.Write("Digite a nota do aluno: ");
                notas[i] = double.Parse(Console.ReadLine());
                //vírgula separa decimal p/ o double.Parse

                media = media + notas[i];
            }

            media = media / quantidade;
            Console.WriteLine("Média: {0}", media);

            for(int i = 0; i < quantidade; i++)
            {
                if(notas[i] >= media)
                {
                    Console.WriteLine("Aluno {0} acima da méd
ia. Nota {1}",
                                        nomes[i], notas[i]);
                }
            }

            Console.WriteLine("Fim do Programa");
        }
    }
}
```

O problema do exemplo anterior não poderia ser resolvido sem a utilização de um *array*, já que é necessário passar por todas as informações, calcular a média e depois repassar por todas elas para descobrir os alunos que estão com notas acima da média.

Outra maneira de inicializar um vetor é, já na declaração, informar os valores que ele recebe. Por exemplo:

```
int [] numeros = {2, 3, 5, 7, 11};
```

O que você acha de fazer mais alguns exercícios?

Caso se sinta preparado, faça estes exercícios:

1. Faça um algoritmo que some o conteúdo de dois vetores e armazene o resultado em um terceiro vetor.

2. Analise um vetor de dez posições e atribua o valor 0 para todos os elementos que contiverem valores negativos.

3. Analise um vetor de dez posições e verifique se existem valores iguais e os escreva.

Estrutura de dados: *struct* (sintaxe, declaração e uso)

Como visto, no exemplo 10 foi necessário usar dois vetores para dar conta de basicamente uma informação: um aluno que tem nome e nota. Há uma estrutura de dados que surge um pouco antes da orientação a objetos. Tal estrutura é chamada no C# de *struct* (ou registro, em algoritmos) e consegue agrupar informações sobre uma mesma entidade.

A sintaxe de declaração do *struct* é a seguinte:

```
<modificador de acesso> struct <nome>
{
  <atributos>
}
```

Nela, cada atributo é definido por:

```
<modificador de acesso> <tipo> <nome>;
```

No exemplo “Aluno”, pode-se definir a estrutura desta forma:

```
struct Aluno
{
    public string nome;
    public double nota;
}
```

Nome e nota são os atributos, e o acesso a eles ocorre usando **“.”** . Veja no exemplo a seguir:

```
Aluno a = new Aluno();
a.nome = "Fulano";
a.nota = 9.75;
Console.WriteLine("Aluno: " + a.nome + "; nota: " + a.nota);
```

Diferentemente de outras linguagens, em C# o funcionamento e a sintaxe do *struct* são muito próximos das classes. De fato, *struct* é considerado uma versão leve das classes e é útil quando, por exemplo, você precisa criar uma matriz grande de objetos e não quer consumir muita memória para tanto. Apesar de compartilhar boa parte da sintaxe das classes, os *structs* são mais limitados.

Exemplo 11 – Programa em C# que calcula uma média de notas entre os alunos informados e mostra aqueles que estão acima da média, utilizando *struct*

```
using System;

namespace teste
{
    struct Aluno
    {
        public string nome;
        public double nota;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Aluno[] alunos;
            double media;
            int quantidade;

            Console.Write("Informe quantos alunos: ");
            quantidade = int.Parse(Console.ReadLine());

            alunos = new Aluno[quantidade];
            media = 0;

            for(int i = 0; i < quantidade; i++){
                alunos[i] = new Aluno();
                Console.Write("Digite o nome do aluno: ");
                alunos[i].nome = Console.ReadLine();
                Console.Write("Digite a nota do aluno: ");
                alunos[i].nota = double.Parse(Console.ReadLin
e());

                media = media + alunos[i].nota;
            }

            media = media / quantidade;
            Console.WriteLine("Média: {0}", media);

            for(int i = 0; i < quantidade; i++)
            {
                if(alunos[i].nota >= media)
                {
                    Console.WriteLine("Aluno {0} acima da méd
ia. Nota {1}",
                                      alunos[i].nome, aluno
s[i].nota);
                }
            }

            Console.WriteLine("Fim do Programa");
```

```
                }
            }
        }
```

Veja no exemplo que o tipo *struct* foi declarado fora da classe **Program** e que, no método **Main()**, criou-se apenas um vetor que contém elementos do tipo “Aluno”. Cada aluno terá consigo um nome e uma nota que podem ser escritos ou lidos.

Os *structs* são usados para criar estruturas de dados cujos objetos são pequenos (no máximo 16 *bytes*) e imutáveis, com um valor que seja único (tenha poucas características) e que não precise ser encapsulado em objetos por referência com frequência. *Structs* são muito úteis (podem otimizar códigos e dar a semântica correta), mas não absolutamente necessários. Na maioria das vezes, você acabará usando classes e objetos.

Métodos (funções): definição, sintaxe, operadores e palavras reservadas

Algumas vezes, quer-se simplificar alguma rotina repetitiva sem precisar reescrever trechos de código. Uma maneira de alcançar isso é usando laços. Contudo, nem sempre essa rotina está envolvida em um processo iterativo.

Imagine que é possível usar algoritmos prontos que tragam funcionalidades para o código que está sendo elaborado sem precisar codificar esse comportamento. Na verdade, não é nem preciso imaginar, pois isso já está sendo feito neste material. Basta pensar em quantas vezes foi utilizado `Console.WriteLine()` em nossos programas e em quantas vezes foi preciso implementar como se utiliza letra por letra de um texto informado, como se identifica a saída de dados, como se envia a informação para essa saída, ou seja, quantas vezes foi necessário implementar a rotina do WriteLine. A rotina já veio pronta para ser usada. WriteLine é um método.

Um método é um encapsulamento de uma computação definida por uma sequência de instruções. Em outras palavras, é um algoritmo autocontido. Também chamado de função ou procedimento, o método precisa ser declarado e ter toda a sua rotina definida para depois ser chamado por outros métodos no programa.

“Funções” e “procedimentos” são termos próprios da programação estruturada (PE). Na programação orientada a objetos (POO), o termo geral é “método”.

A declaração de um método segue esta sintaxe:

```
                [<modificador de acesso>] <tipo de retorno> <nome> (<parametr
os>)
                {
                    <código>
                }
```

Ademais, alguns pontos sobre o exemplo merecem ser destacados:

- O modificador de acesso pode ser *public*, *private*, *protected*, entre outros.
- O tipo de retorno indica que tipo de informação o método pode devolver ao código que o chamou.
- “Nome” é como o método será chamado em nosso código.
- Parâmetros são as informações de entrada que o método pode usar em seu processamento.
- Código é uma sequência de comandos que pode incluir desvios, laços, criação de variáveis etc.

O exemplo a seguir é a declaração mais simples de um método, a qual apenas mostra na tela uma frase pronta:

```
void EscreveAlgo()
{
    Console.WriteLine("Olá, eu sou um método");
}
```

Novamente, mais alguns pontos merecem ser destacados:

- *Void* indica que nenhum valor será gerado pelo método.
- “EscreveAlgo” é o nome do método. No código, o método pode ser chamado digitando `EscreveAlgo();`.
- Os parênteses indicam parâmetros. Nesse caso, não foi usado nenhum parâmetro de entrada para o método.
- Entre **{** e **}**, estão todas as instruções que serão executadas.
- O método do exemplo apresenta a frase “Olá, eu sou um método” no terminal.

Para os projetos do tipo console (como os citados neste material), deve-se fazer uma pequena adaptação: antes de `void`, utiliza-se `static`. Esse modificador será explicado nos materiais das próximas unidades. Por ora, cabe

dizer apenas que ele se trata de uma construção para projetos de console.

Exemplo 12 – Declaração e uso de **EscreveAlgo**

```
using System;

namespace teste
{
    class Program
    { //início do escopo de Program
        static void Main(string[] args)
        {
           EscreveAlgo();
        }

        static void EscreveAlgo()
        {
            Console.WriteLine("Olá, eu sou um método");
        }
    }//fim do escopo de Program
}
```

Um ponto importante a ser notado no exemplo anterior é o posicionamento da declaração do método, o qual precisa ficar entre as chaves que delimitam a classe **Program** e não pode estar dentro do método `Main()`, ou seja, entre as chaves que delimitam o escopo desse método.

Exemplo 13 – Programa em C# que implementa um método que mostra mensagem de “bom dia” ou “boa noite” para o usuário de acordo com o horário atual

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
           string nome = Console.ReadLine();
           Console.WriteLine($"Olá {nome}");
           BoasVindas();
        }

        static void BoasVindas()
        {
            if(DateTime.Now.Hour >= 5 && DateTime.Now.Hour <
18)
            {
                Console.WriteLine("Tenha um bom dia");
            }
            else
            {
                Console.WriteLine("Tenha uma boa noite!");
            }
        }
    }
}
```

Clique ou toque para visualizar o conteúdo.

(#modal-passagem-parametros) (#modal-retorno-valores)

Passagem de parâmetros

Um método sem uma entrada de dados geralmente é menos interessante que um método que receba informações para serem processadas. Para tanto, utilizam-se parâmetros.

Pode-se estabelecer que um método receba zero, um ou mais parâmetros e que cada um deles deva definir um tipo e um nome.

Veja a seguir uma adaptação do exemplo anterior para que o método `BoasVindas` receba um parâmetro com o nome do usuário:

```
                        static void BoasVindas(string usuario)
                        {
                            if(DateTime.Now.Hour >= 5 && DateTime.Now.
Hour < 18)
                            {
                                Console.WriteLine($"Tenha um bom dia
{usuario}");
                            }
                            else
                            {
                                Console.WriteLine($"Tenha uma boa noit
e {usuario}");
                            }
                        }
```

Observe que, apenas com tal mudança, um erro acontece na linha de chamada do método `BoasVindas` com uma mensagem semelhante a esta:

```
Não há nenhum argumento fornecido que corresponde ao parâmetro formal
 necessário "usuario" de "Program.BoasVindas(string)"
```

Isso indica que aquela chamada `BoasVindas();` em **Main()** já não é mais válida. De fato, agora é preciso informar um valor para o parâmetro definido pelo método.

A chamada pode ser adaptada para este código:

```
static void Main(string[] args)
{
   string nome = Console.ReadLine();
   Console.WriteLine($"Olá {nome}");
   BoasVindas(nome);
}
```

Como percebido, o valor da variável “nome” será repassado ao método `BoasVindas` , que o receberá por meio do parâmetro “usuario".

Ainda, pode ser informado mais de um parâmetro, como mostra o exemplo a seguir:

Exemplo 14 – Método “somar” recebe dois valores e mostra o resultado da soma na tela

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            Somar(62, 24);
        }

        static void Somar(int num1, int num2)
        {
            int soma;
            soma = num1 + num2;
            Console.WriteLine("Resultado da so
ma" + soma);
        }
    }
}
```

Todos os valores de parâmetros devem ser informados na hora de chamar um método.

Retorno de valores

Considere o exemplo do método anterior e note que a utilidade deste é muito limitada: ele faz a soma, mostra-a na tela e finaliza o processamento. Se o intuito fosse, por exemplo, usar o método para fazer a soma em uma média, não existiria nenhuma possibilidade, já que o valor que ele gera é mostrado em tela e, ao fim, descartado.

Pode-se, no entanto, adaptar o método para que passe a **retornar um valor**:

```
static int Somar(int num1, int num2)
{
    int soma;
    soma = num1 + num2;
    return soma;
}
```

Observe duas considerações sobre o código mencionado:

1. Em vez de *void*, utilizou-se *int*. Esse será o tipo de informação que será retornado pelo método.
2. No fim do método, há uma cláusula com a palavra reservada `return` . Essa cláusula define o retorno do método e a entrega do valor da variável local “soma” ao trecho do programa que chamou o método.

Como então utilizar o valor que “somar” entrega? A resposta é mais simples do que parece: com uma atribuição ou diretamente em expressões.

```
                    static void Main(string[] args)
                    {
                            int soma = Somar(10, 7); //atribuindo à variá
vel soma o retorno de Somar
                            double media1 = soma / 2.0;
                            Console.WriteLine(media1); //imprime 8.5

                            //usando direto na expressão o retorno de Som
ar
                            double media2 = Somar(6, 9) / 2.0;
                            Console.WriteLine(media2); //imprime 7.5
                    }
```

Indicado um tipo de retorno, é obrigatório que o método finalize com uma cláusula *return*.

Na PE, é comum chamar de **função** a rotina que realiza um retorno de dados, assim como é comum chamar de **procedimento** uma rotina que não devolve informação.

Como dito, por estar sendo abordada uma linguagem orientada a objetos, pode-se usar simplesmente o termo “método” para as duas denominações, sendo que os métodos sem retorno sempre apresentam *void* (vazio) no lugar do tipo de retorno na assinatura do método.

O nome “retorno” não é à toa. Quando as instruções do programa são executadas, se há uma chamada a um método, ocorre um redirecionamento para o trecho onde está definido o código do método, ou seja, há um desvio. Ao fim, o fluxo de execução volta, retornando ao fluxo anterior.

Exemplo 15 – Programa em C# que, dados os catetos de um triângulo retângulo, calcula a hipotenusa

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            double cateto1, cateto2, hipotenusa;
            string continua;

            do{
                Console.Write("Informe cateto oposto:
");
                cateto1 = double.Parse(Console.ReadLi
ne());
                Console.Write("Informe cateto adjacen
te: ");
                cateto2 = double.Parse(Console.ReadLi
ne());

                hipotenusa = Math.Sqrt(Quadrado(catet
o1) + Quadrado(cateto2));
                Console.WriteLine("Hipotenusa: " + hi
potenusa);

                Console.Write("Deseja continuar? [s/
n] ");
                continua = Console.ReadLine();
            }while(continua != "n");
        }

        static double Quadrado(double numero)
        {
            double quadrado = numero * numero;
            return quadrado;
        }
    }
}
```

Veja no exemplo que foi criada a função “quadrado” para calcular a potência de dois de um número. Para fazer a raiz quadrada, usa-se um método pronto da classe **Math – Math.Sqrt()**.

Exemplo 16 – Programa em C# que calcula a área de um retângulo e de um triângulo

```
using System;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            double retangulo = CalculaAreaRetangulo(1
0, 5);
            Console.WriteLine($"Área do retângulo: {r
etangulo}");

            double triangulo = CalculaAreaTriangulo(3
, 4);
            Console.WriteLine($"Área do triângulo: {t
riangulo}");
        }

        static double CalculaAreaRetangulo(double b,
double h)
        {
            double area = b * h;
            return area;
        }

        static double CalculaAreaTriangulo(double b,
double h)
        {
            double area = CalculaAreaRetangulo(b, h);
            area = area / 2;
            return area;
        }
    }
}
```

No exemplo anterior, fica evidente que um método pode ser chamado por outro. Na verdade, isso já era feito com frequência, pois **Main()** também é um método da classe **Program**.

Após essas experiências, podem-se então entender algumas das vantagens da modularização do código por meio da criação de métodos:

Métodos tornam o algoritmo mais fácil de escrever, pois o desenvolvedo não precisa saber todos os detalhes de tudo no código (o profissional pod simplesmente conhecer para que serve o método e chamá-lo).

O código fica mais fácil de ler. Nomes significativos para os método também ajudam.

A criação de métodos eleva o nível de abstração, tornando mais fác entender o que o algoritmo realiza, além de economizar tempo e esforço d desenvolvedor, pois assim pode reutilizar uma função já escrita anteriormente.

Agora que você já leu mais informações sobre métodos, que tal criar alguns?

Caso se sinta à vontade, faça estes exercícios:

1. Crie um programa que contenha um método para realizar o cálculo da área de um círculo. A área é definida pela fórmula **A = π * R²**. O usuário informará o valor do raio (R). Considere **π** igual a 3,14.

2. Crie um programa que defina um vetor de números inteiros e um método para encontrar o menor valor nesse vetor. O vetor precisará ser passado por parâmetro. O programa deve mostrar o menor valor na tela.

3. Em um projeto em C#, programe um método que receba por parâmetro um valor inteiro e positivo e retorne o valor *bool* "*true*", se o valor for primo, e "*false*", caso contrário.

Programações estruturada e orientada a objetos: diferenças

Antes de entrar definitivamente nos conceitos de orientação a objetos, é preciso refletir sobre os dois maiores paradigmas em programação.

Na comunicação, utilizam-se falas, gestos, sinais etc., ou seja, modos diferentes para passar a mensagem desejada. Portanto, **na programação, uma linguagem de programação é o meio de comunicação entre a pessoa e a máquina.** Cada linguagem tem sua estrutura para gerar a comunicação com a máquina.

Um paradigma é uma forma de resolver problemas, um padrão. Como na maioria das coisas dentro do universo da programação, não existe opção certa ou errada, mas, sim, a melhor opção para o projeto. Como programador, você deve desenvolver a capacidade de decidir o paradigma adequado para resolver a problemática do seu projeto atual.

Como visto, a área de desenvolvimento de *software* é muito ampla e diversificada. Existem muitas linguagens de programação que seguem diferentes paradigmas. Os paradigmas, nesse contexto, são definidos pelo que permitem ou não permitem que uma linguagem realize.

Dois dos principais paradigmas são a programação estruturada (PE) e a programação orientada a objetos (POO).

Na maioria das vezes, os estudos em programação são iniciados com linguagens estruturadas, devido ao fato de ser um período de desenvolvimento da lógica computacional. A PE ajuda nesse processo por facilitar o controle do fluxo do código, além de a sua estrutura ser mais simples e fácil de aprender.

A PE tem como objetivo construir programas claros, legíveis, eficientes e de fácil manutenção. É uma forma de escrever códigos sem encapsular dados, sem utilizar restrições de acesso. Não há organização em camadas. Então, todos os

códigos estão, muitas vezes, no mesmo arquivo.

A PE preza pela criação de estruturas simples nos programas, o que a torna mais rígida e restrita (pode utilizar apenas três estruturas: sequência, decisão e iteração [ou repetição]). Com os três tipos de estrutura de controle, é possível construir programas sem usar desvios, ou seja, o código é muito linear.

O C# é um ótimo exemplo de linguagem orientada a objetos (atualmente é a metodologia mais amplamente utilizada). O padrão orientado a objetos se relaciona muito às dificuldades modernas da programação, quais sejam sistemas cada vez mais complexos.

No paradigma orientado a objetos, o problema é dividido em várias partes, a ponto de serem definidos objetos/entidades, visando a aproximar o desenvolvimento de *software* com o mundo real.

A orientação a objetos é uma forma muito eficiente de reaproveitar códigos, pois todos os métodos criados têm uma finalidade e podem ser chamados sempre que necessário.

Além disso, a POO tem quatro pilares: abstração (que refere-se a classes e objetos), encapsulamento (que garante a proteção dos elementos do programa), herança (que otimiza a produção da aplicação em tempo e linhas de código) e polimorfismo (que liga-se à herança e possibilita alterar o funcionamento interno de um método herdado de um objeto pai).

Na PE, os procedimentos são aplicados em toda a aplicação. Já na POO, existem métodos que são aplicados aos dados de cada objeto.

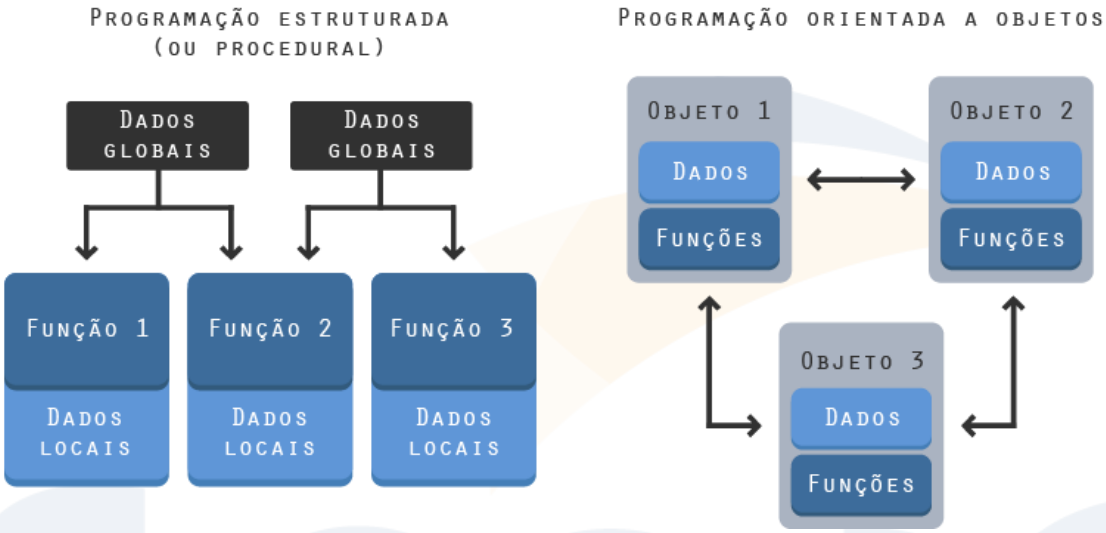

Figura 2 – Comparação entre programação estruturada e programação orientada a objetos
Fonte: adaptado de <https://simplesnippets.tech/java-introduction-to-object-oriented-programming-oop/<.

As funções (ou procedimentos) na PE são independentes, comunicando-se apenas por meio de variáveis globais (acessíveis a todas as funções). Já na POO, os objetos são independentes e se comunicam entre si por meio de chamadas às funções (comportamentos) de cada um.

A PE apresenta um desempenho superior ao da POO. Isso porque, como a estrutura do código é sequencial (cada linha de código é executada após a outra), não existem grandes desvios como os que ocorrem na POO.

Na parte da manutenção de código, a POO segue um padrão que torna mais simples encontrar problemas nos códigos, facilitando também a compreensão por parte dos programadores. Já na PE, comentários devem ter sido feitos pelo programador que criou a aplicação para um melhor entendimento. Contudo, a PE tem um melhor controle do fluxo de código, já que a POO tem conceitos mais complexos que podem dificultar a dissecação do código.

Tanto a orientação a objetos quanto a estrutura permitem reaproveitar o código, mas com a POO podem-se herdar características, relacionar os objetos e reaproveitar o código sem a necessidade de reescrevê-lo de fato. Já a PE é mais limitada e, muitas vezes, inclui duplicar o código e colocá-lo em outra parte do projeto.

Na parte de execução, o código em PE é voltado a resolver um problema específico. Já o código em POO tende a ser mais abrangente para que a mesma solução possa ser utilizada no futuro com a mesma eficiência. Portanto, o problema é que a PE é voltada para como a tarefa deve ser feita e não para o que deve ser feito.

Ambos os paradigmas são muito úteis, mas a POO traz outros pontos que acabam sendo mais interessantes no contexto de aplicações modernas. Sendo assim, os aspectos da orientação a objetos no C# serão o foco de estudo.

Clique ou toque para visualizar o conteúdo.

(#modal-orientacao-objetos) (#modal-classes-objetos)

Orientação a objetos

A POO é uma abordagem de desenvolvimento de *software* em que o problema é dividido em várias partes, a ponto de serem definidos objetos/entidades. As tarefas do software são cumpridas por meio das interações entre os objetos, que trocam mensagens entre si e executam comportamentos próprios.

Pense na rotina que você faz todo dia. Nela, já é utilizado o paradigma da orientação a objetos. Se você precisa ir a uma loja, por exemplo, utilizando um carro para o transporte, interagirá com o objeto “carro”, mandando-lhe mensagens o tempo todo para que execute ações que ele conhece (ligar o carro, virar o volante, acelerar). Ainda, usando um editor de texto, você pode enviar um comando para que ele imprima uma página.

Em ambas as situações, você (objeto “pessoa”) não precisa saber detalhes de como as funções dos outros objetos são executadas (a aceleração do objeto “carro”, por exemplo, ou o processo interno de enviar informação a uma impressora do objeto “editor”). Basta saber como comunicar ao objeto as ações que ele precisa executar.

Os exemplos deixam ainda mais claro como esse paradigma visa a aproximar o desenvolvimento de *software* do mundo real, sendo uma ponte entre o mundo real e o virtual. O objetivo é transferir os elementos do mundo real para os códigos, possibilitando construir sistemas complexos baseados em objetos.

O paradigma da orientação a objetos teve origem na década de 1960, com a linguagem de programação Simula. Porém, ele surgiu com mais notoriedade em 1970, com a linguagem Smalltalk. Em 1980 e 1990, o C++ e o Java popularizaram esse tipo de programação, e, em 2000, o .NET Framework o consolidou.

Algumas das habilidades da POO são:

➔ Manter e implementar alterações nos programas mais rápido eficientemente

➔ Facilitar o trabalho em equipe (separar e integrar tarefas)

➔ Reutilizar o código implementado anteriormente

Objetos

São as bases do paradigma. Os objetos são estruturas que carregam consigo informações e procedimentos (ou comportamentos) que manipulam tais informações. Na vida real, até nós mesmos podemos ser considerados objetos.

Abstração

Esconde detalhes do funcionamento de uma rotina ou de um objeto. A abstração permite que um objeto chame comportamentos de outro objeto sem necessariamente ter controle de como estes ocorrem. Para fazer um carro andar, por exemplo, você não precisa saber como funciona o motor.

Encapsulamento

Permite expor e disponibilizar ou esconder e proteger informações de um objeto. Como exemplo, pense em um setor de recursos humanos: ele só pode disponibilizar informações dos funcionários em determinadas circunstâncias (as informações estão protegidas do acesso geral).

Polimorfismo

Permite que dois objetos de um tipo tenham maneiras diferentes de executar um mesmo comportamento. Pense no método “falar”: um humano conversa, um cachorro late, um gato mia. Cada um tem um jeito de responder ao comportamento “falar”.

Herança

Propaga as características de um tipo de objeto a outro, que pode expandir com comportamentos e dados próprios. Ainda pensando em cachorros, você pode ter uma classificação geral de “cachorro” como um ser que tem quatro patas e late; abaixo, pode criar uma classificação por raças que são do tipo “cachorro”, mas com características próprias.

Agregação

Refere-se à capacidade de um objeto ser composto de outros objetos.

A linguagem C#, como qualquer outra linguagem orientada a objetos, tem processos próprios para implementar cada uma dessas características. Por ora, basta conhecê-las e relacioná-las com o mundo real.

Classes e objetos

Em um mundo orientado a objetos, tudo acaba focando em classes e objetos. Uma classe é uma especificação de uma categoria de objetos/tipos. Observe os exemplos a seguir:

São seres que fazem magia.

São pessoas que foram infectadas por um vírus e que, após a morte, voltam à vida com capacidade cerebral limitada e comem outras pessoas (principalmente o cérebro).

É um pacote de serviços de gerenciamento do dinheiro de uma pessoa realizado por um banco.

Logo, ao definir uma classe, devem-se ter em mente dois tipos distintos de informações. Seguindo no exemplo do mago, “mago” então é uma categoria.

Figura 3 – Três objetos diferentes da classe “mago”
Fonte: <https://www.disneyclips.com/images/fantasia.html> e <https://www.theguardian.com/film/2013/oct/07/peter-jackson-hobbit-production-costs>.

Para conseguir definir o que é um mago, duas perguntas devem ser feitas:

O que os magos têm?

Os magos têm características que conferem a eles a classificação de magos.

O que os magos fazem?

Os magos praticam ações que conferem a eles a classificação de magos.

As características de um mago, então, são: ter um nome, uma quantidade de magia, um valor de força e uma escola de magia, utilizar poderes e se defender de ataques.

Agora, com a definição, pode-se considerar que personagens como Mickey, Gandalf e Harry Potter são todos magos.

Uma classe é uma unidade do sistema. Nela, estão definidos atributos e métodos, que são respectivamente as informações que uma classe pode armazenar e as ações que ela pode desempenhar. Quando a estrutura de um elemento é descrita, cria-se uma especificação básica do que todo elemento daquele tipo deve ter, a qual representa (modela) uma entidade que pode ser do mundo real ou uma abstração de um conceito.

A classe define uma categoria (tipo de dados) que será utilizada para instanciar os objetos, os quais demonstram o mesmo comportamento:

```
class Mago
{
}
```

A diferença fundamental entre classe e objeto está no fato de a classe conter as definições do que essa nova unidade fará e o objeto ser um caso especial de uma classe. É importante saber que, enquanto existir apenas uma definição de classe, podem existir diversos objetos baseados em uma classe.

Classe “mago”
Características
Ações

Um programa pode criar vários objetos da mesma classe. Objetos também são chamados de instâncias e podem ser armazenados em uma variável, uma matriz etc.

Um objeto é uma variável existente na memória do computador e apresenta um estado – valores que os atributos do objeto têm ao longo do ciclo de vida do objeto.

Os objetos de uma mesma classe podem ter valores diferentes nos dados que descrevem as suas características.

Sendo assim, Gandalf e Harry Potter são objetos da classe “magos”, mas têm valores de atributos totalmente diferentes. Um programa orientado a objetos consiste em vários objetos que interagem dinamicamente.

Gandalf (mago)
Nome: Gandalf
Poder: 100
Idade: 300
Harry (mago)
Nome: Harry Potter
Poder: 80
Idade: 39

Às vezes, os termos “classe” e “objeto” são utilizados de forma equivalente. Contudo, as classes descrevem o tipo dos objetos, enquanto os objetos são **instâncias** utilizáveis das classes. O ato de criar um objeto é chamado de **instanciar**.

Figura 4 – Exemplo ilustrativo de como funcionam classes e objetos

Veja na figura 4, à esquerda, a classe “carro”, que define as características e os comportamentos que qualquer carro terá; e, à direita, as instâncias de carro, cada um com um modelo (Polo, Uno e Fusca) e uma cor (verde, azul e vermelho).

Classes no C#: sintaxe, delimitadores e palavras reservadas

Para criar, então, a classe “mago” no C#, abra o Visual Studio Code e crie um novo projeto do tipo console. Para as próximas experiências, as extensões C# e C# Extensions no Visual Studio Code podem ser utilizadas.

Crie um novo arquivo clicando no ícone **NewFile** ao lado do nome da pasta na aba lateral **Explorer**. Após, nomeie o arquivo como **Mago.cs** e inclua o seguinte código:

```
namespace Mago
{
    public class Mago
    {

    }
}
```

A definição inclui:

- **Namespace**: separação lógica do C# que define um conjunto de classes.
- **Public**: modificador de acesso (veja mais informações sobre modificadores de acesso na seção **Encapsulamento**).
- **Class Mago**: definição da classe e do seu nome.

Todas as informações (atributos) e todos os comportamentos (métodos) precisarão ser declarados entre a chave de abertura da classe e a de fechamento.

Clique ou toque para visualizar o conteúdo.

Os atributos são, *grosso modo*, as variáveis da classe (os locais que armazenam as informações referentes àquela classe). Quando um objeto é criado a partir de uma classe, os atributos passam a ter informações específicas, ou seja, são conferidos valores a eles.

Os atributos representam os dados, e os valores daqueles definem o estado do objeto. Tais estados são únicos para cada objeto criado. Por exemplo, os magos Gandalf e Harry podem ter valor “100” em poder, pois teriam a mesma quantidade de força. Contudo, caso fosse modificado o atributo “100” no mago Harry para “80”, o de Gandalf não será afetado.

No C#, os atributos são declarados assim:

```
namespace Mago
{
    public class Mago
    {
        public string nome;
        public int idade;
        public int poder;

    }
}
```

Está-se definindo um atributo *string* chamado “nome”, um atributo “poder” (número inteiro) e um atributo “idade” (número inteiro).

Os comportamentos são definidos como métodos. Você lembra que foram criados métodos na classe **Program**? Aqui, os métodos estarão dentro de cada classe.

Apenas para relembrar, um método é um bloco de código que contém uma série de instruções. Basicamente, um método abrange as ações que o objeto é capaz de realizar, podendo agir por seus próprios atributos ou por meio de interações (envio de mensagem) com outros objetos.

Os métodos “atacar” e “defender” são definidos em nossa classe “mago”:

```
using System;

namespace Mago
{
    public class Mago
    {
        public string nome;
        public int idade;
        public int poder;

        public void Atacar()
        {
            Console.WriteLine("Mago está atacando!");
        }
        public void Defender()
        {
            Console.WriteLine("Mago está defendendo!");
        }
    }
}
```

É possível que, caso você não tenha incluído a cláusula “using System” no topo, uma mensagem de erro tenha surgido no editor com um tracejado vermelho embaixo de “Console”. O problema acontece porque o código não encontrou referência à classe “Console”, ou seja, não a conhece (passe o *mouse* sobre a palavra para ver a mensagem de erro). Observe ainda que aparece uma lâmpada amarela à esquerda. Ela contém algumas ações que resolverão o problema.

Clicando na lâmpada, vê-se uma caixa de menu de contexto, como nesta imagem:

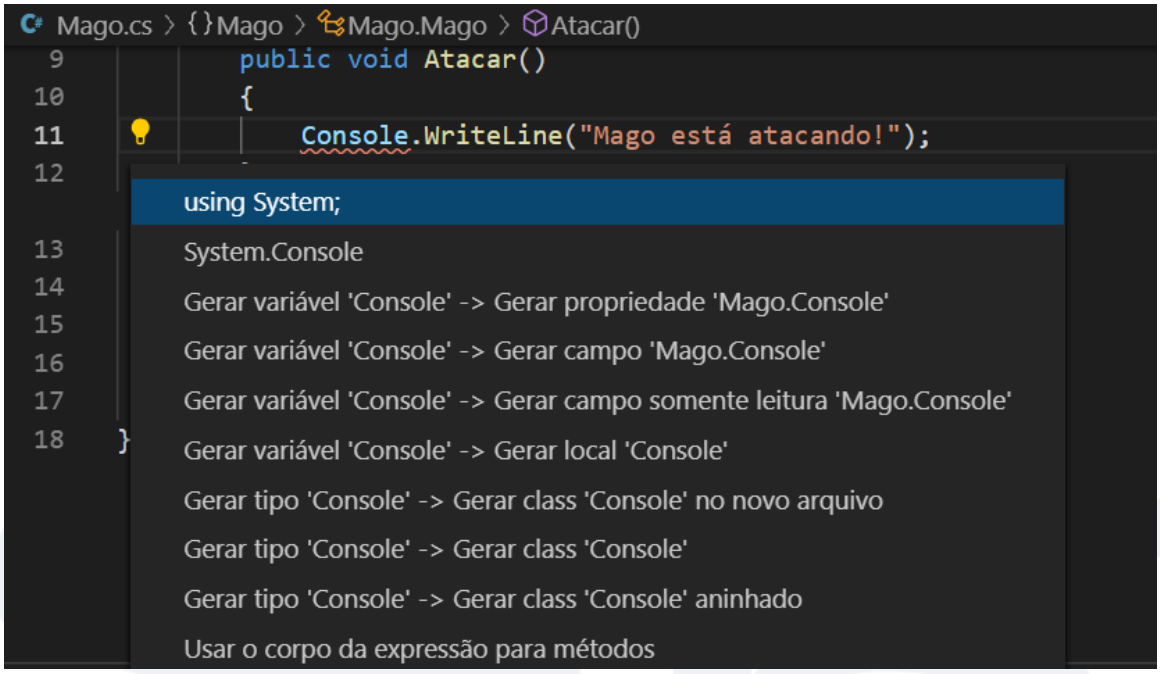

Figura 5 – Detalhe do arquivo **Mago.cs** no Visual Studio Code

A opção “using System;” é a mais adequada nesse caso, pois ela garantirá uma referência à classe “Console”. Clicando nessa opção, a cláusula “using System;” é incluída no topo do arquivo.

Os métodos **Atacar()** e **Defender()** foram definidos para os magos. No corpo do método, há apenas frases, mas obviamente poderia haver códigos mais elaborados. Uma característica importante é que os métodos podem usar os atributos de uma classe sem restrições.

Pode-se fazer então a seguinte alteração em **Atacar()**:

```
public void Atacar()
{
    Console.WriteLine("Mago está atacando!");
    Console.WriteLine($"Ele tem {poder} de poder");
}
```

Outra observação quanto às experiências anteriores é que não é preciso usar a palavra *“static”* na frente do tipo de retorno na assinatura do método. Métodos e classes estáticas serão abordados em outro momento do curso, mas,

em resumo, um método estático é chamado direto da classe. Métodos não estáticos, como o **Atacar()**, são chamados de um objeto.

A instanciação de objetos de uma classe segue esta sintaxe:

```
<Classe> <nome> = new <Classe>();
```

Semelhantemente à instanciação de um *struct* ou de um *array*, a criação de um objeto de mago acabaria assim:

```
Mago mago1 = new Mago();
```

O chamamento de métodos de um objeto ou o acesso a atributos deste ocorre usando o sinal de ponto:

```
mago1.nome = "Nome do Mago";//atributo nome do objeto mago1
mago1.Atacar();//método Atacar do objeto mago1
```

Agora, será utilizada a classe “mago” para criar instâncias dela. Em **Program.cs**, no método **Main()**,as seguintes alterações devem ser feitas:

```
using System;

namespace Mago
{
    class Program
    {
        static void Main(string[] args)
        {
            Mago gand = new Mago();
            Mago harry = new Mago();

            gand.nome = "Gandalf";
            gand.idade = 300;
            gand.poder = 100;

            harry.nome = "Harry Potter";
            harry.idade = 39;
            harry.poder = 80;

            gand.Atacar();
            harry.Defender();
            harry.Atacar();
             gand.Defender();
        }
    }
}
```

As classes **Program** e **Mago** estão sob o mesmo *namespace*. Portanto, elas se “conhecem”. Caso não estivessem, seria necessário incluir uma cláusula `using` no topo do arquivo.

A classe **Conta** guarda as informações de número, dígito verificador, saldo e titular. Construiu-se uma aplicação do tipo console e criou-se um arquivo **Conta.cs** no diretório do projeto, incluindo a seguinte declaração:

Exemplo 17 – Programa em C# usando classes para representar uma aplicação bancária

```
namespace Bancario
{
    public class Conta
    {
        public int numero;
        public int digitoVerificador;
        public double saldo;
        public string titular;

        public bool RelizarSaque(double valor){
            if(saldo  > valor)
            {
                saldo = saldo - valor;
                return true;
            }
            else
            {
                return false; //saldo insuficiente
            }
        }

        public void RealizarDeposito(double valor)
        {
            if(valor > 0)
            {
                saldo = saldo + valor;
            }
        }
    }
}
```

Em **Program.cs**, no método `Main()`, foi incluída a lógica do programa, que receberá uma entrada do usuário e manipulará a conta:

```
using System;

namespace Bancario
{
    class Program
    {
        static void Main(string[] args)
        {
            Conta contaBancaria = new Conta();
            string comando;
            double valor;

            Console.Write("Digite seu nome");
            contaBancaria.titular = Console.ReadLine();

            contaBancaria.numero = 1234;
            contaBancaria.digitoVerificador = 1;

            do
            {
                Console.Write("Digite a operação [d-depósito;
s-saque; x-sair] ");
                comando = Console.ReadLine();

                switch(comando)
                {
                    case "d":
                        Console.Write("Digite o valor a depos
itar: ");
                        valor = double.Parse(Console.ReadLine
());
                        contaBancaria.RealizarDeposito(valo
r);
                        break;

                    case "s":
                        Console.Write("Digite o valor a saca
r: ");
                        valor = double.Parse(Console.ReadLine
());
                        if(contaBancaria.RelizarSaque(valor)
 == false)
                        {
                            Console.WriteLine("Saldo insufici
ente");
                        }
                        break;

                    case "x":
                        Console.WriteLine("Encerrando a aplic
```

```
ação");
                                        break;

                                    default:
                                        Console.WriteLine("Opcao inválida");
                                        break;
                                }

                            }while(comando != "x");

                        }
                    }
                }
```

As classes já estão sendo trabalhadas. Então, o que você acha de aceitar um desafio?

Em caso afirmativo, faça os exercícios a seguir:

1. Modifique o exemplo anterior para criar uma nova opção “Consultar saldo”, a qual mostre o saldo atual da conta bancária.

2. Crie uma nova aplicação console. Para tanto, crie uma classe “Usuario” com as informações “Login” e “Senha”. No programa, cadastre três objetos “Usuario”, solicitando ao usuário do programa que ele digite *log in* e senha para cada objeto. Depois, mostre na tela todos os objetos “Usuario” criados. Você pode usar um vetor.

Construtores

Você já se perguntou por que os parênteses são incluídos logo depois do nome da classe, após a palavra reservada `new`, na instanciação de um objeto? Na verdade, um método especial de uma classe está sendo chamado, cujo nome dado a ele é “construtor”.

O construtor é um método que contém o mesmo nome da classe, que não retorna nenhum valor e que é chamado cada vez que um objeto da classe é criado. Quando nenhum construtor é criado, gera-se um construtor vazio, que não

recebe nenhum parâmetro e não executa nenhum código. É como declarar a classe da seguinte maneira:

```
using System;

namespace Mago
{
    public class Mago
    {
        public string nome;
        public int idade;
        public int poder;

        public Mago()
        {

        }

        public void Atacar()
        {
            Console.WriteLine("Mago está atacando!");
            Console.WriteLine($"Ele tem {poder} de poder");
        }
        public void Defender()
        {
            Console.WriteLine("Mago está defendendo!");
        }
    }
}
```

Observe o trecho que define o construtor:

```
public Mago()
{

}
```

Inclua uma instrução *write* e veja que a aplicação mostrará no terminal a mensagem na hora que acontece a criação dos objetos **Mago**:

```
public Mago()
{
    Console.WriteLine("Este é o construtor executando");
}
```

Os construtores servem, essencialmente, para incluir rotinas que precisam ser executadas no momento da criação do objeto. Também são úteis quando é preciso garantir que o objeto receba alguns valores iniciais aos seus atributos.

No exemplo, provavelmente não faria sentido um objeto **Mago** ficar sem nome e sem poder. Porém, considerando a maneira como é definida a classe, isso é possível, já que, após criar um novo objeto **Mago**, pode-se incluir ou omitir uma atribuição manual às propriedades públicas.

Para evitar tal situação, ao construir o nosso objeto, pode-se obrigar o programa a fornecer o nome e o poder do mago criado, ou seja, é preciso conseguir alterar o comportamento que dirá como será construído o objeto.

Pode-se começar a alteração fazendo o construtor receber o nome por parâmetro:

```
public Mago(string nomeMago)
{
    nome = nomeMago;
    Console.WriteLine("Este é o construtor executando");
}
```

Se você pular a classe **Program**, no método **Main()**, neste momento, verá que há um erro acontecendo:

```
Mago gand = new Mago();
Mago harry = new Mago();
```

Se você passar o *mouse* sobre os termos em vermelho, verá esta mensagem:

```
                Não há nenhum argumento fornecido que corresponde ao parâmetro
formal necessário "nomeMago" de "Mago.Mago(string)" [Mago]
```

Talvez seja necessário executar no terminal o comando **dotnet build** ou o comando **dotnet run** para ver as mensagens de erro, caso a extensão C# não esteja funcionando corretamente.

A mensagem deixa claro que o construtor não pode ser mais usado sem parâmetros.

Pronto! Há agora a obrigatoriedade de informar um nome logo no momento em que o objeto é criado.

```
                Mago gand = new Mago("Gandalf");
                Mago harry = new Mago("Harry Potter");
```

Esse comportamento é similar a um comportamento normal de um método que recebe argumentos, porém com a característica especial de este ser o método que constrói um objeto. Veja que no exemplo anterior era possível remover as linhas

```
                gand.nome = "Gandalf";
```

e

```
                harry.nome = "Harry Potter";
```

de `Main()`, pois os nomes dos magos já foram definidos diretamente no construtor.

Você está pronto para mais um desafio?

Em caso afirmativo, faça a alteração a seguir:

1. Modifique o construtor de **Mago** para receber um segundo parâmetro que servirá para inicializar o atributo `forca`.

Ainda seria possível manter tanto o construtor padrão (sem parâmetros) quanto o construtor com parâmetros. Veja no exemplo a seguir a classe completa de **Mago**:

```
using System;

namespace Mago
{
    public class Mago
    {
        public string nome;
        public int idade;
        public int poder;

        public Mago()
        {
            Console.WriteLine("Construtor vazio");
        }

        public Mago(string nomeMago)
        {
            nome = nomeMago;
            Console.WriteLine("Este é o construtor executand
o");
        }

        public void Atacar()
        {
            Console.WriteLine("Mago está atacando!");
            Console.WriteLine($"Ele tem {poder} de poder");
        }
        public void Defender()
        {
            Console.WriteLine("Mago está defendendo!");
        }
    }
}
```

Nesse caso, seria possível proceder com os dois tipos a seguir de instanciação:

```
Mago gand = new Mago();
Mago harry = new Mago("Harry Potter");
```

Quando diversas versões do construtor são colocadas dentro de uma classe, está-se fazendo uma **sobrecarga de construtores**. O programa saberá o construtor que deve acessar observando os parâmetros que são informados.

Encapsulamento

Agora, observe a seguir a classe **Conta**, já desenvolvida no exemplo 17:

```
namespace Bancario
{
    public class Conta
    {
        public int numero;
        public int digitoVerificador;
        public double saldo;
        public string titular;

        public bool RelizarSaque(double valor){
            if(saldo < valor)
            {
                saldo = saldo - valor;
                return true;
            }
            else
            {
                return false; //saldo insuficiente
            }
        }

        public void RealizarDeposito(double valor)
         {
            if(valor > 0)
            {
                saldo = saldo + valor;
            }
        }
    }
}
```

O saque verifica corretamente se o saldo é suficiente para ocorrer. Entretanto, observe que há liberdade para que o código do sistema (em **Program.Main()**, por exemplo) utilize a classe que contenha uma linha de código:

```
contaBancaria.saldo -= 100;
```

Isso significa que, por mais que o método `RealizarSaque()` esteja fazendo a verificação, outra parte do código – externa à classe **Conta** – pode realizar um saque inválido. Para evitar esse equívoco, é possível proteger o atributo “saldo” para que ele não seja alterado nem acessível fora da classe **Conta**. Portanto, é preciso utilizar o encapsulamento.

O encapsulamento, um dos pilares da orientação a objetos, tem o objetivo de esconder informações ou detalhes de implementação de uma classe. Todo acesso a partir de outras classes deve acontecer por meio de métodos ou atributos públicos.

As partes de uma classe, como o atributo “saldo” na classe **Conta**, podem precisar de proteção para que não sejam acessadas por qualquer outro objeto diretamente e para impedir que os campos sofram alterações acidentais, permitindo que seus valores sejam modificados por métodos específicos e internos à própria classe. Assim, os modificadores de acesso são parte essencial para isso no C#.

Modificadores de acesso: uso, sintaxe e palavras reservadas

Os modificadores de acesso são as palavras-chave utilizadas para especificar a acessibilidade da declaração de um membro ou um tipo. Eles são aplicáveis a atributos, classes, métodos e outras construções do C#.

Existem diferentes modificadores de acesso, cada um com as próprias características. Para exemplificar cada um, é importante voltar brevemente à classe **Mago** e depois resolver o problema da classe **Conta**.

Os modificadores de acesso no C# são estes:

Clique ou toque para visualizar o conteúdo.

Public (público)

O modificador de acesso *public* permite acessar o item por qualquer outro objeto em qualquer lugar do programa, o que o caracteriza como a forma menos segura.

```
                        public class Mago
                        {
                            public string nome;
                            public int idade;
                            public int poder;

                            public Mago()
                            {
                            }

                            public Mago(string nomeMago)
                            {
                                nome = nomeMago;
                            }

                            public void Atacar()
                            {
                                Console.WriteLine("Mago está atacand
o!");
                                Console.WriteLine($"Ele tem {poder} de
poder");
                            }
                            public void Defender()
                            {
                                Console.WriteLine("Mago está defendend
o!");
                            }
                        }
```

A classe **Mago** está, neste momento, constituída basicamente por atributos e métodos públicos. Assim, a classe **Program** (ou qualquer outra classe que existisse no projeto) pode chamar qualquer método ou ler e escrever qualquer atributo.

Private (privado)

O modificador de acesso *private* permite acessar o item somente dentro da classe em que ele foi declarado. O private é a visibilidade padrão quando não é definido um modificador de acesso, sendo a forma mais restritiva.

Veja na classe **Mago** que é possível restringir o acesso ao atributo “nome”, pois este pode ser informado diretamente pelo construtor. Pode-se voltar a forçar que o programador informe um nome ao criar um **Mago**:

```
public class Mago
{
    private string nome;
    public int idade;
    public int poder;

    public Mago(string nomeMago)
    {
        nome = nomeMago;
    }

    public void Atacar()
    {
        Console.WriteLine("Mago está atacando!");
        Console.WriteLine($"Ele tem {poder} de poder"
);

    }
    public void Defender()
    {
        Console.WriteLine("Mago está defendendo!");
    }
}
```

Está-se adicionando, assim, uma camada de proteção à classe **Mago**. Nesse momento, a classe **Main**, caso tenha alguma linha de código com acesso direto ao atributo “nome”, como

```
Mago gand = new Mago("Gandalf");
gand.nome = "Gandalf";
```

emitirá um erro, informando que

```
'"Mago.nome" é inacessível devido ao seu nível de pro
teção.
```

A mensagem informa que o atributo “nome” não está mais visível à classe **Program**, o que é esperado, já que ele agora é privado.

Pode-se ainda ir além e tornar privado também o atributo “poder”, mas, de alguma maneira, o valor de poder precisa ser informado. Por isso, nesse caso, o construtor será modificado.

Se você conseguiu realizar o desafio anterior, o construtor já deve estar adaptado.

```
                        public class Mago
                        {
                            private string nome;
                            public int idade;
                            private int poder;

                            public Mago(string nomeMago, int poderMago)
                            {
                                nome = nomeMago;
                                poder = poderMago;
                            }

                            public void Atacar()
                            {
                                Console.WriteLine("Mago está atacando!");
                                Console.WriteLine($"Ele tem {poder} de poder"
);
                            }
                            public void Defender()
                            {
                                Console.WriteLine("Mago está defendendo!");
                            }
                        }
```

As chamadas ao construtor e ao atributo “poder” em **Program.Main()** precisarão ser adaptadas.

Antes, preste atenção a um detalhe: mesmo estando privado, o atributo “poder” é corretamente usado pelo método **Atacar()**. Isso demonstra que qualquer item, mesmo privado, é acessível dentro da própria classe.

Protected (protegido)

O modificador de acesso *protected* permite acessar apenas a classe que contém o modificador, e os tipos derivados dessa classe têm o acesso. Ele não permite acessos externos, a menos que sejam de classes derivadas dele.

Para exemplificar, uma nova classe “Funcionario” será criada:

```
public class Funcionario
{
    protected string nome;
    protected double salario;
}
```

O modificador *protected* será abordado com mais detalhes quando for estudada a hierarquia de classes.

Internal (interno)

O modificador de acesso *internal* permite o acesso somente dentro do projeto atual/*assembly* (mesmo **.exe ou .dll**):

```
public class Funcionario
{
    protected string nome;
    protected double salario;

    internal int identificacao;
}
```

Também é um modificador de acesso muito específico, sendo utilizado esporadicamente.

Propriedades

A classe **Mago** está um pouco inconsistente. Observe que há alguns atributos públicos e outros privados:

```
private string nome;
public int idade;
private int poder;
```

Assim, permitiu-se acesso total à “idade”, mas restringiu-se totalmente o acesso a “nome” e “poder”. Caso a aplicação queira obter, por exemplo, o valor de “nome” de um objeto a partir de `Progam.Main()` , ela não conseguirá.

```
Mago gand = new Mago("Gandalf", 100);
Console.WriteLine(gand.nome);//ERRO
```

Não é razoável, no entanto, que isso aconteça. Para permitir um acesso controlado a esses elementos de classe, podem-se expor os seus valores por meio de métodos:

```
public class Mago
{
    private string nome;
    public int idade;
    private int poder;

    public Mago(string nomeMago, int poderMago)
    {
        nome = nomeMago;
        poder = poderMago;
    }

    public string GetNome()
    {
        return nome;
    }

    public int GetPoder()
    {
        return poder;
    }

    public void Atacar()
    {
        Console.WriteLine("Mago está atacando!");
        Console.WriteLine($"Ele tem {poder} de poder");
    }
    public void Defender()
    {
        Console.WriteLine("Mago está defendendo!");
    }
}
```

Veja no código citado os métodos `GetNome()` e `GetPoder()`. Eles estão retornando o valor das propriedades privadas “nome” e “poder”, respectivamente.

O código em `Program.Main()` poderia então conter a seguinte chamada:

```
Mago gand = new Mago("Gandalf", 100);
Console.WriteLine(gand.GetNome());//OK! Imprimirá o nome
```

Analogamente, pode-se permitir que, de forma controlada, sejam feitas alterações em “nome” e “poder”.

```
public class Mago
{
    private string nome;
    public int idade;
    private int poder;

    public Mago(string nomeMago, int poderMago)
    {
        nome = nomeMago;
        poder = poderMago;
    }

    public string GetNome()
    {
        return nome;
    }

    public void SetNome(string n)
    {
        nome = n;
    }

    public int GetPoder()
     {
        return poder;
    }

    public void SetPoder(int p)
    {
        poder = p;
    }

    public void Atacar()
    {
        Console.WriteLine("Mago está atacando!");
        Console.WriteLine($"Ele tem {poder} de poder");
    }
    public void Defender()
    {
        Console.WriteLine("Mago está defendendo!");
    }
}
```

Os métodos `SetNome()` e `SetPoder()` foram criados. Agora, em `Program.Main()`, seria possível alterar o “nome” de um mago desta maneira:

```
Mago gand = new Mago("Gandalf", 100);
Console.WriteLine(gand.GetNome()); //Gandalf
gand.SetNome("Saruman");
Console.WriteLine(gand.GetNome()); //Saruman
```

O acesso para alterar e recuperar o dado deve ocorrer de forma “controlada” porque, no método, é possível incluir verificações importantes, como restrições para a quantidade de poder (comparar, por exemplo, se o poder informado em **SetPoder()** é maior que determinado limite e, em caso positivo, não alterar o atributo). Isso quer dizer que há um controle de como a informação é alterada e de como ela é exposta, algo que não existia com atributos públicos.

Também é importante mencionar os métodos *getters* e *setters* (métodos que realizam as operações de recuperar e alterar um valor). *Get* retorna informações do atributo, e *set* modifica informações do atributo. Essa é uma maneira muito utilizada para ter acesso a atributos privados.

O C#, no entanto, conta com uma estrutura simplificada para tal: as propriedades.

Grosso modo, pode-se dizer que propriedades são como atributos com *getters* e *setters*. Na verdade, a propriedade, além disso, também utiliza um atributo.

A sintaxe de declaração é:

```
<modificador de acesso> <tipo> <nome>
{
    get
    {
        <código get>
    }
    set
    {
        <código set>
    }
}
```

Tanto `get` quanto `set` são opcionais, mas ao menos um deles deve ser incluído.

No exemplo **Mago**, será definida uma propriedade para “nome”:

```
public string Nome
{
    get
    {
        return nome;
    }
    set
    {
        nome = value;
    }
}
```

A propriedade retira a necessidade dos métodos `GetNome()` e `SetNome()` separados. Tudo fica encapsulado dentro da definição da propriedade. Podem-se adicionar inclusive validações, como na propriedade “poder” a seguir:

```
        public int Poder
        {
            get
            {
                return poder;
            }
            set
            {
                if(value <= 100)
                {
                    poder = value;
                }
            }
        }
```

Observe que a propriedade (nomeada com **P** maiúsculo) trabalha em conjunto com o atributo (nomeado com **p** minúsculo), retornando ou alterando o valor deste.

Veja também que no `set` o valor, em vez de chegar por um parâmetro, chega pela palavra reservada "`value`".

O aspecto mais importante da propriedade é o seu uso. Em `Program.Main()`, é possível utilizá-la da seguinte forma:

```
        Mago gand = new Mago("Gandalf", 100);
        Console.WriteLine(gand.GetNome()); //Gandalf
        gand.Nome = "Saruman"; //vaio entrar no set da propriedade No
me
        gand.Poder = 50; //vai entrar no set da propriedade Poder
        Console.WriteLine(gand.Nome); //Saruman
```

Os métodos `get` e `set` encapsulados na definição da propriedade serão executados implicitamente quando a propriedade for lida ou escrita.

Reformulando a classe **Mago**, criando propriedades para cada atributo, tem-se o seguinte código final em **Mago.cs**:

```
using System;

namespace Mago
{
    public class Mago
    {
        private string nome;
        private int idade;
        private int poder;

        public string Nome
        {
            get
            {
                return nome;
            }
            set
            {
                nome = value;
            }
        }

        public int Idade
        {
            get
            {
                return idade;
            }
            set
            {
                idade = value;
            }
        }

        public int Poder
        {
            get
            {
                return poder;
            }
            set
            {
                if(value <= 100)
                {
                    poder = value;
                }
            }
        }

        public Mago(string nomeMago, int poderMago)
```

```
        {
            nome = nomeMago;
            poder = poderMago;
        }

        public void Atacar()
        {
            Console.WriteLine("Mago está atacando!");
            Console.WriteLine($"Ele tem {poder} de poder");
        }

        public void Defender()
        {
            Console.WriteLine("Mago está defendendo!");
        }
    }
}
```

No exemplo anterior, o atributo “idade” tornou-se privado, e uma propriedade também foi criada para ele. Além disso, os métodos **GetNome**, **SetNome**, **GetPoder** e **SetPoder** foram removidos, pois já estavam redundantes – as propriedades cumprem o seu papel.

O código de **Program.cs** ficou assim:

```
using System;

namespace Mago
{
    class Program
    {
        static void Main(string[] args)
        {
            Mago gand = new Mago("Gandalf", 100);
            Mago harry = new Mago("Harry Potter", 80);

            gand.Idade = 300;

            harry.Idade = 39;

            gand.Atacar();
            harry.Defender();
            harry.Atacar();
            gand.Defender();

        }
    }
}
```

Há um tipo de propriedade mais simples e direta que, implicitamente, cria um atributo privado. A sintaxe é esta:

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

Tal propriedade define um *getter* e um *setter* padrão, além de, de maneira oculta, criar um atributo privado. Na prática, essa construção é equivalente a um atributo público.

Voltando ao caso da classe **Conta**, como seria possível melhorar a maneira de controlar o acesso ao atributo “valor”? Uma opção é usar uma propriedade:

```
namespace Bancario
{
    public class Conta
    {
        public int numero;
        public int digitoVerificador;
        private double saldo;
        public string titular;

        public double Saldo
        {
            get
            {
                return saldo;
            }
        }

        public bool RelizarSaque(double valor)
        {
            if(saldo > valor)
            {
                saldo = saldo - valor;
                 return true;
            }
            else
            {
                return false; //saldo insuficiente
            }
        }

        public void RealizarDeposito(double valor)
        {
            if(valor > 0)
            {
                saldo = saldo + valor;
            }
        }
    }
}
```

O “saldo” tornou-se privado, e a propriedade “Saldo” (com **S** maiúsculo) foi criada, a qual apenas expõe a maneira de recuperar o valor, e não de alterá-lo (lembre-se de que a propriedade pode conter somente *get* ou *set* ou ambos).

Assim, a única maneira de alterar “saldo” é por meio dos métodos `RealizarSaque()` e `RealizarDeposito()`. Então, a nossa classe está mais protegida.

Em `Program.Main()`, pode ser necessário alterar as chamadas de “saldo” para “Saldo”.

Agora, caso se sinta confiante, faça um desafio simples para treinar propriedades.

Para realizar o desafio, faça este exercício:

1. Crie propriedades para os atributos “numero”, “digitoVerificador” e “titular”, que devem se tornar privados.

Exemplo 18 – Programa em C# que manipula informações de **Cliente** usando classe e propriedades (o programa verifica se idade é maior que zero antes de modificar essa informação)

Veja o código a seguir da classe **Cliente**:

```
using System;

namespace Clientes
{
    public class Cliente
    {
        //Atributos da classe cliente
        private string nome;
        private string endereco;
        private int idade;

        public string Nome
        {
            get { return nome; }
            set { nome = value; }
        }

        public string Endereco
        {
            get { return endereco; }
            set { endereco = value; }
        }

        public int Idade
        {
            get{ return idade; }
            set
            {
                if(value > 0)
                    idade = value;
            }
        }
        //Construtor - chamado quando objeto e instanciado
        public Cliente(string novoNome, string novoEndereco,
int novaIdade)
        {
            nome = novoNome;
            endereco = novoEndereco;
            Idade = novaIdade; //usando propriedade que faz v
alidação
        }

        //Mostra dados do cliente
        public void MostraDados()
        {
            Console.WriteLine("Nome : " + nome);
            Console.WriteLine("Endereco : " + endereco);
            Console.WriteLine("Idade : " + idade);
        }
```

```
    }
}
```

Veja agora o código da classe **Program**:

```
using System;

namespace Clientes
{
    class Program
    {
        static void Main(string[] args)
        {
            //Instancia cliente 1
            Cliente c1 = new Cliente("Fabio", "Rua Y", 30);

            //Chama metodo get idade do cliente 1
            Console.WriteLine("Idade do c1: " + c1.Idade);

            //Set idade - modifica valor da var privada
            c1.Idade = 20;

            //Chama metodo get idade do cliente 1
            Console.WriteLine("Idade do c1: " + c1.Idade);
            c1.MostraDados();

            //Cliente 2
            Cliente c2 = new Cliente("Jonas", "Rua K", 19);
            c2.MostraDados();

        }
    }
}
```

Biblioteca C#: classes que são novas estruturas de dados

O C# conta com uma biblioteca de classes muito grande. O console é uma delas e está no *namespace* **System**. Algumas classes são muito úteis em diversos tipos de aplicação.

Observe a seguir dois exemplos de classes que definem variações ou novas estruturas de dados.

Clique ou toque para visualizar o conteúdo.

(#modal-list) (#modal-dictionary)

List

Você se lembra do *array* e da limitação de tamanho dele? De fato, o programador não consegue aumentar a quantidade de posições em tempo de execução. Sendo assim, para contornar a limitação do *array*, pode-se trabalhar com uma classe do C# chamada *list*. Listas são coleções de dados de um mesmo tipo.

Na linguagem C#, há o *namespace* **System.Collections** (uma biblioteca com implementações de coleções, entre elas as listas). A **List<T>** é uma lista composta por um índice e um valor. Em resumo, *list* é um *array* infinito.

Para utilizar a biblioteca **List<T>**, é necessário inserir no topo do nosso *script* (código) este comando:

```
using System.Collections.Generic;
```

Em seguida, a lista com o seguinte comando deve ser criada:

```
List<T> nomeDaLista = new List<T>();
```

T é o tipo de dado usado (*int*, *string*, *double*, uma classe).

No código anterior, foi definida uma variável do tipo **List<T>** com o identificador “nomeDaLista”. No lado direito do sinal de igual, foi atribuída uma instância da lista utilizando a palavra reservada da linguagem C# “new” e o construtor do objeto lista “(List<T>())”. O tipo do objeto que será armazenado na lista é definido o lugar do “T” em List<T>().

Na prática, caso fosse criada uma lista para armazenar tipos inteiros, ter-se-ia o seguinte código:

```
List<int> listaDeInteiro = new List<int>();
```

Para incluir um item na lista, há o método **Add()**, e o valor passa a ser incluído a este por parâmetro. Veja no exemplo:

```
listaDeInteiro.Add(1);
listaDeInteiro.Add(2);
listaDeInteiro.Add(4);
```

Isso resultaria em uma lista com os valores 1, 2 e 4. Eles foram inseridos na lista em ordem: o valor 1 é o primeiro da lista; o valor 2, o segundo; e o valor 4, o terceiro.

```
using System;
using System.Collections.Generic;

namespace teste
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> listaDeInteiro = new Li
st<int>();

            listaDeInteiro.Add(1);
            listaDeInteiro.Add(2);
            listaDeInteiro.Add(4);


            for(int i = 0; i < listaDeInteir
o.Count; i++)

            {
                Console.WriteLine(listaDeInte
iro[i]);

            }
        }
    }
}
```

Veja no exemplo que, diferentemente do *array*, que definia *length* para o tamanho do vetor, *list* tem uma propriedade *count* que traz o valor de quantos itens há na lista.

Caso seja necessário encontrar o índice de algum elemento na lista, é utilizado o seguinte comando:

```
listaDeInteiro.indexOf(valor_armazenado);
```

A lista permite obter um elemento pelo índice. Caso o intuito fosse obter o primeiro elemento da lista para realizar uma operação, deveria ser utilizado o comando **listaDeInteiro[0]**, como feito com *arrays*.

Também é válido saber remover elementos da lista. Este é o comando que se utiliza nesse caso:

```
listaDeInteiro.Remove(2);
```

A lista ficará somente com os elementos 1 e 4. Observe que, quando há mais de um elemento com o mesmo valor, as operações incidem sobre o primeiro.

Caso esteja preparado, realize o exercício de fixação a seguir.

1. Utilizando **List<Cliente<**, adapte o exemplo 18 para que o usuário possa cadastrar clientes em uma lista. Na aplicação, forneça as opções “incluir”, “listar” e “sair”. O programa usará um laço que termina só quando o usuário digitar o comando “sair”. O comando “inserir” criará um novo objeto “Cliente” e o preencherá com dados de nome e idade informados pelo usuário. Em seguida, o objeto será adicionado à lista. O comando “listar” percorrerá a lista e usará o método **Cliente.MostraDados()** para mostrar os clientes cadastrados.

Dictionary

Quando uma pessoa usa um dicionário, de papel mesmo, sempre haverá uma palavra e um significado associados. Na programação, acontece a mesma coisa com os dicionários em C#. Os dicionários podem ser vistos como listas que armazenam pares de valores. Esses valores são identificados por meio de uma chave e acompanham uma definição.

Para criar um dicionário, é necessário declará-lo:

```
Dictionary<string, string> itens = new Dictio
nary <string, string>();
```

Veja que foram determinados dois tipos diferentes de variáveis, ou seja, não é preciso ter necessariamente uma chave e uma definição de mesmo tipo. Por exemplo, a chave pode ser um *int*; e a descrição, uma *string*.

```
Dictionary<int, string> nomeAlunos = new Dict
ionary <int, string>();
```

Pode-se resumir, de forma simplista, a definição do *dictionary* a uma lista indexada. Para adicionar um elemento ao dicionário, é utilizado **Add**:

```
nomeAlunos.Add(1, "Júlia");
nomeAlunos.Add(45, "Carolina");
itens.Add("indice", "um valor");
```

Para acessar os valores armazenados, basta saber a chave. O comando fica assim:

```
Console.WriteLine(nomeAlunos[1]);
Console.WriteLine(itens["indice"]);
```

Caso seja necessário remover um dos itens, basta a chave. Quando esse comando é utilizado, tanto a chave quanto o valor do item são removidos.

```
nomeAlunos.Remove(1);
itens.Remove("indice");
```

Considerações finais

Conclui-se, assim, o estudo básico do C#, exercitando laços e estruturas de dados e aprendendo a utilizar a orientação a objetos na linguagem.

Tais conhecimentos são fundamentais para compreender a programação para *web*, especialmente no que diz respeito ao código de *back-end*. O domínio na POO, além de ser um pré-requisito básico para um programador hoje em dia, é fundamental para o melhor aproveitamento dos recursos que o ASP.NET Core (o *framework* de programação *web* do .NET Core) fornece.