

Sumário

Prefácio.....	5
Agradecimentos.....	6
1. Primeiros passos	7
1.1 O que é programação.....	7
1.2 Lógica de programação.....	7
1.3 Algoritmos	9
1.4 Visual Studio Code	11
1.5 Visual Studio Code - extensões	12
1.6 .NET	14
1.7 CLR	17
1.8 CIL.....	17
1.9 FCL.....	18
1.10 Como C# e .NET trabalham juntos	20
1.11 Por que escolhi C# para este livro de programação?	20
2. Escrevendo os primeiros códigos	21
2.1 Terminal / Primeiro programa.....	22
2.2 Hello World – Brian Kernighan (Curiosidade).....	24
2.3 Console.WriteLine	24
2.4 Console.ReadLine	25
2.5 Comentários.....	28
2.6 Variáveis	30
2.7 Estrutura de controle	32
2.8 Estrutura de repetição	34
2.9 Desafios	36
3. Tipos de dados, operadores e saída de dados	38
3.1 Tipos de dados	38

3.2 Casting	40
3.3 Tabela ASCII.....	42
3.4 Variáveis e constantes	42
3.5 Regras para criação de variáveis/constantes	44
3.6 Enumeradores	45
3.7 Operadores	49
3.8 Tabela verdade	58
3.9 Concatenação.....	59
3.10 Entrada de dados	61
3.11 Covert.To	63
3.12 Alias.....	65
3.13 Desafios	66
4. Instrução de decisão e repetição	67
4.1 Estrutura de controle	68
4.2 Switch-case.....	73
4.3 For	75
4.4 While	77
4.5 Do-While	79
4.6 Break.....	81
4.7 Continue.....	83
4.8 Desafios	85
5. Vetores e matrizes.....	85
5.1 Vetores.....	86
5.2 Matrizes.....	89
5.3 Copiando vetores ou matrizes.....	92
5.4 Foreach	96
5.5 Redimensionamento de vetores	99

5.6 Ordenação de vetores	102
5.7 Busca de itens	108
5.8 Desafios	112
6. Listas dinâmicas	114
6.1 Criação de lista dinâmica	114
6.2 Adicionando elementos	114
6.3 Acesso a elementos	116
6.4 Pesquisa e filtragem	117
6.5 Remoção de elementos.....	122
6.6 Iteração sobre a lista	128
6.7 Verificando tamanho de uma lista.....	131
6.8 Desafios	131
7. Função	132
7.1 O que é função	133
7.2 Parâmetro e argumento.....	134
7.3 Retorno	136
7.4 Argumento por referência.....	139
7.5 Valores por omissão	141
7.6 Recursividade	144
7.7 Função na prática	147
7.8 Função de callback.....	150
7.9 Função anônima	152
7.10 Desafios	155
8. Memória Stack e Heap.....	156
8.1 Memória Stack	157
8.2 Memória Heap	159
9. Strings.....	163

9.1 O que são strings.....	163
9.2 Compondo palavra usando char	164
9.3 Strings são imutáveis.....	164
9.4 Comparação de strings	166
9.5 Manipulação de strings.....	168
9.6 StringBuilder	171
9.7 Desafios	174
10. Try Catch e Finally.....	175
10.1 Tratamento de exceções.....	175
10.2 Exceções personalizadas.....	178
Desafios	182
11. Programação assíncrona	183
11.1 O que é programação assíncrona.....	183
11.2 Criando um arquivo TXT	183
3. Criando arquivos TXT usando programação assíncrona.....	185
12. O que fazer agora.....	191
12.1 Conteúdo recomendado	192
12.2 Primeiro emprego	192

Prefácio

É com grande satisfação que apresento este livro, que foi cuidadosamente elaborado para guiar você no mundo da programação. Seja você um iniciante, um estudante ou um profissional, este livro foi desenvolvido para ajudá-lo a compreender os conceitos e técnicas fundamentais que um programador precisa saber.

Desde os conceitos básicos de lógica e algoritmos até tópicos mais complexos, como programação assíncrona, abordaremos cada aspecto de forma clara e prática.

No capítulo 1, daremos os primeiros passos, abrangendo conceitos essenciais como o que é programação e lógica de programação. Também exploraremos a utilização das ferramentas Visual Studio Code e .NET.

A partir do capítulo 2, você aprenderá sobre a linguagem de programação C#. Você terá a oportunidade de escrever seus primeiros códigos, compreender estrutura de controle e a manipulação de variáveis.

Nos capítulos 3 e 4, falaremos sobre tipos de dados, operadores, estrutura de repetição e aprofundaremos em estrutura de controle.

As seções seguintes, de 5 a 9, abordarão tópicos como vetores, matrizes, listas dinâmicas, funções, memórias stack e heap.

Por fim, nos Capítulos 10 a 12, exploraremos o tratamento de exceções e programação assíncrona.

A cada capítulo, você encontrará desafios para testar suas habilidades e reforçar o aprendizado. Esses desafios são projetados para ajudá-lo a colocar em prática o que aprendeu e a identificar áreas onde ainda precisa de mais prática.

Todos os códigos e desafios podem ser encontrados no site:
<https://codigodascoisas.com.br/livroprogramacaopariniciantes>

Agradecimentos

Sou profundamente grato a Deus por ter me dado a vida.

Sou imensamente grato a minha família, principalmente aos meus pais, Elizabeth Bafilli e ao meu falecido pai, Jorge Bafilli, que com muito esforço me proporcionaram a oportunidade de estudar tecnologia, também sou imensamente grato à minha esposa, Ana Bafilli, por estar ao meu lado em todos os momentos, sou muito grato ao meu sogro Claudio e à minha sogra Joana por estarem sempre presentes.

Sou grato ao meu amigo Adilson Buratto por ter me apresentado a linguagem de programação C# , e à minha amiga Elenir Dantas por sempre me ajudar quando preciso.

Também sou grato ao meu amigo Lucas Silva Fernandes por sempre compartilhar boas ideias.

Sou muito grato ao Edwin Lindquist, que me ajudou muito no começo da minha carreira. Também quero agradecer ao meu mentor e amigo, Marcos Macedo, por sempre estar disposto sempre que preciso.

Também sou imensamente grato à minha amiga Aretha Perboni, uma verdadeira especialista em gestão de pessoas.

Autor: Bruno Bafilli

1. Primeiros passos

“Aprenda os conceitos básicos, então você pode ser rápido. Se você pular os conceitos básicos, você nunca irá ser rápido.”

1.1 O que é programação

Programação é o processo de desenvolver instruções para um computador executar uma tarefa através de algoritmos, algoritmos são sequências lógicas que resolvem um problema.

Os programas de computador são compostos por um conjunto de instruções que informam ao computador quais ações devem ser executadas. Essas instruções podem incluir cálculos matemáticos, manipulação de dados, tomada de decisões, repetições e interações com o usuário.

A programação permite criar uma ampla variedade de softwares, desde aplicativos para dispositivos móveis e sistemas operacionais, até jogos, sites e aplicações empresariais. Programação é uma habilidade essencial para quem deseja trabalhar na área de tecnologia da informação e oferece um mundo de possibilidades para resolver problemas e desenvolver soluções inovadoras.

Para programar, é necessário aprender pelo menos uma linguagem de programação. Existem várias linguagens de programação disponíveis, como C#, Python, Java, C++, JavaScript, C, Solidity, entre outras, cada uma com suas características e usos específicos.

1.2 Lógica de programação

A lógica de programação é a base fundamental para o desenvolvimento de programas para computadores. A lógica é a

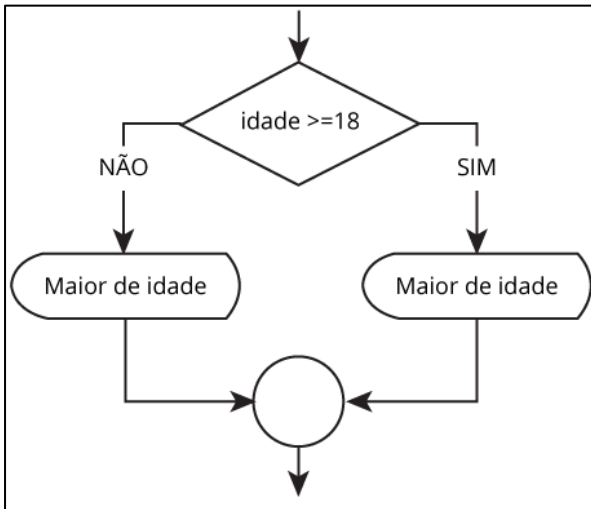
maneira de como o programador organiza as instruções para solucionar um problema de forma eficiente.

A lógica de programação envolve a capacidade de dividir um problema complexo em partes menores, identificar padrões e relações entre os dados, tomar decisões com base em um problema específico e repetir tarefas quando necessário.

Alguns conceitos importantes relacionados à lógica de programação incluem:

- **Algoritmos:** São sequências ordenadas de passos que descrevem a solução de um problema. Um algoritmo é uma representação abstrata de um processo que pode ser implementado em uma linguagem de programação.
- **Estruturas de Controle:** Permitem controlar o fluxo de execução de um programa. As estruturas de controle mais comuns são:
 - Estruturas condicionais (IF-ELSE): Permitem tomar decisões com base em condições lógicas.
 - Estruturas de repetição (LOOP): Permitem executar repetidamente um bloco de código até que uma condição seja satisfeita.
- **Variáveis:** São espaços de memória que armazenam valores temporários ou permanentes. As variáveis permitem que os programas armazenem e manipulem dados durante a execução.
- **Tipos de Dados:** São categorias que definem o tipo de valor que uma variável pode armazenar, como números inteiros, números de ponto flutuante, caracteres, entre outros.
- **Operadores:** Permitem realizar operações matemáticas e lógicas em dados. Alguns exemplos de operadores são soma

(+), subtração (-), multiplicação (*), divisão (/), igualdade (==), e operadores lógicos como AND (&&), OR (||) e NOT (!).



Código 1.2.1

No exemplo Código 1.2.1, é apresentada uma lógica simples de tomada de decisão. Através dela, podemos determinar o fluxo do programa com base na idade fornecida.

Se a idade for **maior** ou **igual** a **18**, a lógica seguirá pelo "**caminho SIM**". Por outro lado, se a idade for **menor** que **18**, o fluxo da lógica será direcionado pelo "**caminho NÃO**".

Essa estrutura de decisão é uma parte fundamental da lógica de programação, permitindo que o programa tome diferentes caminhos com base em condições específicas.

1.3 Algoritmos

Algoritmos são sequências ordenadas de passos que descrevem como resolver um problema de maneira sistemática e lógica. Eles são a base da programação, pois representam uma solução abstrata e detalhada para um determinado problema.

Um algoritmo é composto por instruções precisas e bem definidas, que devem ser executadas em uma determinada ordem para alcançar o resultado desejado. Essas instruções podem incluir operações matemáticas, tomada de decisões, repetições e interações.

Para desenvolver um algoritmo eficiente, é importante considerar a clareza, a simplicidade e a eficiência. Um algoritmo claro é facilmente compreensível, permitindo que outros programadores ou até mesmo você possa entender e dar continuidade ao trabalho. A simplicidade envolve evitar a complexidade desnecessária e buscar soluções diretas e concisas. Já a eficiência refere-se a encontrar a solução mais rápida e otimizada para o problema em questão.

A criação de um algoritmo geralmente envolve as seguintes etapas:

- **Compreender o problema:** Entender completamente o problema a ser resolvido e quais são os requisitos e restrições envolvidos.
- **Definir a entrada e a saída:** Identificar quais dados serão fornecidos como entrada e qual será o resultado esperado.
- **Dividir em etapas:** Decompor o problema em etapas menores e mais gerenciáveis, de modo que cada etapa possa ser abordada separadamente.
- **Desenvolver a lógica:** Determinar a sequência de passos necessários para resolver cada etapa do problema. Isso pode incluir operações matemáticas, condições, iterações, entre outros.
- **Testar e refinar:** Executar o algoritmo em diferentes cenários de teste, verificar se o resultado é o esperado e realizar ajustes, se necessário.

1.4 Visual Studio Code

Visual Studio Code (VS Code) é um editor de código-fonte desenvolvido pela Microsoft. Ele oferece recursos avançados de edição, integração com controle de versão, depuração, terminal integrado e uma comunidade ativa. O VS Code é amplamente utilizado por desenvolvedores de software para projetos diversos, como desenvolvimento web, aplicativos móveis e serviços em nuvem.

Aqui está um tutorial básico para instalar o Visual Studio Code:

Passo 1: Baixe o instalador

- Abra o navegador da web e acesse o site oficial do Visual Studio Code em: <https://code.visualstudio.com/>
- Na página inicial do site, você verá um botão de download. Clique nele para baixar o instalador adequado para o seu sistema operacional (Windows, macOS ou Linux).

Passo 2: Execute o instalador

- Após o download ser concluído, execute o arquivo de instalação que você baixou.

Passo 3: Siga as instruções de instalação

- O instalador do Visual Studio Code será iniciado. Siga as instruções na tela para prosseguir com a instalação.
- Geralmente, você precisa aceitar os termos de uso e escolher as opções de instalação, como o local de instalação padrão.

Passo 4: Conclua a instalação

- Após a instalação ser concluída, você pode optar por iniciar o Visual Studio Code imediatamente ou fechar o instalador e executá-lo posteriormente a partir do menu Iniciar ou da área de trabalho.

1.5 Visual Studio Code - extensões

O motivo de instalar a extensão do C# no Visual Studio Code é que ele fornece um ambiente de desenvolvimento completo e otimizado para trabalhar com a linguagem de programação C#. A extensão oferece diversas vantagens, como:


- **Suporte à linguagem C#:** A extensão do C# no Visual Studio Code oferece recursos avançados de edição e depuração para a linguagem C#. Ele fornece realce de sintaxe, preenchimento automático de código além de sugestões contextuais ao digitar, o que aumenta a produtividade durante o desenvolvimento.
- **Integração com o .NET:** A extensão do C# no Visual Studio Code se integra perfeitamente ao ecossistema do .NET. Ele permite criar, compilar e executar projetos .NET diretamente do Visual Studio Code, facilitando o desenvolvimento de aplicativos para diferentes plataformas, como Windows, macOS e Linux.

Agora, vamos seguir o tutorial para instalar o plugin do C# no Visual Studio Code:

Passo 1: Abra o Visual Studio Code:

- Inicie o Visual Studio Code no seu computador.

Passo 2: Acesse a seção de Extensões:

- No painel lateral esquerdo do Visual Studio Code, clique no ícone com quatro quadrados  ou pressione **Ctrl+Shift+X** (Windows/Linux) ou **Cmd+Shift+X** (macOS) para abrir a seção de Extensões.

Passo 3: Pesquise o plugin do C#:

- Na barra de pesquisa na parte superior, digite "C#" e pressione Enter.
- O primeiro resultado geralmente é o "C#" (Microsoft). Clique em "Install" para instalá-lo.

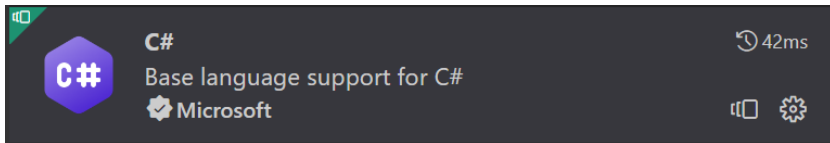


imagem 1.6.1

Passo 4: Aguarde a instalação:

- O Visual Studio Code começará a baixar e instalar a extensão do C#. Aguarde até que a instalação seja concluída.

Passo 5: Reinicie o Visual Studio Code:

- Após a instalação, é recomendado reiniciar o Visual Studio Code para garantir que todas as alterações sejam aplicadas.

Passo 6: Verifique a instalação:

- Após reiniciar, abra um arquivo com extensão **.cs** ou crie um novo arquivo **.cs**.
- Você verá recursos específicos do C# ativados, como destaque de sintaxe, IntelliSense e outras funcionalidades que ajudarão durante o desenvolvimento.

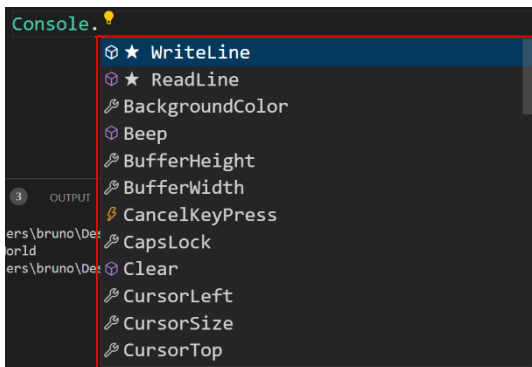
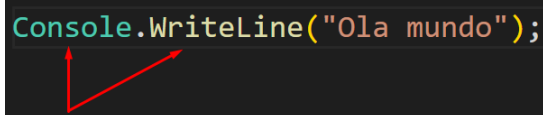


imagem 1.6.2



```
Console.WriteLine("Ola mundo");
```

imagem 1.6.3

Não se preocupe com os códigos apresentados nas imagens. Em breve, você escreverá seus próprios códigos e entenderá o propósito de cada um deles. Os códigos exibidos nas imagens são fornecidos para ajudá-lo a compreender o motivo por trás de cada instalação e para familiarizá-lo com as ferramentas e configurações necessárias. À medida que avançamos, você verá como aplicar esses conhecimentos para criar seus próprios programas em .NET/C#. Portanto, não se preocupe, os códigos podem parecer complexos agora, mas eles serão explicados em detalhes.

1.6 .NET

.NET é uma plataforma de desenvolvimento de software versátil e poderosa, desenvolvida pela Microsoft. Ela oferece um ambiente de execução (Common Language Runtime - CLR) e uma biblioteca de classes (Framework Class Library - FCL), permitindo a criação de aplicativos multiplataforma.

Com o .NET, os desenvolvedores podem escolher entre várias linguagens de programação, como C#, Visual Basic.NET e F#, para desenvolver seus aplicativos. .NET também oferece uma ampla biblioteca de classes que abrange desde tarefas comuns até funcionalidades avançadas.

Além disso, o .NET oferece uma integração perfeita com outras tecnologias e serviços da Microsoft, como o SQL Server, Azure, Windows Presentation Foundation (WPF) e ASP.NET. Isso possibilita a criação de aplicativos que se integram perfeitamente com o ecossistema Microsoft.

O .NET é amplamente utilizado em diversos cenários de desenvolvimento, incluindo aplicativos desktop, web, móveis e serviços

em nuvem. Sua popularidade é impulsionada pela sua robustez, versatilidade e suporte ativo da comunidade de desenvolvedores.

Para instalar o .NET em seu sistema, siga estes passos básicos:

Passo 1: Acesse o site oficial da Microsoft

- Abra o navegador da web e acesse o site oficial do .NET em: <https://dotnet.microsoft.com/>

Passo 2: Escolha a versão do .NET

- Na página inicial do site, você encontrará o botão de download.

Passo 3: Baixe o instalador

- Você será redirecionado para a página de download correspondente. Certifique-se de baixar o .Net 7, que é a versão mais recente disponível no momento da publicação deste livro. É importante ressaltar que as versões podem sofrer alterações ao longo do tempo.
- Após o download ser concluído, execute o arquivo de instalação que você baixou.

Passo 5: Siga as instruções de instalação

- O instalador do .NET será iniciado. Siga as instruções na tela para prosseguir com a instalação.
- Geralmente, você precisa aceitar os termos de uso e escolher as opções de instalação, como o local de instalação padrão.

Passo 6: Conclua a instalação

- Após a conclusão da instalação, é recomendado reiniciar o computador para garantir que todas as alterações sejam aplicadas corretamente. Em seguida, você pode executar alguns comandos de verificação para confirmar se o .NET foi instalado com sucesso.

Verifique se o .NET foi instalado corretamente em seu computador seguindo os seguintes passos:

Passo 1: Abra um prompt de comando ou terminal.

- No Windows: Pressione a tecla **Windows**, digite "**Prompt de Comando**" ou "**cmd**" e pressione **Enter**. Isso abrirá o Prompt de Comando.
- No Linux/macOS: Abra o terminal do sistema.

Passo 2: Digite o comando **dotnet --version**.

- Digite **dotnet --version** exatamente como está, sem aspas.
- Pressione **Enter** para executar o comando.

Passo 3: Aguarde o resultado.

- Após pressionar **Enter**, o comando será executado e exibirá a versão do .NET SDK instalada no seu sistema.

```
PS C:\Users\user> dotnet --version  
7.0.302
```

Terminal 1.7.1

Depois de instalar o .NET, você estará pronto para desenvolver aplicativos usando essa plataforma. Você pode usar o Visual Studio, Visual Studio Code ou outros editores de código para desenvolver seus aplicativos com base no .NET.

É importante mencionar que os detalhes específicos da instalação podem variar dependendo da versão do .NET e do sistema operacional que você está utilizando. É recomendado seguir as instruções fornecidas pela Microsoft durante o processo de instalação para garantir uma instalação bem-sucedida.

1.7 CLR

CLR (Common Language Runtime) é um componente fundamental da plataforma .NET. É responsável por gerenciar a execução de programas escritos em linguagens compatíveis com o .NET, como C#, Visual Basic e outras linguagens de programação.

A principal função do CLR é fornecer um ambiente de execução gerenciado para aplicativos .NET, garantindo que o código seja executado de forma segura e eficiente, independentemente da plataforma em que está sendo executado. Ele oferece recursos como:

- **Coleta automática de lixo:** Coleta automaticamente os objetos que não estão mais sendo usados, liberando memória e evitando vazamentos de memória.
- **Gerenciamento de memória:** Gerencia a memória usada pelos objetos, garantindo que eles não sejam acessados indevidamente e que não haja erros de memória.
- **Suporte a exceções:** Captura e lida com exceções, evitando que aplicativos travem.
- **Manipulação de threads:** Permite que aplicativos executem vários threads simultaneamente, o que pode melhorar o desempenho.

Quando um programa .NET é compilado, ele é transformado em uma linguagem intermediária conhecida como CIL (Common Intermediate Language) ou IL (Intermediate Language). Em tempo de execução, o CLR (Common Language Runtime) converte o CIL em código nativo específico da máquina em que está sendo executado, possibilitando a execução do código .NET em qualquer ambiente compatível com o CLR.

1.8 CIL

CIL (Common Intermediate Language), também conhecida como MSIL (Microsoft Intermediate Language), é uma linguagem de programação de nível intermediário usada pelo .NET Framework.

Quando um código-fonte em uma linguagem .NET (como C#, Visual Basic ou F#) é compilado, ele não é convertido diretamente para o código de máquina específico do processador em que será executado. Em vez disso, é traduzido para o CIL (Common Intermediate Language), que é uma linguagem intermediária.

Aqui está como funciona o processo:

- **Compilação do código-fonte:** O código-fonte escrito em uma linguagem .NET é compilado usando o compilador específico para essa linguagem. Por exemplo, se você estiver usando C#, o código será compilado pelo compilador C#.
- **Criação do CIL:** Durante a compilação, o código-fonte é traduzido para o CIL, que é uma representação intermediária e independente de plataforma.
- **Assembly .NET:** O CIL é armazenado em um arquivo chamado "assembly", que é a unidade básica de implantação no .NET. Um assembly é essencialmente um arquivo que contém o CIL, metadados e outras informações necessárias para executar o programa.
- **Just-in-time (JIT) Compiler:** Quando o aplicativo .NET é executado, o CLR (Common Language Runtime) entra em ação. O CLR é responsável por carregar o assembly, compilar o CIL para código de máquina específico da plataforma em que está sendo executado e, finalmente, executar o código compilado.

1.9 FCL

FCL é um conjunto de bibliotecas de classes que constituem o conjunto padrão de funcionalidades do .NET e inclui uma ampla gama de recursos, como manipulação de arquivos, acesso a bancos de

dados, processamento de XML, criação de interfaces gráficas e muitas outras funcionalidades.

Imagine que você está construindo uma casa com peças de Lego. O FCL seria como uma grande caixa cheia de diferentes peças de Lego que você pode usar para construir sua casa.

Nessa caixa, você encontrará uma grande variedade de peças diferentes, como blocos, rodas, janelas, telhados etc. Cada uma dessas peças possui uma função específica e pode ser usada para diferentes partes da sua casa.

Da mesma forma, o FCL é uma caixa cheia de diferentes "peças" de código que os desenvolvedores podem usar ao criar seus aplicativos. Cada uma dessas "peças" é uma classe ou um conjunto de classes que possui uma funcionalidade específica, como trabalhar com números, manipular arquivos, acessar a internet, entre outras coisas.

Quando você precisa realizar uma tarefa específica em seu aplicativo, em vez de construir tudo do zero, você pode procurar no FCL a "peça" certa para essa tarefa. É como pegar a peça de lego que melhor se encaixa na sua construção.

Assim como os blocos de lego são projetados para se encaixar perfeitamente uns nos outros, as classes no FCL são projetadas para trabalhar harmoniosamente em conjunto. Isso significa que, ao usar essas "peças" de código, você pode construir seu programa de maneira mais rápida e eficiente, aproveitando o trabalho árduo e a qualidade das peças fornecidas pelo FCL.

Essa coleção de peças de Lego - o FCL - é parte fundamental do ecossistema .NET, pois oferece aos desenvolvedores uma base sólida e confiável para a criação de uma ampla variedade de aplicativos. Em vez de reinventar a roda, eles podem usar as peças disponíveis no FCL para criar suas aplicações de forma mais fácil e eficaz.

1.10 Como C# e .NET trabalham juntos

C# é a linguagem de programação usada para desenvolver aplicações .NET. O código é escrito em C#, que é então compilado em CIL e é executado pelo CLR.

```
DateTime dataAtual = DateTime.Now;  
  
Console.WriteLine("Data e hora atual: " +  
dataAtual.ToString());
```

Código 1.10.1

```
Saida: Data e hora atual: 03/04/2022 09:00:25
```

Execução código 1.10.1

Neste exemplo, utilizamos a classe **DateTime** da biblioteca de classes do .NET para obter a data e hora atual através do método **Now()**. Em seguida, utilizamos o método **ToString()** para converter a data e hora em uma representação legível e a exibimos no console usando **Console.WriteLine()**.

Esse exemplo simples demonstra a utilização do .NET para obter informações de data e hora e a integração com a linguagem C# para exibi-las no console.

Se você não entendeu o código 1.10.1 não se preocupe! Eles serão explicados mais detalhadamente ao longo do livro. O importante agora é entender os conceitos.

1.11 Por que escolhi C# para este livro de programação?

Existem várias razões pelas quais vamos usar a linguagem de programação C#.

- **Plataforma abrangente:** C# é uma linguagem que pode ser usada para desenvolver uma ampla variedade de aplicativos, desde aplicativos desktop até aplicativos móveis e aplicações web. Ela é uma linguagem versátil que se integra bem com

várias plataformas, como Windows, iOS e Android.

- **Compatibilidade com a plataforma .NET:** C# é a linguagem de programação principal para a plataforma .NET da Microsoft. A plataforma .NET oferece uma ampla gama de recursos e bibliotecas para desenvolver aplicativos robustos e escaláveis
- **Linguagem orientada a objetos:** C# é uma linguagem orientada a objetos, o que significa que ela suporta conceitos como encapsulamento, herança e polimorfismo. Isso permite uma estrutura de código organizada, modular e reutilizável, facilitando o desenvolvimento e a manutenção de projetos de grande porte.
- **Grande comunidade e suporte:** C# possui uma comunidade ativa de desenvolvedores e um ecossistema robusto. Existem muitos recursos disponíveis, como fóruns, documentação oficial, tutoriais, bibliotecas de terceiros e uma vasta quantidade de exemplos de código disponíveis online. Isso facilita o aprendizado e o suporte durante o processo de desenvolvimento.
- **Oportunidades de carreira:** O conhecimento em C# e na plataforma .NET oferece boas oportunidades de carreira, especialmente no desenvolvimento de software corporativo.

2. Escrevendo os primeiros códigos

“Qualquer tolo pode escrever um código que o computador entenda. Bons programadores escrevem código que os humanos possam entender.”

2.1 Terminal / Primeiro programa

O terminal é uma interface de linha de comando presente em sistemas operacionais, como Windows, macOS e Linux. Ele permite que os usuários interajam com o sistema operacional através de comandos de texto.

Ao abrir o terminal, é apresentada uma linha de comando onde pode digitar comandos específicos do sistema operacional ou de programas instalados. Esses comandos são interpretados pelo sistema operacional e executam ações correspondentes, como listar arquivos, criar diretórios, instalar pacotes, compilar código, executar scripts...

O terminal oferece uma abordagem poderosa e flexível para interagir com o sistema operacional, permitindo que os usuários realizem uma variedade de tarefas. Embora possa ser intimidador para iniciantes, dominar o uso do terminal pode ser extremamente útil para desenvolvedores de software.

Vamos seguir os passos abaixo para criar o seu primeiro programa em .NET usando terminal.

Passo 1: Abra o Terminal:

- No Windows, você pode abrir o Prompt de Comando pressionando as teclas **Win + R** e digitando "cmd" ou abrindo o PowerShell.
- No macOS, abra o Terminal através do Launchpad, localizado na pasta "Outros" ou pela barra de pesquisa do Spotlight.
- No Linux, abra o Terminal usando a combinação de teclas **Ctrl + Alt + T** ou procure por "**Terminal**" no menu de aplicativos.

Passo 2: Navegue até a pasta do projeto:

- Use o comando **cd** seguido do caminho da pasta onde deseja criar seu programa em .NET. Exemplo: **cd /caminho/da/pasta**

Passo 3: Crie um novo projeto em .NET:

- Use o comando **dotnet new** seguido do tipo de projeto que deseja criar. Exemplo: **dotnet new console** para criar um novo projeto de console em .NET.

Passo 4: Acesse a pasta do projeto:

- Use o comando **cd** seguido do nome da pasta do projeto que acabou de criar. Exemplo: **cd nomedoprojeto**

Passo 5: Abra o projeto no editor de código:

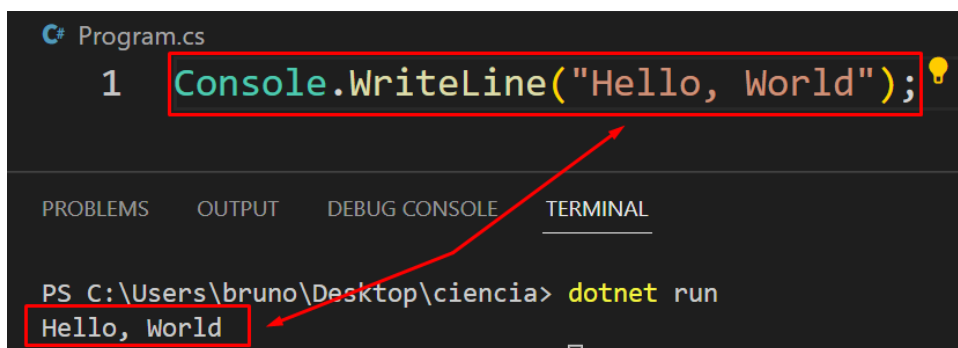
- Use o comando **code .** para abrir a pasta do projeto no Visual Studio Code. (Certifique-se de ter o Visual Studio Code instalado e configurado corretamente no seu sistema).

Passo 6: Edite e compile seu código:

- No Visual Studio Code, você pode editar o código-fonte do seu programa em .NET.
- Use o comando **dotnet build** para compilar seu projeto e verificar se há erros.

Passo 7: Execute seu programa:

- Use o comando **dotnet run** para executar seu programa em .NET.



The screenshot shows the Visual Studio Code interface. At the top, a file named 'Program.cs' is open, containing the code `1 Console.WriteLine("Hello, World");`. The code is highlighted with a red box. Below the editor, the 'TERMINAL' tab is active, showing the command prompt `PS C:\Users\bruno\Desktop\ciencia> dotnet run` and the output `Hello, World`. A red arrow points from the code in the editor to the output in the terminal. The output 'Hello, World' is also highlighted with a red box.

imagem 2.1.1

Ao executar o programa em .NET, você verá a mensagem "Hello, World!" impressa no terminal ou console. Essa é uma maneira simples, mas significativa, de verificar se o ambiente de desenvolvimento está configurado corretamente. O "Hello, World!" é um marco importante para os desenvolvedores iniciantes e marca o começo da jornada na programação em .NET. Parabéns, você criou seu primeiro programa!

2.2 Hello World – Brian Kernighan (Curiosidade)

A frase "Hello, World!" em programação é frequentemente atribuída a Brian Kernighan, um renomado cientista da computação e coautor do livro "The C Programming Language". No livro, publicado em 1978, Kernighan e Dennis Ritchie introduziram a linguagem de programação C e apresentaram o exemplo do programa "Hello, World!".

O programa "Hello, World!" foi criado para demonstrar a estrutura básica de um programa em C, exibindo a mensagem "Hello, World!" na tela. Ele se tornou um exemplo icônico e uma tradição em muitas linguagens de programação, sendo amplamente utilizado como um ponto de partida para novos programadores.

2.3 Console.WriteLine

O Console.WriteLine é uma instrução que exibe uma linha de texto no console de saída do programa. A sintaxe básica do **Console.WriteLine** é a seguinte.

```
Console.WriteLine("Texto");
```

Código 2.3.1

- **Console:** Classe responsável por interagir com o console.
- **WriteLine:** Método que escreve uma linha de texto.
- **Texto:** Informação ou valor que você deseja exibir. Pode ser uma cadeia de caracteres (string), um número, uma variável ou qualquer expressão válida.

Saida: Texto

Execução código 2.3.1

Ao executar o **Console.WriteLine**, a linha de texto é impressa no console e é seguida por uma quebra de linha. Isso significa que o próximo texto ou mensagem será exibido em uma nova linha.

As aspas duplas (" ") são importantes para indicar que o texto dentro delas é uma sequência de caracteres (**string**). As aspas duplas são usadas para delimitar uma **string** literal.

Ao usar o comando **Console.WriteLine** para exibir mensagens na tela, envolvemos o texto desejado com aspas duplas para indicar que é uma **string**.

```
Console.WriteLine("Bem-vindo ao meu programa");
```

Código 2.3.2

Neste exemplo, a mensagem "Bem-vindo ao meu programa" será exibida no console ou na janela de saída do programa.

Saida: Bem-vindo ao meu programa

Execução código 2.3.2

O **Console.WriteLine** é uma ferramenta útil para exibir informações durante a execução do programa, como mensagens de saída, resultados de cálculos e valores de variáveis. Ele ajuda a fornecer feedback para o usuário, facilita a depuração do código e permite acompanhar o progresso do programa em tempo de execução.

2.4 Console.ReadLine

O **Console.ReadLine** é uma instrução que permite o usuário inserir dados a partir do teclado durante a execução do programa. Ele espera que o usuário digite um texto e pressione a tecla enter para confirmar a entrada. A sintaxe básica do **Console.ReadLine** é a seguinte:

```
string entrada = Console.ReadLine();
```

Código 2.4.1

- **Console:** É a classe responsável por interagir com o console.
- **ReadLine:** É o método que lê uma linha de texto a partir do teclado.

A instrução **Console.ReadLine** aguarda a entrada do usuário e armazena o texto digitado na **variável entrada**, que deve ser do tipo **string**.

Aqui está um exemplo para ilustrar o uso do **Console.ReadLine**:

```
Console.WriteLine("Qual é o seu nome?");  
string nome = Console.ReadLine();  
Console.WriteLine("Olá, " + nome + "! Bem-vindo(a).");
```

Código 2.4.2

Neste exemplo, o programa exibirá a mensagem "Qual é o seu nome?" no console e, em seguida, aguardará que o usuário digite seu nome. Assim que o usuário pressionar enter, o texto digitado será armazenado na **variável nome**. Em seguida, o programa exibirá a mensagem "Olá, **[nome]**! Bem-vindo(a)." no console, substituindo **[nome]** pelo valor digitado pelo usuário.

```
Saida: Qual é o seu nome?  
Entrada: Bruno  
Saida: Olá, Bruno! Bem-vindo(a).
```

Execução código 2.4.2

No exemplo anterior, foi criada uma variável chamada **nome** para armazenar a entrada do usuário obtida por meio do **Console.ReadLine**. A variável é declarada como do tipo **string**, foi usada para representar uma sequência de caracteres (texto).

A linha de código **string nome = Console.ReadLine();** possui os seguintes elementos:

- **string**: É o tipo de dados da variável. No caso, string representa uma cadeia de caracteres (texto).
- **nome**: É o nome dado à variável. Você pode escolher qualquer nome válido para a variável, desde que siga as regras de nomenclatura do C#.
- **=**: É o operador de atribuição, que associa o valor obtido do `Console.ReadLine` à variável `nome`.
- **Console.ReadLine()**: É a chamada do método **ReadLine** da classe `Console`, que lê uma linha de texto digitada pelo usuário e retorna essa entrada como uma string.

Ao atribuir o resultado do **Console.ReadLine** à variável **nome**, você pode usar esse valor posteriormente no código, como no exemplo em que é exibida a mensagem de boas-vindas com o nome fornecido pelo usuário.

A variável `nome` é utilizada para armazenar temporariamente a entrada do usuário, permitindo que você acesse e manipule essa informação em outras partes do seu programa.

O **Console.ReadLine** é útil quando você precisa solicitar dados ao usuário para personalizar a interação do programa. Ele pode ser usado para receber entradas de dados, como nome, idade, número de telefone etc. **Console.ReadLine** permite que o programa leia e utilize as informações fornecidas pelo usuário, tornando-o mais dinâmico e interativo.

Não se preocupe se você ainda está um pouco confuso com o conceito de **string**, **variável** e **Console.ReadLine**. Mais adiante, vamos abordar novamente esse assunto com mais detalhes para garantir que você compreenda o seu funcionamento.

2.5 Comentários

Os comentários são trechos de texto que fornecem informações explicativas dentro do código-fonte do programa. Eles são usados para documentar, esclarecer e fornecer contexto sobre o código, ajudando a compreendê-lo tanto para você quanto para outros desenvolvedores que possam ler o código posteriormente.

Existem dois tipos de comentários: comentários de linha única e comentários de várias linhas.

Comentários de linha única

Os comentários de linha única são usados para adicionar informações breves em uma única linha de texto. Eles são representados pelo símbolo `//` e tudo após esse símbolo será considerado um comentário e não será executado pelo compilador.

```
//Este é um exemplo de comentário de linha única
```

Código 2.5.1

Comentários de várias linhas:

Os comentários de várias linhas são usados para adicionar informações mais longas, ocupando várias linhas de texto. Eles são representados pelo símbolo `/*` para iniciar o comentário e `*/` para encerrá-lo. Tudo entre esses símbolos será considerado um comentário.

```
/*
```

```
Este é um exemplo de comentário de várias linhas.
```

```
É Útil para documentar partes mais complexas do código.
```

```
*/
```

Código 2.5.2

```
// Solicita ao usuário para digitar dois números
Console.WriteLine("Digite o primeiro número:");
string input1 = Console.ReadLine();
double numero1 = Convert.ToDouble(input1);
Console.WriteLine("Digite o segundo número:");
string input2 = Console.ReadLine();
double numero2 = Convert.ToDouble(input2);
// Calcula a soma dos dois números
double soma = numero1 + numero2;
// Exibe o resultado
Console.WriteLine("A soma dos números é: " + soma);
```

Código 2.5.3

```
Saida: Digite o primeiro número:
Entrada: 5
Saida: Digite o segundo número:
Entrada: 6
Saida: A soma dos números é: 11
```

Execução código 2.5.3

No exemplo código 2.5.3, os comentários foram adicionados para explicar cada parte do programa de maneira mais detalhada. Eles descrevem o objetivo de cada seção do código, como a solicitação dos números ao usuário, a conversão dos valores de entrada, o cálculo da soma e a exibição do resultado.

Os comentários são opcionais e não afetam a execução do programa. Eles são usados principalmente para explicar a lógica do código,

fornece detalhes sobre a implementação, adicionar observações importantes e fornecer instruções para outros desenvolvedores.

2.6 Variáveis

As variáveis são elementos fundamentais na programação e são usadas para armazenar e manipular dados. Elas são como "recipientes" que podem conter diferentes tipos de valores, como números, textos, booleanos e muitos outros tipos de dados.

Uma variável é composta por três elementos principais:

- **Nome:** É o identificador dado à variável, que permite que você se refira a ela posteriormente em seu código. O nome da variável deve seguir as regras de nomenclatura da linguagem de programação que está sendo utilizada.
- **Tipo de dado:** Define o tipo de valor que a variável pode armazenar. Por exemplo, **int** para números inteiros, **string** para cadeias de caracteres (texto) e **bool** para valores booleanos (verdadeiro ou falso).
- **Valor:** É o dado atribuído à variável. Pode ser um valor literal, como **5** para uma variável de tipo **inteiro**, ou uma expressão que resulta em um valor, como **2 + 3** para uma variável do tipo **inteiro** que armazena o resultado da soma.

As variáveis permitem que você armazene valores e os utilize em diferentes partes do seu programa. Por exemplo, você pode atribuir um valor a uma variável e, em seguida, usar essa variável em cálculos, exibições de saída ou em qualquer outra parte do código onde o valor seja necessário.

Além disso, as variáveis podem ser modificadas, o que significa que você pode alterar o valor armazenado na variável ao longo do tempo, atribuindo um novo valor a ela.

Como criar uma variável:

- **Escolha um nome para a variável:** Comece escolhendo um nome significativo para a variável. O nome deve ser descritivo e refletir o propósito ou o conteúdo que a variável irá armazenar. Por exemplo, se você deseja armazenar a idade de uma pessoa, pode escolher o nome "idade" para a variável.
- **Determine o tipo de dado da variável:** Decida qual tipo de dado será armazenado na variável. Se você estiver armazenando a idade como um número inteiro, o tipo de dado será **int**. Outros exemplos comuns de tipos de dados incluem **string** para texto, **double** para números decimais e **bool** para valores booleanos.
- **Declare a variável:** Para criar uma variável, você precisa declará-la usando a seguinte sintaxe: **tipoDeDado nomeDaVariavel;**
- **Por exemplo:** Se você deseja criar uma variável chamada idade do tipo **int**, a declaração seria:

```
int idade;
```

Código 2.6.1

- **Atribua um valor à variável:** Após declarar a variável, você pode atribuir um valor a ela. Use o operador de atribuição (**=**) para atribuir um valor à variável.

```
int idade = 20;
```

Código 2.6.2

- **Acesse e utilize a variável:** Agora que a variável foi criada e recebeu um valor, você pode utilizá-la em seu código. Por exemplo, pode exibir o valor da variável usando **Console.WriteLine("")**:

```
int idade = 20;

Console.WriteLine("A idade é: " + idade);
```

Código 2.6.3

Saida: A idade é: 20

Execução código 2.6.3

Esses são os passos básicos para criar uma variável. Lembre-se de que, após a declaração inicial, você pode atribuir novos valores à variável, modificando seu conteúdo ao longo do programa, conforme necessário.

2.7 Estrutura de controle

O **if-else** é uma estrutura de controle que permite executar diferentes blocos de código com base em uma condição.

A estrutura **if-else** é composta por duas partes principais: o bloco **if** e o bloco **else**. O bloco **if** é executado se a condição especificada for avaliada como verdadeira. Já o bloco **else** é executado se a condição for avaliada como falsa.

```
bool condicao = true;

if(condicao){
    //Bloco de código a ser executado se a condição for
    verdadeira
} else {
    //Bloco de código a ser executado se a condição for
    falsa
}
```

Código 2.7.1

A variável **condicao** é uma expressão booleana que deve ser avaliada como **true** ou **false**. Se a condição for verdadeira, o bloco de código dentro do **if** será executado. Caso contrário, o bloco de código dentro do **else** será executado.

Aqui está um exemplo que utiliza a estrutura **if-else** para verificar se o valor da variável "idade" é maior ou igual a 18, ou menor que 18:

```
int idade = 20;
if (idade >= 18)
{
    Console.WriteLine("Você é maior de idade");
}
else
{
    Console.WriteLine("Você é menor de idade");
}
```

Código 2.7.2

Saida: Você é maior de idade

Execução código 2.7.2

Nesse exemplo, o programa verifica se a idade é maior ou igual a 18. Se a condição for verdadeira, a mensagem "Você é maior de idade" será exibida. Caso contrário, a mensagem "Você é menor de idade" será exibida.

O **if-else** permite que você controle o fluxo de execução do programa com base em condições específicas, proporcionando uma maior flexibilidade e tomada de decisões em seus programas.

2.8 Estrutura de repetição

A estrutura de repetição, também conhecida como loop, é uma construção fundamental na programação que permite executar um bloco de código repetidamente com base em uma condição. Ela ajuda a automatizar tarefas e simplificar a lógica de um programa.

Existem diferentes tipos de estruturas de repetição, sendo as mais comuns:

- **while:** O loop **while** repete um bloco de código enquanto uma condição especificada for verdadeira. Antes de cada repetição, a condição é verificada, e se for verdadeira, o bloco de código é executado. O loop continua até que a condição seja falsa.
- **do-while:** O loop **do-while** é semelhante ao **while**, mas a verificação da condição ocorre após a execução do bloco de código. Isso significa que o bloco de código é sempre executado pelo menos uma vez, e depois a condição é verificada para determinar se o loop deve continuar.
- **for:** O loop **for** é usado quando é conhecido o número exato de iterações que devem ser executadas. Ele consiste em uma inicialização, uma condição e uma operação de incremento ou decremento. O bloco de código é executado repetidamente enquanto a condição for verdadeira.
- **foreach:** O loop **foreach** é utilizado para iterar sobre uma coleção de elementos. Ele percorre cada elemento da coleção, atribuindo-o a uma variável temporária, e executa um bloco de código para cada elemento.

Cada tipo de estrutura de repetição possui sua própria sintaxe e aplicação adequada, dependendo das necessidades específicas do programa. Para iniciar nosso aprendizado sobre estruturas de repetição, vamos abordar com mais detalhes a estrutura **"for"**.

A sintaxe do "for" é geralmente composta por três partes: a inicialização, a condição de repetição e o incremento. A inicialização

define o valor inicial para a variável de controle do loop. A condição de repetição verifica se a repetição deve continuar ou não. E o incremento atualiza o valor da variável de controle a cada iteração.

Dentro do bloco de código do **"for"**, podemos escrever as instruções que desejamos executar repetidamente. A variável de controle do loop nos permite acompanhar o progresso e realizar ações diferentes com base nesse valor.

Aqui está a sintaxe completa do **for**:

```
for(inicialização; condição; incremento){  
    //Bloco de código a ser repetido  
}
```

Código 2.8.1

Aqui está o significado de cada parte da estrutura **for**:

- A **inicialização** ocorre apenas uma vez no início do loop. É usada para declarar e atribuir um valor inicial à variável de controle. Geralmente, usamos **int i = 0** para iniciar a variável **i** com o valor 0.
- A **condição** é avaliada antes de cada iteração do loop. Se a condição for verdadeira, o bloco de código dentro do **for** será executado. Se for falsa, o loop será encerrado. Por exemplo, podemos usar **i < 5** como condição para executar o loop enquanto **i** for menor que 5.
- O **incremento** (ou decremento) é aplicado após cada iteração do loop. Geralmente, usamos **i++** para incrementar a variável **i** em 1 a cada iteração. No entanto, também podemos usar outros incrementos ou decrementos, como **i += 2** para aumentar **i** em 2 a cada iteração.

Dentro do bloco de código do **for**, podemos incluir as instruções que desejamos repetir. Por exemplo, podemos imprimir valores, realizar

cálculos ou executar outras operações com base nas necessidades do nosso programa.

Vamos dar um exemplo prático para ilustrar o uso do **for**:

```
for(int i = 0; i < 5; i++){  
    Console.WriteLine("Iteração" + i);  
}
```

Código 2.8.2

```
Saida: Iteração0  
Saida: Iteração1  
Saida: Iteração2  
Saida: Iteração3  
Saida: Iteração4
```

Execução código 2.7.2

Neste exemplo, o loop **for** é executado cinco vezes. A variável **i** é inicializada com 0, e a condição **i < 5** é verificada antes de cada iteração. Dentro do bloco de código do **for**, é exibida a mensagem "Iteração" seguida do valor atual de **i**. A cada iteração, o valor de **i** é incrementado em 1.

A estrutura **for** é especialmente útil quando sabemos exatamente quantas vezes queremos repetir um bloco de código. Ela nos permite controlar o comportamento da variável de controle e simplificar a lógica do nosso programa.

2.9 Desafios

1 - Escreva um programa que solicite ao usuário um número e exiba a mensagem "Número par" se o número for par, ou "Número ímpar" se for ímpar.

2 - Escreva um programa que solicite um número ao usuário. Em seguida, o programa deve verificar se o número é positivo, negativo ou zero, exibindo a mensagem correspondente.

3 - Elabore um programa que peça ao usuário para digitar sua nota em uma prova. Com base na nota fornecida, exiba a seguinte classificação:

- Se a nota for maior ou igual a 7, exiba "Aprovado".
- Se a nota for maior ou igual a 5 e menor que 7, exiba "Recuperação".
- Caso contrário, exiba "Reprovado".

4 - Escreva um programa que faça uma contagem regressiva a partir de um número fornecido pelo usuário até zero. Em cada iteração, o programa deve exibir o número atual da contagem.

5 - Crie um programa que solicite ao usuário um número inteiro positivo e calcule a soma de todos os números pares de 1 até o número fornecido. Exiba o resultado da soma.

6 - Crie um programa que solicite ao usuário um número inteiro e exiba a tabuada desse número, multiplicando-o pelos números de 1 a 10.

3. Tipos de dados, operadores e saída de dados

“Medir o progresso da programação por linhas de código é como medir o progresso da construção de aeronaves por peso.”

3.1 Tipos de dados

Tipo de dados é uma classificação que define o tipo de informação que pode ser armazenada ou manipulada em um programa de computador. Existem diversos tipos de dados utilizados em linguagens de programação.

- **Números inteiros (integer):** Representam valores numéricos inteiros, -3, 0, 42, 1000, etc.
- **Números de ponto flutuante (float):** Representam valores numéricos com parte decimal. Podem incluir números fracionários ou números muito grandes ou pequenos. Exemplos de valores de ponto flutuante são -3.14, 0.5, 1.23e-5 (notação científica), etc.
- **Texto (string):** Representa sequências de caracteres, como letras, números e símbolos. As strings são frequentemente utilizadas para armazenar palavras, frases, endereços de e-mail, etc. Exemplos de strings são "Olá, mundo!", "Aprenda a programar!", "12345", "1.3,23.4.5" etc. Em muitas linguagens de programação, as strings são delimitadas por aspas duplas (" ") ou aspas simples (' ').
- **Valores booleanos (boolean):** Representam dois estados lógicos, verdadeiro (true) ou falso (false).

Essa tabela inclui os tipos de dados mais comumente usados:

Tipo de Dado	Descrição	Exemplo
int	Números inteiros	int idade = 25;
double	Números de ponto flutuante (dupla precisão)	double salario = 2500.50;
string	Sequências de caracteres	string nome = "João";
bool	Valores booleanos (verdadeiro/falso)	bool valido = true;
char	Caractere Unicode	char genero = 'M';
DateTime	Data e hora	DateTime dataNascimento = new DateTime(1990, 5, 10);
decimal	Números decimais de alta precisão	decimal valor = 10.99m;
float	Números de ponto flutuante (precisão simples)	float altura = 1.75f;
long	Números inteiros longos	long populacao = 8000000000;
byte	Números inteiros sem sinal (0 a 255)	byte nivel = 3;

Além desses tipos, existem também outros tipos de dados, como arrays (vetores), listas, dicionários, entre outros que veremos futuramente.

3.2 Casting

Casting, em programação, é a conversão de um tipo de dado para outro tipo. É uma operação comum quando trabalhamos com diferentes tipos de dados e é usada para adequar um valor a um tipo específico.

Existem dois tipos principais de casting: casting implícito e casting explícito.

Casting Implícito

O casting implícito ocorre quando uma conversão de tipo é realizada automaticamente pelo compilador sem a necessidade de uma instrução explícita por parte do programador. Isso geralmente acontece quando não há perda de dados ou quando o tipo de destino é capaz de representar todos os valores do tipo de origem. Por exemplo, um número inteiro pode ser atribuído a uma variável de ponto flutuante sem a necessidade de um casting explícito, pois não há perda de dados.

```
int x = 10;  
  
double y = x; //Casting implícito de int para double
```

Código 3.2.1

Casting explícito

Casting explícito, também conhecido como coerção explícita (**int**), (**char**), (**double**) etc. É quando solicitamos a conversão de um tipo de dado para outro. O casting explícito é necessário em situações em que pode haver perda de dados ou quando a conversão não é realizada automaticamente pelo compilador.

```
double a = 3.14;  
  
int b = (int)a; //Casting explícito de double para int  
Console.WriteLine(b);
```

Código 3.2.2

Saida: 3

Execução código 3.2.2

No exemplo código 3.2.2, o valor decimal 3.14 é convertido explicitamente para o inteiro 3 usando o operador de cast (**int**).

Ao realizar um casting explícito, é importante ter cuidado, pois pode ocorrer perda de dados ou comportamentos indesejados. Por exemplo, ao converter um número de ponto flutuante para um número inteiro, a parte decimal é truncada e ocorre perda de informações.

Casting de char

Casting de char é usado para converter um caractere em seu valor numérico equivalente (código ASCII ou Unicode) ou vice-versa.

```
char c = 'A';  
int ascii = (int)c;  
Console.WriteLine(ascii);
```

Código 3.2.3

Saida: 65

Execução código 3.2.3

No exemplo código 3.2.3, o caractere 'A' é convertido em seu valor ASCII correspondente (65) usando o casting (**int**).

```
int ascii = 65;  
char c = (char)ascii;  
Console.WriteLine(c);
```

Código 3.2.4

Saida: A

Saida código 3.2.4

No exemplo código 3.2.5, o valor inteiro 65 é convertido no caractere correspondente 'A' usando o casting (**char**).

3.3 Tabela ASCII

A tabela ASCII (American Standard Code for Information Interchange) é uma codificação que associa um número único a cada caractere utilizado em computadores e dispositivos de comunicação que usam o conjunto de caracteres em inglês. Ela foi desenvolvida no início dos anos 1960 para padronizar a representação de caracteres e facilitar a comunicação e o processamento de texto em sistemas de computadores.

A tabela ASCII é amplamente utilizada em programação e na representação de texto em sistemas computacionais. Cada caractere é representado internamente como um número inteiro e pode ser manipulado através de seus valores correspondentes na tabela ASCII.

Por exemplo, na tabela ASCII, o valor 65 corresponde ao caractere 'A', o valor 97 corresponde ao caractere 'a', o valor 48 corresponde ao dígito '0' e assim por diante.

3.4 Variáveis e constantes

Até o momento, temos utilizado várias variáveis em diversos exemplos, porém, agora iremos nos aprofundar ainda mais no tema.

Variáveis e constantes são elementos fundamentais na programação, usados para armazenar e representar dados durante a execução de um programa. Ambos têm um papel essencial, mas diferem em sua natureza e uso.

Variáveis

- Uma variável é um espaço de armazenamento nomeado que pode conter um valor que pode ser alterado durante a execução do programa.

- Elas são usadas para armazenar dados que podem variar ou serem atualizados ao longo do tempo.
- As variáveis são declaradas com um tipo específico (por exemplo, int, double, string) e um nome único.
- O valor de uma variável pode ser atribuído e modificado durante a execução do programa.
- Através da manipulação de variáveis, é possível armazenar e processar dados dinamicamente, tomar decisões, executar loops e resolver problemas complexos.
- Criação e modificação de uma variável

```
int idade = 10;//Criando  
idade = 20;//Modificando  
Console.WriteLine(idade);
```

Código 3.4.1

```
Saida: 20
```

Execução código 3.4.1

Constantes

- Constante é um valor que é atribuído e não pode ser alterado durante a execução do programa.
- Elas são usadas para representar valores que não devem ser modificados.
- As constantes são declaradas com um tipo específico (por exemplo, int, double, string) e um nome único.

- Uma vez que uma constante recebe um valor, esse valor não pode ser alterado posteriormente.
- Criação de uma constante

```
const double PI = 3.14;
```

Código 3.4.2

3.5 Regras para criação de variáveis/constantes

Ao criar variáveis/constantes, é importante seguir algumas regras e convenções para garantir um código legível, consistente e livre de erros.

- **Nomes válidos:** Os nomes de variáveis/constantes devem começar com uma letra ou sublinhado (_). Eles podem conter letras, números e sublinhados, mas não podem conter espaços ou caracteres especiais. Além disso, os nomes são sensíveis a caracteres maiúsculos e minúsculos, o que significa que "valor" e "Valor" seriam considerados nomes de variáveis diferentes.

```
int valor = 5;  
  
int Valor = 3;
```

Código 3.5.1

- **Convenção de nomenclatura:** É comum seguir convenções de nomenclatura para facilitar a leitura e entendimento do código. Uma convenção comum é o uso de camel case, onde a primeira letra é minúscula e as palavras subsequentes têm a primeira letra maiúscula, por exemplo: "idadeDoUsuario", "nomeCompleto".
- **Palavras reservadas:** Evite utilizar palavras reservadas da linguagem de programação como nomes de variáveis, pois elas têm significado especial e são usadas para fins específicos na linguagem. Por exemplo, em C#, **int** é uma palavra reservada e

não pode ser usada como nome de variável.

- **Significado descritivo:** Dê nomes significativos às variáveis que descrevam o propósito ou o conteúdo que elas representam. Isso torna o código mais compreensível e facilita a manutenção. Evite nomes genéricos como **var1**, **x**, **temp**, que não fornecem informações claras sobre a finalidade da variável.
- **Tamanho máximo:** Verifique as restrições de tamanho máximo de nome de variável impostas pela linguagem de programação que você está utilizando. A maioria das linguagens tem limites para o comprimento dos nomes de variáveis, em C# são 255.

As regras específicas podem variar dependendo da linguagem de programação utilizada, mas as diretrizes gerais são amplamente aplicáveis. É sempre bom consultar a documentação da linguagem específica para obter informações mais detalhadas sobre as regras de criação de variáveis.

3.6 Enumeradores

Enumeradores, também conhecidos como enums, são tipos de dados que permitem definir um conjunto de constantes nomeadas. Os enumeradores são úteis quando desejamos representar um conjunto fixo de valores que são mutuamente exclusivos.

Definição de enum

- Os enumeradores são definidos usando a palavra-chave **enum** seguida pelo nome do enum e seus valores constantes entre chaves.
- Cada valor constante é separado por vírgulas e pode ser atribuído explicitamente a um valor inteiro ou deixado para que a linguagem atribua valores automáticos sequenciais a partir de 0.

- Os nomes das constantes dentro de um enum são exclusivos dentro do escopo do **enum**.

```
enum DiasDaSemana
{
    Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado
}
```

Código 3.6.1

Uso de enum

- Os valores de um enum podem ser acessados usando o nome do enum seguido pelo valor desejado, separados por ponto.
- Os valores de um enum são tratados como constantes e podem ser usados em operações, comparações e atribuições.
- Os valores de um enum são tipados e a linguagem garante que apenas valores válidos do enum sejam usados.

```
DiasDaSemana dia = DiasDaSemana.Segunda;
Console.WriteLine(dia);

enum DiasDaSemana
{
    Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado
}
```

Código 3.6.2

```
Saida: Segunda
```

Execução código 3.6.2

Atribuição de valor inicial

Agora, vamos atribuir manualmente valores personalizados ao enumerador, existem duas formas principais de atribuir valores aos membros do enumerador.

```
enum DiasDaSemana
{
    Domingo = 1,
    Segunda,    // 2
    Terca,      // 3
    Quarta,     // 4
    Quinta,     // 5
    Sexta,      // 6
    Sabado      // 7
}
```

Código 3.6.3

Neste exemplo, o valor de **Domingo** foi definido como **1**, e os demais membros do enumerador são atribuídos automaticamente a valores sequenciais (2, 3, 4, ...).

Atribuição de valores específicos

Nesta abordagem, atribuímos valores específicos a cada membro do enumerador. Isso nos dá controle total sobre os valores associados.

```
enum DiasDaSemana
{
    Domingo = 10,
    Segunda = 20,
    Terca = 30,
    Quarta = 40,
    Quinta = 50,
    Sexta = 60,
    Sabado = 70
}
```

Código 3.6.4

No código 3.6.4, atribuímos manualmente valores específicos a cada membro do enumerador, e esses valores não precisam ser sequenciais ou ter qualquer relação entre si.

Lembre-se de que os valores do enumerador devem ser exclusivos dentro do mesmo enumerador, mas você pode ter valores repetidos em enumeradores diferentes.

Acessando um Enumerador através de um Valor Específico.

Agora que aprendemos a atribuir valores personalizados aos membros do enumerador, é importante entender como acessar um membro específico do enumerador a partir de um valor inteiro. Suponha que temos o valor inteiro 5, que representa a Sexta-feira.


```

DiasDaSemana diasDaSemana = (DiasDaSemana)5;
Console.WriteLine(diasDaSemana);

public enum DiasDaSemana
{
    Domingo,
    Segunda,
    Terca,
    Quarta,
    Quinta,
    Sexta,
    Sabado
}

```

Código 3.6.5

Nesse exemplo, usamos a conversão explícita (**cast**) para converter o valor inteiro **5** para o tipo do enumerador **DiasDaSemana**. O compilador permitirá esse **cast**, pois o enumerador possui um membro cujo valor é **5**, que é **Sexta**. Após a conversão, a variável dia é atribuída com o valor DiasDaSemana.Sexta, que representa o membro do enumerador correspondente.

Saida: Sexta

Execução código 3.6.5

3.7 Operadores

Operadores são símbolos ou palavras-chave que realizam operações em operandos (valores ou variáveis) para produzir um resultado. Eles

são usados para executar cálculos, comparações, atribuições, entre outras operações. Os operadores podem ser classificados em diferentes categorias, incluindo operadores aritméticos, operadores de comparação, operadores lógicos, operadores de atribuição e operadores de incremento/decremento.

Operadores Aritméticos

- São usados para realizar operações matemáticas básicas, como adição, subtração, multiplicação, divisão e módulo (resto da divisão).
- Exemplos de operadores aritméticos: + (adição), - (subtração), * (multiplicação), / (divisão), % (módulo).

```
int a = 5;
int b = 3;
int soma = a + b;
int subtracao = a - b;
int multiplicacao = a * b;
int divisao = a / b;
int modulo = a % b;
Console.WriteLine("Valor de 'a': " + a);
Console.WriteLine("Valor de 'b': " + b);
```



```
Console.WriteLine("Valor da soma: " + soma);  
Console.WriteLine("Valor da subtração: " +  
subtracao);  
Console.WriteLine("Valor da multiplicação: " +  
multiplicacao);  
Console.WriteLine("Valor da divisão: " + divisao);  
Console.WriteLine("Valor do módulo: " + modulo);
```

Código 3.7.1

```
Valor de 'a': 5  
Valor de 'b': 3  
Valor da soma: 8  
Valor da subtração: 2  
Valor da multiplicação: 15  
Valor da divisão: 1  
Valor do módulo: 2
```

Execução código 3.7.1

Operadores de Comparação

- São usados para comparar valores e verificar condições de igualdade, desigualdade, maior que, menor que, etc.
- Exemplos de operadores de comparação: == (igual a), != (diferente de), > (maior que), < (menor que), >= (maior ou igual a), <= (menor ou igual a).

```
int a = 5;
int b = 3;
bool igual = a == b;
bool diferente = a != b;
bool maiorQue = a > b;
bool menorQue = a < b;
bool maiorOuIgual = a >= b;
bool menorOuIgual = a <= b;
Console.WriteLine("Valor de 'igual': " + igual);
Console.WriteLine("Valor de 'diferente': " +
diferente);
Console.WriteLine("Valor de 'maiorQue': " +
maiorQue);
Console.WriteLine("Valor de 'menorQue': " +
menorQue);
Console.WriteLine("Valor de 'maiorOuIgual': " +
maiorOuIgual);
Console.WriteLine("Valor de 'menorOuIgual': " +
menorOuIgual);
```

Código 3.7.2

```
Saida: Valor de 'igual': False
Saida: Valor de 'diferente': True
Saida: Valor de 'maiorQue': True
Saida: Valor de 'menorQue': False
Saida: Valor de 'maiorOuIgual': True
Saida: Valor de 'menorOuIgual': False
```

Execução código 3.7.2

Operadores Lógicos

- São usados para combinar ou inverter condições lógicas e avaliar expressões booleanas.

Exemplos de operadores lógicos: **&&** (AND lógico), **||** (OR lógico), **!** (NOT lógico), **^** (XOR lógico).

```
bool a = true;
bool b = false;
bool resultadoAND = a && b;
bool resultadoOR = a || b;
bool resultadoNOTa = !a;
bool resultadoNOTb = !b;
bool resultadoXOR = a ^ b;
Console.WriteLine("Valor de 'resultadoAND': " +
resultadoAND);
```



```
Console.WriteLine("Valor de 'resultadoOR': " +  
resultadoOR);  
  
Console.WriteLine("Valor de 'resultadoNOTa': " +  
resultadoNOTa);  
  
Console.WriteLine("Valor de 'resultadoNOTb': " +  
resultadoNOTb);  
  
Console.WriteLine("Valor de 'resultadoXOR': " +  
resultadoXOR);
```

Código 3.7.3

```
Saida: Valor de 'resultadoAND': False  
Saida: Valor de 'resultadoOR': True  
Saida: Valor de 'resultadoNOTa': False  
Saida: Valor de 'resultadoNOTb': True  
Saida: Valor de 'resultadoXOR': True
```

Execução código 3.7.3

Operadores de Atribuição

O operador **+=** é um exemplo de operador composto de atribuição. Ele combina uma operação de adição com uma atribuição em uma única expressão.

A sintaxe do operador **+=** é a seguinte: **variável += expressão;**

- A expressão à direita do operador **+=** é avaliada. Isso pode ser uma variável, uma constante ou uma expressão mais complexa.
- O valor resultante da expressão é adicionado ao valor atual da variável à esquerda do operador **+=**.

- O resultado da adição é atribuído à variável à esquerda do operador **+=**.
- Se **a** for igual a 5 e **b** for igual a 3, a expressão **a += b**; é equivalente a **a = a + b**;
- Nesse caso, o valor atual de **a** é 5. A expressão **a + b** resulta em 8 (5 + 3).
- Em seguida, o valor 8 é atribuído à variável **a**, atualizando seu valor para 8.
- Exemplos de operadores de atribuição: **=** (atribuição simples), **+=** (atribuição com adição), **-=**, ***=**, **/=**, entre outros.

```
int a = 5;
int b = 3;
a += b;
Console.WriteLine("Valor de 'a' após a adição: " +
a);
a -= b;
Console.WriteLine("Valor de 'a' após a subtração:
" + a);
a *= b;
Console.WriteLine("Valor de 'a' após a
multiplicação: " + a);
```



```
a /= b;  
  
Console.WriteLine("Valor de 'a' após a divisão: " + a);  
  
a %= b;  
  
Console.WriteLine("Valor de 'a' após o módulo: " + a);
```

Código 3.7.4

```
Saida: Valor de 'a' após a adição: 8  
Saida: Valor de 'a' após a subtração: 5  
Saida: Valor de 'a' após a multiplicação: 15  
Saida: Valor de 'a' após a divisão: 5  
Saida: Valor de 'a' após o módulo: 2
```

Execução código 3.7.4

Operadores de Incremento/Decremento

Os operadores de incremento e decremento são usados para aumentar ou diminuir o valor de uma variável em uma unidade. Eles são representados pelos símbolos **++** e **--**.

Incremento

O operador **++** pode ser usado de duas maneiras: como um operador pré-fixado ou como um operador pós-fixado. Quando usado como um operador pré-fixado, ele aumenta o valor da variável antes de retornar o valor da variável. Quando usado como um operador pós-fixado, ele retorna o valor da variável antes de aumentar o valor da variável.


```
int a = 10;  
int b = ++a;  
Console.WriteLine(b);
```

Código 3.7.5

O código 3.7.5 irá aumentar o valor de **a** para **11** e, em seguida, atribuir o valor de **a** para **b**, resultando em **b = 11**.

Saida: 11

Execução código 3.7.5

Decremento

O operador **--** pode ser usado de forma semelhante ao operador **++**. Quando usado como um operador pré-fixado, ele diminui o valor da variável antes de retornar o valor da variável. Quando usado como um operador pós-fixado, ele retorna o valor da variável antes de diminuir o valor da variável.

```
int a = 10;  
int b = --a;  
Console.WriteLine(b);
```

Código 3.7.6

O código 3.7.6 irá diminuir o valor de **a** para **9** e, em seguida, atribuir o valor de **a** para **b**, resultando em **b = 9**.

Saida: 9

Execução código 3.7.6

Além desses operadores, existem outros, como operadores de acesso a membros (**.**, **->**), operadores de deslocamento, operadores ternários, entre outros.

3.8 Tabela verdade

Mostra todas as combinações possíveis de valores de entrada e o resultado correspondente de uma expressão lógica. A tabela verdade é amplamente usada em lógica booleana para analisar o comportamento de operadores lógicos e expressões condicionais.

Tabela Verdade para o operador AND (&&):

A	B	A && B
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

Tabela Verdade para o operador OR (||):

A	B	A B
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

Tabela Verdade para o operador NOT (!):

A	!A
Verdadeiro	Falso
Falso	Verdadeiro

Tabela Verdade para o operador XOR:

A	B	A XOR B
Verdadeiro	Verdadeiro	Falso
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

3.9 Concatenação

Concatenação é o processo de combinar ou juntar duas ou mais strings em uma única string. Concatenação é amplamente utilizada para criar strings mais longas, incorporando valores de variáveis, constantes ou outros textos em uma única sequência de caracteres.

A concatenação pode ser realizada de diferentes maneiras, aqui estão algumas formas comuns de realizar a concatenação

Operador de Concatenação

- Muitas linguagens de programação possuem um operador específico para a concatenação de strings, geralmente representado pelo símbolo +.
- Ao usar o operador de concatenação, você pode unir duas ou mais strings, variáveis ou constantes para criar uma nova string concatenada.

```
string nome = "João";
int idade = 18;

string mensagem = "Olá, " + nome + "! Você tem " +
idade + " anos.";

Console.WriteLine(mensagem);
```

Código 3.9.1

```
Saida: Olá, João! Você tem 18 anos.
```

Execução código 3.9.1

Interpolação de Strings

- Sintaxe que permite inserir variáveis ou expressões diretamente em uma string sem a necessidade de operadores de concatenação.
- A interpolação de strings geralmente é feita usando um marcador especial, como \$ ou \${}, para indicar os trechos a serem interpolados.

No exemplo código 3.8.2, utilizamos a interpolação de string ao colocar as variáveis **nome** e **idade** dentro de chaves {} dentro de uma string delimitada por \$. Isso permite que as variáveis sejam inseridas diretamente na string sem a necessidade de operadores de concatenação ou conversões explícitas. As expressões dentro das chaves são avaliadas e seus valores são substituídos na string final.

```
string nome = "João";  
  
int idade = 25;  
  
string mensagem = $"Olá, {nome}! Você tem {idade}  
anos.";  
  
Console.WriteLine(mensagem);
```

Código 3.9.2

```
Saida: Olá, João! Você tem 25 anos.
```

Execução código 3.9.2

A interpolação de string oferece uma maneira mais concisa e legível de construir strings com valores de variáveis ou expressões incorporados. Ela é especialmente útil quando você precisa construir strings complexas com várias substituições de valores.

3.10 Entrada de dados

Entrada de dados do usuário pode ser obtida usando a classe **Console** e seus métodos **ReadLine()** e **ReadKey()**.

Entrada de dados com ReadLine()

O método **ReadLine()** é usado para ler uma linha de entrada do usuário. Ele espera que o usuário digite uma sequência de caracteres seguida pela tecla Enter. O valor digitado pelo usuário é retornado como uma string.

```
Console.WriteLine("Digite seu nome:");  
  
string nome = Console.ReadLine();  
  
Console.WriteLine("Seu nome é " + nome);
```

Código 3.10.1

Neste exemplo, o programa solicita ao usuário que digite seu nome. O método **Console.ReadLine()** aguarda até que o usuário digite o nome e pressione Enter. O valor digitado pelo usuário é armazenado na variável **nome** como uma string.

```
Entrada: Digite seu nome:  
  
Saída: Seu nome é João.
```

Execução código 3.10.1

Entrada de dados com ReadKey()

O método **ReadKey()** retorna um objeto do tipo **ConsoleKeyInfo**, que contém informações sobre a tecla pressionada pelo usuário. Esse objeto possui várias propriedades que podem ser utilizadas para obter informações específicas sobre a tecla pressionada.

- **Key**: Essa propriedade retorna o valor da tecla pressionada como um valor do tipo **ConsoleKey**. Por exemplo, se o usuário pressionar a tecla "A", a propriedade **Key** retornará '**A**'.

- **KeyChar**: Essa propriedade retorna o caractere correspondente à tecla pressionada como um valor do tipo **char**. Por exemplo, se o usuário pressionar a tecla "A", a propriedade **KeyChar** retornará o caractere 'A'.
- **Modifiers**: Essa propriedade retorna um valor do tipo **ConsoleModifiers** que representa as teclas de modificação (como Shift, Alt, Ctrl) que foram pressionadas junto com a tecla principal. Ela permite verificar se alguma tecla modificadora foi pressionada ao mesmo tempo que a tecla principal. Por exemplo, quando o usuário pressionar a combinação de teclas "Shift + A", a propriedade **Modifiers** retornará **Shift**. No entanto, se o usuário não pressionar a tecla Shift, a propriedade **Modifiers** retornará **0**.

Neste exemplo, o programa solicita ao usuário que pressione qualquer tecla. O método **Console.ReadKey()** aguarda até que o usuário pressione uma tecla. O objeto **ConsoleKeyInfo** retornado contém informações sobre a tecla pressionada, que são usadas para exibir a tecla no console.

```
Console.WriteLine("Pressione qualquer tecla...");  
  
ConsoleKeyInfo teclaPressionada = Console.ReadKey();  
  
Console.WriteLine("Tecla pressionada: " +  
teclaPressionada.Key);  
  
Console.WriteLine("Caractere correspondente: " +  
teclaPressionada.KeyChar);  
  
Console.WriteLine("Teclas de modificação: " +  
teclaPressionada.Modifiers);
```

Código 3.10.2

Entrada: Pressione qualquer tecla: A

Saida: Tecla pressionada: A

Saida: Caractere correspondente: A

Saida: Teclas de modificação: Shift

Execução código 3.10.2

Esses métodos são úteis para obter interação do usuário e entrada de dados durante a execução de um programa. Com o **ReadLine()**, você pode solicitar ao usuário que digite informações em uma linha completa, enquanto o **ReadKey()** permite a obtenção de uma única tecla pressionada.

3.11 Covert.To

Existe outro método de conversão de dados, chamado **convert**. O **convert** é semelhante ao casting apresentado no **capítulo 3.2**.

Mostrarei como o **convert** funciona e, em seguida, farei uma recapitulação do casting e da diferença entre eles.

Convert

- A conversão usando **Convert** é usada para converter entre tipos de dados diferentes, onde não há uma relação direta entre eles.

```
string numeroTexto = "123";  
int numeroInteiro = Convert.ToInt32(numeroTexto);
```

Código 3.11.1

Neste exemplo, usamos **Convert.ToInt32()** para converter uma string em um valor inteiro. O método **Convert.ToInt32()** trata internamente da conversão e valida se a string pode ser convertida em um valor inteiro.

```
string numeroTexto = "123";  
int numeroInteiro = Convert.ToInt32(numeroTexto);  
Console.WriteLine(numeroInteiro + 5);
```

Código 3.11.2

Saida: 128

Execução código 3.11.2

No exemplo código 3.11.2 efetuamos a soma da variável **(numeroInteiro + 5)**, obtendo o resultado de **128**, dessa forma, o valor original da variável **numeroTexto** foi convertido em um número inteiro antes da soma com o valor 5. O resultado da soma, **128**, indica que a conversão foi bem-sucedida.

Casting

- A conversão casting é usada principalmente para converter entre tipos relacionados, onde há uma relação direta e segura de conversão entre eles.
- O casting é realizado usando o operador de cast (**tipo**), onde você especifica o tipo de dado desejado entre parênteses antes do valor a ser convertido.
- Neste exemplo, usamos o casting (**int**) para converter um valor decimal em um valor inteiro. O casting explicitamente informa ao compilador que queremos tratar o valor decimal como um valor inteiro, mesmo sabendo que pode haver perda de dados.

```
double numeroDecimal = 10.5;  
int numeroInteiro = (int)numeroDecimal;  
Console.WriteLine(numeroInteiro);
```

Código 3.11.3

Saida: 10

Execução código 3.11.3

O `convert` e o `casting` são usados para converter dados de um tipo para outro. A principal diferença entre eles é que o `convert` pode ser usado para converter dados entre tipos diferentes, enquanto o `casting` só pode ser usado para converter dados entre tipos relacionados.

Por exemplo, o `convert` pode ser usado para converter uma string em um número, enquanto o `casting` só pode ser usado para converter um número em um tipo de dado numérico mais específico, como um `int` ou um `float`.

3.12 Alias

Alias é um nome alternativo que podemos atribuir a um elemento existente, como um tipo de dado, uma classe, uma função ou uma variável. O alias permite que você se refira a esse elemento utilizando um nome diferente, o que pode tornar o código mais legível, claro e expressivo.

Você pode criar aliases para vários elementos, incluindo tipos de dados, classes, namespaces e até mesmo para elementos específicos dentro de uma classe, como métodos ou propriedades.

```
using Saudacao = System.String;

Saudacao saudacaoPersonalizada = "Olá, seja bem-vindo!";

Console.WriteLine(saudacaoPersonalizada);
```

Código 3.12.1

No exemplo código 3.12.1, criamos um alias chamado **Saudacao** para o tipo de dado **System.String**, que é a representação de string. Em seguida, declaramos uma variável **saudacaoPersonalizada** do tipo **Saudacao** e atribuímos a ela a string **"Olá, seja bem-vindo!"**.

```
Saida: Olá, seja bem-vindo!
```

Execução código 3.12.1

Os aliases são definidos utilizando a palavra-chave **using**, seguida da diretiva **=** e, em seguida, o nome original e o alias desejado.

Existem palavras-chave reservadas que são usadas para representar diferentes tipos de dados, como **string**, **int**, **float**, **bool** e assim por diante. Essas palavras-chave são nativas da linguagem e não precisam de aliases.

```
string nome = "João";  
int idade = 25;  
float altura = 1.75f;  
bool ativo = true;
```

Código 3.12.2

Portanto, para tipos de dados como **string**, **int**, **float**, **bool**, não é necessário utilizar aliases, pois as palavras-chave reservadas já representam esses tipos de dados nativamente.

3.13 Desafios

1 - Você recebeu um número decimal representado por um tipo de dado **double**, mas precisa exibi-lo como um número inteiro.

2 - Você está desenvolvendo um programa de agendamento e precisa criar um sistema para representar os diferentes meses do ano. Utilize enumeradores para criar os meses (Janeiro, Fevereiro, Março, Abril, Maio, Junho, Julho, Agosto, Setembro, Outubro, Novembro, Dezembro).

3 - Você recebeu três palavras e precisa criar um programa que as concatene em uma única **string**.

4 - Você recebeu dois números inteiros e precisa realizar cálculos entre eles, sendo **A = 23**, **B = 15**;

- A. Soma: $A + B$
- B. Subtração: $A - B$
- C. Divisão: A / B
- D. Multiplicação: $A * B$

5 - Você recebeu a idade e a altura de uma pessoa e precisa realizar algumas verificações utilizando operadores lógicos.

- A. Verifique se a idade é maior ou igual a 18 anos e a altura é maior ou igual a 1.60 metros. Armazene o resultado em uma variável booleana.
- B. Verifique se a idade é menor que 12 anos ou a altura é menor que 1.40 metros. Armazene o resultado em uma segunda variável booleana.

4. Instrução de decisão e repetição

“Software é um ótimo combinador de trabalho humano.”

4.1 Estrutura de controle

Nos capítulos anteriores, fizemos algumas considerações sobre estruturas de decisão. No entanto, é hora de aprofundarmos ainda mais.

A decisão binária é fundamentalmente baseada na avaliação de expressões booleanas, que retornam um valor verdadeiro ou falso. Essas expressões podem ser combinadas usando operadores lógicos, como o "E" lógico (&&), o "OU" lógico (||) e o "NÃO" lógico (!), para criar condições mais complexas.

```
int idade = 25;
if (idade >= 18)
{
    Console.WriteLine("A pessoa é de maior.");
}
else
{
    Console.WriteLine("A pessoa é de menor");
}
```

Código 4.1.1

Nesse exemplo, se a idade for igual ou maior que **18**, o bloco de código dentro do **if** será executado e exibirá a mensagem "**A pessoa é maior de idade.**" Caso contrário, se a idade for menor que **18**, o bloco de código dentro do **else** será executado e exibirá a mensagem "**A pessoa é menor de idade.**"

Saida: A pessoa é de maior.

Execução código 4.1.1

Além disso, também existe o operador ternário, que permite realizar tomadas de decisão binárias de forma concisa em uma única linha de código.

```
condicao ? expressao_verdadeira : expressao_falsa;
```

Exemplo

Aqui está um exemplo usando o operador ternário para tomar uma decisão binária:

```
int numero = 10;  
  
string resultado = (numero > 0) ? "O número é  
positivo." : "O número é negativo ou zero.";  
  
Console.WriteLine(resultado);
```

Código 4.1.2

Nesse exemplo, se o número for maior que **0**, a expressão "**O número é positivo.**" será atribuída à variável **resultado**. Caso contrário, a expressão "**O número é negativo ou zero.**" será atribuída.

```
Saida: O número é positivo.
```

Execução código 4.1.2

A tomada de decisão binária usando **if-else** e o operador ternário é fundamental para controlar o fluxo de execução do programa, permitindo que diferentes caminhos sejam seguidos com base em condições específicas. Essas estruturas são amplamente utilizadas para implementar lógica condicional e criar algoritmos flexíveis.

Decisão binária encadeada

Uma decisão binária encadeada, também conhecida como estrutura de decisão encadeada, é uma extensão da tomada de decisão binária tradicional. Ela permite que múltiplas condições sejam avaliadas sequencialmente e diferentes blocos de código sejam executados com base no resultado dessas condições.

A decisão binária encadeada pode ser implementada usando uma combinação de instruções **if-else**. Cada instrução **if** é seguida por uma condição que é avaliada e, se verdadeira, o bloco de código correspondente é executado. Se a condição não for verdadeira, a próxima instrução **if** é avaliada. O último bloco, associado ao **else**, é executado se todas as condições anteriores forem falsas.

```
int numero = 5;
if (numero < 0)
{
    Console.WriteLine("O número é negativo.");
}
else if (numero == 0)
{
    Console.WriteLine("O número é igual a zero.");
}
else
{
    Console.WriteLine("O número é positivo.");
}
```

Código 4.1.3

Nesse exemplo, se o número for negativo, o bloco de código dentro do primeiro **if** será executado e a mensagem **"O número é negativo."** será exibida. Se a primeira condição for falsa, a próxima condição é avaliada. Se o número for igual a zero, o bloco de código dentro do **else if** será executado e a mensagem **"O número é igual a zero."** será exibida. Se nenhuma das condições anteriores for verdadeira, o bloco de código dentro do **else** será executado e a mensagem **"O número é positivo."** será exibida.

Saida: O número é positivo.

Execução código 4.1.3

A estrutura de decisão encadeada, também conhecida como estrutura de controle de múltiplas condições, é útil quando é necessário avaliar uma série de condições sequencialmente e tomar ações diferentes com base nesses resultados. Ela permite lidar com diferentes cenários de forma mais detalhada, garantindo um fluxo de execução controlado.

Decisão binária aninhada

Decisão binária aninhada, também conhecida como estrutura de decisão aninhada, ocorre quando uma estrutura condicional **if-else** está dentro de outra estrutura **if-else**. Essa abordagem permite lidar com múltiplas condições e executar diferentes blocos de código de acordo com essas condições aninhadas.

```
int idade = 20;

bool possuiCarteiraMotorista = true;

if (idade >= 18)
{
    if (possuiCarteiraMotorista)
    {
```



```
        Console.WriteLine("Você é maior de idade e  
possui carteira de motorista.");  
    }  
    else  
    {  
        Console.WriteLine("Você é maior de idade, mas  
não possui carteira de motorista.");  
    }  
}  
else  
{  
    Console.WriteLine("Você é menor de idade.");  
}
```

Código 4.1.4

Nesse exemplo, a primeira condição avaliada é se a idade é maior ou igual a 18. Se for verdadeira, o código dentro do primeiro **if** é executado. Dentro desse bloco, há outra verificação condicional para determinar se a pessoa possui carteira de motorista. Se essa condição também for verdadeira, a mensagem "**Você é maior de idade e possui carteira de motorista.**" será exibida. Caso contrário, a mensagem "**Você é maior de idade, mas não possui carteira de motorista.**" será exibida. Se a primeira condição for falsa, ou seja, a idade for menor que 18, o bloco de código dentro do **else** será executado e a mensagem "Você é menor de idade." será exibida.

Saida: Você é maior de idade e possui carteira de motorista.

Execução código 4.1.4

Decisão binária aninhada é uma ferramenta poderosa que pode ser usada para lidar com situações mais complexas, em que múltiplas condições precisam ser avaliadas em diferentes níveis. Isso oferece maior flexibilidade para controlar o fluxo do programa com base em diversas combinações de condições.

4.2 Switch-case

A estrutura de controle "switch-case" é uma estrutura de comparação que permite realizar seleção entre várias opções com base no valor de uma expressão. Essa estrutura é especialmente útil quando há necessidade de comparar uma variável ou expressão com diferentes casos e executar diferentes blocos de código dependendo do valor correspondente.

A estrutura **switch-case** é composta por uma expressão que é avaliada uma vez e comparada com os diferentes casos definidos. Cada caso é especificado usando a palavra-chave **case** seguida pelo valor esperado. Se o valor da expressão for igual a um dos casos, o bloco de código correspondente a esse caso é executado. O comando **break** é usado para sair da estrutura **switch-case** após a execução do bloco de código de um caso. Isso evita que os blocos de código dos casos subsequentes sejam executados. É importante lembrar de incluir o **break** em cada caso, a menos que a intenção seja permitir a execução contínua em múltiplos casos.

O **switch-case** também pode ter um caso **default** opcional. O caso **default** é executado quando nenhum dos casos anteriores corresponde ao valor da expressão. Ele pode ser usado para lidar com valores não previstos ou como uma opção de **fallback** também conhecido como **default**.

```
int diaDaSemana = 5;
switch (diaDaSemana)
{
    case 1:
        Console.WriteLine("Segunda-feira");
        break;
    case 2:
        Console.WriteLine("Terça-feira");
        break;
    case 3:
        Console.WriteLine("Quarta-feira");
        break;
    default:
        Console.WriteLine("Dia inválido");
        break;
}
```

Código 4.2.1

Nesse exemplo, se o valor de **diaDaSemana** for **5**, nenhum dos casos especificados será correspondido. Portanto, o bloco de código associado à opção de **fallback (default)** será executado, exibindo a mensagem "**Dia inválido**" no console.

Saida: Dia inválido

Execução código 4.2.1

4.3 For

A estrutura de controle **for** é usada para criar loops ou repetições controladas, permitindo que um bloco de código seja executado repetidamente por um número específico de vezes. O **for** é especialmente útil quando se sabe previamente quantas iterações são necessárias.

- **Inicialização:** É uma expressão que define a variável de controle e seu valor inicial. Geralmente, é onde se declara e/ou atribui um valor à variável de controle do loop.
- **Condição:** É uma expressão booleana que define a condição que deve ser avaliada a cada iteração do loop. Enquanto a condição for verdadeira, o bloco de código dentro do **for** continuará sendo executado. Se a condição for falsa, o loop será encerrado.
- **Expressão de iteração:** É uma expressão que é executada no final de cada iteração do loop, geralmente para atualizar a variável de controle. Isso permite que a variável de controle seja alterada ou incrementada a cada iteração.
- **Bloco de código:** É o conjunto de instruções a serem repetidas enquanto a condição for verdadeira. Esse bloco é delimitado por chaves (**{}**) e pode conter qualquer código válido.

```
for (inicialização; condição; expressão de iteração)
{
    // Bloco de código a ser repetido
}
```

Código 4.3.1

```
for (int i = 0; i < 5; i++)  
{  
    Console.WriteLine("Valor de i: " + i);  
}
```

Código 4.3.2

Nesse exemplo, a variável de controle **i** é inicializada com o valor 0. A condição **i < 5** é avaliada a cada iteração. Enquanto a condição **for** for verdadeira, o bloco de código dentro do **for** é executado. Dentro do bloco, é exibida uma mensagem mostrando o valor atual de **i**. Após cada iteração, a expressão de iteração **i++** é executada, incrementando o valor de **i** em 1. O loop será executado cinco vezes, exibindo os valores de 0 a 4.

```
Saida: Valor de i: 0  
Saida: Valor de i: 1  
Saida: Valor de i: 2  
Saida: Valor de i: 3  
Saida: Valor de i: 4
```

Execução código 4.3.2

A estrutura **for** é especialmente útil quando se sabe antecipadamente quantas iterações são necessárias ou quando se precisa iterar sobre uma coleção de elementos com base em seus índices. É uma ferramenta poderosa para criar loops controlados e repetir um bloco de código de forma eficiente.

Embora possa parecer desafiador encontrar uma utilidade para o comando **for**, ele é extremamente valioso e amplamente empregado em várias aplicações. A fim de elucidar ainda mais sua utilidade, segue mais um exemplo adicional para destacar melhor seu uso.

```
int numero = 5;
Console.WriteLine("Tabuada do " + numero + ":");

for (int i = 1; i <= 5; i++)
{
    int resultado = numero * i;
    Console.WriteLine(numero + " x " + i + " = " +
        resultado);
}
```

Código 4.3.3

No exemplo apresentado, a tabuada do número 5 é exibida, multiplicando o número 5 por cada valor de *i* no intervalo de 1 a 5.

```
Saida: Tabuada do 5:
Saida: 5 x 1 = 5
Saida: 5 x 2 = 10
Saida: 5 x 3 = 15
Saida: 5 x 4 = 20
Saida: 5 x 5 = 25
```

Execução código 4.3.3

4.4 While

A estrutura de controle **while** é usada para criar loops ou repetições baseadas em uma condição. Diferente do **for**, o **while** permite que um bloco de código seja executado repetidamente enquanto a condição especificada for verdadeira

```
while (condição)
{
    // Bloco de código a ser repetido
}
```

Código 4.4.1

Condição

É uma expressão booleana que é avaliada antes de cada iteração do loop. Enquanto a condição for verdadeira, o bloco de código dentro do **while** será executado. Se a condição for falsa, a execução do loop é interrompida e o controle passa para a próxima instrução após o **while**.

Bloco de código

É o conjunto de instruções a serem repetidas enquanto a condição especificada for verdadeira. O bloco de código é delimitado por chaves (**{}**).

É importante ter cuidado ao usar o **while** para evitar criar loops infinitos. Certifique-se de que a condição será eventualmente falsa, ou inclua uma lógica adequada para interromper o loop, como a alteração de uma variável de controle ou a utilização de uma instrução **break**.

```
int contador = 1;
while (contador <= 5)
{
    Console.WriteLine("Contador: " + contador);
    contador++;
}
Console.WriteLine("Loop concluído.");
```

Código 4.4.2

Neste exemplo, a variável contador é inicializada com o valor **1**. Enquanto o valor do contador for menor ou igual a **5**, o bloco de código dentro do **while** é executado. Dentro do bloco, é exibida a mensagem "**Contador:** " seguida do valor atual do contador. Após cada iteração, o contador é incrementado em **1** com a instrução **contador++**. O loop continuará até que o contador seja maior que **5**. Quando a condição se tornar falsa, o loop é concluído e a mensagem "**Loop concluído**" é exibida.

```
Saida: Contador: 1
Saida: Contador: 2
Saida: Contador: 3
Saida: Contador: 4
Saida: Contador: 5
Saida: Loop concluído.
```

Execução código 4.4.2

4.5 Do-While

A estrutura de controle **do-while** é usada para criar um loop ou repetição controlada por uma condição. Ao contrário do **while**, o **do-while** executa o bloco de código primeiro e, em seguida, verifica a condição para determinar se o loop deve continuar ou ser encerrado. Isso significa que o bloco de código é executado pelo menos uma vez, mesmo que a condição seja inicialmente falsa.

```
do
{
    // Bloco de código a ser repetido
}
while (condição);
```

Exemplo

Bloco de código: É o conjunto de instruções a serem repetidas. O bloco de código é delimitado por chaves **{}** e pode conter qualquer código válido.

Condição: É uma expressão booleana que é avaliada após a execução do bloco de código. Se a condição for verdadeira, o loop é executado novamente. Se a condição for falsa, o loop é encerrado e a execução continua na próxima instrução após o **do-while**.

É importante notar que a condição é verificada somente após a execução do bloco de código. Portanto, o bloco de código será executado pelo menos uma vez, independentemente da condição.

```
int contador = 1;
do
{
    Console.WriteLine("Contador: " + contador);
    contador++;
}
while (contador <= 5);

Console.WriteLine("Loop concluído.");
```

Código 4.5.1

Neste exemplo, o bloco de código dentro do **do** é executado primeiro. A mensagem "**Contador:** " seguida do valor atual do contador é exibida. Em seguida, o contador é incrementado em **1** com a instrução **contador++**. Após a execução do bloco de código, a condição **contador <= 5** é avaliada. Se a condição for verdadeira, o loop é executado novamente. Se a condição for falsa, o loop é encerrado e a execução continua na instrução **Console.WriteLine("Loop concluído.")**.

```
Saida: Contador: 1
Saida: Contador: 2
Saida: Contador: 3
Saida: Contador: 4
Saida: Contador: 5
Saida: Loop concluído.
```

Execução código 4.5.1

4.6 Break

A instrução **break** é uma forma de controlar a execução de um loop. Ela permite interromper imediatamente a execução do loop e sair dele, independentemente da condição que está sendo verificada.

Quando a instrução **break** é encontrada dentro de um loop o programa sai do loop e continua a execução na primeira instrução após o loop.

Vamos considerar um exemplo em que queremos realizar um loop para exibir os números de 1 a 10, mas desejamos interromper a execução quando o número 5 for encontrado.

```
int numero = 1;
while (numero <= 10)
{
    Console.WriteLine("Número: " + numero);
    if (numero == 5)
    {
        Console.WriteLine("Número 5 encontrado. Saindo
do loop.");
        break;
    }
    numero++;
}
Console.WriteLine("Loop concluído.");
```

Código 4.6.1

Neste exemplo, iniciamos o loop com a variável `numero` igual a 1. Enquanto o `numero` for menor ou igual a 10, o bloco de código dentro do **while** é executado. Dentro do bloco, exibimos o número atual.

No entanto, adicionamos uma verificação com um **if** para verificar se o número é igual a 5. Se essa condição for verdadeira, exibimos uma mensagem indicando que o número 5 foi encontrado e utilizamos a instrução **break** para interromper imediatamente a execução do loop. Isso significa que o programa sairá do loop e continuará executando na próxima instrução após o loop.

```
Saida: Número: 1
Saida: Número: 2
Saida: Número: 3
Saida: Número: 4
Saida: Número: 5
Saida: Número 5 encontrado. Saindo do loop.
Saida: Loop concluído.
```

Execução código 4.6.1

Outro ponto muito importante, o **break** também pode ser usado em outras estruturas de repetição, como **for** e **do-while**. O objetivo do **break** é interromper imediatamente a execução do loop e sair dele, independentemente do tipo de loop sendo utilizado.

4.7 Continue

A instrução **continue** é utilizada para controlar a execução de um loop.

Quando o **continue** é encontrado dentro de um loop, ele interrompe a iteração atual e passa para a próxima iteração, ignorando o restante do bloco de código que segue o **continue** dentro da mesma iteração.

Vamos considerar um exemplo em que queremos exibir apenas os números ímpares de 1 a 10, utilizando um loop **for** e a instrução **continue**:

```
for (int i = 1; i <= 10; i++)  
{  
    if (i % 2 == 0)  
    {  
        continue;  
    }  
    Console.WriteLine("Número ímpar: " + i);  
}
```

Código 4.7.1

Neste exemplo, utilizamos um loop **for** para iterar de 1 a 10. Dentro do loop, verificamos se o número atual (i) é par, utilizando a condição `i % 2 == 0`. Se essa condição for verdadeira, ou seja, o número é par, utilizamos a instrução **continue**. Isso faz com que o restante do bloco de código seja ignorado para essa iteração e o controle passe para a próxima iteração do loop.

Dessa forma, somente os números ímpares serão exibidos no console. A mensagem "**Número ímpar:** " seguida pelo valor do número ímpar é exibida quando a condição `i % 2 == 0` é falsa.

Saida: Número ímpar: 1

Saida: Número ímpar: 3

Saida: Número ímpar: 5

Saida: Número ímpar: 7

Saida: Número ímpar: 9

Execução código 4.7.1

Observe que os números pares são ignorados e não são exibidos, pois quando a condição `i % 2 == 0` é verdadeira, o **continue** é acionado, pulando a instrução `Console.WriteLine("Número ímpar: " + i)`.

O uso do **continue** é útil para controlar a execução de um loop e pular certas iterações com base em uma condição específica. Isso pode ser útil quando você deseja evitar que uma parte específica do bloco de código seja executada em determinadas situações. Certifique-se de utilizar o **continue** com cuidado para evitar comportamentos inesperados ou loops infinitos.

Importante alertar que o **continue** pode ser utilizada não apenas em loops **for**, mas também em outras estruturas de repetição, como **while** e **do-while**.

4.8 Desafios

- 1 - Exibir os números pares de 1 a 20 utilizando um loop **for**
- 2 - Pedir ao usuário para digitar números até que um número negativo seja digitado utilizando um loop **while**
- 3 - Calcular a média de uma sequência de números digitados pelo usuário até que o número zero seja digitado utilizando um loop **do-while**
- 4 - Verificar se um número digitado pelo usuário é primo utilizando um loop **for**
- 5 - Exibir uma contagem regressiva de 10 a 1 utilizando um loop **while**

5. Vetores e matrizes

“A programação é uma arte disfarçada de ciência e engenharia.”

5.1 Vetores

Vetor é uma estrutura de dados que armazena uma coleção de elementos do mesmo tipo. Os vetores também são conhecidos como arrays unidimensionais.

Declaração e inicialização de um vetor

Você pode declarar e inicializar um vetor usando a seguinte sintaxe:

```
tipo[] nomeVetor = new tipo[tamanho];
```

Exemplo

Onde **tipo** é o tipo de dado que o vetor irá armazenar, **nomeVetor** é o nome que você escolhe para o vetor e **tamanho** é o número de elementos que o vetor pode conter.

Por exemplo, se você deseja criar um vetor de inteiros com 5 elementos, você pode fazer o seguinte:

```
int[] numeros = new int[5];
```

Código 5.1.1

Acessando elementos do vetor

Os elementos em um vetor são acessados usando índices, que são números inteiros que representam a posição do elemento no vetor. Os índices começam em 0 para o primeiro elemento e vão até tamanho - 1 para o último elemento. Para acessar um elemento específico, você usa o nome do vetor seguido do índice entre colchetes.

Por exemplo, para acessar o terceiro elemento do vetor **numeros** que declaramos anteriormente no [código 5.1.2](#), você pode fazer o seguinte:

```
int terceiroNumero = numeros[2];
```

Código 5.1.2

Modificando elementos do vetor

Você também pode modificar elementos individuais do vetor atribuindo um novo valor a eles. Para fazer isso, você usa o nome do vetor

seguido do índice entre colchetes, e então atribui um novo valor ao elemento.

Por exemplo, se você quiser modificar o valor do segundo elemento do vetor **numeros**, você pode fazer o seguinte:

```
numeros[1] = 10;
```

Código 5.1.3

Comprimento do vetor

Você pode obter o comprimento de um vetor usando a propriedade **Length**. Essa propriedade retorna o número total de elementos no vetor.

```
int tamanhoVetor = numeros.Length;
```

Código 5.1.4

Iteração sobre os elementos do vetor

Uma forma comum de percorrer os elementos de um vetor é usando um loop, como o loop for. Você pode usar o comprimento do vetor para definir os limites do loop e, em seguida, acessar cada elemento individualmente.

Por exemplo, para percorrer todos os elementos do vetor **numeros** e imprimi-los no console, você pode fazer o seguinte:

```
int[] numeros = new int[5] { 1, 2, 3, 4, 5 };  
for (int i = 0; i < numeros.Length; i++)  
{  
    Console.WriteLine(numeros[i]);  
}
```

Código 5.1.5

No exemplo [código 5.1.5](#), um vetor chamado **numeros** é declarado e inicializado com 5 elementos inteiros. Em seguida, um loop for é usado para percorrer o vetor e exibir cada elemento no console.

```
Saida: 1  
Saida: 2  
Saida: 3  
Saida: 4  
Saida: 5
```

Execução código 5.1.5

Aqui está mais um exemplo usando um vetor para armazenar uma lista de nomes:

```
string[] nomes = { "João", "Maria", "Pedro", "Ana",  
"Carlos" };  
  
for (int i = nomes.Length - 1; i >= 0; i--)  
{  
    Console.WriteLine(nomes[i]);  
}
```

Código 5.1.6

```
Carlos  
Ana  
Pedro  
Maria  
João
```

Execução código 5.1.6

No código 5.1.6, um vetor chamado **nomes** é criado e inicializado com 5 nomes. Em seguida, um loop for é usado para percorrer o vetor em ordem reversa, começando pelo último elemento e indo até o primeiro. Cada elemento do vetor é exibido no console.

5.2 Matrizes

Matrizes são estruturas de dados bidimensionais que armazenam elementos de um mesmo tipo organizados em linhas e colunas. As matrizes são uma extensão dos vetores unidimensionais, permitindo o armazenamento de dados em uma estrutura tabular.

```
tipo[,] nomeMatriz = new tipo[linhas, colunas];
```

Exemplo

Tipo é o tipo de dado que a matriz irá armazenar, **nomeMatriz** é o nome que você escolhe para a matriz, **linhas** é o número de linhas da matriz e **colunas** é o número de colunas da matriz.

```
int[,] matriz = new int[3, 2];
```

Código 5.2.1

Acessando elementos da matriz

Os elementos em uma matriz são acessados usando índices para indicar a posição do elemento na matriz. Você pode usar os índices de linha e coluna para acessar elementos individuais. Os índices começam em 0 para a primeira linha e a primeira coluna.

```
int elemento = matriz[1, 2];
```

Código 5.2.2

Modificando elementos da matriz

Você também pode modificar elementos individuais da matriz atribuindo um novo valor a eles. Para fazer isso, você usa os índices de linha e coluna para indicar a posição do elemento que deseja modificar.

```
matriz[0, 1] = 10;
```

Código 5.2.3

Comprimento das dimensões da matriz

Você pode obter o comprimento das dimensões de uma matriz usando o método **GetLength()**. Esse método retorna o número de elementos em uma dimensão específica da matriz.

```
int numeroLinhas = matriz.GetLength(0);  
int numeroColunas = matriz.GetLength(1);
```

Código 5.2.4

Iteração sobre os elementos da matriz

Uma forma comum de percorrer os elementos de uma matriz é usando loops aninhados, um para as linhas e outro para as colunas. Você pode usar os métodos **GetLength()** para definir os limites dos loops e, em seguida, acessar cada elemento individualmente.

```

for (int i = 0; i < matriz.GetLength(0); i++)
{
    for (int j = 0; j < matriz.GetLength(1); j++)
    {
        Console.WriteLine(matriz[i, j]);
    }
}

```

Código 5.2.5

Exibindo uma matriz de números inteiros:

```

int[,] matriz = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
Console.WriteLine("Matriz de números inteiros:");
for (int i = 0; i < matriz.GetLength(0); i++)
{
    for (int j = 0; j < matriz.GetLength(1); j++)
    {
        Console.Write(matriz[i, j] + " ");
    }
    Console.WriteLine();
}

```

Código 5.2.6

No código 5.2.6, criamos uma matriz de números inteiros e inicializamos seus valores diretamente na declaração.

Em seguida, usamos **Console.WriteLine()** para exibir uma mensagem indicando que estamos mostrando uma matriz de números inteiros.

Em seguida, usamos dois loops **for** aninhados para percorrer cada elemento da matriz.

O primeiro loop **for** percorre as linhas da matriz, usando a função **GetLength(0)** para obter o tamanho da dimensão 0 (número de linhas).

O segundo loop **for** percorre as colunas da matriz, usando a função **GetLength(1)** para obter o tamanho da dimensão 1 (número de colunas).

Dentro dos loops, usamos **Console.Write()** para exibir cada elemento da matriz separado por um espaço.

Após o segundo loop **for**, usamos **Console.WriteLine()** para imprimir uma nova linha, criando uma nova linha para cada linha da matriz.

```
Saida: 1 2 3
```

```
Saida: 4 5 6
```

```
Saida: 7 8 9
```

Execução código 5.2.6

Matrizes são essenciais quando você precisa trabalhar com dados organizados em formato de tabela. Eles oferecem uma maneira eficiente de armazenar e manipular dados de forma estruturada.

5.3 Copiando vetores ou matrizes

Ao trabalhar com vetores, muitas vezes você precisa copiar um vetor para outro. A cópia de um vetor é útil quando você deseja realizar alterações em uma cópia do vetor original sem modificar os dados originais.

Não se pode simplesmente afirmar que um vetor é igual a outro vetor de forma direta, pois ao fazer isso, não realizaremos a cópia dos dados contidos no vetor, mas sim do endereço de memória para o qual ele aponta, quando você atribui um vetor a outro, o comportamento padrão é copiar o endereço de memória do vetor original para o vetor de destino. Isso significa que as duas variáveis de vetor apontarão para a mesma região de memória, e qualquer alteração em um vetor também será refletida no outro. Este assunto se tornará mais claro quando abordarmos os diferentes tipos de memórias, stack e heap.

```
int[] vetorOriginal = { 1, 2, 3, 4, 5 };
int[] vetorCopia = vetorOriginal;
vetorCopia[0] = 10;

Console.Write("Vetor Original: ");
for (int i = 0; i < vetorOriginal.Length; i++)
{
    Console.Write(vetorOriginal[i]+",");
}
Console.WriteLine();
Console.Write("Vetor Cópia: ");
```



```
for (int i = 0; i < vetorCopia.Length; i++)  
{  
    Console.Write(vetorCopia[i]+",");  
}
```

Código 5.3.1

Saida: Vetor Original: 10, 2, 3, 4, 5,

Saida: Vetor Cópia: 10, 2, 3, 4, 5,

Saida código 5.3.1

Observe que a modificação feita no vetor de cópia também afeta o vetor original, pois ambos apontam para a mesma região de memória.

Se você deseja criar uma cópia dos valores do vetor original para um novo vetor, existem diferentes métodos para copiar vetores.

Método Array.Copy()

O método `Array.Copy()` é uma forma comum de copiar um vetor. Ele permite copiar elementos de um vetor de origem para um vetor de destino. A sintaxe básica do método `Array.Copy()` é a seguinte:

```
Array.Copy(vetorOrigem, vetorDestino, tamanho);
```

Exemplo

- `vetorOrigem` é o vetor que você deseja copiar.
- `vetorDestino` é o vetor para o qual você deseja copiar os elementos.
- `tamanho` é o número de elementos a serem copiados.

```
int[] vetorOrigem = { 1, 2, 3, 4, 5 };  
int[] vetorDestino = new int[vetorOrigem.Length];  
Array.Copy(vetorOrigem, vetorDestino,  
vetorOrigem.Length);
```

Código 5.3.2

Após a execução desse código, o `vetorDestino` conterá uma cópia dos elementos do `vetorOrigem`.

Método `Array.Clone()`

Outra forma de copiar um vetor é usando o método `Array.Clone()`. Esse método cria uma cópia superficial (shallow copy) do vetor, ou seja, os elementos do vetor são copiados, mas se o vetor contiver objetos, as referências para os objetos serão compartilhadas entre o vetor original e a cópia. A sintaxe básica do método `Array.Clone()` é a seguinte:

```
var vetorCopia = (tipo[])vetorOrigem.Clone();
```

Código 5.3.3

- `vetorOrigem` é o vetor que você deseja copiar.
- `tipo` é o tipo de dado que o vetor armazena.

```
int[] vetorOrigem = { 1, 2, 3, 4, 5 };  
int[] vetorCopia = (int[])vetorOrigem.Clone();
```

Código 5.3.4

Após a execução desse código, o **vetorCopia** conterá uma cópia dos elementos do **vetorOrigem**. Importante saber que, se o vetor contiver objetos complexos, as referências para esses objetos serão compartilhadas entre o vetor original e a cópia

Usando iteração manual

Você também pode copiar um vetor manualmente usando iteração. Nesse caso, você percorre cada elemento do vetor de origem e atribui seu valor ao elemento correspondente do vetor de destino.

```
int[] vetorOrigem = { 1, 2, 3, 4, 5 };
int[] vetorDestino = new int[vetorOrigem.Length];
for (int i = 0; i < vetorOrigem.Length; i++)
{
    vetorDestino[i] = vetorOrigem[i];
}
```

Código 5.3.5

Depois de executar esse código, o **vetorDestino** conterá uma cópia dos elementos do **vetorOrigem**.

5.4 Foreach

O **foreach** é uma construção de laço (**loop**) que permite iterar sobre elementos de uma coleção, como um vetor, uma lista ou uma matriz. Ele é projetado para facilitar a iteração por elementos sem a necessidade de gerenciar manualmente os índices ou a contagem dos elementos.

```
foreach (tipo item in colecao)
{
    // Código a ser executado para cada item
}
```

Exemplo

- Tipo é o tipo de dado dos elementos na coleção.
- Item é a variável que representa cada elemento da coleção durante a iteração.
- Coleção é a coleção na qual você deseja iterar, como um vetor, uma lista ou uma matriz.

Como funciona o foreach

O **foreach** itera sequencialmente sobre cada elemento na coleção. A cada iteração, o próximo elemento da coleção é atribuído à variável **item**, e o bloco de código dentro do **foreach** é executado para processar esse item.

Vantagens do foreach

- **Simplifica a iteração:** O foreach simplifica a iteração sobre elementos de uma coleção, eliminando a necessidade de acompanhar manualmente os índices ou a contagem de elementos.
- **Segurança de tipo:** O foreach garante que você itere apenas sobre elementos do tipo esperado na coleção, evitando erros de tipo durante a iteração.
- **Legibilidade do código:** O foreach torna o código mais legível, pois expressa claramente a intenção de iterar sobre os elementos da coleção.

Restrições do foreach

- **Iteração somente para leitura:** O foreach permite apenas iteração para leitura dos elementos da coleção. Você não pode modificar a coleção durante a iteração.

- **Não fornece acesso ao índice:** Ao contrário de um loop for tradicional, o foreach não fornece acesso direto ao índice dos elementos. Se você precisar do índice, pode precisar usar um loop for convencional.

Aqui está um exemplo prático de uso do foreach para iterar sobre um vetor de números e exibir cada elemento no console:

```
int[] numeros = { 1, 2, 3, 4, 5 };  
foreach (int numero in numeros)  
{  
    Console.WriteLine(numero);  
}
```

Código 5.4.1

Neste exemplo, o **foreach** itera sobre cada elemento do vetor **numeros**. Em cada iteração, o elemento atual é atribuído à variável **numero**, e o código dentro do bloco do **foreach** neste caso, **Console.WriteLine(numero)** é executado para processar o elemento.

```
Saida: 1  
Saida: 2  
Saida: 3  
Saida: 4  
Saida: 5
```

Execução código 5.4.1

5.5 Redimensionamento de vetores

Os vetores têm um tamanho fixo após a sua criação. No entanto, existem algumas abordagens que você pode usar para redimensionar vetores, permitindo aumentar ou diminuir o número de elementos que eles armazenam.

Criação de um novo vetor

Uma abordagem comum para redimensionar um vetor é criar um novo vetor com o tamanho desejado e copiar os elementos do vetor original para o novo vetor.

- Declare e inicialize um novo vetor com o tamanho desejado.
- Use um loop for para copiar os elementos do vetor original para o novo vetor.
- Atualize as referências para o vetor original com o novo vetor.

```
int[] vetorOriginal = { 1, 2, 3 };  
int novoTamanho = 5;  
int[] novoVetor = new int[novoTamanho];  
for (int i = 0; i < vetorOriginal.Length; i++)  
{  
    novoVetor[i] = vetorOriginal[i];  
}  
vetorOriginal = novoVetor;  
Console.WriteLine(vetorOriginal.Length);
```

Código 5.5.1

Neste exemplo, o vetor **vetorOriginal** inicialmente tem 3 elementos. Criamos um novo vetor **novoVetor** com um tamanho maior (5) e copiamos os elementos do **vetorOriginal** para o **novoVetor**. Em seguida, atualizamos a referência do **vetorOriginal** para o **novoVetor**, redimensionando efetivamente o vetor.

Saida: 5

Execução código 5.5.1

Uso do método `Array.Resize()`

A classe `Array` fornece um método estático chamado `Resize()` que permite redimensionar um vetor. O método `Resize()` cria um novo vetor com o tamanho especificado e copia os elementos do vetor original para o novo vetor.

```
int[] vetorOriginal = { 1, 2, 3 };  
int novoTamanho = 5;  
Array.Resize(ref vetorOriginal, novoTamanho);  
Console.WriteLine(vetorOriginal.Length);
```

Código 5.5.2

Saida: 5

Execução código 5.5.2

No código 5.5.2, o vetor **vetorOriginal** é redimensionado para o tamanho **5** usando o método **Resize()**. A palavra-chave **ref** é usada para indicar que a referência ao vetor é passada por referência e que as alterações no vetor dentro do método **Resize()** serão refletidas na variável original.

É importante ressaltar que ao reduzir o tamanho de um vetor usando o **Array.Resize()**, os elementos além do novo tamanho são perdidos e a memória associada a esses elementos é liberada.

No exemplo [código 5.5.3](#), temos um vetor **vetorOriginal** inicialmente com **10** elementos. Usamos o **Array.Resize()** para redimensionar o vetor para um tamanho menor, especificando **novoTamanho** como **5**. A palavra-chave **ref** é usada para passar a referência ao vetor por referência, permitindo que o método **Resize()** atualize a referência original.

```
int[] vetorOriginal = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
int novoTamanho = 5;
Array.Resize(ref vetorOriginal, novoTamanho);
foreach (var item in vetorOriginal)
{
    Console.WriteLine(item);
}
```

Código 5.5.3

Após a execução do **Array.Resize()**, o vetor **vetorOriginal** terá seu tamanho reduzido para **5** elementos, mantendo apenas os primeiros **5** elementos originais. Os elementos adicionais do vetor original são descartados.

```
Saida: 1
Saida: 2
Saida: 3
Saida: 4
Saida: 5
```

Execução código 5.5.3

5.6 Ordenação de vetores

Ordenar um vetor é um problema comum e existem vários algoritmos de ordenação disponíveis. Um algoritmo de ordenação é um conjunto de passos que organiza os elementos de um vetor em uma determinada ordem, crescente ou decrescente. Vou explicar dois algoritmos de ordenação amplamente utilizados.

Algoritmo de ordenação por seleção (Selection Sort):

O algoritmo de ordenação por seleção funciona selecionando repetidamente o menor (ou maior) elemento do vetor e colocando-o em sua posição correta.

- O vetor é dividido em duas partes, a parte ordenada e a parte não ordenada.
- Na primeira iteração, o elemento mínimo é encontrado na parte não ordenada e trocado com o primeiro elemento do vetor.
- Na segunda iteração, o segundo menor elemento é encontrado na parte não ordenada e trocado com o segundo elemento do vetor.
- Esse processo continua até que todos os elementos estejam na parte ordenada.

```
int[] vetor = new int[5]{ 55, 19, 32, 28, 1 };
int tamanho = vetor.Length;
for (int i = 0; i < tamanho - 1; i++)
{
    int indiceMinimo = i;
    for (int j = i + 1; j < tamanho; j++)
    {
        if (vetor[j] < vetor[indiceMinimo])
        {
            indiceMinimo = j;
        }
    }
    int temp = vetor[i];
    vetor[i] = vetor[indiceMinimo];
    vetor[indiceMinimo] = temp;
}
foreach (var item in vetor)
{
    Console.WriteLine(item);
}
```

Código 5.6.1

```
Saida: 1  
Saida: 19  
Saida: 28  
Saida: 32  
Saida: 55
```

Execução código 5.6.1

Algoritmo de ordenação por inserção (Insertion Sort)

O algoritmo de ordenação por inserção constrói uma parte ordenada do vetor um elemento de cada vez. A ideia principal é inserir cada elemento na posição correta da parte ordenada, deslocando os elementos maiores (se necessário).

- O vetor é dividido em duas partes: a parte ordenada e a parte não ordenada.
- Na primeira iteração, o primeiro elemento é considerado a parte ordenada.
- Para cada elemento subsequente, ele é comparado com os elementos na parte ordenada e inserido na posição correta.
- Isso é feito deslocando os elementos maiores para a direita até encontrar a posição correta para inserir o elemento.


```
int[] vetor = new int[5]{ 55, 19, 32, 28, 1 };
int tamanho = vetor.Length;
for (int i = 1; i < tamanho; i++)
{
    int valorAtual = vetor[i];
    int j = i - 1;
    while (j >= 0 && vetor[j] > valorAtual)
    {
        vetor[j + 1] = vetor[j];
        j--;
    }
    vetor[j + 1] = valorAtual;
}
foreach (var item in vetor)
{
    Console.WriteLine(item);
}
```

Código 5.6.2

```
Saida: 1  
Saida: 19  
Saida: 28  
Saida: 32  
Saida: 55
```

Execução código 5.6.2

Existem muitos outros algoritmos de ordenação, como o algoritmo de ordenação por bolha (Bubble Sort), algoritmo de ordenação rápida (Quick Sort), algoritmo de ordenação por intercalação (Merge Sort), entre outros. Cada algoritmo tem suas características e complexidades diferentes, sendo adequado para diferentes situações e tamanhos de vetores.

É importante considerar o contexto e os requisitos do seu programa ao escolher um algoritmo de ordenação. Algoritmos como o Quick Sort ou Merge Sort, são recomendados para vetores grandes, enquanto algoritmos mais simples, como o Selection Sort ou Insertion Sort, podem ser mais adequados para vetores menores ou quase ordenados.

Usando recursos do .NET Framework

O .NET Framework oferece recursos para a ordenação de vetores usando o método `Array.Sort()`. Esse método fornece uma implementação de algoritmos de ordenação, como o QuickSort, para classificar os elementos em ordem crescente.

```
int[] vetor = { 5, 2, 8, 3, 1 };  
Array.Sort(vetor);  
foreach (var item in vetor)  
{  
    Console.WriteLine(item);  
}
```

Código 5.6.3

Neste exemplo, temos um vetor inicializado com alguns valores não ordenados. Usamos o método **Array.Sort()** passando o vetor como argumento. Esse método classificará o vetor em ordem crescente.

```
Saida: 1  
Saida: 2  
Saida: 3  
Saida: 5  
Saida: 8
```

Execução código 5.6.3

O **Array.Sort()** ordenará os elementos do vetor usando um algoritmo de ordenação do próprio .NET Framework.

É importante destacar que o **Array.Sort()** altera diretamente o vetor original, em vez de criar um novo vetor ordenado. Se você deseja manter o vetor original inalterado e obter um novo vetor ordenado, pode fazer uma cópia antes de chamar o método **Sort()**.

O método **Array.Sort()** é uma opção conveniente e eficiente para ordenar vetores em ordem crescente. No entanto, é importante lembrar que esse método opera apenas em ordem crescente. Se você precisar de uma ordenação personalizada ou precisar ordenar em ordem

decrecente, pode considerar o uso de algoritmos de ordenação personalizados ou implementar sua própria lógica.

5.7 Busca de itens

Ao trabalhar com vetores, pode ser necessário buscar um item específico dentro do vetor. Existem diferentes abordagens para realizar essa busca, e vou explicar algumas delas:

Busca linear

A busca linear é o método mais simples e direto para encontrar um item em um vetor. Envolve percorrer sequencialmente cada elemento do vetor até encontrar o item desejado ou até percorrer todos os elementos sem encontrá-lo.

```
int[] vetor = { 10, 20, 30, 40, 50 };  
int itemProcurado = 30;  
int indice = -1;  
for (int i = 0; i < vetor.Length; i++)  
{  
    if (vetor[i] == itemProcurado)  
    {
```



```
        indice = i;
        break;
    }
}
if (indice != -1)
{
    Console.WriteLine($"O item {itemProcurado} foi encontrado no índice {indice}.");
}
else
{
    Console.WriteLine($"O item {itemProcurado} não foi encontrado no vetor.");
}
```

Código 5.7.1

Neste exemplo, o vetor contém números inteiros e estamos buscando o valor 30. O laço for percorre cada elemento do vetor e verifica se o elemento é igual ao valor procurado. Se o item for encontrado, o índice correspondente é armazenado na variável indice. Caso contrário, o valor de indice permanece como -1 para indicar que o item não foi encontrado.

O item 30 foi encontrado no índice 2.

Execução código 5.7.1

Busca binária

A busca binária é um algoritmo de busca mais eficiente, mas requer que o vetor esteja previamente ordenado. Segue uma abordagem de divisão e conquista, dividindo o vetor pela metade a cada iteração. A busca é realizada comparando o item procurado com o elemento central do vetor e decidindo se a busca deve continuar na metade esquerda ou direita.

```
int[] vetor = { 10, 20, 30, 40, 50 };  
int itemProcurado = 30;  
int indice = -1;  
int inicio = 0;  
int fim = vetor.Length - 1;  
while (inicio <= fim)  
{  
    int meio = (inicio + fim) / 2;  
    if (vetor[meio] == itemProcurado)  
    {  
        indice = meio;  
        break;  
    }  
    else if (vetor[meio] < itemProcurado)  
    {  
        inicio = meio + 1;  
    }  
}
```



```
        else
        {
            fim = meio - 1;
        }
    }
    if (indice != -1)
    {
        Console.WriteLine($"O item {itemProcurado} foi encontrado no índice {indice}.");
    }
    else
    {
        Console.WriteLine($"O item {itemProcurado} não foi encontrado no vetor.");
    }
}
```

Código 5.7.2

O item 30 foi encontrado no índice 2.

Execução código 5.7.2

No exemplo código 5.7.2, o vetor está ordenado e estamos buscando o valor 30. A busca binária é realizada usando um loop while. A cada iteração, calculamos o índice do elemento do meio do vetor e comparamos com o item procurado. Se forem iguais, o item foi encontrado. Caso contrário, ajustamos os limites início e fim do intervalo de busca com base na comparação e repetimos o processo na metade correspondente.

Embora a busca binária possa ser um pouco mais complexa em termos de implementação do que a busca linear, ela é amplamente utilizada devido à sua eficiência em grandes conjuntos de dados. É especialmente útil quando você tem um vetor ordenado e precisa realizar buscas frequentes.

A complexidade da busca binária é logarítmica, isso significa que a quantidade de operações necessárias para encontrar um item aumenta de forma muito mais lenta à medida que o tamanho do vetor aumenta. Essa característica é devida à natureza do algoritmo de divisão do vetor pela metade a cada iteração.

Lembrando que a escolha entre a busca linear e a busca binária depende do contexto e dos requisitos específicos do seu programa. Se você tiver um vetor pequeno ou não ordenado, a busca linear pode ser uma opção mais simples. Já para vetores grandes e ordenados, a busca binária é a escolha ideal para um desempenho mais eficiente.

5.8 Desafios

- 1** - Crie um programa que receba um vetor de números inteiros como entrada e determine o menor e o maior valor presentes no vetor.
- 2** - Crie um programa que peça ao usuário para inserir os elementos de uma matriz quadrada (mesmo número de linhas e colunas) e verifique se ela é uma matriz diagonal, ou seja, todos os elementos fora da diagonal principal são iguais a zero.
- 3** - Crie um programa que copie os elementos de um vetor original para um novo vetor e exiba o novo vetor como saída.
- 4** - Crie um programa que receba uma lista de nomes como entrada e use um loop foreach para exibir cada nome em uma linha separada.
- 5** - Crie um programa que redimensione um vetor de números inteiros. Peça ao usuário para inserir o tamanho original do vetor e os valores

dos elementos. Em seguida, reduza o tamanho do vetor para a metade e exiba o novo vetor resultante.

6 - Crie um programa que ordene um vetor de números inteiros em ordem crescente utilizando o algoritmo de ordenação por inserção. Peça ao usuário para inserir os elementos do vetor e, em seguida, exiba o vetor ordenado.

7 - Crie um programa que receba um vetor de números inteiros como entrada e um número específico a ser procurado. Implemente a busca binária para verificar se o número está presente no vetor. Exiba uma mensagem indicando se o número foi encontrado ou não.

6. Listas dinâmicas

“O software é como um aspirador de pó: ele só chama a atenção quando não está funcionando.”

6.1 Criação de lista dinâmica

Lista dinâmica é uma estrutura de dados que proporciona flexibilidade para armazenar e manipular coleções de elementos. Ao contrário de matrizes e vetores, que possuem tamanhos fixos, uma lista dinâmica pode crescer ou encolher automaticamente conforme novos elementos são adicionados ou removidos.

List<T> é usada para criar e manipular listas dinâmicas. Tipo **T** é o tipo dos elementos que serão armazenados na lista. Por exemplo, se você deseja criar uma lista de números inteiros, pode usar **List<int>**. Isso garantirá que a lista só pode conter elementos inteiros.

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

Código 6.1.1

No código 6.1.1, a variável **nomeLista** será uma lista dinâmica vazia de inteiros. Você pode adicionar, remover, acessar e modificar elementos nessa lista sem se preocupar com um tamanho fixo predefinido.

6.2 Adicionando elementos

Adicionar dados em uma lista dinâmica é uma operação bastante simples. Você pode usar o método **Add** da classe **List<T>** para adicionar novos elementos à lista. Aqui estão algumas opções para adicionar dados em uma lista dinâmica.

Adicionar um elemento de cada vez

```
List<string> nomes = new List<string>();  
nomes.Add("João");  
nomes.Add("Maria");  
nomes.Add("Carlos");
```

Código 6.2.1

Nesse exemplo, criamos uma lista dinâmica chamada **nomes** e adicionamos os nomes "**João**", "**Maria**" e "**Carlos**" à lista, um de cada vez, usando o método **Add**.

Adicionar vários elementos de uma só vez

```
List<int> numeros = new List<int>() { 1, 2, 3, 4, 5 };
```

Código 6.2.2

Você também pode inicializar a lista com vários elementos na criação, utilizando a sintaxe de inicialização de objetos. No [código 6.2.2](#), criamos uma lista dinâmica chamada **numeros** e adicionamos os números de **1** a **5** à lista em uma única linha.

Adicionar uma coleção de elementos

```
List<string> frutas = new List<string>();  
string[] arrayFrutas = { "Maçã", "Banana", "Laranja" };  
frutas.AddRange(arrayFrutas);
```

Código 6.2.3

Se você já tiver uma coleção de elementos, como um **array**, pode usar o método **AddRange** para adicionar todos os elementos da coleção à

lista dinâmica. No código 6.2.3, criamos uma lista dinâmica chamada **frutas** e adicionamos os elementos do array **arrayFrutas** à lista.

Lembre-se de que o tipo dos elementos que você está adicionando deve corresponder ao tipo especificado ao criar a lista. Por exemplo, se você criar uma **List<int>**, só poderá adicionar números inteiros a essa lista.

Adicionar elementos a uma lista dinâmica não está limitado a um tipo específico de dado. Você pode adicionar objetos de qualquer classe, tipos primitivos (como **int**, **double**, **bool**, etc.), **strings** ou até mesmo outras listas dinâmicas.

É importante notar que as listas dinâmicas são redimensionáveis, o que significa que elas podem crescer automaticamente conforme você adiciona mais elementos. A lista gerencia automaticamente a memória necessária para acomodar os elementos adicionados.

6.3 Acesso a elementos

O acesso aos elementos de uma lista dinâmica pode ser feito de várias maneiras.

Acesso por índice

Você pode acessar um elemento específico da lista utilizando o operador de índice **[]** e especificando o índice desejado. Lembre-se de que os índices começam em zero.

```
List<string> nomes = new List<string>() { "João",  
"Maria", "Carlos" };  
  
string primeiroNome = nomes[0];  
string segundoNome = nomes[1];  
string terceiroNome = nomes[2];  
Console.WriteLine(primeiroNome);  
Console.WriteLine(segundoNome);  
Console.WriteLine(terceiroNome);
```

Código 6.3.1

Nesse exemplo, criamos uma lista de nomes e acessamos os elementos individuais por seus índices.

```
Saida: João  
Saida: Maria  
Saida: Carlos
```

Execução código 6.3.1

6.4 Pesquisa e filtragem

A classe **List<T>** também fornece vários métodos para pesquisar, filtrar e manipular os elementos da lista.

Contains

Verifica se um elemento específico está presente na lista.

```
List<string> cores = new List<string>() { "Vermelho",  
"Verde", "Azul" };  
  
bool contemAzul = cores.Contains("Azul");  
  
bool contemAmarelo = cores.Contains("Amarelo");  
  
Console.WriteLine("Contém Azul: " + contemAzul);  
  
Console.WriteLine("Contém Amarelo: " + contemAmarelo);
```

Código 6.4.1

```
Contém Azul: True  
  
Contém Amarelo: False
```

Execução código 6.4.1

IndexOf

Retorna o índice da primeira ocorrência de um elemento específico na lista.

```
List<string> frutas = new List<string>() { "Maçã",  
"Banana", "Laranja", "Maçã" };  
  
int indiceBanana = frutas.IndexOf("Banana");  
  
int indiceManga = frutas.IndexOf("Manga");  
  
Console.WriteLine("Índice da Banana: " +  
indiceBanana);  
  
Console.WriteLine("Índice da Manga: " + indiceManga);
```

Código 6.4.2

Índice da Banana: 1

Índice da Manga: -1

Execução código 6.4.2

Isso significa que a palavra "**Banana**" está presente na lista e seu índice é **1**, enquanto a palavra "**Manga**" não está presente na lista, por isso o índice é **-1**.

Find

Localiza o primeiro elemento que corresponde a um critério específico.

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };  
  
int primeiroMaiorQueTrinta = numeros.Find(x => x > 30);  
  
Console.WriteLine("Primeiro número maior que 30: " + primeiroMaiorQueTrinta);
```

Código 6.4.3

Primeiro número maior que 30: 40

Execução código 6.4.3

FindAll

Retorna uma lista contendo todos os elementos que correspondem a um critério específico.

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };  
  
List<int> maioresQueVinte = numeros.FindAll(x => x > 20);  
  
foreach (int numero in maioresQueVinte)  
{  
    Console.WriteLine(numero);  
}
```

Código 6.4.4

```
Saida: 30  
Saida: 40  
Saida: 50
```

Execução código 6.4.4

FirstOrDefault e LastOrDefault

Retorna o primeiro ou o último elemento da lista que corresponde a um critério específico, ou o valor padrão se nenhum elemento for encontrado.


```

List<string> frutas = new List<string>() { "Maçã",
"Banana", "Laranja", "Maçã" };

string primeiraMaca = frutas.FirstOrDefault(x => x ==
"Maçã");

string ultimaMaca = frutas.LastOrDefault(x => x ==
"Maçã");

string primeiraUva = frutas.FirstOrDefault(x => x ==
"Uva");

Console.WriteLine("Primeira Maçã: " + primeiraMaca);
Console.WriteLine("Última Maçã: " + ultimaMaca);
Console.WriteLine("Primeira Uva: " + primeiraUva);

```

Código 6.4.5

```

Primeira Maçã: Maçã
Última Maçã: Maçã
Primeira Uva:

```

Execução código 6.4.5

É importante lembrar que, ao acessar elementos por índice, é necessário garantir que o índice esteja dentro dos limites válidos da lista. Caso contrário, ocorrerá uma exceção

ArgumentOutOfRangeException.

Nos exemplos código 6.4.4 e código 6.4.5, encontramos uma expressão **lambda**.

```

(x => x > 20);

```

Código 6.4.6

Se isso pareceu confuso, não se preocupe, no capítulo 7 abordaremos esse assunto de forma mais detalhada.

6.5 Remoção de elementos

A remoção de elementos em uma lista dinâmica é uma operação comum e bastante flexível. Existem várias maneiras de remover elementos de uma lista.

Remoção por valor usando o método Remove

O método **Remove** é usado para remover a primeira ocorrência de um valor específico em uma lista. Ele aceita como argumento o valor a ser removido e retorna um valor booleano indicando se a remoção foi bem-sucedida ou não.

No exemplo a seguir, temos uma lista `numeros` contendo os valores **{10, 20, 30, 40, 50}**. Usamos o método **Remove** para remover o valor **30** da lista:

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };  
  
numeros.Remove(30);  
  
foreach (int numero in numeros)  
{  
    Console.WriteLine(numero);  
}
```

Código 6.5.1

```
Saida: 10  
Saida: 20  
Saida: 40  
Saida: 50
```

Execução código 6.5.1

Nesse caso, o valor **30** é removido da lista. Após a remoção, a lista é ajustada automaticamente, e o elemento subsequente assume a posição vaga. Portanto, o resultado é a lista modificada sem o valor **30**.

Remoção por índice usando o método RemoveAt

O método **RemoveAt** é usado para remover um elemento da lista com base no seu índice. Ele aceita como argumento o índice do elemento a ser removido.

No exemplo a seguir, temos uma lista `nomes` contendo os valores {"João", "Maria", "Carlos"}. Usamos o método **RemoveAt** para remover o elemento no índice **1** (no caso, "Maria"):

```
List<string> nomes = new List<string>() { "João",  
    "Maria", "Carlos" };  
  
nomes.RemoveAt(1);  
  
foreach (string nome in nomes)  
{  
    Console.WriteLine(nome);  
}
```

Código 6.5.2

João

Carlos

Execução código 6.5.2

No código 6.5.2, o elemento **"Maria"** é removido da lista com base no seu índice **1**. Após a remoção, a lista é ajustada automaticamente, e o resultado é a lista modificada sem o elemento removido.

Remoção condicional com o método RemoveAll

O método **RemoveAll** é usado para remover todos os elementos que correspondem a uma determinada condição, especificada por uma expressão lambda. Ele retorna o número total de elementos removidos.

No exemplo a seguir, temos uma lista `numeros` contendo os valores **{10, 20, 30, 40, 50}**. Usamos o método **RemoveAll** em conjunto com a expressão lambda **`x => x > 30`** para remover todos os números maiores que **30** da lista:

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };  
  
numeros.RemoveAll(x => x > 30);  
  
foreach (int numero in numeros)  
{  
    Console.WriteLine(numero);  
}
```

Código 6.5.3

No código 6.5.3, usamos a expressão lambda **`x => x > 30`** para verificar se cada número na lista é maior que **30**. Todos os números maiores que **30** são removidos. Após a remoção, a lista é ajustada automaticamente, e o resultado é a lista modificada com apenas os números menores ou iguais a **30**.

Saida: 10

Saida: 20

Saida: 30

Execução código 6.5.3

Limpeza completa da lista com o método Clear

O método **Clear** é usado para remover todos os elementos da lista, deixando-a vazia.

No exemplo a seguir, temos uma lista nomes contendo os valores {"João", "Maria", "Carlos"}. Usamos o método **Clear** para remover todos os elementos da lista:

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };
numeros.Clear();
foreach (int numero in numeros)
{
    Console.WriteLine(numero);
}
```

Código 6.5.4

No código 6.5.4, todos os elementos da lista são removidos. Após a remoção, a lista estará vazia, e o resultado será uma lista sem nenhum elemento.

Reorganização dos elementos na lista

Ao remover um item de uma lista, os elementos subsequentes são automaticamente reorganizados para preencher o espaço vago. Isso significa que os índices dos elementos restantes serão atualizados, refletindo a nova ordem da lista.

```
List<string> frutas = new List<string>() { "Maçã",  
    "Banana", "Laranja", "Morango" };  
  
for (int i = 0; i < frutas.Count; i++)  
{  
    Console.WriteLine("Índice " + i + ": " +  
        frutas[i]);  
}
```

Código 6.5.5

```
Índice 0: Maçã  
Índice 1: Banana  
Índice 2: Laranja  
Índice 3: Morango
```

Execução código 6.5.5

Suponha que desejamos remover a fruta "Laranja" da lista. Podemos fazer isso usando o método **Remove**

```
List<string> frutas = new List<string>() { "Maçã",  
"Banana", "Laranja", "Morango" };  
frutas.Remove("Laranja");  
for (int i = 0; i < frutas.Count; i++)  
{  
    Console.WriteLine("Índice " + i + ": " +  
frutas[i]);  
}
```

Código 6.5.6

Após a remoção, a lista será reorganizada automaticamente. O elemento "**Laranja**" é removido, e os elementos subsequentes "**Morango**" e "**Banana**" são movidos para preencher o espaço vago.

Agora, a lista frutas contém os elementos "**Maçã**", "**Banana**" e "**Morango**" com seus respectivos índices atualizados.

```
Índice 0: Maçã  
Índice 1: Banana  
Índice 2: Morango
```

Execução código 6.5.6

Essa reorganização dos elementos permite que a lista mantenha uma sequência contínua de índices sem espaços vazios. É importante ter em mente que a reorganização ocorre automaticamente e você não precisa se preocupar em atualizar os índices manualmente após a remoção de um elemento.

6.6 Iteração sobre a lista

O loop `foreach` é uma forma conveniente e simples de iterar sobre uma lista. Ele itera automaticamente por todos os elementos da lista, sem a necessidade de acompanhar índices ou tamanhos.

```
List<string> frutas = new List<string>() { "Maçã",  
    "Banana", "Laranja" };  
  
foreach (string fruta in frutas)  
{  
    Console.WriteLine(fruta);  
}
```

Código 6.6.1

No código 6.6.1, o loop **foreach** percorre todos os elementos da lista `frutas` e imprime cada um deles no console.

Saida: Maçã

Saida: Banana

Saida: Laranja

Execução código 6.6.1

For

O loop `for` é outra opção para iterar sobre uma lista. Ele utiliza um contador e o tamanho da lista para controlar a iteração.


```
List<int> numeros = new List<int>() { 1, 2, 3, 4, 5 };  
for (int i = 0; i < numeros.Count; i++)  
{  
    Console.WriteLine(numeros[i]);  
}
```

Código 6.6.2

No código 6.6.2, o loop **for** itera sobre a lista **numeros** usando o contador **i** para acessar cada elemento pelo seu índice.

```
Saida: 1  
Saida: 2  
Saida: 3  
Saida: 4  
Saida: 5
```

Execução código 6.6.2

While

O loop **while** também pode ser usado para iterar sobre uma lista, especialmente quando você precisa de um controle mais flexível sobre a iteração.

```
List<string> nomes = new List<string>() { "João",  
"Maria", "Carlos" };  
  
int index = 0;  
while (index < nomes.Count)  
{  
    Console.WriteLine(nomes[index]);  
    index++;  
}
```

Código 6.6.3

No código 6.6.3, o loop **while** itera sobre a lista `nomes` enquanto o índice **index** for menor que o tamanho da lista. A cada iteração, o elemento correspondente é impresso no console.

```
Saida: João  
Saida: Maria  
Saida: Carlos
```

Execução código 6.6.3

Essas são as principais formas de iteração sobre uma lista. O **foreach** é a opção mais simples e recomendada quando você só precisa acessar cada elemento. O **for** pode ser usado quando você precisa acompanhar índices ou realizar iterações com condições mais complexas. O **while** é útil quando você precisa de um controle mais granular sobre a iteração.

6.7 Verificando tamanho de uma lista

List<T> fornece uma propriedade chamada **Count** que retorna o número de elementos presentes na lista.

Count

A propriedade **Count** é usada para obter o número de elementos em uma lista. Ela retorna um valor inteiro que representa o tamanho atual da lista.

```
List<int> numeros = new List<int>() { 10, 20, 30, 40, 50 };  
  
int tamanhoDaLista = numeros.Count;  
  
Console.WriteLine("Tamanho da lista: " +  
tamanhoDaLista);
```

Código 6.7.1

O valor 5 é o tamanho atual da lista, ou seja, o número de elementos presentes nela.

```
Saida: Tamanho da lista: 5
```

Execução código 6.7.1

É importante mencionar que a propriedade **Count** reflete o tamanho atual da lista, levando em consideração as adições e remoções de elementos. Ela é atualizada automaticamente conforme a lista é modificada.

6.8 Desafios

1 - Crie uma lista chamada "minhaLista". Verifique se a lista está vazia.

- 2** - Crie uma lista vazia chamada "minhaLista". Adicione os números de 1 a 5 à lista. Imprima a lista para verificar se os elementos foram adicionados corretamente.
- 3** - Crie uma lista com os números de 1 a 5 chamada "minhaLista". Acesse o primeiro elemento da lista e imprima-o.
- 4** - Crie uma lista com os números de 1 a 5 chamada "minhaLista". Verifique se o número 3 está presente na lista. Imprima o resultado.
- 5** - Crie uma lista com os números de 1 a 5 chamada "minhaLista". Remova o número 3 da lista. Imprima a lista para verificar se o elemento foi removido corretamente.
- 6** - Crie uma lista com os números de 1 a 5 chamada "minhaLista". Itere sobre a lista e imprima cada elemento em uma linha separada.
- 7** - Crie uma lista com os números de 1 a 5 chamada "minhaLista". Verifique o tamanho da lista e imprima o resultado.

7. Função

“Em um futuro próximo, ser um analfabeto em tecnologia será tão vergonhoso quanto ser um analfabeto em leitura e escrita”

7.1 O que é função

Função é um bloco de código que realiza uma tarefa específica. Ela agrupa um conjunto de instruções que podem ser chamadas e executadas em qualquer parte do programa.

Pense em uma função como uma "caixa" que recebe uma entrada, executa um conjunto de ações internas e pode retornar um resultado. Ela encapsula um conjunto de instruções lógicas, permitindo que essas instruções sejam reutilizadas em várias partes do programa sem a necessidade de repetição do código.

As funções ajudam a organizar o código em blocos mais gerenciáveis e facilitam a compreensão do programa como um todo. Elas permitem dividir um problema complexo em partes menores cada parte responsável por uma tarefa específica.

Além disso, as funções promovem a modularidade, pois você pode criar uma função separada para realizar uma determinada tarefa e, em seguida, chamar essa função sempre que precisar executar essa tarefa em diferentes partes do programa.

```
void MensagemBemVindo()  
{  
    Console.WriteLine("Bem-vindo(a)!");  
}  
  
MensagemBemVindo();
```

Código 7.1.1

Neste código, temos uma função chamada **MensagemBemVindo**. A função é declarada com o tipo de retorno **void**, o que significa que não retorna nenhum valor

Dentro da função **MensagemBemVindo**, temos uma única instrução que imprime a mensagem "**Bem-vindo(a)!**" no console usando **Console.WriteLine**.

Após a definição da função **MensagemBemVindo**, chamamos essa função usando o nome da função seguido por parênteses (). Essa chamada executa as instruções contidas na função **MensagemBemVindo**.

```
Bem-vindo(a)!
```

Execução código 7.1.1

- Definimos a função **MensagemBemVindo** usando a palavra-chave **void** para indicar que a função não retorna nenhum valor.
- Dentro da função **MensagemBemVindo**, temos uma única instrução **Console.WriteLine** que imprime a mensagem "**Bem-vindo(a)!**" no console.
- Em seguida, fora da definição da função, chamamos a função **MensagemBemVindo** usando o nome da função seguido por parênteses ().
- A chamada da função executa as instruções contidas na função **MensagemBemVindo**, resultando na impressão da mensagem "**Bem-vindo(a)!**" no console.

No exemplo código 7.1.1, a função **MensagemBemVindo** é usada para encapsular o código que imprime a mensagem Bem-Vindo(a)!

7.2 Parâmetro e argumento

Parâmetro e argumento são usados para permitir a passagem de dados entre a função que está sendo chamada e a função que está sendo definida.

Parâmetros: São variáveis declaradas na definição da função. Elas representam os valores que a função espera receber quando for chamada. Os parâmetros são como espaços reservados para receber os valores que serão passados para a função.

Argumentos: São os valores que são passados para a função quando ela é chamada. Eles preenchem os parâmetros da função, fornecendo os dados que a função precisa para realizar suas operações.

A relação entre parâmetros e argumentos é estabelecida quando uma função é chamada. Ao chamar uma função, você fornece os valores dos argumentos, que são atribuídos aos parâmetros correspondentes na definição da função.

```
string nomeUsuario = "João";  
Saudacao(nomeUsuario);  
void Saudacao(string nome)  
{  
    Console.WriteLine("Olá, " + nome + "! Bem-  
vindo(a)!");  
}
```

Código 7.2.1

Neste exemplo, temos uma função chamada **Saudacao** que possui um parâmetro nome do tipo **string**. A função imprime uma mensagem de saudação usando o valor do parâmetro nome.

Declaramos uma variável **nomeUsuario** e atribuímos o valor "**João**" a ela. Em seguida, chamamos a função **Saudacao** e passamos **nomeUsuario** como argumento.

```
Saida: Olá, João! Bem-vindo(a)!
```

Execução código 7.2.1

Os parâmetros permitem que você generalize uma função, tornando-a flexível para trabalhar com diferentes valores. Ao passar argumentos para os parâmetros, você pode personalizar a execução da função com base nos valores fornecidos.

É importante observar que os nomes dos parâmetros na definição da função e os nomes dos argumentos na chamada da função podem ser diferentes, mas é importante que a posição e a tipagem do dado sejam mantidas na ordem em que os argumentos são passados para os parâmetros.

7.3 Retorno

Retorno de uma função é o resultado que a função produz e retorna para quem a chamou. O retorno é usado quando uma função precisa computar um resultado e fornecê-lo para o ponto de chamada.

Para entender o conceito de retorno, vamos usar uma analogia com uma calculadora. Imagine que você tenha uma calculadora que recebe dois números, executa uma operação específica (por exemplo, soma) e retorna o resultado dessa operação.

Uma função pode receber dados de entrada, realizar cálculos e em seguida, retornar um resultado para quem a chamou.

```
int Soma(int a, int b)
{
    int resultado = a + b;
    return resultado;
}
```

Código 7.3.1

No exemplo [código 7.3.1](#), temos uma função chamada **Soma** que recebe dois números inteiros (**a** e **b**) como parâmetros. Dentro da função, realizamos a soma dos dois números e armazenamos o resultado na variável **resultado**.

Em seguida, usamos a palavra-chave **return** para retornar esse resultado para quem chamou a função.

```
int resultadoSoma = Soma(5, 3);  
Console.WriteLine(resultadoSoma);  
  
int Soma(int a, int b)  
{  
    int resultado = a + b;  
    return resultado;  
}
```

Código 7.3.2

Neste trecho de código, estamos chamando a função **Soma** com os argumentos **5** e **3**. A função executa a soma e retorna o valor **8**. Esse valor é armazenado na variável **resultadoSoma** e, em seguida, é impresso no console.

```
Saida: 8
```

Execução código 7.3.2

É possível passar uma função como argumento para outra função, veja o exemplo a seguir.

```
int resultado = SomarNovamente(Soma(5, 3), 2);
Console.WriteLine(resultado);
int Soma(int a, int b)
{
    int resultado = a + b;
    return resultado;
}
int SomarNovamente(int resultadoPrimeiraSoma, int
multiplicar){
    int resultado = resultadoPrimeiraSoma *
multiplicar;
    return resultado;
}
```

Código 7.3.3

- A chamada **Soma(5, 3)** retorna o valor **8**. Esse valor é passado como argumento para a função **SomarNovamente**, juntamente com o valor **2**.
- Dentro da função **SomarNovamente**, o valor **8** é multiplicado por **2**, resultando em **16**.
- O valor **16** é retornado pela função **SomarNovamente** e atribuído à variável **resultado**.
- Por fim, o valor **16** é impresso no console.

Saida: 16

Execução código 7.3.3

O retorno de uma função permite que você utilize esse resultado em outras partes do programa. Dessa forma, você pode reutilizar o código da função e obter o valor para realizar ações adicionais, como armazenar em variáveis, exibir ou passar como argumento para outras funções.

É importante notar que o tipo de dado do retorno precisa ser especificado na declaração da função (por exemplo, `int` para um valor inteiro, `string` para uma sequência de caracteres). Você pode usar outros tipos de dados como retorno, como `float`, `bool`, `object`, entre outros.

7.4 Argumento por referência

O argumento por referência é um conceito onde permite que uma função altere o valor de uma variável que está sendo passada como argumento.

Quando passamos um argumento para uma função, por padrão, é feita uma cópia do valor da variável e essa cópia é passada para a função. Dessa forma, qualquer alteração feita dentro da função não afeta o valor original da variável fora dela. Esse é o comportamento conhecido como "passagem por valor".

No entanto, em certas situações, pode ser necessário que uma função modifique diretamente o valor da variável original. É aí que entra o argumento por referência. Ao utilizar o modificador **ref** na declaração do parâmetro de uma função, estamos indicando que desejamos passar a própria referência da variável para a função, em vez de uma cópia do valor.

```
int numero = 5;

Console.WriteLine("Valor original: " + numero);
IncrementarPorReferencia(ref numero);

Console.WriteLine("Valor após a função
IncrementarPorReferencia: " + numero);

static void IncrementarPorReferencia(ref int valor)
{
    valor++;
}
```

Código 7.4.1

Neste exemplo, declaramos a função **IncrementarPorReferencia** com o parâmetro **ref int valor**. Dentro dessa função, incrementamos o valor do parâmetro **valor** em mais 1.

Declaramos a variável **numero** com o valor inicial de **5**. Em seguida, chamamos a função **IncrementarPorReferencia** passando **numero** como argumento, utilizando o modificador **ref**.

Valor original: 5

Valor após a função IncrementarPorReferencia: 6

Execução código 7.4.1

Perceba que, ao utilizar o argumento por referência, a função **IncrementarPorReferencia** modificou diretamente o valor da variável **numero**, refletindo a alteração fora da função.

É importante destacar que o uso de argumentos por referência pode levar a resultados indesejados e comprometer a legibilidade do código.

7.5 Valores por omissão

Valores por omissão, também conhecidos como valores padrão ou valores default, são utilizados para atribuir um valor predefinido a um parâmetro de função quando nenhum valor é especificado durante a chamada da função. Esse recurso é muito útil para fornecer valores padrão a parâmetros opcionais, evitando a necessidade de fornecer explicitamente um valor toda vez que a função é chamada.

É possível definir valores por omissão para parâmetros de função durante a sua declaração. Isso é feito atribuindo um valor à variável do parâmetro na declaração da função.

```
Saudacao();  
  
Saudacao("Olá, como vai?");  
  
static void Saudacao(string mensagem = "Olá, tudo  
bem?")  
{  
    Console.WriteLine(mensagem);  
}
```

Código 7.5.1

Neste exemplo, temos a função **Saudacao** com um parâmetro **mensagem**, que possui um valor padrão definido como **"Olá, tudo bem?"**.

Chamamos a função **Saudacao** duas vezes. Na primeira chamada, não fornecemos um argumento para o parâmetro **mensagem**, o que resulta na utilização do valor padrão definido na declaração da função. Na segunda chamada, passamos a string **"Olá, como vai?"** como argumento para o parâmetro **mensagem**, substituindo assim o valor padrão.

Saida: Olá, tudo bem?

Saida: Olá, como vai?

Execução código 7.5.1

Podemos ver que, na primeira chamada, onde nenhum argumento é fornecido, o valor padrão "**Olá, tudo bem?**" é utilizado. Já na segunda chamada, o valor padrão é substituído pelo argumento passado, resultando em "**Olá, como vai?**".

É importante deixar claro que, os parâmetros com valores por omissão devem ser declarados por último na lista de parâmetros da função.

```
static void ExemploFuncao(int a, int b, int c = 0)
{
    Console.WriteLine($"{a}, {b}, {c}");
}
```

Código 7.5.2

No exemplo código 7.5.2, temos a função **ExemploFuncao** com três parâmetros: **a**, **b** e **c**. O parâmetro **c** possui um valor por omissão, que é **0**.

Existe uma outra maneira de passar argumentos para uma função, conhecida como passagem de argumentos nomeados. Essa técnica permite especificar explicitamente o nome dos parâmetros ao chamar uma função e fornecer os valores correspondentes para esses parâmetros. Dessa forma, podemos fornecer os argumentos fora da ordem definida na função, o que torna a chamada mais flexível e legível.

A principal vantagem da passagem de argumentos nomeados é que não precisamos nos preocupar com a ordem dos argumentos ao chamar uma função, pois eles são associados aos parâmetros pelos

seus nomes. Isso é especialmente útil quando a função possui muitos parâmetros ou parâmetros opcionais com valores padrão.

```
void ImprimirInformacoes(string nome, int idade,
string cidade)
{
    Console.WriteLine($"Nome: {nome}, Idade: {idade},
Cidade: {cidade}");
}

ImprimirInformacoes(idade: 30, nome: "João", cidade:
"São Paulo");
```

Código 7.5.3

Nesse exemplo, temos uma função chamada **ImprimirInformacoes** que recebe três parâmetros: **nome**, **idade** e **cidade**. Ao chamar a função, fornecemos os argumentos nomeados, especificando explicitamente o nome de cada parâmetro seguido pelo valor correspondente. Observe que a ordem dos argumentos não segue a ordem dos parâmetros na definição da função.

```
Saida: Nome: João, Idade: 30, Cidade: São Paulo
```

Execução código 7.5.3

Ao usar a passagem de argumentos nomeados, podemos fornecer os argumentos na ordem desejada e até mesmo omitir alguns deles, desde que os parâmetros não obrigatórios tenham valores padrão. Por exemplo, podemos chamar a função omitindo o argumento cidade.

```
void ImprimirInformacoes(string nome, int idade,
string cidade = "São paulo")
{
    Console.WriteLine($"Nome: {nome}, Idade: {idade},
Cidade: {cidade}");
}

ImprimirInformacoes(idade: 30, nome: "João");
```

Código 7.5.4

Dessa forma, o valor padrão será usado para o parâmetro cidade. A passagem de argumentos nomeados melhora a legibilidade do código, pois fica mais claro qual valor está sendo atribuído a cada parâmetro, especialmente quando o código é lido por outras pessoas.

```
Saida: Nome: João, Idade: 30, Cidade: São paulo
```

Execução código 7.5.4

7.6 Recursividade

Recursividade é um conceito importante que permite que uma função chame a si mesma repetidamente até que uma condição de parada seja atendida. É uma técnica para resolver problemas que podem ser decompostos em subproblemas semelhantes.

A ideia central da recursividade é quebrar um problema maior em subproblemas menores e resolver cada subproblema chamando a mesma função. Cada chamada recursiva trabalha com um subconjunto menor de dados, avançando em direção à solução do problema original. A recursividade continua até que se atinja um caso-base, que é a condição que determina o fim da recursão.


```
int numero = 5;

int resultado = CalcularFatorial(numero);

Console.WriteLine($"O fatorial de {numero} é {resultado}.");

static int CalcularFatorial(int n)
{
    if (n == 0 || n == 1)
        return 1;

    return n * CalcularFatorial(n - 1);
}
```

Código 7.6.1

No código 7.6.1, temos uma função **CalcularFatorial** que calcula o fatorial de um número inteiro passado como argumento (**n**).

Declaramos uma variável **numero** com o valor **5**, que será o número para o qual queremos calcular o fatorial. Em seguida, chamamos a função **CalcularFatorial** passando **numero** como argumento e atribuímos o resultado à variável **resultado**. Por fim, imprimimos o resultado no console.

Na função **CalcularFatorial**, temos uma verificação utilizando um **if**. Se o valor de **n** for igual a **0** ou **1**, retornamos **1**, pois o fatorial de **0** e **1** é sempre igual a **1**. Caso contrário, calculamos o fatorial chamando a própria função **CalcularFatorial** com **n - 1** como argumento e multiplicamos o resultado pelo valor de **n**. Chamamos a função **CalcularFatorial** com o número **5** e imprimimos o resultado no console.

O fatorial de 5 é 120.

Execução código 7.6.1

A função **CalcularFatorial** é uma função recursiva, pois ela chama a si mesma dentro de sua própria definição. Isso acontece até que a condição de parada seja atingida, que é quando n é igual a **0** ou **1**, reduzindo gradualmente o valor de n até atingir o caso-base.

No caso do fatorial de 5, a recursão ocorre da seguinte forma:

- Chamada inicial: **CalcularFatorial(5)**
- Como n não é **0** nem **1**, a função chama **CalcularFatorial(4)**
- **CalcularFatorial(4)** chama **CalcularFatorial(3)**
- **CalcularFatorial(3)** chama **CalcularFatorial(2)**
- **CalcularFatorial(2)** chama **CalcularFatorial(1)**
- Finalmente, **CalcularFatorial(1)** retorna **1** para a chamada anterior (**CalcularFatorial(2)**), que multiplica por **2**, resultando em **2**
- **CalcularFatorial(2)** retorna **2** para a chamada anterior(**CalcularFatorial(3)**), que multiplica por **3**, resultando em **6**
- Esse processo continua até que **CalcularFatorial(5)** retorne **120**, que é o resultado final.

Dessa forma, a recursividade permite quebrar um problema em partes menores e resolvê-las de forma repetida até atingir o caso-base. O uso da função recursiva **CalcularFatorial** nesse exemplo nos permite calcular o fatorial de qualquer número inteiro positivo.

É importante ter cuidado ao utilizar a recursividade, pois ela pode consumir muita memória e tempo de execução se não for aplicada corretamente, resultando em um estouro da pilha de chamadas recursivas. É essencial garantir que a recursão tenha um caso-base definido e que cada chamada recursiva se aproxime da condição de parada, evitando a ocorrência de chamadas infinitas.

7.7 Função na prática

Vamos considerar uma situação em que você precise realizar várias operações matemáticas, como calcular a soma, a subtração e a multiplicação de dois números diferentes. Em vez de escrever o código repetidamente para cada operação, podemos usar funções para tornar o código mais organizado e reutilizável.

Vamos começar definindo três funções diferentes: Somar, Subtrair e Multiplicar. Cada função receberá dois números como argumentos e retornará o resultado da operação correspondente.

```
int Somar(int a, int b)
{
    int resultado = a + b;
    return resultado;
}

int Subtrair(int a, int b)
{
    int resultado = a - b;
```



```
        return resultado;
    }

    int Multiplicar(int a, int b)
    {
        int resultado = a * b;
        return resultado;
    }
```

Código 7.7.1

Agora, vamos utilizar essas funções em um exemplo prático. Suponha que você queira realizar várias operações matemáticas com diferentes pares de números.

```
int numero1 = 5;
int numero2 = 3;
int resultadoSoma = Somar(numero1, numero2);
Console.WriteLine($"A soma de {numero1} e {numero2} é {resultadoSoma}");
int resultadoSubtracao = Subtrair(numero1, numero2);
Console.WriteLine($"A subtração de {numero1} e {numero2} é {resultadoSubtracao}");
int resultadoMultiplicacao = Multiplicar(numero1, numero2);
Console.WriteLine($"A multiplicação de {numero1} e {numero2} é {resultadoMultiplicacao}");
```

Código 7.7.2

No exemplo [código 7.7.2](#), estamos usando as funções Somar, Subtrair e Multiplicar para realizar as operações matemáticas desejadas. As funções recebem os números numero1 e numero2 como argumentos e retornam o resultado da operação correspondente.

```
int Somar(int a, int b)
{
    int resultado = a + b;
    return resultado;
}

int Subtrair(int a, int b)
{
    int resultado = a - b;
    return resultado;
}

int Multiplicar(int a, int b)
{
    int resultado = a * b;
    return resultado;
}
```



```
int numero1 = 5;
int numero2 = 3;
int resultadoSoma = Somar(numero1, numero2);
Console.WriteLine($"A soma de {numero1} e {numero2} é {resultadoSoma}");
int resultadoSubtracao = Subtrair(numero1, numero2);
Console.WriteLine($"A subtração de {numero1} e {numero2} é {resultadoSubtracao}");
int resultadoMultiplicacao = Multiplicar(numero1, numero2);
Console.WriteLine($"A multiplicação de {numero1} e {numero2} é {resultadoMultiplicacao}");
```

Código 7.7.3

```
A soma de 5 e 3 é 8
A subtração de 5 e 3 é 2
A multiplicação de 5 e 3 é 15
```

Execução código 7.7.3

Observe como o código fica mais organizado e legível ao usar funções. Além disso, caso precisemos realizar as mesmas operações com diferentes pares de números, podemos simplesmente chamar as funções novamente, evitando repetição de código.

As funções nos permitem encapsular blocos de código e reutilizá-los em diferentes partes do programa. Isso promove a modularização, facilitando a manutenção do código e tornando-o mais compreensível.

7.8 Função de callback

Função de callback é uma função que é passada como argumento para outra função e é chamada de volta em algum ponto dessa função

principal. Ela permite que você especifique um comportamento que será executado quando uma determinada condição for atendida ou quando uma operação for concluída.

Vamos considerar um exemplo em que temos uma função chamada **ExecutarOperacao**. Essa função simula uma operação demorada, e quando a operação é concluída, ela chama a função de **callback** fornecida.

```
void ExecutarOperacao(Action callback)
{
    // Simulação de uma operação assíncrona
    Thread.Sleep(2000); // Aguarda 2 segundos para
    execução a função abaixo
    callback();
}

void FuncaoDeCallback()
{
    Console.WriteLine("Operação concluída!");
}

ExecutarOperacao(FuncaoDeCallback);
```

Código 7.8.1

Neste exemplo, a função **ExecutarOperacao** recebe uma função de callback como argumento.

Usamos o **Thread.Sleep** para aguardar 2 segundos (representando a duração da operação). Em seguida, a função de callback é chamada.

A função de callback **FuncaoDeCallback** simplesmente imprime a mensagem "**Operação concluída!**" no console.

Finalmente, chamamos a função **ExecutarOperacao** passando **FuncaoDeCallback** como a função de callback a ser executada após a conclusão da operação.

Quando o código é executado, ocorre um atraso de 2 segundos (devido ao **Thread.Sleep**) para simular a operação demorada, e então a mensagem "**Operação concluída!**" é exibida no console.

7.9 Função anônima

Função anônima é uma função sem nome que pode ser definida e usada imediatamente, em vez de ser declarada como uma função separada. As funções anônimas são úteis quando você precisa de uma função temporária e simples, geralmente como um argumento para outra função.

Expressão Lambda

A palavra "lambda" origina-se do cálculo lambda, um sistema formal desenvolvido por Alonzo Church na década de 1930 para estudar fundamentos da computação e funções matemáticas.

A expressão lambda é uma forma concisa de criar funções anônimas. Ela é denotada pelo operador **=>** (conhecido como operador "lambda" ou "Arrow function"), **(parâmetros) => expressão**.

Aqui está um exemplo de uma expressão lambda que recebe dois inteiros e retorna a sua soma.

```
Func<int, int, int> soma = (a, b) => a + b;  
int resultado = soma(3, 4);  
Console.WriteLine(resultado);
```

Código 7.9.1

Saida: 7

Execução código 7.9.1

A expressão lambda pode ser usada em várias situações, como ordenação, filtragem de coleções, programação assíncrona etc.

Também é possível criar uma função anônima como um argumento para outra função.

```
AplicarOperacao(5, 3, (a, b) => Console.WriteLine(a +
b));

static void AplicarOperacao(int x, int y, Action<int,
int> operacao)
{
    operacao(x, y);
}
```

Código 7.9.2

No código 7.9.2 a primeira chamada **AplicarOperacao(5, 3, (a, b) => Console.WriteLine(a + b));** passamos os argumentos **5, 3** e a expressão lambda **(a, b) => Console.WriteLine(a + b)**. Essa expressão lambda recebe dois parâmetros **a** e **b** e exibe a soma de **a** e **b** usando **Console.WriteLine**. Portanto, essa chamada imprimirá **8** na saída.

A função **AplicarOperacao** é definida como **static void AplicarOperacao(int x, int y, Action<int, int> operacao)**. Ela recebe dois parâmetros inteiros **x** e **y** e um parâmetro **operacao** do tipo **Action<int, int>**. Essa função simplesmente executa a função representada pela expressão lambda passada como argumento, chamando **operacao(x, y)**. Portanto, a função **AplicarOperacao** atua como uma função utilitária que aplica uma operação específica aos valores **x** e **y** dentro da função anônima para exibir o resultado diretamente na tela.

Saida: 8

Execução código 7.9.2

Delegado Anônimo

Delegado anônimo é uma forma mais antiga de criar função anônima. Ele permite criar uma função sem nome usando a sintaxe de um delegado, mas sem a necessidade de declarar explicitamente um delegado separado. Por exemplo, o seguinte código cria um delegado anônimo que recebe um parâmetro *x* e imprime seu valor.

```
Action<int> imprimir = delegate(int x)
{
    Console.WriteLine(x);
};
imprimir(42);
```

Código 7.9.3

Definimos uma variável chamada **imprimir** do tipo **Action<int>**. Essa variável é um **delegate** que representa uma função sem retorno que recebe um argumento inteiro.

Em seguida, temos a sintaxe **delegate(int x) { Console.WriteLine(x); }** para definir uma função anônima. Essa função anônima recebe um parâmetro **x** e imprime o valor de **x** usando **Console.WriteLine**. A função anônima está sendo atribuída à variável **imprimir**.

Saida: 8

Execução código 7.9.3

Por fim, chamamos a função representada pela variável **imprimir**, passando o valor **42** como argumento, **imprimir(42)**. Isso executará a função anônima definida anteriormente e imprimirá 42 na saída.

7.10 Desafios

1 - Crie uma função chamada "calcularAreaRetangulo" que recebe dois parâmetros: largura e altura. A função deve calcular e retornar a área do retângulo. Em seguida, chame a função com valores diferentes para largura e altura e imprima o resultado.

2 - Crie uma função chamada "verificarMaiorNumero" que recebe dois números como parâmetros e retorna o maior entre eles. Em seguida, chame a função com diferentes pares de números e imprima o maior valor retornado.

3 - Crie uma função chamada "incrementar" que recebe um parâmetro inteiro por referência e incrementa seu valor em 1. Em seguida, chame a função passando uma variável inteira como argumento e imprima o valor antes e depois da chamada da função.

4 - Crie uma função chamada "saudacaoPersonalizada" que recebe dois parâmetros: nome e mensagem. O parâmetro mensagem tem um valor padrão "Bem-vindo(a)!" caso não seja fornecido um argumento. A função deve imprimir a mensagem de saudação para o nome especificado. Em seguida, chame a função passando apenas o nome e depois passando o nome e uma mensagem personalizada.

5 - Crie uma função chamada "calcularFatorial" que recebe um número inteiro como parâmetro e retorna o fatorial desse número. Use recursividade para implementar essa função. Em seguida, chame a função com um número de sua escolha e imprima o resultado.

6 - Crie uma função chamada "calcularMedia" que recebe um array de números como parâmetro e retorna a média desses números. Em seguida, crie um array de números e chame a função para calcular a média, imprimindo o resultado.

7 - Crie uma função chamada "processarLista" que recebe uma lista de números e uma função de callback como parâmetros. A função "processarLista" deve aplicar a função de callback a cada elemento da lista e imprimir o resultado. Em seguida, crie uma função de callback

chamada "dobrarNumero" que recebe um número como parâmetro e retorna o dobro desse número. Chame a função "processarLista" passando uma lista de números e a função de callback "dobrarNumero".

8 - Crie uma função anônima some 2 números e exiba na tela.

8. Memória Stack e Heap

“Você não precisa ser um gênio da computação para ser um bom programador, mas ser um bom programador te torna um gênio da computação”

8.1 Memória Stack

A memória stack, também conhecida como pilha, é uma área de memória usada para armazenar informações temporárias relacionadas à execução de funções. Ela é organizada de forma (LIFO - Last-In, First-Out). Isso significa que o último item adicionado à pilha é o primeiro a ser removido.

Vamos imaginar a memória stack como uma pilha física de objetos, como pratos em uma pilha. Você pode adicionar pratos na parte superior da pilha e retirá-los. Cada novo prato adicionado à pilha se torna o topo, e quando você remove um prato, o prato abaixo dele se torna o novo topo.

Na memória stack, em vez de pratos, temos frames (quadros) que representam as chamadas de função. Cada vez que uma função é chamada, um novo frame é criado e adicionado à pilha.

Parâmetros da função: Os valores passados para a função quando ela é chamada. Esses valores são armazenados no frame para que a função possa acessá-los durante a sua execução.

Variáveis locais: São variáveis declaradas dentro da função e que existem apenas durante a execução dessa função. Elas são armazenadas no frame e podem ser acessadas apenas dentro do escopo da função.

Endereço de retorno: É o endereço de memória onde o programa deve continuar sua execução após a função ser concluída. Quando a função é chamada, o endereço de retorno é adicionado ao frame da função atual na pilha.

A cada chamada de função, um novo frame é criado e adicionado ao topo da pilha. Quando uma função é concluída, seu frame correspondente é removido da pilha, e o programa volta para o endereço de retorno armazenado no frame anterior, continuando a execução a partir desse ponto.

```
Funcao1();  
void Funcao1()  
{  
    int x = 10;  
    Funcao2();  
    Console.WriteLine(x);  
}  
void Funcao2()  
{  
    int y = 20;  
}
```

Código 8.1.1

No código 8.1.1, quando a função **Funcao1** é chamada, um novo frame é criado e adicionada à pilha. Esse frame contém a variável **x** com o valor **10**. Em seguida, a função **Funcao2** é chamada, criando um novo frame e adicionando-o à pilha. Esse novo frame contém a variável **y** com o valor **20**.

Saida: 8

Execução código 8.1.1

Quando a execução da função **Funcao2** é concluída, seu frame é removido da pilha e o programa retorna para a função **Funcao1**, continuando a execução de onde parou. A variável **x** ainda existe no frame da **Funcao1**, permitindo que o valor **10** seja impresso no console.

Esse processo continua conforme o programa avança, com novas chamadas de função criando frames e removendo-as da pilha à medida que são concluídas.

8.2 Memória Heap

A memória heap é uma área de memória para alocar e desalocar blocos de memória de forma dinâmica. Diferentemente da memória stack, que é usada para armazenar informações temporárias relacionadas à execução de funções, a memória heap é utilizada para armazenar dados que precisam ser alocados e liberados em momentos específicos durante a execução do programa.

A memória heap é chamada de "heap" porque ela não é organizada de forma estruturada como a pilha (stack). Em vez disso, ela é uma região de memória livre, onde blocos de memória podem ser alocados e desalocados de forma arbitrária.

Vamos usar uma analogia para entender a memória heap. Imagine uma prateleira com livros. Você pode colocar livros na prateleira em qualquer ordem e retirá-los quando necessário. A prateleira não segue uma ordem específica, permitindo que você aloque e libere espaço conforme necessário.

Para alocar memória na heap, podemos usar vetores (arrays). A alocação de um vetor na heap é realizada por meio da solicitação de um bloco de memória de tamanho específico.

```
// Alocação de vetores na heap
int[] array = new int[5];
string[] nomes = new string[3];
// Uso dos vetores alocados na heap
array[0] = 42;
nomes[0] = "João";
nomes[1] = "Maria";
// Impressão dos valores
Console.WriteLine(array[0]);
Console.WriteLine(nomes[0]);
Console.WriteLine(nomes[1]);
// Liberação de memória
array = null;
nomes = null;
```

Código 8.2.1

No código 8.2.1, alocamos vetores na memória **heap**. O **array** é um vetor de inteiros com tamanho **5**, e **nomes** é um vetor de strings com tamanho **3**.

Em seguida, podemos usar esses vetores alocados atribuindo valores a elementos específicos. No exemplo, atribuímos o valor **42** ao primeiro elemento do vetor **array**, e atribuímos nomes às posições **0** e **1** do vetor **nomes**.

Após o uso dos vetores, podemos liberar a memória da **heap** atribuindo **null** às variáveis, indicando que elas não apontam mais para o bloco de memória alocado. Isso permite que o coletor de lixo do sistema operacional libere a memória não utilizada posteriormente.

Saida: 42

Saida: João

Saida: Maria

Execução código 8.2.1

No entanto, não é obrigatório atribuir o valor **null**, uma vez que o coletor de lixo possui inteligência suficiente para determinar o momento adequado para limpar os dados em memória.

Referência

Quando criamos um vetor (ou qualquer objeto) e atribuímos a referência desse vetor a outra variável, ambas as variáveis apontam para o mesmo objeto na memória heap. Isso significa que qualquer modificação feita no objeto através de uma das variáveis será refletida quando você acessar o objeto através da outra variável.

```
int[] vetor1 = { 1, 2, 3 }; // Cria um vetor
int[] vetor2 = vetor1;
vetor2[0] = 10;
Console.WriteLine(vetor1[0]);
```

Código 8.2.2

No código 8.2.2, temos dois vetores, **vetor1** e **vetor2**. Inicialmente, **vetor1** contém os números **1**, **2** e **3**.

Em seguida, atribuímos a referência de **vetor1** à variável **vetor2** usando a declaração **vetor2 = vetor1**; agora, **vetor2** está apontando para o mesmo objeto na memória **heap** que **vetor1**.

Quando modificamos o primeiro elemento do **vetor2** para **10** através da instrução **vetor2[0] = 10**; essa modificação também é refletida no vetor **vetor1**. Isso ocorre porque ambos os vetores estão apontando para o mesmo objeto na memória **heap**.

Ao imprimir o primeiro elemento do vetor **vetor1** usando **Console.WriteLine(vetor1[0]);** o valor impresso será **10**, porque a alteração feita no objeto através de **vetor2** foi refletida em **vetor1**.

Saida: 10

Execução código 8.2.2

Essa é a essência das referências, quando você tem várias variáveis que apontam para o mesmo objeto na memória heap, qualquer modificação feita no objeto através de uma das variáveis, será visível em todas as outras variáveis que estão apontando para o mesmo objeto.

É importante ter em mente esse comportamento ao trabalhar com referências, pois alterações em um objeto podem ter efeitos colaterais em outras partes do código que também estão referenciando esse objeto.

9. Strings

“Seja apaixonado por seu código, mas apaixone-se mais pelo problema que ele resolve.”

9.1 O que são strings

String é uma coleção de caracteres que são tratados como um único objeto. Por exemplo, a frase **"Olá, mundo!"** pode ser armazenada como uma **string**.

```
string nome = "João"
```

Código 9.1.1

Para acessar um caractere específico em uma **string**, podemos usar a indexação, onde cada caractere é associado a um índice. É importante observar que a indexação em **strings** geralmente começa em **zero**, o que significa que o primeiro caractere tem o índice **zero**, o segundo caractere tem o índice **um** e assim por diante.

```
string nome = "João";  
char segundaLetra = nome[1];  
Console.WriteLine(segundaLetra);
```

Código 9.1.2

Por exemplo, se tivermos a **string "nome"** e quisermos acessar o segundo caractere, que é **"o"**, podemos usar a notação **nome[1]**. O índice **1** representa o segundo elemento da **string** e nos permite recuperar a letra **"o"**.

```
Saida: o
```

Execução código 9.1.2

9.2 Composto palavra usando char

Para ilustrar de forma clara que uma string é, de fato, uma cadeia de caracteres.

```
char[] palavra = new char[] { 'H', 'e', 'l', 'l', 'o'
};
string novaPalavra = "";
for(int i = 0; i < palavra.Length; i++){
    novaPalavra += palavra[i];
}
for(int i = 0; i < palavra.Length; i++){
    Console.Write(novaPalavra[i]);
}
```

Código 9.2.1

No código 9.2.1. Foi criado um vetor de caracteres chamado palavra, que representa a palavra "**Hello**". Em seguida, é declarada uma variável **novaPalavra** como uma **string** vazia.

Através de um loop for, cada caractere do vetor palavra é adicionado à variável **novaPalavra**, formando uma nova palavra.

Posteriormente, um novo loop **for** é utilizado para percorrer cada caractere da **novaPalavra** e exibe cada caractere no console.

```
Saida: Hello
```

Execução código 9.2.1

9.3 Strings são imutáveis

Strings são objetos imutáveis, o que significa que uma vez criadas, elas não podem ser modificadas. Isso ocorre porque a classe string foi

projetada para ser imutável, ou seja, o conteúdo de uma string não pode ser alterado após sua criação.

Quando você realiza operações em strings, como concatenação ou substituição de caracteres, na verdade está criando uma nova string com o resultado da operação, em vez de modificar a string original.

```
string texto1 = "Olá";  
string texto2 = texto1 + " mundo!";
```

Código 9.3.1

No código 9.3.1, a variável **texto2** contém uma nova **string** resultante da concatenação dos valores **"Olá"** e **" mundo!"**. A **string** original **texto1** permanece inalterada.

```
string texto1 = "Olá";  
texto1 = "fa";
```

Código 9.3.2

No código 9.3.2 quando atribuímos o valor **"Olá"** à variável **texto1**, uma nova instância da **string** com o valor **"Olá"** é criada na memória. A variável **texto1** passa a fazer referência a essa instância.

Em seguida, quando você atribui o valor **"fa"** à mesma variável **texto1**, uma nova instância da **string** com o valor **"fa"** é criada na memória. Agora, a variável **texto1** é atualizada para fazer referência a essa nova instância.

Nesse caso, a nova instância de **string** com o valor **"Olá"** ainda existe na memória, mas como não há mais nenhuma variável que faça referência a ela, ela se torna elegível para ser coletada pelo coletor de lixo (garbage collector).

No entanto, a imutabilidade das **strings** ainda se mantém. Isso significa que, embora você esteja atribuindo um novo valor à variável **texto1**, não está modificando a **string** original **"Olá"** nem a **string** **"fa"**.

Você está apenas criando novas instâncias de **string** e fazendo com que a variável **texto1** aponte para essas novas instâncias.

```
string texto1 = "Olá";  
texto1[0] = 't';
```

Código 9.3.3

No exemplo 9.3.3 tentamos atribuir o valor **'t'** ao primeiro caractere da string **texto1**. No entanto, essa operação resultará em um erro de compilação, pois como já mencionado, as **strings** em C# são somente leitura. Os caracteres de uma **string** não podem ser modificados diretamente.

9.4 Comparação de strings

A comparação de strings é uma operação comum e envolve a comparação de dois valores de string para determinar se eles são iguais ou diferentes. Existem várias formas de realizar comparações de strings, cada uma com suas características e casos de uso específicos.

Operador de igualdade (==)

O operador de igualdade é utilizado para verificar se duas strings são exatamente iguais. Ele compara os valores de cada caractere das strings, levando em consideração maiúsculas e minúsculas.

```
string texto1 = "Olá";  
string texto2 = "Olá";  
bool saoIguais = (texto1 == texto2);  
Console.WriteLine(saoIguais);
```

Código 9.4.1

No código 9.4.1, o operador **==** compara os valores de **texto1** e **texto2** e retorna **true** porque as duas **strings** têm o mesmo conteúdo.

```
Saida: true
```

Execução código 9.4.1

Método Equals()

O método Equals() é usado para comparar se duas strings são iguais. Ele retorna um valor booleano indicando se as strings são iguais ou não. Ao contrário do operador ==, o método Equals() permite especificar opções de comparação, como ignorar maiúsculas e minúsculas.

```
string texto1 = "Olá";  
string texto2 = "olá";  
  
bool saoIguais = texto1.Equals(texto2,  
    StringComparison.CurrentCultureIgnoreCase);  
  
Console.WriteLine(saoIguais);
```

Código 9.4.2

Nesse caso, **Equals()** é usado com a opção **StringComparison.CurrentCultureIgnoreCase**, que compara as **strings** ignorando diferenças de **maiúsculas** e **minúsculas**.

```
Saida: true
```

Execução código 9.4.2

Além dessas abordagens básicas, também é possível usar outros métodos e classes para realizar comparações de **strings** mais avançadas, como **String.CompareOrdinal()**, **String.CompareOptions** e **StringComparer**. Essas opções oferecem maior flexibilidade e controle sobre a comparação de **strings** em diferentes contextos.

9.5 Manipulação de strings

A manipulação de strings envolve a realização de várias operações, como concatenação, substituição, extração, pesquisa etc. Existem várias classes e métodos que facilitam a manipulação de strings.

Concatenação de strings

A concatenação é a operação de combinar duas ou mais strings em uma única string. Podemos usar o operador + ou o método `Concat()` para realizar a concatenação.

```
string texto1 = "Olá";  
string texto2 = "mundo!";  
string textoConcatenado = String.Concat(texto1, " ",  
texto2);  
Console.WriteLine(textoConcatenado);
```

Código 9.5.1

No código 9.5.1, estamos usando o método **Concat** da classe `String`. O método **Concat** recebe dois ou mais argumentos do tipo `string` e os concatena em uma única string.

No exemplo acima, passamos as variáveis **texto1**, um espaço em branco e **texto2** como argumentos para o método **Concat**. O resultado é armazenado na variável **textoConcatenado**.

```
Saida: Olá mundo!
```

Execução código 9.5.1

Substituição de caracteres

A substituição envolve a troca de um caractere ou conjunto de caracteres por outro valor desejado em uma string. Vamos usar o método `Replace()` para realizar substituições.


```
string texto = "Olá mundo!";  
string novoTexto = texto.Replace("mundo", "amigos");  
Console.WriteLine(novoTexto);
```

Código 9.5.2

Nesse caso, o método **Replace()** é usado para substituir a palavra **"mundo"** pela palavra **"amigos"** na string **texto**, gerando a nova string **novoTexto**.

Saida: Olá amigos!

Execução código 9.5.2

Extração de substrings

A extração de substrings envolve a obtenção de uma parte específica de uma string original. Vamos usar o método **Substring()** para extrair uma substring com base em um índice inicial e um comprimento.

```
string texto = "Olá mundo!";  
string substring = texto.Substring(4, 5);  
Console.WriteLine(substring);
```

Código 9.5.3

Nesse caso, **Substring()** é usado para extrair uma substring de texto começando no índice **4 (letra "m")** e com um comprimento de **5** caracteres, resultando na substring **"mundo"**.

Saida: mundo

Execução código 9.5.3

Pesquisa em strings

A pesquisa envolve encontrar a ocorrência de uma determinada sequência de caracteres em uma string. Podemos usar o método

Contains(), IndexOf(), LastIndexOf() e StartsWith() para realizar diferentes tipos de pesquisas.

```
string texto = "Olá mundo!";  
bool contemMundo = texto.Contains("mundo");  
int indiceM = texto.IndexOf("m");  
Console.WriteLine(contemMundo);  
Console.WriteLine(indiceM);
```

Código 9.5.4

Nesse caso, **Contains()** verifica se a string **texto** contém a sequência **"mundo"**, enquanto **IndexOf()** retorna o índice da primeira ocorrência da letra **"m"** na string **texto**.

Saida: True

Saida: 4

Execução código 9.5.4

Além desses métodos, existem muitos outros e propriedades disponíveis para manipulação de strings, como ToUpper(), ToLower(), Trim(), Split(), Length, Substring(), PadLeft(), PadRight() etc. Cada um deles possui funcionalidades específicas para facilitar a manipulação e transformação de strings.

É importante lembrar que, como as strings são imutáveis, cada operação de manipulação de strings retorna uma nova instância de string com o resultado da operação, em vez de modificar a string original. Portanto, é necessário atribuir o resultado da operação a uma nova variável ou atualizar a variável existente para armazenar a nova string resultante.

9.6 StringBuilder

Ao contrário da classe `string`, que é imutável (não pode ser modificada após a criação), o `StringBuilder` oferece métodos para modificar diretamente seu conteúdo sem a necessidade de criar novas instâncias de `string` a cada modificação. Isso é especialmente útil quando você precisa realizar várias operações de manipulação de strings em sequência, como concatenação, substituição ou inserção de caracteres.

Criação de uma instância do `StringBuilder`

Importe a biblioteca **`System.Text`** no início do seu arquivo adicionando a diretiva **`using System.Text`**.

```
using System.Text;
```

Código 9.6.1

Agora você precisa criar uma instância da classe.

```
using System.Text;  
  
StringBuilder sb = new StringBuilder();
```

Código 9.6.2

Também é possível fornecer um valor inicial para o **`StringBuilder`** através de sua sobrecarga de construtor.

```
using System.Text;  
  
StringBuilder sb = new StringBuilder("Olá");  
  
Console.WriteLine(sb);
```

Código 9.6.3

```
Saida: Olá
```

Execução código 9.6.3

Concatenação de strings

`StringBuilder` oferece o método `Append()` para concatenar strings ao seu conteúdo. Ele permite adicionar strings, caracteres, valores numéricos e outros tipos de dados.

```
using System.Text;

StringBuilder sb = new StringBuilder("Olá");

sb.Append(" mundo!");

Console.WriteLine(sb);
```

Código 9.6.4

Nesse caso, o método **Append()** adiciona a string " **mun**do!" ao conteúdo existente do **StringBuilder**.

Saida: Olá mundo!

Execução código 9.6.4

Substituição de caracteres

`StringBuilder` oferece o método `Replace()` para substituir caracteres ou sequências de caracteres em seu conteúdo. Ele recebe como argumentos a sequência a ser substituída e a sequência de substituição.

```
using System.Text;

StringBuilder sb = new StringBuilder("Olá mundo!");

sb.Replace("mundo", "amigos");

Console.WriteLine(sb);
```

Código 9.6.5

No código 9.6.5, o método **Replace()** substitui a sequência "**mundo**" por "**amigos**" no conteúdo do **StringBuilder**.

Saida: Olá amigos!

Execução código 9.6.5

Inserção de caracteres

`StringBuilder` também oferece o método `Insert()` para inserir caracteres ou sequências de caracteres em posições específicas dentro do seu conteúdo. Ele recebe como argumentos o índice de inserção e a sequência a ser inserida.

```
using System.Text;

StringBuilder sb = new StringBuilder("Olá!");

sb.Insert(4, " mundo");

Console.WriteLine(sb);
```

Código 9.6.6

Nesse caso, o método **`Insert()`** insere a sequência " mundo" na posição de índice 4 do conteúdo do **`StringBuilder`**

Saida: Olá! mundo

Execução código 9.6.6

Conversão para string

Quando você terminar de realizar as operações desejadas com o `StringBuilder`, pode converter seu conteúdo em uma string através do método **`ToString()`**.

```
using System.Text;

StringBuilder sb = new StringBuilder("Olá");

sb.Append(" mundo!");

string resultado = sb.ToString();

Console.WriteLine(resultado);
```

Código 9.6.7

O método **`ToString()`** retorna a representação do conteúdo do **`StringBuilder`** como uma **string**.

```
Saida: Olá! mundo
```

Execução código 9.6.7

Diferente dos métodos tradicionais de concatenação de strings, o `StringBuilder` oferece desempenho superior, minimizando a criação desnecessária de novas instâncias.

9.7 Desafios

- 1** - Escreva um programa que solicite ao usuário que digite seu nome e, em seguida, imprima uma mensagem de boas-vindas usando a string digitada.
- 2** - Escreva um programa que solicite ao usuário que digite uma palavra e, em seguida, exiba cada caractere separadamente em uma nova linha.
- 3** - Escreva um programa que declare uma variável string contendo um texto e, em seguida, tente modificar um caractere específico da string. Observe como ocorre um erro de compilação devido à imutabilidade das strings.
- 4** - Escreva um programa que solicite ao usuário que digite duas palavras e, em seguida, compare as strings para verificar se são iguais. Imprima uma mensagem informando se as palavras são iguais ou diferentes.
- 5** - Escreva um programa que solicite ao usuário que digite uma frase. Em seguida, realize as seguintes operações:
 - Imprima o número total de caracteres na frase.
 - Converta a frase para letras maiúsculas e imprima-a.
 - Verifique se a frase contém a palavra "exemplo" e imprima o resultado.
- 6** - Escreva um programa que declare um objeto `StringBuilder` vazio. Em seguida, solicite ao usuário que digite uma frase e adicione-a ao

objeto `StringBuilder`. Por fim, imprima a frase invertida utilizando o método `Reverse` do `StringBuilder`.

10. Try Catch e Finally

“Não se torne um programador de uma linguagem, torne-se um programador de problemas.”

10.1 Tratamento de exceções

Tratamento de exceções permite lidar com erros e situações inesperadas que podem ocorrer durante a execução de um programa. O bloco de código responsável por tratar exceções é denominado "try-catch-finally".

O bloco "try" é usado para envolver o código que pode gerar uma exceção, é dentro deste bloco que colocamos o código que pode

lançar uma exceção. Se ocorrer uma exceção dentro do bloco "try", ela será "capturada" e tratada pelo bloco "catch".

O bloco "catch" é usado para especificar o tipo de exceção que você deseja capturar e como deseja lidar com ela. É possível ter vários blocos "catch" em um único bloco "try", cada um capturando um tipo diferente de exceção. Quando uma exceção é lançada, o fluxo do programa é desviado para o bloco "catch" correspondente ao tipo de exceção lançada.

```
try
{
    // Código que pode gerar uma exceção
}
catch (TipoDeExcecao1 excecao1)
{
    // Tratamento da excecao1
}
catch (TipoDeExcecao2 excecao2)
{
    // Tratamento da excecao2
}
finally
{
    // Código que sempre será executado,
    independentemente de ocorrer ou não uma exceção
}
```

Exemplo

Dentro do bloco **catch**, você pode escrever código para lidar com a exceção capturada. Isso pode incluir, exibir uma mensagem de erro, registrar informações sobre a exceção, tentar corrigir o problema ou tomar qualquer outra ação.

O bloco **finally** é opcional e usado para especificar um código que será executado independentemente de ocorrer ou não uma exceção. Vale ressaltar que a presença do bloco **finally** não é obrigatória. É possível ter apenas o bloco **try-catch** sem incluir o bloco **finally**.

```
try
{
    int divisor = 0;
    int resultado = 10 / divisor; // Gera uma exceção
    de divisão por zero
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Ocorreu um erro de divisão por
    zero: " + ex.Message);
}
finally
{
    Console.WriteLine("Este bloco será sempre
    executado, independentemente de ocorrer uma exceção ou
    não.");
}
```

Código 10.1.1

No código 10.1.1, o código dentro do bloco **try** tenta executar uma operação de divisão por zero, o que gera uma exceção do tipo

DivideByZeroException. Essa exceção é capturada pelo bloco **catch** correspondente ao tipo de exceção **DivideByZeroException**, onde uma mensagem de erro é exibida. Em seguida, o fluxo do programa segue para o bloco **finally**, onde uma mensagem de finalização é exibida.

Saida: Ocorreu um erro de divisão por zero: Attempted to divide by zero.

Saida: Este bloco será sempre executado, independentemente de ocorrer uma exceção ou não.

Execução código 10.1.1

10.2 Exceções personalizadas

Antes de abordarmos exceções personalizadas, é importante ressaltar que nos depararemos com códigos mais complexos que não foram previamente abordados neste livro. O tema que discutiremos agora está relacionado à programação orientada a objetos, que não faz parte do escopo deste livro. Portanto, é válido mencionar que o conteúdo a seguir pode ser desafiador. Caso você encontre dificuldades em compreender o que está sendo abordado, sugiro que utilize a ideia apresentada como uma “receita de bolo” e busque informações adicionais sobre programação orientada a objetos.

É possível criar exceções personalizadas para lidar com situações específicas do seu programa. Isso pode ser útil quando você deseja tratar erros ou comportamentos personalizados que não são cobertos pelas exceções padrão fornecidas pela linguagem.

Para criar uma exceção personalizada, você deve criar uma classe que herde da classe base `Exception` ou de uma de suas subclasses. Essa classe personalizada deve fornecer construtores e propriedades adicionais, conforme necessário, para fornecer informações relevantes sobre a exceção.

```

public class MeuErroPersonalizadoException : Exception
{
    public MeuErroPersonalizadoException()
    {
    }

    public MeuErroPersonalizadoException(string
mensagem) : base(mensagem)
    {
    }

    public MeuErroPersonalizadoException(string
mensagem, Exception innerException) : base(mensagem,
innerException)
    {
    }
}

```

Código 10.2.1

No exemplo acima, criamos uma classe chamada **MeuErroPersonalizadoException** que herda da classe base **Exception**. A classe possui três construtores: um construtor padrão sem parâmetros, um construtor que aceita uma mensagem de erro e um construtor que aceita uma mensagem de erro e uma exceção interna.

Agora, podemos utilizar a exceção personalizada no código da mesma maneira que utilizamos as exceções padrão.

```
try
{
    // Algum código que pode gerar sua exceção
    personalizada

    throw new MeuErroPersonalizadoException("Ocorreu
    um erro personalizado.");
}
catch (MeuErroPersonalizadoException ex)
{
    Console.WriteLine("Exceção personalizada
    capturada: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("Exceção genérica capturada: " +
    ex.Message);
}
finally
{
    Console.WriteLine("Finalizando o bloco try-catch-
    finally.");
}
```



```

public class MeuErroPersonalizadoException : Exception
{
    public MeuErroPersonalizadoException()
    {
    }

    public MeuErroPersonalizadoException(string
mensagem) : base(mensagem)
    {
    }

    public MeuErroPersonalizadoException(string
mensagem, Exception innerException) : base(mensagem,
innerException)
    {
    }
}

```

Código 10.2.2

No exemplo acima, o código dentro do bloco **try** gera a exceção personalizada **MeuErroPersonalizadoException** usando a palavra-chave **throw**. Essa exceção é capturada pelo bloco **catch** correspondente ao tipo **MeuErroPersonalizadoException**, onde uma mensagem de erro específica é exibida. Se ocorrer outra exceção que não seja da classe **MeuErroPersonalizadoException**, ela será capturada pelo bloco **catch** genérico **Exception**. O bloco **finally** será executado independentemente da exceção ser capturada ou não.

Desafios

1 - Se o usuário digitar um valor não numérico, exiba a mensagem "Entrada inválida! Digite apenas números inteiros."

2 - Crie uma exceção personalizada chamada `NomeInvalidoException` que será lançada quando um usuário digitar um nome vazio ou nulo.

11. Programação assíncrona

“Se você não pode explicar algo de forma simples, você não entende o suficiente.”

11.1 O que é programação assíncrona

É um paradigma de programação que lida com a execução de tarefas de forma não sequencial, permitindo que outras tarefas sejam executadas simultaneamente enquanto se aguarda a conclusão de uma operação assíncrona. Nesse modelo, o fluxo de execução do programa não é bloqueado esperando por uma operação demorada.

As tarefas são executadas em background e notificam quando estão concluídas, em vez de bloquear o programa principal até que a tarefa seja finalizada. Isso permite que o programa seja mais eficiente, pois pode realizar outras operações enquanto espera a resposta de uma operação assíncrona, como requisições de rede, acesso a bancos de dados, leitura ou gravação de arquivos, entre outras.

11.2 Criando um arquivo TXT

Antes de nos aprofundarmos na programação assíncrona, vamos aprender a criar um arquivo de texto. Isso nos ajudará a entender melhor como funciona a programação assíncrona.

```
using System.IO;  
  
string nomeArquivo = "exemplo.txt";  
  
string conteudo = "Conteúdo do arquivo.";
```



```

try
{
    using (StreamWriter writer = new
StreamWriter(nomeArquivo))
    {
        writer.WriteLine(contenido);
    }
    Console.WriteLine("Arquivo criado com sucesso!");
}
catch (Exception ex)
{
    Console.WriteLine("Ocorreu um erro ao criar o
arquivo: " + ex.Message);
}

```

Código 11.2.1

No código 11.2.1, usamos a classe **StreamWriter** para criar um novo arquivo chamado **exemplo.txt** e escrever no arquivo. O bloco **using** garante que o arquivo seja corretamente liberado após o uso.

Caso ocorra algum erro ao criar o arquivo, a exceção será capturada e uma mensagem de erro será exibida. Caso contrário, uma mensagem de sucesso será mostrada no console.

Saida: Arquivo criado com sucesso!

Execução código 11.2.1

Lembre-se de adicionar a declaração **using System.IO** no início do arquivo para utilizar as classes de manipulação de arquivos. Certifique-

se também de ter permissão para escrever no diretório onde o arquivo será criado.

11.3 Criando arquivos TXT usando programação assíncrona

Exemplo de código onde o programa espera um arquivo ser criado antes de criar o próximo arquivo.

```
using System.IO;
using System.Threading.Tasks;
class Program
{
    static async Task CriarArquivoAsync(string
nomeArquivo, string conteudo)
    {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(2));
            using (StreamWriter writer = new
StreamWriter(nomeArquivo))
            {
                await writer.WriteAsync(conteudo);
            }
            Console.WriteLine($"Arquivo
'{nomeArquivo}' criado com sucesso!");
        }
    }
}
```



```
}

    catch (Exception ex)
    {
        Console.WriteLine($"Ocorreu um erro ao
criar o arquivo '{nomeArquivo}': {ex.Message}");
    }
}

static async Task Main()
{
    string nomeArquivo1 = "arquivo1.txt";
    string nomeArquivo2 = "arquivo2.txt";
    string conteudo = "Este é o conteúdo do
arquivo.";

    Console.WriteLine("Início do programa.");

    await CriarArquivoAsync(nomeArquivo1,
conteudo);
    await CriarArquivoAsync(nomeArquivo2,
conteudo);
    Console.WriteLine("Fim do programa.");
}
}
```

Código 11.3.1

Definição do método assíncrono CriarArquivoAsync(): Este método recebe um nome de arquivo e um conteúdo como parâmetros. Ele é marcado com a palavra-chave **async Task**, indicando que é um método assíncrono. Isso permite que ele use a sintaxe **await** para aguardar a conclusão de operações assíncronas.

Simulação de atraso com Task.Delay(): Antes da criação do arquivo, utilizamos o método **Task.Delay(TimeSpan.FromSeconds(2))**. Ele cria uma tarefa que aguarda um determinado período de tempo antes de continuar a execução. No nosso caso, utilizamos um atraso de **2 segundos** para simular uma operação mais demorada.

Criação do arquivo: Após o atraso simulado, o código continua e cria o arquivo utilizando a classe **StreamWriter**. Usamos o método **WriteAsync()** para escrever o conteúdo no arquivo de forma assíncrona.

Tratamento de exceções: O bloco **try-catch** envolve o código da criação do arquivo para capturar e tratar possíveis exceções. Se ocorrer algum erro durante a criação do arquivo, a mensagem de erro será exibida no console.

Chamadas assíncrona: Chamamos o método **CriarArquivoAsync()** duas vezes, cada uma para criar um arquivo diferente. Utilizamos a palavra-chave **await** para aguardar a conclusão de cada chamada assíncrona antes de prosseguir para a próxima linha de código.

```
Saida: Arquivo 'arquivo1.txt' criado com sucesso!
```

```
Saida: Arquivo 'arquivo2.txt' criado com sucesso!
```

```
Saida: Fim do programa.
```

Execução código 11.3.1

Durante a execução, cada chamada assíncrona cria um arquivo e aguardará um atraso de 2 segundos antes de continuar para a próxima chamada. Isso permite que você observe o efeito da programação assíncrona em ação, com atrasos perceptíveis entre cada criação de arquivo.

Também temos a possibilidade de não esperar um arquivo ser criado antes de criar o próximo arquivo. Para fazer isso, basta remover a palavra-chave **await** do código. No entanto, é importante adicionar o método **Console.ReadLine()** no final do código, para que o programa não seja finalizado. Isso ocorre porque o método **await** suspende a execução do programa até que a tarefa assíncrona seja concluída.

```
CriarArquivoAsync(nomeArquivo1, conteudo);  
CriarArquivoAsync(nomeArquivo2, conteudo);  
Console.ReadLine();  
Console.WriteLine("Fim do programa.");
```

Modificação código 11.3.1

Com essas modificações, os arquivos serão criados simultaneamente, sem aguardar um pelo outro. Assim, você poderá observar que a criação dos arquivos ocorre de forma paralela, aproveitando a capacidade de processamento disponível.

Exemplo de código onde o programa cria todos os arquivos simultaneamente.

```
using System.IO;
using System.Threading.Tasks;

class Program
{
    static async Task CriarArquivoAsync(string
nomeArquivo, string conteudo)
    {
        try
        {
            using (StreamWriter writer = new
StreamWriter(nomeArquivo))
            {
                await writer.WriteAsync(conteudo);
            }

            Console.WriteLine($"Arquivo
'{nomeArquivo}' criado com sucesso!");
        }
    }
}
```



```
}

    catch (Exception ex)
    {
        Console.WriteLine($"Ocorreu um erro ao
criar o arquivo '{nomeArquivo}': {ex.Message}");
    }
}

static async Task Main()
{
    string nomeArquivo1 = "arquivo1.txt";
    string nomeArquivo2 = "arquivo2.txt";
    string conteudo = "Este é o conteúdo do
arquivo.";

    Console.WriteLine("Início do programa.");

    Task task1 = CriarArquivoAsync(nomeArquivo1,
conteudo);

    Task task2 = CriarArquivoAsync(nomeArquivo2,
conteudo);

    await Task.WhenAll(task1, task2);

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

Código 11.3.2

Definição do método assíncrono CriarArquivoAsync(): O método **CriarArquivoAsync()** é marcado como `async` e possui a mesma implementação anterior. Ele cria um arquivo de texto com o conteúdo fornecido de forma assíncrona.

Criação das tarefas: Declaramos duas variáveis de tarefa (`Task`), **task1** e **task2**, para representar as chamadas assíncronas para criar os arquivos. Não há espera explícita entre essas tarefas, permitindo que sejam executadas simultaneamente.

Chamada assíncrona e criação simultânea: Chamamos **CriarArquivoAsync()** para criar cada arquivo usando as variáveis de tarefa correspondentes (**task1** e **task2**). As tarefas são iniciadas imediatamente, sem esperar uma pela outra.

Task.WhenAll(): Usamos o método estático **Task.WhenAll()** para aguardar a conclusão de todas as tarefas. Essa função retorna uma nova tarefa que é concluída somente quando todas as tarefas fornecidas estiverem concluídas.

Aguardando a conclusão: Utilizamos a palavra-chave **await** para aguardar a conclusão da tarefa retornada pelo **Task.WhenAll()**. Isso garante que o programa aguarde até que todas as tarefas de criação de arquivo tenham sido concluídas antes de prosseguir.

```
Saida: Arquivo 'arquivo1.txt' criado com sucesso!
```

```
Saida: Arquivo 'arquivo2.txt' criado com sucesso!
```

```
Saida: Fim do programa.
```

Execução código 11.3.2

12. O que fazer agora

“Não se preocupe com o fracasso; você só precisa acertar uma vez.”

12.1 Conteúdo recomendado

Parabéns por ter concluído este livro! Ele forneceu uma base sólida para você criar seus primeiros programas. Agora, é hora de avançar em seus estudos.

Nesse estágio, eu recomendo que você direcione seus estudos para a programação orientada a objetos (POO). A POO é um paradigma importante e amplamente utilizado na programação moderna.

Essa abordagem facilita a construção de programas modulares e de fácil manutenção, além de oferecer benefícios como reutilização de código.

12.2 Primeiro emprego

Se você está em busca do seu primeiro emprego, é fundamental que você demonstre todo seu conhecimento. Recomendo que você faça isso através da publicação dos seus projetos. Existe uma ferramenta essencial chamada GitHub (<https://github.com/>), uma plataforma da Microsoft na qual desenvolvedores compartilham seus projetos com outros profissionais da área.

Não se contente apenas com projetos simples; busque constantemente a evolução e o aprimoramento dos seus projetos. É essencial que você publique regularmente no GitHub para que os contratantes possam acompanhar o seu progresso.

LinkedIn(<https://linkedin.com/>) é uma rede social muito importante para encontrar oportunidades de emprego. Essa ferramenta oferece a possibilidade de se conectar com milhares de profissionais e empresas. Além disso, o LinkedIn oferece a oportunidade de criar um perfil profissional completo, onde você pode destacar suas experiências, habilidades e conquistas.