



K9

TREINAMENTOS

Lógica de Programação

Lógica de Programação em C#

28 de novembro de 2015



| | |
|--|------------|
| Sumário | ii |
| Prefácio | vii |
| 1 Introdução | 1 |
| 1.1 O que é um Computador? | 1 |
| 1.2 Comunicação | 2 |
| 1.3 Formato Binário | 3 |
| 1.4 Unidades | 5 |
| 1.5 Arquiteturas de Processadores | 7 |
| 1.6 O que é um Programa? | 8 |
| 1.7 Linguagem de Máquina | 9 |
| 1.8 Linguagem de Programação | 9 |
| 1.9 Compilador | 10 |
| 1.10 Sistemas Operacionais | 10 |
| 1.11 Máquinas Virtuais | 12 |
| 1.12 Editores de Texto | 14 |
| 1.13 Terminal | 14 |
| 1.14 Hello World em C# | 14 |
| 1.15 O que é o Método Main? | 16 |
| 1.16 Variações do Método Main | 17 |
| 1.17 Argumentos de Linha de Comando | 18 |
| 1.18 Exibindo Mensagens | 20 |
| 1.19 Comentários | 21 |
| 1.20 Indentação | 22 |
| 1.21 Engenharia Reversa | 22 |
| 1.22 Ofuscadores | 23 |
| 1.23 Erro: Compilar um arquivo inexistente | 23 |
| 1.24 Erro: Executar utilizando a extensão .cs | 23 |
| 1.25 Erro: Não fechar os blocos | 23 |
| 1.26 Erro: Não fechar as aspas | 24 |
| 1.27 Erro: Trocar maiúsculas e minúsculas | 24 |
| 1.28 Erro: Esquecer o ponto e vírgula | 25 |
| 1.29 Erro: Esquecer o Main | 25 |
| 1.30 Erro: Utilizar sequências de escape inválidas | 26 |
| 1.31 Resumo | 26 |
| 2 Variáveis | 29 |
| 2.1 O que é uma Variável? | 29 |
| 2.2 Declarando e Inicializando Variáveis | 29 |
| 2.3 Exibindo os Valores das Variáveis | 31 |
| 2.4 Copiando Valores | 31 |
| 2.5 Tipos Simples | 33 |
| 2.6 Nullable Types | 38 |
| 2.7 String | 38 |
| 2.8 Data e Hora | 38 |
| 2.9 Valores Literais | 39 |
| 2.10 Constantes | 44 |
| 2.11 Números Aleatórios | 45 |
| 2.12 Convenções de Nomenclatura | 46 |

| | |
|---|------------|
| 2.13 Regras de Nomenclatura | 48 |
| 2.14 Palavras Reservadas | 49 |
| 2.15 Erro: Variáveis com nomes repetidos | 49 |
| 2.16 Erro: Esquecer a inicialização de uma variável local | 50 |
| 2.17 Erro: Trocar aspas simples por aspas duplas ou vice-versa | 50 |
| 2.18 Erro: Utilizar o separador decimal errado | 51 |
| 2.19 Erro: Valores incompatíveis com os tipos das variáveis | 51 |
| 2.20 Erro: Esquecer dos caracteres de tipagem para float ou decimal | 52 |
| 2.21 Resumo | 53 |
| 3 Operadores | 55 |
| 3.1 Introdução | 55 |
| 3.2 Conversões Entre Tipos Simples | 55 |
| 3.3 Conversão Entre Tipos Simples e String | 57 |
| 3.4 Operadores Aritméticos | 60 |
| 3.5 Tipo do Resultado de uma Operação Aritmética | 62 |
| 3.6 Divisão Inteira | 63 |
| 3.7 Overflow e Underflow | 68 |
| 3.8 Regras para Operações Aritméticas com Valores Especiais | 70 |
| 3.9 Concatenação de Strings | 71 |
| 3.10 Operadores Unários + e - | 73 |
| 3.11 Operadores de Atribuição | 73 |
| 3.12 Operadores de Comparação | 77 |
| 3.13 Operadores Lógicos | 79 |
| 3.14 Operador Ternário ?: | 86 |
| 3.15 Operador de Negação | 87 |
| 3.16 Incremento e Decremento | 89 |
| 3.17 Avaliando uma Expressão | 94 |
| 3.18 Manipulação de Strings | 98 |
| 3.19 Operações com Data e Hora | 102 |
| 3.20 Erro: Utilizar operandos e operadores incompatíveis | 103 |
| 3.21 Erro: Divisão inteira por zero | 104 |
| 3.22 Erro: Armazenamento de valores incompatíveis | 105 |
| 3.23 Erro: Castings não permitidos | 105 |
| 3.24 Resumo | 106 |
| 4 Controle de Fluxo | 109 |
| 4.1 Introdução | 109 |
| 4.2 Instruções de Decisão | 109 |
| 4.3 Instrução if | 109 |
| 4.4 Instrução else | 113 |
| 4.5 Instruções de Decisão Encadeadas | 118 |
| 4.6 Instruções de Repetição | 121 |
| 4.7 Instrução while | 121 |
| 4.8 Instrução for | 126 |
| 4.9 Instruções de Repetição Encadeadas | 131 |
| 4.10 Instrução break | 138 |
| 4.11 Instrução continue | 150 |
| 4.12 Blocos Sem Chaves | 157 |
| 4.13 “Laços Infinitos” | 158 |

| | |
|--|------------|
| 4.14 Instrução switch | 158 |
| 4.15 Instrução do-while | 162 |
| 4.16 Unreachable Code | 165 |
| 4.17 Erro: Não utilizar condições booleanas | 165 |
| 4.18 Erro: Else sem if | 166 |
| 4.19 Erro: Else com condição | 166 |
| 4.20 Erro: Ponto e vírgula excedente | 167 |
| 4.21 Erro: “Laço infinito” | 167 |
| 4.22 Erro: Chave do switch com tipos incompatíveis | 168 |
| 4.23 Erro: Casos do switch com expressões não constantes | 168 |
| 4.24 Erro: Break ou continue fora de um laço | 169 |
| 4.25 Erro: Usar vírgula ao invés de ponto e vírgula no laço for | 169 |
| 4.26 Resumo | 170 |
| 5 Array | 173 |
| 5.1 Introdução | 173 |
| 5.2 O que é um Array? | 173 |
| 5.3 Referências | 174 |
| 5.4 Declaração | 175 |
| 5.5 Inicialização | 175 |
| 5.6 Acessando o Conteúdo de um Array | 176 |
| 5.7 Alterando o Conteúdo de um Array | 177 |
| 5.8 Outras Formas de Inicialização | 178 |
| 5.9 Percorrendo um Array | 179 |
| 5.10 Array Multidimensional | 180 |
| 5.11 Erro: Utilizar valores incompatíveis como índices de um array | 188 |
| 5.12 Erro: Definir outras dimensões ao criar arrays irregulares | 189 |
| 5.13 Erro: Acessar uma posição inválida de um array | 189 |
| 5.14 Resumo | 189 |
| 6 Métodos | 191 |
| 6.1 Introdução | 191 |
| 6.2 Estrutura Geral de um Método | 191 |
| 6.3 Parâmetros | 193 |
| 6.4 Resposta | 193 |
| 6.5 Passagem de Parâmetros | 198 |
| 6.6 Sobrecarga | 200 |
| 6.7 Parâmetros Variáveis | 202 |
| 6.8 Erro: Parâmetros incompatíveis | 204 |
| 6.9 Erro: Resposta incompatível | 205 |
| 6.10 Erro: Esquecer a instrução return | 205 |
| 6.11 Erro: Não utilizar parênteses | 207 |
| 6.12 Resumo | 208 |
| 7 String | 211 |
| 7.1 Referências | 211 |
| 7.2 Pool de Strings | 212 |
| 7.3 Imutabilidade | 213 |
| 7.4 StringBuilder | 214 |
| 7.5 Formatação | 215 |

| | | |
|-----|-------------------------------------|-----|
| 7.6 | Formatação de Data e Hora | 220 |
| 7.7 | Resumo | 221 |





Prefácio

O conteúdo deste livro é uma introdução à lógica de programação e à linguagem C#. Apesar de introdutório, os tópicos deste livro são apresentados com bastante profundidade. Portanto, este material não é adequado para quem procura apenas um conhecimento superficial sobre programação.

O leitor não precisa ter experiência com programação. Mas, é necessário estar acostumado a utilizar computadores no seu dia a dia e ter conhecimento sobre os tópicos básicos de matemática abordados no ensino fundamental.

Organização

No Capítulo 1, serão apresentados os principais elementos de um computador e alguns conceitos básicos como linguagem de máquina, linguagem de programação, compilador, sistema operacional e máquina virtual. Além disso, o leitor terá o primeiro contato com um programa escrito em linguagem C#.

No Capítulo 2, o conceito de variável será apresentado. Veremos o processo de criação de variáveis utilizando a linguagem C#. Além disso, mostraremos os tipos primitivos dessa linguagem e também o tipo `String`.

No Capítulo 3, veremos como as variáveis podem ser manipuladas através dos operadores da linguagem C#. Serão apresentadas as operações de conversão, as aritméticas, as de atribuição, as de comparação e as lógicas. Além disso, mostraremos o funcionamento do operador de negação e do operador ternário.

No Capítulo 4, serão apresentadas as instruções de decisão e de repetição da linguagem C#. Veremos como o fluxo de execução de um programa pode ser controlado com as instruções `if`, `else`, `while`, `for`, `switch`, `do`, `break` e `continue`.

No Capítulo 5, será apresentado o conceito de array. Mostraremos como criar e manipular arrays unidimensionais ou multidimensionais. Além disso, discutiremos os principais erros relacionados aos arrays.

No Capítulo 6, mostraremos como reutilizar código através da criação de métodos. Veremos como definir e utilizar os parâmetros e o retorno de um método. Além disso, apresentaremos o conceito de sobrecarga e varargs.

No Capítulo 7, características específicas dos objetos do tipo `string` serão discutidas. Dentre elas, falaremos sobre imutabilidade, pool de strings e string builder. Além disso, mostraremos os recursos da plataforma .NET para formatação de strings.

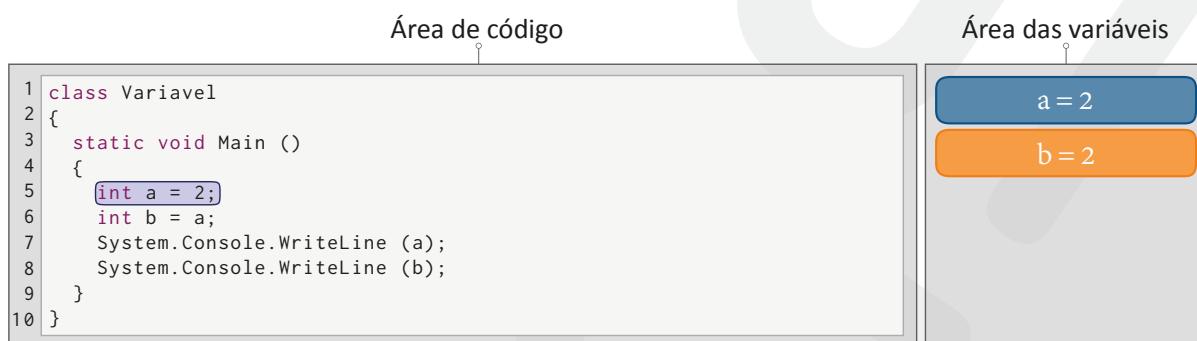
Seções

As seções são classificadas de acordo com o nível de dificuldade. Seções que tratam de assuntos considerados básicos, são marcadas com a figura . A imagem  está associada às seções que

cobrem assuntos com um nível intermediário de dificuldade. Já as seções com o ícone abordam assuntos considerados de nível avançado.

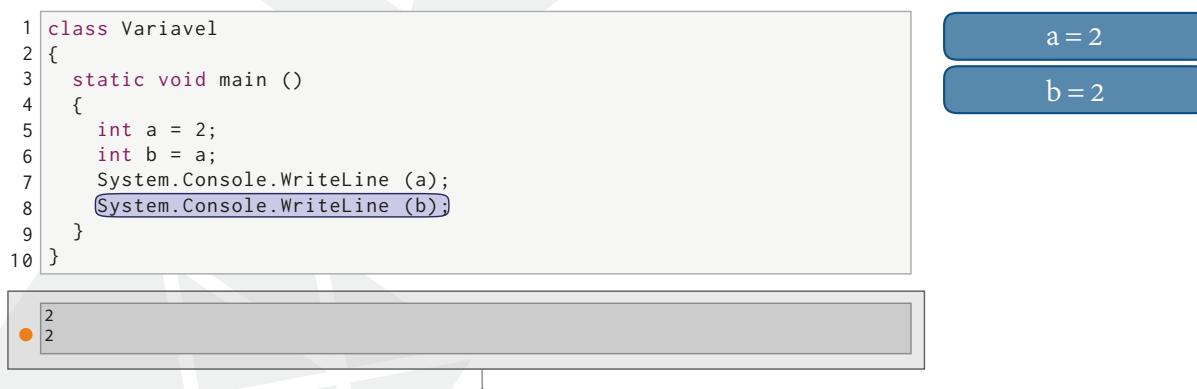
Simulações

Para facilitar o entendimento do leitor, diversas simulações ilustradas foram adicionadas neste livro. O objetivo das simulações é mostrar passo a passo a execução de um programa. A imagem abaixo apresenta a ilustração de um passo de uma simulação.



Na área de código, é apresentado o código do programa cuja a execução está sendo simulada. Na área das variáveis, é possível visualizar as variáveis criadas até o momento e os seus valores atuais. A linha ou instrução que está sendo executada no passo atual é destacada na área de código. Na área das variáveis, as variáveis que foram alteradas no passo atual da simulação são apresentadas em uma caixa de cor laranja e as que não foram alteradas em uma caixa de cor azul.

As mensagens exibidas pelos programas na saída padrão são apresentadas na área de saída. As linhas exibidas no passo atual da simulação serão destacadas com uma círculo laranja.



INTRODUÇÃO

1.1 O que é um Computador?



Atualmente, os computadores estão presentes no cotidiano da maioria das pessoas. Você, provavelmente, já está acostumado a utilizar computadores no seu dia a dia. Mas, será que você conhece o funcionamento básico de um computador? A seguir, listaremos os principais elementos de um computador e suas respectivas funções.

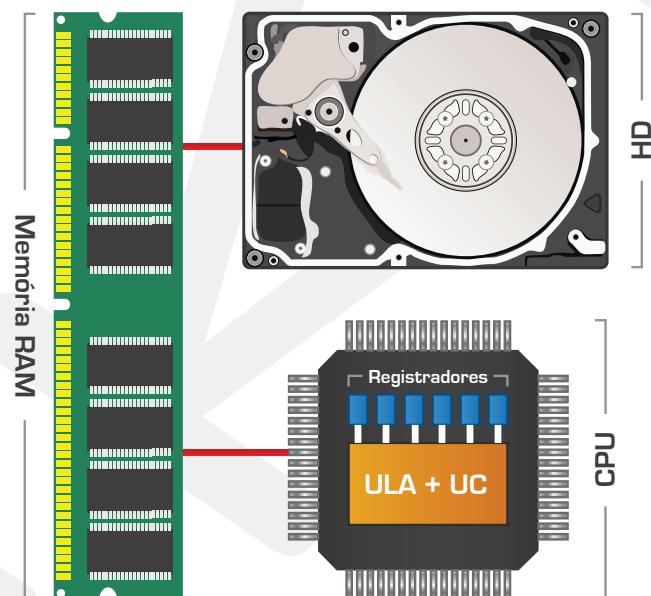


Figura 1.1: Principais elementos de um computador

HD (Disco Rígido): A principal função dos *HDs* é armazenar dados. Geralmente, os documentos que os usuários salvam, por exemplo, arquivos de texto, planilhas eletrônicas, apresentações, imagens, áudios e vídeos são guardados nos *HDs*. Normalmente, os dados e as operações dos programas que os usuários instalam nos computadores também são mantidos nos discos rígidos. O conteúdo armazenado nos *HDs* é persistente, ou seja, não é descartado quando os computadores são desligados. Geralmente, os discos rígidos são capazes de armazenar uma grande quantidade de dados. Contudo, as operações de acesso e de armazenamento de dados não é considerada rápida.

Memória RAM: A principal função da memória *RAM* é armazenar dados. O acesso e o armazenamento de dados na memória *RAM* é bem mais rápido do que nos *HDs*. Por isso, quando os usuários executam um programa, os dados e operações desse programa são copiados do *HD* para a memória *RAM*. Analogamente, os documentos abertos pelos usuários também são copiados do *HD* para a memória *RAM*. Geralmente, a quantidade de dados que podemos armazenar na memória *RAM* é bem menor do que a quantidade de dados que podemos armazenar nos *HDs*. Quando os computadores são desligados, os dados guardados na memória *RAM* são descartados.

CPU (Unidade Central de Processamento - Processador): Basicamente, a tarefa da *CPU* é executar operações aritméticas e operações lógicas. A *UC* (Unidade de Controle), a *ULA* (Unidade Lógica e Aritmética) e os registradores são componentes básicos da *CPU*. Normalmente, a *UC* copia uma operação de um programa armazenado na memória *RAM* e guarda os dados dessa operação nos registradores. Na sequência, a *ULA* executa a operação acessando os dados armazenados nos registradores e guarda o resultado dessa operação também nos registradores. Por fim, a *UC* copia o resultado armazenado nos registradores para a memória *RAM*.

1.2 Comunicação



Os computadores são capazes de se comunicar com dispositivos periféricos como teclado, mouse, monitor, caixa de som, impressoras, projetores, entre outros. Eles também são capazes de se comunicar com outros computadores. Essa comunicação é realizada através das diversas portas físicas que os computadores possuem. A seguir, listaremos algumas portas físicas e as suas respectivas funções.

Ethernet: Utilizada para conectar um computador a uma rede local de computadores. Através dessa porta, um computador pode enviar e receber dados de outros computadores.



Figura 1.2: Porta Ethernet

Paralela: Essa porta foi criada para conectar um computador a uma impressora. Hoje, é utilizada também para conectar computadores a scanners, câmeras de vídeo, entre outros dispositivos.



Figura 1.3: Porta Paralela

PS/2: Teclados e mouses antigos são conectados aos computadores através dessa porta.



Figura 1.4: Porta PS/2

USB: Atualmente, é a porta mais utilizada. Diversos dispositivos são conectados aos computadores através das portas *USB*. Por exemplo, teclados, mouses, impressoras, celulares, HDs externos, entre outros.



Figura 1.5: Porta USB

HDMI: Essa porta é utilizada para transmissão digital de áudio e vídeo.



Figura 1.6: Porta HDMI

1.3 Formato Binário



Os computadores são capazes de receber, armazenar e enviar dados. Contudo, os computadores só trabalham com dados em formato binário. A maior parte das pessoas não está acostumada a utilizar o formato binário no seu dia a dia.



Analogia

Os textos que você está acostumado a escrever ou ler são escritos com as letras do Alfabeto Latino. As 26 letras básicas do Alfabeto Latino são: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y e Z.

Em outras partes do mundo, outros alfabetos são utilizados. Por exemplo, na Grécia, as pessoas utilizam o Alfabeto Grego. No Mundo Árabe, o Alfabeto Árabe. Na China, o Alfabeto Chinês.

Por analogia, podemos dizer que os computadores utilizam o Alfabeto Binário. Esse alfabeto é composto pelo caractere 0 e pelo caractere 1. Todos os dados manipulados por um computador devem ser definidos em formato binário, ou seja, com os caracteres 0 e 1.

Números

As pessoas estão acostumadas a lidar com os números em formato decimal. Os computadores, por outro lado, trabalham com números em formato binário. Veja, a seguir, as representações decimal e binária de alguns números.

| Decimal | Binário | Decimal | Binário | Decimal | Binário |
|---------|---------|---------|---------|---------|---------|
| 0 | 0 | 10 | 1010 | 20 | 10100 |
| 1 | 1 | 11 | 1011 | 21 | 10101 |
| 2 | 10 | 12 | 1100 | 22 | 10110 |
| 3 | 11 | 13 | 1101 | 23 | 10111 |
| 4 | 100 | 14 | 1110 | 24 | 11000 |
| 5 | 101 | 15 | 1111 | 25 | 11001 |
| 6 | 110 | 16 | 10000 | 26 | 11010 |
| 7 | 111 | 17 | 10001 | 27 | 11011 |
| 8 | 1000 | 18 | 10010 | 28 | 11100 |
| 9 | 1001 | 19 | 10011 | 29 | 11101 |

Tabela 1.1: Representação decimal e binária

A quantidade mínima de dígitos binários necessários para definir a representação binária de um número depende da grandeza do mesmo. Por exemplo, para representar o número 4, são necessários pelo menos três dígitos binários. Agora, para representar o número 29, são necessários pelo menos cinco dígitos binários.

Caracteres

Nos computadores, os caracteres de um texto também devem ser definidos em formato binário. Para realizar essa transformação, primeiramente, cada caractere é associado a um valor numérico inteiro. Por exemplo, o caractere “A” e o caractere “?” são normalmente associados aos valores 65 e 63, respectivamente. A representação binária de cada caractere corresponde à representação binária do valor numérico associado a esse caractere.

Os caracteres são mapeados para formato binário através de padrões como *ASCII* (<http://pt.wikipedia.org/wiki/ASCII>) e *Unicode* (<http://pt.wikipedia.org/wiki/Unicode>). Veja, a seguir, a representação binária de alguns caracteres de acordo com o padrão *ASCII*.

| Caractere | Decimal | Binário | Caractere | Decimal | Binário |
|-----------|---------|----------|-----------|---------|----------|
| A | 65 | 01000001 | L | 76 | 01001100 |
| B | 66 | 01000010 | M | 77 | 01001101 |
| C | 67 | 01000011 | N | 78 | 01001110 |
| D | 68 | 01000100 | O | 79 | 01001111 |
| E | 69 | 01000101 | P | 80 | 01010000 |
| F | 70 | 01000110 | Q | 81 | 01010001 |
| G | 71 | 01000111 | R | 82 | 01010010 |
| H | 72 | 01001000 | S | 83 | 01010011 |
| I | 73 | 01001001 | T | 84 | 01010100 |
| J | 74 | 01001010 | U | 85 | 01010101 |
| K | 75 | 01001011 | V | 86 | 01010110 |

Tabela 1.2: Representação binária de caracteres seguindo o padrão ASCII

Normalmente, cada caractere corresponde a uma sequência composta por oito ou dezesseis dígitos binários. A quantidade de dígitos utilizados limita a quantidade de caracteres que podem ser

representados. Por exemplo, com oito dígitos binários, podemos representar no máximo 256 caracteres. Com dezesseis dígitos binários, podemos representar até 65536 caracteres.

Imagens, áudios e vídeos

Como vimos, os números e os caracteres de um texto são facilmente representados em formato binário. Contudo, os computadores também são capazes de manipular imagens, áudio e vídeo. Para esses tipos de dados, a transformação para formato binário é bem mais complicada. Pesquise por *PNG*, *MP3* e *AVI* que são formatos binários de imagens, áudios e vídeos através dos seguintes endereços:

- *PNG* - <http://www.w3.org/TR/PNG/>
- *MP3* - <http://en.wikipedia.org/wiki/MP3>
- *AVI* - http://en.wikipedia.org/wiki/Video_Interleave

Resumidamente, para que um dado possa ser manipulado por um computador, ele deve ser convertido para o formato binário de acordo com algum padrão.

1.4 Unidades



Você deve estar acostumado a medir distâncias utilizando as unidades de comprimento do Sistema Internacional de Medidas (*International System of Units – SI*) como **milímetro**, **centímetro**, **metro** e **quilômetro**. Os americanos e os ingleses utilizam com maior frequência as unidades de medida do *Imperial Unit* como **polegada**, **pé**, **jarda** e **milha**.

De forma análoga, é importante ser capaz de medir a quantidade de dados que um computador pode armazenar ou transmitir. Essa mensuração pode ser realizada com ajuda das unidades de medida. A unidade de medida básica é o **bit**. Cada 0 ou 1 que um computador armazena ou transmite é um bit. Por padrão, um **byte** corresponde a 8 bits. O símbolo utilizado para representar o byte é a letra **B**.

A Comissão Eletrotécnica Internacional (*International Electrotechnical Commission – IEC*) e o Sistema Internacional de Medidas (*International System of Units – SI*) definem unidades de medida relacionadas ao byte. As unidades do padrão *IEC 80000-13* utilizam valores que são potências de 1024. Já as unidades baseadas no *SI* utilizam valores que são potências de 1000. Veja a Tabela 1.3.

| IEC | | | SI | | |
|----------|---------|----------|-----------|---------|----------|
| Nome | Símbolo | Bytes | Nome | Símbolo | Bytes |
| byte | B | 1024^0 | byte | B | 1000^0 |
| kibibyte | KiB | 1024^1 | kilobyte | kB | 1000^1 |
| mebibyte | MiB | 1024^2 | megabyte | MB | 1000^2 |
| gibibyte | GiB | 1024^3 | gigabyte | GB | 1000^3 |
| tebibyte | TiB | 1024^4 | terabyte | TB | 1000^4 |
| pebibyte | PiB | 1024^5 | petabyte | PB | 1000^5 |
| exbibyte | EiB | 1024^6 | exabyte | EB | 1000^6 |
| zebibyte | ZiB | 1024^7 | zettabyte | ZB | 1000^7 |
| yobibyte | YiB | 1024^8 | yottabyte | YB | 1000^8 |

Tabela 1.3: Unidades de medida segundo o padrão IEC 80000-13 e o Sistema Internacional de Unidades

Contudo, não há um consenso na indústria de hardware e software quanto à utilização desses padrões. Muitos sistemas operacionais informam a capacidade dos dispositivos de armazenamento misturando o padrão *SI* e o padrão *IEC 80000-13*. Eles costumam utilizar os símbolos do padrão *SI* com os valores do padrão *IEC 80000-13*. Já os fabricantes dos dispositivos de armazenamento preferem utilizar plenamente as unidades do *SI*.

| Unidades não Padronizadas | | |
|---------------------------|---------|----------|
| Nome | Símbolo | Bytes |
| byte | B | 1024^0 |
| kilobyte | kB | 1024^1 |
| megabyte | MB | 1024^2 |
| gigabyte | GB | 1024^3 |
| terabyte | TB | 1024^4 |
| petabyte | PB | 1024^5 |
| exabyte | EB | 1024^6 |
| zettabyte | ZB | 1024^7 |
| yottabyte | YB | 1024^8 |

Tabela 1.4: Unidades não padronizadas adotadas em muitos sistemas operacionais

Para exemplificar a confusão gerada pela utilização de unidades não padronizadas, considere o disco rígido ST9750420AS fabricado pela *Seagate Technology*.

A capacidade real desse *HD* é 750156374016 bytes. Utilizando o padrão *IEC 80000-13*, podemos dizer que esse disco rígido tem aproximadamente 698,63 GiB. Por outro lado, utilizando o padrão *SI*, podemos dizer que esse disco rígido tem aproximadamente 750,15 GB.

A *Seagate Technology*, assim como as outras fabricantes de discos rígidos, preferem arredondar para baixo a capacidade real dos *HDs* ao anunciar esses produtos aos clientes. Sendo assim, a *Seagate Technology* anuncia que a capacidade do *HD ST9750420AS* é 750 GB. Portanto, a capacidade real é um pouco maior do que a capacidade anunciada. Provavelmente, os compradores não se sentirão prejudicados já que a capacidade real é maior do que a anunciada.

Agora vem a confusão. Sistemas operacionais como *Windows 8* e *OS X* anteriores à versão 10.6 utilizam unidades de medida não padronizadas e informam aos usuários que o *HD ST9750420AS* de

750 GB possui capacidade igual a 698,63 GB. Os usuários que não sabem que esses sistemas operacionais adotam unidades de medida diferentes das adotadas pelos fabricantes de *HD* se sentem enganados.

1.5 Arquiteturas de Processadores



Os processadores só entendem operações definidas em formato binário. Para ilustrar, considere as operações apresentadas na Figura 1.7. Essas operações são fictícias.

| | | | |
|-------|-------|-------------|-------|
| GRAVA | REG-1 | 19 | |
| 0 0 1 | 0 0 1 | 0 1 0 0 1 1 | |
| GRAVA | REG-2 | 11 | |
| 0 0 1 | 0 1 0 | 0 0 1 0 1 1 | |
| SOMA | REG-1 | REG-2 | REG-3 |
| 0 1 0 | 0 0 1 | 0 1 0 | 0 1 1 |
| EXIBE | REG-3 | | |
| 0 1 1 | 0 1 1 | 0 0 0 0 0 0 | |

Figura 1.7: Instruções de processador

Nesse exemplo fictício, os três primeiros bits das instruções definem qual operação o processador deve executar. A operação “GRAVA” é representada pelo código “001”, a operação “SOMA” é representada pelo código “010” e a operação “EXIBE” é representada pelo código “011”.

As operações do tipo “GRAVA” servem para armazenar um valor em um registrador. Por isso, é necessário indicar o valor e o número do registrador onde esse valor deve ser armazenado. Tanto o valor quanto o número do registrador são definidos em formato binário.

As operações do tipo “SOMA” servem para somar os valores armazenados em dois registradores e guardar o resultado em um terceiro registrador. Por isso, é necessário indicar o número de três registradores. Os valores armazenados nos dois primeiros registradores são adicionados e o resultado é armazenado no terceiro registrador.

As operações do tipo “EXIBE” servem para exibir na tela o valor armazenado em um registrador. Por isso, é necessário indicar o número de um registrador. O valor armazenado nesse registrador é exibido na tela.

A primeira instrução indica ao processador que o valor 19 deve ser gravado no *registrador 1*. A segunda instrução indica que o valor 11 deve ser armazenado no *registrador 2*. Já a terceira instrução determina a realização da adição dos valores anteriormente armazenados nos *registradores 1 e 2* além de indicar que o resultado deve ser armazenado no *registrador 3*. Por último, a quarta instrução determina ao processador que o valor do *registrador 3* deve ser exibido na tela.

Não há um padrão universal para o formato das instruções que os processadores podem executar. Consequentemente, as mesmas operações podem ser definidas de formas diferentes em dois processadores distintos. Considere o exemplo fictício a seguir com algumas instruções para dois

processadores de tipos diferentes.

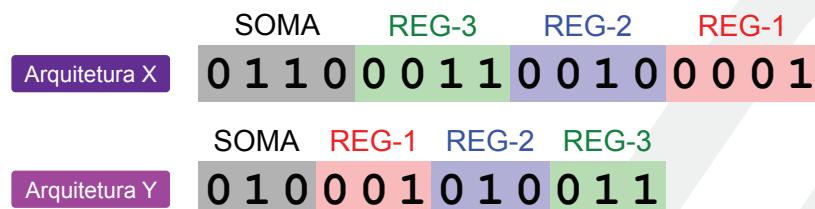


Figura 1.8: Instruções de processadores diferentes

As duas instruções mostradas na Figura 1.8 indicam aos processadores que o valor do *registrador 1* deve ser adicionado ao valor do *registrador 2* e o resultado deve ser armazenado no *registrador 3*. Contudo, as sequências binárias dessas instruções são diferentes porque os processadores são de arquiteturas diferentes.

As instruções que um processador pode executar são definidas pela sua arquitetura. As arquiteturas de processadores mais utilizadas atualmente são *x86*, *x86_64* e *ARM*.

1.6 O que é um Programa?



Os comandos que os processadores dos computadores oferecem são extremamente básicos. Normalmente, são comandos para realizar operações aritméticas como adição, subtração, divisão e multiplicação, bem como operações para armazenar ou recuperar dados do disco rígido, da memória *RAM*, dos registradores e de outros dispositivos de armazenamento. Geralmente, tarefas mais complexas são resolvidas através de sequências desses comandos básicos.

Por exemplo, para calcular a média aritmética dos números 10, 15 e 20, devemos obter o valor do somatório desses números através de operações básicas de adição. Depois, podemos utilizar uma operação básica de divisão para dividir o valor desse somatório por 3 e assim obter o valor 15, que é a média dos números.

Uma sequência de comandos básicos de processador que resolve uma determinada tarefa recebe o nome de **programa**. Os programas são armazenados em arquivos comumente chamados de **executáveis**.

Na prática, os programas são formados por instruções que dependem da arquitetura do processador. Consequentemente, um mesmo programa pode não funcionar em processadores de arquiteturas diferentes.



Analogia

Da mesma forma que pintores são capazes de produzir pinturas sofisticadas utilizando apenas tinta, pincel e quadro, programadores são capazes de criar programas sofisticados a partir dos recursos básicos oferecidos pelos computadores.



1.7 Linguagem de Máquina

Como vimos anteriormente, o formato dos comandos que um computador pode executar depende basicamente da arquitetura do seu processador. Como existem diversas arquiteturas diferentes, um mesmo comando pode funcionar em um computador e não funcionar em outro. O formato dos comandos aceitos por um determinado processador define a **linguagem de máquina ou código de máquina** desse processador.

Comandos definidos em linguagem de máquina são perfeitos para os computadores processarem. Por outro lado, eles são extremamente complexos para as pessoas entenderem. Podemos verificar essa complexidade observando o trecho do programa Chrome exibido na Figura 1.10.

Figura 1.10: Um trecho do programa Chrome

1.8 Linguagem de Programação

Como você já deve ter percebido, é extremamente complexo para uma pessoa escrever um programa diretamente em linguagem de máquina. Para tornar o desenvolvimento de programas uma tarefa viável, foram criadas as **linguagens de programação**. Essas linguagens permitem que pessoas possam criar programas de uma forma muito mais simples. Confira um trecho de código escrito com a linguagem de programação C#:

```
1 double soma = 0;
2 for(int i = 0; i < numeros.Length; i++)
3 {
4     soma += numeros[i];
5 }
6
7 System.Console.WriteLine("A soma é " + soma);
```

Código C# 1.1: Exemplo de código em C#

Por enquanto, você não precisa se preocupar em entender o que está escrito no código acima. Observe apenas que um programa escrito em linguagem de programação é bem mais fácil de ser compreendido do que um programa escrito em linguagem de máquina.

1.9 Compilador



Vimos que os computadores são capazes de processar o código escrito em linguagem de máquina. Também vimos que é inviável desenvolver um programa em linguagem de máquina. Por isso, existem as linguagens de programação. Daí surge uma pergunta: se os computadores entendem apenas comandos em linguagem de máquina, como eles podem executar código escrito em linguagem de programação?

Na verdade, os computadores não executam código escrito em linguagem de programação. Esse código que é denominado **código fonte** deve ser traduzido para código em linguagem de máquina. Essa tradução é realizada por programas especiais chamados **compiladores**.



Figura 1.11: Processo de compilação e execução de um programa

1.10 Sistemas Operacionais



Normalmente, os programas instalados em um computador são armazenados no disco rígido. Para executar um programa, as instruções que definem esse programa devem ser copiadas do disco rígido para a memória *RAM*. Essa cópia é realizada através dos comandos oferecidos pelo processador.

Geralmente, as pessoas não possuem conhecimentos técnicos suficientes para utilizar os comandos dos processadores. Dessa forma, elas não conseguiram copiar as instruções de um programa do disco rígido para a memória *RAM*. Consequentemente, elas não conseguiram executar programas através dos computadores.

Para facilitar a interação entre os usuários e os computadores, foram criados programas especiais denominados **sistemas operacionais**. Os sistemas operacionais funcionam como intermediários entre os usuários e os computadores. Os principais sistemas operacionais atuais oferecem uma interface visual. Através dessa interface os usuários podem controlar o funcionamento dos computadores.



Figura 1.12: Interação entre usuário e sistema operacional

Em sistemas operacionais como o *Windows*, geralmente, o usuário clica duas vezes em um ícone correspondente ao programa que ele deseja executar e o *Windows* se encarrega de copiar as instruções desse programa do disco rígido para a memória *RAM*. Dessa forma, o usuário não precisa conhecer os comandos dos processadores para executar um programa.

Os sistemas operacionais controlam a execução dos programas. Inclusive, eles permitem que vários programas sejam executados simultaneamente. Além disso, oferecem diversas funcionalidades aos usuários, como controlar o volume das caixas de som, o brilho do monitor, o acesso à internet entre outros.

Os sistemas operacionais também oferecem diversos serviços aos próprios programas. Por exemplo, as impressoras configuradas pelos usuários são gerenciadas pelos sistemas operacionais. Qualquer programa que deseja interagir com uma impressora pode utilizar os recursos oferecidos pelos sistemas operacionais para esse propósito. Sendo assim, os sistemas operacionais funcionam como intermediários entre os programas e os computadores.



Figura 1.13: Interação entre programa e sistema operacional

As principais tarefas de um sistema operacional são:

- Gerenciar a execução dos programas.
- Controlar o acesso à memória *RAM* e ao disco rígido.
- Administrar os dispositivos conectados ao computador.
- Simplificar a interação entre os programas e o computador.
- Simplificar a interação entre o usuário e o computador.

Sistemas operacionais diferentes podem oferecer recursos diferentes para os programas. No processo de compilação, geralmente, os programas são preparados para utilizar os recursos de um determinado sistema operacional. Dessa forma, um programa que funciona em um determinado sistema operacional pode não funcionar em outro sistema operacional.



1.11 Máquinas Virtuais

Como vimos anteriormente, o código fonte de um programa deve ser compilado para que esse programa possa ser executado por um computador. Além disso, vimos que os compiladores geram executáveis específicos para um determinado sistema operacional e uma determinada arquitetura de processador. Qual é o impacto disso para quem desenvolve sistemas para múltiplas plataformas?

A empresa que deseja ter uma aplicação disponível para diversos sistemas operacionais (*Windows*, *Linux*, *OS X*, etc) e diversas arquiteturas de processadores (*x86*, *x86_64*, *ARM*, etc) deverá desenvolver e manter um programa para cada plataforma (a combinação de um sistema operacional e uma arquitetura de processador). Consequentemente, os custos dessa empresa seriam muito altos.

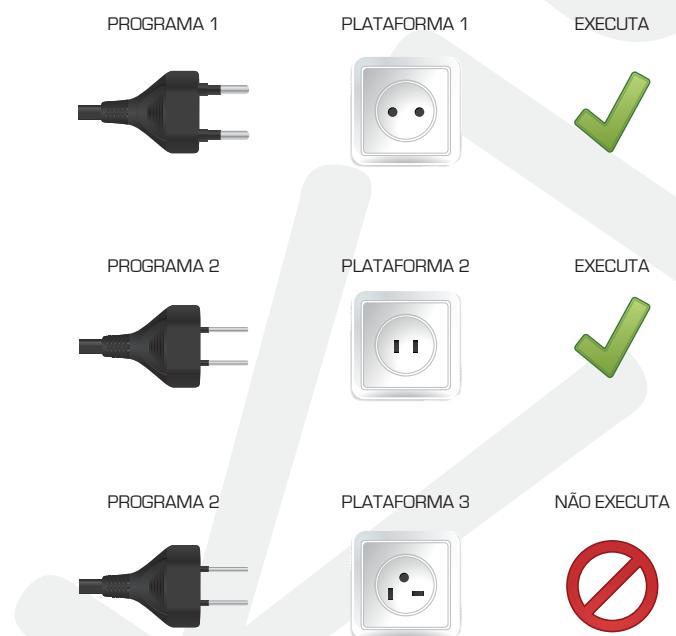


Figura 1.14: Ilustração mostrando que cada plataforma necessita de um executável específico

Para diminuir os custos e aumentar a produtividade, podemos utilizar as chamadas **máquinas virtuais**. As máquinas virtuais são programas especiais que permitem que um programa possa ser executado em diversas plataformas diferentes. Nesse cenário, o desenvolvimento e a execução de um programa são realizados através dos seguintes passos:

1. O programador escreve o código fonte do programa utilizando uma linguagem de programação.
2. O compilador traduz o código fonte para um código intermediário escrito em uma linguagem que a máquina virtual entende.
3. A máquina virtual processa o código intermediário e o traduz para código de máquina.
4. O código de máquina é executado no computador.

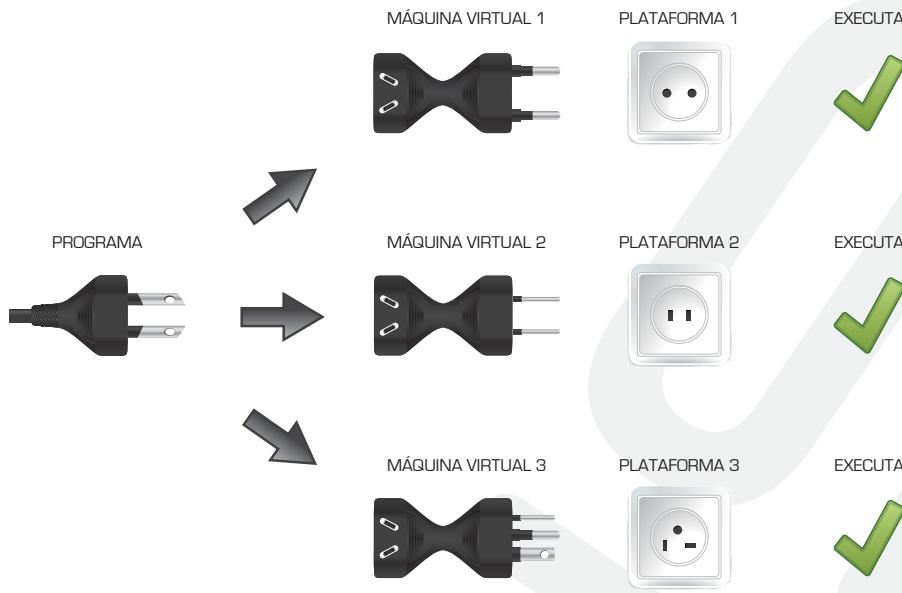


Figura 1.15: Ilustração do funcionamento da máquina virtual

Na analogia da Figura 1.15, o programa seria o plugue, as máquinas virtuais seriam os adaptadores e as plataformas seriam as tomadas. Como as tomadas seguem padrões diferentes, são necessários adaptadores para encaixar o mesmo plugue em todas as tomadas. Analogamente, como as plataformas seguem padrões diferentes, são necessárias máquinas virtuais para executar o mesmo programa em todas as plataformas.

Para cada plataforma, uma máquina virtual específica é necessária. Todas as máquinas virtuais devem saber ler as instruções do programa que desejamos executar para depois traduzi-las para a linguagem de máquina da plataforma correspondente. Dessa forma, as máquinas virtuais atuam como tradutores.

Assim como qualquer coisa, alguém precisa desenvolver as máquinas virtuais. Qualquer pessoa pode desenvolver uma máquina virtual. Contudo, essa é uma tarefa que exige conhecimento técnico muito avançado. Por isso, normalmente, os programadores que desenvolvem os programas não desenvolvem as máquinas virtuais. Geralmente, as máquinas virtuais são desenvolvidas por grandes empresas ou são projetos de código aberto que envolvem programadores experientes do mundo inteiro.

As máquinas virtuais podem ser gratuitas ou pagas. Os maiores exemplos de máquinas virtuais são a *Oracle JVM (Java Virtual Machine)*, a *OpenJDK JVM*, a *Microsoft CLR (Common Language Runtime)* e a *Mono CLR*.

A nossa explicação sobre máquinas virtuais pode dar a entender que elas funcionam apenas como meros tradutores ou adaptadores. Contudo, é importante destacar que as máquinas virtuais oferecem diversos outros recursos como gerenciamento de memória e otimização em tempo de execução.

1.12 Editores de Texto



O código fonte de um programa C# pode ser criado através de editores de texto simples. No *Windows*, recomendamos a utilização do *Notepad* ou do *Notepad++*. No *Linux*, recomendamos a utilização do *gedit*. No *OS X*, recomendamos a utilização do *TextWrangler*. Esses editores são todos gratuitos.

1.13 Terminal



A maior parte dos usuários dos computadores não possui conhecimento sobre programação. Esse usuários interagem com os computadores através das interfaces visuais oferecidas pelos sistemas operacionais. Geralmente, essas interfaces visuais não exigem conhecimentos técnicos.

Os sistemas operacionais oferecem também interfaces baseadas em texto. Essas interfaces não são muito agradáveis para a maior parte dos usuários. Porém, geralmente, elas são mais práticas para os programadores.

No *Windows*, o programa *Command Prompt* e o programa *Windows Power Shell* permitem que os usuários controlem o computador através de uma interface baseada em texto. Nos sistemas operacionais da família *Unix*, há diversos programas que oferecem esse tipo de interface. Geralmente, esses programas são chamados de **Terminal**.

1.14 Hello World em C#



Vamos criar o nosso primeiro programa para entendermos como funciona o processo de codificação, compilação e execução de um programa em C#.



Importante

No sistema operacional *Windows*, para compilar e executar um programa escrito em C#, é necessário ter instalado o *.NET Framework* ou uma implementação alternativa da plataforma *.NET*. A partir do *Windows Vista*, o sistema operacional *Windows* já vem com *.NET Framework* instalado.

Um programa escrito em C# também pode ser compilado e executado em outros sistemas operacionais como o *Ubuntu* e o *OS X* através da plataforma *Mono* (<http://www.mono-project.com>).

O primeiro passo é escrever o código fonte do programa. Qualquer editor de texto pode ser utilizado. No exemplo abaixo, o código fonte foi armazenado no arquivo `HelloWorld.cs`. O nome do arquivo não precisa ser `HelloWorld` nem a extensão precisa ser `.cs`. Contudo, recomendamos que essa extensão seja sempre utilizada nos arquivos que armazenam código fonte C#. Considere que o arquivo `HelloWorld.cs` foi salvo na pasta `introducao`.

```
1 class HelloWorld
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Hello World");
```

```
6 }
7 }
```

Código C# 1.2: HelloWorld.cs

Por enquanto, não se preocupe em entender perfeitamente o código do arquivo `HelloWorld.cs`. Apenas observe que, na primeira linha, foi definida uma classe chamada `HelloWorld`. O corpo dessa classe é delimitado pela chave de abertura da segunda linha e a chave de fechamento da última linha. No corpo da classe `HelloWorld`, o método `Main` foi definido. O corpo desse método é delimitado pela chave de abertura da quarta linha e a chave de fechamento da sexta linha. No corpo do método `Main`, a instrução `System.Console.WriteLine("HelloWorld")` indica que a mensagem "HelloWorld" deve ser exibida na tela. Essa instrução deve terminar com ponto e vírgula.

O próximo passo é abrir um terminal, entrar na pasta `introducao` e depois compilar o arquivo `HelloWorld.cs`.

```
C:\Users\K19> cd introducao
C:\Users\K19\introducao> csc HelloWorld.cs
```

Terminal 1.1: Compilando o arquivo `HelloWorld.cs` no Windows

O comando `cd` altera o diretório atual do terminal. No exemplo acima, esse comando foi utilizado para alterar o diretório atual do terminal para `introducao`.

O comando `csc` aciona o compilador da linguagem C#. No exemplo acima, esse comando foi utilizado para compilar o arquivo `HelloWorld.cs`. O compilador traduz o código fonte desse arquivo e armazena o código compilado em um novo arquivo chamado `HelloWorld.exe`.

Para conferir se o arquivo `HelloWorld.exe` foi criado, podemos utilizar o comando `dir` no *Command Prompt* do Windows. Esse comando lista o conteúdo da pasta atual do terminal.

```
C:\Users\K19\introducao> dir
0 volume na unidade C não tem nome
O Número de Serie do Volume é 063B-6F3D

Pasta de C:\Users\K19\introducao

14-07-2009 06:41    <DIR>      .
14-07-2009 06:41    <DIR>      ..
14-07-2009 06:52        106 HelloWorld.cs
14-07-2009 06:52        425 HelloWorld.exe
              2 File(s)   531 bytes
              2 Dir(s)  57,925,980,160 bytes disponíveis
```

Terminal 1.2: Listando o diretório atual do terminal em ambiente Windows

Agora, podemos executar o arquivo `HelloWorld.exe`.

```
C:\Users\K19\introducao> HelloWorld.exe
Hello World
```

Terminal 1.3: Executando o programa em ambiente Windows



Mais Sobre

A maioria das linguagens de programação são **case sensitive**. Isso significa que elas diferenciam as letras maiúsculas das minúsculas. Portanto, ao escrever o código de um programa, devemos tomar cuidado para não trocar uma letra maiúscula por uma letra minúscula

ou vice-versa.

1.15 O que é o Método Main?



Um programa é basicamente uma sequência de instruções. As instruções de um programa escrito em C# devem ser definidas dentro do método `Main`.

```
1 class Programa
2 {
3     static void Main()
4     {
5         PRIMEIRA INSTRUÇÃO
6         SEGUNDA INSTRUÇÃO
7         TERCEIRA INSTRUÇÃO
8         ...
9     }
10 }
```

Código C# 1.3: Método Main

O método `Main` é o começo do programa. Podemos dizer que o “ponto de partida” de um programa em C# é a primeira instrução do método `Main`. As demais instruções são executadas na ordem em que estão definidas no código. Eventualmente, durante a execução das instruções, algum erro pode ocorrer e interromper o fluxo do processamento.

De acordo com a especificação da linguagem C#, o método `Main` deve ser definido com o modificador `static`. Opcionalmente, ele pode ser definido com o modificador `public` e receber um array de strings como parâmetro. O tipo de retorno pode ser `void` ou `int`.



Lembre-se

O funcionamento dos arrays será abordado no Capítulo ???. Os conceitos de método, parâmetro e tipo de retorno serão apresentados no Capítulo ???.



Lembre-se

Neste livro, não serão abordados o conceito de classe e os modificadores `public` e `static`.

Simulação



Veremos, a seguir, uma simulação de execução de um programa em C#.

- 1 A execução é iniciada na primeira linha do método `Main`. Ou seja, ela começa na linha 5 do código abaixo. A instrução presente nessa linha exibe o caractere “A” na saída padrão.

```

1 class ExibeMensagens
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("A");
6         System.Console.WriteLine("B");
7         System.Console.WriteLine("C");
8     }
9 }
```

● A

- 2 Em seguida, a linha 6 é executada e o caractere “B” é exibido na saída padrão.

```

1 class ExibeMensagens
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("A");
6         System.Console.WriteLine("B");
7         System.Console.WriteLine("C");
8     }
9 }
```

● A

● B

- 3 Seguindo o fluxo de execução, a linha 7 é executada e o caractere “C” é exibido na saída padrão.

```

1 class ExibeMensagens
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("A");
6         System.Console.WriteLine("B");
7         System.Console.WriteLine("C");
8     }
9 }
```

● A

● B

● C



1.16 Variações do Método Main

A forma tradicional do método `Main` é mostrada no código a seguir.

```
1 static void Main()
```

Código C# 1.7: Forma tradicional do método Main

Algumas variações da forma tradicional do método `Main` são aceitas. Por exemplo, acrescentar o modificador `public`.

```
1 public static void Main()
```

Código C# 1.8: Acrescentando o modificador public

Acrescentar um array de strings como parâmetro.

```
1 static void Main(string[] args)
```

Código C# 1.9: Acrescentar um array de strings como parâmetro

Os colchetes podem ser definidos à direita do nome do parâmetro.

```
1 static void Main(string args[])
```

Código C# 1.10: Alterando a posição dos colchetes



1.17 Argumentos de Linha de Comando

Considere um programa que calcula a média das notas dos alunos de uma escola. Esse programa poderia receber as notas de um determinado aluno, calcular a média e depois exibi-la na tela.

Ao executar um programa em C#, podemos passar argumentos na linha de comando para o método `Main`. Por exemplo, suponha que o código compilado desse programa tenha sido armazenado em um arquivo chamado `Programa.exe`. No exemplo abaixo, esse programa foi executado sem nenhum argumento de linha de comando.

```
C:\Users\K19\introducao> Programa.exe
```

Terminal 1.7: Executando o programa sem argumentos de linha de comando

Agora, no próximo exemplo, o programa foi executado com três argumentos de linha de comando: K19, Livros e Lógica.

```
C:\Users\K19\introducao> Programa.exe K19 Livros Lógica
```

Terminal 1.8: Executando o programa com três argumentos de linha de comando

A cada execução do programa, argumentos diferentes podem ser passados na linha de comando. Esses argumentos podem ser recuperados dentro do método `Main`. No exemplo abaixo, criamos um programa que exibe na saída padrão o primeiro, o segundo e o terceiro argumentos da linha de comando. O primeiro argumento é acessado através da variável `args[0]`. O segundo argumento é acessado através da variável `args[1]`. O terceiro argumento é acessado através da variável `args[2]`.

```
1 class Programa
2 {
3     static void Main()
4     {
5         System.Console.WriteLine(args[0]);
6         System.Console.WriteLine(args[1]);
7         System.Console.WriteLine(args[2]);
8     }
9 }
```

Código C# 1.11: Recuperando os argumentos de linha de comando

A seguir apresentamos a saída do programa ao executá-lo com três argumentos de linha de comando.

```
C:\Users\K19\introducao> Programa.exe K19 Livros Lógica
K19
Livros
Lógica
```

Terminal 1.9: Saída do programa ao executá-lo com três argumentos de linha de comando



Mais Sobre

O funcionamento dos arrays será abordado no Capítulo ??.

Você não pode esquecer que, ao executar um programa, os argumentos de linha de comando são separados por espaço e o primeiro parâmetro vem imediatamente à direita do nome do arquivo que contém o código compilado. O índice do primeiro argumento é 0, do segundo é 1, do terceiro é 2 e assim sucessivamente.

```
C:\Users\K19\introducao> Programa.exe arg0 arg1 arg2
```

Terminal 1.10: Argumentos de linha de comando

Para definir um argumento de linha de comando que contém um ou mais espaços, o caractere aspas dupla deve ser utilizado para delimitar esse argumento.

```
C:\Users\K19\introducao> Programa.exe "Rafael Cosentino" "Rafael Lobato" "Marcelo Martins"
```

Terminal 1.11: Argumentos de linha de comando



Simulação

Veremos, a seguir, uma simulação de execução de um programa em C# que exibe os três primeiros argumentos da linha de comando na saída padrão.

- Suponha que o programa armazenado no arquivo `Programa.exe` seja executado com os argumentos “K19”, “Livros” e “Lógica”, como no exemplo abaixo.

```
Programa.exe Argumentos K19 Livros Lógica
```

Como sabemos, o fluxo de execução do programa inicia na primeira linha do método `Main`. Ou seja, ele começa na linha 5 do código abaixo. A instrução presente nessa linha exibe o primeiro argumento da linha de comando, isto é, exibe “K19” na saída padrão.

```
1 class Argumentos
2 {
3     static void Main()
4     {
5         System.Console.WriteLine(args[0]);
6         System.Console.WriteLine(args[1]);
7         System.Console.WriteLine(args[2]);
8     }
9 }
```

● K19

- 2 Em seguida, a linha 6 é executada e o segundo argumento da linha de comando é exibido. Sendo assim, a palavra “Livros” é exibida na saída padrão.

```
1 class Argumentos
2 {
3     static void Main()
4     {
5         System.Console.WriteLine(args[0]);
6         System.Console.WriteLine(args[1]); System.Console.WriteLine(args[1]);
7         System.Console.WriteLine(args[2]);
8     }
9 }
```

K19
● Livros

- 3 Seguindo o fluxo de execução, a linha 7 é executada e o terceiro argumento da linha de comando é exibido na saída padrão, isto é, a palavra “Lógica” é exibida.

```
1 class Argumentos
2 {
3     static void Main()
4     {
5         System.Console.WriteLine(args[0]);
6         System.Console.WriteLine(args[1]);
7         System.Console.WriteLine(args[2]); System.Console.WriteLine(args[2]);
8     }
9 }
```

K19
● Livros
● Lógica

1.18 Exibindo Mensagens



Geralmente, as linguagens de programação possuem comandos para exibir mensagens na saída padrão (tela do terminal). Nos programas em C#, podemos utilizar o seguinte trecho de código para exibir uma mensagem na saída padrão.

```
1 System.Console.WriteLine("MENSAGEM");
```

Código C# 1.15: Exibindo uma mensagem na saída padrão

Para adicionar quebras de linha ou tabulações nas mensagens, é necessário utilizar as chamadas “sequências de escape”. Uma quebra de linha é definida com a sequência de escape “\n”. Uma tabulação, com “\t”.

Na Tabela 1.5, as sequências de escape da linguagem C# são apresentadas.

| Sequência de escape | Descrição |
|---------------------|---|
| \t | Adiciona uma tabulação (<i>tab</i>) |
| \v | Adiciona uma tabulação vertical |
| \b | Volta para o caractere anterior (<i>backspace</i>) |
| \n | Adiciona uma quebra de linha (<i>newline</i>) |
| \r | Volta para o início da linha (<i>carriage return</i>) |
| \f | Adiciona uma quebra de página (<i>formfeed</i>) |
| \' | Adiciona o caractere aspas simples |
| \" | Adiciona o caractere aspas dupla |
| \\\ | Adiciona uma barra invertida |

Tabela 1.5: Sequências de escape

```

1 System.Console.WriteLine("\tRafael");
2 System.Console.WriteLine("Linha1\nLinha2");
3 System.Console.WriteLine("Digite \'sim\'");
4 System.Console.WriteLine("Jonas disse: \"Olá\"");
5 System.Console.WriteLine("C:\\\\K19\\\\Livros");

```

Código C# 1.16: Exemplos de uso das sequências de escape

```

Rafael
Linha1
Linha2
Digite 'sim'
Jonas disse: "Olá"
C:\\K19\\Livros

```

Terminal 1.16: Exemplos de uso das sequências de escape

O método `WriteLine` adiciona uma quebra de linha no final da mensagem exibida. Para exibir mensagens sem quebra de linha, podemos utilizar o método `Write`.

```
1 System.Console.Write("MENSAGEM SEM QUEBRA DE LINHA");
```



1.19 Comentários

Podemos acrescentar comentários em qualquer ponto do código fonte. Geralmente, os comentários funcionam como anotações que o programador adiciona no código fonte para explicar a lógica do programa. Eles são úteis tanto para o próprio programador que os escreveu quanto para outros programadores que, eventualmente, precisam ler e/ou alterar o código fonte.

Os compiladores ignoram os comentários inseridos no código fonte. Portanto, no código de máquina gerado pela compilação do código fonte, os comentários não são inseridos.

Em C#, para comentar uma linha, podemos utilizar a marcação `//`.

```

1 System.Console.WriteLine("K19");
2 // comentário de linha
3 System.Console.WriteLine("Rafael Cosentino");

```

Código C# 1.18: Comentário de linha

Também é possível comentar um bloco com os marcadores `/*` e `*/`.

```
1 System.Console.WriteLine("K19");
2 /* comentário de bloco
3 todo esse trecho
4 está comentado */
5 System.Console.WriteLine("Rafael Cosentino");
```

1.20 Indentação



A organização do código fonte é fundamental para o entendimento da lógica de um programa. Cada linguagem de programação possui os seus próprios padrões de organização. Observe a organização padrão do código fonte escrito com a linguagem de programação C#.

```
1 class Programa
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("K19");
6         System.Console.WriteLine("Lógica de Programação em C#");
7     }
8 }
```

Código C# 1.20: Programa.cs

Para visualizar facilmente a organização dos blocos (trechos delimitados por chaves), o código fonte deve ser indentado. A indentação consiste em deixar uma certa quantidade de espaços em branco ou tabulações no começo de cada linha.

No exemplo acima, as linhas 5 e 6 estão dentro do corpo do método `Main` que por sua vez está dentro do corpo da classe `Programa`. Por isso, essas duas linhas foram indentadas com duas tabulações. As linhas 3, 4 e 7 estão dentro do corpo da classe `Programa` mas não estão dentro do corpo do método `Main`. Por isso, essas três linhas foram indentadas com apenas uma tabulação.

Um código corretamente indentado é mais fácil de ler. Consequentemente, a sua manutenção se torna mais simples.



Mais Sobre

Você pode verificar a convenção de indentação da linguagem C# definida pela Microsoft no seguinte endereço:

<http://msdn.microsoft.com/en-us/library/ff926074.aspx>

1.21 Engenharia Reversa



Talvez você já tenha desmontado um brinquedo ou algum aparelho eletrônico para tentar descobrir como ele funciona. Ao fazer isso, mesmo sem saber, você praticou engenharia reversa.

Muitas empresas praticam engenharia reversa para entender o funcionamento dos produtos dos concorrentes. Países também utilizam esse tipo de abordagem para avaliar a capacidade militar dos outros países.

A engenharia reversa também é aplicada na área de software. As instruções do código de máquina de um programa podem ser traduzidas para alguma linguagem de programação através de programas especiais que são chamados de **decompiladores**.

Normalmente, o código em linguagem de programação gerado a partir da decompilação do código de máquina de um programa não é fácil de entender. Geralmente, é possível, apesar de normalmente ser muito difícil, modificar o funcionamento de um programa para qualquer que seja o propósito utilizando a abordagem da engenharia reversa.

1.22 Ofuscadores



Para dificultar o processo de engenharia reversa, podemos utilizar ferramentas que modificam o código fonte ou o código compilado com o intuito de prejudicar o processo de decompilação. Essas ferramentas são chamadas de **ofuscadores**.

Na maior parte dos casos, a utilização de ofuscadores torna inviável ou muito custosa a aplicação de engenharia reversa com intuito de “copiar” ou “piratear” um software.

1.23 Erro: Compilar um arquivo inexistente



Um erro de compilação comum em C# é compilar um arquivo inexistente. Normalmente, esse erro ocorre porque o arquivo foi salvo em outra pasta ou com um nome diferente.

No exemplo abaixo, o nome do arquivo que deveria ser compilado é `Program.cs`. Contudo, na compilação, esquecemos da letra “a” e solicitamos a compilação de um arquivo chamado `Program.cs`. Como esse arquivo não existe, um erro de compilação é gerado.

```
C:\Users\K19\introducao> csc Program.cs
error CS2001: O arquivo de origem 'Program.cs' não pôde ser encontrado
warning CS2008: Nenhum arquivo de origem especificado
```

Terminal 1.17: Erro de compilação

1.24 Erro: Executar utilizando a extensão .cs



Um erro comum em C# é tentar executar um programa utilizando a extensão `.cs`. Observe, no exemplo a seguir, esse erro ocorrendo.

```
C:\Users\K19\introducao> Program.cs
'Program.cs' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.
```

Terminal 1.18: Erro de tentativa de execução

1.25 Erro: Não fechar os blocos



Um erro de compilação comum em C# é esquecer de fechar os blocos com chave. Observe, nos exemplos a seguir, esse erro ocorrendo.

```
1 class Programa
2 {
3     static void Main()
4     {
5         // instruções
6 }
```

Código C# 1.21: Programa.cs

```
1 class Programa
2 {
3     static void Main()
4     {
5         // instruções
```

Código C# 1.22: Programa.cs

A mensagem que o compilador do C# exibe é semelhante à apresentada abaixo.

```
Programa.cs(6,2): error CS1513: } esperada
```

Terminal 1.19: Erro de compilação

1.26 Erro: Não fechar as aspas

Um erro de compilação comum em C# é esquecer de fechar as aspas. No exemplo a seguir, falta uma aspas dupla na linha 5.

```
1 class Programa
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("K19";
6     }
7 }
```

Código C# 1.23: Programa.cs

Veja a mensagem que o compilador do C# exibe quando as aspas não são fechadas corretamente.

```
Programa.cs(5,28): error CS1010: Newline em constante
```

Terminal 1.20: Erro de compilação

1.27 Erro: Trocar maiúsculas e minúsculas

Um erro de compilação comum em C# é utilizar letras maiúsculas onde deveriam ser utilizadas letras minúsculas ou vice-versa. No exemplo a seguir, o identificador `System` foi escrito com “s”, porém o correto é com “S”.

```
1 class HelloWorld
2 {
3     static void Main()
4     {
5         system.Console.WriteLine("Hello World");
```

```
6 }
7 }
```

Código C# 1.24: HelloWorld.cs

Veja a mensagem de erro do compilador C#.

```
HelloWorld.cs(5,3): error CS0103: O nome 'system' não existe no contexto atual
```

Terminal 1.21: Erro de compilação

1.28 Erro: Esquecer o ponto e vírgula



Para encerrar uma instrução, devemos utilizar o caractere “;”. Não inserir esse caractere no final das instruções gera erro de compilação. No exemplo abaixo, falta um ponto e vírgula no final da linha 5.

```
1 class HelloWorld
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Hello World")
6     }
7 }
```

Código C# 1.25: HelloWorld.cs

Veja a mensagem de erro do compilador C#.

```
HelloWorld.cs(5,42): error CS1002: ; esperado
```

Terminal 1.22: Erro de compilação

1.29 Erro: Esquecer o Main



Todo programa deve ter um “ponto de partida”. Em C#, todo programa precisa do método `Main`. Se você esquecer de definir o método `Main`, obterá um erro de compilação.

```
1 class HelloWorld
2 {
3     static void main()
4     {
5         System.Console.WriteLine("Hello World");
6     }
7 }
```

Código C# 1.26: HelloWorld.cs

Observe que no código C# acima, o método `Main` foi definido com letra minúscula. Contudo, no C#, o correto é com maiúscula. Ao compilar o código, o seguinte erro é exibido.

```
error CS5001: O programa 'c:\Users\K19\introducaoHelloWorld.exe' não contém um método 'Main'
static adequado para um ponto de entrada
```

Terminal 1.23: Erro de compilação

1.30 Erro: Utilizar sequências de escape inválidas



Utilizar sequências de escape inválidas gera erro de compilação. No exemplo abaixo, a sequência “\x” foi utilizada. Contudo, ela não é uma sequência de escape válida.

```
1 class HelloWorld
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Hello\xWorld");
6     }
7 }
```

Código C# 1.27: HelloWorld.cs

Veja a mensagem do compilador C#.

```
HelloWorld.cs(5,34): error CS1009: Seqüência de escape não reconhecida
```

Terminal 1.24: Erro de compilação

1.31 Resumo



- 1 Os principais elementos de um computador são: *CPU*, memória *RAM* e *HD*.
- 2 A *CPU* executa as instruções dos programas.
- 3 Geralmente, os arquivos dos usuários e dados dos programas instalados no computador são armazenados no *HD*.
- 4 Geralmente, quando o usuário executa um programa, os dados desse programa são copiados do *HD* para memória *RAM*.
- 5 Os dados armazenados na memória *RAM* são descartados quando o computador é desligado.
- 6 Os dados armazenados no *HD* não são descartados quando o computador é desligado.
- 7 Os computadores se comunicam com outros computadores ou com dispositivos periféricos através de portas como *ethernet*, *paralela*, *USB* e *HDMI*.
- 8 Os dados manipulados por um computador são definidos em formato binário.

- 9 ► Os principais padrões de codificação de caracteres são *ASCII* e *Unicode*.
- 10 ► Algumas unidades de medida para dados binários do padrão *IEC 80000-13* são: *bit*, *byte (B)*, *kibibyte (KiB)*, *mebibyte (MiB)*, *gibibyte (GiB)* e *tebibyte (TiB)*.
- 11 ► Algumas unidades de medida para dados binários do *SI* são: *bit*, *byte (B)*, *kilobyte (kB)*, *megabyte (MB)*, *gigabyte (GB)* e *terabyte (TB)*.
- 12 ► Os comandos que um processador pode executar são definidos pela sua arquitetura.
- 13 ► Processadores de arquiteturas diferentes entendem comandos diferentes.
- 14 ► Atualmente, as arquiteturas de processador mais utilizadas são: *x86*, *x86_64* e *ARM*.
- 15 ► Um programa é uma sequência de instruções que resolve uma determinada tarefa.
- 16 ► As linguagens de programação são mais fáceis para pessoas entenderem do que as linguagens de máquina.
- 17 ► Os programas são definidos em linguagem de programação.
- 18 ► Os compiladores traduzem o código fonte de um programa para código de máquina.
- 19 ► Os sistemas operacionais gerenciam a execução dos programas; controlam o acesso à memória *RAM* e ao disco rígido; administram os dispositivos conectados ao computador; simplificam a interação entre os programas e o computador; e simplificam a interação entre o usuário e o computador.
- 20 ► As máquinas virtuais permitem a criação de programas portáveis.
- 21 ► Todo programa necessita de um “ponto de partida”. O ponto de partida dos programas escritos em C# é a primeira instrução do método *Main*.
- 22 ► Ao executar um programa em C#, podemos passar argumentos de linha de comando.
- 23 ► No código fonte de um programa em C#, comentários são inseridos com os marcadores “//”, “/*” e “*/”.

- 24 ► A maioria das linguagens de programação são case sensitive.
- 25 ► A indentação melhora a legibilidade do código fonte.
- 26 ► Código escrito em linguagem C# geralmente é armazenado em arquivos com a extensão “.cs”.
- 27 ► Em C#, o método `WriteLine` é utilizado para exibir mensagens com quebra de linha na saída padrão.
- 28 ► Em C#, o método `Write` é utilizado para exibir mensagens sem quebra de linha na saída padrão.

VARIÁVEIS

2.1 O que é uma Variável?



Considere um programa que calcula a média das notas dos alunos de uma escola. Para calcular essa média, o programa precisa realizar operações aritméticas com os valores das notas dos alunos. Para isso, esses valores devem ser armazenados em **variáveis**.

As variáveis são utilizadas para armazenar os dados que um programa deve manipular. Toda variável possui um nome (identificador). Para acessar ou alterar o conteúdo de uma variável, é necessário utilizar o nome dessa variável.

Em C#, toda variável possui um tipo. O tipo de uma variável determina o que pode ou não ser armazenado nela. Por exemplo, podemos determinar que uma variável só pode armazenar números inteiros.

Geralmente, toda variável está associada a uma posição da memória *RAM*. Portanto, quando armazenamos um valor em uma variável, na verdade, estamos armazenando esse valor em algum lugar da memória *RAM*. Dessa forma, os identificadores das variáveis são utilizados para acessar a memória *RAM*.

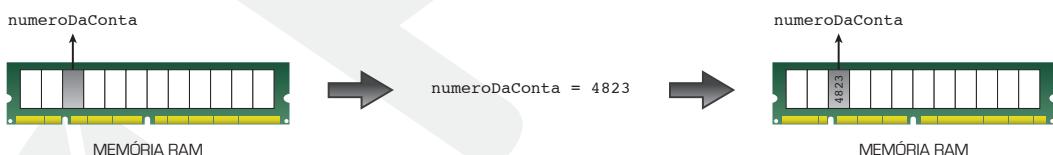


Figura 2.1: Processo de atribuição do valor numérico 4823 à variável numeroDaConta

A Figura 2.1 ilustra o processo de armazenamento do valor 4823 na variável numeroDaConta. Essa variável está associada à terceira posição da memória *RAM*. Lembre-se que esse valor é armazenado em formato binário.

2.2 Declarando e Inicializando Variáveis



Toda variável deve ser declarada antes de ser utilizada. Na declaração, é obrigatório determinar o nome da variável. Por outro lado, o tipo da variável pode ser definido explícita ou implicitamente. No exemplo abaixo, três variáveis foram declaradas: a variável chamada numeroDaConta do tipo `int`,

a variável `saldo` do tipo `double` e a variável `contaAtiva` do tipo `bool`. Os tipos dessas variáveis foram definidos explicitamente.

```
1 int numeroDaConta;
2 double saldo;
3 bool contaAtiva;
```

Código C# 2.1: Declaração de variáveis

Duas ou mais variáveis de um mesmo tipo podem ser declaradas na mesma instrução. Nesse caso, o tipo deve ser definido apenas uma vez. Os nomes das variáveis devem ser separados por vírgula. No exemplo abaixo, três variáveis do tipo `double` foram declaradas: `saldo`, `limite` e `taxa`.

```
1 double saldo, limite, taxa;
```

Código C# 2.2: Declaração de duas ou mais variáveis na mesma instrução

Toda variável deve ser inicializada antes de sua utilização. O processo de inicialização consiste em atribuir a uma variável o seu primeiro valor. No exemplo abaixo, a variável `numeroDaConta` foi declarada na linha 1 e inicializada com o valor 3466 na linha 2.

```
1 int numeroDaConta;
2 numeroDaConta = 3466;
```

Código C# 2.3: Declaração e inicialização de uma variável

Também é possível declarar e inicializar uma variável em apenas uma linha. No exemplo abaixo, a variável `numeroDaConta` foi declarada e inicializada com o valor 3466 na linha 1.

```
1 int numeroDaConta = 3466;
```

Código C# 2.4: Declaração e inicialização de uma variável

Quando uma variável local é declarada e inicializada em uma única instrução, o tipo dessa variável pode ser definido implicitamente. Para isso, basta utilizar a palavra chave `var`. No exemplo abaixo, o tipo da variável `numeroDaConta` foi definido implicitamente devido a utilização da palavra chave `var`. Como a variável `numeroDaConta` foi inicializada com um valor do tipo `int`, ela será do tipo `int`.

```
1 var numeroDaConta = 3466;
```

Código C# 2.5: Declaração e inicialização de uma variável com o comando `var`



Pare para pensar...

O que aconteceria se o compilador C# permitisse utilizar uma variável não inicializada?

Um programador da linguagem C (não C#), responderia essa pergunta facilmente, pois em C é possível utilizar uma variável sem inicializá-la. Quando uma variável é declarada, um espaço na memória *RAM* do computador é reservado para essa variável. Esse espaço pode ter sido utilizado anteriormente, por outro programa e pode conter dados antigos. Dessa forma, se uma variável não inicializada for utilizada, o valor antigo armazenado no espaço de memória *RAM* associado a essa variável será utilizado.

Muitos programadores C esquecem de inicializar suas variáveis com valores adequados. Isso

provoca muitos erros de lógica. Em C#, esse problema não existe pois as variáveis devem sempre ser inicializadas antes de serem utilizadas.

2.3 Exibindo os Valores das Variáveis



O valor de uma variável pode ser exibido na saída padrão. Na primeira linha do exemplo abaixo, a variável `numero` foi declarada e inicializada com o valor 10. Depois, o valor dessa variável foi exibido na tela através do método `WriteLine`.

```
1 int numero = 10;
2
3 System.Console.WriteLine(numero);
```



Pare para pensar...

Qual é a diferença entre as duas linhas a seguir?

```
1 System.Console.WriteLine(numero);
2 System.Console.WriteLine("numero");
```

Na primeira linha, o valor armazenado na variável `numero` é exibido na saída padrão. Na segunda linha, o texto “`numero`” é exibido na saída padrão.

2.4 Copiando Valores



Uma variável pode receber uma cópia do valor armazenado em outra variável. Na primeira linha do exemplo abaixo, a variável `a` do tipo `int` foi declarada e inicializada com o valor 1. Na sequência, a variável `b` também do tipo `int` foi inicializada com uma cópia do valor armazenado na variável `a`.

```
1 int a = 1;
2
3 int b = a;
```

Alterar o valor armazenado em uma variável não afeta o valor armazenado em uma outra variável. No exemplo abaixo, a variável `a` foi inicializada com o valor 1 e a variável `b` foi inicializada com uma cópia do valor armazenado na variável `a`. Depois, os valores armazenados nessas duas variáveis foram modificados. A variável `a` recebeu o valor 2 e a variável `b` recebeu o valor 3. Lembre-se que alterar o valor da variável `a` não afeta o valor armazenado na variável `b` e vice-versa. Portanto, ao exibir os valores armazenados nessas variáveis, os números 2 e 3 serão apresentados na saída padrão.

```
1 int a = 1;
2
3 int b = a;
4
5 a = 2;
6
7 b = 3;
8
9 System.Console.WriteLine(a); // exibe o valor 2
10
11 System.Console.WriteLine(b); // exibe o valor 3
```



Simulação

Para ilustrar a cópia de valores de variáveis, simularemos a execução de um programa em C#.

- 1** A execução é iniciada na primeira linha do método `Main`. Assim, a execução começa na linha 5 do código abaixo. A instrução dessa linha declara a variável `a` do tipo `int` e a inicializa com o valor 2.

```

1 class Variavel
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = a;
7         System.Console.WriteLine(a);
8         System.Console.WriteLine(b);
9     }
10 }
```

a = 2

- 2** Em seguida, a linha 6 é executada. Nessa linha, é declarada a variável `b` do tipo `int` e seu valor é inicializado com o valor armazenado na variável `a`.

```

1 class Variavel
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = a;
7         System.Console.WriteLine(a);
8         System.Console.WriteLine(b);
9     }
10 }
```

a = 2

b = 2

- 3** Seguindo o fluxo de execução, a linha 7 é executada. A instrução dessa linha exibe na saída padrão o valor armazenado na variável `a`. Assim, o número 2 é exibido na saída padrão.

```

1 class Variavel
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = a;
7         System.Console.WriteLine(a);
8         System.Console.WriteLine(b);
9     }
10 }
```

a = 2

b = 2

- 4** Em seguida, a linha 8 é executada. A instrução dessa linha exibe na saída padrão o valor armazenado na variável `b`.

```

1 class Variavel
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = a;
7         System.Console.WriteLine(a);
8         System.Console.WriteLine(b);
9     }
10 }

```

2
2

a = 2

b = 2

2.5 Tipos Simples



Como vimos anteriormente, toda variável possui um tipo. Na linguagem C#, há um conjunto básico de tipos denominados **tipos simples**. Os tipos simples são utilizados com muita frequência e servem como base para a criação de outros tipos. Os tipos simples da linguagem C# são apresentados na Tabela 2.1.

| Nome | Tipo | Intervalo | Espaço |
|---------|----------|---|----------|
| sbyte | Inteiro | De -128 a 127 | 1 byte |
| byte | Inteiro | De 0 a 255 | 1 byte |
| short | Inteiro | De -32768 a 32767 | 2 bytes |
| ushort | Inteiro | De 0 a 65535 | 2 bytes |
| int | Inteiro | De -2147483648 a 2147483647 | 4 bytes |
| uint | Inteiro | De 0 a 4294967295 | 4 bytes |
| long | Inteiro | De -9223372036854775808 a 9223372036854775807 | 8 bytes |
| ulong | Inteiro | De 0 a 18446744073709551615 | 8 bytes |
| char | Inteiro | De 0 a 65535 | 2 bytes |
| float | Real | Aproximadamente de -3.4×10^{38} a 3.4×10^{38} | 4 bytes |
| double | Real | Aproximadamente de -1.7×10^{308} a 1.7×10^{308} | 8 bytes |
| decimal | Real | Aproximadamente de -7.9×10^{28} a 7.9×10^{28} | 16 bytes |
| bool | Booleano | O valor true (verdadeiro) ou o valor false (falso) | 1 byte |

Tabela 2.1: Tipos simples

Os tipos simples serão classificados da seguinte forma:

Numéricos: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double e decimal.

Inteiros: sbyte, byte, short, ushort, int, uint, long, ulong e char.

Reais: float, double e decimal.

Booleano: bool.



Mais Sobre

O tipo `float` e o tipo `double` aceitam também os valores `±Infinity`, `±0` e `NaN` (Not a Number). Os valores `±Infinity` são representados pelas constantes `float.NegativeInfinity`, `float.PositiveInfinity`, `double.NegativeInfinity` e `double.PositiveInfinity`. O valor `NaN` é representado pelas constantes `float.NaN` e `double.NaN`. Veremos mais sobre esses valores adiante.

BigInteger

Para armazenar números inteiros que extrapolam os intervalos de valores dos tipos simples numéricos, podemos utilizar o tipo `BigInteger`.

<http://msdn.microsoft.com/en-us/library/system.numerics.biginteger>

Qualquer número aceito por um tipo simples inteiro pode ser armazenado em uma variável do tipo `BigInteger`. Além desses valores, o tipo `BigInteger` é capaz de armazenar muitos outros. Assim, alguém poderia argumentar que optar por uma variável do tipo `BigInteger` é sempre a melhor escolha.

No entanto, o uso de `BigInteger` tem algumas desvantagens em relação ao uso dos tipos simples inteiros. Uma delas é que uma variável do tipo `BigInteger` ocupa mais espaço. Além disso, operações envolvendo os valores armazenados em variáveis do tipo `BigInteger` são consideravelmente mais lentas do que as operações envolvendo os tipos simples inteiros.



Analogia

Escolher entre os tipos simples numéricos e o tipo `BigInteger` é semelhante a optar pelo uso de um carro ou de um caminhão. De fato, o caminhão é capaz de transportar tudo o que um carro é capaz de transportar e muito mais. Por outro lado, estacionar um caminhão em um shopping, por exemplo, é bem mais complicado do que estacionar um carro. Além disso, o gasto de combustível do caminhão será provavelmente maior do que o gasto do carro.

Valores mínimos e máximos

Os programadores não precisam decorar com exatidão os valores mínimos e máximos aceitos por cada tipo simples numérico. Na linguagem C#, esses valores podem ser acessados através das constantes apresentadas na Tabela 2.2.

| Tipo | Mínimo | Máximo |
|---------|------------------|------------------|
| sbyte | sbyte.MinValue | sbyte.MaxValue |
| byte | byte.MinValue | byte.MaxValue |
| short | short.MinValue | short.MaxValue |
| ushort | ushort.MinValue | ushort.MaxValue |
| int | int.MinValue | int.MaxValue |
| uint | uint.MinValue | uint.MaxValue |
| long | long.MinValue | long.MaxValue |
| ulong | ulong.MinValue | ulong.MaxValue |
| char | char.MinValue | char.MaxValue |
| float | float.MinValue | float.MaxValue |
| double | double.MinValue | double.MaxValue |
| decimal | decimal.MinValue | decimal.MaxValue |

Tabela 2.2: Constantes para os valores mínimos e máximos dos tipos simples

Por exemplo, para exibir o maior valor que uma variável do tipo `int` pode armazenar, podemos utilizar o seguinte código.

```
1 System.Console.WriteLine(2147483647);
```

Contudo, alguém que venha a ler esse código pode não reconhecer esse número como sendo o maior valor do tipo `int`. Uma situação como essa pode comprometer o entendimento do código por parte do leitor. Para melhorar a legibilidade do código, podemos utilizar a constante `int.MaxValue`.

```
1 System.Console.WriteLine(int.MaxValue);
```



Mais Sobre

A propriedade `Epsilon` dos tipos `float` e `double` armazena o menor valor positivo que esses tipos podem armazenar.

Números inteiros

Números inteiros podem ser armazenados em variáveis dos tipos `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong`. Basicamente, para escolher o tipo mais apropriado, devemos considerar a grandeza dos números que desejamos armazenar.

Por exemplo, considere a numeração dos andares de um prédio. Nesse prédio, há 3 subsolos numerados com -1, -2 e -3, o térreo é numerado com 0 e os outros 20 andares com os números de 1 a 20. Precisamos armazenar o número de um andar qualquer desse prédio. De acordo com a Tabela 2.1, o tipo `sbyte` aceita valores entre -128 e 127. Portanto, o tipo `sbyte` é uma escolha adequada para a nossa variável.

Atualmente, os motores dos automóveis comuns não ultrapassam 10.000 *RPM* (rotações por minuto). Dessa forma, é possível armazenar a rotação máxima de um automóvel em uma variável do tipo `short`.

No livro *O cérebro feminino*, a neuropsiquiatra *Louann Brizendine* afirmou que, em média, uma mulher fala 20 mil palavras por dia. Vamos considerar que, em um dia atípico, uma mulher pode falar cerca de 40 mil palavras. Dessa forma, para armazenar a quantidade de palavras que uma mulher fala em um dia, seria razoável utilizar uma variável do tipo `int`.

Atualmente, a população mundial é de aproximadamente 7 bilhões de pessoas. Para armazenar esse valor, devemos utilizar uma variável do tipo `long`.

Na Tabela 2.1, você pode verificar que a quantidade de espaço ocupado por uma variável depende do seu tipo. Para diminuir a quantidade de memória *RAM* utilizada, podemos priorizar o uso dos tipos que ocupam menos espaço.

Números reais

Números reais podem ser armazenados em variáveis dos tipos `float`, `double` e `decimal`. Para escolher o tipo mais apropriado, devemos considerar a grandeza e a precisão dos números que desejamos armazenar. O tipo `double` permite armazenar valores de maior grandeza e de maior precisão quando comparado ao tipo `float`. O tipo `decimal` permite armazenar valores com maior precisão mas com menor grandeza quando comparado aos tipos `double` e `float`.

O menor valor que pode ser representado pelo tipo `double` é aproximadamente -1.7×10^{308} e o maior é aproximadamente 1.7×10^{308} . A quantidade de valores reais entre esses dois números é infinita. Contudo, o tipo `double` permite apenas a representação de uma quantidade finita de valores nesse intervalo.

Por exemplo, o número 1 pode ser representado em `double`. Depois do número 1, o próximo valor que pode ser representado em `double` é 1.0000000000000002220446049250313080847263336181640625 (esse valor é igual a $1 + 2^{-52}$). Depois desse número, o próximo número que pode ser representado é 1.000000000000000444089209850062616169452667236328125 (que é igual a $1 + 2^{-51}$).

De maneira geral, um valor que não pode ser representado em `double` é substituído pelo valor mais próximo que pode ser representado. Por exemplo, o número 1.0000000000000003 não pode ser representado em `double`. Assim, esse número é substituído pelo valor mais próximo que pode ser representado em `double`, que é 1.0000000000000002220446049250313080847263336181640625.

O tipo `float` funciona de forma análoga ao tipo `double`. O menor valor do tipo `float` é aproximadamente -3.4×10^{38} e o maior é aproximadamente 3.4×10^{38} .

O valor especial `NaN` (Not-a-Number) representa o resultado de algumas operações inválidas envolvendo valores do tipo `float` ou do tipo `double`. O valor especial `Infinity` representa o infinito. O resultado de algumas operações envolvendo valores do tipo `float` ou do tipo `double` pode ser `+Infinity` ou `-Infinity`.



Armadilha

Ao exibir números com muitas casas decimais, o método `WriteLine` realiza arredondamentos. No exemplo abaixo, ao tentar exibir o valor 1.0000000000000003, o método `WriteLine` exibe o valor 1.

```
1 System.Console.WriteLine(1.0000000000000003); // exibe 1
```

O tipo `decimal` funciona de forma análoga aos tipos `double` e `float`. O menor valor do tipo `decimal` é aproximadamente -7.9×10^{-28} e o maior é aproximadamente 7.9×10^{28} . Como vimos, a vantagem do tipo `decimal` em relação aos tipos `float` e `double` é a precisão. Por exemplo, o valor 0.1 pode ser representado em `decimal` mas não pode ser representado em `float` ou `double`. Por outro lado, a desvantagem do tipo `decimal` em relação aos tipos `float` e `double` é a grandeza. Por exemplo, não podemos representar números superiores a 7.9×10^{28} com o tipo `decimal` mas podemos com os tipos `float` ou `double`.



Curiosidade

No dia 25 de Fevereiro de 1991, em Dhahran, na Arábia Saudita, durante a Guerra do Golfo, o sistema antimíssil americano chamado *Patriot* falhou e não conseguiu interceptar um míssil iraquiano. Esse míssil atingiu o seu alvo, um alojamento americano. No total, 28 soldados americanos morreram e outras 100 pessoas ficaram feridas.

A falha ocorreu devido a um problema de precisão numérica no software que controlava o sistema *Patriot*. O sistema *Patriot* possui um relógio interno que armazena em décimos de segundos o tempo de funcionamento do equipamento desde a sua última inicialização. O software multiplicava esse valor por 0.1 para obter o tempo de funcionamento do equipamento em segundos. O resultado dessa multiplicação era utilizado para calcular a trajetória dos mísseis que deveriam ser interceptados.

Contudo, o formato numérico utilizado pelo software para armazenar o valor 0.1 é o Ponto Fixo de 24 bits. Esse valor não pode ser representado nesse formato. Sendo assim, o software realizava os cálculos com o valor mais próximo que pode ser representado em Ponto Fixo de 24 bits. Para ser mais específico, o software realizava os cálculos com o valor 0.099999904632568359375.

Depois de 100 horas de funcionamento, essa falta de precisão causava um desvio de aproximadamente 687 metros nos cálculos das trajetórias dos mísseis. Isso causou a falha do dia 25 de Fevereiro de 1991.

Oficiais das forças armadas americanas disseram que uma versão corrigida do software foi finalizada no dia 16 de Fevereiro de 1991. Mas, essa versão só chegou em Dhahran no dia 26 de Fevereiro de 1991. Ou seja, um dia depois da falha fatal.

Se esse software tivesse sido desenvolvido com a linguagem de programação C#, o problema descrito acima poderia ser resolvido com a utilização do tipo `decimal`. O valor 0.1 pode ser representado no formato adotado pelo tipo `decimal`.

Verdadeiro ou falso

Variáveis do tipo `bool` podem armazenar o valor `true` (verdadeiro) ou o valor `false` (falso). Não podemos armazenar números em variáveis do tipo `bool`.

Caracteres

Tecnicamente, uma variável do tipo `char` armazena um número inteiro entre 0 e 65.535. Contudo, o valor armazenado em um variável do tipo `char` representa o código de um caractere de acordo com

a codificação *UTF-16* do padrão *Unicode* (<http://www.unicode.org/>). De forma abstrata, podemos dizer que uma variável do tipo `char` armazena um caractere.

2.6 Nullable Types



Em C#, o valor `null` representa o vazio. Para expressar que uma variável está “vazia”, podemos armazenar o valor `null` nessa variável. Os tipos simples vistos anteriormente não permitem o armazenamento do valor `null`. Daí surgem os **nullable types**. Os nullable types correspondentes aos tipos simples `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` e `bool` são `sbyte?`, `byte?`, `short?`, `ushort?`, `int?`, `uint?`, `long?`, `ulong?`, `char?`, `float?`, `double?`, `decimal?` e `bool?` respectivamente. Observe que o nome dos nullable types correspondentes aos tipos simples são exatamente o nome dos tipos simples com o caractere “?” no final.

2.7 String



Na linguagem C#, o tipo `string` é um dos mais importantes e mais utilizados. O tipo `string` é usado para armazenar texto (sequência de caracteres). No exemplo abaixo, a variável `texto` do tipo `string` foi associada à sequência de caracteres “K19 Treinamentos”.

```
1 string texto = "K19 Treinamentos";
```

Código C# 2.17: Utilizando o tipo `string`

Qualquer caractere definido na codificação *UTF-16* do padrão *Unicode* pode ser utilizado para formar uma `string`.

O espaço utilizado por uma `string` depende da quantidade de caracteres que ela possui. Cada caractere ocupa 16 bits. Portanto, a `string` “K19 Treinamentos”, que possui 16 caracteres (o caractere espaço também deve ser contabilizado), ocupa 256 bits.

O tipo `string` permite o armazenamento o valor `null`.

2.8 Data e Hora



A linguagem C# possui um tipo específico para armazenar data e hora. Em C#, é muito comum utilizarmos o tipo `DateTime`.

```
1 System.DateTime exatamenteAgora = System.DateTime.Now;
```

Código C# 2.18: Data e hora atuais - `DateTime`

No código acima, a data e hora atuais do computador são associadas à variável `exatamenteAgora`. Também podemos definir datas e horas específicas.

```
java.util.Calendar c = new java.util.GregorianCalendar( 1982 , → Ano
                                                       11 , → Mês (0 ~ 11)
                                                       12 , → Dia (1 ~ 31)
                                                       10 , → Hora (0 ~ 23)
                                                       5 , → Minuto (0 ~ 59)
                                                       30 ) → Segundo (0 ~ 59)
```

Figura 2.2: Data e hora específicas - DateTime

No exemplo acima, o primeiro parâmetro define o ano; o segundo o mês; o terceiro o dia; o quarto a hora; o quinto os minutos; e o sexto os segundos. Dessa forma, a data “12 de Dezembro de 1982” e hora “10:05:30” foram associadas à variável `c`.

2.9 Valores Literais



Os valores inseridos diretamente no código fonte são chamados de valores literais.

Null

Considere um programa que utiliza uma variável do tipo `double?` para armazenar a variação do Dólar em relação ao Real. Essa variável deve ser atualizada diariamente.

```
1 double? variacao;
```

No dia 30 de Abril de 2014, a variação foi de $-0,139\%$. Então, nesse dia, o valor -0.139 foi armazenado na variável `variacao`.

```
1 variacao = -0.139;
```

No dia seguinte, 1 de Maio de 2014, devido ao feriado internacional do dia do trabalho, o mercado financeiro não funcionou. Dessa forma, a cotação do Dólar em relação ao Real não sofreu alteração. Nesse dia, qual valor deveria ser armazenado na variável `variacao`?

Provavelmente, nesse caso, o mais intuitivo seria utilizar o valor 0. Contudo, a utilização desse valor gera uma ambiguidade. A variável `variacao` armazenaria o valor 0 quando o mercado financeiro não funciona, ou seja, nos sábados, domingos, feriados e datas extraordinárias. Mas, também armazenaria o valor 0 quando as operações financeiras realizadas em um determinado dia não alteram a cotação do Dólar em relação ao Real. Se o programa precisa diferenciar essas duas situações, o valor 0 não pode ser utilizado para esses dois casos.

Para resolver esse problema, nos dias em que o mercado financeiro não funciona, o valor `null` poderia ser armazenado na variável `variacao`. O `null` é um valor especial que representa o vazio.

```
1 variacao = null;
```



Importante

As variáveis de tipos simples não aceitam o valor `null`. Mas, os nullable types correspondentes aos tipos simples aceitam.



Pare para pensar...

Como vimos, as variáveis do tipo `string` aceitam o valor `null`. No exemplo abaixo, o valor `null` foi armazenado na variável `nome` que é do tipo `string`.

```
1 string nome = null;
```

Se a palavra chave `null` for definida dentro de aspas dupla, a variável `nome` seria associada à sequência de caracteres “`null`” e não armazenaria o valor `null`.

```
1 string nome = "null";
```

Booleanos

O valor “verdadeiro” é representado pelo valor literal `true` e o valor “falso” pelo valor literal `false`.

```
1 bool a = true;
2 bool b = false;
```

Código C# 2.24: Utilizando valores literais booleanos

Inteiros

Números inteiros podem ser escritos nos formatos decimal e hexadecimal. A representação de um número em cada um desses formatos é uma sequência composta por um ou mais dígitos. No formato decimal, são usados os dígitos de 0 a 9. Já no formato hexadecimal, são usados os dígitos de 0 a 9 e as letras de A a F. A letra A corresponde ao valor 10, a letra B corresponde ao valor 11 e assim sucessivamente. A Tabela 2.3 apresenta os números inteiros de 0 a 21 representados em cada um desses formatos.

| Decimal | Hexadecimal |
|---------|-------------|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | A |
| 11 | B |
| 12 | C |
| 13 | D |
| 14 | E |
| 15 | F |
| 16 | 10 |
| 17 | 11 |
| 18 | 12 |
| 19 | 13 |
| 20 | 14 |
| 21 | 15 |

Tabela 2.3: Números de 0 a 21 representados em decimal e hexadecimal

Em C#, valores literais inteiros podem ser escritos no código fonte em decimal ou hexadecimal. Mas como podemos observar na Tabela 2.3, o número 10 em formato decimal tem a mesma representação que o número 16 em formato hexadecimal. Para diferenciá-los, a linguagem C# define algumas regras.

- Para utilizar o formato hexadecimal, devemos acrescentar o prefixo **0x** ou o prefixo **0X**.
- Qualquer outra sequência formada por dígitos de 0 a 9 estará no formato decimal.

No código abaixo, o número 19 é atribuído a variáveis do tipo `int` utilizando os formatos decimal e hexadecimal.

```

1 // 19 em decimal
2 int c = 19;
3
4 // 19 em hexadecimal
5 int d = 0x13;
```

Código C# 2.25: Número 19 nos formatos decimal e hexadecimal

Por padrão, o tipo de um valor literal inteiro será o primeiro dos seguintes tipos no qual esse valor pode ser representado: `int`, `uint`, `long` e `ulong`. Por exemplo, o valor 1000000000 (1 bilhão) pode ser representado com o tipo `int`. Portanto, ele será do tipo `int`. O valor 4000000000 (4 bilhões) não pode ser representado com o tipo `int` mas pode ser representado com o tipo `uint`. Portanto, ele será do tipo `uint`. O valor 10000000000 (10 bilhões) não pode ser representado com o tipo `uint` mas pode ser representado com o tipo `long`. Portanto, ele será do tipo `long`. O valor 10000000000000000000 (10 quinzelhões) não pode ser representado com o tipo `long` mas pode ser representado com o tipo `ulong`. Portanto, ele será do tipo `ulong`.

O tipo de um valor literal inteiro terminado com o sufixo **U** (u maiúsculo) ou **u** (u minúsculo) será o primeiro dos seguintes tipos no qual esse valor pode ser representado: `uint` e `ulong`. Por exemplo, o valor 1000000000U (1 bilhão) pode ser representado com o tipo `uint`. Portanto, ele será do tipo `uint`. O valor 10000000000000000000U (10 quinzelhões) não pode ser representado com o tipo `uint` mas pode ser representado com o tipo `ulong`. Portanto, ele será do tipo `ulong`.

Agora, o tipo de um valor literal inteiro terminado com o sufixo **L** (éle maiúsculo) ou **l** (éle minúsculo) será o primeiro dos seguintes tipos no qual esse valor pode ser representado: `long` e `ulong`. Por

exemplo, o valor 1000000000L (1 bilhão) pode ser representado com o tipo `long`. Portanto, ele será do tipo `long`. O valor 10000000000000000000000000L (10 quinquilhões) não pode ser representado com o tipo `long` mas pode ser representado com o tipo `ulong`. Portanto, ele será do tipo `ulong`.

Por fim, um valor literal inteiro terminado com um dos seguintes sufixos: **UL**, **Ul**, **uL**, **ul**, **LU**, **Lu**, **IU** ou **lu** será do tipo `ulong`.

Reais

Valores literais reais são definidos com o separador de casas decimais “.” (ponto). Veja alguns exemplos:

```
1 double a = 19.19;
2
3 double b = 0.19;
4
5 double c = .19;
```

Código C# 2.26: Valores literais reais

Por padrão, independentemente da grandeza do número, os valores literais reais são tratados como `double`. Por exemplo, considere o valor `19.09`. Esse valor poderia ser tratado como `float`, `double` ou `decimal`. Contudo, por padrão, ele será tratado como `double`. Dessa forma, o código a seguir gera um erro de compilação.

```
1 float a = 19.09;
```

Código C# 2.27: Erro de compilação

Para resolver esse problema, devemos utilizar o sufixo **F** (éfe maiúsculo) ou **f** (éfe minúsculo). Ao utilizar um desses sufixos, indicamos ao compilador que o valor literal real deve ser tratado como `float`.

```
1 float a = 19.09F;
2
3 float b = 19.09f;
```

Código C# 2.28: Utilizando o sufixo *F* e *f*

De forma análoga, podemos indicar que um valor literal é do tipo `decimal` utilizando o sufixo **M** (eme maiúsculo) ou **m** (eme minúsculo).

Para indicar que um valor literal é do tipo `double`, podemos utilizar o sufixo **D** (dê maiúsculo) ou **d** (dê minúsculo). Esses sufixos são redundantes para valores literais reais pois, por padrão, esses valores já são do tipo `double`.

Também podemos definir valores literais reais na forma exponencial. No exemplo abaixo, a variável `a` foi inicializada com o valor “`1.74e3`”. Esse valor é o resultado da multiplicação do número à esquerda do caractere **e** (é minúsculo) por 10 elevado ao número à direita do caractere **e** (é minúsculo). Em outras palavras, esse valor é igual a 1.74×10^3 , ou seja, igual a 1740.

```
1 double a = 1.74e3; // 1740 na forma exponencial
```

Código C# 2.29: Forma exponencial

O caractere **E** (é maiúsculo) pode ser utilizado no lugar do caractere **e** (é minúsculo).

```
1 double a = 1.74E3; // 1740 na forma exponencial
```

Código C# 2.30: Forma exponencial

Caracteres

Caracteres literais são definidos dentro de aspas simples. No exemplo abaixo, a variável **a** foi inicializada com o código numérico do caractere **K**.

```
1 char a = 'K';
```

Código C# 2.31: Caracteres literais

O código do caractere **K** é 75. Dessa forma, tecnicamente, a variável **a** do exemplo acima armazena o valor 75.



Importante

Apenas um caractere pode ser definido dentro de aspas simples.



Importante

Para definir os caracteres \, ' e " (barra invertida, aspas simples e aspas duplas), devemos acrescentar uma \ (barra invertida) à frente desses caracteres. Assim, devemos utilizar as sequências de escape \\, \' e \" . Veja o exemplo a seguir.

```
1 char a = '\\';
2
3 char b = '\'';
4
5 char c = '\"';
```

A quantidade de caracteres definidos pela codificação *UTF-16* do padrão *Unicode* é muito maior do que a quantidade de teclas do seu teclado. Dessa forma, você não pode digitar a maior parte dos caracteres aceitos pela linguagem C#.

Para definir caracteres que não estão associados às teclas do seu teclado, você pode utilizar o código de cada caractere. Por exemplo, para definir o caractere Ω (ômega), podemos utilizar o código 937.

No exemplo abaixo, utilizamos o valor \u03A9 dentro de aspas simples. O número à direita do caractere **u** (u minúsculo) é o código do caractere desejado em notação hexadecimal com 4 dígitos.

```
1 char b = '\u03A9'; // 937 em notação hexadecimal com 4 dígitos
```

Código C# 2.33: Utilizando o código dos caracteres

Também podemos utilizar o prefixo \x em conjunto com o código hexadecimal do caractere desejado.

```
1 char c = '\x3A9'; // 937 em notação hexadecimal
```

Código C# 2.34: Utilizando o código dos caracteres

Strings

Strings literais são definidas dentro de aspas duplas. No exemplo abaixo, a variável `a` foi associada à sequência de caracteres “K19 Treinamentos”.

```
1 string a = "K19 Treinamentos";
```

Código C# 2.35: Strings literais

Para simplificar, uma string literal é uma sequência de caracteres literais dentro de aspas duplas.



Importante

Para utilizar os caracteres \ e " (barra invertida e aspas duplas) dentro de uma string, devemos usar as sequências de escape \\ e \" , respectivamente.

No exemplo abaixo, ocorre um erro de compilação.

```
1 string a = "C:\k19\rafael\cosentino";
```

Código C# 2.36: Erro de compilação

Para solucionar esse problema, podemos utilizar a sequência de escape \\.

```
1 string a = "C:\\k19\\\\rafael\\\\cosentino";
```

Código C# 2.37: Utilizando a sequência de escape \\

Outra forma de solucionar esse problema é acrescentar o caractere @ no começo da string.

```
1 string a = @"C:\\k19\\\\rafael\\\\cosentino";
```

Código C# 2.38: Utilizando o caractere @

2.10 Constantes



Podemos utilizar a palavra reservada `const` para declarar constantes. Uma constante é como uma variável cujo conteúdo não pode ser alterado. No exemplo abaixo, a constante `a` do tipo `int` foi declarada com o uso da palavra reservada `const` e inicializada com o valor 10. Esse valor não pode ser mais alterado.

```
1 const int a = 10;
```

Por exemplo, no código abaixo, a constante `a` foi inicializada com o valor 10. Depois dessa inicialização, na tentativa de trocar o valor dessa constante, um erro de compilação é gerado.

```

1 const int a = 10;
2
3 a = 5; // erro de compilação

```

Código C# 2.40: Tentando alterar o valor de uma constante

A inicialização de uma constante precisa ser realizada junto com a sua declaração.

```

1 const int a; // erro de compilação

```

Código C# 2.41: Declarando uma constante sem inicializá-la



2.11 Números Aleatórios

Para realizar mostrar alguns exemplos, utilizaremos números aleatórios. Em C#, esses números podem ser gerados facilmente. No exemplo a seguir, utilizamos a classe `Random` e o método `NextDouble` para gerar números aleatórios do tipo `double` maiores ou iguais a **0** e menores do que **1**.

```

1 System.Random gerador = new System.Random();
2
3 double numero = gerador.NextDouble();

```

Código C# 2.42: Gerando números aleatórios

Podemos adaptar o intervalo dos números gerados com algumas operações matemáticas. Suponha que você queira gerar aleatoriamente um número que seja maior ou igual a um certo valor mínimo e menor do que um certo valor máximo. O código abaixo exemplifica como essa tarefa pode ser feita.

```

1 double minimo = -5.0;
2 double maximo = 17.3;
3
4 // minimo <= numero < maximo
5 double numero = minimo + gerador.NextDouble() * (maximo - minimo);

```

Código C# 2.43: Gerando números aleatórios num intervalo específico

A classe `Random` também possui o método `Next`, que gera um número inteiro não negativo aleatório menor do que um determinado valor. No exemplo abaixo, usamos esse método para gerar um número inteiro aleatório entre 0 e 10.

```

1 int limite = 11;
2
3 // 0 <= numero < limite
4 int numero = gerador.Next(limite);

```

Código C# 2.44: Gerando números inteiros aleatórios



Simulação

Vamos simular a execução de um programa em C# que gera um número aleatório através do método `NextDouble` e o exibe na saída padrão.

- 1 A execução inicia na primeira linha do método Main. Assim, a execução começa na linha 5 do código abaixo. A instrução dessa linha cria um objeto capaz de gerar números aleatórios.

```

1 class NumeroAleatorio
2 {
3     static void Main()
4     {
5         System.Random gerador = new System.Random();
6         double a = gerador.NextDouble();
7         System.Console.WriteLine(a);
8     }
9 }
```

- 2 Agora, o fluxo de execução segue para a linha 6. A instrução dessa linha declara a variável a do tipo double e a inicializa com o valor devolvido pelo método NextDouble.

```

1 class NumeroAleatorio
2 {
3     static void Main()
4     {
5         System.Random gerador = new System.Random();
6         double a = gerador.NextDouble();
7         System.Console.WriteLine(a);
8     }
9 }
```

a = 0.87

- 3 Em seguida, a linha 7 é executada. A instrução presente nessa linha exibe o valor da variável a na saída padrão.

```

1 class NumeroAleatorio
2 {
3     static void Main()
4     {
5         System.Random gerador = new System.Random();
6         double a = gerador.NextDouble();
7         System.Console.WriteLine(a);
8     }
9 }
```

a = 0.87

2.12 Convenções de Nomenclatura



Os nomes das variáveis são fundamentais para o entendimento do código fonte. Considere o exemplo a seguir.

```

1 int j;
2 int f;
3 int m;
```

Você consegue deduzir quais dados serão armazenados nas variáveis j, f e m? Provavelmente, não. Vamos melhorar um pouco os nomes dessas variáveis.

```
1 int jan;
```

```
2 int fev;  
3 int mar;
```

Agora, talvez, você tenha uma vaga ideia. Vamos melhorar mais um pouco os nomes dessas variáveis.

```
1 int janeiro;  
2 int fevereiro;  
3 int marco;
```

Agora sim! Você já sabe para que servem essas variáveis? Se você parar para pensar ainda não sabe muita coisa sobre elas. Então, é importante melhorar mais uma vez o nome dessas variáveis.

```
1 int numeroDePedidosEmJaneiro;  
2 int numeroDePedidosEmFevereiro;  
3 int numeroDePedidosEmMarco;
```

Finalmente, os nomes das variáveis conseguem expressar melhor a intenção delas. Consequentemente, a leitura e o entendimento do código fonte seria mais fácil.

Geralmente, bons nomes de variáveis são compostos por várias palavras como no exemplo a seguir.

```
1 int numeroDeCandidatosAprovados;
```

Quando o nome de uma variável é composto, é fundamental adotar alguma convenção para identificar o início e o término das palavras. A separação natural das palavras na língua portuguesa são os espaços. Contudo, os nomes das variáveis em C# não podem possuir espaços. Não adotar uma convenção de nomenclatura para identificar o início e o término das palavras é como escrever um texto em português sem espaços entre as palavras. Em alguns casos, o leitor não saberia como separar as palavras. Considere o exemplo abaixo.

salamesadia

O que está escrito no texto acima? A resposta depende da divisão das palavras. Você pode ler como “sala mesa dia” ou “salame sadia”. Dessa forma, fica clara a necessidade de deixar visualmente explícita a divisão das palavras.

Em algumas linguagens de programação, delimitadores são utilizados para separar as palavras que formam o nome de uma variável.

numero_de_candidatos_aprovados;
numero-de-candidatos-aprovados;

Em outras linguagens de programação, letras maiúsculas e minúsculas são utilizadas para separar as palavras.

NumeroDeCandidatosAprovados;
numeroDeCandidatosAprovados;

Em C#, a convenção de nomenclatura adotada para separar as palavras que formam o nome de uma variável é o *Camel Case*, que consiste em escrever o nome da variável com a primeira letra de cada palavra em maiúscula com exceção da primeira letra da primeira palavra.

```
1 int numeroDaConta; // segue a convenção
2 int Numerodaconta; // não segue a convenção
```

Código C# 2.53: Convenção para nomes de variáveis

Também devemos nos lembrar que a linguagem C# é **case sensitive**. Dessa forma, `numeroDaConta` e `Numerodaconta` são consideradas variáveis diferentes.



Importante

Considere um código C# que declara uma variável chamada `pontuação`. Note o uso dos caracteres “ç” e “ã” no nome dessa variável. Geralmente, os códigos desses caracteres são diferentes em cada padrão de codificação. Por exemplo, no *UTF-8*, o código do caractere “ç” é 50087, enquanto que no *ISO-8859-1* é 231.

Suponha que esse código tenha sido salvo em um arquivo que utiliza a codificação *UTF-8*. Se ele for aberto em um editor que utiliza a codificação *ISO-8859-1*, o caractere “ç” não será apresentado corretamente, dificultando a leitura ou a modificação do código fonte.

Para evitar esse tipo de problema, a recomendação é utilizar apenas as letras de A a Z (tanto maiúsculas quanto minúsculas) e os dígitos de 0 a 9 pois, geralmente, os códigos desses caracteres não variam de codificação para codificação.

2.13 Regras de Nomenclatura



A linguagem C# possui regras técnicas relacionadas à nomenclatura das variáveis. O nome (identificador) de uma variável é uma sequência limitada de caracteres que:

1. Deve começar com uma letra ou com o caractere `_`.
2. Se for igual a uma palavra reservada (ver Tabela 2.4), é necessário adicionar o prefixo `@`.
3. Pode conter letras e dígitos.

```
1 // válido
2 int numeroDaConta;
3
4 // inválido pois o nome de uma variável não pode começar com dígito
5 int 2outraVariavel;
6
7 // inválido pois o nome de uma variável não pode ser igual a uma palavra reservada
8 double double;
9
10 // inválido pois o nome de uma variável não pode conter espaços
11 double saldo da conta;
12
13 // válido
14 int umaVariavelComUmNomeSuperHiperMegaUltraGigante;
15
16 // válido
```

```

17 int numeroDaContaCom8Digitos_semPontos;
18
19 // inválido pois o caractere # não é considerado uma letra
20 int #telefone;
21
22 // válido pois apesar de ser igual a uma palavra reservada possui o prefixo @
23 int @static;

```

Código C# 2.54: Exemplos de nomes de variáveis válidos e inválidos



2.14 Palavras Reservadas

Toda linguagem de programação possui um conjunto de palavras reservadas. Em geral, essas palavras representam os comandos da linguagem. Na Tabela 2.4, são apresentadas as palavras reservadas da linguagem C#.

| | | | | |
|----------|----------|------------|-----------|-----------|
| abstract | as | base | bool | break |
| byte | case | catch | char | checked |
| class | const | continue | decimal | default |
| delegate | do | double | else | enum |
| event | explicit | extern | false | finally |
| fixed | float | for | foreach | goto |
| if | implicit | in | int | interface |
| internal | is | lock | long | namespace |
| new | null | object | operator | out |
| override | params | private | protected | public |
| readonly | ref | return | sbyte | sealed |
| short | sizeof | stackalloc | static | string |
| struct | switch | this | throw | true |
| try | typeof | uint | ulong | unchecked |
| unsafe | ushort | using | virtual | void |
| volatile | while | | | |

Tabela 2.4: Palavras reservadas da linguagem C#



2.15 Erro: Variáveis com nomes repetidos

Um erro de compilação comum em C# ocorre quando duas ou mais variáveis são declaradas com nome repetido em um mesmo bloco. No exemplo abaixo, três variáveis com o mesmo nome foram declaradas.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 10;
6
7         double a = 10.7;
8
9         int a = 5;
10    }
11 }

```

Código C# 2.55: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(7,10): error CS0128: A local variable named 'a' is already defined
    in this scope
Programa.cs(7,14): error CS0266: Cannot implicitly convert type 'double' to
    'int'. An explicit conversion exists (are you missing a cast?)
Programa.cs(9,7): error CS0128: A local variable named 'a' is already defined in
    this scope
```

Terminal 2.4: Erro de compilação

2.16 Erro: Esquecer a inicialização de uma variável local

Outro erro de compilação comum em C# ocorre quando utilizamos uma variável local não inicializada. No exemplo abaixo, a variável a foi utilizada sem antes ter sido inicializada.

```
1 class Programa
2 {
3     static void Main()
4     {
5         int a;
6
7         System.Console.WriteLine(a);
8     }
9 }
```

Código C# 2.56: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(7,28): error CS0165: Use of unassigned local variable 'a'
```

Terminal 2.5: Erro de compilação

2.17 Erro: Trocar aspas simples por aspas duplas ou vice-versa

Mais um erro comum em C# ocorre quando utilizamos aspas simples onde deveríamos usar aspas duplas ou vice-versa. Veja um exemplo de programa em C# que utiliza aspas duplas onde deveria haver aspas simples.

```
1 class Programa
2 {
3     static void Main()
4     {
5         char c = "A";
6     }
7 }
```

Código C# 2.57: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,12): error CS0029: Cannot implicitly convert type 'string' to
  'char'
```

Terminal 2.6: Erro de compilação

Agora, veja um exemplo de programa em C# que utiliza aspas simples onde deveria haver aspas duplas.

```
1 class Programa
2 {
3     static void Main()
4     {
5         string s = 'K19 Treinamentos';
6     }
7 }
```

Código C# 2.58: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,14): error CS1012: Too many characters in character literal
```

Terminal 2.7: Erro de compilação

2.18 Erro: Utilizar o separador decimal errado

Outro erro de compilação comum em C# ocorre quando não utilizamos o separador decimal correto. No exemplo abaixo, as casas decimais não foram separadas com o caractere “.” (ponto).

```
1 class Programa
2 {
3     static void Main()
4     {
5         double d = 19,09;
6     }
7 }
```

Código C# 2.59: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,17): error CS1001: Identifier expected
```

Terminal 2.8: Erro de compilação

2.19 Erro: Valores incompatíveis com os tipos das variáveis

Também é um erro de compilação comum em C# atribuir valores incompatíveis com os tipos das variáveis. No exemplo abaixo, tentamos armazenar um valor do tipo `double` em uma variável do tipo `int`.

```
1 class Programa
2 {
3     static void Main()
4     {
```

```

5     int a = 19.09;
6 }
7 }
```

Código C# 2.60: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,11): error CS0266: Cannot implicitly convert type 'double' to
'int'. An explicit conversion exists (are you missing a cast?)
```

Terminal 2.9: Erro de compilação

2.20 Erro: Esquecer dos caracteres de tipagem para float ou decimal

Quando desejamos utilizar valores literais dos tipos `float` ou `decimal`, não podemos esquecer dos caracteres de tipagem (“F”, “f”, “M” e “m”). Veja alguns exemplos de programa em C# com esse problema.

```

1 class Programa
2 {
3     static void Main()
4     {
5         float a = 3.14;
6     }
7 }
```

Código C# 2.61: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,13): error CS0664: Literal of type double cannot be implicitly
converted to type 'float'; use an 'F' suffix to create a literal of this
type
```

Terminal 2.10: Erro de compilação

```

1 class Programa
2 {
3     static void Main()
4     {
5         decimal a = 3.14;
6     }
7 }
```

Código C# 2.62: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(5,15): error CS0664: Literal of type double cannot be implicitly
converted to type 'decimal'; use an 'M' suffix to create a literal of
this type
```

Terminal 2.11: Erro de compilação



2.21 Resumo

- 1 ► As variáveis são utilizadas para armazenar os dados que um programa deve manipular.
- 2 ► Toda variável possui um nome (identificador).
- 3 ► Em C#, as variáveis são classificadas em tipos (“tipadas”).
- 4 ► Para armazenar um valor em uma variável, esse valor deve ser compatível com o tipo da variável.
- 5 ► Em C#, as variáveis devem ser inicializadas antes de serem utilizadas.
- 6 ► A linguagem C# possui treze tipos simples: byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal, bool e char.
- 7 ► As variáveis de tipos simples não aceitam o valor null.
- 8 ► Para armazenar números inteiros “grandes”, podemos utilizar o tipo BigInteger.
- 9 ► Uma variável do tipo string pode ser associada a uma sequência de caracteres.
- 10 ► Em C#, para trabalhar com datas e horas, podemos usar o tipo System.DateTime.
- 11 ► Em C#, os valores literais do tipo float devem possuir o sufixo “f” ou ”F”.
- 12 ► Em C#, o separador de casas decimais é o “.” (ponto).
- 13 ► Em C#, os valores literais booleanos são true e false.
- 14 ► Strings literais são definidas dentro de aspas duplas.
- 15 ► O valor de uma constante não pode ser modificado.

- 16 ► Podemos gerar números aleatórios em C# com o método `NextDouble()` da classe `System.Random`.
- 17 ► As convenções de nomenclatura de variáveis são importantes para melhorar a legibilidade do código.
- 18 ► Em C#, as convenções de nomenclatura de variáveis são baseadas em letras maiúsculas e minúsculas.

OPERADORES

3.1 Introdução



Para manipular os valores literais ou os dados armazenados nas variáveis de uma aplicação, devemos utilizar os operadores oferecidos pela linguagem de programação que estamos utilizando. Os principais tipos de operações são:

- Conversões
- Aritméticas (+ - * / %)
- Atribuições (= += -= *= /= %= ++ --)
- Comparações (== != < <= > >=)
- Lógicas (& | ^ && || ! ? :)

3.2 Conversões Entre Tipos Simples



Considere um número inteiro dentro do intervalo de valores do tipo `int`. Esse valor pode ser armazenado em uma variável do tipo `long`, pois todos os valores que estão no intervalo do tipo `int` também estão no intervalo do tipo `long`.

Por causa disso, podemos copiar diretamente qualquer valor armazenado em uma variável do tipo `int` para uma variável do tipo `long`. Veja o exemplo a seguir.

```
1 int a = 19;  
2 long b = a;
```

Código C# 3.1: Compatibilidade

Agora, considere um número inteiro dentro do intervalo de valores do tipo `long`. Não podemos garantir que esse valor possa ser armazenado em uma variável do tipo `int` porque o intervalo do tipo `long` é mais abrangente do que o intervalo do tipo `int`. Por exemplo, o número **2147483648** está no intervalo do tipo `long` mas não está no intervalo do tipo `int`.

Por causa disso, não podemos copiar diretamente um valor armazenado em uma variável do tipo `long` para uma variável do tipo `int`. A tentativa de realizar esse tipo de cópia gera erro de compilação mesmo que o valor armazenado na variável do tipo `long` seja compatível com `int`. Veja o exemplo a seguir.

```
1 long a = 19;
2 int b = a;
```

Código C# 3.2: Erro de compilação - Incompatibilidade

Observe, na Tabela 3.1, a compatibilidade entre os tipos simples. Note, por exemplo, que um valor do tipo `int` pode ser convertido automaticamente para `long`, `float` ou `double`. Por outro lado, um valor do tipo `long` não pode ser convertido automaticamente para `byte`, `short`, `char` ou `int`.

| Para → De ↓ | sbyte | byte | short | char | ushort | int | uint | long | ulong | float | double | decimal |
|-------------|-------|------|-------|------|--------|-----|------|------|-------|-------|--------|---------|
| sbyte | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| byte | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| short | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| char | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ushort | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| int | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| uint | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| long | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| ulong | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| float | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| double | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| decimal | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

Tabela 3.1: Compatibilidade dos tipos simples

Para copiar o valor armazenado em uma variável para outra variável de um tipo incompatível, é necessário realizar uma operação de **casting**. Esse tipo de operação força a conversão dos valores. No exemplo abaixo, o valor da variável `a`, que é do tipo `long`, foi copiado para a variável `b`, que é do tipo `int`, com uma operação de casting.

```
1 long a = 19;
2 int b = (int)a;
```

Código C# 3.3: Casting

Operações de casting podem gerar resultados indesejados. Considere que uma variável do tipo `long` armazena o valor **3000000000**. Se uma operação de casting for aplicada para copiar esse valor para uma variável do tipo `int`, o valor obtido na cópia será **-1294967296**.

```
1 long a = 3000000000;
2 int b = (int)a; // b = -1294967296
```

Código C# 3.4: Valores indesejados com casting

Em geral, quando há o risco de obter valores indesejados, os compiladores exigem a operação de casting. Isso funciona como um alerta para o programador. Contudo, em alguns casos, mesmo com esse risco, os compiladores não exigem a operação de casting. No exemplo abaixo, a variável do tipo `long` armazena o valor **9223372036854775807**. Ao copiar o conteúdo dessa variável para uma variável do tipo `float`, o valor obtido é **92233720000000000000**.

```

1 long a = 9223372036854775807;
2 float b = a; // b = 922337200000000000000000

```

Código C# 3.5: Valores indesejados mesmo sem casting

O tipo `bool` não é compatível com os outros tipos simples. Além disso, não é possível utilizar operações de casting para realizar conversões de booleanos. No exemplo abaixo, a variável `a` é do tipo `int` e a variável `b` é do tipo `bool`. A tentativa de copiar o valor da variável `b` para a variável `c`, que é do tipo `int`, com casting gera erro de compilação. Analogamente, a tentativa de copiar o valor da variável `a` para variável `d`, que é do tipo `bool`, com casting também gera erro de compilação.

```

1 int a = 10;
2 bool b = true;
3
4 int c = (int)b; // erro de compilação
5
6 bool d = (bool)a; // erro de compilação

```

Código C# 3.6: Conversão de booleanos



Curiosidade

No dia 6 de Junho de 1996, a Agência Espacial Europeia lançava o voo 501 do foguete Ariane 5. O objetivo da missão era transportar satélites para o espaço. Esses satélites seriam utilizados para analisar o impacto da atividade solar sobre a atmosfera terrestre. O desenvolvimento desse projeto levou uma década e custou cerca de 7 bilhões de dólares.

O lançamento do foguete, no entanto, não ocorreu como o esperado. Em menos de um minuto após o início do lançamento, o foguete desviou-se de sua trajetória e explodiu, causando um prejuízo direto de cerca de 360 milhões de dólares.

A explosão foi causada por uma falha no software que controlava o foguete. Essa falha ocorreu devido a uma operação de conversão de um valor que estava no formato ponto flutuante de 64 bits para o formato inteiro de 16 bits. Como o valor era superior a 32767, ele não poderia ser representado no formato inteiro de 16 bits. Dessa forma, o valor obtido na conversão desencadeou uma falha no sistema de controle do foguete, levando à sua explosão.

Fazendo um paralelo com a linguagem C#, esse problema poderia ocorrer na conversão de um valor do tipo `double` para um valor do tipo `short`.



3.3 Conversão Entre Tipos Simples e String

Considere uma variável do tipo `string` associada à sequência de caracteres “19”. Não podemos copiar o valor dessa variável para uma variável do tipo `int`, pois um erro de compilação seria gerado.

```

1 string a = "19";
2 int b = a;

```

Código C# 3.7: Erro de compilação - Incompatibilidade

Nesses casos, é necessário realizar uma conversão de `string` para `int`. Em C#, essa conversão pode ser realizada com o uso do método `ToInt32` da classe `System.Convert`. Confira o exemplo abaixo.

```
1 string a = "19";
2 int b = System.Convert.ToInt32(a);
```

Código C# 3.8: Conversão de string para int

A Tabela 3.2 a seguir mostra os métodos utilizados para converter valores do tipo `string` para valores simples.

| De <code>string</code> para | Método |
|-----------------------------|----------------------------------|
| <code>byte</code> | <code>Convert.ToByte()</code> |
| <code>sbyte</code> | <code>Convert.ToSByte()</code> |
| <code>short</code> | <code>Convert.ToInt16()</code> |
| <code>ushort</code> | <code>Convert.ToUInt16()</code> |
| <code>int</code> | <code>Convert.ToInt32()</code> |
| <code>uint</code> | <code>Convert.ToUInt32()</code> |
| <code>long</code> | <code>Convert.ToInt64()</code> |
| <code>ulong</code> | <code>Convert.ToUInt64()</code> |
| <code>float</code> | <code>Convert.ToSingle()</code> |
| <code>double</code> | <code>Convert.ToDouble()</code> |
| <code>decimal</code> | <code>Convert.ToDecimal()</code> |
| <code>bool</code> | <code>Convert.ToBoolean()</code> |

Tabela 3.2: Conversão

O método `ToBoolean()` devolve `true` se a `string` passada como parâmetro for igual a “true” independentemente das letras serem maiúsculas ou minúsculas (por exemplo, “TRUE”, “tRuE”, etc). Caso contrário, esse método devolve `false` se a `string` passada como parâmetro for igual a “false” independentemente das letras serem maiúsculas ou minúsculas (por exemplo, “FALSE”, “fAlSe”, etc). Caso contrário, um erro de execução é gerado.

Para os demais métodos, se o argumento não respeitar as regras estabelecidas na documentação da plataforma .NET, um erro de execução será gerado. O código abaixo exemplifica casos que provocam erros de execução.

```
1 // FormatException
2 System.Convert.ToByte("");
3 System.Convert.ToInt16("abc");
4 System.Convert.ToInt32("18 + 1");
5 System.Convert.ToDouble("K19");
```

Código C# 3.9: Erros de conversão



Simulação

Veremos, a seguir, a simulação de um programa em C# para ilustrar alguns tipos de conversões.

- 1 Ao executar a primeira linha do método `Main`, uma variável do tipo `string` chamada `s` será criada e associada à sequência de caracteres “3.14”.

```

1 class Conversao
2 {
3     static void Main()
4     {
5         string s = "3.14";
6         double d = System.Convert.ToDouble(s);
7         int i = (int)d;
8         System.Console.WriteLine(i);
9     }
10 }
```

`s = "3.14"`

- 2 Em seguida, na execução da linha 6, o método `ToDouble` da classe `System.Convert` converte a `string` “3.14” para um valor do tipo `double`. Esse valor é armazenado na variável `d`.

```

1 class Conversao
2 {
3     static void Main()
4     {
5         string s = "3.14";
6         double d = System.Convert.ToDouble(s);
7         int i = (int)d;
8         System.Console.WriteLine(i);
9     }
10 }
```

`s = "3.14"`

`d = 3.14`

- 3 Na execução da linha 7, o valor do tipo `double` armazenado na variável `d` é convertido para o tipo `int` através de uma operação de casting. O valor obtido nessa conversão é armazenado na variável `i`. Como o tipo `int` não permite o armazenamento de casas decimais, o resultado da conversão é o valor inteiro 3.

```

1 class Conversao
2 {
3     static void Main()
4     {
5         string s = "3.14";
6         double d = System.Convert.ToDouble(s);
7         int i = (int)d;
8         System.Console.WriteLine(i);
9     }
10 }
```

`s = "3.14"`

`d = 3.14`

`i = 3`

- 4 Ao executar a linha 8, o valor armazenado na variável `i` é exibido na saída padrão.

```

1 class Conversao
2 {
3     static void Main()
4     {
5         string s = "3.14";
6         double d = System.Convert.ToDouble(s);
7         int i = (int)d;
8         System.Console.WriteLine(i);
9     }
10 }
```

● 3

s = "3.14"

d = 3.14

i = 3

3.4 Operadores Aritméticos



Os operadores aritméticos funcionam de forma muito semelhante aos operadores da Matemática. Os operadores aritméticos são:

- + (adição)
- - (subtração)
- * (multiplicação)
- / (divisão)
- % (módulo)

Nesta seção, vamos considerar a utilização dos operadores aritméticos apenas com valores dos tipos simples. Veja no código a seguir alguns exemplos de uso desses operadores.

```

1 System.Console.WriteLine(1 + 2); // exibe: 3
2 System.Console.WriteLine(3 - 1); // exibe: 2
3 System.Console.WriteLine(2 * 3); // exibe: 6
4 System.Console.WriteLine(10 / 2); // exibe: 5
5 System.Console.WriteLine(10 % 4); // exibe: 2
```

Código C# 3.14: Exemplo de uso dos operadores aritméticos



Armadilha

Como a precisão dos tipos `float` e `double` é limitada, algumas operações aritméticas podem gerar valores incorretos. Por exemplo, o resultado da operação “`0.9F - 0.8F`” deveria ser `0.1`. Contudo, na linguagem C#, nem o valor `0.9` e nem o valor `0.8` podem ser representados em `float` ou `double`. Assim, valores aproximados são utilizados. O número `0.9` é substituído pelo valor `0.89999997615814208984375` e o número `0.8` é substituído pelo valor `0.800000011920928955078125`. Portanto, o resultado da operação não é `0.1`, mas `0.09999964237213134765625`. O método `WriteLine` exibe esse resultado arredondado.

```
1 System.Console.WriteLine(0.9F - 0.8F); // exibe: 0.099999964237213134765625
```



Mais Sobre

As operações de potenciação, raiz quadrada e valor absoluto podem ser realizadas através dos métodos `System.Math.Pow`, `System.Math.Sqrt` e `System.Math.Abs`.

```

1 System.Console.WriteLine(System.Math.Pow(2, 5)); // exibe 32
2 System.Console.WriteLine(System.Math.Sqrt(9)); // exibe 3
3 System.Console.WriteLine(System.Math.Abs(-19.3)); // exibe 19.3
4
5

```

Código C# 3.16: Potenciação, raiz quadrada e valor absoluto

O método `Sqrt` devolve `NaN` quando é aplicado a valores negativos.

Módulo

Em Matemática, o módulo de um número x é o valor numérico de x desconsiderando o seu sinal (valor absoluto). Por exemplo, o módulo de -2 é 2 e expressamos esse valor da seguinte forma: $|-2| = 2$. Em C#, a palavra módulo tem outro significado. Ela indica o resto da divisão de um número por outro.

Podemos calcular o resto da divisão de um número por outro através do operador `%`. Por exemplo, o resto da divisão do número 6 pelo número 5 é 1 . Para calculá-lo, utilizamos a operação $6 \% 5$.

Nas operações de módulo, o sinal do resultado é igual ao sinal do dividendo. Veja alguns exemplos abaixo.

```

1 System.Console.WriteLine(6 % 5); // exibe: 1
2 System.Console.WriteLine(6 % -5); // exibe: 1
3 System.Console.WriteLine(-6 % 5); // exibe: -1
4 System.Console.WriteLine(-6 % -5); // exibe: -1

```

Divisão por 0

Nas operações de divisão ou de módulo envolvendo números inteiros, se o divisor for uma variável que armazena o valor 0 , o erro de execução `System.DivideByZeroException` ocorrerá.

```

1 int a = 0;
2 int b = 10 / a; // Ocorrerá DivideByZeroException

```

Agora, se o divisor for a constante 0 , um erro de compilação ocorrerá.

```

1 const a = 0;
2 int b = 10 / a; // Ocorrerá um erro de compilação

```

`error CS0020: Division by constant zero`

Por outro lado, nas operações de divisão ou módulo envolvendo pelo menos um número real, o resultado será `Infinity`, `-Infinity` ou `NaN`.

```

1 double b = 10.0 / 0; // O resultado será Infinity
2 double c = -10.0 / 0; // O resultado será -Infinity
3 double d = 0.0 / 0; // O resultado será NaN

```

3.5 Tipo do Resultado de uma Operação Aritmética



No exemplo abaixo, a variável `b1` foi inicializada com o valor 1, a variável `b2` foi inicializada com o valor 2 e a variável `b3` foi inicializada com resultado da operação `b1 + b2`. Contudo, na linguagem C#, operações aritméticas envolvendo valores do tipo `byte` devolvem valores do tipo `int`. Dessa forma, o resultado da operação `b1 + b2` não pode ser armazenado diretamente na variável `b3`.

```

1 byte b1 = 1;
2
3 byte b2 = 2;
4
5 byte b3 = b1 + b2; // erro de compilação

```

Código C# 3.21: Resultado das operações aritméticas

Neste caso, poderíamos aplicar uma operação de casting no resultado da operação `b1 + b2`.

```

1 byte b1 = 1;
2
3 byte b2 = 2;
4
5 byte b3 = (byte)(b1 + b2);

```

Código C# 3.22: Utilizando casting

Para saber o tipo do resultado de uma operação aritmética, devemos aplicar as seguintes regras:

- Se pelo menos um dos operandos for do tipo `decimal`, o resultado será do tipo `decimal`.
- Caso contrário, se pelo menos um dos operandos for do tipo `double`, o resultado será do tipo `double`.
- Caso contrário, se pelo menos um dos operandos for do tipo `float`, o resultado será do tipo `float`.
- Caso contrário, se pelo menos um dos operandos for do tipo `ulong`, o resultado será do tipo `ulong`.
- Caso contrário, se pelo menos um dos operandos for do tipo `long`, o resultado será do tipo `long`.
- Caso contrário, se um dos operandos for do tipo `uint` e o outro do tipo `sbyte`, `short` ou `int`, o resultado será do tipo `long`.
- Caso contrário, se pelo menos um dos operandos for do tipo `uint`, o resultado será do tipo `uint`.
- Caso contrário, o resultado será do tipo `int`.

De acordo com as regras acima, no exemplo a seguir, o resultado da operação `a + b` é um valor do tipo `int`. O resultado da operação `b + c` é um valor do tipo `double`. Por fim, o resultado da operação `a + c` também é um valor do tipo `double`.

```

1 byte a = 1;
2

```

```

3 short b = 2;
4
5 double c = 3.14;
6
7 int resultado1 = a + b;
8
9 double resultado2 = b + c;
10
11 double resultado3 = a + c;

```

Código C# 3.23: Determinando o tipo do resultado de uma operação aritmética



Importante

Não são permitidas operações binárias envolvendo um valor do tipo decimal e um valor do tipo double ou float. Também não são permitidas operações binárias envolvendo um valor do tipo ulong e um valor do tipo sbyte, short, int ou long.

3.6 Divisão Inteira



Considere uma operação de divisão entre valores inteiros. Por exemplo, uma divisão entre valores do tipo int.

```

1 int a = 5;
2 int b = 2;
3 System.Console.WriteLine(a / b); // exibe: 2

```

Código C# 3.24: Divisão inteira

Matematicamente, o resultado da operação $5/2$ é 2.5 . Contudo, no exemplo acima, o valor obtido na divisão a / b é 2. Quando ocorre uma divisão entre dois valores inteiros, a parte fracionária do resultado é descartada.

Podemos, explicitamente, converter um dos valores envolvidos na divisão ou até mesmo os dois para algum tipo que aceita números reais. Dessa forma, a divisão não seria inteira e a parte fracionária não seria descartada. Essas conversões podem ser realizadas com operações de casting. No exemplo abaixo, o resultado de cada uma das operações de divisão é 2.5. Lembre-se que as operações de casting são realizadas antes das operações aritméticas.

```

1 int a = 5;
2 int b = 2;
3
4 // convertendo o valor armazenado na variável "a"
5 System.Console.WriteLine((double)a / b); // exibe: 2.5
6
7 // convertendo o valor armazenado na variável "b"
8 System.Console.WriteLine(a / (double)b); // exibe: 2.5
9
10 // convertendo os valores armazenados nas variáveis "a" e "b"
11 System.Console.WriteLine((double)a / (double)b); // exibe: 2.5

```

Código C# 3.25: Castings



Pare para pensar...

Considerando o que foi discutido anteriormente a respeito de divisão inteira e casting,

qual é o resultado da operação do exemplo a seguir?

```
1 double d = (double)(5 / 2);
```



Simulação

Nessa simulação, realizaremos operações aritméticas e operações de casting. Além disso, mostraremos as diferenças entre divisão inteira e divisão real.

- 1 Ao executar o programa, a primeira linha do método Main será processada. O valor 0 será armazenado na variável a.

```
1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }
```

a = 0

- 2 Na sequência, a linha 6 será executada. O operador % calculará o resto da divisão entre os números 13 e 4. O resultado dessa operação é 1.

```
1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }
```

a = 0

- 3 O fluxo de execução prossegue e a linha 7 será executada. A divisão entre os números inteiros 10 e 4 será realizada com a utilização do operador /. Como os dois operandos dessa operação são valores inteiros, ocorre uma divisão inteira. Por isso, o resultado será 2 e não 2.5.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }
```

1
2

a = 0

- 4 Em seguida, a linha 8 é executada. Assim como na linha 7, o operador / será utilizado para calcular a divisão entre os números 10 e 4. Contudo, uma operação de casting converte o literal 10, que é do tipo int, para double antes da divisão. Como pelo menos um dos operandos é um valor real, ocorrerá uma divisão real. Sendo assim, o resultado terá casas decimais e será exibido o número 2.5 na saída padrão.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }
```

1
2
2.5

a = 0

- 5 Na sequência, a linha 9 é executada. Assim como nas linhas 7 e 8, o operador / será utilizado para calcular a divisão entre os números 10 e 4. Antes da divisão, uma operação de casting converte o literal 4, que é do tipo int, para double. Como pelo menos um dos operandos é um valor real, ocorrerá uma divisão real. Sendo assim, o resultado será 2.5.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }

```

1
2
2.5
2.5

a = 0

- 6 Continuando, a linha 10 é executada. Assim como nas linhas 7, 8 e 9, o operador / será utilizado para calcular a divisão entre os números 10 e 4. Note que essa operação foi delimitada com parênteses. Sendo assim, a divisão será executada antes da operação de casting. Como os dois operandos dessa divisão são números inteiros, o resultado não terá casas decimais. Sendo assim, esse resultado será 2. Em seguida, o casting transformará esse valor inteiro em real. Consequentemente, será exibido o número 2 na saída padrão.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }

```

1
2
2.5
2.5
2

a = 0

- 7 A próxima linha que será executada é a 11. Nessa linha, a divisão entre o literal 10.0 que é do tipo double e o valor armazenado na variável a que é 0 é realizada com a utilização do operador /. Como pelo menos um dos operandos é um valor real, ocorrerá a divisão real. Sendo assim, o resultado dessa operação é Infinity.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }

```

1
2
2.5
2.5
2
● Infinity

a = 0

- 8 A próxima linha que será executada é a 12. Nessa linha, a divisão entre o literal 0.0 que é do tipo double e o valor armazenado na variável a que é 0 é realizada com a utilização do operador /. Como pelo menos um dos operandos é um valor real, ocorrerá a divisão real. Sendo assim, o resultado dessa operação é NaN.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }

```

1
2
2.5
2.5
2
● Infinity
NaN

a = 0

- 9 Continuando a execução, a linha 13 é executada e a divisão do literal 0 que é do tipo int pelo valor armazenado na variável a que é 0 é calculada com o operador /. Essa operação é considerada uma divisão inteira porque os dois operandos são valores inteiros. Uma divisão inteira por 0 gera um erro de execução, o DivideByZeroException.

```

1 class Operacoes
2 {
3     static void Main()
4     {
5         int a = 0;
6         System.Console.WriteLine(13 % 4);
7         System.Console.WriteLine(10 / 4);
8         System.Console.WriteLine((double)10 / 4);
9         System.Console.WriteLine(10 / (double)4);
10        System.Console.WriteLine((double)(10 / 4));
11        System.Console.WriteLine(10.0 / a);
12        System.Console.WriteLine(0.0 / a);
13        System.Console.WriteLine(0 / a);
14    }
15 }

```

1
2
2.5
2.5
2
Infinity
NaN
● Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
at Programa.Main()

a = 0

3.7 Overflow e Underflow



Considere duas variáveis do tipo `int` chamadas `a` e `b`. A variável `a` armazena o maior valor do tipo `int`, ou seja, armazena o valor 2147483647. A variável `b` armazena o menor valor do tipo `int`, ou seja, armazena o valor -2147483648.

Em C#, o resultado da operação `a + 1` é do tipo `int` porque os dois operandos são do tipo `int`. Matematicamente, o resultado dessa operação é 2147483648. Contudo, o maior valor aceito pelo tipo `int` é 2147483647. Nesse caso, a operação `a + b` gera um **overflow** e o valor obtido é -2147483648. Analogamente, a operação `b - 1` causa um **underflow** e o seu resultado é 2147483647.

```

1 System.Console.WriteLine(a + 1); // overflow: -2147483648
2 System.Console.WriteLine(b - 1); // underflow: 2147483647

```

Lembrando que o menor valor aceito pelo tipo `long` é `long.MinValue` e o maior é `long.MaxValue`, as operações do exemplo abaixo causam underflow e overflow.

```

1 int a = 1;
2 System.Console.WriteLine(long.MinValue - a); // underflow: 9223372036854775807
3 System.Console.WriteLine(long.MaxValue + a); // overflow: -9223372036854775808

```

Nas operações aritméticas envolvendo valores do tipo `float` ou `double`, também pode ocorrer overflow ou underflow. No overflow, o resultado pode ser $\pm\infty$. No underflow, o resultado será 0.

```

1 int a = 2;
2 System.Console.WriteLine(float.MaxValue * a); // overflow: Infinity
3 System.Console.WriteLine(float.MinValue * a); // overflow: -Infinity
4 System.Console.WriteLine(float.Epsilon / a); // underflow: 0.0
5
6 System.Console.WriteLine(double.MaxValue * a); // overflow: Infinity
7 System.Console.WriteLine(double.MinValue * a); // overflow: -Infinity
8 System.Console.WriteLine(double.Epsilon / a); // underflow: 0.0

```



Simulação

Nessa simulação, mostraremos um exemplo de overflow e de underflow.

- 1 Ao executar o programa, a primeira linha do método Main será processada. Nessa linha, é declarada a variável a do tipo int e inicializada com o maior valor do tipo int.

```

1 class OverflowUnderflow
2 {
3     static void Main()
4     {
5         int a = int.MaxValue;
6         int b = int.MinValue;
7         System.Console.WriteLine(a + 1);
8         System.Console.WriteLine(b - 1);
9     }
10 }
```

a = 2147483647

- 2 Na linha 6, é declarada a variável b do tipo int e inicializada com o menor valor do tipo int.

```

1 class OverflowUnderflow
2 {
3     static void Main()
4     {
5         int a = int.MaxValue;
6         int b = int.MinValue;
7         System.Console.WriteLine(a + 1);
8         System.Console.WriteLine(b - 1);
9     }
10 }
```

a = 2147483647

b = -2147483648

- 3 Na linha 7, é exibido o valor de a + 1. Como o resultado dessa expressão ultrapassa o valor máximo do tipo int, ocorre um overflow e o resultado dessa operação será -2147483648.

```

1 class OverflowUnderflow
2 {
3     static void Main()
4     {
5         int a = int.MaxValue;
6         int b = int.MinValue;
7         System.Console.WriteLine(a + 1);
8         System.Console.WriteLine(b - 1);
9     }
10 }
```

a = 2147483647

b = -2147483648

- 4 Na linha 8, é exibido o valor de b - 1. Como o resultado dessa operação é menor do que o limite mínimo do tipo int, ocorre um underflow e o resultado dessa operação será 2147483647.

```

1 class OverflowUnderflow
2 {
3     static void Main()
4     {
5         int a = int.MaxValue;
6         int b = int.MinValue;
7         System.Console.WriteLine(a + 1);
8         System.Console.WriteLine(b - 1);
9     }
10}

```

-2147483648
2147483647

a = 2147483647

b = -2147483648

3.8 Regras para Operações Aritméticas com Valores Especiais



Os resultados das operações aritméticas com `float` ou `double` envolvendo os valores `±Infinity`, `±0` e `NaN` seguem as seguintes regras:

- Se pelo menos um dos operandos for `NaN`, o resultado será `NaN`.
- A soma de infinitos de sinais opostos é `NaN`.
- A soma de `+Infinity` com `+Infinity` é `+Infinity`, assim como a soma de `-Infinity` com `-Infinity` é `-Infinity`.
- A soma de `+Infinity` com um valor finito é `+Infinity`, assim como a soma de `-Infinity` com um valor finito é `-Infinity`.
- A soma de zeros de sinais opostos é `+0`.
- A soma de `+0` com `+0` é `+0`, assim como a soma de `-0` com `-0` é `-0`.
- Tanto na multiplicação quanto na divisão, o sinal do resultado é determinado como na Matemática.
- Multiplicar `±Infinity` por `0` resulta em `NaN`.
- Multiplicar `±Infinity` por `±Infinity` resulta em `±Infinity`.
- Dividir `±Infinity` por `±Infinity` resulta em `NaN`.
- Dividir `±Infinity` por um valor finito resulta em `±Infinity`.
- Dividir um valor finito por `±Infinity` resulta em `±0`.
- Dividir `±0` por `±0` resulta `NaN`.
- Dividir `±0` por um valor finito diferente de `± 0` resulta em `± 0`.
- Dividir um valor finito diferente de `± 0` por `± 0` resulta em `±Infinity`.
- Nas operações de módulo, o sinal do resultado é o sinal do dividendo.
- Nas operações de módulo, se o dividendo for `±Infinity` ou o divisor for `±0`, o resultado é `NaN`.
- Nas operações de módulo, se o dividendo for `±Infinity` e o divisor for `±Infinity`, o resultado é `NaN`.
- Nas operações de módulo, se o dividendo for `±0` e o divisor finito, o resultado é igual ao dividendo.

3.9 Concatenação de Strings



Como vimos anteriormente, o operador + é utilizado para realizar a operação aritmética de adição. Mas, ele também pode ser utilizado para concatenar strings.

```

1 string s1 = "Marcelo";
2 string s2 = " ";
3 string s3 = "Martins";
4
5 // "Marcelo Martins"
6 string s4 = s1 + s2 + s3;

```

No exemplo abaixo, o operador + foi aplicado a valores do tipo `int` e do tipo `string`. Nesse caso, o valor do tipo `int` é automaticamente convertido para `string` e a concatenação é realizada. De maneira geral, quando o operador + for aplicado a um valor do tipo `string` e outro diferente de `string`, esse último será convertido para `string` e a concatenação será realizada.

```

1 string s1 = "Idade: ";
2 int idade = 30;
3
4 // "Idade: 30"
5 string s2 = s1 + idade;

```



Pare para pensar...

As expressões são avaliadas da esquerda para a direita. Dessa forma, considere o seguinte trecho de código:

```

1 System.Console.WriteLine(1 + 2 + 3 + " testando");
2 System.Console.WriteLine("testando" + 1 + 2 + 3);

```

O que seria exibido nesse caso?



Pare para pensar...

Como vimos, quando um dos operandos do operador + é do tipo `string`, a operação que será realizada é a de concatenação. Dessa forma, o que seria exibido na saída padrão com o código abaixo?

```
1 System.Console.WriteLine('a' + 'a');
```

Simulação



Nessa simulação, realizaremos operações de concatenação.

- 1 Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `s` do tipo `string` será associada à string “K”.

```

1 class Concatenacoes
2 {
3     static void Main()
4     {
5         string s = "K";
6         s = "Cursos " + s;
7         s = s + 19;
8         System.Console.WriteLine(s);
9     }
10 }
```

s = "K"

- 2 Em seguida, a linha 6 será executada e a string “Cursos” será concatenada à esquerda da string “K”. Essa concatenação produz a string “Cursos K”. Essa string é associada à variável s através do operador =.

```

1 class Concatenacoes
2 {
3     static void Main()
4     {
5         string s = "K";
6         s = "Cursos " + s;
7         s = s + 19;
8         System.Console.WriteLine(s);
9     }
10 }
```

s = "Cursos K"

- 3 Na sequência, a linha 7 é executada. O literal 19, que é do tipo int, é convertido automaticamente para string. Em seguida, a string “19” é concatenada à direita da string “Cursos K”. Essa concatenação produz a string “Cursos K19”. Essa string é associada à variável s através do operador =.

```

1 class Concatenacoes
2 {
3     static void Main()
4     {
5         string s = "K";
6         s = "Cursos " + s;
7         s = s + 19;
8         System.Console.WriteLine(s);
9     }
10 }
```

s = "Cursos K19"

- 4 Por fim, a linha 8 será executada e a mensagem “Cursos K19” será exibida na saída padrão.

```

1 class Concatenacoes
2 {
3     static void Main()
4     {
5         string s = "K";
6         s = "Cursos " + s;
7         s = s + 19;
8         System.Console.WriteLine(s);
9     }
10 }
```

s = "Cursos K19"

● Cursos K19



3.10 Operadores Unários + e -

Nesta seção, vamos considerar a utilização dos operadores unários apenas com valores dos tipos simples. O operador unário + preserva o sinal dos valores numéricos enquanto o operador unário - inverte o sinal desses valores. O resultado gerado por esses operadores segue as seguintes regras.

- O resultado é do tipo `int` quando o operando é do tipo `sbyte`, `byte`, `short`, `ushort`, `char` ou `int`.
- O resultado é do tipo `long` quando o operando é do tipo `uint` ou `long`.
- O resultado é do tipo `ulong` quando o operando é do tipo `ulong`. O operador unário - não pode ser aplicado a valores do tipo `ulong`.
- O resultado é do tipo `float` quando o operando é do tipo `float`.
- O resultado é do tipo `double` quando o operando é do tipo `double`.
- O resultado é do tipo `decimal` quando o operando é do tipo `decimal`.

```
1 System.Console.WriteLine(+1); // exibe: 1
2 System.Console.WriteLine(-1); // exibe: -1
```



3.11 Operadores de Atribuição

Um operador de atribuição altera o valor armazenado em uma variável. Nesta seção, vamos considerar a utilização dos operadores de atribuição com valores dos tipos simples.

- `=` (atribuição simples)
- `+=` (incremental)
- `-=` (decremental)
- `*=` (multiplicativa)
- `/=` (divisória)
- `%=` (modular)
- `++` (incremento)
- `--` (decremento)

Operador de atribuição simples

O operador `=` é chamado de operador de atribuição simples. Esse operador armazena o valor da expressão à sua direita na variável à sua esquerda. O resultado devolvido por essa operação é o valor armazenado na variável.

Essa operação só é permitida se o tipo do valor da expressão à direita é compatível com o tipo da variável à esquerda (ver Tabela 3.1).

No exemplo abaixo, são declaradas duas variáveis do tipo `int`. Na primeira linha, o operador `=` é utilizado para armazenar o valor 1 na variável `a`. Na segunda linha, esse operador é utilizado para armazenar em `b` uma cópia do valor armazenado na variável `a`.

```
1 int a = 1;
2 int b = a;
```

Operadores de atribuição compostos

Os operadores de atribuição `+=`, `-=`, `*=`, `/=`, `%=`, `++` e `--` são chamados de operadores compostos, pois além de modificar o valor de uma variável, eles realizam a operação aritmética correspondente. Por exemplo, a operação `a += b` faz com que a variável `a` passe a armazenar o valor de `a + b`.

Confira, no código abaixo, exemplos de utilização desses operadores.

```
1 int valor = 1;
2
3 valor += 2; // valor = 3
4
5 valor -= 1; // valor = 2
6
7 valor *= 6; // valor = 12
8
9 valor /= 3; // valor = 4
10
11 valor %= 3; // valor = 1
12
13 valor++; // valor = 2
14
15 valor--; // valor = 1
```

Código C# 3.53: Exemplo de uso dos operadores de atribuição

Nas instruções com os operadores de atribuição compostos, há uma operação de casting implícita. Assim, as instruções do exemplo acima são equivalentes às instruções do código abaixo.

```
1 int valor = 1;
2
3 valor = (int)(valor + 2); // valor = 3
4
5 valor = (int)(valor - 1); // valor = 2
6
7 valor = (int)(valor * 6); // valor = 12
8
9 valor = (int)(valor / 3); // valor = 4
10
11 valor = (int)(valor % 3); // valor = 1
12
13 valor = (int)(valor + 1); // valor = 2
14
15 valor = (int)(valor - 1); // valor = 1
```

Código C# 3.54: Utilizando os operadores aritméticos

Para utilizar os operadores de atribuição compostos, a operação de atribuição simples e a operação aritmética correspondente devem ser válidas se aplicadas individualmente aos mesmos operandos. No exemplo abaixo, a variável `a` é do tipo `int` e a variável `b` é do tipo `short`. Para determinar se a operação `a += b` é válida, devemos verificar se as operações `a = b` e `a + b` são permitidas. Como vimos na Seção 3, a operação `a = b` é válida. Também vimos na Seção 3 que a operação `a + b` é válida. Portanto, a operação `a += b` também é válida.

```
1 int a = 1;
```

```
2 short b = 1;
3 a += b; // é válida, pois a = b é válida e a + b é válida
```

Agora, no exemplo a seguir, a operação `a = b` é inválida. Portanto, a operação `a += b` também é inválida.

```
1 ulong a = 1;
2 short b = 1;
3 a += b; // é inválida, pois a = b é inválida
```

Os operadores de atribuição compostos reduzem a quantidade de código escrito. Eles funcionam como “atalhos” para realizar operações aritméticas em conjunto com operações de atribuição.



Mais Sobre

Qual é o resultado de uma operação de atribuição? O resultado de uma operação de atribuição é o valor do segundo operando. No exemplo abaixo, a operação `a = 1` devolve o valor 1, que é do tipo `int`.

```
1 int a;
2
3 System.Console.WriteLine(a = 1); // exibe: 1
```

Código C# 3.57: Utilizando os operadores aritméticos



Simulação

Nessa simulação, mostraremos um exemplo de utilização dos operadores de atribuição.

- 1 Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `a` do tipo `int` é declarada e inicializada com o valor 7.

```
1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 7;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10        System.Console.WriteLine(a);
11    }
12 }
```

a = 7

- 2 Em seguida, a linha 6 é executada. Utilizando o operador `+=`, o valor 3 é adicionado ao valor da variável `a`. O resultado dessa operação é armazenado na própria variável `a`. Dessa forma, depois da execução dessa linha, o valor contido nessa variável será 10.

```

1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 7;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10        System.Console.WriteLine(a);
11    }
12 }
```

a = 10

- 3 Na sequência, a linha 7 é executada. Utilizando o operador `*=`, o valor armazenado na variável `a` é multiplicado por 2. O resultado dessa operação é armazenado na própria variável `a`. Dessa forma, após a execução dessa linha, o valor contido nessa variável será 20.

```

1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 7;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10        System.Console.WriteLine(a);
11    }
12 }
```

a = 20

- 4 Agora, a linha 8 é executada. Utilizando o operador `++`, o valor 1 é adicionado ao valor armazenado na variável `a`. O resultado dessa operação é armazenado na própria variável `a`. Dessa forma, depois dessa linha, o valor contido nessa variável será 21.

```

1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 7;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10        System.Console.WriteLine(a);
11    }
12 }
```

a = 21

- 5 Continuando a execução, a linha 9 será processada. Como a variável `a` armazena o valor 21, a operação `a -= 2` faz com que a variável `a` passe a armazenar o valor 19.

```

1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 7;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10    System.Console.WriteLine(a);
11 }
12 }
```

a = 19

- 6 Por fim, a linha 10 é executada e o valor da variável a é exibido na saída padrão.

```

1 class Atribuicao
2 {
3     static void Main()
4     {
5         int a = 5;
6         a += 3;
7         a *= 2;
8         a++;
9         a -= 2;
10    System.Console.WriteLine(a);
11 }
12 }
```

a = 19

3.12 Operadores de Comparação



Muitas vezes precisamos determinar a equivalência entre dois valores ou a relação de grandeza (se é maior ou menor) entre eles. Nessas situações, utilizamos os operadores de comparação (também chamados de operadores relacionais). As operações realizadas com os operadores relacionais devolvem valores do tipo `bool`. Os operadores relacionais são:

- `==` (igualdade)
- `!=` (desigualdade)
- `<` (menor)
- `<=` (menor ou igual)
- `>` (maior)
- `>=` (maior ou igual)

Nesta seção, vamos considerar a utilização dos operadores de comparação com valores dos tipos simples. Veja, no código abaixo, alguns exemplos de utilização desses operadores.

```

1 int valor = 2;
2 bool b;
3 b = (valor == 2); // b = true
4 b = (valor != 2); // b = false
5 b = (valor < 2); // b = false
6 b = (valor <= 2); // b = true
```

```
7 b = (valor > 1); // b = true
8 b = (valor >= 1); // b = true
```

Código C# 3.64: Exemplo de uso dos operadores relacionais em C#

Não é permitida a utilização desses operadores para comparar valores dos tipos simples numéricos com valores do tipo `bool`. Além disso, não é possível usar esses operadores quando um operando é do tipo `long` e o outro é do tipo `ulong`. Uma última restrição é a de que os operadores que envolvem ordem (`<`, `<=`, `>` e `>=`) não podem ser aplicados a operandos do tipo `bool`.



Mais Sobre

Se um dos operandos do operador `!=` for `NaN` então o resultado será `true`. Para os demais operadores, se um dos operandos for `NaN` então o resultado será `false`.

Simulação



Nessa simulação, mostraremos um exemplo de utilização dos operadores de comparação.

- Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `a` do tipo `int` é declarada e inicializada com o valor 1.

```
1 class Comparacoes
2 {
3     static void Main()
4     {
5         int a = 1;
6         System.Console.WriteLine(a == 1);
7         System.Console.WriteLine(a > 5);
8         System.Console.WriteLine(a <= 3);
9     }
10 }
```

a = 1

- Na sequência, a linha 6 será executada. O operador `==` foi utilizado para verificar se o valor armazenado na variável `a` é igual a 1. Como a variável `a` armazena o valor 1, essa operação devolve `true`.

```
1 class Comparacoes
2 {
3     static void Main()
4     {
5         int a = 1;
6         System.Console.WriteLine(a == 1);
7         System.Console.WriteLine(a > 5);
8         System.Console.WriteLine(a <= 3);
9     }
10 }
```

a = 1

True

- Na sequência, a linha 7 será executada. O operador `>` foi utilizado para verificar se o valor armazenado na variável `a` é maior do que 5. Como a variável `a` armazena o valor 1, essa operação devolve

false.

```

1 class Comparacoes
2 {
3     static void Main()
4     {
5         int a = 1;
6         System.Console.WriteLine(a == 1);
7         System.Console.WriteLine(a > 5);
8         System.Console.WriteLine(a <= 3);
9     }
10 }
```

a = 1

True
False

- 4 Em seguida, a linha 8 será executada. O operador `<=` foi utilizado para verificar se o valor armazenado na variável `a` é menor ou igual a 3. Como a variável `a` armazena o valor 1, essa operação devolve `true`.

```

1 class Comparacoes
2 {
3     static void Main()
4     {
5         int a = 1;
6         System.Console.WriteLine(a == 1);
7         System.Console.WriteLine(a > 5);
8         System.Console.WriteLine(a <= 3);
9     }
10 }
```

a = 1

True
False
True

3.13 Operadores Lógicos



Nesta seção, apresentamos os operadores lógicos considerando suas aplicações apenas a operandos do tipo `bool`.

Considere um jogo de dados no qual o jogador faz dois lançamentos de dado. Se o resultado do primeiro lançamento for menor do que 3 e o resultado do segundo for maior do que 4, o jogador ganha. Caso contrário, ele perde. Suponha que o resultado do primeiro lançamento seja armazenado em uma variável chamada `a` e o resultado do segundo em uma variável chamada `b`.

Como verificar se o jogador ganhou? Podemos utilizar um operador lógico. Os operadores lógicos da linguagem C# são:

- `&` (E simples)
- `&&` (E duplo)
- `|` (OU simples)
- `||` (OU duplo)
- `^` (OU exclusivo)

Os operandos de uma operação lógica são valores do tipo `bool` e o resultado de uma operação lógica também é um valor do tipo `bool`.

- Os operadores & (E simples) e && (E duplo) devolvem true se e somente se as duas condições forem true.

```

1 System.Random gerador = new System.Random();
2 int a = gerador.Next(6) + 1;
3 int b = gerador.Next(6) + 1;
4
5 System.Console.WriteLine(a < 3 & b > 4);
6 System.Console.WriteLine(a < 3 && b > 4);

```

Código C# 3.69: Exemplo de uso dos operadores & e &&

A **tabela verdade** é uma forma prática de visualizar o resultado dos operadores lógicos. Veja a seguir a tabela verdade dos operadores & e &&.

| a < 3 | b > 4 | a < 3 & b > 4 | a < 3 && b > 4 |
|-------|-------|---------------|----------------|
| V | V | V | V |
| V | F | F | F |
| F | V | F | F |
| F | F | F | F |

Tabela 3.3: Tabela verdade dos operadores & e &&

- Considere agora uma mudança nas regras do jogo. Suponha que o jogador ganha se no primeiro lançamento o resultado for menor do que 3 ou no segundo lançamento o resultado for maior do que 4. Para verificar se o jogador ganha o jogo, podemos usar os operadores OU. Os operadores | (OU simples) e || (OU duplo) devolvem true se pelo menos uma das condições for true.

```

1 System.Random gerador = new System.Random();
2 int a = gerador.Next(6) + 1;
3 int b = gerador.Next(6) + 1;
4
5 System.Console.WriteLine(a < 3 | b > 4);
6 System.Console.WriteLine(a < 3 || b > 4);

```

Código C# 3.70: Exemplo de uso dos operadores | e ||

Também, podemos utilizar a tabela verdade para visualizar o resultado dos operadores | e ||.

| a < 3 | b > 4 | a < 3 b > 4 | a < 3 b > 4 |
|-------|-------|---------------|----------------|
| V | V | V | V |
| V | F | V | V |
| F | V | V | V |
| F | F | F | F |

Tabela 3.4: Tabela verdade dos operadores | e ||

- Agora, suponha que o jogador ganha o jogo se uma das situações abaixo ocorrer:
 - No primeiro lançamento, o resultado é menor do que 3 e, no segundo lançamento, o resultado **não** é maior do que 4.
 - No primeiro lançamento, o resultado **não** é menor do que 3 e, no segundo lançamento, o resultado é maior do que 4.

Nesse tipo de jogo, podemos utilizar o operador OU exclusivo para verificar se o jogador é vencedor. O operador ^ (OU exclusivo) devolve true se e somente se exatamente uma das condições for true (ou seja, uma delas deve ser true e a outra deve ser false).

```

1 System.Random gerador = new System.Random();
2 int a = gerador.Next(6) + 1;
3 int b = gerador.Next(6) + 1;
4
5 System.Console.WriteLine(a < 3 ^ b > 4);

```

Código C# 3.71: Exemplo de uso do operador ^

Vamos visualizar o resultado do operador ^ através da tabela verdade.

| a < 3 | b > 4 | a < 3 ^ b > 4 |
|-------|-------|---------------|
| V | V | F |
| V | F | V |
| F | V | V |
| F | F | F |

Tabela 3.5: Tabela verdade do operador ^

Os operadores & e && produzem o mesmo resultado. Então, qual é a diferença entre eles? O operador & sempre avalia as duas condições. Por outro lado, o operador && não avalia a segunda condição se o valor da primeira condição for false. De fato, esse comportamento é plausível, pois se o valor da primeira condição for false, o resultado da operação é false independentemente do valor da segunda condição. Dessa forma, podemos simplificar a tabela verdade do operador &&.

| a < 3 | b > 4 | a < 3 && b > 4 |
|-------|-------|----------------|
| V | V | V |
| V | F | F |
| F | ? | F |

Tabela 3.6: Tabela verdade do operador &&

Analogamente, podemos deduzir a diferença entre os operadores | e ||. As duas condições sempre são avaliadas quando utilizamos o operador |. Agora, quando utilizamos o operador ||, a segunda condição é avaliada se e somente se o valor da primeira condição for false. Realmente, esse comportamento é aceitável, pois o resultado da operação é true quando o valor da primeira condição for true independentemente do valor da segunda condição. Dessa forma, podemos simplificar a tabela verdade do operador ||.

| a < 3 | b > 4 | a < 3 b > 4 |
|-------|-------|----------------|
| V | ? | V |
| F | V | V |
| F | F | F |

Tabela 3.7: Tabela verdade do operador ||



Pare para pensar...

Considerando o comportamento dos operadores lógicos &, &&, | e ||, o que seria exibido com as seguintes instruções?

```

1 int i = 10;
2

```

```

3 System.Console.WriteLine(i > 100 & i++ < 500);
4 System.Console.WriteLine(i > 100 && i++ < 500);
5 System.Console.WriteLine(i > 0 | i++ < 500);
6 System.Console.WriteLine(i > 0 || i++ < 500);
7 System.Console.WriteLine(i);

```



Pare para pensar...

A linguagem C# possui os operadores lógicos `&` e `&&`. Também possui os operadores `|` e `||`. Agora, a pergunta que não quer calar: por quê não existe o operador `^^`?



Simulação

Nessa simulação, mostraremos um exemplo de utilização dos operadores lógicos.

- 1 Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `a` do tipo `int` é declarada e inicializada com o valor 2.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

`a = 2`

- 2 Na sequência, a linha 6 será executada e uma variável chamada `b` do tipo `int` é declarada e inicializada com o valor 10.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

`a = 2`

`b = 10`

- 3 Agora, na linha 7, a operação `a > 0` será executada. Como o valor armazenado na variável `a` é 2, o resultado dessa operação será `true`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

a = 2

b = 10

- 4 Em seguida, ainda na linha 7, o operador `||` devolverá `true` sem avaliar o lado direito (`b < 0`) pois, como vimos no passo anterior, o lado esquerdo (`a > 0`) devolveu `true`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

a = 2

b = 10

- 5 Para finalizar a linha 7, o resultado da expressão `a > 0 || b < 0`, que é `true`, será exibido na saída padrão.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

a = 2

b = 10

True

- 6 Em seguida, na linha 8, a operação `a == 2` será executada. Como a variável `a` armazena o valor 2, o resultado dessa operação é `true`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True

a = 2

b = 10

- 7 Na sequência, ainda na linha 8, como o lado esquerdo (`a == 2`) do operador `&&` é `true`, o lado direito (`b != 10`) desse operador deve ser avaliado. Como a variável `b` armazena o valor 10, o resultado de `b != 10` é `false`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && [b != 10]);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True

a = 2

b = 10

- 8 Continuando na linha 8, o operador `&&` devolverá o valor `false` pois, como vimos no passo anterior, o lado direito (`b != 10`) devolveu `false`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && [b != 10]);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True

a = 2

b = 10

- 9 Para finalizar a linha 8, o resultado da expressão `a == 2 && b != 10`, que é `false`, será exibido na saída padrão.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True
False

a = 2

b = 10

- 10 Agora, na linha 9, a operação `a <= 0` será executada. Como a variável `a` armazena o valor 2, o resultado dessa operação é `false`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine([a <= 0] ^ b >= 1);
10    }
11 }

```

True
False

a = 2

b = 10

- 11 Em seguida, ainda linha 9, a operação `b >= 1` será executada. Como a variável `b` armazena o valor 10, o resultado dessa operação é `true`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ [b >= 1]);
10    }
11 }

```

True
False

a = 2

b = 10

- 12 Continuando na linha 9, o operador `^` devolverá `true` pois, como vimos nos passos anteriores, o lado esquerdo (`a <= 0`) é `false` e o lado direito (`b >= 1`) é `true`.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True
False

a = 2

b = 10

- 13 Para finalizar a linha 9, o resultado da expressão `a <= 0 ^ b >= 1`, que é true, será exibido na saída padrão.

```

1 class Logicos
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a > 0 || b < 0);
8         System.Console.WriteLine(a == 2 && b != 10);
9         System.Console.WriteLine(a <= 0 ^ b >= 1);
10    }
11 }

```

True
False
● True

a = 2

b = 10

3.14 Operador Ternário ?:

Considere um programa que controla as notas dos alunos de uma escola. Para exemplificar, vamos gerar a nota de um aluno aleatoriamente.

```

1 System.Random gerador = new System.Random();
2 double nota = 10.0 * gerador.NextDouble();

```

O programa deve exibir a mensagem “aprovado” se nota de um aluno for maior ou igual a 5 e “reprovado” se a nota for menor do que 5. Esse problema pode ser resolvido com o operador ternário.



Figura 3.1: Operador ternário

Quando a “condição” (`nota >= 5`) é true, o operador ternário devolve o primeiro valor (“aprovado”). Caso contrário, devolve o segundo valor (“reprovado”). Podemos guardar o resultado do operador ternário em uma variável ou simplesmente exibi-lo.

```
1 string resultado = nota >= 5 ? "aprovado" : "reprovado";
2 System.Console.WriteLine(nota >= 5 ? "aprovado" : "reprovado");
```

No exemplo anterior, o operador ternário foi utilizado com valores do tipo `string`. Contudo, podemos utilizá-lo com qualquer tipo de valor. Veja o exemplo a seguir.

```
1 int i = nota >= 5 ? 1 : 2;
2 double d = nota >= 5 ? 0.1 : 0.2;
```

3.15 Operador de Negação



Valores booleanos podem ser invertidos com o operador de negação `!`, ou seja, o valor `true` pode ser substituído por `false` e vice-versa. Por exemplo, podemos verificar se uma variável do tipo `double` armazena um valor maior do que 5 de duas formas diferentes. A primeira delas verifica diretamente se o número é maior do que 5 com o uso do operador `>`.

```
1 d > 5
```

A segunda forma utiliza o operador de negação e verifica se o número não é menor ou igual a 5.

```
1 !(d <= 5)
```

Simulação



Nessa simulação, mostraremos um exemplo de utilização do operador ternário e do operador de negação.

- 1 Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `nota` do tipo `double` é declarada e inicializada com o valor 6.3.

```
1 class TernarioNegacao
2 {
3     static void Main()
4     {
5         double nota = 6.3;
6         bool reprovado = !(nota >= 5);
7         string s = reprovado ? "REPROVADO" : "APROVADO";
8         System.Console.WriteLine(s);
9     }
10 }
```

nota = 6.3

- 2 Em seguida, na linha 6, a operação `nota >= 5` é executada. Como a variável `nota` armazena o valor 6.3, o resultado dessa operação é `true`.

```

1 class TernarioNegacao
2 {
3     static void Main()
4     {
5         double nota = 6.3;
6         bool reprovado = !(nota >= 5);
7         string s = reprovado ? "REPROVADO" : "APROVADO";
8         System.Console.WriteLine(s);
9     }
10 }
```

nota = 6.3

- 3 Ainda na linha 6, o operador de negação é aplicado ao resultado da operação `nota >= 5`. Como vimos no passo anterior, essa operação devolveu `true`. Dessa forma, a expressão `!(nota >= 5)` devolve `false`. Esse valor é armazenado na variável `reprovado`.

```

1 class TernarioNegacao
2 {
3     static void Main()
4     {
5         double nota = 6.3;
6         bool reprovado = !(nota >= 5);
7         string s = reprovado ? "REPROVADO" : "APROVADO";
8         System.Console.WriteLine(s);
9     }
10 }
```

nota = 6.3

reprovado = false

- 4 Agora, na linha 7, o operador ternário é utilizado. Como o valor armazenado na variável `reprovado` é `false`, esse operador devolve “APROVADO”. Essa string é associada à variável `s`.

```

1 class TernarioNegacao
2 {
3     static void Main()
4     {
5         double nota = 6.3;
6         bool reprovado = !(nota >= 5);
7         string s = reprovado ? "REPROVADO" : "APROVADO";
8         System.Console.WriteLine(s);
9     }
10 }
```

nota = 6.3

reprovado = false

s = “APROVADO”

- 5 Por fim, a linha 8 é executada e a mensagem “APROVADO” é exibida na saída padrão.

```

1 class TernarioNegacao
2 {
3     static void Main()
4     {
5         double nota = 6.3;
6         bool reprovado = !(nota >= 5);
7         string s = reprovado ? "REPROVADO" : "APROVADO";
8         System.Console.WriteLine(s);
9     }
10 }
```

● APROVADO

nota = 6.3

reprovado = false

s = “APROVADO”



3.16 Incremento e Decremento

Os operadores `++` e `--` podem ser utilizados de duas formas diferentes, antes ou depois de uma variável de tipo simples numérico.

```
1 int i = 10;
2 i++; // pós incremento
3 i--; // pós decremento
```

Código C# 3.96: Pós incremento e pós decremento

```
1 int i = 10;
2 ++i; // pré incremento
3 --i; // pré decremento
```

Código C# 3.97: Pré incremento e pré decremento

No primeiro exemplo, o operador `++` foi utilizado à direita da variável `i`. Já no segundo exemplo, ele foi utilizado à esquerda da variável `i`. A primeira forma de utilizar o operador `++` é chamada de pós incremento. A segunda é chamada de pré incremento. Analogamente, o operador `--` foi utilizado na forma de pós decremento no primeiro exemplo e pré decremento no segundo exemplo.

Mas, qual é a diferença entre pré incremento e pós incremento (ou entre pré decremento e pós decremento)? As operações de incremento (ou decremeno) alteram o valor de uma variável e devolvem um valor. Tanto o pré incremento quanto o pós incremento adicionam 1 ao valor armazenado em uma variável. O pré incremento devolve o valor armazenado na variável após o seu incremento. O pós incremento, por outro lado, devolve o valor armazenado na variável antes de seu incremento. O comportamento é análogo para as operações de pré e pós decremento. Vejamos alguns exemplos a seguir.

```
1 int i = 10;
2 System.Console.WriteLine(i++ == 10); // exibe: true
3 System.Console.WriteLine(i); // exibe: 11
```

Observe que o operador `++` foi utilizado nas expressões do exemplo acima em conjunto com o operador `==`. Como dois operadores foram utilizados na mesma expressão, você pode ter dúvida em relação a quem será executado primeiro. O incremento com o operador `++` será realizado antes ou depois da comparação com o operador `==`?

O incremento ocorrerá antes da comparação. Contudo, como o operador `++` foi utilizado na forma de pós incremento, a operação `i++` devolverá o valor antigo da variável `i`. Dessa forma, a comparação utilizará o valor armazenado na variável `i` antes do incremento.

Analogamente, a comparação utilizaria o valor antigo da variável `i` se o operador `--` fosse utilizado na forma de pós decremento.

Agora, considere a utilização do operador `++` na forma de pré incremento.

```
1 int i = 10;
2 System.Console.WriteLine(++i == 10); // exibe: false
3 System.Console.WriteLine(i); // exibe: 11
```

Nesse último exemplo, a operação `++i` devolverá o valor novo da variável `i`. Dessa forma, a com-

paração utilizará o valor armazenado na variável `i` depois do incremento.

Analogamente, a comparação utilizaria o valor novo da variável `i` se o operador `--` fosse utilizado na forma de pré decremento.



Pare para pensar...

Considere o comportamento do pré incremento, pós incremento, pré decremento e pós decremento. O que seria exibido no exemplo abaixo?

```

1 int i = 10;
2 int j = 10;
3
4 System.Console.WriteLine(i++ == i - 1);
5 System.Console.WriteLine(++j == j);

```

Simulação



Nessa simulação, mostraremos um exemplo de utilização dos operadores de incremento e decremento.

- Ao executar o programa, a primeira linha do método `Main` será processada. Nessa linha, uma variável chamada `a` do tipo `int` é declarada e inicializada com o valor 2.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

`a = 2`

- Em seguida, a linha 6 será executada. Nessa linha, uma variável chamada `b` do tipo `int` é declarada e inicializada com o valor 10.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

`a = 2`

`b = 10`

- 3 Em seguida, na linha 7, a operação `a++` será executada. Dessa forma, a variável `a` passa a armazenar o valor 3. Como o operador `++` foi utilizado na forma de pós incremento, essa operação devolve o valor antigo da variável `a`, ou seja, devolve 2.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine([a++ == 2 || b++ == 10]);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }
```

a = 3

b = 10

- 4 Agora, ainda na linha 7, a operação de igualdade `a++ == 2` será executada para verificar se o resultado da operação `a++` é igual a 2. No passo anterior, vimos que a operação `a++` devolveu 2. Dessa forma, o resultado da expressão `a++ == 2` será `true`.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine([a++ == 2 || b++ == 10]);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }
```

a = 3

b = 10

- 5 Continuando na linha 7, o operador lógico `||` devolverá `true` sem avaliar o lado direito (`b++ == 10`) porque, como vimos no passo anterior, o lado esquerdo (`a++ == 2`) devolveu `true`.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine([a++ == 2 || b++ == 10]);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }
```

a = 3

b = 10

- 6 Para finalizar a linha 7, o valor da expressão `a++ == 2 || b++ == 10`, que é `true`, será exibido na saída padrão.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True

a = 3

b = 10

- 7 Em seguida, na linha 8, a operação `++a` será avaliada. Dessa forma, o valor armazenado na variável `a` passa a ser 4. Como o operador `++` foi utilizado na forma de pré incremento, a operação `++a` devolve o novo valor da variável `a`, ou seja, devolve 4.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True

a = 4

b = 10

- 8 Na sequência, ainda na linha 8, a operação de desigualdade `++a < 4` será avaliada. No passo anterior, vimos que a operação `++a` devolveu 4. Dessa forma, a expressão `++a < 4` devolverá `false`.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True

a = 4

b = 10

- 9 Continuando na linha 8, o operador `&&` devolverá `false` sem avaliar o lado direito (`--b == 10`) pois, como vimos no passo anterior, o lado esquerdo (`++a < 4`) devolveu `false`.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True

a = 4

b = 10

- 10 Para finalizar a linha 8, o resultado da expressão `++a < 4 && --b == 10`, que é `false`, será exibido na saída padrão.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True
False

a = 4

b = 10

- 11 Ao executar a linha 9, o valor da variável `a` será exibido na saída padrão.

```

1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }

```

True
False
4

a = 4

b = 10

- 12 Por fim, ao executar a linha 10, o valor da variável `b` será exibido na saída padrão.

```
1 class IncrementoDecremento
2 {
3     static void Main()
4     {
5         int a = 2;
6         int b = 10;
7         System.Console.WriteLine(a++ == 2 || b++ == 10);
8         System.Console.WriteLine(++a < 4 && --b == 10);
9         System.Console.WriteLine(a);
10        System.Console.WriteLine(b);
11    }
12 }
```

```
True
False
4
10
```

a = 4

b = 10

3.17 Avaliando uma Expressão



A ordem de avaliação das operações de uma expressão determina o resultado dessa expressão. Por exemplo, considere a expressão $2 + 3 * 5$. Se a operação de adição for efetuada primeiro, o resultado dessa expressão será 40. Por outro lado, se a multiplicação for efetuada antes da adição, o resultado será 17. Dizemos que essa expressão é ambígua.

Para avaliar uma expressão, devemos primeiramente eliminar qualquer ambiguidade presente na expressão. Para isso, vamos utilizar a Tabela 3.8.

| Precedência | Operador | Operação | Desempate |
|-------------|---|---|--------------------|
| 1 | <code>++</code> <code>--</code> | Pós incremento Pós decremento | |
| 2 | <code>++</code> <code>--</code> <code>+</code> <code>-</code> <code>!</code> <i>(tipo)</i> | Pré incremento Pré decremento Mais unário Menos unário Negação Casting | Direita → Esquerda |
| 3 | <code>*</code> <code>/</code> <code>%</code> | Multiplicação Divisão Módulo | Esquerda → Direita |
| 4 | <code>+</code> <code>-</code> | Adição Subtração | Esquerda → Direita |
| 5 | <code><</code> <code><=</code> <code>></code> <code>>=</code> | Menor Menor ou igual Maior Maior ou igual | Esquerda → Direita |
| 6 | <code>==</code> <code>!=</code> | Igualdade Diferença | Esquerda → Direita |
| 7 | <code>&</code> | E simples | Esquerda → Direita |
| 8 | <code>^</code> | Ou exclusivo | Esquerda → Direita |
| 9 | <code> </code> | Ou simples | Esquerda → Direita |
| 10 | <code>&&</code> | E duplo | Esquerda → Direita |
| 11 | <code> </code> | Ou duplo | Esquerda → Direita |
| 12 | <code>? :</code> | Ternário | Direita → Esquerda |
| 13 | <code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> | Atribuição Incremental Decremental Multiplicativa Divisória Modular | Direita → Esquerda |

Tabela 3.8: Precedência de operadores



Importante

O fato de um operador ter precedência sobre outro não significa que ele será executado primeiro. Por exemplo, de acordo com a Tabela 3.8, o operador de pré incremento (`++`) possui precedência sobre o operador de divisão (`/`). Sabendo disso, considere o código abaixo.

```

1 class DivisaoPorZero {
2     static void Main() {
3         int x = 0;
4         System.Console.WriteLine(1 / x + ++x);
5     }
6 }
```

Código C# 3.113: DivisaoPorZero.cs

Se a ordem de processamento das operações de uma expressão fosse a mesma da ordem indu-

zida pela precedência, a operação `++x` seria executada antes da operação de divisão. Nesse caso, o código acima não apresentaria erro de execução e exibiria o valor 1.

Contudo, na expressão `1 / x + ++x`, a primeira operação realizada é a de divisão. Assim, esse código produz um erro de execução causado por uma divisão por zero.

De acordo com a Tabela 3.8, o operador de multiplicação tem precedência sobre o operador de adição. Dessa forma, a multiplicação tem maior prioridade do que a adição. Sendo assim, a expressão `2 + 3 * 5` é equivalente à expressão `(2 + (3 * 5))`. Portanto, o resultado dessa expressão é 17.

Agora, considere a expressão `1 - 1 + 1`. Se a operação de subtração for executada antes da adição, o resultado será 1. Por outro lado, se a adição for efetuada antes da subtração, o resultado será `-1`. Nessa expressão, todos os operadores possuem a mesma precedência. Nesse caso, devemos nos orientar pela última coluna da Tabela 3.8 para determinar qual operador tem maior prioridade. Como o desempate desses operadores é realizado da esquerda para a direita, a subtração tem maior prioridade do que a adição nessa expressão em particular. Portanto, a expressão `1 - 1 + 1` é equivalente à expressão `((1 - 1) + 1)`. Logo, o resultado dessa expressão é 1.

Então, para eliminar ambiguidades em uma expressão, devemos acrescentar parênteses de acordo com a prioridade dos operadores. Vamos exemplificar essa tarefa com a seguinte expressão, onde os operadores estão destacados com a cor vermelha.

```
1 + 7 * 3 / 4 / (double)(int)x - y
```

As operações de casting possuem maior precedência nessa expressão. De acordo com a Tabela 3.8, quando há mais de uma operação de casting, a operação mais à direita possui prioridade sobre as outras. Dessa forma, devemos adicionar parênteses para envolver essa operação.

```
1 + 7 * 3 / 4 / (double)((int)x) - y
```

Considerando os operadores restantes (na cor vermelha), a outra operação de casting possui prioridade. Assim, ela deve ser envolvida com parênteses.

```
1 + 7 * 3 / 4 / ((double)((int)x)) - y
```

Agora, há um empate na precedência dos operadores de multiplicação e divisão. Como, nesse caso, a prioridade é do operador mais à esquerda, a multiplicação é a próxima operação.

```
1 + (7 * 3) / 4 / ((double)((int)x)) - y
```

Na sequência, o operador de divisão mais à esquerda possui prioridade sobre os outros.

```
1 + (7 * 3) / 4 / ((double)((int)x)) - y
```

Entre os operadores restantes, o operador de divisão possui prioridade sobre os outros.

$$1 + ((7 * 3) / 4) / ((\text{double})((\text{int})x)) - y$$

Agora, restam os operadores de adição e de subtração. Como o operador de adição está mais à esquerda, ele tem prioridade.

$$(1 + (((7 * 3) / 4) / ((\text{double})((\text{int})x)))) - y$$

Por fim, o último operador é o de subtração.

$$((1 + (((7 * 3) / 4) / ((\text{double})((\text{int})x))))) - y$$

Ao final desse processo, obtemos uma expressão sem ambiguidade. Agora, basta processar as operações de acordo com a seguinte regra: executar sempre o par de parênteses interno mais à esquerda até o valor da expressão ser obtido. Um par de parênteses é formado por um “abre parênteses” e um “fecha parênteses”. Dizemos que um par de parênteses é interno se ele não contém parênteses. No exemplo abaixo, os três pares de parênteses em destaque são internos.

$$((1 + ((7 * 3) / 4) / ((\text{double})((\text{int})x)))) - y$$

Suponha que as variáveis x e y sejam do tipo `double` e armazenem os valores 4.57 e 2.25, respectivamente. O par de parênteses interno mais à esquerda envolve a expressão $7 * 3$.

$$((1 + ((7 * 3) / 4) / ((\text{double})((\text{int})x)))) - y$$

O resultado dessa expressão é 21.

$$((1 + ((21 / 4) / ((\text{double})((\text{int})x)))) - y)$$

Agora, o par de parênteses interno mais à esquerda envolve a expressão $21 / 4$.

$$((1 + ((21 / 4) / ((\text{double})((\text{int})x)))) - y)$$

Como essa é uma divisão entre dois valores inteiros, o resultado dessa expressão é 5.

$$((1 + (5 / ((\text{double})((\text{int})x)))) - y)$$

Em seguida, o par de parênteses interno mais à esquerda envolve a expressão `(int)x`

$$((1 + (5 / ((\text{double})((\text{int})x)))) - y)$$

Como o valor de x é 4.57, o resultado dessa expressão é 4.

$((1 + (5 / ((double)4))) - y)$

Agora, o par de parênteses interno mais à esquerda envolve a expressão `(double)4`.

$((1 + (5 / ((double)4))) - y)$

O valor dessa expressão é 4.0.

$((1 + (5 / 4.0)) - y)$

Em seguida, o par de parênteses interno mais à esquerda envolve a expressão $5 / 4.0$.

$((1 + (5 / 4.0)) - y)$

O valor dessa expressão é 1.25.

$((1 + 1.25) - y)$

Agora, o par de parênteses a ser processado é o que envolve a expressão $1 + 1.25$.

$((1 + 1.25) - y)$

Como essa expressão é composta por um número do tipo `int` e um número do tipo `double`, o valor dessa expressão é 2.25.

$(2.25 - y)$

Finalmente, processamos o último par de parênteses da expressão.

$(2.25 - y)$

Como o valor armazenado em y é 2.25, o valor dessa última expressão é 0.0.

0.0

3.18 Manipulação de Strings



Nesta seção, mostraremos como manipular strings.

Quantidade de caracteres

A propriedade `Length` devolve a quantidade de caracteres (comprimento) de uma string.

```

1 string s = "Rafael Cosentino";
2
3 int length = s.Length;
4
5 System.Console.WriteLine(length); // exibe: 16

```

Caractere em determinada posição

Podemos acessar um determinado caractere de uma string através do operador `[]`. No exemplo abaixo, a variável `s` armazena uma referência para a string com conteúdo “Rafael Cosentino”. Para acessar o terceiro caractere dessa string, por exemplo, podemos fazer `s[2]`.

```

1 string s = "Rafael Cosentino";
2
3 char a = s[0]; // obtém o primeiro caractere
4 char b = s[9]; // obtém o décimo caractere
5 char c = s[15]; // obtém o décimo sexto (último) caractere
6
7 System.Console.WriteLine(a); // exibe: R
8 System.Console.WriteLine(b); // exibe: s
9 System.Console.WriteLine(c); // exibe: o

```

O número entre colchetes deve estar entre zero e o comprimento da string menos um. Um erro de execução ocorrerá se um número fora desse intervalo for usado. Note que um erro de execução ocorrerá se esse operador for utilizado com a string vazia.

Contains

O método `Contains` verifica se uma determinada sequência de caracteres está contida em uma string. Esse método recebe como parâmetro uma sequência de caracteres e devolve `true` se e somente se essa sequência está contida em um determinada string.

No exemplo abaixo, o método `Contains` é aplicado à string cuja referência está armazenada na variável `s`. Como essa string possui a sequência de caracteres “Objetos”, a instrução da linha 6 exibe “true”. Como essa string não contém a sequência de caracteres “objetos”, a instrução da linha 7 exibe “false”.

```

1 string s = "K11 - Orientação a Objetos em C#";
2
3 bool resultado1 = s.Contains("Objetos");
4 bool resultado2 = s.Contains("objetos");
5
6 System.Console.WriteLine(resultado1); // exibe: True
7 System.Console.WriteLine(resultado2); // exibe: False

```

EndsWith

É possível verificar se uma string termina com uma determinada sequência de caracteres. Para isso, podemos usar o método `EndsWith`. Esse método recebe uma sequência de caracteres como parâmetro. Ele devolve `true` se e somente se uma determinada string termina com essa sequência de caracteres.

No exemplo a seguir, o método `EndsWith` foi aplicado à string cuja referência está armazenada na variável `s`. Como essa string termina com “C#”, a instrução da linha 6 exibe “True”. Como essa string não termina com “Objetos”, a instrução da linha 7 exibe “False”.

```
1 string s = "K11 - Orientação a Objetos em C#";
2
3 bool resultado1 = s.EndsWith("C#");
4 bool resultado2 = s.EndsWith("Objetos");
5
6 System.Console.WriteLine(resultado1); // exibe: True
7 System.Console.WriteLine(resultado2); // exibe: False
```

StartsWith

É possível também verificar se uma string começo com uma determinada sequência de caracteres. Podemos fazer isso com o método `StartsWith`. Esse método recebe uma sequência de caracteres como parâmetro. Ele devolve `true` se e somente se uma determinada string começo com essa sequência de caracteres.

No exemplo abaixo, o método `StartsWith` é aplicado à string cuja referência está armazenada na variável `s`. Como essa string não começo com “C#”, a instrução da linha 6 exibe “False”. Como essa string começo com “K11”, a instrução da linha 7 exibe “True”.

```
1 string s = "K11 - Orientação a Objetos em C#";
2
3 bool resultado1 = s.StartsWith("C#");
4 bool resultado2 = s.StartsWith("K11");
5
6 System.Console.WriteLine(resultado1); // exibe: False
7 System.Console.WriteLine(resultado2); // exibe: True
```

Replace

Podemos realizar substituições em uma string com o método `Replace`. Esse método recebe duas strings como parâmetros e devolve um objeto do tipo `string`. Considere o exemplo abaixo.

```
1 string s1 = "Matamos o tempo, o tempo nos enterra"; // Machado de Assis
2
3 s1 = s1.Replace("o tempo", "um coveiro");
4
5 System.Console.WriteLine(s1); // exibe: Matamos um coveiro, um coveiro nos enterra
```

Na linha 3, o método `Replace` substituirá cada ocorrência de “o tempo” na string cuja referência está em `s1` por “um coveiro”. Assim, ele devolve “Matamos um coveiro, um coveiro nos enterra”.

Na verdade, o primeiro parâmetro desse método é uma expressão regular, mas esse assunto está fora do escopo deste livro.

Substring

É possível extrair um trecho de uma string com o método `Substring`. Há duas formas de utilizar esse método. Na primeira, ele recebe apenas um parâmetro. Esse parâmetro indica a posição do primeiro caractere do trecho desejado. O término desse trecho será o último caractere da string. Veja o exemplo abaixo.

```
1 string s1 = "Porta corta fogo";
2
3 string s2 = s1.Substring(9);
4
5 System.Console.WriteLine(s2); // exibe: ta fogo
```

Na linha 3, o método `Substring` é invocado com o argumento 9. Assim, ele devolve a substring de “Porta corta fogo” que começa na posição 9 e se estende até o fim. Logo, a instrução da linha 5 exibe “ta fogo”.

Na segunda forma de utilizar o método `Substring`, dois parâmetros do tipo `int` são necessários: *a* e *b*. O método então devolve o trecho que começa com o caractere na posição *a* e possui *b* caracteres. Vejamos um exemplo.

```
1 string s1 = "O elevador chegou";
2
3 string s2 = s1.Substring(2, 8);
4
5 System.Console.WriteLine(s2); // exibe: elevador
```

Na linha 3, o método `Substring` é invocado com os argumentos 2 e 8. Assim, ele devolve a substring de “O elevador chegou” que começa na posição 2 e possui 8 caracteres. Logo, a instrução da linha 5 exibe “elevador”.

ToUpper

Podemos transformar em maiúsculas todas as letras contidas em uma string com o método `ToUpper`. No exemplo abaixo, esse método é aplicado à string cujo conteúdo é “Rafael Cosentino”. Esse método então devolve a string com todas as letras maiúsculas, ou seja, devolve “RAFAEL COSENTINO”.

```
1 string s1 = "Rafael Cosentino";
2
3 string s2 = s1.ToUpper();
4
5 System.Console.WriteLine(s2); // exibe: RAFAEL COSENTINO
```

ToLower

Também podemos transformar em minúsculas todas as letras contidas em uma string com o método `ToLower`. No exemplo a seguir, esse método é aplicado à string cujo conteúdo é “Rafael Cosentino”. Assim, esse método devolve a string “rafael cosentino”.

```
1 string s1 = "Rafael Cosentino";
2
3 string s2 = s1.ToLower();
4
5 System.Console.WriteLine(s2); // exibe: rafael cosentino
```

Trim

Com o método `Trim` das strings, podemos eliminar os espaços em branco do começo e do término de uma string. Veja o exemplo abaixo.

```
1 string s1 = "      Rafael     Cosentino      ";
2
3 string s2 = s1.Trim();
4
5 // "Rafael     Cosentino"
6 System.Console.WriteLine(s2);
```

No código acima, a chamada ao método `Trim` devolve a string que não possui espaços antes de “Rafael” e não possui espaços após “Cosentino”. Observe que os espaços entre “Rafael” e “Cosentino” não são removidos.

3.19 Operações com Data e Hora



Algumas operações são específicas para data e hora. A seguir, apresentaremos algumas dessas operações.

Add

Podemos modificar uma data e hora acrescentando ou subtraindo milisegundos, segundos, minutos, horas, dias, meses ou anos. Essa tarefa pode ser realizada com os métodos `AddMilliseconds`, `AddSeconds`, `AddMinutes`, `AddHours`, `AddDays`, `AddMonths` e `AddYears` da classe `DateTime`.

```
1 DateTime dt = new DateTime(2010, 8, 27);
2
3 // Acrescentando 140 dias
4 dt = dt.AddDays(140);
5
6 // Subtraindo 2 anos
7 dt = dt.AddYears(-2);
8
9 // Acrescentando 72 horas
10 dt = dt.AddHours(72);
```

Na primeira linha do código acima, é criado um objeto do tipo `DateTime` para representar a data 27 de Agosto de 2010. Na linha 4, foram adicionados 140 dias a essa data. Assim, obtemos a data 14 de Janeiro de 2011. Na linha 7, são subtraídos dois anos dessa data. Assim, a data obtida é 14 de Janeiro de 2009. Na última linha, são adicionadas 72 horas. Assim, a data obtida é 17 de Janeiro de 2009.

Comparações

Podemos comparar datas e horas com os operadores de comparação. Veja o exemplo abaixo.

```

1 DateTime dt1 = new DateTime(2010, 8, 27);
2 DateTime dt2 = DateTime.Now; // data e hora atuais
3
4 System.Console.WriteLine(dt1 < dt2); // exibe: True
5
6 System.Console.WriteLine(dt1 > dt2); // exibe: False

```

Na primeira linha do código acima, é criado um objeto do tipo `DateTime` que representa a data 27 de Agosto de 2010. Na segunda linha, um objeto do tipo `DateTime` é criado para representar a data no momento de execução dessa instrução. Na linha 4, usamos o operador `<` para verificar se a primeira data é anterior à segunda. Na linha 6, usamos o operador `>` para verificar se a primeira data é posterior à segunda.

3.20 Erro: Utilizar operandos e operadores incompatíveis



Um erro de compilação comum ocorre quando um operador é aplicado a valores incompatíveis. Veja alguns exemplos de programas em C# com esse problema.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string s1 = "K19";
6         string s2 = "Treinamentos";
7
8         System.Console.WriteLine(s1 - s2);
9     }
10 }

```

Código C# 3.127: *Programa.cs*

O operador `-` não pode ser aplicado a valores do tipo `string`. A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(8,28): error CS0019: Operator '-' cannot be applied to operands of
type 'string' and 'string'
```

Terminal 3.35: Erro de compilação

No código abaixo, o operador `>` é usado com operandos do tipo `bool`.

```

1 class Programa
2 {
3     static void Main()
4     {
5         bool b1 = true;

```

```

6     bool b2 = false;
7
8     System.Console.WriteLine(b1 > b2);
9 }
10 }
```

Código C# 3.128: Programa.cs

Apenas os operadores de igualdade e desigualdade podem ser aplicados a valores do tipo `bool`. A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(8,28): error CS0019: Operator '>' cannot be applied to operands of
type 'bool' and 'bool'
```

Terminal 3.36: Erro de compilação

No código a seguir, o operador de negação `!` é aplicado a um valor do tipo `int`.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int i = 1;
6
7         System.Console.WriteLine(!i);
8     }
9 }
```

Código C# 3.129: Programa.cs

O operador de negação não pode ser aplicado a valores de tipos primitivo numéricos. A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(7,28): error CS0023: Operator '!' cannot be applied to operand of
type 'int'
```

Terminal 3.37: Erro de compilação

3.21 Erro: Divisão inteira por zero



Um erro de execução comum ocorre quando um valor numérico inteiro é dividido pelo valor inteiro 0. Veja um exemplo de programa em C# com esse problema.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 10;
6         int b = 100 / (a - 10);
7     }
8 }
```

Código C# 3.130: Programa.cs

Ao executar o programa, a instrução da linha 6 provoca um erro de execução devido à divisão do valor inteiro 100 pelo valor inteiro 0.

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
```

```
at Programa.Main()
```

Terminal 3.38: Erro de execução

3.22 Erro: Armazenamento de valores incompatíveis



Um erro comum ocorre quando tentamos armazenar em uma variável um valor que não é compatível com o tipo dessa variável. A seguir, apresentamos alguns exemplos onde esse tipo de problema acontece. Considere o código abaixo.

```
1 class Programa
2 {
3     static void Main()
4     {
5         double a = 10.0;
6         int b = 5 + a;
7     }
8 }
```

Código C# 3.131: Programa.cs

Nesse exemplo, declaramos a variável `a` do tipo `double` e a variável `b` do tipo `int`. Na linha 6, tentamos armazenar o valor de uma expressão cujo resultado é do tipo `double` na variável `b`. Como `b` é do tipo `int`, isso provoca um erro de compilação.

```
Programa.cs(6,11): error CS0266: Cannot implicitly convert type 'double' to
'int'. An explicit conversion exists (are you missing a cast?)
```

Terminal 3.39: Erro de compilação

No exemplo abaixo, declaramos uma variável do tipo `float` e tentamos inicializá-la com o valor 3.14.

```
1 class Programa
2 {
3     static void Main()
4     {
5         float a = 3.14;
6     }
7 }
```

Código C# 3.132: Programa.cs

Como 3.14 é um valor do tipo `double`, isso provoca um erro de compilação.

```
Programa.cs(5,13): error CS0664: Literal of type double cannot be implicitly
converted to type 'float'; use an 'F' suffix to create a literal of this
type
```

Terminal 3.40: Erro de compilação

3.23 Erro: Castings não permitidos



Algumas operações de castings não são permitidas em C#. Por exemplo, considere a conversão de um número inteiro para `string`.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 19;
6         string b = (string)a;
7     }
8 }
```

Código C# 3.133: Programa.cs

A conversão de um número inteiro para `string` não é permitida com o uso de casting. A classe `Programa` acima não compila e apresenta o seguinte erro de compilação.

```
Programa.cs(6,14): error CS0030: Cannot convert type 'int' to 'string'
```

Terminal 3.41: Erro de compilação

Para converter um número inteiro para `string`, podemos usar o método `ToString`. A conversão de `string` para `int` também não é permitida usando uma operação de casting.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string b = "19";
6         int a = (int)b;
7     }
8 }
```

Código C# 3.134: Programa.cs

O código acima apresenta o seguinte erro de compilação.

```
Programa.cs(6,11): error CS0030: Cannot convert type 'string' to 'int'
```

Terminal 3.42: Erro de compilação

Para converter uma `string` para um `int`, podemos usar o método `ToInt32` da classe `System.Convert`.

3.24 Resumo



- 1 ► Os operadores são utilizados para manipular os valores armazenados nas variáveis ou valores literais.
- 2 ► Não é necessário fazer casting para copiar o valor armazenado em uma variável de tipo simples para outra variável de tipo simples desde que o tipo da primeira variável seja compatível com o tipo da segunda. Ver Tabela 3.1.
- 3 ► Apesar do tipo `long` ser compatível com o tipo `float`, copiar valores armazenados em uma variável do tipo `long` para uma variável do tipo `float` pode gerar resultados indesejados.

- 4 Operações de casting podem gerar resultados bem diferentes dos desejados.
- 5 A conversão de valores dos tipos byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal e bool para string pode ser realizada através dos métodos ToByte, ToSByte,ToInt16, ToUInt16,ToInt32, ToUInt32,ToInt64, ToUInt64, ToSingle, ToDouble, ToDecimal e ToBoolean da classe System.Convert, respectivamente.
- 6 As operações aritméticas de adição, subtração, multiplicação, divisão e resto são realizadas respectivamente através dos operadores +, -, *, / e %.
- 7 Na divisão entre valores inteiros, se o divisor for zero, ocorrerá o erro de execução System.DivideByZeroException.
- 8 Na divisão entre valores reais, se o divisor for zero, o resultado poderá ser +Infinity, -Infinity ou NaN.
- 9 A divisão entre valores inteiros desconsidera a parte fracionária do resultado.
- 10 Operações aritméticas podem produzir overflow e underflow.
- 11 O operador + também é utilizado para realizar a concatenação de strings.
- 12 O conteúdo de uma variável pode ser modificado através dos operadores de atribuição: =, +=, -=, *=, /=, %=, ++ e --.
- 13 Podemos comparar o conteúdo das variáveis ou os valores literais através dos operadores relacionais: ==, !=, <, <=, > e >=.
- 14 Operadores relacionais devolvem valores booleanos.
- 15 As operações lógicas E, OU e OU EXCLUSIVO são realizadas através dos operadores: &, &&, |, || e ^.
- 16 O operador && não avalia o segundo operando se o valor do primeiro operando for false.
- 17 O operador || não avalia o segundo operando se o valor do primeiro operando for true.
- 18 O primeiro argumento do operador ternário ?: deve ser um valor booleano.

- 19 ► O operador de negação ! inverte os valores booleanos.
- 20 ► O operador ++ pode ser utilizado na forma de pré e pós incremento.
- 21 ► O operador -- pode ser utilizado na forma de pré e pós decremento.

CONTROLE DE FLUXO

4.1 Introdução



Neste capítulo, mostraremos instruções que permitem controlar o fluxo de execução de um programa. Essas instruções aumentam a “inteligência” do código. Basicamente, as linguagens de programação oferecem dois tipos de instruções para controlar o fluxo de execução dos programas: instruções de **decisão** e de **repetição**.

4.2 Instruções de Decisão



Considere um parque de diversões como os da Disney. Nesses parques, para garantir a segurança, alguns brinquedos possuem restrições de acesso. Em geral, essas restrições estão relacionadas à altura dos visitantes. Em alguns parques, a altura do visitante é obtida por sensores instalados na entrada dos brinquedos e um programa de computador libera ou bloqueia o acesso de acordo com altura obtida. Então, o programa deve decidir se executa um trecho de código de acordo com uma condição. Essa decisão pode ser realizada através das instruções de decisão oferecidas pelas linguagens de programação.

Nos exemplos vistos nos capítulos anteriores, a ordem da execução das linhas de um programa é exatamente a ordem na qual elas foram definidas no código fonte. As instruções de decisão permitem decidir se um bloco de código deve ou não ser executado. As instruções de decisão são capazes de criar um “desvio” no fluxo de execução de um programa.

Neste capítulo, veremos as seguintes instruções de decisão: `if`, `else` e `switch`.

4.3 Instrução if



A instrução de decisão `if` é utilizada quando um determinado trecho de código deve ser executado apenas se uma condição for `true`. A sintaxe da instrução `if` é a seguinte:

```
1 if (condição)
2 {
3     bloco de comandos
4 }
```

Como funciona a instrução `if`? Se a condição for `true`, o bloco de comandos será executado. Caso

contrário, ou seja, se a condição for `false`, o bloco de comandos não será executado.

A condição da instrução `if` deve ser um valor do tipo `bool`. A Figura 4.1 ilustra o fluxo de execução da instrução `if`.

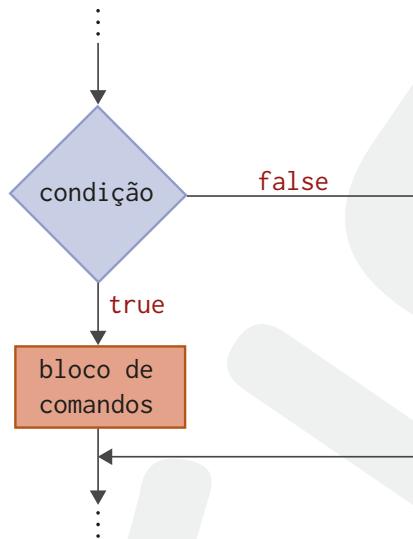


Figura 4.1: Fluxograma da instrução `if`

Simulação



A altura mínima para o ingresso na atração “The Barnstormer” do parque temático da Disney “Magic Kingdom” é **0.89 metros**. Vamos simular a execução do programa que controla o acesso dos visitantes a essa atração.

- 1 Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");
  
```

- 2 Ao executar a linha 2, um número aleatório do tipo `double` é gerado com o método `NextDouble()`. Vamos utilizar esse número para representar a altura de um visitante que deseja ingressar na atração “The Barnstormer”. Esse valor é armazenado na variável `altura`. Suponha que o valor 0.75 tenha sido gerado.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 3 Em seguida, a linha 3 é executada e o valor armazenado na variável altura é exibido no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 4 Na sequência, a linha 4 é executada e a comparação da condição do if devolve true pois o valor da variável altura é menor do que 0.89.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 5 A linha 6 é executada porque a condição do if da linha 4 é verdadeira. Dessa forma, a mensagem “Acesso bloqueado” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

0.75
● Acesso bloqueado

- 6 Por fim, a linha 8 é executada e a mensagem “The Barnstormer” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

0.75
Acesso bloqueado
The Barnstormer

altura = 0.75



Simulação

- 1 Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

- 2 Na sequência, na linha 2, um número aleatório do tipo `double` é gerado com o método `NextDouble()`. Vamos utilizar esse número para representar a altura de um visitante que deseja ingressar na atração “The Barnstormer”. Esse valor é armazenado na variável `altura`. Suponha que o valor 0.97 tenha sido gerado.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

altura = 0.97

- 3 Prosseguindo, na linha 3, o valor armazenado na variável `altura` é exibido no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

0.97

altura = 0.97

- 4 Na linha 4, a comparação da condição do `if` devolve `false` pois o valor da variável `altura` não é menor do que 0.89.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

0.97

altura = 0.97

- 5 A linha 6 não é executada porque a condição do `if` da linha 4 é falsa. Dessa forma, o fluxo de execução vai direto para a linha 8 e a mensagem “The Barnstormer” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 System.Console.WriteLine("The Barnstormer");

```

0.97
● The Barnstormer

altura = 0.97

4.4 Instrução else



Muitas vezes, queremos executar um bloco de comandos caso uma condição seja verdadeira e outro bloco de comandos caso essa condição seja falsa. Para isso, podemos utilizar as instruções de decisão `if` e `else`. Veja abaixo, a estrutura dessas instruções.

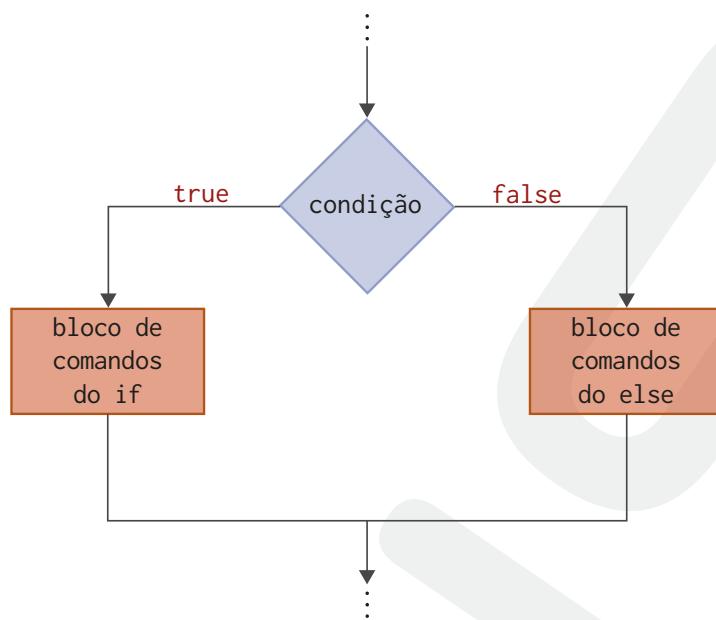
```

1 if([condição])
2 {
3     bloco de comandos
4 }
5 else
6 {
7     bloco de comandos
8 }

```

No exemplo acima, se a condição do `if` for `true`, o bloco de comandos do `if` será executado. Caso contrário, ou seja, se a condição for `false`, o bloco de comandos do `else` será executado.

A instrução `else` não pode aparecer sozinha no código sem estar vinculada a uma instrução `if`. A instrução `else` pode ser traduzida em português para “senão”. Em português, assim como em C#, não faz sentido dizer “senão” sem antes dizer “se”. Por isso, não podemos utilizar a instrução `else` sem antes ter utilizado a instrução `if`. A Figura 4.2 exibe o fluxograma das instruções `if` e `else`.

Figura 4.2: Fluxograma das instruções `if` e `else`

Simulação



A altura mínima para o ingresso na atração “The Barnstormer” do parque temático da Disney “Magic Kingdom” é **0.89 metros**. Vamos simular a execução do programa que controla o acesso dos visitantes a essa atração.

- 1 Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");
    
```

- 2 Prosseguindo, na linha 2, um número aleatório do tipo `double` é gerado com o método `NextDouble()`. Vamos utilizar esse número para representar a altura de um visitante que deseja ingressar na atração “The Barnstormer”. Esse valor é armazenado na variável `altura`. Suponha que o valor 0.75 tenha sido gerado.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 3 Na linha 3, o valor armazenado na variável altura é exibido no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 4 Na linha 4, a comparação da condição do if devolve true pois o valor da variável altura é menor do que 0.89.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.75

- 5 A linha 6 é executada porque a condição do if da linha 4 é verdadeira. Dessa forma, a mensagem "Acesso bloqueado" é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

0.75
● Acesso bloqueado

```

altura = 0.75

- 6 Por fim, o fluxo de execução “pula” para a linha 12 e a mensagem “The Barnstormer” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

0.75
● Acesso bloqueado
● The Barnstormer

```

altura = 0.75

Simulação



- 1 Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

- 2 Em seguida, na linha 2, um número aleatório do tipo `double` é gerado com o método `NextDouble()`. Vamos utilizar esse número para representar a altura de um visitante que deseja ingressar na atração

“The Barnstormer”. Esse valor é armazenado na variável altura. Suponha que o valor 0.97 tenha sido gerado.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.97

- 3 Na linha 3, o valor armazenado na variável altura é exibido no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.97

- 4 Na linha 4, a comparação da condição do if devolve false pois o valor da variável altura não é menor do que 0.89.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

```

altura = 0.97

- 5 A linha 6 não é executada porque a condição do if da linha 4 é falsa. Dessa forma, o fluxo de execução vai direto para a linha 10 e a mensagem “Acesso liberado” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

0.97
● Acesso liberado

```

altura = 0.97

- 6 Por fim, o fluxo de execução continua e a linha 12 é executada exibindo a mensagem “The Barnstormer”.

```

1 System.Random gerador = new System.Random();
2 double altura = gerador.NextDouble();
3 System.Console.WriteLine(altura);
4 if(altura < 0.89)
5 {
6     System.Console.WriteLine("Acesso bloqueado");
7 }
8 else
9 {
10    System.Console.WriteLine("Acesso liberado");
11 }
12 System.Console.WriteLine("The Barnstormer");

0.97
● Acesso liberado
● The Barnstormer

```

altura = 0.97

4.5 Instruções de Decisão Encadeadas



Considere um programa de computador que controla os saques efetuados nos caixas eletrônicos de um banco. Nesse banco, os saques efetuados das 6 horas até as 22 horas não podem ser superiores a R\$ 5.000,00. Por outro lado, os saques efetuados depois das 22 horas e antes das 6 horas não podem ser superiores a R\$ 400,00. Podemos implementar essa lógica utilizando as instruções de decisão oferecidas pelas linguagens de programação.

```

1 if(hora >= 6 && hora <= 22)
2 {
3     if(valor <= 5000)
4     {
5         System.Console.WriteLine("Saque efetuado com sucesso");
6     }
7     else
8     {
9         System.Console.WriteLine("Valor máximo de saque é R$ 5000,00");
10    }
11 }
12 else
13 {
14     if(valor <= 400)
15     {
16         System.Console.WriteLine("Saque efetuado com sucesso");

```

```

17 }
18 else
19 {
20     System.Console.WriteLine("Valor máximo de saque é R$ 400,00");
21 }
22 }

```

Simulação



Nessa simulação, mostraremos um exemplo de utilização de instruções de decisão encadeadas.

- 1 Ao executar a primeira linha, uma variável chamada `nota` do tipo `double` é declarada e inicializada com o valor 8.2.

```

1 double nota = 8.2;
2 if (nota >= 5.0)
3 {
4     System.Console.WriteLine("Aprovado");
5     if (nota < 8.0)
6     {
7         System.Console.WriteLine("Sua nota foi regular");
8     }
9     else
10    {
11        System.Console.WriteLine("Sua nota foi ótima");
12    }
13 }
14 else
15 {
16     System.Console.WriteLine("Reprovado");
17 }

```

nota = 8.2

- 2 Na sequência, a linha 2 será executada. Como a variável `nota` armazena o valor 8.2, a operação `nota >= 5.0` devolve `true`.

```

1 double nota = 8.2;
2 if (nota >= 5.0)
3 {
4     System.Console.WriteLine("Aprovado");
5     if (nota < 8.0)
6     {
7         System.Console.WriteLine("Sua nota foi regular");
8     }
9     else
10    {
11        System.Console.WriteLine("Sua nota foi ótima");
12    }
13 }
14 else
15 {
16     System.Console.WriteLine("Reprovado");
17 }

```

nota = 8.2

- 3 Como a condição do `if` da linha 2 é verdadeira, a linha 4 é executada e a palavra “Aprovado” é exibida na saída padrão.

```

1 double nota = 8.2;
2 if (nota >= 5.0)
3 {
4     System.Console.WriteLine("Aprovado");
5     if (nota < 8.0)
6     {
7         System.Console.WriteLine("Sua nota foi regular");
8     }
9     else
10    {
11        System.Console.WriteLine("Sua nota foi ótima");
12    }
13 }
14 else
15 {
16     System.Console.WriteLine("Reprovado");
17 }

```

● Aprovado

nota = 8.2

- 4 Continuando a execução, a linha 5 será processada. Como a variável nota armazena o valor 8.2, a operação nota < 8.0 devolve false.

```

1 double nota = 8.2;
2 if (nota >= 5.0)
3 {
4     System.Console.WriteLine("Aprovado");
5     if (nota < 8.0)
6     {
7         System.Console.WriteLine("Sua nota foi regular");
8     }
9     else
10    {
11        System.Console.WriteLine("Sua nota foi ótima");
12    }
13 }
14 else
15 {
16     System.Console.WriteLine("Reprovado");
17 }

```

Aprovado

nota = 8.2

- 5 Como a condição do if da linha 5 é falsa, o fluxo de execução é direcionado para a linha 11. Assim, a mensagem “Sua nota foi ótima” é exibida na saída padrão.

```

1 double nota = 8.2;
2 if (nota >= 5.0)
3 {
4     System.Console.WriteLine("Aprovado");
5     if (nota < 8.0)
6     {
7         System.Console.WriteLine("Sua nota foi regular");
8     }
9     else
10    {
11        System.Console.WriteLine("Sua nota foi ótima");
12    }
13 }
14 else
15 {
16     System.Console.WriteLine("Reprovado");
17 }

```

nota = 8.2

● Aprovado
Sua nota foi ótima

4.6 Instruções de Repetição



Considere um programa que gera bilhetes de loteria. O número do primeiro bilhete é 1000, do segundo 1001, do terceiro 1002 e assim por diante até o último bilhete numerado com 9999. Para esse tipo de tarefa, podemos utilizar as instruções de repetição oferecidas pelas linguagens de programação.

Como vimos, as instruções de decisão permitem que um determinado trecho de código seja executado ou não. Agora, as instruções de repetição permitem que um determinado trecho de código seja executado várias vezes.

Veremos neste capítulo as instruções de repetição `while`, `for` e `do-while`.

4.7 Instrução while



A instrução de repetição `while` recebe como parâmetro uma condição. O código no corpo do `while` será executado de acordo com essa condição. A sintaxe dessa instrução é a seguinte:

```
1 | while([condição])
2 | {
3 |     bloco de comandos
4 | }
```

Traduzindo para o português a instrução `while` como **enquanto**, fica mais fácil entender o seu funcionamento. O código acima poderia ser lido da seguinte forma: “Enquanto a condição for verdadeira, execute o bloco de comandos do `while`”.

A condição da instrução `while` deve ser um valor do tipo `bool`. Veja na Figura 4.3 o fluxo de execução da instrução `while`.

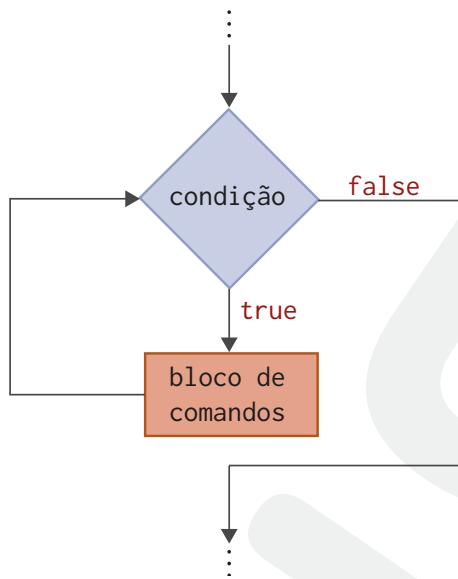


Figura 4.3: Fluxograma da instrução while

Considere um programa que exibe na tela cem mensagens de acordo com o seguinte padrão:

```

Mensagem número 1
Mensagem número 2
...
Mensagem número 100

```

Terminal 4.19: Programa que exibe mensagens

Esse programa poderia ser implementado sem as instruções de repetição.

```

1 System.Console.WriteLine("Mensagem número 1");
2 System.Console.WriteLine("Mensagem número 2");
3 ...
4 System.Console.WriteLine("Mensagem número 100");

```

Código C# 4.33: Exibindo a frase "Mensagem número x"

O código acima teria cem linhas de instruções. Contudo, utilizando a instrução while, o código fica bem mais simples.

```

1 // Variável que indica o índice da próxima mensagem que deve exibida.
2 int i = 1;
3
4 while(i <= 100)
5 {
6     System.Console.WriteLine("Mensagem número " + i);
7     i++;
8 }

```

Código C# 4.34: Exibindo a frase "Mensagem número x"

Até agora, o uso da instrução while parece ser mais uma conveniência do que uma necessidade. Vamos mudar um pouco o exemplo anterior para verificar a importância das instruções de repetição. Suponha que a frase “Mensagem número x” tenha de ser exibida um número aleatório de vezes, mas que não seja possível determinar o número máximo de vezes que ela será executada. Por exemplo,

suponha que a mensagem deva ser exibida enquanto o valor sorteado pelo método `NextDouble` for menor do que 0.99999.

```

1 int i = 0;
2
3 System.Random gerador = new System.Random();
4
5 while (gerador.NextDouble() < 0.99999)
6 {
7     System.Console.WriteLine("Mensagem número " + (i + 1));
8     i++;
9 }
```

Código C# 4.35: Exibindo a frase “Mensagem número x” um número aleatório de vezes

A cada vez que é executado, o programa acima pode exibir uma quantidade diferente de mensagens. Esse comportamento seria possível sem a utilização de uma instrução de repetição?

Simulação



Vamos simular a execução de um programa que gera bilhetes de loteria. Para não alongar muito a simulação, apenas 3 bilhetes serão gerados. Esses bilhetes devem ser numerados sequencialmente iniciando com o número 1000.

- 1 Na linha 1, a variável `numero` é declarada e inicializada com o valor 1000.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");
```

numero = 1000

- 2 Na linha 2, a condição do `while` é testada. Como o valor da variável `numero` é menor ou igual a 1002, a condição `numero <= 1002` devolve `true`.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");
```

numero = 1000

- 3 Como a condição da linha 2 devolveu `true`, o corpo do `while` será executado. Ao executar a linha 4, a mensagem “Bilhete 1000” é exibida no terminal.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000

numero = 1000

- 4 Ao executar a linha 5, a variável `numero` é incrementada para 1001.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000

numero = 1001

- 5 O fluxo de execução volta para a linha 2 e a condição do `while` é testada novamente. Mais uma vez, o valor da variável `numero` é menor ou igual a 1002. Dessa forma, a condição `numero <= 1002` devolve `true`.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000

numero = 1001

- 6 Como a condição da linha 2 devolveu `true`, o corpo do `while` será executado. Ao executar a linha 4, a mensagem “Bilhete 1001” é exibida no terminal.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000

Bilhete 1001

numero = 1001

- 7 Ao executar a linha 5, a variável `numero` é incrementada para 1002.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001

numero = 1002

- 8 Agora, o fluxo de execução volta para a linha 2 e a condição do `while` é testada novamente. O valor da variável `numero` é igual a 1002. Dessa forma, a condição `numero <= 1002` ainda devolve `true`.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001

numero = 1002

- 9 Como a condição da linha 2 devolveu `true`, o corpo do `while` será executado. Ao executar a linha 4, a mensagem “Bilhete 1002” é exibida no terminal.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
● Bilhete 1002

numero = 1002

- 10 Ao executar a linha 5, a variável `numero` é incrementada para 1003.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
Bilhete 1002

numero = 1003

- 11 Mais uma vez, o fluxo de execução volta para a linha 2 para testar a condição do `while`. Finalmente, o valor da variável `numero` não é menor ou igual a 1002. Dessa forma, a condição devolve `false`.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

Bilhete 1000
Bilhete 1001
Bilhete 1002

```

numero = 1003

- 12** Como a condição da linha 2 é falsa, o corpo do `while` não será mais executado. Portanto, o laço é interrompido e o fluxo de execução “pula” para a linha 7. Ao executar essa linha, a mensagem “FIM” é exibida no terminal.

```

1 int numero = 1000;
2 while(numero <= 1002)
3 {
4     System.Console.WriteLine("Bilhete " + numero);
5     numero++;
6 }
7 System.Console.WriteLine("FIM");

Bilhete 1000
Bilhete 1001
Bilhete 1002
● FIM

```

numero = 1003

4.8 Instrução for



A instrução `for` é outra instrução de repetição e tem a mesma finalidade da instrução `while`. Podemos resolver problemas que envolvem repetições com a instrução `while` ou `for`. Veja a sintaxe da instrução `for`:

```

1 for([inicialização]; [condição]; [atualização])
2 {
3     [bloco de comandos]
4 }

```

No lugar da **inicialização**, devemos inserir os comandos que serão executados antes do início do laço. No lugar da **condição**, devemos inserir uma expressão booleana que será verificada antes de cada iteração (repetição) do laço. Assim, como na instrução `while`, o laço continuará a ser executado enquanto essa condição for verdadeira. No lugar da **atualização**, devemos inserir os comandos que serão executadas ao final de cada iteração do laço. A Figura 4.4 exibe o fluxograma da instrução `for`.

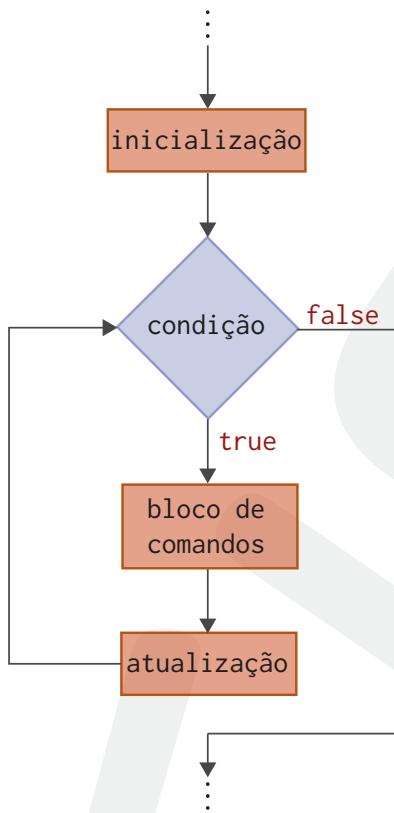


Figura 4.4: Fluxograma da instrução `for`



Importante

O termo **iteração** é utilizado quando nos referimos à repetição de uma ou mais ações. O código do corpo de um laço pode ser executado diversas vezes. A cada vez que ele é executado, dizemos que ocorre uma iteração.

Considere o seguinte trecho de código que utiliza a instrução de repetição `while`.

```

1 // inicialização
2 int i = 1;
3
4 // condição
5 while(i <= 100)
6 {
7     // comandos
8     System.Console.WriteLine("Mensagem número " + i);
9
10    // atualização
11    i++;
12 }
```

Podemos reescrever esse código com a instrução de repetição `for`.

```

1 // inicialização; condição; atualização
2 for(int i = 1; i <= 100; i++)
3 {
4     // comandos
5     System.Console.WriteLine("Mensagem número " + i);
6 }
```

Perceba que o código ficou mais compacto sem prejudicar a compreensão. Na linha em destaque, declaramos e inicializamos a variável `i` (`int i = 1`), definimos a condição de execução (`i <= 100`) e definimos que ao final de cada iteração devemos atualizar a variável `i` (`i++`). Diferentemente do `while`, no `for`, a inicialização, a condição e a atualização do laço são definidas na mesma linha.



Mais Sobre

Vimos que a instrução `for` possui 3 argumentos: inicialização, condição e atualização. Esses argumentos podem ser mais complexos do que os utilizados anteriormente. Podemos declarar e/ou inicializar diversas variáveis na inicialização. Podemos definir condições mais sofisticadas com uso dos operadores lógicos. Podemos atualizar o valor de diversas variáveis na atualização. Veja um exemplo.

```
1 for(int i = 1, j = 2; i % 2 != 0 || j % 2 == 0; i += j, j += i)
2 {
3     // comandos
4 }
```



Mais Sobre

Os três argumentos da instrução `for` (inicialização, condição e atualização) são opcionais. Consequentemente, o seguinte código é válido apesar de ser estranho no primeiro momento.

```
1 for(;;)
2 {
3     // comandos
4 }
```

O segundo argumento do `for`, a condição, possui o valor padrão `true`.



Pare para pensar...

Sabendo que o segundo argumento do `for`, a condição, possui o valor padrão `true`, como podemos interromper a execução do laço do exemplo a seguir?

```
1 for(;;)
2 {
3     // comandos
4 }
```



Simulação

Novamente, vamos simular a execução de um programa que gera bilhetes de loteria. Mas agora, vamos utilizar a instrução de repetição `for`. Para não alongar muito a simulação, apenas 3 bilhetes serão gerados. Esses bilhetes devem ser numerados sequencialmente iniciando com o número 1000.

- Na linha 1, a variável `número` é declarada e inicializada com o valor 1000.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

numero = 1000

- 2 Na linha 1, a condição do for é testada. Como o valor da variável numero é menor ou igual a 1002, a condição devolve true.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

numero = 1000

- 3 Como a condição da linha 1 devolveu true, o corpo do for será executado e a mensagem “Bilhete 1000” é exibida no terminal.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

numero = 1000

Bilhete 1000

- 4 Agora, o fluxo de execução volta para a linha 1 e a atualização do for é executada. Dessa forma, a variável numero é incrementada para 1001.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

numero = 1001

Bilhete 1000

- 5 Depois da atualização, a condição do for é testada novamente. Mais uma vez, o valor da variável numero é menor ou igual a 1002. Dessa forma, a condição devolve true.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

numero = 1001

Bilhete 1000

- 6 Como a condição da linha 1 devolveu true, o corpo do for será executado. Ao executar a linha 3, a mensagem “Bilhete 1001” é exibida no terminal.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
● Bilhete 1001

numero = 1001

- 7 Mais uma vez, o fluxo de execução volta para a atualização do `for` da linha 1. Dessa forma, a variável `numero` é incrementada para 1002.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001

numero = 1002

- 8 Agora, a condição do `for` é testada novamente. O valor da variável `numero` ainda é menor ou igual a 1002. Dessa forma, a condição devolve `true`.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001

numero = 1002

- 9 Como a condição da linha 1 devolveu `true`, o corpo do `for` será executado. Ao executar a linha 3, a mensagem “Bilhete 1002” é exibida no terminal.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
● Bilhete 1002

numero = 1002

- 10 Mais uma vez, o fluxo de execução retorna para executar a atualização do `for` da linha 1. Assim, a variável `numero` é incrementada para 1003.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
Bilhete 1002

numero = 1003

- 11 Agora, a condição do `for` é testada novamente. Finalmente, o valor da variável `numero` não é menor ou igual a 1002. Dessa forma, a condição devolve `false`.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
Bilhete 1002

numero = 1003

- 12 Como a condição do `for` da linha 1 é falsa, o laço é interrompido e o fluxo de execução “pula” para a linha 5. Ao executar essa linha, a mensagem “FIM” é exibida no terminal.

```

1 for(int numero = 1000; numero <= 1002; numero++)
2 {
3     System.Console.WriteLine("Bilhete " + numero);
4 }
5 System.Console.WriteLine("FIM");

```

Bilhete 1000
Bilhete 1001
Bilhete 1002
● FIM

numero = 1003

4.9 Instruções de Repetição Encadeadas



Considere o programa de computador que gera os ingressos das apresentações de um determinado teatro. Esse teatro foi dividido em 4 setores com 200 cadeiras cada. Os ingressos devem conter o número do setor e o número da cadeira. Podemos utilizar laços encadeados para implementar esse programa.

```

1 for(int i = 1; i <= 4; i++)
2 {
3     for(int j = 1; j <= 200; j++)
4     {
5         System.Console.WriteLine("SETOR: " + i + " CADEIRA: " + j);
6     }
7 }

```

No exemplo acima, para cada iteração do laço externo, há 200 iterações do laço interno. Portanto, o corpo do laço interno executa 800 vezes. Esse valor é exatamente a quantidade de ingressos.

Além de encadear `fors`, podemos encadear `whiles`. Veja algumas variações do exemplo anterior.

```

1 int i = 1;
2 while(i <= 4)
3 {
4     int j = 1;
5     while(j <= 200)
6     {
7         System.Console.WriteLine("SETOR: " + i + " CADEIRA: " + j);
8         j++;
9     }
10    i++;
11 }

```

```

1 int i = 1;
2 while(i <= 4)
3 {
4     int j = 1;
5     for(int j = 1; j <= 200; j++)
6     {
7         System.Console.WriteLine("SETOR: " + i + " CADEIRA: " + j);
8     }
9     i++;
10 }
```

```

1 for(int i = 1; i <= 4; i++)
2 {
3     int j = 1;
4     while(j <= 200)
5     {
6         System.Console.WriteLine("SETOR: " + i + " CADEIRA: " + j);
7         j++;
8     }
9 }
```

Simulação



A seguir, simulamos a execução de dois laços do tipo `for` encadeados.

- Na linha 1, temos a execução de um laço `for`. Primeiramente, ocorre a declaração de uma variável do tipo `int` chamada `i`, que é inicializada com o valor 0.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

`i = 0`

- Em seguida, a condição do laço é verificada. Como o valor armazenado na variável `i` é 0, a operação `i <= 2` devolve `true`.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

`i = 0`

- Como a condição do laço da linha 1 é verdadeira, a linha 3 é executada. Nessa linha, temos um laço `for`. Assim, sua execução começa pela inicialização. A variável `j` do tipo `int` é declarada e inicializada com o valor de `i+1`. Como `i` armazena o valor 0, a variável `j` é inicializada com o valor 1.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0

j = 1

- 4 Em seguida, a condição do laço da linha 3 é verificada. Como a variável j armazena o valor 1, a operação j <= 2 devolve true.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0

j = 1

- 5 Como a condição do laço da linha 3 é verdadeira, a linha 5 é executada. Assim, os valores armazenados nas variáveis i e j são exibidos na saída padrão.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1

i = 0

j = 1

- 6 Após a execução do corpo do laço da linha 3, o fluxo é desviado para a atualização do laço. Assim, a operação j++ é executada e a variável j passa a armazenar o valor 2.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1

i = 0

j = 2

- 7 Após a atualização do laço da linha 3, a condição desse laço é verificada. Como a variável j armazena o valor 2, a operação j <= 2 devolve true.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1

```

i = 0

j = 2

- 8** Como a condição do laço da linha 3 é verdadeira, o corpo desse laço é executado. Assim, a instrução da linha 5 é executada e os valores das variáveis i e j são exibidos na saída padrão.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
● i = 0, j = 2

```

i = 0

j = 2

- 9** Após a execução do corpo do laço da linha 3, o fluxo é direcionado para a atualização do laço. Assim, a operação j++ é executada e a variável j passa a armazenar o valor 3.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 0

j = 3

- 10** Após a atualização do laço da linha 3, a condição desse laço é verificada. Como a variável j armazena o valor 3, o resultado da operação j <= 2 é false e a execução desse laço é encerrada.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 0

j = 3

- 11** Ao final da execução do corpo do laço da linha 1, o fluxo de execução é direcionado para a atualização desse laço. Assim, a operação i++ é processada e o valor 1 é atribuído à variável i.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 1

- 12** Após a atualização do laço da linha 1, a condição desse laço é avaliada. Como o valor armazenado em i é 1, a operação `i <= 2` devolve true.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 1

- 13** Como a condição do laço da linha 1 é verdadeira, o corpo desse laço é executado. Assim, o laço da linha 3 é executado. O laço da linha 3 começa pela declaração e inicialização da variável j do tipo int. Como a variável i armazena o valor 1, o resultado da operação `i+1` é 2. Portanto, o valor 2 é atribuído à variável j.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 1

j = 2

- 14** Em seguida, a condição do laço da linha 3 é testada. Como o valor armazenado na variável j é 2, o resultado da operação `j <= 2` é true.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }

i = 0, j = 1
i = 0, j = 2

```

i = 1

j = 2

- 15** Como a condição do laço da linha 3 é verdadeira, o corpo desse laço é executado. Assim, a instrução da linha 5 é processada e os valores armazenados nas variáveis i e j são exibidos na saída

padrão.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1
i = 0, j = 2
● i = 1, j = 2

i = 1

j = 2

- 16 Ao final da execução do corpo do laço da linha 3, o valor da variável j é atualizado. Como j armazena o valor 2, a operação j++ atribui o valor 3 à variável j.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1
i = 0, j = 2
● i = 1, j = 2

i = 1

j = 3

- 17 Após a atualização do laço da linha 3, sua condição é testada. Como a variável j armazena o valor 3, a operação j <= 2 devolve false e a execução desse laço é encerrada.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1
i = 0, j = 2
● i = 1, j = 2

i = 1

j = 3

- 18 Depois da execução do corpo do laço da linha 1, o valor da variável i é atualizado. Como i armazena o valor 1, a operação i++ atribui o valor 2 à variável i.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

i = 0, j = 1
i = 0, j = 2
● i = 1, j = 2

i = 2

- 19 Após a atualização do valor da variável i, a condição do laço da linha 1 é verificada. Como a

variável i armazena o valor 2, o resultado da operação $i \leq 2$ é true.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

```

i = 0, j = 1
i = 0, j = 2
i = 1, j = 2
```

i = 2

20 Como a condição do laço da linha 1 é verdadeira, o corpo desse laço é executado. Sendo assim, o laço da linha 3 é processado. Isso começa com a declaração e inicialização da variável j. Como i armazena o valor 2, a variável j é inicializada com o valor 3.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

```

i = 0, j = 1
i = 0, j = 2
i = 1, j = 2
```

i = 2

j = 3

21 Em seguida, a condição do laço da linha 3 é testada. Como a variável j armazena o valor 3, a operação $j \leq 2$ devolve false. Assim, a execução desse laço é encerrada.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

```

i = 0, j = 1
i = 0, j = 2
i = 1, j = 2
```

i = 2

j = 3

22 Ao final da execução do corpo do laço da linha 1, o valor da variável i é atualizado. Como i armazena o valor 2, essa variável passa a armazenar o valor 3 após a operação $i++$.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

```

i = 0, j = 1
i = 0, j = 2
i = 1, j = 2
```

i = 3

- 23 Em seguida, a condição do laço da linha 1 é verificada. Como a variável `i` armazena o valor 3, a operação `i <= 2` devolve `false`. Assim, a execução do laço da linha 1 é encerrada.

```

1 for (int i = 0; i <= 2; i++)
2 {
3     for (int j = i + 1; j <= 2; j++)
4     {
5         System.Console.WriteLine("i = " + i + ", j = " + j);
6     }
7 }
```

`i = 0, j = 1
i = 0, j = 2
i = 1, j = 2`

`i = 3`

4.10 Instrução break



Considere um jogo de dados no qual o jogador ganha quando a soma dos números obtidos em lançamentos consecutivos de um dado ultrapassar um determinado valor. Antes de começar o jogo, é necessário definir a quantidade máxima de lançamentos e o valor que deve ser ultrapassado para obter a vitória. Eventualmente, se o valor desejado for ultrapassado antes do último lançamento, não é necessário continuar jogando o dado pois a vitória já está garantida. Podemos implementar um programa de computador para simular a execução desse jogo. Nesse programa, podemos utilizar a instrução `break` para interromper os lançamentos se o valor desejado for ultrapassado.

A instrução `break` não é uma instrução de repetição, mas está fortemente relacionada às instruções `while` e `for`. Ela pode ser utilizada para forçar a parada de um laço.

No exemplo abaixo, a lógica para simular o jogo descrito anteriormente considera que a quantidade máxima de lançamentos é 100 e o valor desejado é 360.

```

1 int soma = 0;
2
3 System.Random gerador = new System.Random();
4
5 for(int i = 1; i <= 100; i++)
6 {
7     System.Console.WriteLine("Lançamento: " + i);
8
9     int numero = (int)(gerador.NextDouble() * 6 + 1);
10
11    System.Console.WriteLine("Número: " + numero);
12
13    soma += numero;
14
15    System.Console.WriteLine("Soma: " + soma);
16
17    if(soma > 360)
18    {
19        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
20        break;
21    }
22}
23 System.Console.WriteLine("Jogue Novamente");
```

O trecho `(int)(gerador.NextDouble() * 6 + 1)` gera um número inteiro entre 1 e 6. Esse trecho simula o lançamento de um dado. A variável `soma` acumula os números gerados. A instrução `if` é utilizada para verificar se a soma ultrapassou o valor desejado. Dentro do corpo do `if`, utilizamos o comando `break` para interromper as iterações do laço `for` pois, quando a soma ultrapassa 360, a

vitória já está garantida.



Importante

A instrução `break` deve aparecer somente dentro do corpo de um laço ou dentro do corpo de um `switch`. Veremos a instrução `switch` neste capítulo.



Importante

Quando a instrução `break` é utilizada dentro de uma cadeia de laços, ela interrompe o laço mais interno. No exemplo abaixo, três laços encadeados foram definidos.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         for(int k = 0; k < 10; k++)
6         {
7             ...
8         }
9     }
10 }
```

Se a instrução `break` for utilizada em qualquer lugar dentro do corpo do terceiro laço, esse laço será interrompido.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         for(int k = 0; k < 10; k++)
6         {
7             ...
8             break; // interrompe o terceiro laço
9             ...
10        }
11    }
12 }
```

Caso contrário, se a instrução `break` for utilizada dentro do corpo do segundo laço mas fora do terceiro laço, o segundo laço será interrompido.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         ...
6         break; // interrompe o segundo laço
7         ...
8         for(int k = 0; k < 10; k++)
9         {
10            ...
11        }
12    }
13 }
```

Caso contrário, se a instrução `break` for utilizada dentro do primeiro laço mas fora do segundo laço, o primeiro laço será interrompido.

```

1 for(int i = 0; i < 10; i++)
2 {
```

```

3 ...
4 break; // interrompe o primeiro laço
5 ...
6 for(int j = 0; j < 10; j++)
7 {
8     for(int k = 0; k < 10; k++)
9     {
10    }
11 }
12 }
```

Simulação



Vamos simular a execução do jogo de dados descrito anteriormente. Para não alongar muito a simulação, considere que o número máximo de lançamentos é 2 e o valor que deve ser ultrapassado é 7.

- Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");
```

- Na linha 2, a variável `soma` é declarada e inicializada com o valor 0.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

soma = 0

- 3 Na linha 3, a inicialização do for é executada. A variável i é criada e inicializada com o valor 1.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

soma = 0

i = 1

- 4 Na linha 3, a condição do for é testada. Como o valor da variável i é menor ou igual a 2, a condição devolve true.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

soma = 0

i = 1

- 5 Como a condição da linha 3 devolveu true, o corpo do for será executado. Ao executar a linha 5, a mensagem “Lançamento: 1” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

soma = 0

i = 1

- 6 Na sequência, a linha 6 é executada. Um número aleatório entre 1 e 6 é gerado e armazenado na variável numero. Suponha que o número 5 tenha sido gerado.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1

soma = 0

i = 1

numero = 5

- 7 Prosseguindo, a linha 8 é executada e a mensagem “Número: 5” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5

soma = 0

i = 1

numero = 5

- 8 Adiante, a linha 9 é executada e o valor da variável `numero` é incrementado na variável `soma`. Dessa forma, a variável `soma` passa a armazenar o valor 5.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5

soma = 5

i = 1

numero = 5

- 9 Na sequência, a linha 11 é executada e a mensagem “Soma: 5” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5

soma = 5

i = 1

numero = 5

- 10 Prosseguindo, a linha 12 é executada e a condição do if é testada. Como o valor da variável soma não é maior do que 7, a condição devolve false.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5

soma = 5

i = 1

numero = 5

- 11 Como a condição da linha 12 é falsa, o corpo do if não será executado e o fluxo de execução vai para a atualização do for na linha 3. Dessa forma, a variável i é incrementada para 2.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5

soma = 5

i = 2

- 12 Novamente, na linha 3, a condição do for é testada. Como o valor da variável i é menor ou igual a 2, a condição devolve true.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5

soma = 5

i = 2

- 13** Como a condição da linha 3 devolveu true, o corpo do for será executado. Ao executar a linha 5, a mensagem “Lançamento: 2” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5
● Lançamento: 2

soma = 5

i = 2

- 14** Na sequência, a linha 6 é executada. Um número aleatório entre 1 e 6 é gerado e armazenado na variável numero. Suponha que o número gerado tenha sido o 3.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2

soma = 5

i = 2

numero = 3

- 15 Prosseguindo, a linha 8 é executada e a mensagem “Número: 3” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2
Número: 3

soma = 5

i = 2

numero = 3

- 16 Adiante, a linha 9 é executada e o valor da variável `numero` é incrementado na variável `soma`. Dessa forma, a variável `soma` passa a armazenar o valor 8.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2
Número: 3

soma = 8

i = 2

numero = 3

- 17 Na sequência, a linha 11 é executada e a mensagem “Soma: 8” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2
Número: 3
● Soma: 8

soma = 8

i = 2

numero = 3

- 18 Prosseguindo, a linha 12 é executada e a condição do if é testada. Como o valor da variável soma é maior do que 7, a condição devolve true.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
 Número: 5
 Soma: 5
 Lançamento: 2
 Número: 3
 Soma: 8

soma = 8

i = 2

numero = 3

- 19** Como a condição da linha 12 é verdadeira, o corpo do if será executado. Ao executar a linha 14, a mensagem “Você ganhou com 2 lançamentos” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

Lançamento: 1
 Número: 5
 Soma: 5
 Lançamento: 2
 Número: 3
 Soma: 8
 ● Você ganhou com 2 lançamentos

soma = 8

i = 2

numero = 3

- 20** Agora, a linha 15 é executada. Dessa forma, a instrução break interrompe o laço.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2
Número: 3
Soma: 8
Você ganhou com 2 lançamentos

```

soma = 8

i = 2

numero = 3

- 21 Por fim, a linha 18 é executada e a mensagem “Jogue Novamente” é exibida no terminal.

```

1 System.Random gerador = new System.Random();
2 int soma = 0;
3 for(int i = 1; i <= 2; i++)
4 {
5     System.Console.WriteLine("Lançamento: " + i);
6     int numero = (int)(gerador.NextDouble() * 6 + 1);
7
8     System.Console.WriteLine("Número: " + numero);
9     soma += numero;
10
11    System.Console.WriteLine("Soma: " + soma);
12    if(soma > 7)
13    {
14        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
15        break;
16    }
17 }
18 System.Console.WriteLine("Jogue Novamente");

```

```

Lançamento: 1
Número: 5
Soma: 5
Lançamento: 2
Número: 3
Soma: 8
Você ganhou com 2 lançamentos
● Jogue Novamente

```

soma = 8

4.11 Instrução continue



Considere uma variação do jogo de dados proposto anteriormente. Nessa nova versão, somente valores pares devem ser somados. Em outras palavras, os valores ímpares devem ser descartados. Nesse caso, podemos utilizar a instrução `continue`. Essa instrução permite que, durante a execução

de um laço, uma determinada iteração seja abortada, fazendo com que o fluxo de execução continue para a próxima iteração.

O código abaixo simula o jogo de dados discutido anteriormente com a variação proposta.

```

1 int soma = 0;
2
3 System.Random gerador = new System.Random();
4
5 for(int i = 1; i <= 100; i++)
6 {
7     System.Console.WriteLine("Lançamento: " + i);
8
9     int numero = (int)(gerador.NextDouble() * 6 + 1);
10
11    System.Console.WriteLine("Número: " + numero);
12
13    if(numero % 2 != 0)
14    {
15        continue;
16    }
17
18    soma += numero;
19
20    System.Console.WriteLine("Soma: " + soma);
21
22    if(soma > 180)
23    {
24        System.Console.WriteLine("Você ganhou com " + i + " lançamentos");
25        break;
26    }
27 }
28 System.Console.WriteLine("Jogue Novamente");

```

No trecho destacado, calculamos o resto da divisão do número gerado aleatoriamente por dois. Além disso, na condição do `if`, verificamos se esse valor é diferente de zero. Se essa condição for verdadeira, significa que o número gerado aleatoriamente é ímpar e consequentemente deve ser descartado. No corpo do `if`, utilizamos a instrução `continue` para abortar a iteração atual.



Importante

Quando aplicada a laços `while`, a instrução `continue` “pula” para a condição do laço. Por outro lado, quando aplicada a laços `for`, ela “pula” para a atualização do laço.



Importante

Quando a instrução `continue` é utilizada dentro de uma cadeia de laços, ela afeta o laço mais interno. No exemplo abaixo, três laços encadeados foram definidos.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         for(int k = 0; k < 10; k++)
6         {
7         }
8     }
9 }

```

Se a instrução `continue` for utilizada em qualquer lugar dentro do corpo do terceiro laço, esse laço

será afetado.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         for(int k = 0; k < 10; k++)
6         {
7             ...
8             continue; // afeta o terceiro laço
9             ...
10        }
11    }
12 }
```

Caso contrário, se a instrução `continue` for utilizada dentro do corpo do segundo laço mas fora do terceiro laço, o segundo laço será afetado.

```

1 for(int i = 0; i < 10; i++)
2 {
3     for(int j = 0; j < 10; j++)
4     {
5         ...
6         continue; // afeta o segundo laço
7         ...
8         for(int k = 0; k < 10; k++)
9         {
10            }
11        }
12 }
```

Caso contrário, se a instrução `continue` for utilizada dentro do terceiro laço mas fora do segundo laço, o primeiro laço será afetado.

```

1 for(int i = 0; i < 10; i++)
2 {
3     ...
4     continue; // afeta o laço mais externo
5     ...
6     for(int j = 0; j < 10; j++)
7     {
8         for(int k = 0; k < 10; k++)
9         {
10            }
11        }
12 }
```

Simulação



Vamos simular a execução de um programa que gera aleatoriamente dois números inteiros entre 1 e 100 e exibe no terminal apenas os ímpares.

- 1 Na linha 1, um objeto do tipo `System.Random` é criado. Esse objeto será utilizado para gerar números aleatórios.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

- 2 Em seguida, na linha 2, a variável `i` é declarada e inicializada com o valor 1.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 1

- 3 Na sequência, a condição do `for` é testada. Como o valor da variável `i` é menor ou igual a 2, essa condição devolve true.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 1

- 4 O corpo do `for` é executado porque a condição da linha 2 devolveu true. Ao executar a linha 4, um número aleatório entre 1 e 100 é gerado e armazenado na variável `numero`. Suponha que o valor gerado seja 38.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 1

numero = 38

- 5 Na linha 5, verificamos se o resto da divisão do valor da variável `numero` por 2 é igual a 0. Como essa variável está armazenando o valor 38, a condição do `if` devolve `true`, pois o resto da divisão de 38 por 2 é 0.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 1
numero = 38

- 6 Como a condição da linha 5 devolveu `true`, o corpo do `if` é executado. Ao executar a linha 7, a instrução `continue` “pula” para a próxima iteração.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 1
numero = 38

- 7 Devido ao desvio causado pela instrução `continue`, o fluxo de execução vai para a atualização do `for` na linha 2. Dessa forma, a variável `i` é incrementada para 2.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; [i++])
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 2

- 8 Na sequência, a condição do `for` é testada. Como valor da variável `i` é menor ou igual a 2, essa condição devolve `true`.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; [i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 2

- 9 O corpo do `for` é executado porque a condição da linha 2 devolveu `true`. Ao executar a linha 4, um número aleatório entre 1 e 100 é gerado e armazenado na variável `numero`. Suponha que o valor gerado seja 97.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 2

numero = 97

- 10 Na linha 5, verificamos se o resto da divisão do valor da variável `numero` por 2 é igual a 0. Como essa variável está armazenando o valor 97, a condição do `if` devolve `false`, pois o resto da divisão de 97 por 2 não é 0.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 2

numero = 97

- 11 Como a condição da linha 5 devolveu `false`, o corpo do `if` não é executado. Dessa forma, o fluxo de execução vai direto para a linha 9 e o valor 97 é exibido no terminal.

```

1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 2

numero = 97

- 12 Prosseguindo, o fluxo de execução vai para a atualização do `for` na linha 2. Dessa forma, a variável `i` é incrementada para 3.

```
1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 3

97

- 13 Na sequência, a condição do for é testada. Como valor da variável i não é menor ou igual a 2, essa condição devolve false. Dessa forma, o laço é finalizado.

```
1 System.Random gerador = new System.Random();
2 for(int i = 1; i <= 2; i++)
3 {
4     int numero = (int)(gerador.NextDouble() * 100 + 1);
5     if(numero % 2 == 0)
6     {
7         continue;
8     }
9     System.Console.WriteLine(i);
10 }
```

i = 3

97

4.12 Blocos Sem Chaves



Normalmente, os blocos de código associados às instruções de decisão `if` e `else` ou às instruções de repetição `while` e `for` são delimitados com chaves “`{ }`”. Contudo, as chaves podem ser omitidas nos blocos que possuem apenas um comando. Veja alguns exemplos.

```
1 if(a < 10)
2   a = a * 2 + 1;
3 else
4   a = a / 2 + 1;
```

```
1 while(a < 10)
2   a = a * 2 + 1;
```

```
1 for(int i = 1; i < 10; i++)
2   a = a * 2 + 1;
```



Lembre-se

Na linguagem C#, apenas blocos de código com apenas um comando podem ser associados às instruções de decisão `if` e `else` ou às instruções de repetição `while` e `for`.

Normalmente, não delimitar com chaves os blocos de código com dois ou mais comandos gera erros de lógica ou até mesmo erros de compilação. Para evitar esses problemas, a utilização das chaves mesmo em blocos com apenas um comando é recomendada.

Considere o seguinte trecho de código.

```
1 if(a < 10)
2   a = a * 2 + 1;
3 else
4 {
5   if(a < 20)
6     a = a * 3 + 1;
7   else
8     a = a * 4 + 1;
9 }
```

O trecho em destaque, apesar de conter diversas linhas de código, é considerado um comando único. Dessa forma, podemos omitir as chaves que envolvem esse trecho. Reescrevendo o código, teríamos o seguinte resultado:

```
1 if(a < 10)
2   a = a * 2 + 1;
3 else if(a < 20)
4   a = a * 3 + 1;
5 else
6   a = a * 4 + 1;
```

Os leitores mais desavisados desse código podem assumir a existência da instrução `else if`. Contudo, essa instrução não existe na linguagem C#. Na verdade, nesse exemplo, o segundo `if` pertence ao corpo do primeiro `else`.



4.13 “Laços Infinitos”

Um laço é interrompido quando a condição de um laço for falsa ou quando utilizamos a instrução `break`. Dessa forma, considere os seguintes laços.

```
1 int i = 1;
2 while(i < 10)
3 {
4     System.Console.WriteLine("K19");
5 }
```

```
1 for(int i = 1; i < 10;)
2 {
3     System.Console.WriteLine("K19");
4 }
```

Observe que a condição nunca devolverá o valor `false`. Como a variável `i` é inicializada com o valor 1 e não é mais atualizada, a condição `i < 10` será sempre verdadeira. Dessa forma, os laços acima nunca serão interrompidos. Eles serão executados indefinidamente. Esses laços são chamados popularmente de “laços infinitos”.



4.14 Instrução switch

A instrução `switch` é uma instrução de decisão. Ela permite definir o que deve ser executado quando o valor de uma determinada chave for igual a certos valores. Veja a seguir a sintaxe dessa instrução.

```
1 switch([chave])
2 {
3     case [expressao1]: [bloco1]
4     case [expressao2]: [bloco2]
5     case [expressao3]: [bloco3]
6     default: [bloco4]
7 }
```

Código C# 4.144: Sintaxe da instrução `switch`

Vamos considerar a utilização da instrução `switch` com chaves dos tipos `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char` e `string`.

Para cada caso, exceto para caso padrão (`default`), é necessário definir uma expressão constante compatível com o tipo da chave do `switch`. Não é permitido definir dois casos com expressões de mesmo valor. Também não é permitido definir mais do que um caso padrão.

Quando o valor da chave for igual ao valor da “`expressao1`”, o bloco 1 será executado. Quando o valor da chave for igual ao valor da “`expressao2`”, os bloco 2 será executado. Quando o valor da chave for igual ao valor da “`expressao3`”, o bloco 3 será executado. Quando o valor da chave não for igual ao valor da “`expressao1`”, “`expressao2`” ou “`expressao3`”, apenas o bloco 4 será executado.

No final de cada bloco, é necessário utilizar a instrução `break` para interromper o `switch` ou acionar outro bloco com o comando `goto`. No exemplo abaixo, quando a chave for igual a 0, o programa exibirá 0, 2 e “`default`”. Quando a chave for igual a 1, o programa exibirá apenas 1. Quando a chave

for igual a 2, o programa exibirá 2 e “default”. Caso contrário, o programa exibirá apenas “default”.

```

1 System.Random gerador = new System.Random();
2 int chave = (int)(gerador.NextDouble() * 10);
3 switch(chave)
4 {
5     case 0:
6         System.Console.WriteLine(0);
7         goto case 2;
8     case 1:
9         System.Console.WriteLine(1);
10    break;
11    case 2:
12        System.Console.WriteLine(2);
13        goto default;
14    default:
15        System.Console.WriteLine("default");
16 }
```



Mais Sobre

O número de casos que podem ser declarados em um `switch` é ilimitado. Além disso, o caso padrão (`default`) é opcional. Tecnicamente, um `switch` sem nenhum caso é válido. Portanto, o código a seguir está correto.

```

1 int x = 1;
2 switch(x)
3 {
4 }
5 }
```

Código C# 4.146: Um switch vazio

O mesmo bloco pode ser associado a vários casos. No exemplo abaixo, se a nota for ‘A’ ou ‘B’, a mensagem “Ótimo” será exibida na saída padrão. Se a nota for ‘C’, a mensagem “Regular” será exibida na saída padrão. Se a nota for ‘D’ ou ‘E’, a mensagem “Ruim” será exibida na saída padrão.

```

1 char nota = 'A';
2 switch(nota)
3 {
4     case 'A':
5     case 'B':
6         System.Console.WriteLine("Ótimo");
7         break;
8     case 'C':
9         System.Console.WriteLine("Regular");
10    break;
11    case 'D':
12    case 'E':
13        System.Console.WriteLine("Ruim");
14        break;
15 }
```

Código C# 4.147: Um bloco associado a vários casos



Simulação

Nessa simulação, mostraremos um exemplo de utilização da instrução `switch`.

- 1 Ao executar a linha 1, uma variável do tipo `int` chamada `tipoDeSeguro` é declarada e inicializada com o valor 2.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }
```

`tipoDeSeguro = 2`

- 2 Em seguida, a linha 2 é executada. A instrução `switch` verifica o valor armazenado na variável `tipoDeSeguro` para decidir qual `case` deve ser selecionado.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }
```

`tipoDeSeguro = 2`

- 3 Como o valor da variável `tipoDeSeguro` armazena o valor 2, o fluxo de execução é direcionado para a linha 8. Na execução dessa linha, a mensagem “GUINCHO E VIDROS” é exibida na saída padrão.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }
```

`tipoDeSeguro = 2`

● GUINCHO E VIDROS

- 4 A próxima linha a ser executada é a linha 9. Ao executar essa linha, o fluxo de execução é direcionado para a linha 11.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }

```

GUINCHO E VIDROS

tipoDeSeguro = 2

- 5 A próxima linha a ser executada é a linha 11. Ao executar essa linha, a mensagem “ROUBO E COLISÃO” será exibida na saída padrão.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }

```

GUINCHO E VIDROS
● ROUBO E COLISÃO

tipoDeSeguro = 2

- 6 A próxima linha a ser executada é a linha 12. Ao executar essa linha, a instrução `break` interrompe o `switch`.

```

1 int tipoDeSeguro = 2;
2 switch(tipoDeSeguro)
3 {
4     case 1:
5         System.Console.WriteLine("CARRO RESERVA");
6         goto case 2;
7     case 2:
8         System.Console.WriteLine("GUINCHO E VIDROS");
9         goto case 3;
10    case 3:
11        System.Console.WriteLine("ROUBO E COLISÃO");
12        break;
13 }

```

GUINCHO E VIDROS
ROUBO E COLISÃO

tipoDeSeguro = 2



4.15 Instrução do-while

A instrução de repetição do-while é análoga à instrução while. A sintaxe do do-while é:

```
1 do
2 {
3     bloco_de_comandos
4 } while([condição]);
```

Código C# 4.154: Sintaxe da instrução do-while

A condição deve ser um valor do tipo bool. Veja na Figura 4.5 o fluxograma da instrução do-while.

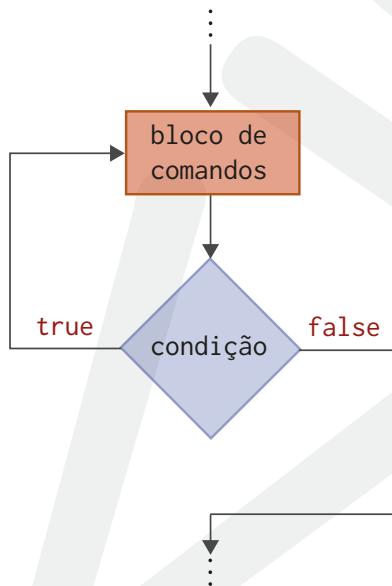


Figura 4.5: Fluxograma da instrução do-while

No exemplo abaixo, a variável `x` que é do tipo `int` foi inicializada com o valor 10. Na primeira iteração, a condição do `while` é falsa, pois o valor da variável `x` não é menor do que 10. Consequentemente, o bloco do `while` não será executado nem ao menos uma vez.

```
1 int x = 10;
2 while(x < 10)
3 {
4     System.Console.WriteLine(x);
5     x++;
6 }
```

Código C# 4.155: Um while que não executa

No exemplo abaixo, substituímos a instrução `while` pela instrução `do-while`. Nesse caso, o bloco do laço sempre será executado ao menos uma vez, pois a condição do laço é verificada somente no final das iterações e não no começo como ocorre no `while`.

```
1 int x = 10;
2 do
3 {
4     System.Console.WriteLine(x);
```

```
5     x++;
6 } while(x < 10);
```

Código C# 4.156: Utilizando a instrução do-while



Simulação

Nessa simulação, mostraremos um exemplo de utilização da instrução do-while.

- 1 Na execução da linha 1, uma variável do tipo `int` chamada `a` é declarada e inicializada com o valor 1.

```
1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");
```

`a = 1`

- 2 Em seguida, a linha 4 é executada e o valor armazenado na variável `a` é exibido na saída padrão.

```
1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");
```

`a = 1`

- 3 Agora, a linha 5 é executada e o valor armazenado na variável `a` é atualizado com o operador `++` e essa variável passa a armazenar o valor 2.

```
1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");
```

`a = 2`

- 4 Na sequência, a linha 6 é executada. Como a variável `a` armazena o valor 2, a operação `a < 3` devolve `true`. Sendo assim, o laço continua executando.

```

1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");

```

1

a = 2

- 5 Continuando a execução, a linha 4 é processada e o valor armazenado na variável a é exibido na saída padrão.

```

1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");

```

1
2

a = 2

- 6 Com a execução da linha 5, o valor da variável a é atualizado com o operador ++ e essa variável passa a armazenar o valor 3.

```

1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");

```

1
2

a = 3

- 7 Agora, a linha 6 é executada. Como a variável a armazena o valor 3, a operação a < 3 devolve false. Sendo assim, o laço terminará.

```

1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");

```

1
2

a = 3

- 8 Por fim, a linha 7 é executada e a mensagem “FIM” é exibida na saída padrão.

```

1 int a = 1;
2 do
3 {
4     System.Console.WriteLine(a);
5     a++;
6 } while(a < 3);
7 System.Console.WriteLine("FIM");

```

1
2
● FIM

a = 3

4.16 Unreachable Code



O compilador tenta identificar instruções que nunca seriam executadas. Algumas dessas instruções geram o aviso **unreachable code**. No exemplo abaixo, a instrução `contador++` nunca seria executada, pois ela é precedida pela instrução `break`. Nesse caso, ocorreria um unreachable code.

```

1 int contador = 0;
2 while(contador < 100)
3 {
4     break;
5     contador++; // aviso - unreachable code
6 }

```

Analogamente, o mesmo problema ocorre no código abaixo. A constante `contador` foi inicializada com o valor `10`. Por ser uma constante, não podemos alterar o seu conteúdo. Dessa forma, o corpo do `while` nunca seria executado. Nesse caso, novamente ocorreria um unreachable code. Se `contador` fosse uma variável, nenhum aviso seria gerado pelo compilador.

```

1 const int contador = 10;
2 while(contador < 10)
3 {
4     System.Console.WriteLine(contador); // aviso - unreachable code
5 }

```

Também podemos observar esse problema no exemplo abaixo. A constante `contador` foi inicializada com o valor `0`. Dessa forma, a condição do `while` é sempre verdadeira. Como não há nada que interrompa o laço, as linhas depois do corpo do `while` nunca seriam executadas. Nesse caso, mais uma vez ocorreria um unreachable code.

```

1 const int contador = 0;
2 while(contador < 10)
3 {
4     System.Console.WriteLine(contador);
5 }
6 System.Console.WriteLine("FIM"); // aviso - unreachable code

```

4.17 Erro: Não utilizar condições booleanas



Um erro de compilação comum em C# ocorre quando não utilizamos condições booleanas nas instruções `if`, `while` ou `for`.

```

1 class Programa
2 {
3     static void Main()
4     {
5         System.Random gerador = new System.Random();
6         double a = gerador.NextDouble();
7         double b = gerador.NextDouble();
8
9         if (a + b)
10        {
11            a *= 2;
12        }
13        else
14        {
15            a /= 2;
16        }
17    }
18 }
```

Código C# 4.168: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```
Programa.cs(10,7): error CS0029: Cannot implicitly convert type 'double' to
'bool'
```

Terminal 4.90: Erro de compilação

4.18 Erro: Else sem if



Um erro de compilação comum em C# ocorre quando o comando `else` não está associado ao comando `if`.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 1;
6         if(a < 10)
7             System.Console.WriteLine(a);
8             System.Console.WriteLine("Menor");
9         else
10            System.Console.WriteLine(a);
11            System.Console.WriteLine("Maior");
12     }
13 }
```

Código C# 4.169: Programa.cs

Observe que o corpo do comando `if` possui apenas uma instrução, já que as chaves não foram utilizadas. Dessa forma, a instrução `System.Console.WriteLine("Menor")` separa o comando `else` do `if`.

```
Programa.cs(9,3): error CS1525: Invalid expression term 'else'
Programa.cs(9,7): error CS1002: ; expected
```

Terminal 4.91: Erro de compilação

4.19 Erro: Else com condição



Um erro de compilação em C# ocorre quando a instrução `else` é seguida por uma condição.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 1;
6         if(a < 10)
7             System.Console.WriteLine("Menor");
8         else(a > 10)
9             System.Console.WriteLine("Maior");
10    }
11 }
```

Código C# 4.170: Programa.cs

No código acima, ocorre um erro de compilação na linha 8, pois a instrução `else` não admite uma condição como na instrução `if`.

`Programa.cs(8,15): error CS1002: ; expected`

Terminal 4.92: Erro de compilação



4.20 Erro: Ponto e vírgula excedente

Um erro de lógica comum em C# pode ocorrer quando o caractere “;” é adicionado em excesso.

```

1 class Programa
2 {
3     static void Main()
4     {
5         for (int i = 0; i < 10; i++);;
6         {
7             System.Console.WriteLine("*****");
8         }
9     }
10 }
```

Código C# 4.171: Programa.cs

Observe o caractere “;” depois dos argumentos do `for`. Na verdade, não há erros de compilação nesse código. Contudo, podemos considerar que há um erro de lógica, pois o laço não tem **corpo**. O bloco depois do `for` executará apenas uma vez pois não está associado ao laço.

Veja o resultado da execução desse programa.

`*****`

Terminal 4.93: Erro de lógica



4.21 Erro: “Laço infinito”

Um erro de lógica comum em C# pode ocorrer quando a condição de um laço é sempre verdadeira.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 1;
6         while (a < 10)
7         {
8             System.Console.WriteLine(a);
9         }
10    }
11 }

```

Código C# 4.172: Programa.cs

Observe que a condição `a < 10` é sempre verdadeira porque a variável `a` foi inicializada com o valor 1 e ela nunca é alterada. Consequentemente, o corpo do `while` será executado indefinidamente.

4.22 Erro: Chave do switch com tipos incompatíveis



Um erro de compilação comum em C# ocorre quando a chave da instrução `switch` é uma expressão de um tipo incompatível.

```

1 class Programa
2 {
3     static void Main()
4     {
5         double a = 1;
6         switch (a)
7         {
8             case 0.0: System.Console.WriteLine(0);
9             case 1.0: System.Console.WriteLine(1);
10        }
11    }
12 }

```

Código C# 4.173: Programa.cs

No código acima, ocorre um erro de compilação na linha 7, pois a chave da instrução `switch` é do tipo `double`.

```
Programa.cs(7,11): error CS0151: A switch expression or case label must be a bool, char, string, integral, enum, or corresponding nullable type
```

Terminal 4.94: Erro de compilação

4.23 Erro: Casos do switch com expressões não constantes



Em C#, uma expressão constante é uma expressão que deve ser avaliada em tempo de compilação. Esse tipo de expressão envolve apenas valores literais, variáveis do tipo `const` ou `enum` types. Um erro de compilação comum ocorre quando um dos casos da instrução `switch` não é uma expressão constante.

```

1 class Programa
2 {
3     static void Main()
4     {

```

```

5     int a = 1;
6     int b = 2;
7
8     switch (a)
9     {
10        case 1:
11            System.Console.WriteLine(1);
12            break;
13        case b:
14            System.Console.WriteLine(b);
15            break;
16    }
17 }
18 }
```

Código C# 4.174: Programa.cs

No código acima, ocorre um erro de compilação na linha 13, pois a expressão do segundo caso da instrução `switch` não é constante.

```
Programa.cs(13,9): error CS0150: A constant value is expected
```

Terminal 4.95: Erro de compilação

4.24 Erro: Break ou continue fora de um laço



Um erro de compilação ocorre quando a instrução `break` é utilizada fora de um laço ou de um `switch` ou quando a instrução `continue` é utilizada fora de um laço. Confira o exemplo abaixo.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 1;
6
7         if (a == 1)
8         {
9             System.Console.WriteLine("Igual a 1");
10            break;
11        }
12        else
13        {
14            System.Console.WriteLine("Diferente de 1");
15            continue;
16        }
17    }
18 }
```

Código C# 4.175: Programa.cs

No código acima, ocorrem erros de compilação nas linhas 10 e 15.

```
Programa.cs(10,4): error CS0139: No enclosing loop out of which to break or
                     continue
Programa.cs(15,4): error CS0139: No enclosing loop out of which to break or
                     continue
```

Terminal 4.96: Erro de compilação

4.25 Erro: Usar vírgula ao invés de ponto e vírgula no laço for



Ocorrem erros de compilação quando os argumentos do laço `for` são separados por vírgula ao invés de ponto e vírgula.

```
1 class Programa
2 {
3     static void Main()
4     {
5         for(int i = 0, i < 10, i++)
6         {
7             System.Console.WriteLine(i);
8         }
9     }
10 }
```

Código C# 4.176: Programa.cs

No código acima, ocorrem erros de compilação na linha 5.

```
Programa.cs(5,20): error CS1002: ; expected
Programa.cs(5,22): error CS1002: ; expected
```

Terminal 4.97: Erro de compilação

A vírgula só pode ser utilizada na separação de inicializações de variáveis no primeiro argumento do `for` ou na separação de instruções válidas no terceiro argumento do `for`.

4.26 Resumo



- 1 ► Os programas de computador utilizam as instruções de decisão para determinar se um bloco de código será executado ou não de acordo com determinada condição.
- 2 ► Para utilizar a instrução de decisão `if`, devemos definir uma condição e um corpo. O corpo é o bloco de código que será executado se e somente se a condição for `true`.
- 3 ► Para utilizar a instrução `else`, devemos definir um corpo. Essa instrução sempre está associada à instrução `if`. O corpo do `else` é executado se e somente se a condição do `if` correspondente for `false`.
- 4 ► Instruções de decisão podem ser encadeadas. Em outras palavras, podemos definir `ifs` e `elses` dentro de `ifs` ou `elses`.
- 5 ► Os programas de computador utilizam as instruções de repetição para executar repetidas vezes um determinado bloco de código.
- 6 ► Para utilizar a instrução de repetição `while`, devemos definir uma condição e um corpo. O corpo é executado se e somente se a condição for `true`. Após cada execução do corpo, a condição é reavaliada para decidir se o corpo deve ser executado novamente.

- 7 ► Para utilizar a instrução de repetição `for`, devemos definir uma inicialização, uma condição, uma atualização e um corpo. Primeiro, a inicialização é executada e em seguida a condição é avaliada. Se a condição for `true`, o corpo é executado. Caso contrário, a execução do laço é encerrada. Após cada execução do corpo, a atualização é processada e a condição é reavaliada para decidir se o corpo deve ser executado novamente.
- 8 ► Instruções de repetição podem ser encadeadas. Em outras palavras, podemos definir `whiles` e `fors` dentro de `whiles` ou `fors`.
- 9 ► A instrução `break` interrompe a execução de um laço.
- 10 ► A instrução `continue` interrompe a execução de uma iteração. No `while`, o `continue` desvia o fluxo de execução para a condição. No `for`, o `continue` desvia o fluxo de execução para a atualização.
- 11 ► Quando o corpo do `if` possui apenas um comando, ele não precisa ser delimitado com chaves. A mesma regra vale para o `else`, `while`, `do-while` e `for`.
- 12 ► O corpo de um laço `do-while` sempre é executado pelo menos uma vez.
- 13 ► Se a condição de um laço é sempre `true`, o corpo desse laço será executado repetidamente sem parar (“laços infinitos”).



ARRAY

5.1 Introdução



Considere um programa de computador que realiza cálculos matemáticos com os preços dos produtos de um supermercado. Por exemplo, esse programa calcula a média dos preços e encontra o produto mais barato.

Para manipular os preços dos produtos, dentro de um programa, esses valores devem ser armazenados em variáveis.

```
1 double preco1;
2 double preco2;
3 double preco3;
4 ...
```

Como uma variável do tipo `double` armazena somente um valor de cada vez, seria necessário criar uma variável para cada produto. Considerando que em um supermercado existem milhares de produtos, essa abordagem não é prática, pois seria necessário criar uma grande quantidade de variáveis. Nesses casos, podemos utilizar **arrays**.

5.2 O que é um Array?



Um array é uma estrutura de dados capaz de armazenar uma coleção de variáveis. Todo array possui uma capacidade. Essa capacidade é a quantidade de variáveis que o array armazena. As variáveis contidas em um array não possuem nome. Para identificá-las, elas são numeradas de 0 até a *capacidade* - 1. Dessa forma, o índice da primeira variável é 0, o índice da segunda variável é 1, o índice da terceira variável é 2 e assim sucessivamente. Como as variáveis dentro de um array são organizadas de forma sequencial, é comum utilizar o termo **posição** para se referir a essas variáveis. Por exemplo, utilizaremos “posição 10” ou invés de “variável 10”.

Basicamente, um array é como um armário com gavetas numeradas a partir do número 0.

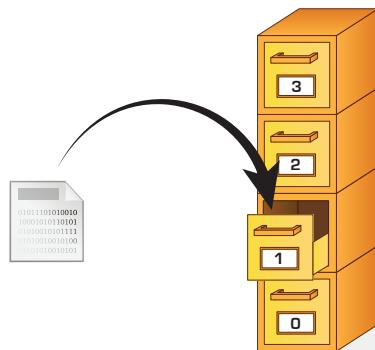


Figura 5.1: Um armário com gavetas numeradas

Quando um array é criado, é necessário definir o tipo de valor que pode ser armazenado em cada posição. Na analogia com armário, seria como se tivéssemos que definir o que pode ser guardado em cada gaveta. Por exemplo, se definirmos que um armário deve guardar livros, então somente livros podem ser armazenados nas gavetas desse armário. Não poderemos guardar revistas ou jornais.

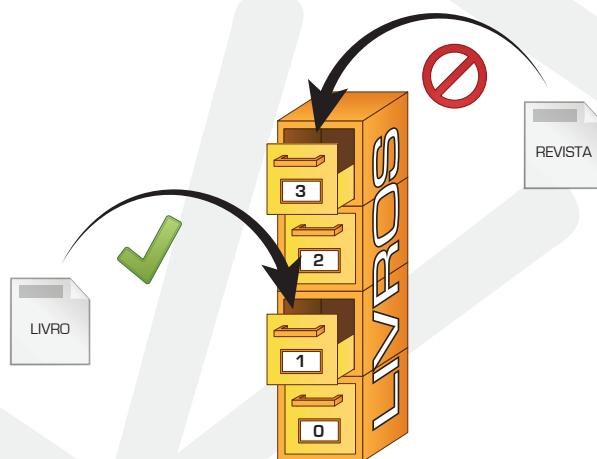


Figura 5.2: Um armário de livros não pode guardar revistas

5.3 Referências



Em C#, os arrays são objetos. Para manipular (controlar) um objeto, é necessário possuir uma referência do mesmo. Uma referência de um array é como o controle remoto de uma TV. Através do controle remoto, podemos controlar a TV. Através da referência, podemos controlar o array.

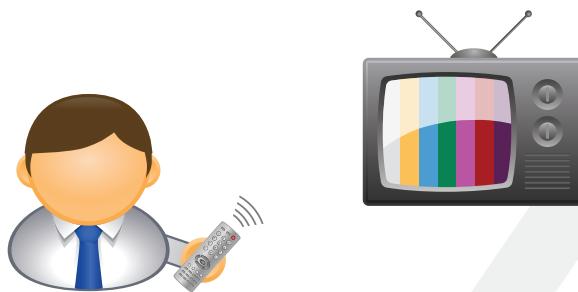


Figura 5.3: Controle remoto de uma TV



Importante

Os arrays não são armazenados em variáveis. Somente as referências dos arrays são armazenadas em variáveis.

5.4 Declaração



Antes de criar um array, é necessário declarar uma variável para armazenar a referência desse array. Nessa declaração, o tipo da variável é o tipo das variáveis contidas no array seguido dos caracteres “[” e “]” (colchetes).

```
1 int[] numeros;
```

Código C# 5.2: Declaração

Duas ou mais variáveis podem ser declaradas na mesma instrução. No exemplo abaixo, as variáveis `numeros`, `códigos` e `matrículas` são do tipo array de `int`. Os nomes das variáveis devem ser separados por vírgula.

```
1 int[] numeros, códigos, matrículas;
```

Código C# 5.3: Declarando diversas variáveis na mesma instrução

Na declaração das variáveis que armazenam referências de arrays, não é permitido informar a capacidade dos arrays.

```
1 int[10] numeros; // erro de compilação
```

Código C# 5.4: Informando a capacidade na declaração

5.5 Inicialização



Para criar um array, podemos utilizar o operador `new`. Na criação de um array, a capacidade deve ser informada dentro dos colchetes. Essa capacidade pode ser qualquer valor não negativo que pode ser convertido implicitamente para os tipos simples `int`, `uint`, `long` ou `ulong`.

No exemplo abaixo, o operador `new` cria um array com 10 posições para armazenar valores do tipo `int` e devolve a referência do mesmo. Essa referência é armazenada na variável `numeros`.

```

1 int[] numeros; // declaração
2
3 numeros = new int[10]; // inicialização

```

Código C# 5.5: Inicialização

A declaração e a inicialização podem ser realizadas na mesma linha.

```
1 int[] numeros = new int[10]; // declaração e inicialização
```

Código C# 5.6: Declaração e inicialização na mesma linha



Figura 5.4: Declaração e inicialização de um array

Quando um array é criado, todas as posições são inicializadas com valores padrão. Se as variáveis contidas no array forem de um tipo simples numérico, todas são inicializadas com o valor 0. Se elas forem do tipo simples bool, todas são inicializadas com o valor false. Se elas forem de um tipo não simples, todas são inicializadas com o valor null.

```

1 int[] numeros = new int[10]; // as posições são inicializadas com 0
2
3 bool[] aprovados = new bool[10]; // as posições são inicializadas com false
4
5 string[] nomes = new string[10]; // as posições são inicializadas com null

```

Código C# 5.7: Valores padrão

5.6 Acessando o Conteúdo de um Array



Para acessar as posições de um array, é necessário indicar o índice da posição desejada dentro de colchetes. No exemplo abaixo, a variável `numeros` guarda a referência de um array com 10 posições para armazenar valores do tipo `int`. Essa variável foi utilizada para acessar a posição 5 (sexta posição) do array.

```

1 int[] numeros = new int[10];
2
3 System.Console.WriteLine(numeros[5]); // acessando a posição 5

```

Código C# 5.8: Acessando o conteúdo de um array

5.7 Alterando o Conteúdo de um Array



Para alterar o conteúdo das posições de um array, é necessário indicar o índice da posição desejada dentro de colchetes. No exemplo abaixo, a variável `numeros` guarda a referência de um array com 10 posições para armazenar valores do tipo `int`. Essa variável foi utilizada para alterar o conteúdo da posição 5 (sexta posição) do array.

```
1 int[] numeros = new int[10];
2
3 numeros[5] = 10;
```

Código C# 5.9: Alterando o conteúdo de um array

Simulação



Nessa simulação, mostraremos um exemplo de utilização de array.

- 1 Ao executar a linha 1, um array com três variáveis do tipo `double` é criado. Essas três variáveis são inicializadas com o valor real 0.0.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

precos[0] = 0.0

precos[1] = 0.0

precos[2] = 0.0

- 2 Em seguida, a linha 2 é executada e o valor real 17.54 é armazenado na variável de índice 0 do array criado na linha 1.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

precos[0] = 17.54

precos[1] = 0.0

precos[2] = 0.0

- 3 Na sequência, a linha 3 é executada e o valor real 23.81 é armazenado na variável de índice 1 do array criado na linha 1.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

precos[0] = 17.54

precos[1] = 23.81

precos[2] = 0.0

- 4 Agora, a linha 4 será executada e o valor armazenado na variável de índice 0 do array criado na linha 1 é exibido na saída padrão.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

● 17.54

precos[0] = 17.54

precos[1] = 23.81

precos[2] = 0.0

- 5 Continuando, a linha 5 será executada e o valor armazenado na variável de índice 1 do array criado na linha 1 é exibido na saída padrão.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

● 17.54
23.81

precos[0] = 17.54

precos[1] = 23.81

precos[2] = 0.0

- 6 Por fim, a linha 6 será executada e o valor armazenado na variável de índice 2 do array criado na linha 1 é exibido na saída padrão.

```
1 double[] precos = new double[3];
2 precos[0] = 17.54;
3 precos[1] = 23.81;
4 System.Console.WriteLine(precos[0]);
5 System.Console.WriteLine(precos[1]);
6 System.Console.WriteLine(precos[2]);
```

● 17.54
23.81
0

precos[0] = 17.54

precos[1] = 23.81

precos[2] = 0.0

5.8 Outras Formas de Inicialização



Na criação de um determinado array, é possível definir o seu conteúdo inicial. No exemplo abaixo, um array para armazenar valores do tipo `int` foi criado com capacidade para três variáveis. A primeira posição foi inicializada com o valor 1, a segunda, com o valor 2 e a terceira, com o valor 3. A capacidade do array é a quantidade de elementos dentro das chaves. Nessa abordagem, a capacidade pode ou não ser informada dentro dos colchetes.

```
1 int[] numeros = new int[]{1, 2, 3};
```

```
1 int[] numeros = new int[3]{1, 2, 3};
```

O mesmo array pode ser criado de uma forma mais simples. No exemplo abaixo, novamente, um array para armazenar valores do tipo `int` foi criado com capacidade para três variáveis. A primeira posição foi inicializada com o valor 1, a segunda, com o valor 2 e a terceira foi inicializada com o valor 3.

```
1 int[] numeros = {1, 2, 3};
```



5.9 Percorrendo um Array

Ao utilizar um array, uma das tarefas mais comuns é acessar todas ou algumas posições desse array de forma sistemática para realizar algum processamento sobre o conteúdo dessas posições.

Para percorrer um array, podemos utilizar qualquer instrução de repetição. Contudo, normalmente, a instrução de repetição `for` é a mais utilizada para essa tarefa. No exemplo abaixo, um array com 100 posições para armazenar valores do tipo `int` foi criado. Todas as posições foram inicializadas com o valor padrão 0. A referência desse array foi armazenada na variável `numeros`. Em seguida, no primeiro laço, os múltiplos de 2 a partir do 0 foram armazenados nesse array. No segundo laço, os valores armazenados no array foram exibidos na saída padrão.

```
1 int[] numeros = new int[100];
2
3 for(int i = 0; i < 100; i++)
4 {
5     numeros[i] = i * 2;
6 }
7
8 for(int i = 0; i < 100; i++)
9 {
10    System.Console.WriteLine(numeros[i]);
11 }
```

Código C# 5.19: Percorrendo um array para inserir e acessar valores

A propriedade Length

No exemplo anterior, a capacidade do array foi “inserida” diretamente no código fonte em três pontos diferentes. Essa prática pode dificultar algumas mudanças na lógica do código. Em particular, para alterar a capacidade do array de 100 posições para 50 posições, três lugares devem ser alterados. Basicamente, todos os lugares onde a capacidade do array foi “inserida” diretamente no código teriam de ser alterados.

```
1 int[] numeros = new int[100];
2
3 for(int i = 0; i < 100; i++)
4 {
5     numeros[i] = i * 2;
6 }
7
8 for(int i = 0; i < 100; i++)
9 {
10    System.Console.WriteLine(numeros[i]);
11 }
```

Código C# 5.20: Capacidade “hard-coded”

Para facilitar esse tipo de modificação, podemos utilizar a propriedade `Length`. Todo array possui essa propriedade e ela armazena a capacidade do array. No exemplo abaixo, o array foi criado com 100 posições. Nos laços, utilizamos a propriedade `Length` para recuperar a capacidade do array.

```

1 int[] numeros = new int[100];
2
3 for(int i = 0; i < numeros.Length; i++)
4 {
5     numeros[i] = i * 2;
6 }
7
8 for(int i = 0; i < numeros.Length; i++)
9 {
10    System.Console.WriteLine(numeros[i]);
11 }

```

Código C# 5.21: Utilizando a propriedade Length

Agora, para alterar a capacidade do array de 100 posições para 50 posições, um único ponto do código deve ser alterado.

O laço foreach

Como vimos, o `for` tradicional pode ser utilizado para percorrer as posições de um array.

```

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

```

Código C# 5.22: Percorrendo um array com o for tradicional

Se o objetivo for somente percorrer as posições de um array sem modificar o seu conteúdo, podemos utilizar o laço **foreach**. O exemplo abaixo apresenta a sintaxe desse laço.

```

1 foreach(int numero in numeros)
2 {
3     System.Console.WriteLine(numero);
4 }

```

Código C# 5.23: Sintaxe do laço foreach

À direita da palavra chave `in`, é necessário indicar a variável que contém uma referência do array que desejamos percorrer. No exemplo acima, a variável `numeros` foi indicada. À esquerda da palavra chave `in`, é necessário declarar uma variável compatível com o tipo das variáveis armazenadas no array. No exemplo acima, a variável `numero` do tipo `int` foi declarada. Na primeira iteração do laço, o valor armazenado na primeira posição do array é copiado e guardado na variável `numero`. Na segunda iteração, o valor da segunda posição do array é copiado e guardado na variável `numero` e assim sucessivamente até a última posição do array.

5.10 Array Multidimensional



Até agora, utilizamos apenas arrays unidimensionais. Contudo, os arrays podem ser multidimensionais, ou seja, podemos criar arrays com duas ou mais dimensões. Por exemplo, um array bidimensional pode ser utilizado para representar uma tabela, uma matriz ou até um tabuleiro de batalha naval.

Utilizando a analogia anterior, um array bidimensional é como um armário no qual é possível armazenar em cada gaveta outro armário. Essa analogia pode ser expandida para arrays tridimensionais, quadridimensionais e etc.

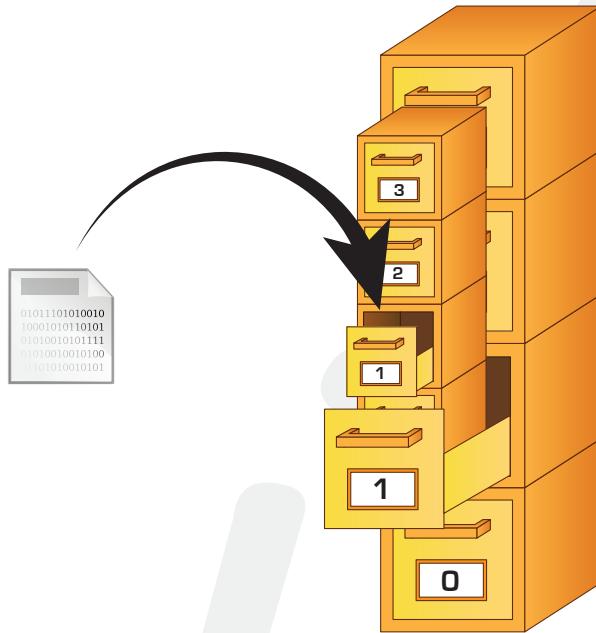


Figura 5.5: Abstração de um array multidimensional

Basicamente, os arrays multidimensionais são arrays de arrays. Há dois tipos de arrays multidimensionais: os retangulares (rectangular arrays) e os irregulares (jagged arrays).

Declaração

No exemplo abaixo, a variável `numeros` pode armazenar uma referência de um array bidimensional retangular. A quantidade de dimensões é igual à quantidade de vírgulas definidas dentro dos colchetes mais um.

```
1 int[,] numeros;
```

Código C# 5.24: Array bidimensional retangular

Agora, no próximo exemplo, a variável `numeros` pode guardar uma referência de um array bidimensional irregular. A quantidade de dimensões é igual à quantidade de pares de colchetes mais um.

```
1 int[][][] numeros;
```

Código C# 5.25: Array bidimensional irregular

No exemplo abaixo, a variável `x` pode guardar uma referência de um array bidimensional retangular, a variável `y` pode guardar uma referência de um array tridimensional retangular e a variável `z` pode guardar uma referência de um array quadridimensional retangular.

```
1 int[,] x; // bidimensional retangular
```

```
2 int[, , ] y; // tridimensional retangular
3
4 int[, , , ] z; // quadridimensional retangular
```

Código C# 5.26: Arrays multidimensionais retangulares

No exemplo abaixo, a variável `x` pode guardar uma referência de um array bidimensional irregular, a variável `y` pode guardar uma referência de um array tridimensional irregular e a variável `z` pode guardar uma referência de um array quadridimensional irregular.

```
1 int[][][] x; // bidimensional irregular
2
3 int[][][] y; // tridimensional irregular
4
5 int[][][][] z; // quadridimensional irregular
```

Código C# 5.27: Arrays multidimensionais irregulares

Na declaração das variáveis que armazenam referências de arrays, não é permitido informar a capacidade dos arrays.

```
1 int[10, 10] a; // erro de compilação
2 int[10][10] b; // erro de compilação
```

Código C# 5.28: Informando a capacidade na declaração

Inicialização

Na criação de um array multidimensional retangular, a capacidade de todas as dimensões deve ser definida. No exemplo abaixo, um array bidimensional retangular foi criado. A primeira dimensão foi definida com capacidade igual a 10 e a segunda dimensão com capacidade igual a 20. Dessa forma, temos espaço para armazenar 200 (isto é, 10×20) valores do tipo `int`.

```
1 int[,] numeros = new int[10, 20];
```

Código C# 5.29: Inicializando um array multidimensional retangular

Por outro lado, na criação de um array multidimensional irregular, apenas a capacidade da primeira dimensão deve ser definida. No exemplo abaixo, um array bidimensional irregular foi criado. A primeira dimensão foi definida com capacidade igual a 5. Em cada posição desse array, podemos armazenar uma referência de um array unidimensional. Podemos definir capacidades diferentes para esses arrays unidimensionais.

```
1 int[][] numeros = new int[5][];
2
3 numeros[0] = new int[1];
4 numeros[1] = new int[2];
5 numeros[2] = new int[3];
6 numeros[3] = new int[4];
7 numeros[4] = new int[5];
```

Código C# 5.30: Inicializando um array multidimensional irregular

O próximo código cria 1 array bidimensional e 3 unidimensionais. O array bidimensional tem capacidade para guardar 30 referências de arrays unidimensionais que armazenam valores do tipo int. O primeiro array unidimensional tem 10 posições, o segundo tem 20 posições e o terceiro tem 30 posições. As referências do primeiro, do segundo e do terceiro arrays unidimensionais foram armazenadas respectivamente na primeira, na segunda e na terceira posição do array bidimensional. As demais posições do array bidimensional continuam “vazias”, ou seja, com o valor null. Dessa forma, temos espaço para armazenar 60 (ou seja, $10 + 20 + 30$) valores do tipo int.

```
1 int[][] numeros = new int[30][];
2
3 numeros[0] = new int[10];
4 numeros[1] = new int[20];
5 numeros[2] = new int[30];
```

Código C# 5.31: Inicializando um array multidimensional irregular

Outras formas de inicialização

No exemplo abaixo, foi criado um array bidimensional retangular. A capacidade da primeira dimensão é 3 e a capacidade da segunda dimensão é 2.

```
1 int[,] numeros = new int[,]{{1, 2}, {3, 4}, {5, 6}};
```

O mesmo array do exemplo anterior pode ser criado da seguinte forma.

```
1 int[,] numeros = {{1, 2}, {3, 4}, {5, 6}};
```

No próximo exemplo, foi criado um array bidimensional irregular com capacidade 2. Na primeira posição desse array, armazenamos uma referência de um array unidimensional com capacidade 3. Na segunda posição, armazenamos uma referência de outro array unidimensional com capacidade 2. No primeiro array unidimensional, os valores 1, 2 e 3 foram armazenados. No segundo array unidimensional, os valores 4 e 5 foram armazenados.

```
1 int[][] numeros = new int[][] { new int[] { 1, 2, 3 }, new int[] { 4, 5 } };
```

Acesso

Para acessar uma posição em um array multidimensional retangular, é necessário definir um índice para cada dimensão. No exemplo abaixo, armazenamos o valor 1 na posição [2, 3].

```
1 int[,] numeros = . . .
2
3 numeros[2, 3] = 1;
```

Código C# 5.35: Acessando as posições de um array multidimensional retangular

Nesse outro exemplo, armazenamos o valor 1 na posição [1, 2, 3].

```
1 int[, ,] numeros = . . .
2
```

```
3 | numeros[1, 2, 3] = 1;
```

Código C# 5.36: Acessando as posições de um array multidimensional retangular

O próximo exemplo armazena o valor 1 na posição [0, 1, 2, 3].

```
1 | int[,,,] numeros = ...  
2 |  
3 | numeros[0, 1, 2, 3] = 1;
```

Código C# 5.37: Acessando as posições de um array multidimensional retangular

Agora, para acessar uma posição em um array multidimensional irregular, também é necessário definir um índice para cada dimensão. No exemplo abaixo, armazenamos o valor 1 na quarta posição (posição 3) do terceiro (posição 2) array unidimensional.

```
1 | int[][] numeros = ...  
2 |  
3 | numeros[2][3] = 1;
```

Código C# 5.38: Acessando as posições de um array multidimensional irregular

Nesse outro exemplo, armazenamos o valor 1 na quarta posição (posição 3) do terceiro (posição 2) array unidimensional do segundo (posição 1) array bidimensional.

```
1 | int[][][] numeros = ...  
2 |  
3 | numeros[1][2][3] = 1;
```

Código C# 5.39: Acessando as posições de um array multidimensional irregular

O próximo exemplo armazena o valor 1 na quarta posição (posição 3) do terceiro (posição 2) array unidimensional do segundo (posição 1) array bidimensional do primeiro (posição 0) array tridimensional.

```
1 | int[][][][] numeros = ...  
2 |  
3 | numeros[0][1][2][3] = 1;
```

Código C# 5.40: Acessando as posições de um array multidimensional irregular

Percorrendo um array multidimensional

Para percorrer um array multidimensional retangular, podemos utilizar laços encadeados. A quantidade de laços encadeados é igual à quantidade de dimensões desse array. No exemplo abaixo, o primeiro `for` percorre a primeira dimensão e o segundo `for` a segunda dimensão. O método `GetLength` recebe o índice de uma dimensão e devolve a capacidade dessa dimensão.

```
1 | int[,] numeros = new int[10, 20];  
2 |  
3 | for(int i = 0; i < numeros.GetLength(0); i++)  
4 | {  
5 |   for(int j = 0; j < numeros.GetLength(1); j++)  
6 |   {  
7 |     numeros[i, j] = i * j;
```

```
8 } }
9 }
```

Código C# 5.41: Percorrendo um array bidimensional retangular

No próximo exemplo, percorremos todas as posições de um array tridimensional retangular utilizando três laços encadeados.

```
1 int[, ,] numeros = new int[10, 20, 30];
2
3 for(int i = 0; i < numeros.GetLength(0); i++)
4 {
5     for(int j = 0; j < numeros.GetLength(1); j++)
6     {
7         for(int k = 0; k < numeros.GetLength(2); k++)
8         {
9             numeros[i, j, k] = i * j * k;
10        }
11    }
12 }
```

Código C# 5.42: Percorrendo um array tridimensional retangular

Analogamente, para percorrer um array multidimensional irregular, também podemos utilizar laços encadeados. No exemplo abaixo, utilizamos dois laços encadeados para percorrer um array bidimensional irregular.

```
1 int[][] numeros = new int[10][];
2
3 for(int i = 0; i < numeros.Length; i++)
4 {
5     numeros[i] = new int[i + 1];
6
7     for(int j = 0; j < numeros[i].Length; j++)
8     {
9         numeros[i][j] = i * j;
10    }
11 }
```

Código C# 5.43: Percorrendo um array bidimensional irregular

No próximo exemplo, três laços encadeados foram utilizados para percorrer um array tridimensional irregular.

```
1 int[][][] numeros = new int[10][][];
2
3 for(int i = 0; i < numeros.Length; i++)
4 {
5     numeros[i] = new int[i + 1][];
6
7     for(int j = 0; j < numeros[i].Length; j++)
8     {
9         numeros[i][j] = new int[j + 1];
10
11         for(int k = 0; k < numeros[i][j].Length; k++)
12         {
13             numeros[i][j][k] = i * j * k;
14         }
15     }
16 }
```

Código C# 5.44: Percorrendo um array tridimensional irregular



Simulação

Nessa simulação, mostraremos um exemplo de utilização de array multidimensional.

- 1 Ao executar a linha 1, um array bidimensional retangular 2 por 2 do tipo `int` é criado. Todas as posições desse array são inicializadas com valor padrão do tipo `int`, ou seja, todas as posições são inicializadas com o valor 0.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

x[0][0] = 0
x[0][1] = 0
x[1][0] = 0
x[1][1] = 0

- 2 Em seguida, a linha 2 é executada e o valor inteiro 1 é armazenado na variável de índice [0, 0] do array criado na linha 1.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

x[0][0] = 1
x[0][1] = 0
x[1][0] = 0
x[1][1] = 0

- 3 Na sequência, a linha 3 é executada e o valor inteiro 2 é armazenado na variável de índice [0, 1] do array criado na linha 1.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

x[0][0] = 1
x[0][1] = 2
x[1][0] = 0
x[1][1] = 0

- 4 Agora, a linha 4 é executada e o valor inteiro 3 é armazenado na variável de índice [1, 0] do array criado na linha 1.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 0

- 5 Continuando, a linha 5 é executada e o valor inteiro 4 é armazenado na variável de índice [1, 1] do array criado na linha 1.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 4

- 6 Em seguida, a linha 6 será executada e o valor armazenado na variável de índice [0, 0] do array criado na linha 1 é exibido na saída padrão.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

● 1

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 4

- 7 Na sequência, a linha 7 será executada e o valor armazenado na variável de índice [0, 1] do array criado na linha 1 é exibido na saída padrão.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

● 2

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 4

- 8 Na sequência, a linha 8 será executada e o valor armazenado na variável de índice [1, 0] do array criado na linha 1 é exibido na saída padrão.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

1
2
3

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 4

- 9 Por fim, a linha 9 será executada e o valor armazenado na variável de índice [1, 1] do array criado na linha 1 é exibido na saída padrão.

```

1 int[,] x = new int[2, 2];
2 x[0, 0] = 1;
3 x[0, 1] = 2;
4 x[1, 0] = 3;
5 x[1, 1] = 4;
6 System.Console.WriteLine(x[0, 0]);
7 System.Console.WriteLine(x[0, 1]);
8 System.Console.WriteLine(x[1, 0]);
9 System.Console.WriteLine(x[1, 1]);

```

1
2
3
4

x[0][0] = 1

x[0][1] = 2

x[1][0] = 3

x[1][1] = 4

5.11 Erro: Utilizar valores incompatíveis como índices de um array

Um erro de compilação ocorre quando usamos como índice de um array um valor de tipo incompatível com int.

```

1 class Programa
2 {
3     static void Main()
4     {
5         double i = 1;
6         double[] numeros = new double[5];
7         System.Console.WriteLine(numeros[i]);
8     }
9 }

```

Código C# 5.54: Programa.cs

No código acima, ocorre um erro de compilação na linha 7, pois um valor de tipo double foi utilizado como índice do array.

```
Programa.cs(7,36): error CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
```

Terminal 5.8: Erro de compilação

5.12 Erro: Definir outras dimensões ao criar arrays irregulares



Na criação de um array irregular, ocorre um erro de compilação quando definimos outras dimensões além da primeira dimensão.

```

1 class Programa
2 {
3     static void Main() {
4         int[][] a = new int[3][5];
5     }
6 }
```

Código C# 5.55: Programa.cs

No código acima, declaramos um array de duas dimensões. Em sua inicialização, na linha 4, definimos a segunda dimensão do array e isso provoca um erro de compilação.

```
Programa.cs(4,26): error CS0178: Invalid rank specifier: expected ',' or ']'
```

Terminal 5.9: Erro de compilação

5.13 Erro: Acessar uma posição inválida de um array



Ocorre um erro de execução quando tentamos acessar uma posição inválida de um array.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int[] a = new int[5];
6         System.Console.WriteLine(a[5]);
7     }
8 }
```

Código C# 5.56: Programa.cs

No código acima, tentamos acessar a posição de número 5 do array a. Apesar do array possuir 5 posições, elas devem ser acessadas com índices que variam de 0 a 4.

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at Programa.Main()
```

Terminal 5.10: Erro de execução



5.14 Resumo

- Os arrays são estruturas de dados simples que permitem o armazenamento **sequencial** de dados.
- As posições de um array são numeradas sequencialmente iniciando com **0**.

3 ► A capacidade de um array retangular pode ser obtida através do método `GetLength` e a capacidade de um array irregular pode ser obtida através do atributo `Length`.

4 ► Na tentativa de acesso à uma posição inexistente de um array, um **erro de execução** é gerado.

MÉTODOS

6.1 Introdução



Considere o sistema de uma empresa que precisa gerar diversos tipos de documentos como recibos, atestados e relatórios. Os dados da empresa devem aparecer no cabeçalho desses documentos.

```
1 System.Console.WriteLine("----- K19 Treinamentos -----");
2 System.Console.WriteLine("----- contato@k19.com.br -----");
```

Código C# 6.1: Cabeçalho

As duas linhas de código acima exibem o cabeçalho dos documentos. Toda vez que um documento é gerado, esse trecho de código deve ser executado. A primeira abordagem para resolver esse problema é replicar essas duas linhas toda vez que for necessário. Contudo, essa replicação tornará a manutenção do sistema mais complicada.

Por exemplo, suponha que o sistema já esteja funcionando e o trecho de código que exibe o cabeçalho dos documentos tenha sido replicado muitas vezes. Agora, considere uma mudança simples no cabeçalho dos documentos. O telefone da empresa deve aparecer depois do email. Para atender a essa nova regra, será necessário modificar o código fonte em vários lugares.

De forma geral, toda vez que houver uma alteração no cabeçalho, será necessário modificar muitos lugares do código fonte. Consequentemente, a manutenção do sistema será mais demorada. Para facilitar eventuais mudanças no cabeçalho, podemos utilizar o conceito de **método**. Um método permite que um determinado trecho de código possa ser reutilizado várias vezes.

6.2 Estrutura Geral de um Método



Então, definiremos um método para exibir o cabeçalho dos documentos na saída padrão e reproveitá-lo sempre que for necessário. Observe a declaração do método `ExibeCabecalho` no exemplo abaixo.

```
1 static void ExibeCabecalho()
2 {
3     System.Console.WriteLine("----- K19 Treinamentos -----");
4     System.Console.WriteLine("----- contato@k19.com.br -----");
5 }
```

A palavra chave `static` permite que o método `ExibeCabecalho` possa ser utilizado sem a criação de um objeto. O conceito de objeto não faz parte do conteúdo deste livro. Portanto, utilizaremos o modificador `static` na declaração de todos os métodos desse material.

A palavra chave `void` indica que o método `ExibeCabecalho` não devolverá nenhuma resposta depois de ser executado.

À direita da palavra chave `void`, definimos o nome do método. Em nosso caso, o nome do método é `ExibeCabecalho`. Os nomes dos métodos são utilizados para chamá-los posteriormente.

Depois do nome do método, os parâmetros são definidos dentro de parênteses. Como o método `ExibeCabecalho` não precisa de parâmetros, nada foi definido dentro dos parênteses.

Por fim, à direita dos parâmetros, foi definido o corpo do método `ExibeCabecalho`. No corpo de um método, colocamos as instruções que devem ser executadas quando esse método for chamado.

Observe, no código abaixo, o método `ExibeCabecalho` sendo chamado duas vezes.

```
1 class Programa
2 {
3     static void Main()
4     {
5         // chamando o método ExibeCabecalho
6         ExibeCabecalho();
7         System.Console.WriteLine("Recibo: R$ 545,00");
8
9         System.Console.WriteLine();
10
11        // chamando método ExibeCabecalho
12        ExibeCabecalho();
13        System.Console.WriteLine("Atestado de Matrícula: Jonas Keizo Hirata");
14    }
15
16    static void ExibeCabecalho()
17    {
18        System.Console.WriteLine("----- K19 Treinamentos -----");
19        System.Console.WriteLine("----- contato@k19.com.br -----");
20    }
21 }
```

```
----- K19 Treinamentos -----
----- contato@k19.com.br -----
Recibo: R$ 545,00

----- K19 Treinamentos -----
----- contato@k19.com.br -----
Atestado de Matrícula: Jonas Keizo Hirata
```

Agora, acrescentar o telefone da empresa no cabeçalho dos documentos é muito fácil. Basta alterar o código do método `ExibeCabecalho`.

```
1 static void ExibeCabecalho()
2 {
3     System.Console.WriteLine("----- K19 Treinamentos -----");
4     System.Console.WriteLine("----- contato@k19.com.br -----");
5     System.Console.WriteLine("----- 11 2387-3791 -----");
6 }
```

6.3 Parâmetros



Considere um programa de computador que realiza operações financeiras como o cálculo de juro simples por exemplo. Para evitar repetição de código, podemos definir um método para realizar esse cálculo e reutilizá-lo toda vez que for necessário.

```
1 static void CalculaJuroSimples()
2 {
3     double juro = 10000 * 0.015 * 12;
4 }
```

Observe que o método acima considera um capital fixo de R\$ 10.000,00, uma taxa de juro fixa de 1,5% e um período fixo de 12 meses. De fato, esse método não é muito útil porque toda vez que ele for chamado, ele realizará o cálculo com esses valores fixos.

Para tornar o método `CalculaJuroSimples` mais útil, devemos parametrizá-lo. Basicamente, os parâmetros de um método são variáveis que permitem que valores diferentes sejam passados a cada chamada desse método.

```
1 static void CalculaJuroSimples(double capital, double taxa, int periodo)
2 {
3     double juro = capital * taxa * periodo;
4 }
```

No código acima, três parâmetros foram definidos para o método `CalculaJuroSimples`: `capital`, `taxa` e `periodo`. O primeiro parâmetro é do tipo `double`, o segundo também é do tipo `double` e o terceiro é do tipo `int`.

Agora, nas chamadas do método `CalculaJuroSimples`, devemos passar os três valores necessários para o cálculo do juro simples. No exemplo a seguir, o método `Main` chama o método `CalculaJuroSimples` duas vezes. Na primeira chamada, os valores passados como parâmetros são 10000, 0.015 e 12. Na segunda chamada, os valores passados como parâmetros são 25400, 0.02 e 30.

```
1 class Programa
2 {
3     static void Main()
4     {
5         CalculaJuroSimples(10000, 0.015, 12);
6
7         CalculaJuroSimples(25400, 0.02, 30);
8     }
9
10    static void CalculaJuroSimples(double capital, double taxa, int periodo)
11    {
12        double juro = capital * taxa * periodo;
13    }
14 }
```

6.4 Resposta



O valor calculado dentro do método `CalculaJuroSimples` é armazenado em uma variável local. Essa variável não pode ser acessada dentro do método `Main`. Em outras palavras, o método `Main` não tem acesso ao valor do juro que foi calculado dentro do método `CalculaJuroSimples`.

Todo método pode, ao final do seu processamento, devolver uma resposta para quem o chamou. O comando `return` indica o valor de resposta de um método.

```

1 static double CalculaJuroSimples(double capital, double taxa, int periodo)
2 {
3     double juro = capital * taxa * periodo;
4     return juro;
5 }
```

Observe as duas modificações realizadas no método `CalculaJuroSimples`. A primeira alteração é a retirada da palavra reservada `void` e a inserção da palavra reservada `double` em seu lugar. A palavra `void` indicava que o método não devolvia nenhuma resposta ao final do seu processamento. A palavra `double` indica que o método devolverá um valor do tipo `double` ao final do seu processamento. A segunda modificação é a utilização do comando `return` para devolver como resposta o juro calculado, que é um valor do tipo `double`.

Agora, a resposta pode ser recuperada no método `Main` e armazenada em uma variável por exemplo.

```

1 class Programa
2 {
3     static void Main()
4     {
5         double resposta1 = CalculaJuroSimples(10000, 0.015, 12);
6
7         double resposta2 = CalculaJuroSimples(25400, 0.02, 30);
8
9         System.Console.WriteLine("Juro: " + resposta1);
10        System.Console.WriteLine("Juro: " + resposta2);
11    }
12
13     static double CalculaJuroSimples(double capital, double taxa, int periodo)
14     {
15         double juro = capital * taxa * periodo;
16         return juro;
17     }
18 }
```

Um método pode devolver outros tipos de valores. Para isso, basta modificar a marcação de retorno definindo o tipo de valor que o método devolverá. Veja alguns exemplos.

```

1 static int Metodo()
2 {
3     // corpo de um método que devolve int
4 }
```

```

1 static char Metodo()
2 {
3     // corpo de um método que devolve char
4 }
```

```

1 static float Metodo()
2 {
3     // corpo de um método que devolve float
4 }
```



Simulação

Nessa simulação, mostraremos um exemplo de chamada de método.

- Na execução da linha 5, uma variável do tipo `int` chamada `a` é declarada e inicializada com o valor 1. Essa variável é uma variável local do método `Main`.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }
```

`amain = 1`

- Na sequência, a linha 6 é executada e uma variável do tipo `int` chamada `b` é declarada e inicializada com o valor 2. Essa variável é uma variável local do método `Main`.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }
```

`amain = 1`

`bmain = 2`

- Em seguida, na linha 7, o método `soma` é chamado. Os valores armazenados nas variáveis `a` e `b` do método `Main` são passados como argumentos nessa chamada de método.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }
```

a_{main} = 1b_{main} = 2

- 4 Agora, as variáveis `a` e `b` do método `soma` recebem os valores passados como argumentos no passo anterior. Essas variáveis são variáveis locais do método `soma`.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }
```

a_{main} = 1b_{main} = 2a_{soma} = 1b_{soma} = 2

- 5 Na sequência, ao executar a linha 12, a soma dos valores armazenados nas variáveis `a` e `b` do método `soma` é armazenada na variável `c`. Essa variável `c` é uma variável local do método `soma`

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }
```

a_{main} = 1b_{main} = 2a_{soma} = 1b_{soma} = 2c_{soma} = 3

- 6 Em seguida, ao executar a linha 13, o valor armazenado na variável `c` do método `soma` é devolvido como resposta para o método `Main`.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }

```

a_{main} = 1b_{main} = 2a_{soma} = 1b_{soma} = 2c_{soma} = 3

- 7 Agora, o fluxo de execução volta para o método `Main`. Na linha 7, a variável `c` desse método recebe a resposta da chamada do método `soma`.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }

```

a_{main} = 1b_{main} = 2c_{main} = 3

- 8 Por fim, na execução da linha 8, o valor da variável `c` do método `Main` é exibido na saída padrão.

```

1 class Metodos
2 {
3     static void Main()
4     {
5         int a = 1;
6         int b = 2;
7         int c = soma(a, b);
8         System.Console.WriteLine(c);
9     }
10    static int soma(int a, int b)
11    {
12        int c = a + b;
13        return c;
14    }
15 }

```

a_{main} = 1b_{main} = 2c_{main} = 3



6.5 Passagem de Parâmetros

No exemplo abaixo, duas variáveis foram definidas no método `Main`: `a` e `b`. Analogamente, duas variáveis com os mesmos nomes foram definidas no método `ExibeSoma`. Apesar de terem nomes iguais, as variáveis do método `Main` são independentes das variáveis do método `ExibeSoma` e vice e versa.

```

1 class Programa
2 {
3     static void Main()
4     {
5         double a = 5.6;
6         double b = 7.1;
7
8         ExibeSoma(a, b);
9     }
10
11    static double ExibeSoma(double a, double b)
12    {
13        System.Console.WriteLine(a + b);
14    }
15 }
```

Código C# 6.21: Programa.cs

Na linha 8, quando o método `ExibeSoma` é chamado, o valor armazenado na variável `a` do método `Main` é copiado para a variável `a` do método `ExibeSoma`. Analogamente, o valor armazenado na variável `b` do método `Main` é copiado para a variável `b` do método `ExibeSoma`.

Alterações nos valores armazenados nas variáveis do método `ExibeSoma` não afetam o conteúdo das variáveis do método `Main` e vice e versa.

Para exemplificar esse comportamento, considere o código abaixo. Na linha 5, a variável `a` do método `Main` foi declarada e inicializada com o valor 1. Na chamada do método `Teste`, na linha 7, o valor armazenado nessa variável é copiado para a variável `a` do método `Teste`. O conteúdo da variável `a` do método `Teste` é alterado na linha 14 e essa variável passa a armazenar o valor 2. Quando o fluxo de execução volta para o método `Main`, o método `WriteLine` é utilizado para exibir o valor da variável `a` do método `Main`. Essa variável armazena o valor 1. Portanto, na saída padrão, o número 1 será exibido.

```

1 class Programa
2 {
3     static void Main()
4     {
5         int a = 1;
6
7         Teste(a);
8
9         System.Console.WriteLine(a); // exibe 1
10    }
11
12    static void Teste(int a)
13    {
14        a = 2;
15    }
16 }
```

Código C# 6.22: Programa.cs



Simulação

Nessa simulação, mostraremos um exemplo de passagem de parâmetros.

- Na execução da linha 5, uma variável do tipo `int` chamada `a` é declarada e inicializada com o valor 1. Essa variável é uma variável local do método `Main`.

```

1 class Parametros
2 {
3     static void Main()
4     {
5         int a = 1;
6         Troca(a);
7         System.Console.WriteLine(a);
8     }
9     static void Troca(int a)
10    {
11        a = 2;
12    }
13 }
```

`amain = 1`

- Em seguida, na linha 6, o método `Troca` é chamado. O valor armazenado na variável `a` do método `Main` é passado como argumento nessa chamada de método.

```

1 class Parametros
2 {
3     static void Main()
4     {
5         int a = 1;
6         Troca(a);
7         System.Console.WriteLine(a);
8     }
9     static void Troca(int a)
10    {
11        a = 2;
12    }
13 }
```

`amain = 1`

- Agora, a variável `a` do método `Troca` recebe o valor passado como argumento no passo anterior. Essa variável é uma variável local do método `Troca`.

```

1 class Parametros
2 {
3     static void Main()
4     {
5         int a = 1;
6         Troca(a);
7         System.Console.WriteLine(a);
8     }
9     static void Troca(int a)
10    {
11        a = 2;
12    }
13 }
```

`amain = 1`

`atroca = 1`

- 4 Na execução da linha 11, o valor da variável `a` do método `Troca` é alterado. Essa alteração não afeta o valor da variável `a` do método `Main`, pois essas variáveis são independentes.

```

1 class Parametros
2 {
3     static void Main()
4     {
5         int a = 1;
6         Troca(a);
7         System.Console.WriteLine(a);
8     }
9     static void Troca(int a)
10    {
11        a = 2;
12    }
13 }
```

a_{main} = 1a_{Troca} = 2

- 5 O fluxo de execução volta para o método `Main`. Ao executar a linha 7, o valor da variável `a` do método `Main` é exibido na saída padrão, ou seja, o número 1 é exibido na saída padrão.

```

1 class Parametros
2 {
3     static void Main()
4     {
5         int a = 1;
6         Troca(a);
7         System.Console.WriteLine(a);
8     }
9     static void Troca(int a)
10    {
11        a = 2;
12    }
13 }
```

a_{main} = 1

● 1



6.6 Sobrecarga

No exemplo abaixo, o método `Maximo` recebe como parâmetro exatamente dois valores do tipo `double` e devolve o maior deles.

```

1 static double Maximo(double a, double b)
2 {
3     if(a > b)
4     {
5         return a;
6     }
7     else
8     {
9         return b;
10    }
11 }
```

Esse método pode ser utilizado toda vez que temos dois valores do tipo `double` e desejamos descobrir qual é o maior. As chamadas abaixo mostram a utilização do método `Maximo`.

```

1 double m1 = Maximo(7.8, 9.8);
2 double m2 = Maximo(3.2, 1.7);
3 double m3 = Maximo(5.4, 4.9);
```

Agora, considere que, com bastante frequência, é necessário descobrir o máximo entre três valores do tipo `double`. Para resolver esse problema, o método `MaximoEntreTresValores` foi definido.

```

1 static double MaximoEntreTresValores(double a, double b, double c)
2 {
3     if(a >= b && a >= c)
4     {
5         return a;
6     }
7     else if(b >= a && b >= c)
8     {
9         return b;
10    }
11   else
12   {
13       return c;
14   }
15 }
16
17 static double Maximo(double a, double b)
18 {
19     if(a >= b)
20     {
21         return a;
22     }
23     else
24     {
25         return b;
26     }
27 }
```

Os dois métodos acima possuem objetivos muito parecidos. Um deles descobre quem é o maior entre dois valores do tipo `double`. O outro descobre quem é o maior entre três valores do tipo `double`. Esses métodos podem ser utilizados de forma semelhante ao apresentado no código abaixo.

```

1 double m1 = Maximo(7.8, 9.8);
2 double m2 = MaximoEntreTresValores(3.2, 1.7, 4.1);
```

De forma análoga e conforme a necessidade, poderíamos definir o método `MaximoEntreQuatroValores` ou `MaximoEntreCincoValores`. Do ponto de vista prático, não é interessante ter métodos com objetivos tão semelhantes mas com nomes diferentes.

Para simplificar, podemos definir diversos métodos com o mesmo nome. No exemplo abaixo, três métodos chamados `Maximo` foram definidos. As quantidades de parâmetros que esses métodos recebem são diferentes.

```

1 static double Maximo(double a, double b, double c, double d)
2 {
3     // código
4 }
5
6 static double Maximo(double a, double b, double c)
7 {
8     // código
9 }
10
11 static double Maximo(double a, double b)
12 {
13     // código
14 }
```

Nas chamadas desses métodos, eles são diferenciados de acordo com a quantidade de argumentos utilizados. No código a seguir, a primeira chamada aciona o método `Maximo` que recebe dois parâmetros. A segunda chamada aciona o método `Maximo` que recebe três parâmetros. A terceira chamada aciona o método `Maximo` que recebe quatro parâmetros.

```
1 double m1 = Maximo(7.8, 9.8);
2 double m2 = Maximo(3.2, 1.7, 4.1);
3 double m3 = Maximo(7.5, 6.3, 9.7, 8.8);
```

A possibilidade de criar métodos com o mesmo nome é denominada **sobrecarga**.

Regra

Dois métodos podem ter o mesmo nome se as respectivas listas de parâmetros são diferentes em quantidade ou tipo. Se essa regra não for respeitada ocorrerá erro de compilação.

No exemplo abaixo, há uma sobrecarga inválida porque os dois métodos que possuem o mesmo nome também possuem listas de parâmetros equivalentes. Os dois recebem dois parâmetros do tipo `double`.

```
1 static double Metodo(double a, double b)
2 {
3     // código
4 }
5
6 static double Metodo(double c, double d)
7 {
8     // código
9 }
```

6.7 Parâmetros Variáveis



No exemplo abaixo, o método `CalculaSomatorio` recebe como parâmetro exatamente três valores do tipo `double`. Depois, ele calcula e devolve o somatório desses valores.

```
1 static double CalculaSomatorio(double a, double b, double c)
2 {
3     return a + b + c;
4 }
```

As chamadas abaixo são exemplos de utilização do método `CalculaSomatorio`.

```
1 double s1 = CalculaSomatorio(1.5, 2.7, 6.4);
2 double s2 = CalculaSomatorio(5.1, 1.7, 3.2);
3 double s3 = CalculaSomatorio(5.5, 4.2, 4.7);
4 double s4 = CalculaSomatorio(1.9, 2.1, 5.0);
```

Observe que o método `CalculaSomatorio` só pode ser utilizado para calcular o somatório de exatamente três valores do tipo `double`. Contudo, ele seria mais útil se pudesse calcular o somatório de zero ou mais valores do tipo `double`. Para isso, o método `CalculaSomatorio` precisa ser capaz de receber zero ou mais valores como parâmetro.

Agora, no exemplo abaixo, o parâmetro do método `CalculaSomatorio` é um **parameter-array**. O que caracteriza esse parâmetro ser um parameter-array é a utilização da palavra chave `params`.

```
1 static double CalculaSomatorio(params double[] valores)
2 {
3
4 }
```

A utilização de um parameter-array na definição do método `CalculaSomatorio` permite que zero ou mais valores do tipo `double` sejam passados como argumento. Dessa forma, as chamadas abaixo são todas válidas.

```
1 double s1 = CalculaSomatorio();
2 double s2 = CalculaSomatorio(5.1);
3 double s3 = CalculaSomatorio(5.5, 4.2);
4 double s4 = CalculaSomatorio(1.9, 2.1, 5.0);
```

A cada chamada, os argumentos do método `CalculaSomatorio` são armazenados em um array. Uma referência desse array é armazenada no parâmetro `valores`. Dentro do corpo do método `CalculaSomatorio`, o array pode ser manipulado normalmente.

```
1 static double CalculaSomatorio(params double[] valores)
2 {
3     double soma = 0;
4     for(int i = 0; i < valores.Length; i++)
5     {
6         soma += valores[i];
7     }
8     return soma;
9 }
```

Agora, considere um método que calcula e devolve a média aritmética de valores do tipo `double`. Matematicamente, não faz sentido calcular a média aritmética de zero números. Portanto, esse método deve receber pelo menos um argumento do tipo `double`.

No exemplo abaixo, o método `CalculaMedia` foi definido com um parâmetro do tipo `double` e um parameter-array de `double`. Sendo assim, a cada chamada desse método, é necessário passar como argumento pelo menos um valor do tipo `double`.

```
1 static double CalculaMedia(double valor, params double[] valores)
2 {
3     double soma = valor;
4     for(int i = 0; i < valores.Length; i++)
5     {
6         soma += valores[i];
7     }
8     return soma / (1 + valores.Length);
9 }
```

As seguintes chamadas são exemplos de utilização do método `CalculaMedia`.

```
1 double m1 = CalculaMedia(3.5);
2 double m2 = CalculaMedia(5.1, 6.7);
3 double m3 = CalculaMedia(5.5, 4.2, 8.9);
4 double m4 = CalculaMedia(1.9, 2.1, 5.0, 8.8);
```

Regra

Há uma regra fundamental relacionada à utilização de parameter-arrays. O parameter-array, se existir, deve ser o último parâmetro do método correspondente. Dessa forma, as listas de parâmetros dos métodos abaixo estão corretas.

```

1 static void Metodo1(params int[] valores) { }
2 static void Metodo2(int a, params string[] valores) { }
3 static void Metodo3(double a, char c, params double[] valores) { }
4 static void Metodo4(string a, params char[] valores) { }

```

Por outro lado, os métodos abaixo estão incorretos.

```

1 static void Metodo1(params int[] valores, double a) { }
2 static void Metodo2(params string[] valores, params int[] numeros) { }
3 static void Metodo3(params double[] valores, char c, int i) { }

```

Uma conclusão imediata da regra de utilização de parameter-arrays é que todo método pode ter no máximo um parameter-array.

6.8 Erro: Parâmetros incompatíveis



Um erro de compilação comum em C# ocorre quando um método é chamado com parâmetros incompatíveis.

```

1 class Programa
2 {
3     static void Main()
4     {
5         Metodo();
6         Metodo(10.1, 10.1, "k19");
7         Metodo("10", "10.1", "k19");
8     }
9
10    static void Metodo(int a, double b, string c)
11    {
12        return a + b + c;
13    }
14 }

```

Código C# 6.44: Programa.cs

A mensagem de erro de compilação seria semelhante à apresentada abaixo.

```

Programa.cs(5,3): error CS1501: No overload for method 'Metodo' takes 0
arguments
Programa.cs(10,14): (Location of symbol related to previous error)
Programa.cs(6,3): error CS1502: The best overloaded method match for
    'Programa.Metodo(int, double, string)' has some invalid arguments
Programa.cs(6,10): error CS1503: Argument 1: cannot convert from 'double' to
    'int'
Programa.cs(7,3): error CS1502: The best overloaded method match for
    'Programa.Metodo(int, double, string)' has some invalid arguments
Programa.cs(7,10): error CS1503: Argument 1: cannot convert from 'string' to
    'int'

```

```
Programa.cs(7,16): error CS1503: Argument 2: cannot convert from 'string' to
                  'double'
Programa.cs(12,3): error CS0127: Since 'Programa.Metodo(int, double, string)'
                   returns void, a return keyword must not be followed by an object
                   expression
```

Terminal 6.4: Erro de compilação

6.9 Erro: Resposta incompatível



Um erro de compilação comum em C# ocorre quando armazenamos a resposta de um método em variáveis de tipos incompatíveis.

```
1 class Programa
2 {
3     static void Main()
4     {
5         int a = Metodo();
6         double b = Metodo();
7         bool c = Metodo();
8     }
9
10    static string Metodo()
11    {
12        return "k19";
13    }
14 }
```

Código C# 6.45: Programa.cs

A mensagem de erro de compilação seria semelhante à apresenta abaixo.

```
Programa.cs(5,11): error CS0029: Cannot implicitly convert type 'string' to
                  'int'
Programa.cs(6,14): error CS0029: Cannot implicitly convert type 'string' to
                  'double'
Programa.cs(7,12): error CS0029: Cannot implicitly convert type 'string' to
                  'bool'
```

Terminal 6.5: Erro de compilação

6.10 Erro: Esquecer a instrução return



Um erro de compilação comum em C# ocorre quando um método que deve devolver uma resposta (um método não void) não a devolve. Isso pode acontecer, por exemplo, quando a instrução `return` não está presente no corpo do método. Veja o exemplo abaixo.

```
1 class Programa
2 {
3     static void Main()
4     {
5         int resposta = Teste();
6     }
7
8     static int Teste()
9     {
10        System.Console.WriteLine(1);
11    }
12 }
```

Código C# 6.46: Programa.cs

Note que o método `Teste` não devolve uma resposta. Como esse método não é `void` ocorre um erro de compilação. A mensagem de erro de compilação seria semelhante à apresenta abaixo.

```
Programa.cs(8,13): error CS0161: 'Programa.Teste()': not all code paths return a
value
```

Terminal 6.6: Erro de compilação

O código a seguir possui o mesmo problema.

```
1 class Programa
2 {
3     static void Main()
4     {
5         int resposta = Teste();
6     }
7
8     static int Teste()
9     {
10        System.Random gerador = new System.Random();
11        double valor = gerador.NextDouble();
12        if(valor > 0.5)
13        {
14            return 1;
15        }
16    }
17 }
```

Código C# 6.47: Programa.cs

No exemplo acima, se o valor gerado pelo método `NextDouble` for menor ou igual a 0.5, o método `Teste` não devolverá resposta. Novamente, como esse método não é `void` ocorre um erro de compilação. A mensagem de erro de compilação seria semelhante à apresenta abaixo.

```
Programa.cs(8,13): error CS0161: 'Programa.Teste()': not all code paths return a
value
```

Terminal 6.7: Erro de compilação

6.11 Erro: Não utilizar parênteses



Um erro de compilação comum em C# ocorre quando não utilizamos parênteses na declaração ou na chamada de um método. No exemplo abaixo, o método `Teste` foi declarado sem parênteses.

```
1 class Programa
2 {
3     static void Main()
4     {
5         Teste();
6     }
7
8     static void Teste
9     {
10        System.Console.WriteLine("K19");
11    }
12 }
```

Código C# 6.48: Programa.cs

A mensagem de erro de compilação seria semelhante à apresenta abaixo.

```
Programa.cs(10,3): error CS1014: A get or set accessor expected
```

Terminal 6.8: Erro de compilação

Nesse outro exemplo, o método `Teste` foi declarado corretamente, mas foi chamado sem parênteses.

```
1 class Programa
2 {
3     static void Main()
4     {
5         Teste;
6     }
7
8     static void Teste()
9     {
10        System.Console.WriteLine("K19");
11    }
12 }
```

Código C# 6.49: Programa.cs

A mensagem de erro de compilação seria semelhante à apresenta abaixo.

```
Programa.cs(5,3): error CS0201: Only assignment, call, increment, decrement,
await, and new object expressions can be used as a statement
```

Terminal 6.9: Erro de compilação

6.12 Resumo



- 1 ► Para evitar a repetição de um determinado trecho de código, podemos criar um método.
- 2 ► Um método possui uma marcação de retorno, um nome, uma lista de parâmetros e um corpo.
- 3 ► Um método pode ter zero ou mais parâmetros.
- 4 ► Um parâmetro é uma variável local de um método.
- 5 ► A palavra reservada `void` é utilizada em métodos que não devolvem resposta.
- 6 ► A palavra reservada `return` é utilizada para finalizar um método e devolver uma resposta caso o método não seja `void`.
- 7 ► As variáveis locais de um método não podem ser alteradas em outro método.

- 8 ► Sobrecarga é a possibilidade de definir métodos com o mesmo nome.
- 9 ► Métodos com o mesmo nome devem ter lista de parâmetros diferentes.
- 10 ► Um método pode receber uma quantidade variável de parâmetros utilizando parameter-array.





STRING

7.1 Referências



Em C#, as strings são objetos. Para controlar um objeto, é necessário possuir uma referência desse objeto. Uma referência de uma string é como um controle remoto de uma TV. Através de um controle remoto, podemos controlar uma TV. Através de uma referência, podemos controlar uma string.

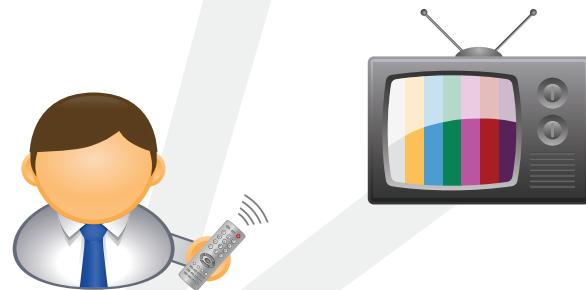


Figura 7.1: Controle remoto de uma TV



Importante

As variáveis do tipo `string` não armazenam os objetos do tipo `string`. Elas armazenam referências para esses objetos.

Ao executar o código abaixo, um objeto do tipo `string` será criado com o conteúdo “Rafael”. A referência desse objeto será armazenada na variável `nome`.

```
1 class Programa
2 {
3     static void Main()
4     {
5         string nome = "Rafael";
6     }
7 }
```



7.2 Pool de Strings

Um objeto do tipo `string` pode ter até 2147483647 de caracteres. Cada caractere ocupa 16 bits. Potencialmente, o espaço necessário para manter objetos do tipo `string` é bem maior do que o espaço necessário para manter objetos da maior parte dos outros tipos. Por isso, em diversas linguagens de programação, inclusive em C#, objetos do tipo `string` são compartilhados para diminuir o espaço ocupado por eles.

Por exemplo, na linha 5 do código abaixo, ao utilizar a string literal “Rafael”, um objeto do tipo `string` é criado para armazenar essa sequência de caracteres. Esse objeto é colocado no **pool de strings**. Uma referência desse objeto é armazenada na variável `a`.

Na linha 6, a string literal “Rafael” é utilizada pela segunda vez. O objeto criado na linha anterior está no pool de strings e possui o mesmo conteúdo. Sendo assim, ao invés de criar um novo objeto, reaproveita-se o mesmo. Uma referência desse objeto é armazenada na variável `b`. Portanto, as variáveis `a` e `b` guardam referências para o mesmo objeto.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string a = "Rafael";
6         string b = "Rafael";
7     }
8 }
```

Código C# 7.2: strings literais

O pool de strings é um repositório no qual objetos do tipo `string` são armazenados para serem reaproveitados. No pool de strings não há strings repetidas (objetos do tipo `string` que armazenam a mesma sequência de caracteres).



Importante

Objetos do tipo `string` criados com o operador `new` não são colocados no pool de strings. Dessa forma, eles não são reaproveitados. No exemplo abaixo, dois objetos do tipo `string` com o mesmo conteúdo serão criados. A variável `a` armazenará uma referência do primeiro objeto e a variável `b` armazenará uma referência do segundo objeto.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string a = new string(new char[]{ 'R', 'a', 'f', 'a', 'e', 'l' });
6         string b = new string(new char[]{ 'R', 'a', 'f', 'a', 'e', 'l' });
7     }
8 }
```

Código C# 7.3: Criando strings com o operador `new`



Mais Sobre

Considere um objeto do tipo `string`. Esse objeto pode estar ou não no pool de strings. Através do método `Intern`, podemos obter uma referência para um objeto do tipo `string` com o

mesmo conteúdo que esteja no pool de strings.

No exemplo abaixo, um objeto do tipo `string` foi criado na linha 5 com o operador `new`. Consequentemente, esse objeto não é armazenado no pool de strings. Na linha 6, o método `Intern` devolve uma referência do objeto do tipo `string` que possui o mesmo conteúdo do objeto criado na linha 5.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string a = new string(new char[]{'R','a','f','a','e','l'});
6         string b = string.Intern(a);
7     }
8 }
```

Código C# 7.4: Utilizando o método Intern

No exemplo seguinte, um objeto do tipo `string` foi criado na linha 5 com a utilização da string literal “Rafael”. Consequentemente, esse objeto é armazenado no pool de strings. Na linha 6, o método `intern` devolve uma referência do objeto criado na linha 5.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string a = "Rafael";
6         string b = string.Intern(a);
7     }
8 }
```

Código C# 7.5: Utilizando o método intern

7.3 Imutabilidade



O conteúdo de um objeto do tipo `string` não pode ser alterado. Por isso, as strings são consideradas imutáveis. Alguns métodos dos objetos do tipo `string` podem sugerir que o conteúdo de uma string pode ser alterado. Contudo, esses métodos, geralmente, criam uma cópia da string original com determinadas alterações.

Por exemplo, na linha 5 do código abaixo, uma string foi criada de forma literal com o conteúdo “rafael”. Uma referência dessa string foi armazenada na variável `a`. Na linha 6, o método `ToUpper` foi utilizado. Esse método cria uma cópia da string original substituindo as letras minúsculas por letras maiúsculas e devolve uma referência dessa cópia. Dessa forma, a variável `b` armazenará uma referência de um objeto do tipo `string` com o conteúdo “RAFAEL”.

```

1 class Programa
2 {
3     static void Main()
4     {
5         string a = "rafael";
6         string b = a.ToUpper();
7
8         System.Console.WriteLine(a); // rafael
9         System.Console.WriteLine(b); // RAFAEL
10    }
```

11 }

Código C# 7.6: Utilizando o método toUpperCase

7.4 StringBuilder

Em algumas situações, é necessário “montar” uma string através de concatenações sucessivas. No exemplo abaixo, uma lista de números é criada através de uma sequência de concatenações.

```
1 class Programa
2 {
3     static void Main()
4     {
5         string numeros = "";
6         for(int i = 0; i < 1000; i++)
7         {
8             numeros = numeros + i + " ";
9         }
10        System.Console.WriteLine(numeros);
11    }
12 }
```

Código C# 7.7: Montando uma lista de números com concatenações

A cada concatenação, um novo objeto do tipo `string` é criado. No `for` do código anterior, duas mil concatenações serão realizadas ao total. Dessa forma, dois mil objetos do tipo `string` serão criados. A criação de muitos objetos prejudica a performance dos programas.

Nessas situações, podemos utilizar um objeto do tipo `StringBuilder`. Na linha 5 do exemplo abaixo, um `string builder` vazio foi criado. No corpo do `for`, o método `Append` foi utilizado para adicionar conteúdo ao `string builder`. Depois do laço, o método `Tostring` foi utilizado para obter uma `string` com o conteúdo armazenado no `string builder`.

```
1 class Programa
2 {
3     static void Main()
4     {
5         System.Text.StringBuilder builder = new System.Text.StringBuilder();
6         for(int i = 0; i < 1000; i++)
7         {
8             builder.Append(i);
9             builder.Append(" ");
10        }
11        string numeros = builder.ToString();
12        System.Console.WriteLine(numeros);
13    }
14 }
```

Código C# 7.8: Montando uma lista de números com StringBuilder

Diferentemente das concatenações com o operador `+`, o método `Append` não cria novos objetos. Portanto, a performance do programa não é comprometida.



7.5 Formatação

Considere um programa que registra as compras dos clientes nos caixas de um supermercado. Quando uma compra é finalizada, o programa deve exibir o cupom fiscal. Esse cupom deve conter o nome, o preço unitário e a quantidade de cada produto, além do valor total por produto e o valor total da compra. O preço unitário deve ser exibido com 2 casas decimais e alinhado à direita. A quantidade deve ser exibida sempre com 3 dígitos. O preço total por produto deve ser exibido com 2 casas decimais e alinhado à direita. Veja o exemplo abaixo.

| CUPOM FISCAL | | | |
|---------------|----------|-----|--------|
| PRODUTO | UNITÁRIO | QTD | TOTAL |
| Sorvete | 18.75 | 002 | 37.50 |
| Chocolate | 5.00 | 003 | 15.00 |
| Refrigerante | 3.89 | 014 | 54.46 |
| Bolacha | 1.80 | 058 | 104.40 |
| TOTAL: 211.36 | | | |

Separando as informações do cupom fiscal

Considere que a largura da tela do dispositivo permite a exibição de 42 caracteres por linha. Para separar as informações presentes no cupom fiscal, podemos utilizar linhas formadas por 42 caracteres iguais. No exemplo acima, escolhemos o caractere “-” e utilizamos o seguinte comando para exibir cada uma dessas linhas:

```
1 System.Console.WriteLine("-----");
```

Exibindo o título do cupom fiscal

O título “CUPOM FISCAL” deve ser exibido na segunda linha de forma centralizada. Como o dispositivo permite a exibição de 42 caracteres por linha e esse título possui 12 caracteres, são necessários 15 espaços em branco à esquerda do texto “CUPOM FISCAL” para centralizar esse título. Para resolver esse problema, podemos usar o método `WriteLine` acrescentado os 15 espaços em branco diretamente no código.

```
1 System.Console.WriteLine("          CUPOM FISCAL");
```

Até agora, utilizamos o método `WriteLine` com apenas um parâmetro. Contudo, há uma versão desse método que recebe uma `string` e uma quantidade variável de parâmetros. Essa `string` é chamada de **string de formatação**. Os demais argumentos são combinados com a `string` de formatação para definir a mensagem que será exibida.

No exemplo abaixo, quando a string de formatação for combinada com o segundo argumento do método `WriteLine`, o trecho “{0}” será substituído por “CUPOM FISCAL”. Dessa forma, o título do cupom fiscal será exibido da forma desejada.

```
1 System.Console.WriteLine("          {0}", "CUPOM FISCAL");
```

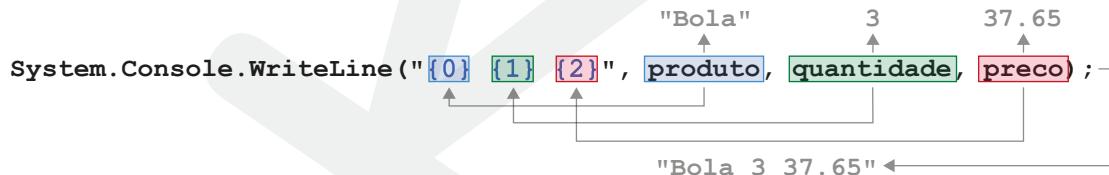
Os 15 espaços em branco acrescentados diretamente no código fonte podem gerar algumas dificuldades. Por exemplo, não é fácil verificar se há exatamente 15 espaços em branco olhando o código fonte. Para melhorar a legibilidade do código, podemos trocar o trecho “{0}” por “{0, 27}”. O número 27 indica a quantidade mínima de caracteres que devem ser utilizados para formar a mensagem que será exibida.

```
1 System.Console.WriteLine("{0,27}", "CUPOM FISCAL");
```

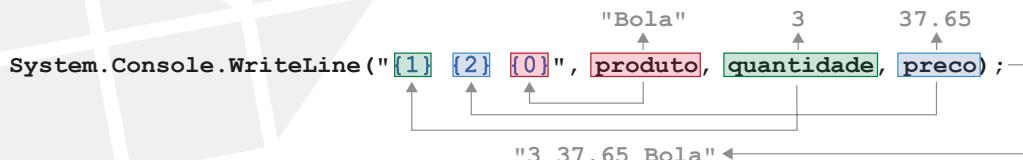
No exemplo acima, quando a string de formatação for combinada com o segundo argumento do método `WriteLine`, o trecho “{0,27}” será substituído por “CUPOM FISCAL”. Contudo, 15 espaços em branco serão adicionados à esquerda desse texto para completar a quantidade mínima de caracteres que é 27.

Parâmetros da string de formatação

Podemos definir zero ou mais parâmetros na string de formatação. Cada parâmetro deve ser definido dentro de colchetes. A correspondência entre os parâmetros da string de formatação e os argumentos do método `WriteLine` é definida através da indexação desses parâmetros. O índice de um parâmetro é um número inteiro que deve aparecer imediatamente após o caractere “{”.

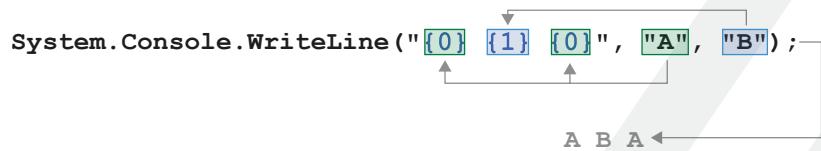


A ordem de utilização dos parâmetros na string de formatação não precisa ser igual à ordem dos demais argumentos do método `WriteLine`. No exemplo abaixo, o primeiro parâmetro da string de formatação foi combinado com o argumento “3”. O segundo parâmetro com o argumento “37.65”. Por fim, o terceiro parâmetro com o argumento “Bola”.



Dois ou mais parâmetros na string de formatação podem ser associados a um mesmo argumento do método `WriteLine`. Para isso, basta que esses parâmetros possuam o mesmo índice. No exemplo

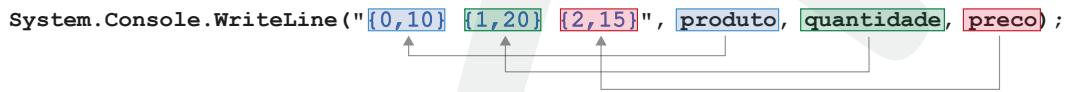
abaixo, o primeiro e o terceiro parâmetros da string de formatação foram associados ao argumento “A” e o segundo parâmetro ao argumento “B”. Dessa forma, a mensagem “A B A” seria exibida.



Definindo a quantidade mínima de caracteres

Para cada parâmetro da string de formatação, podemos definir a quantidade mínima de caracteres que devem ser utilizados quando a string de formatação é combinada com os demais argumentos do método `WriteLine`.

No exemplo abaixo, o primeiro parâmetro ocupa pelo menos 10 caracteres, o segundo parâmetro ocupa pelo menos 20 caracteres e o terceiro pelo menos 15 caracteres.



Alinhamento

No exemplo abaixo, quando a string de formatação for combinada com os demais argumentos do método `WriteLine`, o parâmetro “[0,16]” será substituído pelo argumento “PRODUTO”.

```
1 System.Console.WriteLine("{0,16}", "PRODUTO");
```

Como o parâmetro “[0,16]” define que a quantidade mínima de caracteres é 16 e o argumento “PRODUTO” possui apenas 7 caracteres, 9 espaços são adicionados à esquerda desse argumento. Sendo assim, seria exibido na saída padrão a string “ PRODUTO”. Portanto, o argumento “PRODUTO” seria alinhado à direita.

Os espaços utilizados para completar a quantidade mínima de caracteres podem ser inseridos à direita do argumento “PRODUTO”. Para isso, basta substituir o parâmetro “[0,16]” por “[0,-16]”. Veja o exemplo a seguir.

```
1 System.Console.WriteLine("{0,-16}", "PRODUTO");
```

Exibindo os títulos das colunas do cupom fiscal

O cupom fiscal possui quatro colunas para apresentar o nome, o preço unitário, a quantidade e o preço total de cada produto. Devemos definir a largura de cada coluna e o espaçamento entre elas, lembrando que cada linha permite no máximo 42 caracteres.

Para cada produto, vamos reservar 16 caracteres para o nome, 10 para o preço unitário, 3 para a quantidade e 10 para o preço total. Com essa divisão, as quatro colunas ocupam 39 caracteres na horizontal. Para separar as colunas utilizaremos 1 espaço em branco. Dessa forma, os 42 caracteres de cada linha serão ocupados.

| PRODUTO | UNITÁRIO | QTD | TOTAL |
|----------------------|----------|-----|--------|
| Sorvete | 18.75 | 002 | 37.50 |
| Chocolate | 5.00 | 003 | 15.00 |
| Refrigerante | 3.89 | 014 | 54.46 |
| Bolacha | 1.80 | 058 | 104.40 |
| TOTAL: 211.36 | | | |

Figura 7.2: Larguras das colunas do cupom fiscal

No exemplo abaixo, a string de formatação do método `WriteLine` possui quatro parâmetros para os títulos das colunas. Observe as quantidades mínimas de caracteres definidas para cada parâmetro. Note também que o sinal “-” foi utilizada no primeiro parâmetro para alinhar o primeiro título à esquerda.

```
1 System.Console.WriteLine("{0,-16} {1,10} {2,3} {3,10}", "PRODUTO", "UNITÁRIO", "QTD", "TOTAL") ;
```

Exibindo os nomes dos produtos

No cupom fiscal, a coluna que apresenta os nomes dos produtos deve ocupar 16 caracteres horizontalmente. Na string de formatação, podemos definir a quantidade mínima de caracteres como vimos anteriormente.

```
1 System.Console.WriteLine("{0,16}", nome);
```

Com a string de formatação acima, se o nome de um produto tiver menos do que 16 caracteres, espaços em branco serão acrescentados à esquerda desse nome para completar a quantidade mínima de caracteres. Dessa forma, os nomes dos produtos serão alinhados à direita.

Contudo, de acordo com a especificação do cupom fiscal, os nomes dos produtos devem ser alinhados à esquerda. Sendo assim, os espaços em branco utilizados para completar os 16 caracteres devem ser acrescentados à direita dos nomes dos produtos.

Para alinhar os nomes dos produtos à esquerda, devemos acrescentar o caractere “-” na string de formatação como no exemplo abaixo.

```
1 System.Console.WriteLine("{0,-16}", nome);
```

Agora, devemos considerar os nomes dos produtos que possuem mais do que 16 caracteres. Nesse caso, podemos utilizar o método `Substring` para extrair os primeiros 16 caracteres desses nomes.

```
1 nome = nome.Substring(0, System.Math.Min(nome.Length, 16));
2 System.Console.WriteLine("{0,-16}", nome);
```

Exibindo os preços unitários dos produtos

No cupom fiscal, a coluna que apresenta os preços unitários dos produtos deve ocupar 10 caracteres horizontalmente. Na string de formatação, podemos definir a quantidade mínima de caracteres como vimos anteriormente.

```
1 System.Console.WriteLine("{0,-16} {1,10}", nome, preco);
```

Os preços unitários devem ser exibidos com duas casas decimais. Para controlar a quantidade de casas decimais, devemos utilizar a formatação de números reais. Para isso, é necessário adicionar, na string de formatação, no parâmetro correspondente aos preços dos produtos, o trecho “:N”.

```
1 System.Console.WriteLine("{0,-16} {1,10:N}", nome, preco);
```

Por padrão, quando utilizamos a formatação de números reais, os valores são exibidos com duas casas decimais. Podemos modificar esse comportamento acrescentando a quantidade desejada de casas decimais imediatamente depois do caractere “N”. No exemplo abaixo, os preços seriam exibidos com 4 casas decimais.

```
1 System.Console.WriteLine("{0,-16} {1,10:N4}", nome, preco);
```

Exibindo as quantidades dos produtos

No cupom fiscal, a coluna que apresenta as quantidades dos produtos deve ocupar 3 caracteres horizontalmente. Na string de formatação, podemos definir a quantidade mínima de caracteres como vimos anteriormente.

```
1 System.Console.WriteLine("{0,-16} {1,10:N} {2,3}", nome, preco, quantidade);
```

As quantidades dos produtos são números inteiros. Para poder utilizar as opções de formatação de números inteiros, devemos adicionar o caractere “D”.

```
1 System.Console.WriteLine("{0,-16} {1,10:N} {2,3:D}", nome, preco, quantidade);
```

Como as quantidades dos produtos devem ser exibidas sempre com três caracteres, eventualmente, zeros devem ser adicionados à esquerda dessas quantidades. Para que isso ocorra automaticamente, devemos substituir o parâmetro “{2,3:D}” por “{2:D3}” na string de formatação.

```
1 System.Console.WriteLine("{0,-16} {1,10:N} {2:D3}", nome, preco, quantidade);
```

Exibindo os preços totais dos produtos

No cupom fiscal, a coluna que apresenta os preços totais dos produtos deve ocupar 10 caracteres horizontalmente. Na string de formatação, podemos definir a quantidade mínima de caracteres como vimos anteriormente.

```
1 System.Console.WriteLine("{0,-16} {1,10:N} {2:D3} {3,10:N}", nome, preco, quantidade, total);
```

Lembre-se que por padrão, na formatação de números reais, os valores são exibidos com duas casas decimais.

Exibindo o valor total da compra

O total da compra deve ser exibido com duas casas decimais em uma linha separada. Para isso podemos utilizar a string de formatação abaixo.

```
1 System.Console.WriteLine("TOTAL: {0:N}", totalDaCompra);
```



Mais Sobre

Mais detalhes sobre o funcionamento das formatações podem ser obtidos no endereço:

<http://msdn.microsoft.com/en-us/library/txafckwd.aspx>

7.6 Formatação de Data e Hora



Normalmente, o formato padrão para exibir data e hora varia de país para país ou de região para região. Por exemplo, os brasileiros estão mais acostumados com o formato de data “dia/mês/ano”. Por outro lado, os americanos costumam utilizar o formato “mês/dia/ano”.

Em C#, podemos formatar datas e horas facilmente. No código exemplo abaixo, a formatação “dia/mês/ano hora:minuto:segundos” está sendo aplicada.

```
1 System.DateTime fundacaoK19 =
2     new System.DateTime(2010, 7, 27, 10, 32, 15);
3
4 string fundacaoK19Formatada = fundacaoK19.ToString("dd/MM/yyyy HH:mm:ss");
```

Código C# 7.27: Aplicando o formato "dia/mês/ano hora:minuto:segundos"

Na máscara de formatação, devemos utilizar os caracteres especiais para definir o formato desejado. Veja o que cada caractere indica.

d: dia

M: mês

y: ano

H: hora

m: minutos

s: segundos

Quando o caractere **d** é utilizado de forma simples na máscara de formatação, os dias de 1 até 9 são formatados com apenas um dígito. Quando utilizamos **dd**, os dias de 1 até 9 são formatados com apenas dois dígitos (01, 02, 03, ..., 09). Analogamente, para o mês, ano, hora, minutos e segundos.

7.7 Resumo



- 1 ► Para controlar um objeto do tipo `string`, é necessário utilizar uma referência desse objeto.
- 2 ► Strings literais são armazenadas no pool de strings.
- 3 ► O pool de strings permite que objetos do tipo `string` sejam reutilizados a fim de diminuir o consumo de memória.
- 4 ► O conteúdo de um objeto do tipo `string` não pode ser alterado.
- 5 ► Ao invés de realizar concatenações sucessivas para “montar” uma string, devemos utilizar objetos do tipo `System.Text.StringBuilder`, pois um número excessivo de concatenações pode prejudicar o desempenho dos programas em C#.
- 6 ► O método `WriteLine` pode ser utilizado para exibir strings formatadas na saída padrão.
- 7 ► Datas e horas podem ser formatados com através do método `Tostring`.