Aula 1 - Introdução ao .NET

Docupedia Export

Author:Silio Leonardo (CtP/ETS)
Date:28-Mar-2023 17:42

Table of Contents

- Introdução
- Linguagens Compiladas e Interpretadas
- Ahead-Of-Time Compilation, Representações Intermediárias e Linguagens Híbridas
- Java e Just-In-Time Compilation
- Breve História do C#
- Tipagem
- Paradigmas de Programação
- Ecossistema .NET
- dotnet CLI
- Função Main e Arquivo Top-Level
- Console, Input e Output
- Variáveis
- Conversões
- Operadores

1 Introdução

Este curso é introdutório ao C# bem como vários aspectos da linguagem e da computação em si, perfeito para quem deseja desenvolver-se mais profundamente na área. Contudo, ele não é um curso introdutório de programação. Recomenda-se fortemente que tenha-se lógica antes de adentrar o curso, visto que não se irá ensinar com cuidado coisas como For, If, conceitos básicos de variávies, etc. Apenas o necessário de como usar tais coisas no C#.

2 Linguagens Compiladas e Interpretadas

Para que um programa seja executado em um computador é necessário que exista alguma forma de transformar o texto do código fonte em algo que o computador compreenda, o então chamado código de máquina.
Para isso, existem o que chamamos de **Compilador**. Um Compilador é um programa que recebe arquivos de código fonte de uma linguagem de programação e faz a sua conversão para código de máquina.
É importante perceber que isso significa que o programador entra com um código e recebe como saída um executável (.exe) que é capaz de ser executado em um computador. Mas note que:

- Cada com sistema operacional pode ter dependências (bilbiotecas base) e ainda ser 32 ou 64 bits (veremos isso mais a frente).
- Cada processador tem um código de máquina único.

Assim você precisa compilar para cada arquitetura computacional. Quando trabalhar com linguagens compiladas você precisa ter isso em mente. Chamamos o alvo da compilação simplismente de target.
Exemplos de linguagens compiladas: C, C++, Fortran, Cobol.
Por outro lado existe também outra técnica de tradução que é o **Interpretador**. A ideia é simples porém brilhante: Faça um programa que leia um código fonte e execute imediatamente. Por exemplo: Ao ler uma linha de código 'print("ola")' o programa imediatamente executa o 'printf' da linguagem C, por exemplo, tendo como parâmetro 'ola'. Se ele ler uma linha 'x = 4', ele imediatamente salva em algum lugar o valor 4 apontando que o mesmo se encontra numa variável chamada 'x'.
A ideia é inteligente pois podemos fazer um Interpretador para cada arquitetura e um mesmo programa pode rodar em diferentes arquiteturas, pois será interpretado por diferentes programas para diferentes arquiteturas mas que tem um mesmo funcionamento.
Por isso, linguagens como Python rodam em todo lugar: Basta ter um interpretador Python (que é escrito em C) compilador para a arquitetura destino.
O mesmo acontece com JavaScript, Css e Html (as duas últimas não são linguagens de programação, apenas linguagens de estilização e estruturação, respectivamente). Um navegador não é nada mais, nada menos, que um interpretador que recebe código online e apresenta o seu processamento.
A desvantagem das linguagens interpretadas é que costumam ser mais lentas, afinal, é preciso ler cada linha de código, interpretar o que ela faz e então executar o código de máquina equivalente escrito em C. Além disso, para que uma aplicação funcione no seu cliente, é necessário que você exponha o código original do seu software para ele.
Outra vantagem é que na linguagem interpretada você não precisa passar por um processo para que ela se torne uma linguagem executável (.exe) toda vez que quiser testá-la, tornando-a, em geral, mais ágil.

Característica	**Compilada**	**Interpretada**
Velocidade de Compilação	Lenta	Inexistente
Velocidade de Execução	Rápida	Mais Lenta
Compatibilidade sem Recompilação	Não	Sim

3 Ahead-Of-Time Compilation, Representações Intermediárias e Linguagens Híbridas

Uma técnica poderosa utilizada pelas linguagens modernas é a **Compilação Antecipada**, Ahead-Of-Time, ou simplismente AOT. Está técnica consiste em traduzir um código mais alto-nível (mais inteligível para humanos) em um código mais baixo-nível (mais próximo ao código de máquina).
Ou seja, você pode transformar JavaScript em C++, por exemplo. Mas é muito mais do que isso. A utilização mais comun é criar uma representação intermediária.
Essa representação intermediária (IR) do código pode se utilizada para diminuir o trabalho da compilação.
Por exemplo, você pode ter várias linguagens que se transformam em uma mesma IR, assim todas elas seriam compatíveis, ao passo que seria necessário apenas transformar o IR em código de máquina uma única vez.
Entenda: Se nós temos 3 linguagens e 5 arquiteturas desejadas, seguindo os conceitos tradicionais precisariamos desenvolver 15 compiladores, 1 para cada linguagem sendo convertida para cada arquitetura. Usando AOT, precisamos de 3 compiladores, de cada linguagem para a IR, e 5 compiladores, da IR para cada arquitetura. Assim 8 no total e, de quebra, ganhamos alta compatibilidade entre as linguagens.
E o mais poderoso: Podemos construir um compilador para transformar da linguagem fonte na IR e criarmos um interpretador para a IR executar. Assim temos linguagens de compilação **Híbridas**.

4 Java e Just-In-Time Compilation

A grande vantagem das linguagens Híbridas são linguagens como o **Java**. Muitas linguagens como Java, Kotlin, Scala, são convertidas na mesma IR que é levada ao cliente e interpretada em diferentes máquinas. Basta ter um software de tempo de execução chamada JVM ou Java Virtual Machine, instalada na arquitetura alvo.
Isso também permite que o mesmo código uma vez compilado (IR) execute em praticamente qualquer arquitetura, basta existir uma implementação da JVM para ela.
Além disso, o Java utliza uma ténica chamada de **Just-In-Time** ou JIT. O JIT é uma forma de interpretação aperfeiçoada: Ela compila a representação intermediária no momento em que ela deveria ser executada, possibilitando a execução diretamente na CPU do computador, realizando ainda optimizações que não poderiam ser feitas em outro momento, possibilitando que as desvantagens de linguagens interpretadas sejam liquidadas, desempenhando muito bem em diferentes arquiteturas.
Além disso, apenas o código usado é convertido, partes do código não executadas não são convertidas e as partes convertidas ficam salvas, fazendo com que em futuras execuções o desempenho da aplicação melhore ainda mais.
Ou seja, JIT é poderoso, contudo, complexo de se implementar, mas dá um alto nível de flexibilidade e compatibilidade.

5 Breve História do C#

A Microsoft desejava otimizar o Java ainda mais para que tivesse um desempenho melhor no Windows, tentando tornar seu sistema operacional (e carro chefe da empresa) ainda mais atrativo. Assim foi criado o J++, basicamente, o mesmo Java, porém que rodaria em uma nova implementação da JVM que só funcionária no Windows.

A Sun, criadora do Java, não gostou disso e como havia algumas quebras de patentes foi instaurado um processo contra a Microsoft que perdeu nos tribunais. Assim o J++ morreu e a empresa de Bill Gates resolveu criar sua própria linguagem concorrente do Java.

Chamada inicialmente de Cool, essa linguagem também usaria uma máquina virtual e JIT para a compilação. Naquele tempo Steven Ballmer, CEO da Microsoft, tinha um cabeça, digamos, um pouco fechada. A nova linguagem vinha para ter um foco em Windows e até por isso, e por no começo estar incompleta, C#, como então foi batizada a linguagem, virou motivo de piada, uma cópia de Java.

O que não se esperava é que a linguagem vingaria tanto. Mudando os preceitos para rodar em mais lugares e melhor, trazendo novas atualizações e características únicas (algumas coisas implementadas no C# 2007 só vieram aparecer no Java depois de 2015, ou seja, Java começou a se inspirar na linguagem da mesma forma que o contrário ocorreu), C# se tornou extremamente competitiva e poderosa. Embora a velha guarda custe a perceber.

Com a compra da Unity pela Microsoft, do Xamarin entre outras novas soluções, hoje é possível usar C# para programar para: Desktop, Android, IOS, Web Frontend, Web Backend, Nuvem, Microcontroladores, aplicações IOT, Playstation, Xbox, Switch, entre outros.

6 Tipagem

Um tópico interessante da Ciência da Computação que nos ajuda a compreender melhor as características da linguagem C#, bem como outras, é a **Tipagem**.
A tipagem é como um plano Cartesiano que classifica as linguagens em 2 eixos:

- Tipagem Dinâmica vs Tipagem Estática
- Tipagem Forte vs Tipagem Fraca

Uma Tipagem Estática é uma linguagem onde os tipos de todas as variáveis são definidos ainda na fase de compilação. Ou seja, se a variável 'x' contém um número, isso será identificado ainda na compilação e 'x' só poderá receber um número. Isso acontece em linguagens como C ou Java.
Uma Tipagem Dinâmica é uma linguagem onde as variáveis podem mudar de tipos o tempo todo. Como Python e JavaScript, por exemplo. Então escrever 'x = 2' e em seguida 'x = "oi"' é aceitável numa linguagem Dinâmica, mas não em uma Estática.
Já um Tipagem Forte é onde os dados tem tipo bem definido. Por exemplo, vamos supor que criamos um variável 'x' valendo 4. Ao tentar obter o resulado da seguinte expressão 'x[0]' teríamos um erro pois 4 não é um vetor então não podemos acessar a posição zero dele.
Para uma Tipagem Fraca a expressão retornaria algum valor, mesmo que o mesmo seja tratado como 'indefinido'. O fato de não aparecer um erro faz com que não percebamos que algo deu errado. Isso pode trazer flexibilidade mas aumenta a quantidade de bugs.
Abaixo uma pequena tabela com exemplos:

Linguagem	Força	Dinamicidade
C K&R (bem antigo)	Fraca	Estática
C Ansi (mais moderno)	Forte	Estática
Java	Forte	Estática
C++	Forte	Estática
JavaScirpt	Fraca	Dinâmica
PHP	Fraca	Dinâmica
Python	Forte	Dinâmica
Ruby	Forte	Dinâmica

Nota: Aqui você percebe que Java e JavaScript tem muito pouco em comum.

Aqui vale uma ressalva: Permitir que várias conversões automáticas aconteçam como no JS torna uma linguagem mais fraca. Por exemplo ao computar '{} + {}', um dicionário vazio mais outro dicionário vazio se obtém '[object Object][object Object]', pois o JS tenta converter para um texto apresentável. O exemplo anterior 'x[0]' ou até mesmo '4[0]' apresenta valor indefinido. Isso é muito diferente do que fazer isso em uma linguagem como o C antigamente, onde 'x[0]' é literalmente acessar a memória do computador no endereço 4, pois o C entende (ou entendia) tudo como números e mais números e não da significado para seus valores. Assim é fácil perceber que existem diferentes níves de linguagem fraca.
Neste contexto, onde o C# se encontra? Bem, ao procurar online você verá que C# é uma linguagem Forte e Estática, é assim que a usamos. Porém, é importante salientar que C# possui recursos de tipagem dinâmica incluso no seu baú de recursos. E não só isso, C# possui várias características que podem tornar a linguagem um pouco menos forte, embora jamais deixe de ser uma linguagem inerentemente forte.

7 Paradigmas de Programação

Um paradigma de programação é a maneira como uma linguagem de programação se estrutura para orientar o pensamento do programador. Existem diversas abordagens para se programar; estamos mais acostumados com a programação Imperativa e Estruturada, ou seja, dizemos como o programa deve fazer o processamento de dados e estruturamos isso em várias funções e variáveis, executando estruturalmente linha a linha.
Contudo, existem outras propostas: Ainda como linguagem imperativa temos a Orientação a Objetos, que muda a forma como modularizamos o estado (variáveis) e comportamento (métodos) do nosso programa, escondendo dentro de escopos e afastando da maneira estruturada onde deixamos tudo no mesmo programa.
Por outro lado, temos os paradigmas declarativos, onde se fala o que se deseja fazer, não se importando no como. Dentro deste mundo existem paradigmas como o Funcional, onde a definição de funções, uso de funções como dados (o que sim, parece confuso e de fato é), além de vários recursos prontos tornam a programação muito diferente da que estamos acostumados, mas pode aumentar muito a produtividade.
O C# lhe permite escrever código estruturado, embora seja uma linguagem inerentemente Orientada a Objetos. E dentro da programação imperativa, temos ainda a programação Orientada a Eventos que não será abordado neste curso, mas C# da suporte a mesma. C# também tem suporte a programação funcional entre outros aspectos de programação declarativa. Ou seja, C# é uma linguagem vasta considerada multi-paradigma.

8 Ecossistema .NET

.NET é a plataforma de desenvolvimento criado pela Microsoft. Ela, em si, é muito maior que somente o C#. Por isso, os profissionais se dizem programaodres .NET, e não apenas programadores C#. Vamos explorar rapidamente a fim de entender o que é o .NET. Para isso vamos aprender os seus componentes:

- .NET possui 3 linguagens principais: C#, Visual Basic e F# (uma interessante linguagem funcional). Todas são convertidas para uma linguagem intermediária (ou seja, um linguagem que é uma representação intermediária/IR) chamada CIL ou Common Intermediate Language. Ou seja, as linguagens são intercambiáveis e totalmente integráveis.
- CLR, Common Language Runtime é a máquina virtual que transforma o CIL em código nativo capaz de rodar em muitas arquiteturas diferentes.
- .NET Framework é uma espécie de biblioteca padrão com centenas de recursos para realizar qualquer tarefa: Trabalhar com arquivos, Criptografia, Comunicação em Redes, Desenvolvimento Web e afins. Para que programas rodem você precisa ter instalado na sua máquina as bibliotecas do .NET Framework na versão desejada.
- .NET Core: Aqui entra um discussão interessante que confunde muitos. Como anteriormente mencionado, o C# era voltado apenas para Windows nos anos 2000 até 2016. Embora C# pudesse rodar em qualquer lugar, o .NET Framework muitas vezes tinha implementações apenas para Windows. Com isso, no final de 2014 anunciou-se o .NET Core que vinha como uma nova implementação melhorada, tornando até mesmo várias funções antigas mais rápidas. Assim, por muito tempo tivemos dois frameworks principais: .NET Framework ou simplesmente .NET avançado até as versões 4.X, e o .NET Core, avançado até a versão 3.1. Neste momento, o .NET Framework 'morreu'. A partir do .NET Framework 5.0, temos na verdade o .NET Core que a partir desta versão, chamamos apenas de .NET ou .NET Framework. Então se você está usando o .NET na versão 4.6, por exemplo, você está usando o antigo .NET. Se você está usando o .NET Core, ou o .NET 5 em diante, você está usando o novo framework que surgiu em 2016.
- .NET Standard é uma especificação que aponta o mínimo que um Framework precisa ter para se tornar mais compatível. Explicando: Quando desenvolvedores queriam criar bibliotecas compatíveis tanto com .NET Core quanto .NET Framework, eles utilizavam apenas o que era assegurado estar no .NET Standard, que seria basicamente as implementações compartilhadas por ambos os frameworks. Utilizar algo que não estava no .NET Standar poderia fazer com que a biblioteca ficasse incompatível, ou com .NET Framework, ou com o .NET Core.
- .NET SDK ou SDK (Software Development Kit) é o Kit de desenvolvimento do .NET. É possível instalar apenas os recursos de execução, sendo um usuário, e executar programas .NET sem ter o Kit de desenvolvimento. Para os desenvolvedores, basta instalar o SDK e ter acesso as ferramentas para construir novas aplicações.
- .NET CLI (Command Line Interface) é um programa que utilizamos para criar projetos, testar, executar entre várias operações no momento de gerenciar projetos em .NET. Com o .NET SDK instalado, basta utilizar o comando 'dotnet' no terminal para utilizar o .NET.

9 dotnet CLI

Como já comentado, o dotnet CLI é um programa que te ajuda a desenvolver em C#. Apartir dele você irá criar e executar aplicações. Aqui uma lista de comandos que você pode executar em qualquer terminal PowerShell ou semelhante:

- dotnet --version: Mostra a versão instalada do dotnet
- dotnet --help: Mostra lista de comandos disponíveis
- dotnet new --list: Mostra a lista de tipos de projetos que você pode desenvolver
- dotnet new console: Cria uma nova aplicação console
- dotnet new gitignore: Cria um arquivo gitignore especial para C# a ser utilizado junto com o github
- dotnet run: Roda o programa
- dotnet build: Compila o projeto e mostra os possíveis erros de compilação, sem executar a aplicação
- dotnet clean: Limpa todos os arquivos de compilação. Pode ser útil para se livrar de alguns erros desconhecidos
- dotnet add package : Baixa e instala um pacote da internet (nuget).

10 Função Main e Arquivo Top-Level

Assim como programas C/C++, todo programa C# se inicia na função Main. É comum ver exemplos do famoso Hello World apresentados da seguinte forma:

```
1  using static System.Console;
2
3  namespace MeuProjeto
4  {
5      public class Program
6      {
7          public static void Main(string[] args)
8          {
9              WriteLine("Olá mundo!");
10         }
11     }
12 }
```

Mas não se preocupe com a aparente complexidade de um programa simples, vamos entender cada um desses 3 elementos que você vê (namespace, class e a função Main) ao longo do curso. Por enquanto vamos falar da função Main, já que ela não é comum para programadores de funções como Python e JavaScript.
Uma função Main é o ponto inicial de todo programa C# e só devemos ter uma única função Main. Não se preocupe com a declaração da função e seus detalhes pois vamos abordar cada aspecto separadamente, bem como a forma de se declarar uma função no C#.
De inicio, é bom salientar que void significa que a função não deve retornar nada. Se você aprendeu funções através de Python, JavaScript, entre outras linguagens não está acostumado a declarar o tipo de retorno da função, mas isso acontece em C# e temos a opção void, isentando a função de retornar qualquer coisa.
Outro aspecto é o vetor de string chamado comumente de args. Esses são os argumentos para o programa. Todo programa possui argumentos que mudam seu comportamento. O melhor exemplo é o dotnet CLI, onde ao executar o programa 'dotnet', mandamos a string 'new' como parâmetro. A cada espaço de divisão, um novo parâmetro é gerado. Note que na grande maioria das vezes esse parâmetro é ignorado.
Apesar disso tudo, o C# evoluiu para ser mais simples e menos confuso. Então na versão 6.0 em diante temos os arquivos Top-Level. Em todo programa você pode ter um único arquivo Top-Level. Ele não possui nenhuma das estruturas no exemplo anterior. Todo seu código será jogado dentro de uma função Main artificial gerada na compilação. Assim o código a seguir, torna-se válido, útil e simples:

```
1  using static System.Console;
2
3  WriteLine("Olá mundo!");
```

Note que 2 arquivos Top-Level geram um erro pois acabam por gerar 2 funções Main o que é inválido.
Algumas coisas que podem passar pela sua cabeça são as seguintes:

- **Então eu não consigo mais acessar o vetor 'args'?** Na verdade consegue, bastar usar a palavra reservada args.
- **Não é possível colocar uma função em um arquivo Top-Level? Já que isso seria colocar uma função dentro da outra.** Felizmente, C# suporta funções declaradas dentro de outras funções.

- **Posso usar outras estruturas como no primeiro exemplo? Existe alguma limitação?** Existem algumas, mas a maioria das estruturas que não são códigos executáveis são automaticamente jogadas para fora da função Main.

Outro detalhe importante é que você precisa de código executável no Top-Level para que ele reconheça a existência de uma função Main. Você verá que existem estruturas que se colocadas sozinhas no arquivo Top-Level acabam por indicar para o compilador que na verdade aquele não é um arquivo Top-Level e que não existe uma função Main e isso acarreta em um erro de compilação.

11 Console, Input e Output

Como você pode observar, para apresentar informações na tela basta usar o print do C#, chamado de WriteLine. Ele é muito útil para várias coisas, como checar erros ou fazer aplicações para Console, que é o nome dado para aplicações de tela preta/Terminal. Para que você use-a é necessário importá-la da seguinte maneira:

using static System.Console;

Depois de incluir esse código no topo do seu arquivo você tem acesso a muitas funcionalidades:

- WriteLine: Escreve algo na tela e pula uma linha logo a seguir.
- Write: Escreve algo na tela.
- ReadLine: Lê a próxima linha digitada pelo usuário e retorna o valor.
- ReadKey: Lê apenas um caracter digitado pelo usuário e retorna suas informações. Você pode mandar o parâmetro 'true' para evitar que o que for digitado apareça na tela.
- Beep: Emite um som em determinada frequência.
- Clear: Limpa o Console.
- BackgroundColor: Variável onde você pode definir a cor do fundo do que será escrito.
- ForegroundColor: Variável onde você pode definir a cor da fonte do que será escrito.
- Title: Variável onde você pode definir o titulo do Console.
- CursorLeft: Variável onde você pode definir a posição horizontal do cursor.
- CursorTop: Variável onde você pode definir a posição vertical do cursor.
- CursorVisible: Variável onde você pode definir se o cursor é visível.
- WindowWidth: Variável onde você pode definir a largura da janela do Console.
- WindowHeight: Variável onde você pode definir a altura da janela do Console.

Entre outras funções. Segue um pequeno exemplo:

```
1  using static System.Console;
2
3  WriteLine("Digite algo...");
4  string texto = ReadLine();
5
6  BackgroundColor = ConsoleColor.White;
7  ForegroundColor = ConsoleColor.Blue;
8  CursorVisible = false;
9
10 Clear();
11
12 Write("O usuário digitou: ");
13
14 ForegroundColor = ConsoleColor.Red;
```

```
15  WriteLine(texto);
```

12 Variáveis

Como você pode perceber no último exemplo, para declarar uma variável em C# você só precisa digitar seu tipo seguido do nome e atribuição. Como discutido anteriormente, C# tem tipagem estática em 99% do tempo e você precisa apontar o tipo usado. Existem uma grande variedade de tipos:

```
1  byte b1 = 0; // Inteiro com 2^8 valores de 0 a 255 (naturalmente sem sinal).
2  sbyte b2 = 0; // Inteiro com sinal (signed) com 2^8 valores de -128 a 127.
3
4  short s1 = 0; // Inteiro com 2^16 valores de -32'768 a 32'767.
5  ushort s2 = 0; // Inteiro sem sinal (unsigned) com 2^16 valores de 0 a 65'535.
6
7  int i1 = 0; // Inteiro com 2^32 valores de -2'147'483'648 a 2'147'483'647.
8  uint i2 = 0; // Inteiro sem sinal (unsigned) com 2^32 valores de 0 a 4'294'967'295.
9
10 long l1 = 0; // Inteiro com 2^64 valores de -9'223'372'036'854'775'808 a 9'223'372'036'854'775'807.
11 ulong l2 = 0; // Inteiro sem sinal (unsigned) com 2^64 valores de 0 a 18'446'744'073'709'551'615.
12
13 nint n1 = 0; // int se o sistema for de 32 bits, longo se o sistema for de 64 bits, ou seja, um inteiro nativo.
14 nuint n2 = 0; // O mesmo que o nint, porém, podendo ser uint ou ulong.
15
16 char c = 'a'; // Caracter.
17 string s = "texto"; // Cadeia de caracteres/texto.
18
19 bool b = true; // Booleano, ou seja, verdadeiro ou falso (true/false).
20
21 float f = 0f; // Tipo de ponto flutuante, isso é, armazena números com virgula até uma determinada potência com uma
   quantidade limitada de casas decimais.
22 double d = 0.0; // Semelhante ao float, porém com o dobro de armazenamento, podendo ter mais casas decimais e representar
   maiores números.
23 decimal m = 0m; // Tipo desenvolvido para guardar dinheiro. Dobro do armazenamento do double. Menor range de valores mas
   muito mais casas decimais.
24
25 var x = 0; // Descobre automaticamente o tipo da variável de forma Implícita, sem precisar escrever.
26 // x = "oi"; // Mas não permite a mudança do tipo da variável.
27
28 dynamic y = 0; // Desabilita verificações e guarda qualquer tipo de dado.
29 y = "oi"; // Pode mudar de tipo ao longo do programa, porém, pouco utilizado.
30
31 int i3 = int.MaxValue; // Obtém maior valor possível para int.
```

```
32  int i4 = 0b_0000_0101; // Valor binários.
33
34  //int i4 = 5; // Erro, não é possível declarar duas variáveis com mesmo nome.
35  i4 = 6; // Ok.
36
37  //byte b3 = 300; // Erro, valor muito grande para byte.
38
39  int[] vetor = new int[10]; // Criando um vetor com 10 posições. Todas as posições devem ser um int. Se iniciam todas as
    posições como 0.
40  var vetor2 = new string[] { "Olá", "Mundo", "!" }; // Inicialização com valores em uma variável Implícita usando var.
```

Usar variáveis ocupa menos espaço e pode ser útil e muitas aplicações, então vale a pena compreender um pouco dos tipos e seus usos.

13 Conversões

Converter valores é uma funcionalidade importante, básica e corriqueira durante a programação. Assim sendo é importante aprender a converter tipos em uma linguagem com a tipagem do C#. Duas são muito comuns:

- String Parse

```
1  string numeroEmTexto = ReadLine();
2  int numero = int.Parse(numeroEmTexto);
```

- Type Cast

```
1  int valor = 100;
2  // byte outroValor = valor; // Erro! valor pode ser um número muito grande para uma variável byte, ou até negativo. Por
   isso o C# bloqueia essa operação.
3  byte outroValor = (byte)valor; // Ok.
```

14 Operadores

Também temos vários operadores em C#, bem como em outras linguagens. A seguir uma lista de várias operações váldas no C#:

```
1  int a = 3;
2  int b = 2;
3
4  int r1 = a + b; // 5
5  int r2 = a - b; // 1
6  int r3 = a * b; // 6
7  int r4 = a / b; // 1 (divisão de inteiros)
8  int r5 = a % b; // 1 (resto da divisão)
9  int r6 = a << b; // 12 shift (operação binária) equivalente a 'a * 2^b'.
10 bool r7 = a > b; // true
11 bool r8 = a <= b; // false
12 bool r9 = a == b; // false
13 bool r10 = r7 && r8; // Operador And
14 bool r11 = r7 || r8; // Operador Or
15 bool r12 = !r7; // Operador Not
16
17 int r13 = r1++; // 5. Retorna r1 (5, no caso) e depois soma 1 a variável r1 que agora vale 6.
18 int r14 = ++r1; // 7. Soma 1 na variável r1 que agora vale 7 e retorna este valor após isso.
```