

Curso iniciante C# / .NET

<https://docs.microsoft.com/pt-br/learn/paths/csharp-logic/>

Expressões booleanas

Ao trabalhar com instruções de decisão, estamos interessados em expressões booleanas. Em uma expressão booleana, o runtime avalia os valores, operadores e métodos para retornar um único valor true ou false.

Igualdade == : Se dois valores em ambos os lados do operador de igualdade forem equivalentes, a expressão retornará true. Caso contrário, retornará false.

Desigualdade != : Talvez você deseje verificar se os dois valores não são iguais. Você usa o operador de desigualdade != entre dois valores para avaliar a igualdade.

Exemplo1:

Editor do .NET

```
1 Console.WriteLine("a" == "a");
2 Console.WriteLine("a" != "a");
3 Console.WriteLine("a" == "A");
4 Console.WriteLine("a" != "A");
```

Saída

```
True
False
False
True
```

Exemplo2:

Editor do .NET

```
1 string myValue = "a";
2
3 Console.WriteLine(myValue == "a");
4 Console.WriteLine(myValue != "a");
```

Saída

```
True
False
```

Moldar dados.

Você pode ficar surpreso com o fato de que a linha **Console.WriteLine("a" == "A");** gera a saída **false**. Ao comparar cadeias de caracteres, o uso de maiúsculas e minúsculas faz diferença.

Além disso, considere esta linha de código:

```
Editor do .NET
1 Console.WriteLine("a" == " a");
```

Ao adicionarmos um espaço em branco na cadeia de caracteres também gera uma saída **false**.

Em alguns casos, isso é perfeitamente aceitável. No entanto, você pode querer uma verificação sem esse tipo de distinção. Para isso precisamos moldar os dados antes de verificar uma comparação.

ToUpper(): Verifica se ambas as cadeias de caracteres são todas maiúsculas ou minúsculas.

ToLower(): Verifica qualquer valor de cadeia de caracteres.

Trim(): Remover um espaço em branco à esquerda ou à direita em qualquer valor de cadeia de caracteres.

```
Editor do .NET
1 string value1 = " a";
2 string value2 = "A";
3
4 Console.WriteLine(value1.Trim().ToLower() == value2.Trim().ToLower());
```

```
Saída
True
```

Avaliar comparações.

Ao trabalhar com tipos de dados numéricos, você desejará determinar se um valor é maior, menor ou igual a outro valor.

- Maior que.
- < Menor que.
- == Igual a.
- <= Menor ou igual a.
- >= Maior ou igual a.

Editor do .NET

```
1 Console.WriteLine(1 > 2);  
2 Console.WriteLine(1 < 2);  
3 Console.WriteLine(1 == 1);  
4 Console.WriteLine(2 <= 1);  
5 Console.WriteLine(2 >= 2);
```

Saída

```
False  
True  
True  
False  
True
```

Métodos que retornam um valor booliano.

⚠ Observação

Alguns tipos de dados têm métodos que executam tarefas de utilitário bastante úteis. O tipo de dados `String` tem muitos deles. Vários retornam um valor booliano incluindo `Contains()`, `StartsWith()` e `EndsWith()`. Você pode aprender mais sobre eles no módulo do Microsoft Learn "Manipular dados alfanuméricos usando métodos da classe `String` no C#".

Expressão de invocação de método.

Editor do .NET

```
1 string pangram = "The quick brown fox jumps over the lazy dog.";
2
3 Console.WriteLine(pangram.Contains("fox"));
4 Console.WriteLine(pangram.Contains("cow"));
```

Saída

```
True
False
```

O método **Contains()** verificar se determinada cadeia de caracteres possui uma cadeia de caracteres específica.

Negação lógica.

O termo "negação lógica" refere-se ao operador **!**. A adição do operador **!** antes de uma expressão condicional, tal como uma chamada de método, verifica se a expressão é falsa.

Editor do .NET

```
1 Console.WriteLine(! (1 > 2));
```

Saída

```
True
```

Esse código nega que 1 seja maior que 2 então a saída é true.

Editor do .NET

```
1 string pangram = "The quick brown fox jumps over yhe lazy dog.";
2
3 Console.WriteLine(!pangram.Contains("fox"));
4 Console.WriteLine(!pangram.Contains("cow"));
```

Saída

```
False
True
```

A linha 3 nega que exista a cadeia de caracteres “fox” na variável **pangram**.

A linha 4 nega que exista a cadeia de caracteres “cow” na variável **pangram**.

Então o resultado é **false** e **true** respectivamente.

Operador condicional ternário (? :).

Avalia uma expressão booliana e retorna o resultado da avaliação de uma de duas expressões, dependendo se a expressão booliana é avaliada como verdadeira ou falsa.

Embora seja certamente possível usar a construção de **ramificação if ... elseif ... else** para expressar essa regra o operador condicional ternário usa um formato compacto que economiza algumas linhas de código e, possivelmente, torna a intenção do código mais clara.

Este é o formato básico:

<avalie esta condição> **?** <se for verdadeiro, retorne isso> **:** <se for falso, retorne isso>

Editor do .NET

```
1  int saleAmount = 1001;
2
3  int discount = saleAmount > 1000 ? 100 : 50;
4
5  Console.WriteLine($"Discount: {discount}");
```

Saída

```
Discount: 100
```

A regra de negócio seria: Se a compra for maior que 1000 o desconto será 100, mas se a compra foi igual a 1000 ou menor o desconto será 50.

Editor do .NET

```
1  int saleAmount = 1001;
2
3  //int discount = saleAmount > 1000 ? 100 : 50;
4  //Console.WriteLine($"Discount: {discount}");
5
6  Console.WriteLine($"Discount: {(saleAmount > 1000 ? 100 : 50)}");
```

Esse formato embutido retorna o mesmo resultado mas nem sempre é uma boa ideia combinar linhas de código se isso afeta negativamente a legibilidade geral da linha. Geralmente, essa é uma decisão subjetiva.

Desafio.

Nesse desafio, você lançará uma moeda para exibir heads ou tails.

Jogo do cara ou coroa.

Editor do .NET

```
1 Random jogarMoeda = new Random();
2 int moeda = jogarMoeda.Next(0, 2);
3
4 Console.WriteLine((moeda == 0) ? "heads" : "tails");
5 Console.WriteLine($"{(moeda == 0 ? "heads" : "tails" )}");
```

Saída

```
tails
tails
```

Desafio.

Neste desafio, você implementará regras de negócios que restringem o acesso a usuários com base nas respectivas permissões e nível. Você exibirá uma mensagem diferente para cada usuário, dependendo das respectivas permissões e nível.

- Se o usuário for um administrador com um nível maior que 55, exiba a mensagem: Welcome, Super Admin user.
- Se o usuário for um administrador com um nível menor ou igual a 55, exiba a mensagem: Welcome, Admin user.
- Se o usuário for um gerente com um nível maior ou igual a 20, exiba a mensagem: Contact an Admin for access.
- Se o usuário for um gerente com um nível menor que 20, exiba a mensagem: You do not have sufficient privileges.
- Se o usuário não for um administrador nem um gerente, exiba a mensagem: You do not have sufficient privileges.

```

8  string[] permission = {"Admin", "Manager"};
9  int level = 56;
10
11  string permissionAccess = permission[0];
12  //Console.WriteLine(permissionAccess);
13
14  /*
15  foreach(string permissionAccess in permission){
16      Console.WriteLine(permissionAccess);
17  }
18  */
19
20  if (permissionAccess.Contains("Admin"))
21  {
22      if (level > 55)
23      {
24          Console.WriteLine("Welcome, Super Admin user.");
25      }
26      else
27      {
28          Console.WriteLine("Welcome, Admin user.");
29      }
30  }
31  else if (permissionAccess.Contains("Manager"))
32  {
33      if (level >= 20)
34      {
35          Console.WriteLine("Contact an Admin for access.");
36      }
37      else
38      {
39          Console.WriteLine("You do not have sufficient privileges.");
40      }
41  }
42  else
43  {
44      Console.WriteLine("You do not have sufficient privileges.");
45  }

```

Saída

Welcome, Super Admin user.

Blocos de código e escopo de variável.

Um bloco de código é uma ou mais instruções do C# que definem um caminho de execução.

O escopo da variável é a visibilidade da variável para o outro código em seu aplicativo. Uma variável com escopo local só pode ser acessada dentro do bloco de código no qual ela está definida.

```
Editor do .NET ▶ Executar  
1 bool flag = true;  
2 if (flag)  
3 {  
4     int value = 10;  
5     Console.WriteLine($"Inside of code block: {value}");  
6 }
```

```
Saída  
  
Inside of code block: 10
```

Aqui acessamos a variável **value** definida dentro do bloco de código.

```
Editor do .NET ▶ Executar  
1 bool flag = true;  
2 if (flag)  
3 {  
4     int value = 10;  
5     Console.WriteLine($"Inside of code block: {value}");  
6 }  
7 Console.WriteLine($"Outside of code block: {value}");
```

```
Saída  
  
(7,45): error CS0103: The name 'value' does not exist in the current context
```

A variável não é acessível fora do bloco de código no qual está definida. Se tentarmos acessar fora do bloco de código retornará erro.

Para acessar uma variável tanto de um bloco de código externo quanto de um interno precisamos declarar a variável **value** fora do bloco de código.

Editor do .NET ▶ Executar

```
1  bool flag = true;
2  int value;
3
4  if (flag)
5  {
6      int value = 10;
7      Console.WriteLine($"Inside of code block: {value}");
8  }
9  Console.WriteLine($"Outside of code block: {value}");
```

Saída

```
(6,9): error CS0136: A local or parameter named 'value' cannot be declared in
this scope because that name is used in an enclosing local scope to define a
local or parameter
(9,45): error CS0165: Use of unassigned local variable 'value'
```

Mesmo declarando a variável fora do bloco de código obtemos erro. Isso porque precisamos inicializar a variável com algum valor.

Editor do .NET ▶ Ex

```
1  bool flag = true;
2  int value = 0;
3
4  if (flag)
5  {
6      value = 10;
7      Console.WriteLine($"Inside of code block: {value}");
8  }
9  Console.WriteLine($"Outside of code block: {value}");
```

Saída

```
Inside of code block: 10
Outside of code block: 10
```

Agora sim conseguimos compilar nosso código da forma correta acessando a variável dentro e fora do bloco de código.

- Quando você define uma variável dentro de um bloco de código, a visibilidade dela é local para esse bloco de código e inacessível fora desse bloco de código.
- Para tornar uma variável visível dentro e fora de um bloco de código, você deve definir a variável fora do bloco de código.
- Não se esqueça de inicializar nenhuma variável cujo valor seja definido em um bloco de código, assim como uma instrução if.

Estruturas de nível superior.

Nestes módulos, que apresentam unidades C# para iniciantes, usamos o Editor do .NET para compor e executar aplicativos.

No Editor do .NET baseado em navegador, todo o código que você escreve é executado dentro de um método oculto Main(), que esperamos que tenha diminuído bastante a complexidade de sua experiência inicial.

Para criar aplicativos reais, você escreverá métodos e os organizará em classes e namespaces.

```
C#  
  
using System;  
  
namespace MyNewApp  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

Aqui, você vê três níveis de blocos de código, começando do bloco de código mais interno e prosseguindo de dentro para fora:

- Um método Main()
- Uma classe Program
- Um namespace MyNewApp

Método: é um bloco de código que é uma unidade de execução. Em outras palavras, depois que o método for chamado pelo respectivo nome, o método inteiro será executado.

O nome do método Main() é especial. Quando o programa for executado, por padrão, o tempo de execução do .NET pesquisará por um método chamado Main() para usar como ponto de partida ou ponto de entrada para o programa.

Classe: é um contêiner para membros como métodos, propriedades, eventos, campos e assim por diante.

Namespace: realiza a desambiguação de nomes de classe. Há um número tão grande de classes na Biblioteca de Classes do .NET que é possível ter duas classes com o mesmo nome. O namespace garante que você possa instruir o compilador sobre com qual classe e método você deseja trabalhar, especificando também um namespace.

Você pode criar namespaces adicionais em seu código conforme necessário e criar uma série hierárquica de namespaces usando o operador ponto. Portanto, vamos supor que queríamos criar um segundo nível de namespaces para as classes de nosso aplicativo. Poderíamos adicionar um namespace filho como este:

```
3 namespace MyNewApp.Business
4 {
5     /*
6     No namespace MyNewApp.Business, esperamos adicionar classes
7     que implementem a lógica de negócios do nosso aplicativo.
8     */
9 }
10
11 namespace MyNewApp.Data
12 {
13     /*
14     No namespace MyNewApp.Data, esperamos adicionar classes
15     que implementem recursos de acesso a dados do nosso aplicativo.
16     */
17 }
```

Podemos adicionar quantos namespaces quisermos. Podemos criar namespaces com qualquer número de níveis de profundidade que for necessário.

Chamar um método na mesma classe.

```
1  using System;
2
3  namespace MyNewApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              string value = "Microsoft Learn";
10             string reversedValue = Reverse(value);
11             Console.WriteLine($"Secret message: {reversedValue}");
12         }
13
14         static string Reverse(string message)
15         {
16             char[] letters = message.ToCharArray();
17             Array.Reverse(letters);
18             return new string(letters);
19         }
20     }
21 }
```

Observe a linha 10. Já que o método **Reverse()** é definido na mesma classe, o código que chama o método não precisa qualificar o nome do método com o nome de classe.

Chamar um método de uma classe diferente

```
1  using System;
2
3  namespace MyNewApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              string value = "Microsoft Learn";
10             string reversedValue = Utility.Reverse(value);
11             Console.WriteLine($"Secret message: {reversedValue}");
12         }
13     }
14
15     class Utility
16     {
17         public static string Reverse(string message)
18         {
19             char[] letters = message.ToCharArray();
20             Array.Reverse(letters);
21             return new string(letters);
22         }
23     }
24 }
```

Aqui separamos o método **Reverse()** da classe **Program** na nova classe **Utility**, também adicionamos a palavra **public** ao método, caso contrário ele estaria inacessível para o método **Main()** da classe **Program**.

E no método **Main()** na linha 10 precisamos usar o nome da nova classe **Utility** ao acessar o método **Reverse()**.

Referenciar uma classe em um namespace diferente.

```
1  using System;
2
3  namespace MyNewApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              string value = "Microsoft Learn";
10             string reversedValue = MyNewApp.Utilities.Utility.Reverse(value);
11             Console.WriteLine($"Secret message: {reversedValue}");
12         }
13     }
14 }
15
16 namespace MyNewApp.Utilities
17 {
18     class Utility
19     {
20         public static string Reverse(string message)
21         {
22             char[] letters = message.ToCharArray();
23             Array.Reverse(letters);
24             return new string(letters);
25         }
26     }
27 }
```

1. Criamos um novo namespace **MyNewApp.Utilities**.
2. Movemos a classe **Utility** para o novo namespace.
3. Na linha 10 do método **Main()** adicionamos o nome do novo namespace ao chamar o método **Reverse()**.

A instrução **using** ajuda o compilador a resolver namespaces.

A instrução **using** é adicionada à parte superior de um arquivo de código. Ela resolve os nomes de classe que são usados no arquivo, instruindo o compilador a examinar a lista de namespaces para localizar todos os nomes de classe.

```
1  using System;
2  using MyNewApp.Utilities;
3
4  namespace MyNewApp
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             string value = "Microsoft Learn";
11             string reversedValue = Utility.Reverse(value);
12             Console.WriteLine($"Secret message: {reversedValue}");
13         }
14     }
15 }
16
17 namespace MyNewApp.Utilities
18 {
19     class Utility
20     {
21         public static string Reverse(string message)
22         {
23             char[] letters = message.ToCharArray();
24             Array.Reverse(letters);
25             return new string(letters);
26         }
27     }
28 }
29
```

A instrução **using** informa ao compilador para procurar aqui ao tentar resolver quaisquer nomes de classe que ele precise localizar. Agora, podemos chamar o método **Reverse()** usando apenas o nome de classe na linha 11 do método **Main()**.

Using System;

```
C#
using System;
```

Isso possibilita chamar **Console.WriteLine()** em vez de **System.Console.WriteLine()**.

Remover blocos de código em instruções if.

Se o bloco de código precisar de apenas uma linha de código, é provável que você não precise definir um bloco de código formal usando chaves, nem precise separar seu código em várias linhas.

Editor do .NET

```
1 bool flag = true;
2 if (flag)
3     Console.WriteLine(flag);
```

Saída

True

Como a instrução if e a chamada de método para Console.WriteLine() são curtas, poderíamos optar por combiná-las em uma única linha.

Editor do .NET

```
1 bool flag = true;
2 if (flag) Console.WriteLine(flag);
```

Apenas para demonstrar como usar as instruções else if e else sem blocos de código.

Editor do .NET

```
1 string name = "steve";
2 if (name == "bob") Console.WriteLine("Found Bob");
3 else if (name == "steve") Console.WriteLine("Found Steve");
4 else Console.WriteLine("Found Chuck");
```

Ou

Editor do .NET

```
1 string name = "steve";
2 if (name == "bob")
3     Console.WriteLine("Found Bob");
4 else if (name == "steve")
5     Console.WriteLine("Found Steve");
6 else
7     Console.WriteLine("Found Chuck");
```

Saída

Found Steve

Desafio.

Use o que você aprendeu neste módulo para corrigir este código mal escrito e que não executa. Há muitas melhorias que você pode fazer. Boa sorte!

```
C# Copiar

int[] numbers = { 4, 8, 15, 16, 23, 42 };
foreach (int number in numbers)
{
    int total;
    total += number;
    if (number == 42)
    {
        bool found = true;
    }
}
if (found)
{
    Console.WriteLine("Set contains 42");
}
Console.WriteLine($"Total: {total}");
```

Solução.

```
C# Copiar

int[] numbers = { 4, 8, 15, 16, 23, 42 };
int total = 0;
bool found = false;

foreach (int number in numbers)
{
    total += number;
    if (number == 42) found = true;
}

if (found) Console.WriteLine("Set contains 42");

Console.WriteLine($"Total: {total}");
```

```
Saída

Set contains 42
Total: 108
```

As maiores alterações no código problemático incluíam:

- Mover as variáveis total e found fora da instrução foreach.
- Inicializar as variáveis total e found com valores padrão razoáveis.
- Remover os blocos de código e os feeds de linha extras das instruções if.

Switch Case.

Switch é uma instrução de seleção que escolhe uma única seção de opção para executar com base em uma lista de candidatos com base em uma correspondência de padrões com a expressão de correspondência. Uma instrução switch inclui uma ou mais seções de opção.

- Use a instrução switch quando você tiver um valor com várias correspondências possíveis, cada uma delas exigindo um branch em sua lógica de código.
- Uma única seção de opção que contém a lógica de código que pode ser correspondida usando um ou mais rótulos definidos pela palavra-chave case.
- Use a palavra-chave default opcional para criar um rótulo e uma seção de opção para serem usados quando nenhum outro rótulo de caso corresponder.

```
1  /* switc case */
2
3  int employeeLevel = 200;
4  string employeeName = "John Smith";
5
6  string title = "";
7
8  switch (employeeLevel)
9  {
10     case 100:
11         title = "Junior Associate";
12         break;
13     case 200:
14         title = "Senior Aassociate";
15         break;
16     case 300:
17         title = "Manager";
18         break;
19     case 400:
20         title = "Senior Manager";
21         break;
22     default:
23         title = "Aassociate";
24         break;
25 }
26
27 Console.WriteLine($"{employeeName}, {title}");
```

Safda

John Smith, Senior Associate


1. A palavra-chave `switch` define a finalidade do bloco de código. Ao lado da palavra-chave, a expressão que será comparada entre parênteses (`employeeLevel`).
2. Dentro do bloco de código, uma ou mais *seções de opção*. Cada seção de opção tem um ou mais rótulos. Um rótulo começa com a palavra-chave `case`.
3. Depois que o runtime encontrar um rótulo correspondente, ele executará o código nessa seção de opção específica.

break: é uma de várias maneiras de encerrar uma seção de opção e literalmente interromper a instrução de opção.

default: Se não houver rótulos correspondentes, essa opção será executada.

Para permitir que dois rótulos executem a mesma seção de opção utilizamos a seguinte sintaxe.

C#

 Copiar

```
case 100:  
case 200:  
    title = "Senior Associate";  
    break;
```

Agora os rótulos de valor 100 e 200 recebem a mesma instrução.

Desafio.

Suponha que trabalhamos para uma loja de lembranças em uma cidade universitária que vende camisetas, moletons e outros presentes com o logotipo e as cores da instituição. Um relatório mensal de vendas usa a descrição completa, assim como a SKU (Unidade de Manutenção de Estoque) dos produtos vendidos. Pediram que reescrevêssemos determinadas partes do código para ficarem mais legíveis. Uma das tarefas é simplificar a conversão de um SKU em uma descrição usando a instrução switch.

O código a seguir converte um SKU em uma descrição de formato longo (por exemplo, o SKU **01-MN-L** será **Large Maroon Sweat shirt**).

```
1  /* seletor de roupa - com switch case */
2
3  string sku = "01-MN-L";
4
5  string[] product = sku.Split('-');
6
7  string type = "";
8  string color = "";
9  string size = "";
10
11  switch (product[0]){
12      case "01":
13          type = "Sweat shirt";
14          break;
15      case "02":
16          type = "T-Shirt";
17          break;
18      case "03":
19          type = "Sweat pants";
20          break;
21      default:
22          type = "Other";
23          break;
24  }
25
26  switch (product[1]){
27      case "BL":
28          color = "Black";
29          break;
30      case "MN":
31          color = "Maroon";
32          break;
33      default:
34          color = "White";
35          break;
36  }
```

```

37
38     switch (product[2]){
39         case "S":
40             size = "Small";
41             break;
42         case "M":
43             size = "Medium";
44             break;
45         case "L":
46             size = "Large";
47             break;
48         default:
49             size = "One Size Fits ALL";
50             break;
51     }
52
53     Console.WriteLine($"Product: {size} {color} {type}");

```

Saída

Product: Large Maroon Sweat shirt

Usamos o switch-case como uma substituição do constructo if-elseif-else para expressar sucintamente nossa intenção de converter um SKU (Unidade de Manutenção de Estoque) em uma descrição de formato longo.

A instrução da decisão if-elseif-else, a instrução da decisão switch-case e o operador condicional fornecem três maneiras de ramificar nossa lógica de código. Agora você sabe como escolher a melhor ferramenta do C# para expressar sua intenção em seu código.

Instrução de iteração for.

A instrução **for** itera por meio de um bloco de código um número específico de vezes. Isso torna a instrução **for** exclusiva entre as outras instruções de iteração. A instrução **foreach** itera por meio de um bloco de código uma vez para cada item em uma sequência de dados como uma matriz ou coleção. A instrução **while** itera por meio de um bloco de código até que uma condição seja atendida.

Além disso, a instrução **for** oferece muito mais controle sobre o processo de iteração expondo as condições de iteração.

C# Copiar Exec

```
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine(i);  
}
```

Saída Copiar

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Há seis partes para a instrução **for**.

1. A palavra-chave **for** indica um loop.
2. Um conjunto de parênteses que define as condições da iteração for. Ele contém três partes distintas, separadas por ponto e vírgula (**inicializador ; condição ; iterador**).
3. **Inicializador**: inicializa a variável de iterador. Neste exemplo: `int i = 0`.
4. **Condição**: continuará iterando sobre o código no bloco de código embaixo da instrução **for**, enquanto **i** for menor que **10**.
5. **Iteração**: após cada iteração, **i++** incrementará o valor de **i** em **1**.
6. Finalmente, o bloco de código. Esse é o código que será executado para cada iteração.

A instrução for deve ser usada quando você sabe o número de vezes que precisa iterar por meio de um bloco de código antes do tempo.

```
C# Copiar Executar

for (int i = 10; i >= 0; i--)
{
    Console.WriteLine(i);
}
```

```
Saída

10
9
8
7
6
5
4
3
2
1
0
```

E se desejamos contar regressivamente em vez de progressivamente?

A instrução **for** permite que você controle a maneira como cada iteração é manipulada.

1. Inicializamos a variável de iteração como 10.
2. Alteramos a condição de conclusão para sair da instrução **for** quando **i** for menor que **0**.
3. Alteramos o padrão do iterador para subtrair **1** de **i** toda vez que concluirmos uma iteração.

```
Editor do .NET

1 for (int i = 0; i < 10; i += 3){
2     Console.WriteLine(i);
3 }
```

```
Saída

0
3
6
9
```

Outro exemplo de iteração com **for**. Onde a cada iteração será somado 3 ao inicializador.

Usar a palavra-chave break para causar um curto-circuito na instrução de iteração.

E se for necessário sair da instrução de iteração prematuramente com base em alguma condição? Podemos usar a palavra-chave break.

C# Copiar Executar

```
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine(i);  
    if (i == 7) break;  
}
```

Saída

```
0  
1  
2  
3  
4  
5  
6  
7
```

Percorrer cada elemento de uma matriz.

Um uso comum da instrução `for` é iterar por meio de uma matriz de elementos, principalmente se você precisar ter algum controle sobre a maneira como ocorre a iteração. Embora o **foreach** itere por meio de cada elemento da matriz, a instrução **for** pode ser ajustada para fornecer mais personalização.

Editor do .NET

```
1 string[] names = {"Alex", "Eddie", "David", "Michael"};  
2  
3 for (int i = names.Length - 1; i >= 0; i--){  
4     Console.WriteLine(names[i]);  
5 }
```

Saída

```
Michael  
David  
Eddie  
Alex
```


Nesse caso, iteramos por meio da matriz para trás, algo que não seria possível fazer com a instrução **foreach**. Usamos a propriedade **Length** para inicializar a variável de iterador e subtraímos uma de **i** com cada iteração. Dentro do bloco de código, indexamos na matriz usando a variável de iteração.

Limitação da instrução foreach.

```
C# Copiar

string[] names = { "Alex", "Eddie", "David", "Michael" };
foreach (var name in names)
{
    // Can't do this:
    if (name == "David") name = "Sammy";
}
```

Se tentar executar esse código receberá um erro. Em outras palavras, não é possível reatribuir o valor de **name** porque ele faz parte da implementação interna da iteração **foreach**.

```
Editor do .NET

1  string[] names = {"Alex", "Eddie", "David", "Michael"};
2
3  for (int i = 0; i < names.Length; i++){
4  |      if (names[i] == "David") names[i] = "Sammy";
5  |  }
6  foreach (var name in names) Console.WriteLine(name);

Saída

Alex
Eddie
Sammy
Michael
```

No entanto, é possível realizar a mesma funcionalidade usando a instrução **for**.

Como a matriz não faz parte diretamente da implementação da instrução de iteração, é possível alterar os valores dentro da matriz.

Desafio FizzBuzz.

Regras do FizzBuzz:

1. Valores de saída de 1 a 100, um número por linha.
2. Quando o valor atual é divisível por 3, imprima o termo **Fizz** ao lado do número.
3. Quando o valor atual é divisível por 5, imprima o termo **Buzz** ao lado do número.
4. Quando o valor atual é divisível tanto por 3 quanto por 5, imprima o termo **FizzBuzz** ao lado do número.

```
2
3  string fizz = "Fizz";
4  string buzz = "Buzz";
5  string fizzBuzz = "FizzBuzz";
6
7
8  for (int i = 1; i <= 100; i++){
9
10     if ((i % 3 == 0) && (i % 5 == 0)){
11         Console.WriteLine($"{i} - {fizzBuzz}");
12     }
13     else if (i % 3 == 0){
14         Console.WriteLine($"{i} - {fizz}");
15     }
16     else if (i % 5 == 0){
17         Console.WriteLine($"{i} - {buzz}");
18     }
19     else{
20         Console.WriteLine(i);
21     }
22
23 }
24
```

Saída

```
1
2
3 - Fizz
4
5 - BUZZ
6 - Fizz
7
8
9 - Fizz
10 - BUZZ
11
12 - Fizz
13
14
15 - FizzBuzz
16
17
18 - Fizz
19
20 - BUZZ
21 - Fizz
22
.
.
.
```

1. A instrução `for` é importante porque permite que você itere o bloco de código 100 vezes.
2. O `if-elseif-else` permite que você verifique se há divisores de 3 e de 5.
3. O `%`, o operador mod, permite que você determine se 3 ou 5 são divididos em outro número sem resto.
4. E o operador `&&` verifica se um número pode ser dividido em 3 e 5 para a condição FizzBuzz.

Instruções "do-while", "while" e "continue"

As instruções **do-while** e **while** permitem controlar o fluxo de execução de código fazendo um loop por meio de um bloco de código até que uma condição seja atendida. Ao trabalhar com a instrução **foreach**, iteramos uma vez para cada item em sequência, como uma matriz. A instrução **for** nos permite iterar um número predeterminado de vezes e controlar o processo de iteração. As instruções **do-while** e **while** permitem iterar por meio de um bloco de código com a intenção que a lógica dentro do bloco de código afetará quando for possível parar a iteração.

O que é a instrução do-while?

```
C#Copiar  
  
do  
{  
    // This code executes at least one time  
} while (true)
```

O código é executado pelo menos uma vez e, em seguida, a expressão booliana ao lado da palavra-chave **while** é avaliada. Se a expressão booliana retornar **true**, o bloco de código será executado novamente.

Editor do .NET

```
1 Random random = new Random();
2 int current = 0;
3
4 do
5 {
6     current = random.Next(1, 11);
7     Console.WriteLine(current);
8 }while (current != 7);
```

Saída

```
3
3
7
```

O código é executado pelo menos uma vez, e depois a expressão é avaliada. Se o número for diferente de 7 o loop continua, quando a condição se torna **false** o loop encerra.

A instrução while.

Editor do .NET

```
1 Random random = new Random();
2 int current = random.Next(1, 11);
3
4 while (current >= 3)
5 {
6     Console.WriteLine(current);
7     current = random.Next(1, 11);
8 }
9 Console.WriteLine($"Last number: {current}");
```

Saída

```
4
9
Last number: 1
```

No loop **while** a expressão é avaliada antes de entrar no bloco de código se for **false** o bloco de código não será executado, mas se for **true** a cada execução a expressão será avaliada novamente até que a expressão seja **false** e saia do loop.

A palavra continue.

```
Editor do .NET
1 Random random = new Random();
2 int current = random.Next(1, 11);
3
4 do
5 {
6     current = random.Next(1, 11);
7
8     if (current >= 8) continue;
9
10    Console.WriteLine(current);
11 } while (current != 7);
```

Saída

```
6
5
7
```

A chave para esta etapa do exercício é a seguinte linha de código 8.

Ao executar o código você não verá nenhum valor 8 ou maior na janela de saída antes que a execução do código termine com o valor 7.

Como você pode ver, continue ignorará a execução da iteração atual para que nada maior que 7 seja impresso.

Desafio.

Desafio de batalha em RPG.

Um herói e um monstro começam com a mesma pontuação de integridade. Durante a rodada do herói, ele gerará um valor aleatório que será subtraído da integridade do monstro. Se a integridade do monstro for maior que zero, ela terá sua rodada e atacará o herói. Desde que tanto o herói quanto o monstro tenham integridade maior que zero, a batalha continuará.

Regras do Jogo:

1. O herói e o monstro começarão com dez pontos de integridade.
2. Todos os ataques serão um valor entre 1 e 10.
3. O herói atacará primeiro.
4. Imprima a quantidade de integridade que o monstro perdeu e a integridade que resta a ele.
5. Se a integridade do monstro for maior que zero, ele poderá atacar o herói.
6. Imprima a quantidade de integridade que o herói perdeu e a integridade que resta a ele.

7. Continue esta sequência de ataque até que a integridade do monstro ou do herói seja zero ou menos.
8. Imprima quem foi o vencedor.

```
22 int hero = 10;
23 int monster = 10;
24
25 Random dice = new Random();
26
27 do
28 {
29     int roll = dice.Next(1, 11);
30     monster -= roll;
31     Console.WriteLine($"Monster was damaged and lost {roll} health and now has {monster} health.");
32
33     if (monster <= 0) continue;
34
35     roll = dice.Next(1, 11);
36     hero -= roll;
37     Console.WriteLine($"Hero was damaged and lost {roll} health and now has {hero} health.");
38 } while(hero > 0 && monster > 0);
39
40
41 Console.WriteLine(hero > monster ? "Hero wins!" : "Monster wins!");
```

Saída

```
Monster was damaged and lost 4 health and now has 6 health.
Hero was damaged and lost 3 health and now has 7 health.
Monster was damaged and lost 4 health and now has 2 health.
Hero was damaged and lost 5 health and now has 2 health.
Monster was damaged and lost 6 health and now has -4 health.
Hero wins!
```

Nossa meta era usar as instruções **do-while** e **while** para executar uma iteração. Conforme aprendemos, o que faz com que as instruções **do** e **while** sejam exclusivas é o modo como o corpo do bloco de código determina se o fluxo de execução deve continuar ou parar.

Usando a instrução **do-while**, executamos um bloco de código uma vez antes de avaliar uma expressão booliana e, possivelmente, sair da iteração. Usando a instrução **while**, realizamos a avaliação da expressão booliana imediatamente e continuamos a avaliá-la para sair da iteração. Usamos a instrução **continue** para passar diretamente para a expressão booliana.