Introdução ao C#

1. História do C#

A linguagem é fortemente tipada, orientada a objeto e orientada a componentes e utiliza um sistema de tipo unificado. Diferentemente de C ou C++, o C# lida com gerenciamento de memória e recursos para o desenvolvedor, emprestando o conceito de código gerenciado.

O C# usa um mecanismo de coleta de lixo para liberar memória e recursos que não são mais mencionados no código do aplicativo, ajudando a evitar problemas de vazamento de memória.

Alguns recursos do C# são:

- Fortemente tipado: os idiomas reforçam a verificação de tipo nos objetos no código, o que significa que é seguro para o tipo;

- Orientado a objetos: o C# oferece ao desenvolvedor todos os princípios do OOP, como encapsulamento, herança e polimorfismo;

- Orientado a componentes: C# permite a criação de componentes de software para pacotes de funcionalidade independentes e autoexplicativos;

- Sistema de tipos unificado: todos os tipos de C#, dos primitivos aos tipos de referência, herdam de uma única raiz conhecida como Objeto;

2. Tipos de dados

Todos os aplicativos armazenam e manipulam dados na memória do computador. O C# suporta dois tipos de dados usados para representar informações do mundo real, tipos de valor e tipos de referência.

Os tipos de valor são denominados porque contêm o valor real dos dados que armazenam. Por exemplo, você pode ter um tipo *int* que armazena o valor 3. O valor literal de 3 é armazenado na variável que você declara mantê-lo.

Os tipos de referência também são conhecidos como objetos. Os tipos de referência são criados a partir de arquivos de classe, abordados no curso Programação Orientada a Objetos em C#.

Um tipo de referência armazena uma referência ao local na memória do objeto. Se você estiver familiarizado com C / C++, poderá pensar em uma referência ao local da memória como igual a um ponteiro. C# não requer o uso de ponteiros.

Type	Description	Size (bytes)	.NET Type	Range
int	Whole numbers	4	System.Int32	-2,147,483,648 to 2,147,483,647
long	Whole numbers (bigger range)	8	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Floating-point numbers	4	System.Single	+/-3.4 x 10^38
double	Double precision (more accurate) floating-point numbers	8	System.Double	+/-1.7 x 10^308
decimal	Monetary values	16	System.Decimal	28 significant figures
char	Single character	2	System.Char	N/A
bool	Boolean	1	System.Boolean	True or false
DateTime	Moments in time	8	System.DateTime	0:00:00 on 01/01/0001 to 23:59:59 on 12/31/9999
string	Sequence of characters	2 per character	System.String	N/A

3. Identificadores

Em C#, um identificador é um nome que você atribui aos elementos do seu programa. Os elementos do seu programa incluem;

- Namespaces: O .NET Framework usa namespaces como uma maneira de separar arquivos de classe em buckets ou categorias relacionados. Também ajuda a evitar colisões de nomes em aplicativos que podem conter classes com o mesmo nome;

- Classes: Classes são os projetos para tipos de referência. Eles especificam a estrutura que um objeto terá quando você criar instâncias da classe;

- Métodos: Abordados um pouco mais adiante, são peças discretas de funcionalidade em um aplicativo. Eles são análogos a funções no mundo da programação não orientada a objetos;

- Variáveis: São identificadores ou nomes que você cria para manter valores ou referências a objetos no seu código. Uma variável é essencialmente um local de memória nomeado;

4. Operadores

Type	Operators
Arithmetic	+, -, *, /, %
Increment, decrement	++, --
Comparison	==, !=, <, >, <=, >=, is
String concatenation	+
Logical/bitwise operations	&, |, ^, !, ~, &&, ||
Indexing (counting starts from element 0)	[ ]
Casting	( ), as
Assignment	=, +=, -=, =, /=, %=, &=, |=, ^=, <<=, >>=, ??
Bit shift	<<, >>
Type information	sizeof, typeof
Delegate concatenation and removal	+, -
Overflow exception control	checked, unchecked
Indirection and Address (unsafe code only)	*, ->, [ ], &
Conditional (ternary operator)	?:

5. Conversão de tipos de dados

O C# suporta dois tipos inerentes de conversão para tipos de dados, implícitos e explícitos. O C# usará a conversão implícita onde puder, principalmente no caso em que uma conversão não resultará em perda de dados ou quando a conversão for possível com um tipo de dado compatível.

```
int myInt = 2147483647;
long myLong= myInt;
```

O tipo *long* tem um tamanho de 64 bits na memória enquanto o tipo *int* usa 32 bits. Portanto, o *long* pode acomodar facilmente qualquer valor armazenado no tipo *int*. Ir de um *long* para um *int* pode resultar em perda de dados e você deve usar a conversão explícita para isso, se souber quais dados serão perdidos e isso não afetará seu código.

As conversões explícitas são realizadas de uma das duas maneiras, conforme demonstrado no seguinte exemplo de código.

```
double myDouble = 1234.6;
int myInt;
// Cast double to int by placing the type modifier ahead of the type to be converted
// in parentheses
myInt = (int)myDouble;
```

A segunda opção é usar os métodos fornecidos no .NET Framework.

```
double myDouble = 1234.6;
int myInt;
// Cast double to int by using the Convert class and the ToInt32() method.
// This converts the double value to a 32-bit signed integer
myInt = Convert.ToInt32(myDouble);
```

O C# também fornece outro mecanismo para lidar com tipos de conversão. O uso dos métodos *TryParse()* e *Parse()* também pode ajudar na conversão. Esses métodos são anexados aos tipos em C# em vez da classe Convert. Um exemplo ajudará a demonstrar.

```
// TryParse() example
bool result = Int32.TryParse(value, out number);

// Parse() example
int number = Int32.Parse(value);
```

No exemplo *TryParse()*, o método retorna um resultado booleano indicando se a conversão foi bem-sucedida. No exemplo *Parse()*, se a conversão não for bem-sucedida, uma exceção será lançada.

6. Estrutura de decisão

As estruturas de decisão em C# fornecem lógica no código do aplicativo que permite a execução de diferentes seções do código, dependendo do estado dos dados no aplicativo.

A instrução condicional primária no Visual C# é a instrução *if*. Uma alternativa para a instrução *if* é uma instrução *switch*. Como você verá na seção sobre a instrução *switch*, você pode usá-lo para decisões mais complexas.

6.1. If

Em C#, as instruções *if* estão relacionadas a lógica booleana. Se a instrução for verdadeira, o bloco de código associado à instrução *if* será executado. Se a instrução for falsa, o controle cairá para a linha após a instrução *if* ou após o fechamento de um bloco de instrução *if*.

Em C#, as instruções *if* também podem ter cláusulas *else* associadas. A cláusula *else* é executada quando a instrução *if* é falsa.

O exemplo de código a seguir mostra como usar um *if else* para executar código quando uma condição é falsa.

```
//if else Statements

string response;
if (response == "connection_failed")
{
    // Block of code executes if the value of the response variable is "connection_failed".
}
else
{
    // Block of code executes if the value of the response variable is not "connection_failed".
}
```

As instruções *if* também podem ter cláusulas *else if* associadas. As cláusulas são testadas na ordem em que aparecem no código após a declaração *if*. Se alguma das cláusulas retornar verdadeira, o bloco de código associado a essa instrução será executado e o controle deixará o bloco de código associado a toda a instrução *if else*.

O exemplo de código a seguir mostra como usar um se declaração com uma cláusula *else if*.

```
//else if Statements

string response;
if (response == "connection_failed")
{
    // Block of code executes if the value of the response variable is "connection_failed".
}
else if (response == "connection_error")
{
    // Block of code executes if the value of the response variable is "connection_error".
}
else
{
    // Block of code executes if the value of the response variable is neither above responses.
}
```

6.2. Switch

Se houver muitas outras declarações *if*, o código pode ficar confuso e difícil de seguir. Nesse cenário, uma solução melhor é usar uma instrução *switch*. A instrução *switch* simplesmente substitui várias instruções *if*.

O exemplo a seguir mostra como você pode usar uma instrução *switch* para substituir uma coleção de cláusulas *else if*.

```
//switch Statement

string response;
switch (response)
{
   case "connection_failed":
      // Block of code executes if the value of response is "connection_failed".
      break;
   case "connection_success":
      // Block of code executes if the value of response is "connection_success".
      break;
   case "connection_error":
      // Block of code executes if the value of response is "connection_error".
      break;
   default:
      // Block executes if none of the above conditions are met.
      break;
}
```

Observe que há um bloco rotulado default:. Este bloco de código será executado quando nenhum dos outros blocos corresponder.

7. Estruturas de repetição

Repetição é essencialmente o conceito de fazer algo de maneira repetitiva. Na programação, você normalmente usa a repetição para iterar nos itens de uma coleção ou para executar a mesma tarefa repetidamente para produzir o efeito desejado em seu programa.

7.1. For

o ciclo *for* executa um bloco de código repetidamente até que a expressão especificada seja avaliada como falsa. Você pode definir um *loop for* da seguinte maneira.

```
for ([initializers]; [condition]; [iterator])
{
   // code to repeat goes here
}
```

A parte [*initializers*] é usada para inicializar um valor como um contador para o *loop*. Em cada iteração, o *loop* verifica se o valor do contador está dentro do intervalo para executar o *loop for*, especificado na parte [*condition*]. E, em caso afirmativo, execute o corpo do *loop*. No final de cada iteração de *loop*, a seção [*iterator*] é responsável por incrementar o contador de *loop*.

7.2. Foreach

Embora um *loop for* seja fácil de usar, ele pode apresentar alguns desafios, dependendo da situação. Como exemplo, considere iterar sobre uma coleção ou uma matriz de valores. Você precisaria saber quantos elementos há na coleção ou matriz. Em muitos casos você saberá disso, mas às vezes pode ter coleções ou matrizes dinâmicas e não dimensionadas em tempo de compilação. Se o tamanho da coleção ou matriz mudar durante o tempo de execução, talvez seja uma opção melhor usar um *foreach*.

O exemplo de código a seguir mostra como usar um *foreach* para iterar uma matriz de string.

```
//foreach Loop

string[] names = new string[10];

// Process each name in the array.
foreach (string name in names)
{
    // Code to execute.
}
```

O C# lida com a determinação de quantos itens há na matriz e parará de executar o *loop* quando o final for alcançado. O uso do *foreach* pode ajudar a evitar erros de índice fora dos limites nas matrizes.

7.3. While

Um *loop while* permite executar um bloco de código enquanto uma determinada condição é verdadeira. Por exemplo, você pode usar um *loop while* para processar a entrada do usuário até que o usuário indique que não há mais dados para inserir. O *loop* pode continuar solicitando ao usuário até que ele decida encerrar a interação digitando um valor de sentinela. O valor da sentinela é responsável por finalizar o *loop*.

O exemplo de código a seguir mostra como usar um *loop while*.

```
//while Loop

string response = PromptUser();
while (response != "Quit")
{
    // Process the data.
    response = PromptUser();
}
```

7.4. Do

Um *loop do* às vezes também chamado de *do...while loop*, é muito semelhante a um *while loop*, com a exceção de que um *do loop* sempre executará o corpo do *loop* pelo menos uma vez. Em um *while loop*, se a condição for falsa desde o início, o corpo do *loop* nunca será executado.

Você pode usar um *loop* se souber que o código será executado apenas em resposta a um prompt de dados do usuário. Nesse cenário, você sabe que o aplicativo precisará processar pelo menos um dado e, portanto, pode usar um *loop do*.

O exemplo de código a seguir mostra o uso de um *loop do*.

```
//do Loop

do
{
    // Process the data.
    response = PromptUser();
} while (response != "Quit");
```

8. Métodos

Métodos são um conceito fundamental dentro de um programa C# e, na verdade, são fundamentais para a programação orientada a objetos, porque nos permitem encapsular comportamento e funcionalidade dentro dos objetos que criamos em nosso código.

8.1. Declarações de métodos

A capacidade de definir e chamar métodos é um componente fundamental da programação orientada à objetos, porque os métodos permitem encapsular operações que protegem os dados armazenados em um tipo.

Os métodos podem ser projetados para uso interno por um tipo e, como tal, estão ocultos de outros tipos. Os métodos públicos podem ser projetados para permitir que outros tipos solicitem que um objeto execute uma ação e sejam expostos fora do tipo.

Um método é declarado usando uma assinatura e um corpo de método. A parte da assinatura é responsável por fornecer o modificador de acesso, o tipo de retorno do método, o nome do método e a lista de parâmetros. O corpo contém a implementação para o que o método se destina a fazer. Cada componente de assinatura do método é explicado aqui:

- Modificador de acesso: usado para controlar a acessibilidade do método (de onde ele pode ser chamado);
  - *Private:* mais restritivo e permite acesso ao método somente de dentro da classe ou estrutura que o contém;
  - *Public:* menos restritivo, permitindo acesso a partir de qualquer código no aplicativo;
  - *Protected:* permite acesso de dentro da classe que contém ou de dentro de classes derivadas;
  - *Internal:* acessível a partir de arquivos no mesmo conjunto;
  - *Static:* indica que o método é um membro estático da classe e não um membro de uma instância de um objeto específico;
- Tipo de retorno: usado para indicar qual tipo o método retornará. Use *void* se o método não retornar um valor ou qualquer tipo de dado suportado;
- Nome do método: todos os métodos precisam de um nome para que você saiba o que chamar no código. As regras de identificador também se aplicam aos nomes dos métodos;
- Lista de parâmetros: uma lista de parâmetros separados por vírgula para aceitar argumentos passados para o método;

```
public Boolean StartService(string serviceName)
{
   // code to start the service
}
```

8.2. Chamada de métodos

Para chamar um método, especifique o nome do método e forneça quaisquer argumentos que correspondam aos parâmetros do método entre parênteses.

O exemplo de código a seguir mostra como chamar o método *StartService*, passando variáveis *int* e booleanas para atender aos requisitos de parâmetro da assinatura do método.

```
int upTime = 2000;
bool shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.
void StartService(int upTime, bool shutdownAutomatically)
{
   // Perform some processing here.
}
```

8.3. Retornando dados de métodos

Se o método retornar um valor, você especificará como lidar com esse valor, normalmente atribuindo-o a uma variável do mesmo tipo, no seu código de chamada.

Pode haver momentos em que você prefira retornar vários valores de um método. Existem três abordagens que você pode adotar para fazer isso:

- Retornar uma matriz ou coleção
- Use a palavra-chave *ref*
- Use a palavra-chave *out*

Neste primeiro exemplo de código, é feita uma chamada para o método *ReturnMultiOut*. Os parâmetros para este método usam a palavra chave *out* para indicar que os valores serão retornados para esses parâmetros. Observe que não precisamos chamar esse método com uma instrução de atribuição.

```
ReturnMultiOut(out first, out sValue);
Console.WriteLine($"{first.ToString}, {sValue}");

static void ReturnMultiOut(out int i, out string s)
{
    i = 25;
    s = "using out";
}
```

Neste novo exemplo de código, o palavra chave *ref* é usada para retornar vários valores do método. Tipicamente a palavra chave *ref* requer que as variáveis usadas sejam inicializadas primeiro.

```
// Using ref requires that the variables be initialized first
string sValue = "";
int first = 0;
ReturnMultiRef(ref first, ref sValue);
Console.WriteLine($"{first.ToString()}, {sValue}");

 void ReturnMultiRef(ref int i, ref string s)
 {
        i = 50;
        s = "using ref";
 }
```

8.4. Sobrecarga de métodos

Ao definir um método, você pode perceber que ele requer conjuntos diferentes de informações em diferentes circunstâncias. Você pode definir métodos sobrecarregados para criar vários métodos com a mesma funcionalidade que aceitam parâmetros diferentes, dependendo do contexto em que são chamados.

Métodos sobrecarregados têm o mesmo nome um do outro para enfatizar sua intenção comum. No entanto, cada método sobrecarregado deve ter uma assinatura exclusiva, para diferenciá-lo das outras versões sobrecarregadas do método na classe.

A assinatura de um método inclui seu nome e sua lista de parâmetros. O tipo de retorno não faz parte da assinatura. Portanto, você não pode definir métodos sobrecarregados que diferem apenas no tipo de retorno. Você também não pode definir métodos sobrecarregados que diferem em posição dos parâmetros.

O exemplo de código a seguir mostra três versões do método *StopService*, todas com uma assinatura exclusiva.

```
void StopService()
{
   // This method accepts no arguments
}

void StopService(string serviceName)
{
   // This method overload accepts a single string argument
}

void StopService(int serviceId)
{
   // This method overload accepts a single integer argument
}
```

Quando você invoca o método *StopService*, você tem escolha de qual versão sobrecarregada você usa. Você simplesmente fornece os argumentos relevantes para satisfazer uma sobrecarga específica e, em seguida, o compilador descobre qual versão chamar, com base nos argumentos que você passou.

Como observado acima, você não pode criar um método sobrecarregado usando apenas a posição dos argumentos nem o tipo de retorno. O código abaixo mostra um exemplo de tentativas incorretas de sobrecarregar métodos.

```
void StopService(string serviceName, int seconds)
{
   // This method overload accepts a string argument
   // and an integer argument
}

void StopService(int seconds, string serviceName)
{
   // This method overload accepts an integer argument
   // and a string argument.  It appears different than
   // the preceding method because the arguments are
   // passed in differently, however, it is still not
   // a valid overload
}

bool StopService(string serviceName, int seconds)
{
   // This method overload accepts a string argument
   // and an integer argument but returns a boolean
   // result.  The difference between this method and
   // the others is the return type.  This is still not
   // a valid overload based on overloading rules.
}
```

8.5. Parâmetros opcionais e nomeados

Parâmetros opcionais também são úteis em outras situações. Eles fornecem uma solução compacta e simples quando não é possível usar sobrecarga porque os tipos de parâmetros não variam o suficiente para permitir que o compilador faça a distinção entre implementações. Ao definir métodos que usarão parâmetros opcionais, é importante observar que você deve especificar todos os parâmetros não opcionais primeiro e depois listar os parâmetros opcionais.

O exemplo de código a seguir mostra como definir um método que aceita um parâmetro obrigatório (*forceStop*) e dois parâmetros opcionais (*serviceName*, *serviceID*). Observe que o mecanismo usado para indicar um parâmetro opcional é a inclusão de um valor padrão.

```
void StopService(bool forceStop, string serviceName = null, int serviceId =1)
{
   // code here that will stop the service
}
```

Tradicionalmente, ao chamar um método, a ordem e posição dos argumentos na chamada de método corresponde para a ordem dos parâmetros na assinatura do método. Se os argumentos estiverem desalinhados e os tipos incompatíveis, você receberá um erro de compilação.

No Visual C#, você pode especificar parâmetros por nome e, portanto, fornecer argumentos em uma sequência diferente da definida na assinatura do método. Para usar argumentos nomeados, forneça o nome do parâmetro e o valor correspondente separados por dois pontos.

O exemplo de código a seguir mostra como chamar o método *StopService* usando argumentos nomeados para passar o parâmetro *serviceID*.

StopService(true, serviceID: 1);

Ao usar argumentos nomeados em conjunto com parâmetros opcionais, você pode omitir facilmente parâmetros. Quaisquer parâmetros opcionais receberão seu valor padrão. No entanto, se você omitir parâmetros obrigatórios, seu código não será compilado.

Você pode misturar argumentos posicionais e nomeados. No entanto, você deve especificar todos os argumentos posicionais antes de qualquer argumento nomeado.

9. Tratamento de exceções

Uma exceção é uma indicação de um erro ou condição excepcional. Um método pode gerar uma exceção quando detecta que algo inesperado aconteceu, por exemplo, o aplicativo tenta abrir um arquivo que não existe.

9.1. Propagação da exceção

Quando um método lança uma exceção, o código de chamada deve estar preparado para detectar e manipular essa exceção. Se o código de chamada não detectar a exceção, o código será abortado e a exceção será propagada automaticamente para o código que chamou o código de chamada. Esse processo continua até que uma seção do código assuma a responsabilidade pelo tratamento da exceção. A execução continua nesta seção de código após a conclusão da lógica de tratamento de exceções. Se nenhum código manipular a exceção, o processo será encerrado e exibirá uma mensagem para o usuário.

Às vezes, isso também é chamado de passagem da exceção na pilha ou na pilha de chamada.

9.2. Tratando exceções

O bloco *try / catch* é a construção de programação principal que permite implementar o tratamento estruturado de exceções em seus aplicativos escritos em C#.

Você quebra o código que pode falhar e causa uma exceção em um bloco *try* e adiciona um ou mais blocos *catch* para manipular quaisquer exceções que possam ocorrer. A estratégia recomendada a seguir com os blocos de captura é capturar primeiro exceções mais específicas e, depois, exceções mais genéricas.

Por exemplo, se você espera encontrar uma exceção em torno do acesso a arquivos, capturará o *FileNotFoundException* no primeiro bloco de captura e, em seguida, talvez crie um segundo bloco de captura que observaria a classe *Exception* genérica capturar qualquer outra exceção além da *FileNotFoundException*.

O exemplo de código a seguir mostra a sintaxe para definir um bloco *try / catch*.

```
try
{
    // Try block.
}
catch (FileNotFoundException fnfEx)
{
    // Catch block 1.
}
catch (Exception e)
{
    // Catch block n.
}
```

9.3. Bloco finally

Alguns métodos podem conter código crítico que sempre deve ser executado, mesmo se ocorrer uma exceção não tratada. Por exemplo, um método pode precisar garantir que ele feche um arquivo no qual estava gravando ou libere alguns outros recursos antes de terminar. Um bloco *finally* permite que você lide com essa situação.

Você especifica um *finally* após qualquer manipulador de captura em um bloco *try / catch*. Ele especifica o código que deve ser executado quando o bloco terminar, independentemente de

ocorrer alguma exceção, manipulada ou não. Se uma exceção for capturada e manipulada, o manipulador de exceções no bloco *catch* será executado antes do bloco *finally*.

O exemplo de código a seguir mostra como implementar um bloco *try / catch / finally*.

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
finally
{
   // Code that always runs to close files or release resources.
}
```

9.4. Lançando exceções

Você pode criar uma instância de uma classe *Exception* no seu código e lançar a exceção para indicar que ocorreu uma exceção. Quando você lança uma exceção, a execução do bloco de código atual termina e o CLR passa o controle para o primeiro manipulador de exceções disponível que captura a exceção.

Para lançar uma exceção, você usa a palavra chave *throw* e especifica o objeto de exceção a ser lançado.

O exemplo de código a seguir mostra como criar uma instância da classe *NullReferenceException* e depois a joga no objeto ex.

```
var ex = new NullReferenceException("The 'Name' parameter is null.");
throw ex;
```

Uma estratégia comum é um método ou bloco de código capturar quaisquer exceções e tentar lidar com elas. Se o bloco de captura de uma exceção não puder resolver o erro, ele poderá repetir a exceção para propagar o código de chamada.

O exemplo de código a seguir mostra como relançar uma exceção que foi capturada em um bloco de captura.

```
try
{
   // Code that could cause an exception
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Attempt to handle the exception
    ...
    // If this catch handler cannot resolve the exception,
    // throw it to the calling code
    throw;
}
```