

FRAMEWORKS JAVA, .NET &  
WEBSERVICES

# *A LINGUAGEM C* **MAIS AFIADA DO QUE NUNCA**



**3B**

## LISTA DE FIGURAS

Figura 1 – Declaração de classe C# .....	18
Figura 2 – Instanciando objetos em C# .....	18
Figura 3 – Declaração de atributos .....	19
Figura 4 – Construtores.....	22
Figura 5 – Erro de acesso a atributos .....	26
Figura 6 – Erro de acesso aos construtores.....	26
Figura 7 – Erro de acesso a métodos .....	27
Figura 8 – Membros herdados da classe base.....	29

## LISTA DE QUADROS

Quadro 1 – Tipos primitivos.....	7
Quadro 2 – Precedência de operadores .....	11
Quadro 3 – Modificadores de acesso.....	24
Quadro 4 – Coleções mais utilizadas .....	44

EMANIP

## LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Declaração de variáveis numéricas .....	8
Código-fonte 2 – Atribuição de variáveis numéricas .....	9
Código-fonte 3 – Conversão de <i>long</i> para <i>int</i> .....	9
Código-fonte 4 – Operadores matemáticos .....	10
Código-fonte 5 – Operadores de atribuição .....	10
Código-fonte 6 – Condição <i>if...else</i> simples .....	12
Código-fonte 7 – Condição <i>if...else</i> operador condicional AND .....	12
Código-fonte 8 – Condição <i>else if</i> .....	13
Código-fonte 9 – Condição <i>Switch</i> .....	14
Código-fonte 10 – Estrutura <i>for</i> .....	15
Código-fonte 11 – Estrutura <i>while</i> .....	15
Código-fonte 12 – Estrutura <i>do...while</i> .....	16
Código-fonte 13 – Estrutura <i>foreach</i> .....	16
Código-fonte 14 – Criação do método .....	20
Código-fonte 15 – Criação de métodos com retorno .....	21
Código-fonte 16 – Utilizando construtores .....	23
Código-fonte 17 – Modificadores de acesso .....	25
Código-fonte 18 – Herança .....	28
Código-fonte 19 – Herança, validando métodos .....	30
Código-fonte 20 – Herança, virtual e override .....	31
Código-fonte 21 – Uso de classe e método <i>abstract</i> .....	33
Código-fonte 22 – Criando e usando interfaces .....	34
Código-fonte 23 – Lançando uma exceção .....	35
Código-fonte 24 – Capturando exceções .....	36
Código-fonte 25 – Criando uma <i>exception</i> personalizada .....	37
Código-fonte 26 – Lançando a <i>exception</i> personalizada .....	37
Código-fonte 27 – Tratando a <i>exception</i> personalizada .....	38
Código-fonte 28 – Criando <i>arrays</i> .....	39
Código-fonte 29 – Acessando conteúdo do <i>array</i> .....	40
Código-fonte 30 – Criando uma lista de Cursos .....	41
Código-fonte 31 – Importando o <i>namespace Generics</i> .....	41
Código-fonte 32 – Adicionando elementos na lista .....	42
Código-fonte 33 – Removendo elementos da lista .....	43
Código-fonte 34 – Coleção do tipo <i>SortedList</i> .....	45
Código-fonte 35 – Coleção do tipo <i>Dictionary</i> .....	46

## SUMÁRIO

1 A LINGUAGEM C MAIS AFIADA DO QUE NUNCA .....	6
1.1 Introdução .....	6
1.1.1 Tipos de variáveis .....	7
1.1.2 Operadores .....	9
1.1.3 Precedência de operadores .....	10
1.1.4 Controle de fluxo .....	11
1.1.5 Estruturas de repetições.....	14
1.1.6 Estrutura <i>for</i> .....	15
1.1.7 Estrutura <i>while</i> .....	15
1.1.8 Estrutura <i>do... while</i> .....	16
1.1.9 Estrutura <i>foreach</i> .....	16
1.2 Orientação a objetos .....	17
1.2.1 Classe e objeto.....	17
1.2.2 Atributos .....	18
1.2.3 Métodos.....	19
1.2.4 Construtores.....	21
1.2.5 Modificadores de acesso.....	23
1.2.6 Herança.....	27
1.2.7 Virtuais .....	29
1.2.8 Abstract .....	31
1.2.9 Interface .....	33
1.3 Exceções.....	34
1.3.1 Lançamento de <i>exception</i> .....	35
1.3.2 Tratamento de <i>exception</i> .....	35
1.3.3 Personalizando <i>exceptions</i> .....	36
1.4 Coleções .....	38
1.4.1 <i>Arrays</i> .....	38
1.4.2 Listas .....	40
1.4.3 Adicionando elementos na lista.....	41
1.4.4 Removendo elementos da lista .....	42
1.4.5 Conjuntos .....	43
CONCLUSÃO.....	48
REFERÊNCIAS.....	49

# 1 A LINGUAGEM C MAIS AFIADA DO QUE NUNCA

## 1.1 Introdução

A plataforma .NET permite o desenvolvimento de sistemas nas mais diferentes linguagens de programação, porém, nenhuma delas conquistou mais o mercado do que a linguagem C Sharp (C#).

Neste capítulo, veremos a importância dos conceitos de programação orientada a objeto (POO) aplicados a essa linguagem, que foi lançada no início do ano 2000 e é considerada relativamente nova, em comparação com outras linguagens de programação, como Pascal, Basic etc.

O C# é uma linguagem simples, orientada a objetos, combinando produtividade e o poder de linguagem como C e C++. Devido ao fato de ser uma linguagem relativamente nova, não apresenta problemas com documentação, pois oferece um grande acervo de documentos e exemplos on-line, além de livros, artigos, fóruns de discussões, blogs, repositórios de exemplos e outros materiais de referência.

Na primeira parte deste capítulo, apresentaremos instruções e comandos básicos, tais como: operações matemáticas, tipo de dados, estruturas de controle e repetição. Na segunda parte, abordaremos a base para orientação a objeto, descrevendo conceitos de classes, modificadores de acesso, construtores e herança.

Na terceira e na quarta partes, serão vistos os tópicos mais avançados de coleções, listas e conjuntos de dados, além de exceções, abstração e interfaces.

Agora é hora de mergulhar na linguagem C#, vamos?

### 1.1.1 Tipos de variáveis

O Quadro “Tipos primitivos”, a seguir, apresenta os tipos primitivos de variáveis do C#. Estes são conhecidos como tipos primitivos ou *value types*. Na linguagem C#, todas as variáveis e constantes são fortemente tipadas, toda declaração de método requer a especificação do tipo de cada parâmetro de entrada e a especificação do tipo do retorno.

Tipo	Tamanho	Valores aceitos
bool	1 byte	true e false
byte	1 byte	0 a 255
sbyte	1 byte	-128 a 127
short	2 bytes	-32768 a 32767
ushort	2 bytes	0 a 65535
int	4 bytes	-2147483648 a 2147483647
uint	4 bytes	0 to 4294967295
long	8 bytes	-922337203685477508 a 922337203685477507
ulong	8 bytes	0 a 18446744073709551615
float	4 bytes	-3.402823E38 a 3.402823E38
double	8 bytes	-1.79769313486232e308 a 1.79769313486232e308
decimal	16 bytes	números com até 28 casas decimais
char	2 bytes	Caracteres Unicode. Exemplo 'a','b','ç'

Quadro 1 – Tipos primitivos  
Fonte: Microsoft MSDN (2015)

As informações de um tipo de variável podem conter detalhes, como:

- Espaço em memória que o tipo utiliza.
- Tipo base que é herdado.
- Endereço de memória.
- Valor mínimo e valor máximo que pode armazenar.

## A linguagem C mais afiada do que nunca

- Operações permitidas.

Os tipos primitivos do C# são muito similares aos tipos primitivos do Java, por exemplo: `byte`, `char`, `double`, `float`, `long` usam a mesma palavra reservada nas duas linguagens. Já o tipo `boolean` do Java foi abreviado para `bool`.

Com os tipos primitivos do C# apresentados, podemos entender como devemos declarar variáveis, de que maneira elas funcionam na aplicação e como interagem em tipos diferentes de dados.

O Código-Fonte “Declaração de variáveis numéricas” apresenta a declaração de algumas variáveis numéricas e a interação entre elas. Veja:

```
// Exemplo utilizando Console Application .NET
// Não existe a necessidade da declaração do método Main

// Método Main comentado
//static void Main(string[] args)
//{
    int valorInt = 100;

    // convertendo inteiro para long
    long valorLong = valorInt;

    // convertendo long para double
    double valorDouble = valorLong;

    // Imprimindo conteúdo da variável
    Console.WriteLine("Valor Inteiro:" + valorInt);
    Console.WriteLine("Valor Long:" + valorLong);
    Console.WriteLine("Valor Double:" + valorDouble);

    // Para a execução até o usuário teclar Enter.
    Console.Read();
//}
```

Código-fonte 1 – Declaração de variáveis numéricas  
Fonte: Elaborado pelo autor (2022)

O caso anterior não apresenta incompatibilidade na associação das variáveis do tipo *int*, *long* e *double*, pois todas as interações são feitas atribuindo a variável de menor tamanho para a variável de maior tamanho. Vamos adicionar uma linha ao exemplo e tentar associar o valor da variável *long* para a variável *int*. Segue o exemplo:



## A linguagem C mais afiada do que nunca

```
static void Main(string[] args)
{
    int valorInt = 100;
    long valorLong = valorInt;
    double valorDouble = valorLong;

    // Tentando converter long para int
    valorInt = valorLong;
}
```

Código-fonte 2 – Atribuição de variáveis numéricas  
Fonte: Elaborado pelo autor (2022)

A última linha de código apresenta um erro que impossibilita a compilação do projeto. A variável do tipo **int** não aceita um valor do tipo **long** sem a declaração de uma conversão; dessa forma, alteraremos a última linha para adicionar a conversão e executaremos o programa de exemplo, conforme o Código-Fonte “Conversão de *long* para int” abaixo:

```
static void Main(string[] args)
{
    int valorInt = 100;
    long valorLong = valorInt;
    double valorDouble = valorLong;

    // declaração de conversação (Parse)
    valorInt = (int) valorLong;

    Console.WriteLine(valorInt);
    Console.WriteLine(valorLong);
    Console.WriteLine(valorDouble);

    Console.Read();
}
```

Código-fonte 3 – Conversão de *long* para int  
Fonte: Elaborado pelo autor (2022)

### 1.1.2 Operadores

Vimos, na seção anterior os tipos de dados da linguagem C#, bem como a associação e a simples conversão entre os tipos. Nesta seção, serão apresentadas algumas operações básicas de atribuição, cálculos numéricos, lógicos e relacionais. Os atributos do tipo lógico e relacional são encontrados com estruturas de controle e repetição e serão apresentados futuramente; para os operadores numéricos e de atribuições, a seguir, veja o Código-Fonte “Operadores matemáticos” como exemplo:

## A linguagem C mais afiada do que nunca

```
static void Main(string[] args)
{
    // Operadores para Cálculos
    int soma = 10 + 15 + 3;
    int subtracao = soma - 10;
    int multiplicacao = soma * subtracao;
    double divisao = multiplicacao / subtracao;

    Console.WriteLine(soma);
    Console.WriteLine(subtracao);
    Console.WriteLine(multiplicacao);
    Console.WriteLine(divisao);
    Console.Read();
}
```

Código-fonte 4 – Operadores matemáticos  
Fonte: Elaborado pelo autor (2022)

```
static void Main(string[] args)
{
    // Atribuição
    int soma = 10;
    soma += 1; // Valor final 11

    int subtracao = soma;
    subtracao -= 10; // Valor final 1

    int multiplicacao = soma * subtracao;
    multiplicacao *= 3; // Valor final 33

    double divisao = multiplicacao;
    divisao /= soma; // Valor final 3

    // ... Inserir bloco para impressão dos valores
    Console.Read();
}
```

Código-fonte 5 – Operadores de atribuição  
Fonte: Elaborado pelo autor (2022)

**IMPORTANTE:** Conforme o comentário no Código-Fonte, “Operadores de atribuição”, é necessário inserir comandos para impressão a fim de apresentar os valores das variáveis na tela.

### 1.1.3 Precedência de operadores

Quando uma expressão C# for composta por vários operadores, a linguagem determina a sequência de execução de cada um deles. O Quadro “Precedência de

A linguagem C mais afiada do que nunca

operadores” ilustra os operadores da linguagem e sua precedência, na ordem do mais alto para o mais baixo, ou seja, os primeiros da lista são executados primeiro.

Categoria	Operador
Primário	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unário	+, -, !, ~, ++x, --x, (T)x
Multiplicativos	*, /, %
Aditivos	+, -
Troca	<<, >>
Relacionais	<, >, <=, >=, is
Igualdade	==
Lógico AND	&
Lógico XOR	^
Lógico OR	
Condicional AND	&&
Condicional OR	
Condicional	?:
Associação	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=,  =

Quadro 2 – Precedência de operadores  
Fonte: Elaborado pelo autor (2018)

O C# adota os mesmos operadores das linguagens Java, C e C++ e também a mesma precedência destas.

#### 1.1.4 Controle de fluxo

O controle de fluxo é o tema mais comum em qualquer projeto de software, assim, todas as linguagens de programação possuem estruturas condicionais muito similares.

A estrutura **if...else** é uma das mais utilizadas, e, sempre, é a primeira a ser introduzida no aprendizado de lógica de programação ou no aprendizado de uma

A linguagem C mais afiada do que nunca

linguagem. Nos próximos parágrafos, vamos percorrer exemplos de códigos-fonte de estruturas, condições e operadores lógicos.

Segue o exemplo de condição **if...else** simples:

```
static void Main(string[] args)
{
    int idade = 17;
    if (idade >= 18)
    {
        Console.WriteLine("É maior de idade");
    } else
    {
        Console.WriteLine("Ação não permitida para menores de 18 anos");
    }
    Console.Read();
}
```

Código-fonte 6 – Condição *if...else* simples  
Fonte: Elaborado pelo autor (2022)

Adicionando o operador condicional AND para validar o fluxo de uma condição mais complexa:

```
static void Main(string[] args)
{
    int idade = 17;
    if (idade >= 18 && idade < 21)
    {
        Console.WriteLine("Você pode jogar na categoria SUB-20");
    } else
    {
        Console.WriteLine("Você NÃO pode jogar na categoria SUB-20");
    }

    Console.Read();
}
```

Código-fonte 7 – Condição *if...else* operador condicional AND  
Fonte: Elaborado pelo autor (2022)

Com a estrutura *if...else*, é possível adicionar mais blocos de condição. No exemplo a seguir, foram criadas condições para definir uma categoria de acordo com a idade. Para isso, devemos usar a estrutura **else if**, conforme o exemplo do Código-Fonte “Condição *else if*” a seguir:

```
int idade = 20;

if (idade > 15 && idade < 18)
{
    Console.WriteLine("SUB-17");
}
else if (idade > 17 && idade <= 21)
{
    Console.WriteLine("SUB-20");
}
else if (idade > 21 && idade < 24)
{
    Console.WriteLine("SUB-23");
}

Console.Read();
```

Código-fonte 8 – Condição *else if*  
Fonte: Elaborado pelo autor (2022)

Por fim, temos a estrutura de controle para escolhas chamada *Switch*. No Código-Fonte “Condição *Switch*” é apresentado o código para definição da categoria com o uso da estrutura do comando *Switch*.

```
int idade = 16;

switch (idade)
{
    case 15:
        Console.WriteLine("SUB-15");
        break;
    case 16:
        Console.WriteLine("SUB-17");
        break;
    case 17:
        Console.WriteLine("SUB-17");
        break;
    case 18:
        Console.WriteLine("SUB-20");
        break;
    default:
        Console.WriteLine("Categoria não definida");
        break;
}

Console.Read();
```

Código-fonte 9 – Condição *Switch*  
Fonte: Elaborado pelo autor (2022)

Se compararmos os códigos-fonte deste subcapítulo com um exemplo em Java, notaremos que há duas pequenas diferenças. A primeira é no nome do método **Main**, que, em Java, seria **main**, e a segunda é no comando para imprimir na tela o resultado. Ou seja, as palavras-chave e a estrutura de código para controlar o fluxo são iguais em ambas as linguagens.

### 1.1.5 Estruturas de repetições

Passamos das estruturas condicionais para o controle de fluxo e vamos agora falar sobre as estruturas de repetição. Essas estruturas são responsáveis pela execução e pelo controle de comandos executados repetidamente.

Também conhecidas como laços, as estruturas de repetição disponíveis na linguagem C# são: *for*, *while*, *do...while* e *foreach*.

### 1.1.6 Estrutura *for*

Para escrever um laço usando a estrutura *for* é necessária a declaração de três partes, sendo elas: inicialização de uma variável, condição e atualização da variável. A separação de cada uma das partes é feita pelo símbolo “;”, como pode ser visto no exemplo do Código-Fonte “Estrutura *for*”, em que é executada uma contagem de 0 até 100.

```
// Contando de 0 a 100
for(int i = 0; i < 101; i += 1)
{
    Console.WriteLine(i);
}
Console.Read();
```

Código-fonte 10 – Estrutura *for*  
Fonte: Elaborado pelo autor (2022)

### 1.1.7 Estrutura *while*

Com uma sintaxe diferente do *for*, a estrutura *while* necessita apenas da declaração da condição. É uma condição mais simples de entender, porém, requer mais linhas de código se comparada ao *for*. Segue o exemplo do código-fonte anterior (Código-fonte “Estrutura *for*”) usando a estrutura *while*.

```
// Contando de 0 a 100
int i = 0;
while(i < 101)
{
    Console.WriteLine(i);
    i += 1;
}
Console.Read();
```

Código-fonte 11 – Estrutura *while*  
Fonte: Elaborado pelo autor (2022)

### 1.1.8 Estrutura *do... while*

Essa estrutura é uma variação da estrutura *while*, a diferença é que a condição é verificada após a execução do bloco de código. Assim, ao menos uma vez, o bloco será executado, conforme apresenta o exemplo do Código-Fonte “Estrutura *do... while*”:

```
// Contando de 0 a 100
int i = 0;
do
{
    Console.WriteLine(i);
    i += 1;
} while (i < 101);
Console.Read();
```

Código-fonte 12 – Estrutura *do...while*  
Fonte: Elaborado pelo autor (2022)

### 1.1.9 Estrutura *foreach*

O *foreach* é uma estrutura de laço utilizada basicamente para percorrer *arrays*, lista e coleção, conceitos que serão apresentados em capítulos futuros. Pode ser considerado uma forma resumida do *for* para percorrer dados em uma lista. No exemplo a seguir, vamos percorrer uma lista de nomes e exibir cada um deles na tela. Veja o Código-Fonte “Estrutura *foreach*”:

```
string[] lista = { "Fiap", "Fiap On", "Fiap School" };

foreach (string nome in lista)
{
    Console.WriteLine(nome);
}

Console.Read();
```

Código-fonte 13 – Estrutura *foreach*  
Fonte: Elaborado pelo autor (2022)

Mais uma vez, podemos notar a semelhança entre C# e Java nos comandos e nas palavras-chave usadas para controle de repetições.



A linguagem C mais afiada do que nunca

## 1.2 Orientação a objetos

### 1.2.1 Classe e objeto

A definição para Classe é um conjunto de objetos com as mesmas características, formado de propriedades e comportamentos por meio dos seus métodos. Podemos pensar em uma classe como a forma de organizar o código e o nosso sistema.

O objeto, também conhecido como instância de uma classe, é responsável por armazenar os conteúdos de suas propriedades e executar comportamento e ações por meio de seus métodos.

Para entender melhor os conceitos de classe e objeto, usaremos a linguagem C# e criaremos uma classe e algumas instâncias.

Vamos imaginar que haja a necessidade de criar uma aplicação para controlar cursos; para cada curso, precisamos das seguintes informações: código, nome do curso, nome do instrutor, carga horária equantidade mínima e máxima de alunos. Essas informações são o nosso modelo de curso para o nosso sistema.

Uma classe na linguagem C# é criada pela descrição dos modificadores de acesso, acrescidos da palavra **class** e do nome da classe.

**[modificador de acesso] class [NomeDaClasse] { }**

As Figuras “Declaração de classe C#” e “Instanciando objetos em C#” exibem a declaração da classe Curso e a forma de criar instâncias ou objetos:

```
namespace FiapHelloWord
{
    public class Curso
    {
    }
}
```

Definição da Classe

Figura 1 – Declaração de classe C#  
Fonte: Elaborado pelo autor (2022)

```
namespace FiapHelloWord
{
    class Program
    {
        static void Main(string[] args)
        {
            Curso cursoXamarin = new Curso();
            Curso cursolonic = new Curso();
        }
    }
}
```

Criação dos objetos cursoXamarin e cursolonic

Figura 2 – Instanciando objetos em C#  
Fonte: Elaborado pelo autor (2022)

### 1.2.2 Atributos

As propriedades definidas na seção anterior (código, nome do curso, nome do instrutor, carga horária, quantidade mínima e máxima de alunos) serão transformadas em atributos da nossa classe. Estes serão os responsáveis por armazenar as informações do objeto.

No código escrito para definir a classe, os atributos são declarados como variáveis, podendo ser do tipo de outro objeto ou do tipo primitivo do C#. A declaração de um atributo é igual à declaração de uma variável, porém, na boa prática de C#, os atributos devem ser escritos com a primeira letra em maiúsculo (*UperCamelCase*).

As variáveis que definem um atributo em uma classe são chamadas de variáveis de instância, pois só é possível armazenar informação nessas variáveis após a instanciação da Classe, ou seja, no objeto (YAMAMOTO, 2017). Veja a imagem da classe Curso e suas propriedades:

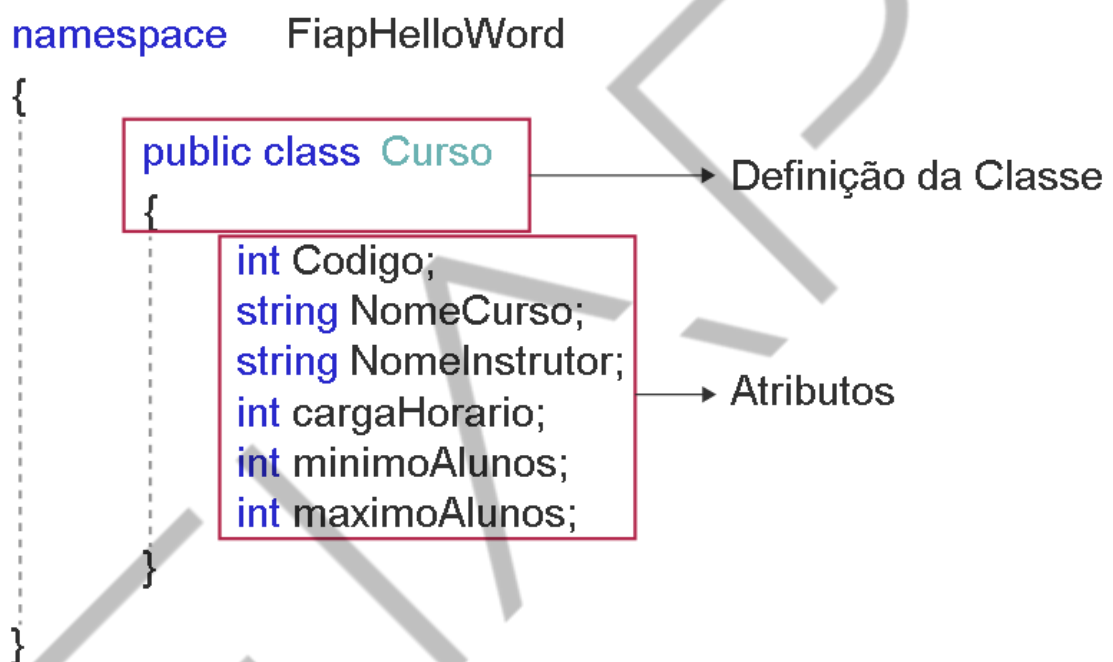


Figura 3 – Declaração de atributos  
Fonte: Elaborado pelo autor (2018)

### 1.2.3 Métodos

São os responsáveis pela execução das ações nos objetos. Eles dão comportamento ao objeto e são executados ao receber uma mensagem em tempo de execução da classe.

Diferentemente da linguagem Java, em C# todo método deve ter seu nome iniciado com letra maiúscula, assim como os atributos de uma classe. Todo o método tem acesso aos dados armazenados nas propriedades da instância, sendo capaz de controlá-los e alterá-los.

## A linguagem C mais afiada do que nunca

Todos os métodos necessitam de quatro informações para sua implementação, são elas: modificador de acesso, tipo de retorno, nome do método e argumentos (não obrigatório).

Para nossa classe `Curso`, criaremos o primeiro método, o qual terá a responsabilidade de criar um curso. Esse método receberá o nome do curso e o nome do instrutor. Veja o código-fonte no Código-Fonte “Criação do método”.

```
public class Curso
{
    int Codigo;
    string NomeCurso;
    string NomeInstrutor;
    int CargaHorario;
    int MinimoAlunos;
    int MaximoAlunos;

    public void CriarCurso(string nome, string instrutor)
    {
        this.NomeCurso = nome;
        this.NomeInstrutor = instrutor;
    }
}
```

Código-fonte 14 – Criação do método  
Fonte: Elaborado pelo autor (2018)

Note o uso da palavra reservada ***void***, ela indica que esse método não retorna nenhuma informação depois da execução.

Agora, temos a necessidade de criar dois novos métodos. O primeiro é responsável por matricular um aluno no curso. O segundo, tem a função de recuperar a quantidade máxima de alunos aceitos pelo curso. Assim, precisamos definir um tipo de dado que será retornado no método e também codificar o método para retornar à informação necessária.

O retorno de informações por um método é feito por meio da palavra reservada ***return***.

Veja, no Código-Fonte “Criação de métodos com retorno”, os dois exemplos de criação dos métodos para a classe `curso`:

```
public bool MatricularAluno(string nomeAluno)
{
    // Verificar a quantidade de alunos
    return true;
}

public int ConsultarMaximoAlunos()
{
    // Retorno o valor do atributo
    return this.maximoAlunos;
}
```

Código-fonte 15 – Criação de métodos com retorno  
Fonte: Elaborado pelo autor (2018)

### 1.2.4 Construtores

De forma resumida, um construtor é um método especial que é executado assim que uma nova instância da classe é criada. Na maioria dos casos, os construtores são responsáveis pela alocação de recursos e definição inicial dos atributos do objeto.

Todas as classes possuem pelo menos um construtor. Caso o desenvolvedor não implemente nenhum na classe, a linguagem cria o construtor-padrão ou *default*, esse não recebe nenhum parâmetro e não possui nenhum bloco de código implementado.

Existem três particularidades no construtor que o diferenciam de um método, são elas:

- O construtor não tem especificação de retorno.
- Não utiliza a palavra **return**, pois nunca retorna nenhum valor.
- É obrigatório ter o mesmo nome da classe.

Para fixar o conhecimento, vamos adicionar alguns construtores à classe Curso, com a ideia de inicializar os objetos com valores predefinidos. Seguem os exemplos na Figura “Construtores” a seguir:

A linguagem C mais afiada do que nunca

```
public class Curso
{
    0 references
    public Curso()
    {
        //Construtor padrão
    }

    0 references
    public Curso(string nome, string instrutor)
    {
        this.NomeCurso = nome;
        this.NomeInstrutor = instrutor;
    }

    0 references
    public Curso(string nomeCurso, int minimoAlunos, int maximoAlunos)
    {
        NomeCurso = nomeCurso;
        MinimoAlunos = minimoAlunos;
        MaximoAlunos = maximoAlunos;
    }
}
```

→ Padrão

→ Construtores personalizados

Figura 4 – Construtores  
Fonte: Elaborado pelo autor (2022)

Agora podemos criar objetos do tipo `Curso` com três formas de instanciar a classe. A primeira dela foi mantida como padrão, a segunda podemos afirmar que substitui o método “CriarCurso” implementado nos exemplos anteriores e, por fim, o terceiro construtor inicializa o objeto do tipo curso com nome e capacidades mínima e máxima já definidas.

Nesse código-fonte, podemos visualizar a forma de uso, com os três construtores sendo utilizados.

```
static void Main(string[] args)
{
    // Construtor padrão
    Curso cursoXamarin = new Curso();
    cursoXamarin.CriarCurso("Xamarin", "Flavio Moreni");

    // Definindo nome do curso e instrutor
    Curso cursoIonic = new Curso("Ionic", "Antonio
Coutinho");

    // Definindo nome do curso e capacidade mínima e máxima
    Curso cursoNode = new Curso("Node.js", 5, 40);
}
```

Código-fonte 16 – Utilizando construtores  
Fonte: Elaborado pelo autor (2022)

### 1.2.5 Modificadores de acesso

A razão do uso de modificadores de acesso é prover segurança entre os componentes de um sistema. Os modificadores são palavras-chave que determinam o nível de acesso em classes, já os construtores são métodos e propriedades.

A linguagem C# possui cinco modificadores de acesso, são eles: *public*, *protected internal*, *protected*, *internal* e *private*. Para cada tipo de objeto, o C# tem um modificador padrão, ou seja, quando o desenvolvedor não declara nenhum modificador, o .NET Framework define automaticamente os seguintes modificadores:

- Classes – padrão *internal*.
- Atributos de classe – padrão *private*.
- Membros de estrutura – padrão *private*.
- Namespace, interfaces e enumeradores – padrão *public*, esses tipos não podem sofrer alteração nos modificadores, sempre serão públicos.

Além das definições de modificadores padrão, cada modificador tem uma definição de acesso. O quadro “Modificadores de acesso” apresenta todos os modificadores e os componentes que podem ser aplicados e os níveis de acesso permitidos.

## A linguagem C mais afiada do que nunca

Modificador	Componentes	Descrição de acesso
<i>public</i>	classe e atributos	Nenhuma restrição
<i>protected</i>	atributos	Acesso para classe e seus filhos
<i>internal</i>	classe e atributos	Sem restrição dentro do mesmo projeto
<i>protected internal</i>	atributos	Acesso dentro do mesmo projeto e classes filhas
<i>private</i>	atributos	Acesso somente pela classe

Quadro 3 – Modificadores de acesso  
Fonte: Elaborado pelo autor (2022)

Para validar os modificadores e os acessos permitidos, vamos aplicar algumas alterações na classe `Curso`. Nos atributos da classe, devemos aplicar cada um dos tipos do Quadro “Modificadores de acesso” em um dos construtores, aplicaremos o modificador *private*, e assim faremos em cada um dos métodos. Observe o Código-Fonte “Modificadores de acesso”:

```
namespace FiapHelloWorld
{
    public class Curso
    {
        #region atributos
        int Codigo;
        internal string NomeCurso;
        public string NomeInstrutor;
        private int CargaHorario;
        protected int MinimoAlunos;
        protected internal int MaximoAlunos;
        #endregion

        public Curso()
        {
            // Construtor padrão.
        }

        protected internal Curso(string nome, string
instrutor)
        {
            this.NomeCurso = nome;
            this.NomeInstrutor = instrutor;
        }

        private Curso(string nome, int minimo, int maximo)
```



```
{
    this.NomeCurso = nome;
    this.MaximoAlunos = maximo;
    this.MinimoAlunos = minimo;
}

public void CriarCurso(string nome, string instrutor)
{
    this.NomeCurso = nome;
    this.NomeInstrutor = instrutor;
}

private bool MatricularAluno(string nomeAluno)
{
    // Verificar a quantidade de alunos
    return true;
}

private int ConsultarMaximoAlunos()
{
    // Retorno o valor do atributo
    return this.MaximoAlunos;
}
}
```

Código-fonte 17 – Modificadores de acesso  
Fonte: Elaborado pelo autor (2022)

Com as alterações na classe `Curso`, podemos usar nossa classe **Program.cs** para validar os acessos. Na classe **Program.cs**, criaremos uma instância da classe `Curso` e tentaremos acessar todos os atributos, conforme a Figura “Erro de acesso a atributos”:

```
static void Main(string[] args)
{
    Curso curso1 = new Curso();
    curso1.Codigo = 1;
    curso1.NomeCurso = "Nome do Curso";
    curso1.NomeInstrutor = "Instrutor";

    curso1.CargaHorario = 40;
    curso1.MinimoAlunos = 10;
    curso1.MaximoAlunos = 50;
}
```

Figura 5 – Erro de acesso a atributos  
Fonte: Elaborado pelo autor (2022)

Podemos notar que três linhas ficaram sinalizadas e apresentam problemas de compilação. A razão desses problemas é a permissão de acesso que foi concedida aos atributos "Codigo", "CargaHoraria" e "MinimoAlunos", impossibilitando o acesso pela classe Program.

Em seguida, efetuaremos os testes com os construtores. A Figura "Erro de acesso aos construtores" exibe três objetos do tipo Cursos sendo criados, cada um deles com um construtor diferente. Veja:

```
static void Main(string[] args)
{
    Curso curso1 = new Curso();
    Curso curso2 = new Curso("Curso", "Instrutor");
    Curso curso3 = new Curso("Node.js", 5, 40);
}
```

Figura 6 – Erro de acesso aos construtores  
Fonte: Elaborado pelo autor (2022)

A linguagem C mais afiada do que nunca

Note que a instância **curso3** apresenta erro, pois seu perfil de acesso foi declarado como *private*, assim, não é permitido o acesso de fora da classe Curso.

Para finalizar, o último exemplo são os acessos aos métodos. Na Figura “Erro de acesso a métodos”, é possível notar que o método **MatricularAluno** e **ConsultarMaximoAlunos** apresentarão acesso na chamada da classe Program.cs. Segue a figura com o erro:

```
static void Main(string[] args)
{
    Curso curso1 = new Curso();
    curso1.CriarCurso("nome", "instrutor");
    curso1.MatricularAluno("aluno");
    curso1.ConsultarMaximoAlunos();
}
```

Figura 7 – Erro de acesso a métodos  
Fonte: Elaborado pelo autor (2022)

A forma mais fácil de corrigir esses problemas é declarando todos os atributos, construtores e métodos como públicos, assim não haverá mais problemas de acesso.

Mas muita atenção: essa estratégia é apenas para resolver os erros e continuar executando nossa aplicação. Projetos profissionais requerem níveis bem definidos de acesso aos componentes.

**DICA:** Não declare todos os atributos, métodos e construtores como públicos. Analise componente a componente e organize-os em *namespaces* de domínios similares, essa estratégia facilitará nas definições de acesso em seu sistema.

### 1.2.6 Herança

A vantagem do uso do conceito de herança é a reutilização de código. Assim como o encapsulamento e o polimorfismo, a herança é uma característica da orientação ao objeto.

A linguagem C mais afiada do que nunca

Temos dois conceitos de classe para a herança. A primeira é a base, classe que terá seu código reaproveitado. A segunda é a classe derivada, que é especialização da classe base.

Toda classe derivada é formada implicitamente por todos os membros da classe base, exceto construtores e finalizadores. Assim, todo o código da classe base fica disponível para utilização na classe derivada, além de ser possível adicionar novos comportamentos e atributos, transformando assim a classe derivada em uma classe mais especializada da classe base.

Uma classe derivada pode ter apenas uma classe base, porém, a herança é transitiva, ou seja, se sua classe base for uma classe derivada, sua classe final herdará todos os membros declarados nas duas classes base.

Para entender a herança em C#, criaremos uma classe chamada **CursoFerias**, que será derivada da nossa classe **Curso**. A forma de implementar a herança no C# é usando o “:” depois do nome da classe e seguido do nome da classe base. Segue o exemplo no Código-Fonte “Herança”:

```
using System;

namespace FiapHelloWorld
{
    public class CursoFerias: Curso
    {
    }
}
```

Código-fonte 18 – Herança  
Fonte: Elaborado pelo autor (2022)

Criando uma instância da classe **CursoFerias** é possível notar que temos acesso a todos os membros da classe **Curso**. A Figura “Membros herdados da classe base”, a seguir, apresenta a tela do Visual Studio com as opções dos membros da classe **CursoFerias**.

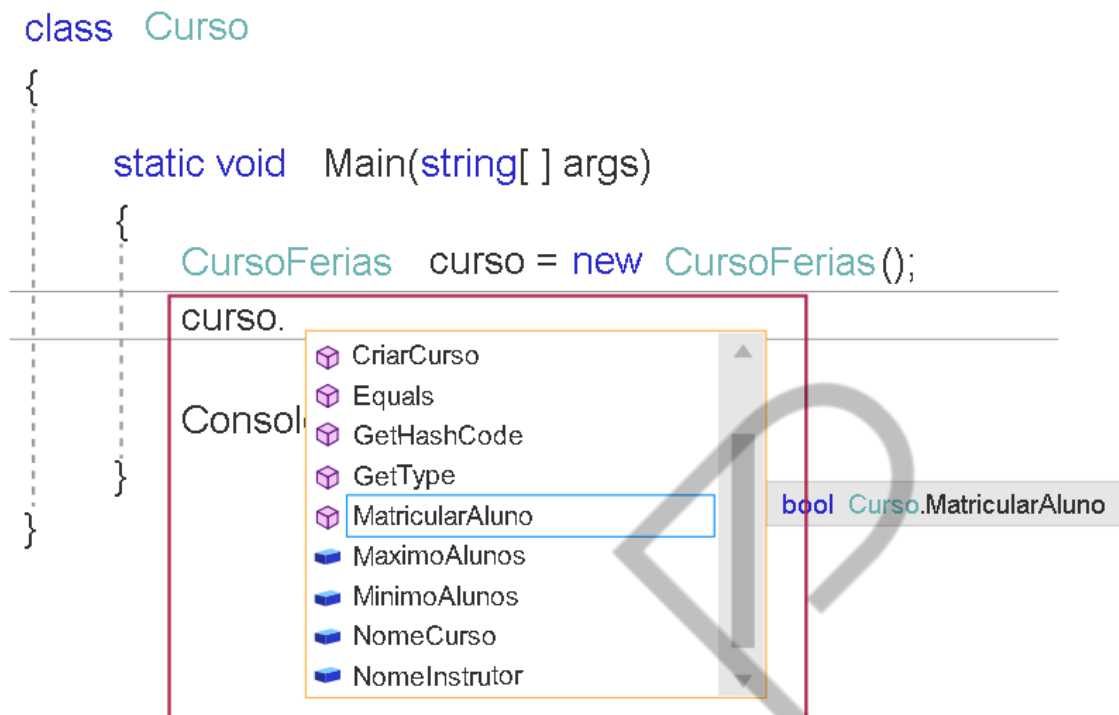


Figura 8 – Membros herdados da classe base  
Fonte: Elaborado pelo autor (2022)

### 1.2.7 Virtuais

A palavra-chave **virtual** indica para o método que uma classe derivada pode substituir o método por sua própria implementação. O método da classe derivada precisa ser declarado com a palavra-chave **override**. O não uso dos termos **virtual** e **override** não traz erros de compilação, porém, apresenta resultados diferentes.

Para entender melhor, vamos usar o código-fonte abaixo. Três classes são criadas para o exemplo, a primeira é a classe base, a segunda é a classe derivada e a terceira é a classe de execução. Veja o Código-Fonte “Herança, validando métodos” sem o uso das palavras **virtual** e **override**:

```
public class ClasseBase
{
    public void Metodo()
    {
        Console.WriteLine("Método ClasseBase");
    }
}

public class Derivada: ClasseBase
```

```
{
    public void Metodo()
    {
        Console.WriteLine("Método Derivada");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        ClasseBase a = new ClasseBase();
        a.Metodo();

        Derivada b = new Derivada();
        b.Metodo();

        ClasseBase c = new Derivada();
        c.Metodo();

        Console.Read();
    }
}
```

Código-fonte 19 – Herança, validando métodos  
Fonte: Elaborado pelo autor (2022)

Após a execução do programa, as mensagens na tela informam que a instância “c” executou o método da classe base. O resultado esperado são as três linhas de mensagem: **Método ClasseBase, Método Derivada e Método ClasseBase**.

Para resolver o problema da última execução do método, usaremos a declaração de **virtual** e **override**. A seguir, temos o Código-Fonte “Herança, virtual e override” alterado:

```
public class ClasseBase
{
    public virtual void Metodo()
    {
        Console.WriteLine("Método ClasseBase");
    }
}

public class Derivada: ClasseBase
{
    public override void Metodo()
    {

```

```
        Console.WriteLine("Método Derivada");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        ClasseBase a = new ClasseBase();
        a.Metodo();

        Derivada b = new Derivada();
        b.Metodo();

        ClasseBase c = new Derivada();
        c.Metodo();

        Console.Read();
    }
}
```

Código-fonte 20 – Herança, virtual e override  
Fonte: Elaborado pelo autor (2022)

Após a execução, é possível notar que a última impressão foi a mensagem **Método Derivada**, ou seja, o método da classe base foi realmente substituído pelo método da classe derivada.

As palavras **virtual** e **override** apresentam comportamentos diferentes na linguagem Java. Em Java, quando queremos que um método não seja sobrescrito, devemos declará-lo como final; sem o modificador final ou qualquer método pode ser sobrescrito. Em Java, não temos a palavra **override**; para sobrescrever um método, basta codificar na classe filho com o mesmo nome, retorno e parâmetros de entrada.

### 1.2.8 Abstract

A palavra-chave *abstract* pode ser usada em classes e métodos, seu objeto ou uso é permitir que classes ou métodos que estão incompletos sejam implementados nas classes que herdam a abstração, as classes derivadas.

Tanto uma classe abstrata quanto um método abstrato possuem particularidades, seguem algumas:

### Classes

- Não pode ser instanciada. Não é permitido criar um objeto a partir de uma classe abstrata.
- Geralmente é usada como classe base para outras classes.
- Pode conter métodos abstratos e métodos comuns.
- Pode possuir construtores e propriedades.
- Não pode ser estática (*static*).
- Pode herdar de outra classe abstrata.

### Métodos

- Não possui implementação na classe abstrata. É composto apenas por sua assinatura.
- A classe que deriva de uma classe abstrata precisa implementar seus métodos abstratos. Caso contrário, um erro de compilação é apresentado.
- Método abstrato é **virtual** e deve ser implementado usando o modificador **override**.
- Somente pode existir em uma classe abstrata.
- Não pode usar os modificadores **static** e **virtual**.

Abaixo, segue o Código-fonte “Uso de classe e método *abstract*” com o exemplo de uso do **abstract**.

```
public abstract class ClasseBase
{
    public virtual void Metodo()
    {
        Console.WriteLine("Método ClasseBase");
    }

    public abstract void MetodoAbstrato();
}

public class Derivada: ClasseBase
{
    public override void Metodo()
```



```
{
    Console.WriteLine("Método Derivada");
}

public override void MetodoAbstrato()
{
    Console.WriteLine("Método MetodoAbstrato");
}
}
public class Program
{
    static void Main(string[] args)
    {
        Derivada b = new Derivada();
        b.Metodo();

        ClasseBase c = new Derivada();
        c.Metodo();
        c.MetodoAbstrato();

        Console.Read();
    }
}
```

Código-fonte 21 – Uso de classe e método *abstract*  
Fonte: Elaborado pelo autor (2022)

### 1.2.9 Interface

O conceito de interface tem uma leve semelhança com *abstract*. É outra forma de herança e de atribuir comportamentos.

A primeira diferença entre uma interface e uma classe abstrata é que uma pode derivar de mais de uma interface. A segunda é que uma interface define apenas assinaturas de métodos, nunca possui a sua implementação. A terceira, e última, é que, por convenção, o C# usa a letra I como prefixo em todas as interfaces.

Veja um exemplo de uso de interfaces no Código-Fonte “Criando e usando interfaces”.

```
public interface IAluno
{
    void CriarAluno();
}

public interface IInstrutor
{
    void CriarInstrutor();
}

public class Curso : IAluno, IInstrutor
{
    public void CriarAluno()
    {
        Console.WriteLine("Criando Aluno");
    }

    public void CriarInstrutor()
    {
        Console.WriteLine("Criando Instrutor");
    }
}
```

Código-fonte 22 – Criando e usando interfaces  
Fonte: Elaborado pelo autor (2022)

### 1.3 Exceções

Exceções são condições de erro no fluxo de execução, indicam que um evento inesperado ocorreu durante a execução. Quando eventos inesperados acontecem, objetos de exceção são criados e, em seguida, lançados para a classe que enviou a mensagem para execução.

O .NET Framework é responsável pelo lançamento de várias exceções (por exemplo, tentativa de abrir um arquivo inexistente no sistema de arquivos). Por outro lado, os desenvolvedores podem ou devem lançar exceções no código. Seguem algumas situações em que exceções devem ser lançadas:

- método não pode concluir toda a sua funcionalidade;
- valores incorretos de argumentos dos métodos;
- chamadas a componentes inexistentes ou sem instância em memória;

A linguagem C mais afiada do que nunca

- falhas em conexão com recursos externos (por exemplo, banco de dados).

Para trabalhar com *exceptions* e entender seu uso, é preciso tratar sobre três tópicos. O primeiro é o lançamento de *exceptions*, o segundo é o tratamento de *exceptions* e o último é a criação de *exceptions* personalizadas.

### 1.3.1 Lançamento de exception

O lançamento de uma *exception* é feito pela palavra reservada **throw**. A palavra *throw* sinaliza que uma situação não esperada aconteceu durante a execução. Todas as exceções em C# são herdadas da classe *System.Exception*.

Veja o exemplo de lançamento de uma exceção na validação de dados no parâmetro de um método:

```
public void CriarAluno(string nome)
{
    if (nome == null)
    {
        // Lançando uma exceção
        throw new Exception("Nome do aluno inválido");
    }
}
```

Código-fonte 23 – Lançando uma exceção  
Fonte: Elaborado pelo autor (2022)

### 1.3.2 Tratamento de exception

A forma de tratar *exception* em C# é usando o bloco de código *try...catch*. O código a seguir apresenta o exemplo para capturar duas exceções, a primeira delas é *System.NullReferenceException* e a segunda é *Exception*. Segue o exemplo:

```
static void Main(string[] args)
{
    try
    {
        string nome = null;
        Console.WriteLine(nome.Substring(2));
        new Curso().CriarAluno(nome);
    }
    // Capturando uma exceção de referência nula.
    // Similar ao NullPointerException do Java
    catch (NullReferenceException)
    {
        Console.WriteLine("Nome do aluno incorreto");
    }
    catch (Exception)
    {
        Console.WriteLine("Problema na execução a
        operação");
        throw new Exception("Problema na execução a
        operação");
    }
}
```

Código-fonte 24 – Capturando exceções  
Fonte: Elaborado pelo autor (2022)

### 1.3.3 Personalizando exceptions

A linguagem C# permite o lançamento de exceções do *namespace* **System**, assim como possibilita a criação de exceções personalizadas derivando de **System.Exception**. Para criar uma exceção personalizada, a classe derivada precisa definir quatro construtores. Segue o exemplo de uma *exception* personalizada no Código-Fonte “Criando uma *exception* personalizada”.

## A linguagem C mais afiada do que nunca

```
public class PersonalizadaException: Exception
{
    public PersonalizadaException() : base() { }
    public PersonalizadaException(string message) :
base(message) { }
    public PersonalizadaException(string message,
System.Exception inner) : base(message, inner) { }

    protected PersonalizadaException(
        System.Runtime.Serialization.SerializationInfo
info,
        System.Runtime.Serialization.StreamingContext
context) { }
}
```

Código-fonte 25 – Criando uma *exception* personalizada  
Fonte: Elaborado pelo autor (2022)

Agora, podemos alterar a regra de validação do nome e substituir a exceção **System.Exception** por **PersonalizadaException**. Então é necessário também alterar o bloco *try...catch* e tratar a nova *exception*. Veja abaixo como ficou o Código-Fonte “Lançando a *exception* personalizada”:

```
public void CriarAluno(string nome)
{
    if (nome == null)
    {
        throw new PersonalizadaException("Nome do aluno
inválido");
    }
}
```

Código-fonte 26 – Lançando a *exception* personalizada  
Fonte: Elaborado pelo autor (2022)

```
public class Program
{
    static void Main(string[] args)
    {
        try
        {
            string nome = null;
            new Curso().CriarAluno(nome);
            Console.WriteLine(nome.Substring(2));
        }
        catch (PersonalizadaException p)
        {
            Console.WriteLine(p.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Problema na execução a
operação");
            throw new Exception("Problema na execução a
operação");
        }
    }
}
```

Código-fonte 27 – Tratando a *exception* personalizada  
Fonte: Elaborado pelo autor (2022)

## 1.4 Coleções

### 1.4.1 Arrays

Em qualquer linguagem de programação, a codificação de *arrays* é muito importante. Um dos problemas mais comuns no uso de *arrays* é a limitação de tamanho, é necessário ser bastante preciso com esse detalhe, caso contrário, a solução implementada com *arrays* pode virar um grande problema.

Em resumo, um *array* é uma estrutura de dados que facilita o armazenamento de diversos elementos, tornando a leitura e o acesso fáceis para os desenvolvedores. O exemplo apresenta as duas formas de criação de um *array* e o acesso e à manipulação dos dados de algumas posições do *array*, veja o Código-Fonte “Criando *arrays*” a seguir:

```
// Exemplo 1
string[] nomes1 = { "João", "Maria", "José" };

// Exemplo 2
string[] nomes2 = new string[3];
nomes1[0] = "João";
nomes2[1] = "Maria";
nomes2[3] = "José";
```

Código-fonte 28 – Criando *arrays*  
Fonte: Elaborado pelo autor (2022)

Nota-se que foram criados dois *arrays* com valores iguais, o primeiro *array* foi inicializado com conteúdo de nomes e não teve seu tamanho especificado pelo desenvolvedor, já o segundo foi criado com o tamanho especificado e o seu conteúdo foi adicionado em cada posição.

A criação de um *array* é semelhante à criação de uma instância de objeto, com a diferença do uso de colchetes “[ ... ]” adicionados ao tipo do objeto.

**DICA:** A posição inicial de um *array* é 0 (zero), logo, as posições vão de 0 (zero) até tamanho -1 (menos um).

No exemplo do Código-Fonte “Criando *arrays*”, elaboramos um *array* para armazenar uma lista de nomes, mas podemos enriquecer nosso exemplo e criar *array* de classes específicas do nosso sistema. Assim, o código-fonte abaixo apresenta um exemplo de criação de *array* da classe *curso* e a utilização da estrutura de repetição *foreach* para acesso e exibição dos valores na tela. Veja o Código-Fonte “Acessando conteúdo do *array*”:

```
public class Curso
{
    public int Codigo;
    public string NomeCurso;

    public Curso(int cod, string nome)
    {
        this.Codigo = cod;
        this.NomeCurso = nome;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        // Criando um array de curso
        Curso[] listaCursos = new Curso[3];

        // Criando os itens do array
        listaCursos[0] = new Curso(1, "Curso 1");
        listaCursos[1] = new Curso(2, "Curso 2");
        listaCursos[2] = new Curso(3, "Curso 3");

        // Navegando pelo array e imprimindo o conteúdo
        foreach (Curso curso in listaCursos)
        {
            Console.WriteLine(curso.NomeCurso);
        }

        Console.Read();
    }
}
```

Código-fonte 29 – Acessando conteúdo do *array*  
Fonte: Elaborado pelo autor (2022)

### 1.4.2 Listas

Podemos classificar as listas como uma evolução dos *arrays*. Com os *arrays*, é fácil guardar e acessar elementos, como apresentado no capítulo anterior. Porém, o que não é fácil de fazer com *arrays* é a manipulação, ou seja, adicionar e remover elementos de forma rápida.

Para facilitar a manipulação e resolver problemas com *arrays*, temos as listas, que na linguagem C# são representadas pela classe *List* e a interface *ICollection*.



A linguagem C mais afiada do que nunca

A criação de uma lista em C# segue o padrão da criação de um objeto, mas exige que seja especificada qual a classe de objetos que será armazenada na lista. O bloco abaixo apresenta a declaração de uma instância de *List*, na qual armazenaremos uma coleção de objetos do tipo *Curso*. Veja o Código-Fonte “Criando uma lista de Cursos”:

```
// Exemplo 1
string[] nomes1 = { "João", "Maria", "José" };

// Exemplo 2
string[] nomes2 = new string[3];
nomes1[0] = "João";
nomes2[1] = "Maria";
nomes2[3] = "José";
```

Código-fonte 30 – Criando uma lista de Cursos  
Fonte: Elaborado pelo autor (2022)

```
using System;
using System.Collections.Generic;

List<Curso> lista = new List<Curso>();
```

Código-fonte 31 – Importando o *namespace Generics*  
Fonte: Elaborado pelo autor (2022)

### 1.4.3 Adicionando elementos na lista

Com um objeto do tipo *List* instanciado, podemos iniciar a manipulação de elementos e entender seu funcionamento. Por meio do método *Add*, adicionamos instâncias do objeto *Curso*. A inclusão de elementos na lista também pode ser feita com o método *Insert*. Apesar de parecerem similares, possuem comportamentos diferentes, para o método *Add*, o novo elemento é adicionado no final da lista, já no método *Insert*, o desenvolvedor precisa informar qual a posição em que o objeto deve ser inserido.

Veja o exemplo do Código-Fonte “Adicionando elementos na lista”:

```
// Criando a lista dos objetos curso
List<Curso> lista = new List<Curso>();

// Adicionando cursos na lista
lista.Add(new Curso(1, "Curso 1"));
lista.Add(new Curso(2, "Curso 2"));
lista.Add(new Curso(4, "Curso 4"));

// Inserindo um curso em uma posição específica.
lista.Insert(2, new Curso(3, "Curso 3"));

foreach (var curso in lista)
{
    Console.WriteLine(curso.NomeCurso);
}

Console.Read();
```

Código-fonte 32 – Adicionando elementos na lista  
Fonte: Elaborado pelo autor (2022)

Os objetos do tipo `List` e `ICollection` em C# são muito parecidos com `ArrayList` e `List` do Java. No exemplo anterior, a única diferença entre C# e Java é o nome do método usado para incluir elementos na lista: **Add** no C# e **add** no Java.

#### 1.4.4 Removendo elementos da lista

Assim como a operação de adição de elementos em `List`, temos dois métodos que podemos usar para remover itens de uma lista. São eles:

- *Remove* – remove a primeira instância de um objeto específico de `List`.
- *RemoveAt* – remove o objeto de uma posição específica.

Aproveitamos o código-fonte anterior e codificamos um exemplo de uso dos dois métodos de remoção. Veja, a seguir, o Código-Fonte “Removendo elementos da lista”:

```
List<Curso> lista = new List<Curso>();
Curso c1 = new Curso(1, "Curso 1");
lista.Add(c1);
lista.Add(new Curso(2, "Curso 2"));
lista.Add(new Curso(4, "Curso 4"));
lista.Insert(2, new Curso(3, "Curso 3"));

// Removendo um objeto de uma determinada posição
lista.RemoveAt(3);

// Removendo o objeto pela referência de c1
lista.Remove(c1);

foreach (var curso in lista)
{
    Console.WriteLine(curso.NomeCurso);
}

Console.Read();
```

Código-fonte 33 – Removendo elementos da lista  
Fonte: Elaborado pelo autor (2022)

#### 1.4.5 Conjuntos

Até aqui, trabalhamos com conjunto de objetos do tipo *arrays* e *List*, mas a linguagem C# oferece outros recursos que permitem ao desenvolvedor implementar operações de busca de informações de uma forma mais simples e poderosa, esses recursos são denominados **coleções**.

As coleções em C# são encontradas no *namespace* **System.Collections**. Veja o Quadro “Coleções mais utilizadas” abaixo:

## A linguagem C mais afiada do que nunca

Classe	Descrição
<i>Dictionary</i>	Representa uma coleção de pares chave/valor que são organizados com base na chave.
<i>List</i>	Representa uma lista de objetos que podem ser acessados por índice. Fornece métodos para pesquisar, classificar e modificar listas.
<i>Queue</i>	Representa uma coleção de objetos, em que o primeiro a entrar é o primeiro a sair.
<i>SortedList</i>	Representa uma coleção de pares chave/valor que são classificados por chave, com base na implementação de <i>IComparer</i> associada.
<i>Stack</i>	Representa uma coleção, em que o último a entrar é o primeiro a sair.
<i>ArrayList</i>	Representa uma matriz de objetos cujo tamanho é dinamicamente ampliado, caso necessário.
<i>Hashtable</i>	Representa uma coleção de pares de chave/valor que são organizados com base no código Hash da chave.
<i>HashSet</i>	É uma coleção ordenada de itens exclusivos, ou seja, não pode ter itens duplicados e nenhuma ordem é mantida.

Quadro 4 – Coleções mais utilizadas  
Fonte: Microsoft MSDN (2022)

Para fixar o conhecimento, veremos dois exemplos de trabalho com coleções. No primeiro, utilizaremos uma coleção do tipo **SortedSet**, na qual executaremos operações de busca e verificaremos o mecanismo de ordenação automática. No segundo, faremos uso de uma coleção do tipo **Dictionary**, executando os operadores de adição, interação e a forma de acesso aos objetos da coleção.

Veja os exemplos:

```
// Criando uma lista ordenada
SortedSet<string> alunos = new SortedSet<string>();

// Adicionando elementos na lista
alunos.Add("Alberto");
alunos.Add("Giovanna");
alunos.Add("Fabio");
alunos.Add("Victor");
alunos.Add("Carlos");

Console.Write("Encontrou o aluno Carlos: ");
// Procurando na lista um determinado elemento
Console.WriteLine( alunos.Contains("Carlos") );
Console.WriteLine("");

foreach (string aluno in alunos)
{
    Console.WriteLine(aluno);
}

Console.Read();
```

Código-fonte 34 – Coleção do tipo *SortedList*  
Fonte: Elaborado pelo autor (2022)

```
public class Program
{
    static void Main(string[] args)
    {
        Curso c1 = new Curso(1, "Curso 1");
        Curso c2 = new Curso(2, "Curso 2");
        Curso c3 = new Curso(3, "Curso 3");

        // Criando uma lista de objeto na estrutura chave +
valor
        Dictionary<string, Curso> dicionario = new
Dictionary<String, Curso>();
        dicionario.Add(c1.NomeCurso, c1);
        dicionario.Add(c2.NomeCurso, c2);
        dicionario.Add(c3.NomeCurso, c3);

        // procurando um determinado elemento
        Console.Write("Encontrou o Curso 2: ");
        Console.WriteLine(dicionario["Curso 4s"] == null ?
false : true);
        Console.WriteLine("");

        // Navegando pela coleção e imprimindo os objetos.
        foreach (KeyValuePair<string, Curso> itemCurso in
dicionario)
        {
            string chave = itemCurso.Key;
            Curso c = dicionario[chave];
            Console.WriteLine(c.NomeCurso);
        }

        Console.Read();
    }
}

public class Curso
{
    public int Codigo;
    public string NomeCurso;

    public Curso(int cod, string nome)
    {
        this.Codigo = cod;
        this.NomeCurso = nome;
    }
}
```

Código-fonte 35 – Coleção do tipo *Dictionary*  
Fonte: Elaborado pelo autor (2022)

A linguagem C mais afiada do que nunca

DICA: Você pode consultar toda a especificação de Listas e Coleções da linguagem C# no site da Microsoft: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/collections>.

EMSE

## CONCLUSÃO

Neste capítulo, apresentamos os conceitos básicos da linguagem C#, as instruções básicas, os tipos de variáveis, os operadores, as estruturas de repetições e os conceitos de Orientação a Objeto. Tornando possível, assim, a criação de uma aplicação simples do tipo **console**. Abordamos também o uso de exceções, permitindo a criação personalizada, o tratamento e o lançamento delas.

Para aprofundar o conhecimento, demos alguns exemplos de uso de listas e coleções mais comuns da linguagem.



## REFERÊNCIAS

ANDRADE, C. **Trabalhando com Arrays no C#**. Microsoft. Disponível em: <https://msdn.microsoft.com/pt-br/library/cc564862.aspx>. Acesso em: 13 jan. 2021.

ARAÚJO, E. C. **C# e Visual Studio – Desenvolvimento de aplicações desktop**. São Paulo: Casa do Código, 2015.

ARAÚJO, E. C. **Orientação a Objetos em C# – Conceitos e implementações em .NET**. São Paulo: Casa do Código, 2017.

CARDOSO, G. S. **Criando aplicações para o seu Windows Phone**. São Paulo: Casa do Código, 2014.

LIMA, E. **C# e .NET – Guia do Desenvolvedor**. Rio de Janeiro: Campus, 2012.

MICROSOFT. **Criando e lançando exceções (Guia de Programação em C#)**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/exceptions/creating-and-throwing-exceptions>. Acesso em: 13 jan. 2021.

MICROSOFT. **Herança (Guia de Programação em C#)**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/inheritance>. Acesso em: 13 jan. 2021.

YAMAMOTO, T. T. I. **Classe, Métodos, Atributos**. São Paulo: FIAP, 2017.