FMX Academy

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
01 Lógica de Programação e C# Básico	03 - Introdução aos Métodos	Guilherme Paracatu

1. Revisão rápida e dúvidas

Recapitulando lógica de Fluxo!

Operadores: Qual o propósito dos operadores lógicos (&&, ||, !)? E a diferença entre == e =?

Condicionais (if/else if/else, switch): Como fazemos o programa tomar decisões? Quando usar switch em vez de if?

Loops (for, while, do-while, foreach): Qual a diferença entre for e while? Quando usar do-while? Para que servem break e continue?

2. Introdução aos Métodos: O que são e porque usá-los?

Métodos: Blocos de Construção do Código

Organize, Reutilize, Simplifique!

Imagine seu código como um grande projeto de construção. Se você coloca tudo em um só lugar, vira uma bagunça! Métodos são como 'subcontratados' especializados, que cuidam de tarefas específicas.

Analogia: Pense em um aplicativo de banco. Há uma função para "sacar dinheiro", outra para "ver extrato", outra para "fazer pix". Cada uma faz uma coisa específica.

2.1 – O problema do código Monolítico

Código "Macarrônico"

No início, escrevemos todo o código no Main.
Mas e se precisamos fazer a mesma sequência de ações várias vezes?
Ou, e se o Main fica gigantesco e difícil de entender?

Exemplo de código ruim :

```
// Imagine este código repetido várias vezes no seu Main
Console.WriteLine("--------------------");
Console.WriteLine("Calculadora Simples");
Console.WriteLine("--------------------");
Console.Write("Digite o primeiro número: ");
double num1 = Convert.ToDouble(Console.ReadLine());
Console.Write("Digite o segundo número: ");
double num2 = Convert.ToDouble(Console.ReadLine());
```

```
double resultado = num1 + num2;
Console.WriteLine($"Resultado da soma: {resultado}");
Console.WriteLine("--------------------");

// ... 1000 linhas de código depois, você precisa fazer outra soma ...
Console.WriteLine("--------------------");
Console.WriteLine("Calculadora Simples");
Console.WriteLine("--------------------");
Console.Write("Digite o primeiro número: ");
double numA = Convert.ToDouble(Console.ReadLine());
Console.Write("Digite o segundo número: ");
double numB = Convert.ToDouble(Console.ReadLine());
double resultado2 = numA + numB;
Console.WriteLine($"Resultado da soma: {resultado2}");
Console.WriteLine("--------------------");
```

Problemas: Repetição de código (duplicação), difícil de ler, difícil de manter (se mudar a lógica da soma, tem que mudar em vários lugares).

2.2 – A Solução: Métodos!

Métodos no Resgate!

- Métodos (ou funções/procedimentos) são blocos de código nomeados que realizam uma tarefa específica.
- Podemos 'chamar' ou 'invocar' um método sempre que precisarmos daquela tarefa.

Exemplo Simplificado de Solução:

```
void RealizarSoma() // Isso é um método!
{
    Console.WriteLine("--------------------");
    Console.WriteLine("Calculadora Simples");
    Console.WriteLine("--------------------");
    Console.Write("Digite o primeiro número: ");
    double num1 = Convert.ToDouble(Console.ReadLine());
    Console.Write("Digite o segundo número: ");
    double num2 = Convert.ToDouble(Console.ReadLine());
    double resultado = num1 + num2;
    Console.WriteLine($"Resultado da soma: {resultado}");
    Console.WriteLine("--------------------");
}

// No seu Main, você só 'chama' o método
// RealizarSoma(); // Chama a tarefa
// RealizarSoma(); // Chama a tarefa de novo, sem repetir o código
```

2.3 – Vantagens dos Métodos

Por Que Usar Métodos?

1. **Reutilização de Código:** Escreva uma vez, use em muitos lugares.
2. **Organização e Legibilidade:** Código dividido em blocos lógicos é mais fácil de ler e entender.
3. **Manutenção Facilitada:** Se precisar mudar a lógica de uma tarefa, mude em apenas um lugar.

4. **Colaboração:** Diferentes desenvolvedores podem trabalhar em diferentes métodos simultaneamente.
5. **Depuração (Debugging) Mais Fácil:** Problemas ficam isolados em métodos específicos.

2.3 – Analogia com Delphi: Procedures e Functions

Métodos em C# vs. Delphi: Procedures e Functions

- Em Delphi, vocês têm procedure (não retorna valor) e function (retorna um valor).
- Em C#, ambos são chamados de Métodos.
- A diferença é controlada pelo tipo de retorno do método:
  void: É o equivalente a uma procedure (não retorna nada).
  int, string, double, bool (etc.): É o equivalente a uma function (retorna um valor daquele tipo).

3. Estrutura Básica de um Método em C#

A Anatomia de um Método

Sintaxe Geral:

```
ModificadorDeAcesso TipoDeRetorno NomeDoMetodo(TipoParametro1 nomeParametro1, TipoParametro2 nomeParametro2, ...)
{
    // Corpo do Método: Onde o código da tarefa é executado
    // Pode haver um 'return' aqui se o tipo de retorno não for 'void'
}
```

Exemplo Esquemático:

```
public static void ExibirMensagem(string mensagem)
{
    Console.WriteLine(mensagem);
}
```

3.1 – Modificadores de Acesso (public, private, static)

Modificadores de Acesso: Quem Pode Chamar?

Definem onde o método pode ser acessado.

public:
O método pode ser acessado de qualquer lugar do seu programa (e até de outros programas/bibliotecas que o usem).

private:
O método só pode ser acessado de dentro da mesma classe onde foi definido. (É o padrão, se não especificar).

static:
"Significa que o método pertence à classe, não a um objeto específico dela."

"Para chamar um método static do Main (que também é static), o método que você criar também deve ser static no início."
"Vamos usar static para todos os nossos métodos por enquanto, para que o Main possa chamá-los diretamente."

Exemplo: public static void ...

3.2 – Tipo de Retorno (void, tipos de dados)

O que o método devolve?
Define o tipo de valor que o método 'devolve' (ou 'retorna') para quem o chamou.

- **void**: "Significa que o método não retorna nenhum valor." (Equivalente a procedure em Delphi).
- **int, string, double, bool, etc.**: "Significa que o método retorna um valor daquele tipo." (Equivalente a function em Delphi).

Palavra-chave return: "Para métodos que retornam um valor, a palavra-chave return é usada para enviar esse valor de volta e encerrar o método."

Exemplo:
public static void ExibirMensagem(...) (não retorna nada)
public static int SomarNumeros(...) (retorna um int)

3.3 – Nome do método

Identificação clara
Deve ser descritivo e indicar a tarefa que o método realiza.
Convenção: **PascalCase** (primeira letra de cada palavra em maiúscula).
Ex: CalcularMedia, ExibirBoasVindas, ObterDadosCliente

3.4 – Parâmetros (lista de entrada)

Entradas para o Método
São as informações que o método precisa para realizar sua tarefa.
Definidos entre parênteses () na assinatura do método.
Cada parâmetro tem um Tipo e um Nome.
Se não houver parâmetros, os parênteses ficam vazios: ().

Exemplo:
```
public static void ExibirMensagem(string mensagem) (mensagem é o parâmetro)
public static int SomarNumeros(int num1, int num2) (num1 e num2 são os parâmetros)
```

Analogia:
"Pense em uma máquina de café. Os parâmetros são os 'ingredientes' que você coloca: pó de café, água, açúcar. O método não pode fazer café sem eles!"

3.5 – Corpo do método

A tarefa em si
O código que implementa a lógica da tarefa do método.
Delimitado por chaves {}.

Exemplo:

```
public static void ExibirMensagem(string mensagem)
{
    // ESTE É O CORPO DO MÉTODO
    Console.WriteLine(mensagem);
}
```

4. Métodos sem Parâmetros e sem Retorno (void sem ())

4.1 - Métodos sem Parâmetros e sem Retorno (void)

- Não recebem nenhuma informação externa (parênteses vazios).
- Não devolvem nenhum valor (tipo de retorno void).
- São usados para executar uma sequência de ações.

4.2 – Métodos com Parâmetros

Personalizando ações

- Recebem uma ou mais informações (parâmetros) para personalizar sua execução.
- Continuam sendo void se não retornam nada.

4.3 – Métodos com Retorno

Devolvendo um Valor

- Realizam uma tarefa e 'devolvem' um valor como resultado para quem os chamou.
- O tipo de retorno **não é void**; é o tipo do valor que será devolvido (int, string, bool, etc.).
- Obrigatório usar a palavra-chave **return** para especificar o valor a ser devolvido.

5. Parâmetros Especiais: ref e out

5.1 – Passagem por Valor (Padrão)

- É a forma padrão como os parâmetros são passados para métodos em C#.
- Uma **cópia** do valor do argumento é feita e enviada para o método.
- Qualquer alteração feita no parâmetro dentro do método **não afeta** a variável original fora do método.

5.2 – Passagem por Referência (ref)

- Permite que o método trabalhe diretamente com a **variável original**, não com uma cópia.
- Alterações feitas no parâmetro dentro do método **afetam** a variável original fora do método.
- Precisa da palavra-chave ref **tanto na declaração do parâmetro quanto na chamada do método.**
- A variável original **precisa ser inicializada** antes de ser passada com ref.

Sintaxe :

```
public static void MeuMetodoRef(ref int valor) { /* ... */ }
// Chamada: MeuMetodoRef(ref minhaVariavel);
```

- **Analogia**: É como dar o seu documento original para alguém. Se a pessoa rabiscar, seu documento original será afetado.
- **Uso**: Trocar valores entre variáveis, múltiplos retornos indiretos (menos comum com C# moderno).

5.3 – Passagem de Saída (out)

- Similar ao ref, permite que o método altere uma variável externa.
- Diferença principal: A variável original NÃO precisa ser inicializada antes de ser passada com out. O método é obrigado a atribuir um valor ao parâmetro out antes de retornar.
- Usado quando você quer que um método retorne múltiplos valores (além do valor de retorno principal, se houver).
- Precisa da palavra-chave out tanto na declaração do parâmetro quanto na chamada do método.

Sintaxe :

```
public static void MeuMetodoOut(out int resultado) { /* ... */ }
// Chamada: MeuMetodoOut(out meuResultado);
```

- **Analogia**: É como se você desse uma caixa vazia para alguém e pedisse para a pessoa preenchê-la com algo antes de te devolver. Você não se importa com o que tinha na caixa antes, só com o que ela vai colocar lá dentro.
- **Uso comum**: Métodos que tentam uma operação (ex: conversão) e, além de indicar sucesso/falha, também retornam o resultado da operação.

6. Sobrecarga de Métodos (Method Overloading)

Sobrecarga de Métodos (Overloading)

Um Nome, Várias Versões!

- É a capacidade de ter **múltiplos métodos com o mesmo nome** na mesma classe.
- O C# (e o compilador) sabe qual método chamar com base na **assinatura** do método (número, tipo e ordem dos parâmetros).
- O tipo de retorno sozinho **NÃO** é suficiente para sobrecarregar um método.
- **Uso**: Criar métodos flexíveis que podem lidar com diferentes tipos ou quantidades de dados, mas que realizam a mesma "ideia" de tarefa. Ex: um método CalcularArea que pode receber (lado) para um quadrado ou (largura, altura) para um retângulo.

7. Exercício prático final

Desafio Final: Gerenciador de Operações!
Um Nome, Várias Versões!

Crie um novo projeto de console chamado GerenciadorOperacoes.

Crie os seguintes métodos dentro da classe Program (além do Main):

1. public static void ExibirMenu(): Exibe um menu com opções para o usuário (Ex: "1. Somar", "2. Multiplicar", "S. Sair"). Não recebe nem retorna nada.

2. public static double ObterNumero(string prompt): Recebe um string prompt (ex: "Digite o primeiro número: ") e retorna um double lido do console. Use double.Parse(Console.ReadLine()).

3. public static double SomarNumeros(double n1, double n2): Recebe dois doubles e retorna a soma.

4. public static double MultiplicarNumeros(double n1, double n2): Recebe dois doubles e retorna a multiplicação.

5. No método Main:
   Use um loop do-while para exibir o menu repetidamente até que o usuário escolha "S" (Sair).

   Use um switch para chamar os métodos de operação (SomarNumeros, MultiplicarNumeros) com base na escolha do usuário.

   Para somar/multiplicar, chame o método ObterNumero duas vezes para pegar os dois números do usuário. Exiba o resultado.

8. Dúvidas e próximos passos

8.1 - O que vimos!

Métodos: O Segredo do Código Organizado!

- A importância de **métodos** para organizar e reutilizar código.
- A **estrutura** de um método (modificadores, tipo de retorno, nome, parâmetros, corpo).
- Tipos de métodos na prática (sem/com parâmetros, com/sem retorno).
- **Passagem por valor (ref, out)** e por referência.
- **Sobrecarga de métodos** (mesmo nome, assinaturas diferentes).

8.2 – Próxima aula!

Métodos: O Segredo do Código Organizado!

- **Classes e Objetos (Programação Orientada a Objetos - POO):** Entender como C# é uma linguagem orientada a objetos e como criar nossos próprios tipos de dados complexos.

- Vamos criar nossos próprios "tipos de coisa", como Cliente, Produto, etc.!