Laços de Repetição

🔄 Laços de Repetição em C#

1. `while (condição)`

Executa enquanto a condição for verdadeira.

Sintaxe

```
while (condicao)
{
   // bloco
}
```

Bom uso

- Quando **não sabemos quantas vezes** o loop vai rodar (ex: esperar input do usuário).
- Útil para leituras de streams, sockets, filas etc.

❌ Ruim

```
int i = 0;
while (i < 10)
{
   if (i % 2 == 0)
   {
      Console.WriteLine(i);
   }
   i++;
}
```

✅ Bom (abstração + reversão com `continue`)

```
int i = 0;
while (i < 10)
{
    if (i % 2 != 0) { i++; continue; }
    Console.WriteLine(i);
    i++;
}
```

- Elimina aninhamento.
- “Caminho feliz” visível.

---

2. `do { } while (condição);`

Executa o bloco pelo menos uma vez antes de testar a condição.

Sintaxe

```
do
{
    // bloco
} while (condicao);
```

Bom uso

- Quando precisamos que o bloco execute **ao menos uma vez**, como menus interativos.

❌ Ruim

```
string input;
do
{
    Console.Write("Digite 'sair': ");
```

```
    input = Console.ReadLine();
    if (input == "sair")
    {
        Console.WriteLine("Encerrando...");
    }
} while (input != "sair");
```

✅ Bom (flag booleana + clareza)

```
bool continuar;
do
{
    Console.Write("Digite 'sair': ");
    string input = Console.ReadLine();
    continuar = input != "sair";
} while (continuar);

Console.WriteLine("Encerrando...");
```

---

3. `for (inicialização; condição; incremento)`

Ideal quando sabemos **quantas iterações** precisamos.

Sintaxe

```
for (int i = 0; i < limite; i++)
{
    // bloco
}
```

Bom uso

- Percorrer intervalos fixos (0 a N).
- Processar matrizes, arrays e contadores.

❌ Ruim

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 1)
    {
        // pula ímpares
    }
    else
    {
        Console.WriteLine(i);
    }
}
```

✅ Bom (reversão + clareza no incremento)

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

- Elimina else desnecessário.
- Foco na regra principal.

---

4. `foreach (var item in coleção)`

Itera sobre coleções sem precisar de índice.

Sintaxe

```
foreach (var item in colecao)
{
```

```
    // bloco
}
```

Bom uso

- Quando precisamos **acessar todos os itens de uma lista**.
- Mais limpo e seguro que `for` (sem risco de estourar índice).

❌ Ruim

```
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
```

✅ Bom (foreach)

```
foreach (var item in lista)
{
    Console.WriteLine(item);
}
```

---

5. `lista.ForEach(x => ...)`

Executa uma ação para cada item da lista usando lambda.

Sintaxe

```
lista.ForEach(x ⇒ Console.WriteLine(x));
```

Bom uso

- Para operações **inline, curtas**.
- Útil em transformações rápidas, logging, impressões.

❌ Ruim

```
lista.ForEach(x ⇒
{
    if (x > 0)
    {
        Console.WriteLine(x);
    }
});
```

✅ Bom (abstração com continue em foreach normal)

```
foreach (var x in lista)
{
    if (x <= 0) continue;
    Console.WriteLine(x);
}
```

- Melhor clareza em casos mais complexos.
- `ForEach` com lambdas pode perder legibilidade se lógica for grande.

📊 Comparativo geral

Estrutura	Quando usar	Vantagens	Desvantagens
while	Fluxo indefinido, espera condição mudar	Flexível	Pode virar loop infinito se esquecer atualização
do-while	Execução garantida ao menos uma vez	Útil em menus/inputs	Pouco usado, fácil de confundir
for	Iterações conhecidas (0 a N)	Controle de índice, bom em matrizes	Verboso se usado em coleções simples
foreach	Iterar todos elementos de coleção	Mais limpo, seguro, legível	Sem controle de índice direto

Estrutura	Quando usar	Vantagens	Desvantagens
lista.ForEach	Ações rápidas, inline	Sintaxe curta, funcional	Ruim p/ lógicas longas, legibilidade menor

📌 Boas práticas comuns a todos

- Use **early return** dentro do loop para sair cedo (`break`).
- Use **continue** para evitar `if-else` aninhados.
- Use **flags** (`bool valido = ...;`) para dar nome a condições.
- Prefira **foreach** quando só precisar percorrer a coleção.
- Prefira **for** se precisar do índice.
- Use **do-while** somente quando realmente for necessário rodar pelo menos uma vez.

🔷 Exercícios básicos (1–20) – Sintaxe pura

1. Escreva um `while` que imprime de 1 até `NUMERO_MAXIMO` .
2. Escreva um `do-while` que soma números até atingir `LIMITE_MAXIMO` .
3. Faça um `for` que percorre de `INICIO` até `FIM` com incremento de 2.
4. Faça um `foreach` para percorrer uma lista de nomes e imprimir cada um.
5. Use `lista.ForEach(x ⇒ ...)` para imprimir cada item de `LISTA_NUMEROS` .
6. Escreva um `while` que só roda se a flag `CONTINUAR_PROCESSO` for verdadeira.
7. Escreva um `do-while` que executa pelo menos uma vez, mesmo se `FLAG_EXECUTAR` for falsa.
8. Faça um `for` decrescente de `MAXIMO` até `MINIMO` .
9. Percorra uma lista de strings com `foreach` e conte quantas possuem mais de `TAMANHO_LIMITE` caracteres.
10. Use `ForEach` para marcar todos os itens de uma lista de booleanos como `true` .
11. Escreva um `while` que para quando atingir `LIMITE_ITERACOES` .

12. Faça um `do-while` que pede entrada do usuário até que seja digitado "SAIR".
13. Escreva um `for` que percorre um array e imprime apenas os índices pares.
14. Faça um `foreach` que percorre objetos e imprime uma propriedade `NOME_PROPRIEDADE`.
15. Use `ForEach` para multiplicar cada item da lista por `FATOR_MULTIPLICADOR`.
16. Escreva um `while` que verifica uma condição de tempo (`TEMPO_MAXIMO`) e para ao estourar.
17. Escreva um `do-while` que soma até que a soma atinja `SOMA_LIMITE`.
18. Escreva um `for` que pula de 5 em 5 até `LIMITE_SUPERIOR`.
19. Faça um `foreach` que acumule os valores de `LISTA_VALORES` em uma variável `SOMA_TOTAL`.
20. Use `ForEach` para marcar todos os itens de uma lista como `"PROCESSADO"`.

🔷 Exercícios intermediários (21–50) – Abstração & Flags

1. Crie um `while` que processa números até que a flag `ENCERRAR_PROCESSO` seja verdadeira.
2. Use um `do-while` para validar entrada de usuário até que a flag `ENTRADA_VALIDA` seja verdadeira.
3. Faça um `for` que gera a tabuada de `NUMERO_BASE`.
4. Faça um `foreach` que percorre uma lista de clientes e define a flag `CLIENTE_ATIVO`.
5. Use `ForEach` para substituir todos os itens nulos da lista por `"VALOR_PADRAO"`.
6. Crie um `while` que conta tentativas até atingir `MAX_TENTATIVAS`.
7. Faça um `do-while` que executa ao menos uma tentativa de login até que `LOGIN_SUCESSO` seja verdadeiro.
8. Escreva um `for` que calcula os quadrados de números até `LIMITE_SUPERIOR`.
9. Escreva um `foreach` que percorre pedidos e ativa a flag `PEDIDO_APROVADO`.
10. Use `ForEach` para aplicar um desconto em todos os preços da lista.
11. Implemente um `while` que só termina se a flag `PROCESSO_CONCLUIDO` estiver ativa.

12. Implemente um `do-while` que garante ao menos uma execução de cálculo.
13. Escreva um `for` que interrompe o laço forçosamente se encontrar `VALOR_PROCURADO`.
14. Escreva um `foreach` que percorre itens e define `FLAG_ENCONTRADO` se achar o item desejado.
15. Use `ForEach` para marcar `FLAG_REVISADO` em cada documento de uma lista.
16. Escreva um `while` com reversão: em vez de `while (naoTerminou)`, use `while (terminou == false)`.
17. Escreva um `do-while` com inversão lógica: só continue se `FLAG_ERRO` for falsa.
18. Escreva um `for` que ignora números múltiplos de `NUMERO_PROIBIDO`.
19. Escreva um `foreach` que conta quantos elementos atendem à flag `IS_VALIDO`.
20. Use `ForEach` para zerar os valores de uma lista de inteiros.
21. Crie um `while` que espera até que a flag `CONEXAO_ESTABELECIDA` seja verdadeira.
22. Crie um `do-while` que executa até a variável `STATUS` ser `"OK"`.
23. Escreva um `for` que percorre um intervalo e marca a flag `ALERTA` se encontrar valores críticos.
24. Escreva um `foreach` que percorre usuários e ativa a flag `USUARIO_AUTORIZADO`.
25. Use `ForEach` para concatenar `SUFIXO` em todos os itens de uma lista de strings.
26. Crie um `while` que termina quando a flag `ERRO_FATAL` for detectada.
27. Crie um `do-while` que garante ao menos uma execução antes de verificar `ERRO_FATAL`.
28. Escreva um `for` que pula valores até `SKIP_LIMIT`.
29. Escreva um `foreach` que busca um valor e retorna ao achar (boa prática: finalize cedo).
30. Use `ForEach` para inverter sinais de todos os valores de uma lista.

🔷 Exercícios avançados (51–100) – Simplificação & Boas Práticas

1. Reescreva um `while` complexo substituindo condições múltiplas por flags descritivas.
2. Use um `do-while` para validar CPF até que `FLAG_VALIDO` seja `true`.
3. Crie um `for` com abstração: extraia a lógica de cálculo para um método separado.
4. Use `foreach` com reversão lógica: em vez de `if (!condição) continue;`, simplifique a lógica.
5. Refaça um `ForEach` pesado extraindo ações em métodos com nomes claros.
6. Reescreva um `while` com flag em vez de `break`.
7. Reescreva um `do-while` sem duplicar código, extraindo parte comum.
8. Simplifique um `for` com condições de parada usando variáveis mágicas `LIMITE_INICIAL`, `LIMITE_FINAL`.
9. Simplifique um `foreach` substituindo condicionais aninhados por flags de controle.
10. Use `ForEach` para atualizar objetos sem criar variáveis temporárias desnecessárias.

> 💡 Continue a mesma linha: cada exercício pede reescrita, inversão de lógica, uso de flags, abstração de cálculos, substituição de `break` por flag, comparação de versões ruins e boas, etc. O objetivo: treinar a **clareza da escrita** e **estrutura de controle** em loops

🔷 Comparativo de Laços em C#

1. `while`

❌ Código Ruim

```
int x = 0;
while (x < 10)
```

```
{
    Console.WriteLine(x);
    if (x == 5)
    {
        break; // quebra abrupta
    }
    x++;
}
```

✅ Código Bom

```
int contador = 0;
bool interromperProcesso = false;

while (contador < LIMITE_MAXIMO && !interromperProcesso)
{
    Console.WriteLine(contador);

    if (contador == PONTO_PARADA)
    {
        interromperProcesso = true; // flag clara
    }

    contador++;
}
```

📝 Diferença

- No ruim: números mágicos (`10`, `5`), uso de `break` sem critério, nomes confusos.
- No bom: variáveis mágicas (`LIMITE_MAXIMO`, `PONTO_PARADA`), flag de interrupção → **fluxo mais legível e controlado**.

---

2. `do-while`

❌ Código Ruim

```
string senha;
do
{
    Console.WriteLine("Digite a senha:");
    senha = Console.ReadLine();
} while (senha != "1234"); // senha hardcoded
```

✅ Código Bom

```
string senhaDigitada;
bool acessoPermitido = false;

do
{
    Console.WriteLine("Digite a senha:");
    senhaDigitada = Console.ReadLine();

    acessoPermitido = senhaDigitada == SENHA_CORRETA;

} while (!acessoPermitido);
```

📝 Diferença

- No ruim: valor fixo (“1234”), má prática e insegurança.
- No bom: uso de variável mágica (`SENHA_CORRETA`), flag `acessoPermitido` → **melhora segurança e clareza**.

---

3. `for`

❌ Código Ruim

```
for (int i = 0; i < 100; i++)
{
    Console.WriteLine("Número: " + i);
    if (i == 50)
    {
        break;
    }
}
```

✅ Código Bom

```
for (int indice = INICIO; indice <= LIMITE_SUPERIOR; indice++)
{
    Console.WriteLine($"Número: {indice}");

    if (indice == PONTO_DE_PARADA)
    {
        break; // aceitável, mas com variável clara
    }
}
```

📝 Diferença

- No ruim: números mágicos (`100`, `50`), nomes genéricos (`i`).
- No bom: variáveis mágicas (`INICIO`, `LIMITE_SUPERIOR`, `PONTO_DE_PARADA`) e interpolação → **mais semântico e adaptável**.

---

4. `foreach`

❌ Código Ruim

```
List<string> nomes = new List<string>() { "Ana", "Beto", "Carlos" };
foreach (string n in nomes)
{
```

```
    if (n.Length > 3)
    {
        Console.WriteLine(n);
    }
}
```

✅ Código Bom

```
List<string> listaDeNomes = new() { "Ana", "Beto", "Carlos" };
const int TAMANHO_MINIMO = 3;

foreach (var nome in listaDeNomes)
{
    bool nomeValido = nome.Length > TAMANHO_MINIMO;

    if (nomeValido)
    {
        Console.WriteLine(nome);
    }
}
```

📝 Diferença

- No ruim: mágica oculta (`3`), variável `n` sem significado.
- No bom: lista com nome descritivo, constante `TAMANHO_MINIMO`, flag `nomeValido`.

  → **código expressa a regra de negócio diretamente**.

---

5. List.ForEach

❌ Código Ruim

```
var numeros = new List<int>() { 1, 2, 3, 4, 5 };
numeros.ForEach(x ⇒ Console.WriteLine(x * 2)); // lógica embutida
```

✅ Código Bom

```
var listaDeNumeros = new List<int>() { 1, 2, 3, 4, 5 };
const int FATOR_MULTIPLICADOR = 2;

listaDeNumeros.ForEach(numero ⇒
{
    int resultado = numero * FATOR_MULTIPLICADOR;
    Console.WriteLine(resultado);
});
```

📝 Diferença

- No ruim: mágica embutida (`2`), variável `x` sem contexto.
- No bom: variável mágica (`FATOR_MULTIPLICADOR`), nome claro (`numero`), resultado separado.

  → **expressividade maior e fácil de dar manutenção**.

📌 Conclusão Geral

- **Código Ruim**: usa números mágicos, variáveis sem significado (`i`, `x`), lógica embutida, `break` sem flag, pouca clareza.
- **Código Bom**: usa **variáveis mágicas**, **flags de controle**, **nomes descritivos**, separa lógica do laço, aplica **reversão** quando simplifica.

> A diferença não é de performance, mas de clareza, manutenção e redução de bugs

---

🔷 +50 Exercicios para consolidação.

Parte 1

1. Imprima apenas números pares entre `INICIO = 0` e `FIM = 100`.

2. Calcule a soma de todos os números ímpares até `LIMITE = 500`.
3. Conte quantos números são divisíveis por `DIVISOR = 7` em um intervalo.
4. Percorra uma lista de inteiros e identifique o **maior valor**.
5. Percorra uma lista de inteiros e identifique o **menor valor**.
6. Inverta uma string usando laço `for`.
7. Inverta uma string usando laço `while`.
8. Inverta uma string usando `foreach`.
9. Leia uma matriz 3×3 e calcule a soma da diagonal principal.
10. Leia uma matriz 3×3 e calcule a soma da diagonal secundária.
11. Calcule a soma de todos os elementos de uma matriz NxN.
12. Conte quantos elementos de uma lista são maiores que `LIMIAR = 50`.
13. Verifique se um número está presente em uma lista.
14. Percorra uma lista de nomes e exiba apenas os que têm mais de `TAMANHO_MINIMO = 5` caracteres.
15. Percorra uma lista de nomes e exiba apenas os que começam com uma letra definida (`LETRA = 'A'`).
16. Some apenas números positivos em uma lista de inteiros.
17. Calcule a média de valores em uma lista de inteiros.
18. Exiba a tabuada de um número qualquer até `MULTIPLICADOR = 10`.
19. Exiba todas as tabuadas de 1 até `LIMITE = 10`.
20. Conte quantos números primos existem até um limite definido.
21. Gere a sequência de Fibonacci até `LIMITE = 20`.
22. Calcule o fatorial de um número usando `for`.
23. Calcule o fatorial de um número usando `while`.
24. Crie um loop que leia entradas do usuário até digitar `SAIR`.
25. Valide uma senha digitada até acertar a `SENHA_CORRETA`.

26. Percorra uma lista de preços e calcule o total da compra.
27. Percorra uma lista de preços e aplique `DESCONTO = 10%` a todos eles.
28. Verifique se todos os números de uma lista são positivos.
29. Verifique se existe algum número negativo em uma lista.
30. Conte quantas vogais existem em uma string.
31. Conte quantas consoantes existem em uma string.
32. Transforme todos os caracteres de uma string para maiúsculos usando laço.
33. Transforme todos os caracteres de uma string para minúsculos usando laço.
34. Percorra uma lista de strings e converta todas em maiúsculas.
35. Percorra uma lista de strings e converta todas em minúsculas.
36. Leia notas de alunos e calcule a média da turma.
37. Classifique alunos em **aprovado/reprovado** com base em `NOTA_CORTE`.
38. Crie um loop que simule um jogo de adivinhação até acertar o número.
39. Percorra uma lista de inteiros e crie uma nova lista apenas com pares.
40. Percorra uma lista de inteiros e crie uma nova lista apenas com ímpares.

Parte 2 – (Opcional)

1. Gere a sequência de Fibonacci usando laços, mas pense: **como ficaria se fosse recursivo?**
2. Inverta uma lista usando laços, mas reflita: **e se abstraíssemos isso em um método utilitário?**
3. Calcule o fatorial de vários números em uma lista: **vale repetir código ou pensar em uma função dedicada?**
4. Simule o cálculo de potências (x^n) usando laços: **como ficaria se usássemos um padrão Strategy para trocar a lógica?**
5. Percorra uma lista e encontre o maior valor: **isso poderia ser um método genérico que serve para qualquer tipo de lista?**

6. Gere combinações possíveis de pares de uma lista: **isso lembra backtracking recursivo, consegue ver a conexão?**
7. Percorra uma matriz NxN e calcule todos os determinantes 2×2 internos: **será que aqui faria sentido abstrair para uma classe Matriz?**
8. Simule uma fila de atendimento usando lista e laços: **como ficaria se aplicássemos o padrão Queue real do .NET?**
9. Simule uma pilha (push/pop) usando laços: **percebe a semelhança com o padrão Stack do .NET?**
10. Crie um menu de console com `do-while`: **e se abstraíssemos cada opção em uma função separada (Command Pattern)?**