Coleções e Tuplas

🔷 Tipos de Coleções em .NET

1. **Array (`T[]`)**

- **Definição**: Estrutura **fixa** de elementos, indexados a partir de `0`.
- **Vantagens**:
  - Acesso rápido por índice (`O(1)`).
  - Mais leve em memória do que estruturas dinâmicas.
  - Bom para cenários em que o **tamanho não muda**.
- **Desvantagens**:
  - Tamanho imutável.
  - Inserções e remoções caras.
- **Quando usar**:
  - Quando o número de elementos é **conhecido e fixo**.
  - Ex.: armazenar dias da semana, constantes pré-definidas.

```
int[] numeros = { 1, 2, 3, 4, 5 };
Console.WriteLine(numeros[2]); // 3
```

---

2. **List (`List<T>`)**

- **Definição**: Lista **dinâmica** baseada em array, permite adicionar e remover itens.
- **Vantagens**:
  - Cresce e diminui dinamicamente.
  - Métodos prontos (`Add`, `Remove`, `Contains`, `Sort`).

    - Mantém **ordem de inserção**.
- **Desvantagens**:
    - Inserções no meio podem ser custosas (`O(n)`).
    - Mais consumo de memória comparado ao array.
- **Quando usar**:
    - Quando precisa de **flexibilidade** no número de elementos.
    - Ex.: lista de usuários conectados, carrinho de compras.

```
var lista = new List<string> { "Ana", "Beto" };
lista.Add("Carlos");
Console.WriteLine(lista[1]); // Beto
```

---

3. Dictionary (`Dictionary<TKey, TValue>`)

- **Definição**: Estrutura de chave-valor. A chave deve ser **única**.
- **Vantagens**:
    - Busca por chave muito rápida (`O(1)` em média).
    - Ideal para mapear relações (ex.: ID → Nome).
- **Desvantagens**:
    - Não mantém ordem de inserção (até .NET Core 3.1; a partir do .NET 5 mantém).
    - Consome mais memória.
- **Quando usar**:
    - Quando precisa **mapear e buscar valores pela chave**.
    - Ex.: cadastro de clientes por CPF, cache de dados.

```
var dicionario = new Dictionary<int, string>();
dicionario[1] = "João";
```

```
dicionario[2] = "Maria";
Console.WriteLine(dicionario[1]); // João
```

4. IEnumerable (`IEnumerable<T>`)

- **Definição**: Interface que representa uma **sequência enumerável**.
- **Vantagens**:
  - Base para **LINQ**.
  - Lazy evaluation (avaliação sob demanda).
  - Permite abstração sobre qualquer coleção (array, list, etc.).
- **Desvantagens**:
  - Apenas iteração, sem acesso direto por índice.
  - Pode não permitir múltiplas iterações (ex.: streams).
- **Quando usar**:
  - Quando quer **iterar** sobre qualquer coleção sem se preocupar com a implementação concreta.
  - Ex.: consultas LINQ que retornam dados, pipelines de leitura.

```
IEnumerable<int> numeros = Enumerable.Range(1, 5);
foreach (var n in numeros)
{
   Console.WriteLine(n); // 1, 2, 3, 4, 5
}
```

📊 Comparativo Resumido

Tipo	Estrutura	Vantagens	Desvantagens	Melhor Uso
Array	Tamanho fixo, indexado	Rápido, leve	Imutável, pouco flexível	Dados fixos e conhecidos

Tipo	Estrutura	Vantagens	Desvantagens	Melhor Uso
List	Dinâmica, genérica	Flexível, muitos métodos	Inserção/remover no meio caro	Coleções mutáveis e dinâmicas
Dictionary	Mapeamento chave-valor	Busca rápida, chave única	Mais memória, ordem só recente	Relacionar dados (ex.: ID → Objeto)
IEnumerable	Interface de iteração	Abstração, LINQ, lazy	Só leitura sequencial	Iteração genérica, consultas LINQ

Comportamentos

🔷 1. Array

- Estrutura fixa (tamanho imutável após criação).
- Acesso rápido via índice.
- Ideal para cenários onde você sabe a quantidade de elementos de antemão.

Loops:

✅ `for`

```
int[] arr = { 1, 2, 3, 4 };
for (int i = 0; i < arr.Length; i++)
    Console.WriteLine(arr[i]);
```

➡️ Vantajoso: índice direto, bom controle.

⚠️ Desvantagem: fácil errar limites (`i <= arr.Length` → exception).

✅ `foreach`

```
foreach (var n in arr)
    Console.WriteLine(n);
```

➡️ Simples, limpo, evita erros de índice.

⚠️ Não dá para alterar direto os elementos.

❌ `while` / `do-while`

```
int i = 0;
while (i < arr.Length)
    Console.WriteLine(arr[i++]);
```

➡️ Funciona, mas verboso demais para Arrays.

🔷 2. List<T>

- Estrutura dinâmica (cresce/diminui).
- Muito usada em aplicações modernas.
- Possui métodos (`Add`, `Remove`, `Find`, `ForEach`).

Loops:

✅ `foreach`

```
List<int> list = new() { 1, 2, 3 };
foreach (var n in list)
    Console.WriteLine(n);
```

➡️ Natural, expressivo.

⚠️ Não permite remover dentro do loop sem dar erro (modificação em tempo de iteração).

✅ `for`

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

➡️ Bom quando precisa remover/alterar.

⚠️ Performance ligeiramente pior que Array, mas aceitável.

✅ `.ForEach()` **(lambda)**

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

➡️ Limpo, conciso, combina com LINQ.

⚠️ Menos flexível em cenários mais complexos (break/continue não funciona fácil).

🔷 3. Dictionary<TKey, TValue>

- Estrutura **chave-valor**.
- Lookup rápido (ideal para buscas).
- Mais poderoso que List, mas mais caro em memória.

Loops:

✅ `foreach` **(KeyValuePair)**

```
Dictionary<string, int> dict = new() { ["A"] = 1, ["B"] = 2 };
foreach (var kvp in dict)
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
```

➡️ O mais natural, fácil.

⚠️ Ordem de iteração não é garantida.

✅ `for` (quase nunca usado)

```
var keys = dict.Keys.ToList();
for (int i = 0; i < keys.Count; i++)
    Console.WriteLine($"{keys[i]}: {dict[keys[i]]}");
```

➡️ Usável, mas indireto e pesado.

⚠️ Desnecessário em 99% dos casos.

✅ **LINQ / abstração**

```
dict.ToList().ForEach(x ⇒ Console.WriteLine($"{x.Key} ⇒ {x.Value}"));
```

➡️ Bom para transformar em pipeline.

⚠️ Cuidado com overhead.

🔷 4. IEnumerable<T>

- É **interface de iteração** (não é uma coleção concreta).
- Usada em LINQ, streaming, consultas a BD.
- Só garante: você pode **iterar**.

Loops:

✅ `foreach`

```
IEnumerable<int> nums = Enumerable.Range(1, 5);
foreach (var n in nums)
    Console.WriteLine(n);
```

➡️ Ideal e natural.

⚠️ Não dá para acessar por índice (a menos que converta para List/Array).

❌ `for`

```
// Não funciona diretamente, IEnumerable não expõe Count nem índice.
```

⚠️ **while** pode ser útil em casos de streaming:

```
IEnumerator<int> enumerator = nums.GetEnumerator();
while (enumerator.MoveNext())
    Console.WriteLine(enumerator.Current);
```

➡️ Baixo nível, usado em implementações customizadas de iteradores.

⚠️ Verboso demais para uso normal.

📊 Resumo comparativo

Estrutura	Melhor loop	Por quê
Array	`for`, `foreach`	Índice rápido (for), ou leitura simples (foreach).
List<T>	`foreach`, `for`	Dinâmica; foreach é limpo, for útil quando precisa alterar/remover.
Dictionary	`foreach`	Iteração natural com `KeyValuePair`. For só em casos raros.
IEnumerable	`foreach`	É a alma do IEnumerable; qualquer outro é gambiarra.

⚡ Conclusões rápidas:

- **Arrays** → bons para performance bruta e cenários fixos.
- **Lists** → flexibilidade + fácil de usar no dia a dia.
- **Dictionaries** → lookup por chave, mas não use se precisar de ordem fixa.
- **IEnumerable** → ótimo para abstrações, LINQ e streams, mas não para manipulação direta.

🟦 Arrays (25 exercícios)

1. Iterar um array de inteiros e imprimir apenas os pares.
2. Criar um método que receba um array de strings e retorne apenas as que têm mais de 5 caracteres.
3. Somar todos os elementos de um array sem usar variáveis mágicas no loop.
4. Inverter a ordem dos elementos de um array usando `for`.
5. Inverter um array sem usar índice (abstrair para `Array.Reverse`).
6. Encontrar o menor número em um array (comparar `for` vs `foreach`).

7. Substituir todos os valores negativos por zero em um array.
8. Criar uma função que recebe um array e retorna um novo com apenas os elementos únicos.
9. Implementar uma busca sequencial em array.
10. Implementar uma busca binária em array já ordenado.
11. Concatenar dois arrays em um novo array.
12. Calcular a média dos elementos de um array.
13. Contar quantos elementos repetidos existem em um array.
14. Usar `foreach` para montar uma string com todos os valores separados por vírgula.
15. Multiplicar todos os elementos de um array por 2 usando abstração (`Select`).
16. Validar se todos os elementos de um array são positivos.
17. Validar se existe pelo menos um número primo em um array.
18. Usar `while` para iterar até encontrar um valor específico.
19. Criar um algoritmo para rotacionar os elementos de um array (ex: [1,2,3] → [2,3,1]).
20. Fazer a soma cumulativa dos elementos em um novo array.
21. Ordenar um array manualmente com `for` (bubble sort).
22. Comparar a ordenação manual com `Array.Sort`.
23. Criar um método que receba um array e devolva um dicionário com a contagem de cada elemento.
24. Implementar uma flag para parar a iteração quando encontrar determinado valor.
25. Comparar performance entre `for` e `foreach` em array com 1 milhão de itens.

🟩 List<T> (25 exercícios)

1. Criar uma lista de strings e imprimir apenas as que começam com vogal.

2. Iterar com `for` e remover os elementos maiores que 10.
3. Iterar com `foreach` e mostrar por que não é seguro remover dentro dele.
4. Usar `.ForEach()` para aplicar uma ação em cada item da lista.
5. Criar uma função que receba `List<int>` e retorne os números em ordem crescente.
6. Validar se todos os elementos são únicos.
7. Encontrar o primeiro elemento que atende uma condição (`Find`).
8. Encontrar todos os elementos que atendem a uma condição (`FindAll`).
9. Usar `Any` para verificar se existe um valor maior que 100.
10. Usar `All` para verificar se todos são maiores que 0.
11. Criar um método para converter uma lista em string formatada.
12. Rotacionar elementos de uma lista.
13. Usar `while` para esvaziar a lista removendo item por item.
14. Criar uma função que recebe `List<string>` e devolve um `Dictionary<string, int>` com a frequência.
15. Usar `LINQ` para transformar uma lista de inteiros em seus quadrados.
16. Implementar uma flag para parar ao encontrar um valor duplicado.
17. Comparar `foreach` com `.ForEach()` em termos de clareza de código.
18. Ordenar a lista usando `Sort`.
19. Ordenar a lista usando LINQ (`OrderBy`).
20. Filtrar itens que contenham substring específica.
21. Usar `RemoveAll` com predicate.
22. Iterar com `for` e acessar índice e valor.
23. Implementar um método de paginação (retorna “n” itens por página).
24. Testar `ConvertAll` para mudar tipo de lista (`List<int>` → `List<string>`).
25. Comparar performance entre `for`, `foreach` e `.ForEach()` em lista grande.

🟨 Dictionary<TKey, TValue> (25 exercícios)

1. Criar um dicionário de alunos (nome → nota) e imprimir todos.
2. Buscar um aluno específico usando chave.
3. Validar se uma chave existe antes de acessar.
4. Iterar em `foreach` com `KeyValuePair`.
5. Iterar apenas pelas chaves (`Keys`).
6. Iterar apenas pelos valores (`Values`).
7. Usar `TryGetValue` para evitar exceções.
8. Criar uma função que retorna o maior valor dentro de um dicionário.
9. Criar uma função que inverte chave e valor em um novo dicionário.
10. Contar quantos alunos têm nota acima de 7.
11. Atualizar valores existentes em um loop.
12. Criar uma flag para parar quando achar determinada nota.
13. Converter o dicionário em uma lista de strings formatadas.
14. Usar LINQ para ordenar dicionário por valor.
15. Usar LINQ para ordenar por chave.
16. Criar um método que merge dois dicionários (resolvendo conflitos de chave).
17. Iterar em `for` pegando `Keys.ToList()`.
18. Mostrar porque `for` não é natural para dicionários.
19. Criar um método para retornar todas as chaves que possuem valor duplicado.
20. Iterar com `foreach` e transformar em JSON (string simulada).
21. Usar `GroupBy` em valores do dicionário (ex: agrupar notas iguais).
22. Criar dicionário aninhado e iterar (ex: aluno → matérias → nota).
23. Criar flag para parar a iteração quando encontrar chave específica.

24. Comparar iteração de `Dictionary` com `SortedDictionary`.
25. Comparar performance entre `foreach` e LINQ em dicionário grande.

🟪 IEnumerable<T> (25 exercícios)

1. Iterar sobre um `Enumerable.Range(1,10)` com `foreach`.
2. Iterar sobre `IEnumerable` com `while` e `GetEnumerator`.
3. Criar uma função que retorna `IEnumerable<int>` (yield return).
4. Mostrar preguiça (lazy) em `IEnumerable` com LINQ.
5. Filtrar `IEnumerable` usando `Where`.
6. Projetar em novo tipo usando `Select`.
7. Validar se todos os elementos são positivos com `All`.
8. Verificar se existe um múltiplo de 7 com `Any`.
9. Somar todos os elementos (`Sum`).
10. Usar `Take` para pegar apenas os 5 primeiros.
11. Usar `Skip` para pular os 5 primeiros.
12. Concatenar dois `IEnumerable` com `Concat`.
13. Mostrar diferença entre `ToList()` e `IEnumerable` em execução.
14. Criar uma função recursiva que retorna `IEnumerable`.
15. Criar flag para parar iteração em streaming (simular `while`).
16. Criar `SelectMany` para explodir listas internas.
17. Usar `Distinct` para remover duplicados.
18. Criar `GroupBy` em `IEnumerable` e iterar.
19. Ordenar com `OrderBy`.
20. Comparar `foreach` direto vs materializar com `ToList()`.
21. Iterar com `foreach` sobre `Directory.EnumerateFiles()`.

22. Criar pipeline que só executa quando chamar `ToList()`.
23. Demonstrar que não há índice em `IEnumerable`.
24. Converter `IEnumerable` em array e iterar com `for`.
25. Comparar performance de `IEnumerable` vs `List` em loop grande.

🔷 O que é `yield`

`yield` é uma palavra-chave que permite criar **iteradores** de forma **mais simples e eficiente**, sem precisar criar manualmente coleções intermediárias ou `IEnumerator`.

Existem dois usos principais:

1. `yield return <valor>` → retorna cada elemento da sequência, **um de cada vez**.
2. `yield break` → encerra a sequência antecipadamente.

O compilador cria automaticamente o estado interno necessário para continuar a iteração na próxima chamada.

🔷 Por que usar `yield`

1. **Lazy Evaluation** (avaliação preguiçosa):
   - Elementos só são gerados quando necessários.
   - Não precisa carregar toda a coleção na memória.
2. **Economia de memória**:
   - Em vez de criar `List<T>` grande, você gera os elementos sob demanda.
3. **Fluxo controlado**:
   - Permite pausar e continuar a execução do loop interno sem complicações.
4. **Abstração limpa**:
   - Você encapsula a lógica de geração de sequência sem expor detalhes internos.

🔷 Quando utilizar `yield`

- Sequências **grandes ou infinitas**.
- Quando não quer **materializar uma lista inteira** na memória.
- Para criar **pipelines de dados** ou iteradores customizados.
- Quando a função precisa **iterar sob demanda** com condições complexas.

Evite quando:

- A coleção final precisa ser **acessada aleatoriamente** por índice.
- O consumo de cada elemento precisa ser repetido várias vezes.

🔷 Exemplos práticos

1. Iterador simples

```
IEnumerable<int> GetEvenNumbers(int max)
{
    for (int i = 0; i <= max; i++)
        if (i % 2 == 0)
            yield return i;
}

// Uso
foreach (var n in GetEvenNumbers(10))
    Console.WriteLine(n); // 0,2,4,6,8,10
```

2. Encerrando antecipadamente

```
IEnumerable<int> GetNumbers(int max)
{
    for (int i = 0; i <= max; i++)
    {
        if (i > 5) yield break; // interrompe a sequência
```

```
        yield return i;
    }
}
```

3. Sequência infinita

```
IEnumerable<int> Fibonacci()
{
    int a = 0, b = 1;
    while (true)
    {
        yield return a;
        int temp = a;
        a = b;
        b = temp + b;
    }
}

// Uso: pegar só os 10 primeiros
foreach (var n in Fibonacci().Take(10))
    Console.WriteLine(n);
```

🔷 Diferença prática vs lista normal

```
// Sem yield
List<int> GenerateNumbers(int max)
{
    var result = new List<int>();
    for (int i = 0; i <= max; i++)
        result.Add(i);
    return result;
}
```

```
// Com yield
IEnumerable<int> GenerateNumbersLazy(int max)
{
    for (int i = 0; i <= max; i++)
        yield return i;
}
```

- `List<int>` cria todos os elementos na memória.
- `yield` cria **sob demanda**, mais eficiente para grandes volumes.

🔷 Resumo

Aspecto	Lista/Array	Yield
Memória	Alta para grande listas	Baixa, gera sob demanda
Avaliação	Eager (imediata)	Lazy (preguiçosa)
Reutilização	Fácil (acesso por índice)	Apenas sequência iterável
Complexidade	Menos abstrata	Limpa para iteradores complexos
Uso recomendado	Coleções pequenas/fixas	Sequências grandes, pipelines, streams

🔷 1. Array

Código bom

- Usando `for` ou `foreach` corretamente.
- Evita variáveis mágicas, mantém índice claro.

```
int[] numeros = { 1, 2, 3, 4, 5 };

// Usando for
for (int i = 0; i < numeros.Length; i++)
{
    Console.WriteLine(numeros[i]);
}
```

```
// Usando foreach
foreach (var n in numeros)
{
    Console.WriteLine(n);
}
```

Código ruim

- Acessando índice fora do limite ou usando números mágicos.

```
int[] numeros = { 1, 2, 3, 4, 5 };
for (int i = 0; i <= 5; i++) // Erro: índice 5 não existe
{
    Console.WriteLine(numeros[i]);
}
```

Resumo:

- `for` bom quando precisa de índice.
- `foreach` bom para iterar simples.
- Ruim: usar constantes fixas que quebram se o array mudar.

🔷 2. List<T>

Código bom

- Usando `foreach` ou `.ForEach()` e abstração de métodos.

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

// Foreach
foreach (var nome in nomes)
    Console.WriteLine(nome);
```

```
// Lambda ForEach
nomes.ForEach(nome ⇒ Console.WriteLine(nome));
```

Código ruim

- Modificando a lista dentro do `foreach` diretamente.

```
List<int> numeros = new() { 1, 2, 3, 4 };
foreach (var n in numeros)
{
    if (n % 2 == 0)
        numeros.Remove(n); // Erro: coleção modificada durante iteração
}
```

Resumo:

- `for` útil se for alterar/remover itens.
- `foreach` seguro apenas para leitura.
- `.ForEach()` conciso para ações simples.

🔷 3. Dictionary<TKey, TValue>

Código bom

- Iterando com `KeyValuePair`.
- Evitando acesso direto sem checagem.

```
Dictionary<int, string> alunos = new() { [1] = "Ana", [2] = "Beto" };
foreach (var kvp in alunos)
    Console.WriteLine($"{kvp.Key} ⇒ {kvp.Value}");

if (alunos.TryGetValue(2, out var nome))
    Console.WriteLine(nome);
```

Código ruim

- Acessando chave inexistente sem checagem.

```
Dictionary<int, string> alunos = new() { [1] = "Ana", [2] = "Beto" };
Console.WriteLine(alunos[5]); // Erro: KeyNotFoundException
```

Resumo:

- `foreach` é natural.
- Evite `for` direto.
- Sempre verificar se a chave existe (`TryGetValue`).

🔷 4. IEnumerable<T>

Código bom

- Usando `foreach` e LINQ.
- Avaliação preguiçosa, sem materializar coleção.

```
IEnumerable<int> numeros = Enumerable.Range(1, 10)
                           .Where(n ⇒ n % 2 == 0);
foreach (var n in numeros)
    Console.WriteLine(n);
```

Código ruim

- Tentando acessar índice direto ou iterar várias vezes sem materializar, causando repetição de execução.

```
IEnumerable<int> numeros = Enumerable.Range(1, 5)
                           .Select(n ⇒ {
                              Console.WriteLine("Gerando " + n);
                              return n;
                           });
```

```
Console.WriteLine(numeros.ElementAt(2)); // Funciona, mas força iteração co
mpleta
Console.WriteLine(numeros.ElementAt(2)); // Executa tudo de novo
```

Resumo:

- Use `foreach` para iteração direta.
- Materialize (`ToList()`) se precisar múltiplas iterações.
- LINQ + IEnumerable = ótimo para pipelines.

🔷 5. yield

Código bom

- Cria iteradores sob demanda, memória eficiente.

```
IEnumerable<int> Fibonacci(int max)
{
    int a = 0, b = 1;
    for (int i = 0; i < max; i++)
    {
        yield return a;
        int temp = a;
        a = b;
        b = temp + b;
    }
}

// Uso
foreach (var n in Fibonacci(10))
    Console.WriteLine(n);
```

Código ruim

- Criar lista completa sem necessidade.

```
List<int> Fibonacci(int max)
{
    List<int> resultado = new();
    int a = 0, b = 1;
    for (int i = 0; i < max; i++)
    {
        resultado.Add(a);
        int temp = a;
        a = b;
        b = temp + b;
    }
    return resultado;
}
```

Resumo:

- `yield` = eficiente, lazy, ideal para sequências grandes ou infinitas.
- Evite criar listas desnecessárias.

📌 Resumo geral comparativo de laços e coleções

Estrutura	Melhor loop	Boas práticas	Erros comuns
Array	for / foreach	Índices claros, evitar variáveis mágicas	Índices fora do limite
List<T>	foreach / ForEach	Foreach para leitura, for se alterar	Modificar coleção no foreach
Dictionary	foreach	Usar TryGetValue, KeyValuePair	Acessar chave inexistente
IEnumerable	foreach / LINQ	Lazy evaluation, materializar se necessário	Acesso por índice direto
yield	foreach	Sequências grandes, lazy	Criar listas desnecessárias

Erros Comuns e como evita-los

🔷 1. Array

Erro comum	Por que acontece	Como evitar / corrigir
Índice fora do limite (`IndexOutOfRangeException`)	Usar `i <= array.Length` ou acessar índices inválidos	Sempre usar `i < array.Length`; evite números mágicos; prefira `foreach` se não precisa do índice
Acesso a array vazio	Array não inicializado ou tamanho 0	Verificar `array.Length > 0` antes de acessar
Modificação indevida durante iteração	Tentativa de alterar tamanho em loops que assumem tamanho fixo	Arrays têm tamanho fixo; não tente `Add/Remove`

🔷 2. List<T>

Erro comum	Por que acontece	Como evitar / corrigir
Modificar a lista dentro de `foreach`	`foreach` não permite alterações diretas	Use `for` reverso ou `.RemoveAll()`
Acesso a índice inválido (`ArgumentOutOfRangeException`)	Índice maior que `Count-1`	Sempre usar `i < list.Count`
Comparações com valores mágicos	Hard-coded numbers no loop	Usar propriedades `.Count` e constantes nomeadas
Uso excessivo de `.ForEach()` com lógica complexa	Não suporta `break` / `continue`	Prefira `foreach` ou `for` quando precisar controlar fluxo

🔷 3. Dictionary<TKey, TValue>

Erro comum	Por que acontece	Como evitar / corrigir
Acesso a chave inexistente (`KeyNotFoundException`)	Usar `dict[chave]` sem verificar	Usar `TryGetValue()` ou `ContainsKey()`
Iteração e remoção simultânea	Modificação durante `foreach`	Iterar sobre `Keys.ToList()` ou coletar chaves a remover primeiro
Chaves duplicadas	Tentativa de `Add` com chave já existente	Garantir unicidade ou usar `dict[key] = value` para atualizar

🔷 4. IEnumerable<T>

Erro comum	Por que acontece	Como evitar / corrigir
Tentativa de acessar índice (`IEnumerable` não tem índice)	Usar `ElementAt` sem materializar	Converter para `ToList()` ou `ToArray()` se precisa de índice
Iteração múltipla gera execução repetida	Lazy evaluation	Materializar em lista ou array se precisar iterar várias vezes
Modificação da coleção base durante iteração	Enumerador se mantém fixo	Evitar modificar coleção usada ou materializar antes de iterar

🔷 5. yield / Iteradores

Erro comum	Por que acontece	Como evitar / corrigir
Tentativa de acessar elementos fora do iterador	`yield return` gera elementos sob demanda	Iterar sempre via `foreach` ou LINQ; não tentar acessar por índice
Consumo múltiplo do mesmo iterador	Sequência é reavaliada a cada iteração	Materializar em lista (`ToList()`) se precisar reutilizar
Confusão entre `yield return` e `return`	`return` finaliza método inteiro	Use `yield break` para parar sequência, `yield return` para emitir valor
Sobrecarga de memória se gerar lista temporária desnecessária	Criar `List<T>` dentro do iterador grande	Apenas `yield return` sem criar listas grandes

🔷 6. Boas práticas gerais para evitar erros

1. **Evitar variáveis mágicas** → use `Length`, `Count` ou constantes nomeadas.
2. **Abstração e funções** → encapsule lógica de iteração, filtragem ou transformação.
3. **Flag / controle de fluxo** → quando precisar parar a iteração antecipadamente, use `break`, `return` ou `yield break`.
4. **Escolher loop correto** → `foreach` para leitura, `for` quando precisa de índice ou remover elementos.
5. **Materialização quando necessário** → converter `IEnumerable` para `List` ou `Array` se precisar de múltiplas iterações ou acesso por índice.
6. **Testar limites e coleções vazias** → sempre verificar se a coleção não é nula ou vazia antes de iterar.
7. **Evitar alterações durante iteração** → especialmente em List e Dictionary.

🟧 Exercícios mistos (20 exercícios)

1. Receber array e retornar List só com números pares.
2. Iterar List e criar Dictionary de contagem de elementos.
3. Converter Dictionary em IEnumerable filtrado por valor > X.
4. Receber IEnumerable, materializar em List e inverter.
5. Criar função que recebe array e retorna IEnumerable que gera quadrados dos elementos.
6. Iterar List e parar ao encontrar o primeiro valor negativo usando flag.
7. Converter List de strings em Dictionary chave = string, valor = tamanho.
8. Criar pipeline usando IEnumerable: filtrar, mapear, pegar primeiros 10.
9. Implementar função que retorna IEnumerable de Fibonacci usando yield.
10. Receber Dictionary e criar lista de chaves ordenadas por valor.
11. Iterar array e remover duplicados criando novo array.

12. Receber List e calcular média dos valores positivos.
13. Iterar IEnumerable infinito e parar com flag ao chegar em limite.
14. Criar Dictionary a partir de List de objetos (Id → Nome).
15. Receber array de números e retornar IEnumerable com apenas primos.
16. Receber List e criar pipeline de transformação em strings formatadas.
17. Materializar IEnumerable, ordenar e imprimir.
18. Iterar Dictionary e atualizar valores multiplicando por 2 usando foreach seguro.
19. Criar função que recebe List e retorna IEnumerable de elementos alternados (0,2,4...).
20. Receber array e criar Dictionary de contagem de números pares e ímpares.

🔷 1. Tuplas: Conceito

- Uma **tupla** é um **agrupamento de valores heterogêneos** em uma única estrutura.
- Permite **retornar múltiplos valores de um método** sem criar uma classe ou struct.
- Pode ser **imutável** (Tuple<>) ou **mutável** ((...)).

🔷 2. Tuple<> (explícita)

- Classe genérica tradicional.
- Imutável, **não permite nomes de campos diretamente**.
- Ideal para interoperabilidade com APIs antigas ou quando nomes não importam.

Exemplo

```
// Declaração explícita
Tuple<int, string, bool> pessoa = new Tuple<int, string, bool>(1, "Ana", true);

// Acesso
Console.WriteLine(pessoa.Item1); // 1
Console.WriteLine(pessoa.Item2); // Ana
Console.WriteLine(pessoa.Item3); // true
```

Observações:

- Não suporta **desconstrução direta** (antes do C# 7).
- Uso limitado em código moderno devido a legibilidade.

🔷 3. Tupla implícita ((...))

- Introduzida no C# 7.
- Permite **nomes de campos**, melhorando legibilidade.
- Pode ser **mutável** se armazenada em variável `var` ou explicitamente.

Exemplo básico

```
// Tupla implícita com nomes
(int Id, string Nome, bool Ativo) pessoa = (1, "Ana", true);

// Acesso
Console.WriteLine(pessoa.Id);    // 1
Console.WriteLine(pessoa.Nome);  // Ana
Console.WriteLine(pessoa.Ativo); // true

// Desconstrução
var (id, nome, ativo) = pessoa;
Console.WriteLine(nome); // Ana
```

Vantagens:

- Mais legível do que `Item1`, `Item2`.
- Facilita **desconstrução e passagem de múltiplos valores**.
- Ideal para **retorno de métodos**.

🔷 4. Boas práticas com tuplas

1. **Use tupla implícita ((...)) com nomes claros** sempre que possível.
2. **Evite tupla quando a estrutura de dados cresce muito** → prefira uma classe ou struct.
3. **Use tupla para retornos rápidos e temporários**, não para armazenar estado complexo.
4. **Desconstrução**: aproveite para criar variáveis locais de forma limpa.
5. **Não confunda com List ou Array** → tupla tem tamanho fixo e tipos heterogêneos.

🔷 5. Comparativo Tuple<> vs (...):

Aspecto	Tuple<>	(...)/ValueTuple
Mutabilidade	Imutável	Mutável
Nome dos campos	Item1, Item2...	Pode nomear
Desconstrução	Não nativa	Suporte nativo
Sintaxe	Verbosa	Concisa
Código moderno	Legado/compatibilidade	Recomendado
Boas práticas	Usar apenas se necessário	Retornos rápidos, claros

---

🔷 6. Exemplos de uso prático

Retornando múltiplos valores de um método

```
// Retorno explícito
Tuple<int, int> SomaESubtracao(int a, int b)
{
    return new Tuple<int, int>(a + b, a - b);
}

// Retorno implícito
(int soma, int subtracao) SomaESubtracaoModerno(int a, int b)
{
    return (a + b, a - b);
}

// Uso
var resultado = SomaESubtracaoModerno(5, 3);
Console.WriteLine(resultado.soma);      // 8
Console.WriteLine(resultado.subtracao);  // 2
```

Iterando com tuplas em lista

```
var pessoas = new List<(int Id, string Nome)>
{
    (1, "Ana"),
    (2, "Beto")
};

foreach (var (id, nome) in pessoas)
    Console.WriteLine($"{id} - {nome}");
```

Vantagens: clareza, fácil acesso, integração com loops e LINQ.

🔷 7. Integração com dicionários

Tuple<> (explícita) como chave ou valor:

```
var dict = new Dictionary<Tuple<int, int>, string>();
dict[new Tuple<int, int>(1, 2)] = "Par 1-2";

foreach (var kvp in dict)
    Console.WriteLine($"{kvp.Key.Item1}-{kvp.Key.Item2} ⇒ {kvp.Value}");
```

Problemas / ruim:

- Verboso, difícil leitura se chave tiver mais de 2 itens.
- `Item1`/`Item2` → não descritivo.

Tupla implícita (ValueTuple):

```
var dict = new Dictionary<(int X, int Y), string>();
dict[(1, 2)] = "Par 1-2";

foreach (var kvp in dict)
    Console.WriteLine($"{kvp.Key.X}-{kvp.Key.Y} ⇒ {kvp.Value}");
```

Vantagens / bom:

- Campos nomeados → código claro.
- Funciona como chave em dicionário (`ValueTuple` é comparável).
- Permite loops claros e manutenção simplificada.

🔷 8. Boas práticas de uso de tuplas

1. **Prefira ValueTuple** (`(...)`) com nomes descritivos.
2. **Evite criar tuplas grandes (>4-5 elementos)** → prefira classe/struct.
3. **Use tuplas para retorno de métodos simples**, não para armazenar estado complexo.
4. **Desconstrua** quando for iterar ou manipular múltiplos valores.

5. **Não abuse do Tuple<> legado** em código moderno, só em interoperabilidade com APIs antigas.
6. **Combine tuplas com loops e flags** para lógica de interrupção ou reversão.

🔷 9. Exemplos de código ruim x bom resumido

Cenário	Código ruim (Tuple<>)	Código bom ((...))
Retorno múltiplos valores	`return new Tuple<int,int>(a,b)`	`return (soma:a,b)`
Iteração lista de pessoas	`p.Item1`, `p.Item2`	`foreach(var (id,nome) in lista)`
Integração com dicionário	`dict[new Tuple<int,int>(x,y)]`	`dict[(X:x,Y:y)]`
Desconstrução	Não suporta	`var (x,y) = tupla;`

🟦 1. Tuplas básicas (10 exercícios)

1. Criar uma tupla `(int, string)` com Id e Nome e imprimir.
2. Criar uma tupla `Tuple<int, string, bool>` e acessar seus itens usando `Item1`, `Item2`, `Item3`.
3. Desconstruir uma tupla implícita `(int Id, string Nome)` em variáveis separadas.
4. Criar uma função que retorna `(int soma, int subtracao)` e imprimir os resultados.
5. Criar uma função que retorna `Tuple<int,int,int>` com soma, subtração e multiplicação de dois números.
6. Modificar valores de uma tupla implícita mutável e imprimir antes/depois.
7. Criar uma tupla com elementos heterogêneos `(int, string, double, bool)` e acessar cada valor.
8. Iterar sobre um array de tuplas `(int, string)` e imprimir os elementos.
9. Criar uma tupla com valores negativos e criar flag para interromper impressão no primeiro negativo.

10. Criar função que retorna tupla implícita com Id e Nome somente se Id > 0; caso contrário, retorna `(0,"")`.

🟩 2. Tuplas em listas (10 exercícios)

1. Criar `List<(int Id, string Nome)>` e imprimir usando `foreach`.
2. Iterar `List<Tuple<int,string>>` usando `for` e `Item1`, `Item2`.
3. Adicionar novos elementos a uma lista de tuplas.
4. Remover elementos da lista onde `Id` < 5 usando `RemoveAll`.
5. Criar flag para parar o loop quando encontrar `Nome = "Beto"`.
6. Ordenar lista de tuplas por `Nome`.
7. Ordenar lista de tuplas por `Id` decrescente.
8. Desconstruir cada tupla da lista dentro do loop e imprimir apenas `Nome`.
9. Criar uma lista de tuplas, filtrar apenas Id pares e retornar nova lista.
10. Criar uma função que recebe lista de tuplas e retorna apenas os nomes em array de strings.

🟨 3. Tuplas em dicionários (10 exercícios)

1. Criar `Dictionary<int, (string Nome, bool Ativo)>` e iterar.
2. Adicionar novos elementos a dicionário usando ValueTuple.
3. Atualizar valor da tupla em dicionário.
4. Criar flag para interromper loop ao encontrar um usuário inativo.
5. Converter dicionário em lista de tuplas e iterar.
6. Ordenar dicionário por valor `Nome`.
7. Verificar se uma chave existe antes de acessar a tupla.
8. Criar função que recebe dicionário e retorna média de Ids.
9. Criar dicionário com tupla como chave `(int X, int Y)` e iterar.

10. Comparar acesso de `Tuple<>` vs `(...)` como chave de dicionário.

🟪 4. Tuplas com métodos e desconstrução (10 exercícios)

1. Criar função que retorna `(int, int, int)` com soma, subtração e multiplicação, e desconstruir no método chamador.
2. Criar função que retorna tupla implícita de dados de pessoa `(Id, Nome, Ativo)` e desconstruir em variáveis locais.
3. Criar método que retorna tupla `(bool sucesso, string mensagem)` e usar em condicional `if`.
4. Criar função que retorna tupla de coordenadas `(X,Y,Z)` e iterar usando `for`.
5. Criar função que retorna tupla `(min, max)` de um array de números.
6. Criar função que recebe tupla `(int, int)` e retorna `true` se a soma for par.
7. Criar função que recebe lista de tuplas e retorna tupla `(maiorId, nomeMaiorId)`.
8. Criar função que retorna tupla de índices do primeiro número negativo em um array 2D.
9. Criar função que retorna tupla implícita `(valor, mensagem)` baseada em condição e usar flag no loop.
10. Criar função que retorna tupla de booleanos `(par, positivo)` para cada número em array e imprimir usando desconstrução.

🟧 5. Exercícios avançados / mistos (10 exercícios)

1. Iterar `IEnumerable<(int, string)>` usando `foreach` e desconstrução.
2. Criar função `yield return` que retorna tupla `(int Fibonacci, bool éPar)` para N elementos.
3. Criar lista de tuplas, filtrar elementos ativos e imprimir.
4. Receber array de tuplas `(int, string)`, ordenar e imprimir.

5. Iterar dicionário de tuplas e criar nova lista só com elementos que satisfazem condição.
6. Criar função que retorna tupla com múltiplos valores e usar `switch` baseado em um campo da tupla.
7. Criar flag para parar iteração em lista de tuplas quando encontrar tupla com nome específico.
8. Criar função que recebe lista de tuplas e retorna tupla `(min, max, média)`.
9. Iterar lista de tuplas e modificar valores (ValueTuple mutável) durante iteração.
10. Criar pipeline: array → tupla → lista → filtrar → desconstruir → imprimir.