



# Programação .NET C#

## Coleções e LINQ

Aprenda a manipular dados de forma eficiente com Arrays, Listas, Dicionários, LINQ e muito mais

# Arrays e Listas Genéricas: conceitos e diferenças

## Arrays

- ✓ Tamanho fixo, definido na criação
- ✓ Acesso por índice:  $O(1)$
- ✓ Sintaxe: `tipo[] nome = new tipo[N];`

## ? Quando usar cada um?

### Array

- > Tamanho fixo
- > Alta performance

## List<T>

- ✓ Tamanho dinâmico (cresce/encolhe)
- ✓ Métodos: Add, Remove, Find, Clear
- ✓ Sintaxe: `List<tipo> nome = new();`

### List<T>

- > Tamanho variável
- > Mais flexível

## </> Exemplos

### Array:

```
int[] numeros = new int[5];
numeros[0] = 10;
// Erro se tentar adicionar além do tamanho
```

### List<T>:

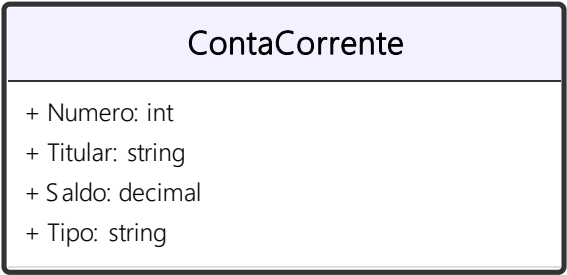
```
List<int> numeros = new();
numeros.Add(10); // Adiciona elementos
numeros.Remove(10); // Remove elementos
```

# Exercício 1: Arrays e List<T> (List<ContaCorrente>)

## Objetivo

Praticar o uso de List<T> para armazenar objetos e demonstrar operações de coleções.

## Diagrama UML da Classe



### Dica

Compare o comportamento dinâmico da List<T> vs array fixo para entender as vantagens de cada estrutura.

## Critérios de Aceitação

- Lista criada e populada
- Operações (Add, Remove, Find) funcionando
- Comparação com array demonstrada
- Total de contas exibido

## Passo a passo

### 1 Crie a classe ContaCorrente

Implemente a classe seguindo o diagrama UML ao lado, com as propriedades Numero (int), Titular (string), Saldo (decimal) e Tipo (string).

### 2 Declare uma List<ContaCorrente>

No programa principal, declare uma lista para armazenar as contas correntes.

### 3 Popule a lista com 5 contas variadas

Adicione pelo menos 5 contas com dados variados, incluindo saldos positivos, negativos e diferentes tipos ("Comum", "Especial").

### 4 Demonstre as operações básicas de coleções:



#### Add()

Adicionar uma nova conta à lista.



#### Remove()

Remover uma conta (ex: com condição específica).



#### Find()

Buscar uma conta por número ou critério.



#### Count

Exibir a quantidade total de contas.

### 5 Compare array (fixo) com List (dinâmica)

Crie um array de tamanho fixo (ex: 3) e tente adicionar mais elementos. Compare com o comportamento da List, que cresce dinamicamente. Mostre na prática quando cada estrutura é mais adequada.

### 6 Apresente os resultados

Exiba o total de contas após as operações e mostre exemplos de como a lista dinâmica se comporta em comparação com o array de tamanho fixo.



# Outras Coleções: Dictionary, HashSet, Queue, Stack

## Dictionary<TKey,TValue>

- ✓ Pares chave-valor, acesso  $O(1)$  por chave única
- ✓ Métodos: Add, TryGetValue, ContainsKey
- 💡 Use para: lookups rápidos, mapeamentos

## HashSet<T>

- ✓ Conjunto sem duplicatas, verificação  $O(1)$
- ✓ Métodos: Add, Contains, UnionWith
- 💡 Use para: unicidade, operações de conjuntos

## Queue<T> (FIFO)

- ✓ Primeiro a entrar = primeiro a sair
- ✓ Métodos: Enqueue, Dequeue, Peek
- 💡 Use para: filas de processamento, buffers

## Stack<T> (LIFO)

- ✓ Último a entrar = primeiro a sair
- ✓ Métodos: Push, Pop, Peek
- 💡 Use para: desfazer, navegação histórica

## Como escolher a coleção ideal?

- Dictionary: quando precisa de chaves únicas
- HashSet: para verificar unicidade
- Queue: para processar na ordem de chegada
- Stack: para operações aninhadas/reversíveis

# Exercício 2: Mudar armazenamento para Dictionary<int, ContaCorrente>

## Objetivo

Migrar de List<ContaCorrente> para Dictionary<int, ContaCorrente> para acesso O(1) por chave.

### Por que Dictionary?

- Acesso rápido O(1) por chave (número da conta)
- Evita busca sequencial em listas grandes
- Perfeito para lookups frequentes por ID

### Critérios de Aceitação

- Dictionary implementado corretamente
- Buscas funcionando por número
- Tratamento de chaves duplicadas
- Operações básicas funcionando

## Passo a passo

- Substitua List por Dictionary**

Altere a declaração da coleção para usar Dictionary<int, ContaCorrente>, onde a chave é o número da conta e o valor é o objeto ContaCorrente.
- Migre os dados existentes**

Transfira todas as contas da List para o Dictionary, usando o número da conta como chave. Implemente uma verificação para evitar a duplicação de chaves (números de conta).
- Implemente as operações básicas**

**Buscar conta**  
Use TryGetValue para buscar uma conta pelo número com acesso O(1)

**Remover conta**  
Use Remove(key) para excluir uma conta pelo seu número

**Listar todas as contas**  
Itere pelo Dictionary usando foreach e KeyValuePair

**Verificar existência**  
Use ContainsKey para verificar se uma conta existe
- Implemente um HashSet de titulares (opcional)**

Crie um HashSet para rastrear titulares únicos, demonstrando outro tipo de coleção especializada para unicidade.
- Compare o desempenho**

**List**  
Busca: Complexidade O(n)  
Lenta para coleções grandes

**Dictionary**  
Busca: Complexidade O(1)  
Rápida independente do tamanho

### Importante

O uso do Dictionary exige que a chave seja única. Certifique-se de tratar adequadamente casos onde o número da conta já existe no dicionário.

# Introdução ao LINQ (I): conceitos e sintaxes

## 💡 O que é LINQ

- ✓ Language INtegrated Query: consultas integradas à linguagem
- ✓ Consulta qualquer fonte: coleções, XML, bancos de dados

## 🔗 Namespaces e Execução

- ✓ Importe System.Linq para usar LINQ
- ✓ Execução adiada vs. materialização (ToList(), Count())

## </> Sintaxes de Consulta LINQ

### 📄 Query Syntax

```
var resultado = from c in contas
where c.Saldo < 0
orderby c.Titular
select c;
```

- > Inspirada em SQL, mais legível para consultas complexas

### 🔗 Method Syntax

```
var resultado = contas
.Where(c => c.Saldo < 0)
.OrderBy(c => c.Titular);
```

- > Encadeamento de métodos, mais flexível

### ✅ Boas Práticas

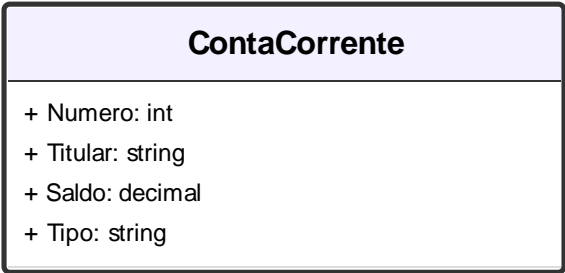
- ✓ Escolha a sintaxe mais legível
- ✓ Trate coleções nulas
- ✓ Materialize quando necessário
- ✓ Evite consultas repetidas

# Exercício 3: LINQ básico (busca e filtro)

## Objetivo

Praticar consultas LINQ para buscar e filtrar dados no Dictionary<int, ContaCorrente>.

## Diagrama UML da Classe



## Dica

Compare as duas sintaxes de LINQ para entender suas diferenças e quando cada uma é mais apropriada.

## Critérios de Aceitação

- Conta encontrada corretamente por número
- Lista completa de contas com saldo negativo
- Ambas sintaxes demonstradas
- Resultados materializados com ToList()

## Passo a passo

- 1 Implemente a busca de uma conta pelo número**  
Utilize LINQ para buscar uma conta específica pelo seu número (ex: 1001) no Dictionary de contas. Implemente usando as duas sintaxes: consulta (query syntax) e método (method syntax).
- 2 Filtre as contas com saldo negativo**  
Crie uma consulta LINQ que retorne todas as contas que possuem saldo menor que zero. Novamente, implemente usando as duas sintaxes diferentes.
- 3 Entenda as duas sintaxes LINQ**

**Method Syntax:**  
Utiliza encadeamento de métodos com expressões lambda. Recomendada para operações simples e encadeadas. Oferece acesso a mais operadores específicos.

**Query Syntax:**  
Similar à sintaxe SQL com palavras-chave como from, where e select. Geralmente mais legível em consultas complexas com múltiplas operações.
- 4 Materialize e exiba os resultados**  
Use ToList() para materializar os resultados imediatamente  
Exiba informações da conta encontrada (número, titular, saldo)  
Mostre quantidade de contas com saldo negativo encontradas  
Imprima os detalhes de cada conta com saldo negativo

### Importante:

Observe a execução adiada vs. imediata em LINQ. Sem materialização (ToList(), ToArray(), etc.), a consulta não é executada até que seus resultados sejam iterados.

# LINQ (II): Where, OrderBy, Select, GroupBy

## 🔍 Where

- ✓ Filtra elementos com base em predicado

```
</> contasp.Where(c => c.Saldo < 0)
```

- 💡 Reduz dados antes de operações complexas

## 🔧 Select

- ✓ Projeta para nova forma (transformação)

```
</> Select(c => new { c.Titular, c.Saldo })
```

- 💡 Reduz dados transferidos, selecionando apenas o necessário

## ⬇️ OrderBy/OrderByDescending

- ✓ Ordenação por uma ou mais propriedades

```
</> OrderBy(c => c.Saldo).ThenBy(c => c.Titular)
```

- 💡 Use OrderByDescending para ordem decrescente

## 📁 GroupBy

- ✓ Agrupa elementos com chave em comum

```
</> var grupos = contasp.GroupBy(c => c.Tipo);
```

- 💡 Acessar: Key, Count(), Sum(c => c.Saldo)

### Exemplo de encadeamento:

```
contasp.Where(c => c.Saldo > 1000) .OrderByDescending(c => c.Saldo) .GroupBy(c => c.Tipo) .Select(g => new { Tipo = g.Key, Total = g.Count(), SaldoMedio = g.Average(c => c.Saldo) });
```

### 💡 Dicas:


Encadeie operações; materialize ao final (ToList/ToArray); evite múltiplas enumerações.

# Exercício 4: LINQ avançado (agrupar, ordenar, projetar)



## Objetivo

Praticar LINQ para agrupar por tipo, ordenar por saldo e projetar dados.

 **Dica**

Use GroupBy com .Key para acessar e agregar dados de cada grupo.

### Critérios de Aceitação

- ☒ Grupos por Tipo de conta
- ☒ Contagem por grupo
- ☒ Ordenação: Saldo (desc), Titular
- ☒ Projeção para Titular e Saldo

ContaCorrente
+ Numero: int
+ Titular: string
+ Saldo: decimal
+ Tipo: string



## Passo a passo

- 1

**Verifique a propriedade Tipo em ContaCorrente**  
Certifique-se de que a classe ContaCorrente possui a propriedade Tipo (string) com valores como "Comum", "Especial", etc. Esta propriedade será usada para agrupamento.
- 2

**Agrupe as contas por Tipo e conte**  
Use o método GroupBy para agrupar as contas por Tipo. Depois, para cada grupo, conte o número de elementos usando Count(). Você pode acessar a chave do grupo com a propriedade Key.  
Exiba o nome do tipo e a quantidade de contas em cada grupo.
- 3

**Ordene por Saldo (decrescente) e Titular**  
Implemente a ordenação usando o método OrderByDescending para ordenar por Saldo (maior para menor) e depois use ThenBy para desempates pelo Titular (ordem alfabética).  
Você pode usar tanto a *Method Syntax* (com métodos encadeados) quanto a *Query Syntax* (estilo SQL com from/orderby).
- 4

**Projete para tipo anônimo (Titular e Saldo)**  
Use o método Select para criar uma projeção que inclua apenas as propriedades Titular e Saldo, criando um novo tipo anônimo que contém apenas esses dados.  
Exiba a lista projetada em formato de tabela ou lista.
- 5

**Combine todas as operações**  
Crie uma consulta LINQ completa que:  
Agrupe as contas por Tipo  
Para cada grupo, calcule a quantidade de contas  
Ordene as contas dentro de cada grupo por Saldo (decrescente)  
Projete apenas Titular e Saldo para cada conta  
Exiba os resultados de forma estruturada no console.

Observação: Este exercício é sobre praticar a sintaxe LINQ. Não se preocupe em memorizar os métodos, mas entenda o conceito de cada operação (agrupamento, ordenação e projeção).

## ” Interpolação de String

- ✓ Prefixo \$: \$"Olá, {nome}!"
- ✓ Formatação: \$"Saldo: {saldo:C}", \$"Data: {data:d}"

## A Formatação

- ✓ Padrão: "C" (moeda), "d" (data curta)
- ✓ Cultura: ToString("C", CultureInfo.GetCultureInfo("pt-BR"))

## ✂ Métodos Úteis de String

- ✓ Manipulação: Substring(), Replace(), Trim()
- ✓ Verificação: StartsWith(), EndsWith(), Contains()
- ✓ Helpers: Join(), Split(), IsNullOrEmpty()

## 📅 DateTime

- ✓ Obter: Now, UtcNow, Today
- ✓ Converter: Parse(), TryParse()
- ✓ Manipular: AddDays(), AddMonths()
- ✓ Diferenças: TimeSpan ts = data2 - data1;

## </> Exemplo Prático

```
var hoje = DateTime.Now;  
var vencimento = hoje.AddDays(30);  
var dias = (vencimento - hoje).Days;  
Console.WriteLine($"Vence em {dias} dias ({vencimento:dd/MM/yyyy})");
```

## ★ Boas Práticas

- ✓ Padronize formatos para consistência visual
- ✓ Use DateTimeOffset para considerar fuso horário
- ✓ Use StringBuilder para múltiplas concatenações

# Exercício 5: Registro de transações com data/hora



## Objetivo

Criar registro de transações bancárias com data/hora e formatar a saída de forma amigável.



## Modelo UML

### Transacao

- + Id: Guid
- + ContaNumero: int
- + Valor: decimal
- + Tipo: string
- + Descricao: string
- + DataHora: DateTime



### Dica

Use CultureInfo("pt-BR") para formatação monetária regional.



## Critérios de Aceitação

- Transações com timestamp correto
- Valores formatados (pt-BR)
- Datas: dd/MM/yyyy HH:mm
- Extrato legível e organizado



## Passo a passo

- 1 Crie a classe Transacao conforme o diagrama UML**  
Use as propriedades exatamente como definidas, inicialize Id com Guid.NewGuid() e DataHora com DateTime.Now
- 2 Crie uma lista para armazenar as transações**  
Declare uma lista global de transações (List<Transacao>) e implemente um método para registrar novas transações que atualize o saldo da conta
- 3 Implemente formatação de datas e valores**  
Use CultureInfo("pt-BR") para formatação monetária e utilize os formatos corretos (dd/MM/yyyy HH:mm para datas e "C" para valores monetários)
- 4 Crie método de extrato usando LINQ**  
Filtre por número da conta, ordene por data decrescente (mais recente primeiro), limite pelo número de transações solicitado e formate a saída
- 5 Teste com transações variadas**  
Registre diferentes tipos de transação (créditos e débitos) e gere extratos para verificar a formatação e organização



### Exemplo de saída esperada:

EXTRATO DA CONTA 1001 - João Silva			
22/10/2025 14:25	Crédito	R\$ 500,00	Depósito...
22/10/2025 14:20	Débito	R\$ 150,50	Pagamento...



# Encerramento e próximos passos

## ✓ Recap do conteúdo

- ✓ Arrays vs List<T>: tamanho fixo vs dinâmico
- ✓ Coleções especializadas: Dictionary, HashSet, Queue, Stack
- ✓ LINQ: filtros, ordenação, agrupamento e projeção
- ✓ Strings e Datas: interpolação, formatação, operações comuns

## 🏠 Próximos passos

- LINQ avançado e expressões assíncronas
- Performance de coleções e otimizações
- LINQ to Entities (Entity Framework)
- Testes automatizados para LINQ

## 📖 Recursos de aprendizado

### 🌐 Documentação

- 🔗 Microsoft Learn: [learn.microsoft.com/dotnet](https://learn.microsoft.com/dotnet)
- 🔗 Docs .NET: [docs.microsoft.com/dotnet](https://docs.microsoft.com/dotnet)

### 💻 Prática

- 🔗 GitHub: [github.com/dotnet/samples](https://github.com/dotnet/samples)
- 🔗 Stack Overflow: [\[c#\]](#) [\[linq\]](#)