Exceções em C# — Fundamentos Didáticos e POO

Ao final deste estudo você será capaz de:

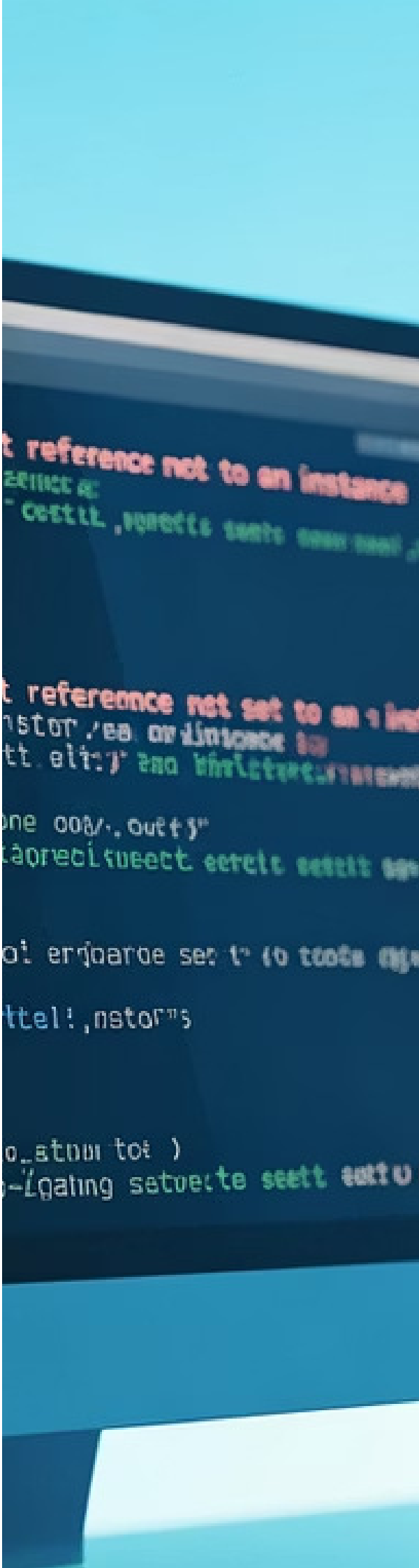
1. **Explicar o que é uma exceção e por que ela existe**
2. **Diferenciar erros esperados de falhas excepcionais**
3. **Usar try / catch / finally, throw, encadeamento via InnerException e mensagens úteis**
4. **Escolher tipos de exceção adequados e criar exceções de domínio**

   Ex.: ArgumentException, InvalidOperationException
5. **Projetar pontos de lançamento e pontos de captura em uma aplicação por camadas**

   Onde *lançar* e onde *tratar*
6. **Tratar exceções em código assíncrono**

   async / await, Task.WhenAll
7. **Aplicar boas práticas**

   fail-fast, não engolir exceções, não controlar fluxo com exceções, logging e comunicação amigável ao usuário



Por que exceções?

Separar o caminho feliz do caminho de erro

Leitura e manutenção ficam mais claras quando casos raros não poluem o fluxo principal.

Falhar rápido (fail-fast)

Quanto antes detectarmos uma inconsistência, menor o custo de propagação do erro.

Proteger invariantes de domínio

Impedir estados inválidos (ex.: saldo negativo) melhora a confiabilidade do software.

Composição de camadas

Cada camada decide o que **significa** uma falha: domínio lança, aplicação traduz, UI comunica ao humano.

Comparação didática

Sem exceções:

- Funções retornam códigos (bool, ints)
- Fácil de ignorar
- Chamadas aninhadas geram "if-else pirâmide"

Com exceções:

- O código expressa o que é **normal**
- O que não é normal **interrompe** o fluxo automaticamente

Vocabulário essencial

Exceção

Objeto que representa uma **anomalia** em tempo de execução.

Lançar (throw)

Sinalizar que algo quebrou uma regra/expectativa.

Capturar (catch)

Lidar com a falha (traduzir, retentar, registrar, informar).

finally

Bloco que **sempre** roda (limpeza de recursos).

Unchecked

Em C# **não** há verificação de exceções em tempo de compilação: responsabilidade do projeto de tratamento, não do compilador.

Árvore importante (parcial):

System.Exception → System.SystemException / ApplicationException (evite herdar de ApplicationException) → especializadas (ArgumentException, InvalidOperationException, IOException, TimeoutException, ...).

Onde lançar? Onde capturar?

Lançar

Onde a **causa real** acontece (e onde a regra é conhecida):

- Domínio
- Validações de argumentos
- Serviços de infraestrutura

Capturar

Em **fronteiras**:

- Camada de aplicação
- Endpoints
- UI
- Jobs

Nesses pontos você **traduz** a falha (mensagem amigável, HTTP 400/404/409/500, etc.).

> Regra didática: "Perto do erro eu lanço; longe do erro (na borda) eu decido como responder."

Guard Clauses (validar cedo)

```
public static class Guard
{
   public static void AgainstNull(object? value, string paramName)
   {
      if (value is null)
        throw new ArgumentNullException(paramName);
   }

   public static void AgainstOutOfRange(bool condition, string message,
        string paramName)
   {
      if (condition)
        throw new ArgumentOutOfRangeException(paramName,
          message);
   }
}
```

Uso:

```
public class Produto
{
 public string Codigo { get; }
 public string Nome { get; private set; }
 public double PrecoUnitario { get; private set; }
 public int Quantidade { get; private set; }

 public Produto(string codigo, string nome, double preco, int qtd)
 {
   Guard.AgainstNull(codigo, nameof(codigo));
   if (string.IsNullOrWhiteSpace(nome))
     throw new ArgumentException("Nome inválido", nameof(nome));

   Guard.AgainstOutOfRange(preco < 0, "Preço não pode ser negativo",
     nameof(preco));
   Guard.AgainstOutOfRange(qtd < 0, "Quantidade não pode ser negativa",
     nameof(qtd));
   Codigo = codigo;
   Nome = nome;
   PrecoUnitario = preco;
   Quantidade = qtd;
 }

 public void Remover(int qtd)
 {
   Guard.AgainstOutOfRange(qtd <= 0, "Quantidade deve ser positiva",
     nameof(qtd));
   Guard.AgainstOutOfRange(qtd > Quantidade, "Estoque insuficiente",
     nameof(qtd));
   Quantidade -= qtd;
 }
}
```

Tipos comuns e quando usar

Tipo de exceção	Quando usar	Exemplo de mensagem
ArgumentNullException	parâmetro obrigatório ausente	"clienteId não pode ser nulo"
ArgumentOutOfRangeException	valor fora do domínio permitido	"percentual deve estar entre 0 e 1"
ArgumentException	argumento inválido em geral	"CPF em formato inválido"
InvalidOperationException	operação proibida no estado atual	"Pedido já foi fechado"
KeyNotFoundException	chave/registro não encontrado	"Produto C001 não encontrado"
TimeoutException	operação excedeu tempo	"Timeout ao consultar serviço de pagamento"
IOException	falha de E/S	"Erro ao salvar arquivo de relatório"
DomainException (custom)	violação de regra de negócio	"Saldo insuficiente para saque"

Exceção de domínio (custom)

```
public class DomainException : Exception
{
    public DomainException(string message) : base(message) { }
    public DomainException(string message, Exception inner) : base(message,
        inner) { }
}
```

Uso:

```
public class ContaBancaria
{
 public decimal Saldo { get; private set; }
 public void Sacar(decimal valor)
 {
   if (valor <= 0)
     throw new ArgumentOutOfRangeException(nameof(valor));
   if (valor > Saldo)
     throw new DomainException("Saldo insuficiente");
   Saldo -= valor;
 }
}
```

try/catch/finally, rethrow e limpeza de recursos

```
try
{
   // código que pode falhar
}
catch (IOException ex)
{
   // trate o tipo específico que você sabe lidar
   Console.Error.WriteLine($"Falha de I/O: {ex.Message}");
   // opcional: re-lançar com contexto
   throw new IOException("Falha ao exportar relatório anual", ex);
}
finally
{
   // sempre executa: fechar conexões, liberar locks, etc.
}
```

Rethrow correto

`throw;` preserva o *stack trace* original.

`throw ex;` **reseta** o *stack trace* (evite).

using / IDisposable

```
// usando declaração (C# 8+)
using var stream = File.OpenRead(caminho);
// processa o stream; liberação automática ao sair do escopo
```

Assíncrono (async / await)

- Exceções lançadas dentro de async surgem quando você **await** a Task.
- Com Task.WhenAll, múltiplas exceções surgem como AggregateException se você não usar await diretamente em cada tarefa.

```
try
{
 await OperacaoRemotaAsync();
}
catch (HttpRequestException ex)
{
 Console.Error.WriteLine($"Falha HTTP: {ex.Message}");
}

// WhenAll
var t1 = BaixarAsync(url1);
var t2 = BaixarAsync(url2);

try
{
 await Task.WhenAll(t1, t2);
}
catch
{
 // t1.Exception ou t2.Exception podem conter detalhes
 if (t1.IsFaulted)
    Console.Error.WriteLine(t1.Exception?.GetBaseException().Message);
 if (t2.IsFaulted)
    Console.Error.WriteLine(t2.Exception?.GetBaseException().Message);
}
```

Boas práticas (Do & Don't)

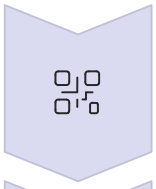
Do

- Lance **cedo** e com **mensagem clara**.
- Prefira **exceções específicas**; use Exception apenas em último caso.
- Propague contexto: use InnerException ao re-lançar.
- Capture em **pontos de borda** para **traduzir** (ex.: para HTTP 400/404/409/500) e registrar.
- Teste cenários de falha tanto quanto os de sucesso.

Don't

- Não use exceções para **controle de fluxo** normal (ex.: sair de laços, escolher caminho trivial).
- Não engula exceções (catch {} vazio) — registre, traduza ou re-lançe.
- Evite jogar exceções dentro de loops muito sensíveis a performance.
- Não exponha mensagens técnicas diretamente ao usuário final.

Estratégia por camadas (exemplo)

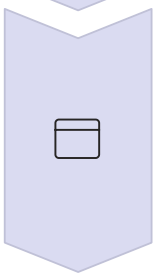


Domínio

Garante invariantes, lança DomainException / InvalidOperationException / Argument*.

Aplicação/Serviço

Orquestra, **não** engole; pode traduzir falhas de infraestrutura.



Borda (UI/API/Worker)

Captura e **traduz** em resposta amigável (ex.: HTTP 400 para ArgumentException, 404 para KeyNotFoundException, 409 para DomainException, 500 para desconhecidos). Também faz **logging**.

Exercício mental: para cada exceção lançada no domínio, decida qual **status**/mensagem deve aparecer na borda.

Laboratórios guiados

Lab A — Primeiros passos com try/catch

1. Escreva um método que lê um arquivo pelo caminho informado.
2. Lance ArgumentException se o caminho vier vazio e trate FileNotFoundException.
3. Registre a mensagem de erro e continue a execução.

Desafio: adicione finally para imprimir "Fim da tentativa" sempre.

Lab B — Invariantes no domínio

Implemente Produto (código, nome, preço, quantidade) com validações do construtor e métodos Adicionar / Remover:

- Lance ArgumentOutOfRangeException para valores negativos e DomainException para regras de negócio (ex.: estoque insuficiente).
- Escreva testes no Program.cs que tentem quebrar as regras e mostre as mensagens.

Lab C — Mapeamento para API (conceitual)

Simule uma borda de API: um método ExecutarAcao() chama o domínio. Faça um try/catch e imprima o status que devolveria:

Argument* → 400	KeyNotFoundException → 404
DomainException (conflito de regra) → 409	outras → 500

Lab D — Assíncrono

Crie BuscarDadosAsync() que pode lançar HttpRequestException. Escreva dois cenários: sucesso e timeout. Trate corretamente no await.

Exercícios com saída esperada

1

Rethrow correto

Demonstre a diferença entre throw; e throw ex; (espere manter stack com throw;).

2

Guards reutilizáveis

Extraia uma classe Guard e use em três lugares diferentes do domínio.

3

Tabela de mapeamento

Escreva um switch que mapeia exceções para mensagens de UI (amigáveis) e códigos (quando aplicável).

Checklist de consolidação

1. Sei explicar por que exceções existem e quando usá-las.
2. Identifico claramente onde **lançar** e onde **capturar**.
3. Uso tipos de exceção específicos e mensagens claras.
4. Trato exceções em código assíncrono.
5. Não uso exceções para fluxo normal.
6. Tenho uma **estratégia por camadas** para erros.

Próximos passos

Middleware/filtros de exceção em APIs ASP.NET Core

Tradução centralizada de exceções para respostas HTTP.

Políticas de retry e circuit breaker

Estratégias para lidar com falhas transitórias em sistemas distribuídos.

Padronização de mensagens e códigos

Criação de catálogo de erros internos consistentes em toda a aplicação.

Fechamento

> Exceções são a linguagem do .NET para sinalizar e **conter** anomalias. Usadas com critério, elas protegem seus **invariantes**, simplificam o **fluxo feliz** e tornam o sistema mais previsível e fácil de evoluir.

O segredo está em **lançar cedo**, **capturar nas bordas** e **traduzir** de modo adequado ao usuário e ao contexto técnico.

Exemplo prático: Implementação de exceções em sistema bancário

Domínio

```
public class ContaBancaria
{
    public string Numero { get; }
    public decimal Saldo { get; private set; }

    public ContaBancaria(string numero, decimal saldoInicial)
    {
        Guard.AgainstNull(numero, nameof(numero));
        Guard.AgainstOutOfRange(
            saldoInicial < 0,
            "Saldo inicial não pode ser negativo",
            nameof(saldoInicial));

        Numero = numero;
        Saldo = saldoInicial;
    }

    public void Depositar(decimal valor)
    {
        if (valor <= 0)
            throw new ArgumentOutOfRangeException(
                nameof(valor),
                "Valor de depósito deve ser positivo");

        Saldo += valor;
    }

    public void Sacar(decimal valor)
    {
        if (valor <= 0)
            throw new ArgumentOutOfRangeException(
                nameof(valor),
                "Valor de saque deve ser positivo");

        if (valor > Saldo)
            throw new DomainException("Saldo insuficiente");

        Saldo -= valor;
    }
}
```

Aplicação

```
public class ServicoTransferencia
{
 private readonly IRepositorioContas _repositorio;
 private readonly ILogger _logger;

 public ServicoTransferencia( IRepositorioContas repositorio, ILogger logger)
 {
   _repositorio = repositorio;
   _logger = logger;
 }

 public async Task TransferirAsync(string contaOrigem, string contaDestino,
   decimal valor)
 {
   try
   {
     var origem = await _repositorio.BuscarPorNumeroAsync(contaOrigem)
       ?? throw new KeyNotFoundException(
       $"Conta origem {contaOrigem} não encontrada");

     var destino = await _repositorio.BuscarPorNumeroAsync(contaDestino)
       ?? throw new KeyNotFoundException(
       $"Conta destino {contaDestino} não encontrada");

     origem.Sacar(valor);
     destino.Depositar(valor);

     await _repositorio.SalvarAlteracoesAsync();
   }
   catch (Exception ex)
   {
     _logger.LogError(ex, "Erro ao transferir {Valor} de {Origem} para
       {Destino}", valor, contaOrigem, contaDestino);
     throw;
   }
 }
}
```

Exemplo prático: Tratamento na borda da API

```
[ApiController]
[Route("api/transferencias")]
public class TransferenciasController : ControllerBase
{
 private readonly ServicoTransferencia _servico;

 public TransferenciasController(ServicoTransferencia servico)
 {
   _servico = servico;
 }

 [HttpPost]
 public async Task Transferir(TransferenciaRequest request)
 {
   try
   {
     await _servico.TransferirAsync(
     request.ContaOrigem,
     request.ContaDestino,
     request.Valor);

     return Ok(new { Mensagem = "Transferência realizada com sucesso" });
   }
   catch (ArgumentException ex)
   {
     return BadRequest(new { Erro = ex.Message });
   }
   catch (KeyNotFoundException ex)
   {
     return NotFound(new { Erro = ex.Message });
   }
     catch (DomainException ex)
   {
     return Conflict(new { Erro = ex.Message });
   }
```

```
[    catch (Exception)
    {
        return StatusCode(500, new {
            Erro = "Ocorreu um erro ao processar sua solicitação. " +
            "Por favor, tente novamente mais tarde."
        });
    }
 }
}
```