






Notas sobre Tratamento de Erros e Fronteiras

Type	Info
Topics	 Studies
Notebooks	 Programação Profissional com Clean Code
Related Notes	 Ementa do Curso

Práticas de Exceções

- Tratamento de erros
 - Permite que um programa lide com situações inesperadas, mantendo a robustez e confiabilidade do sistema
 - Garantir que, ao ocorrer um erro, o programa continue executando, fornecendo informações úteis para a correção do problema
- Exemplos
 - Usar exceções para tornar o código mais limpo

```
public class Processador
{
    // Exemplo Ruim: Uso de códigos de retorno
    public int ProcessarDados(string entrada)
    {
        if (string.IsNullOrEmpty(entrada))
        {
            return -1; // Código de erro
        }
        // Processamento...
        return 0; // Sucesso
    }
}
```

```

    }

    // Exemplo Bom: Uso de exceções
    public void ProcessarDadosComExcecao(string entrada)
    {
        if (string.IsNullOrEmpty(entrada))
        {
            throw new ArgumentException("Entrada não pode ser nula ou vazia");
        }
        // Processamento...
    }
}

```

- Definir blocos try catch para facilitar o manejo de erros

```

public class LeitorArquivo
{
    public void LerArquivo(string caminho)
    {
        try
        {
            // Código para leitura do arquivo
        }
        catch (IOException ex)
        {
            Console.WriteLine($"Erro ao ler o arquivo: {ex.Message}");
        }
        finally
        {
            // Limpeza de recursos
        }
    }
}

```

- Usar exceções não verificadas (não precisa declarar exceções que um método pode lançar)

```
public class Calculadora
{
    public int Dividir(int numerador, int denominador)
    {
        if (denominador == 0)
        {
            throw new InvalidOperationException("Denominador não pode ser zero");
        }
        return numerador / denominador;
    }
}
```

- Fornecer contexto com mensagens detalhadas para facilitar o diagnóstico

```
public class BancoDados
{
    // Exemplo ruim: Lançando nova exception
    public void Conectar(string stringConexao)
    {
        try
        {
            // Código para conexão ao banco de dados
        }
        catch (SQLException ex)
        {
            throw new InvalidOperationException("Erro ao conectar ao banco de dados", ex);
        }
    }

    // Exemplo bom: Log personalizado
    public void ConectarBom(string stringConexao)
```

```

    {
        try
        {
            // Código para conexão ao banco de dados
        }
        catch (SqlException ex)
        {
            // Entenda como seu mecanismo de log
            Console.WriteLine("Erro ao conectar ao ba
nco de dados", ex);
            throw;
        }
    }
}
public class SqlException : Exception { }

```

- Usar pelo menos três construtores ao criar as próprias classes de exceção

```

public class ExcecaoSaldoInsuficiente : Exception
{
    public ExcecaoSaldoInsuficiente() : base() { }
    public ExcecaoSaldoInsuficiente(string mensagem)
: base(mensagem) { }
    public ExcecaoSaldoInsuficiente(string mensagem,
Exception inner) : base(mensagem, inner) { }
}

public class ContaBancaria
{
    public void Sacar(decimal quantia)
    {
        if (quantia > Saldo)
        {
            throw new ExcecaoSaldoInsuficiente("Saldo
insuficiente para a operação.");
        }
        Saldo -= quantia;
    }
}

```

```

    }

    public decimal Saldo { get; private set; }
}

```

Boas práticas com Null

- Exemplos
 - Evitar retornar null em métodos, mas utilizar objetos especiais, coleções vazias ou outras técnicas para representar ausência de valor

```

public class Repositorio
{
    // Exemplo Ruim: Retornando null
    public Produto ObterProdutoPorId(int id)
    {
        // Simulação de busca no repositório
        return null; // Produto não encontrado
    }

    // Exemplo Bom: Usando Optional ou retornando uma coleção vazia
    public Produto? ObterProdutoPorIdSeguro(int id)
    {
        // Simulação de busca no repositório
        Produto? produto = /* lógica para obter o produto */ null;

        return produto; // Pode retornar null, mas uso de ? indica que o consumidor deve estar ciente
    }

    // Alternativa: Retornando uma lista vazia para múltiplos resultados
    public List<Produto> ObterProdutos()
    {
        // Simulação de busca no repositório
        List<Produto> produtos = /* lógica para obter

```

```

produtos */ null;

        return produtos ?? new List<Produto>(); // Re
torna uma lista vazia se não houver produtos
    }

    public record Produto(int Id, string Nome);
}

```

- Não utilizar null como argumento ao chamar métodos, mas sim utilizar objetos padrão ou valores especiais, evitando a necessidade de lidar com null no código

```

// Exemplo Ruim: Passando null
public class ProcessadorPedido
{
    // Exemplo Ruim: Aceitando e processando null sem
validação
    public void Processar(Pedido pedido)
    {
        // Processamento do pedido sem validar se é n
ull...
        // Isso é ruim porque permite que null seja p
rocessado
        // e cause falhas no código posteriormente.
        // Falha de execução se pedido for null.
        var nome = pedido.Nome; // Isso causará uma N
ullReferenceException
    }

    // Exemplo Bom: Validando argumento e lançando ex
ceção
    public void ProcessarSeguro(Pedido pedido)
    {
        if (pedido == null)
        {
            throw new ArgumentNullException(nameof(pe
dido), "Pedido não pode ser null.");
        }
    }
}

```

```

    }

    // Processamento do pedido...
    var nome = pedido.Nome; // Isso é seguro porque
    // null foi evitado
    }

    // Exemplo Alternativo: Em algum momento pode ser
    // possível receber null e o código sabe disso
    // Nesse caso o compilador não vai apontar a possibilidade
    // do erro
    public void ProcessarSeguroAlternativo(Pedido? pedido)
    {
        if (pedido == null)
        {
            throw new ArgumentNullException(nameof(pedido),
                "Pedido não pode ser null.");
        }

        // Processamento do pedido...
        var nome = pedido.Nome; // Isso é seguro porque
        // null foi evitado
    }
}
public record Pedido(int Id, string Nome);

```

- C# lidando com null

```

public class Demo
{
    static void Main()
    {
        // Exemplo de tipo nullable
        int? nullableInt = null;
        Console.WriteLine($"Nullable int: {nullableInt}");
    }
}

```

```

        // Verificando se uma variável nullable tem v
    alor
        if (nullableInt.HasValue)
        {
            Console.WriteLine("nullableInt tem um val
or.");
        }
        else
        {
            Console.WriteLine("nullableInt é nulo.");
        }

        // Operador de coalescência nula (??)
        int valorDefault = nullableInt ?? 10;
        Console.WriteLine($"Valor após coalescência n
ula: {valorDefault}");

        // Operador condicional null (?) para acessa
r membros de objetos potencialmente nulos
        string mensagem = null;
        int? tamanhoMensagem = mensagem?.Length;
        Console.WriteLine($"Tamanho da mensagem: {tam
anhoMensagem}");

        // Operador de coalescência nula (??) com str
ings
        string nome = null;
        string nomeOuPadrao = nome ?? "Nome padrão";
        Console.WriteLine($"Nome: {nomeOuPadrao}");

        // Usando o operador null-forgiving (!)
        string? nomeNullable = null;
        // Console.WriteLine(nomeNullable!.Length);
        // Isso lançará uma exceção em tempo de execução se n
omeNullable for nulo

        // Tratamento de nulidade em classes
        Pessoa pessoa = new Pessoa();

```



```

        pessoa.Nome = null; // Nome é um string nullable

        // Acessando propriedades de um objeto potencialmente nulo com o operador condicional null (?)
        string primeiroNome = pessoa.Nome?.Split(' ')[0] ?? "Primeiro nome não disponível";
        Console.WriteLine($"Primeiro nome: {primeiroNome}");

        // Usando o operador is para verificar nulidade
        if (pessoa.Nome is null)
        {
            Console.WriteLine("O nome da pessoa é nulo.");
        }

        // Usando o operador null-forgiving com mais segurança
        if (pessoa.Nome is not null)
        {
            Console.WriteLine($"Comprimento do nome: {pessoa.Nome.Length}");
        }

        // Usando o operador switch com valores nulos
        switch (nullableInt)
        {
            case null:
                Console.WriteLine("nullableInt é nulo no switch.");
                break;
            case int valor:
                Console.WriteLine($"nullableInt tem valor no switch: {valor}");
                break;
        }

```

```

    }
}

// <Project Sdk="Microsoft.NET.Sdk">
//   <PropertyGroup>
//     <OutputType>Exe</OutputType>
//     <TargetFramework>net8.0</TargetFramework>
//     <ImplicitUsings>enable</ImplicitUsings>
//     <Nullable>enable</Nullable>
//   </PropertyGroup>
// </Project>

// Habilitado(<Nullable>enable</Nullable>) :

// Força a consideração de nullabilidade nas variáveis de referência.
// Ajuda a prevenir erros de referência nula com avisos do compilador.
// Requer verificações explícitas de nullabilidade ou uso de operadores como ?..

// Desabilitado(<Nullable>disable</Nullable>) :

// Comportamento tradicional do C#, onde variáveis de referência podem ser nulas sem avisos.
// Maior risco de exceções de referência nula em tempo de execução.

class Pessoa
{
    public string? Nome { get; set; } // Propriedade nullable
}

```

Integração com Código de Terceiros

- Definição: código de terceiro pode ser uma dll, um sistema legado, pacote externo, biblioteca, etc; essa integração requer cuidado para não comprometer a qualidade da aplicação
- Exemplos
 - Encapsular a responsabilidade em um wrapper já que existe muita dependência desses valores; é importante avaliar critérios como confiabilidade, documentação, suporte e comunidade ativa

```
// Exemplo Ruim
// Dependência direta de biblioteca de terceiros
public class ExemploRuim
{
    private static readonly ILog log = LogManager.Get
Logger(typeof(ExemploRuim));

    public void FazerAlgo()
    {
        log.Info("Fazendo algo...");
    }
}

// Exemplo Bom
// Encapsulamento da biblioteca de terceiros
public interface IMeuLogger
{
    void Info(string message);
}

public class MeuLogger : IMeuLogger
{
    private static readonly ILog log = LogManager.Get
Logger(typeof(MeuLogger));

    public void Info(string message)
    {
        log.Info(message);
    }
}
```

```

public class ExemploBom
{
    private readonly IMeuLogger logger;

    public ExemploBom(IMeuLogger logger)
    {
        this.logger = logger;
    }

    public void FazerAlgo()
    {
        logger.Info("Fazendo algo...");
    }
}

```

- Testar o comportamento do código exteno através de testes; contratos claros para comunicação entre os componentes

```

// Exemplo Ruim
// Falta de testes exploratórios
public class ExploracaoRuim
{
    public void ConfigurarLogger()
    {
        var log = LogManager.GetLogger(typeof(ExploracaoRuim));
        log.Info("Logger configurado");
    }
}

// Exemplo Bom
// Testes exploratórios para entender a biblioteca
public class ExploracaoBoa
{
    public void TestarLog4Net()
    {
        var log = LogManager.GetLogger(typeof(ExploracaoBoa));
        log.Info("Testando Log4Net");
    }
}

```

```

caoBoa));
    log.Info("Teste de info");
    log.Warn("Teste de warn");
    log.Error("Teste de erro");
}
}

```

Práticas de Fronteira

- Exemplos
 - Implementar interfaces ou subsistemas que ainda não existem, o que envolve abstrações e contratos para integração; mocks e stubs para simulação de comportamento

```

// Exemplo Ruim
// Desenvolvimento bloqueado pela falta de interface
public class DesenvolvimentoRuim
{
    private Transmitter transmitter;

    public void EnviarDados(string dados)
    {
        //transmitter.Send(dados); // Transmitter ain
da não definido
    }
}

public record Transmitter();

// Exemplo Bom
// Uso de interface temporária para continuar o desen
volvimento
public interface ITransmitter
{
    void Send(string data);
}

public class MockTransmitter : ITransmitter

```

```

{
    public void Send(string data)
    {
        Console.WriteLine($"Mock sending: {data}");
    }
}

public class DesenvolvimentoBom
{
    private readonly ITransmitter transmitter;

    public DesenvolvimentoBom(ITransmitter transmitt
er)
    {
        this.transmitter = transmitter;
    }

    public void EnviarDados(string dados)
    {
        transmitter.Send(dados);
    }
}

```

- Manter separação clara entre o código do projeto e terceiros

```

// Exemplo Ruim
// Dependência direta de biblioteca de terceiros sem
encapsulamento
public class FronteiraRuim
{
    public void RealizarOperacao()
    {
        var cliente = new ApiClient(); // Uso direto
de uma biblioteca externa
        cliente.FazerRequisicao();
    }
}

```

```

// Exemplo Bom
// Encapsulamento da interação com a biblioteca de terceiros
// A ideia aqui além de encapsular é adaptar todo o contexto externo ao interno
public interface IApiClient
{
    void FazerRequisicao();
}

public class ApiClienteWrapper : IApiClient
{
    private readonly ApiClient cliente = new ApiClient();

    public void FazerRequisicao()
    {
        cliente.FazerRequisicao();
    }
}

public class FronteiraBoa
{
    private readonly IApiClient apiCliente;

    public FronteiraBoa(IPApiClient apiCliente)
    {
        this.apiCliente = apiCliente;
    }

    public void RealizarOperacao()
    {
        apiCliente.FazerRequisicao();
    }
}

public class ApiClient()

```

```
{  
    public void FazerRequisicao() { }  
}
```