UTFPR-MD

VALIDAÇÃO DE REGRAS DE NEGÓCIO E USO DE PROPRIEDADES NA PROGRAMAÇÃO ORIENTADA A OBJETOS COM C#

Professor: Everton Coimbra de Araújo

MATEUS S. NEINAS

VICTOR HUGO WILLE

BRUNO HENRIQUE OLIVEIRA GARCIA

```
0 references
class Program
{
    0 references
    static void Main()
    {
        try
        {
            Pessoa pessoa = new Pessoa("João", 17);
            Console.WriteLine($"Pessoa criada: {pessoa.Nome}, {pessoa.Idade} anos");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Erro ao criar pessoa: " + ex.Message);
        }
```

```
public class Pessoa
{
    2 references
    public string Nome { get; }
    2 references
    public int Idade { get; }

    1 reference
    public Pessoa(string nome, int idade)
    {
        if (idade < 18)
            throw new ArgumentException("É necessário ter 18 anos ou mais.");

        Nome = nome;
        Idade = idade;
    }
}
```

REGRAS DE NEGÓCIO

- São condições, restrições ou diretrizes que definem ou restringem como um processo de negócio deve funcionar.
- Elas representam o conhecimento e as políticas da organização, sendo essenciais para garantir que o sistema siga os objetivos e limites do negócio.

Exemplo geral:

Os funcionários de uma loja só podem entrar na loja em horário comercial.

Exemplo :

```
.public bool HorarioValido(DateTime horario){
        return horario.Hour >= 8 && horario.Hour < 18;
}
```

ENCAPSULAMENTO

O encapsulamento consiste em proteger os dados internos da classe contra acessos diretos indevidos.

- As variáveis (atributos) são privadas.
- O acesso a elas é feito somente por métodos públicos.
- Isso permite controlar e validar o que entra e sai da classe.

Exemplo: A data e a lista de horários são privadas. Só podem ser modificadas usando métodos como RegistrarData

```
// Campo privado: não pode ser acessado diretamente de fora da classe
5 references
private decimal saldo;

// Propriedade somente leitura (pública): o titular pode ser lido, mas só definido no construtor
2 references
public string Titular { get; private set; }

// Construtor da classe: define o titular e o saldo inicial
1 reference
public ContaBancaria(string titular, decimal saldoInicial)
{
    Titular = titular;
    saldo = saldoInicial;
}
```

CONCEITO DE PROPRIEDADES

Propriedades são membros de uma classe que fornece acesso controlado a campos privados.

São declaradas usando { get; set; }

Exemplo:
public string Nome { get; set; }

```
1 reference
public class Tarefa
{
    1 reference
    public string Nome { get; set; }
    1 reference
    public DateTime CriadaEm { get; }
    2 references
    public string Status { get; private set; }
    0 references
    public Tarefa(string nome)
    {
        Nome = nome;
        CriadaEm = DateTime.Now;
        Status = "Pendente";
    }

    0 references
    public void Concluir()
    {
        Status = "Concluída";
    }
}
```

Boas Práticas com Propriedades

- Use propriedades automáticas ({ get; set; }) sempre que possível para simplicidade.
- Use modificadores de acesso (private set) para proteger estados internos.
- Não exponha campos diretamente (public string nome; é má prática).
- Propriedades ajudam a manter o encapsulamento e a integridade dos dados.

RegistrarData

- Verifica se a data é válida (DateTime.TryParse);
- Garante que a data não seja no passado;
- Se a regra não for respeitada, lança um erro com throw.new.exception().

Quando os dados não atendem às regras de negócio, usamos throw para interromper a execução e avisar que há um problema.

A exceção pode ser capturada no código externo (try/catch) para mostrar uma mensagem clara ao usuário.

Isso mantém o código mais seguro, organizado e fácil de entender!

```
static void Main()
{
    var compromisso = new Compromisso();

    try
    {
        Console.Write("Digite a data do compromisso (yyyy-MM-dd): ");
        string data = Console.ReadLine();    Converting null literal
        compromisso.RegistrarData(data);    Possible null reference a

        compromisso.ValidarCompromisso();
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("Erros ao registrar o compromisso:");
        Console.WriteLine(ex.Message);
    }
}
```

```
public void RegistrarData(string textoData)
 {
 if (!DateTime.TryParse(textoData, out data))
 {
throw new ArgumentException("Data inválida. Use o formato correto (ex: 05/05/2025).");
 }
 if (data < DateTime.Today)
 {
 throw new ArgumentException("A data não pode estar no passado.");
 }

 Console.WriteLine("Data registrada com sucesso: " + data.ToShortDateString());
```

ValidarCompromisso

Esse método serve para reunir todos os erros acumulados e, se houver algum:

- Cria uma mensagem com todos os erros;
- Lança uma exceção com throw new ArgumentException().

Exemplo:

```
public void ValidarCompromisso()
{
    if (errosDeValidacao.Count > 0)
    {
        throw new ArgumentException(string.Join("; ", errosDeValidacao));
    }

    Console.WriteLine($"Compromisso agendado para {Data:dd/MM/yyyy} às {Hora}");
}
```

OBJECT INITIALIZERS VS CONSTRUTORES EM C#

O QUE SÃO OBJECT INITIALIZERS?

- SÃO UMA FORMA PRÁTICA E RÁPIDA DE CRIAR OBJETOS ATRIBUINDO VALORES DIRETAMENTE ÀS PROPRIEDADES PÚBLICAS COM SET.

O QUE SÃO CONSTRUTORES?

- UM CONSTRUTOR É UM MÉTODO ESPECIAL DE UMA CLASSE QUE É CHAMADO AUTOMATICAMENTE QUANDO O OBJETO É CRIADO;
- ELE SERVE PARA INICIALIZAR O OBJETO E GARANTIR QUE ELE COMECE EM UM ESTADO VÁLIDO.

REQUISITOS PARA USAR OBJECT INITIALIZERS:

- SÓ FUNCIONAM COM PROPRIEDADES PÚBLICAS COM SET ACESSÍVEL;
- NÃO EXECUTAM VALIDAÇÕES AUTOMATICAMENTE — APENAS ATRIBUEM VALORES DIRETAMENTE.

POR QUE USAR CONSTRUTOR É PREFERÍVEL EM ALGUNS CASOS?

- QUANDO HÁ DADOS CRÍTICOS, COMO DATA E HORA, QUE PRECISAM DE VALIDAÇÃO:
- O CONSTRUTOR GARANTE QUE OS VALORES SEJAM VERIFICADOS ANTES DE O OBJETO SER USADOS;
- EVITA A CRIAÇÃO DE OBJETOS INVÁLIDOS.

CONSTRUTOR

```
// Cria a reserva (o construtor faz a validação!)
Reserva reserva = new Reserva(sala, dataHora);
reserva.ExibirDetalhes();
```

OBJECT INITIALIZERS

```
// Inicialização usando Object Initializer
Pessoa pessoa = new Pessoa
{
    Nome = "Ana"
};
```

CRITÉRIO	OBJECT INITIALIZER	CONSTRUTOR
Simplicidade	Mais simples para objetos leves	Mais detalhado
Validação de dados	Não faz validação automaticamente	Pode validar internamente
Segurança do estado	Pode criar objetos inválidos	Garante consistência
Reutilização de lógica	Repetição em cada uso	Lógica centralizada no construtor
Leitura de código	Leitura clara e objetiva	Levemente mais verboso
Recomendado para	DTOs, modelos simples	Objetos com lógica e regras de negócio

BOAS PRÁTICAS REFORÇADAS

- VALIDAÇÃO CENTRALIZADA NA CLASSE DE DOMÍNIO;
- USO DE PRIVATE SET EM PROPRIEDADES CRÍTICAS;
- SEPARAÇÃO ENTRE LÓGICA DE APRESENTAÇÃO E LÓGICA DE NEGÓCIO;
- CONSTRUTORES COM VALIDAÇÃO OBRIGATÓRIA → CONSISTÊNCIA DO SISTEMA;
- CÓDIGO DEVE SER AUTODOCUMENTADO POR MEIO DE NOMES DESCRITIVOS.

```
PUBLIC CLASS TAREFA
{
  // PROPRIEDADES
  PUBLIC STRING NOME { GET; }
  PUBLIC STRING STATUS { GET; PRIVATE SET; } = "PENDENTE";

  // CONSTRUTOR COM VALIDAÇÃO OBRIGATÓRIA
  PUBLIC TAREFA(STRING NOME)
  {
    IF (ISNULLORWHITESPACE(NOME))
      THROW NEW ARGUMENTEXCEPTION("O NOME DA TAREFA NÃO PODE SER VAZIO.");
    NOME = NOME;
  }

  // LÓGICA DE NEGÓCIOS: ALTERAR O STATUS DA TAREFA
  PUBLIC VOID CONCLUIR()
  {
    VALIDARSTATUSALTERACAO();
    STATUS = "CONCLUÍDA";
  }

  PUBLIC VOID CANCELAR()
  {
    VALIDARSTATUSALTERACAO();
    STATUS = "CANCELADA";
  }

  // VALIDAÇÃO CENTRALIZADA PARA GARANTIR QUE O STATUS NÃO SEJA ALTERADO SE JÁ ESTIVER FINALIZADO
  PRIVATE VOID VALIDARSTATUSALTERACAO()
  {
    IF (STATUS == "CONCLUÍDA" || STATUS == "CANCELADA")
      THROW NEW INVALIDOPERATIONEXCEPTION("NÃO É POSSÍVEL MODIFICAR UMA TAREFA JÁ CONCLUÍDA OU
CANCELADA.");
  }

  // MÉTODO PARA EXIBIR INFORMAÇÕES DA TAREFA
  PUBLIC VOID EXIBIRINFORMACOES()
  {
    CONSOLE.WRITELINE($"TAREFA: {NOME} | STATUS: {STATUS}");
  }
}

CLASS PROGRAM
{
  STATIC VOID MAIN()
  {
    TRY
    {
```

CONCLUSÕES

E RECOMENDAÇOES

- USAR PROPRIEDADES COM PRIVATE SET, OU PROPRIEDADES SOMENTE LEITURA, GARANTE QUE O ESTADO DO OBJETO SÓ SEJA ALTERADO DE FORMA CONTROLADA;
- O MODELO DE DOMÍNIO (EX: COMPROMISSO, PEDIDO, RESERVA) DEVE SER RESPONSÁVEL POR VALIDAR E APLICAR AS REGRAS DE NEGÓCIO;
- USE CONSTRUTORES COM VALIDAÇÃO PARA ENTIDADES DE DOMÍNIO CRÍTICAS, RESERVE OBJECT INITIALIZERS PARA TESTES, DTOS OU MODELOS SEM REGRAS COMPLEXAS;
- LANÇAR EXCEÇÕES COM MENSAGENS CLARAS AJUDA O DESENVOLVEDOR A ENTENDER RAPIDAMENTE O QUE DEU ERRADO.

OBRIGADO PELA ATENÇÃO