Novos conceitos de Orientação a Objetos no Sistema Fiesta!!!

Este documento explica os novos conceitos de Orientação a Objetos (OO) e os comandos utilizados no código do sistema **Fiesta!!!**,: `abstract class` , `<T>` , `where T : class, new()` , `protected abstract` , `: BaseCRUD<...>` , e `protected override` . Cada conceito é descrito com sua utilidade no contexto do sistema.

1. `abstract class`

Conceito

Uma **classe abstrata** (`abstract class`) em C# é uma classe que não pode ser instanciada diretamente, ou seja, não é possível criar objetos dela usando `new` . Ela funciona como um modelo ou "molde" para outras classes derivadas (subclasses) que herdam dela. Classes abstratas são usadas para definir comportamentos comuns que várias classes relacionadas devem compartilhar, mas que não devem ser usados diretamente como objetos.

Utilidade no Código

No sistema **Fiesta!!!**, a classe `BaseCRUD<T>` é definida como abstrata:

```
public abstract class BaseCRUD<T> where T : class, new()
```

- **Finalidade**: A classe `BaseCRUD<T>` fornece uma estrutura genérica para operações CRUD (Create, Read, Update, Delete) que podem ser aplicadas a diferentes tipos de dados, como `ClienteDTO` e `TemaDTO` . Ela contém métodos e propriedades comuns que as classes derivadas (`ClienteCRUD` e `TemaCRUD`) reutilizam ou implementam.
- **Por que abstrata?** Como `BaseCRUD<T>` é um modelo genérico e não faz sentido instanciá-la diretamente (sem especificar o tipo de dado), ela é marcada como `abstract` . Isso obriga a criação de classes específicas, como `ClienteCRUD` e `TemaCRUD` , que implementam os detalhes

necessários para seus respectivos tipos de dados.

Exemplo no Código

- A classe `BaseCRUD<T>` define métodos comuns, como `ExecutarCRUD`, `BuscarCodigo`, `IncluirRegistro` e `ExcluirRegistro`, que são compartilhados por todas as classes CRUD.
- Métodos abstratos, como `InicializarCampos` e `GetCodigo`, são declarados sem implementação, forçando as subclasses a fornecê-las:

```
protected abstract void InicializarCampos(); // Define os rótulos dos campos
protected abstract int? GetCodigo(T registro); // Obtém o código do registro
```

- Esses métodos garantem que cada classe derivada defina como os campos e códigos são tratados para seu tipo específico.

Benefícios

- **Reutilização de código**: Evita repetir a lógica de CRUD em várias classes, centralizando-a em `BaseCRUD<T>`.
- **Organização**: Agrupa comportamentos comuns, tornando o código mais claro e fácil de manter.
- **Flexibilidade**: Permite que diferentes tipos de dados (como clientes e temas) usem a mesma estrutura base.

2. `<T>` (Genéricos)

Conceito

O símbolo `<T>` indica o uso de **genéricos** em C#. Genéricos permitem que uma classe, método ou interface seja definida de forma a trabalhar com diferentes tipos de dados sem especificar o tipo exato durante a definição. O `T` é um **parâmetro de tipo**, um espaço reservado que será substituído por um tipo concreto (como `ClienteDTO` ou `TemaDTO`) quando a classe for utilizada.

Utilidade no Código

Na definição de `BaseCRUD<T>`:

```
public abstract class BaseCRUD<T> where T : class, new()
{
    protected List<T> lista; // "Banco de dados"
```

- **Finalidade**: O `<T>` permite que `BaseCRUD` seja reutilizada para gerenciar qualquer tipo de dado que atenda às restrições definidas (ser uma classe com construtor padrão). Por exemplo, `BaseCRUD<ClienteDTO>` gerencia uma lista de clientes, enquanto `BaseCRUD<TemaDTO>` gerencia uma lista de temas.
- A propriedade `lista` é do tipo `List<T>`, o que significa que ela armazena objetos do tipo especificado quando a classe é instanciada (conforme, `List<ClienteDTO>` ou `List<TemaDTO>`).

Exemplo no Código

- Em `ClienteCRUD : BaseCRUD<ClienteDTO>`, o parâmetro `T` é substituído por `ClienteDTO`, fazendo com que `lista` seja uma `List<ClienteDTO>`.
- Isso permite que a lógica genérica de `BaseCRUD` (como buscar, incluir ou excluir registros) seja aplicada a clientes sem necessidade de duplicar código para outros tipos, como temas.

Benefícios

- **Flexibilidade**: Uma única classe (`BaseCRUD`) pode gerenciar diferentes tipos de dados, como clientes e temas, sem reescrever a lógica.
- **Segurança de tipo**: O compilador garante que apenas o tipo correto seja usado (não é possível adicionar um `TemaDTO` a uma `List<ClienteDTO>`).
- **Reutilização**: Reduz a necessidade de criar classes separadas para cada tipo de dado, economizando tempo e effort.

3. `where T : class, new()`

Conceito

A cláusula `where T : class, new()` é uma **restrição de tipo** aplicada ao parâmetro genérico `T`. Ela especifica que:

- **`class`**: O tipo `T` deve ser um tipo de referência (uma classe), não um tipo primitivo como `int` ou `double`.
- **`new()`**: O tipo `T` deve ter um construtor público sem parâmetros (construtor padrão).

Utilidade no Código

Na definição de `BaseCRUD<T>`:

```
public abstract class BaseCRUD<T> where T : class, new()
```

- **Restrição `class`**: Garante que `T` seja uma classe (como `ClienteDTO` ou `TemaDTO`), permitindo o uso de tipos de referência complexos que contêm propriedades e métodos.
- **Restrição `new()`**: Assegura que `BaseCRUD` possa criar instâncias de `T` usando `new T()`, como no trecho:

```
this.registro = new T();
```

- Isso é essencial porque `BaseCRUD` precisa criar registros em branco durante operações CRUD.

Exemplo no Código

- As classes `ClienteDTO` e `TemaDTO` atendem às restrições porque são classes e possuem construtores padrão (sem parâmetros):

```
public ClienteDTO()
{
    Codigo = 0;
    Nome = "";
    Email = "";
    Telefone = "";
}
```

- Sem o construtor padrão, o compilador geraria um erro ao tentar usar `new T()` em `BaseCRUD`.

Benefícios

- **Segurança**: Garante que apenas tipos apropriados sejam usados, evitando erros durante a compilação.
- **Facilidade**: Permite criar objetos dinamicamente sem precisar saber o tipo exato no momento da codificação.
- **Clareza**: Define claramente as exigências para os tipos usados com `BaseCRUD`, facilitando a compreensão do código.

4. `protected abstract`

Conceito

O modificador `protected abstract` é aplicado a métodos em uma classe abstrata:

- **`protected`**: O método só pode ser acessado pela própria classe ou por suas subclasses (classes derivadas).
- **`abstract`**: O método não possui implementação na classe base e deve ser implementado pelas subclasses.

Utilidade no Código

Na classe `BaseCRUD<T>`:

```
protected abstract void InicializarCampos(); // Define os rótulos dos campos
protected abstract int? GetCodigo(T registro); // Obtém o código do registro
```

- **Finalidade**: Esses métodos são declarados como `protected abstract` para obrigar as subclasses (`ClienteCRUD` e `TemaCRUD`) a fornecerem implementações específicas para seus tipos de dados.
- **`InicializarCampos`** : Define os rótulos dos campos específicos para cada tipo ("Código", "Nome", "Email" para clientes ou "Código", "Nome", "Categoria" para temas).
- **`GetCodigo`** : Especifica como obter o código de um registro (acessar a propriedade `Codigo` de `ClienteDTO` ou `TemaDTO`).

Exemplo no Código

Em `ClienteCRUD` :

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código   :", "Nome     :", "Email    :", "Telefone :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}
```

- Este método implementa `InicializarCampos` para definir os rótulos específicos de clientes.
- Em `GetCodigo` :

```
protected override int? GetCodigo(ClienteDTO registro)
{
    return registro.Codigo;
}
```

- Retorna o código do cliente, implementando o método abstrato.

Benefícios

- **Obrigatoriedade**: Garante que as subclasses implementem funcionalidades essenciais, evitando omissões.
- **Encapsulamento**: O acesso `protected` limita o uso dos métodos às subclasses, protegendo a lógica interna.
- **Personalização**: Permite que cada tipo de dado tenha sua própria lógica sem alterar a estrutura geral do CRUD.

5. `: BaseCRUD<...>` (Herança com Genéricos)

Conceito

A notação `: BaseCRUD<ClienteDTO>` indica que a classe `ClienteCRUD` **herda** de `BaseCRUD<T>`, especificando que o parâmetro genérico `T` será substituído por `ClienteDTO`. Isso combina **herança** com genéricos, permitindo que a classe derivada herde toda a funcionalidade da classe base, mas a especialize para um tipo específico de dado.

Utilidade no Código

Na definição de `ClienteCRUD`:

```
public class ClienteCRUD : BaseCRUD<ClienteDTO>
```

- **Finalidade**: `ClienteCRUD` herda toda a lógica genérica de `BaseCRUD` (como os métodos `ExecutarCRUD`, `BuscarCodigo`, `IncluirRegistro`, etc.) e a especializa para gerenciar objetos do tipo `ClienteDTO`.
- A herança permite que `ClienteCRUD` reuse métodos genéricos, enquanto implementa métodos abstratos (como `InicializarCampos`) para lidar com os detalhes específicos de clientes.

Exemplo no Código

- `ClienteCRUD` usa o método herdado `ExecutarCRUD`, que gerencia o fluxo geral de operações CRUD (solicitar código, buscar registro, cadastrar, alterar, excluir).
- A implementação de `InicializarCampos` define campos específicos para clientes:

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código   :", "Nome     :", "Email    :", "Telefone :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}
```

- A propriedade `lista` de `BaseCRUD<T>` se torna `List<ClienteDTO>` em `ClienteCRUD`.

Benefícios

- **Reutilização**: Evita reescrever a lógica de CRUD para cada tipo de dado, usando a implementação genérica de `BaseCRUD`.
- **Especialização**: Permite personalizar o comportamento para tipos específicos, como `ClienteDTO` ou `TemaDTO`.
- **Manutenibilidade**: Alterações na lógica genérica em `BaseCRUD` são automaticamente refletidas em todas as subclasses.

6. `protected override`

Conceito

O modificador `protected override` é usado em uma classe derivada para **substituir** (override) a implementação de um método abstrato ou virtual definido na classe base. O `protected` restringe o acesso à classe base e suas subclasses, enquanto `override` indica que o método está fornecendo uma nova implementação para um método herdado.

Utilidade no Código

Em `ClienteCRUD`:

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código   :", "Nome     :", "Email    :", "Telefone :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}
```

- **Finalidade**: Substitui o método abstrato `InicializarCampos` de `BaseCRUD<T>` com uma implementação específica para clientes, definindo os rótulos dos campos que aparecem na interface do CRUD.
- O `override` é necessário porque `InicializarCampos` é marcado como `abstract` em `BaseCRUD`, exigindo que cada subclasse forneça sua própria implementação.

Exemplo no Código

- Em `TemaCRUD`, o método `InicializarCampos` também é sobrescrito para temas:

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código    :", "Nome      :", "Categoria :", "Disponível:", "Valor     :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}
```

- Aqui, os campos são personalizados para incluir "Categoria", "Disponível" e "Valor", que são específicos para temas.

Benefícios

- **Personalização**: Permite que cada subclasse defina como um método deve funcionar para seu tipo de dado específico.
- **Flexibilidade**: Mantém a estrutura genérica de `BaseCRUD` intacta, permitindo variações específicas nas subclasses.

- **Segurança**: O compilador verifica se o método sobrescrito corresponde a um método abstrato ou virtual na classe base, evitando erros.

7. Polimorfismo

Conceito

Polimorfismo é a capacidade de objetos de diferentes classes serem tratados de forma uniforme por meio de uma interface ou classe base comum, executando implementações específicas de métodos conforme o tipo do objeto. Em C#, o polimorfismo é geralmente implementado por meio de herança, com métodos `abstract` ou `virtual` na classe base que são sobrescritos (`override`) nas subclasses, ou por interfaces. O termo vem do grego, significando "muitas formas", pois um mesmo método pode ter comportamentos diferentes dependendo da classe que o executa.

Utilidade no Código

No sistema **Fiesta!!!**, o polimorfismo está presente na classe abstrata `BaseCRUD<T>` , que define métodos abstratos (`protected abstract`) que as subclasses `ClienteCRUD` e `TemaCRUD` implementam de forma específica. Esses métodos são chamados de forma uniforme pelo método `ExecutarCRUD` , mas executam implementações distintas dependendo da subclasse, demonstrando polimorfismo de subtipo.

Exemplo no Código

Na classe `BaseCRUD<T>` :

```
public abstract class BaseCRUD<T> where T : class, new()
{
    protected abstract void InicializarCampos();
    protected abstract int? GetCodigo(T registro);
    protected abstract void MontarTela(int coluna, int linha);
    protected abstract void EntrarDados(int qual);
    protected abstract void MostrarDados();
```

```
protected abstract void AlterarRegistro();

public void ExecutarCRUD()
{
    while (true)
    {
        this.MontarTela(this.coluna, this.linha); // Chamada polimórfica
        this.registro = new T();
        this.tela.centralizar("Deixe o campo vazio para sair.");
        this.EntrarDados(1); // Chamada polimórfica
        if (this.registro == null || this.GetCodigo(this.registro) == 0) break; // Chamada polimórfica
        bool achou = this.BuscarCodigo(this.GetCodigo(this.registro).Value);
        if (!achou)
        {
            this.tela.centralizar("Registro não encontrado. Deseja cadastrar (S/N): ");
            string resp = Console.ReadLine();
            if (resp.ToLower() == "s")
            {
                this.EntrarDados(2); // Chamada polimórfica
                this.tela.centralizar("Confirma cadastro (S/N) : ");
                resp = Console.ReadLine();
                if (resp.ToLower() == "s")
                {
                    this.IncluirRegistro();
                }
            }
        }
        else
        {
            this.MostrarDados(); // Chamada polimórfica
            this.tela.centralizar("Deseja Voltar/Alterar/Excluir (V/A/E) : ");
            string resp = Console.ReadLine();
            if (resp.ToLower() == "a")
            {
                this.tela.centralizar("Digite apenas o dado que deseja alterar");
```

```
                    this.EntrarDados(2); // Chamada polimórfica
                    this.tela.centralizar("Confirma alteração (S/N) : ");
                    resp = Console.ReadLine();
                    if (resp.ToLower() == "s")
                    {
                        this.AlterarRegistro(); // Chamada polimórfica
                    }
                }
                // ... (código para exclusão)
            }
        }
    }
}
```

- **Métodos Abstratos**: Métodos como `InicializarCampos`, `GetCodigo`, `MontarTela`, `EntrarDados`, `MostrarDados` e `AlterarRegistro` são declarados sem implementação, forçando as subclasses a fornecê-las.
- **Implementações nas Subclasses**:
  - Em `ClienteCRUD`:

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código   :", "Nome     :", "Email    :", "Telefone :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}

protected override int? GetCodigo(ClienteDTO registro)
{
    return registro.Codigo;
}
```

- Em `TemaCRUD`:

```
protected override void InicializarCampos()
{
    this.campos.AddRange(new[] { "Código    :", "Nome      :", "Categoria :", "Disponível:", "Valor     :" });
    this.larguraDados = this.larguraTotal - this.campos[0].Length;
}

protected override int? GetCodigo(TemaDTO registro)
{
    return registro.Codigo;
}
```

- **Polimorfismo em Ação**: Quando `ExecutarCRUD` chama métodos como `InicializarCampos` ou `MontarTela`, a implementação específica da subclasse (`ClienteCRUD` ou `TemaCRUD`) é executada, mesmo que a chamada seja feita em uma referência do tipo `BaseCRUD<T>`.

Como o Polimorfismo Funciona no Fiesta!!!

- O método `ExecutarCRUD` trata todas as subclasses de `BaseCRUD<T>` de forma uniforme, chamando métodos abstratos sem saber qual subclasse está sendo usada.
- Por exemplo, quando executamos:

```
ClienteCRUD clienteCRUD = new ClienteCRUD();
clienteCRUD.ExecutarCRUD();
```

- `InicializarCampos` executa a implementação de `ClienteCRUD`, definindo campos como "Código", "Nome", "Email" e "Telefone".
- Para temas:

```
TemaCRUD temaCRUD = new TemaCRUD();
temaCRUD.ExecutarCRUD();
```

- A mesma chamada a `InicializarCampos` executa a implementação de `TemaCRUD`, definindo campos como "Código", "Nome", "Categoria", "Disponível" e "Valor".
- Isso é polimorfismo porque o mesmo método (`InicializarCampos`) é chamado de forma uniforme, mas o comportamento varia conforme a classe derivada.

Benefícios

- **Reutilização de Código**: A lógica geral do CRUD está centralizada em `BaseCRUD<T>`, permitindo que `ExecutarCRUD` seja usado para gerenciar diferentes tipos de dados (clientes, temas) sem duplicação.
- **Flexibilidade**: Novas classes CRUD (e.g., para "Fornecedores") podem herdar de `BaseCRUD<T>` e implementar os métodos abstratos, sem alterar o código existente.
- **Manutenibilidade**: Alterações em `ExecutarCRUD` afetam todas as subclasses automaticamente, enquanto implementações específicas permanecem isoladas.
- **Consistência**: Garante que todas as operações CRUD sigam o mesmo fluxo (montar tela, entrar dados, buscar, etc.), mas com detalhes personalizados.
- **Extensibilidade**: O uso de genéricos combinado com polimorfismo permite estender o sistema para novos tipos de dados, mantendo a estrutura polimórfica.

Resumo dos Benefícios Gerais

Os conceitos e comandos explicados são fundamentais para tornar o sistema **Fiesta!!!** modular, reutilizável e fácil de manter:

- **Classe abstrata (`abstract class`)**: Fornece uma base comum para operações CRUD, evitando duplicação de código.
- **Genéricos (`<T>`)**: Permitem que a mesma lógica seja aplicada a diferentes tipos de dados (clientes, temas) sem redundância.
- **Restrições (`where T : class, new()`)**: Garantem que os tipos usados sejam classes com construtores padrão, assegurando compatibilidade e segurança.

- **Métodos abstratos (`protected abstract`)**: Forçam as subclasses a implementar detalhes específicos, mantendo a consistência.
- **Herança com genéricos (`: BaseCRUD<...>`)**: Combina reutilização de código com especialização para tipos específicos.
- **Sobrescrita (`protected override`)**: Permite personalizar comportamentos mantendo a estrutura geral do sistema.
- **Polimorfismo**: Permite tratar objetos de diferentes subclasses de forma uniforme, executando implementações específicas, o que aumenta a flexibilidade e escalabilidade.

Impacto no Sistema Fiesta!!!

No contexto do **Fiesta!!!**, esses conceitos permitem gerenciar clientes, temas e aluguéis de forma eficiente e consistente:

- A classe `BaseCRUD<T>` centraliza a lógica de CRUD, reduzindo a repetição de código.
- O uso de genéricos (`<T>`) e restrições (`where T : class, new()`) torna o sistema flexível para lidar com diferentes tipos de dados.
- Métodos abstratos e sobrescrita (`protected abstract` e `protected override`) permitem que `ClienteCRUD` e `TemaCRUD` personalizem a interface e a lógica para seus respectivos tipos.
- A herança com genéricos (`: BaseCRUD<ClienteDTO>`) garante que a lógica genérica seja reutilizada, enquanto os detalhes específicos são implementados nas subclasses.
- O polimorfismo permite que `ExecutarCRUD` execute operações CRUD de forma uniforme, com comportamentos específicos para cada tipo de dado.

Esses conceitos ajudam a construir sistemas escaláveis e organizados, onde o código é modular e suporta diferentes tipos de dados. Tente criar um outro CRUD, como por exemplo, de **Itens de Festa** (onde será possível cadastrar copos, enfeites de mesa, toalhas, guardanapos etc.) e veja como a tarefa de programar se torna mais fácil e rápida.