# Curso de C#
# Aula 4

*Professores*
Célio Alencar de Assis
Fábio Duarte Machado

# *O que será abordado Hoje*

- Classes Genericas
- Delegação
- Eventos
- Delegates
- Métodos anonimos
- Expressões Lambdas
- Action Delegate
- Predicate Delegate
- Func Delegate

# Generic Types

- Permite parametrizar código, sendo que o parametro não é um objeto, é um tipo(type);
- Permite criar estruturas mais flexíveis e reutilizáveis;
- Reduz acoplamento entre classes;
- Reduz o uso de interfaces ou classes abstratas;
- Reduz a necessidade do uso de tipos mais genéricos para efetuar operação sobre vários tipos (ex. Object);
- Maior performance pois o tipo é resolvido estaticamente, não necessitando “type casts” dinamicos;

# Generic Com Coleções (List<T>)

```
//Lista não genérica, coloco qualquer coisa
var lista = new ArrayList();
lista.add(10);
lista.add("manuel");
lista.add(new Button());

//Lista tipada só coloco pessoa
var  listaPessoa = new List<Pessoa>();
listaPessoa.add(new Pessoa(){ID = 1, Nome =
"Fulano"});
listaPessoa.add(new Pessoa(){ID = 2, Nome =
"Beltrano"});
listaPessoa.add(new Pessoa(){ID = 3, Nome =
"Siclano"});
```

# Definindo Classes Genéricas

```
public  class Pilha<T>
{
        int pos = 0;
        T[] itens = new T[100];
        public void Push(T item)
        {
                itens[pos] = item;
                pos ++;
        }
        public T Pop()
        {
                pos--;
                return itens[pos];
        }
}
```

# Usando Classes Genéricas

```
var pes1 = new Pessoa() {ID=1,
Nome="Fulano"};
var pes2 = new Pessoa() {ID=2,
Nome="Beltrano"};

Pilha<Pessoa> pilha = new Pilha<Pessoa>();

pilha.Push(pes1);
pilha.push(pes2);

Console.WriteLine(pilha.pop().Nome);
Console.WriteLine(pilha.pop().Nome);
```

1. Criar uma classe Genérica conforme o exemplo do slide que possa ser criado uma pilha de Cliente e outra de Fornecedor e fazer com que seja listado o Nome das duas pilhas.

2. Criar uma Classe Genérica para o Padrão Repository

# Delegação

# Delegação

- Oque é delegar?
- Não se pode fazer tudo sozinho;
- Significado: “Dar Algo que é legalmente próprio, algo sobre o qual se tem autoridade, dar uma tarefa para que outra pessoa execute”;
- Na POO, a delegação é usada quando um objeto repassa a outro objeto a execução de determinada tarefa, procedimento, resolução de um problema.

# Delegação

- Ideal para não sobrecarregar uma única classe com múltiplas  responsabilidades, criando uma Classe Deus (God Class), que realiza tudo;
- Neste sentido, é importante saber delegar tarefas a outras classes;
- Cuidado para não delegar demais e causar o efeito contrário, criando assim uma Classe Preguiçosa (Lazy Class);

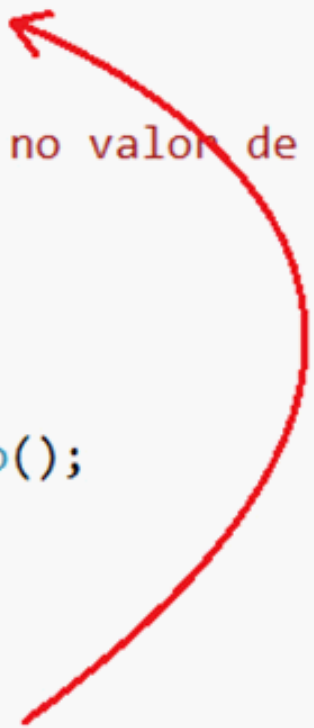
# Delegação

- Prefira a delegação em privado (private), ocultando o objeto ao qual irá ser repassada a tarefa, ou seja, a delegação propriamente dita;
- Além disso, delegação privada permite modificar o objeto ao qual se delega, sem causar efeitos de dependência;
- Delegação privada reduz acoplamento e permite simular herança múltipla;

# Delegação

```
public class Boleto
{
    public void Pagar(double valor)
    {
        Console.WriteLine("Boleto pago no valor de " + valor);
    }
}
public class Pedido
{
    private Boleto _boleto = new Boleto();

    public void Fechar(double valor)
    {
        // delegação...
        this._boleto.Pagar(valor);
    }
}
```



1. Criar a aplicação demonstrada em sala baseada no slide anterior.

# Eventos

# Eventos

- Podem ser disparados por alguma ação do usuário na aplicação ou definidos pelo próprio sistema;
- Exemplos de eventos de UI: Click de um Button, Texto alterado em uma TextBox;
- Eventos definidos pelo próprio sistema usam a palavra reservada event;
- Esse event deve ser declarado como sendo de um tipo *delegate* (estudado a seguir);

# Eventos

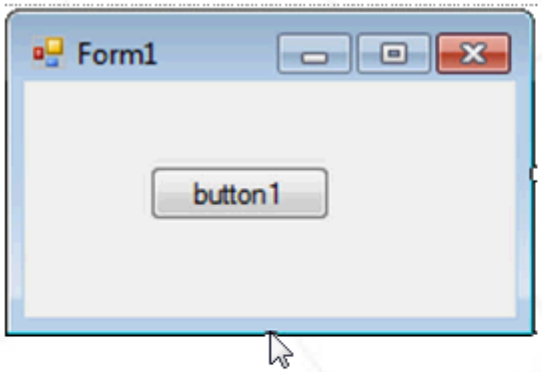
- Uma propriedade de uma determinado objeto que seja de um tipo *delegate* (um evento) pode apontar então para um ou mais métodos;
- Neste caso dizemos que um evento nada mais é que um ponteiro para um método;
- Quando um evento dispara mais de um método, ele é chamado *multi-cast event*;

# Eventos

- O método que manipula, ou seja aponta para o evento, é chamado de *event handler*;
- O método deve ter os mesmos parametros declarados na assinatura do tipo delegate;
- Neste sentido é muito comum confundir nomenclaturas, o que é evento, propriedade, método, delegate, handler, etc. Vamos a um exemplo prático.

# Eventos

- Exemplo de delegação de evento de uma aplicação Win Forms:

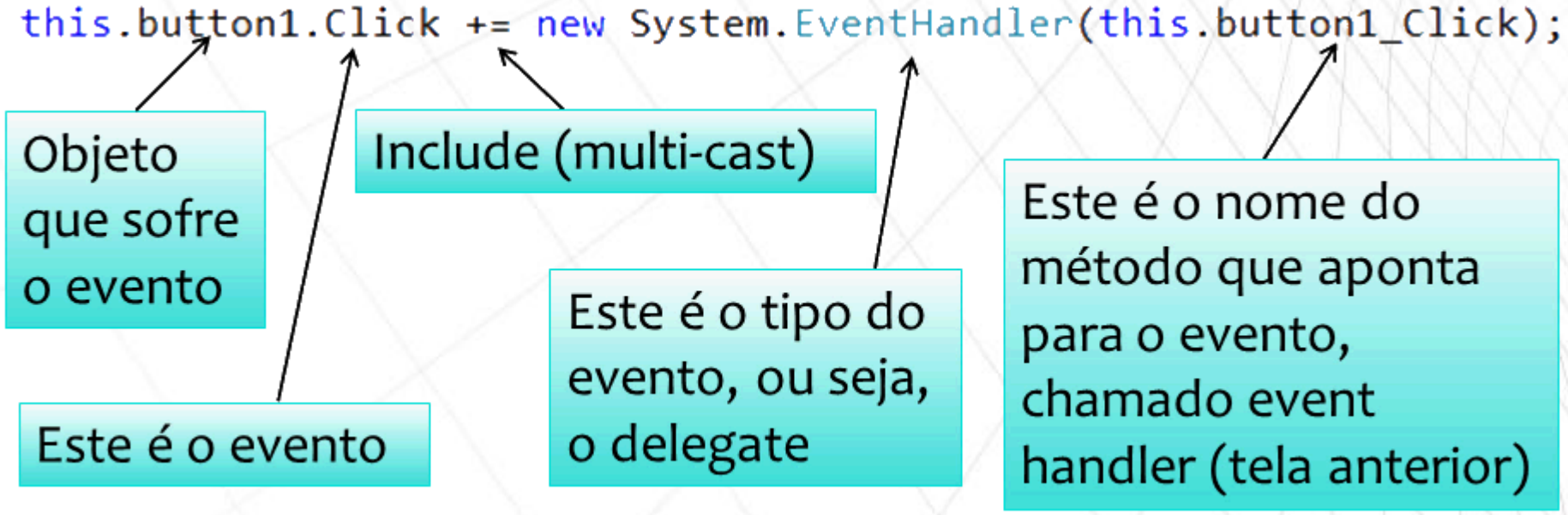



Isto é um método, não é um evento

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Teste");
}
```

# Eventos

- Agora vamos analisar o vínculo feito pela IDE no arquivo de designer do formulário:



# Delegates

# Delegates

- Permite definir um ponteiro para um método que terá uma chamada com vínculo ou atribuição dinâmica;
- Por exemplo, podemos chamar um evento “Pagar” que pode assumir diferentes formas.
- Permite criar pontos de injeção de código em frameworks abertos;

# Delegates

- O exemplo a seguir mostra uma classe “Pedido” que tem um método chamado “pagar”, que invoca o evento “fechar”, que pode apontar para um método que tenha a assinatura definida pelo delegate;
- Uma classe externa pode então injetar código dinamicamente que é invocado pela Classe “Pedido”;
- Nesse sentido, dizemos que a classe “Pedido” **delega** a responsabilidade do fechamento (método fechar) para um método externo de um tipo normal, anônimo ou expressão lambda;

# Delegates

```
public delegate void FecharNotify(double valor);

public class Pedido
{
    // evento do tipo delegate
    public event FecharNotify Fechar;
    // método simples
    public void Pagar(double valor)
    {
        // chama o evento do tipo delegate
        Fechar(valor);
        // código do pagamento aqui...
    }
}
```

Ponto de injeção de código, pode ser um método normal, anônimo ou expressão lambda

# Delegates – Método Simples

```
static void Fechamento(double valor)
{
    Console.Write("Pago valor de " + valor);
}

static void Main(string[] args)
{
    var obj = new Pedido();
    // atribuição de método nomeado ao evento
    // ligação tardia, dinâmica
    obj.Fechar += new FecharNotify(Fechamento);
   // o método Pagar internamente vai chamar
    // o método Fechamento declarado acima
    obj.Pagar(1250.50);
```

Injetando código...

# Métodos Anônimos

# Métodos Anônimos

- Permite atribuir uma porção de código a um delegate, sem que para isso seja preciso definir um método com nome, que esteja declarado na classe;
- Torna a programação mais fluente;
- Mesmo efeito do exemplo anterior com delegates, exceto que não é necessário definir formalmente um método na atribuição;

# Métodos Anônimos

```
var obj = new Pedido();
// atribuição de código anônimo a evento
obj.Pagar  += delegate (double valor) {
    Console.Write("Pago valor de " + valor);
};
// o método Fechar internamente vai chamar
// o bloco de código acima
obj.Fechar(1250.50);
```

Bloco de código, somente corpo do método

# Expressões lambda

# Expressões lambda

- É uma forma mais resumida de atribuir código de métodos anônimos a eventos(delegates), tornando a programação ainda mais enxuta e fluente;

# Delegates

```
public delegate void FecharNotify(double valor);

public class Pedido
{
    // evento do tipo delegate
    public event FecharNotify Fechar;
    // método simples
    public void Pagar(double valor)
    {
        // chama o evento do tipo delegate
        Fechar(valor);
        // código do pagamento aqui...
    }
}
```

Ponto de injeção de código, pode ser um método normal, anônimo ou expressão lambda

# Delegates – Método Simples

```
static void Fechamento(double valor)
{
    Console.Write("Pago valor de " + valor);
}

static void Main(string[] args)
{
    var obj = new Pedido();
    // atribuição de método nomeado ao evento
    // ligação tardia, dinâmica
    obj.Fechar += new FecharNotify(Fechamento);
   // o método Pagar internamente vai chamar
    // o método Fechamento declarado acima
    obj.Pagar(1250.50);
```

Injetando código...

# Métodos Anônimos

```
var obj = new Pedido();
// atribuição de código anônimo a evento
obj.Pagar  += delegate (double valor) {
    Console.Write("Pago valor de " + valor);
};
// o método Fechar internamente vai chamar
// o bloco de código acima
obj.Fechar(1250.50);
```

Bloco de código, somente corpo do método

# Expressões lambda

```
static void Main(string[] args)
{
    var obj = new Pedido();
    // atribuição de método anônimo a evendo usando expressão lambda
    obj.Pagar += v => MessageBox.Show("Boleto Pago no valor de " + v.ToString());
    // o método Fechar internamente vai chamar
    // o bloco de código acima
    obj.Fechar(1250.50);
}
```

# Action<T> Delegate

- Representa um método que tem um único parâmetro e não retorna valor;

```
public delegate void Action<in T>(T obj);
```

# Action<T> Delegate

```
List<Cliente> clientes = new List<Cliente>
    {
        new Cliente{ID=1,Nome="Fulano"},
        new Cliente{ID=2, Nome="Beltrano"},
        new Cliente{ID=3, Nome="Fábio"},
        new Cliente{ID=4, Nome="Marcio"}
    };
```

```
private void btnAction_Click(object sender, EventArgs e)
{
    Action<Cliente> minhaAction = new Action<Cliente>(delegate(Model.Cliente c) {
        MessageBox.Show("O Nome do Cliente é: " + c.Nome);
    });

    clientes.ForEach(minhaAction);
}
```

# Predicate<T> Delegate

- Representa um método que define um conjunto de critérios e determina se o objeto especificado atende a este critério;

```
public delegate bool Predicate<in T>(T obj);
```

# Predicate<T> Delegate

```
List<string> Nomes = new List<string> { "Fulano", "Beltrano", "Fábio", "Márcio" };
foreach (var nome in Nomes)
{
    if (nome.Contains('r'))
        MessageBox.Show(nome);
}
MessageBox.Show("Usando Predicate");
Predicate<string> mValidaNomes = new Predicate<string>(n => n.Contains('r'));
foreach(var nome in Nomes.FindAll(mValidaNomes))
{
    MessageBox.Show(nome);
}
```

# Func<T,TResult> Delegate

- Representa um método que tem um parâmetro e retorna um valor do tipo especificado no parâmetro TResult.

```
public delegate TResult Func<in T, out TResult>(T arg);
```

# Func<T,TResult> Delegate

```
Func<Cliente, bool> mFunBool = new Func<Cliente, bool>(c => c.Nome.Contains('r'));

foreach (var c in clientes.Where(mFunBool))
{
    MessageBox.Show("O Cliente que contém a letra r é: " + c.Nome);
}
```