Delegates

Prof. Ricardo Frohlich da Silva

Delegate

- Delegate é um tipo de dado que representa referências a métodos com uma assinatura específica.
- Ele permite que você trate métodos como objetos, fornecendo uma maneira flexível de realizar chamadas de método indiretas.
- Delegates são frequentemente usados para implementar callbacks, eventos e outros padrões de design.

Para que serve um delegate?

- Encapsulação de Métodos:
  - Delegates permitem encapsular um método com uma assinatura específica, tornando-o um objeto que pode ser passado, armazenado e chamado como parâmetro para outros métodos ou classes.
- Callback e Eventos:
  - Delegates são frequentemente usados para implementar callbacks, onde um método é passado como argumento para outro método e é chamado de volta quando necessário.
  - Em eventos, delegates são usados para notificar objetos quando um evento específico ocorre.
- Flexibilidade:
  - Delegates proporcionam flexibilidade, permitindo que você escreva código mais genérico e extensível.

Delegate – Encapsulamento de métodos

```
// Definindo um delegate chamado OperacaoMatematica
public delegate int OperacaoMatematica(int a, int b);

internal class Program
{
    static void Main(string[] args)
    {
        // Instanciando o delegate com um método correspondente
        OperacaoMatematica somaDelegate = new OperacaoMatematica(Somar);

        // Chamando o método através do delegate
        int resultado = somaDelegate(3, 5);

        Console.WriteLine("Resultado da Soma: " + resultado);
    }
    static int Somar(int a, int b)
    {
        return a + b;
    }
}
```

Delegate

- OperacaoMatematica é um delegate que representa métodos que aceitam dois inteiros como parâmetros e retornam um inteiro.
- Somar é um método que corresponde à assinatura do delegate.
- O delegate é instanciado com o método Somar, e então, chamamos o delegate como se fosse um método normal.

```csharp
// Definindo um delegate chamado OperacaoMatematica
public delegate int OperacaoMatematica(int a, int b);

internal class Program
{
    static void Main(string[] args)
    {
        // Instanciando o delegate com um método correspondente
        OperacaoMatematica somaDelegate = new OperacaoMatematica(Somar);

        // Chamando o método através do delegate
        int resultado = somaDelegate(3, 5);

        Console.WriteLine("Resultado da Soma: " + resultado);

        OperacaoMatematica multiplicaDelegate = new OperacaoMatematica(Multiplicar);

        resultado = multiplicaDelegate(3, 5);

        Console.WriteLine("Resultado da Multiplicação: " + resultado);

    }
    static int Somar(int a, int b)
    {
        return a + b;
    }

    static int Multiplicar(int a, int b)
    {
        return a * b;
    }
}
```

Action

- Action é uma das classes de delegados pré-definidas que podem ser usadas para representar métodos que não retornam nenhum valor.

- A Action pode ser parametrizada com até quatro tipos de parâmetros.

Action

```
static void Main(string[] args)
{
    // Exemplo de uso do Action com 4 parâmetros
    Action<int, double, string, bool> meuDelegate = MeuMetodo;

    // Chamando o método encapsulado pelo delegate
    meuDelegate(42, 3.14, "Olá, mundo!", true);

    Console.ReadKey();
}

// Método que corresponde à assinatura do Action com 4 parâmetros
static void MeuMetodo(int parametro1, double parametro2, string parametro3, bool parametro4)
{
    Console.WriteLine("Parâmetros: "+ parametro1+", "+parametro2 + ", "+parametro3 + ", "+parametro4);
}
```

Action

- Action<int, double, string, bool> é um delegate que representa um método que aceita quatro parâmetros (um int, um double, uma string e um bool) e não retorna nenhum valor.

- Em seguida, instanciamos esse delegate com o método MeuMetodo que corresponde à assinatura esperada.

- Ao chamar o delegate (meuDelegate), ele executa o método encapsulado (MeuMetodo) com os valores fornecidos como argumentos.

Func

- Func é outra classe de delegados pré-definida que pode ser usada para representar métodos que retornam um valor.

- A classe Func pode ser parametrizada com até 16 tipos de parâmetros, sendo que o último tipo é sempre o tipo de retorno.

Func

```
static void Main(string[] args)
{
    // Exemplo de uso do Func com até 4 parâmetros e retorno do tipo string
    Func<int, double, string, bool, string> meuDelegate = MeuMetodo;

    // Chamando o método encapsulado pelo delegate
    string resultado = meuDelegate(42, 3.14, "Olá, mundo!", true);
    Console.WriteLine("Resultado: " + resultado);

    Console.ReadKey();
}

// Método que corresponde à assinatura do Func com até 4 parâmetros e retorno do tipo string
static string MeuMetodo(int parametro1, double parametro2, string parametro3, bool parametro4)
{
    return "Parâmetros: " + parametro1 + ", " + parametro2 + ", " + parametro3 + ", " + parametro4;
}
```

Func

- Func<int, double, string, bool, string> é um delegate que representa um método que aceita quatro parâmetros (um int, um double, uma string e um bool) e retorna um valor do tipo string.
  - O último tipo especificado (string) é o tipo de retorno.
- Instanciamos esse delegate com o método MeuMetodo, que corresponde à assinatura esperada.
- Ao chamar o delegate, ele executa o método encapsulado com os valores fornecidos como argumentos e retorna o resultado.
- Note que o número de tipos na assinatura do Func determina o número de parâmetros, e o último tipo é o tipo de retorno.

Event

- Um evento é uma maneira de fornecer notificação de que algo interessante aconteceu dentro de um objeto.

- Eventos são frequentemente utilizados em conjunto com delegados.

Event

```
public delegate void MeuDelegate(string mensagem);

// Classe que contém o evento e métodos associados
public class Publicador
{
    // Declarando o evento usando o delegate
    public event MeuDelegate MeuEvento;

    // Método para simular algum evento ocorrendo
    public void OcorreuAlgoImportante()
    {
        Console.WriteLine("Algo importante aconteceu!");

        // Verificando se há assinantes para o evento antes de invocá-lo
        if (MeuEvento != null)
        {
            // Disparando o evento
            MeuEvento("O evento ocorreu!");
        }
    }
}
```

Event

```
public class Assinante
{
    // Método que será chamado quando o evento ocorrer
    public void TratarEvento(string mensagem)
    {
        Console.WriteLine("Evento tratado: "+mensagem);
    }
}
```

Event

```
static void Main(string[] args)
{
    // Criando instâncias do publicador e do assinante
    Publicador publicador = new Publicador();
    Assinante assinante = new Assinante();

    // Associando o método TratarEvento ao evento MeuEvento
    publicador.MeuEvento += assinante.TratarEvento;

    // Simulando ocorrência de algo importante (dispara o evento)
    publicador.OcorreuAlgoImportante();
}
```

Event

- MeuDelegate é o delegate que define a assinatura dos métodos que podem ser associados ao evento.
- Publicador é a classe que contém o evento (MeuEvento) e um método (OcorreuAlgoImportante) que simula algum evento importante.
- Assinante é uma classe que possui um método (TratarEvento) que será chamado quando o evento ocorrer.
- No método Main, uma instância do Publicador é criada, e um método da instância do Assinante é associado ao evento MeuEvento.
- Quando OcorreuAlgoImportante é chamado, verifica-se se há assinantes para o evento antes de invocar o delegate associado a ele.
- Essencialmente, os eventos fornecem uma camada de segurança adicional para garantir que o delegate associado só seja invocado se houver pelo menos um assinante para o evento.

Mensagerias

- Em C#, a mensageria refere-se à comunicação entre diferentes partes de um sistema de software por meio de mensagens.

- Isso é comumente usado em arquiteturas distribuídas para permitir a troca de informações entre componentes de forma assíncrona e desacoplada.

- A mensageria é uma abordagem poderosa para criar sistemas escaláveis e flexíveis, uma vez que os componentes podem se comunicar sem depender diretamente uns dos outros.

Mensagerias

- Existem várias maneiras de implementar a mensageria em C#, mas uma abordagem comum é usar um sistema de mensagens baseado em fila. Abaixo estão alguns conceitos e elementos chave associados à mensageria em C#:
- Fila de Mensagens:
  - Uma fila de mensagens é uma estrutura de dados que armazena mensagens na ordem em que foram recebidas.
  - As mensagens na fila podem ser processadas por diferentes partes do sistema, permitindo uma comunicação assíncrona.
- Produtor e Consumidor:
  - O produtor é responsável por criar e enviar mensagens para a fila.
  - O consumidor é responsável por receber e processar as mensagens da fila.

Mensagerias

- Serialização de Mensagens:
  - As mensagens frequentemente precisam ser serializadas antes de serem enviadas para a fila e desserializadas ao serem recebidas.
- Padrões de Mensageria:
  - Padrões comuns incluem mensageria ponto a ponto, publicação/assinatura e filas de trabalho.
- Bibliotecas de Mensageria:
  - Existem várias bibliotecas e frameworks em C# que facilitam a implementação de sistemas de mensageria, como o RabbitMQ, Apache Kafka, Azure Service Bus, entre outros.

Mensagerias

- Serialização de Mensagens:
  - As mensagens frequentemente precisam ser serializadas antes de serem enviadas para a fila e desserializadas ao serem recebidas.
- Padrões de Mensageria:
  - Padrões comuns incluem mensageria ponto a ponto, publicação/assinatura e filas de trabalho.
- Bibliotecas de Mensageria:
  - Existem várias bibliotecas e frameworks em C# que facilitam a implementação de sistemas de mensageria, como o RabbitMQ, Apache Kafka, Azure Service Bus, entre outros.

Mensagerias

- A mensageria pode ser implementada usando o padrão de eventos e delegados para permitir a comunicação entre objetos de maneira desacoplada.

- Vamos criar um exemplo simples usando esse conceito.

```
// Definindo um argumento personalizado para a mensagem
public class MensagemEventArgs : EventArgs
{
    public string Conteudo { get; set; }

    public MensagemEventArgs(string conteudo)
    {
        Conteudo = conteudo;
    }
}
```

Mensagerias

```
public class ReceptorMensagens
{
    // Método que será chamado quando uma mensagem for recebida
    public void AoReceberMensagem(object sender, MensagemEventArgs e)
    {
        Console.WriteLine("Mensagem Recebida: "+e.Conteudo);
    }

}
```

Mensagerias

```
// Classe que envia mensagens
public class EmissorMensagens
{
    // Delegado para o evento de mensagem
    public delegate void MensagemEventHandler(object sender, MensagemEventArgs e);

    // Evento associado ao delegado
    public event MensagemEventHandler MensagemEnviada;

    // Método para enviar uma mensagem
    public void EnviarMensagem(string conteudo)
    {
        Console.WriteLine($"Enviando mensagem: {conteudo}");

        // Disparando o evento de mensagem
        OnMensagemEnviada(conteudo);
    }

    // Método protegido para chamar o evento
    protected virtual void OnMensagemEnviada(string conteudo)
    {
        // Verifica se há assinantes do evento
        MensagemEnviada?.Invoke(this, new MensagemEventArgs(conteudo));
    }
}
```

```
// Classe que envia mensagens
public class EmissorMensagens
{
    // Delegado para o evento de mensagem
    public delegate void MensagemEventHandler(object sender, MensagemEventArgs e);

    // Evento associado ao delegado
    public event MensagemEventHandler MensagemEnviada;

    // Método para enviar uma mensagem
    public void EnviarMensagem(string conteudo)
    {
        Console.WriteLine($"Enviando mensagem: {conteudo}");

        // Disparando o evento de mensagem
        OnMensagemEnviada(conteudo);
    }

    // Método protegido para chamar o evento
    protected virtual void OnMensagemEnviada(string conteudo)
    {
        // Verifica se há assinantes do evento
        MensagemEnviada?.Invoke(this, new MensagemEventArgs(conteudo));
    }
}
```

Mensagerias

```
static void Main(string[] args)
{
    // Criando instâncias do emissor e receptor
    EmissorMensagens emissor = new EmissorMensagens();
    ReceptorMensagens receptor = new ReceptorMensagens();

    // Associando o método AoReceberMensagem ao evento MensagemEnviada
    emissor.MensagemEnviada += receptor.AoReceberMensagem;

    // Enviando uma mensagem
    emissor.EnviarMensagem("Olá, isso é uma mensagem!");

    // Aguardando entrada para visualizar a saída
    Console.ReadLine();

}
```

Mensagerias

Neste exemplo, a classe EmissorMensagens envia mensagens e a classe ReceptorMensagens recebe essas mensagens.

O padrão de eventos e delegados é usado para estabelecer a comunicação entre essas duas classes de maneira flexível e desacoplada.

Lambda

- As expressões lambda em C# permitem escrever funções anônimas de forma concisa.

```
// Declarando um delegate
delegate int MeuDelegate(int x, int y);
static void Main(string[] args)
{
    // Usando uma expressão lambda para criar uma instância do delegate
    MeuDelegate somar = (x, y) => x + y;

    // Chamando o delegate
    int resultado = somar(3, 5);
    Console.WriteLine("Resultado: "+resultado);
}
```

Lambda – Em uma classe

```
public class Calculadora
{
    // Método com expressão lambda
    public int Somar(int x, int y) => x + y;
}
```

```
static void Main(string[] args)
{
    // Criando uma instância da classe
    Calculadora calculadora = new Calculadora();

    // Chamando o método com expressão lambda
    int resultado = calculadora.Somar(3, 5);
    Console.WriteLine("Resultado: "+resultado);
}
```

Lambda – Método anônimo

```
static void Main(string[] args)
{
    // Usando um método anônimo com expressão lambda
    Func<int, int, int> somar = (x, y) => x + y;

    // Chamando o método anônimo
    int resultado = somar(3, 5);
    Console.WriteLine("Resultado: "+resultado);
}
```

Lambda – Em uma lista

```
static void Main(string[] args)
{
    var numeros = new[] { 1, 2, 3, 4, 5 };
    var quadrados = Array.ConvertAll(numeros, x => x * x);

    Console.Write("Quadrados: ");
    foreach (var quadrado in quadrados)
    {
        Console.Write(quadrado+" ");
    }
}
```

Lambda - Event

```
public class Publicador
{
    // Declarando um evento com expressão lambda
    public event Action<string> MeuEvento = mensagem => Console.WriteLine("Evento: "+mensagem);

    // Método que dispara o evento
    public void DispararEvento(string mensagem)
    {
        MeuEvento?.Invoke(mensagem);
    }
}
```

Lambda - Event

```
static void Main(string[] args)
{
    // Criando uma instância do publicador
    Publicador publicador = new Publicador();

    // Associando um assinante ao evento usando expressão lambda
    publicador.MeuEvento += mensagem => Console.WriteLine("Assinante: "+mensagem);

    // Disparando o evento
    publicador.DispararEvento("Alguma coisa aconteceu!");
}
```