Delegates

Centro de Cálculo
Instituto Superior de Engenharia de Lisboa

F. Miguel Carvalho (mcarvalho@cc.isel.ipl.pt)

Agenda

- Ponteiros para função em C/C++
- Objectos função em C++
- *Delegates* em C#

Agenda

- Ponteiros para função em C/C++
- Objectos função em C++
- *Delegates* em C#

Exemplo em C++: Processar *array* usando função genérica

```
char toLower(char ch) {
        if (isalpha(ch) != 0)
                return ch + ' ';
        return ch;
}
// Sintaxe 1
void process(char* arr, char (*proc)(char)) {
        if (proc == 0) {
                cout << "Function doesn't exist" << endl;
                return;
        }

        char* ptr = arr;
        while (*ptr != '\0') {
                *ptr = proc(*ptr);
                ++ptr;
        }
}
```

⇔ `char proc(char)`

Exemplo em C++: Processar *array* usando função genérica

```
int main() {
        char (*fp)(char); // Apontador para função
        // Atribuir valor ao ponteiro
        fp = toLower; // <=> fp = &toLower;

        // Obter carácter  'A'
        char ch = fp('A'); // <=> char ch = (*fp)('A');
        cout << ch << endl;

        char arr[] = {'B','O','M',' ','D','I','A'};
        process(arr, fp);
        //process(arr, toLower);
        //process(arr, &toLower);
        //process(arr, 0);  // 0 <=> null

        // Imprimir array
        ...
}
```

Usando typedef

```
typedef char (*FP)(char);

// Sintaxe 2
void process(char* arr, FP proc) {
        // Código igual ao anterior
        ...
}


int main() {
        // Apontador para função usando typedef
        FP fp;
        fp = toLower;
        //fp = &toLower;
        ...
}
```

Ponteiros para métodos estáticos (C++)

```
class X {
   public:
        static char toLower(char ch);
        ...
};
char X::toLower(char ch) {
        if (isalpha(ch) != 0)
                return ch + ' ';
        return ch;
}
...
int main() {
        FP fp;
        fp = X::toLower;
        // Ou:
        //fp = &X::toLower;
        ...
}
```

demo

Ponteiros para função

Demo 01

CENTRO DE CÁLCULO
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Agenda

- Ponteiros para função em C/C++
- Objectos função em C++
- *Delegates* em C#

Objectos-função em C++

- As classes podem definir o *overload* do operador *chamada a função*
  - `void operator()( parâmetros ...)`

- As instâncias (objectos) de um tipo que defina o *overload* do operador chamada a função designam-se objectos-função e usam a seguinte sintaxe de invocação do operador:

```
MyClass obj();
obj(parâmetros); // <=> obj.operator()(parâmetros)
```

  - os objectos podem ser invocados como se fossem funções

Objectos-função em C++

```
class Counter {
        int c, sum;
public:
        Counter() :  c(0), sum(0) {}

        void operator()(int value) {
                ++c;
                sum += value;
        }
        int getC() {
                return c;
        }
        int getSum() {
                return sum;
        }
};
```

Objectos-função em C++

Instância passada por cópia

Optimização: passar por referência

```
Counter process(int arr[], int len, Counter c) {
        for (int i = 0; i < len; ++i)
                //c(arr[i]);
                // operador pode ser invocado explicitamente
                c.operator()(arr[i]);
        return c;
}

int main() {
        int arr[] = {1, 2, 3, 4, 5};

        Counter c = process(arr, 5, Counter());

        cout << "Count " << c.getC() << " values" << endl;
        cout << "Sum " << c.getSum() << " values" << endl;
        return 0;
}
```

demo

Objectos função

Demo 02

CENTRO DE CÁLCULO
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Objectos-função em C++

- Vantagens:
  - Dado que o operador é um método de instância, podem usar-se campos da instância para armazenar os resultados de processamento
  - Os operadores podem ter qualquer tipo e número de parâmetros e qualquer tipo de retorno
  - A *Standard Template Library* (STL) do C++ está desenhada tendo em conta a utilização de objectos função

Agenda

- Ponteiros para função em C/C++
- Objectos função em C++
- *Delegates* em C#

Delegates

- Um *delegate* define um tipo cujas instâncias encapsulam um ponteiro para função
- Pode armazenar métodos de instância ou estáticos e com qualquer acessibilidade
- Se o método for de instância, o *delegate* contém a referência para o respectivo objecto
- Suporta cadeia de *delegates*

Definição e utilização

```
delegate int DT(int i);
class Del {
        public static int f1(int i) {
                Console.WriteLine("f1 called i={0}", i);
                return 10;
        }
        public static int f2(int i) {
                Console.WriteLine("f2 called i={0}", i);
                return 20;
        }
        public static void Main() {
                DT d = f1;
                d(1);
                // d.Invoke(1);
                d = f2;
                d(2);
        }
}
```

demo

Delegates

Demo 03

Delegates

- Quando se define a linha de código:

```
internal delegate void Feedback(Int32 value);
```

- O compilador gera o seguinte tipo:

```
internal class Feedback : System.MulticastDelegate {
   // Constructor
   public Feedback(Object object, IntPtr method);
   // Method with same prototype as specified by the source code
   public virtual void Invoke(Int32 value);
   // Methods allowing the callback to be called asynchronously
   public virtual IAsyncResult BeginInvoke(Int32 value,
   AsyncCallback callback, Object object);
   public virtual void EndInvoke(IAsyncResult result);
 }
```

Delegates

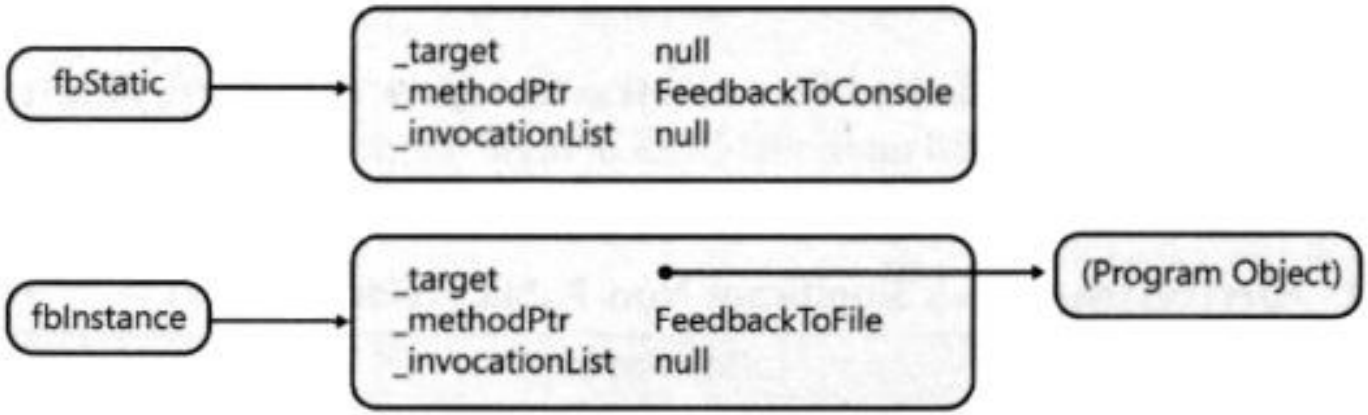
Table 15-1 MulticastDelegate's Significant Non-Public Fields

Field	Type	Description
_target	**System.Object**	When the delegate object wraps a static method, this field is **null.** When the delegate objects wraps an instance method, this field refers to the object that should be operated on when the callback method is called. In other words, this field indicates the value that should be passed for the instance method's implicit **this** parameter.

Field	Type	Description
_methodPtr	**System.IntPtr**	An internal integer the CLR uses to identify the method that is to be called back.
_invocationList	**System.Object**	This field is usually **null.** It can refer to an array of delegates when building a delegate chain (discussed later in this chapter).

Delegates

- `Feedback fbStatic = new Feedback(Program.FeedbackToConsole);`
- `Feedback fbInstance = new Feedback(new Program().FeedbackToFile);`



demo

Delegates

Demo 03 – Del3.cs

Resumo

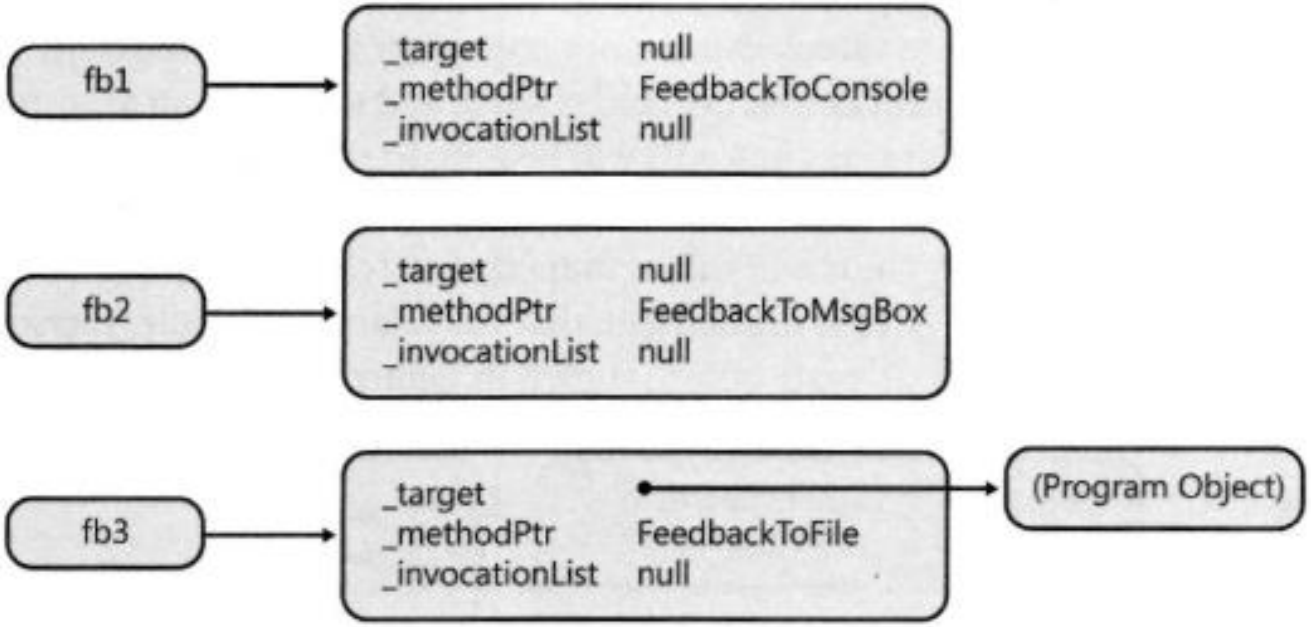
- *Delegates* suportam encadeamento:
  - método estático `System.Delegate.Combine`
  - operador += em C# usa `Combine`

- Resultado do encadeamento de duas instâncias de *delegate* é uma terceira instância *delegate*:
  - As instâncias (não as referências) de *delegate* são imutáveis
  - quando invocada, resulta na invocação de todas as instâncias de *delegate* da cadeia

Cadeias de *delegates* (1)

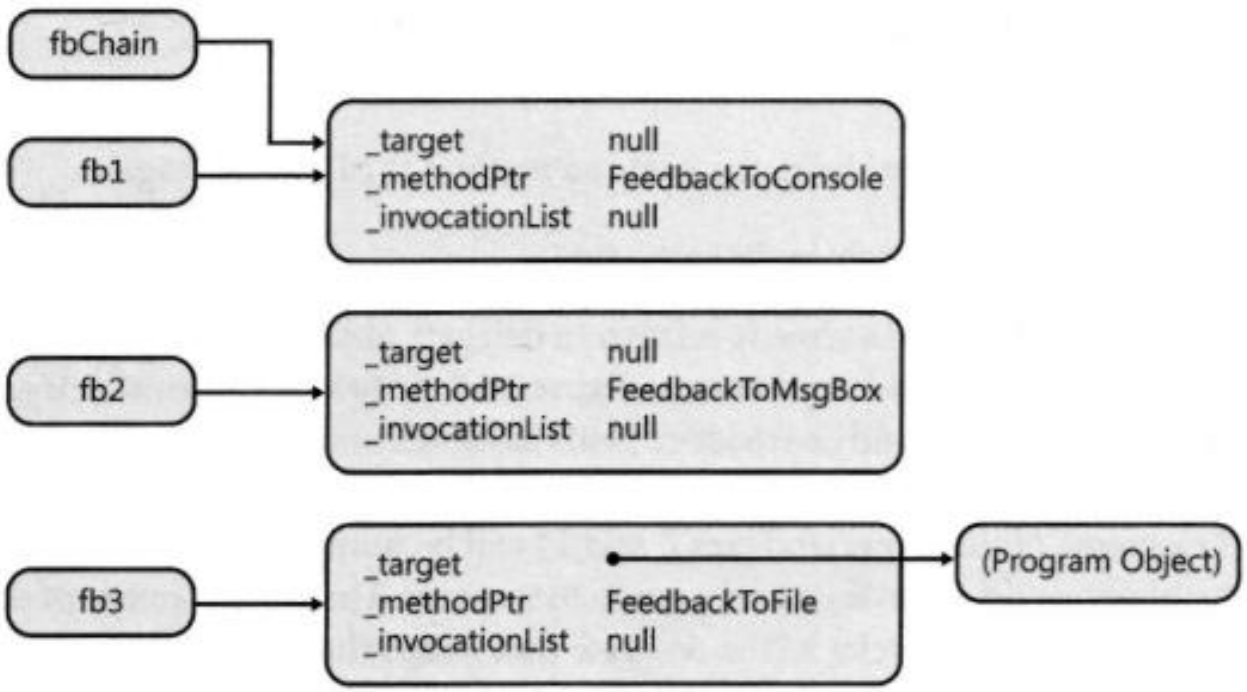
```
...

Feedback fb1 = FeedbackToConsole;
Feedback fb2 = FeedbackToMsgBox;
Feedback fb3 = p.FeedbackToFile;

// Criação de cadeia de delegates
Feedback fbChain = null;
fbChain += fb1;
fbChain += fb2;
fbChain += fb3;

fbChain(10);
```

Cadeias de *delegates* (2)



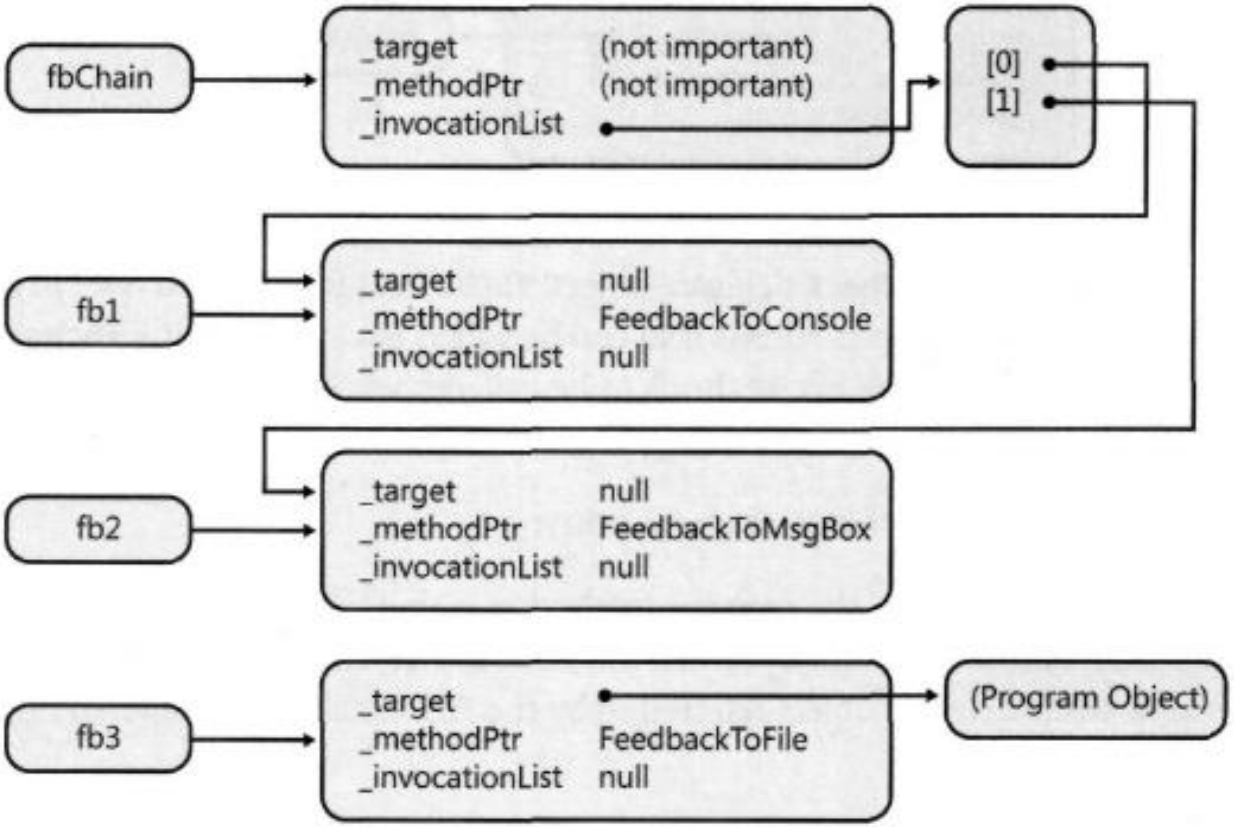
Cadeias de *delegates* (3)

```
Feedback fbChain = null;
fbChain += fb1;
// fbChain = (Feedback)Delegate.Combine(fbChain, fb1)
```



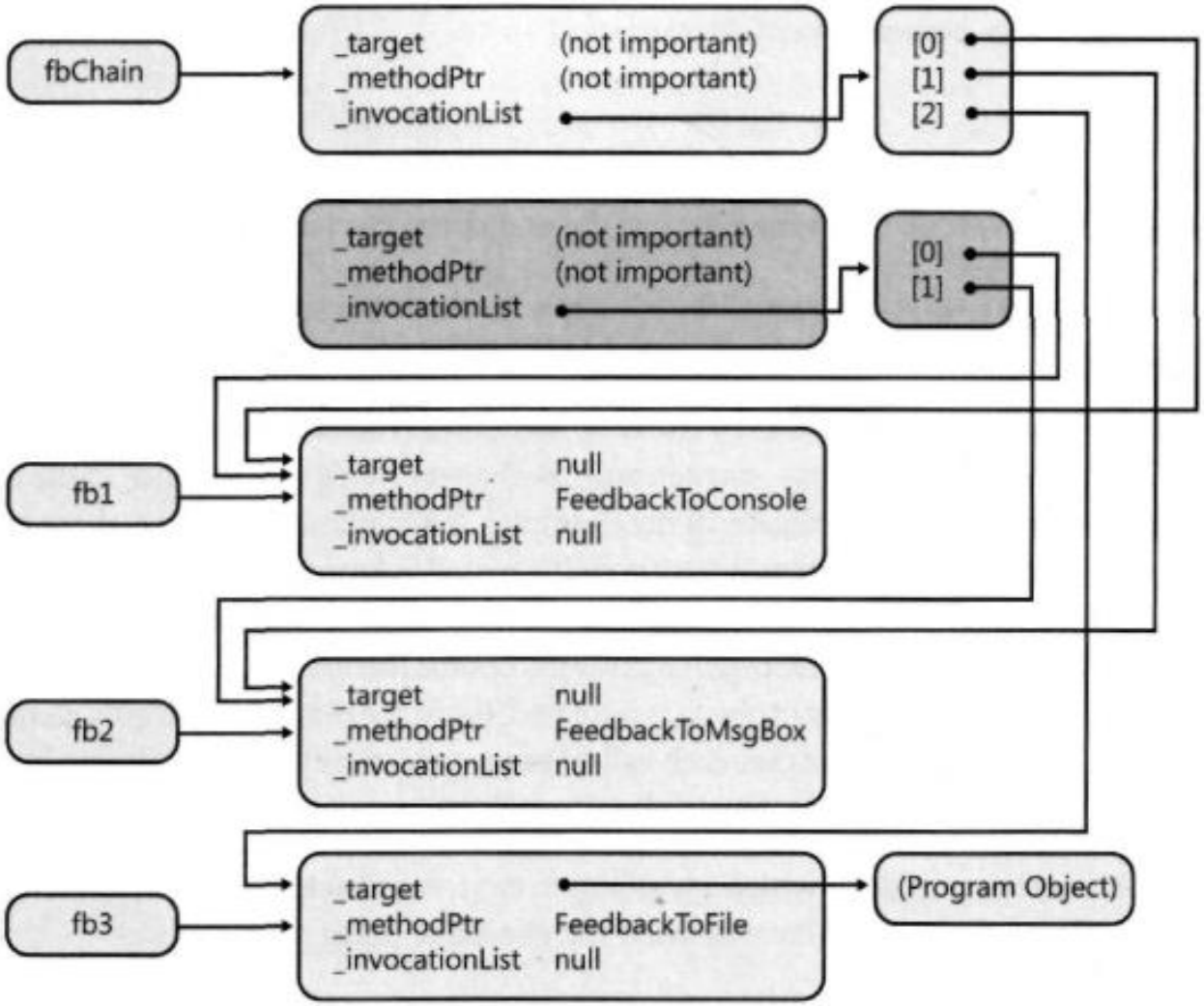
Cadeias de *delegates* (4)

```
...
fbChain += fb2;
```



Cadeias de *delegates* (5)

```
fbChain += fb3;
```



Pseudo-código da função Invoke

```
public void Invoke(Int32 value) {
   Delegate[] delegateSet = _InvocationList as Delegate[];
   if (delegateSet != null) {
   // This delegate's array indicates the delegates
   // that should be called
      foreach (Feedback d in delegateSet)
         d(value); // Call each delegate
   }
   else {
      // This delegate identifies a single method to be called back
      // Call the callback method on the specified target object.
      _methodPtr.Invoke(_target, value);
     // The line above is an approximation of the actual code.
     // What really happens cannot be expressed in C#.
   }
}
```

Propriedades Target e Method

- A classe Delegate define duas propriedades públicas *readonly*
  - Target e Method
- Target devolve a referência armazenada no campo _target
- A propriedade Method devolve uma instância de MethodInfo obtida a partir de _methodPtr
  - MethodInfo é um tipo que representa um método
    - faz parte do sistema de tipos de reflexão

```
Boolean DelegateRefersToInstanceMethodOfType(
        MulticastDelegate d, Type type) {
  return((d.Target != null) && d.Target.GetType() == type);
}
Boolean DelegateRefersToMethodOfName(
        MulticastDelegate d, String methodName) {
  return(d.Method.Name == methodName);
}
```

Lista de invocação

- É possível aceder programaticamente à lista de invocação
  - A classe `MulticastDelegate` disponibiliza o método de instância `GetInvocationList`

```
public abstract class MulticastDelegate : Delegate
{
    // Creates a delegate array where each elements refers
    // to a delegate in the chain.
    public sealed override Delegate[] GetInvocationList();
}

Utilização:

foreach (Feedback fb in fbChain.GetInvocationList())
    // aplica o valor param ao pipeline de delegates
    param = fb(param);
```

Covariância e contra-variância

- O compilador de C# e o CLR suportam *covariância* e *contra-variância* de tipos referência quando associam um método a um *delegate*
- *Covariância* significa que o tipo de retorno do método pode ser um tipo derivado do tipo de retorno do *delegate*
- *Contra-variância* significa que um método pode receber um parâmetro que é um tipo base do tipo de parâmetro do *delegate*

Covariância e contra-variância

```
public class Vehicle { ... }
public class Car : Vehicle { ... }
public class Bike : Vehicle { ...  }

delegate Vehicle DelegateA();
delegate void DelegateB(Bike b);

public static class Examples {
  public static Car MethodA() { ... }
  public static void MethodB(Vehicle v) { ... }
  public static void DelegateVarianceExample() {
    DelegateA delA = MethodA;
    DelegateB delB = MethodB;
    Vehicle v = delA(); // Ok
    delB( new Bike() ); // Ok
    delB( new Car() ); // Erro
    delB( new Vehicle () ); // Erro
    // O tipo do argumento tem que ser compatível com Bike
  } }
```

demo

Covariância e contra-variância

Demo 04

CENTRO DE CÁLCULO
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Covariância e contra-variância - Conclusão

- Co-variância e contra-variância tornam os *delegates* mais flexíveis

- Na contra-variância, pode associar-se um método que recebe um parâmetro mais geral, `MethodB(Vehicle)`, a um *delegate* que recebe um parâmetro mais específico (`Bike`)

  – Se tiver um método geral que sabe processar `Vehicles` posso usá-lo como callback com qualquer delegate que receba um tipo compatível, `del1(Bike)`, `del2(Car)`, ...

  – Contudo, quando se invoca tem que se respeitar o tipo especificado no *delegate*: `del1(Bike)` só é possível passar instâncias compatíveis com `Bike`

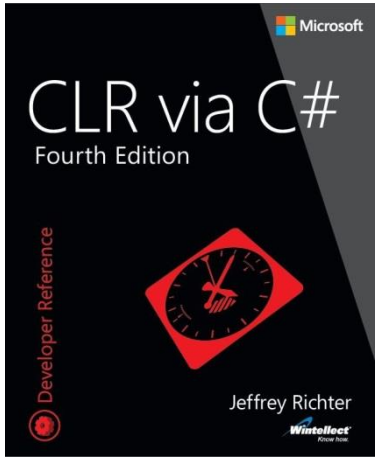
Delegates - Resumo

- Versão *Object-Oriented* e *type-safe* de ponteiro para função;
  - ponteiro para método de tipo ou de instância.
- Tipos *delegate:*
  - derivados de `System.Delegate` e de `System.MulticastDelegate`;
  - definem assinatura de método.
- Instâncias de tipos *delegate* referem:
  - um método com a assinatura definida no tipo *delegate* (`_methodPtr`);
  - um objecto do tipo que contém o método (ou *null*, no caso de métodos estáticos) (`_target`).
- Parametrização com comportamento:
  - Por exemplo: indicar critério de comparação a um algoritmo genérico de ordenação.
- Notificações
  - Por exemplo: indicar reacção a botão pressionado em aplicação visual.

Referências

Jeffrey Richter, “CLR via C#, Second Edition”,

Microsoft Press; 4nd edition, 2012



- Don Box, “Essential .NET, Volume I: The Common Language Runtime ”,
  Addison-Wesley Professional, 1st edition, 2002