Curs07

Tipul delegate

Tipul `delegate` permite încapsularea metodelor (ca referinţă) în obiecte.

Este similar pointerilor la funcţii din C++, dar de data aceasta varianta `object oriented`, prin intermediul unei referinţe. Dacă scriem o aplicatie în care se doreşte executarea unor anumite acţiuni, pentru care cunoaştem `semnătura` acestora (tip returnat, listă parametrii), dar nu ne interesează neapărat de implementarea lor (aceasta va putea fi stabilită ulterior), putem utiliza tipul `delegate`.

În acest fel aplicaţia capătă un caracter general, în sensul că acţiunea poate fi oricând modificată|optimizată|schimbată.

Sintaxă

```
public delegate tip_returnat nume_delegate([lista_param]);
```

Exemplu

```
public delegate int Operation(int a, int b);
```

Acest `delegate` poate încapsula orice funcţie (metodă) care returnează `double` şi are doi parametrii de tip `int`.

Cum se ataşează o metodă la un `delegat` (instanţierea delegatului)

```
namespace DelegateApp {
  class Methods {
    public static int Add(int a, int b) {
      return a + b;
    }
    public static int Mult(int a, int b) {
      return a * b;
    }
  }

  public delegate int Operation(int a, int b);
  public delegate int Operation1();

  class InstanceMethods {
    public int A { get; set; }
    public int B { get; set; }
```

```
    public int Add() {
      return A + B;
    }
    public int Mult() {
      return A * B;
    }
  }

  class Program {
    static void Main(string[] args) {
      // op este un obiect de tip delegat
      Operation op;
      op = new Operation(Methods.Mult);

      // Apelarea delegatului se face ca si cum am apela metoda incapsulata
      Console.WriteLine(op(3, 4));
      InstanceMethods obj = new InstanceMethods() { A = 10, B = 20 };
      Operation1 op1 = new Operation1(obj.Add);
      Console.WriteLine(op1());

      Console.ReadKey();
    }
  }
}
```

Observaţie: Un `delegate` poate încapsula de fapt mai multe metode (`multicast` cu `+` sau `+=`)

```
static void Main(string[] args) {
  // op este un obiect de tip delegat
  Operation op;
  op = new Operation(Methods.Mult);
  op += Methods.Add;

  // Apelarea delegatului se face ca si cum am apela metoda incapsulata
  Console.WriteLine(op(3, 4));

  //InstanceMethods obj = new InstanceMethods() { A = 10, B = 20 };
  //Operation1 op1 = new Operation1(obj.Add);
  //Console.WriteLine(op1());

  Console.ReadKey();
}
```

Observaţie: Un delegat se implementează de fapt ca o clasa in C#. La declararea unui delegat se generează o clasă derivată din clasa `MulticastDelegate`.

Metode anonime

Orice `delegat`, pentru a putea fi utilizat, trebuie să fie iniţializat (cu o referinţă către o metodă). Dacă nu dorim să avem o clasă specială pentru metoda respectivă, se poate utiliza ad-hoc, la momentul respectiv, o metodă anonimă.

```
namespace DelegateAppNew {
  public delegate void Log(string msg);

  class Program {
```

```
    static void Main(string[] args) {
      Log log = delegate(string m) {
        Console.WriteLine(m);
      }; // Aceasta se numeste metoda anonima

      log("Hello, this is a console logger!");
      log = delegate(string m) {
        File.WriteAllText("Log.txt", m);
      };
      log("Hello, this is a file logger!");

      Console.ReadKey();
    }
  }
}
```

Expresii lambda

C#, pentru a simplica lucrul cu tipuul delegate, a definit niste delegati de forma urmatoare:

Action, Action, Action<T1,T2>,... (pana la 16 param.)

Func, Func<T,R>, Func<T1,T2,R>,....

Delegatii de tip Action incapsuleaza metode (cu zero sau mai multi param care returneaza void)
Delegatii de tip Func incapsuleaza metode (cu zero sau mai multi param care returneaza o anumita valoare)

```
// 1.
class Program {
  static void Main(string[] args) {
    Action<string> log = m => Console.WriteLine(m); // Aceasta se numeste
expresie lambda

    log("Hello, this is a console logger!");
    log = m => { File.WriteAllText("Log.txt", m); };
    log("Hello, this is a file logger!");

    Console.ReadKey();
  }
}

// 2.
class Program {
  public static void Log(Action<string> log, string msg) {
    if (log != null)
      log(msg);
  }

  static void Main(string[] args) {
    //Action<string> log = m => Console.WriteLine(m); // Aceasta se numeste
expresie lambda
    //log("Hello, this is a console logger!");

    Log(m => Console.WriteLine(m), "Hello, this is a console logger!");
    //log = m => { File.WriteAllText("Log.txt", m); };
    //log("Hello, this is a file logger!");
```

```
    Log(m => File.WriteAllText("Log.txt", m), "Hello, this is a file logger!");

    Console.ReadKey();
  }
}

// 3.
class Program {
  public static void Log(Action<string> log, string msg) {
    if (log != null)
      log(msg);
  }

  static void Main(string[] args) {
    //Action<string> log = m => Console.WriteLine(m); // Aceasta se numeste
expresie lambda
    //log("Hello, this is a console logger!");

    //Log(m => Console.WriteLine(m), "Hello, this is a console logger!");
    //log = m => { File.WriteAllText("Log.txt", m); };
    //log("Hello, this is a file logger!");
    //Log(m => File.WriteAllText("Log.txt", m), "Hello, this is a file logger!");

    Func<string> f = () => { return "cocolino"; };
    Func<int, string> f1 = (x) => "hello, " + x;
    Console.WriteLine(f());
    Console.WriteLine(f1(25));

    Console.ReadKey();
  }
}
```

Metode extinse

Pe scurt, o metodă extinsş permite extinderea tipurilor deja existente cu noi metode, fără modificarea codului sursă al acestora. Sunt declarate doar în clase statice, cu referinţa `this`.

De exemplu, ca să extindem clasa `string` cu o nouă metodă `m`, procedăm astfel:

```
public static Extension {
  public static void m(this string s) {
    //...
  }
}
```