Delegati, događaji i lambde

Veliki deo korisničkog interfejsa se bazira na događajima. Kako bi događaji funkcionisali, potrebno je specificirati koje metode je moguće pozvati, što se radi putem delegata i lambdi.

Tipovi delegata

Na delegat se donekle može gledati kao na interfejs sa jednom metodom. Delegat specificira potpis metode, a kada imamo instancu delegata, možemo je pozvati kao metodu sa istim potpisom. Delegati imaju i druge odlike, ali mogućnost pozivanja metoda sa određenim potpisom predstavlja razlog postojanja koncepta delegata. Delegati sadrže referencu na metodu, kao i referencu na ciljani objekat nad kojim bi tu metodu trebalo pozvati.

Tipovi delegata se deklarišu pomoću ključne reči **delegate**. Mogu se pojavljivati zasebno, ili ugnježđeni unutar neke klase, kao što je prikazano u sledećem primeru:

```
namespace DelegateArticle
{
    public delegate string FirstDelegate (int x);

    public class Sample
    {
        public delegate void SecondDelegate (char a, char b);
    }
}
```

Ovaj kod deklariše dva tipa delegata. Prvi je **DelegateArticle.FirstDelegate**, koji prihvata jedan argument tipa **int** i vraća **string** vrednost. Drugi je **DelegateArticle.Sample.SecondDelegate**, koji ima dva ulazna **char** parametra i nema povratnu vrednost.

Ključna reč **delegate** ne znači uvek deklaraciju delegatskog tipa. Ista ključna reč se upotrebljava za kreiranje instanci delegatskog tipa, upotrebom anonimnih metoda.

Tipovi delegata koji su ovde deklarisani izvode se iz **System.MulticastDelegate** klase. Svaki tip delegata ima jedan konstruktor i tri dodatne metode: **Invoke**, **BeginInvoke** i **EndInvoke**.

```
public string Invoke (int x);
public System.IAsyncResult BeginInvoke(int x, System.AsyncCallback callback, object state);
public string EndInvoke(IAsyncResult result);
```

Ovi članovi će biti spominjani u nastavku dokumenta.

Instance delegata

Kao što je već spomenuto, najvažniji podaci unutar bilo koje instance delegata jesu metoda na koju se delegat odnosi, i referenca nad kojom se poziva metoda(ciljani objekat). Za statičke metode, nema potrebe za navođenjem ciljanog objekta.

Da bi se kreirala instanca delegata koristi se izraz za kreiranje delegata, koji ima formu ***new delegate-type (expression)***. Izraz(*eng. expression*) mora biti drugi delegat, istog tipa, ili grupa metoda – ime metode i opciono ciljani objekat, specificirani kao da se metoda poziva ali bez argumenata i zagrada. Neki primeri su dati u nastavku:

```
// The following two creation expressions are equivalent,
// where InstanceMethod is an instance method in the class
// containing the creation expression (or a base class).
// The target is "this".
FirstDelegate d1 = new FirstDelegate(InstanceMethod);
FirstDelegate d2 = new FirstDelegate(this.InstanceMethod);
// Here we create a delegate instance referring to the same method
// as the first two examples, but with a different target.
FirstDelegate d3 = new FirstDelegate(anotherInstance.InstanceMethod);

// This delegate instance uses an instance method in a different class,
// specifying the target to call the method on
FirstDelegate d4 = new FirstDelegate(instanceOfOtherClass.OtherInstanceMethod);

// This delegate instance uses a static method in ths class containing
// the creation expression (or a base class).
FirstDelegate d5 = new FirstDelegate(StaticMethod);

// This delegate instance uses a static method in a different class
FirstDelegate d6 = new FirstDelegate(OtherClass.OtherStaticMethod);
```

Konstruktor delegata ima dva parametra, **object** i **IntPtr**. Objekat predstavlja referencu na ciljani objekat(ili **null** vrednost kod statičkih metoda), a **IntPtr** je pokazivač na metodu.

Ono što treba napomenuti jeste da delegati mogu da se referenciraju na metode i ciljane objekte, koji obično ne bi bili vidljivi u trenutku poziva. Na primer, privatna metoda može biti upotrebljena kako bi se napravila instanca delegata, koja kasnije može biti vraćena putem javnog polja(člana). Takođe, ciljani objekat instance delegata može biti objekat o kojem pozivalac ne zna ništa. Međutim, potrebno je da ciljani objekat, kao i metoda, budu pristupačni prilikom kreiranja instance delegata. Dozvole pristupa se zanemaruju prilikom pozivanja instance delegata, što nas dovodi i do sledećeg poglavlja.

Pozivanje instance delegata

Instance delegata se pozivaju kao i same metode. Primer poziva delegata **d1** sa ulaznom vrednošću **10** dat je u nastavku.

```
string result = d1(10);
```

Metoda na koju se odnosi instanca delegata se poziva nad ciljanim objektom(ukoliko postoji), i vraća se rezultat. Sledi primer programa koji prikazuje delegat sa statičkom metodom, kao i delegat sa instancnom metodom. Imena klasa su ostavljena kako bi bilo jasno kako se referenciraju metode iz drugih klasa, iako to nije prikazano u ovom primeru.

```
using System;

public delegate string FirstDelegate (int x);

class DelegateTest
{
    string name;

    static void Main()
    {
        FirstDelegate d1 = new FirstDelegate(DelegateTest.StaticMethod);

        DelegateTest instance = new DelegateTest();
        instance.name = "My instance";
        FirstDelegate d2 = new FirstDelegate(instance.InstanceMethod);

        Console.WriteLine (d1(10)); // Writes out "Static method: 10"
        Console.WriteLine (d2(5));  // Writes out "My instance: 5"
    }

    static string StaticMethod (int i)
    {
        return string.Format ("Static method: {0}", i);
    }

    string InstanceMethod (int i)
    {
        return string.Format ("{0}: {1}", name, i);
    }
}
```

C# sintaksa predstavlja skraćenu verziju pozivanja **Invoke** metode, obezbeđene od strane svakog tipa delegata. Delegati takođe mogu biti pokretani asinhrono upotrebom **BeginInvoke**/**EndInvoke** metoda.

Kombinovanje delegata

Delegati mogu biti kombinovani tako da se prilikom poziva delegata pozove čitava lista metoda – potencijalno sa različitim ciljanim objektima. Kada je ranije napomenuto da delegat sadrži ciljani objekat i metodu, vršeno je određeno pojednostavljenje. To sadrže instance delegata koje predstavljaju jednu metodu, odnosno **jednostavni delegati**. Alternativa su instance delegata koje predstavljaju listu jednostavnih delegata, istog tipa(sa istim potpisom). Ovi delegati se zovu **kombinovani delegati**. Kombinovani delegati mogu biti kombinovani međusobno, stvarajući veliku listu jednostavnih delegata.

Bitno je znati da su instance delegata nepromenljive(*eng. immutable*). Svako njihovo kombinovanje stvara novu instancu delegata, koja predstavlja novu listu ciljanih objekata/metoda za pozivanje.

Kombinovanje dve instance delegata se obično vrši pomoću znaka +, kao da su instance delegata stringovi ili brojevi. Oduzimanje instanci se vrši pomoću znaka –. Treba napomenuti da oduzimanje kombinovanog delegata od drugog funkcioniše po principu lista. Ukoliko lista za oduzimanje nije pronađena u originalnoj listi, rezultat je samo originalna lista. U suprotnom, uklanja se poslednja pojava liste. Ovo se najbolje vidi na primerima. Umesto koda, dati su prosti primeri sa listama jednostavnih delegata **d1**, **d2** i **d3**. Na primer, delegat **[d1, d2, d3]** predstavlja kombinovani delegat, koji poziva **d1**, **d2** i **d3**, kada se izvrši. Prazna lista se predstavlja kao **null** vrednost, umesto kao prava instanca delegata.

Izraz	Rezultat
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] - d1	d2
[d1, d2] - d2	d1
[d1, d2, d1] - d1	[d1, d2]
[d1, d2, d3] - [d1, d2]	d3
[d1, d2, d3] - [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] - [d1, d2]	[d1, d2, d3]
[d1, d2] - [d1, d2]	null

Instance delegata mogu biti kombinovane i upotrebom statičke metode **Delegate.Combine**, oduzimanje može biti izvršeno pmoću statičke metode **Delegate.Remove**. C# kompajler konvertuje operatore + i – u pozive ovih metoda. Zbog toga što su ove metode statičke, mogu da rade i sa **null** vrednostima.

Operatori za sabiranje i oduzimanje uvek rade kao deo dodele: **d1 ±= d2** ili **d1 = d1 ± d2**. Originalni delegat ostaje nepromenjen, a vrednost za delegat **d1** se menja tako da predstavlja referencu na novi kombinovani delegat.

Ukoliko je tip delegata deklarisan tako da vraća neku vrednost i pozove se kombinovana instanca delegata, prilikom izvršavanja će biti vraćena vrednost poslednjeg jednostavnog delegata iz liste.

Događaji

Prva stvar koju treba spomenuti u vezi sa događajima jeste da događaji nisu instance delegata. C# dopušta da se oni upotrebljavaju na isti način u određenim situacijama, ali razlika između njih postoji.

Najlakši način za razumevanje događaja je gledajući na njih kao na svojstva. Iako svojstva izgledaju kao polja, ona nisu polja, i mogu se deklarisati bez ikakve upotrebe istih. Slično tome, događaji liče na instance delegata po načinu izražavanja operatora za dodavanje i oduzimanje, ali oni to nisu.

Događaji predstavljaju parove metoda **add** i **remove**, odgovarajuće opisanih u IL-u(*eng. Intermediate Language*) kako bi bile povezane i kako bi jezici znali da te metode predstavljaju događaje. Metode odgovaraju na operacije dodavanja i uklanjanja, od kojih svaka kao parametar prihvata instancu delegata istog tipa(tipa događaja). Tipična primena ovih operacija jeste u dodavanju ili uklanjanju delegata iz liste rukovaoca(*eng. handler*) za događaj. Kada se događaj aktivira(šta god da bio aktivator – klik na dugme, timeout…) pozivaju se rukovaoci. U C#-u, pozivanje rukovaoca događajem ne predstavlja deo samog događaja.

Za pozivanje **add** i **remove** metoda koristi se sledeća sintaksa:

eventName ±= delegateInstance

Parametar **eventName** može biti kvalifikovan kao referenca(npr. **myForm.Click**) ili kao naziv tipa(npr. **myClass.SomeEvent**). Statički događaji se retko koriste.

Događaji se sami po sebi mogu deklarisati na dva načina. Prvi način je upotrebom eksplicitnih **add** i **remove** metoda, na sličan način kao kod svojstava, ali uz ključnu reč **event**. Sledi primer događaja za delegat tipa **System.EventHandler**.

```
using System;

class Test
{
    public event EventHandler MyEvent
    {
        add
        {
            Console.WriteLine ("add operation");
        }

        remove
        {
            Console.WriteLine ("remove operation");
        }
    }

    static void Main()
    {
        Test t = new Test();

        t.MyEvent += new EventHandler (t.DoNothing);
        t.MyEvent -= null;
    }

    void DoNothing (object sender, EventArgs e)
    {
    }
}
```

Ovde treba primetiti da događaj zapravo ne radi ništa sa instancama delegata koje se prosleđuju **add** i **remove** metodama. On samo ispisuje koja operacija je pozvana. Takođe treba primetiti i da se **remove** operacija poziva, iako joj je rečeno da ukloni **null** vrednost. Ovaj način ignorisanja vrednosti je redak, ali se primenjuje ukoliko postoji velik broj događaja, pri čemu je korisnik pretplaćen samo na određene događaje.

***Field-like* događaji**

C# obezbeđuje način za istovremeno deklarisanje delegatske promenljive i događaja. Ovo se naziva *field-like* događaj i deklariše se vrlo jednosatvno – slično kao i "dugačka" verzija, ali bez tela:

```
public event EventHandler MyEvent;
```

Ovo kreira delegatsku promenljivu i događaj, oba istog tipa. Pristup događaju je određen preko modifikatora pristupa događaja, ali je delegatska promenljiva uvek **private** tipa. Implicitno definisano telo događaja je dato u nastavku:

```
private EventHandler _myEvent;
```

```
public event EventHandler MyEvent
{
    add
    {
        lock (this)
        {
            _myEvent += value;
        }
    }
    remove
    {
        lock (this)
        {
            _myEvent -= value;
        }
    }
}
```

Za statičke članove, promenljiva delegata je takođe statička.

Aktiviranje događaja preko korsiničkog interfejsa

Unutar ove sekcije biće dat primer koda, koji radi nešto kada korisnik pritisne dugme. Za ovo će nam biti potrebne sledeće stvari:

1. Klasa **Button**, koja je obezbeđena preko neke tehnologije korisničkog interfejsa(*eng. UI*).
2. Event član **Button** klase, nazvan **Clicked**.
3. UI brine o tome kada će događaj **Clicked** biti aktiviran. U ovom primeru biće korišćena **SimulateClick** metoda.
4. Događaj ima tip delegata. Samo metode koje odgovaraju potpisu tog tipa delegata mogu biti dodeljene istom.
5. Kod bi trebalo da definiše metodu koja se poziva prilikom aktivacije događaja.
6. Metoda koju napišemo mora imati potpis koji odgovara tipu delegata datog događaja. Ukoliko se potpisi ne poklope, kompajler nam neće dozvoliti da dodelimo metodu tom delegatu.

Kao što se može videti, delegati imaju puno pokretnih delova. Posebno je važno zapamtiti da oni sprečavaju dodeljivanje proizvoljne metode nekom događaju. Sledi primer koji definiše delegat i klasu sa događajem tipa tog delegata.

```
using System;

public class ClickEventArgs : EventArgs
{
    public string Name { get; set; }
}

public delegate void ClickHandler(object sender, ClickEventArgs e);

public class CalculatorButton
{
    public event ClickHandler Clicked;
}
```

Događaj može biti član klase ili interfejsa. Ukoliko je član interfejsa, klasa koja implementira taj interfejs mora imati događaj u svojoj definiciji.

Ukoliko delegat služi svrsi koja nam je potrebna, možemo ga upotrebiti. FCL ima mnoge tipove, koje je moguće koristiti višekratno. **EventHandler** tip se skoro poklapa sa potpisom **ClickHandler**

tipa, gde **sender** predstavlja izvor događaja, a **EventArgs** osnovnu klasu, koja može biti nasleđena kako bi se delile informacije između metoda i poziva događaja, kada se aktivira.

Sledeći primer simulira događaj, kako bi demonstrirao kako se piše metoda koja rukuje događajima.

```
using System;

public class CalculatorButton
{
    public event ClickHandler Clicked;
    public void SimulateClick()
    {
        if (Clicked != null)
        {
            ClickEventArgs args = new ClickEventArgs();
            args.Name = "Add";

            Clicked(this, args);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Program prg = new Program();
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += new ClickHandler(prg.CalculatorBtnClicked);
        calcBtn.Clicked += prg.CalculatorBtnClicked;

        calcBtn.SimulateClick();

        Console.ReadKey();
    }

    public void CalculatorBtnClicked(object sender, ClickEventArgs e)
    {
        Console.WriteLine(
            $"Caller is a CalculatorButton: {sender is CalculatorButton} and is named
{e.Name}");
    }
}
```

CalculatorButton klasa ima novu metodu **SimulateClick**, koja demonstrira pravilan način za okidanje događaja u sopstvenom kodu. Pre aktivacije, bitno je proveriti da li je korisnik dodelio metode događaju, proveravajući da li je on **null** vrednosti. **SimulateClick** setuje **ClickEventArgs** parametre. U ovom slučaju, to je samo svojstvo **Name**. Okidanje događaja se dalje vrši pozivanjem događaja kao metode. Ovo izaziva pozivanje svih metoda dodeljenih događaju. **Main** metoda nam pokazuje kako se metode dodeljuju događaju. Prva dodela kreira novu instancu **ClickHandler** delegata. Druga dodela dodeljuje instancnu metodu **CalculatorBtnClicked** našem događaju, pri čemu **prg** instanca obezbeđuje pristup toj metodi prilikom dodele.

Za kraj, pozivom **SimulateClick** metode nad **CalculatorButton** instancom, **calcBtn**, aktivira se događaj koji je prethodno opisan. **SimulateClick** metoda je morala biti definisana unutar **CalculatorButton** klase, jer eksterni kod ne može aktivirati događaj. U ovom slučaju, **CalculatorBtnClicked** metoda će biti pozvana dva puta.

Rad sa lambdama

Lambda predstavlja bezimenu, odnosno anonimnu metodu. Ponekad imamo kod koji služi samo jednoj specifičnoj svrsi i nije ga potrebno definisati kao metodu. Lambde predstavljaju brz i jednosatvan način za dodelu i izvršavanje bloka koda.

Kao i metode, lambde mogu imati ulazne parametre, telo, i povratnu vrednost. Sledeći kod predstavlja primer lambde.

```
using System;

public class Program
{
    public static void Main()
    {
        Action hello = () => Console.WriteLine("Hello!");
        hello();

        Console.ReadKey();
    }
}
```

Action je delegat **.NET Framework-a** za višekratnu upotrebu(*eng. reusable*), a **hello** predstavlja promenljivu **Action** tipa delegata. Lambda počinje sa praznom listom parametara, što znači da nema ulaznih parametara. Operator => odvaja listu parametara i telo lambde. Dalje, vidimo telo, koje je dato u vidu jednog iskaza. Kako je **hello** delegat, može biti pozvan kao metoda i izvršiće lambda funkciju, koja ispisuje **"Hello!"** u konzoli.

Ukoliko postoji samo jedna naredba, nema potrebe za vitičastim zagradama, ali u slučaju više naredbi one se moraju uključiti, kao u narednom primeru:

```
using System;

public class Program
{
    public static void Main()
    {
        Predicate<string> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }



    public static void ValidateInput(Predicate<string> validator)
    {
        string input = "Hello";
```

```
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Prethodni primer dodeljuje lambdu **Predicate** delegatu, koji se osmišljen tako da vraća bool vrednost. Lambda ima jedan parametar, **word**, tipa **string**. Kako lambda ima više od jedne naredbe, moramo definisati telo unutar vitičastih zagrada. Ovaj primer pokazuje kako je lambdu moguće proslediti i kao parametar i kao čitavu lambdu.

Predicate predstavlja generički delegat. Tip parametra se postavlja na **<string>**, što znači da je ulazni parametar lambde **string** tipa.

ValidateInput metoda prosleđuje **string** delegatu **validator** i dodeljuje vrednost **isValid** promenljivoj. Ovo je isto kao poziv metode, samo što metoda ne postoji, odnosno imamo samo kod.

Još jedan način za upotrebu lambdi jeste u kombinaciji sa događajima. Naredni primer pokazuje kako se uz pomoć lambde može napisati metoda za rukovanje događajem **Clicked**, koji predstavlja **event** polje klase **CalculatorButton**.

```
using System;

public class Program
{
    public static void Main()
    {
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += (object sender, ClickEventArgs e) =>
        {
            Console.WriteLine(
                $"Caller is a CalculatorButton: {sender is CalculatorButton} and is
named {e.Name}");

            Console.WriteLine(message);
        };

        calcBtn.SimulateClick();

        Console.ReadKey();
    }
}
```

Prva stvar koju treba primetiti jeste da je delegat dodeljen **Clicked** događaju zapravo lambda. Takođe, treba primetiti i da ukoliko lambda ima dva ili više parametara, njih je potrebno ubaciti u obične zagrade, kao listu razdvojenu zarezima.

Još neki FCL tipovi delegata

Pored **Action** i **Predicate<T>** koje smo spomenuli do sada, **FCL**(*Framework Class Library*) ima i set delegata **Func<T>.** U nastavku je ponovljen primer sa **Predicate<string>** delegatom, ali je ovaj put umesto umesto njega korišćen **Func<T, TResult>** delegat**.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class Program
{
    public static void Main()
    {
        Func<string, bool> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }

    public static void ValidateInput(Func<string, bool> validator)
    {
        string input = "Hello";
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Kao što je već spomenuto, i **Func<T, TResult>** i **Predicate<T>** su generički delegati. Tipovi specificirani unutar zagrada < > se odnose na parametre i povratne tipove. Sledeći kod prikazuje definiciju delegata **Predicate<T>** unutar FCL-a.

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

On se odnosi na metodu koja vraća **bool** vrednost, ali prihvata parametar tipa **T**. **Predicate<string>** znači da je parametar referentne metode tipa **string.** Slično ovome, data je i definicija delegata **Func<T, TResult>.**

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

Ovo nam govori da delegat **Func<T, TResult>** prihvata **T** tip kao parametar i ima povratnu vrednost tipa **TResult**. **Func<string, bool>** se odnosi na metodu sa parametrom tipa **string** i povratnom vrednošću tipa **bool**. Zbog jednostavnosti, FCL nudi 18 preklapanja **Func** delegata, dozvoljavajući između 0 i 16 ulaznih parametara i 1 povratni parametar. Ovo pokriva mnogo scenarija i zbog toga možemo kreirati sopstvene delegate samo kada je to neophodno.