.NET Okruženje

Vježba 3: C# - LINQ, async

Sadržaj

C# - napredni koncepti

U narednom poglavlju bit će obrađeni napredni koncepti u C# jeziku – konkretnije lambda izrazi i LINQ mehanizam. Stabla izraza (expression trees) su posebne strukture u C# jeziku koje omogućavaju definiranje koda koji će tek kasnije biti izvršen. LINQ koristi stabla izraza za manipulaciju kolekcijama, gdje se pomoću stabla izraza definira upit koji će tek naknadno biti izveden (ako se radi o upitu na bazu podataka kroz Entity Framework ili sličan ORM alat). U ovom poglavlju bit će obrađene osnove LINQ izraza za manipulacije kolekcijama te Func<>, Predicate<> i Action<> objekti kojima se definira funkcija u obliku varijable.

Lambda izrazi i LINQ

Lambda izrazi se u osnovi mogu poistovjetiti s anonimnim ili inline funkcijama. U ovom dijelu bit će navedeni primjeri korištenja lambda izraza u svrhu enumeracije kolekcija. Ideja manipulacije kolekcijama kroz LINQ je da svaka kolekcija, pozivom neke LINQ naredbe se transformira u novu kolekciju[1].

Lambda izrazi

Pomoću lambda izraza se definira anonimna funkcija ili stablo izraza koje se koristi pri izvršavanju primjerice LINQ upita. Sintaksa lambda izraza je sljedeća:

```
p => p.Id < 3
```

Primjerice, u gornjem izrazu, prvi parametar p (s lijeve strane operatora =>) označava jedan ulazni objekt kolekcije. S desne strane operatora => se nalazi izraz koji koristi objekt p. U ovom slučaju, povratna vrijednost izraza s desne strane je „bool“. Štoviše, gornji izraz je ekvivalentan ovom:

```
p =>
{
    if (p.Id < 3)
        return true;
    return false;
}
```

Naravno, ovisno o kontekstu, povratna vrijednost se mijenja.

LINQ – naredba where

Vjerojatno najčešće korištena metoda prilikom manipulacija kolekcijama – služi za filtriranje i obradu samo određenih podataka.

```
var quizesWithLowId = listaKvizova
    .Where(p => p.Id < 3);
```

Lambda izraz – kao rezultat se u konačnici vraćaju samo oni elementi iz kolekcije za koje je zadovoljen ovaj izraz; tj., za koje ova funkcija vraća true.

Kolekcija objekata tipa Quiz – primjerice, **List<Quiz>**; nad njom se izvršava **where** naredba.

[1] Puno primjera korištenja LINQ naredbi može se naći ovdje: http://www.dotnetperls.com/linq

Zadatak 3.1

Napisati funkciju u klasi Fakultet koja pomoću where naredbe dohvaća samo studente rođene nakon 1991. godine. Funkcija se treba zvati **DohvatiStudente91**, a povratna vrijednost treba biti IEnumerable<Student>.

1. Izdvojiti iste objekte, ali bez korištenja LINQ naredbi, u funkciji **DohvatiStudente91NoLinq**.

Zadatak 3.2

Napisati funkciju **StudentiNeTvzD** u klasi Fakultet koja će dohvatiti samo studente kojima JMBAG NE počinje s „0246“ i kojima prezime počinje s „D“.

LINQ – naredba ToList

Svaku kolekciju možemo eksplicitno pretvoriti u listu pozivom ToList metode koja će stvoriti novi List<> objekt, i sve elemente kolekcije prebaciti u listu.

Zadatak 3.3

Kreirati funkciju **DohvatiStudente91List** koja, kao u zadatku 3.1 dohvaća studente rođene nakon '91, ali ih vraća u obliku „List“ objekta.

LINQ – naredbe First, Single, FirstOrDefault i SingleOrDefault

Gore navedene naredbe služe za izdvajanje samo jednog elementa iz kolekcije. Metode imaju sljedeće razlike:

- **First** – ukoliko nema niti jednog elementa u kolekciji (primjerice, tražimo sve studente kojima ime počinje s „Đ“), tada će poziv te naredbe baciti iznimku
- **FirstOrDefault** – ukoliko nema niti jednog elementa u kolekciji, tada poziv naredbe vraća default vrijednost, ovisno koji tipovi su u kolekciji. Primjerice, za klasu Student, default vrijednost je null, dok za cjelobrojne vrijednosti default vrijednost je 0.
- **Single** – ukoliko ima više od jednog elementa u kolekciji, ili je kolekcija prazna, bacit će se iznimka
- **SingleOrDefault** – slično kao Single, no neće se baciti iznimka već se vraća default vrijednost

LINQ – naredba OrderBy i OrderByDescending

Koristi se ukoliko je potrebno poredati kolekciju. OrderByDescending primjenjuje poredak u obrnutom redoslijedu od OrderBy poziva. Sve LINQ naredbe se mogu zajedno kombinirati.

Zadatak 3.4

Kreirati funkciju **NajboljiProsjek(int god)** u klasi Fakultet koja dohvaća studenta s najboljim prosjekom rođenog godine koja je proslijeđena kao parametar „god“. Ukoliko student ne postoji, funkcija treba vratiti null.

Zadatak 3.5

Kreirati funkciju **StudentiGodinaOrdered(int god)** u klasi Fakultet koja dohvaća studente rođene one godine koja je proslijeđena kao parametar „god“, poredano po prosjeku silazno.

Zadatak 3.6

Kreirati funkciju **SviProfesori(bool asc)** u klasi Fakultet koja dohvaća sve profesore poredano abecedno po prezimenu zatim po imenu ako je parametar **asc = true**, inače poredane abecedno ali unatrag.

LINQ – naredba Count

Kao rezultat vraća broj elemenata u kolekciji. Može se kombinirati s ostalim LINQ upitima i naredbama.

Zadatak 3.7

Kreirati funkciju **KolikoProfesoraUZvanju(Zvanje zvanje)** u klasi Fakultet koja vraća ukupan broj profesora u zadanom zvanju. Funkcija prima kao parametar traženo zvanje.

LINQ – podupiti

Čest je slučaj da je potrebno izdvojiti elemente kolekcije na temelju kolekcija koje sadrže. Unutar bilo kojeg LINQ upita, može se postaviti dodatni upit nad bilo kojom kolekcijom definiranom nekim svojstvom.

Zadatak 3.8

Kako bi mogli testirati podupite, potrebno je uz već postojeće klase, dodati i klasu **Predmet**, te u klasi Profesor omogućiti spremanje popisa svih predmeta koje profesor predaje (svojstvo nazvati **Predmeti**). Klasa Predmet treba imati šifru predmeta (int **Sifra**), broj ECTS bodova (int **ECTS**), naziv predmeta (**Naziv**).

Zadatak 3.9

Kreirati funkciju **NeaktivniProfesori(int x)** u klasi Fakultet koja vraća profesore koji su u zvanju Predavač ili Viši Predavač a koji imaju manje od **x** predmeta na kojima drže nastavu (x se proslijeđuje kao parametar).

Zadatak 3.10

Kreirati funkciju **AktivniAsistenti(int x, int minEcts)** u klasi Fakultet koja vraća profesore koji su u zvanju Asistent a koji imaju više od **x** predmeta s barem **minEcts** ECTS bodova na kojima drže nastavu.

Func, Predicate i Action objekti

Nešto slično kao što su pokazivači na funkcije, u C# je uveden novi set objekata koji omogućavaju spremanje funkcije u varijablu, te kasnije pozivanje te funkcije s željenim parametrima i dohvaćanje rezultata. Primjerice, u sljedećem programskom odsječu pozivamo funkciju GetRandomOperation() koja vraća objekt **Func<int,int,int>**. Taj objekt zapravo predstavlja funkcijju koja ima dva parametra tipa **int** i povratnu vrijednost tipa **int**. Kada bi ista funkcija vraćala double vrijednost, Func objekt bi bio Func<int,int,double>. Zadnji parametar predloška je uvijek tip povratne vrijednosti funkcije. Pogledajmo sljedeći primjer:

Program.cs

```
namespace LINQHelperApp.Console
{
    class Program
    {
        static void Main(string[] args)
        {
            var calc = new Calculator();

            Func<int,int,int> fOp = calc.GetRandomOperation();

            int x = 10, y = 5;
            int rez = fOp(x, y);

            System.Console.WriteLine(rez);
            System.Console.ReadKey();
        }
    }
}
```

Varijabla fOp je upravo objekt koji tretiramo kao funkciju, vidimo u nastavku poziv fOp(x,y) gdje se „pravimo“ da je fOp zapravo neka funkcija. Pogledajmo kako izgleda kod Calculator klase (u nastavku):

Calculator.cs

```
namespace LINQHelperApp.Console
{
    public class Calculator
    {
        public Func<int, int, int> GetRandomOperation()
        {
            var rand = new Random();
            switch (rand.Next() % 3)
            {
                case 0:
                    return Summation;
                case 1:
                    return Substraction;
                case 2:
                    return Multiplication;
            }
            return null;
        }

        public int Summation(int a, int b)
        {
            return a + b;
        }

        public int Substraction(int a, int b)
        {
            return a - b;
        }

        public int Multiplication(int a, int b)
        {
            return a * b;
        }
    }
}
```

Također, moguće je iste funkcije definirati *inline*:

Calculator.cs

```
namespace LINQHelperApp.Console
{
    public class Calculator
    {
        public Func<int, int, int> GetRandomOperation()
        {
            var rand = new Random();
            switch (rand.Next() % 3)
            {
                case 0:
                    return (a, b) => {
                        return a + b;
                    };
                case 1:
                    return ((a,b) => a - b);
                case 2:
                    return (a,b) => a * b;
            }
            return null;
        }
    }
}
```

Iz gornjeg primjera se može uočiti sličnost između korištenja lambda izraza u zadacima sa LINQ operacijama, te korištenja na ovaj način, preko Func, Action i Predicate objekata. Zapravo, LINQ funkcije koriste upravo te objekte za obradu kolekcija.

Kao i svaki drugi objekt, moguće je i ovakav objekt proslijediti kao parametar nekoj drugoj metodi ili funkciji:

Program.cs

```
namespace LINQHelperApp.Console
{
    class Program
    {
        static void PrintOperation(int x, int y, Func<int, int, int> op)
        {
            int rez = op(x, y);
            System.Console.WriteLine(rez);
        }

        static void Main(string[] args)
        {
            var calc = new Calculator();

            Func<int,int,int> fOp = calc.GetRandomOperation();

            PrintOperation(10, 5, fOp);

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

Gore navedena svojstva se koriste kod korištenja LINQ proširenja za manipulaciju kolekcijama. Ideja je da se operacija koju je potrebno izvesti nad svakim članom kolekcija proslijedi u obliku Func<>, Predicate<> ili Action<> objekta. Predicate<> je isto što i Func<> objekt, ali s povratnom vrijednošću bool, dok je Action<> isto što i Func<> objekt, samo bez povratne vrijednosti.

Zadatak 3.11

U klasi Fakultet potrebno je kreirati funkciju **IzmjeniProfesore(Action<Profesor> action)** koja kao parametar prima Action objekt kojem je jedini parametar tipa Profesor. Funkcija mora proći po svim profesorima u fakultetu, i izvršiti akciju nad svima njima. Konkretna funkcionalnost akcije se ubacuje u automatskom testu.

Async arhitektura

Od verzije .NET radnog okvira 4.5 uvedene su velike optimizacije za rad s višedretvenošću. Konkretno, uvedene su ključne naredbe async i await koje omogućavaju izuzetno lagano korištenje pozadinskih dretvi za izvršavanje procesa koji bi inače potrošili prijeko potrebne resurse. Također, postoje performansni benefiti od korištenja *async* arhitekture u odnosu na sinkronu – najčešće u desktop/mobile okruženju gdje je korištenje UI dretve za dulje operacije nepovoljno.

Osim u desktop/klijentskim aplikacijama, async postaje bitan u kontekstu velikog broja paralelnih zahtjeva na server, jer ako pozive na bazu podataka odrađuje pozadinska dretva, oslobađaju se resursi za odrađivanje „običnih" zahtjeva za to vrijeme.

Klasa Task

U C# async arhitekturi, ključnu ulogu ima klasa Task, kao i ključna riječ „async". Pogledajmo sljedeći primjer i proanalizirajmo ga:

Vjezba.AsyncConsole/Program.cs

```
Task t1 = Task.Run(() =>
{
    Console.WriteLine($"Sleeping started");
    Thread.Sleep(1000);
    Console.WriteLine($"Sleeping completed");
});
Console.WriteLine($"Waiting on task..");
t1.Wait();
```

Kad bi riječima opisali ovaj kod, rekli bi ovako:

- Kreiramo Task na način da mu definiramo što radi (zeleno uokvireno) – to je primjer Action objekta iz prošlog poglavlja
  - Pozivom Task.Run() se taj task automatski pokreće
  - Spremamo ga u varijablu t1. U ovom trenutku task se već počeo izvršavati (vjerojatno)
- Main program nastavlja daljnje izvođenje, iduća linija je označena plavo.
- Nakon te linije, zaustavljamo daljnje izvođenje dok god Task t1 ne završi: pozivom t1.Wait().
  - Alternativa t1.Wait bi bila statička metoda klase Task koja omogućava čekanje više taskova odjednom: Task.WaitAll(t1)

Zadatak 3.12

Na temelju gornjeg primjera, kreirati 2 taska, jedan koji čeka 1s, drugi koji čeka 1.5s. Koristiti Task.WaitAll(). U svim potrebnim trenucima ispisati poruku iz koje će biti jasno koji task se izvršava, a koji je završio (slično kao gore). Kreirati posebnu konzolnu aplikaciju za ovaj i idući zadatak.

- Ako umjesto Task.WaitAll() koristimo Task.WaitAny(), što se mijenja?

Async metode

Kada bi htjeli postići istu funkcionalnost kao u gornjem primjeru pomoću async metode, to bi izgledalo ovako:

Vjezba.AsyncConsole/Program.cs

```
private static async Task DoSomeSleepingAsync()
{
    Console.WriteLine($"Sleeping started");
    await Task.Delay(1000);
    Console.WriteLine($"Sleeping completed");
}

static void Main(string[] args)
{
    Task t1 = DoSomeSleepingAsync();
    Console.WriteLine($"Waiting on task..");
    t1.Wait();

    ...
```

Razlika koju možemo uočiti je da smo umjesto Thread.Sleep() koristili await Task.Delay(). Pronalizirajmo malo izvođenje gornjeg koda:

1. U Main Metodi se kreira Task t1 pozivom async metode
    a. Primjetiti da metoda DoSomeSleeping ima definiranu povratnu vrijednost tipa Task, ali ne vraća nikakav rezultat (nema „return“ naredbe). To određuje ključna riječ „async“.
2. Izvršava se prva linija (označena zelenom točkicom)
3. Nakon toga se dolazi do await naredbe. U tom trenutku (tek tada) se interno kreira nova dretva (thread) i izvršava spavanje od 1000ms. Za to vrijeme, u Main funkciji se nastavlja izvođenje paralelno (linija označena plavom točkom)
4. Kod poziva t1.Wait() se čeka da gornji Task završi, slično kao u prošlom primjeru – iduća linija koja se izvršava je označena crvenom točkom.

Generalno, često će async metode imati sufix „async“ tako da ih je vrlo lako prepoznati. Također, razne popularne biblioteke koje se koriste će često imati istu metodu napisanu u sinkronom kontekstu i async kontektstu.

Zadatak 3.13

Kreirati async funkciju **SleepF1** koja unutar sebe poziva drugu async funkciju **SleepF2** koristeći await operator. U svakoj od tih funkcija napraviti **await Task.Delay()** željeni broj milisekundi i ispisati odgovarajuće poruke koje pokazuju koji je slijed izvršavanja.