1. Uvod

Na prvim vježbama se upoznajemo s funkcionalnim i višenitnim mogućnostima C# i .NET tehnologije. Ove mogučnosti ćemo koristiti kroz nekoliko sljedećih vježbi u izradi jednostavnih raspodijeljenih aplikacija.

Teme kojima posvećujemo ove vježbe su:

- Task;
- Delegat;
- Lambda izraz.

Naravno, sve navedeno se može koristiti u bilo kojoj aplikaciji izrađenoj uz pomoć C# i .NET tehnologije. Task-ove koristimo kada nam samo glavna nit u aplikaciji nije dovoljna. Delegati nam služe za referenciranje metoda i funkcija, a lambda izrazi za skraćeno stvaranje anonimnih funkcija.

1.1 Task

Task-ovi predstavljaju asinkronu operaciju. Predstavljaju veću razine abstrakcije od niti (engl. *Thread*) i *ThreadPool*-a. ThreadPool predstavlja skupinu niti koje sustav instancira za obavljanje manjih poslova. Broj niti i poslova ne mora biti jednak, već se broj niti određuje prema računskim mogućnostima sustava. Prednosti Task-ova u odnosu na Thread i ThreadPool:

- učinkovitija i skalabilnija upotreba sistemskih resursa. ThreadPool koristi napredne algoritme uz pomoć kojih se na temelju dostupnih resursa za aplikaciju instancira skupina niti. Task-ovi koji se stvaraju se stavljaju u red za ThreadPool kako bi im se dodijelila nit na kojoj će se izvršavati. Zato kažemo da u pozadini Task Parallel Library koji je uveo Task kao koncept i klasu u .NET koristi ThreadPool. Zbog ovoga su task-ovi jeftini kada su u pitanju resursi te ih se može kreirati velik broj;
- Programer ima više kontrole nad Task-om nego nad običnom niti. Task-ovi nude programeru širok spektar API-ova (*Application Programming Interface*) koji podržavaju čekanje na task, prekid, nadovezivanje, upravljanje pogreškama, detaljan uvid u stanje, itd.

Zbog oba ova razloga, TPL (kratica za Task Parallel Library) je preporučeni način za izradu asinkronih, paralelnih i višenitnih aplikacija. Već je rečeno da se niti u threadpool-u koriste za obavljanje manjih poslova. Što onda sa zadatkom koji će se obavljati duže vremena? Npr. što ako aplikacija mora uploadati podatak koji zauzima 100 ili više MB memorije? U tom slučaju TPL-u se može narediti da stvori nit izvan thread poola koju će koristiti za takav zadatak.

Prvi primjer demonstrira najjednostavniji način izrade Task-a.

Primjer 1.1: Stvaranje Task-ova

```
1
2  using System;
3  using System.Threading;
4
5  // Task-ovi se nalaze u ovome namespaceu:
6  using System.Threading.Tasks;
7
8  namespace StvaranjeTaskOva
9  {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14             for(int i = 0; i < 10; i++)
15             {
16                 // Instanciramo task pri cemu mu saljemo kao
                       argument metodu koju mora izvrsiti
17                 Task t = new Task(WriteInfo);
18
19                 // Pokrecemo stvoreni task
20                 t.Start();
21             }
22
23             Console.WriteLine("[Main] ID: " +
                   Thread.CurrentThread.ManagedThreadId + "
                   Milliseconds: " + DateTime.Now.Millisecond);
24
25             Console.ReadLine();
26         }
```

```
27
28          private static void WriteInfo()
29          {
30              Console.WriteLine("ID: " +
                    Thread.CurrentThread.ManagedThreadId + "
                    Milliseconds " + DateTime.Now.Millisecond);
31          }
32      }
33  }
```

U primjeru 1.1 stvaramo 10 task-ova koji ispisuju ID niti na kojoj se izvršavaju te trenutan datum i vrijeme. Ujedno, unutar main metode ispisujemo ID niti na kojoj je main pokrenut. Ovim primjerom se želi pokazati sljedeće:

- task-ovi se pokreću na različitim nitima;
- jednom kada je novi task pokrenut, glavna nit ne čeka da se pokrenuti task završi već nastavlja sa svojim radom;
- osim što ih je potrebno kreirati task-ove je potrebno i pokrenuti;
- različiti task-ovi mogu biti pokrenuti na istoj niti.

Pokretanjem ovog programa dobije se ispis sličan sljedećem:

```
1
2           ID: 6 Milliseconds: 808
3           [Main] ID: 9 Milliseconds: 809
4           ID: 14 Milliseconds: 837
5           ID: 10 Milliseconds: 812
6           ID: 13 Milliseconds: 837
7           ID: 15 Milliseconds: 837
8           ID: 11 Milliseconds: 836
9           ID: 6 Milliseconds: 839
10          ID: 12 Milliseconds: 836
11          ID: 16 Milliseconds: 837
12          ID: 15 Milliseconds: 839
```

Obratite pažnju da u kodu konstruktoru Task-a šaljemo metodu. Ovome ćemo se detaljnije posvetiti u nastavku vježbe. Poslana metoda je zadatak koji Task mora obaviti. Task neće biti izvršen do kraja dok god ne obavi zadatak koji mu je poslan. Ovdje smo oprezni u izražavanju jer je moguće da operacijski sustav suspendira nit na kojoj se task izvršava te procesorsko vrijeme dodijeli drugoj niti. Prisjetite se gradiva s kolegija Operacijski sustavi. Različita stanja u kojima se task može nalaziti demonstriramo primjerom 1.2.

Primjer 1.2: Stvaranje Task-ova

```
1
2   using System;
3   using System.Collections.Generic;
4   using System.Threading;
5   using System.Threading.Tasks;
6
7   namespace StanjeTaskova
8   {
9       class Program
10      {
11          static void Main(string[] args)
12          {
```

```
13              Dictionary<int, Task> taskDict = new Dictionary<int,
                    Task>();
14
15              for (int i = 0; i < 10; i++)
16              {
17                  Task t = new Task(WriteInfo);
18
19                  t.Start();
20
21                  taskDict.Add(i, t);
22              }
23
24              foreach(var el in taskDict)
25              {
26                  Console.WriteLine("Key " + el.Key + " State " +
                        el.Value.Status);
27              }
28
29              // zaustavljamo trenutnu nit za 200 milisekundi
30              Thread.Sleep(200);
31
32              foreach (var el in taskDict)
33              {
34                  Console.WriteLine("Key " + el.Key + " State " +
                        el.Value.Status);
35              }
36
37              Console.ReadLine();
38          }
39
40          private static void WriteInfo()
41          {
42              Console.WriteLine("ID: " +
                    Thread.CurrentThread.ManagedThreadId + "
                    Milliseconds: " + DateTime.Now.Millisecond);
43
44              // Zaustavljamo trenutnu nit za 100 milisekundi
45              Thread.Sleep(100);
46          }
47      }
48  }
```

U linijama 30 i 45 primjera 1.2 koristimo metodu kojom zaustavimo trenutnu nit za određeni broj milisekundi. Ovu metodu ćemo kroz vježbe koristiti kada želimo demonstrirati nešto ili simulirati posao koji dugo traje. Ispis ovoga primjera je sličan ovome:

```
1  Key 0 State WaitingToRun
2  ID: 9 Milliseconds: 601
3  ID: 11 Milliseconds: 603
4  ID: 12 Milliseconds: 601
5  ID: 16 Milliseconds: 613
6  ID: 14 Milliseconds: 613
7  ID: 10 Milliseconds: 610
8  ID: 15 Milliseconds: 617
9  ID: 13 Milliseconds: 616
10 Key 1 State Running
```

```
Key 2 State Running
Key 3 State Running
Key 4 State Running
Key 5 State Running
Key 6 State Running
Key 7 State Running
Key 8 State WaitingToRun
Key 9 State WaitingToRun
ID: 9 Milliseconds: 720
ID: 11 Milliseconds: 720
Key 0 State RanToCompletion
Key 1 State RanToCompletion
Key 2 State RanToCompletion
Key 3 State RanToCompletion
Key 4 State RanToCompletion
Key 5 State RanToCompletion
Key 6 State RanToCompletion
Key 7 State RanToCompletion
Key 8 State RanToCompletion
Key 9 State RanToCompletion
```

Uočimo da se taskovi nalaze u različitim momentima u različitm stanjima.

Ranije smo naveli da različiti taskovi mogu biti pokrenuti na istim nitima. Naime, ovo je svojstvo TPL-a kojim se želi optimizirati iskoristivost niti. TPL može koristi istu nit za izvršenje više taskova prije nego tu nit vrati u threadpool. Uočite da ne znamo nad kojim nitima će TPL izvršiti koji task.

Ovo nije jedini način za pokretanje taskova. Preporučeni način za ove vježbe je dan u primjeru 1.3.

Primjer 1.3: Preporučeni način pokretanja taskova

```
1
2  using System;
3  using System.Threading.Tasks;
4
5  namespace PripmjeriTaskova
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             // task koji nema rezultat
12             Task t1 = Task.Factory.StartNew(WriteMsg);
13             // task kojemu ce rezultati biti tipa int
14             Task<int> t2 = Task.Factory.StartNew<int>(Get42);
15             // task kojemu ce rezultat biti tipa string
16             Task<string> t3 =
                   Task.Factory.StartNew<string>(GetMsg);
17
18             Console.WriteLine("t1 is completed: " + t1.Status);
19
20             Console.WriteLine("t2 value: " + t2.Result);
21             Console.WriteLine("t3 value: " + t3.Result);
22
23             Console.ReadLine();
```

```
24          }
25
26          private static void WriteMsg()
27          {
28              Console.WriteLine("Hello world");
29          }
30
31          private static int Get42()
32          {
33              return 42;
34          }
35
36          private static string GetMsg()
37          {
38              return "Hello world";
39          }
40      }
41  }
```

Uz pomoć `Task.Factory.StartNew` metode se istovremeno instancira i pokreće task te se instancirani task dobije kao povratna vrijednost koja se može spremiti u varijablu. Od taska se može očekivati rezultat nekog tipa. Od taska t1 ne očekujemo rezultat. Od taskova t2 i t3 očekujemo rezultate tipa int i string, redom. Rezultatu taska pristupamo preko njegovog svojstva `Result`. **Ukoliko pristupamo svojstvu Result nekoga taska, nit koja pokušava pristupiti rezultatu će biti blokirana dok god rezultat ne bude dostupan, odnosno, dok se task ne izvrši do kraja**. Blokiranje niti nije preporučljivo u komercijalnim aplikacijama.

1.1.1 Prekid taska

Za prekid taskova se preporuča korištenje `CancellationToken`-a. Ovaj token je moguće dobiti instanciranjem `CancellationTokenSource` objekta te pristupom njegovom svojstvu `Token`. Dakako, moguće je da sami napravimo task u kojemu nam `CancellationToken` neće biti potreban, ali postoji mogućnost da nam takav pristup neće odgovarati pri izradi složenijih aplikacija. Token posjeduje svojstvo `IsCancellationRequested` koje će, kao što samo ime kaže, vratiti vrijednost istine ako je zatražen prekid taska. Za zatražiti prekid taska, token ima metodu `Cancel`.

1.1.2 Metode s argumentima

Taskovi ne bi bili previše korisni kada ne bismo u njih mogli poslati metodu koja prima parametre. U primjeru 1.4 se instancira i pokreće nekoliko takvih taskova. U svim primjerima se kao prvi argument u metodu `StartNew` šalje akcija koja očekuje jedan parametar tipa `object`, a kao drugi argument se šalje parametar koji poslana metoda mora primiti. S obzirom na to da akcija koja se šalje mora primiti object, u svakoj od poslanih metoda se primljeni parametar mora castat (pretvoriti) u tip podatka koji je metodi potreban da obavi svoj zadatak.

Primjer 1.4: Primjer slanja argumenata u metodu taska

```

2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
```

```
8  namespace PrimjerTaskMetodaSParametrima
9  {
10     class Info
11     {
12         private int _broj;
13
14         public int Broj
15         {
16             get { return _broj; }
17             set { _broj = value; }
18         }
19
20         private int _potencija;
21
22         public int Potencija
23         {
24             get { return _potencija; }
25             set { _potencija = value; }
26         }
27
28         public Info(int broj, int potencija)
29         {
30             _broj = broj;
31             _potencija = potencija;
32         }
33     }
34
35     class Program
36     {
37         static void Main(string[] args)
38         {
39             string msg = "Hello world";
40
41             Task t1 = Task.Factory.StartNew(Print, msg);
42
43             int broj = 4;
44             Task<double> t2 = Task.Factory.StartNew(Square,
                   broj);
45
46             int potencija = 3;
47
48             Info info = new Info(broj, potencija);
49             Task<double> t3 = Task.Factory.StartNew(Pow, info);
50
51             Console.WriteLine(t2.Result);
52             Console.WriteLine(t3.Result);
53
54             Console.ReadLine();
55         }
56
57         static double Square(object n)
58         {
59             return Math.Pow((int)n, 2);
60         }
61
62         static double Pow(object info)
```

```
63          {
64              Info i = (Info)info;
65
66              int m = i.Broj;
67              int n = i.Potencija;
68
69              return Math.Pow(m, n);
70          }
71
72          static void Print(object msg)
73          {
74              Console.WriteLine(msg);
75          }
76      }
77  }
```

Nadalje, metode Pow prima objekt `Info` koji u sebi sadrži dva svojstva, a ta svojstva su informacije koje metoda koristi da nam da nekakav rezultat. Zašto ta dva broja nismo direktno poslali kao drugi argument taska? Zato što `StartNew` očekuje metodu koja prima **jedan** argument tipa objekt. Kao posljedica ovoga, sama metoda `StartNew` ima predviđen samo jedan parametar za argument poslane metode. Kako bismo zaobišli ovaj problem, vrijednosti koje metoda koristi smo "zamotali" u novi objekt tipa `Info` te njega poslali, a zatim pustili samu metodu da primljeni objekt pretvori u odgovarajući tip te izvuče iz njega potrebne podatke.
Ovakav način prenošenja argumenata do poslanih metoda je dosta ograničavajuć te podrazumjeva dosta koda za pretvaranje iz jednog tipa podatka u drugi. Rješenje: **Zašto ne bismo metodu koja prima parametar "zamotali" u funkciju koja ne prima nijedan, ali će sa sobom "ponjeti" sve vrijednosti koje su "zamotanoj" metodi potrebne.** O ovome ćemo pričati u kasnijem poglavlju kada se upoznamo s lambda izrazima.

1.2 Delegati

Delegat je tip podatka koji nam služi za referenciranje metoda odgovarajućeg potpisa. Prisjetite se što je potpis metode. Omogućuje nam aspekte funkcionalnog programiranja u C#-u, npr. spremanje metode u varijablu, slanje jedne metode u drugu.

Slanje jedne metode u drugu smo već vidjeli u prethodnome poglavlju. Studenti koji su položili Objektno-orijentirano programiranje bi trebali biti svjesni da je i konstruktor nekakva metoda kojoj vraća novo stvoreni objekt. Nazvati konstruktor metodom je pomalo subjektivno i treba uzeti s dozom pažnje. Unatoč tome, vidjeli smo da smo uspješno odmah u prvome primjeru u konstruktor taska poslali jednu metodu. U kasnijim primjerima smo u metodu `StartNew` slali metodu koju smo željeli da task izvršava. Ovo ne bi bilo moguće da nema delegata.

U posljednjem potpoglavlju smo spomenuli da metoda `StartNew` kao prvi parametar očekuje metodu koja prima parametar tipa `object`. Ubuduće ćemo govoriti da metoda očekuje delegat nekog tipa. Budući da metoda `StartNew` očekuje delegat koji prima object ili delegat koji ne prima parametar, nikakav treći delegat ne možemo poslati u metodu.

Delegati i aspekti funkcionalnog programiranja koji su ugrađeni u C# omogućuju veću razinu abstrakcije i iskoristivost koda.

Delegati imaju sljedeća svojstvaČ

- Delegati su slični pokazivačima na funkciju u C++-u, ali kompajler može validirati tip delegata koji se koristi;
- Delegati omogućuje slanje metoda kao parametara
- Delegati se mogu koristiti za definiranje takozvanih *callback* metoda. To su metoda kojima je jedan od parametara delegat, a sama metoda unutar svoga koda odlučuje kad će i hoće li

pozvati poslani delegat;

Svoj delegat možemo definirati koristeći ključnu riječ `delegate`, kao npr:

```
1         delegate int MyDelegate(int x, int y);
```

Jednom kada smo definirali delegat, možemo ga koristiti u kodu kao i druge tipove podataka. Delegat definiran poviše može referencirati metode koje primaju dva parametra tipa int i vraćaju rezultat tipa int.

1.2.1 Action i Func

Na vježbama ćemo koristiti delegate koji već postoje u C# programskom jeziku, a to su `Action` i `Func`. Prvoga koristite kada želite referencirati metodu koja nema povratnu vrijednost, odnosno povratna vrijednost je void, a drugoga kada želite referencirati metodu koja ima povratnu vrijednost.

Generički oblik ovih delegata možemo zapisati na sljedeći način:

- `Action<T1, T2, ..., Tn>` - gdje T1 do Tn označavaju tipove podataka za svaki od parametara koje referencirana metoda prima. U najjednostavnijem obliku ovaj delegat izgleda ovako: Action, a to je delegat za metodu koja ne prima parametre i nema povratnu vrijednost;
- `Func<T1, T2, ..., Tn, Tout>` - gdje T1 do Tn označavaju tipove podataka za svaki od parametara koje referencirana metoda prima, Tout označava povratnu vrijednost metode. U najjednostavnijem obliku ovaj delegat izgleda: Func<T>, a to je delegat kojim se mogu referencirati metode koje ne primaju parametre, a povratna vrijednost im je tipa T.

Primjer 1.5: Delegati

```
1
2  using System;
3
4  namespace Delegati
5  {
6      class Program
7      {
8          delegate double MyDelegate(bool b, double d);
9          delegate void NestoSaStrane(string msg);
10
11         static private double SquareOrExp(bool b, double d)
12         {
13             return b ? Math.Pow(d, 2) : Math.Exp(d);
14         }
15
16         static private void Ispis(string message)
17         {
18             Console.WriteLine(message);
19         }
20
21         static void Main(string[] args)
22         {
23             NestoSaStrane ispis = Ispis;
24             MyDelegate mojaMetoda = SquareOrExp;
25
26             ispis("Hello world");
27
28             // Pozivamo i ispisujemo rezultat metode spremljene
                   u varijablu mojaMetoda
29             Console.WriteLine(mojaMetoda(true, 2));
```

```
30              // Pozivamo i ispisujemo rezultat metode spremljene
                   u varijablu mojaMetoda
31              Console.WriteLine(mojaMetoda(false, 5));
32
33              // Koristimo ugradeni delegat Action za akcije -
                   akcija je bilo koja metoda kojoj je povratna
                   vrijednost "void"
34              Action<string> akcija1 = Ispis;
35
36              akcija1("Hello world");
37
38              // Koristimo ugradeni delegat Func za funkcije -
                   funkcija je bilo koja metoda kojoj je povratna
                   vrijednost razlicita od "void"
39              Func<bool, double, double> funkcija = SquareOrExp;
40
41              Console.WriteLine(funkcija(true, 2));
42
43              Console.ReadLine();
44          }
45      }
46  }
```

U primjeru 1.5 demonstriramo upotrebu delegata. Primjetite da jednom spremljenu metodu u varijablu, npr. akcija1, možemo pozvati preko imena varijable, npr. akcija1().

Delegat tipa `Func<T, bool>` nazivamo predikat. Delegat koji referencira metodu koja primi jedan parametar i vraća bool vrijednost. Ovakav delegat se najčešće koristi u ispitivanju uvjeta kod određenih elemenata ili skupova elemenata.

1.3 Lambda izrazi

Postoje situacije kod kojih neku funkciju koristimo na jednome mjestu u kodu i nigdje više. Da ne bismo morali definirati potpunu funkciju, u C#-u i još nekim jezicima, postoje takozvane anonimne funkcije. Anonimne funkcije su funkcije koje definiramo unutar neke klase ili metode, a ona ne mora imati svoje ime. Ako funkciju definiramo samo na mjestu na kojem ju koristimo, nije ju uopće potrebno spremati u varijablu. Anonimne funkcije su se u .NET-u 2.0 deklarirale uz pomoć ključne riječi delegate, npr:

```
1           delegate(string s) { Console.WriteLine(s); }
```

U .NET-u 3.0 su uvedeni lambda izrazi kao skraćeni zapis za anonimne funkcije. Funkcija jednaka upravo navedenoj, ali napisana preko lambda izraza bi izgledala:

1. `(String s) => Console.WriteLine(s);`
2. `s => Console.WriteLine(s)` - još kraće

Uočite da nismo morali definirati tip varijable s. To je zbog svojstva koje se zove *Type inference*, a omogućuje kompajleru da samostalno deducira tip podatka. Općeniti oblik anonimne funkcije izgleda:

```
1           (parametri) => { kod koji se mora izvrsiti }
```

U slučaju da u tijelu funkcije nemamo slijed izraza, dakle, ako imamo samo jedan izraz, onda nije potrebno pisati vitičaste zagrade. U suprotnome, ako imamo slijed izraza u tijelu funkcije, onda se koriste viričaste zagrade te se nakon svake linije piše točka-zarez. U slučaju da funkcija uz to vraća rezultat, obavezna je ključna riječ return. Primjer:

```
        () => {
                int a = 5;
                return a + 2;
        }
```

Par primjera:

- `() =>` - funkcija ne prima parametre i ne vraća ništa
- `() => 4` - ne prima parametre, ali vraća broj 4
- `(a, b) => a + b` - prima 2 parametra i vraća njihov zbroj
- `x => x % 2 == 0` - prima 1 parametar i vraća bool vrijednost

Lambda izraze ćemo (između ostalog) koristiti za "zamotati" metodu u funkciju skupa s potrebnim parametrima te novu funkciju poslati u task. O tome smo pričali u potpoglavlju 1.1.2.

Primjer 1.6: Lambda izrazi

```
using System;

namespace LambdaIzrazi
{
    class Program
    {
        static void Main(string[] args)
        {
            Action<string> a1 = (string x) =>
               Console.WriteLine(x);
            Action<string> a2 = x => Console.WriteLine(x);

            Action prazno = () => { };

            Action<string> akcijaSViseLinija = x =>
            {
                x = "Pozdrav " + x;
                Console.WriteLine(x);
            };

            Func<int> f1 = () => 42;
            Func<int, int> f2 = x => x + 42;
            Func<int, int, int> f3 = (x, y) => x + y;

            // Pozivi definiranih akcija
            a1("Hello world");
            a2("Hello world");

            prazno();

            akcijaSViseLinija("Marin");

            // Pozivi definiranih funkcija
            Console.WriteLine(f1());
            Console.WriteLine(f2(5));
            Console.WriteLine(f3(5, 10));

            // Slanje funkcije u metodu kao parametra
            Console.WriteLine(PozoviFunkciju(f2));
```

```
39
40              Console.ReadLine();
41          }
42
43          private static int PozoviFunkciju(Func<int, int> f)
44          {
45              int x = -345;
46
47              return f(x);
48          }
49      }
50  }
```

U primjeru 1.6 je prikazano stvaranje nekoliko lambda izraza i referenciranje istih varijablama.

Primjer 1.7: Taskovi s lambdama

```
1
2   using System;
3   using System.Threading;
4   using System.Threading.Tasks;
5
6   namespace TaskoviSlanjeLambdaIzraza
7   {
8       class Info
9       {
10          private int _broj;
11
12          public int Broj
13          {
14              get { return _broj; }
15              set { _broj = value; }
16          }
17
18          private int _potencija;
19
20          public int Potencija
21          {
22              get { return _potencija; }
23              set { _potencija = value; }
24          }
25
26          public Info(int broj, int potencija)
27          {
28              _broj = broj;
29              _potencija = potencija;
30          }
31      }
32
33      class Program
34      {
35          static double Square(object n)
36          {
37              return Math.Pow((int)n, 2);
38          }
39
```

```
40          static double Square2(int n)
41          {
42              return Math.Pow((int)n, 2);
43          }
44
45          static double Pow(object info)
46          {
47              Info i = (Info)info;
48
49              int m = i.Broj;
50              int n = i.Potencija;
51
52              return Math.Pow(m, n);
53          }
54
55          static double Pow(int m, int n)
56          {
57              return Math.Pow(m, n);
58          }
59
60          static void Print(object msg)
61          {
62              Console.WriteLine(msg);
63          }
64
65          static void Main(string[] args)
66          {
67              string msg = "Hello world";
68
69              // Pokrecemo task i saljemo msg kao objekt u poslanu
                    metodu,
70              // ali u nekim slucajevima onda taj objekt moramo
                    castati (pretvoriti) nazad u originalni tip
                    podatka koji nam je potreban
71              Task t1 = Task.Factory.StartNew(Print, msg);
72
73              // Primjer sa slanjem broja kao objecta. U metodi
                    Square moramo poslani object pretvoriti u int
                    ponovno da bi ga mogli koristiti
74              int broj = 4;
75              Task<double> t2 = Task.Factory.StartNew(Square,
                    broj);
76
77              int potencija = 3;
78
79              Info info = new Info(broj, potencija);
80              Task<double> t3 = Task.Factory.StartNew(Pow, info);
81
82              Console.WriteLine(t2.Result);
83              Console.WriteLine(t3.Result);
84
85              // Da bismo izbjegli pretvaranje parametara u object
                    i nazad u originalni tip podatka, mozemo metodu
86              // koju zelimo poslati u task zapakirati u novu
                    funkciju koja ce sa sobom ponjeti sve
                    informacije koje su potrebne
```

```
87              // metodi da bi ista mogla obaviti svoj posao
88              Task tt1 = Task.Factory.StartNew(() => Print(msg));
89
90              // Uocite razlike izmedu Square i Square2 - nema
                   pretvaranja iz object u int te Square2 prima int
91              Task<double> tt2 = Task.Factory.StartNew(() =>
                   Square2(broj));
92
93              // Uocite da smo zamotali varijable broj i potencija
                   u anonimnu funkciju skupa s metodom koju pozivamo
94              Task<double> tt3 = Task.Factory.StartNew(() =>
                   Pow(broj, potencija));
95
96              Console.WriteLine(tt2.Result);
97              Console.WriteLine(tt3.Result);
98
99              Console.ReadLine();
100         }
101     }
102 }
```

Primjer 1.7 je modificirani primjer 1.4 samo što sada "zapakiramo" metode u lambda izraze skupa s argumentima koji su potrebni metodama te rezultatni lambda izraz pošaljemo u task.

Primjer 1.8: Task + lambda pogreške

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  namespace PakiranjeLambdaIzrazom
6  {
7      class Primjer
8      {
9          private int _a;
10         private int _b;
11
12         public int A
13         {
14             get { return _a; }
15             set { _a = value; }
16         }
17
18         public int B
19         {
20             get { return _b; }
21             set { _b = value; }
22         }
23     }
24
25     class Program
26     {
27         private static void PosaoKojiDugoTraje()
28         {
29             Thread.Sleep(1000);
30         }
```

```
31
32          private static int Zbroji(Primjer primjer)
33          {
34              Console.WriteLine("Primljeno: " + primjer.A + " + "
                    + primjer.B);
35
36              PosaoKojiDugoTraje();
37
38              Console.WriteLine("Zbraja se: " + primjer.A + " + "
                    + primjer.B);
39
40              return primjer.A + primjer.B;
41          }
42
43          private static void Zbroji(int x)
44          {
45              Console.WriteLine("For petlja, prolaz: " + x);
46          }
47
48          static void Main(string[] args)
49          {
50              int a = 5;
51              int b = 3;
52
53              Primjer primjer = new Primjer();
54
55              primjer.A = a;
56              primjer.B = b;
57
58              Task<int> t1 = Task.Factory.StartNew(() =>
                    Zbroji(primjer));
59
60              Thread.Sleep(100);
61
62              primjer.A = primjer.A + 1;
63
64              Console.WriteLine(t1.Result);
65
66              // Pogledajmo jednu cestu pogresku koja nastaje
                    koristenjem ovog pakiranja uz pomoc funkcija
67              for(int i = 0; i < 10; i++)
68              {
69                  Task t = Task.Factory.StartNew(() => Zbroji(i));
70              }
71
72              Console.WriteLine("Malo drugacije");
73
74              for (int i = 0; i < 10; i++)
75              {
76                  Task t = Task.Factory.StartNew(() =>
                        Console.WriteLine(i));
77              }
78
79              Console.ReadLine();
80          }
81      }
```

```
82 }
```

U primjeru 1.8 se kreira i pokreće nekoliko taskova uz pomoć lambda izraza. Ujedno demonstriramo neke tipične probleme kada se lambda izrazi koriste za pokretanje taska.

Prvi od tih je na liniji 58 gdje se objekt šalje po referenci, zatim se čeka s poslom, a u međuvremenu se objekt promijeni. Ono što se dobije jest pogrešan rezultat jer objekt poslan po referenci. Ovo ukazuje na problem koji nastaje kod dijeljenih objekata. Jedna nit promijeni neku vrijednost, a druga zbog toga dobije pogrešan rezultat. Ovo je primjer data race-a.

Drugi od tih je u for petlji na liniji 67. Problem koji se javlja jest da se i može promijeniti prije nego se task pokrene te se stoga mogu dobiti različite vrijednosti u različitm taskovima.

Treći od problema se javlja u for petlji na liniji 74. Naime, lambda izraz pokupi referencu na varijablu i te se pošalje u taskove, dok se taskovi pokrenu, vrijednost i se promijeni prije nego ju stignu ispisat. Stoga svi taskovi ispišu promijenjenu (pogrešnu) vrijednost varijable i.

1.4 Zadatci

1. Napravite program koji generira niz od 200 slučajnih brojeva u rasponu od 0 do 100. Implementirajte metodu Filter koja prima izgeneriranu listu i 1 predikat (Func<T, bool>) te vraća novu listu filtriranih vrijednosti kao rezultat. Korisnika program pita za unos dok ne unese 0. Korisnik odabire hoće li u listi ostati samo parni, neparni ili prosti elementi. Koristiti metodu Filter.
2. Napravite program koji pokreće 2 taska koji igraju ping pong. Oba taska smiju zapisivati u zajedničko polje klase Program. Jedan task zapisuje Ping, a drugi Pong. Svaki task zapisuje svoju vrijednost u polje samo ako ono već ne sadrži njegovu vrijednost. Istovremeno task ispiše vrijednost koju je zapisao i ID niti.
3. Napišite program u kojemu korisnik odabire hoće li se poruka koju unese spremiti u datoteku ili ispisati na konzolu. Odabrani zadatak pokrenite u novome tasku. Nakon što je task gotov, potrebno je ispisati poruku korisniku te zatražiti ponovni unos. Na task možete čekati koristeći metodu Wait taska (npr. ako je task spremljen u varijablu a – a.Wait()).
4. Napravite metodu Compose koja će primiti dva lambda izraza generičkog tipa (Func<T, U> i Func<U, V>) te vratiti kompoziciju tih dviju funkcija Func<T, V>. Compose mora raditi s generičkim tipovima.
5. Napravite metodu Curry koja prima lambda izraz koji prima dva parametra i vraća vrijednost te ga pretvori u njegov *curried* oblik, odnosno vrati delegat koji prima jedan parametar i vraća delegat koji prima drugi parametar, a vraća rezultat. Curry mora raditi s generičkim tipovima.
6. Napravite metodu UnCurry koja je obrnuti postupak od metode Curry. Uncurry prima delegat koji prima jedan parametar i vraća drugi delegat koji prima drugi paramater te vraća rezultat. Uncurry vraća delegat koji prima dva parametra i vraća rezultat. UnCurry mora raditi s generičkim tipovima.
7. Napišite program koji generira niz od 1000 slučajnih brojeva. Korisnik unosi broj n između 1 i 10. Program zatim podijeli niz (ili listu) na n približno jednakih dijelova te za svaki dio pokrene task. Svaki od tih taskova dobiveni dio spremi u datoteku pod imenom brojevi_<ID>.txt. Program korisniku ispiše kada sve niti završe s spremanjem. Koristite **Task.WaitAll()** i niz (ili listu) taskova.
8. Napravite program u kojemu glavna nit pokrene dva nova taska. Prvi task, A, će simulirati podizanje događaja, a drugi, B, stvaranje konekcije na način opisan u nastavku. Task A svako 5 sekundi podiže događaj kojim se vrijednost polja "kreni" postavlja na true (u početku je false). Task B unutar while petlje provjerava vrijednost polja "kreni" te u trenutku kada primjeti da je polje `true` stvori novi task, a zatim polje "kreni" vrati na `false`. Taj novi task

u ovome zadatku smatramo simuliranom konekcijom i želimo da prvo ispiše poruku "Spaja se task s id: <ID>". Zatim taj task u petlji koja se ponavlja N puta svako 1 sekundu ispiše svoj ID i trenutni datum i vrijeme. Kada se petlja izvrši do kraja, ovaj task ispisuje poruku "Odjavljujem se <ID>", a zatim završi s poslom. Za podizanje događaja svako 5 sekundi unutar taska A koristite klasu Timer (nije obavezno).