# Arhitektura distribuiranih sistema u elektroenergetici

Praktikum (2020/2021)

Izvođači nastave :
Bojan Jelačić – bojan.jelacic@uns.ac.rs
Jelena Sekulić – jelena.sekulici@uns.ac.rs
Zorana Babić – zbabic@uns.ac.rs

Sadržaj

Uvod

Polaganje predmeta :

- Predispitne obaveze – 60 bodova
- Ispitne obaveze – 40 bodova
    - Test – 20 bodova
    - Usmeni – 20 bodova

Vežbe :

- Alat koji se koristi prilikom izrade zadataka je Visual Studio (verzija po želji).
- Prisustvo na vežbama je obavezno (maksimalan broj izostanaka je dva).
- Predispitni bodovi se sakupljaju polaganjem dva kolokvijuma.
    - Svaki od kolokvijuma nosi 30 bodova i traje 2,5 sata.
    - Kolokvijum se smatra položenim ukoliko student osvoji minimalno 15.5 bodova
    - Kako bi se položile predispitne obaveze student mora da **položi oba kolokvijuma.**
    - K1 – 24.04.2021. (subota)
    - Oblasti za K1 :
        - V1 - Osnove programskog jezika C#
        - V2 - Klase i osnovni koncepti OOP
        - V3 – Kolekcije podataka
        - V4 – Napredni koncepti
        - V5 – Windows Communication Foundation (1)
    - K2 – 05.06.2021. (subota)
        - Oblasti za K2 - sve oblasti koje su rađene tokom semestra.
    - Popravni kolokvijum
        - Izlaskom na popravni kolokvijum se poništavaju prethodno stečeni bodovi.
        - Studenti mogu da popravljaju samo **jedan** od kolokvijuma.

# Vežba 1 – Osnove programskog jezika C#

Cilj ove vežbe je da pruži osnovno razumevanje *.NET framework* softverske platforme, kroz objašnjenje osnovnih koncepata *programskog jezika C#* razvijenih u .NET softverskom okruženju, u okviru *Microsoft Visual Studio* razvojnog okruženja.

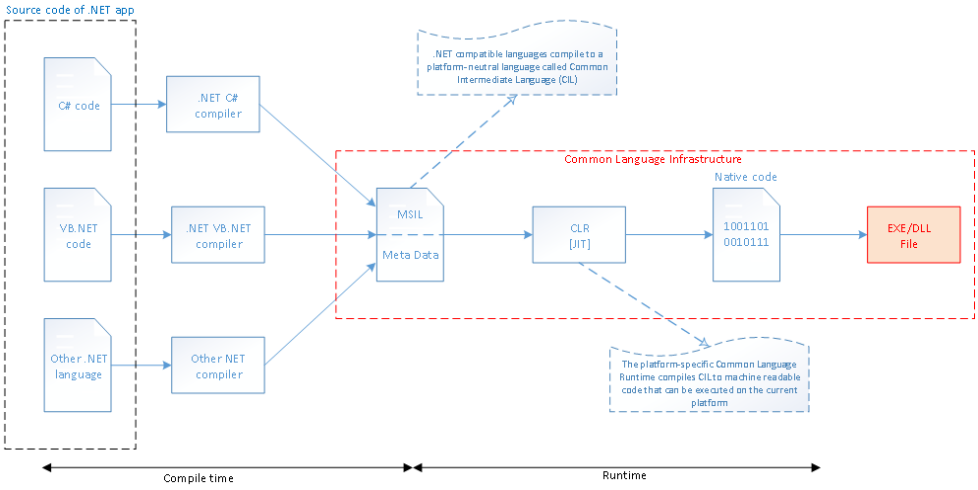
## .NET Framework

.NET Framework predstavlja kontrolisano programsko okruženje za razvoj distribuiranih aplikacija na Windows operativnom sistemu. .NET podržava više programskih jezika (Visual Basic, C++, C#) izvršava u softverskom okruženju *Common Language Runtime (CLR)*, koje u potpunosti kontroliše interakciju sa operativnim sistemom. Na taj način se obezbeđuje pisanje aplikacija nezavisno od platforme na kojoj će se aplikacija izvršavati. *CLR* je virtuelna mašina koja omogućava kontrolisano izvršavanje programa pisanih za .NET, uključujući i kontrolu izvršavanja niti, upravljanja memorijom, rukovanje izuzecima i bezbednost aplikacije.

Kod koji se izvršava od strane CLR-a, odnosno u .NET runtime okruženju, naziva se *managed code*. Managed code je izolovan u smislu da mu se ne može pristupiti van *runtime* okruženja, niti se može pozvati direktno van *runtime* okruženja, što ga čini pouzdanijim i robusnijim u odnosu na *unmanaged code*. Za razliku od *managed* koda, *unmanaged code* podrazumeva direktnu interakciju sa operativnim sistemom što predstavlja određeni sigurnosni rizik, npr. preko povećane verovatnoće pojave nenamernih grešaka u izvornom kodu (tzv. bug-ova).

.NET Framework uključuje i velik broj biblioteka klasa (tzv. *Framework Class Library*) koje pružaju širok spektar mogućnosti, uključujući klase za razvoj korisničkog interfejsa, pristup bazama podataka, razvoj distribuiranih i veb aplikacija, podršku za konkurentno programiranje, itd. *.NET Framework Class Library* su biblioteke nezavisne od programskog jezika, a organizovane su u logičke grupe, tzv. prostore imena (engl. *namespace*).

Na slici 1. je prikazano *.NET runtime okruženje*. Prilikom kompajliranja, izvorni kod pisan u nekom od .NET programskih jezika se prevodi u *Common Intermediate Language (CIL)*, poznat i kao *Microsoft Intermediate Language (MSIL)*. CIL, odnosno MSIL, predstavlja skup instrukcija koji je nezavisan od platforme, a koji se može efikasno prevesti u mašinski/native code. Rezultat kompajliranja je *.NET assembly* koji enkapsulira sve podatke o aplikaciji, čime predstavlja osnovnu jedinicu za isporuku, verzionisanje i ponovnu upotrebu aplikacije. *Just In Time (JIT) compiler* konvertuje MSIL kod u skup mašinskih instrukcija u toku izvršavanja. CLR obezbeđuje različite JIT programske prevodioce (tj. kompajlere), specifične za određenu arhitekturu. .NET CLR rezultuje DLL ili EXE fajlom.




*Slika 1. .NET Runtime Okruženje*

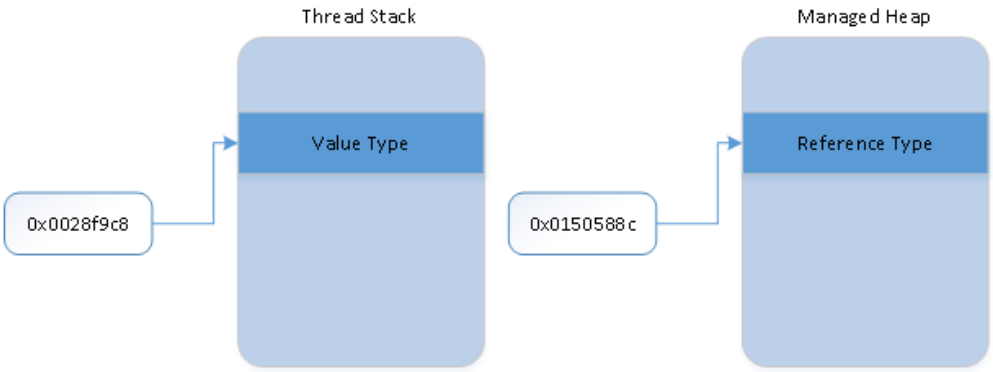
## Osnovni koncepti C# programskog jezika

C# je objektno-orijentisan programski jezik razvijen u okviru .NET softverskog okruženja.

### Osnovni tipovi podataka i osnovni operatori u C#

Tip (*type*) je osnovna jedinica programabilnosti u .NET. Postoje dve kategorije tipova podataka:

1. **Vrednosni tipovi (eng. *value types*).** Vrednosni tipovi podataka se izvode iz klase **System.ValueType**. U ovu grupu spadaju: primitivni tipovi podataka (*int, double, bool, char, itd.*), strukture (*struct*) i enumeracije (*enum*). Vrednosti se ovim promenljivama dodeljuju direktno, odnosno svaka promenljiva sadrži direktno dodeljenu vrednost. Ukoliko se prilikom definisanja promenljive ne navede inicijalna vrednost, biće dodeljena podrazumevana vrednost u zavisnosti od tipa. Ovi tipovi podataka se smatraju relativno malim podacima koji se čuvaju direktno na steku(stack) aktivne niti.
2. **Referentni tipovi (eng. *reference types*).** Referentni tipovi podataka ne sadrže vrednost promenljive, već predstavljaju referencu na memorijsku lokaciju na kojoj se promenljiva skladišti. Prema tome, ovaj tip ima dva dela: objekat i referencu na taj objekat. Prilikom dodele instance referentnog tipa drugoj instanci, kopira se referenca instance, ali ne i objekat. Na taj način, više promenljivih ovog tipa mogu da pokazuju na istu memorijsku lokaciju (isti objekat), što implicira da će se izmena vrednosti od strane jedne promenljive reflektovati i na druge promenljive. Promenljive ovog tipa se skladište u okviru posebne memorijske strukture (*heap*) i u potpunosti je kontrolisano od strane *Garbage Collectora (GC)*. Treba imati u vidu da je skladištenje podataka na *heap*-u i njihova kontrola od strane GC-a je skupa operacija. Iz tog razloga je objekte koji su relativno mali i kratkog životnog veka (*short-lived*) efikasnije skladištiti na steku niti u okviru koje se kreiraju. U referentne tipove spadaju klase, nizova, delegati i interfejsi.




*Slika 2. Skladištenje Value i Reference tipova podataka*

## Grananja u C#

Grananje, odnosno *decision making structure*, je iskaz koji podrazumeva izvršavanje različitih izraza u zavisnosti od toga da li su određeni uslovi ispunjeni ili ne, odnosno da li je vrednost uslova tačna (*true*) ili netačna (*false*). Tipične strukture grananja u C# su navedene ispod.

1. **IF struktura grananja.** Ukoliko je *Boolean* izraz tačan, kod unutar *if* bloka će biti izvršen. Ukoliko je *Boolean* izraz netačan, kod unutar *if* bloka će biti preskočen, i program će nastaviti sa prvom sledećom naredbom van *if* bloka.

```
if (uslov)
{
    // kod koji će biti izvršen ukoliko je uslov zadovoljen
}
```

2. **IF ... ELSE struktura grananja.** Ukoliko je Boolean izraz tačan, kod unutar if bloka će biti izvršen. Ukoliko je Boolean izraz netačan, kod unutar else bloka će biti izvršen.

```
if (uslov)
{
    // kod koji će biti izvršen ukoliko je uslov tačan
}
else
{
    // kod koji će biti izvršen ukoliko je uslov netačan
}
```

3. **IF ... ELSE IF struktura grananja.** Struktura koja se koristi ukoliko postoji više različitih uslova u zavisnosti od kojih će se izvršavati različiti programski blokovi. Kada se naiđe na prvi tačan izraz u okviru *if* uslova, ostali uslovi se neće proveravati. U slučaju da se poslednji *else* izraz ne navede, ukoliko nijedan od navedenih izraza nije bio tačan, nastavlja se sa prvom sledećom naredbom van ove strukture.

```
if (uslov1)
{
    // kod koji će biti izvršen ukoliko je uslov1 zadovoljen
}
else if (uslov2)
{
    // kod koji će biti izvršen ukoliko je uslov2 zadovoljen
}
// dodatni else if blokovi
else
{
    // kod koji će se izvršava ako nijedan od uslova nije
zadovoljen
}
```

4. **SWITCH-CASE struktura grananja.** Omogućuje izvršavanje određenog programskog bloka na osnovu provere jednakosti promenljive (odnosno *switch* izraza) sa listom vrednosti koje su definisane kao konstante tipa kome taj izraz odgovara (a koje se nazivaju *case*). *Default* blok je opcioni. Ukoliko postoji, a nijedan prethodni iskaz nije tačan, kod u okviru podrazumevanog bloka će biti izvršen.

```
switch (izraz)
{
case vrednost1:
    // kod koji se izvršava ako je izraz jednak vrednost1
    break;
case vrednost2:
    // kod koji se izvršava ako je izraz jednak vrednost2
    break;
    // dodatne case grane
default:
    // kod koji se izvršava ako izraz ne spada u grupu
vrednosti
    break;
}
```

### Petlje u C#

Petlje (*loops*) predstavljaju kontrolne strukture koje obezbeđuju da se određeni programski blok izvrši određeni broj puta. Tipične kontrolne strukture, odnosno petlje u C# su navedene ispod.

1. **FOR petlja.** Sintaksa *for* petlje je data u nastavku. ***Init*** je korak koji se izvršava prvi put, i samo jednom. U ovom koraku mogu se deklarisati i inicijalizovati promenljive kontrolne strukture, npr. brojači. ***Condition*** je uslov koji se računa u svakom prolazu kroz petlju i ukoliko je rezultat tačan, telo petlje će se izvršiti. U suprotnom, telo petlje se neće izvršiti, a program nastavlja sa prvom naredbom nakon for petlje. ***Increment*** je skup izraza koji se izvršava nakon što je izvršeno telo petlje, npr. da bi se ažurirala vrednost brojača. Nakon ovog koraka, ponovo se proverava uslov zadat kroz *condition* i sve dok je ta vrednost tačna, telo petlje će se izvršavati.

```
for (int i=0; i<10; i++)
{
    // kod koji se ponavlja
}
```

2. **WHILE petlja.** Sintaksa while petlje je data u nastavku. While petlja je kontrolna struktura koja obezbeđuje ponavljanje programskog bloka sve dok je ***condition*** tačan. Na ovaj način, ukoliko je pre početka izvršavanja while petlje uslov netačan, telo petlje se neće izvršiti nijednom.

```
while (uslov)
{
    // kod koji se ponavlja
}
```

3. **DO-WHILE petlja**. Za razliku od FOR ili WHILE petlji koje pre izvršavanja tela petlje proveravaju da li je uslov zadovoljen, DO-WHILE petlja proverava uslov nakon izvršavanja tela petlje. Na taj način se garantuje da će telo petlje biti izvršeno najmanje jednom, a izvršavaće se sve dok je uslov (*condition*) tačan. Sintaksa DO-WHILE petlje u C# je data u nastavku.

```
do
{
    // kod koji se ponavlja
} while (condition);
```

### *Break i Continue naredbe*

**Break** i **continue** naredbe su iskazi koji menjaju sekvencijalni redosled izvršavanja programa. Kada se ***break iskaz*** koristi u telu petlje, on zaustavlja izvršavanje tela petlje, i prebacuje izvršavanje programa na prvu naredbu nakon petlje. ***Continue iskaz*** prekida izvršavanje tela petlje, i prebacuje izvršavanje programa na proveru uslova iste petlje, a u zavisnosti od rezultata uslova petlje, telo petlje će se izvršiti ili će program izaći iz petlje.

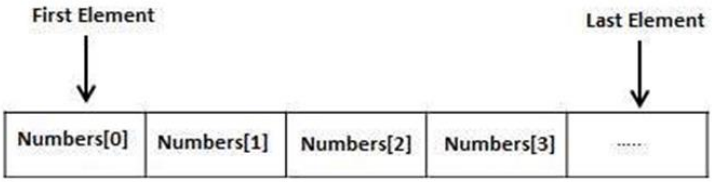
### *Beskonačne petlje*

Beskonačne petlje su petlje u kojima uslov nikada neće postati netačan, i samim tim izvršavanje programa nikada neće izaći van tela petlje. Tipičan način pisanja beskonačne petlje je: **for (; ;)**. Naime, kada uslov nije naveden podrazumeva se da je on tačan. Ukoliko je potrebno, inicijalizacija i inkrement u okviru petlje se mogu navesti, ali bez navedenog uslova petlja je beskonačna. Izlazak iz ove petlje je moguć korišćenjem naredbe *break*.

## Nizovi u C#

Niz je sekvencijalna kolekcija elemenata istog tipa. Umesto deklarisanja svake promenljive posebno (*npr. int num0, int num1, ... int numN*), moguće je definisati niz promenljivih (istog tipa) kojima se pristupa preko indeksa.

Niz predstavlja kolekciju promenljivih istog tipa smeštenih u susedne memorijske lokacije – najniža memorijska adresa odgovara prvom elementu, a najviša adresa odgovara poslednjem elementu niza (vidi Slika).




*Slika 3. Zauzimanje memorije prilikom inicijalizacije niza*

### *Deklarisanje i inicijalizacija niza*

Deklarisanje niza ne podrazumeva i njegovu inicijalizaciju, odnosno zauzimanje memorijskog prostora za elemente tog niza. Niz je referentni tip i prilikom inicijalizacije potrebno je koristiti ključnu reč *new*. Primer inicijalizacije niza:

```
uint N = 10;
double[] Numbers = new double[N + 1];
Numbers[0] = 4500.5;
```

U listingu iznad je prikazan C# kod za inicijalizaciju niza koji sadrži realne brojeve. U poslednjoj linijia listinga iznad je prikazan način pristupa i dodele vrednosti elementima niza se koristi indeks niza.

### Višedimenzionalni nizovi

U C# je moguće definisati i takozvane višedimenzionalne nizove. Najjednostavnija forma višedimenzionalnog niza je 2-dimenzionalni niz, a koji predstavlja niz čiji su elementi 1-dimenzionalni nizovi. Dvodimenzionalni niz se može zamisliti kao matrica, odnosno tabela sa X vrsta i Y kolona (kao na slici 2). Elementima ovakvog niza (matrice) se pristupa preko dva indeksa – *i* i *j*, na sledeći način: ***a[i, j]***.

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

Slika 2. Primer dvodimenzionalnog niza sa 3 vrste i 4 kolone

### Kreiranje novog C# projekta u Visual Studio

Postupak kreiranja novog projekta je izuzetno jednostavan u VS i sastoji se od sledećih koraka:

- Kreirati novi C# projekat:
  - *File* → *New* → *Project* → Izabrati Visual C# project template
- Podešavanjekonfiguracije projekat u zavisnosti od platforme na kojoj se pokreće VS:
  - *Configuration Manager* → Podesiti aktivnu konfiguraciju projekta, kao i platformu
    - *Debug* konfiguracija
    - *Release* konfiguracija
  - *Project properites* → *Build* → *Output path*

## C# Programska struktura

Struktura svakog C# datoteka sastoji se iz sledećih elemenata: deklaracija prostora imena, deklaracija klase, komentari i ostali izrazi i iskazi koji definišu ponašanje procesa. Ove delove programa ćemo detaljnije prikazati ispod.

### *Namespace deklaracija.*

*Namespace* je način organizacije različitih klasa u logičke grupe, čime se omogućuje kontrola opsega klasa i metoda u okviru većih projekata. Da bi se u programu koristile klase iz određenog *namespace*-a, navodi se ključna reč *using* kojom se definiše da dati program koristi imena iz navedenog namespace-a. Prva linija *using System;* označava da će prostor imena *System* biti uključen u program. Moguće je navesti više *using* izraza. Nakon toga sledi definicija prostora imena u okviru projekta sa ključnom reči *namespace*, nakon čega se navodi ime *namespace*-a (videti listing ispod).

```
namespace v01_uvod
{
    // izvorni kod, npr. interfejsi, klase
}
```

### *Deklaracija klase*

Deklaracija C# klase se sastoji od imena klase, deklaracije promenljivih i metoda klase. Osnovna klasa, class *Program*, sadrži jednu statičku metodu po imenu Main, koja predstavlja ulaznu tačku svake C# aplikacije.

**Mini zadatak:** Preimenovati klasu Program u *MyFirstCSharpProgram*. Obratiti pažnju da prilikom promene imena razvojno okruženje VS nudi mogućnost promene imena promenljive/metode na nivou celog projekta kako bi se sprečile potencijalne greške.

### *Komentari*

Dobra programerska praksa nalaže da komentari treba da budu sastavni elementi svakog komada izvornog koda. Iako programski prevodilac ignoriše linije koda koje predstavljaju komentar, treba ih navoditi radi lakšeg razumevanja i održavanja napisanog koda. U zavisnosti od potrebe, komentari se mogu definisati na sledeća tri načina (komentar u jednoj liniji, više-linijski, opis klase/metoda) prikazan u listinzima ispod:

```
// komentar u jednoj liniji
```

```
/* višelinijski komentar - prva linija
druga linija
…
poslednja linija komentara */
```

```
/// opis klase i/ili, metoda, „otvara” se sa tri kose crte /
/// <summary>
///
/// </summary>
/// <param name="args"></param>
```

*Rukovanje ulazima i izlazima u konzolnom prozoru*

Klasa Console definisana u okviru prostora imena *System* je klasa za ispis na konzolu, kao i čitanje sa konzole.

U nastavku su kroz praktične primere objašnjeni osnovni koncepti potrebni za razumevanje osnova programiranja u programskom jeziku C#. Kao razvojno okruženje koristi se Microsoft Visual Studio (VS).

## Zadatak 1. Hello World

Ispisati “Hello World!” na konzoli. Obezbediti čekanje na pritisak tastera sa konzole kako bi program nastavio sa radom. Ispisati vrednost prethodno unetu kroz konzolu od strane korisnika.

## Zadatak 2. Int vs double

Prodiskutovati *result = x / y* u zavisnosti od tipa promenljivih (*int* ili *double*).

```
int res;
double a = 7, b = 3;
res = a / b;
```

Eksplicitno kastovanje daje na znanje programskom prevodiocu u kom formatu se želi tumačiti vrednost rezultujuće promenljive. Ukoliko se ne kastuje eksplicitno, programski prevodilac će implicitno da kastuje u trenutku kada to bude neophodno. Treba voditi računa da nije moguće **implicitno** kastovati vrednosti tipa koji ima veći opseg u vrednost tipa koji ima manji opseg, npr. *double* vrednost u *int* u listingu izad.

## Zadatak 3. Postfix i prefix operatori

Prodiskutovati razliku između *prefix* i *postfix* inkrementa (**++x** vs **x++)**. Šta je rezultat izvršavanja sledećeg programa?

```
int a, b, x = 10;
a = ++x;
b = x++;
Console.WriteLine("a ={0}, b ={ 1}, x ={ 2}", a, b, x);
```

Uveriti se da

1. *Prefix* inkrement vraća vrednost x nakon ažuriranja (uvećanje za 1) lokacije promenljive x
2. *Postfix* inkrement vraća vrednost promenljive x, a zatim ažurira lokaciju te promenljive (uvećava za jedan).

## Zadatak 4. Osnovni unos podataka

Napisati program koji proverava da li je uneti odgovor (*string answe*r) potvrdan ("da" ili "yes", dozvoljene su sve kombinacije malog i velikog slova), i u skladu sa tim postaviti promenljivu *bool isAffirmative* na odgovarajuću vrednost.

Napomena: Za poređenje string vrednosti koristiti metode klase **System.String**.

## Zadatak 5. Ternarni iskaz

Umesto if-else iskaza, zadatak 4. implementirati korišćenjem uslovnog (*conditional*) operatora. Forma *conditional* operatora je:

**t ? x : y** – ukoliko je iskaz t tačan, izračunaj i vrati vrednost x; u suprotnom, izračunaj i vrati vrednost y

## Zadatak 6. Višestruka grananja

Za IF-ELSE iskaz u primeru ispod napisati ekvivalentan *switch-case* iskaz. Uporediti brzinu izvršavanja *if-else* u odnosu na ekvivalentan *switch-case* iskaze.

```
int a;
if (a == 1)
{
    Console.WriteLine("a = 1");
}
else if (a == 2)
{
    Console.WriteLine("a = 2");
}
//..
else if (a == 5)
{
    Console.WriteLine("a = 5");
}
else
{
    Console.WriteLine("Nepoznata vrednost.");
}
```

Ukoliko *swith-case* iskaz sadrži više od pet *case* blokova, implementira se koristeći *lookup* tabelu ili *hash* listu. Na taj način se obezbeđuje da je za svaki *case* izraz potrebno isto vreme pristupa, za razliku od liste *if-else* izraza gde se izrazima pristupa redosledom kojim su i navedeni. To znači da pre nego što se dođe do poslednjeg *if-else* izraza potrebno je proći kroz sve prethodne izraze, čime se može znatno produžiti vreme izvršavanja koda.

## Zadatak 7. Efikasno korišćenje petlji #1

Napisati program koji na korisničkoj konzoli prikazuje zvezdice (*) kao na slici.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

## Zadatak 8. Efikasno korišćenje petlji #2

Napisati program koji inicijalizuje niz od deset elemenata tipa *int* tako da vrednost svakog elementa niza odgovara vrednosti indeksa tog elementa u nizu uvećana za 100. Nakon toga, potrebno je ispisati sve elemente tog niza.

## Zadatak 9. Foreach petlja

Izmeniti način iteriranja kroz niz koristeći **foreach** petlju.
S obzirom da niz predstavlja kolekciju podataka, za prolazak kroz sve elemente niza moguće je koristiti *foreach* petlju. Sintaksa *foreach* petlje u C# je data ispod. Dakle, izvršavanje tela petlje se završava kada se prođe kroz sve elemente niza (u opštem slučaju, umesto niza može biti bilo koja kolekcija podataka. Slično kao i kod ostalih petlji, i izvršavanje tela *foreach* petlje moguće je prekinuti naredbom break, nakon čega program nastavlja sa izvršavanjem prve sledeće naredbe van petlje.

```
foreach (double vrednost in Numbers)
{
    // kod za obradu vrednosti
}
```

## Zadatak 10. Višedimenzioni nizovi

Proizvoljno inicijalizovati 2-dimenzionalni niz sa N=10 vrsta i kolona, i sabrati sve elemente niza ispod glavne dijagonale (uključujući i elemente na samoj dijagonali).

# Vežba 2 – Klase i osnovni koncepti objektno-orijentisanog programiranja

Cilj vežbe je da pruži razumevanje pojma klase i osnovnih koncepata objektno-orijentisanog programiranja.

Klasa predstavlja strukturu podataka koja omogućuje definisanje novog tipa koji enkapsulira ponašanje objekata iste klase. Kroz klasu se definišu promenljive i metode koje opisuju ponašanje objekata date klase. Objekat predstavlja instancu klase koji zauzima određen memorijski prostor prilikom inicijalizacije. Klasa je referentni tip (eng. *reference type*) tako da se objekti skladište na heap-u.

## Klase u C#

Definicija klase u programskom jeziku C# počinje ključnom reči **class**, zatim ime klase, nakon čega slede definicije članova klase, tj. promenljiivh i metoda koji čine klasu. Ime klase postaje novi tip koji se može koristiti za kreiranje objekata klase, odnosno promenljivih koje su tipa te klase. Kreiranje, odnosno inicijalizacija objekata se obavlja u konstruktoru klase. Ukoliko se konstruktor ne definiše eksplicitno, poziva se podrazumevani konstruktor bez parametara.

Specifikator pristupa (*access specifier*) definiše nivo pristupa kako za objekte određene klase, tako i za članove klase:

- ***public:*** članovi klase definisani kao *public* su vidljivi i van te klase, odnosno dostupni su izvan klase kako u okviru istog asemblija tako i iz drugih assemblija koji referenciraju taj.
- ***private:*** članovi klase definisani kao *private* su zaštićeni od spoljnog pristupa, i njima mogu pristupati isključivo funkcije članice iste klase.
- ***protected:*** članovi klase kojima mogu pristupati funkcije članice iste klase, ali i iz klase nastale nasleđivanjem te klase.
- ***internal:*** članovi klase koji su dostupni samo u okviru istog assemblija.
- ***protected internal:*** članovi klase koji su dostupni u okviru istog assemblija, ali i iz klasa definisanih u drugom assembliju ukoliko nasleđuju datu klasu.

Podrazumevani nivo pristupa je *internal* za klase, odnosno *private* za članice klasa. Članovima klase se pristupa navođenjem tačke (.) nakon imena objekta čijem članu se pristupa.

## Property

U programskom jeziku C# je polje klase (*field*) promenljiva bilo kog tipa definisana u okviru klase. **Property** je član klase kojim se obezbeđuje pristup poljima te klase, odnosno kojim je moguće obezbediti manipulisanje (čitanje, pisanje) vrednosti polja. Praksa za pisanje lepog i održivog koda je da su polja klase uvek definisana kao *private*, i da se izlažu spoljašnjem svetu preko Property-a. Property-ju se, kao i ostalim članovima klase, može pristupati u zavisnosti od definisanog specifikatora pristupa. Sintaksa Property-a u C# je data ispod. *Property* Ime obezbeđuje manipulisanje *poljem ime* van klase.

```
class Student
{
    private string ime;
    public string Ime
    {
        get { return ime; }
        set { ime = value; }
    }
}
```

## Statički članovi klase u C#

Kada je član klase definisan kao statički (ključna reč **static**), to znači da bez obzira koliko je objekata ove klase instancirano, postoji tačno jedna kopija tog člana klase zajednička za sve objekte. Statičkim članovima klase se može pristupati bez obzira da li postoji instanciran objekat te klase. Statičke promenljive se uglavnom koriste za definisanje konstanti koje su zajedničke za sve objekte date klase. Statičke metode mogu pristupati isključivo statičkim poljima klase, i takođe mogu se pozivati i pre kreiranja objekata. Ukoliko je klasa definisana kao statička, to znači da nije moguće instancirati objekte ove klase. Statičkim članovima klase se pristupa koristeći ime klase, umesto imena konkretnog objekta klase.

## Osnovni koncepti OOP

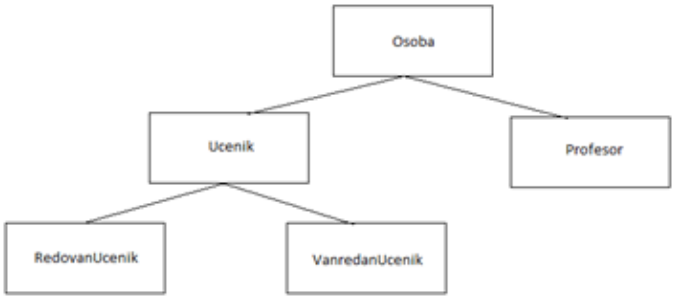
Osnovni koncepti objektno-orijentisanog programiranja (OOP) su:

1) enkapsulacija (*encapsulation*),
2) nasleđivanje (*inheritance*),
3) polimorfizam (*polymorphism*).

Pod **enkapsulacijom** se najčešće podrazumeva mogućnost programskog jezika koja omogućava grupisanje i zaštitu srodnih podataka, npr. u okviru klase.

**Nasleđivanje** je mogućnost da se na osnovu postojećih klasa izvedu nove klase koje treba da prošire, iskoriste ili izmene ponašanje definisano u postojećim klasama. Klasa koja se nasleđuje se naziva bazna klasa (*base class*), a koja nasleđuje drugu klasu se naziva izvedena (*derived class*). Nasleđivanje može biti jednostruko ili višestruko. C# podržava jednostruko nasleđivanje, odnosno jedna klasa može imati najviše jednu baznu klasu. Jednu baznu klasu može da nasledi više izvedenih klasa, čiji broj može biti neograničen.

Grupa klasa koje su povezane nasleđivanjem formiraju strukturu koja se naziva *hijerarhija klasa*. Dubina hijerarhije predstavlja broj nivoa nasleđivanja. Preporuka je da hijerarhija bude najviše dubine 7. Na slici 4 je dat primer hijerarhije dubine dva. Hijerarhiju čini bazna klasa *Osoba*, koju nasleđuju klase *Učenik* i *Profesor*. Klasa *Učenik* ima dve implementacije, *RedovanUčenik* i *VanredanUčenik*.




*Slika 3. Hijerarhija uloga*

U primeru ispod prikazana je sintaksa nasleđivanja. Javni članovi bazne klase su implicitno i javni članovi izvedene klase. Nasleđivanje ne podrazumeva da će izvedena klasa imati pristup svim članovima bazne klase. Privatni članovi bazne klase, iako su nasleđeni, su dostupni isključivo članovima bazne klase. Članovi bazne klase sa modifikatorom *protected* su jedino dostupni unutar bazne klase i direktno i indirektno izvedenim klasama. Ukoliko modifikator pristupa nije naveden podrazumeva se da je *private*. Dostupnost izvedene klase je uslovljena dostupnošću bazne klase. Ako je bazna klasa privatna, izvedena ne može biti javna. Osim korišćenja implementiranih funkcionalnosti ili implementacije apstraktnih metoda, u izvedenoj klasi se mogu dodati i novi članovi, specifični samo za datu klasu. Za poziv konstruktora bazne klase iz konstruktora izvedene klase se koristi ključna reč **base**.

```
class BaznaKlasa
{
    public BaznaKlasa()
    {
    }
    public BaznaKlasa(int i)
    {
    }
    public void PosaljiPoruku(string poruka)
    {
    }
}
```

```
class IzvedenaKlasa : BaznaKlasa
{
    // Konstruktor izvedene klase
    public IzvedenaKlasa(string s)
    {
    }

    // Esplicitni poziv konstruktora bazne klase
    public IzvedenaKlasa() : base()
    {
    }

    public IzvedenaKlasa(int i) : base(i)
    {
    }
}
```

```
// kreiranje instance izvedene klase i poziv metoda bazne klase
IzvedenaKlasa izv = new IzvedenaKlasa(6);
izv.PosaljiPoruku("Tekst poruke");
```

**Polimorfizam** se zasniva na ideji da metoda, deklarisana u osnovnoj klasi, može biti implementirana na više različitih načina u različitim izvedenim klasama. Polimorfizam se realizuje preko virtuelnih metoda. Za deklaraciju virtuelnih metoda se koristi ključna reč **virtual**. Virtuelne metode se mogu re-implementirati u izvedenim klasama, kada se koristi ključna reč **override**. Pritom, virtuelna i re-implementirana metoda moraju imati 1) isto ime, 2) isti modifikator pristupa, 3) isti tip rezultata i 4) iste tipove parametara. Sintaksa virtuelne i re-implementirane metode može se videti u primeru ispod.

```
class BaznaKlasa
{
    public virtual void PosaljiPoruku(string poruka)
    {
        // virtuelni metod
    }
    }
}
```

```
class IzvedenaKlasa : BaznaKlasa
{
    public override void PosaljiPoruku(string poruka)
    {
        // dodatne akcije za slanje poruke
    }
}
```

Ukoliko ne želimo da drugi klase nasleđuju neku klasu, to možemo učiniti pomoću ključne reči **sealed**, a takve klase se nazivaju *zapečaćene klase*. Definicija takve klase data je u primeru ispod.

```
public sealed class ZapecacenaKlasa
{
}
```

## Interfejsi

Interfejs definiše zajedničke elemente (metode i property-e) svih klasa koje nasleđuju taj interfejs. Interfejs sadrži deklaraciju članova klase bez implementacije istih. Klase koje nasleđuju interfejse moraju implementirati sve članove interfejsa. Za deklaraciju interfejsa koristi se ključna reč **interface**, nakon čega sledi ime interfejsa. Kao nepisano pravilo, imena interfejsa uvek počinju velikim slovom 'I'. C# omogućava jednostruko nasleđivanje klasa (tj. moguće je naslediti samo jednu klasu), dok je moguće naslediti više interfejsa.

**Apstraktne klase** su koncept usko povezan sa interfejsima. Za ove klase se ne mogu instancirati objekti, mada za razliku od interfejsa (koji nikad ne sadrže implementaciju), pojedine metode apstraktne klase mogu biti implementirane. Ključna razlika između interfejsa i apstraktne klase je činjenica da klase mogu implementirati neograničen broj interfejsa, dok najviše jednu apstraktnu klasu mogu naslediti. Klasa koja nasledi jednu apstraktnu, i dalje može implementirati neograničen broj interfejsa. Značaj apstraktnih klasa se ogleda u tome što je neke funkcionalnosti moguće implementirati kao zajedničke za sve klase naslednice, dok se druge metode specifične samo za klase naslednice mogu naknadno implementirati kada i sama klasa naslednica. Dakle, svaka konkretna klasa koja implementira interfejs mora implementirati sve njegove metode, dok se u slučaju apstraktne klase neke članice mogu implementirati, a neke članice se deklarisati kao apstraktne i implementirati u klasi naslednici.

Deklaracija apstraktne klase u C# počinje ključnom reči **abstract**, nakon čega sledi standardna C# deklaracija klase. Takođe, metode za koje se očekuje da budu implementirane u klasama naslednicama se označavaju ključnom reči **abstract**. Apstraktna klasa je klasa koja se može naslediti, ali se ne može instancirati. Može posedovati apstraktne metode koje nemaju svoju implementaciju, ali ih klase naslednice moraju implementirati korišćenjem ključne reči **override**.

## Zadatak 1. Klasa za rad sa pravougaonicima

Definisati klasu *Rectangle* koja opisuje pravougaonik. Objekat pravougaonika je opisan određenom dužinom (*double width*) i širinom (*double length*). Podrazumevani objekti ove klase treba da imaju dužinu i širinu 1. Omogućiti kreiranje objekta proizvoljnih dimenzija. Za objekte klase *Rectangle* obezbediti izračunavanje površine i obima objekta.

## Zadatak 2. Članice klase za pristup atributima

Obezbediti javni pristup privatnim poljima klase *Rectangle*.

## Zadatak 3. Statički metodi

Implementirati statički metod u okviru klase *Rectangle* koja vraća broj ukupno kreiranih objekata ove klase. U okviru funkcije *Main* pozvati ovaj metod i ispisati povratnu vrednost.

## Zadatak 4. Interfejs klase Rectangle

Napisati interfejs koji odgovara klasi *Rectangle* implementiranoj kroz prethodne primere. Dodatno, u interfejsu definisati i novi metod *void ShowInfo()* koja treba da ispiše podatke (u konzoli) o dimenzijama objekata *Rectangle*.

## Zadatak 5. Rad sa bankovnim računima

Napisati program koji omogućava uplatu i isplatu na devizni i tekući račun, kao i ispis stanja računa nakon datih uplata i isplata. Provizija na uplatu na tekući račun je 0, a provizija na isplatu je 3%. Provizija na uplatu i isplatu sa deviznog računa je 5%. I devizni i tekući račun su definisati svojim jedinstvenom brojem (broj računa), kao i početnim stanjem računa.

# Vežba 3 – Kolekcije podataka

Cilj ove vežbe je da unapredi znanje u radu sa različitim tipovima kolekcija podataka u okviru programskog jezika C#.

## Liste u C#

Lista predstavlja C# kolekciju podataka istog tipa kojima se pristupa preko indeksa. Odnosno, lista predstavlja dinamički niz ili vektor. Pristupanje elementima niza preko indeksa je brza operacija. Prilikom dodavanja elemenata u listu, korisnik ima potpunu kontrolu kada je u pitanju redosled elemenata te liste. Međutim, dodavanje i uklanjanje elemenata sa početka ili sredine liste je prilično skupa operacija, jer podrazumeva pomeranje (engl. shift) svih elemenata na steku u zavisnosti od toga da li se element dodaje ili uklanja.

### Inicijalizacija liste

Podrazumevano se liste inicijalizuju sa kapacitetom nula (tj. prazna lista). U situaciji kada je potrebno dodati novi element u listu, a kapacitet liste je dostignut, veličina liste se udvostručava. To znači da ukoliko je podrazumevani kapacitet liste nula, prilikom dodavanja prvog elementa lista će biti reinicijalizovana na četiri, kada dostigne kapacitet biće ponovo reinicijalizovana na dužinu osam, itd. U slučaju liste sa velikim brojem elemenata ovo može uzrokovati nepotrebno zauzimanje memorijskog prostora.

```
// Podrazumevana inicijalizacija liste:
List<int> listObj = new List<int>();
```

Zbog gore navedenog je prilikom inicijalizacije liste potrebno voditi računa da se lista kreira sa unapred definisanim kapacitetom kad god je to moguće. Na taj način se znatno poboljšava rukovanje memorijom. U sledećem listingu je prikazana inicijalizacija liste sa unapred definisanim kapacitetom:

```
// Inicijalizacija liste sa unapred definisanim kapacitetom:
List<int> listObj = new List<int>(157345);
```

Kad god su vrednosti elemenata liste unapred poznati, najbolja praksa je definisati vrednosti tih elemenata prilikom inicijalizacije liste, umesto kasnijeg višestrukog pozivanja metoda *Add*. Dakle, umesto:

```
List<int> listObj = new List<int>();
listObj.Add(5);
listObj.Add(15);
listObj.Add(25);
```

elemente liste treba inicijalizovati na sledeći način:

```
List<int> listObj = new List<int>() { 5, 15, 25 };
```

## Zadatak 1. Inicijalizacija liste

Inicijalizovati listu celobrojnih (*integer*) vrednosti od ukupno 154357 elemenata. Vrednost svakog elementa liste inicijalizovati na sledeći način:

- ukoliko je indeks elementa deljiva sa 4, upisati vrednost indeksa,
- u suprotnom, upisati vrednost indeksa sa negativnim predznakom.

Sabrati sve elemente liste koji su deljivi sa 17 i ispisati to na konzolu.

### Metod Contains

Metod Contains vraća vrednost true/false u zavisnosti od toga da li se element naveden kroz parametar nalazi u listi ili ne. Međutim, treba voditi računa da korišćenje ovog metoda može uticati na performanse aplikacije iz razloga što se vreme pretrage koristeći ovaj metod povećava linearno sa povećanjem kapaciteta liste. Preporuka je da se ovaj metod koristi isključivo za male kolekcije podataka.

## HashSet u C#

HashSet je C# struktura podataka koja, slično kao i liste, sadrži skup objekata istog tipa, ali je za razliku od liste optimizovana za efikasniju pretragu elemenata. Naime, objekti se u HashSet strukturi skladište tako da indeks odgovara hashcode-u objekta koji se dodaje. Pozitivne osobine ove strukture podataka je što onemogućava skladištenje istog objekta više puta, kao i brza detekcija postojanja elementa u HashSet-u. Iz tog razloga, da bi se optimizovale pretrage elemenata u listama velikog kapaciteta, preporuka je da se umesto liste koristi HashSet struktura podataka (ponekad je i više od $10^6$ puta brža – npr. 1h naspram 1ms).

Negativna strana čuvanja podataka u HashSet strukturi je otežan prolazak kroz strukturu, kao i pristupanje elementima HashSet-a. Naime, HashSet je neuređena kolekcija podataka (engl. *unordered*), odnosno prilikom skladištenja neće biti sačuvan redosled dodavanja elemenata, jer se elementi skladište na osnovu svog hashcode-a.

## Zadatak 2. Korišćenje HashSet kontejnera

Datu listu iz prethodnog zadatka prepakovati u HashSet strukturu podataka.

Prilikom prepakivanja podataka iz jedne strukture u drugu treba voditi računa da se to radi u okviru konstruktora strukture i uz oslanjanje na (ugrađene) mogućnosti .NET platforme. Nije dozvoljeno pisanje petlji i prepakivanje elemenata jedan po jedan.

## Zadatak 3. Uporedna analiza brzine pretrage

Verifikovati performanse prethodno kreiranih *HashSet* i *List* kolekcija prilikom pretrage. Izvršiti eksperiment u kojem se traži element -100001 u svakoj od kolekcija. Pretragu ponoviti 100.000 puta (npr. u for petlji) u slučaju obe kolekcije i uporediti dobijena vremena.

## Dictionary u C#

Dictionary je možda i najčešće korišćena kolekcija u C# koja predstavlja asocijativni kontejner. Asocijativni iz razloga što se prilikom skladištenja podacima dodeljuje ključ (*key*) koji služi za manipulisanje podacima u kolekciji. ***Dictionary<Tkey, TValue>*** važi za najbržu asocijativnu kolekciju jer u osnovi koristi *HashTable* strukturu. To znači da su vrednosti ključeva hash vrednosti, čime se znatno poboljšavaju performanse u radu sa ovom vrstom kolekcija. Razlika između *HashTable* i *Dictionary* kolekcije je u tome što je dictionary generički tip, što ovu kolekciju čini tipski bezbednom (eng. *type safe*) strukturom, odnosno nije moguće dodati slučajan tip elementa u kolekciju, a samim tip nije potrebno kastovanje podataka prilikom čitanja elemenata iz kolekcije. Vreme potrebno za dodavanje, uklanjanje i pretraga je približno konstantno bez obzira na veličinu kolekcije.

### Inicijalizacija Dictionary kolekcije

Slično kao i liste, dictionary je treba inicijalizovati sa unapred definisanim kapacitetom kad god je to moguće u cilju poboljšanja rukovanja memorijom.

```
//Umesto podrazumevane inicijalizacije dictionary:
Dictionary<long, object> dictObj = new Dictionary<long, object>();

// Inicijalizacija dictionary sa unapred definisanim kapacitetom:
Dictionary<long, object> dictObj = new Dictionary<long, object>(157354);
```

### Try-Get metod

Elementima dictionary-a se pristupa preko vrednosti ključa.

```
Dictionary<int, double> dictObj = new Dictionary<int, double >(100);
dictObj.Add(5, 135.6);
// Pristup elementu po ključu.
// Ukoliko element ne postoji, baca se izuzetak KeyNotFoundException.
long temp = dictObj[5];
```

Ukoliko nismo sigurni da li se ključ po kom pristupamo elementu dictionary-a zaista nalazi u kontejneru, potrebno je izvršiti odgovarajuću proveru pre pristupa. Ta provera se najčešće izvršava koristeći metod *ContainsKey*, kao što je navedeno:

```
double a;
if (dictObj.ContainsKey(5))
{
    a = dictObj[5];
}
```

Međutim, ukoliko postoji potreba za čestim pristupom elementima sa većom verovatnoćom da ključ postoji u dictionary-u, efikasnije je koristiti metod *TryGetValue*. Ovaj metod proverava da li navedeni ključ postoji u kolekciji, a zatim vraća vrednosti elementa koji je dodeljen tom ključu. U slučaju da ključ ne postoji, biće vraćena podrazumevana vrednost koja odgovara tipu elementa. Na osnovu gore iznetog, za pristup elementima preko ključa se predlaže korišćenje metoda *TryGetValue* zbog povećane pouzdanosti i efikasnosti.

```
if (dictObj.TryGetValue(1, out a))
{
    // kod za obradu
}
```

*TryGetValue* metod pokazuje bolje performanse kada je broj uspešnih pronalazaka elemenata po ključu veći (tj. kada se traže elementi koji stvarno postoje u kolekciji). Razlog je način implementacije *TryGetValue* metod koji podrazumeva poziv metoda *ContainsKey*, a zatim i pretragu niza u slučaju da ključ postoji. U listingu ispod su navedene implementacije oba metoda dobijene dekompajliranjem.

```
public bool TryGetValue(TKey key, out TValue value)
{
    int index = this.FindEntry(key);
    if (index >= 0)
    {
        value = this.entries[index].value;
        return true;
    }
    value = default(TValue);
    return false;
}

public bool ContainsKey(TKey key)
{
    return (this.FindEntry(key) >= 0);
}
```

### Add ili Update metod

Ukoliko je potrebno, u zavisnosti od toga da li ključ već postoji u kolekciji, odgovarajući par *<key, value>* dodati (*add*) ili ažurirati (*update*) najčešći pristup u implementaciji je provera da li određeni ključ postoji.

```
if (!dictObj.ContainsKey(5))
{
    dictObj.Add(5, 135.6);
}
else
{
    dictObj[5] = 135.6;
}
```

Efikasniji način da se ovo implementira je samo pristup kolekciji preko ključa, što u pozadini radi dodavanje ili ažuriranje u zavisnosti od toga da li ključ postoji u kolekciji.

```
dictObj[5] = 135.6;
```

### Dictionary čija je vrednost lista

Jedan od problema kada je u pitanju dictionary kolekcija čija je vrednost tipa liste je dodavanje elementa u listu, u zavisnosti od toga da li određeni ključ postoji u dictionary-u. Najčešći pristup je provera da li ključ postoji, i u zavisnosti od toga se nova lista prvo inicijalizuje:

```
Dictionary<long, List<object>> dictlist = new Dictionary<long,
List<object>>(10);
if (!dictlist.ContainsKey(13))
{
    dictlist.Add(13, new List<object>(1));
}
// očitavanje kolekcije sa ključem 13
List<object> objList = dictlist[13];
// dodavanje u listu sa ključem 13
dictlist[13].Add("Dodat element");
```

Efikasniji način kako da se ovo implementira je:

```
Dictionary<long, List<object>> dictlist = new Dictionary<long,
List<object>>(10);
List<object> objList = null;
if (!dictlist.TryGetValue(1, out objList))
{
    objList = new List<object>(1);
    dictlist.Add(1, objList);
}
objList.Add("Dodat element");
```

***KeyValuePair*** je struktura podataka koja predstavlja par (ključ, vrednost) u kolekciji, koju treba iskoristiti u ovom zadatku.

### Konvertovanje kontejnera

U nekim situacijama tokom razvoja softverskog proizvoda je potrebno izvršiti konvertovanje kolekcije podataka u drugi tip kolekcije. Ovakve operacije se relativno jednostavno implementiraju u C# programskom jeziku, u kome nije potrebno pisanje koda za prepakivanje elemenata iz jednog tipa kontejnera u drugi, već se koriste ugrađeni mehanizmi platforme.

Ako je npr. potrebno da kontejner tipa dictionary konvertujemo u listu, onda je dovoljno prilikom prolaska kroz dictionary kreirati listu čiji su elementi KeyValuePair structure (videti listing ispod).

```
Dictionary<long, long> dictlongs = new Dictionary<long, long>();
List<KeyValuePair<long, long>> kvp = dictlongs.ToList();
```

## Zadatak 4. Uporedna analiza brzine pristupa

Verifikovati performanse *TryGetValue* i *ContainsKey* metod prilikom pristupa elementu dictionary kolekcije preko ključa. Operaciju pristupa ponoviti 1000000 (tj. milion) puta, jednom u slučaju da ključ postoji u kolekciji, a zatim i za slučaju u kojem ključ ne postoji u kolekciji.

## Zadatak 5. Kontejneri u dictionary objektima

Proizvoljno inicijalizovati *Dictionary* čiji ključ je tipa ***int***, a vrednost tipa ***liste stringova***. Iterirati kroz celu kolekciju, i ispisati sve elemente svake liste u kolekciji u sledećem formatu: *<dictionary_key> - <index_of_list_item> - <string_value>*.

## Zadatak 6. Spajanje kolekcija

Inicijalizovati dve proizvoljne dictionary kolekcije istog tipa, i proizvoljno ih inicijalizovati. Kreirati novu dictionary kolekciju koja predstavlja spoj dve liste kreirane u prethodnom koraku.

# Vežba 4 – Napredni koncepti C# programskog jezika

Cilj ove vežbe je da proširi osnovno znanje programskog jezika C# sa konceptima koji će kasnije biti minimalno potrebni za implementaciju distribuiranih aplikacija.

## Eksplicitno kastovanje podataka

Eksplicitno kastovanje daje na znanje programskom prevodiocu u kom formatu želimo da tumačimo vrednost promenljive. Ukoliko ne kastujemo eksplicitno, programski prevodilac će implicitno da kastuje u trenutku kada to bude neophodno. Postoji dva tipa eksplicitnog kastovanja u C#:

1) kastovanje korišćenjem prefiksa,
2) *as* kastovanje.

```
Prefix cast:
BaseType g = ...;
SpecificType t = (SpecificType) g;

As cast:
BaseType g = ...;
SpecificType t = g as SpecificType;
```

Postoje značajne razlike između ove dve vrste kastovanja koje treba razmotriti pre opredeljenja za konkretan tip kastovanja. Razlike se odnose na ponašanje aplikacije u slučaju neuspešnog pokušaja kastovanja, kao i na performanse. U slučaju da nije moguće izvršiti kastovanje (npr. ukoliko *SpecificType* ne nasleđuje *BaseType*) u slučaju prefiks kastovanja biće bačen izuzetak. *As* kastovanje će vratiti null i nastaviti regularno sa izvršavanjem programa, a problem će se manifestovati tek prilikom korišćenja promenljive, npr. u okviru drugog metoda ili klase. Ovakve greške (eng.*bug*-ove) u kodu je mnogo teže otkriti nego što je to u slučaju da se odmah baci izuzetak i na taj način jasno ukaže šta je problem. S druge strane, *as* kastovanje je značajno brža operacija u odnosu na prefiks kastovanje (oko pet puta je brže). Na osnovu ovoga, prefiks kastovanje se još naziva i pouzdano kastovanje, a *as* kastovanje se naziva brzo kastovanje.

Problem vraćanja vrednosti null u slučaju korišćenja *as* kastovanja može se izbeći proverom da li je vrednost null, ali to svakako nosi i određene probleme, kao što je: 1) činjenica da se na taj način ne vidi da li je problem u kastovanju ili je i originalna promenljiva bila null, 2) prilikom bilo kakve dodatne provere gube se prednosti kada su u pitanju performanse *as* kastovanja.

## Zadatak 1.

Napisati program koji inicijalizuje niz od 30.000.000 elemenata tipa object na sledeći način:

- elementima sa indeksom i | (i%3 == 0) dodeliti vrednost null,
- elementima sa indeksom i | (i%3 == 1) dodeliti proizvoljnu *string* vrednost,
- elementima sa indeksom i | (i%3 == 2) dodeliti novu instancu tipa *object*.

Proći kroz sve elemente niza, i naći zbir dužina svakog stringa. Proveru da li je element niza string implementirati:

1) koristeći *prefix-cast* elemenata niza. Umesto hvatanja izuzetka u slučaju neuspešnog kastovanja, proveriti da li je element niza string, koristeći **is** operator;
2) koristeći *as-cast*.

Uporediti vreme izvršavanja programa pod gore navedenim tačkama 1 i 2.

## Rukovanje izuzecima u C#

Izuzeci su greške koje se dešavaju u toku izvršavanja programa. Ukoliko je moguće izuzetke bi uvek trebalo uhvatiti, uz odgovarajuće akcije kako bi se sprečilo defektivno ponašanje i obezbedio robustan program.

Implementiranjem izuzetaka omogućuje se transfer kontrole iz jednog dela programa u drugi. Izuzeci se bacaju pomoću ključne reči **throw** sa mesta u izvornom kodu gde se greška desila. Kada u nekom delu programa očekujemo izuzetak (jer znamo da taj deo programa može da dovede do defektivnog stanja), pravimo **try-catch** blokove. Ovi blokovi uvek idu u paru, ***try*** blok sadrži izvorni kod gde očekujemo izuzetak, dok se ***catch*** blok izvrši samo u slučaju kad se navedeni izuzetak desi. Postoji mogućnost korišćenja više *catch* blokova u slučaju da očekujemo više tipova izuzetka u određenom delu izvornog koda. Ukoliko je to situacija, blokove treba poređati tako da u njima očekivani izuzeci budu poređani od specijalizovanih ka opštijim tipovima izuzetaka. Opcioni blok prilikom rukovanja izuzecima je **finally** blok koji će se izvršiti bez obzira da li se izuzetak desio ili ne. U skladu sa tim, ovaj blok uglavnom sadrži naredbe koje se odnose na oslobađanje resursa zauzetih u try bloku, npr. ukoliko je fajl otvoren, potrebno je zatvoriti taj fajl bez obzira da li se izuzetak desio ili ne. Sintaksa try-catch-finally bloka je data ispod.

```
try
{
    // izvorni koji može da generiše izuzetak
}
catch (ExceptionName1 e1)
{
    // programski blok koji se izvršava na izuzetak ExceptionName1
}
// blokovi za obradu ostalih tipova izuzetaka
catch (ExceptionNameN eN)
{
    // programski blok koji se izvršava na izuzetak ExceptionNameN
}
finally
{
    // kod koji se uvek izvršava, npr. oslobađanje resursa
}
```

## Zadatak 2.

Izmeniti zadatak 1. tako da se u slučaju neuspešnog kastovanja obradi odgovarajući izuzetak. Ukoliko je element tipa string, vrednost stringa upisati u fajl. Takođe, nakon prolaska kroz sve elemente niza potrebno je osloboditi zauzete resurse.

*System.IO.File* je osnovna klasa za rukovanje fajlovima u .NET-u. Za rad sa tekstualnim fajlovima može se iskoristiti i *System.IO.TextWriter* klasa.

### Prosleđivanje izuzetaka

Ukoliko je potrebno izuzetak bačen u jednom metodu proslediti u drugi metod veoma je važno ispravno rukovati izuzetkom, jer u suprotnom *stack trace* originalnog izuzetka može biti uništen. Ovakav primer rukovanja izuzetkom nije ispravan.

```
try
{
    // izvorni koji može da generiše izuzetak
}
catch (Exception e)
{
    throw e; // briše stek trejsa originalnog izuzetka
}
```

Umesto toga, originalni izuzetak treba proslediti na sledeći način:

```
try
{
    // izvorni koji može da generiše izuzetak
}
catch (Exception e)
{
    // kod za ispis informacija o izuzetku u audit log, tj. detalji
greške se proslede log serveru ili zapišu u log fajl
    throw;
}
```

Takođe, moguće je "prepakovati" originalni izuzetak kako bi se prilagodio metodu kom se prosleđuje, ali ponovo treba voditi računa da se prilikom prepakivanja ne uništi stack trace. Primer ispod prikazuje kako se uhvaćeni Exception1 može prepakovati u drugi Exception2 bez gubljenja stack trace.

```
try
{
    // izvorni koji može da generiše izuzetak
}
catch (Exception1 e)
{
    // izuzetak se prepakuje uz očuvanje stek trejsa originalnog
izuzetka
    throw new Exception2("My custom exception message", e);
}
```

### Izbegavanje bacanja suvišnih izuzetaka

Generalno, bacanje izuzetaka je skupa operacija i kad god je moguće, efikasnije je koristiti različite validacije u kodu umesto bacanja izuzetka, npr.

- Iz prethodnog primera, korišćenje *is* operatora je efikasnije nego rukovanje izuzetkom u slučaju neuspešnog kastovanja.
- Na primeru programa koji deli dva broja, umesto korišćenja try-catch bloka kao zaštite od deljenja nulom, mnogo je efikasnije proveriti da li je delilac nula u dodatnom if-else bloku i u zavisnosti od rezultata nastaviti sa izvršavanjem koda.

## Using blok

**Using** blok je ekvivalentan **try-finally** bloku, pri čemu try blok ima isto telo kao using blok, dok se u finally delu poziva *Dispose()* metoda objekta koji se zada kao parameter using bloka. Iz tog razloga, da bi se objekat koristio kao parametar *using* bloka, mora se implementirati interfejs *IDisposable*. Sintaksa using bloka je navedena ispod. Novije verzije razvojnog okruženja Visual Studio (npr. 2017) sadrže funkcionalnost za automatsko generisanje implementacije tog interfejsa.

```
using (Resurs r = new Resurs())
{
    // kod za obradu resursa
}
```

## Zadatak 3.

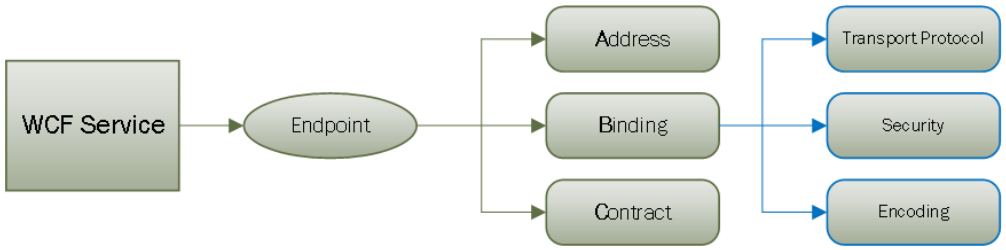
Koristeći using blok, implementirati kreiranje tekstualnog fajla na disku.

# Vežba 5 – Windows Communication Foundation (WCF) (1)

*Windows Communication Foundation (WCF)* je deo .NET razvojnog okruženja za razvoj distirbuiranih aplikacija. U opštem slučaju se distribuirani sistem implementiran uz pomoć WCF-a sastoji od sledećih elemenata:

1. Opis usluge servisa (tj. interfejs), koji se u kontekstu WCF-a naziva *Service Contract* i u okviru kog se deklarišu metodi i polja potrebni za pružanje opisane usluge;
2. Implementacija usluge, koji se najčešće naziva WCF servis i ima jednu ili više tačaka pristupa;
3. Korisnici usluga, tj. WCF klijenti, koji pristupaju WCF servisima i koriste njihove usluge.

Endpoint sadrži skup neophodnih informacija koje su potrebne WCF klijentu za pristup servisu. Svaki endpoint mora da sadrži informacije o adresi na kojoj servis sluša (***address***), o komunikacionom protokolu (***binding***), kao i informacije o opisu usluge (***contract***) koje servis pruža klijentima. Osnovni elementi endpointa su prikazani na Slika 4.




*Slika 4. Struktura WCF Endpoint -a*

**Adresa.** Predstavlja adresu na kojoj servis sluša dolazne zahteve. Predstavlja se u standardnom Uniform Resource Identifier (URI) formatu, na sledeći način:

*Scheme*:**//**<*MachineName*>[:*Port*]/*Path*, gde je:

- Scheme: transportni protokol (TCP, HTTP, itd.),
- Machine name: ime ili adresa računara nam kom je pokrenut (u WCF kontekstu se koristi još i izraz „hostovan“) servis,
- Port: ovaj parametar je opcioni, i ukoliko se ne navede koristi se podrazumevani broj porta za odgovarajući protokol, npr. 443 za HTTPS.
- Path: putanja do servisa.

**Binding.** Definiše „jezik“ komunikacije između klijenta i servisa, tj. komunikacioni protokol. Osim specificiranja mrežnog protokola, binding-om se mogu specificirati i druge karakteristike za komunikaciju kao što su bezbednost i kodiranje koje određuje format poruke – binarni ili tekstualni.

**Contract.** Definiše usluge koje određeni WCF servis izlaže. Deklaracija WCF interfejsa se dekoriše sa atributom *ServiceContract*. Svakom metodu koji će biti dostupan preko interfejsa se dodaje atribut *OperationContract*. Ukoliko je potrebno da i određeni složeni podatak (npr. WCF klasa)

bude dostupan preko WCF interfejsa, onda se isti obeležava se atributom *DataContract*, a njegovi članovi prostog tipa sa atributom *DataMember*.

## Zadatak 1. Implementacija WCF servisa

Implementirati WCF servis koji nudi metode za dodavanje i uklanjanje osoba opisanih uz pomoć jedinstvenog matičnog broja građana (JMBG) i punog imena, kao i ispis podataka o svakoj osobi.

**Smernice:** WCF servis se implementira u sledećim koracima:

1. Opis interfejsa;
2. Implementacija interfejsa;
3. Konfigurisanje servisa, koje se najčešće sastoji od opisa tačaka pristupa, tj. endpoint-a;
4. Pokretanje WCF servisa.

### Opis interfejsa

Prvi korak prilikom implementacije je deklarisanje interfejsa WCF servisa, tj. opisivanje usluga koje će budući WCF servis nuditi klijentima. Praksa lepog programiranja je da se kontrakti čuvaju u okviru posebne bibilioteke (*Visual C# Class Library, tj. DLL file*) koja će biti referencirana od strane svih WCF servisa i WCF klijenata koje implementiraju, odnosno koriste dotičnu uslugu.

Ako se ne koriste ugrađeni šabloni razvojnog okruženja za kreiranje WCF servisa i njegovih interfejsa, onda se ovo rešava u sledećim koracima:

1) **Kreiranje novog projekta** u razvojnom okruženju tipa Class Library, pod imenom Common
2) **Dodavanje reference** na .NET biblioteku klasa System.ServiceModel u kojoj se nalaze tipovi koji su potrebni za izradu WCF aplikacija. Ukoliko se koristi Visual Studio, onda se u Solution Explorer delu prozora, u delu stabla koji odgovara projektu kreiranom u prethodnom koraku uradi desni klik na References granu i potom se sa spiska dodaju reference na sve potrebne biblioteke.
3) **Deklaracija interfejsa** koji opisuje servis iz teksta zadatka, tj. omogućava unos i brisanje lica, odnosno kreiranje listinga svih lica – videti listing ispod. Obratiti pažnju na korišćenje atributa ServiceContract i OperationContract za označavanje interfejsa, odnosno metoda dostupnih preko WCF-a u okviru opisa interfejsa. Takođe ja važno dodavanje ključne reči *public* ispred interfejsa, jer u suprotnom interfejs neće biti dostupan iz drugih projekata, što jeste cilj ovog koraka. Napomena: pozvati se na System.ServiceModel prostor imena (ključna reč *using*) – bez toga gore navedeni atributi nisu „vidljivi“.

```
// Opis interfejsa
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        [OperationContract]
        void DodajLice(long jmbg, string ime, string prezime);

        [OperationContract]
        void ObrisiLice(long jmbg);

        [OperationContract]
        string SpisakLica();
    }
}
```

## Implementacija interfejsa

Sledeći korak je implementacija WCF servisa, koja treba da sadrži implementaciju svih potrebnih WCF interfejsa u jednoj ili više klasa. Za rešavanje zadatka 1 je dovoljno uraditi sledeće:

1) **Kreiranje novog projekta** tipa Console Application. Ovaj projekat se kreira u okviru istog Solution-a desnim klikom na koren stabla u Solution Explorer delu prozora i biranjem opcije Add → New project…, odnosno preko opcije glavnog menija File → New → Project… Projekat imenovati na odgovarajući način (npr. EvidencijaLica) i ubaciti ga u tekući Solution pored ranije kreirane biblioteke klasa. Uočiti da se automatski kreira klasa Progam sa metodom Main, tj. ulaznom tačkom WCF servisa.
2) **Dodavanje reference** na biblioteku .NET klasa System.ServiceModel na ranije opisan način.
3) **Dodati referencu** na biblioteku klasa iz prethodnog koraka. Ovo se postiže desnim klikom na References, Add Reference, izbor opcije Projects sa leve strane i potom označavanje biblioteke klasa ZajednickiElementi.
4) **Dodavanje klase koja implementira interfejs ili interfejse.** Klasa treba da nasledi ranije kreiran interfejs i implementira sve njegove metode. Novije verzije Visual Studio-a (npr. 2017) će ponuditi automatsku implementaciju metoda interfejsa – videti listing ispod.

```
namespace EvidencijaLica
{
    public class ServisFizickihLica : IFizickaLica
    {
        public void DodajLice(int jmbg, string ime, string prezime)
        {
            throw new NotImplementedException();
        }

        public void ObrisiLice(int jmbg)
        {
            throw new NotImplementedException();
        }

        public string SpisakLica()
        {
            throw new NotImplementedException();
        }
    }
}
```

## Konfiguracija i pokretanje WCF servisa

U cilju pojednostavljenja implementacije prvog WCF servisa će koraci 3 i 4 za izradu servisa, tj. njegovo konfigurisanje i pokretanje, biti objedinjeni.

Kasnije će ova dva koraka biti značajno drugačije implementirani, npr. konfiguracije tačke pristupa će biti izdvojena u posebnu XML datoteku. To je naravno jedan od razloga za upoznavanje sa radom sa XML datotekama u programskom jeziku C# u okviru ranijih vežbi.

Za konfigurisanje i pokretanje je potrebno uraditi sledeće:

1) U datoteci Program.cs dodati pozvati se na System.ServiceModel prostor imena (ključna reč *using*).
2) U klasi Program, u okviru metoda Main definisati primerak klase ServiceHost za rukovanje sa WCF servisom. Kao tip servisa navesti naziv klase iz prethodnog koraka (tj. ServisFizickihLica).
3) Dodati tačku pristupa.
4) Pokrenuti servis (metod *Open*).
5) Dodati kod za zaustavljanje servisa. Ovo je najjednostavnije implementirati tako što će servis biti dostupan sve dok se ne pritisne taster Enter na tastaturi.

Gore navedene korake je moguće implementirati C# kodom iz listinga ispod. Dodata tačka pristupa podrazumeva binarni protokol razmene podataka (NetTcpBinding) i pokretanje servisa na lokalnom računaru, na TCP portu 4000.

```
namespace EvidencijaLica
{
    class Program
    {
        static void Main(string[] args)
        {
            ServiceHost svc = new ServiceHost(typeof(ServisFizickihLica));
            svc.AddServiceEndpoint(typeof(ZajednickiElementi.IFizickaLica),
                        new NetTcpBinding(),
                        new Uri("net.tcp://localhost:4000/IFizickaLica"));
            svc.Open();

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje servisa.");
            Console.ReadLine();
        }
    }
}
```

## Zadatak 2. Implementacija WCF klijenta

Implementirati WCF klijenta koji se povezuje na servis iz prethodnog zadatka, dodaje i briše lica, odnosno ispisuje podatke.

Smernice za izradu WCF klijenta:

1) **Kreirati novi projekat** tipa Console Application.
2) **Dodavanje reference** na biblioteku .NET klasa System.ServiceModel na ranije opisan način.
3) **Dodati referencu** na biblioteku klasa sa opisom interfejsa WCF servisa. Ovo se postiže desnim klikom na References, Add Reference, izbor opcije Projects sa leve strane i potom označavanje biblioteke klasa ZajednickiElementi.
   **Napomena:** Koraci 2 i 3 su isti kod izrade WCF servisa i klijenta.
4) **Otvaranje komunikacionog kanala**, tj. povezivanje sa WCF servisom preko tačke pristupa. Razvojna platforma .NET nudi dve klase za implementaciju proxy-a ka servisu: ClientBase i ChannelFactory. U ovom zadatku biće prikazan način korišćenja ChannelFactory klase.
   **Napomena:** Obratiti pažnju da se u WCF klijentu nigde ne referencira klasa koja implementira WCF servis, već samo interfejs koji opisuje pružanu uslugu.
5) Dodati izvorni kod za **korišćenje usluga WCF servisa** u skladu sa tekstom zadatka, odnosno potrebama distribuirane aplikacije.

Jedna moguća implementacija WCF klijenta za rešavanje zadatka 2 je prikazana u listingu ispod. Nakon pokretanja se klijent povezuje na WCF servis, dodaje dva lica, ispisuje sve podatke u konzoli i na kraju briše jedno lice. Poslednja linija u listingu ispod obezbeđuje da se izvršavanje klijenta ne prekine, tj. da se proces ne zaustavi sve dok se ne pritisne taster Enter.

```
namespace TestKlijent
{
    class Program
    {
        static void Main(string[] args)
        {
            ChannelFactory<IFizickaLica> factory = new ChannelFactory<IFizickaLica>(
                new NetTcpBinding(),
                new EndpointAddress("net.tcp://localhost:4000/IFizickaLica"));

            IFizickaLica kanal = factory.CreateChannel();
            kanal.DodajLice(0803996800033, "Marko", "Markovic");
            kanal.DodajLice(1503996800033, "Petar", "Petrovic");
            string spisak = kanal.SpisakLica();
            Console.WriteLine("Spisak lica: {0}", spisak);
            kanal.ObrisiLice(1503996800033);

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje klijenta.");
            Console.ReadLine();
        }
    }
}
```

## Najčešće greške u izradi WCF aplikacija

Najčešće greške u izradi WCF servisa i klijenata:

- Pogrešno navođenje adrese u konstruktoru klase EndpointAddress – videti listing iznad. Uglavnom se greši u navođenju porta ili se navede pogrešan naziv interfejsa u poslednjem delu URI-ja.
  **Napomena:** URI koji se navodi u konfiguraciji servisa treba da se u potpunosti poklapa sa URI-jem koji se navodi u kodu ili konfiguraciji klijenta sve dok se servis ne izmesti na drugi računar, u kom slučaju se umesto „localhost“ navodi adresa udaljenog računara.
- Pogrešno referenciranje neophodnih biblioteka, npr. ServiceModel ili biblioteka u kojoj je deklarisan interfejs. Podvrsta ove greške je izostavljena using deklaracija za ServiceModel ili sopstvenu biblioteku klasa.
- Greške navođenju prostora imena, prvenstveno u konfiguracionim datotekama – naročito kada se interfejs, servis i klijent nalaze u različitim prostorima imena.

# Vežba 6 – Windows Communication Foundation (WCF) (2)

## Rad se složenim podatacima

WCF koristi *DataContract* serijalizaciju prilikom prenosa podataka. Svi ugrađeni .NET primitivni tipovi, kao i neki ugrađeni izvedeni tipovi mogu biti serijalizovani i deserijalizovani pomoću *DataContract* serijalizatora.

Korisnički-definisane tipove podataka je potrebno označiti (dekorisati) sledećim atributima (nalaze se u *System.Runtime.Serialization* prostoru imena iz):

1. *DataContract*: eksplicitno definiše novi tip podatka za serijalizaciju, koristi se za dekorisanje klasa, struktura i enumeracija;
2. *DataMember*: definiše članove klase koji će biti uključeni u proces serijalizacije;
3. *IgnoreDataMember*: navodi se kako bi se preskočila serializacija određenog člana klase.

## Zadatak 1.

Izmeniti metode servisa kreiranog u prethodnoj vežbi tako da se umesto parametara JMBG, ime i prezime koristi objekat klase FizickoLice sa odgovarajućim poljima. Prilagoditi WCF servis i klijenta ovim izmenama.

Smernice za rešavanje zadatak:

1. U deljenoj biblioteci klasa uvesti novu klasu FizickoLice. Klasu dekorisati gore navedenim atributima (videti listing ispod) i na taj način omogućiti serijalizaciju njenih primeraka i prenos preko komunikacionih kanal otvorenih preko WCF-a.
   Napomena: referencirati na biblioteku *System.Runtime.Serialization*.

```
// Klasa i clanice klase dekorisane WCF atributima DataContract i DataMember
namespace ZajednickiElementi
{
    [DataContract]
    public class FizickoLice
    {
        public FizickoLice(long jmbg, string ime, string prezime)
        {
            this.Jmbg = jmbg;
            this.Ime = ime;
            this.Prezime = prezime;
        }
        [DataMember]
        public long Jmbg { get => jmbg; set => jmbg = value; }
        [DataMember]
        public string Ime { get => ime; set => ime = value; }
        [DataMember]
        public string Prezime { get => prezime; set => prezime = value; }

        private long jmbg;
        private string ime;
        private string prezime;
    }
}
```

2. Izmeniti opis interfejsa u skladu sa tekstom zadatka, tj. metod za unos fizičkog lica treba da umesto tri prosta parametra dobije jedan parametar tipa FizickoLice. Obratiti pažnju

da nije neophodno proslediti primerak klase FizickoLice u metodi za brisanje lica, jer je JMBG jedinstveni identifikator čije navođenje je dovoljno za operaciju brisanja.

```
// Klasa i clanice klase dekorisane WCF atributima DataContract i DataMember
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        [OperationContract]
        void DodajLice(FizickoLice lice);

        [OperationContract]
        void ObrisiLice(long jmbg);

        [OperationContract]
        List< FizickoLice> SpisakLica();
    }
}
```

3. Uskladiti minimalnu implementaciju servisa u klasi koja nasleđuje i implementira gore navedeni interfejs sa izmenom interfejsa.
4. Uskladiti kod WCF klijenta sa ovim izmenama, tj. kreirati primerke klase FizickoLice i njih proslediti kroz pozive udaljenih metoda.

## Rad sa WCF izuzecima

WCF podržava propagaciju grešaka od servisa do klijenata preko izuzetaka, tj. omogućava da se izuzetak kreira na strani servisa, propagira kroz komunikacioni kanal do klijenta, gde ga klijent može (i treba) uhvatiti sa odgovarajućim kodom za obradu grešaka.

Rad sa izuzecima se omogućava u sledeća tri koraka:

1. Opisu metoda se doda atribut u kojem se navodi tip izuzetka koji metod može da generiše. Moguće je korišćenje ugrađenih ili korisnički-definisanih tipova izuzetaka. Za ovo drugo je potrebno definisati klasu koja će sadržati opis greške.
2. Servis nakon ulaska u granu izvršavanja sa greškom generiše („baci“) WCF izuzetak.
3. Klijent hvata WCF izuzetak.

Navođenje izuzetka koji WCF metod može da baci je moguće navođenjem atributa *FaultContract* na način prikazan u isečku interfejsa u sledećem listingu.

```
[OperationContract]
[FaultContract(typeof(ServisFizickaLicaIzuzetak))]
void DodajLice(FizickoLice lice);
...
```

Prilikom bacanja WCF izuzetka, inicijalizuje se primerak klase sa opisom greške kao što je navedeno u listingu ispod.

```
ServisFizickaLicaIzuzetak iz = new ServisFizickaLicaIzuzetak()
{
        Razlog = "Fizičko lice sa navednim JMBG-om postoji."
};
throw new FaultException<ServisFizickaLicaIzuzetak>(iz);
```

Izuzetak se „hvata“ sa druge strane komunikacionog kanala na način prikazan u sledećem listingu.

```
try
{
        kanal.DodajLice(new FizickoLice(0803996800033, "Marko", "Markovic"));
} catch (FaultException<ServisFizickaLicaIzuzetak> iz)
{
        Console.WriteLine(iz.Detail.Razlog);
}
```

**Napomena:** try blok treba okruži samo one delove koda za koje je poznato da mogu da uzrokuje greške određenog tipa. try blok može biti praćen većim brojem catch blokova, u kojima se navode tipovi mogućih izuzetaka u opadajućem redosledu specifičnosti, npr. *FaulException* (koji je specifičniji, tj. izvedeni tip) prvi, a opšti *Exception* drugi. Takođe je poželjno da se svaki od većeg broja sukcesivnih poziva za dodavanje lica stavi u poseban try-catch blok da bi se tačno znalo u kom pozivu dolazi do greške.

## Zadatak 2. Bacanje izuzetaka

Proširiti implementaciju primera iz prethodnog zadatka sa radom sa izuzecima. Na strani WCF servisa rukovati sa greškama tipa dodavanje ranije dodatog lica i brisanje nepostojećih lica. Sve greške na ovom interfejsu opisati preko jedne zajedničke klase, tj. koristiti primerke iste klase za opis oba gore navedena tipa greške. Sa strane WCF klijenta dodati kod za obradu svih mogućih tipova grešaka deklarisanih preko atributa u opisu interfejsa, tj. staviti svaki poziv metoda na WCF servisu u try-catch blok ako on može da „baci“ izuzetak.

Smernice za rešavanje zadatak:

1. U deljenoj biblioteci klasa dodati novu klasu koja će se koristiti za opisivanje grešaka do kojih može doći tokom obrade podataka u WCF servisu.
   **Napomena:** klasu i značajna polja označiti sa odgovarajućim atributima za serijalizaciju (*DataContract* i *DataMember*). Omogućiti korišćenje tih atributa uvođenjem *System.Runtime.Serialization* prostora imena.

```
namespace ZajednickiElementi
{
    [DataContract]
    public class ServisFizickaLicaIzuzetak
    {
        [DataMember]
        public string Razlog { get => razlog; set => razlog = value; }
        private string razlog;
    }
}
```

2. Modifikovati implementaciju interfejsa dodavanjem atributa FaultContract u opisima metoda koji mogu baciti neki tip greške. Uočiti masnim slovima izdvojene linije koda sa atributima FaultContract u listingu ispod. To su jedine dve dodate linije po kojima se ovaj listing razlikuje od osnovne verzije opisa interfejsa.

```
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void DodajLice(FizickoLice lice);

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void ObrisiLice(long jmbg);

        [OperationContract]
        List< FizickoLice > SpisakLica();
    }
}
```

3. U implementaciju metoda za dodavanje lica dodati IF blok u kojem se proverava JMBG fizičkog lica i ukoliko je on jednak sopstvenom JMBG-u, onda „baciti“ izuzetak sa porukom „Fizičko lice sa navedenim JMBG-om postoji.“
   **Napomena:** WCF servis u ovim primerima je *stateless* tipa, tj. ne pamti dodata fizička lica. U tom kontekstu je korišćenje gore navedenog uslova jednostavan način izbegavanja bezuslovnog bacanja izuzetaka.
4. U implementaciju metoda za uklanjanje lica dodati IF blok u kojem se proverava JMBG za brisanje i ukoliko je različit od sopstvenog, onda „baciti“ izuzetak sa porukom „Fizičko lice sa navedenim JMBG-om **ne** postoji.“
5. Na strani klijenta dodati try-catch blokove za obradu svih mogućih izuzetaka za koje znamo da WCF servis može da generiše – videti primer u listingu ispod.

```
namespace TestKlijent
{
    class Program
    {
        static void Main(string[] args)
        {
            ChannelFactory<IFizickaLica> factory = new ChannelFactory<IFizickaLica>(
                new NetTcpBinding(),
                new EndpointAddress("net.tcp://localhost:4000/IFizickaLica"));

            IFizickaLica kanal = factory.CreateChannel();
            try
            {
                kanal.DodajLice(new FizickoLice(2803996800033, "Marko", "Markovic"));
            } catch (FaultException<ServisFizickaLicaIzuzetak> ex)
            {
                Console.WriteLine(ex.Detail.Razlog);
            }

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje klijenta.");
            Console.ReadLine();
        }
    }
}
```

## Konfiguracija WCF aplikacija

Platforma .NET podržava konfigurisanje izvršnih aplikacija pomoću konfiguracione datoteke koja sadrži konfiguraciju definisanu na XML jeziku. Prevođenjem projekta se pravi kopija ove datoteke sa promenjenim imenom ***ime_aplikacije.exe.config***. Konfiguraciju .NET aplikacija je moguće menjati u tim datotekama.

Ovakve datoteke se dodaju u projekte koje su izvršnog tipa (konzolne i Windows aplikacije) na sledeći način: Desni klik na ciljani projekat → Add New Item → odabrati opciju ”Application Configuration File”. Za rad sa konfiguracionim datotekama potrebno je uključiti **System.Configuration** namespace.

## Zadatak 3. Korišćenje konfiguracionih fajlova

WCF servis i klijenta iz prethodnog zadatka izmeniti tako da se koriste konfiguracioni fajlovi za definisanje tačaka pristupa preko kojih se ostvaruje njihova komunikacija.

### Konfiguracija WCF servisa

U dosadašnjim primerima je tačka pristupa (eng. *endpoint*) WCF servisa bila konfigurisana u izvornom kodu. Mana takvog rešenja je nemogućnost izmene konfiguracije tačke pristupa (npr. izmena TCP porta na kojem će se pokrenuti) bez prevođenja cele aplikacije.

Gore navedeni problem je moguće rešiti preko konfiguracione datoteke WCF aplikacija. Konfiguracija se opisuje XML-om i njegova izmena je moguća bez prevođenja izvornog koda.

Izvlačenje konfiguracije tačke pristupa WCF servisa je moguće uraditi u sledećim koracima:

1. Dodati konfiguracionu datoteku u projekat WCF servisa.
   **Napomena:** novije verzije razvojnog okruženja Visual Studio (npr. 2017 Enterprise) podrazumevano dodaju konfiguracionu datoteku u projekte tipa konzolna aplikacija (ime datoteke App.config).
2. Definisati opis tačke (ili tačaka) pristupa po ugledu na XML opis iz listinga ispod. Ključni elementi su XML tagovi service, add i endpoint, preko kojih se unosi ime klase, URI i ime interfejsa, respektivno. Ovi elementi su označeni masnim fontom u listingu ispod.
   **Napomena #1:** navode se puna imena klase i interfejsa sa celim putanjama u prostoru imena.
   **Napomena #2:** ogromna većina primera WCF servisa i klijenata u ovom praktikumu koristi binarni komunikacioni protokol, tj. NetTcpBinding. Neke od mogućih alternativa su BasicHttpBinding i NetMsmqBinding, koji će biti ilustrovan kroz primere kasnije.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <system.serviceModel>
    <services>
      <service name="EvidencijaLica.ServisFizickihLica">
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:4000/IFizickaLica" />
          </baseAddresses>
        </host>
        <!-- Service Endpoints -->
        <endpoint address="" binding="netTcpBinding" contract="ZajednickiElementi.IFizickaLica" />
        </service>
      </services>
  </system.serviceModel>
</configuration>
```

3. Obrisati C# kod za programsko dodavanje tačke pristupa iz datoteke Program.cs. Ovo praktično znači da je potrebno obrisati poziv metoda AddServiceEndpoint.

## Konfiguracija WCF klijenta

Slično WCF servisu iz ranijih primera, dosad izrađeni WCF klijenti takođe imaju „zakodiranu“, tj. u samom C# kodu definisanu, tačku pristupa WCF servisa sa kojim komuniciraju. Ovu manjkavost klijenata je moguće ispraviti na sličan način, preko XML konfiguracione datoteke.

Dodavanje XML konfiguracione datoteke i prebacivanje opisa tačke pristupa WCF servisa je moguće implementirati u sledećim koracima:

1. Dodati konfiguracionu datoteku u projekat WCF klijenta na ranije opisan način.
   **Napomena:** novije verzije razvojnog okruženja Visual Studio automatski dodaju konfiguracionu datoteku u projekte tipa konzolna aplikacija.
2. Dodati opis tačke pristupa WCF servisa u konfiguracionu datoteku – videti listing ispod. Ključni elementi su označeni masnim fontom. Prvo je ime interfejsa koje se posle navodi u C# kodu klijenta. Drugo je URI, koji mora da se poklapa sa URI-jem servisa. Treće je ime interfejsa.
   **Napomena:** ime klase koja implementira WCF servis se ne navodi nigde u kodu, odnosno konfiguraciji WCF klijenta. Ovo omogućava lakšu izmenu implementacije servisa.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <system.serviceModel>
    <client>
      <endpoint name="ServisLica"
        address="net.tcp://localhost:4000/IFizickaLica"
        binding="netTcpBinding"
        contract="ZajednickiElementi.IFizickaLica" />
    </client>
  </system.serviceModel>
</configuration>
```

3. C# kod za programsko opisavanje tačke pristupa zameniti sa imenom interfejsa po ugledu na listing ispod. Umesto eksplicitnog navođenja protokola i adrese se navodi samo ime stavke u konfiguracionoj datoteci koja sadrži opis tačke pristupa.
   **Napomena:** ime interfejsa mora da se poklopi sa imenom navedenim u atributu *name* u okviru XML taga *endpoint* u okviru konfiguracione datoteke – videti listing iznad.

```
ChannelFactory<IFizickaLica> factory = new ChannelFactory<IFizickaLica>("ServisLica");
        //new NetTcpBinding(),
        //new EndpointAddress("net.tcp://localhost:4000/IFizickaLica"));
IFizickaLica kanal = factory.CreateChannel();
```

## Zadatak 4. Kalkulator kompleksnih brojeva

Implementirati kalkulator kompleksnih brojeva koji podržava četiri osnovne operacije nad kompleksnim brojevima. WCF servis nudi usluge izvršavanja operacija kalkulatora, dok klijenti unose kompleksne brojeve i željene operacije i po prijemu rezultata štampaju iste. U slučaju da klijent pokuša da deli s nulom server treba da baci odgovarajući izuzetak koji klijent treba da uhvati i ispiše njegove detalje.

## Najčešće greške u konfigurisanju WCF aplikacija

Najčešće greške u konfigurisanju WCF servisa i klijenata:

- greška u navođenju IP adrese, odnosno imena računara na kom se nalazhi WCF servis – ovo je posebno važno u konfiguraciji WCF klijenta;
- greške u navođenju imena klase koja implementira WCF interfejs – obratiti pažnju da je potrebno navesti celo ime klase zajedno sa punim prostorom imena;
- greške u URI-ju, npr. pogrešno navedeno ime interfejsa na kraju simboličke adrese;
- greške u navođenju imena interfejsa – obratiti pažnju da je potrebno navesti celo ime interfejsa zajedno sa punim prostorom imena;
- razlike u URI-jima u konfiguraciji klijenta i servisa – mogu oba URI-ja da pojedinačno budu ispravna, ali da njihove razlike onemoguće povezivanje klijenta na servis;
- greška u navođenju imena interfejsa u kodu klijenta – imena interfejsa u kodu i u XML konfigraciji moraju da se poklapaju.

# Vežba 7 – Mehanizmi bezbednosti

Cilj ovih vežbi je da se uvedu bezbednosni mehanizmi za autentifikaciju i autorizaciju klijenata na WCF servisima.

## Autentifikacija

Autentifikacija se sastoji od jednoznačnog utvrđivanja identiteta WCF klijenata. U ovoj vežbi ćemo vršiti autentifikaciju unošenjem korisničkog imena i lozinke (tj. šifre). To praktično znači da ćemo implementirati jedno-faktorsku autentifikaciju digitalnog identiteta (korisničko ime WCF klijenta) preko nečega što korisnik zna (tj. šifre).

Iako .NET i WCF sadrže bogat skup ugrađenih bezbednosnih mehanizama, ovde će u edukativne svrhe biti prikazana jednostavna implementacija autentifikacije koja se ne oslanja na njih.

## Zadatak 1.

Proširiti WCF server iz prethodnih vežbi sa autentifikacijom WCF klijenata. Onemogućiti pristup interfejsu korisnicima koji nisu jednoznačno autentifikovani.

### Direktorijum korisnika

WCF servis implementiran u prethodnim poglavljima je bio *stateless* tipa, tj. nije pamtio nikakve informacije o svojim klijentima, niti je snimao podatke koje je dobijao od njih. Praktično su svi pozivi inicirani od strane WCF klijenata bili međusobno nezavisni i nikako nisu uticali na stanje WCF servisa. Za implementaciju mehanizma autentifikacije će biti potrebno uvesti laganu bazu (tj. direktorijum) korisnika kojima će biti dozvoljen pristup WCF servisu.

Za kreiranje direktorijuma će biti potrebno uraditi sledeće:

1) Uvođenje C# klase koja opisuje korisnika;
2) Uvođenje skladišta u kom se čuvaju podaci korisnika;
3) Unos podataka korisnika kojima je dozvoljen pristup unošenjem uređenih parova (korisničko ime, lozinka).

Opis korisnika WCF servisa je dat u sledećem listingu:

```
namespace EvidencijaLica
{
    class Korisnik
    {
        string korisnik;
        string lozinka;
        bool autentifikovan = false;
        string token;

        public string KIme { get => korisnik; set => korisnik = value; }
        public string Lozinka { get => lozinka; set => lozinka = value; }
        public bool Autentifikovan { get => autentifikovan;
                        set =>    autentifikovan = value; }
        public string Token { get => token; set => token = value; }

        public Korisnik(string ime, string lozinka)
        {
            this.KIme = ime;
            this.Lozinka = lozinka;
        }
    }
}
```

Pošto se podaci autentifikovanih korisnika čuvaju samo u okviru WCF servisa, ovu klasu je potrebno uvesti samo u projektu WCF servisa, tj. ne dodaje se u deljeni projekat tipa biblioteka klasa. Klasa enkapsulira polja za čuvanje korisničkog imena, lozinke, statusa autentifikacije korisnika i token asociran sa komunikacionom sesijom klijenta. Mogući statusi su da je autentifikovan, odnosno da nije.

Opisi korisnici će biti čuvani u strukturi podataka tipa Dictionary, koji omogućava brzu pretragu po ključu. U produkcionim (distribuiranim) sistemima bi se skladištili u odgovarajućoj bazi podataka, npr. Active Directory (AD). Sledeći listing prikazuje kod za kreiranje direktorijuma i dodavanje dva korisnika, čija su korisnička imena i lozinke takođe date u izvornom kodu jednostavnosti radi – ovim se ceo primer pojednostavljuje, npr. nije potrebno postojanje AD-a i izvornog koda za pristup istom.

```
namespace EvidencijaLica
{
    class DirektorijumKorisnika
    {
        private Dictionary<string, Korisnik> korisnici
            = new Dictionary<string, Korisnik>();
        private Dictionary<string, string> autentifikovani
            = new Dictionary<string, string>();
        private const string _pepper = "P&0myWHq";

        public DirektorijumKorisnika()
        {
            DodajKorisnika("pera", "P3rA");
            DodajKorisnika("admin", "pr3Ax4dmin");
        }

        private string KodiranTekst(string lozinka)
        {
            using (var sha = SHA256.Create())
            {
                var computedHash = sha.ComputeHash(
                    Encoding.Unicode.GetBytes(lozinka + _pepper));
                return Convert.ToBase64String(computedHash);
            }
        }

        public void DodajKorisnika(string ime, string lozinka)
        {
            korisnici.Add(ime, new Korisnik(ime, this.KodiranTekst(lozinka)));
        }

        public string AutentifikacijaKorisnika(string ime, string lozinka)
        {
            if (korisnici.ContainsKey(ime)
                && this.KodiranTekst(lozinka) == korisnici[ime].Lozinka)
            {
                korisnici[ime].Autentifikovan = true;
                string token = this.KodiranTekst(ime + _pepper);
                this.autentifikovani.Add(token, ime);
                return token;
            }
            else
            {
                throw new FaultException<BezbednosniIzuzetak>(
                    new BezbednosniIzuzetak("Neispravno korisnicko ime i/ili
lozinka."));
            }
        }

        public bool KorisnikAutentifikovan(string token)
        {
            if (autentifikovani.ContainsKey(token))
                return true;
            else
                throw new FaultException<ZajednickiElementi.BezbednosniIzuzetak>(
                    new ZajednickiElementi.BezbednosniIzuzetak("Korisnik nije
autentifikovan"));
        }
    }
}
```

Metod AutentifikacijaKorisnika iz gornjeg listinga baca WCF izuzetak ako je pozvana sa nepostojećim korisničkim imenom, odnosno ako se uz postojeće korisničko ime prosledi neispravna lozinka. U oba slučaja se baca WCF izuzetak i detalju se nalazi primerak klase BezbednosniIzuzetak.

Obratiti pažnju na metod KodiranTekst, koji kodira lozinke sa jednosmernom funkciojom SHA256. U cilju povećanja složenosti primenjenog postupka kodiranja lozinki se koristi „biber“ (engl.

*pepper*) koji se konkatenacijom tekstualnih promenljivih dodaje na lozinku i povećava složenost ulaznog teksta. Tokeni komunikacionih kanala asociranih sa uspešno autentifikovanim korisnicima se čuvaju u strukturi pod nazivom „autentifikovani“, u kom su tokeni ključevi, a korisnička imena vrednosti.

Primerak direktorijuma korisnika dodat u klasu koja implementira WCF servis kao statički član. Iako je u opštem slučaju poželjno izbegavati korišćenje statičkih promenljivih, ovde se korišćenjem tog tipa promenljive moguće osigurati da će u jednom primerku WCF servisa postojati samo jedan primerak direktorijuma korisnika. Videti masnim slovima izdvojen red u listingu ispod.

```
namespace EvidencijaLica
{
    public class ServisFizickihLica : IFizickaLica, IBezbednosniMehanizmi
    {
        static readonly DirektorijumKorisnika direktorijum = new DirektorijumKorisnika();
         …
```

## Autentifikacija WCF korisnika

Za proveru podataka dobijenih od WCF korisnika je potrebno uraditi sledeće:

1) Dodati novi WCF interfejs koji opisuje dostupne mehanizme bezbednosti, npr. autentifikacija WCF korisnika. Metod za autentifikaciju korisnika treba da vrati promenljivu tipa tekst u slučaju uspešne autentifikacije. Taj tekst treba da bude jedinstven tekst kreiran od strane WCF servisa i asociran sa komunikacionom sesijom klijenta.

```
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IBezbednosniMehanizmi
    {
        [OperationContract]
        [FaultContract(typeof(BezbednosniIzuzetak))]
        string Autentifikacija(string korisnik, string lozinka);
    }
}
```

2) Naslediti novo-kreirani bezbednosni interfejs u klasi u WCF servisu koja implementira interfejs za radom sa opisima fizičkih lica.

   Napomena: u programskom jeziku C# jedna klasa može implementirati veći broj interfejsa.

3) Implementirati metod za autentifikaciju WCF korisnika. U ovom metodu se primljeni uređeni par (korisničko ime, lozinka) poredi sa parovima u direktorijumu korisnika. U listingu ispod uočiti da je autentifikacija korisnika delegirana direktorijumu korisnika i da se statičkom članu klase pristupa preko imena klase (tj. ne preko primerka klase).

4) Napomena: WCF klijent mora da vodi računa o tome da ovaj metod može da baci WCF izuzetak.

```
public string Autentifikacija(string korisnik, string lozinka)
{
        return ServisFizickihLica.direktorijum.AutentifikacijaKorisnika(
                korisnik, lozinka);
}
```

5) Metod za autentifikaciju korisnika je jednostavan i njegov poziv može imati jedan od sledeća dva rezultata:

   a. Uspešna autentifikacija korisnika – u ovom slučaju metod vraća tajni tekstualni token asociran sa komunikacionom sesijom korisnika. Ovaj token je poznat samo WCF servisu i klijentu.

   b. Neuspešna autentifikacija korisnika (npr. neispravna lozinka) i metod baca WCF izuzetak.

6) Pored gore navedenih izmena je potrebno proširiti opise ranije deklarisanih WCF interfejsa prosleđivanjem dobijenog bezbednosnog tokena – videti listing ispod.
   Napomena: u produkcionim sistemima bi identifikacija autentifikovanih komunikacionih sesija bilo implementirano na drugi način, korišćenjem rešenja ugrađenih u .NET okvir.

```
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void DodajLice(FizickoLice lice, string token);

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void ObrisiLice(long jmbg, string token);

        [OperationContract]
        string SpisakLica(string token);
    }
}
```

7) Implementacije gore navedenih metoda takođe treba prilagoditi ovim izmenama – videti listing ispod u kom je masnim fontom prikazan poziv metoda za proveru statusa korisnika.

```
public void DodajLice(FizickoLice lice, string token)
{
        ServisFizickihLica.direktorijum.KorisnikAutentifikovan(token);
        …
```

8) Modifikovati konfiguraciju WCF servisa navođenjem pristupne tačke preko koje su dostupni metodi na bezbednosnom interfejsu – videti masnim fontom označenu dodatnu tačku pristupa u listingu ispod.

```
  <system.serviceModel>
    <services>
      <service name="EvidencijaLica.ServisFizickihLica">
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:4000/ServisFizickihLica" />
          </baseAddresses>
        </host>
        <!-- Service Endpoints -->
        <endpoint address="" binding="netTcpBinding"
contract="ZajednickiElementi.IFizickaLica" />
        <endpoint address="" binding="netTcpBinding"
contract="ZajednickiElementi.IBezbednosniMehanizmi" />
      </service>
    </services>
```

## Zadatak 2.

Proširiti WCF klijenta iz prethodnih vežbi tako da učita korisničko ime i lozinku iz komandne linije, autentifikuje se na WCF servisu i koristi dobijeni token komunikacione sesije u svim kasnijim pozivima.

Smernice za rešavanje zadatka:

1) Navesti pristupnu tačku bezbednosnog servisa u konfiguracionom fajlu WCF klijenta – videti masnim fontom označen deo listinga ispod.
   Napomena: obratiti pažnju na to da je adresa ista za obe tačke pristupa i da se one razlikuju samo po imenu interfejsa koji se navodi u XML atributu „*contract*“.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5" />
    </startup>
  <system.serviceModel>
    <client>
      <endpoint name="ServisLica"
        address="net.tcp://localhost:4000/ServisFizickihLica"
        binding="netTcpBinding"
        contract="ZajednickiElementi.IFizickaLica" />
      <endpoint name="BezbednosniServis"
        address="net.tcp://localhost:4000/ServisFizickihLica"
        binding="netTcpBinding"
        contract="ZajednickiElementi.IBezbednosniMehanizmi" />
    </client>
  </system.serviceModel>
</configuration>
```

2) Otvoriti dodatni komunikacioni kanal ka bezbednosnom servisu i autentifikovati korisnika.
3) Uneti korisničko ime i lozinku preko konzolnog prozora. Memorisati dobijeni bezbednosni token u lokalnoj promenljivoj.
4) Proširiti pozive postojećih metoda dodavanjem tokena u sve pozive.
5) Obraditi bezbednosne izuzetke.

Rešenja za korake 2-5 su prikazana u listingu ispod. Obratiti pažnju na sledeće elemente listinga:

- Otvaraju se dva nezavisna komunikaciona kanala: kroz prvi se pozivaju bezbednosni metodi, a kroz drugi se pozivaju metodi za rukovanje sa opisima fizičkih lica.

- U izvornom kodu se ne navode ni korisnička imena, ni lozinke, već se oni učitavaju iz konzole. Na ovaj način se povećava bezbednost tih osetljivih podataka.
- Metodima za rukovanje opisima fizičkih lica se pored od ranije poznatih argumenata prosleđuje i token koji je dobijen samo ako je korisnik uspešno autentifikovan.
- U donjem delu listinga se „hvataju“ dva tipa WCF izuzetaka: oni koji se generišu ukoliko se proslede nevalidni bezbednosni podaci, odnosno oni koji se generiše ukoliko se navode dupli podaci ili se pokuša očitati nepostojeći podataka.

```
namespace TestKlijent
{
    class Program
    {
        static void Main(string[] args)
        {
            ChannelFactory<IBezbednosniMehanizmi> bezbednost =
                new ChannelFactory<IBezbednosniMehanizmi>("BezbednosniServis");
            IBezbednosniMehanizmi kanalB = bezbednost.CreateChannel();
            string token = "";
            try
            {
                Console.WriteLine("Unesite korisnicko ime:");
                string korisnik = Console.ReadLine();
                Console.WriteLine("Unesite lozinku:");
                string lozinka = Console.ReadLine();
                token = kanalB.Autentifikacija(korisnik, lozinka);
            } catch (FaultException<BezbednosniIzuzetak> ex)
            {
                Console.WriteLine(ex.Detail.Razlog);
            }

            ChannelFactory<IFizickaLica> servislica =
                new ChannelFactory<IFizickaLica>("ServisLica");
            IFizickaLica kanal = servislica.CreateChannel();
            try
            {
                kanal.DodajLice(new FizickoLice(2803996800033, "Marko", "Markovic"),
token);
                kanal.DodajLice(new FizickoLice(1604988800033, "Petar", "Petrovic"),
token);

                string spisak = kanal.SpisakLica(token);
                Console.WriteLine("Spisak lica: {0}", spisak);

                kanal.ObrisiLice(1604988800033, token);
            }
            catch (FaultException<BezbednosniIzuzetak> bex)
            {
                Console.WriteLine(bex.Detail.Razlog);
            }
            catch (FaultException<ServisFizickaLicaIzuzetak> ex)
            {
                Console.WriteLine(ex.Detail.Razlog);
            }

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje klijenta.");
            Console.ReadLine();
        }
    }
}
```

## Autorizacija

Dodela prava pristupa i kontrola pristupa nam dodatno povećava nivo informacione bezbednosti u distribuiranim sistemima preko selektivnog omogućavanja, odnosno onemogućavanja pristupa WCF servisima, njihovim interfejsima, odnosno metodima na njihovim interfejsima u zavisnosti od prava pristupa koja WCF klijenti poseduju.

## Zadatak 3.

Proširiti WCF klijent-servis iz prethodnih zadataka sa kontrolom pristupa. Definisati dva nivoa pristupa, jedan za čitanje, drugi za ažuriranje podataka. Dodati barem po jednog korisnika koji poseduju različita prava pristupa (tj. jedan samo čitanje, drugi čitanje i pisanje) i prikazati uspešnu, odnosno neuspešnu proveru prava pristupa. Pokrenuti dva WCF klijenta: u prvom demonstrirati upis i čitanje podataka od strane korisnika koji ima sva prava. U drugom WCF klijentu prikazati uspešno čitanje i neuspešan upis podataka usled nedovoljnih prava pristupa.

Napomena: Iako .NET razvojno okruženje sadrži klase i tipove koji olakšavaju implementaciju kontrole pristupa, ovde ćemo (u edukativne svrhe) „od nule“ implementirati taj bezbednosni mehanizam.

Koraci rešavanja zadatka:

1) **Definicija prava pristupa.** Prava pristupa ćemo definisati u obliku enumeracije koja sadrži sva moguća prava određene klase. Sledeći listing sadrži definiciju enumeracije koju je potrebno uključiti u direktorijum korisnika definisan ranije u ovom poglavlju.

```
public enum EPravaPristupa { Citanje, Azuriranje };
```

2) **Dodela prava pristupa korisnicima.** Za memorisanje dodeljenih prava ćemo uvesti novu strukturu podataka u kojoj je ključ korisničko ime, a vrednost niz (jedinstvenih) prava koja dotični korisnik ima. Ovu strukturu ćemo takođe uključiti u direktorijum korisnika. Uočiti u listingu ispod da smo za implementaciju ove funkcionalnosti ugnjezdili dve strukture: spolja je direktorijum u kom je ključ korisničko ime, a vrednosti u direktorijumu su takođe složene strukture podataka tipa set. Set se koristi jer želimo da svako pravo bude uskladišteno samo jednom.

```
private Dictionary<string, SortedSet<EPravaPristupa>> prava
            = new Dictionary<string, SortedSet<EPravaPristupa>>();
```

3) **Konfiguracija prava pristupa korisnika.** U opštem slučaju bi prava pristupa bili konfigurisana u eksternom servisu ili barem u posebnom konfiguracionom fajlu. U cilju pojednostavljenja rešenja ćemo u našem slučaju dodeliti prava pristupa ranije definisanim korisnicima u C# izvornom kodu konstruktora direktorijuma korisnika – videti sledeći isečak listinga u kojem su masnim fontom označeni dosad dodati delovi izvornog koda za uvođenje mehanizma kontrole pristupa.

```
namespace EvidencijaLica
{
    public enum EPravaPristupa { Citanje, Azuriranje };

    class DirektorijumKorisnika
    {
        private Dictionary<string, Korisnik> korisnici
            = new Dictionary<string, Korisnik>();
        private Dictionary<string, string> autentifikovani
            = new Dictionary<string, string>();
        private Dictionary<string, SortedSet<EPravaPristupa>> prava
            = new Dictionary<string, SortedSet<EPravaPristupa>>();

        private const string _pepper = "P&0myWHq";

        public DirektorijumKorisnika()
        {
            DodajKorisnika("pera", "P3rA");
            DodajKorisnika("admin", "pr3Ax4dmin");

            SortedSet<EPravaPristupa> citanje
                = new SortedSet<EPravaPristupa>();
            citanje.Add(EPravaPristupa.Citanje);
            prava.Add("pera", citanje);

            SortedSet<EPravaPristupa> azuriranje
                = new SortedSet<EPravaPristupa>();
            azuriranje.Add(EPravaPristupa.Azuriranje);
            prava.Add("admin", azuriranje);
        }
        …
```

4) **Metod za kontrolu prava pristupa.** Cilj ovog metoda je da proveri da li korisnik ima potrebna prava za čitanje, odnosno ažuriranje opisa fizičkih lica. U listingu ispod je prikazano C# kod koji implementira ovu funkcionalnost u sledećim koracima (u okviru if uslova), tj. proverava:

   a. da li korisnik sa dobijenim tokenom postoji;

   b. da li korisnik postoji u strukturi sa dodeljenim pravima – ako ne postoji, to znači da nema nikakva prava;

   c. da li korisnik poseduje pravo pristupa navedeno kao ulazni parametar metoda.

Funkcija omogućava proveru jednog prava i ukoliko bilo koji od gore navedena tri uslova nije zadovoljen, baca WCF izuzetak.

```
public bool KorisnikAutorizovan(string token, EPravaPristupa pravo)
{
    if (autentifikovani.ContainsKey(token)
        && prava.ContainsKey(autentifikovani[token])
        && prava[autentifikovani[token]].Contains(pravo))
            return true;
    else
        throw new FaultException<BezbednosniIzuzetak>(
            new BezbednosniIzuzetak("Korisnik nema pravo: "
            + pravo.ToString()));
}
```

5) **Dodavanje opisa mogućih grešaka u interfejs.** Ukoliko metod WCF interfejsa može da baca određeni tip izuzetka, onda je potrebno da isti bude naveden u okviru deklaracije interfejsa. U našem slučaju je potrebno označiti metode za rukovanje sa opisima fizičkih interfejsa da mogu da bace bezbednosne izuzetke – videti nove linije naglašene masnim fontom u listingu sa opisom interfejsa ispod.

```
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        [FaultContract(typeof(BezbednosniIzuzetak))]
        void DodajLice(FizickoLice lice, string token);

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        [FaultContract(typeof(BezbednosniIzuzetak))]
        void ObrisiLice(long jmbg, string token);

        [OperationContract]
        [FaultContract(typeof(BezbednosniIzuzetak))]
        string SpisakLica(string token);
    }
}
```

6) **Provera prava pristupa.** Ovo je potrebno implementirati u svim metodima na WCF interfejsu za rad sa opisima fizičkih lica. Funkcija za listanje podataka o korisnicima treba da proveri da li korisnik ima pravo čitanja, a funkcije za dodavanje i uklanjanje opisa fizičkih lica treba da provere da li korisnik poseduje pravo ažuriranja. U listingu ispod je masnim fontom prikazan umetnut deo koda za proveru postojanja prava ažuriranja.

```
public void ObrisiLice(long jmbg, string token)
{
    ServisFizickihLica.direktorijum.KorisnikAutentifikovan(token);
    ServisFizickihLica.direktorijum.KorisnikAutorizovan(token,
        EPravaPristupa.Azuriranje);

    Console.WriteLine(DateTime.Now.ToString()
        + " - Detalji obrisanog lica:");
    Console.WriteLine(" - JMBG: " + jmbg.ToString());
}
```

Napomena: proveru postojanja adekvatnih prava pristupa je potrebno dodati u sva tri metoda.

Uočiti da je ovde prikazana implementacija mehanizma autorizacije korisnika potpuno transparentna sa stanovišta WCF klijenta. Ova karakteristika se oslikava u tome da WCF klijent uopšte nismo morali menjati, tj. sve izmene su bile lokalizovane u kodu WCF servisa. Ovo važi pod pretpostavkom da smo u izvornom kodu WCF klijenta od ranije imali kod za obradu bezbednosnog izuzetka.

### Testiranje

Za testiranje gore navedenih novih delova izvornog koda je potrebno više puta pokrenuti WCF klijenta i izvršiti barem sledeće testove:

1. Ulogovati se kao korisnik „admin“ i uočiti da će uspeti da doda i obriše korisnike, ali da neće imati potrebna prava za listanje podataka svih korisnika, tj. da će listanje podataka baciti bezbednosni izuzetak.
2. Ulogovati se kao korisnik „pera“ i uočiti da će on uspeti da se autentifikuje, ali da će prvi poziv za ažuriranje opisa fizičkih lica (npr. dodavanje opisa lica) baciti bezbednosni izuzetak, jer „pera“ ne poseduje pravo ažuriranja podataka.

Napomena: ukoliko od ranije postoje delovi izvornog koda u WCF servisu koji bezuslovno bacaju WCF izuzetke, onda je njih potrebno staviti pod komentar. Korisna prečica za komentarisanje dela izvornog koda u razvojnom okruženju Visual Studio je: (1) označavanje dela izvornog koda koji želimo da komentarišemo, (2) Ctrl + K, (3) Ctrl + C.

## Zadatak 4.

Pojednostaviti gornji listing objedinjavanjem poziva za autentifikaciju i autorizaciju korisnika. Klasu korisnik izmeniti tako da sadrzi podatke o tome da li je korisnik autentifikovan, koja su njegova prava i vrednost tokena.

### Serijalizacija podataka

Serijalizacija je proces konvertovanja objekata u tok bajtova kako bi se isti smeštali u memoriju, bazu podataka, datoteku ili slali preko računarske mreže. Glavni cilj serializacije jeste perzistencija stanja objekta i njegovo učitavanje iz trajnih skladišta podataka. Suprotna operacija od serializacije se zove deserializacija.

Kada se objekti serijalizuju u tok, ne zapisuju se samo podaci iz objekta već i informacije o tipu objekta, verziji objekta, lokalizaciji, biblioteci u kojoj se nalazi itd. Tokovi implementirani u .NET platformi imaju ugrađene klase koje služe za serializaciju i deserializaciju objekata. Ugrađeni serijalizatori su binarni, XML i SOAP, ali je moguće napraviti korisnički-definisane klase za serijalizaciju. Ovaj kurs pokriva i objašnjava samo XML serializaciju jer je njihov izlaz čitljiv za čoveka (i pogodan u edukativne svrhe) i jer čini osnovu Simple Object Access Protocol (SOAP) serializacije koja se koristi i u okviru Windows Communication Foundation (WCF), koji će biti korišćen u kasnijim primerima.

Kako bi klasa Korisnik mogla da se serijalizuje neophodno je dodati atribut [Serializable].

Kako bi se izvrsila serijalizacija neophodno je napraviti primerak klase *XmlSerializer* koji serializuje objekte tipa List<Korisnik>.

Napomena: potrebno je uključiti namespace System.Xml.Serialization.

Koristeći instancu klase *TextWriter* (namespace System.IO), pomoću objekta za serijalizaciju izvrsiti serijalizaciju.

```
XmlSerializer serializer = new XmlSerializer(typeof(List<Korisnik>));
using (TextWriter textWriter = new StreamWriter("../../korisnici.xml"))
{
       // ispis u XML datoteku
       serializer.Serialize(textWriter, korisniciList);
}
```

Izlazna datoteka sadrži sve XML elemente koji opisuju klasu i vrednost svakog elementa je vrednost koja je upisana u serijalizovani objekat u programu.

Podrazumevano ponašanje klase XmlSerializer je da sve javne članove i atribute prosleđenih objekata pretvora u XML elemente. Serijalizacija ne sadrži informacije o tipu klase. Klasa čije primerke želimo da serijalizujemo na ovaj način mora imati podrazumevani konstruktor. Članovi ispisane klase čiji upis nije dozvoljen (eng. ReadOnly) se ne serializuju.

Kako bi se kontrolisalo kako se serijalizuju članovi klase, ciljani članovi se mogu dekorisati sledećim atributima:

- *XmlElement*: Označava da će član klase biti XML element – ovo je podrazumevano.
- *XmlAttribute*: Označava da će član klase biti snimljen kao atribut u XML datoteci.
- *XmlIgnore*: Član klase se neće razmatrati prilikom serializacije.
- *XmlRoot*: Označava da će element biti korenski element XML datoteke.

Deserijalizacija objekata se radi na uz pomoc istog objekta klase Serializer razilika je u tome sto se pozvia metoda Deserialize:

- Napraviti primerak klase *XmlSerializer* koji serijalizuje objekte tipa Student.
- Koristeći primerak klase *TextReader* učitati XML datoteku i njen sadržaj deserijalizovati u primerak klase *object.*
- Eksplicitnim *cast*-om dobijeni *object* pretvoriti u primerak klase Student.

```
XmlSerializer serializer = new XmlSerializer(typeof(List<Korisnik>));
using (TextReader textReader = new StreamReader("../../korisnici.xml"))
{
       // ispis iz XML datoteke
       List<Korisnik> list = List<Korisnik>)serializer.Deserialize(textReader);
}
```

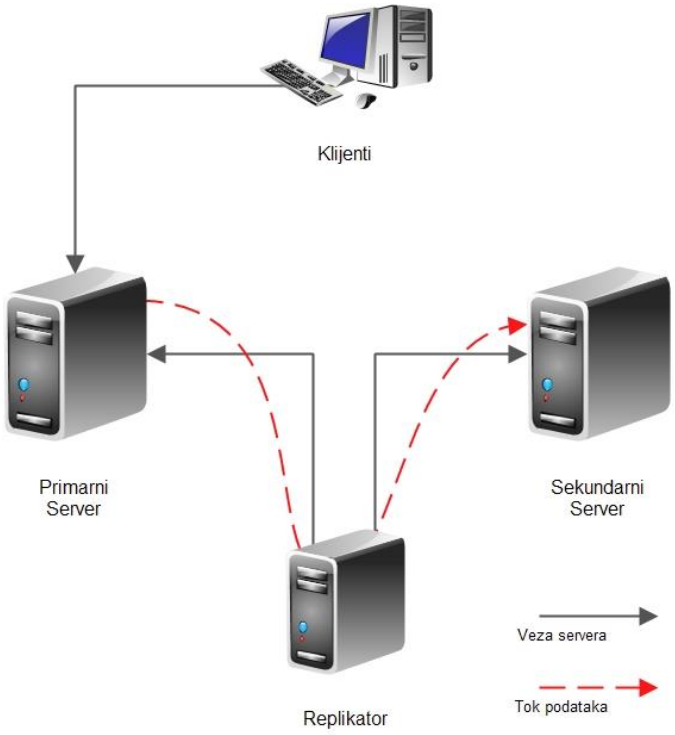
## Zadatak 5.

Ukloniti zakodirana korisnička imena i lozinke iz izvornog koda. Uređene parove (korisničko ime, kodirana lozinka, prava pristupa) prebaciti u XML datoteku, odvojiti deo koda koji sluzi za serijalizaciju i deserijalizaciju korisnika u XMl fajl.

# Vežba 8 – Replikacija

Replikacija se uvodi u distribuirane sisteme radi poboljšanja preformansi sistema i povećavanja dostupnosti istog. Moguća je (1) replikacija hardvera uvođenjem dodatnog hardvera koji stoji u raspolaganju ukoliko dođe do kvara opreme, (2) replikacija servisa kreiranjem više od jedne instance procesa koji pruža određenu uslugu u distribuiranom sistemu, odnosno (3) replikacija podataka u cilju izbegavanja gubitka podataka ukoliko dođe do kvara opreme ili softvera.

Cilj ovih vežbi je da prikaže sva tri tipa replikacije, sa naglaskom na replikaciji podataka. U cilju praktične implementacije mehanizma replikacije potrebno je klijent – server arhitekturu distribuiranog sistema proširiti, rezervnim serverima i dodatnim procesom replikatora podataka – videti sliku ispod za detalje.




Primerni i sekundarni servis pružaju istu uslugu i u optimalnom slučaju su pokrenuti na različitim fizičkim računarima, tj. jednostruki ispad hardvera ne dovodi do ispada oba servisa. Osnovni zadatak procesa replikatora je replikacija podataka sa jednog servera na drugi server.

## Redundantni servisi

### Zadatak 1.

Pokrenuti dva servisa za rukovanje sa opisima fizičkih lica.

Smernice za rešavanje zadatka:

1) Unaprediti interfejs servisa za rad sa opisima fizičkih lica dodavanjem metoda za čitanje i upis svih opisa lica.

```
namespace ZajednickiElementi
{
    [ServiceContract]
    public interface IFizickaLica
    {
        ...

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        List<FizickoLice> OcitavanjeLica();

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void UpisLica(List<FizickoLice> lica);
    }
}
```

2) Implementirati modifikovani interfejs.

```
namespace EvidencijaLica
{
    public class ServisFizickihLica : IFizickaLica
    {
        static List<FizickoLice> listaLica = new List<FizickoLice>();

        public void DodajLice(FizickoLice lice)
        {
            ...
        }

        public void ObrisiLice(long jmbg)
        {
            ...
        }

        public List<FizickoLice> OcitavanjeLica()
        {
            Console.WriteLine(DateTime.Now.ToString()
                + " - Ocitavanje lica inicirano.");
            return ServisFizickihLica.listaLica;
        }

        public void UpisLica(List<FizickoLice> lica)
        {
            Console.WriteLine(DateTime.Now.ToString()
                + " - Upis lica iniciran.");
            ServisFizickihLica.listaLica = lica;
        }
    }
}
```

Smernice za testiranje rešenja:

1) Pokrenuti po jedan servis za rukovanje sa opisima fizičkih lica na dva fizička (opciono virtuelna) računara. Nije potrebno modifikovati konfiguracije, tj. oba servisa mogu (i treba) da se pokrenu na istim TCP portovima. Ukoliko se servisi pokrecu na istim racunatima neophodno je
2) Diskutovati uticaj jednostrukog ispada servisa na dostupnost sistema od dva servisa.
3) Diskutovati uticaj jednostrukog ispada fizičkog (ili virtuelnog) računara na dostupnost distribuiranog sistema od dva servisa.

## Replikator podataka

### Zadatak 2.

Proširiti gore opisan distribuirani sistem sa replikatorom podataka. Zadatak procesa replikatora je da se periodično poveže na onaj servis koji proglasimo primarnim i da kopira sve podatke na onaj servis koji je proglašen rezervnim.

Smernice za rešavanje zadataka:

1) Konfigurisati tačke pristupa oba servisa podataka u XML konfiguraciji klijenta.

```
<system.serviceModel>
  <client>
    <endpoint name="Izvor"
      address="net.tcp://192.168.100.100:4000/IFizickaLica"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IFizickaLica" />
    <endpoint name="Odrediste"
      address="net.tcp:// 192.168.100.101:4000/IFizickaLica"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IFizickaLica" />
  </client>
</system.serviceModel>
```

   **Napomena:** u gornjem listingu su masnim fontom označene IP adrese servisa koje je u opštem slučajno poželjno zameniti simboličkim imenima računara na kojima su odgovarajući servisi pokrenuti.
2) U kod klijenta ubaciti beskonačnu petlju koja pauzira izvršavanje na 5 sekundi. Unutar beskonačne petlje ubaciti otvaranje komunikacionih kanala i ka oba servisa, čitanje svih podataka sa jednog, i upis svih podataka na drugom.

```
namespace Replikator
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                try
                {
                    ChannelFactory<IFizickaLica> cfIzvor
                        = new ChannelFactory<IFizickaLica>("Izvor");
                    ChannelFactory<IFizickaLica> cfOdrediste
                        = new ChannelFactory<IFizickaLica>("Odrediste");
                    IFizickaLica kIzvor = cfIzvor.CreateChannel();
                    IFizickaLica kOdrediste = cfOdrediste.CreateChannel();

                    kOdrediste.UpisLica(kIzvor.OcitavanjeLica());

                    Thread.Sleep(5000);
                }
                catch (FaultException<ServisFizickaLicaIzuzetak> ex)
                {
                    Console.WriteLine(ex.Detail.Razlog);
                }
            }
        }
    }
}
```

## Zadatak 3.

Uvesti dodatnog klijenta za programsko generisanje izmena podataka koji će svake tri sekunde raditi izmenu određenog podatka. Proveriti da li se tako izmenjeni podaci odmah repliciraju na sekundarni server.

## Zadatak 4.

Proširiti prethodni zadatak tako da je za pozivanje metoda za replikaciju neophodno da korisnik ima pravo pristupa Replicate, kako bi izvrsio replikaciju.

## Zadatak 5.

Unaprediti servise dodavanjem funkcionalnosti za očitavanje i upis samo onih opisa fizičkih lica koji su menjani od poslednje uspešne replikacije podataka.

Smernice za rešavanje zadatka:

1) Klasu *FizickoLice* proširiti sa poljem tipa datum i vreme koje predstavlja vreme kreiranja/izmene datog objekta. U metodu za očitavanje spiska fizičkih lica iskoristiti dato polje tako da se obezbedi očitavanje samo onih fizičkih lica koja su modifikovana i/ili dodata posle zadatog vremena.
2) Opis metoda za očitavanje opisa fizičkih lica proširiti sa ulaznim parametrom datum i vreme.
3) Implementaciju metoda za očitavanje opisa fizičkih lica proširiti i uvažiti novi ulazni parametar tipa datum i vreme.
4) U kodu replikatora čuvati vreme poslednje sinhronizacije i slati ga na servis prilikom očitavanja podatka.

5) Modifikovati kod za upis spiska fizičkih lica tako da se ne prepisuje celo skladište fizičkih lica, već da se uradi provera svakog prispelog opisa lica i donese odluka o upisivanju novog, odnosno ažuriranju postojećeg opisa.

## Zadatak 6.

Unaprediti rešenje prethodnog zadatka sa mehanizmom brisanja opisa fizičkih lica. Obratiti pažnju na replikaciju akcija brisanja.

Smernice za rešavanje zadatka, tj. moguće opcije rešavanja ovog problema:

1) preći na potpuno osvežavanje, tj. čitanje i upis **svih** opisa u replikama;
2) prepraviti ceo mehanizam replikacije na replikaciju operacija, tj. da se umesto podataka repliciraju operacije - u našem slučaju bi replicirali uređene parove (operacija, opis/jmbg).
   **Napomena:** Motivisati studente da smisle alternativna rešenja ovog problema.

## Zadatak 7.

Unaprediti proces replikatora sa ugrađenim baferom u kom se privremeno čuvaju podaci u slučaju ispada rezervnog servisa, tj. odredišta repliciranih podataka. Analizirati memorijsko zauzeće procesa replikatora u slučaju dugotrajnog ispada rezervnog servisa.

## Zadatak 8.

Implementirati distribuiran sistem u kom je moguć upis podataka preko bilo kog servisa. Unaprediti replikacionog klijenta sa mogućnošću repliciranja podataka u takvom okruženju. Rukovati sa potencijalnom cirkularnom replikacijom podataka.

# Vežba 9 – Otpornost na otkaze

U distribuiranim sistemima se otpornost na otkaze povećava uvođenjem bezbednosnih kopija, tj. replika. Moguće je replicirati hardver, softver (tj. servise) i podatke. U prethodnim vežbama je pokazano jedno jednostavno rešenje za replikaciju podataka. Cilj ove vežbe je da pokaže da dodavanje rezervnog servisa (tj. dodatnog serverskog procesa) omogućava distribuiranom sistemu da se automatski oporavi od jednostrukog ispada servisa.

## Redundantni servisi

Ukoliko u DS postoji veći broj servisa, onda oni mogu biti ravnopravni i da se zahtevi klijenata dele između njih. Takva implementacija je karakteristična za veb servere, koji pokreću veći broj paralelnih procesa za opsluživanje zahteva. Servisi u industrijskim sistemima su uglavnom složeniji i njihove redundantne konfiguracije se najčešće sastoje od primarnog servisa i jednog ili više rezervnih servisa. Klijenti u normalnom radu komuniciraju sa primarnim servisom. Ukoliko dođe do njegovog ispada, njegovu ulogu preuzima jedan (ili jedini) rezervni servis. U ovim vežbama ćemo prikazati upravo ovaj način replikacije servisa.

## Zadatak 1.

Omogućiti uvođenje rezervnog servisa za obradu podataka o fizičkim licima. Omogućiti sledeća stanja servisa: nepoznato, primarni i rezervni. Omogućiti konfigurisanje inicijalnog stanja servisa preko XML konfiguracione datoteke. Dodati i implementirati WCF interfejs za proveru statusa servisa (tj. da li je aktivan) i ažuriranje stanja servisa (tj. postavljanje jednog od gore navedena tri stanja). Pokrenuti jedan primarni i jedan rezervni servis. Povezati WCF klijenta na primarni servis.

Smernice za rešavanje zadatka:

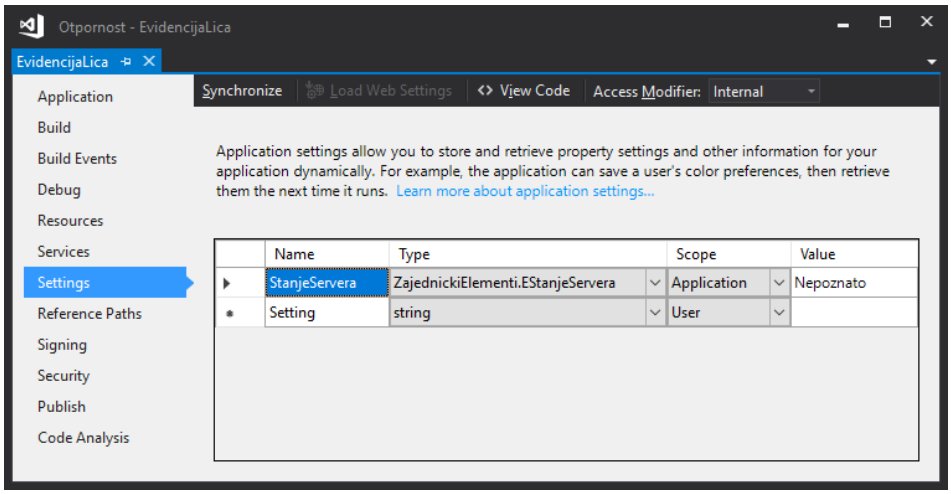
1) Krenuti od naprednog WCF klijent-server rešenja sa uvedenim radom sa izuzecima i XML konfiguracionim datotekama, ali bez autentifikacije i replikacije podataka. Ovo praktično znači da se u rešavanje ovog zadatka kreće kreiranjem kopije projekata tog WCF servisa i klijenta.
2) Dodati interfejs koji sadrži po jedan metod za očitavanje i postavljanje (tj. ažuriranja) stanja servisa – videti listing ispod. Uočiti da su stanja uvedena u obliku enumeracije i da metodi za čitanje i ažuriranje stanja bacaju tip izuzetka koji je uveden ranije – ovde nije uveden novi tip izuzetka koji bi bio specifičan za ovaj interfejs u cilju pojednostavljenja rešenja.
3) Elementi enumeracije su označeni sa odgovarajućim atributom koji omogućava njihov prenos kroz WCF komunikacioni kanal.Napomena: ukoliko se ne koriste gore navedene atributi za označavanje elemenata enumeracija, dolaziće do WCF izuzetaka i neće biti moguće pozivanje metoda koji razmenjuju primerke dotične enumeracije.

```
namespace ZajednickiElementi
{
    [DataContract(Name = "EStanjeServera")]
    public enum EStanjeServera {
        [EnumMemberAttribute]
        Nepoznato,
        [EnumMemberAttribute]
        Primarni,
        [EnumMemberAttribute]
        Sekundarni }

    [ServiceContract]
    public interface IStanjeServisa
    {
        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        EStanjeServera ProveraStanja();

        [OperationContract]
        [FaultContract(typeof(ServisFizickaLicaIzuzetak))]
        void AzuriranjeStanja(EStanjeServera stanje);
    }
}
```

4) Dodati konfiguracioni element u XML konfiguracionu datoteku. Ovo se postiže u sledećim koracima:
   a. Desni klik na projekat WCF servisa;
   b. U novom prozoru biranje opcije Settings u meniju sa leve strane;
   c. Dodavanje zapisa sa sledećim detaljima: ime StanjeServera, tip EStanjeServera (tj. naša enumeracija iz listinga iznad), vrednost Nepoznato – videti izgled prozora za unos konfigracionog zapisa ispod;




Napomena: otvoriti App.config u razvojnom okruženju i uočiti dodatnu sekciju u XML datoteci koja odgovara ovom zapisu.

5) Dodati klasu za opis konfiguracije WCF servisa. Ova klasa u svom konstruktoru treba da učita vrednost stanja servisa iz konfiguracione datoteke. Uočiti da konstruktor pored učitavanja vrši i ispis trenutne konfiguracije servisa.

```
namespace EvidencijaLica
{
    class KonfiguracijaServisa
    {
        private EStanjeServera stanjeServera;

        public EStanjeServera StanjeServera {
            get => stanjeServera;
            set => stanjeServera = value; }

        public KonfiguracijaServisa()
        {
            this.stanjeServera
                = Properties.Settings.Default.StanjeServera;

            Console.WriteLine("Stanje servisa je: "
                + this.StanjeServera.ToString());
        }
    }
}
```

6) Implementirati interfejs za rukovanje sa stanjem servisa. U cilju jasnog razdvajanja izvornog koda koji implementira raznorodne funkcionalnosti, uvešćemo dodatnu klasu koja implementira ovaj interfejs – videti listing ispod.

```
namespace EvidencijaLica
{
    public class ObradaStanja : IStanjeServisa
    {
        private static KonfiguracijaServisa konfiguracija
            = new KonfiguracijaServisa();

        public void AzuriranjeStanja(EStanjeServera stanje)
        {
            ObradaStanja.konfiguracija.StanjeServera = stanje;
        }

        public EStanjeServera ProveraStanja()
        {
            return ObradaStanja.konfiguracija.StanjeServera;
        }
    }
}
```

Napomena: uočiti da gore navedena klasa sadrži konfiguraciju servisa (tj. primerak odgovarajuće klase iz ranijeg listinga) kao statičkog člana.

7) Konfigurisati tačku pristupa gore implementiranom WCF servisu u XML konfiguracionoj datoteci – videti masnim slovima označen relevantan deo konfiguracije u listingu ispod. Praktično je dodat opis novog servisa koji će biti podignut na portu 4001.

```
  <system.serviceModel>
    <services>
      <service name="EvidencijaLica.ServisFizickihLica">
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:4000/IFizickaLica" />
          </baseAddresses>
        </host>
        <!-- Service Endpoints -->
        <endpoint address="" binding="netTcpBinding"
contract="ZajednickiElementi.IFizickaLica" />
      </service>
      <service name="EvidencijaLica.ObradaStanja">
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:4001/IStanjeServisa"
/>
          </baseAddresses>
        </host>
        <!-- Service Endpoints -->
        <endpoint address="" binding="netTcpBinding"
contract="ZajednickiElementi.IStanjeServisa" />
      </service>
    </services>
  </system.serviceModel>
```

8) Dodati C# kod za pokretanje dodatnog servisa – videti masnim slovima označen deo u listingu metoda Main u okviru klase Program ispod.

```
namespace EvidencijaLica
{
    class Program
    {
        static void Main(string[] args)
        {
            ServiceHost svc = new ServiceHost(typeof(ServisFizickihLica));
            svc.Open();

            ServiceHost svcStanja = new ServiceHost(typeof(ObradaStanja));
            svcStanja.Open();

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje servisa.");
            Console.ReadLine();
        }
    }
}
```

Za testiranje gore navedenog rešenja je potrebno uraditi sledeće:

1) prevesti projekat;
2) kopirati dobijene binarne i konfiguracione fajlove u odvojene foldere, npr. kreirati foldere „primarni“ i „sekundarni“ u direktorijumu /Temp;
3) promeniti XML konfiguraciju obe kopije fajlova servisa na sledeći način:
   a. u folderu „primarni“ staviti inicijalno stanje servisa da bude „Primarni“ unošenjem odgovarajućeg teksta u App.config
   b. u folderu „sekundarni“ staviti inicijalno stanje servisa da bude „Sekundarni“ unošenjem odgovarajućeg teksta u App.config;
4) interfejse sekundarnog servisa konfigurisati na portovima 5000 i 5001 – ukoliko se ovo ne uradi, neće biti moguće pokretanje sekundarnog servisa ukoliko je primarni već pokrenut (ili obrnuto);
5) pokrenuti obe kopije WCF servisa iz gore navedenih foldera; i pokrenuti WCF test klijenta.

Napomena: gore naveden postupak ima jednu značajnu manjkavost, jer ukoliko nađemo grešku u izvornom kodu, ispravimo je i ponovo prevedemo WCF servis, onda ćemo morati da binarne fajlove (ne i konfiguracione) ponovo kopiramo u dodatne foldere.

U ovom trenutku klijent nije svestan postojanja sekundarnog servisa i uvek se povezuje na primarni servis je tako konfigurisan još od ranije (tj. konekcija preko TCP porta 4000).

## Praćenje stanja servisa

U gore navedenom zadatku i njegovom rešenju su serveri potpuno nezavisni i u zavisnosti od njihove konfiguracije imaju različita, ali potencijalno i ista unutrašnja stanja, npr. dva primarni servera. Postojanje dva primarna servera je često neželjena pojava i na engleskom se naziva „split brain“, jer u tom slučaju imamo dva procesa koja mogu imati različite podatke (tj. imati nekonzistentne podatke) i opsluživati podskup klijenata u distribuiranom sistemu.

Za rešavanje gore navedenog problema, pa i njemu sličnih problema se često uvodi dodatni proces, koji nadzire status svih servisa i upravlja sa njima. U kontekstu operativnih sistema se takvi upravljački procesi nazivaju „*watchdog*“ procesima.

## Zadatak 2.

Dodati dodatni proces koji nadzire i podešava stanja WCF servisa. Nakon pokretanja sa povezuje na oba procesa, jedan podešava da bude primarni, drugi sekundarni. Periodično proverava stanja oba servisa. Detektuje ispad primarnog servisa ukoliko ne uspe da očita njegovo stanje i proglašava sekundarni servis primarnim.

Smernice za rešavanje zadatka:

1) Dodati novi projekat tipa konzolna aplikacija. Imenovati ga Monitor ili SistemMonitor.

2) Dodati tačke pristupa interfejsima za rad sa stanjima servisa na oba servisa u konfiguracionu datoteku – videti listing ispod.

```
<system.serviceModel>
  <client>
    <endpoint name="Primarni"
      address="net.tcp://localhost:4001/IStanjeServisa"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IStanjeServisa" />
    <endpoint name="Sekundarni"
      address="net.tcp://localhost:5001/IStanjeServisa"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IStanjeServisa" />
  </client>
</system.serviceModel>
```

Uočiti da se konfiguracije tačaka pristupa osim imena razlikuju samo po TCP portu na kom su servisi dostupni.

3) Otvoriti po jedan komunikacioni kanal na oba servisa iz metoda Main u okviru klase Program. Konfigurisati servis na portu 4001 da bude primarni pozivanjem odgovarajućeg metoda na interfejsu za rad sa stanjem servisa. Na sličan način konfigurisati servis na portu 5001 da bude sekundarni.

```
IStanjeServisa primarni = null;
IStanjeServisa sekundarni = null;
try
{
    ChannelFactory<IStanjeServisa> cfPrimarni
        = new ChannelFactory<IStanjeServisa>("Primarni");
    primarni = cfPrimarni.CreateChannel();
    primarni.AzuriranjeStanja(EStanjeServera.Primarni);
} catch (CommunicationException cex)
{
    Console.WriteLine("Primarni servis nedostupan. Razlog: "
        + cex.Message);
}
try
{
    ChannelFactory<IStanjeServisa> cfSekundarni
        = new ChannelFactory<IStanjeServisa>("Sekundarni");
    sekundarni = cfSekundarni.CreateChannel();
    sekundarni.AzuriranjeStanja(EStanjeServera.Sekundarni);
}
catch (CommunicationException cex)
{
    Console.WriteLine("Sekundarni servis nedostupan. Razlog: "
        + cex.Message);
}

if (primarni is null && sekundarni is null)
{
    Console.WriteLine("Neuspelo povezivanje na servise.");
    Environment.Exit(0);
}
```

Napomena: poslednji deo listinga sa if uslovom prekida izvršavanje procesa u slučaju kada ne uspe povezivanje ni na primarni, ni na sekundarni servis.

4) Ubaciti beskonačnu while petlju u okviru koje se očitava status oba servisa i u zavisnosti od njihovih vrednosti ili ne izvršavaju nikakve radnje, ili se sekundarni servis proglašava primarnim ukoliko dođe do njegovog ispada.

```
while (true)
{
    EStanjeServera stanjePrimar = EStanjeServera.Nepoznato;
    EStanjeServera stanjeSekundar = EStanjeServera.Nepoznato;
    try
    {
        stanjePrimar = primarni.ProveraStanja();
    } catch (Exception ex)
    {
        Console.WriteLine("Greska na primarnom: " + ex.Message);
    }
    try
    {
        stanjeSekundar = sekundarni.ProveraStanja();
    } catch (Exception ex)
    {
        Console.WriteLine("Greska na sekundarnom: " + ex.Message);
    }

    Console.WriteLine("Stanja servisa.");
    Console.WriteLine("- primarni:   " + stanjePrimar.ToString());
    Console.WriteLine("- sekundarni:    " + stanjeSekundar.ToString());

    if (stanjePrimar == EStanjeServera.Nepoznato)
    {
        if (stanjeSekundar == EStanjeServera.Sekundarni)
        {
            sekundarni.AzuriranjeStanja(EStanjeServera.Primarni);
            Console.WriteLine("Sekundarni proglasen primarnim.");
        }
    }
    if (stanjeSekundar == EStanjeServera.Nepoznato)
    {
        Console.WriteLine("Sekundarni nije operativan.");
    }
    Thread.Sleep(5000);
}
```

Uočiti da se ispad sekundarnog servisa takođe detektuje u if uslovu na kraju listinga iznad i da se vrši ispis odgovarajuće poruke u komandnoj liniji.

Za proveru ispravnog funkcionisanja procesa za praćenje statusa servisa je potrebno izvršiti (barem) sledeće testove:

1) Pokrenuti oba servisa i pratiti rad monitora.
2) Zaustaviti primarni servis i pratiti ispis.
3) Zaustaviti sekundarni servis i pratiti ispis.

## Zadatak 3.

Omogućiti (ponovno) povezivanje na servise unutar beskonačne petlje. Ovim se dodaje podrška za privremeni ispad i ponovno pokretanje jednog ili oba servisa.

## Zadatak 4.

Omogućiti rad sa većim brojem ravnopravnih servisa od kojih se samo jedan proglašava primarnim i svi ostali sekundarnim, tj. rezervnim servisima. Napomena: dodati niz komunikacionih kanala, npr. korišćenjem kontejnera List.

## Klijenti redundantnih servisa

Naglasak gore navedenih primera je bio na postojanju redundatnih servisa i na njihovom nadzoru i konfigurisanju. Korisnik tih servisa nije menjan i ne sadrži logiku za prevezivanje na rezervni servis u slučaju ispada primarnog, aktivnog servisa.

## Zadatak 5.

Unaprediti korisnika usluga gore implementiranih servisa (tj. WCF klijenta) dodavanjem koda za detekciju ispada primarnog servisa i za automatsko prevezivanje na sledeći primarni servis ukoliko on postoji. Tačke pristupa redundantnih servisa podesiti na uobičajen način preko XML konfiguracione datoteke klijenta.

Smernice za rešavanje zadatka:

1) Uneti dve tačke pristupa u XML konfiguracionu datoteku.

```
<system.serviceModel>
  <client>
    <endpoint name="ServisLica_0"
      address="net.tcp://localhost:4000/IFizickaLica"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IFizickaLica" />
    <endpoint name="ServisLica_1"
      address="net.tcp://localhost:5000/IFizickaLica"
      binding="netTcpBinding"
      contract="ZajednickiElementi.IFizickaLica" />
  </client>
</system.serviceModel>
```

   Napomena #1: imena tačaka pristupa su formirana tako da se olakša njihovo programsko formiranje preko indeksa u nizu.
   Napomena #2: uočiti da se tačke pristupa pored imena razlikuju samo po TCP portu. Ovako je omogućeno testiranje sa primarnim i sekundarnim servisom na istom računaru. U opštem slučaju bi se oni nalazili na različitim fizičkim računarima.

2) Proširiti izvorni kod WCF klijenta dodavanjem for petlje za programsko generisanje imena tačke pristupa i pokušati povezivanje na sve konfigurisane tačke pristupa. Ispisati ishod pokušaja povezivanje, kako pozitivan, tako i negativan.

```
namespace TestKlijent
{
    class Program
    {
        static void Main(string[] args)
        {
            ChannelFactory<IFizickaLica> cfLica = null;
            IFizickaLica servisLica = null;
            for (uint i = 0; i <= 1; i++) {
                try
                {
                    cfLica = new ChannelFactory<IFizickaLica>("ServisLica_"
                        + i.ToString());
                    servisLica = cfLica.CreateChannel();
                    servisLica.SpisakLica(); // poziv kao test
                    Console.WriteLine("Uspesno povezivanje na ServisLica_"
                        + i.ToString());
                    break;
                } catch (Exception ex)
                {
                    Console.WriteLine("Neuspelo povezivanje na ServisLica_"
                        + i.ToString());
                    cfLica = null; servisLica = null;
                }
            }

            if (servisLica is null)
            {
                Environment.Exit(0);
            }

            Console.WriteLine("Pritisnite [Enter] za zaustavljanje klijenta.");
            Console.ReadLine();
        }
    }
}
```

Za testiranje WCF klijenta izvršiti (barem) sledeće testove:

1) Pokrenuti primarni servis iz komandnog prompta i potom pokrenuti klijenta i posmatrati ispis u komandnoj liniji;
2) Zaustaviti primarni servis, pokrenuti samo sekundarni servis (na portu 5000), pokrenuti klijenta i posmatrati ispis u komandnoj liniji;
3) Zaustaviti oba servisa, pokrenuti WCF klijenta i posmatrati njegov ispis u komandnoj liniji.

## Zadatak 6.

Pokrenuti ukupno tri servisa na portovima 4000, 5000 i 6000. Podesiti prvi da bude u primaran, a preostala dva da budu sekundarna. U XML konfiguracioni fajl klijenta uneti sve tri tačke pristupa i modifikovati izvorni kod tako da proba pristupiti svim konfigurisanim servisima. Uopštiti rešenje na proizvoljan broj servisa, npr. 4.

## Zadatak 7.

Onemogućiti pristup servisima koji nisu u odgovarajućem stanju. Ovo praktično znači da servisi treba da bace izuzetak iz svih metoda na interfejsu za rukovanje opisima fizičkih lica ukoliko nisu u stanju „Primarni“.