VIRTUELIZACIJA PROCESA

Praktikum
2023/2024

Profesor:
Bojan Jelačić

Asistenti :
Zorana Babić
Jelena Petrić
Sandra Brkić

Uvod

Polaganje predmeta :

- Predispitne obaveze – 70 bodova
- Ispitne obaveze – 30 bodova

Vežbe :

- Alat koji se koristi prilikom izrade zadataka je Visual Studio (verzija po želji).
- Prisustvo na vežbama je **obavezno** i nosi 5 bodova.
- Tokom semestra studenti imaju dva kolokvijuma.
- Prvi kolokvijm će se održati u nedelji 15.04.2024. - 19.04.2024 u terminima vežbi i nosi 30 bodova, na prvi kolovijum ide gradivo koje se izučava u prvih 5 termina.
- Drugi kolokvijum će se održati u nedelji 27.05.2024. - 31.05.2024. u terminima vežbi i nosi 35 bodova, na drugi kolokvijum ide gradivo koje se izučava u drugih 5 termina.
- Poprvani kolokvijum će se održati u nedelji 03.06.2024. - 07.06.2024. Za popravni kolokvijum student je dužan da se prijavi sa naznakom koji kolokvijum popravlja. Na popravnom kolokvijumu može da se popravlja ***isključivo jedan kolokvijum***, izlaskom na popravni kolokvijum poništavaju se prethodno stečeni bodovi kolokvijuma koji se popravlja.
- Da bi student položio predispitne obaveze neophodno je da skupi minimum 36 bodova u zbiru (vežbe + K1 + K2).

Sadržaj

Vežba 1 - Osnove WCF-a

Windows Communication Foundation (WCF) je framework koji služi za pravljenje servisno-orijentisanih aplikacija. Uz pomoć WCF-a imamo mogućnost slanja asinhronih poruka sa jednog na drugi servis.

Svaki WCF servis definiše minimalno jedan endpoint koji sadrži skup neophodnih informacija koje su potrebne WCF klijentu za pristup servisu.

Svaki endpoint mora da sadrži sledeće informacije

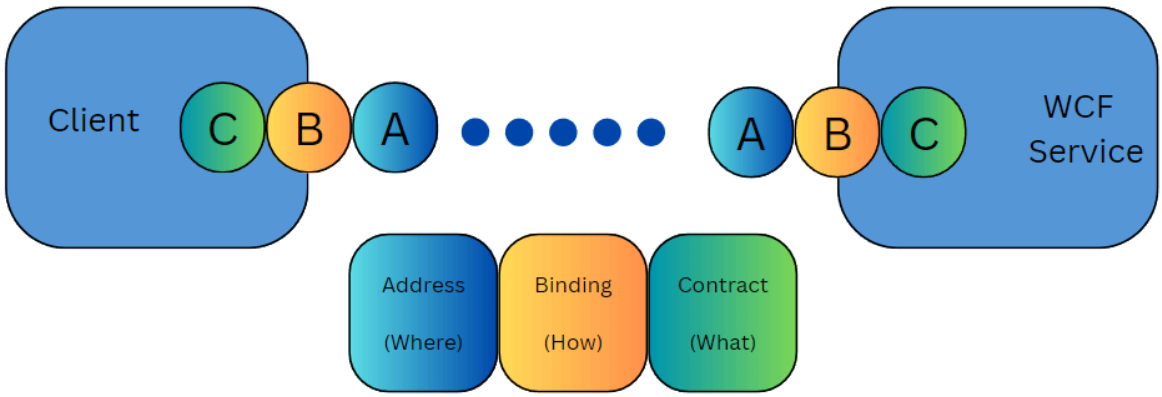
A. Informacije o adresi na kojoj servis sluša (address) koja je predstavljena u standardnom URI formatu na sledeći način:

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.

B. O komunikacionom protokolu (binding),

C. Informacije o opisu usluge (contract) koje servis pruža klijentima.



Slika 1. Osnovni elementi WCF Endpoint -a

Konfiguracija i pokretanje servisa

Kako bismo pokrenuli neki servis prethodno moramo definisati interfejs koji ćemo koristiti kao Contract. Da bi se jedan interfejs koristio kao contract neophodno ga je dekorisati atributom *ServiceContract*, a metode koje želimo da izlaže sa *OperationContract* atributom (nalaze se u *System.ServiceModel* biblioteci). Primer interfeja se nalazi ispod.

```
[ServiceContract]
public interface IExample
{
    [OperationContract]
    string TestFunction();
}
```

Rad se složenim podatacima

Pored prenošenja ugrađenih tipova WCF nudi mogućnost slanja korisnički definisanih tipova podataka. U WCF-u se koristi *DataContract* serijalizacija 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.

```
[DataContract]
public class Knjiga
{
    public Knjiga(string nazivKnjige, string imeAutora, string prezimeAutora)
    {
        this.NazivKnjige = nazivKnjige;
        this.ImeAutora = imeAutora;
        this.PrezimeAutora = prezimeAutora;
    }

    [DataMember]
    public string NazivKnjige{ get => nazivKnjige; set => nazivKnjige= value; }
    [DataMember]
    public string ImeAutora{ get => imeAutora; set => imeAutora= value; }
    [DataMember]
    public string PrezimeAutora{ get => prezimeAutora; set => prezimeAutora= value; }

    private string imeAutora;
    private string imeAutora;
    private string prezimeAutora;
}
```

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.

Navođenje izuzetka koji WCF metod može da baci je moguće navođenjem atributa *FaultContract* iznad metode koja može da baci izuzetak.

```
[OperationContract]
[FaultContract(typeof(Exception))]
    string TestFunction();
```

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 koji su izvršnog tipa. Za rad sa konfiguracionim datotekama potrebno je uključiti **System.Configuration** namespace.

U donjem listingu je prikazano na koji način se konfiguriše server.

```
ServiceHost svc = new ServiceHost(typeof(ServiceName));
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="NamespaceOfService.ServiceName">
        <host>
          <baseAddresses>
            <add   baseAddress="net.tcp://localhost:4000"
          /> </baseAddresses>
        </host>
        <!-- Service Endpoints -->
        <endpoint
                  address="Service"
                  binding="netTcpBinding"
                  contract="NamespaceOdInterface.Interface" />
        </service>
      </services>
  </system.serviceModel>
</configuration>
```

U donjem listingu je prikazano na koji način se konfiguriše klijent.

```
ChannelFactory<InterfaceName> factory =
    new ChannelFactory<InterfaceName>("NazivServisa");
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="NazivServisa"
        address="net.tcp://localhost:4000/Service"
        binding="netTcpBinding"
        contract="NamespaceOdInterface.InterfaceName" />
    </client>
  </system.serviceModel>
</configuration>
```

Zadatak 1. WCF

Neophodno je implementirati klijent-server arhitekturu koristeći konfiguracioni fajl prilikom konfigurisanja servisa i klijenta.

Servis treba da izloži sledeće metode :

- Dodavanje novih knjiga u biblioteku
- Brisanje postojećih knjiga iz biblioteke, metoda treba da baci CustomException izuzetak ukoliko pokuša da se obriš knjiga koja ne postoji
- Izlistavanje svih knjiga koje postoje u biblioteci
- Izlistavanje svih knjiga po nazivu autora (ime + prezime)
- Izlistavanje svih knjiga na osnovu zanra
- Izlistavanje svih knjiga koje su izdate odredjene godine

Treba implementirati klase Knjiga i CustomException
Klasa Knjiga treba da sadrži :

- ID knjige (predstavlja redni broj kreiranog objekta klase)
- Ime knjige
- Ime autora
- Prezime
- Zanr (predstavljen enumeracijom)
- Datum izdavanja
- Za svako odgovarajuce polje obezbediti Property
- Za inicijalizaciju objekta NE koristiti default-ni konstruktor

Klasa CustomException treba da sadrzi

- Message i odgovarajuci Property

Metode interfejsa implementirati na strani Servisa, sa strane servisa neophodno je da postoji kolekcija Knjiga.

Dodatak : Uputstvo za kreiranje projekta

- Prilikom kreiranja novog projekta neophodno je obratiti pažnju da svi projekti budu **.NETFramework** (Slika 2)

Slika 2. Odabir projekta u VS

- **Common** projekat je tipa **ClassLibrary**, u njega se piše sve ono što je neophodno da bude vidljivo i klijentu i serveru (interfejsi, klase…)
- Biblioteke **System.ServiceModel** i **System.Runtime.Serialization** u sebi sadrže sve neophodno za kreiranje klijenta (ChannelFactory) i servera (ServiceHost) kao i attribute kojima oznacavamo ServiceContract i DataContract stoga one moraju biti dodate u one projekte gde je to neophodno
- Servisa implementira interfejs i vrši hostovanje kako bi klijent mogao pristupi ovim metodama
- Klijent otvara kanal ka servisu i vrši pozivanje ovih metoda
- Neophodno je dobro konfigurisati app.config kako bi klijent i servis mogli da se povežu uspešno

Vežba 2 – Kolekcije podataka

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>();
```

Prilikom inicijalizacije liste potrebno je 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 };
```

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 106 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.

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 metoda

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.
long temp = dictObj[5];
// Ukoliko element ne postoji, baca se izuzetak KeyNotFoundException.
```

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
objList.Add("Novi 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("Novi element");
```

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();
```

Zadaci

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.

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.

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

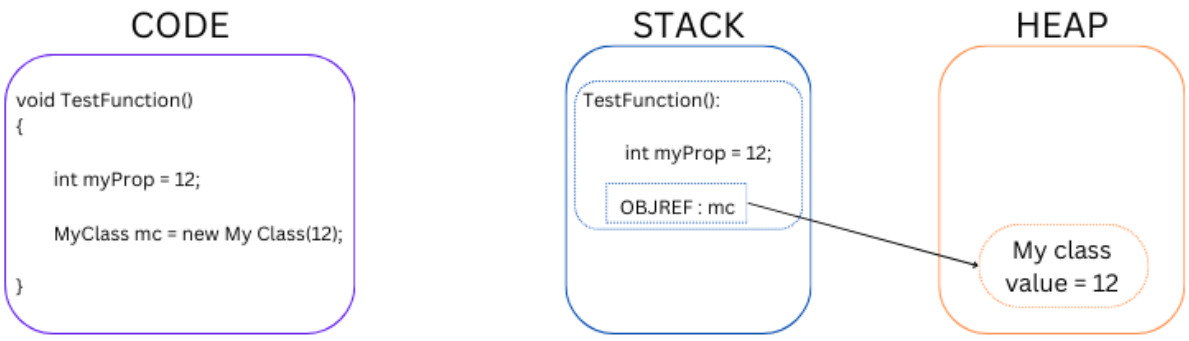
Inicijalizovati dve proizvoljne dictionary kolekcije gde je ključ tipa celobrojne vrednosti, a vrednost predstavlja listu stringova. Kreirati novu dictionary kolekciju koja predstavlja spoj dve kolekcije kreirane u prethodnom koraku, tako da ukoliko se ključevi ponavljaju u obe kolekcije vrednost predstavlja spojenu listu stringova iz obe kolekcije.

Vezbe 3 - Manipulacija memorijom

Osnovni tipovi podataka

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)*. U referentne tipove spadaju klase, nizova, delegati i interfejsi.



Slika 3. Skladistenja podataka na stack-u i heap-u

Memorijski prostor

Sve podatke vezane za određenu metodu: ulazne parametre, lokalno definisane varijable, adresu linije koda koja treba da se izvrši nakon završetka metode se čuvaju na **Stack**-u. Kada se metoda završi, deo memorije (container) sa vrha se briše, tok izvršavanja se nastavlja na liniji koja je bila zapisana i na vrhu **Stack**-a se opet nalazi metoda koja se trenutno izvršava.

Kada se kreiraju instance referentnog tipa njihova referenca (adresa) se čuva na **Stack**-u dok se sama instanca čuva na Heap-u.

Postoje dve vrste **Heap**-a:

1. Small Object Heap (SOH) za skladištenje objekata veličine manje od 85K
2. Large Object Heap (LOH) za skladištenje objekata veličine veće od 85K

Treba imati u vidu da je skladištenje podataka na ***Heap***-u i njihova kontrola od strane **Garbage Collectora** 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.

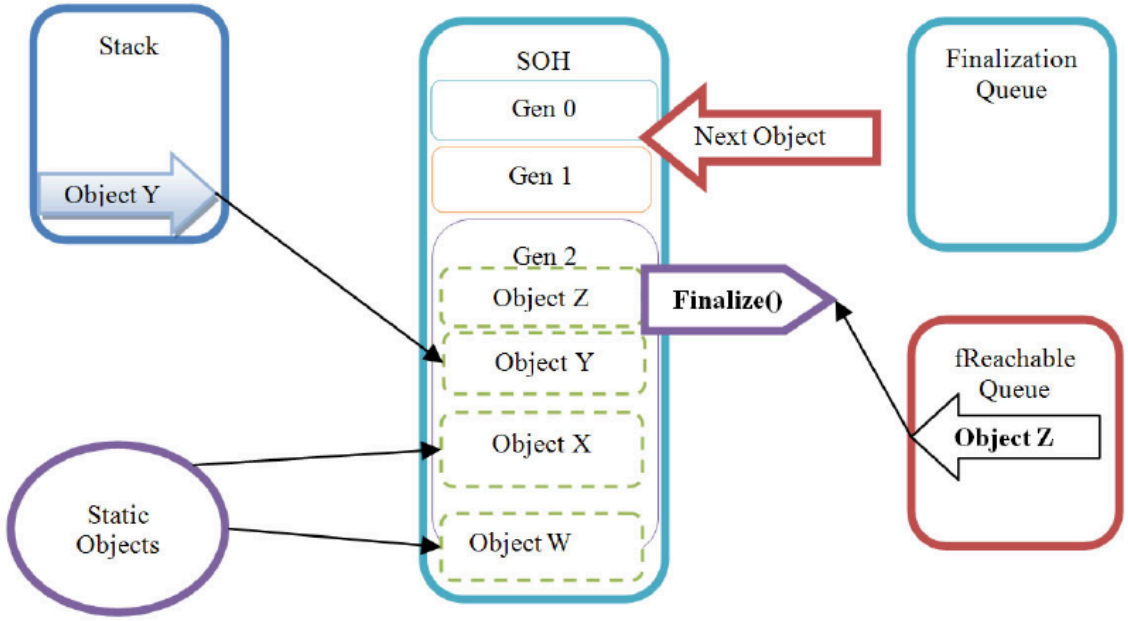
Glavni modul za čišćenje memorije je Garbage Collector (GC). Sve što ovaj modul radi je da prati Heap i registruje sve objekte koji nemaju referencu. Referenca može postojati na Stack-u ali mogu postojati i drugi izvori: globalna promenljiva, CPU registar, referenca na objekat u finalizaciji, interop reference. Sve ove vrste referenci se nazivaju root references ili krace GC roots. Ako neki objekat nema root referencu, praktično ne može biti iskorišćen od strane koda. Postaje beskorisna i zauzeta memorija koja treba da bude počišćena.

GC prikupi sve root reference i krećući se kroz stablo objekata markira sve objekte koji su trenutno u upotrebi (živi objekti). Svi objekti koji nisu markirani treba da budu pokupljeni (collected). Kada se završi markiranje, GC započinje proces čišćenja Heap-ova. Na kraju se resetuje lista objekata koji su još u upotrebi i GC je spreman za novi ciklus.

Ako se piše kod koji koristi neki resurs koji nije managed, recimo rad sa fajlovima, mrežnim resursima, bazama podataka, tada moramo povesti računa da na adekvatan način oslobodimo ove resurse pre nego što se objekat uništi. Ovo se rešava destruktorom ili Finalize metodom. Način na koji se piše destruktor je prikazan u listingu ispod.

```
class TestClass
{
    ~TestClass()
    {
        // code of finalize function
    }
}
```

Svi objekti koji imaju Finalize metodu dobijaju dodatnu root referencu koja se smešta u Finalization queue. Nakon prvog prolaska, kada je objekat spreman za collect, ova referenca se briše iz Finalization queue i premešta u fReachable queue. U ovom slučaju je ovo i jedina referenca i Finalize metoda će biti izvršena. Finalize metoda na ovaj način produžava vek objektu i samim tim nepotrebno zauzima memoriju. Ako imamo veliki broj objekata koji se često alociraju i dealociraju i koji su pri tom veliki, možemo lako doći u situaciju da imamo veliki deo memorije zauzet a da zapravo naša aplikacije ne koristi aktivno tu memoriju.



Slika 4. Zauzimanje i oslobadanje resursa od strane GC

Dispose pattern

Da bi se unapredio ovaj mehanizam dealociranja objekata sa Heap-a koji imaju u sebi unmanaged resurse, predložen je efikasniji način čišćenja resursa. Dispose pattern predlaže da se za sve klase implementira Dispose metoda koja će počistiti alocirane unmanaged resurse i u isto vreme briše referencu iz Finalization queue. Metoda se poziva ručno na mestu gde objekat više nije potreban i gde resursi mogu biti oslobođeni.

```
public void Dispose()
{
    CleanUp(true);
    GC.SuppressFinalize(this);
}

private void CleanUp(bool disposing)
{
    if (disposing)
    {
        // Thread-specific code goes here
    }
}
```

Klasa u kojoj se implementira Dispose patter bi trebala da implementria interfejs IDisposable. Pravilna implementacija Dispose pattern je prikazana u listingu ispod.

```
public class UnmanagedWrapper : IDisposable
{
    private bool disposed = false;
    public UnmanagedWrapper()
    {
        // creates the unmanaged resource...
    }
    ~UnmanagedWrapper()
    {
        Dispose(false);
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            // Free the unmanaged resource anytime.
            if (disposing)
            {
                // Free any other managed objects here.
            }
            disposed = true;
        }
    }
```

Dispose pattern je praktično obavezan alat u radu sa unmanaged resursima. Finalize metodu treba implementirati na nacin da se poziva CleanUp jer će na taj način resursi biti počišćeni iako se u kodu izostavi eksplicitno pozivanje Dispose metode (dodatna sigurnost).

Zadatak 1 - Implementacija Dispose patterna

Napraviti klasu koja implementira IDisposable interfejs i Dispose pattern i služi za manipulaciju tekstualnim fajlovima.

Ova klasa treba da ima polje putanja, kao i polja za TextWriter i TextReadr koja će posle biti oslobođena prilikom implementacije Dispose patterna. Klasa poseduje konstruktora koji sadrži parametar putanju, ova putanja ne može da se menja tokom zivota jednog objekta, ali u svakom momentu moze da se dođe do informacije koja je njena vrednost.

Ova klasa treba da ima metode za dodavanje teksta na kraj fajla, brisanje celog teksta iz fajla, funkciju za čitanje celog teksta iz fajla i metodu koja prima neograničen broj parametara i broji pojave u tekstu, pod pojavama se podrazumevaju da se prosledjeni parametri poklapaju sa celom rečju.

U okviru klase Program.cs napraviti meni koji nudi korisniku mogućnost da vrši manipulaciju sa fajlom čije ime navede sa mogućnošću poziva svih metoda koje su gore navedene, kao i mogucnost da završi manipulaciju nad fajlom, prilikom završetka manipulacije nad određenim fajlom neophodno je pozvati metodu za cišćenje resurs i omogućiti korisniku ponovno unošenje putanje ukoliko želi da radi nad drugim fajlom ili završetak programa. Svaku od metoda pozivati u okviru try-cath bloka.

Vezbe 4 - Rad sa fajlovima

Biblioteka koja sadrži sve neophodno za manipulaciju nad fajlovima i strimovima (čitanje, pisanje) je *System.IO Namespace.*

Klasa File and FileInfo

Klasa *File* pruža statičke metode za kreiranje, kopiranje, brisanje, premeštanje i otvaranje datoteka. Pomaže u kreiranju *FileStream* objekata. Ova klasa se koristi ukoliko je neophodno da izvršite neku operaciju nad fajlom u datom momentu, pored toga pruža mogućnost izmene informacija o samom datumu kreiranja fajla.

Neke od metode su:

- **ReadAllText** - otvara tekstualnu datoteku i celokupan tekst iz nje učitava u string
- **ReadAllLines** - otvara tekstualnu datoteku i iz nje učitava sve redove u niz stringova.
- **ReadAllBytes** - otvara binarnu datoteku i iz nje učitava sadržaj u niz bajtova
- **ReadLines** - čita pojedinačne redove tekstualne datoteke i upisuje ih u niz stringova. Metoda pogodna za velike tekstualne datoteke.
- **WriteAllText** - kreira novu tekstualnu datoteku i upisuje u nju tekstualni sadržaj.
- **WriteAllLines** - kreira novu datoteku i upisuje u nju jedan ili više stringova
- **WriteAllBytes** - kreira novu binarnu datoteku i upisuje u nju prosleđeni niz bajtova
- **AppendAllText** - otvara datoteku i dodaje tekst na kraj datoteke
- **AppendAllLines** - ovara datoteku i dodaje sve redove na kraj datoteke
- **Open -** otvara FileStream na specifičnoj putanju
- **Copy** - kopira postojeći fajl u novi fajl
- **Create** - kreira ili ukoliko fajl postoji prepiše ga
- **Delete** - briše fajl

Klasa *FileInfo* pruža informacije o datotekama. Omogućava kreiranje, kopiranje, brisanje, premeštanje i otvaranje datoteka. Podržava kreiranje *FileStream* objekata.

Neke od metoda su:

- **AppendText** - kreira StreamWriter koji dodaje tekst u fajl
- **Open** - otvara fajl
- **CopyTo** - kopira postojeći fajl u novi fajl
- **Create** - kreira fajl
- **Delete** - trajno briše fajl

Klasa *File* se koristi kada je neophodno izvršavati samo jednu operaciju nad fajlom, dok se klasa *FileInfo* koristi ukoliko treba da se izvršava više operacija. Još jedna od osnovnih razlika jeste ta da kada želimo da koristimo klasu *FileInfo* neophodno je da napravimo objekat te klase i onda nad tim objektom da pozivamo metode, dok je klasa *FIle* statička i ne postojivmogućnost kreiranja njene instance.

Klasa Directory i DirectoryInfo

Klasa *Directory* sadrži statičke metode za kreiranje, premeštanje i prolazak kroz direktorijume i poddirektorijume.

Neke od metoda su :

- **CreateDirectory** - kreira direktorijum sa poddirektorijumima
- **Delete** - briše direktorijum
- **Exists** - vrši proveru da li direktorijum postoji
- **Move** - vrši pomeranje direktorijuma sa jedne destinacije na drugu
- **GetDirectories** - vraća informaciju o imenima direktorijuma kao i njihovim putanjama
- **GetFiles** - vraća informaciju o nazivima fajlova kao i njihovim putanjama

Klasa *DirectoryInfo* pruža informacije o direktorijumima. Omogućava kreiranje, premeštanje i prolazak kroz direktorijume i poddirektorijume.

Neke od metoda su:

- **Create** - kreira direktorijum sa poddirektorijumima
- **Delete** - briše direktorijum
- **Exists** - vrši proveru da li direktorijum postoji
- **MoveTo** - vrši pomeranje direktorijuma sa jedne destinacije na drugu
- **GetDirectories** - vraća informaciju o imenima direkrtorijuma kao i njihovim putanjama
- **GetFiles** - vraća informaciju o nazivima fajlova kao i njihovim putanjama

Razlika između ove dve klase je u tome što *DirectoryInfo* zahteva kreiranje objekta nad kojim će se vršiti pozivanje metoda dok *Directory* predstavlja statičku klasu koja ne može da se instancira. Ukoliko nad nekim resursom treba da se izvršavaju različite operacije više puta preporuka je da se koristi *DirectoryInfo*.

Tokovi podataka (Streams)

Stream-ovi su nizovi bajtova koji se mogu koristiti za upisivanje i čitanje iz različitih vrsta skladišta podataka.

Apstraktna klasa *Stream* podržava čitanje i pisanje bajtova. Sve klase koje implementiraju stream-ove nasleđuju klasu *Stream*. Klasa *Stream* i njene izvedene klase pružaju zajednički pogled na izvore podataka i spremišta. Klasa *Stream* i njene izvedene klase apstrahuju detalje operativnog sistema i hardware-a. Prednosti stream-ova je u inkrementalnom procesiranju podataka, apstrakcija rada sa skladištima podataka kao i fleksibilnosti.

Klasa *Stream* implementira *IDisposable* interfejs. Pošto koristi eksterne resurse, stream se čisti garbage collector-om, te ga je potrebno ručo počistiti korišćenjem *Dispose*() metode.

Klase StreamReader i StreamWriter

Klasa *StreamWriter* nasleđuje apstraktnu klasu *TextWriter* služi za upsivanje karaktera u Stream u određenom enkodingu.

Neke od metoda su :

- **Write** - Upisuje string u stream

- **WriteLine** - Upisuje string u stream i dodaje karakter za novi red na kraju
- **Close** - Zatvara stream
- **Dispose** - Oslobađa se resursa

Klasa *StreamReader* nasleđuje apstraktnu klasu *TextReader* i služi za čitanje karaktera iz *Stream*-a u odredjenom enkodingu.

Neke od metoda su:

- **Peek** - Vraca prvi sledeći karakter
- **Read** - Čita sledeći karakter u stream-u
- **ReadLine** - Čita sledeću liniju i vraća je kao string
- **ReadToTheEnd** - čita karaktere od određene pozicije do kraja

Klasa FileStream

FileStream klasa implementira stream za datoteku, podržavajući operacije čitanja i pisanja u datoteku i iz datoteke.

Neke od metoda su :

- **BeginRead** - operacija čitanja u asinhronom režimu
- **BeginWrite** - operacija pisanja u asinhronom režimu
- **Read**- metoda pomoću koje se vrši čitanje niza bajtova iz fajla
- **Write** - metoda pomocu koje se vrši pisanje niza bajtova u fajl
- **Close** - metoda koja zatvara stream

U donjem listingu je dat primeri kako se koristi *FileStream* prilikom čitanja i pisanja bajtova u fajl.

```
using (FileStream fs = File.Create(path))
{
    string line = "Some text for writing in file";
    byte[] lineInBytes = new UTF8Encoding(true).GetBytes(line);
    fs.Write(lineInBytes, 0, lineInBytes.Length);
}

using (FileStream fs = File.OpenRead(path))
{
    byte[] buffer = new byte[1024];
    UTF8Encoding temp = new UTF8Encoding(true);
    fs.Read(buffer, 0, buffer.Length)
    temp.GetString(buffer, 0, buffer.Length);
}
```

Klasa MemoryStream

MemoryStream implementira stream za memoriju, podržavajući operacije čitanja i pisanja u memoriju i iz memorije.

Neke od metoda su :

- **BeginRead** - operacija čitanja u asinhronom režimu
- **BeginWrite** - operacija pisanja u asinhronom režimu
- **Read**- metoda pomoću koje se vrši čitanje iz fajla
- **Seek** - metoda sa kojom se vrši pozicioniranje na određeni deo fajla
- **WriteTo** - metoda pomocu koje se vrši pisanje u fajl
- **Close** - metoda koja zatvara stream

U donjem listingu je dat primer upisa podataka bajt po bajt u *MemoryStream*, kao i samo čitanje iz *MemoryStream*-a.

```
UnicodeEncoding uniEncoding = new UnicodeEncoding();
byte[] textInBytes = uniEncoding.GetBytes(text);

using (MemoryStream memStream = new MemoryStream())
{
    // Write string to the stream byte by byte.
    int count = 0;
    while (count < textInBytes.Length)
    {
        memStream.WriteByte(textInBytes[count++]);
    }
}

using (MemoryStream memStream = new MemoryStream())
{
    byte[] byteArray = new byte[memStream.Length];
    int count = memStream.Read(byteArray, 0, byteArray.Length);
    char[] charArray = new char[uniEncoding.GetCharCount(byteArray, 0, count)];
    uniEncoding.GetDecoder().GetChars(byteArray, 0, count, charArray, 0);
}
```

Zadatak 1 - Rad sa direktorijuma i fajlovima

Napraviti program koji služi za manipulaciju sa direktorijumimam i fajlovima i ima sledeće metode :

- **AddDirectory(string path)** - metoda za dodavanje direktorijuma
- **RemoveDirectory(string path)** - metoda za uklanjanje direktorijuma
- **GetListOfAllFiles(string path)** - metoda koja vraca spisak svih fajlova u odredjenom direktorijumu
- **CreateEmptyFiles(string directoryPath, params string[] fileNames)** - metoda koja kreira fajlove na određenoj putanji, pomoću ove metode može da se kreira neogranicen broj fajlova
- **RemoveFile(string path)** - metoda koja uklanja fajl
- **ReadFile(string path)** - metoda koja vraća ceo sadržaj fajla (koristiti klasu StreamReader)
- **AddTextToFile(string path, string text)** - metoda koja dodaje tekst u fajl (koristiti klasu StreamWriter)

Sve ove metode testirati u okviru klase Program.cs

Zadatak 2 - File stream

U okviru Programa.cs napraviti funkcionalnost čitanja i pisanja u fajl koristeći *FileStream* klasu. Na početku pokretanja programa korisnik treba da unese putanju fajla koji treba da kreira, nakon toga unosi tekst koji želi da se nađe u fajlu. Mogućnost završetka unošenja teksta se korisniku omogućavata na taj način da unese END u novi red. Kada je korisnik uneo sav tekst neophodno je na konzolu ispisati sav tekst koji se nalazi u fajlu.

Zadatak 3 - Memory stream

U okviru Programa.cs koristeći *MemoryStream* klasu serijalizovati teks koji je korisnik uneo na konzolu i upisivati ga bajt po bajt u objekat *MemoryStream*-a, potom serijalizovati niz bajtova u string i ispisati ga na konzolu. Sve ovo uraditi u jednom *using*-u odnosno kreirati samo jedan objekat *MemoryStream*-a.

Vezbe 5 - Rad sa fajlovima preko mreže

Sistem za upravljanje bazom podataka (DBMS) predstavlja softverski alat koji korisnicima omogućava da lako upravljaju bazom podataka. Ovakvi sistemi imaju mogućnost čuvanja datoteka u tabelama, najćešće korišćenjem BLOB (binary large object) tipova. Ovakvi objekti mogu biti do reda veličine 2GB.

Prilikom slanja podataka preko mreže neophodno je fajl serijalizovati i tako ga preneti preko mreže. MemoryStream klasa pruža mogućnost konverzije fajlova u bajtove i obrnuto.

WCF i MemoryStream

Parametri koji se prosleđuju kroz WCF i sadrže MemoryStream objekte treba sami da implentiraju IDisposable interfejs, gde Dispose metoda treba da se oslobodi svih resursa koji se više ne koriste.

Kada neka od metoda interfejsa sadrži parametar koji implementira *IDisposable* interfejs preko atributa *OperationBehaviour* možemo da naglasimo da li će resurse koji se prosleđuju kroz parametre oslobađati server. Ukoliko je vrednost podešena na *true*, što je i podrazumevana vrednost onda računamo da hoće. Ovaj atribut se dodaje iznad implementacije metode interfejsa.

```
[OperationBehavior(AutoDisposeParameters = true)]
```

Zadatak - manipulacija fajlovima preko mreže

Neophodno je napraviti klijent-server arhitekturu.

Deljeni interfejs treba da sadrži metode za slanje fajla i za dobavljanje fajlova.

```
FileManipulationResults SendFile(FileManipulationOptions options);
FileManipulationResults GetFiles(FileManipulationOptions options);
```

U okviru deljene biblioteke treba da se nađu i klase *FileManipulationResults* i *FileManipulationOptions*, obe ove klase treba da implementiraju interface *IDisposable*. Klasa *FileManipulationOptions* sadrži polja memoryStream (tipa *MemoryStream*) i keyWord (tipa *string*). Klasa *FileManipulationResults* sadrži polja resultType(tipa *enum ResultType* koja može da ima vrednosti *Success*, *Warning*, *Failed*), resultMessage (tipa *string*) i memoryStremCollection (tipa *Dictionary<string, MemoryStream>*).

U listingu dole je prikazano na koji način se definiše varijabla u konfiguracionom fajlu.

```
<appSettings>
    <add key="path" value="Files"/>
</appSettings>
```

U listingu dole je prikazano na koji način se pristupa varijabli iz koda.

```
var fileDirectoryPath = ConfigurationManager.AppSettings["path"];
```

Metode interfejsa je neophodno impelentirati sa strane sarvisa tako da:

SendFile - Na putanju koja je definisana u konfiguracionom fajlu treba upisati prosleđen fajl, treba voditi računa da ukoliko putanja nije definisana ili je prosleđen prazan fajl servisa vraća upozorenje sa porukom, ukoliko se uspešno sačuva fajl šalje se informacija da je fajl uspešno sačuvan, dok ukoliko je došlo do neke greške vraća se informacija da operacija nije uspešno izvršena.

GetFiles - Za putanju koja je definisana u konfiguracionom fajlu treba isčitati sve fajlove koji počinju sa ključnom rečju koja je prosleđena kroz parametar, treba voditi računa da ukoliko putanja nije definisana servis vraća upozorenje sa porukom, ukoliko se uspešno isčitaju fajlovi šalje se informacija da je operacija uspešno izvršena, dok ukoliko je došlo do neke greške vraća se informacija da operacija nije uspešno izvršena.

Sa strane klijenta neophodno je napraviti meni koji nudi mogućnost pozivanja metoda servisa, neophodno je obraditi izuzetak ukoliko se desi, prilikom vraćanja fajlova sa servisa neophodno je iste upisati u direktorijum.