PREDMET: Razvoj Programskih Rješenja	**NASTAVNI ANSAMBL**
AK. GOD. : 2017/2018 **RESURS:** Laboratorijska vježba 2 **DATUM OBJAVE:** 20.10.2017	**PROFESOR:** Dženana Đonko (ddonko@etf.unsa.ba) **ANSAMBL:** Hasić Haris, Hodžić Kerim Đenašević Emir, Hasanbašić Denis, Mahmutović Mustafa, Ramić Benjamin

Cilj vježbi:

- Upoznavanje sa interfejsima, delegatima, anonimnim funkcijama i lambda izrazima
- Upoznavanje sa kolekcijama u C#

Napomena: Prije izrade vježbe je obavezno pročitati predavanja vezana za tematiku vježbe i upoznati se sa osnovnim konceptima jezika C# i objektno orjentisanog programiranja.

LABORATORIJSKA VJEŽBA 2
Interfejsi, delegati, lamda izrazi u C#

Interfejsi

Interfejs je najbolje konceptualno zamisliti kao ugovor sa klasom. Ako klasa potpiše ugovor (deklariše da implementira interfejs), tada će ili klasa znati ponuditi određene usluge propisane ugovorom ili će biti prijavljeno kršenje ugovora (program se neće kompajlirati). Konkretno, interfejs je potpuna definicija atributa (Properties, ne polja ili Fields!) i metoda bez implementacije pa je sam za sebe sličan apstraktnoj klasi (nemoguće je instancirati interfejs). Štaviše, u nekim programskim jezicima se interfejs i implementira kao apstraktna klasa sa svim virtuelnim metodama. Ipak, iako dijele slične mehanizme, naslijeđivanje i korištenje interfejsa nisu jedna te ista stvar. Naslijeđivanjem klase B iz klase A tvrdimo da je svaka instanca klase B ujedno i instanca klase A (ne vrijedi obratno), tj. B **jeste** A (šta klasa **jeste**), dok s druge strane ako klasa B implementira interfejs A tvrdimo da klasa B, između ostalog, ima atribute (propertije) i metode koje svojim definicijama odgovaraju definiciji iz interfejsa, tj. B zna/može sve po specifikaciji A (šta klasa **zna**). Naravno, interfejs ne definira kako će klasa implementirati svaku od funkcionalnosti. Jedan od čestih primjera upotrebe interfejsa je polimorfno duboko kopiranje objekata gdje se definiranjem interfejsa iCloneable (interfejsi se često imenuju sa "iInterfejs") koji sadrži metodu clone() koja vraća Object omogućava svakoj klasi da implementira duboko kopiranje na sebi svojstven način. Kako je u C# zabranjeno višestruko naslijeđivanje, to je upotreba interfejsa jedan od načina prevazilaženja navedenog ograničenja. Također, interfejsi su jedan od načina za formiranje mehanizama dodavanja plug-ina ili novih dodataka u postojeću hijerarhiju klâsa, posebno kada se hijerarhiju izvozi kao biblioteku klâsa.

Primjer interfejsa je dat u nastavku:

```
interface IMojInterfejs
{
    void NekaMetoda();
}

class KlasaKojaImplementiraInterfejs : IMojInterfejs
{
    // Eksplicitna implementacija metode interfejsa:
    void IMojInterfejs.NekaMetoda()
    {
        // Implementacija metode
    }

    static void Main()
    {
        // Deklaracija instance interfejsa
        IMojInterfejs obj = new KlasaKojaImplementiraInterfejs();

        // Poziv implementirane metode
        obj.NekaMetoda();
    }

 }
```

Delegati, anonimne funkcije i lambda izrazi

Delegati predstavljaju reference na funkcije. Delegat definiše ključne osobine funkcije, a to su njen povratni tip i ulazni argumenti sa tipovima (tzv. function signature ili prototip). Delegat se ponaša kao tip - vrijednost koju može poprimiti bilo koja funkcija sa istim prototipom (signatureom). Na taj način je moguće proslijeđivati funkcije kao parametar drugim funkcijama. Ovo omogućava da se, naprimjer, programski mijenja poziv metode, ili da se ubaci novi kod u već postojeće klase.

Anonimne funkcije su funkcije koje imaju popis parametara i tijelo funkcije, ali nemaju imena. Lambda izrazi predstavljaju poseban vid matematske notacije anonimnih funkcija (definisanih inline) koje je moguće iskoristiti za kreiranje delegata. Na taj način je moguće lako proslijediti funkciju (npr. kriterija što ste susretali koristeći biblioteku <algorithm> u C++) kao parametar drugoj funkciji. Da bi se kreirao labda izraz potrebno je s lijeve strane lambda znaka `=>` naznačiti ulazne parametre ako ih ima, a sa desne strane izraz koji treba da se izvrši. Naprimjer izraz `x => x*x` specificira parametar x i vraća njegov kvadrat. Ovaj izraz je moguće dodijeliti delegatu na sljedeći način:

```
delegate int delegat(int i);
static void Main(string[] args)
{
    delegat mojDelegat = x => x * x;
    int j = mojDelegat(5); //j = 25
}
```

Tradicionalni način pisanja delegata bez lambda izraza je sljedeći:

```
public delegate void MojDelegat(string arg1, string arg2);

public static void pokreniTest()
{
    MojDelegat delegat = printajNaKonzolu;
    delegat("Ime", "Prezime");
}

public static void printajNaKonzolu(string s1, string s2)
{
    Console.WriteLine(s1);
    Console.WriteLine(s2);
}
```

Lambda način pisanja delegata je sljedeći:

```
public static void pokreniLambdaTest()
{
    MojDelegat delegat = (s1, s2) =>
    {
        Console.WriteLine(s1);
        Console.WriteLine(s2);
    };

    delegat("Lambda Ime", "Lambda Prezime");
}
```

Kolekcije u C#

.NET framework omogućava bogat set tipova podataka za rad i manipulaciju kolekcijama. Imenovani prostori vezani za kolekcije prikazani su sljedećom tabelom:

Namespace	Sadrži
`System.Collections`	Negeneričke klase i interfejsi za kolekcije
`System.Collections.Specialized`	Predefinirane negeneričke kolekcijske klase
`System.Collections.Generic`	Generičke klase i interfejsi za kolekcije
`System.Collections.ObjectModel`	Bazne klase za izvedene kolekcije
`System.Collections.Concurrent`	Kolekcije za rad sa threadovima

U nastavku su najprije objašnjeni klasični C# nizovi koji ne spadaju u `System.Collections` namespace-u. Nakon toga su prezentirane neke najvažnije klase za rad sa kolekcijama koje se nalaze u `System.Collections` i `System.Collections.Generic`.

Klasični nizovi (Array)

Klasični nizovi u C# programskom jeziku predstavljeni su klasom ***Array***. Ta klasa je implicitna bazna klasa za sve jednodimenzionalne i višedimenzionalne nizove. Ona omogućava unifikaciju tipova što svim nizovima bez obzira na način deklaracije i vrstu elemenata koju pohranjuju omogućava isti set metoda za manipulaciju nizovima i njhovim elementima. Definisana je u glavnom System namespace-u.

Klasa ***Array*** sadrži nekoliko pomoćnih metoda za manipulaciju nizovima. Neke od najkorištenijih metoda prezentirane su u sljedećoj tabeli:

Naziv metode	Opis
`Clear`	Elemente niza u definiranom rasponu postavlja na njihovu podrazumijevanu vrijednost (0, *false* ili *null*)
`CopyTo (Array, Int32)`	Kopira sve elemente iz izvornog niza u niz proslijeđen kao drugi parametar, počevši od specificiranog indeksa
`GetValue (Int32)`	Vraća elemenat jednodimenzionalnog niza na specificiranoj poziciji
`IndexOf (Array, Object)`	Vraća indeks elementa proslijeđenog kao drugi parametar u nizu proslijeđenom kao prvi parametar
`Reverse (Array)`	Postavlja elemente u obrnutom redosljedu u proslijeđenom nizu
`SetValue (Object, Int32)`	Postavlja specificirani element na poziciju proslijeđenu kao drugi parametar
`Resize(niz, Int32)`	Mijenja veličinu niza zadanog prvim parametrom u broj zadan u drugom paramteru
`Sort (Array)`	Sortira elemente niza korištenjem IComparable implementacije svakog elementa

Zanimljivo je da je većina metoda za rad sa nizovima zapravo statička u okviru klase Array, te je stoga prilikom traženja i korištenja takvih metoda neophodno obratiti pažnju na taj implementacijski detalj (iako postoji i nekoliko metoda i atributa koji su vezane za instancu niza). Najkorišteniji atributi niza su Length (dužina niza), Rank (dimenzionalnost niza), IsReadOnly (indikacija da je niz nepromjenjiv) itd. Način korištenja prethodno opisanih metoda dat je kroz sljedeći ***Primjer 1***:

```
        static void Main(string[] args)
        {
            int[] niz = { 34, 72, 13, 44, 25, 30, 10 };
            int[] temp = new int[niz.Length];
            niz.CopyTo(temp, 0);
            Console.Write("Originalni niz: ");
            foreach (int i in niz) Console.Write(i + " ");
            Console.WriteLine();
            // Reverse
            Array.Reverse(temp);
            Console.Write("Obrnuti niz: ");
```

```
            foreach (int i in niz) Console.Write(i + " ");
            Console.WriteLine();
            // Kopiranje
            Console.Write("Kopirani niz: ");
            foreach (int i in temp) Console.Write(i + " ");
            Console.WriteLine();
            // Sort
            Array.Sort(temp);
            Console.Write("Sortiran niz: ");
            foreach (int i in niz) Console.Write(i + " ");
            Console.WriteLine();
            // Clear
            Array.Clear(niz, 0, niz.Length);
            Console.Write("Clear niz: ");
            foreach (int i in niz) Console.Write(i + " ");
            Console.WriteLine();
            Console.Read();
        }
```

Primjer 1 između ostalog demonstrira i upotrebu metode za sortiranje elemenata niza. U slučaju cijelih brojeva, ova metoda podrazumijevano će sortirati elemente u rastućem poretku. No, šta ukoliko korisnik želi sortirati u opadajućem poretku ili želi sortirati elemente koji nisu prosti tip podatka kao što je cijeli broj, već korisnički definiran tip (klasa). Kao opcionalni parametar Sort metoda prima generički delegat ***Comparison<T> (public delegate int Comparison<T> (T x, T y))*** koji referencira metodu za poređenje dva objekta istog tipa. Ova metoda treba da vrati nulu u slučaju da su dva objekta jednaka, negativan cijeli broj u slučaju da je prvi manji od drugog i pozitivan cijeli broj u slučaju da je drugi manji od prvog. Prema tome, da bi se sortiranje niza cijelih brojeva iz primjera 1 izvršilo u opadajućem poretku potrebno je modifikovati poziv metode Sort proslijeđujući joj metodu kriterija sortiranja:

```
        private static int CompareInt(int x, int y)
        {
            return y - x;
        }

        Array.Sort(niz, CompareInt);
```

Prethodno prikazano može se skraćeno napisati instanciranjem delegata koji Sort metoda očekuje pomoću anonimne metode ili lambda izrazom:

```
        Array.Sort(niz, delegate(int x, int y) { return x - y; }); // Anonimna metoda
        Array.Sort(niz, ((x, y) => y - x)); // Lambda izraz
```

Po istom principu moguće je specificirati kriterij sortiranja i za složene tipove podataka, pri čemu funkcija kriterija kao parametre prima odgovarajući tip tog podatka, odnosno klasu. Ovakav način definisanja kriterija sortiranja koristi se i kod ostalih tipova kolekcija kao što su generičke liste i sl.

Više o klasi Array: https://msdn.microsoft.com/en-us/library/system.array(v=vs.110).aspx

Liste (List<T>)

Generička klasa List i negenerička klasa ArrayList su među najkorištenijim klasama za rad sa kolekcijama u C# programskom jeziku. Omogućavaju dinamičko mijenjanje veličine kolekcije na način da interno implementiraju klasični niz te nakon dosezanja njegovog maksimalnog kapaciteta ga zamjenjuju sa novim proširenim nizom. Ove klase intuitivno su jednostavne za koristiti i omogućavaju veliki broj pomoćnih metoda za njihovu manipulaciju.

Elementima liste može se pristupati preko indeksnog operatora analogno nizovima. Potrebno je paziti na to da liste koje sadrže referentni tip podataka nije moguće jednostavno kopirati bez iteriranja kroz listu i kopiranja (instanciranja) svakog individualnog elementa. Prilikom deklaracije i instanciranja nove liste nije neophodno specificirati njen incijalni kapacitet, koji u tom slučaju postaje 0 te se prilikom ubacivanja prvog novog elementa lista interno proširuje. Korisna mogućnost jeste proslijeđivanje niza u konstruktor liste, prilikom čega se lista popunjava elementima tog niza (koji će u slučaju referentnog tipa podatka biti samo reference na elemente niza).

Neke od najkorištenijih metoda za rad sa generičkim listama prezentirane su u sljedećoj tabeli:

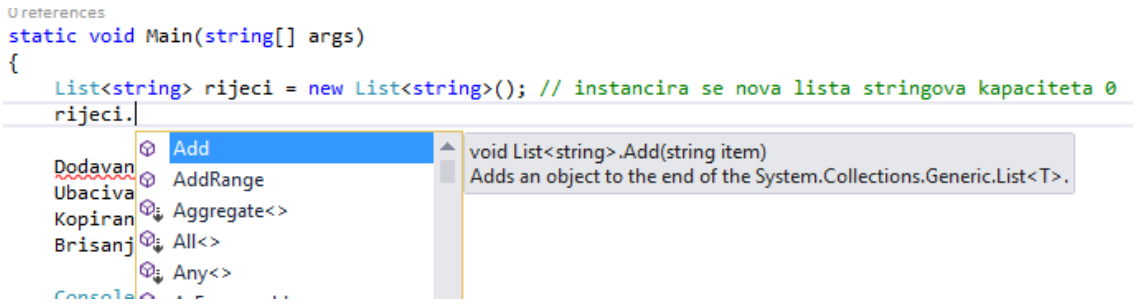
Naziv metode	Opis
`Clear`	Briše sve elemente liste
`Add`	Dodaje novi elemenat na kraj liste
`AddRange (Array)`	Dodaje kolekciju na kraj liste
`Insert (Int32)`	Ubacuje novi elemenat na specificiranu poziciju
`InsertRange (Array, Int32)`	Ubacuje kolekciju elemenata na specificiranu poziciju
`Remove`	Briše elemenat iz liste
`RemoveAt (Int32)`	Briše elemenat iz liste na specificiranoj poziciji
`Contains`	Provjerava da li specificirani elemenat postoji u listi
`IndexOf`	Vraća indeks elementa u listi (-1 ako element ne postoji)

U nastavku je dat ***Primjer 2*** koji demonstrira osnovne operacije (dodavanje, ubacivanje, brisanje) sa listom stringova.

```
        static void Main(string[] args)
        {
            List<string> rijeci = new List<string>(); // Nova lista stringova kapaciteta 0
            DodavanjeElemenata(rijeci);
            UbacivanjeElemenata(rijeci);
            BrisanjeElemenata(rijeci);
            Console.Read();
        }

        static void DodavanjeElemenata(List<string> rijeci)
        {
            rijeci.Add("test");
            rijeci.Add("asdf");
```

```
            // string dvijeRijeci = new string[] { "predzadnja", "zadnja" };
            // Na kraj liste se dodaju dvije riječi
            rijeci.AddRange(new string[] { "predzadnja", "zadnja" });
            foreach (string s in rijeci) // Iteracija kroz listu pomoću foreach petlje
                Console.Write(s + (rijeci.Last().Equals(s) ? Environment.NewLine : ", "));
        }

        static void UbacivanjeElemenata(List<string> rijeci)
        {
            rijeci.Insert(0, "prva"); // Nova riječ se ubacuje na prvu poziciju (indeks 0)
            // Dvije riječi se dodaju na drugu poziciju
            rijeci.InsertRange(1, new string[] { "druga", "treća" });
            for (int i = rijeci.Count - 1; i >= 0; i--) // Iteracija unazad pomoću for
                Console.Write(rijeci[i] + (rijeci.First().Equals(rijeci[i]) ?
Environment.NewLine : ", "));
        }

        static void BrisanjeElemenata(List<string> rijeci)
        {
            rijeci.Remove("prva"); // Brisanje elementa 'prva' iz liste
            rijeci.RemoveRange(0, 2); // Brisanje prva dva elementa iz liste
            foreach (string s in rijeci)
                Console.Write(s + (rijeci.Last().Equals(s) ? Environment.NewLine : ", "));
            rijeci.Clear(); // brisanje svih elemenata iz liste
            Console.WriteLine("Ukupno elemenata u listi: " + rijeci.Count);
        }
```

Radi jednostavnije analize rada prikazanog kôda koristiti debugging opcije za zaustavljanje izvršenja na kontrolnim tačkama (koje mogu biti tri metode koje se pozivaju u glavnom programu) i pratiti ispis u konzoli prilikom izvršenja svake od metoda. Uvid u sve raspoložive metode nad instancom generičke liste moguće je (kao i za sve druge tipove) dobiti kroz IntelliSense pozicioniranjem na instancirani objekat liste ili pritiskom Ctrl + .



Primjer 2 demonstrirao je neke od najosnovnijih operacija nad listama korištenjem stringa kao tipa podatka koji se pohranjuje u listi. Ekvivalentno tome moguće je u listu smjestiti bilo koji tip podatka. Sljedeći primjer demonstrira rad sa listama čiji su elementi klase. Ovaj primjer obuhvata nekoliko bitnih pojedinosti koje su u nastavku obrazložene. Implementirane su dvije jednostavne klase, Osoba i Glumac, pri čemu klasa Glumac nasljeđuje klasu Osoba. U okviru klase Osoba nalazi se virtuelna metoda za ispis, koja je reimplementirana u klasi Glumac. U nastavku je prikazana struktura korištenih klasa.

Primjer 3

```
    class Osoba
    {
        private string _Ime, _Prezime;
        private int _Starost;
        public int Starost
        {
            get { return _Starost; }
            set { _Starost = value; }
        }
        public string Prezime
        {
            get { return _Prezime; }
            set { _Prezime = value; }
        }
        public string Ime
        {
            get { return _Ime; }
            set { _Ime = value; }
        }
        public virtual void Ispisi()
        {
            Console.WriteLine(_Ime + " " + _Prezime + "  " + _Starost + " god");
        }
    }

    class Glumac : Osoba
    {
        private List<string> _TopFilmovi;
        public List<string> TopFilmovi
        {
            get { return _TopFilmovi; }
            set { _TopFilmovi = value; }
        }
        public override void Ispisi()
        {
            Console.Write(Ime + " " + Prezime + "  " + Starost + " god ||| TOP 3: ");
            if (_TopFilmovi == null)
            {
                Console.WriteLine();
                return;
            }
            foreach (string s in _TopFilmovi)
                Console.Write(s + (_TopFilmovi.Last().Equals(s) ? Environment.NewLine : ",
"));
        }
    }
```

U nastavku je prikazan ostatak primjera 3 koji predstavlja glavni program u okviru kojeg se vrše određene manipulacije sa generičkom listom osoba. Nakon ovog primjera dato je kratko objašnjenje najbitnijih dijelova.

```
    class Program
    {
        static void Main(string[] args)
        {
            List<Osoba> osobe = new List<Osoba>();
            osobe.Add(new Osoba() { Ime = "Imenko", Prezime = "Prezimenko", Starost = 23
});
            osobe.Add(new Glumac()
            {
                Ime = "Jim",
                Prezime = "Carter",
                Starost = 52,
```

```
                TopFilmovi = new List<string>() { "The Mask", "Dumb and Dumber", "Ace
Ventura" }
            });
            osobe.Add(new Glumac()
            {
                Ime = "Matthew",
                Prezime = "McConaughey",
                Starost = 45,
                TopFilmovi = new List<string>() { "Interstellar", "Dallas Buyers Club",
"Magic Mike" }
            });
            osobe.Add(new Glumac()
            {
                Ime = "Denzel",
                Prezime = "Washington",
                Starost = 60,
                TopFilmovi = new List<string>() { "The Equalizer", "Man on Fire", "Safe
House" }
            });
            Console.WriteLine("Osobe sortirane po starosti: ");
            osobe.Sort((a, b) => a.Starost - b.Starost);
            foreach (Osoba o in osobe)
                o.Ispisi(); // pozvat će se metoda Ispisi() iz naslijeđene klase ukoliko je
override-ana
            Console.WriteLine();
            Console.WriteLine("Osobe mlađe od 60 godina: ");
            foreach (Osoba o in osobe.Where(o => o.Starost < 60))
                o.Ispisi();
            Console.WriteLine();
            Console.WriteLine("Prosječan broj godina: " + osobe.Average(o => o.Starost));
            Console.WriteLine("Broj osoba mlađih od 60: " + osobe.Count(o => o.Starost <
60));
            Console.WriteLine("Najduže prezime: " + osobe.Max(o => o.Prezime));
            Console.WriteLine();
            // neophodno je eksplicitno pretvoriti bazni objekat u izvedeni da bi se
pristupilo njegovim metodama
            foreach (Osoba o in osobe)
            {
                Glumac g = o as Glumac;
                if (g != null)
                    Console.WriteLine(g.Ime + " " + g.Prezime + ": " +
g.TopFilmovi.FirstOrDefault());
            }
            Console.Read();
        }
    }
```

Posebnu pažnju potrebno je obratiti na nekoliko elemenata u ovom primjeru. Dodavanje objekta u listu može se vršiti na nekoliko načina. Osnovni način je deklaracija i instanciranje novog objekta i popunjavanje vrijednosti njegovih atributa (pomoću konstruktorske ili inicijalizacijske sintakse), te zatim korištenje metode Add nad listom kako bi se instancirani objekat dodao. U gornjem primjeru se spomenuto instanciranje i inicijaliziranje objekta vrši unutar metode Add.

Prikazano je korištenje nekoliko ekstenzijskih metoda nad listom, kao što su Sort, Where, Count, Average, Max itd. Ove, ali i mnoge druge ekstenzijske metode dio su LINQ mogućnosti .NET frameworka predstavljenih prvobitno 2008. godine.

Iteracija kroz generičku listu vrši se klasičnom foreach petljom. Obzirom da se u ovom slučaju radi o hijerarhiji klasa (lista sadrži objekat tipa Osoba kao i objekat tipa Glumac koji nasljeđuje klasu Osoba) tokom iteracije je neophodno pretvoriti svaki objekat liste u bazni tip, u ovom slučaju Osoba. Tada polimorfizam omogućava da se prilikom izvršenja ovog koda pozove metoda Ispisi iz one klase kakva je izvorno u samoj listi. Ovakva iteracija omogućava i eksplicitno pozivanje određenih metoda i atributa specifičnih za neku klasu. Primjerice, klasa Glumac sadrži vlastitu listu filmova i da bi se pristupilo toj listi iteracijom kroz sve elemente glavne liste neophodno je pretvoriti objekat Osoba u Glumac. Ovakvo eksplicitno pretvaranje je moguće samo ukoliko je dati objekat zbilja tipa Glumac, što u gornjem primjeru nije slučaj za prvi elemenat liste. Stoga se može iskoristiti "as" operator koji će vratiti vrijednost null u slučaju da konverzija nije moguća.

Upcasting i downcasting su dva bitna pojma vezana za polimorfizam. Upcasting je proces pretvaranja izvedenog objekta u općenitiji bazni objekat (konverzija iz izvedene klase u baznu). Downcasting predstavlja obrnut proces pretvaranja baznog objekta u izvedeni (baznu klasu u izvedenu). U prethodnom primjeru 4 tokom iteracije kroz listu osoba vrši se upcasting svakog objekta u tip Osoba, jer objekti tipa Glumac nasljeđuju klasu Osoba. S druge strane, downcasting objekta tipa Osoba u specifičniji tip Glumac nije moguće uvijek, obzirom da se bazni tip Osoba ne može pretvoriti u naslijeđeni tip Glumac ukoliko se radi o konkretnoj instanci klase Osoba, te je u tom slučaju neophodno izvršiti eksplicitno provjeru tipa prije downcastinga ili koristiti operator "as" kao u primjeru 4, jer on ne baca izuzetak u slučaju greške.

ArrayList

Manje korištena kolekcija objekata u odnosu na generičke liste je ArrayList. Za razliku od generičke liste koja se instancira specificirajući tip elemenata koji će biti pohranjivani u njoj, ArrayList može primiti bilo koji tip podatka. Mana ovakvog pristupa je što programer mora pažljivo manipulirati sa vrstom svakog elementa u listi i provjeravati kojeg je on tipa. Pored toga, ArrayList nosi značajni performansni deficit u odnosu na generičke liste, ponajprije zbog činjenice da je svaki elemenat liste implicitno pretvoren u tip Object (koji je ujedno bazni tip za sve referentne tipove podataka u .NET frameworku) što podrazumijeva proces boxinga. Iz toga je jasno da je ArrayList zapravo ekvivalentan listi sa Object tipom podataka (List<Object>). Primjer jednostavne ArrayListe sa nekoliko različitih tipova podataka dat je u sljedećem ***Primjeru 4***:

```
class Program
{
    static void Main(string[] args)
    {
        ArrayList lista = new ArrayList();
        lista.Add("Beast");
        lista.Add(666);
        lista.Add("Eddie");
        lista.Add(0.001);
        lista.Add('c');

        foreach (Object o in lista)
            Console.WriteLine(o);

        Console.Read();}}
```

Dictionary i Tuple

Dictionary, odnosno riječnik, je kolekcija podataka čiji su objekti parovi ključ/vrijednost. Riječnik se najčešće koristi zbog brzih performansi pretraživanja i prilikom rada sa sortiranim listama. .NET framework nudi nekoliko vrsta klasa za rad sa riječnicima, no ovdje će se demonstrirati način rada sa najkorištenijom klasom od tih, a to je generička klasa Dictionary<TKey, TValue> koja je interno implementirana kao hash tabela. Za demonstraciju rada Dictionary klase može poslužiti klasa Glumac iz prethodnih primjera. Sljedeći ***Primjer 5*** pokazuje način na koji se Dictionary instancira specificirajući tip ključa (u ovom slučaju string) i tip objekta koji se pohranjuje (klasa Glumac).

```
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<string, Glumac> glumci = new Dictionary<string, Glumac>();
            glumci.Add("Jimmy", new Glumac()
            {
                Ime = "Jim",
                Prezime = "Carrey",
                Starost = 52,
                TopFilmovi = new List<string>() { "The Mask", "Dumb and Dumber", "Ace
Ventura" }
            });
            glumci.Add("Matt", new Glumac()
            {
                Ime = "Matthew",
                Prezime = "McConaughey",
                Starost = 45,
                TopFilmovi = new List<string>() { "Interstellar", "Dallas Buyers Club",
"Magic Mike" }
            });
            Glumac g;
            if (glumci.TryGetValue("Jimmy", out g))
                g.Ime = "Jimmy";
            foreach (string key in glumci.Keys)
                Console.WriteLine(key);
            foreach (Glumac value in glumci.Values)
                value.Ispisi();
            Console.Read();
        }
    }
```

Ključ riječnika u datom primjeru predstavlja string sa nadimkom glumca. Prilikom pretraživanja riječnika preporučljivo je koristiti metodu TryGetValue koja prima vrijednost ključa čije postojanje je neophodno provjeriti kao i referencu na objekat u koji će se pohraniti vrijednost elementa za dati ključ. Ova metoda je optimizirana da vrši provjeru i dodijeljivanje vrijednosti u jednom prolazu što omogućava maksimalno efikasan rad sa riječnikom. Iteracija kroz sve ključeve ili vrijednosti iz riječnika moguća je pomoću *foreach* petlje nad atributima Keys i Values, no za iteraciju kroz kompletnu strukturu riječnika preporučljivo je koristiti generičku klasu KeyValuePair (foreach (KeyValuePair<T1, T2> pair in dictionary)). U situacijama kada se ključ riječnika sastoji iz više podataka (npr. String i Integer) moguće je definisati riječnik u riječniku, gdje će ključ prvog riječnika biti string, a njegova vrijednost novi riječnik čiji je ključ Integer. Ipak, u ovakvim situacijama jednostavnije je koristiti Tuple.

Tuple je klasa koja predstavlja generičku kolekciju nepromjenjivih podataka, pri čemu svaki podatak može biti prozvoljnog tipa. Time se omogućava pohranjivanje različitih podataka bez potrebe za njihovom enkapsulacijom u klasu, u situacijama kada to naravno nije potrebno. Vrlo je korisna za slanje i preuzimanje više podataka kroz jedan parametar. Tuple je ograničen na maksimalno 8 elemenata, no definiranjem bilo kojeg elementa kao novi Tuple moguće je taj kapacitet proširivati za novih 8 elemenata. Svakom elementu pristupa se preko atributa ItemX, gdje je X bilo koji broj od 1 do 8. Primjer instanciranja Tuple-a sa dva elementa u njegova upotreba kao ključ Dictionary-a dati su u sljedećem ***Primjeru 6***:

```
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<Tuple<string, int>, Glumac> glumci = new Dictionary<Tuple<string,
int>, Glumac>();
            Tuple<string, int> kljuc1 = Tuple.Create<string, int>("Jimmy", 1);
            Tuple<string, int> kljuc2 = Tuple.Create<string, int>("Matt", 2);
            glumci.Add(kljuc1, new Glumac()
            {
                Ime = "Jim",
                Prezime = "Carrey",
                Starost = 52,
                TopFilmovi = new List<string>() { "The Mask", "Dumb and Dumber", "Ace
Ventura" }
            });
            glumci.Add(kljuc2, new Glumac()
            {
                Ime = "Matthew",
                Prezime = "McConaughey",
                Starost = 45,
                TopFilmovi = new List<string>() { "Interstellar", "Dallas Buyers Club",
"Magic Mike" }
            });
            Glumac g;
            if (glumci.TryGetValue(Tuple.Create<string, int>("Jimmy", 1), out g))
                g.Ime = "Jimmy";
            foreach (Tuple<string, int> key in glumci.Keys)
                Console.WriteLine(key.Item1 + " " + key.Item2);
            foreach (Glumac value in glumci.Values)
                value.Ispisi();
            foreach (KeyValuePair<Tuple<string, int>, Glumac> pair in glumci)
                pair.Value.Ispisi();
            Console.Read();
        }
    }
```

Interesantno je da se prilikom poziva metode TryGetValue kao ključ može proslijediti nova instanca Tuple objekta, jer se prilikom poređenja jednakosti dva Tuple objekta vrši najprije poređenje broja i tipova elemenata, a potom i jednakost vrijednosti svakog individualnog elementa. Takvo ponašanje, odnosno poređenje Tuple objekata po vrijednosti specifično je za samo Tuple klasu, iako se radi o referentnom tipu podatka.

Delegati i lambda izrazi u kolekcijama

Česta primjena lambda izraza u C# je u svrhu skraćivanja raznih operacija nad kolekcijama objekata. Analizirajmo sljedeću liniju koda:

```
Klijenti.Single(klijent => klijent.Id == id);
```

Single je ekstenzijska metoda nad kolekcijom (u ovom slučaju listom). Single uzima kao svoj parametar lambda izraz koji mapira objekat tipa sadržanog u kolekciji (u ovom slučaju je to neki klijent) kojeg identificira kao varijablu "klijent" u izraz koji evaluira u tačno ili netačno. Vidimo da slobodno možemo koristiti atribute objekta tipa Klijent u okviru poređenja, a također lambda izraz "vidi" i varijablu koja je u vidokrugu kôda u kojem deklarišemo lambda izraz. Single će u konkretnom slučaju vratiti prvi objekat iz kolekcije koji zadovolji uvjet tj. lambda se evaluira u true. Pored single postoji veći broj već implementiranih ekstenzijskih metoda nad kolekcijama koje programerima omogućavaju da u malom broju linija kôda riješe česte probleme poput traženja prvog elementa koji zadovoljava uvjet, pronalaska svih objekata koji zadovoljavaju uvjet, konverzije svih objekata u neki drugi tip i slično.

Kreiranje i korištenje .dll file

DLL (Dynamically linked library) je dinamički linkovana biblioteka koja se povezuje sa programom koji je koristi u trenutku izvršavanja programa. Na ovaj način je moguće kreirati kreirati jednu ili više klasa kao Class library i ponovo ih iskorištavati dodajući ih u druge aplikacije. Također, na taj način kao developeri dijela koda možemo taj kod dati drugima na korištenje, ali ne dajemo im i mogućnost izmjene istog.

Način kreiranja:

1. Prilikom kreiranja kao vrstu projekta odabrati *Class library*.
2. Implementirati logiku klase/a
3. Uraditi *build solutiona*
4. Potražiti *dll file* u imeaplikacije/debug/release/ime.dll

Način korištenja:

1. Kreirati novi projekat
2. Dodati *dll file* kao referencu u projekat (References->Add reference->odaberite *dll file*)
3. Ako se radi o projektu sa drugačijim namespaceom potrebno ga je uključiti sa *using imenaspacea*

Zadaci za samostalan rad i vježbu

Zadatak 1

Banka iz Vježbe 1 – Primjer 1 je proširila svoj biznis i sada omogućava otvaranje i štednih računa za korisnike (pored tekućih računa). Štedni račun sadrži dodatnu informaciju o valuti. Tu nije kraj promjenama - predstavnici banke očekuju da će se uskoro omogućiti i pravnim licima da koriste sve usluge banke kao i fizička lica. Pravno lice opisano je nazivom i adresom. Vaš zadatak je preuzeti do sada urađeno programsko rješenje (solution) sa stranice predmeta i prilagoditi klase u oba imenska prostora tako da podržite sve nove zahtijeve. Prodiskutujte rješenja sa asistentima.

Zadatak 2

Banka iz Vježbe 1 – Primjer 1 je zadovoljna izmjenama koje ste napravili kroz prethodni zadatak. Novi zahtijev je da omogućite logiranje (trajno bilježenje) svake akcije promjene stanja na računu. Pri tome banka napominje da još nisu odlučili koji logger će koristiti i da li će podatke snimati u bazu podataka, flat file u tekstualnom ili XML formatu, tako da je sve otvoreno. Kako bilo, žele da vaš codebase bude spreman. Prilagodite klase za bankovne račune da kao tekstualnu poruku zapišu sve relevantne informacije u logger. Za sada možete simulirati logiranje kroz vaš ispis u konzolu (DummyLogger).
Hint: Dobro razmislite o dizajnu loggera. Ne znate ništa o njemu osim da ima jednu funkcionalnost i da mu se šalje tekstualna poruka. Vjerovatno će vam trebati više klâsa da napravite sve kako treba. Trebati će vam neka vrsta klase za globalno upravljanje loggerima i klasa kojom ćete simulirati neku konkretnu vrstu loggera. Trebati će vam i interfejs. Diskutovati. Konkretni poziv logovanja bi trebao biti ***Logger.getLogger().log("hello world!");*** Na početku maina možete postaviti logger sa ***Logger.setLogger(loggerInstance);***

Zadatak 3

Banka iz Vježbe 1 – Primjer 1 vam se javila sa novostima: za logger će se koristiti komponenta koju je razvila kompanija RPR-ANSAMBL koja podatke bilježi između ostalog i u konzolu. Logger se isporučuje kroz ponovno iskoristivu biblioteku klâsa. Povežite logger sa vašim codebaseom i demonstrirajte da ste savladali sve izložene principe.

Hint: Importujte .dll tako što ćete uraditi desni klik na Project References u Solution Exploreru i odabrati opciju Add Reference, Browse i pronađite .dll datoteku koju ste preuzeli kao materijal za ove vježbe. (vježba/dll). Problem spajanja tuđe komponentu o kojoj inicijalno ne znate ništa sa apstrakcijom koju ste koristili u prvom zadatku nije nimalo trivijalan. Iskoristite object browser da istražite sadržaj importovanog dll-a i vidite šta imate na raspolaganju. Očito ćete morati iskoristiti neki adapter da biste prevazišli eventualne razlike između komponenti.

Zadatak 4

Vježbe 1, Primjer 1 – Dodajte u programsko rješenje za banku novu mogućnost: svaki objekat tipa Customer treba validirati prije nego ga se spasi u kolekciju klijenata. Validacija se treba obavljati korištenjem korisnički definirane validacijske funkcije za svaki tip klijenata.

Hint: Očekuje se sljedeći interfejs:

```
Poly.Bank b1 = new Poly.Bank(isPersonValid);
…

static Boolean isPersonValid(Poly.Person p)
{
    return p.Name.Length > 0;
}
```

Zadatak 5

Vježbe 1, Primjer 1 – Dodajte u programsko rješenje za banku novu mogućnost: moguće je pronaći sve osobe koje su klijenti banke na osnovu dijela imena. Obavezno koristiti ekstenzijsku metodu nad kolekcijom klijenata i lambda izraz.

Hint: Očekuje se sljedeći interfejs:

```
public List<Person> findPersons(String nameSubstr);
```

Zadatak 6

Vježbe 1, Primjer 1 – Dodajte u programsko rješenje za banku novu mogućnost: moguće je pronaći imena svih osoba koje su klijenti banke. Obavezno koristiti ekstenzijske metode nad kolekcijom klijenata i lambda izraze.

Hint: Očekuje se sljedeći interfejs:

```
public List<String> getAllNames();
```

Zadatak 7

Uporedite performanse unosa elemenata klase List<T> i ArrayList za slučaj prostih tipova podataka kao što su cijeli i realni brojevi (koristiti tip Int32 i Decimal) i referentnog tipa (klasa sa nekoliko atributa). Testni primjeri listi trebaju imati dovoljno velik i reprezentativan broj elemenata da se razlike u rezultatima mogu jasno usporediti. Koristiti klasu Stopwatch iz System.Diagonstics namespace za mjerenje vremena izvršenja programa.

Zadatak 8

Implementirati klasu Fakultet koja pohranjuje tri vrste kolekcija. Prva vrsta kolekcije su studenti, koju je neophodno realizovati kao riječnik, pri čemu je ključ riječnika broj indeksa svakog studenta, a vrijednost objekat tipa Student. Druga vrsta kolekcije su profesori, koju treba realizovati kao riječnik čiji se ključ sastoji iz titule profesora (docent, vanredni, redovni) i JMB, a vrijednost objekat tipa Profesor. Treća vrsta kolekcije su predmeti, predstavljeni kao lista objekata tipa Predmet. Sve klase unutar glavne klase Fakultet potrebno je smisleno organizovati i povezati u skladu sa naučenim elementima iz dosadašnjih vježbi. Klasa Fakultet treba zaštititi pristup svim kolekcijama koje pohranjuje i omogućiti metode kroz koje ćete demonstrirati upotrebu barem tri različite ekstenzijske metode za svaku od definiranih kolekcija. Primjer takve metode može biti pretraživanje studenata prema imenu, broj redovnih profesora na fakultetu, prosjek ocjena itd.

Zadatak 9

Implementirajte metodu ToSortedArray (List<string> lista) koja će kao parametar primiti listu stringova, a kao rezultat vratiti novi niz stringova (string []) sa svim elementima iz liste pri čemu elementi u nizu trebaju biti sortirani prema dužini (najkraći string prvi, najduži zadnji). Obratiti pažnju na činjenicu da je string referentni tip podatka.