ZADATAK ZA BONUS

Potrebno je Konzolnu aplikaciju i kreirati sleće tipove:

- **ClockAlarm** koji ima sedeća stanja i ponašanje:
  - Celobrojnu promenljivu **brojSekundiDoZvona,** sa podrazumevanom vrednošću 10
  - Metod **Snooze** koji odlaže alarm za 10 sekundi
  - Metod **Dismis** koji isključuje alarm
  - Javni event **VremeJeZaUstajanje** na koji se mogu prijaviti sve metode koje imaju povratni tip ***void*** i prihvataju instancu na klasu **ClockAlarm**
  - Javni metod StartAlarm koji nakon **brojSekundiDoZvona** podiže događaj **VremeJeZaUstajanje**
- **Student** koji ima sledeća stanja i ponašanje:
  - Metod **AlarmZvoni** koji nema povratni tip i prihvata instancu klase **ClockAlarm.** Metod u konzoli postavlja pitanje „Da li želite da odložite alarm?" i student odgovara sa **da/ne**. U zavisnosti od odgovora alarm se odlaže ili isključuje.
- **Program** sa Main metodom u kojoj se kreira jedan **ClockAlarm** i student pri cemu se on prijavljuje na event **VremeJeZaUstajanje** klase **ClockAlarm**.

DINAMIČKE/ANONIMNE FUNKCIJE

U prethodnom delu su definisane delegatske promenljive koje su pokazivale na statičke metode nekih klasa. Nekada je malo nezgodno da se predefinišu sve funkcije koje su potrebne u kodu. Umesto statičkih metoda omogućeno je dinamičko kreiranje funkcija i dodeliti ih delegatima. Primer je dat u nastavku:

```
class Program
{
    public delegate double AritmetickaFunkcija(double x, double y);

    static void Main()
    {
        AritmetickaFunkcija funkcija = delegate (double x, double y) { return x * y; };
        Console.WriteLine(funkcija(3, 5));
    }

}
```

Pokazivači na anonimne funkcije se koriste ako je potrebno da koristimo funkciju samo na jednom mestu. U suprotnom, ako želimo funkciju više puta da koristimo onda je dobra praksa definisati je u okviru neke klase i referencirati se na nju. Na primer, ako je potreban uslov da je neki broj prost onda je bolja praksa da se ova funkcija definiše u nekoj klasi, međutim ako je potreban uslov da je broj veći od 5 onda nema smisla staviti to u statičku funkciju jer je ova funkcija specifična za dati problem.

LAMBDA IZRAZI

Lambda izrazi u C# programskom jeziku sluze za kompaktniji način predstavljanja dinamičkih anonimnih funkcija. Sve što je potrebno za definisanje funkcije je da se navede lista argumenata i telo. Lambda izrazi imaju jednostavnu sintaksu kojom funkcije predstavljamo u obliku:

(lista argumenata) => (telo)

(lista argumenata) => (izraz)

Dok bi u C# kodu to izgledalo ovako:

```
class Program
    {
        public delegate double AritmetickaFunkcija(double x, double y);

        static void Main()
        {
            AritmetickaFunkcija funkcijaLambda = (x, y) => x * y;
            Console.WriteLine("Rezultat funkcije je: " + funkcijaLambda(3, 5));
        }

    }
```

Ovaj primer je identičan prethodnom primeru, s tim što je umesto anonimne funkcije delegatu dodeljen lambda izraz. Funkcija pokazuje na lambda izraz i prilikom poziva za argumente 3 i 5 vratiće rezultat 15. Kao i anonimna funkcija, lambda izraz ima dva dela - deo ispred => kojim se definiše lista argumenata i deo iza => koji predstavlja telo funkcije. Ako se ovaj oblik uporedi sa anonimnom funkcijom predstavljenom u formi delegate(<>){<>} primećuje se da su potpuno ekvivalentni. U slučaju da postoji samo jedan argument nije potrebno okružiti argumente zagradama. Na primer x => x

PREDIKATI

Predikati predstavljaju funkcije kojima se definišu uslovi. Radi se o pokazivačima na funkcije koje vraćaju vrednosti tačno ili netačno. Predikati su opisuju klasom Predicate gde su T1,T2,...,Tn tipovi argumenata funkcije. Vrednost koju vraća predikat je logička promenljiva tako da nije potrebno definisati tip rezultate. U principu Predicate je ekvivalentno Func. Funkcija koja određuje da li je broj paran i njena upotreba uz pomoć predikata je prikazana u sledećem primeru:

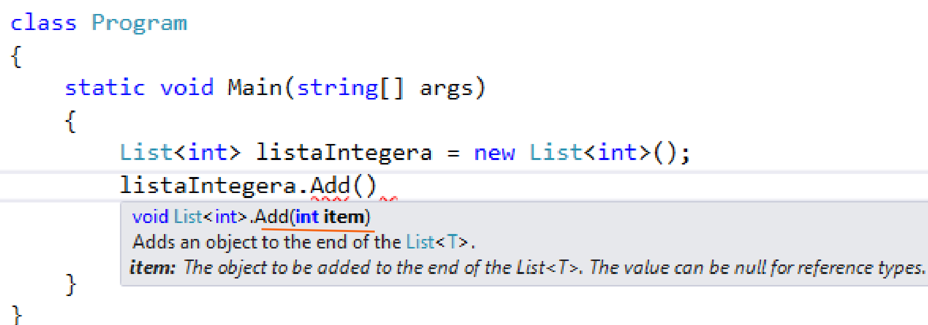
```
 class Program
{
    public static bool Paran(int broj)
    {
        return broj % 2 == 0;
    }
    static void Main(string[] args)
    {
        bool rezultat;

        Predicate<int> predikat = Paran; // Predikat pokazuje na funkciju paran
        rezultat = predikat(4); //rezultat = true
        Console.WriteLine("Broj 4 je paran = " + rezultat);

        predikat = x => { return x % 2 == 1; }; // Predikat pokazuje na funkciju neparan

        rezultat = predikat(9);
        Console.WriteLine("Broj 9 je neparan = " + rezultat);

        rezultat = predikat(14); // rezultat = false
        Console.WriteLine("Broj 14 je neparan = " + rezultat);
    }

}
```

Rad sa predikatima je sličan radu sa funkcijama – moguće ih je definisati i dodeliti im postojeće metode, anonimne funkcije/delegate ili lambda izraze. Jedina razlika u odnosu na funkcije je u činjenici da je rezultat uvek logičkog tipa. Predikati se najčešće koriste kao uslovi u kodu.

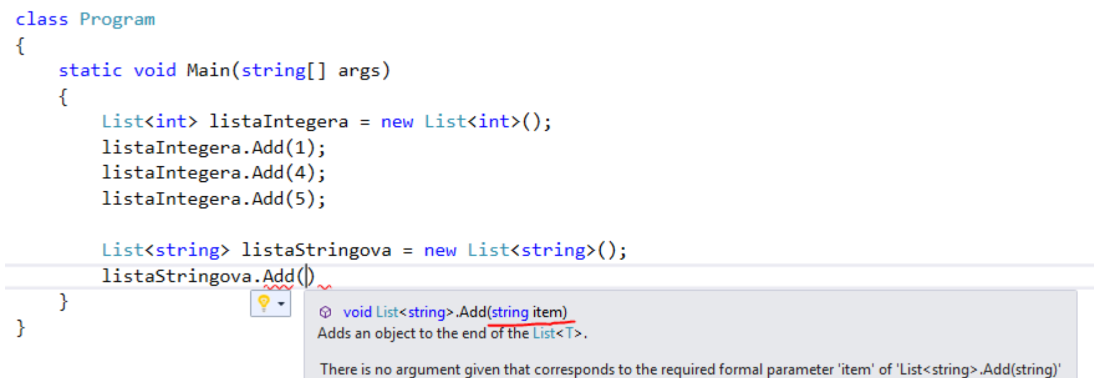
GENERIČKE KLASE I METODE

Generičke klase omogućavaju parametrizovanje tipova, to znači da se može definisati klasa ili metoda sa generičkim (opštim) tipom. Jedna od postojećih generičkih klasa u C#-u koju smo koristili jeste ***List***. Pomoću ove generičke klase možemo da kreiramo listu brojeva, stringova, objekata, struktura...

U listingu ispod je prikazan primer liste brojeva tipa *int*, i kao što se vidi, prilikom poziva metode **Add**, očekuje se da ulazni parametar bude *int*.



Slično, ako se kreira lista string-ova, prilikom poziva metode **Add**, očekuje se da ulazni parametar bude string.



Tip podatka koji će se čuvati u listi se navodi prilikom instanciranja objekta. U prikazanom primeru, brojevi i stringovi su konkretni tipovi koji zamenjuju generičke (opšte).

PRIMER 1

Definisati klasu Test koja čuva promenljivu generičkog (opšteg) tipa, sadrži konstruktor koji prihvata promenljivu i setuje je, sadrži metod koji na standardnom izlazu ispisuje string reprezentaciju promenljive koja se čuva.

```
class Test<T>
{
    T _value;

    public Test(T t)
    {
        // The field has the same type as the parameter.
        this._value = t;
    }

    public void Write()
    {
        Console.WriteLine(this._value.ToString());
    }
}

public class Student
{
    int indeks;
    int godina;
    string ime;

    public Student(int indeks, int godina, string ime)
    {
        this.indeks = indeks;
        this.godina = godina;
        this.ime = ime;
    }

    public override string ToString()
    {
        return "" + indeks + "/" + godina + " - " + ime;
    }
}

class Program
{
    static void Main()
    {
        // Use the generic type Test with an int type parameter.
        Test<int> test1 = new Test<int>(5);
        // Call the Write method.
        test1.Write();

        // Use the generic type Test with a string type parameter.
        Test<string> test2 = new Test<string>("cat");
        test2.Write();

        // Use the generic type Test with a Student type parameter.
        Test<Student> test3 = new Test<Student>(new Student(01,2016,"Pera Peric"));
        test3.Write();
    }

}
```

Karakteristika ovako definisane klase Test ima osobinu da može čuvati bilo koji tip iz programskog jezika C#. Sa kojim tipom će klasa test raditi, navodi se prilikom njenog instanciranja.

OGRANIČENJA GENERIČKIH TIPOVA

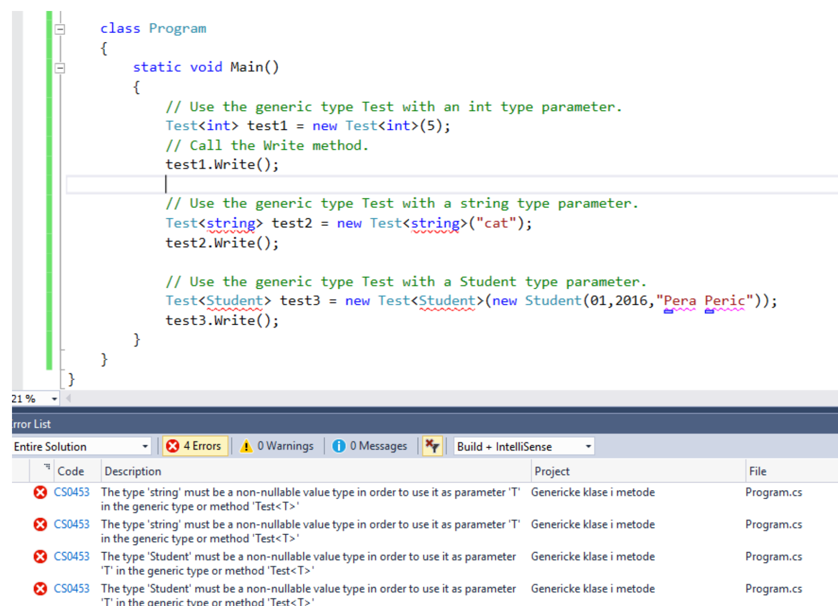
Prilikom definisanja generičke klase, moguće je uključiti ograničenja na tipove koji se koriste prilikom instanciranja klase. Ako tip kojim se pokušava instancirati klasa nije dozvoljen, prema definisanom ograničenju, javiće se kompajlerska greška. Ograničenja se navode ključnom rečju **where**.

PRIMER 2

Uvedimo ograničenje da prethodno definisanu klasu mogu koristiti samo **vrednosni** tipovi.

```
class Test<T> where T : struct
    {
        T _value;

        public Test(T t)
        {
            // The field has the same type as the parameter.
            this._value = t;
        }

        public void Write()
        {
            Console.WriteLine(this._value.ToString());
        }

    }
```

Naredbama `where T : struct` je onemogućeno da da prilikom instanciranja klase test, koristimo referentne tipove. Ova promena će imati za rezultat kompajlersku grešku, kod string-a i Stuent-a.



Spisak ograničenja koji se može definisati je prikazan u nastavku:

Ograničenje	Opis
where T: struct	Tip mora biti vrednosni.
where T : class	Tip mora biti referentni.
where T : new()	Tip mora imati javni konstruktor bez parametara.
where T : <base class name>	Tip mora da nasleđuje klasu <base class name>
where T : <interface name>	Tip mora da implamentira interfejs <interface name>

GENERIČKI DELEGATI

U predhodnom delu smo kreirali tipizirane delegate koji su moglu da referenciraju samo odgovarajuće funkcije. Funkcije koje su im odgovarale po tipovima argumenata i njihovom broju. Korišćenjem generika moguće je generalizovati delegate koji mogu da rade sa različitim tipovima. Primer takve generičke definicije delegata je prikazan u sledećem kodu:

```
class Program
{
    public delegate T GenerickaFunkcija<T>(T x, T y);
    static void Main(string[] args)
    {
        GenerickaFunkcija<int> f1 = delegate (int x, int y) { return x + y; };
        GenerickaFunkcija<string> f2 = delegate (string x, string y)
        { return x + " " + y; };

        Console.WriteLine("Rezultat funkcije je: " + f1(3, 6));
        Console.WriteLine("Rezultat funkcije je: " + f2("Genericki", "delegat"));
    }

}
```

U prethodnom primeru f1 predstavlja funkciju koja sabira brojeve i u promenljivu x upisuje rezultat 9, dok druga funkcija spaja stringove i u promenljivu s upisuje string "Genericki delegat". Kao što se može videti jedan generički delegat je potreban da bi se prilagodio različitim metodama, što nije bio slučaj kod tipiziranih delegata.