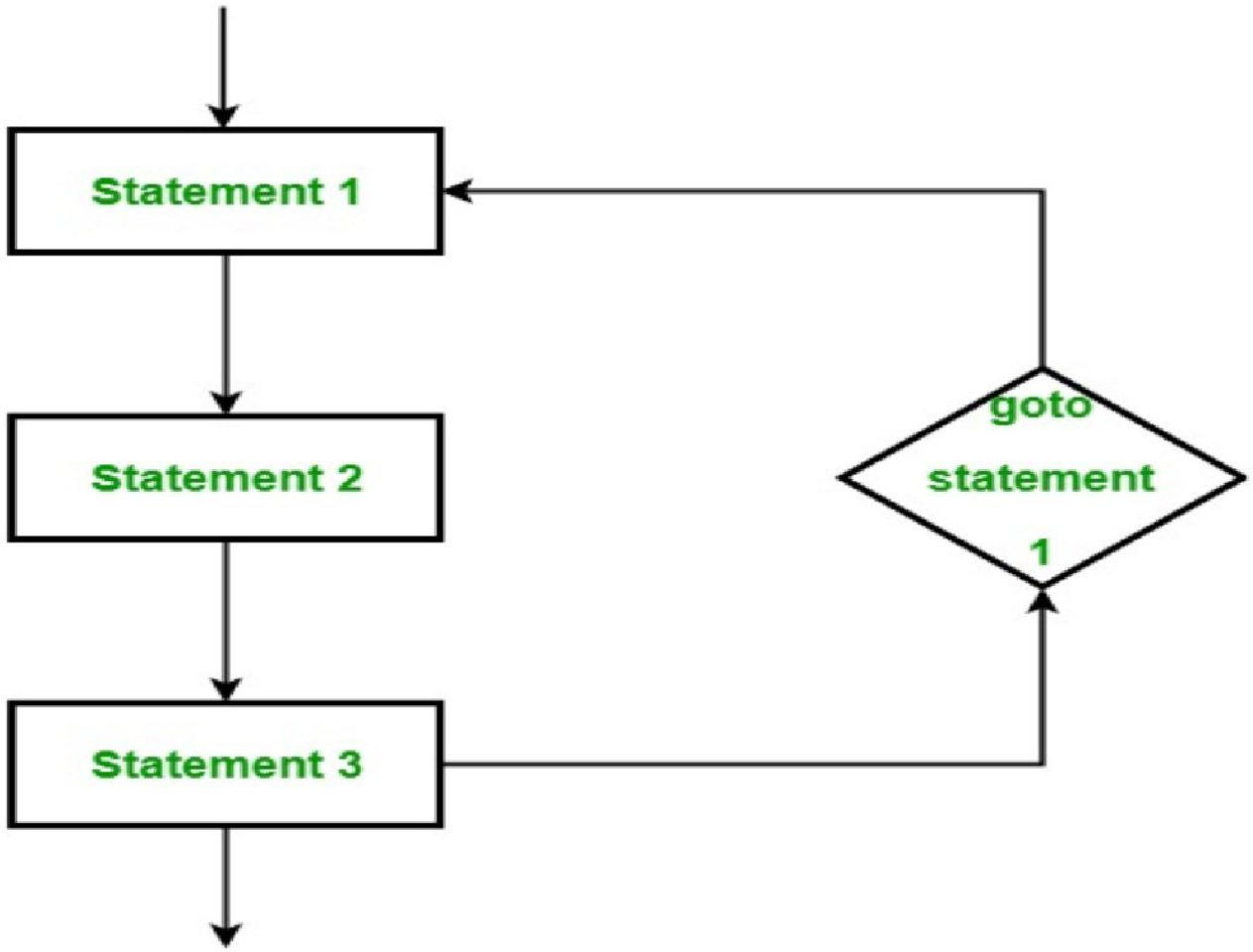
C#

Il deo

Goto izraz

- *Goto* naredba se koristi za prenos kontrole toka izvršavanja na neku određenu labelu.
- Smislena upotreba
  - *switch* naredba
  - iskakanje iz duboko ugnjezdenih petlji
  - ponovno izvrsavanje koda nakon unosenja ispravki

Goto izraz



```
string coffeeType = Console.ReadLine();

switch (coffeeType) {
    case "milk":
      Console.WriteLine("Can I have a milk coffee?");
      break;
    case "black":
      Console.WriteLine("Can I have a black coffee?");
      // transfer code to case "milk"
      goto case "milk";
    default:
      Console.WriteLine("Not available.");
        break;
}
```

```
for (int i = 0; i < 10; i++){
    for (int j = 0; j < 10; j++){
        while (someCondition) {
            //  ....
            if (someOtherCondition){
                goto finished;
            }
        }
    }
}
finished:
```

Decimalni tip

- Ne postoji u Javi
- Koristi se za predstavljanje realnih brojeva
- Pored njega u C# postoje i *float* i *double*
- Pruža veću preciznost i manji raspon od *float* i *double* brojeva
- Koristi se za finansijske i monetarne račune
- Tip realnog literala određuje se pomoću sufiksa:
  - D –*Double*
  - F –*Float*
  - M –*Decimal*

Preklapanje operatora

- Korisnički definisani tip može da redefiniše predefinisane C# operatore. Odnosno, tip može da pruži prilagođenu implementaciju operacije u slučaju da su jedan ili dva operanda tog tipa.
- Deklaracija operatora mora da ispunjava sledeća pravila:
  - Sadrži *public* i *static* modifikator pristupa
  - Unarni operator ima jedan ulazni parametar, a binarni dva ulazna parametra od kojih najmanje jedan mora da bude istog tipa kao i tip nad kojim se definiše operator.

Preklapanje operatora

- Funkcije sa posebnim imenima:
  - Ključna reč *operator*
  - Simbol operatora koji se definiše

```
// Overload + operator to add two Box objects.
public static Box operator+ (Box b, Box c) {
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

Primer 4.

For petlja

- *For* petlja služi za izvršavanje bloka izraza više puta sve dok je neki uslov ispunjen. Sastoji se iz 3 dela, a to su:
  - Inicijalizacija promenljive
    - U ovom delu je potrebno definisati i inicijalizovati promenljivu koja će se koristiti u uslovnom izrazu i u delu za promenu koraka. Izvršava se samo jednom i to na početku *for* petlje.
  - Uslov
    - Predstavlja relacioni izraz, koji vraća *True* i *False*. Od ovog izraza zavisi da li će se telo *for* petlje izvršiti ili ne. Provera uslova se dešava pre svake iteracije.
  - Korak
    - U ovom delu definišemo kako se menja promenljiva koju smo definisali u prvom delu. Ovaj deo *for* petlje se izvršava svaki put nakon izvršavanja tela *for* petlje. Obično je to inkrement ili dekrement izraz.

For petlja

- Koraci izvršavanja *for* petlje:

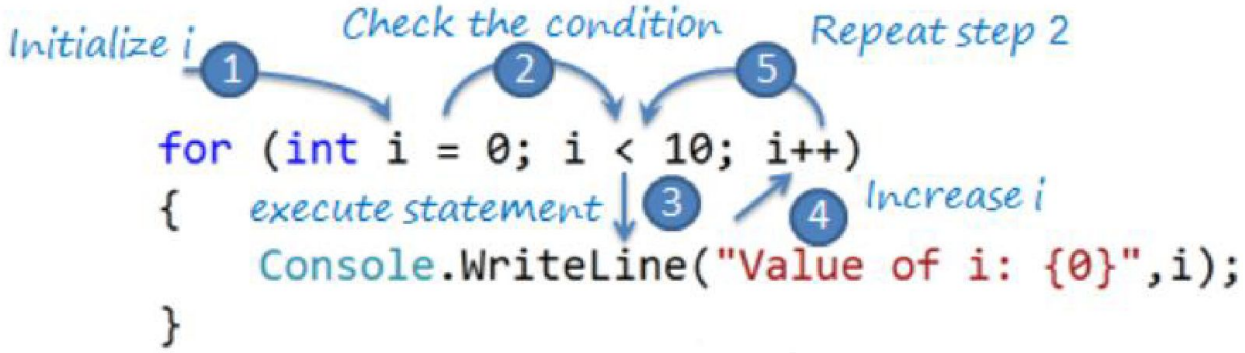
  1.Definisanje i inicijalizacija promenljive

  2.Provera da li je uslov ispunje ili ne

  3.Ako je uslov ispunjen onda se izvršava telo petlje

  4.Evaluacija koraka

  5.Ponovna provera da li je uslov ispunje ili ne



Foreach petlja

- *Foreach* petlju koristimo za iteraciju kroz neku kolekciju podataka ili niz.

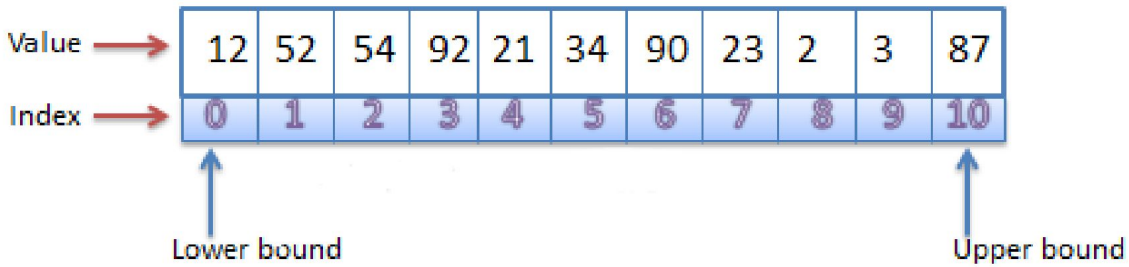
```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

Ključna reč ***in*** služi za ponavljanje iterabilne stavke. Ona na početku svake iteracije bira stavku iz kolekcije i smešta je u promenljivu.

Foreach petlja

- U prvoj iteraciji je prva stavka sačuvana u promenljivoj, ukoliko u kolekciji ima još elemenata onda se izvršava sledeća iteracija, na čijem početku se bira nova stavka iz kolekcije.
- Broj izvršavanja *foreach* petlje zavisi od broja stavki u kolekciji.

Niz



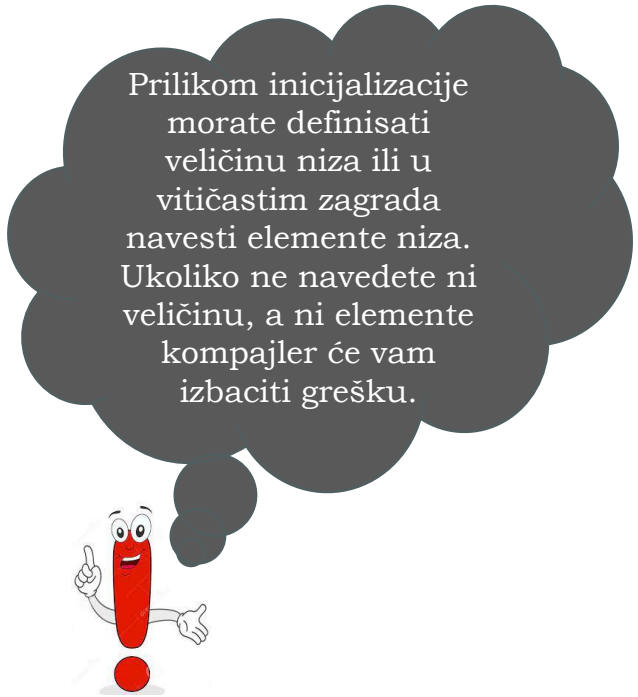
- Struktura podataka koja se koristi za čuvanje fiksnog broja elemenata istog tipa podataka.
- Deklariše se na isti način kao i promenljiva, ali uz upotrebu uglastih zagrada.
- *System.Array* – klasa koja sadrži metode za kreiranje, manipulaciju, pretragu i sortiranje nizova.
- Elementi niza se čuvaju sekvencijalno u memoriji, što utiče na performanse.
- Svaki element niza ima jedinstven indeks koji počinje sa 0 i naknadno se uvećava za 1. Indeks prvog elementa je donja granica, a indeks poslednjeg je gornja granica.
- Može da se prosledi kao parametar metode.
- Nizovi su referentni tipovi.

Deklaracija

```
int[] intArray;
bool[] boolArray;
string[] string Array;
double[] doubleArray;
byte[] byteArray;
Student[] customClassArray;
```

Inicijalizacija

```
#defining array with size 5, add values later on
int[] intArray1 = new int[5];

#defining array with size 5, add adding values at same time
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};

#defining array with size 5 elements which indicates the size of an array
int[] intArray3 = {1, 2, 3, 4, 5};

int[] intArray = new int[]; //compile error: must give size of an array
```

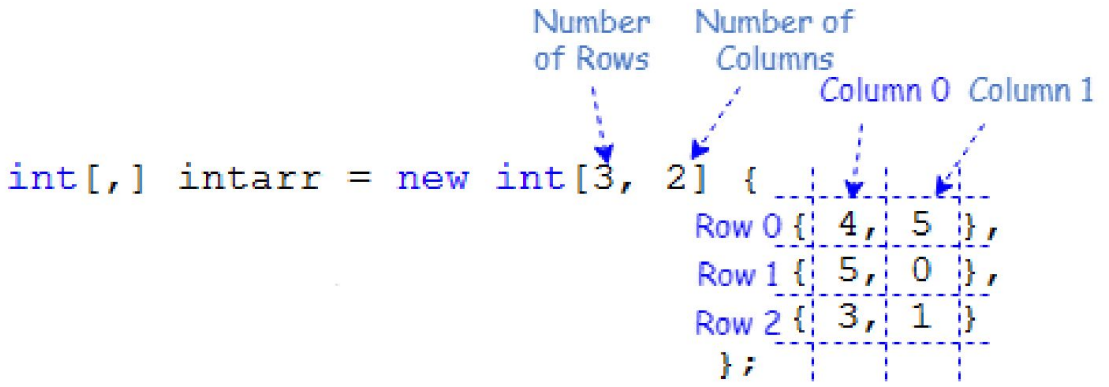


Pristup elementima niza

```
int[] intArray = new int[3]{10, 20, 30 };

for(int i = 0; i < intArray.Length; i++)
    Console.WriteLine(intArray[i]);

intArray[0];  //returns 10

intArray[2];  //returns 30

int[] evenNums = { 2, 4, 6, 8, 10};
string[] cities = { "Mumbai", "London", "New York" };

foreach(var item in evenNums)
    Console.WriteLine(item);

foreach(var city in cities)
    Console.WriteLine(city);
```

Višedimenzionalni niz

- Višedimenzionalni niz je niz koji sadrži više dimenzija koje predstavljaju elemente u tabelarnom formatu poput redova i kolona.
- Deklarišu se specificiranjem vrste podataka elemenata praćene uglastim zagradom sa separatorom zarez ([,]). Prvi element u zagradi predstavlja broj redova, a drugi broj kolona.
- Broj kolona je u svakom redu isti.



Inicijalizacija višedimenzionalnog niza

- Prilikom inicijalizacije moramo navesti koliko redova i kolona imamo
- Ako hoćemo možemo odmah da navedemo i elemente niza upotrebom vitičastih zagrada, kao što je prikazano na primeru

```
int[,] intArray = new int[3,2]{
                                {1, 2},
                                {3, 4},
                                {5, 6}
                              };
```

Pristup elementima niza

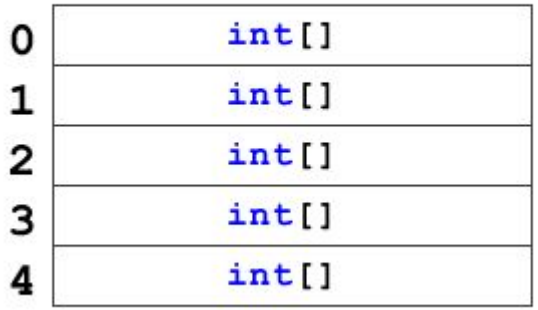
- Elementima niza pristupamo navođenjem indeksa reda i indeksa kolone unutar uglastih zagrada
- Indeksi počinju od 0

```
int[,] intArray = new int[3,2]{
                                {1, 2},
                                {3, 4},
                                {5, 6}
                              };

intArray[0,0]; //Output: 1
intArray[0,1]; // 2

intArray[1,0]; // 3
intArray[1,1]; // 4

intArray[2,0]; // 5
intArray[2,1]; // 6
```

Jagged array

- Jagged nizovi su poznati kao matrice, odnosno niz nizova. Oni umesto jedne vrednosti skladište nizove.
- Deklarišu se navođenjem tipa podatka iza kojeg slede dva para uglastih zagrada ([][]). Prva zagrada određuje veličinu matrice, a druga zagrada dimenziju niza koji će se čuvati.
- Prilikom inicijalizacije ovog niza potrebno je navesti veličinu matrice, odnosno koliko nizova će da sadrži.
- Nizovi koji se skladište mogu da budu različitih veličina.

```
int[][] jagArray = new int[5][];
```



Inicijalizacija jagged niza

- Potrebno je prvo da specificirate veličinu matrice, a zatim da inicijalizujete nizove koji će da se nalaze u matrici.

```
int[][] intJaggedArray = new int[2][];

intJaggedArray[0] = new int[3]{1, 2, 3};

intJaggedArray[1] = new int[2]{4, 5 };
```

Pristup elementima jagged niza

- Elementima niza pristupamo upotrebom dva para uglastih zagrada, u prvoj zagradi navodimo redni broj niza kojem pristupamo, a u drugoj indeks elementa u nizu kojem pristupamo

```
int[][] intJaggedArray = new int[2][];

intJaggedArray[0] = new int[3]{1, 2, 3};

intJaggedArray[1] = new int[2]{4, 5 };

Console.WriteLine(intJaggedArray[0][0]); // 1

Console.WriteLine(intJaggedArray[0][2]); // 3

Console.WriteLine(intJaggedArray[1][1]); // 5
```

Primer 1.

Lista

- Generički tip podataka
- List (C#) ⇔ ArrayList (Java)
  - Implementirana kao IList<T>
- Sadrži elemente određenog tipa podataka.
- Sadrži metode za sortiranje, pretragu i modifikacije.

Kreiranje liste i dodavanje elemenata u listu

```
var cities = new List<string>();

cities.Add("New York");

cities.Add("London");

cities.Add("Mumbai");

cities.Add("Chicago");

cities.Add(null);// nulls are allowed for reference type list

//adding elements using collection-initializer syntax

var bigCities = new List<string>()
                    {
                        "New York",
                        "London",
                        "Mumbai",
                        "Chicago"
                    };
```

```
// Create a list

List<string> AuthorList = new List<string>();



// Add items using Add method

AuthorList.Add("Mahesh Chand");

AuthorList.Add("Praveen Kumar");
// Add a range of items

string[] authors = { "Mike Gold", "Don Box",
                     "Sundar Lal", "Neel Beniwal"
};

AuthorList.AddRange(authors);
```

Ubacivanje elementa u listu

- Pomoću metode *Insert* možemo da ubacimo element na određenu poziciju u listi.

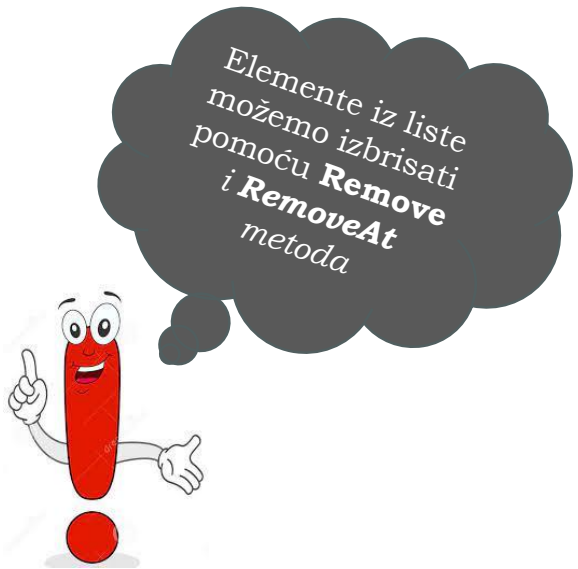
```
var numbers = new List<int>() {10, 20, 30, 40};

numbers.Insert(1, 11); //inserts 11 at 1st index: after 10

foreach(var num in numbers){
    Console.Write(num);
}
```

Pristup elementima liste

```
List<int> numbers = new List<int>() { 1, 2, 5, 7, 8, 10 };

Console.WriteLine(numbers[0]); // prints 1

Console.WriteLine(numbers[1]); // prints 2

Console.WriteLine(numbers[2]); // prints 5

Console.WriteLine(numbers[3]); // prints 7


// using foreach LINQ method

numbers.ForEach(num => Console.WriteLine(num + ", "));//prints 1, 2, 5, 7, 8, 10,


// using for loop

for(int i = 0; i < numbers.Count; i++)

    Console.WriteLine(numbers[i]);
```

Brisanje elemenata iz liste

```
var numbers = new List<int>(){ 10, 20, 30, 40, 10 };

numbers.Remove(10); // removes the first 10 from a list

numbers.RemoveAt(2); //removes the 3rd element (index starts from 0)

//numbers.RemoveAt(10); //throws ArgumentOutOfRangeException

foreach (var el in intList)
    Console.Write(el); //prints 20 30
```



Dictionary

- *Dictionary* (C#) ⇔ *HashMap* (Java)
- Čuva parove (ključ, vrednost)
  - Ključevi moraju biti jedinstveni i ne mogu biti null
  - Vrednost može da bude null ili duplikat
  - Vrednosti se pristupa prosleđivanjem povezanog ključa
- Element se čuva kao *KeyValuePair<TKey, TValue>* objekat.

Kreiranje rečnika i dodavanje elemenata

```
IDictionary<int, string> numberNames = new Dictionary<int, string>();
numberNames.Add(1,"One"); //adding a key/value using the Add() method
numberNames.Add(2,"Two");
numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.
//numberNames.Add(3, "Three");

foreach(KeyValuePair<int, string> kvp in numberNames)
    Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

//creating a dictionary using collection-initializer syntax
var cities = new Dictionary<string, string>(){
    {"UK", "London, Manchester, Birmingham"},
    {"USA", "Chicago, New York, Washington"},
    {"India", "Mumbai, New Delhi, Pune"}
};

foreach(var kvp in cities)
    Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);
```

Pristup elementima rečnika

```
var cities = new Dictionary<string, string>(){
    {"UK", "London, Manchester, Birmingham"},
    {"USA", "Chicago, New York, Washington"},
    {"India", "Mumbai, New Delhi, Pune"}
};

Console.WriteLine(cities["UK"]); //prints value of UK key
Console.WriteLine(cities["USA"]);//prints value of USA key
//Console.WriteLine(cities["France"]); // run-time exception: Key does not exist

//use ContainsKey() to check for an unknown key
if(cities.ContainsKey("France")){
    Console.WriteLine(cities["France"]);
}
```

```
//use TryGetValue() to get a value of unknown key
string result;

if(cities.TryGetValue("France", out result))
{
    Console.WriteLine(result);
}

//use ElementAt() to retrieve key-value pair using index
for (int i = 0; i < cities.Count; i++)
{
    Console.WriteLine("Key: {0}, Value: {1}",
                                        cities.ElementAt(i).Key,
                                        cities.ElementAt(i).Value);
}
```

Izmena rečnika

```
var cities = new Dictionary<string, string>(){
    {"UK", "London, Manchester, Birmingham"},
    {"USA", "Chicago, New York, Washington"},
    {"India", "Mumbai, New Delhi, Pune"}
};

cities["UK"] = "Liverpool, Bristol"; // update value of UK key
cities["USA"] = "Los Angeles, Boston"; // update value of USA key
//cities["France"] = "Paris"; //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){
    cities["France"] = "Paris";
}
```

Brisanje elemenata iz rečnika

```
var cities = new Dictionary<string, string>(){
    {"UK", "London, Manchester, Birmingham"},
    {"USA", "Chicago, New York, Washington"},
    {"India", "Mumbai, New Delhi, Pune"}
};

cities.Remove("UK"); // removes UK
//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it
    cities.Remove("France");
}

cities.Clear(); //removes all elements
```

HashSet

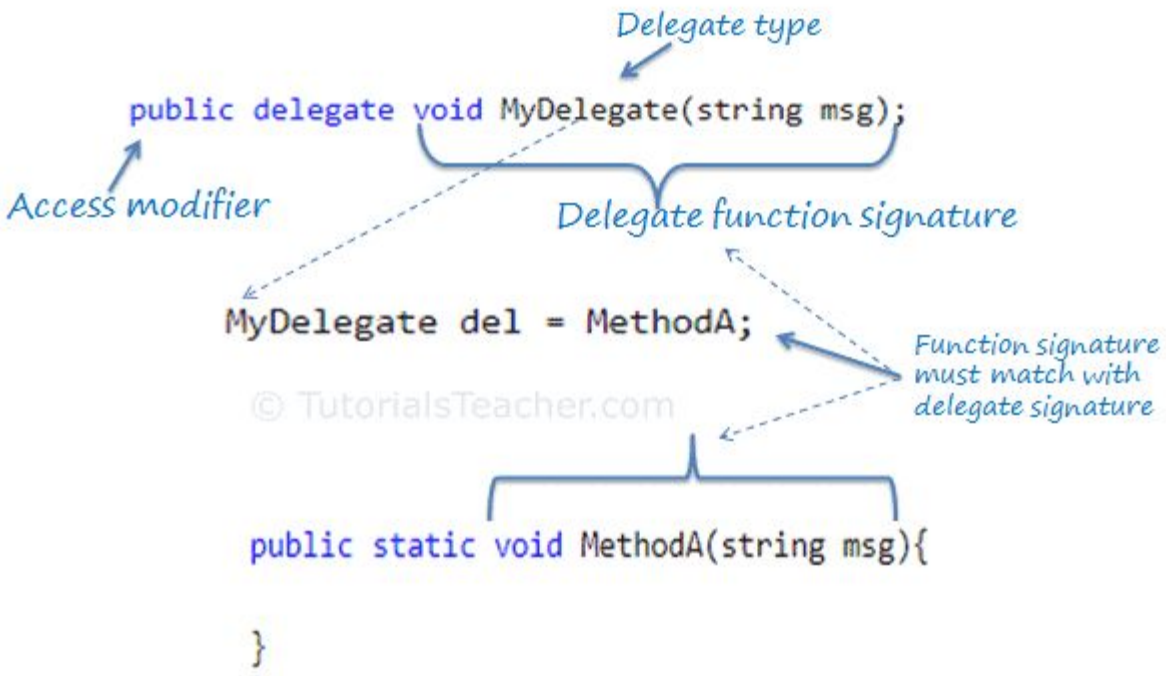
- *HashSet* (C#) ⇔ *HashSet* (Java)

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}

Console.Write("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);

Console.Write("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);
```

Primer 7.

Delegati

- Tip koji predstavlja reference na metode sa određenim potpisom parametara i povratnim tipom. Kada instancirate delegat, onda njegovu instancu možete da povežete sa bilo kojom metodom koja je kompatibilna sa potpisom delegata.
- Koristimo kada hoćemo da prosledimo funkciju kao parametar, a obrađivači događajima nisu ništa drugo do metode koje se pozivaju putem delegata.
- Bilo koju metodu iz bilo koje dostupne klase ili strukture koja odgovara tipu delegata možete da dodelite delegatu.
- Deklarišu se izvan klase upotrebom ključne reči ***delegate***, nakon čega sledi potpis funkcije.
- Kreiranje instance delegata obezbeđuje fleksibilnost, zato što se kreira red metoda koje treba da budu pozvane u određenom redosledu
- Omogućava izvršavanje više akcija istovremeno
- Omogućavaju da implementiramo događaje za slanje poruka između objekata koji ne treba da znaju ništa jedan o drugom


Delegate type
public delegate void MyDelegate(string msg);
Access modifier
Delegate function signature
MyDelegate del = MethodA;
Function signature must match with delegate signature
© TutorialsTeacher.com
public static void MethodA(string msg){
}


Potpis funkcije uključuje i povratnu vrednost funkcije.

Kreiranje delegata

```
public delegate void MyDelegate(string msg); // declare a delegate

// set target method
MyDelegate del = new MyDelegate(MethodA);
// or
MyDelegate del = MethodA;
// or set lambda expression
MyDelegate del = (string msg) =>  Console.WriteLine(msg);

// target method
static void MethodA(string message)
{
    Console.WriteLine(message);
}
```

Pozivanje delegata

- Mogu da se pozovu na dva načina:
  - Kao funkcija

```
Print printDel = PrintNumber;
printDel(10000);
```

  - Pomoću *Invoke* metode
    - Ovaj način nam omogućava da pomoću operatora ? proverimo da li neki delegat postoji ili ne pre njegovog poziva.
    - Primer: *printDel?.Invoke(10000);*

```
Print printDel = PrintNumber;
printDel.Invoke(10000);
```

Delegati kao parametri metode

- Metoda može da ima parametar tipa delegara i može pozvati parametar delegata, kao što je prikazano u datom primeru.

```
public static void PrintHelper(Print delegateFunc, int numToPrint)
{
    delegateFunc(numToPrint);
}
```

Uvezivanje delegata

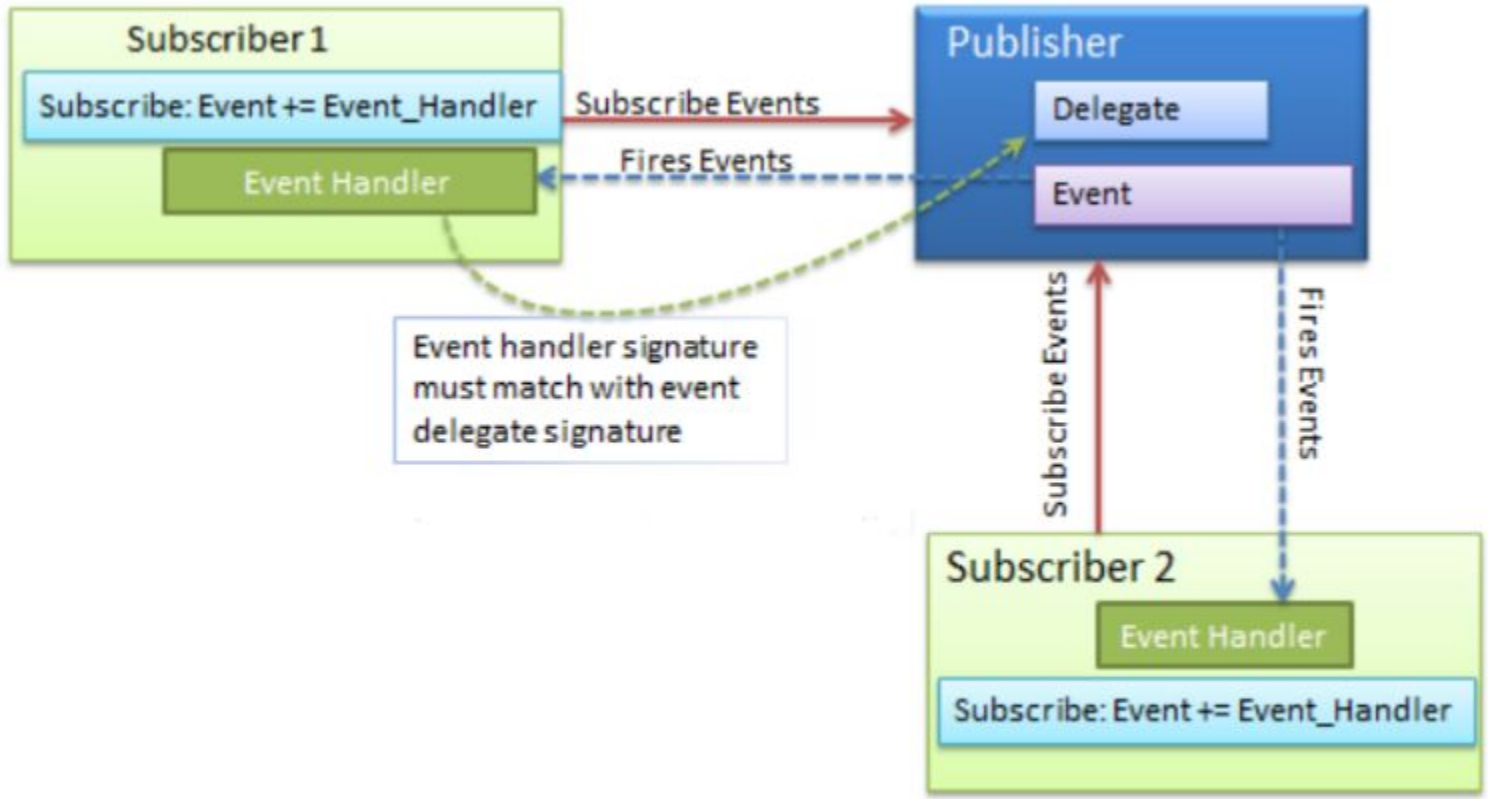
- Delegati mogu da povezuju više sličnih funkcija
- Funkcija se na delegat dodaje operatorom +, a brisanje iz delegata se radi sa operatorom -

```
Print printDel = PrintNumber;
printDel += PrintHexadecimal;
printDel += PrintMoney;

printDel(1000);

printDel -=PrintHexadecimal;
printDel(2000);
```

Primer 5.

Događaji

- Događaj je obaveštenje koje objekat šalje da signalizira da se neka radnja desila. Npr, ako korisnik klikne na dugme kreiraće se događaj.
- Obezbeđuju razmenu poruka između dva objekta.
- Takođe, za događaje može da kažemo da su to enkapsulirani delegati. Delegat definiše potpis za metodu za rukovanje događajima.



Event publisher-Subscriber

Događaji

- Svojstva:
  - Izdavač utvrđuje kada se događaj pokreće, a pretplatnici obrađuju šta će da se desi u odgovoru na događaj.
  - Događaj može da ima više pretplatnika. Pretplatnik može da obrađuje viđe događaja od više izdavača.
  - Događaji koji nemaju pretplatnike se nikada ne priređuju.
  - Obično se koriste za signalizaciju korisničkih radnji, kao što su klik na dugme ili izbor iz menija.
  - Kada događaj ima više pretplatnika, obrađivači događaja se sinhrono pozivaju kada se događaj desi.

Delegati vs događaji

1. Delegat je objekat koji se koristi kao pokazivač na funkciju, a događaji su apstrakcije delegata.
2. Delegat se deklariše izvan klase (mogu da se definišu i u klasi), a događaj unutar klase.
3. Delegati su fleksibilniji.
4. Delegati su nezavisni od događaja, a događaji se ne mogu koristiti bez delegata.

Lambda izrazi

1. Promenljiva x je sama po sebi lambda izraz
2. Ako su λ i x lambda izrazi, onda je λx, λ lambda izraz poznat kao lambda apstrakcija (definicija funkcije).
3. Ako su t i s lambda izrazi, onda je (ts) takođe lambda izraz poznat kao lambda konkretizacija ili lambda primena

Lambda izrazi

Lambda izraz

- Telo lambda funkcije se sastoji od samo jednog izraza

```
(input-parameters) => expression
```

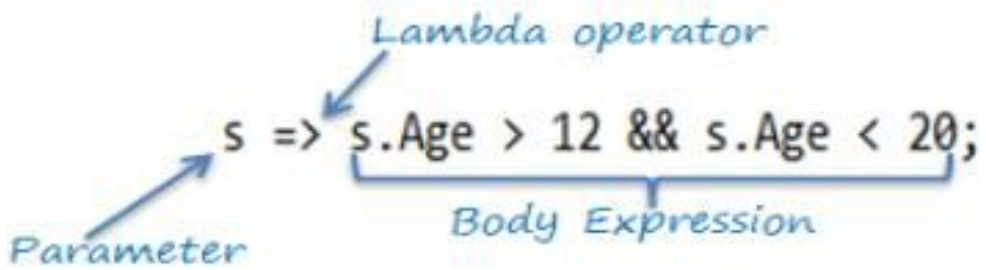
Lambda iskaz

- Telo lambda iskaza se sastoji od više izraza i oni se nalaze unutar vitičastih zagrada

```
(input-parameters) => { <sequence-of-statements> }
```

Lambda izrazi

- Sintaksa lambda izraza
  - Pomoću lambda operatora (=>) odvajamo listu parametara od tela lambda funkcije



  - Kada kreiramo lambda izraz, prvo treba da odredimo ulazne parametre i napišemo ih na levoj strani lambda operatora, a izraz ili blok izraza na drugoj strani.

Lambda izrazi

- Ulazni parametri
  - Nema parametara

```
Action line = () => Console.WriteLine();
```

  - Jedan parametar

```
Func<double, double> cube = x => x * x * x;
```

  - Više parametara

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Lambda izrazi

- Sa više parametara
  - Lambda izraz može da sadrži više parametara i ako je to slučaj onda se parametri navode u zagradama.

```
(Student s,int youngAge) => s.Age >= youngage;
```

- Bez parametara

```
() => Console.WriteLine("Parameter less lambda expression")
```

- Višelinijski izraz

```
(s, youngAge) =>
{
  Console.WriteLine("Lambda expression with multiple statements in the body");

  Return s.Age >= youngAge;
}
```

Lambda izrazi

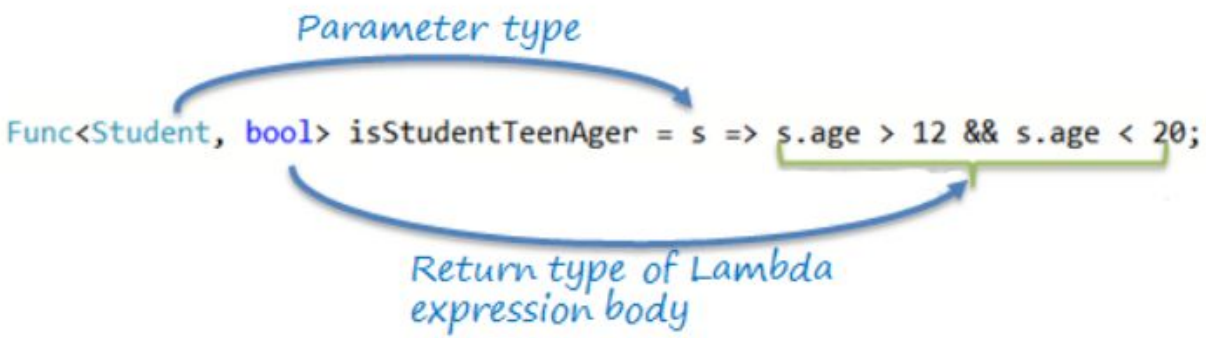
- Deklaracija lokalne promenljive
  - Unutar tela lambda izraza mogu da se definišu i lokalne promenljive i one važe samo u opsegu lambda izraza

```
s =>
{
    int youngAge = 18;

    Console.WriteLine("Lambda expression with multiple statements in the body");

    return s.Age >= youngAge;
}
```

Lambda izrazi i delegati

- Lambda izraz može da se dodeli delegatu tipa *Func<in T, out TResult>*
  - Poslednji tip parametra u funkcijskom delegatu je povratni tip, a ostatak su ulazni parametri

```
Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

Student std = new Student() { age = 21 };

bool isTeen = isStudentTeenAger(std);// returns false
```



LINQ

- Skup tehnologija zasnovanih na integraciji mogućnosti upita direktno u C# jeziku, bez potrebe za poznavanjem izvora podataka.
- Pomoću sintakse upita možete izvršiti:
  - filtriranje
  - sortiranje
  - grupisanje

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Lambda izrazi i LINQ upiti

- Lambda izrazi mogu da se kombinuju i sa LINQ upitima pomoću funkcijskih delegata

```
IList<Student> studentList = new List<Student>(){...};

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

var teenStudents = studentList.Where(isStudentTeenAger).ToList<Student>();
```

Primer 9 i 10.

Deterministički object cleanup

- Oslobađanje resursa kojima ne upravlja runtime, kao što su veze ka bazi podataka.
- Implementacija interfejsa *IDisposable* i upotreba metode *Finalize*.
- Smernice za upotrebu metode *finalize:*
  - Implementirati je samo nad objektima koji zahtevaju završetak, zato što ponekad možemo implementacijom ove metode da pogoršamo performanse.
  - Implementacija interfejsa *IDisposable* kako bi omogućili izbegavanje pozivanja metode *finalize*.
  - *Finalize* metoda bi trebala da bude privatna metoda, a ne javna.
  - Trebala bi da oslobodi sve resurse koje zauzima objekat za koji se poziva, ne sme da se odnosi na druge objekte.
  - Ne treba je pozivati nad objetkom, ako nije direktna instanca klase, nego iz metode *finalize* klase izvedenice pozovite metodu *finalize* iz roditeljske klase.

Deterministički object cleanup

- IDisposable interfejs se koristi za determinističko otpuštanje resursa. On sadrži dve Dispose metode od kojih je jedna private, a druga public.
  - Metodu public Dispose će pozvati programer, koristeći tip koji implementira ovaj interfejs. Svrha ove metode je da oslobodi resurse kojima se ne upravlja, izvrši generalno čišćenje i ukaže da finalizator, ako postoji, ne mora da se pokrene.
  - Metod private Dispose sa parametrom bool se koristi interno za implementiranje otpuštanja resursa. Potrebno je da se provere parametar *disposing* i oznaka *disposed*, jer, ako je završni metod već pokrenut, treba da se otpuste samo neupravljeni resursi.
- Da bi ste bili sigurni da je javna metoda *Dispose* pozvana potrebno je da koristite *using* iskaz.

```
using (Resurs r = new Resurs(this))
{
    r.uradiNesto();
}
```

Primer 11.