PROGRAMIRANJE

priručnik za treći i četvrti razred

radna verzija

Sombor, 2024.

SADRŽAJ

UVOD

Ovaj dokument za svrhu ima da pomogne pri savladavanju gradiva objektno orijentisanog programiranja tokom izučavanja predmeta *programiranje* u trećem i četvrtom razredu smera *Elektrotehničar informacionih tehnologija*.

Trenutna verzija nije konačna i biće naknadno ispravljana i dopunjavana. Spisak izmena i dopuna tokom školske 2023/24. godine će se nalaziti na ovoj stranici.

25.10. - Ažurirano poglavlje *Nizovi*.
14.1. - Ažurirano poglavlje za rad sa chart-om.

OBJEKTNO ORIJENTISANO PROGRAMIRANJE

Objektno orijentisano programiranje (OOP) predstavlja stil programiranja u kom se problemi rešavaju upotrebom i međusobnom interakcijom objekata. Ovaj način rešavanja problema je vrlo sličan ljudskom načinu razmišljanja i rešavanju problema.

Sastoji se od :

1. identifikovanja problema,
2. identifikovanja objekata koji su potrebni za njegovo rešenje,
3. identifikovanje poruka koje će objekti međusobno slati i primati,
4. kreiranja sekvence poruka objektima, koje će rešavati problem ili probleme.

Klasa je osnovni gradivni element objektno orijentisanog programa. Ona predstavlja šablon na osnovu kog se kreiraju konkretni objekti. Opisuje se nazivom, atributima i metodama (ponašanjem).

- *Naziv* klase je uvek imenica koja označava šta klasa predstavlja (npr. Učenik, Dnevnik, Utakmica...).
- *Atributi* opisuju osobine objekta, odnosno omogućavaju nam da čuvamo informacije u njima za vreme izvršavanja programa (npr. ime, prezime, godište...).
- *Metode* opisuju ponašanje klase, koriste se da tražimo od klase da nešto uradi (npr. Ispiši, Dodaj, Obriši...).

Objekat je instanca klase (njen primerak). Objekti sadrže konkretne podatke, odnosno dodeljuju konkretne podatke atributima definisanim u klasi da bi se nakon toga definisane metode izvršavale nad tim podacima. Objekat se može smatrati promenljivom čiji je tip podataka zapravo klasa. Objekat može da menja svoje stanje i to se najčešće postiže pozivanjem metoda koje su deo objekta.

Klasa :

Naziv :
Učenik
Atributi :
ime
prezime
razred
smer
brojIzostanaka
Metode :
UpišiIzostanak()
IzmeniRazred()
IzmeniSmer()

Objekti :

Naziv :	Naziv :
u14	u22
Atributi :	Atributi :
Zoran	Danica
Vujović	Crnogorčević
3-8	4-1
El. računara	El. IT
6	29
Metode :	Metode :
UpišiIzostanak()	UpišiIzostanak()
IzmeniRazred()	IzmeniRazred()
IzmeniSmer()	IzmeniSmer()

Naziv :
Objava
Atributi :
opis
datum
Metode :
Lajkuj()
Komentariši()
Podeli()

Naziv :	Naziv :
o23	o25
Atributi :	Atributi :
obaveštenje o događaju	utisci sa letovanja
6.8.2021.	25.8.2021.
Metode :	Metode :
Lajkuj()	Lajkuj()
Komentariši()	Komentariši()
Podeli()	Podeli()

U gore prikazanim primerima prvo je predstavljena klasa *Učenik*. Ona za atribute ima osnovne podatke o učeniku dok se metode odnose na rad sa tim podacima. Kreirana metode *upišiIzostanak()* bi omogućila korisniku da izabranom učeniku doda izostanak ukoliko se on ne pojavi na času. Metode *izmeniRazred()* i *izmeniSmer()* korisniku omogućuju da izmeni razred i/ili smer ukoliko učenik pređe u viši razred ili promeni smer u školi.

Klasa *Objava* za cilj ima da predstavi objavu postavljenu na društvenu mrežu, a za atribute ima *opis* te objave i *datum* kada je objava postavljena. Metode su vezane za opcije koje se pružaju korisniku u interakciji sa

određenom objavom, označavanje da se objava sviđa korisniku metodom *Lajkuj()* komentarisanje objave metodom *Komentariši()* ili deljenje objave sa drugima na različite načine metodom *Podeli()*. Naziv objekta se dodeljuje tako da jasno označava šta predstavlja, u ovim slučajevima nazivi su davani početnim slovom klase i rednim brojem učenika u dnevniku (*u14*, *u22*) odnosno rednim brojem objave (*o23*, *o25*).

Enkapsulacija je jedan od temelja objektno orijentisanog programiranja. Svrha mu je skrivanje implementacije. Sakrivanjem implementacije se ujedno štiti i kontroliše pristup podacima koji se nalaze u objektu. Bez enkapsulacije korisnik bi imao potpun pristup objektu i njegovim podacima što bi takođe moglo dovesti do uticaja na ispravnost podataka (npr. korisnik bankovnog softvera bi lako mogao da samostalno menja iznos na svom računu).
Na primeru prethodno kreirane klase *Objava* enkapsulacija se ogleda u tome što korisnik može da poziva metode *Lajkuj()*, *Komentariši()* i *Podeli()* ali nema uvid u detaljan prikaz svega što se dešava da bi se metoda izvršila niti ima mogućnost da menja dostupne podatke, odnosno opis i datum objave.

Nasleđivanje je mogućnost da nova klasa bude izvedena iz postojeće. Time izvedena klasa preuzima atribute i metode uz mogućnost dopuna i izmena. Osnovna klasa (bazna, klasa ,,roditelj") time dobija svoje podklase (izvedena, klasa ,,dete"). Ovaj mehanizam je dobro koristiti kada prepoznamo da dve klase imaju zajedničko ponašanje, pored zajedničkih podataka. Jedna klasa može da bude nasleđena više puta. Nasleđivanje se može vršiti i nad izvedenom klasom, odnosno klasa koja nasleđuje može biti i sama nasleđena.
Na primeru prethodno kreirane klase *Objava* nasleđivanje se može predstaviti tako što kreiramo klasu *FejsbukObjava*. Klasa *FejsbukObjava* će zatim naslediti sve atribute i metode klase *Objava* što donosi uštedu u pisanju koda, odnosno nema potrebe da se iznova piše kod za atribute i metode koje su već kreirane baznom klasom.

Osnovna klasa :

Naziv :
Objava
Atributi :
opis
datum
Metode :
Lajkuj()
Komentariši()
Podeli()

Izvedena klasa :

Naziv :
FejsbukObjava
Atributi :
opis
datum
vrsta
Metode :
Lajkuj()
Komentariši()
Podeli()
Prijavi()

Možemo primetiti da nasleđivanjem klase *Objava* izvedena klasa *FejsbukObjava* automatski nasleđuje sve njene atribute (*opis*, *datum*) kao i metode (*Lajkuj()*, *Komentariši()*, *Podeli()*) i pored toga sadrži sopstvene atribute (*vrsta*) i metode (*Prijavi()*).

Polimorfizam predstavlja koncept prema kom se metoda ponaša različito u različitim objektima. S tim u vidu polimorfizam nam omogućuje da redefinišemo kako nešto radi.
Na primeru klase *Objava* ukoliko bi pored izvedene klase *FejsbukObjava* kreirali i klasu *InstagramObjava* sve tri klase bi sadržale metodu bazne klase *Lajkuj()*. Polimorfizam bi omogućio da menjamo tekst koji korisnik dobija u vidu obaveštenja nakon što neko pozove metodu *Lajkuj()* odnosno označi da mu se sviđa sadržaj korisnika. Ukoliko bi neko označio da mu se sviđa vaša objava u slučaju klase *FejsbukObjava* na Vaš fejsbuk profil bi stiglo obaveštenje „Ime prezime se sviđa Vaš status/snimak/fotografija" dok bi u slučaju klase *InstagramObjava* na Vaš instagram profil stiglo obaveštenje „Korisnik nadimak kaže da mu se sviđa vaša fotografija". Tako bi zahvaljujući polimorfizmu mogli da iskoristimo istu nasleđenu metodu u oba slučaja prilagođenu posebno izvedenoj klasi, u ovom slučaju da manipulišemo sa tekstom obaveštenja bez da kreiramo novu metodu.

C# PROGRAMSKI JEZIK

Osnove sintakse

- Naredba predstavlja iskaz kojim se zadaje komanda računaru da izvrši neku operaciju. Kraj svake naredbe završava se simbolom tačka-zarez ("**;**").
- Naredbe se mogu pisati jedna za drugom u istom redu, ali najčešće se pišu u posebnim redovima.
- Naredbe se pišu u blokovima oivičenim vitičastim zagradama **{ }**. Blokovi određuju oblast vidljivosti promenljivih, odnosno oblast dejstva nekog koda. Moguće je da postoji više ugnježdenih blokova.
- Programski kod C# jezika razlikuje velika i mala slova ("case sensitive" – ime i Ime nisu iste promenljive).
- Komentari se koriste da bi se neki deo programskog koda objasnio. Koristimo **//** za svaki red ili **/* */** za blok.

Tipovi podataka

- **int** – brojčana vrednost
- **float** – decimalna brojčana vrednost
- **double** – decimalna brojčana vrednost (precizniji od float)
- **char** – jedan karakter
- **string** – niz karaktera, odnosno tekst
- **bool** – logički tip, odnosno provera tačnosti

Promenljive

Promenljive predstavljaju nazive koji su dati memorijskim lokacijama u kojima se smeštaju podaci. Podaci u tim promenljivima se u toku izvršavanja programa mogu menjati. Promenljive su određene imenom, memorijskom lokacijom, tipom podatka i vrednošću.
Pravila prilikom deklaracije/definicije promenljive :

- Naziv promenljive mora početi ili slovom ili donjom crtom, a nakon toga u sebi može sadržati slova, brojeve i donje crte.
- Nije dozvoljen razmak unutar naziva promenljive.
- Rezervisane reči poput int, float, using se ne mogu koristiti kao nazivi promenljivih.
- Kada je promenljiva sa određenim nazivom deklarisana taj naziv se više ne može koristiti za deklarisanje novih promenljivih.

Razlikujemo:

- Deklaracija promenljive – određivanje imena i tipa podatka promenljive.

```
int broj;
float decimalanBroj;
char karakter;
string tekst;
bool proveraTacnosti;
```

- Definicija promenljive – uključuje deklaraciju i dodeljivanje početne vrednosti.

```
int broj = 10;
float decimalanBroj = 0.7F;
char karakter = 'A'; (piše se između po jednog navodnika)
string tekst = "Pise se izmedju navodnika";
bool proveraTacnosti = true;
```

- Deklaracija više promenljivih istog tipa podataka

```
int x, y, z;
```

- Definicija više promenljivih istog tipa podataka

```
int x=1, y=2, z=3;
```

Naredbe

jedna naredba - naredba;
blok naredbi – { blok naredbi }

Operatori

+ sabiranje, spajanje stringova, spajanje stringova i promenljivih

- oduzimanje

***** množenje

/ deljenje

% ostatak od deljenja

+= x+=2 je isto što i x=x+2

-= x-=2 je isto što i x=x-2

***=** x*=2 je isto što i x=x*2

/= x/=2 je isto što i x=x/2

%= x%=2 je isto što i x=x%2

++ x++ je isto što i x=x+1

-- x-- je isto što i x=x-1

== jednakost – vraća vrednost *tačno* ako su vrednosti promenljivih identične

!= nije jednako – vraća vrednost *tačno* ako su vrednosti promenljivih različite

< manje – vrednost izraza je *tačna* ako je promenljiva sa leve strane znaka manja od one sa desne strane

> veće - vrednost izraza je *tačna* ako je promenljiva sa leve strane znaka veća od one sa desne strane

<= manje ili jednako – vrednost izraza je *tačna* ako je promenljiva sa leve strane znaka manja ili jednaka od one sa desne strane

>= veće ili jednako - vrednost izraza je *tačna* ako je promenljiva sa leve strane znaka veća ili jednaka od one sa desne strane

&& uslovno logičko AND - ako je prvi operand *netačno*, drugi se ni ne proverava

|| uslovno logičko OR - ako je prvi operand *tačno*, drugi se ni ne proverava

t ? x : y ako je t tačno, proveri i vrati vrednost x, inače proveri i vrati vrednost y

Ispisivanje na ekran

Console.Write("Tekst"); ispisuje se sve između navodnika

Console.WriteLine("Tekst"); ispisuje se sve između navodnika dok se naredni ispis prebacuje u sledeći red

Console.WriteLine("{0} je veće od {1} dok je {2} jednako {3}", a, b, c, d);

Ispisuje vrednost promenljivih a, b, c i d redosledom kojim su nabrojane nakon zatvorenih navodnika i zareza, a na mestima brojeva oivičenih vitičastom zagradom unutar navodnika.

Console.WriteLine("Rezultat je: " + zbir);

Ispisuje vrednost promenljive zbir nakon teksta između navodnika.

Učitavanje iz konzole

Console.ReadLine();

Učitava sve što korisnik ispiše u konzolu pre nego što pritisne *Enter* (u kom god da je obliku čita ga kao promenljivu tipa string).

Console.ReadKey();

Čeka da korisnik pritisne neko dugme pre nego što nastavi sa izvršavanjem naredbi ili zatvaranjem programa.

Transformacija tipova podataka

int.Parse(nazivPromenljive);

Menja vrednost tipa podataka string u integer, ako je ta vrednost broj.

float.Parse(nazivPromenljive);

Menja vrednost tipa podataka string u float, ako je ta vrednost broj.

double.Parse(nazivPromenljive);

Menja vrednost tipa podataka string u double, ako je ta vrednost broj.

char.Parse(nazivPromenljive);

Menja vrednost tipa podataka string u char, ako je ta vrednost jedan karakter.

(float)nazivPromenljive;

Menja vrednost tipa podataka integer u float (isto je izvodivo i za double).

(int)nazivPromenljive;

Menja vrednost tipa podataka float ili double u integer.

nazivPromenljive.ToString(".##");

Menja vrednost decimalnog broja u string i određuje broj decimala (koliko # toliko će se decimala prikazivati).

Convert.ToString(nazivPromenljive);

Menja vrednost promenljive u string.

if

if (uslov) {naredba;}

Ako se ispuni uslov izvršava se naredba, u suprotnom program se nastavlja bez izvršavanja navedene naredbe (isto važi i za blokove naredbi)

if (uslov) {naredba;} else {naredba2;}

Ako se ispuni uslov izvršava se naredba, u suprotnom izvršava se druga navedena naredba (isto važi i za blokove naredbi)

if (uslov) {if (uslov2) {naredba;} else {naredba2;}} else {naredba3;}

If uslov može da bude ugnježden unutar drugog if uslova što znači da ukoliko prvi uslov nije tačan ide se na izvršavanje naredba3, a ukoliko jeste proverava se uslov2 i u zavisnosti od njegove tačnosti izvršavaju se ili naredba ili naredba2.

if (uslov) {naredba;} else if (uslov2) {naredba2;} else {naredba3;}

U ovom slučaju se nakon provere uslova izvršava naredba ako je uslov tačan, a ako nije proverava se naredni uslov, uslov2, i ukoliko je on tačan izvršava se naredba2, a ukoliko nije naredba3.

switch

```
switch (nazivPromenljive)
{
case vrednost1:
naredba;
break;
case vrednost2:
naredba2;
break;
default :
naredba3;
break;
}
```

Switch se izvršava nad promenljivom, proverava se vrednost i ukoliko jeste identična postavljenoj nakon reči *case* izvršava se naredba a reč *break* zaustavlja izvršavanje svega što je unutar switch bloka nakon toga. Ukoliko vrednost promenljive nije jednak nekoj od ponuđenih vrednosti program izvršava naredbu iz *default* dela. Default deo nije obavezno postaviti.

```
switch (nazivPromenljive)
{
case vrednost1:
case vrednost2:
naredba;
```

```
break;
case vrednost3:
naredba2;
break;
default :
naredba3;
break;
}
```

Ukoliko je potrebno proveriti više vrednosti pre izvršavanja neke naredbe provere se pišu jedna ispod druge a naredba (ili blok naredbi) koja treba da se izvrši nakon poslednje provere.

while

while (uslov) {naredba;} Ukoliko je uslov ispunjen naredba će se izvršiti i vratiti se na proveru uslova nakon čega će se ponovo izvršiti ukoliko je uslov i dalje ispunjen a ukoliko nije preskočiće se izvršavanje naredbe.

do while

do {naredba;} while (uslov) Nakon izvršavanja naredbe proveriće se uslov, ukoliko je uslov ispunjen ponovo će se izvršiti ista naredba i ponovo će se proveravati uslov nakon izvršenja sve dok uslov ne prestane biti ispunjen.

for

for (inicijalizacija; uslov; poslednja komanda)
{ naredbe; } Unutar for petlje se prvenstveno upisuje inicijalizacija promenljive, a nakon toga uslov koji se proverava za pomenutu promenljivu i izvršava poslednja komanda pre nove provere uslova.

Randomiziranje broja

Random imeObjekta = new Random();

int imePromenljive = imeObjekta.Next(min, max);

Ovim kreiramo objekat klase Random, a promenljivoj dodeljujemo vrednost nakon poziva metode Next. Ukoliko unutar zagrada postavimo vrednosti za minimum i maksimum promenljiva će dobiti nasumičnu vrednost broja koji spada između minimuma i maksimuma (bez vrednosti maksimuma).

NIZOVI

- Deklarisanje niza:

tipPodatka[] nazivNiza;
Ovom naredbom je samo deklarisan niz - C#-u još uvek nije saopšteno da treba da kreira niz sa specificiranim brojem elemenata.

- Inicijalizovanje niza:

tipPodatka[] nazivNiza = new tipPodatka[brojČlanovaNiza];
Operatorom new saopštava se C#-u da treba da kreira određeni broj promenljivih izabranog tipa podataka i da treba da omogući nizu pristup do njih.

- Načini dodeljivanja vrednosti inicijalizovanom nizu:

nazivNiza[indeks] = vrednost;

- Načini dodeljivanja vrednosti prilikom inicijalizacije niza:

tipPodatka[] nazivNiza = {vrednosti};
tipPodatka[] nazivNiza = new tipPodatka[] {vrednosti};

nazivNiza.Length

Daje broj članova niza.

Array.Sort(nazivNiza);

Sortira niz.

Definisanje nizovne promenljive kada znamo koji su članovi niza:

```
// Prvi način
string[] predmeti = { "Srpski", "Matematika", "Programiranje" };
int[] ocene = { 4, 3, 5 };
// Drugi način
string[] predmeti = new string[] { "Srpski", "Matematika", "Programiranje" };
// Treći način
string[] predmeti = new string[3];
predmeti[0] = "Srpski";
predmeti[1] = "Matematika";
predmeti[2] = "Programiranje";
int[] ocene = new int[3];
ocene[0] = 4;
ocene[1] = 3;
ocene[2] = 5;
```

Definisanje nizovne promenljive kada znamo samo broj članova niza:

```
string[] predmeti = new string[3];
int[] ocene = new int[3];
```

Definisanje nizovne promenljive kada korisnik unosi broj članova niza:

```
// Niz uvek definisati tek nakon što imamo vrednost za promenljivu koju smeštamo u uglaste zagrade
Console.WriteLine("Unesite broj članova niza:");
brojClanova = int.Parse(Console.ReadLine());
string[] predmeti = new string[brojClanova];
```

Učitavanje članova niza od korisnika (bez korišćenja for i foreach petlje) :

```
string[] predmeti = new string[3];
Console.WriteLine("Unesite članove za niz predmeti:");
predmeti[0] = Console.ReadLine();
predmeti[1] = Console.ReadLine();
predmeti[2] = Console.ReadLine();
```

Ispis članova niza (bez korišćenja for i foreach petlje) :

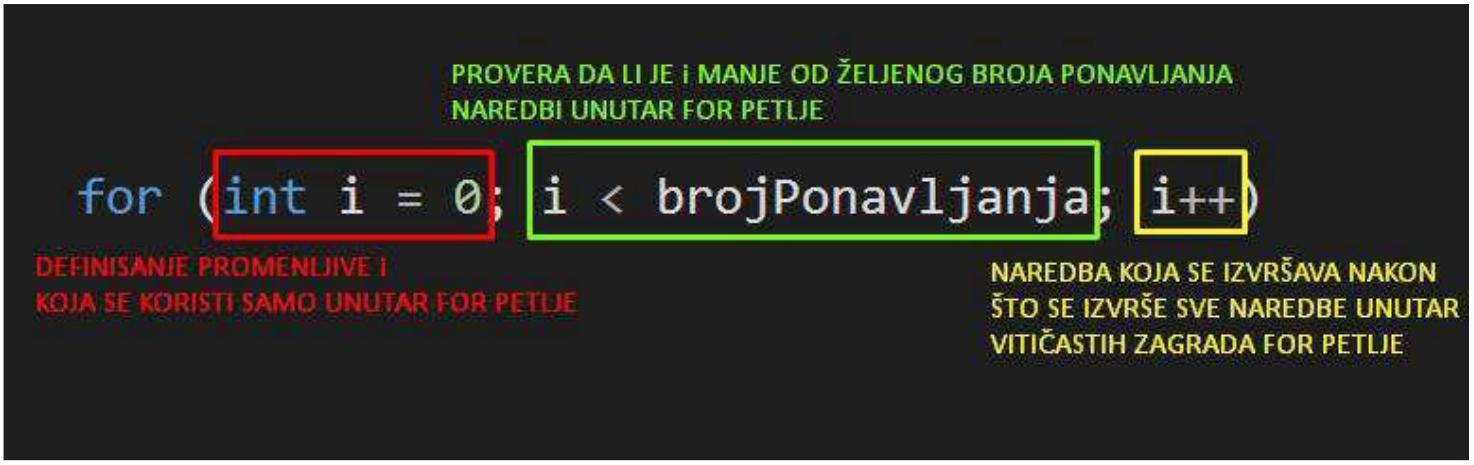
```
Console.WriteLine("Članovi niza predmeti su:");
Console.WriteLine(predmeti[0]);
Console.WriteLine(predmeti[1]);
```

```
Console.WriteLine(predmeti[2]);
```

Sortiranje niza (ako je niz tekstualni sortira se po abecednom redu, ako je numerički od manjeg ka većem broju) :

```
Array.Sort(predmeti);
```

for



Primer – ukoliko želimo da tri puta ispišemo vrednost tekstualne promenljive:

```
string smer = "Elektrotehnicar racunara";
for (int i = 0; i < 3; i++)
{
Console.WriteLine(smer);
}
```

Isto se moglo postići i sa while petljom :

```
int i = 0;
while (i < 3)
{
Console.WriteLine(smer);
i++;
}
```

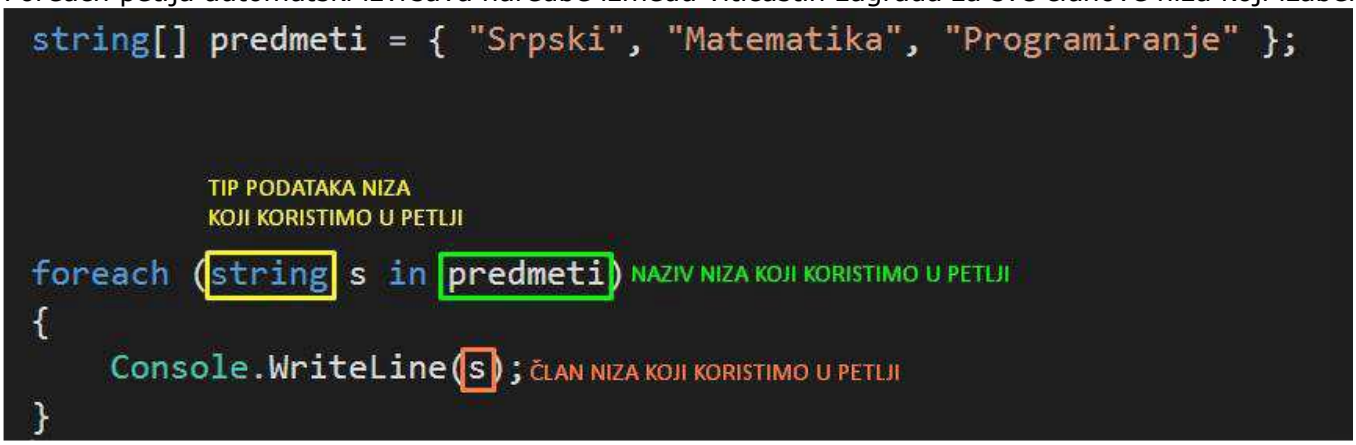
Međutim kada znamo broj ponavljanja preporučljivo je koristiti for petlju.
Primer for petlje u radu sa nizovima:

```
// nizA.Length predstavlja broj članova niza A što znači da će se petlja ponavljati za svaki od
članova
for (int i = 0; i < nizA.Length; i++)
{
// nizA[i] predstavlja člana niza sa indeksom i (indeks kreće od nule) - u ovom slučaju se
učitava vrednost iz konzole i smešta unutar nizovne promenljive
nizA[i] = int.Parse(Console.ReadLine());
}
```

foreach

foreach (tipPodataka nazivPromenljive in nazivNiza)
{ naredbe; } Izvršava naredbe za sve članove niza.

Foreach petlja automatski izvršava naredbe između vitičastih zagrada za sve članove niza koji izaberemo:



Binarna pretraga elementa u nizu

Binarno pretraživanje je nalaženje zadate vrednosti u sortiranom nizu elemenata. U svakom koraku, dok se ne pronađe tražena vrednost, skup se deli na dva dela i pretraga se nastavlja samo u jednom – odbacuje se deo koji sigurno ne sadrži zadatu vrednost.

Pretraživanje se vrši na sledeći način: pronalazi se srednji element niza, proverava se da li je jednak zadatoj vrednosti i ako jeste vraća se njegov indeks, a ako nije pretraživanje se nastavlja nad nizom svih manjih (ako je srednji element niza veći od zadate vrednosti) ili svih većih elemenata (ako je srednji element niza manji od zadate vrednosti).

Uslov za binarno pretraživanje jeste da niz bude sortiran u rastućem poretku. Ukoliko niz nije sortiran poziv funkcije za binarno pretraživanje će vratiti neočekivanu vrednost.

Funkcija za binarno pretraživanje se poziva na sledeći način : `Array.BinarySearch(ime niza, traženi broj);`

```
static void Main(string[] args)
   {
       int[] brojevi = { 7, 16, 2, 48, 15, 20 };

       int x = Array.BinarySearch(brojevi, 16); // x će dobiti neočekivanu vrednost jer niz brojevi
nije sortiran

       int[] cifre = { 5, 8, 12, 17, 23, 30, 38 };

       int y = Array.BinarySearch(cifre, 38); // y će dobiti vrednost 6 jer je to indeks traženog broja
   }
```

Dvodimenzionalni nizovi (matrice)

Dvodimenzionalan niz je niz koji ima elemente koji su nizovi skalarnih podataka. Dvodimenzionalan niz se još naziva i matrica. To su nizovi koji se sastoje iz vrsta i kolona.

```
// Načini kreiranja matrice
int[,] matrica = new int[3, 3];
int[,] matrica = new int[,] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
int[,] matrica = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Prva cifra unutar uglastih zagrada predstavlja broj redova, a druga broj kolona.

```
// Matrica sa dva reda i tri kolone
int[,] matrica = new int[2, 3] { { 1, 5, 9 }, { 2, 6, 10 } };
```

1	5	9
2	6	10

matrica[0,0]	matrica[0,1]	matrica[0,2]
matrica[1,0]	matrica[1,1]	matrica[1,2]

```
// Pristup svim elementima matrice preko for petlji
for (int i = 0; i < matrica.GetLength(0); i++) // Pristup redovima
    for (int j = 0; j < matrica.GetLength(1); j++) // Pristup kolonama
    {
        Console.WriteLine(matrica[i, j]); // Pristup i ispis elementa matrice
    }

// Pristup svim elementima matrice preko foreach petlje
foreach (int broj in matrica) Console.WriteLine(broj);
```

GENERIČKA LISTA – List<T>

Lista je skup objekata, istog tipa – generička lista, bez unapred poznatog broja, odnosno sa mogućnošću da se naknadno u već formiranu listu dodaju ili izbacuju elementi.

Svaka lista ima sledeće osobine :

- Lista može biti prazna,
- Moguće je ubaciti novi element na bilo koju poziciju u listi,
- Moguće je izbaciti bilo koji element liste,
- Lista može da ima svoju veličinu, tj. broj elemenata

Nizovi i liste su namenjeni za čuvanje i pretraživanje velikog broja elemenata istog tipa. Osnovna razlika između niza i liste jeste što broj elemenata u listi nije fiksan.

```
// Ispravni načini kreiranja liste
List<string> polja = new List<string>();
List<int> ocene = new List<int>(10);
List<char> znakovi = new List<char>() { '!', '@', '#' };
```

Metode listi

- **Add(podatak tipa kog je i lista)** - Dodavanje elemenata na kraj liste
- **Count()** - Vraćanje broja elemenata liste (prebrojavanje)
- **Insert(indeks na koji želimo da ubacimo element, podatak tipa kog je i lista)** – Ubacivanje elementa na izabrano mesto
- **RemoveAt(indeks sa kog želimo da uklonimo element)** – Uklanjanje elementa sa izabranog mesta

```
List<int> ocene = new List<int>() { 3, 5, 5, 4, 5 };

// Dodavanje elementa na kraj liste
ocene.Add(5);
// Dodavanje elementa na poziciju sa izabranim indeksom
ocene.Insert(2, 5);
// Uklanjanje elementa sa izabranog indeksa
ocene.RemoveAt(0);
// Prebrojavanje članova liste
int brojOcena = ocene.Count;
```

METODE

Metode (funkcije) koristimo kako bi mogli da ih koristimo umesto ponovnog pisanja istog koda u momentima kada nam je potrebno da se taj kod izvrši.

Kreiranje metode bez povratne vrednosti i bez parametara

```
static void IspisiPoruku()
{
Console.WriteLine("Pogrešan unos.");
}
```

`void` – označava da metoda ne vraća vrednost već se samo izvršava kod unutar nje
`IspisiPoruku()` – naziv metode, zagrade su prazne jer nema parametara koji se preuzimaju iz glavnog programa
Pozivanje metode bez povratne vrednosti i bez parametara : `IspisiPoruku();`

Kreiranje metode bez povratne vrednosti i sa parametrima

```
static void IspisiIme(string imeKorisnika)
{
Console.WriteLine("Dobrodošli {0}!", imeKorisnika);
}
```

`(string imeKorisnika)` – označava da će prilikom poziva metode biti potrebno proslediti parametar tekstualnog tipa.

Pozivanje metode bez povratne vrednosti i sa parametrima : `IspisiIme(ime);`

Promenljiva koju prosleđujemo metodi ne mora da bude istog naziva kao i naziv parametra prilikom kreiranja metode (u ovom slučaju u metodi je naziv stringa imeKorisnika, a unutar glavnog programa ime promenljive je samo ime). Takođe kao parametar možemo da prosledimo i samo vrednost koja je tipa podataka kao i parametar koji se traži (`IspisiIme("Tamara")`).

Kreiranje metode sa povratnom vrednošću i bez parametara

```
static string UcitajUnos()
{
return Console.ReadLine();
}
```

`string` – označava tip povratne vrednosti metode odnosno onoga što dobijamo nakon što pozovemo metodu
`return` – piše se pre promenljive ili izraza koji će činiti povratnu vrednost metode

U gornjem primeru smo praktično samo preimenovali metodu Console.ReadLine(), tako da na isti način možemo koristiti metodu UcitajUnos().
Primer pozivanja gore kreirane metode : `broj = int.Parse(UcitajUnos());`

Kreiranje metode sa povratnom vrednošću i sa parametrima

```
static int PronadjiMaksimum(int broj1, int broj2)
{
int max;
if (broj1 > broj2)
{
max = broj1;
}
else
{
max = broj2;
}
return max;
}
```

Pozivanje metode sa povratnom vrednošću i sa parametrima : `PronadjiMaksimum(x, y);`

Metoda sa povratnom vrednošću se tretira kao promenljiva tipa podataka istog kojeg je tipa i sama metoda. To znači da bi nad metodom iz gornjeg primera mogli vršiti matematičke operacije jer se radi o numeričkoj promenljivoj, npr. :
PronadjiMaksimum(x,y) * 2 – bi dalo kvadrat većeg od dva broja.
Console.WriteLine(PronadjiMaksimum(x,y)) – ukoliko bi želeli da ispišemo na ekran vrednost većeg broja.

Preklapanje imena metode (preopterećenje metode, overloading)

Isto ime metode možemo koristiti ukoliko imamo različiti tip parametara ili različiti broj parametara.

```
static void Saberi(int x, int y)
{
Console.WriteLine("Zbir je {0}", x + y);
}
static void Saberi(int x, int y, int z)
{
Console.WriteLine("Zbir je {0}", x + y + z);
}
```

Ukoliko koristimo različiti broj parametara prilikom kreiranja metode možemo koristiti isti naziv metode jer će program automatski pozvati ispravnu metodu u zavisnosti od toga koliko promenljivih stavimo u zagradu prilikom pozivanja :

```
Saberi(2, 8); // Poziva metodu sa dva parametra (x i y)
Saberi(1, 3, 5); // Poziva metodu sa tri parametra (x, y i z)
```

Drugi način preklapanja imena metode :

```
static int Saberi(int x, int y)
{
return x + y;
}
static float Saberi(float x, float y)
{
return x + y;
}
```

Ukoliko koristimo različit tip podataka za povratnu vrednost možemo koristiti isti naziv metode, a program će na osnovu tipa podataka promenljivih unutar zagrada pozvati ispravnu metodu :

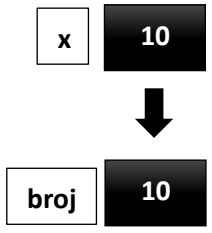
```
Saberi(6, 4); // Poziva metodu koja sabira cele brojeve
Saberi(2.5F, 1.3F); // Poziva metodu koja sabira decimalne brojeve
```

Prenos parametara po vrednosti i referenci

Prilikom poziva metode koja zahteva parametre razlikujemo prenos parametara po vrednosti i po referenci.
Tipovi podataka koji se kao parametri prenose po vrednosti : int, float, string, char
Tipovi podataka koji se kao parametri prenose po referenci : svi nizovni tipovi podataka

Primer prenosa parametara po vrednosti :

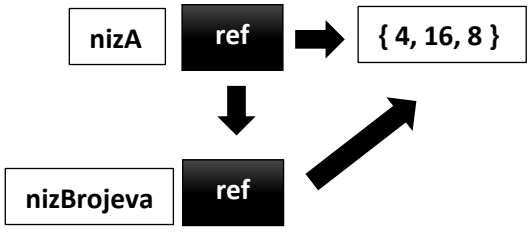
```
    public class Program
    {
        public static void Main(string[] args)
        {
            int x = 10;
            // Na ekran će se ispisati broj 10
            Console.WriteLine(x);
            /* Vrednost promenljive x (10) će se dodeliti promenljivoj broj metode Povecaj
               Nakon toga se naredba broj++ izvršava nad promenljivom broj i bez ikakvog uticaja na
promenljivu x */
            Povecaj(x);
            // Na ekran se ponovo ispisuje broj 10, s obzirom da nije bilo promene vrednosti
promenljive x
            Console.WriteLine(x);
        }

        static void Povecaj(int broj)
        {
            broj++;
        }
    }
```



Nakon poziva metode Povecaj vrednost promenljive x se prebacuje u promenljivu broj i tu prestaje svaka veza sa promenljivom x.

Primer prenosa parametara po referenci :

```
    public class Program
    {
        public static void Main(string[] args)
        {
            int[] nizA = { 4, 16, 8 };
            // Na ekran se ispisuju vrednosti članova niza nizA : 4, 16, 8
            foreach (int i in nizA)
            {
                Console.WriteLine(i);
            }
            // Pozivom metode PrebaciUNegativno prosleđuje joj se referenca ka nizu nizA
            PrebaciUNegativno(nizA);
            // Na ekran se ispisuju vrednosti članova niza nizA nakon izmena u metodi PrebaciUNegativno
: -4, -16, - 8
            foreach (int i in nizA)
            {
                Console.WriteLine(i);
            }
        }

        static void PrebaciUNegativno(int[] nizBrojeva)
        {
            for (int i = 0; i < nizBrojeva.Length; i++)
            {
                nizBrojeva[i] = nizBrojeva[i] * -1;
            }
        }
    }
```

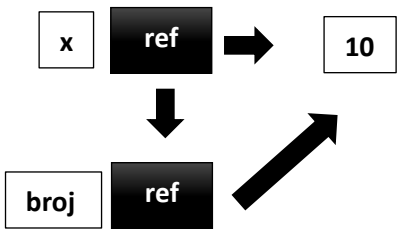


Parametri nizovnog tipa se prenose po referenci što znači da sve naredbe koje se izvršavaju unutar metode se izvršavaju unutar samog niza koji je naveden kao parametar i iz tog razloga direktno utiču na njega.

Ukoliko želimo da neki od tipova podataka koji se prenose po vrednosti (int, float, char, string) prenesemo po referenci dodajemo službenu reč **ref** pre deklarisanja parametra i unutar poziva metode.

Primer :

```
    public class Program
    {
        public static void Main(string[] args)
        {
            int x = 10;
            // Na ekran će se ispisati broj 10
            Console.WriteLine(x);
            /* Metodi Povecaj će se proslediti referenca ka promenljivoj x
               Nakon toga se naredba broj++ izvršava nad promenljivom x i menja njenu vrednost */
            Povecaj(ref x);
            // Na ekran se ponovo ispisuje broj 11
            Console.WriteLine(x);
        }

        static void Povecaj(ref int broj)
        {
            broj++;
        }
    }
```



Metode sa promenljivim brojem parametara

Da bi smo kreirali metodu sa promenljivim brojem parametara moramo koristiti ključnu reč params.

Params – ključnu reč params koristimo kada pišemo metod i ne znamo koliko parametara ćemo imati. Nakon korišćenja ključne reči params navodimo tip podatka (tip params parametra mora biti jednodimenzionalnog (nizovnog) tipa) i naziv parametra nakon čega se ne sme navoditi više parametara. Ključnu reč params

možemo koristiti samo jednom po metodi. Drugi parametri se mogu navoditi pre korišćenja ključne reči params.

```
        // Ispravni načini korišćenja ključne reči params

        public void Posalji(params string[] poruke) { }

        public void Primi(int broj, string primalac, params string[] poruke) { }
```

Rekurzivne metode

Rekurzivan metod je onaj metod koji u nekoj svojoj instrukciji sadrži poziv samog sebe. Svakako, prilikom kreiranja rekurzivnog metoda moramo voditi računa da ne dođe do beskonačne rekurzije. Uvek moramo imati proveru da li je ispunjen uslov za napuštanje rekurzije (ili da li je ispunjen uslov za dalji ulazak u rekurziju).

```
        void/string/int... Metod(parametar1, parametar2 ...)
        {
            if (uslov za bazni slučaj)
            {
                //bazni slučaj
            }
            //rekurzivni deo
        }
```

Bazni slučaj : Najmanja verzija problema za koju već znamo rešenje ili krajnji uslov kada funkcija može odmah vratiti rezultat.

Rekurzivni deo : Pronalaženje rešenja problema putem rešenja njegovih manjih potproblema. Ovde funkcija poziva samu sebe da razbije trenutni problem na jednostavniji nivo.

```
        // Metoda koja proverava da li je prosleđeni string palindrom
        public static bool palindrom(string s)
        {
            // Ako prosleđeni string ima samo jedan ili nijedan karakter znači da je sigurno palindrom
            if (s.Length <= 1) return true; //bazni slučaj
            // Ako je prosleđeni string duži od jednog karaktera proveravamo jednakost prvog i
poslednjeg slova
            // Ako prvo i poslednje slovo nisu jednak reč nije palindrom
            else if (s[0] != s[s.Length - 1]) return false;
            // Ako prvo i poslednje slovo jesu jednaki pozivamo funkciju palindrom, a string skraćujemo
za prvo i poslednje slovo
            else return palindrom(s.Substring(1, s.Length - 2)); // rekurzija
        }
```

Korisne metode za rad sa tekstom

Zahvaljujući .NET framework-u možemo koristiti brojne metode bez da ih samostalno kreiramo. Neke od onih najčešće korišćenih pri radu sa tekstom su :

- ResetText() – Resetuje vrednost tekstualnog svojstva na početnu vrednost (*nazivPolja.ResetText();*).
- ToLower() – Menja izabrani tekst u samo mala slova (*tekst.ToLower();*).
- ToUpper() – Menja izabrani tekst u samo velika slova (*tekst.ToUpper();*).

UPRAVLJANJE GREŠKAMA I IZUZECIMA

Greške možemo podeliti na tri grupe :
1. **Sintaksne greške (syntax errors)** – rezultat pogrešno napisane naredbe, nedeklarisane varijable i slično. Ove greške najčešće nisu problematične jer program koji ih sadrži ni ne može da se kompajlira (streng umesto string, zaboraviti ; na kraju naredbe...).
2. **Greške u vreme izvršavanja (runtime errors)** – greške koje se mogu ali i ne moraju dogoditi za vreme izvršavanja programa. Najčešće su rezultat neproverenih vrednosti promenljivih i pogrešnog unosa korisnika (uneti string kada kompajler očekuje integer, pokušati pristupiti članu niza sa indeksom koji je van granica tog niza...).
3. **Logičke greške (logical errors)** – program funkcioniše, ne događaju se nikakve greške, ali dobijaju se pogrešni rezultati. Ovo su greške koje je veoma teško naći i koje nastaju zbog pogrešno projektovanog programa ili algoritma (ispravno napisani ali pogrešno postavljeni uslovi, loš redosled naredbi unutar programa...).

Obrada greške se u C# vrši pomoću naredbi *try* i *catch*.

```
try
{
// linije koda koje mogu da generišu grešku
}
catch (Exception naziv)
{
// obrada greške
}
```

Unutar naredbe finally se ispisuje kod koji će se izvršiti bez obzira da li do greške dođe ili ne.

```
try
{
// linije koda koje mogu da generišu grešku
}
catch (Exception naziv)
{
// obrada greške
}
finally
{
// kod koji se izvršava bez obzira da li do greške dođe ili ne
}
```

Finally naredbu nije obavezno pisati. Takođe mogu se pisati i samo naredbe try i finally međutim to neće sprečiti prekid programa ukoliko dođe do greške.

Logika mehanizma :

1. Tok izvršavanja prelazi na try blok
2. Ukoliko unutar try bloka ne dođe do greške izvršavanje se nastavlja do kraja ovog bloka, a nakon toga se izvršavanje prenosi na finally blok
3. Ukoliko dođe do greške unutar try bloka, izvršavanje se prenosi na catch blok
4. U catch bloku se vrši obrada greške. Na kraju catch bloka izvršavanje se automatski prenosi na finally blok
5. Finally blok se izvršava

Pošto postoji dosta različitih tipova izuzetaka, moguće je navesti i više catch blokova, pri čemu svaki catch blok obrađuje određeni tip izuzetka.

```
public static void Main(String[] args)
{
try
{
int x, y;
Console.WriteLine("Unesite vrednosti za x i y:");
x = int.Parse(Console.ReadLine());
y = int.Parse(Console.ReadLine());
Console.WriteLine("Količnik je : ", x / y);
}
catch (DivideByZeroException izuzetak)
{
// obrada greške deljenja sa nulom
Console.WriteLine("Ne možete deliti sa nulom!");
}
catch (Exception izuzetak)
{
// obrada bilo koje druge greške sem greške deljenja sa nulom
Console.WriteLine("Došlo je do greške!");
}
}
```

Automatski se hvata instanca izuzetka i prosleđuje isključivo na osnovu njenog tipa odgovarajućem catch bloku.

Prilikom pisanja catch blokova mora se obratiti pažnja na redosled navođenja. Izvršava se jedan catch blok i to prvi koji je kompatibilan, stoga se navođenje vrši od najspecifičnijeg izuzetka ka najopštijem izuzetku. Ukoliko se navede prvo opštiji, a zatim specifičniji tip izuzetka program neće moći da se kompajlira.

Catch blok čiji je tip izuzetka Exception se naziva opšti catch blok. Koristi se da bi se uhvatili izuzeci koji nisu obrađeni posebnim catch blokovima. On može da obradi bilo koji izuzetak bez obzira na njegov tip, obzirom da su svi izuzeci izvedeni iz klase System.Exception. Može se navesti samo jedan opšti catch blok i, ukoliko je naveden, mora biti poslednji u nizu. Ne smeju se navesti dva catch bloka koji imaju isti tip izuzetka.

Message koristimo kako bi došli do opisa greške koja je “uhvaćena”.

```
try
{
int x, y;
Console.WriteLine("Unesite vrednosti za x i y:");
x = int.Parse(Console.ReadLine());
y = int.Parse(Console.ReadLine());
Console.WriteLine("Količnik je : ", x / y);
}
catch (DivideByZeroException izuzetak)
{
// obrada greške deljenja sa nulom
Console.WriteLine("Ne možete deliti sa nulom!");
}
// opšti catch blok
catch (Exception izuzetak)
{
// Message koristimo da prikažemo opis greške koja je "uhvaćena"
Console.WriteLine(izuzetak.Message);
}
```

Eksplicitni izuzeci - nastaju navođenjem naredbe *throw* usled čega istog trenutka i bezuslovno nastaje izuzetak. Bitno je izabrati onu klasu izuzetka koja u najboljoj meri opisuje nastalu grešku. Nakon podizanja izuzetka koji je naveden posle *throw* naredbe naredba koja sledi naredbu *throw* nikada neće biti izvršena.

```
try
{
int x, y;
Console.WriteLine("Unesite vrednosti za x i y:");
x = int.Parse(Console.ReadLine());
y = int.Parse(Console.ReadLine());
if (y == 0)
{
// eksplicitno bacanje izuzetka DivideByZeroException
throw new DivideByZeroException("Poruka koja će se ispisati nakon pozivanja Message-a");
}
Console.WriteLine("Količnik je : ", x / y);
}
catch (DivideByZeroException izuzetak)
{
// ispisaće se poruka koju smo sami postavili prilikom eksplicitnog bacanja izuzetka
Console.WriteLine("Ne možete deliti sa nulom! {0}", izuzetak.Message);
}
catch (Exception izuzetak)
{
Console.WriteLine(izuzetak.Message);
}
```

Često korišćene klase izuzetaka :

ArithmeticException – osnovna klasa za izuzetke koji nastaju prilikom izvršavanja aritmetičkih operacija kao što su *DivideByZeroException* i slični.
DivideByZeroException – podiže se prilikom pokušaja deljenja celobrojne vrednosti nulom.
ArrayTypeMismatchException – podiže se prilikom pokušaja da se u niz ubaci element čiji tip nije kompatibilan sa tipom elementa niza.
IndexOutOfRangeException – podiže se prilikom pokušaja pristupa nepostojećem elementu niza.
NullReferenceException – podiže se prilikom pokušaja korišćenja nepostojećeg objekta.
ArgumentException – podiže se kada vrednost nekog od prosleđenih parametara nije ispravna.

KLASE U C#

Koristi se ključna reč **class** iza koje se navodi ime klase, a zatim se unutar vitičastih zagrada navode članovi klase. Članom klase se podrazumeva bilo koji podatak ili funkcija koji su definisani unutar klase. Pod funkcijom se podrazumeva bilo koji član koji sadrži kôd (metode, svojstva, konstruktori i preklopljeni operatori). Redosled navođenja članova klase nije značajan.

```
// kreiranje klase – reč class nakon čega ide naziv klase
class Prozor
{
    // atributi klase
    float duzina;
    float sirina;
    bool jeOtvoren;

    // metode klase
    void Otvori()
    {
        jeOtvoren = true;
    }

    void Zatvori()
    {
        jeOtvoren = false;
    }

    static void Main(string[] args)
    {
        // kreiranje objekta - tip podataka je naziv klase nakon čega se kreira preko reči new
        Prozor prozor01 = new Prozor();
        // dodeljivanje vrednosti atributu klase
        prozor01.duzina = 0.9f;
        // pozivanje metode klase
        prozor01.Otvori();
    }
```

Klasa je referentni tip. Deklaracijom se ne kreira instanca klase (objekat), već referenca. Instanciranje klase, odnosno kreiranje objekta vrši se iz dva koraka:

- postavljanjem operatora **new**.
- inicijalizacija objekta se vrši pomoću konstruktora kojim se objekat inicijalizuje (u ovom slučaju je korišćen podrazumevani konstruktor Prozor() – više o konstruktorima u nastavku lekcije).

Modifikatori pristupa

Kontrola mogućnosti pristupa članovima entiteta se ostvaruje navođenjem **modifikatora pristupa** (public i private).

Uvođenjem modifikatora pristupa omogućava se razdvajanje klase na javni deo (čine ga članovi koji su označeni sa modifikatorom pristupa **public** i pristup im nije ograničen) i privatni deo (čine ga članovi koji su označeni sa modifikatorom pristupa **private** i mogu mu pristupiti samo članovi klase).

Modifikator	Opis
public	Pristup bez ograničenja.
protected internal	Pristup je moguć iz projekta u kome je definisan i izvedenih klasa klase u kojoj se nalazi, a koje su van tog projekta.
internal	Pristup je moguć iz projekta u kome je definisan.
protected	Pristup je moguć iz klase u kojoj je definisan i svih izvedenih klasa.
private	Pristup je moguć samo iz klase u kojoj je definisan.

Podrazumevane (default) vrednosti modifikatora pristupa su **internal** za klasu, odnosno **private** za sve njene članove.

To znači da ako napišemo samo **class NazivKlase** ta klasa će biti **internal** i pristup će joj biti moguć samo iz projekta u kom je definisana. A ukoliko njene članove definišemo sa **int broj**, **void Uradi()**, ti članovi će biti posmatrani kao **private** i pristup će im biti moguć samo iz klase u kojoj su i definisani. Da bi klasi i njenim članovima pristup bio moguć bez ograničenja pišemo **public** modifikator pristupa.

```
using System;

namespace KlaseVezba
{
    class Turnir
    {
        // javna polja klase
        public string organizator;
        public string sport;

        // privatno polje klase
        private int nagrada;

        // javna metoda klase
        public void OrganizujeSport()
        {
            Console.WriteLine("{0} je organizovao turnir u {1}!", organizator, sport);
        }

        // privatna metoda klase
        private void PovecajNagradu()
        {
            nagrada += 50000;
        }

    }


        class Program
    {

        static void Main(string[] args)
        {
            // kreiranje objekta
            Turnir kup = new Turnir();

            // pristupanje javnim poljima klase
            kup.organizator = "UEFA";
            kup.sport = "Liga konferencije";

            // pozivanje javne metode klase
            kup.OrganizujeSport();

            /*
             kup.nagrada i kup.PovecajNagradu se ne mogu pisati i kompajler će prijavljivati grešku,
             jer su to polje i metoda privatni i ne može im se pristupati van klase u kojoj su
definisani.
            */
        }
    }
}
```

Svojstva - Visual C# definiše properties tj. „svojstva" (svojstva objekata tj. klasa) kao kombinaciju tj. spoj polja i metode, naime spolja gledano to je polje, a iznutra gledano je metoda. Svojstvu se pristupa kao da je polje (koristi se sintaksa karakteristična za polja), ali kompjuter automatski prevodi ovu sintaksu, i pri tome poziva pristupne metode.

```
namespace KlaseVezba
{
    class Turnir
    {
        public string organizator;
        public string sport;

        private int nagrada;

        // svojstvo (property)
        public int Nagrada
```

```
        {
            // preuzimanje vrednosti preko metode get
            get
            {
                return nagrada;
            }
            // postavljanje vrednosti preko metode set
            set
            {
                nagrada = value;
            }
        }

        public void OrganizujeSport()
        {
            Console.WriteLine("{0} je organizovao turnir u {1}!", organizator, sport);
        }

        private void PovecajNagradu()
        {
            nagrada += 50000;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Turnir kup = new Turnir();

            // dodeljivanje vrednosti svojstvu
            kup.Nagrada = 12500;
            // preuzimanje vrednosti svojstva
            Console.WriteLine(kup.Nagrada);
        }
    }
}
```

Preko javnih svojstava (*public property*) može se ostvariti pristup privatnim članovima klase. Postavljanje vrednosti privatnog polja vrši se naredbom **set**, dok se čitanje vrednosti privatnog polja vrši naredbom **get** u okviru javnog svojstva. Ukoliko bi svojstvo u gornjem primeru pisali bez **set** metode onda bi mogli samo da čitamo vrednost *Nagrada*, a ukoliko bi ga napisali bez **get** metode mogli bi samo da upisujemo vrednost u svojstvo *Nagrada*.

Svrha svojstava jeste enkapsulacija podataka i kontrola pristupa poljima klase. Unutar **get** i **set** metoda se mogu pisati dodatna ograničenja i uopšteno bilo kakva pravila koja bi dodatno obezbedila podatke.

```
        private int nagrada;

        // svojstvo koje omogućava samo preuzimanje vrednosti privatnog polja
        public int Nagrada
        {
            get
            {
                return nagrada;
            }
        }
```

```
        private int nagrada;

        // svojstvo koje omogućava samo postavljanje vrednosti privatnom polju
        public int Nagrada
        {
            set
            {
                if (value > 0 && value < 5000000)
                {
                    nagrada = value;
```

```
                }
                else
                {
                    Console.WriteLine("Uneli ste neispravnu vrednost.");
                }
            }
        }
```

Konstruktor

Konstruktor je deo klase koji služi da nam olakša kreiranje objekata. Nakon modifikatora pristupa piše se naziv klase i otvaraju se zagrade unutar kojih se upisuju parametri ako ih ima, nakon čega se unutar vitičastih zagrada piše kôd konstruktora.

```
using System;

namespace KlaseVezba
{
    class Patike
    {
        public string marka;
        public string model;
        public int broj;

        // Konstruktor
        public Patike(string _marka, string _model, int _broj)
        {
            marka = _marka;
            model = _model;
            broj = _broj;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            // Kreiranje objekta pomoću definisanog konstruktora
            Patike adidas = new Patike("Adidas", "Adistar", 40);
        }
    }
}
```

Podrazumevani konstruktor – Ukoliko sami ne definišemo konstruktor klasa automatski kreira sopstveni podrazumevani konstruktor. On nema parametre i nikakav dodatni kod (public nazivKlase() { }).

```
using System;

namespace KlaseVezba
{
    class Patike
    {
        public string marka;
        public string model;
        public int broj;
    }

    class Program
    {

        static void Main(string[] args)
        {
            // Kreiranje objekta pomoću podrazumevanog konstruktora
            Patike adidas = new Patike();
        }
    }
}
```

Onog momenta kada sami definišemo konstruktor podrazumevani konstruktor se ne kreira automatski.

Konstruktor kopije – Ukoliko nam je potrebno da vrednosti jedne instance klase prosledimo drugoj instanci klase možemo koristiti konstruktor kopije. Njegova svrha je da kreira objekat tako što kopira vrednosti drugog objekta. Njegov parametar je klasnog tipa, klase kojoj i sam konstruktor pripada.

```
using System;

namespace KlaseVezba
{
    class Pas
    {
        public string rasa;
        public string boja;
        public int godine;

        // Konstruktor
        public Pas(string _rasa, string _boja, int _godine)
        {
            rasa = _rasa;
            boja = _boja;
            godine = _godine;
        }

        // Konstruktor kopije - za parametar ima klasu kao tip podataka
        public Pas(Pas p)
        {
            // Preuzimanje vrednosti od objekta koji je korišćen kao parametar
            rasa = p.rasa;
            boja = p.boja;
            godine = p.godine;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            // Kreiranje objekta pomoću definisanog konstruktora
            Pas dzeki = new Pas("labrador", "zlatna", 3);
            // Kreiranje objekta pomoću konstruktora kopije - imaće iste vrednosti kao i objekat dzeki
            Pas reks = new Pas(dzeki);
        }
    }
}
```

Destruktor – Za uništavanje objekta koristi se destruktor. Potrebe za destruktorom u C# gotovo i da nema jer se nepotrebni objekti automatski uništavaju.

Ključna reč this

Ključna reč this se koristi kada želimo da napravimo razliku između parametara metode ili konstruktora i polja u klasi. Takođe može poslužiti i pri pozivu metoda koje su deo kreiranog objekta, najčešće nije potrebno stavljati reč this pri pozivu metoda ali nije greška ni koristiti je. U ovim slučajevima reč this možemo smatrati zamenom za naziv kreiranog objekta.

```
using System;

namespace KlaseVezba
{
    class Raspust
    {
        public int duzina;
        public string naziv;
        public bool uToku;

        public Raspust(int duzina, string naziv, bool uToku)
        {
            // this označava da se radi o promenljivoj duzina koja pripada kreiranom objektu
            // ovim se omogućava da se atributu klase dodeli vrednost parametra sa istim nazivom
            this.duzina = duzina;
```

```
            this.naziv = naziv;
            this.uToku = uToku;
            // this označava da se radi o metodi skratiRaspust koja pripada kreiranom objektu
            // ključna reč this može, a ne mora stojati pri pozivu klasinih metoda unutar nje same
            this.skratiRaspust(2);
        }

        public void skratiRaspust(int brojDana)
        {
            if (brojDana <= duzina)
            {
                duzina -= brojDana;
            }
            else
            {
                Console.WriteLine("Raspust se ne može skratiti za uneti broj dana.");
            }
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            // kreiranje objekta
            Raspust zimski = new Raspust(12, "zimski", false);
        }
    }
}
```

Ključna reč this se takođe može koristiti i kada želimo da pri kreiranju konstruktora pozovemo izvršavanje nekog drugog konstruktora. Tada nakon ispisivanja naziva i eventualnih parametara konstruktora dodajemo **:this(parametri ako ih ima)**.

```
using System;

namespace KlaseVezba
{
    class Kabinet
    {
        public string naziv;
        public int kapacitet;
        public bool otvorenaVrata;
        public bool otvoreniProzori;

        public Kabinet()
        {
            otvorenaVrata = true;
            otvoreniProzori = true;
        }

        // Pozivanje drugog konstruktora se vrši pomoću :this(parametri ako ih ima)
        public Kabinet(string naziv, int kapacitet):this()
        {
            this.naziv = naziv;
            this.kapacitet = kapacitet;
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            // Kreiranje objekta - pozivaju se i konstruktor Kabinet(string naziv, int kapacitet) i
Kabinet()
            Kabinet k2 = new Kabinet("k2", 18);
        }
    }
}
```

Statički i nestatički članovi klase

Članovi klase su polja (atributi) i metode. Ti članovi klase se dele na statičke (klasne) i nestatičke (objektne). Statički članovi se nazivaju i klasnim zato što pripadaju klasi i nije potrebno kreirati objekat da bi se oni koristili. Njima se pristupa preko naziva klase. Sa druge strane nestatički članovi su pristupačni samo preko kreiranog objekta i zato se nazivaju i objektni.
Statičke metode mogu da koriste samo statičke članove klase dok nestatičke metode mogu da koriste i statičke i nestatičke članove klase.

```
using System;

namespace KlaseVezba
{
    class Film
    {
        public string naziv;
        public string zanr;
        // Statički članovi klase se definišu dodavanjem reči static nakon modifikatora pristupa
        // Statička polja pripadaju klasi i iste su vrednosti za svaki kreirani objekat
        public static string tip = "dugometražni";
        public static int brojFilmova;

        public Film(string naziv, string zanr)
        {
            this.naziv = naziv;
            this.zanr = zanr;
            brojFilmova++;
        }

        public void Prikazi(string bioskop)
        {
            Console.WriteLine("Film " + naziv + " se prikazuje u " + bioskop + ".");
            // Nestatička metoda može da koristi i statička polja (brojFilmova u ovom slučaju)
            Console.WriteLine("{0} je jedan od {1} filmova.", naziv, brojFilmova);
        }

        // Statičke metode mogu u sebi da sadrže samo statička polja klase
        // Nestatičke metode mogu da koriste i statička i nestatička polja klase
        public static void PromeniTip(string noviTip)
        {
            tip = noviTip;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Film Hangover = new Film("Hangover", "Komedija");
            // Statički članovi klase se pozivaju nazivom klase i zatim nazivom željenog atributa ili
metode
            Film.PromeniTip("kratkometražni");
            // Nestatički članovi klase se pozivaju nazivom objekta i zatim nazivom željenog atributa
ili metode
            Hangover.Prikazi("Somborski bioskop");
        }
    }
}
```

Parcijalne klase

Klasa može da sadrži više metoda, polja i konstruktora. Neke klase mogu biti glomazne. Zato, u jeziku C# postoji mogućnost da podelite izvorni kod klase u nekoliko odvojenih fajlova (datoteka) tako da neka glomazna klasa može da se organizuje kao skup nekoliko manjih klasa. Npr. deo klase koji može da se menja je u jednom fajlu, a deo klase koji se ne menja je u drugom fajlu.
Kada se neka klasa podeli u nekoliko fajlova, onda se definišu tzv. „parcijalne klase" pomoću ključne reči **partial.**

Npr. klasa Kvadrat može da se podeli u dva fajla: kvadrat1.cs (koja sadrži konstruktore) i kvadrat2.cs (koja sadrži metode i polja):

```
partial class Kvadrat
{
    public Kvadrat() { ... } //podrazumevani konstruktor
    public Kvadrat(...) { ... } //nepodrazumevani konstruktor
}

partial class Kvadrat
{
    public float Povrs() { ... }
    private float strana;
}
```

Kompajler automatski grupiše parcijalne klase (iz razdvojenih fajlova) u jednu klasu.

Apstraktne klase

Svaka klasa koja sadrži ključnu reč **abstract** i koja ima najmanje jednu apstraktnu metodu se naziva apstraktna klasa. Apstraktna metoda je metoda koja sadrži ključnu reč abstract i koja nema ni telo ni implementaciju. **Apstraktna klasa ne može da se instancira** dok svaka njena apstraktna metoda mora da bude nadjačana (override-ovana) u svakoj neapstraktnoj klasi koja nasleđuje apstraktnu klasu. To jednostavno znači da apstraktnu klasu i njene apstraktne metode koristite kad niste sigurni kakva će biti implementacija neke metode ali vam je ta metoda neophodna. Apstraktna klasa je u svakom slučaju nepotpuna klasa.
Na primer, pretpostavite da u vašem projektu morate imati klasu koja predstavlja konekciju sa bazom podataka, međutim vi ne znate kakve će sve baze koristiti kompanija za koju radite projekat.

```
public abstract class Konekcija
{
    public abstract string StringZaKonekciju();
}
```

Tada pravite apstraktnu klasu sa apstraktnom metodom koja predstavlja konekciju sa bazom podataka i ostavljate kompaniji da prilikom korišćenja vaše apstraktne klase u drugoj klasi nadjačaju metodu za konekciju i definišu njeno ponašanje.

```
class Program : Konekcija
{
    public override string StringZaKonekciju()
    {
        return "Tekst konekcije za bazu podataka";

    }
}
```

Na ovaj način svako ko na primer hoće da vidi tabelu zaposlenih mora implementirati konekciju sa bazom podataka u kojoj se podaci zaposlenih nalaze. Možda kompanija ima posebnu bazu podataka i tabelu zaposlenih u Beogradu i jednu drugu u Novom Sadu i podaci su fizički odvojeni, što nije praksa ali se dešava kad velika kompanija kupi malu. Onda vaša apstraktna klasa je idealno rešenje za sve vaše konekcije sa bazama podataka.

Interfejsi

Intefejs klasa je predstavnik jedne ili više klasa, predstavlja obrazac o tome šta će biti implementirano. Sakriva ko i kako implementira svojstva i metode. **Sadrži deklaraciju metoda ali ne i definiciju (telo) metoda**. Konkretna klasa nasleđuje klasu interfejsa (preko simbola : kao i pri nasleđivanju bilo koje druge klase) i implementira njene okvirne elemente. Interefejsi mogu sadržati svojstva i metode ali ne i polja.

```
// interfejs
interface IZivotinja
{
    void zvuk(); // metoda interfejsa (ne sadrži telo metode)
    void pokret(); // metoda interfejsa (ne sadrži telo metode)
}
```

Preporuka je da se nazivi interfejsa pišu sa početnim velikim slovom I da bi se znalo da se radi o interfejsu a ne o običnoj klasi.
Podrazumevane vrednosti članova interfejsa su abstract i public.

Klasa Object

Object klasa je bazna klasa svim klasama u C#. Deo je system namespace-a i u njoj su definisane osnovne metode koje svaka izvedena klasa može da redefiniše za sebe. Object klasa služi da bi osigurala kontrolu na najnižem mogućem nivou za svaku klasu.

Metode klase Object

ToString() – predstavlja metodu koju možemo pozvati na objektu svake klase i vratiće nam taj objekat reprezentovan u stringu.

```
public virtual String ToString()
        {
            return GetType().ToString();
        }
```

Equals – predstavlja metodu koja proverava da li je objekat nad kojim se poziva metoda isti kao objekat prosleđen kao parametar metode.

```
public virtual bool Equals(Object obj)
        {
            return RuntimeHelpers.Equals(this, obj);
        }
```

```
        static void Main(string[] args)
        {
            // Kreiranje objekata tipa podatka Automobil
            Automobil fiat = new Automobil();
            Automobil opel = new Automobil();
            // Kreiranje objekta tipa podatka Object
            Object golf = new Automobil();
            Object skoda = new Object();

            Console.WriteLine(fiat.Equals(opel)); // Ispisaće se True
            Console.WriteLine(fiat.Equals(golf)); // Ispisaće se False

            Console.WriteLine(fiat.ToString()); // Ispisaće se Tekst override-ovane metode
            Console.WriteLine(skoda.ToString()); // Ispisaće se System.Object
        }

        class Automobil
        {
            int godiste;

            public bool Equals(Automobil auto)
            {
                return this.godiste == auto.godiste;
            }

            public override string ToString()
            {
                return "Tekst override-ovane metode";
            }
        }
```

NASLEĐIVANJE

Nasleđivanje nam omogućava da kreiramo klasu nastalu iz već postojeće klase. Jedno je od ključnih segmenata objektno orijentisanog programiranja.
Klasa koja je nasleđena se naziva **bazna klasa** (osnovna klasa, nadklasa, klasa roditelj) dok se klasa koja nasleđuje naziva **izvedena klasa** (podklasa, klasa dete).
Izvedena klasa nasleđuje polja i metode bazne klase (**ne nasleđuje konstruktore**). Svrha ovoga je mogućnost upotrebe istog koda bez obaveze da se svaki put piše iznova.

Simbol :

Naziv bazne klase (one koju želimo da nasledimo) pišemo nakon kreirane izvedene klase i simbola **:** .

```
        // bazna klasa - Zivotinja
        class Zivotinja
        {
            public string rasa;
            public int godiste;

            public void Ispisi()
            {
                Console.WriteLine(rasa + " " + godiste);
            }
        }

        // izvedena klasa - Macka
        class Macka : Zivotinja
        {
            public string boja;

            public void Mjaukni()
            {
                Console.WriteLine("Mjau mjau!");
            }
        }
```

Kreiranje objekta izvedene klase

Ukoliko kreiramo objekat izvedene klase u Main delu neke druge klase (u ovom primeru klase Program) možemo pristupiti svim public poljima i metodama bazne i izvedene klase.

```
    class Program
    {
        public static void Main(string[] args)
        {
            // Kreiranje objekta izvedene klase
            Macka macka01 = new Macka();

            // Pristup poljima i metodi bazne klase
            macka01.rasa = "Sijamska";
            macka01.godiste = 2015;
            macka01.Ispisi();

            // Pristup poljima i metodi izvedene klase
            macka01.boja = "Bela";
            macka01.Mjaukni();
        }
    }
```

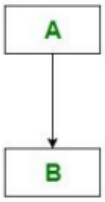
U primeru iznad je kreirana izvedena klasa **Macka** na osnovu bazne klase **Zivotinja**. Samim tim klasa **Macka** može da pristupi poljima i metodama klase **Zivotinja**. Objekat izvedene klase može biti kreiran i sa tipom podataka bazne klase (u gornjem primeru bi to bilo npr. `Zivotinja feliks = new Macka();`).
Izvedena klasa ima smisla ako se može povezati veznikom **je** sa baznom klasom (mačka **je** životinja, automobil **je** vozilo, jabuka **je** voće...).

Modifikatori pristupa (nasleđivanje)

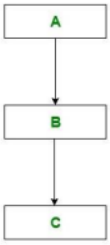
Izvedena klasa nasleđuje sva polja i metode bazne klase bez obzira na njihov modifikator pristupa.
public – polja i metode su vidljivi i mogu se koristiti unutar bazne i izvedene klase kao i van njih.
internal – polja i metode su vidljivi i mogu se koristiti unutar istog projekta.
protected – polja i metode su vidljivi i mogu se koristiti samo unutar bazne klase i izvedenih klasa koje je nasleđuju.
private – polja i metode su vidljivi i mogu se koristiti samo unutar klase u kojima su kreirani.
sealed – koristimo kao modifikator pristupa same klase kada želimo da onemogućimo njeno nasleđivanje.

Tipovi nasleđivanja

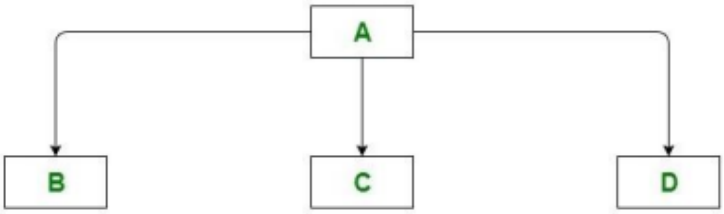
Prosto (jednostruko) nasleđivanje – Kod ovog tipa nasleđivanja jedna klasa nasleđuje baznu klasu. Dakle klasa A je bazna, a klasa B izvedena klasa.



Višeslojno nasleđivanje – Kod ovog tipa nasleđivanja, iz jedne bazne klase (A) izvodi se izvedena klasa (B) koja je opet bazna za izvedenu klasu (C).



Hijerarhijsko nasleđivanje – Kod ovog tipa nasleđivanja iz jedne bazne klase izvodi se nekoliko klasa. U prikazanom slučaju, iz klase A izvode se klase B, C i D.



Konstruktori u nasleđivanju

Izvedena klasa ne nasleđuje konstruktore bazne klase.
Kreiranjem objekta izvedene klase obavezno se prvo izvršava konstruktor bazne klase pa zatim i konstruktor izvedene klase.
Ukoliko bazna klasa nema eksplicitno kreiran konstruktor sa parametrima konstruktor bez parametara će biti automatski pozvan prilikom pozivanja konstruktora izvedene klase.
Ukoliko bazna klasa ima eksplicitno kreiran konstruktor sa parametrima obavezno je pozvati ga unutar izvedene klase pomoću ključne reči **base**.

```
// bazna klasa - Zivotinja
public class Zivotinja
{
    public string naziv;
}

// izvedena klasa - Macka
class Macka : Zivotinja
{
    public string boja;
/* Kreiranje eksplicitnog konstruktora u izvedenoj klasi u slučaju
kada ne postoji eksplicitni konstruktor sa parametrima u baznoj klasi */
    public Macka()
    {
        Console.WriteLine("Boja mačke imena {0} je {1}", naziv, boja);
    } }
```

```
// bazna klasa - Zivotinja
public class Zivotinja
{
    public string naziv;

    public Zivotinja(int broj)
    {
        Console.WriteLine("Broj životinja je {0}.", broj);
    }
}

// izvedena klasa - Macka
class Macka : Zivotinja
{
    public string boja;

/* Kreiranje eksplicitnog konstruktora u izvedenoj klasi u slučaju
kada postoji eksplicitni konstruktor sa parametrima u baznoj klasi */
    public Macka() : base(10)
    {
        Console.WriteLine("Boja mačke imena {0} je {1}", naziv, boja);
    }
}
```

Preopterećenje (overloading) metoda (u nasleđivanju)

Ukoliko želimo da unutar bazne i izvedene klase koristimo metodu sa istim nazivom prvi način je da te dve metode imaju **različit tip parametara ili različit broj parametara**.
Ukoliko želimo da predefinišemo metodu bazne klase (da promenimo ono što ona treba da radi) potrebno je da unutar bazne klase nakon modifikatora pristupa postavimo službenu reč **virtual**, a nakon toga da u izvedenoj klasi nakon modifikatora pristupa postavimo službenu reč **override**. Ovim će metoda bazne klase u izvedenoj dobiti drugu svrhu.
Ukoliko želimo samo da sakrijemo metodu bazne klase unutar izvedene, a ne da je predefinišemo, onda koristimo službenu reč **new**.

```
// bazna klasa - Zivotinja
class Zivotinja
{
    public string naziv;

    /* Metoda istog naziva u baznoj i izvedenoj klasi
    sa različitim brojem parametara*/
    public void Setaj(int km)
    {
        Console.WriteLine("Životinja je šetala {0} kilometara", km);
    }

    // Metoda koja se može predefinisati u izvedenoj klasi
    public virtual void Hrani()
    {
        Console.WriteLine("{0} jede hranu.", naziv);
    }
}

// izvedena klasa - Macka
class Macka : Zivotinja
{
    /* Metoda istog naziva u baznoj i izvedenoj klasi
    sa različitim brojem parametara*/
    public void Setaj(int km, string dan)
    {
        Console.WriteLine("Životinja je šetala {0} kilometara, dana {1}", km, dan);
    }

    // Metoda koja se može predefinisati u izvedenoj klasi
    public override void Hrani()
    {
        Console.WriteLine("{0} jede mačiju hranu.", naziv);
    }
```

```
        // Metoda koja sakriva metodu iz bazne klase
        public new void Setaj(int km)
        {
            Console.WriteLine("Mačka je šetala {0} kilometara", km);
        }
    }
```

Ključna reč base

Ključnu reč base, pored korišćenja pri pozivanju konstruktora bazne klase, možemo koristiti unutar izvedene klase i pri pozivanju polja i metoda bazne klase.
Ključnom rečju base ne možemo pozivati privatna polja i metode bazne klase.

Pozivanje metoda

Koja metoda de biti pozvana, metoda bazne ili izvedene klase se određuje na osnovu tipa instance koju promenljiva referencira, a ne na osnovu tipa same promenljive (na osnovu konstruktora koji pozivamo pri kreiranju objekta). To znači da ukoliko koristimo konstruktor izvedene klase, a tip podatka bazne klase (`BaznaKlasa objekat1 = new IzvedenaKlasa();`) biće pozvana override-ovana verzija metode ukoliko postoji unutar izvedene klase.

```
public class Prodavnica
{
    public virtual void Prodaj()
    {
        Console.WriteLine("Prodat je artikl.");
    }
}

public class Univerexport : Prodavnica
{
    public override void Prodaj()
    {
        Console.WriteLine("Prodat je Univerexportov artikl.");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Prodavnica pro1 = new Prodavnica();
        Prodavnica univer = new Univerexport();

        pro1.Prodaj(); // Na ekranu će se ispisati Prodat je artikl.
        univer.Prodaj(); // Na ekranu će se ispisati Prodat je Univerexportov artikl.
    }
}
```

```
public class Napitak
    {
        public virtual void Sipaj()
        {

Console.WriteLine("Napitak.");
        }
    }

    public class Caj : Napitak
    {
        public override void Sipaj()
        {

Console.WriteLine("Čaj.");
        }
    }

    public class LedeniCaj : Caj
    {
        public override void Sipaj()
        {

Console.WriteLine("Ledeni čaj.");
        }
    }
```

```
public class Program
    {
        static void Main(string[] args)
        {
            Napitak napitak1 = new Napitak();
            Napitak napitak2 = new Caj();
            Napitak napitak3 = new LedeniCaj();
            Caj caj1 = new Caj();
            Caj caj2 = new LedeniCaj();
            LedeniCaj ledeniCaj1 = new LedeniCaj();

            napitak1.Sipaj(); // Ispisaće se Napitak.
            napitak2.Sipaj(); // Ispisaće se Čaj.
            napitak3.Sipaj(); // Ispisaće se Ledeni čaj.
            caj1.Sipaj(); // Ispisaće se Čaj.
            caj2.Sipaj(); // Ispisaće se Ledeni čaj.
            ledeniCaj1.Sipaj(); // Ispisaće se Ledeni čaj.
        }
    }
```

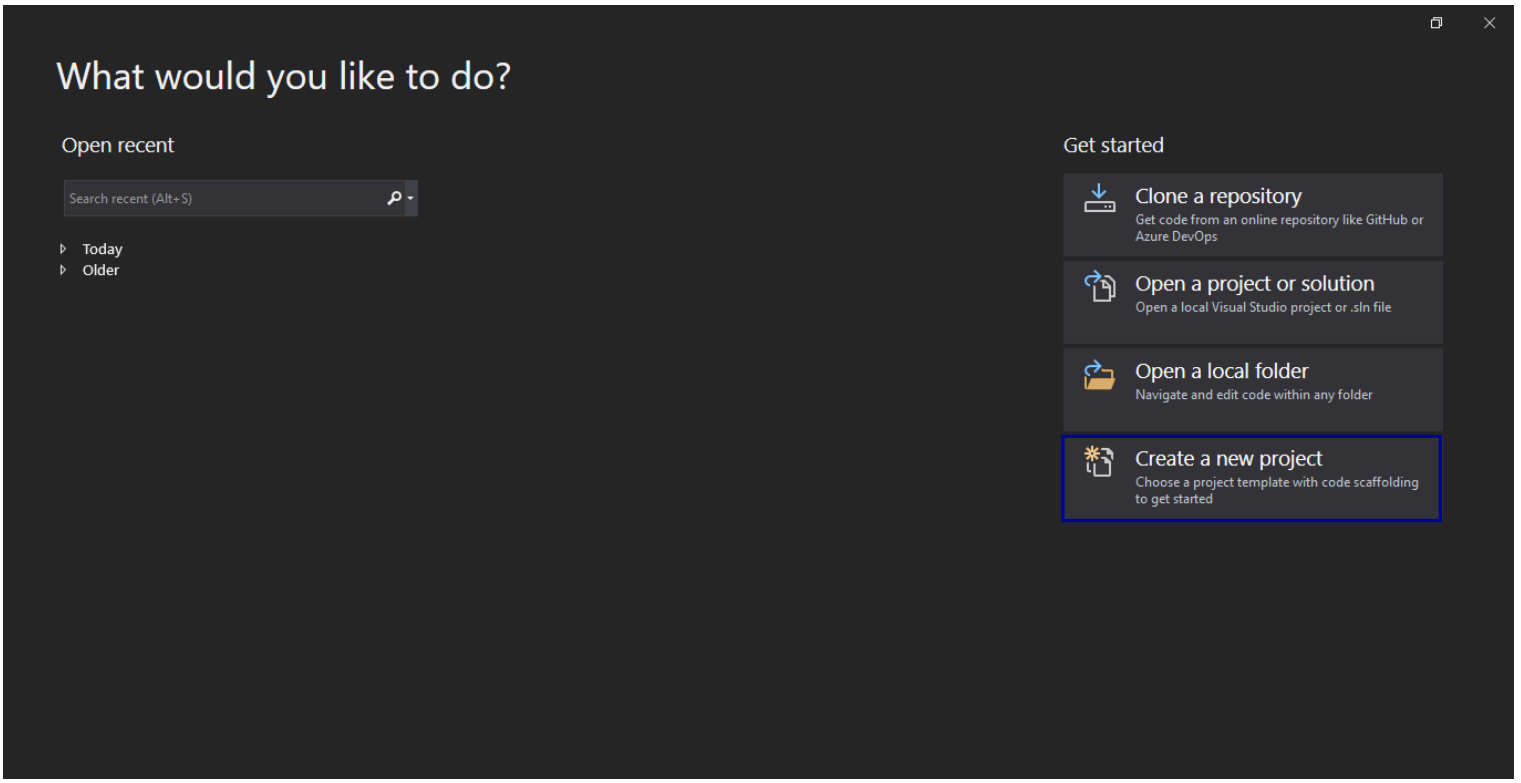
Ukoliko je unutar izvedene klase pozivanoj metodi dodata službena reč **new** ta metoda će biti pozvana jedino ako je tip podatka prilikom kreiranja objekta isti kao i izvedena klasa unutar koje se nalazi ta metoda.

```
    public class Napitak
    {
        public virtual void Sipaj()
        {

Console.WriteLine("Napitak.");
        }
    }

    public class Caj : Napitak
    {
        public override void Sipaj()
        {
        Console.WriteLine("Čaj.");
        }
    }

    public class LedeniCaj : Caj
    {
        public new void Sipaj()
        {
        Console.WriteLine("Ledeni čaj.");
        }
    }
```

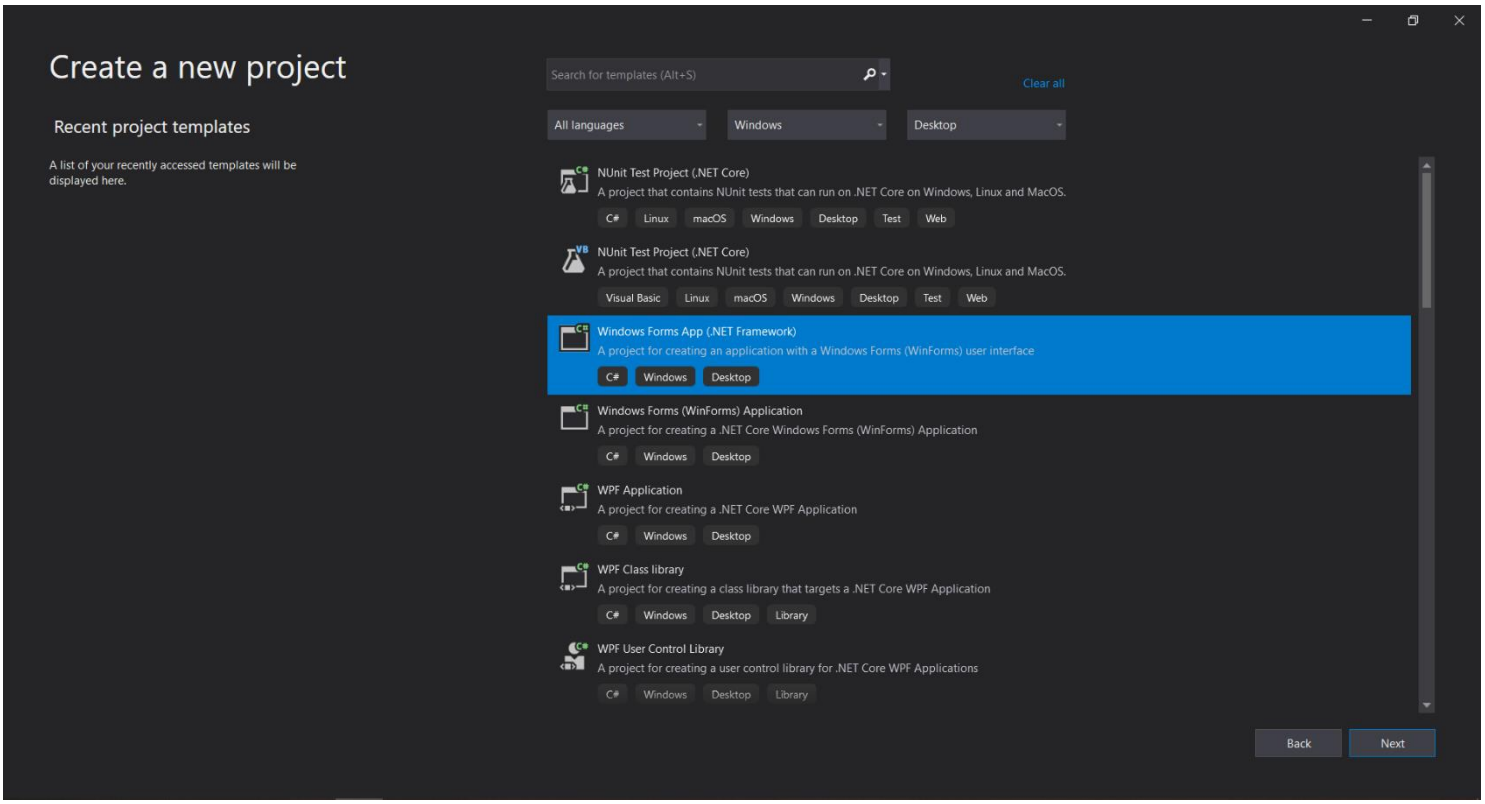
```
    public class Program
    {
        static void Main(string[] args)
        {
            Napitak napitak1 = new Napitak();
            Napitak napitak2 = new Caj();
            Napitak napitak3 = new LedeniCaj();
            Caj caj1 = new Caj();
            Caj caj2 = new LedeniCaj();
            LedeniCaj ledeniCaj1 = new LedeniCaj();

            napitak1.Sipaj(); // Ispisaće se Napitak.
            napitak2.Sipaj(); // Ispisaće se Čaj.
            napitak3.Sipaj(); // Ispisaće se Čaj.
            caj1.Sipaj(); // Ispisaće se Čaj.
            caj2.Sipaj(); // Ispisaće se Čaj.
            ledeniCaj1.Sipaj(); // Ispisaće se Ledeni čaj.
        }
    }
```

WINDOWS FORME

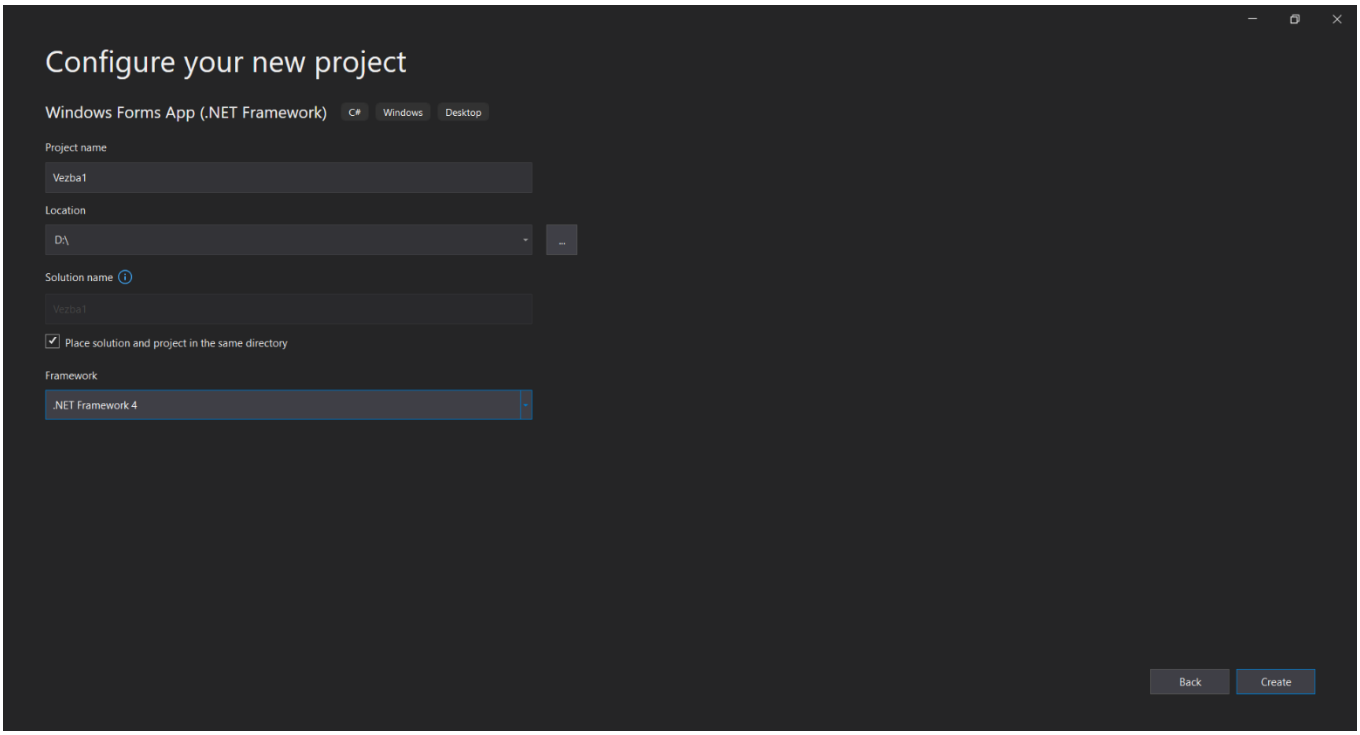
Nakon pokretanja *Visual Studio 2019 Community* programa biramo opciju *Create a new project*.



Nakon toga biramo *Windows Forms App (.NET framework)* i opciju *Next* u donjem desnom uglu.



U sledećem prozoru biramo naziv aplikacije, lokaciju gde ćemo je smestiti i čekiramo opciju *Place solution and project in the same directory.* Nakon čega biramo Framework (najbolje da izaberete .Net Framework 4) i opciju *Create* u donjem desnom uglu.



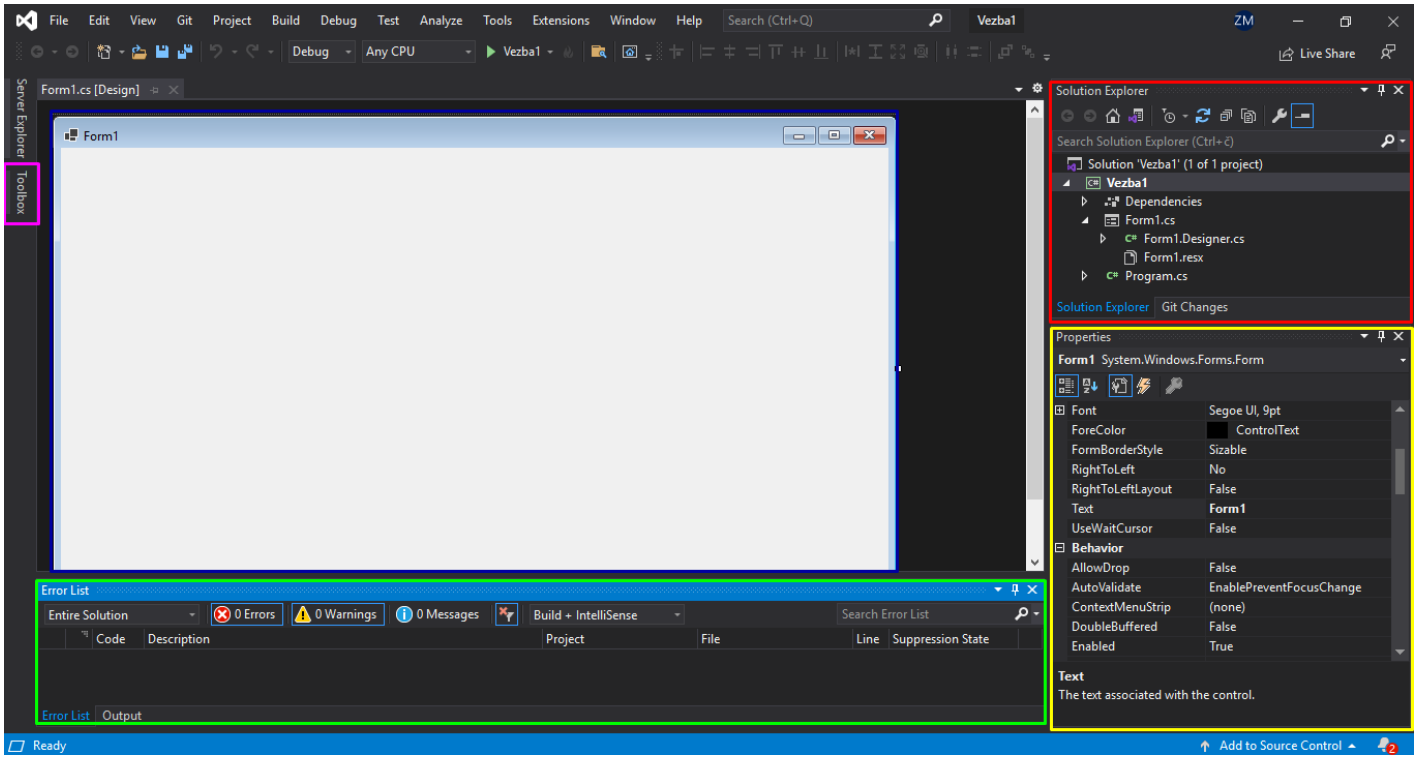
Nakon što je aplikacija kreirana dobijamo prozor sa **formom** (uokvireno plavom bojom na slici) koja će predstavljati ono što korisnik vidi kada pokrene naš program.

Ispod forme nalazi se deo sa listom grešaka (**Error List**) koji takođe može biti postavljen i na opciju *Output* (uokvireno zelenom bojom).

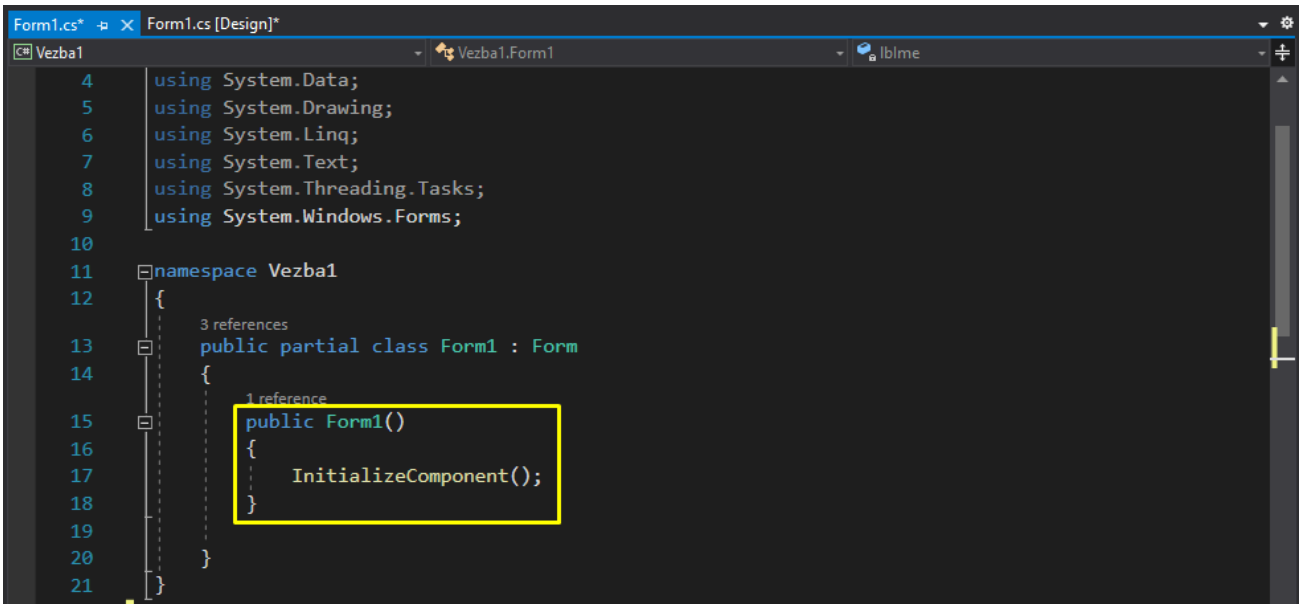
Na desnoj strani nalazi se **Solution Explorer** u kom se nalazi sve što je deo našeg programa (uokvireno crvenom bojom).

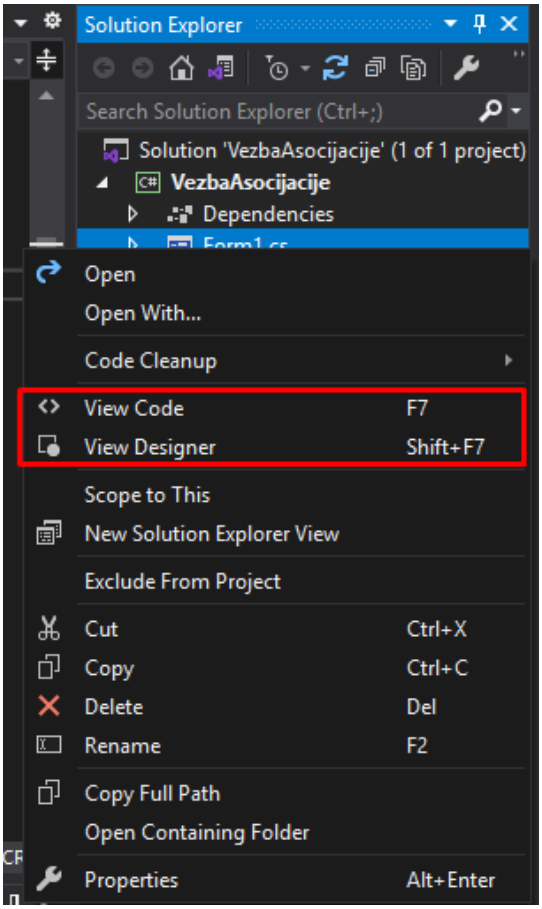
Ispod *Solution Explorer*-a nalaze se svojstva selektovane forme ili kontrole (**Properties** – uokvireno žutom bojom).

Sa leve strane imamo pristup **Toolbox**-u (uokvireno ljubičastom bojom – takođe se može pristupiti i preko opcije *View -> Toolbox*)



Kodu forme se može pristupiti desnim klikom na samu formu i izborom opcije *View Code*. Kod koji se automatski dobija u formi jeste klasa sa nazivom forme koja nasleđuje klasu *Form* i već kreirani konstruktor sa metodom koja inicijalizuje sve ono što postavimo na formu (uokvireno žutom bojom). Sav naš kod pišemo unutar klase, a ništa unutar već kreiranog konstruktora.

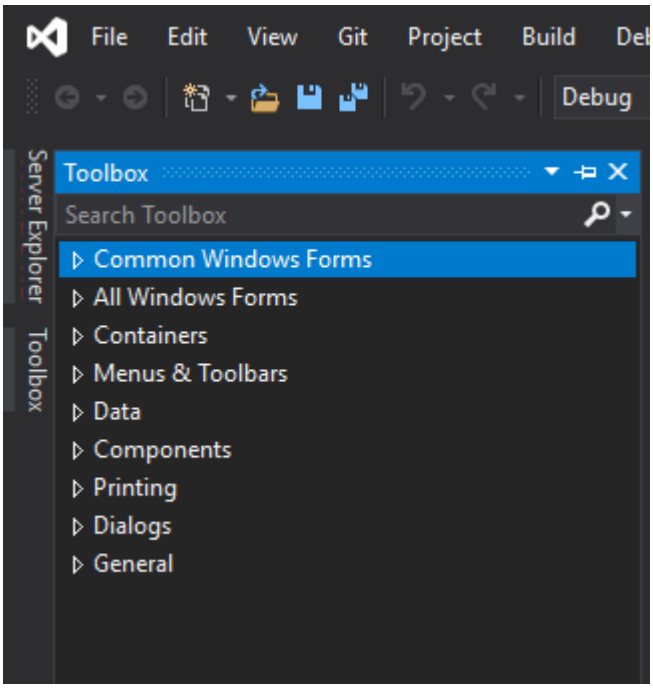




Ukoliko kod i/ili dizajner nisu otvoreni možemo im pristupiti desnim klikom na formu u *Solution Explorer*-u i izborom *View Code* ili *View Designer* kao i preko skraćenica *F7* odnosno *Shift+F7*. Na taj način možemo pokušati pokrenuti dizajner ukoliko se tokom njegovog učitavanja pojavi *IntelliSense Error* (prvo zatvaramo dizajner preko *x*-a).

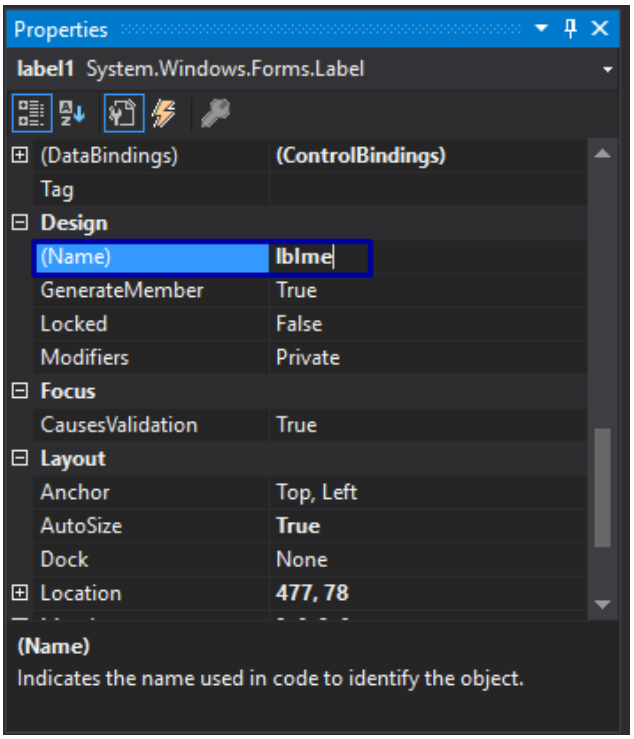
Tipovi kontrola

Unutar *Toolbox*-a se nalaze različite kontrole koje se na formu mogu ubaciti ili prevlačenjem mišem ili duplim klikom na neku od njih.



Unutar odeljka *Common Windows Forms* se nalaze najčešće korišćene kontrole.

Preporuka je da se unutar svojstva (*Properties*) kontrole koja se postavi na formu doda naziv sa skraćenicom tipa kontrole i zatim nazivom onoga što ona označava (*lbIme* za label koji služi za ime i slično).



Kontrole sa kojima će najčešće biti rađeno :

Label - Služi za ispis teksta i obično je opisnog, odnosno informativnog, karaktera (opis šta se treba uneti u polje za unos i slično).

TextBox - Polje za unos teksta.

Button - Dugme, služi da klikom na njega pokrenemo određena dešavanja.

RadioButton - Koristimo kada imamo više opcija a korisniku želimo da omogućimo da izabere samo jednu.

CheckBox - Koristimo kada imamo više opcija a korisniku želimo da omogućimo da izabere više njih ili nijednu od ponuđenih.

ComboBox - Padajući meni sa opcijama koje se unose unutar svojstva *Items*.

DateTimePicker - Služi za pružanje mogućnosti korisniku da izabere datum. Svojstva *MinDate* i *MaxDate* služe za postavljanje minimalnog i maksimalnog dostupnog datuma za izbor a svojstvo *Format* za izbor prikaza formata datuma unutar kontrole.

PictureBox - Služi za umetanje slike unutar forme.

MessageBox - Iskačući prozor (poziva se pomoću metode *Show* (MessageBox.Show("Tekst koji se pojavljuje u iskačućem prozoru")).

Svojstva kontrola

BackColor - Odnosi se na pozadinsku boju.

ForeColor - Odnosi se na boju slova.

Text - Predstavlja tekst ispisan na samoj kontroli.

Font - Podešavanje tipa i veličine fonta.

(Name) - Predstavlja naziv kontrole unutar koda.

AcceptButton (svojstvo forme) - Daje izbor dugmeta koje će biti aktivirano kada korisnik pritisne *Enter*.

CancelButton (svojstvo forme) - Daje izbor dugmeta koje će biti aktivirano kada korisnik pritisne *ESC*.

StartPosition (svojstvo forme) - Daje izbor gde na ekranu će se nalaziti forma po pokretanju aplikacije.

MaximizeBox (svojstvo forme) - Daje izbor da li da forma ima mogućnost povećanja preko celog ekrana.

MinimizeBox (svojstvo forme) - Daje izbor da li da forma ima mogućnost ,,minimiziranja".

Enabled - Ima opcije *true* i *false* i označava da li će kontrola biti dostupna korisniku za interakciju sa njom.

Checked (svojstvo kontrola za izbor) - Označava da li je određena kontrola izabrana (,,čekirana'' – true) ili ne (false).

DropDownStyle (svojstvo ComboBox-a) - Označava da li će korisnik moći da unosi tekst unutar njega (*DropDown*) ili ne (*DropDownList*).

MinDate (svojstvo kontrola za datum) - Izbor najranijeg dostupnog datuma koji će korisnik moći da izabere.

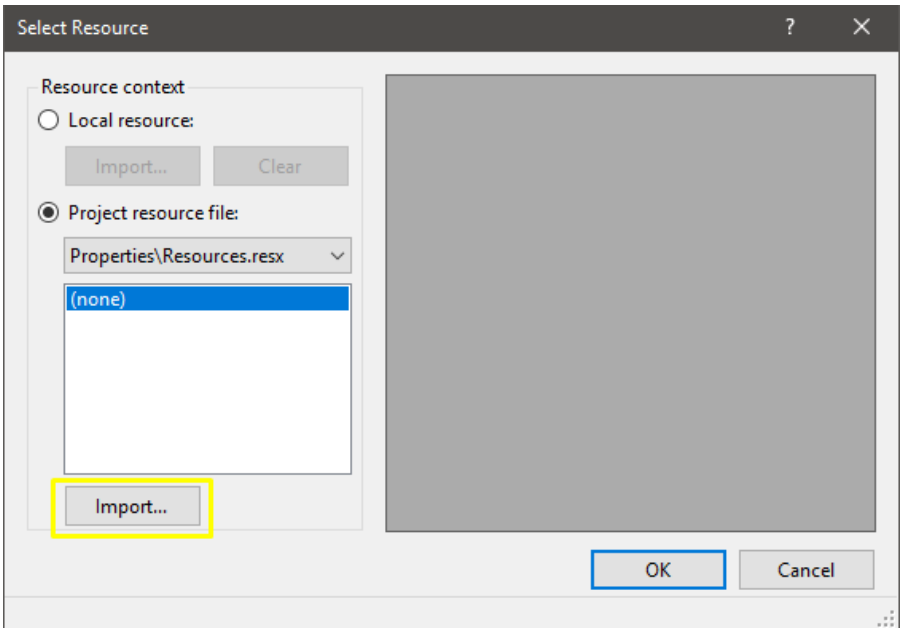
MaxDate (svojstvo kontrola za datum) - Izbor najkasnijeg dostupnog datuma koji će korisnik moći da izabere.

Format (svojstvo kontrola za datum) - Izbor formata datuma (načina na koji će biti prikazan – moguće je izabrati i *Custom* nakon čega se u svojstvu *CustomFormat* može izabrati format u kom želimo da datum bude prikazan).

Text align (svojstvo TextBox-a) - Izbor centriranja teksta (*left, right, center*).

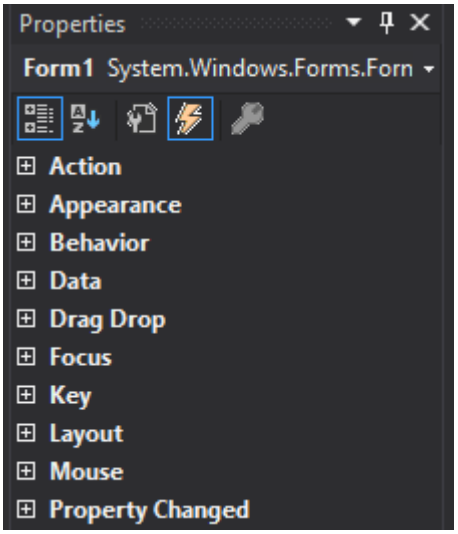
PlaceholderText (svojstvo TextBox-a) - Tekst koji će stojati unutar TextBox-a pre nego što se počne sa unosom.

BackgroundImage – Image - Izbor slike koju za koju želimo da bude na pozadini ili unutar okvira za sliku.



Preko opcije *Import* slika se automatski ubacuje u resurse programa.

Kreiranje događaja



Unutar kartice sa svojstvima nalazi se i kartica sa događajima (*Events*). Događaji su poređani u (biće navedeni samo neki od najčešće korišćenih) :

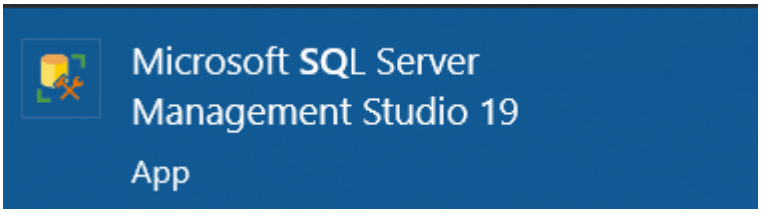
- **Action** –
  - Click – duplim klikom na njega kreira se metoda koja se izvršava kada se klikne na formu ili kontrolu
  - DoubleClick – duplim klikom na njega kreira se metoda koja se izvršava kada se klikne dvaput na formu ili kontrolu
- **Appearance**
- **Behavior**
- **Data**
- **Drag Drop**
- **Focus**
- **Key**
  - KeyDown – Izvršava se čim korisnik pritisne taster na tastaturi.
  - KeyPress – Izvršava se za tastere kada se pritisnu i puste, za razliku od KeyDown i KeyUp ne pokreće se za tastere koji ne predstavljaju karaktere (Shift, Ctrl...).
  - KeyUp – Izvršava se nakon što korisnik pusti taster na tastaturi.

    - *Provera koji taster je pritisnut se za KeyPress vrši preko* if (e.KeyChar == (char)Keys.Enter) *(primer je za taster Enter, može se koristiti za ostale tastere tastature), a za KeyDown i KeyUp preko* if (e.KeyCode == Keys.Enter).
- **Layout**
- **Mouse**
    - MouseDown – Izvršava se nakon klika mišem na kontrolu.
    - MouseEnter – Izvršava se nakon što pristupimo kontroli kursorom miša.
    - MouseHover – Izvršava se kada zadržimo kursor miša na kontroli.
    - MouseLeave – Izvršava se kada pomerimo kursor miša sa kontrole.
- **Property Changed**
    - TextChanged – Izvršava se kada se promeni tekst unutar kontrole.

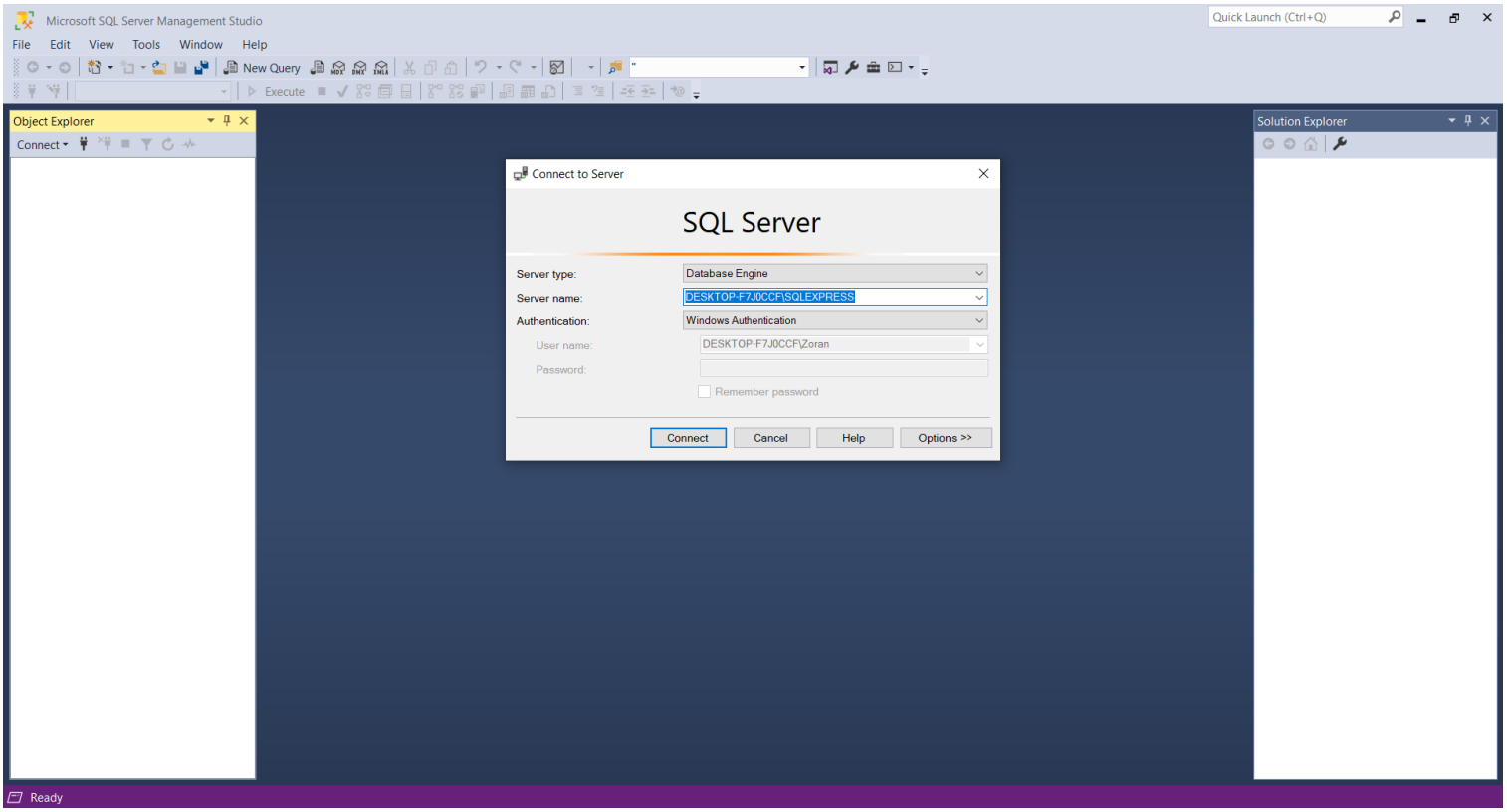
RAD SA BAZAMA PODATAKA

SQL Server Management Studio 19 – početni koraci

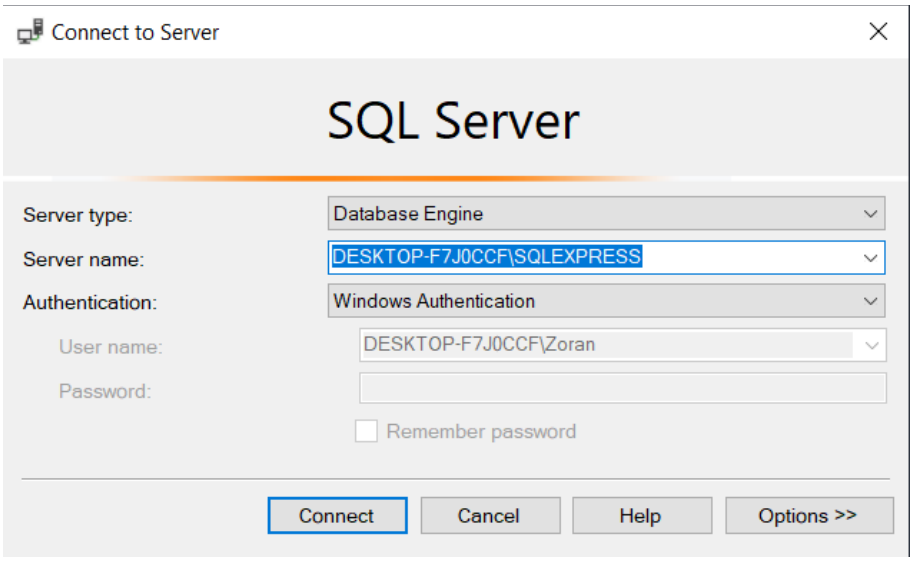
Pokretanje *Microsoft SQL Server Management Studio 19* okruženja :



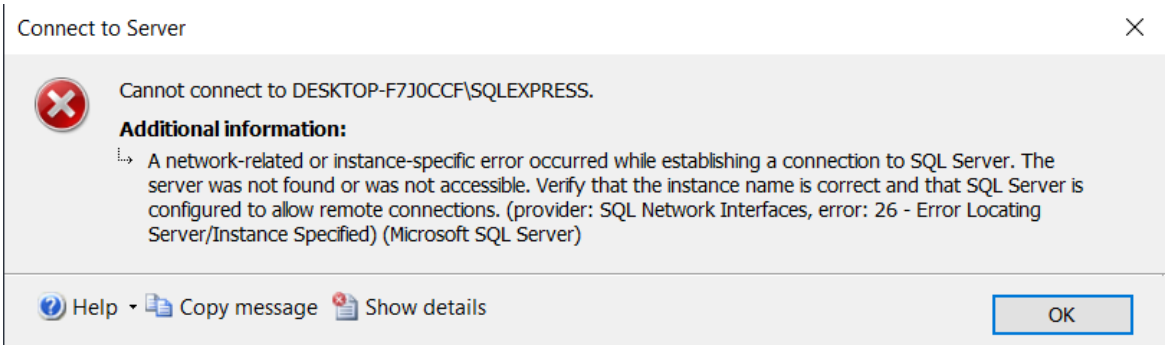
Nakon pokretanja programa dobija se radno okruženje i otvoren prozor *Connect to Server* :



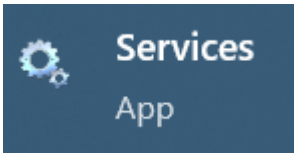
U prozoru *Connect to Server* potrebno je izabrati opciju *Connect*. Na mestu *Server name* će stojati ime_računara\SQLEXPRESS ako prilikom instalacije nije drugačije naznačeno.

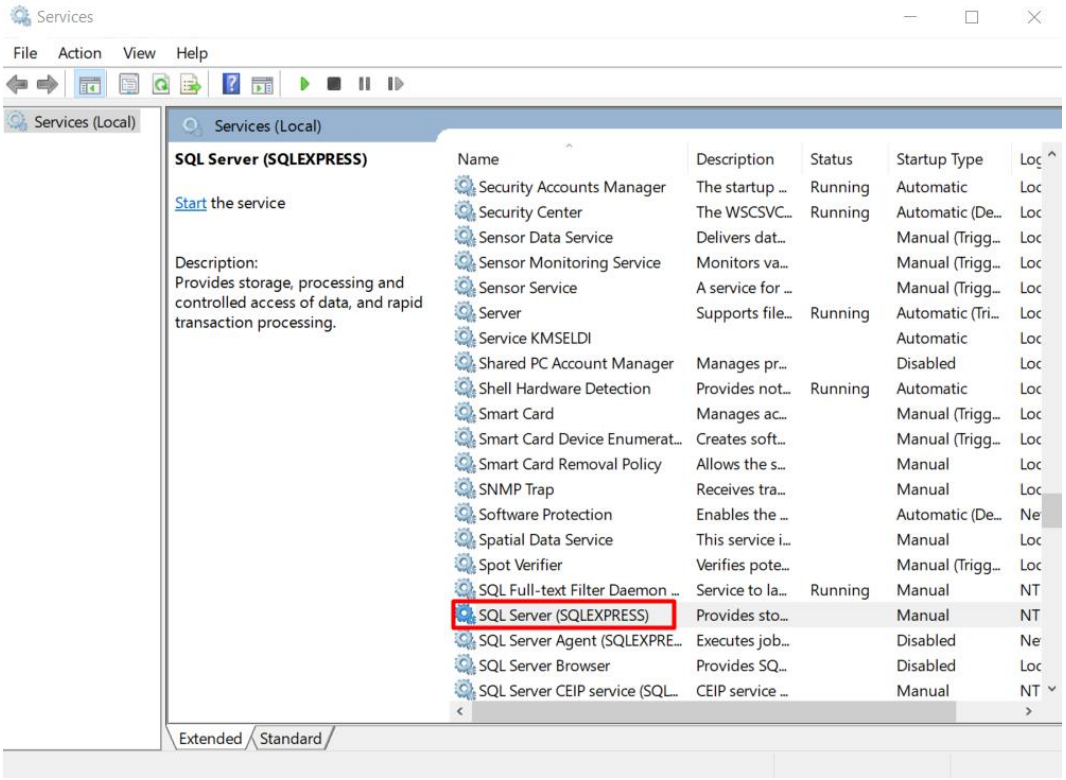


Ukoliko prilikom konektovanja na server dođe do greške prikazane na slici ispod, to je znak da SQL Server nije pokrenut na računaru.

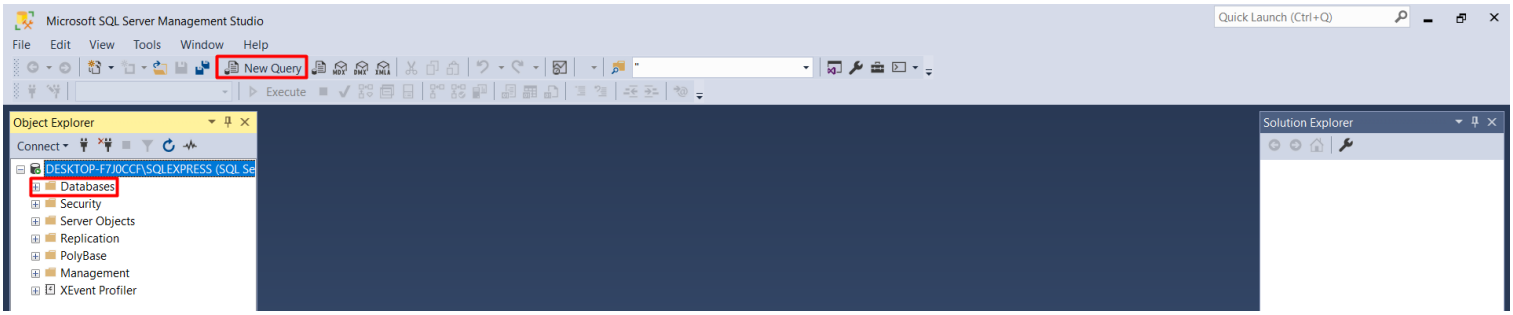


Da bi se SQL Server pokrenuo pristupa se servisima (*Services*) i pronalazi se servis *SQL Server (SQLEXPRESS)* i pokreće se desnim klikom na njega i izborom opcije *Start*.





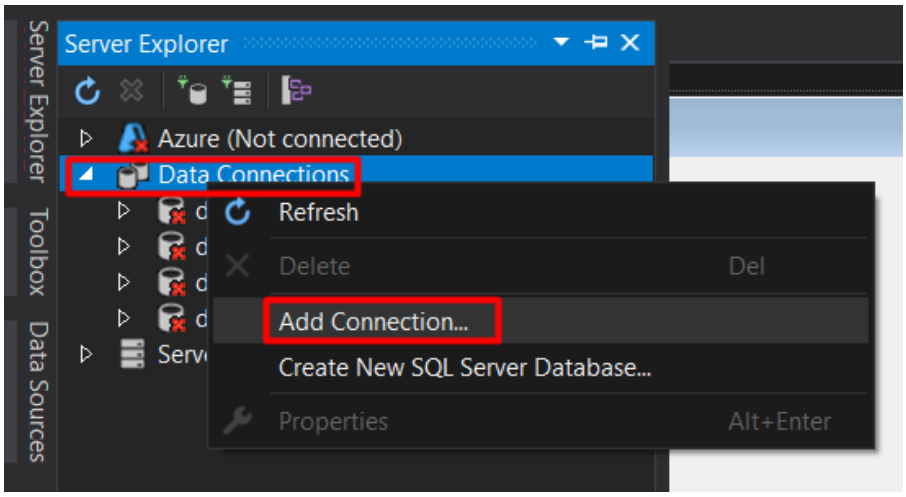
Nakon pokretanja potrebnog servisa ponovo biramo opciju *Connect* u prozoru *Connect to Server* i dobijamo radno okruženje sa *Object Explorer*-om na levoj, *Solution Explorer*-om na desnoj strani i praznim mestom u sredini. Upite kreiramo izborom opcije *New Query* iz gornje trake sa opcijama (ili kombinacijom tastera *CTRL* i *N*). Unutar *Object Explorer*-a će se nalaziti sve baze podataka koje kreirate, zajedno sa njihovim tabelama, pogledima itd.



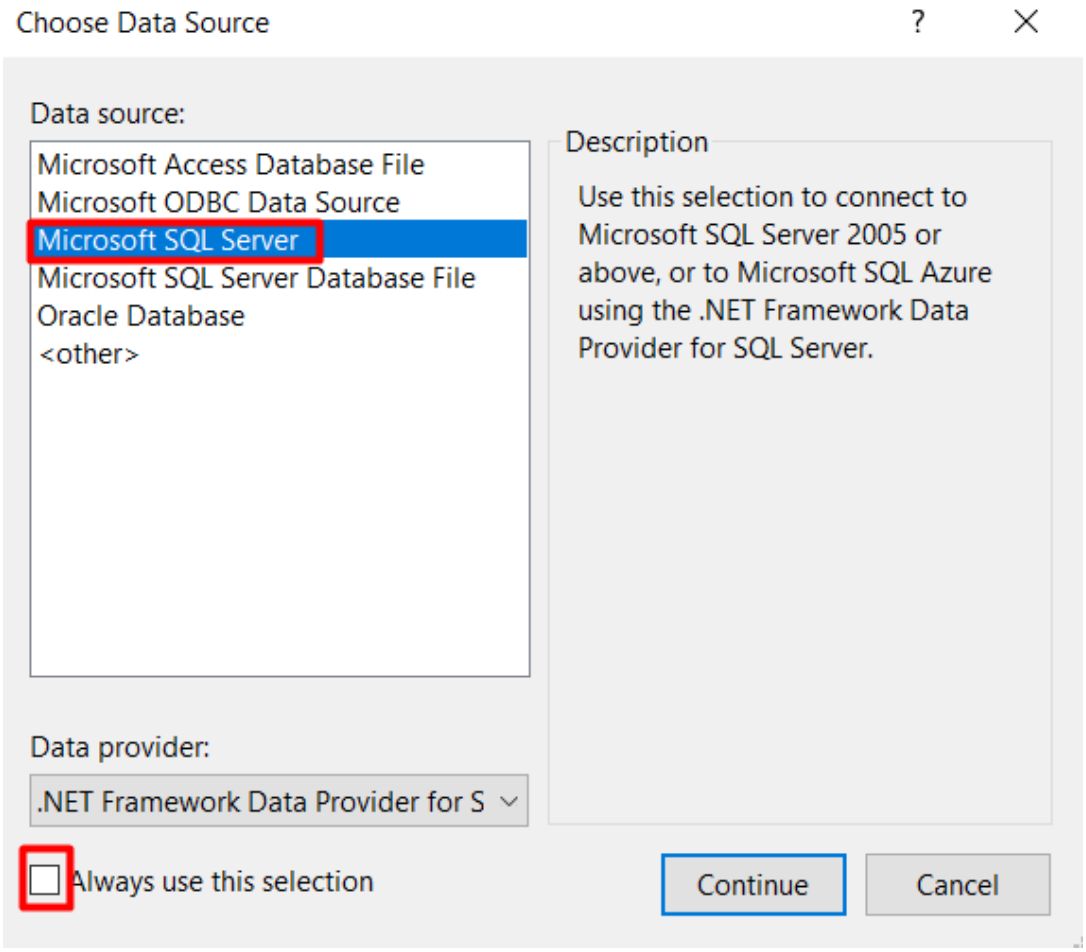
Nakon kreiranja novog upita možemo birati nad kojom bazom podataka želimo da ga izvršimo, izborom baze iz padajućeg menija u gornjoj traci sa alatima. Izborom opcije *Execute* izvršavamo kreirani upit.

Povezivanje baze podataka sa projektom – početni koraci

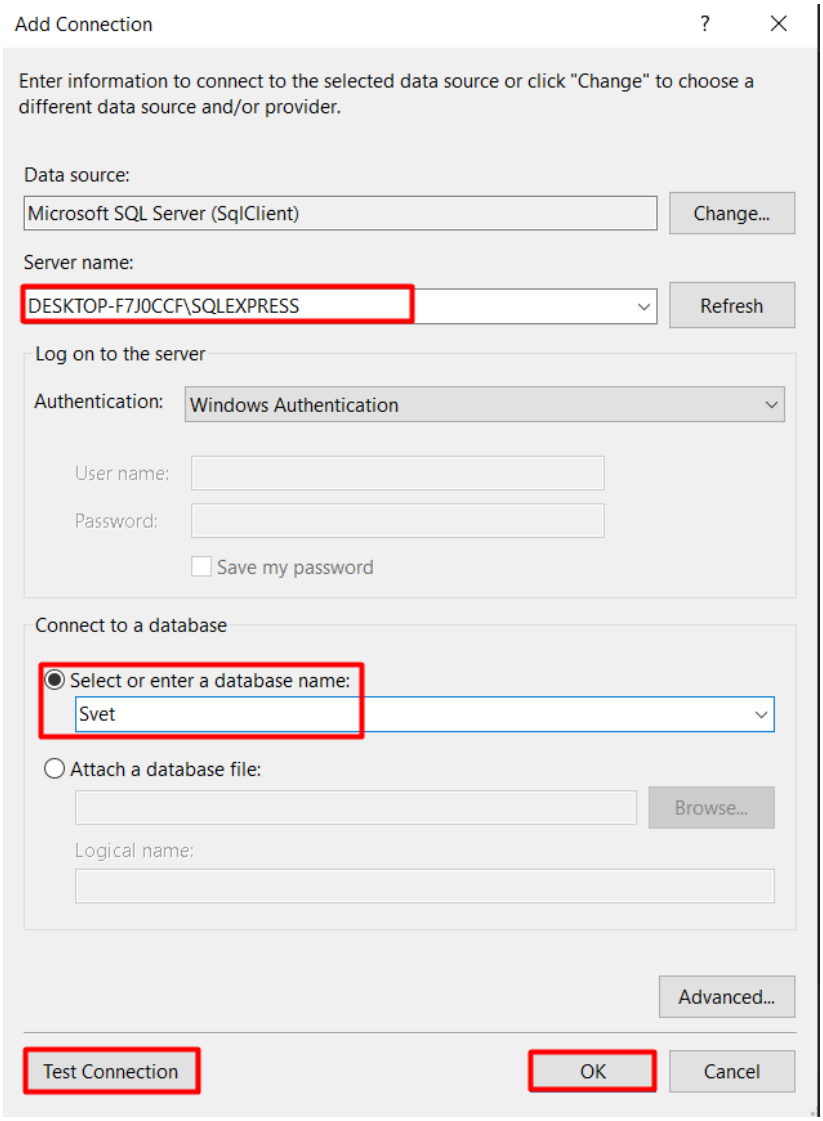
Unutar *Server Explorer*-a desnim klikom na *Data Connections* biramo opciju *Add Connection…* (ukoliko se *Server Explorer* ne nalazi na levoj strani radne površine otvaramo ga iz gornjeg menija biranjem opcija *View -> Server Explorer*.



U prozoru *Choose Data Source* biramo opciju *Microsoft SQL Server*, a polje *Always use this selection* ostavljamo nečekirano. Zatim biramo opciju *Continue*.

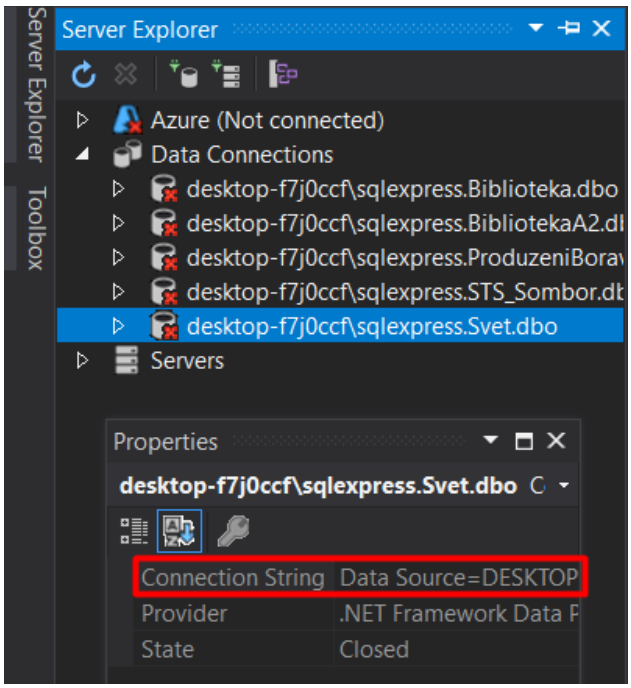


U polje *Server name:* upisujemo ime servera (isto koje je i u SQL Management Studio-u). Zatim u padajućem meniju biramo bazu podataka za koju želimo da dodamo konekciju. Nakon toga pritiskom na dugme *Test Connection* dobijamo povratnu informaciju da li je konekcija uspešna (*Test connection succeeded* ako jeste). Ukoliko je konekcija uspešna biramo dugme *OK*.



Konekcioni string

Konekcioni string koji nam je kasnije potreban za povezivanje sa bazom možemo naći u Properties-u od kreirane konekcije na bazu podataka.



Konekcioni string nam koristi da bi projekat znao na koji server i koju bazu podataka je potrebno da se poveže.

Kopirani tekst konekcionog stringa čuvamo unutar tekstualne promenljive proizvoljnog naziva (u primerima i priručniku promenljiva će biti nazivana *konekcioniString*).

```
string konekcioniString = @"Data Source=DESKTOP-F7J0CCF\SQLEXPRESS;Initial Catalog=Biblioteka;Integrated Security=True";
```

Klase za rad sa bazama podataka

Da bi koristili klase i metode za rad sa SQL Server bazom podataka na vrhu našeg projekta moramo dodati biblioteku *System.Data.SqlClient*.

```
using System.Data.SqlClient;
```

SqlConnection

SqlConnection klasa u C# predstavlja vezu između aplikacije napisane u C# i baze podataka u kojoj se nalaze podaci koje aplikacija treba koristiti. Ova klasa omogućava otvaranje, zatvaranje i upravljanje vezom između aplikacije i baze podataka.

Da bi se koristila SqlConnection klasa, prvo se uključuje biblioteka za rad sa bazama podataka (*System.Data.SqlClient*). Zatim se kreira nova instanca SqlConnection klase i prosleđuje joj se string koji sadrži informacije o željenoj bazi podataka, kao što su naziv servera, naziv baze podataka, korisničko ime i lozinka (konekcioni string).

```
// Kreiranje instance klase SqlConnection
SqlConnection konekcija = new SqlConnection(konekcioniString);

// Otvaranje veze ka bazi podataka
konekcija.Open();

// Zatvaranje veze ka bazi podataka
konekcija.Close();
```

SqlCommand

SqlCommand klasa u C# je deo ADO.NET biblioteke i koristi se za izvršavanje SQL naredbi nad bazom podataka. SqlCommand klasa omogućuje da izvršimo različite vrste SQL naredbi, kao što su SELECT, INSERT, UPDATE, DELETE, itd.

```
/* Kreiranje instance klase SqlCommand
   (s tim da konekcija predstavlja instancu SqlConnection klase) */
SqlCommand komanda = new SqlCommand("SELECT * FROM Tabela", konekcija);

// SQL upit može da se čuva i unutar promenljive
string select = "SELECT * FROM Tabela";
SqlCommand komanda = new SqlCommand(select, konekcija);
```

Ukoliko se unutar upita komande nalaze i parametri njih dodajemo na sledeći način :

```
SqlCommand komanda = new SqlCommand(SQLupit, konekcija);

komanda.Parameters.AddWithValue("@ImeParametra", vrednost);
```

SqlDataReader

SqlDataReader klasa u C# omogućava programerima da pročitaju podatke iz SQL Server baze podataka. Ova klasa se koristi u kombinaciji sa SqlCommand klasom koja se koristi za izvršavanje SQL upita na bazi podataka. Kada se izvrši SQL upit korištenjem SqlCommand klase, SqlDataReader klasa se koristi za čitanje podataka iz rezultujućeg skupa podataka. SqlDataReader klasa čita podatke u jednom smeru, red po red, tako da omogućava brzo i efikasno čitanje velikih skupova podataka.

```
            string select = "SELECT * FROM Tabela";
            SqlCommand komanda = new SqlCommand(select, konekcija);

            // Izvršavanje čitanja podataka nad upitom iz komande
            SqlDataReader citac = komanda.ExecuteReader();

            while (citac.Read())
            {
                // Kod koji se izvršava dok se čitaju podaci
            }

            // Zatvaranje čitača - prekid čitanja podataka iz komande
            citac.Close();
```

SqlDataAdapter

SqlDataAdapter klasa u C# se koristi za preuzimanje podataka iz baze podataka i njihovo skladištenje u *DataSet* ili *DataTable* objekat. *DataSet* objekat može sadržati više *DataTable* objekata koji su usklađeni s tabelama u bazi podataka. Nakon što su podaci učitani u *DataSet* ili *DataTable* objekat, aplikacija ih može koristiti na razne načine.

```
            // Kreiranje instance klase SqlDataAdapter
            SqlDataAdapter adapter = new SqlDataAdapter(SQLupit, konekcija);

            // Kreiranje tabele - instance klase DataTable
            DataTable podaci = new DataTable();

            // Popunjavanje kreirane tabele podacima iz adapter-ovog upita
            adapter.Fill(podaci);
```

Rad sa ListView-om

ListView kontrola se na formu dodaje preuzimanjem iz *Toolbox*-a.

- ***Dodavanje kolona*** - Kolone se unutar ListView-a mogu dodavati ili preko svojstva **Columns** unutar *Properties*-a ili unutar koda forme (imeListView.Columns.Add("ImeKolone");).
- ***Prikaz kolona*** – Da bi videli naslove i podatke u kreiranim kolonama svojstvo **View** menjamo na vrednost **Details** (takođe ili preko *Properties*-a ili unutar koda (imeListView.View = View.Details;).
- ***Selektovanje celog reda*** – Ukoliko želimo da se klikom na red u ListView-u selektuju sve kolone tog reda menjamo svojstvo **FullRowSelect** u vrednost **True**.
- ***Selektovanje većeg broja redova*** – Ukoliko želimo da omogućimo/onemogućimo selektovanje većeg broja redova unutar ListView-a menjamo svojstvo **MultiSelect** u vrednost **True/False**.

Ukoliko želimo da određeni red bude selektovan :

```
imeListViewa.SelectedIndices.Add(indeks od željenog reda);
```

Ukoliko želimo da određeni red bude vidljiv (ukoliko je lista dugačka i potrebno je skrolovanje metoda *EnsureVisible()* će automatski dovesti do željenog reda) :

```
imeListViewa.Items[indeks od željenog reda].EnsureVisible();
```

Popunjavanje kolona podacima

Da bi popunili kolone podacima koristimo ranije pomenute klase za rad sa bazama podataka.

Primer :

```
// Dok čitač prolazi kroz podatke na osnovu prosleđenog upita kod unutar petlje će se izvršavati
            while (citac.Read())
            {
                /* Kreira se stavka ListView-a, a u konstruktor se postavlja prva željena vrednost u
tekstualnoj vrednosti */
                ListViewItem prikaz = new ListViewItem(citac[0].ToString());
```

```
                /* Na kreiranu stavku se dodaju SubItem-i, onoliko njih koliko podataka želimo da
izvučemo iz upita
                   Broj polja ne sme biti veći od broja polja koje vraća upit */
                prikaz.SubItems.Add(citac[1].ToString());
                prikaz.SubItems.Add(citac[2].ToString());
                prikaz.SubItems.Add(citac[3].ToString());
                prikaz.SubItems.Add(citac[4].ToString());
                // Kreirana stavka sa svojim SubItem-ima se dodaje ListView-u
                imeListViewa.Items.Add(prikaz);
            }
```

Rad sa ComboBox-om

Popunjavanje *ComboBox*-a podacima iz baze podataka se vrši pomoću korišćenja klasa *SqlDataAdapter* i *DataTable*.

```
            SqlDataAdapter adapter = new SqlDataAdapter(SQLupit, konekcija);

            DataTable podaci = new DataTable();
            adapter.Fill(podaci);

            // Popunjena tabela se dodaje kao DataSource
            imeComboBoxa.DataSource = podaci;

            // Za ValueMember se uzima unikatna vrednost iz upita
            imeComboBoxa.ValueMember = "UnikatnoPoljeIzTabele";

            // Za DisplayMember se uzima polje čija vrednost treba da se prikaže u ComboBox-u
            imeComboBoxa.DisplayMember = "PoljeIzTabele";
```

Rad sa DataGridView-om

DataGridView kontrola se na formu dodaje preuzimanjem iz *Toolbox*-a.

```
            SqlDataAdapter dataAdapter = new SqlDataAdapter(SQLupit);

            DataTable podaci = new DataTable();
            dataAdapter.Fill(podaci);

            // Podatke iz tabele dodeljujemo DataGridView-u
            imeDataGridViewa.DataSource = podaci;
```

Preko svojstva *ReadOnly* se omogućava/onemogućava da se vrši unos u polja *DataGridView*-a.

Rad sa Chart-om

Chart kontrola se na formu dodaje preuzimanjem iz *Toolbox*-a.

```
            // Dodeljujemo izvor podataka našem chart-u
            imeCharta.DataSource = dataAdapter;
            // Na x osu postavljamo kolonu po kojoj želimo da se mere podaci
            imeCharta.Series[0].XValueMember = "Kolona X ose";
            // Na y osu postavljamo kolonu koja po kojoj će se meriti podaci sa x ose
            imeCharta.Series[0].YValueMembers = "Kolona Y ose";
            // Unosimo podatke unutar chart-a
            imeCharta.DataBind();
```