Klase i objekti u C#

UVOD

Uvod

'U ovom predavanju govori se o klasama i objektima, kao i o „svojstvima". Takodje, o skrivanju podataka, konstruktorima , i dodavanju polja i dodavanju „svojstava" klasama.

Ključna pitanja:

Kako se kreiraju klase i objekti u Visual C#?

Šta su to "svojstva"?

Napomena: Pogledati Java kurs na Digiš-u.

„Svojstva“ u Visual C# ?

01

KLASE ?

Telo klase sadrži metode i polja, naime, promenjive u klasi se nazivaju polja.

Pre nego objasnimo šta su to „svojstva“ u Visual C#, podsetimo se prvo šta su to klase. „Klasa“ tj. klasa podataka je specijalan tip podataka, složen tip podataka koji ima niz komponenti, i koji je definisan od strane programera da opiše tj. imitira (simulira) realno postojeće objekte (*real world objects*). Komponente klase su „atributi“ tj. članovi klase („*members*“) i akcije tj. „metode“ („*methods*“). Npr. ako posmatramo klasu Pas, neki objekat u klasi Pas, npr. objekat Pas1 ima atribute: težinu, starost, boju, itd., i može da obavlja akcije: spavati, jesti, lajati, itd. Klasni mehanizam u C# ili Javi ili C++ omogućuje da se kreiraju virtuelni psi u okviru nekog programa pomoću kreiranja odgovarajuće klase (klase Pas). Pogledajmo klasu Pas, sa samo jednim atributom:

```
class Dog{private int age;
public int GetAge(){ return age;}
public void SetAge(int value){ age = value;}
}
```

A klasa Dog se koristi pomoću njenih *setter* i *getter* metoda, npr.

```
Dog Dog1; Dog1 = new Dog(); //kreiranje objekta Dog1
Dog1.setAge(10);
MessageBox.Show(Dog1.getAge().ToString());
```

Telo klase sadrži metode i polja, naime, promenjive u klasi se nazivaju polja. Pomoću ključnih reči *private* i *public* u definicijama polja i meode, može se kontrolisati pristup spolja. Metoda/polje je privatno ako je dostupno samo unutar klase. Ovo se postiže dodavanjem reči *private* ispred deklaracije polja/metode. Ako se ne stavi ni *private* ni *public,* onda podrazumeva se (*default) private. Metoda/polje je javno, public*, ako je dostupno i izvan i unutar klase. Ovo se dobija ako stavite reč *public* ispred deklaracije polja/metode. Za svako polje/metodu posebno se defineše da li je *private* ili *public*.

SVOJSTVA ?

Visual C# definiše properties tj. „svojstva“ (svojstva objekata tj. klasa) kao kombinaciju tj. spoj polja i metode.

Ako proanaliziramo gornji program koji opisuje klasu, gde se za pristup privatnom polju ***age*** koriste metode GetAge() i SetAge(), vidimo da se za pristup privatnom polju umesto

Dog1.age, koje bi se koristilo da je polje ***age*** javno, koristi Dog1.setAge() i Dog1.getAge(), jer je ovo polje privatno, što je nezgrapno i konfuzno, jer se za dobijanje vrednosti polja koristi sintaksa bazirana na metodama. Naime, korišćenje javnih polja je jasnije i elegantnije u odnosu na upotrebu *setter* i *getter* metoda kod privatnih polja. „Svojstva“ tj. „*properties*“ u Visual C# omogućuju da se koristi sintaksa bazirana na poljima a ne metodama, kada želimo da pristupimo nekom privatnom polju.

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. Naime, svojstvu se pristupa kao da je polje (koristi se sintaksa karkteristična za polja), ali kompjuter automatski prevodi ovu sintaksu, i pri tome poziva pristupne metode.

„SVOJSTVO“-DEFINICIJA

Evo sintakse kojase koristi za „svojstva“.

U Visual C#, potrebno je deklarisati svojstvo (*property)*, i sintaksa je ova:

```
Modifier Type PropertyName
{
get
{
…..//instrukcije get metode
}
set
{
….//instrukcije set metode
}
}
```

SVOJSTVA-PRIMER

Evo primera upotrebe svojstava klase.

Npr.

```
public int Age {
get
{
…..//instrukcije get metode
}
set
{
….//instrukcije set metode
}
}
```

Takodje potrebno je deklarisati polje gde se memoriše svojstvo. Npr.

private int mAge; tj.

Modifier Type mPropertyName;

UPOTREBA SVOJSTAVA

Evo primera upotrebe svojstava klase.

Konvencija je da se ime tj. naziv javnih svojstva piše velikim početnim slovom, a privatnih polja malim početnim slovima. Takodje, u *set* metodi se koristi skriveni (implicitni) parametar nazvan *value*, što će biti ilustrovano kasnije, a koji se koristi da se zada vrednost polju.

Upotrebom svojstava, npr. u klasi Dog, umesto

```
Dog1.setAge(10);
MessageBox.Show(Dog1.getAge().ToString());
```

imamo :

```
Dog1.Age=10;
MessageBox.Show(Dog1.Age.ToString());
```

Ovde je vrednost skrivenog (implicitnog) parametra *value* jednaka 10, i odredjena je gornjom instrukcijom:

```
Dog1.Age=10;
```

Dakle implicitni parametar se predaje set metodi preko operatora =, a ne preko malih zagrada ().

Klase i objekti u Visual C#

02

KLASE I OBJEKTI U VISUAL C#

Visual C# je napravljen tj. dizajniran tako da omogućuje lako kreiranje, i lako korišćenje klasa i objekata.

C# je u potpunosti objektno orijentisan jezik, gde se koriste pojam „klasa“, i pojam „objekat“. Klasa je u stvari jedan specijalan tip podatka, složen tip podatka, tip podatka koji sam programer kreira prema potrebi, i to da bi opisao realno postojeće objekte. A objekat je konkretan primerak tog složenog tipa podatka. „Klase“ obuhvataju ne samo varijable već i funkcije tj. metode objekata. Metode tj. funkcije objekata opisuju akcije koje mogu neki objekti da obavljaju. Jasno je da neki objekat nije okarakterisan samo atributima koji opisuju “izgled” nekog objekta već i „metodama“ koje objekat može da vrši.

Do sada smo koristili C# i bez znanja o programiranju objekata i klasa, medjutim vrlo je važno naučiti kako se koristi o.o. programiranje u C#, ako se želi primeniti C# u složenijim projektima tj. u realno-postojećim projektima, dakle projektima kakve možemo očekivati u realnom životu. Visual C# je napravljen tj. dizajniran tako da omogućuje lako kreiranje, i lako korišćenje klasa i objekata. Dole je dat primer gde je opisano kako da se kreira klasa i objekat pomoću Visual C#.

KLASE-PRIMER:

Evo primera kreiranja klasa u C#.

1. Započeti nov projekat, i izabrati **Add New Item** iz **File-menu**.

2. U **Add-*New-Item***-dijalogu, izabrati **Class,** i nazvati novoformiranu klasu **Knjiga.cs**

3. U **Add-*New-Item***-dijalogu kliknuti **Open** da bi se otvorila klasa **Knjiga.cs**. Onda je potrebno ukucati programski kod, kao što je dole prikazano, da bi se definisala klasa Knjiga:

```
public class Knjiga{
        private string mNaslov;
        public string Naslov
{
        get
        {
                return mNaslov;
        }
        set
        {
                mNaslov = value;
        }
}
```

KLASE-PRIMER:

Evo primera kreiranja klasa u C#.

4. Vratiti se na „formu" u projektu, dodati „dugme", i pomoću ***Click-event-hendler**-a*, treba dodati programski kod koji je dole prikazan.

```
Knjiga Knjiga1; Knjiga1 = new Knjiga ();
Knjiga1.Naslov = ‘’Dictionary’’;
MessageBox.Show (’’Naslov knjige je:’’ + Knjiga1.Naslov);
```

5. A onda izvršiti aplikaciju tj projekat.

Proanalizirajmo detalje programskog koda u koracima 3 i 4. U koraku 3, kreirali smo klasu „Knjiga", i pri tome definisali samo jedno „svojstvo" objekata, a to je "Naslov". A u koraku 4, smo u cilju korišćenja klase Knjiga, izvršili kreiranje objekta „Knjiga1". Kreiranje objekta obuhvatilo je dva koraka, prvo, deklarisanje varijable Knjiga1 tipa Knjiga, i drugo, kreiranje tj. Inicijalizacija konkretnog objekta tipa Knjiga pomoću ključne reči ***new***. Bitno je razlikovati izmedju klase i objekta, naime klasa je tip podataka, složen ali vrlo koristan tip podataka, koji definiše tip objekta, a objekat je konkretan primerak („instanca") tog tipa podataka koji zovemo klasa. Deklaracija i inicijalizacija objekta Knjiga1 se može uraditi i ovako:

```
Knjiga Knjiga1= new Knjiga ();
```

IMPLICITNI PARAMETAR

Važno je uočiti, da metoda set koristi jedan implicitni parameter, a to je parameter „value", dakle, taj parameter value nije dat u listi parametara izmedju dve male zagrade metode,

Pogledajmo definiciju klase Knjiga. Javno svojstvo Naslov je memorisano kao "privatna" varijabla "mNaslov" (memorisan Naslov). *Definicija svojstva* Naslov obuhvata dve metode, *get* i *set*, koje koristimo za učitavanje i menjanje varijable mNaslov. I ova varijabla mNaslov se može učitavati ili menjati samo pomoću ove dve metode, jer mNaslov je privatna, *private*, varijabla, a get i set su javne, *public*, metode. Alternativno, može se izbaciti metoda *set,* ako se želi dozvoliti samo učitavanje, a zabraniti menjanje varijable mNaslov. Ako bi se varijabla (tj. polje) mNaslov proglasila da je javna, *public*, onda bi se moglo izostaviti definisanje metoda *get* i *set,* ali upotreba metoda *get* i *set* ima prednosti, npr. ove metode se mogu tako definisati da ulazni podatak je validan, npr. provera da ulazni pdatak počinje velikim slovom i da nije prazna varijabla (*blank*).

Važno je uočiti, da metoda *set* koristi jedan implicitni parameter, dakle implicitni ulazni podatak, a to je parameter „*value",* dakle, taj parameter *value* nije dat u listi parametara izmedju dve male zagrade metode, kao što bi mogli očekivati, dakle nije eksplicitan parameter. Medjutim, Visual C# omogućuje da se koristi kao implicitni parametar, naime, kod zadavanja vrednosti, npr.

Knjiga1.Naslov = ‘'Dictionary'';

automatski je podešeno da se unosi vrednost tj podatak za parameter *value*. Dakle, pomoću operatora = zadaje se vrednost parametra *value*, konkretno u ovom slučaju je *value* jednako "Dictionary". Takodje, tip ovog parametra se automatski zadaje da je isti kao tip podatka mNaslov.

ELEMENTI KLASE

Po pravilu svaka klasa je poseban fajl, ali to nije obavezno, jer se može npr. dodati definicija neke klase onom fajlu koji sadrži program za „formu" (Form code file).

Napomenimo da klasa može obuhvatati tzv. „statičke" članove. Naime, statički članovi (statičke metode i svojstva) mogu se pozivati a da nije potrebno kreirati objekte. Naime, postoje i takve klase gde postoje osobine i metode koje nisu asocirane objektima već pripadaju samoj klasi, i ove osobine i metode se zovu statički članovi klase, *static members*, gde se koristi ključna reč *static*.

Po pravilu svaka klasa je poseban fajl, ali to nije obavezno, jer se može npr. dodati definicija neke klase onom fajlu koji sadrži program za „formu" (*Form code file*).

Elementi klase tj fajla koji definiše neku klasu su elementi:

using statements,

namespace declaration,

comments,

class statement,

class-level variables,

method declarations,

directives (# instrukcije kompajleru),

generated code (delovi programa automatski generisani od strane form dizajnera, *Form Designer*).

KONSTRUKTOR

Kod Visual C#, automatski se kreira jedan inicijalni konstruktor kada se kreira tj. dodaje nova klasa.

Dva prelomna momenta u trajanju tj korišćenju nekog objekta su kreiranje objekta i uništavanje objekta. Konstruktor – šta je to? E pa konstruktor je specijalna metoda koja omogućuje kreiranje objekta na jedan kontrolisan način, a destruktor je specijalna funkcija koja biva pozvana kod uništavanje objekta Medjutim, destruktori se mnogo manje koriste od strane C# programera, jer pomoću *.NET Framework* se destrukcija objekata obavlja automatski pomoću tehnike nazvane „garbage collection".

Konstruktor je dakle specijalna metoda, i ona ima isto ime kao klasa, npr.

```
public Knjiga ()
{
.....
}
```

I obično je ta metoda javna, *public*. Kod Visual C#, automatski se kreira jedan inicijalni konstruktor kada se kreira tj. dodaje nova klasa pomoću:

Project > Add class.

Medjutim, programer može da kreira dodatne konstruktore.

DESTRUKTOR

Konstruktor se koristi kod većine klasa. Pri tome ideja je da se tako definiše konstruktor da se obezbedi da su objekti uvek validni

Takodje, automatski se i kreira i poziva jedan destruktor. A destruktor je metoda sa istim imenom kao i klasa, a ispred toga se stavlja znak ~. Npr.

```
~Knjiga ()
{
.....
}
```

Konstruktor se koristi kod većine klasa. Pri tome ideja je da se tako definiše konstruktor da se obezbedi da su objekti uvek validni. Dakle, konstruktor se po pravilu koristi da bi se kreirao objekat, i to sa ciljem da bi se obezbedilo ispravno kreiranje objekata. Prema tome, upotrebom konstruktora se izbegavaju greške kod kreiranja objekata. Npr. kod klase „Knjiga“, može se napraviti takav konstruktor koji osigurava da svaki objekat u klasi „Kniga“ ima obavezno definisanog autora i naslov.

Napomenimo, može se definisati klasa i sa nekoliko konstruktora (*overloading constructors*) a ne samo sa jednim konstruktorom, pri čemu svaki pojedini konstruktor ima različite liste parametara (argumenata). Dakle, C# programeri se dosta bave programiranjem konstruktora, dok destruktori se koriste automatski.

Add *Field*, i Add *Property*

03

ADD *FIELD*, I ADD *PROPERTY*

Proširujemo definiciju klasu Knjiga da liči na pravu klasu, koristeći ***Add Field*** *(Dodaj polje) i* ***Add Property*** *(Dodaj svojstvo).*

Prethodno je kreirana klasa Knjiga, kao i objekat te klase: Knjiga1. Pri tome smo uveli samo jedno svojstvo (tj. osobinu), *property*, koje se zvalo „Naslov“. Sada ćemo klasu Knjiga proširiti da ima još jedno svojstvo, i to svojstvo „Autor“, i tada će klasa Knjiga početi da liči na pravu klasu koja može da ima čitavu seriju članova, npr. ISBN, Publisher, Year, Pages, itd. Kada neka klasa ima puno članova i puno objekata, onda u punoj meri dolazi do izražaja moć koncepta o.o. programiranja tj. efekat uvodjenja i korišćenja klasa i objekata.

Proširujemo definiciju klasu Knjiga da liči na pravu klasu, koristeći **Add Field** (Dodaj polje) i **Add Property (**Dodaj svojstvo**)**, i to na sledeći način:

ADD *FIELD*, I ADD *PROPERTY-PRIMER*

Proširujemo definiciju klasu Knjiga da liči na pravu klasu, koristeći ***Add Field*** *(Dodaj polje) i* ***Add Property*** *(Dodaj svojstvo). Dole je primer.*

1. Svojstvo **(tj.** osobina) Autor je slična sa svojstvom Naslov, i treba je dodati klasi Knjiga. Medjutim, ovaj put ćemo koristiti ugradjene alate u okviru Visual C#, koji štede kucanje. Naime, treba izabrati:

View > Class View,

i pronaći prethodno kreiranu klasu Knjiga, i zatim izabrati:

Add > Add Field.

2. U dijalogu Add Field, specificirati *private* za Field access, i *string* za Field type**,** i *mAutor* za Field name**,** i onda kliknuti Finish.

3. Kliknuti (desni klik) ponovo klasu Knjiga, i izabrati

Add Property.

U dijalogu Add Propery, specificirati *Property name* kao Autor, i specificirati *Property access* kao *public*, i *Property type* kao *string*, i specificirati *Accessors* kao *get/set Acccessors,* i kliknuti Finish.

ADD *FIELD*, I ADD *PROPERTY-PRIMER*

Dole je primer upotrebe „add field“ i „add property“.

4. Ići na View > Class View, i onda duplo kliknuti *property* Autor. Ovim se otvara kod-editor na početku svojstva Autor, i onda treba ukucati metode *get* i *set,* i to izgleda ovako:

```
public string Autor{
get
{return mAutor;
}
set
{                    mAutor = value;
}            }
```

5. Visual C# automatski kreira konstruktor za klasu koja se kreira pomoću Project > Add class. Možemo pronaći ovaj konstruktor pod imenom Knjiga(). Ovaj konstruktor se poziva kad neki objekat u klasi Knjiga treba da se kreira. Medjutim, sada ćemo modifikovati ovaj konstruktor Knjiga(), pa umesto ovog inicijalnog konstruktora Knjiga() bez ijednog parametra, imaćemo konstruktor sa dva parametra: Knjiga(autor, naslov):

```
public Knjiga (string autor, string naslov)
{                                   mAutor = autor;
                                    mNaslov = naslov;
}
```

AUTOMATSKI KONSTRUKTOR

Izvršena je modifikacija automatskog konstruktora.

Rezultat gore prikazanih modifikacija konstruktora, koji je od Knjiga() postao Knjiga(autor, naslov), je da se više ne može kreirati neki objekat u klasi Knjiga a da se ne specificira autor i naslov, naime, ako se ukuca:

```
Knjiga Knjiga1 = new Knjiga();
```

neće se moći izvršiti kompilacija, javiće se kompilaciona greška, jer konstruktor Knjiga() bez parametara više ne postoji, već sada postoji konstruktor Knjiga(autor, naslov), tako da mora da se ukuca npr.:

```
Knjiga Knjiga1 = new Knjiga (”Vujaklija”, ”Recnik”);
```

I time se vrši inicijalizacija objekta.

Da napomenemo, da ako je „polje“, *Field*, okarakterisano kao *private*, onda je obezbedjeno da se može prisatupiti polju samo preko metode *set* **i** *get***.** Ako se umesto *get/set* pristupa izabere ili *get* ili *set*, ona će se izabrati ili *read-only* ili *write-only* pristup. Takodje da napomenemo, znamo da klase se mogu definisati da može postojati ne samo jedan, već više konstruktora, ali sa različitim listama argumenata, npr. Knjiga(autor, naslov) i Knjiga(naslov).

Sakrivanje podataka - primer

04

DATA HIDING

Kada pozivate metodu ConsoleWriteLine, ne interesuje vas detalji te metode i te klase, već jednostavno želite da upotrebite tu metodu.

U cilju zaštite podataka, bolje je organizovati klase da nisu lako dostupne iz ostalih delova programa. Ova tehnika se zove ‘’Sakrivanje podataka”, *Data hiding*, a sakrivanje podataka se obezbedjuje pomoću ključne reči *private*. Ako ne želimo sakrivanje podataka onda koristimo reč *public*. Medjutim ako koristimo sakrivanje podataka, onda moramo da obezbedimo pristupne metode, *accessor methods*, i to kao javne, *public methods*, delove klase. Ove pristupne metode se zovu, *set* i *get* metode. I koriste se za učitavanje i menjanje atributa tj varijabli klase. Visual C# koristi vizuelne alate, a takodje i automatizuje, i pojednostavljuje pisanje delova programa.

Radni okvir .NET sadrži hiljade i hiljade već kreiranih i testiranih klasa spremnih za upotrebu, a neke smo već spomenuli, npr. klase Console i Exception. Klase pružaju pogodan šablon i mehanizam za modelovanje raznih entiteta, fizičkih entiteta kao što je npr. mačka, stan, itd., ili apstraktnih entiteta kao što je npr. transakcija. „Enkapsulacija“ tj. „skrivanje informacija“, *data hidi*ng, je važan aspekt definisanja klase. Npr. kada pozivate metodu ConsoleWriteLine, ne interesuje vas detalji te metode i te klase, već jednostavno želite da upotrebite tu metodu.

KLASA KRUG

Evo primera klase koja se zove Krug.

Pogledajmo sledeću klasu nazvanu Krug, napisanu u jeziku C#:

```
class Krug
{
        double Povrs()
        {return Math.PI*rad*rad;}
 int rad;
```

} Ova klasa sadrži jednu „običnu“ metodu (metodu koja nije glavna metoda *main()*), metodu Povrs(), i jedno polje, polje „rad“. Napomenimo, da klasa Math (iz .NET okvira) sadrži metode za izvršavanje matematičkih operacija, i ova klasa takodje sadrži polje Math.PI što je broj 3.1415.....Korišćenje klase je slično sa korišćenjem drugih tipova varijabli, a to obuhvata deklarisanje varijabli i inicijalizacija, i pri tome se koristi ključna reč *new*, npr.

```
Krug c1;
c1 = new Krug();
```

ili

```
Krug c1, c2;
c1 = new Krug();
c2 = c1;
```

KLASA KRUG

Evo primera klase koja se zove Krug.

Klasa Krug je praktčno neupotrebljiva, jer je pristup njoj potpuno zatvoren jer se nigde ne koristi ključna reč *public*. Ako enkapsulirate metode i varijable unutar klase, klasa je nepristupačna. Polje „radius“ i metoda „Povrs“ su definisani unutar klase i nemogu se „videti“ izvan klase. Tako da iako kreirate objekat npr. c1, ne možete da pristupite njegovom polju niti da pozovete metodu Povrs(). Pomoću ključne reči *private* se vrši enkapsulacija, a ako se ne stavi ta reč, onda se ona podrazumeva. Da bi se neko polje ili metoda proglasili javnim koristi se reč „public“. Npr.

```
class Krug{
                public double Povrs()
{
 return Math.PI*rad*rad;
}
private int rad;
}
```

Ovde je polje „rad“ deklarisano kao privatno polje, pa je dostupno unutar klase. Zato, metoda Povrs() može da pristupi polju „rad“, jer je ova metoda unutar klase. Metoda Povrs() je javna. Ali, i dalje ne možete da inicijalizujete polje „radius“ izvan klase, pa da bi se ovo uradilo može se koristiti konstruktor, i to javni konstruktor.

AUTOMATSKI KONSTRUKTOR

Evo primera klase „Krug“, sa konstruktorom koji je automatski generisan.

U jeziku C# svaka klasa mora da ima konstruktor. Ako ne definišete konstruktor za neku klasu, kompajler će automatski napisati konstruktor umesto vas. Evo primera klase „Krug“, sa konstruktorom koji je automatski generisan, i koji inicijalizuje automatski polje „rad“ na vrednost nula:

```
class Krug{
        public Krug() //automatski konstruktor postoji
        (
                rad = 0;
        }
public double Povrs()
{ return Math.PI*rad*rad;}
private int rad;
}
```

U jeziku C#, automatski konstruktor ne prima nikakve parametre. Mogu se napisati novi konstruktori, koji su različiti od ovog automatskog. Npr.

```
public Krug(int initRad) //preopterećeni konstruktor
        (       radius = initRad;}
```

Objekti u ListBox-u

05

OBJEKTI U LISTBOX-U?

U praksi, često se traži da se prikažu objekti neke klase u „kontroli“ ListBox

U praksi, često se traži da se prikažu objekti neke klase u „kontroli“ *ListBox*, gde je *ListBox* već pomenuta ranije kao jedan od najćešće korišćenih predefinisanih objekata u Toolbox-u u Visual C#. Tada smo koristili ListBox da prikažemo listu elemenata tipa „string“. Ali, u praksi se obično zahteva da se prikazuje lista objekata a ne lista „string“-ova. Svaki objekat u Visual C# može da koristi metodu

ToString().

A „kontrola“ ListBox koristi metodu ToString() da prikaže objekte. Ovo se može iskoristiti da se npr. prikažu objekti iz klase Knjiga, klasu koju smo kreirali u prethodnoj glavi.

Medjtim, metoda ToString() se može modifikovati, da prikazuje objekat po vašoj želji. Evo kako to može da izgleda, npr.

OBJEKTI U LISTBOX-PRIMER

Metoda ToString() se može modifikovati.

npr.

1. Sledeći programski kod treba ukucati u *Knjiga.cs* za modifikovanu metodu ToString()*:*

```
public override string ToString()        { return Naslov + ”od” + Autor; }
```

2. Dodati „kontrolu“ ListBox na „formu“ u vašem projektu, i dodati „dugme“ sa sledećim programskim kodom za *Click-event-handler*:

```
listBox1.Items.Clear(); //brisanje postojecih objekata u ListBox-u
listBox1.Items.Add(new Knjiga (”benson””, ”dictionary”)); //dodavanje objekta
listBox1.Items.Add(new Knjiga (”andric”, ”cuprija”)); //dodavanje objekta
//itd.
//instrukcije za listanje objekata, gde se koristi indeks koji indeksira redove tj. objekte u ListBox-u
```

3. Egzekutovati aplikaciju da bi se prikazala lista knjiga, koja je u stvari isprogramirana kao jedna klasa. Gore je korišćena ključna reč *override*, da „nadjača“ postojeću metodu ToString(), a ona će biti detaljnije objašnjena u sledećem predavanju.

FORMA SA „KONTROLAMA“ LISTBOX I BUTTON

Na slici je forma sa „kontrolama“ ListBox i Button.

Sl.1: ListBox objekti

Svojstva - primer

06

SVOJSTVA-PRIMER

Bez upotrebe svojstava pristup poljima je nezgrapan.

Pomoću „svojstava“, *properties*, može pristupiti privatnim poljima (*fields*) u klasi. Naime, u jeziku C# se koriste „svojstva“ da bi se pristupilo privatnim poljima, što je drukčije od klasičnog prilaza, npr. u jeziku C++ se ne koriste „svojstva“ za pristup poljima, pa je sintaksa pristupa privatnim poljima značajno komplikovanija u poredjenju sa C#. I u okviru C# je moguće koristiti ovaj prilaz bez upotrebe „svojstava“, ali onda je sintaksa nespretna tj. komplikovana. Zato je u jeziku C# ugradjen mehanizam koji rezultira u jednostavniju i elegantniju sintaksu, kao što ćemo dole detaljno objasniti. Pogledajmo sledeći primer klasičnog načina pristupa promenljivama, bez upotrebe svojstava:

```
class Position{
public Position(int x0, int y0){ x= x0; y=y0;}
public int GetX(){return x;}
public void SetX(int newX){x = newX;}
public int GetY(){return y;}
public void SetY(int newY){y = newY;}
private int x, y;}
```

Pristup poljima je komplikovan jer ne koristi sintaksu baziranu na poljima već na metodama. Npr.

```
Position xy0= new Position(0,0);
xy0.SetX(10); xy0.SetY(100);
```

A da je polje javno, imali bi: xy0.x=10;

SVOJSTVA-PRIMER

Evo primera upotrebe svojstava.

Umesto ovog „klasičnog“ načina, mogu se primeniti „svojstva“ da bi se definisao pristup poljima. Sintaksa je sledeća,

AccessModifier Type PropertyName

{get{…}

set{….}}

Dakle, svojstvo tj. *property* je kombinacija polja i metode, izgleda kao polje a ponaša se kao metoda. Definicija svojstva, *property*, uključuje u sebe metodu *get* i metodu *set.* Svojstvu pristupate koristeći istu sintaksu koja se koristi za pristup polju, ali kompajler automatski prevodi sintaksu polja u sintaksu metode. Evo primera primene „svojstava“ za pristup poljima,

```
class Position{public Position(int x0, int y0){x = x0; y = y0;}
public int XX{
get { return x; }
set { x = value; }}
public int YY{
get { return y; }
set { y = value; }}
private int x, y;}
```

Kada je potrebno da se koristi svojstvo za pristup, za učitavanje (čitanje) ili za zadavanje (pisanje) vrednosti, može se koristiti sintaksa polja, npr.

```
xy0.XX = 10; xy0.YY = 100;
```

READ -ONLY I WRITE-ONLY SVOJSTVA

Evo primera read-only ili write-only svojstava.

Svojstva se mogu definisati i kao *read-only* ili *write-only*, npr.

```
class Position
{
...
public int XX {get { return x; }}
public int YY {get { return y; }}
private int x,y;
}
```

ili

```
class Position
{
...
public int XX{set { x = value; }}
public int YY{set { y = value; }}
private int x,y;
}
```

MODIFIKATOR PRISTUPA SVOJSTVU

Unutar deklaracije svojstva može da se nadjača (override) pristup svojstvu za pristupne metode get i set.

Može da se navede modifikator pristupa svojstvu (public, private, protected) kada se deklariše svojstvo. Takodje, unutar deklaracije svojstva može da se nadjača (*override*) pristup svojstvu za pristupne metode *get* i *set*. Npr.

```
class Position
{
…..
public int XX
{
get{ return x;}
private set{x = value;}
}
public int YY
{
get{ return y;}
private set{y = value;}
}
private int x,y;
}
```

Automatsko generisanje „svojstava“

07

AUTOMATSKO GENERISANJE „SVOJSTAVA“

Može se primeniti „automatsko“ generisanje svojstava, gde se primenjuje kondenzovana sintaksa.

Jezik C# je tako napravljen da što više olakšava posao programeru, i zato se i zove C *sharp*. U tom kontekstu, treba pomenuti mogućnost automatskog generisanja svojstava. Naime, ako napišete sledeće instrukcije:

```
class Square
{public int Side{ get; set; }
...}
```

Onda će kompajler automatski generisati instrukcije koje otprilike izgledaju ovako:

```
class Square
{private int _side;
public int Side{
get{return this._side;}
set{this._side = value;}
}
...}
```

Prema tome, vrlo lako i brzo se može primeniti pristup poljima preko „svojstava“. Ali, treba naglasiti da ako koristite automatsko generisanje *get* i *set* metode, onda morate da pomenete obe ove metode, a ne samo jednu od njih, dakle ovo automatsko kodiranje ne može se koristiti kao *read-only* ili *write-only*.

Operatori „is“ i „as“

08

OPERATORI „IS“ I „AS“

Operator „is“ ima vrednost ili „true“ ili „false“.

Operator „is“ se koristi za proveru da li je tip objekta onaj koji mislite da jeste, npr.:

```
object o1 = k1;
if ( o1 is Kvadrat)
{.....      }
```

Operator „is“ ima vrednost ili „true“ ili „false“. Npr. ako je tip o1 isti sa Kvadrat onda je vrednost true.

Pored operatora “is“ postoji operator „as“ koji je sličan i ima sličnu ulogu.

Osobine klasa

09

KLASA-PRIMER

Evo primera kreiranja klase.

```
class Kvadrat{double Povrs()
{return strana*strana;}
double strana;}
```

Generalno gledano, telo neke klase sadrži:

metode (npr. Povrs()), i „polja“ (npr. „strana“). Primetimo da se promenljive unutar klase nazivaju polja.

Klasa definiše jedan specijalan tip podataka, tip koji obuhvata i metode i polja. Evo primera upotrebe klase:

```
Kvadrat k1; //kreiranje promenljive k1 tj. objekta k1 tj. instance k1
k1 = new Kvadrat(); //inicijalizacija
```

Pomoću „new“ se kreira nova instanca klase tj. objekat klase.

Dozvoljeno je sledeće kopiranje objekta u objekat:

```
Kvadrat k1; //kreiranje objekta
k1 = new Kvadrat();
Kvadrat k2; //kreiranje objekta (instance klase)
k2 = k1;
```

Dakle, može se inicijalizovati neki objekat i bez upotrebe „new“ već direktno pomoću operatora „=“.

AUTOMATSKA INICIJALIZACIJA

Može se inicijalizovati neki objekat i bez upotrebe „new“ već direktno pomoću operatora „=“.

Polje ili metoda unutar klase mogu se proglasiti (deklarisati) kao ili „private“ ili „public“. Ako se proglasi „private“ onda su dostupni samo unutar klase a ne i izvan klase. Ako se izostavi „private“ ili „public“ onda se podrazumeva da je „private“ („private“ je po default-u), ali je bolje eksplicitno napisati da li je polje ili metoda privatna (private) ili javna (public) nego da se to podrazumeva, jer je program tako manje konfuzan. Evo primera gde je metoda deklarisana kao javna a polje je deklarisano kao privatno:

```
class Kvadrat{
public double Povrs()
{return strana*strana;}
private double strana;
}
```

Svako polje ili metoda se individualno deklariše kao privatno ili javno (za razliku od C++ gde se to radi grupno)

Polja u klasi se automatski inicijalizuju sa :

0

False

Null

U zavisnosti od tipa polja.

KONVENCIJE

Postoje konvencije za za davanje imena poljima i metodama.

Sledeće su konvencije što se tiče davanja imena poljima i metodama:

Imena (tj. tzv. identifikatori) koja su „public“ preporučuje se da počinju velikim slovima, npr. : public double Povrs()..

Imena koja su „private“ preporučuje se da počinju malim slovom, npr. polje „strana“ je privatno pa počinje malim slovom:

private double strana;

Imena klasa počinju velikim slovom

Imena konstruktora su ista kao imena klasa

Svaka klasa ima konstruktor. Čak ako i ne napišete konstruktor klase, kompajler automatski kreira tzv. default-konstruktor (podrazumevani konstruktor). Evo primera podrazumevanog konstruktora (default konstruktora):

```
class Kvadrat{public Kvadrat()
{strana = 0;}
public double Povrs(){return strana*strana;}
private double strana;}
```

Kao što vidimo, podrazumevani konstruktor ne prima parametre (lista parametara je prazna). Medjutim, konstruktori se mogu „preopteretiti“, tj. mogu se napisati tzv. „nepodrazumevani“ konstruktori, i ti nepodrazumevani konstruktori primaju parametre tj. imaju listu parametara. Konstruktor je deklarisan kao „public“, a ako se izostavi ova ključna reč onda konstruktor postaje privatan, i onda se ne može koristiti izvan klase npr. u nekoj drugoj klasi,npr. glavnoj klasi.

AUTOMATSKA PROMENLJIVA

Metoda Povrs() ne prima parametre, već je automatski promenljiva tj. polje „strana“ raspoloživo u toj metodi

Evo primera upotrebe objekta:

Kvadrat k1; //kreiranje objekta

k1 = new Kvadrat(); double p = k1.Povrs();

Primetimo da metoda Povrs() ne prima parametre, već je automatski promenljiva tj. polje „strana“ raspoloživo u toj metodi i nije potrebno ga stavljati na listu parametara.

U klasu Kvadrat {.....} može se dodati sledeći nepodrazumevani konstruktor, koji koristi parametar „pocetnaStrana“:

public Kvadrat(double pocetna Strana)

{strana = pocetnaStrana}

Pri tome, redosled konstruktora u klasi nije bitan, tj. proizvoljan je.

Nepodrazumevani konstruktor se koristi da se kreira neki objekat, npr.:

Kvadrat k1; //kreiranje objekta

k1 = new Kvadrat(5.0);

Bitno je napomenuti: ako napišete konstruktor za neku klasu, onda kompajler neće automatski generisati podrazumevani konstruktor. Pa ako ste napisali nepodrazumevani konstruktor, a želite da imate i podrazumevani konstruktor, onda morate i nega da napišete sami a ne da se oslonitena kompajler.

PARCIJALNE KLASE:

Evo definicije parcijalne klase.

Klasa može da sadrži više metoda, polja i konstruktora. Neke klase mogu biti glomazne. Zato, ujeziku 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. Ovo se koristi kod WPF (Windows Presentation Foundation) aplikacija. 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 double Povrs()......
private double strana;
}
```

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

STATIČKE METODE I PODACI:

može se koristiti klučna reč „static“ da se definišu neke metode ili polja klase, onda možete da pozovete takvu „statičku“ metodu ili da pristupite takvom „statičkom“ polju bez upotrebe objekta.

U jeziku C“ svaka metoda mora biti definisana u okviru neke klase. Ali, može se koristiti klučna reč „static“ da se definišu tj. deklarišu neke metode ili neka polja klase, onda možete da pozovete takvu „statičku“ metodu ili da pristupite takvom „statičkom“ polju bez upotrebe objekta (tj. imena objekta) već upotrebom imena klase. Npr.

```
class Yyyyy
{public static double Xxxx(....){.....}
.....}
```

Ali, statička metoda nema pristup poljima te klase koja nisu „static“ već može da koristi samo polja koja su deklarisana kao „static“. Takodje, statička metoda može da pozove neku drugu statičku metodu jer pozivanje nestatične metode zsahteva kreiranje objekta.

Polja koja se deklarišu kao „static“, tzv. statička polja, su polja koja mogu da koriste svi objekti neke klase. Npr.

```
class Yyyyy{.....
public static int int1 = 0;}
```

U ovoj klasi polje „int1“ može se koristiti u bilo kojoj nestatičkoj metodi.

Napomenimo takodje da ključna reč „Const“ ispred polja može da se koristi da se neko polje deklariše kao „statičko“. Naime, i ako polje polje deklarisano kao „Const“ (skraćeno od Constant), dakle kao polje čija vrednost se neće menjati, ono i ako nije deklarisano eksplicitno da je „static“ ono je ipak „static“ ali implicitno. Ali „Const“ se koristi samo za tipove: int, double, string, enum.

STATIČKE KLASE

Svrha neke statičke klase je da služi kao skladište uslužnih metoda i polja.

Neka klasa se može deklarisati da je „statička“, i ovo podrazumeva da takva klasa ima samo statičke članove tj. elemente. Svrha neke statičke klase je da služi kao skladište uslužnih metoda i polja. Statička klase ne sme da sadrži nestatičke metode i polja. Nema smisla kreirati objekat statičke klase, i ako pokušate ompajler će vas upozoriti na grešku. Statička klasa može da ima podrazumevani konstruktor pod uslovom da je proglašen kao „static“. Bilo koji drugi tip konstruktora nije dozvoljen, i kompajler će vas upozoriti ako grešite.

Evo kako npr. izgleda statička klasa:

```
public static class Yyyy
{
public static double X1(..){.....}
public static double X2(..){.....}
......
public static int int1 = 0;
.....................}
```

ANONIMNE KLASE

Evo objašnjenja šta su to anonimne klase. Ovakve klase imaju ponekad svoju primenu.

Anonimne klase nemaju ime. Ovakve klase imaju ponekad svoju primenu. Anonimna klasa se kreira na sledeći način, npr.:

anonimniObjekt = new { Prezime = “Pera“, Telefon = 123456 };

Dakle kreirali smo objekt a ne znamo ime klase i ne koristimo ime klase? Ova anonimna klasa sadrži dva polja: „Prezime“ i „Telefon“. Kompajler automatski zaključuje koji su tipovi ovih polja na osnovu podataka koji se koriste za inicijalizaciju. Takodje, kompajler automatski generiše neko ime za tu klasu, ali vi ne znate koje je to ime.

Može se koristiti ključna reč „var“ da se deklariše „anonimniObjekt“ kao promenljiva implicitnog tipa:

var anonimniObjekt = new { Prezime = “Pera“, Telefon = 123456 };

Ključna reč „var“ obezbedjuje da kompajler kreira promenljivu istog tipa kao i izraza koji je korišćen kod inicijalizacije.

Objekti anonimne klase mogu se koristiti pomoću tačka-operatora, npr.:

Console.WriteLine(anonimniObjekt.Prezime, anonimniObjekt.Telefon);

Takodje možete da kreirate drugi objekat iz iste anonimne klase:

var anonimniObjekt2 = new { Prezime = “Mika“, Telefon = 1234567 };

Kompajler će sam zaključiti da li dva objekta anonimne klase imaju isti tip. Ako dva objekta anonimne klase imaju polja istog imena i tipa i rasporeda, onda kompajler zaključuje da ova dva objekta pripadaju istoj anonimnoj klasi.

Medjutim, anonimne klase imaju dosta ograničenja. Npr. anonimne klase mogu da sadrže samo javna polja, polja moraju biti inicijalizovana, polja ne mogu biti statička, i anonimne klase ne sadrže nijednu metodu

Vežba 5

10

METODE

Metodama u programskom jeziku C# postižemo funkcionalnost nekog dela programa, odnosno njom izvršavamo neku akciju.

Metodama u programskom jeziku C# postižemo funkcionalnost nekog dela programa, odnosno njom izvršavamo neku akciju.

Osnovna sintaksa bilo kakve metode je:

<modifikator_pristupa>_<povratna_vrednost>_<ime_metode>(<argumenti_metoda>)

{

<telo_metode>

}

Argumenti metoda i povratne vrednosti

U programskom jeziku postoje različite varijacije metoda koje se mogu pojaviti u upotrebi, pa tako imamo nekoliko različitih varijanti:

Najjdenostavnija metoda je metoda koja ne vraća nikakvu vrednost, a ne prosleđuju joj se nikakvi parametri.

Primer je:

```
private void metodaBezArgumenataNeVracaVrednost()
{
UradiNesto();
}
```

Važno je zapaziti da za ovakve metode mora biti definisana povratna vrednost na void. Nešto složenija je metoda koja vraća nekakvu povratnu vrednost:

Primer je:

```
Primer je:
private String
metodaBezArgumenataVracaStringovnuVrednost()
{
String nasaVrednost = "testVrednost";
return nasaVrednost;
}
```

METODE DRUGI DEO

Sa druge strane, moguće je imati metodu koja ne vraća nikakvu vrednost ali kao argumente prima određene vrednosti:

Sa druge strane, moguće je imati metodu koja ne vraća nikakvu vrednost ali kao argumente prima određene vrednosti:

```
private void metodaSaArgumentimaNeVracaVrednost
(String nasaVrednost, String drugaVrednost)
{
nasaVrednost += drugaVrednost;

}
```

Postoji mogućnost da nam je potrebno da metodi prosledimo argumente I da nam ta metoda vrati odgovarajuću vrednost.

U tom slučaju ćemo imati sledeće:

```
private String metodaSaArgumentimaVracaVrednost
(String nasaVrednost, String drugaVrednost)
{
String sabraniStringovi;
sabraniStringovi = nasaVrednost +
drugaVrednost;
return sabraniStringovi;
}
```

Primer upotrebe metoda možemo videti u prilikom realizacije osnovnih matematičkih operacija, gde pozivanjem metode matematičkeOperacije vršimo sabiranje brojeva 2 i 4 tako što svakoj metodi koja vrši određenu operaciju prosleđujemo kao argumente prvi i drugi broj, a kao rezultat rada metode prihvatamo vrednost.

```
public class MatematickeOperacije
{

public String matematickeOperacije()
{
int prviBroj = 2;
int drugiBroj = 4;
String zbir = "Zbir je " + sabiranje(prviBroj, drugiBroj) + ".";
String razlika = "Razlika je " + oduzimanje(prviBroj, drugiBroj) + ".";
String proizvod = "Proizvod je " + mnozenje(prviBroj, drugiBroj) + ".";; String kolicnik = "Kolicnik je " + deljenje(prviBroj,
drugiBroj) + "."; return zbir + "\n" + razlika + \n" + proizvod + +"\n" + kolicnik;
}

public int sabiranje(int prviBroj, int drugiBroj)
{
int zbir = prviBroj + drugiBroj;
return zbir;
}

public int oduzimanje(int prviBroj, int drugiBroj)
{
int razlika = prviBroj - drugiBroj;
return razlika;
```

KONSTRUKTORI

Sa druge strane, moguće je imati metodu koja ne vraća nikakvu vrednost ali kao argumente prima određene vrednosti:

Konstruktor je jedina metoda u klasi koja nema deklarisanu povratnu vrednost i ima ime isto kao i ime klase. Konstruktorskoj metodi se mogu proslediti i argumenti, ali i ne moraju.

Konstruktorske metode služe da naprave objekat klase i stavljaju ga u ispravno stanje. Ukoliko konstruktor u kodu nije definisan, CLR će izvršiti kreiranje konstruktora umesto Vas.

```
public class TestKlasa
{
public TestKlasa()
{
}
}
```

Preklapanje metoda i konstruktora

U realnom radu, veoma često će odgovarati situacija da imate vise istoimenih funkcija.

Ono što razlikuje jednu metodu od druge metode je njen potpis (signature). Potpis jedne metode je određen njenim imenom i listom argumenata. Dve metode imaju različite potpise ako nemaju identične liste parametara iako imaju isto ime.

Kako smo u prethodnom delu koristili računske operacije, možemo recimo dati i primer preklapanja u tom duhu. Recimo, potrebno nam je da izvršimo sabiranje, znamo da će nam biti potrebno da izvršimo sabiranje dva, tri ili četiri sabirka, i trebaju nam sve te mogućnosti. Logična je varijanta da se sve metode i zovu isto jer upravo i rade isto.

Prema tome, naše rešenje bi bilo preklapanje metoda prema sledećem:

```
public class MatematickeOperacije
{

public int sabiranje(int prviBroj, int drugiBroj)
{
int zbir = prviBroj + drugiBroj;
return zbir;
}

public int sabiranje(int prviBroj, int drugiBroj, int treciBroj)
{
int zbir = prviBroj + drugiBroj + treciBroj;
return zbir;
}

public int sabiranje(int prviBroj, int drugiBroj, int treciBroj, int cetvrtiBroj)
{
int zbir = prviBroj + drugiBroj + treciBroj + cetvrtiBroj;
```

KONSTRUKTORI DRUGI DEO

U realnom radu, često, biće Vam potrebno preklapanje konstruktora. Evo i jednog takvog primera:

U realnom radu, često, biće Vam potrebno preklapanje konstruktora. Evo i jednog takvog primera:

```
public class Vreme
{
//privatne promenljive clanice klase private int godina;
private int mesec; private int dan; private int sat;
private int minut; private int sekund;

public void PrikaziVreme()
{
System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}", dan,
mesec, godina, sat, minut, sekund);
}

public Vreme(System.DateTime dt)
{
godina = dt.Year;
mesec = dt.Month;
dan = dt.Day;
sat = dt.Hour;
minut = dt.Minute;
sekund = dt.Second;
}

public Vreme(int godina, int mesec, int dan, int sat, int
minut, int sekund)
{
this.godina = godina;
this.mesec = mesec; this.dan = dan; this.sat = sat;
this.minut = minut; this.sekund = sekund;
}
}

public class Test
```

KLASE I OBJEKTI U VISUAL C#

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa.

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa, onosno razlog postojanja klasa i objekata.

Konkretno, u proceduralnom programiranju, na osnovu koncepta izdovijio se niz nedostataka koji su zahtevali promene. Na primer, u proceduralnom programiranju, svaki podprogram može da pristupi svakom podatku, zatim, svaka promena u podacima može dovesti do neuspešnog izvršavanja podprograma. Još jedna bitna stvar koja je indirektno dovela do razmatranja je činjenica da je sa povećanjem veličine programa sve je teže vršiti izmene, a kod nije bilo moguće ponovno koristiti.

Uvođenjem objektno orijentisanog pristupa, rešavaju se ovi problemi, a pore toga ostvaruju se i sledeće ključne benefiti:

- U objektno orijentisanom programiranju se polazi od objekata kojima želimo da manipulišemo, a ne od logike koja je potrebna za tu manipulaciju.
- U realnom sistemu se identifikuju objekti i veze koje postoje između njih. Definišu se tri osnovna koncepta objektno orijentisanog programiranja:
- učaurenje
- nasleđivanje
- polimorfizam

Uvođenjem objektno orijentisanog pristupa, uvedeni su pojmovi klase i objekta.

KLASE I OBJEKTI

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa.

Ako u najkraćem sažmemo predavanje br. 8 u delu objašnjenja pojma klasa i objekata, i pokušamo da odgovorimo na pitanje šta su to klase i objekti, to možemo uraditi na sledeži način:

Klasa je struktura podataka koju bismo trebali posmatrati kao novi tip, i ona prestavlja generičku definiciju objekata koji imaju zajeničku strukturu i ponašanje.

Nasuprot klasi, objekat je instanca klase i on se definiše kao entitet koji je sposoban da čuva svoja stanja i koji okolini stavlja na raspolagaje skup operacija preko kojih se tim stanjima pristupa.

Prema tome, objekat karakterišu:

• identitet - omogućava razlikovanje objekata među sobom

• ponašanje - dinamički aspekt objekta, definiše se metodama koje sadrži objekat

• stanje - statički aspekt objekta, definiše se podacima objekta i njegovim vezama sa drugim objektima u sistemu.

Kreiranje klase

Kada kreiramo neku sopstvenu klasu, potrebno je da definišemo: njena svojstva (properties) ili varijable kojima će dodeljivati vrednosti1 i metode koje određuju njeno ponašanje. Naredba za kreiranje nove klase počinje ključnom reči class, zatim dolazi naziv klase, zatim deklaracija svojstava (properties) koje se definišu za tu klasu, a zatim metode koje pripadaju klasi.

```
public class mojaKlasa
{
//deklaracija i inicijalizacija svojstava ili varijabli int mojaVarijabla;

//metode
public void mojaMetoda()
{
...
}
}
```

Gornjim naredbama kreirali smo klasu mojaKlasa, i njoj pripadajuću celobrojnu varijablu mojaVarijabla kojoj će se unutar klase mojaKlasa dodjeljivati vrednosti. U nastavku, još jedan primer kreiranja klase, ovoga puta, klasa Račun.

```
public class Racun
{
private long brojRacuna;
private decimal stanje;
private decimal provizija;

public void Uplata(decimal iznos)
```

KREIRANJE OBJEKATA

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa.

U prethonom delu smo definisali kreiranje klasa i vrlo je bitno da shvatimo da je klasa referentni tip. Samim tim, jasno nam je da se deklaracijom klase ne kreira instanca klase (objekat), već referenca.

Kreiranje objekata se realizuje na drugačiji način. Instanciranje klase, odnosno kreiranje objekta vrši se iz dva koraka:

- alokacija memorije se vrši pomoću operatora new.
- inicijalizacija objekta se vrši pomoću konstruktora kojim se alocirana memorija "pretvara" u objekat i objekat inicijalizuje.

Pristup članovima objekta ostvaruje se navođenjem punog kvalifikovanog imena člana.

```
class Student
{


// deklaracija svojstava koja pripadaju klasi int Ocena;

public void Main()
{
Student Nikola = new Student();
Nikola.Ocena = 3;
}
}
```

Korišćenje ključne reči "this"

Ključna reč "this" odnosi se na trenutnu instancu neke klase (ili drugog objekta). "this" je skriveni pokazivač (pointer) na svaku nestatičku metodu neke klase. Postoje tri slučaja upotrebe ključne reči this:

1. slučaj upotrebe: U svrhu kvalifikacije instance ako se ona zove isto kao i parametar koji se prosljeđuje nekoj metodi, pa tada "this" omogućava da se odredi na koju se vrednost se misli.

```
public void NasMetod (int varijabla)
{
this.varijabla = varijabla;
}
```

2. slučaj upotrebe: U svrhu prosleđivanja trenutno aktivnog objekta kao parametra drugoj metodi.

```
public void FirstMethod(OtherClass otherObject)
{
otherObject.SecondMethod(this);
}
```

KREIRANJE OBJEKATA DRUGI DEO

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa.

Ovaj primer uspostavlja dve klase; prva - klasa koja ima metodu FirstMethod(), I druga - klasa OtherClass koja ima metodu SecondMethod().

Ako unutar prve metode želimo da pozovemo metodu koja pripada drugoj klasi, tada tu drugu metodu moramo pozvati iz te druge klase. Tu klasu ne možemo koristiti preko njenog naziva, nego moramo proslediti parametar – varijablu koja je tipa OtherClass.

3. Slučaj upotrebe: korišćenjem indeksera

Primer korišćenja ključne reči this: Ako this koristimo u svrhu određivanja čije svojstvo (property) će se koristiti:

Gornjom naredbom u tekstu labele label4 će se ispisati vrednost svojstva ImePrezimeStudenta koje pripada aktuelnoj klasi (u kojoj se nalazi ta metoda i naredba).

Gornjom naredbom u tekstu textBox1 će se ispisati vridnost svojstva Ocena koje pripada aktuelnoj klasi (u kojoj se nalazi ta metoda i naredba).

KONSTRUKTORI I DESTRUKTORI

U prethodnim vežbama obrađena je sintaksa konstruktora u okviru metoda, ali nismo zašli "ispod haube"

U prethodnim vežbama obrađena je sintaksa konstruktora u okviru metoda, ali nismo zašli "ispod haube" i nismo videli karakteristike konstruktora, njihove osnovne primene i nismo dali nikakve preporuke. Kako je sintaksa već obrađena, na nju se nećemo vraćati, ali ćemo konstruktore obraditi za potrebe razumevanja njihovog korišćenja.

Pre nego što budemo dali ilustrativni primer, bitno je da se zapamte sledeće činjenice:

Šta su konstruktori? Konstruktori su specijalne metode koje služe za inicijalizaciju objekata nakon njihovog kreiranja. Oni obezbeđuju da objekat ima dobro definisano početno stanje pre nego što se upotebi. Ako ne uspe inicijalizacija neće postojati objekat.

Za imenovanje konstruktora koristi se ime klase (metoda ima isto ime kao i klasa) iza koga slede zagrade.

Konstruktori nemaju povratnu vrednost, pa čak ni tipa void. Postoje dve vrste konstruktora i to:

- konstruktori instance (vrše inicijalizaciju objekata)
- statički konstruktori (vrše inicijalizaciju klasa)

Kada se kreira objekat .NET kompajler, ukoliko nije eksplicitno naveden konstruktor, automatski generiše podrazumevani konstruktor.

```
class Datum
{
private int godina;
private int mesec;
private int dan;
// public Datum () { ... } default konstruktor koji se automatski generiše pošto nije naveden konstruktor
}

class Test
{
static void Main()
{
Datum danas = new Datum(); //kreira se objekat i poziva konstruktor
...
}
}
```

Vrlo je bitno da zapamtite osobine podrazumevanih konstruktora zbog njegove dalje upotrebe u raznim Vašim programima (koristićete ih u svakoj klasi koju napravite). Osobine podrazumevanog konstruktora su: Podrazumevani konstruktor ne prima parametre. Podrazumevani konstruktor implicitno inicijalizuje sva nestatička polja na njihove podrazumevane vrednosti i to:

- numerička polja (int, double,decimal) na nulu
- logička polja na false
- polja referentnog tipa na null
- polja tipa zapis tako da su svi elementi zapisa inicijlizovani na njihove podrazumevane vrednosti.

Primer, kroz koji možemo da vidimo navedene činjenice je

KONSTRUKTORI I DESTRUKTORI DRUGI DEO

U prethodnim vežbama obrađena je sintaksa konstruktora u okviru metoda, ali nismo zašli "ispod haube"

Modifikator pristupa je public.

Konstruktor može primiti jedan ili više parametara koji se koriste za inicijalizaciju polja. Ako se u klasi deklariše bar jedan konstruktor, kompajler neće generisati podrazumevani konstruktor.

```
public class Datum
{
public int godina, mesec, dan;
public Datum(int g, int m, int d)
{
godina = g;
mesec = m;
dan = d;
}
}
class Test
{
static void Main()
{
Datum danas = new Datum(2011,09,21);
}
}
```

Sva polja koja nisu inicijalizovana u korisnički definisanom konstruktoru zadržavaju svoju podrazumevanu inicijalizaciju.

```
public class Datum
{
public int godina, mesec, dan;


public Datum(int g, int m, int d)
{
godina = g; mesec = m; dan = d;
}
}
public class Primer
```

Za jednu klasu može se definisati više konstruktora.

NAPOMENA: lista parametara svakog od njih mora biti jedinstvena, ili po broju ili po tipu parametara (odnosno moraju imati različite potpise)

Na prethodnoj vežbi, u delu metoda, objašnjeno je preklapanje, tako da se na te primere nećemo ponovo vraćati, ali ćemo samo napomenuti da se može desiti da preklopljeni konstruktori sadrže isti kod. Inicijalizatorska lista omogućava da jedan konstruktor poziva drugi koji je deklarisan unutar iste klase čime se omogućava da se konstruktor implementira pozivanjem preklopljenog konstruktora. Sintaksa za ove navode je sledeća:

Prilikom pravljenja konstruktora navodi se : iza kojih sledi ključna reč this, a u zagradi su navedeni parametri.

NAPOMENA: Inicijalizatorske liste se mogu koristiti samo kod konstruktora!

```
public class Datum
{
public int godina, mesec, dan;
public Datum()
: this(2010, 3, 30)
{
}
public Datum(int g, int m, int d)
```

KONSTRUKTORI I DESTRUKTORI TREĆI DEO

Destruktori su metode koje su u potpunosti suprotne konstruktorima.

Destruktori su metode koje su u potpunosti suprotne konstruktorima. Služe da inicirane objekte unište, i to im je osnovna namena. U velikoj meri desktruktore nećete koristiti jer programski jezik C# radi pod VM koja obezbeđuje garbage collection, te samim tim vrši destrukciju objekata, ali nekada ćete imati potrebu da eksplicitno uništite objekat. U tom slučaju ćete upotrebiti destructor. Ukoliko nemate adekvatni konstruktor u klasi, prilikom aktiviranja destruktora, destruktor će uništiti instancirani objekat podrazumevanog konstruktora (setite se priče o podrazumevanim konstruktorima).

Osnovne osobine destruktora su sledeće:

- Destruktori se ne mogu definisati na strukturama, koriste sa isključivo sa klasama.
- Klasa može imati samo jedan destruktor.
- Destruktor ne može se nasljeđuje ili preopterećuje.
- Destruktor se ne može pozvati. On se automatski poziva.
- Destruktor ne koristi modifikatore ni parametre. Primer korišćenja destruktora je sledeći:

Kod:

```
class Auto
{
~Auto()  // destructor klase Auto
{
// naredbe čišćenja
}
}
```

Ako idemo malo dalje u razmatranje, možemo postaviti i drugačiji primer:

```
class PrvaKlasa
{
~PrvaKlasa()
{
System.Diagnostics.Trace.WriteLine("Desktruktor prve klase je pozvan.");
}

class DrugaKlasa : PrvaKlasa
{
~DrugaKlasa()
{
System.Diagnostics.Trace.WriteLine("Desktruktor druge klase je pozvan.");
}

class TrecaKlasa : DrugaKlasa
{
~TrecaKlasa()
{
```

Startovanjem ovog malog primera dobijamo sledeći izlaz:

Desktruktor trece klase je pozvan.

Desktruktor druge klase je pozvan.

Desktruktor prve klase je pozvan.

KONSTRUKTORI I DESTRUKTORI ČETVRTI DEO

Destruktori su metode koje su u potpunosti suprotne konstruktorima.

Iz njega jasno vidimo da destruktore ne pozivamo, ne prosleđujemo nikakve vrednosti, ali da oni vrše realizaciju bez obzira što mi to nismo uradili.

Šta u stvari destruktori rade prilikom realizacije?

Destruktor implicitno poziva metodu Finalize na osnovu bazne klase objekta. Ako pogledamo naš primer klase Auto, upotreba destruktora će se implicitno prevesti u sledeći kod:

```
protected override void Finalize()
{
try
{

}
// naredbe čišćenja
finally
{
base.Finalize();
}
```

Ovaj kod će biti jasniji posle završetka vežbe izuzeci.

MODIFIKATORI PRISTUPA I ENKAPSULACIJA

Enkapsulacija objekata je jedan od osnovnih koncepata objektno orijentisanog programiranja I predstavlja dodatnu apstrakciju kojom se "sakrivaju" detalji implementacije objekta.

NAPOMENA: Za realizaciju ovog dela vežbe ponovite modifikatore pristupa obrađene u vežbi 6. Ako niste usvojili informacije o modifikatorima u jeziku C#, uradite to sad.

Enkapsulacija objekata je jedan od osnovnih koncepata objektno orijentisanog programiranja I predstavlja dodatnu apstrakciju kojom se "sakrivaju" detalji implementacije objekta.

Postoje dva bitna aspekta enkapsulacije:

- objedinjavanje podataka i funkcija u jedinstven entitet (klasa)
- kontrola mogućnosti pristupa članovima entiteta (modifikatori pristupa)

NAPOMENA: Posle ove vežbe Vam mora ostati ova činjenica urezana duboko u pamćenje: direktan pristup podacima je potpuno nepotreban i nepoželjan!.

Objedinjavanje podataka i funkcija u jedinstven entitet se ostvaruje se pomoću klasa, a određuje se granicom entiteta.

Kontrola mogućnosti pristupa članovima entiteta ostvaruje se navođenjem modifikatora pristupa.

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

```
public  void  Uplata(decimal   iznos)
{
stanje  = stanje  + (iznos*(1-provizija));
}

public  void  Isplata(decimal   iznos){...}
public  void  Prikazi(){...}

public  long  brojRacuna;  public  decimal  stanje;  public  decimal  provizija;
}
...
Racun  racun1  = new Racun();
racun1.Uplata(10000);
racun1.stanje   = 150000;
```

Na osnovu principa objektno orijentisanog programiranja I dobre programerske prakse, preporuka je da podaci objekta treba da se nalaze u privatnom delu. Naravno, ukoliko to ne želite da uradite ne morate, ali tako sami sebi usložnjavate održavanje, a samim tim I mogućnost nastanka bug-ova. Od mesta deklaracije određenog člana zavisi i vrsta modifikatora pristupa koji se može koristiti.

MODIFIKATORI PRISTUPA I ENKAPSULACIJA DRUGI DEO

Enkapsulacija objekata je jedan od osnovnih koncepata objektno orijentisanog programiranja I predstavlja dodatnu apstrakciju kojom se "sakrivaju" detalji implementacije objekta.

Ukoliko nije naveden, određuje se podrazumevani modifikator:

- za imenovani prostor nije dozvoljeno navođenje modifikatora, jer se podrazumeva da su javni
- tipovi koji se deklarišu unutar imenovanog prostora mogu biti public ili internal. Podrazumevani modifikator je internal.
- članovi klase mogu da imaju bilo koji od navedenih pet modifikatora. Podrazumevani modifikator je private.
- članovi zapisa mogu biti public, internal ili private. Podrazumevani modifikator je private.
- članovi interfejsa ne mogu imati modifikatore pristupa. Implicitno su public.
- članovi nabrajanja ne mogu imati modifikatore pristupa. Implicitno su public.

Domen iz koga se može pristupiti određenom tipu nikad ne sme biti uži od domena iz koga se može pristupiti članu koji je deklarisan unutar tog tipa.

```
namespace primer
{
internal class Racun
{
public decimal stanje; //greška
```

Postoje dva razloga:

• omogućavanje kontrole korišćenja - Objekat se može koristiti isključivo preko javnih metoda.

• smanjenje uticaja promena - Ukoliko su detalji implementacije objekta privatni mogu se promeniti, a da te promene ne utiču direktno na korisničke objekte (koje jedino mogu da pristupe javnim metodama).

```
public class Racun {
public decimal DajStanje()
{
return stanje;
}

public void Prikazi(){...}
private decimal stanje;
}


public class Klijent {

public void Prikazi()
{
Console.WriteLine( "Klijent ...");
Console.WriteLine( string.Format("Stanje: {0}" racun.DajStanje() );
}
```

Ovim promena ne utiče na korisnika klase.

```
public class Racun {
public decimal DajStanje()
```

SVOJSTVA GET I SET

Svojstvo daje (postavlja) informacije o objektu kome pripada. Svojstvo je jedan metod ili par metoda (što se klijentskog koda tiče), ali se ponaša kao polje.

Svojstvo daje (postavlja) informacije o objektu kome pripada. Svojstvo je jedan metod ili par metoda (što se klijentskog koda tiče), ali se ponaša kao polje. Na primer, kada pomerate neku kontrolu svojstvo

Locationdaje vam informaciju o poziciji gde se ta kontrola nalazi.Tekstualni sadržaj je opisan svojstvom Text, itd.Svojstvo može da da informaciju o objektu i/ili da postavlja nekuvrednost za objekat. Drugim rečima svojstva mogu da budu read-write, ali i samo jedno od ta dva, bilo koje.Svojstva su neobična i po tome što su preuzeta iz programskog jezikaVB, a ne iz C++ ili Jave. Svojstva se ponašaju kao atributi, ali zarazliku od njih svojstva ne predstavljaju samo vrednost nekepromenljive, već su po prirodi funkcije kao i metode.Na primer, upotreba jednog svojstva definisanog u klasi Form

Prilikom izvršenja ovog koda prozor će dobiti visinu 500, a korisnik ćemoći da vidi ovu promenu na ekranu. Sintaksno, kôd je isti kao da smoizvršili podešavanje nekog polja, ali suštinska razlika je u tome što jepozvan metod koji vrši promenu visine.

Sintaksa za definisanje jednog svojstva najbolje se vidi iz sledećegprimera:

```
public string mojeSvojstvo
{
get
{
return "Vracena stringovna vrednost!";
}
set
{
string x = "Podesavanje stringovne vrednosti";
}
}
```

Svojstvo se definiše u okviru neke klase kojom se opisuje objekat. Ugornjem primeru svojstvo je tipa

string

i nosi naziv **myProperty**

. Drugimrečima, vrši se čitanje i podešavanje vredosti nekog stringa uz sveostale akcije koje su skrivene iza metoda koje definišu svojstvo.Ukoliko svojstvo ima metode

get i

set

za njega kažemo da je **read-write**

SVOJSTVA GET I SET DRUGI DEO

Svojstvo daje (postavlja) informacije o objektu kome pripada. Svojstvo je jedan metod ili par metoda (što se klijentskog koda tiče), ali se ponaša kao polje.

Ukoliko ima samo metodu get

kažemo da je r**ead-only**

, a ako imasamo metodu set

, kažemo da je **write-only**

.

Programski jezik C# ne dozvoljava podešavanje različitih modifikatorapristupa (

private, public ili

protected

) pristupnim metodama **get**

i set.

Primer: U klasi

Property

dodato je svojstvo

pokazana je upotreba tog svojstva.

Name

, a u klasi

Class1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
public class Svojstvo
{
public string Ime
{
get
{
return ime;
}
set
{
ime = value;
}
}
private string ime;
}

class TestKlasa
{
```

KREIRANJE SVOJSTVA IZ IDE

Svojstvo daje (postavlja) informacije o objektu kome pripada. Svojstvo je jedan metod ili par metoda (što se klijentskog koda tiče), ali se ponaša kao polje.

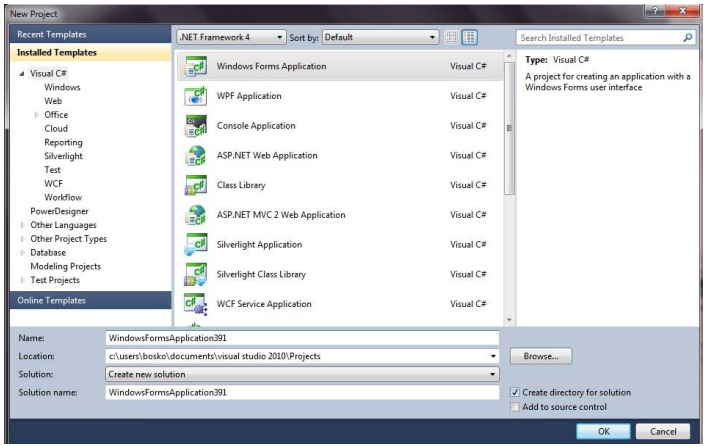
Svojstva se mogu kreirati i preko IDE preko ugrađenog alata pomoću koga možete dodati proizvoljno svojstvo nekoj klasi.

Primer kreiranja klase i svojstva kroz alat IDE:

1. Kreirajte projekat: File->New->Project i na New Project dijalogu u Project

Types odaberite Visual C#.

2. U Templates listi oaberite Windows Application.



Slika-1: Novi projekat

3. U Name tekst polju zamenite ime.

4. U kombo polju Location prihvatite predlog ili otkucajte sami lokaciju.

5. OK

6. Pritisnite F5 da biste testirali program.

7. Zatvorite ga i vraćate se u VS.Ovo je standardni deo za pravljenje okvira jedne windows aplikacije ukome dalje radimo.

8. Otvorite pogled na klase u vašoj aplikaciji Class View.

9. Desnim tasterom miša kliknite na klasu Form1 u koju dodajemo novi metod.

10. Pošto se otvara pomoćni meni, odaberite stavku Add

11. A zatim odaberite stavku Property.

12. Otvara se novi dijalog u kome podešavate parametre koji se tiču svojstva koje dodajete.

ZADACI ZA SAMOSTALNI RAD

U programskom jeziku postoji više naredbi koje mogu da uslove različite tokove i izvrše grananja.

Proradite sve varijante upotrebe metoda, prosledjujte argumente, prihvatajte vrednosti i pokušajte da elaborirate sve veze jer je to od ključnog značaja za nastavak i Vaš dalji uspešan rad.

Napravite po dve klase i obavezno kreirajte konstruktore i destruktore. Provežbajte uništavanje objekata korišćenjem destruktora.

Definišite klasu koja u sebi ima polja, metode i lokalne promenljive. Definišite da sva polja i promenljive ne smeju biti vidljive van klase. Pokušajte pristup iz različitih varijanti (iz klase, iz paketa, itd.) i menjajte modifikatore.

Provežbajte upotrebu modifikatora, posebno provežbajte upotrebu protected, internal i protected internal modifikatora.

Zaključak

ZAKLJUČAK

Zaključak

C# je potpuno objektno orijentisan jezik, gde se koriste pojmovi klasa, i objekat. Radni okvir .NET sadrži hiljade i hiljade već kreiranih i testiranih klase spremne za upotrebu, a neke smo već spomenuli, npr. klase Console i Exception. Klase pružaju pogodan šablon i mehanizam za modelovanje raznih entiteta, fizičkih entiteta kao što je Mačka, Stan, itd., ili apstraktnih entiteta kao što je npr. transakcija. „Enkapsulacija“ tj. „skrivanje informacija“, *data hidi*ng, je važan aspekt definisanja klase. Npr. kada pozivate metodu ConsoleWriteLine, ne interesuje vas detalji te metode i te klase, već jednostavno želite da upotrebite tu metodu.

Nizovi, arrays, su veoma koristan tip varijabli, ali imaju i nedostatke. Radni okvir .NET nudi nekoliko klasa koje okuplaju slične elemente, ali na druge posebne načine u odnosu na nizove. To su klase kolekcija, i one se nalaze u prostoru *SystemCollections*. Klase kolekcija okuplaju objekte.

U praksi, često se traži da se prikažu objekti neke klase u „kontroli“ *ListBox*, gde je *ListBox* već pomenuta ranije kao jedan od najćešće korišćenih predefinisanih objekata u Toolbox-u u Visual C#. Tada smo koristili ListBox da prikažemo listu elemenata tipa „string“. Ali, u praksi se obično zahteva da se prikazuje lista objekata a ne lista „string“-ova. Svaki objekat u Visual C# može da koristi metoduToString(). A „kontrola“ ListBox koristi ovu metodu da prikaže objekte.