

Osnove programskog jezika C#.NET

Za jezik se može reći da predstavlja jednu od najvećih tekovina ljudske civilizacije. Prirodni jezici koji su u upotrebi u svakodnevnoj komunikaciji ljudi, nastali su iz potrebe za međusobnim sporazumevanjem u obavljanju različitih aktivnosti. Bogatstvo rečnika jednog prirodnog jezika zavisi od starosti i kulture naroda koji ga koristi, ali isto tako i od delatnosti kojima se taj narod bavi.

Pored prirodnih postoje i veštački jezici koji su nastali kao unapred definisan skup pravila za sporazumevanje u određenoj oblasti. Najbolji primer za to su matematika i računarske nauke gde su u širokoj upotrebi skupovi veštačkih entiteta poznati po imenu formalni jezici. Formalni jezici se, kao i prirodni, odlikuju skupovima karaktera, semantičkim i gramatičkim pravilima. Posebnu grupu veštačkih jezika čine programski jezici. Nastaju sa razvojem računara iz potrebe da se pojednostavi pisanje programa kojima se upravlja tokom obrade podataka na njima. Zbog toga se može reći da su programski jezici sredstvo komunikacije između ljudi i računara. Do danas je razvijen veliki broj programskih jezika i njihov broj prelazi 4000. U isto vreme broj aktivnih prirodnih (ljudskih) jezika se kreće oko brojke od pet hiljada.

Programski jezici se mogu klasifikovati u više kategorija po više različitih kriterijuma. Jedan od često korišćenih kriterijuma je i zavisnost od mašine na kojoj se izvršava. Tako postoje: **mašinski zavisni** i **mašinski nezavisni jezici**. U mašinski zavisne jezike svrstavaju se jezici koji su u direktnoj zavisnosti sa procesorom na kome se program izvršava. To su pre svega mašinski jezik i mašini orijentisani assemblerski i makro assemblerski jezici. Oni čine grupu jezika niskog nivoa. Grupu mašinski nezavisnih jezika čine programski jezici visokog nivoa ili proceduri orijentisani programski jezici kao i programski jezici veoma visokog nivoa, aplikativni ili problemu orijentisani programski jezici.

Sa razvojem programskih jezika uporedo je išao i razvoj programskih paradigmi i stilova programiranja pa tako imamo sledeće vrste programiranja: modularno, strukturno, rekurzivno, objektno, logičko, paralelno, simboličko, itd.

Najznačajnije programske paradigme među programskim jezicima višeg nivoa su **proceduralna** i **objektno orijentisana**. Tradicionalno proceduralno programiranje se sastoji od niza funkcija i njihovih poziva dok se objektno orijentisano bazira na **objektima** i njihovoj međusobnoj komunikaciji. Kako se svaki realan sistem može opisati kao skup objekata koji se nalaze u nekoj interakciji sasvim je prirodno da objektno orijentisana paradigma bolje i skladnije oslikava realni svet i omogućava logičnije projektovanje i realizaciju programa. Normalno, jedan problem se može rešiti i pomoću tradicionalnog proceduralnog programiranja i pomoću objektnog programiranja. Razlike su samo u naporu koji se ulaže u razvoj a kasnije i održavanje programa. Uopšteno rečeno objektno orijentisana paradigma donosi novine koje je čine osnovom velikog broja savremenih programskih jezika. Takav jezik je i **C#.NET** (nadalje u tekstu će biti pisano samo C#) koji će biti predmet ovog kursa. Još jedna značajna karakteristika programskog jezika C# je i mogućnost razvijanja različitih vrsta aplikacija: konzolnih, Windows, Web. Zahvaljujući tome, sintaksom koja je nastala na bazi sintaksi široko rasprostranjenih jezika C++ i Java, kao i Microsoft-ovom razvojnom okruženju izuzetno jednostavnom za korišćenje, C# danas spada u kategoriju najpopularnijih programskih jezika.

Napomena

C# je programski jezik koji nam omogućuje pisanje kako konzolnih tako i Windows aplikacija, pa čak i aplikacija za Web. U prvom delu koji se odnosi na osnove programskog jezika C# primeri će se odnositi na delove koda neke konzolne aplikacije dok će se u drugom delu detaljnije objašnjavati razvoj Windows aplikacija te će i primeri biti sa odgovarajućim grafičkim korisničkim interfejsom.

Svaki programski jezik ima elemente koji ga karakterišu i to su:

- azbuka,
- rezervisane reči,
- konstante,
- promenljive.

Azbuku programskog jezika C# čine mala i velika slova engleskog alfabeta, cifre, specijalni znaci (npr. za aritmetičke operacije, relacije, itd.) i razdelnici (tačka, dvotačka, tačka zarez, itd.).

Promenljive

Programski jezik C# spada u grupu tipiziranih jezika. Svaka promenljiva mora da se deklarise pre upotrebe. Deklaracijom promenljive definiše se:

- naziv,
- tip,
- opciono njena početna vrednost i
- vidljivost tj. prava pristupa (ako je promenljiva članica neke klase)

Sintaksa je sledeća:

vidljivost tip naziv = početna vrednost ;

Na primer:

```
public string Ime ;  
private int BrojUcenika = 1 ;  
int x, z ;  
double alfa, Plata, UDALJENOST ;
```

Programski jezik C# kao i većina drugih jezika (na primer C, C++) **pravi razliku između malih i velikih slova** (case sensitive) te su promenljive alfa i Alfa dve različite promenljive.

Napomena:

1. Treba uočiti da se u okviru jedne linije koda može deklarirati jedna ili pak više promenljivih.
2. Može se uočiti i da se linije završavaju simbolom tačka zarez (;) (slično kao i kod programskih jezika C i C++).
3. **Sintaksa programskog jezika C# se u velikoj meri poklapa sa sintaksom programskih jezika C i C++**
!!! (Ovo je bio i jedan od razloga zašto se kao implementacioni jezik u okviru ovog kursa koristi upravo C# a ne neki drugi objektno orijentisani jezik kao na primer Java.)

Tipovi podataka

Svaka promenljiva kao karakteristiku ima:

- ime,
- tip.

Ime promenljive je niz alfanumeričkih karaktera pri čemu prvi znak ne može biti cifra. Specijalni znaci ne mogu biti deo imena osim znaka _ (donja crta) koji može biti i prvi znak. Mogu se koristiti i mala i velika slova pri čemu se pravi razlika između velikih i malih slova tj promenljive čija su imena alfa i AlFa su dve različite promenljive.

Tip određuje: skup vrednosti, način predstavljanja u memoriji računara i skup operatora koji se mogu primenjivati.

Primer: Jedan od celobrojnih tipova je tip **byte**. Skup vrednosti koje može imati jedan podatak tipa byte je skup celih brojeva u opsegu od -128 do 127. Za memorisanje jednog podatka tipa byte potrebno je 8 bita tj. jedan bajt. Za memorisanje jednog podatka tipa int potrebno je 4 bajta tj. 32 bita. Operacije koje su dozvoljene nad podacima tipa int su aritmetičke operacije (sabiranje, oduzimanje, deljenje, množenje).

Napomena: 8 bita čine jedan bajt.

Za sve tipove je potrebno poznavati broj bajtova koji se koristi predstavljanje kao i način predstavljanja jer njihovo nepoznavanje može dovesti do nepredviđenih grešaka u programima.

Na primer: ako imamo celobrojnu brojačku promenljivu neophodno je da znamo koji je najveći ceo broj koji se može predstaviti promenljivom da bi se izbegle situacije do kojih se dolazi kada se dosegne maksimalna vrednost. Postavlja se pitanje šta se dešava kada se na maksimalnu vrednost doda 1 ili neka veća celobrojna konstanta? Da li će program da se zaustavi ili će nastaviti da radi? Ako nastavlja sa radom koju vrednost će tada imati brojačka promenljiva? Da li je to izvor grešaka?

Osnovni tipovi podataka

Tipovi podataka se mogu podeliti u 3 grupe:

- numeričke,
- znakovne,
- logičke.

Numeričkim tipovima podataka se predstavljaju celi i realni brojevi. Za predstavljanje celih brojeva postoji više tipova koji su prikazani u tabeli 1.

Tabela 1: Celobrojni tipovi

tip	Opseg vrednosti	Zauzeće memorije u bajtovima
sbyte	-128 do 127	1
byte	0 do 255	1
short	-32768 do 32767	2
ushort	0 do 65535	2
int	-2 147 483 648 do 2 147 483 647	4
uint	0 do 4 294 967 295	4
long	-9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	8
ulong	0 do 18 446 744 073 709 551 615	8

U tabeli 1 se mogu uočiti tipovi koji počinju slovom u što znači da su neoznačeni (unsigned) tj. da opseg vrednosti čine samo pozitivni celi brojevi.

Napomena: 1 bajt se sastoji od 8 bitova tj 8 pozicija.

Realni brojevi se su predstavljeni tipovima:

- float,
- double i
- decimal.

Ovi tipovi se razlikuju po opsegu vrednosti i broju značajnih cifara tj. decimala kao što je to prikazano u tabeli 2.

Tabela 2:

tip	Opseg vrednosti	preciznost	Zauzeće memorije u bajtovima
float	$\pm 1.5 \times 10^{-45}$ do $\pm 3.4 \times 10^{38}$	7 cifara	4
double	$\pm 5 \times 10^{-324}$ do $\pm 1.7 \times 10^{308}$	15-16	8
decomal	$\pm 1 \times 10^{-28}$ do $\pm 7.9 \times 10^{28}$	28-29	16

Napomena: Koji od postojećih tipova za predstavljanje će biti korišćen zavisi od konkretne namene tj. potrebe za opsegom vrednosti, potrebe za preciznošću, veličine dostupne memorije. Pravilan izbor tipova podatka značajno utiče na kvalitet softvera.

Znakovni tip je char. Dozvoljene vrednosti su UNICODE znak tj. ceo broj u opsegu od 0 do 65536 što znači da se svaki znak kodira sa dva bajta. Konstante tipa char se pišu navođenjem znaka između apostrofa (npr 'a', 'A', 's')

Logički tip je bool i ima samo dve vrednosti: **true** (logička istina tj. tačno) i **false** (logička neistina tj. netačno). Jedan bajt se koristi za podatak tipa bool.

Operatori

Tipom podatka je između ostalog određen i skup operatora koji mogu da se koriste nad podacima datog tipa. Operatori se mogu klasifikovati po više kriterijuma. Najčešće se koriste klasifikacija po broju operanada i klasifikacija u odnosu na vrstu izraza u kojima se koriste. Po prvoj klasifikaciji operatori mogu biti:

- unarni,
- binarni,
- ternarni.

Unarni operatori su operatori koji imaju samo jedan operand. Takav je na primer operator negacije. Binarni operatori se primenjuju nad dva operanda i oni su najčešći. Primer binarnih operatora su operatori za sabiranje, oduzimanje, množenje, itd. Ternarni operator ima tri operanda i u programskom jeziku C# postoji samo jedan takav operator (taj operator je operator „?:” koji će naknadno biti opisan).

Po drugoj klasifikaciji operatori se mogu podeliti na:

- aritmetičke (numeričke),
- logičke,
- relacijske i
- operatore za rad sa tekstualnim podacima .

Sintaksno ispravna kombinacija operanada i operatora predstavlja jedan izraz. Izraz (tj tačnije rečeno vrednost izraza) takođe ima svoj tip koji može da bude jedan od standardnih tipova (celobrojni, realni, logički, ...).

Primer: Izrazi su

a+beta

alfa – 3 * pom

Ako se u jednom izrazu pojavi više operatora postavlja se pitanje kojim redosledom će oni biti primenjivani tj izvršavani. Svaki operator ima svoj prioritet. Redosled primene operatora zavisi od prioriteta operatora koji se sreću u jednom izrazu. Ako imamo više operatora istog prioriteta onda se u nekim slučajevima oni primenjuju s leva u desno tj onako kako se pojavljuju u izrazu (na primer operatori sabiranja, oduzimanja, itd) dok se u ostalim slučajevima operatori primenjuju s desna u levo (na primer operator dodele). Iz navedenog se može zaključiti da pored poznavanja načina funkcionisanja operatora treba znati i prioritet operatora i redosled izvršavanja ako se javi više njih uzastopno.

Ako eksplicitno želimo odgovarajući redosled izvršavanja operatora onda koristimo zagrade. U opštem slučaju se najpre primenjuju oni operatori koji se nalaze u zagradama. Ako imamo slučaj ugnježđenih zagrada onda se najpre primenjuju najdublje zagrade.

Preporuka: Ako niste sigurni u prioritet operatora koristite zagrade. Na taj način će redosled izvršavanja operatora biti onakav kako želite a vreme izvršavanja neće biti promenjeno.

Aritmetički operatori su:

- + (sabiranje)
- - (oduzimanje)
- * (množenje)
- / (deljenje)
- % (ostatak celobrojnog deljenja - modulo)

Logički operatori su:

- ! - negacija
- || - logičko "ili" (OR)
- && - logičko "i" (AND)

Relacijski operatori su:

- == - ekvivalencija tj jednakost
- != - neekvivalencija tj različito
- < - manje
- > - veće
- <= - manje ili jednako
- >= - veće ili jednako

Operatori za rad sa bitovima su:

- ~ - negacija na nivou bita
- & - I na nivou bita
- | - Ili na nivou bita
- ^ - ekskluzivno ili na nivou bita
- << - pomeranje u levo
- >> - pomeranje u desno

Tekstualni operator je konkatencija u oznaci + i predstavlja nadovezivanje drugog operanda na prvi operand.

Pored prethodno navedenih operatora koji se najčešće koriste postoje i drugi operatori kao što su operatori dodele, operatori za inkrementiranje, operatori za dekrementiranje, ternarni operator itd.

Osnovni operator dodele je =. U opštem slučaju operator dodele se koristi na sledeći način:

promenljiva = izraz

Najpre se izračunava izraz a zatim se izračunata vrednost izraza dodeljuje promenljivoj pri čemu tipovi izraza i promenljive moraju da se slože.

Pored osnovnog operatora dodele postoje i složeni oblici operatora dodele koji imaju opšti oblik

operator = gde je *operator* bilo koji aritmetički operator. Na primer

```
a += b;  
je isto što i  
a = a + b;
```

Operatori inkrementiranja u oznaci ++ i dekrementiranja u oznaci -- mogu biti prefiksni i postfiksni. Operator inkrementiranja povećava vrednost operanda za jedan a operator dekrementiranja smanjuje vrednost operanda za jedan i to bez obzira da li su prefiksni ili postfiksni. Razlika se uočava jedino kada su ovi operatori deo nekog složenijeg izraza. U tom slučaju se operatori inkrementiranja i dekrementiranja ako su u prefiksnom obliku izvršavaju pre računanja vrednosti izraza za razliku od slučaja kada se javljaju u postfiksnom obliku kada se izvršavaju tek nakon izračunatog izraza. Jednostavno rečeno ako se u nekom izrazu operatori inkrementiranja i dekrementiranja javljaju u prefiksnom obliku vrednost izraza se izračunava sa novim vrednostima operanada nad kojim se primenjuju ovi operatori odnosno u slučaju postfiksnih operatora izraz se računa sa starim vrednostima operanada. I u jednom i u drugom slučaju se što se tiče samih operanada njihova vrednost poveća za jedan (za inkrement) odnosno smanji za jedna (za dekrement). Ilustrujmo to sledećim primerom koji predstavlja deo nekog koda.

Primer

```
int i = 5;  
int y = 10;  
int x = 0;  
  
i++; // vrednost i se povećava za jedan tj i ima vrednost 6;  
++i; // vrednost i se smanjuje za jedan tj i ima vrednost 7;  
i--; // vrednost i se smanjuje za jedan tj i ima vrednost 6;  
--i; // vrednost i se smanjuje za jedan tj i ima vrednost 5;  
  
x = y + ++i; // x=16 jer se vrednost i poveća za jedan pa se sabere sa y  
x = y + i++; // x=15 jer se vrednost i sabere sa y i dodeli x a zatim se i poveća za jedan  
x = y + --i; // x=14 jer se vrednost i smanji za jedan pa se sabere sa y  
x = y + i--; // x=15 jer se vrednost i sabere sa y i dodeli x a zatim se i smanji za jedan
```

Napomena: Česte greške

Pri korišćenju nekih operatora veoma često dolazi do grešaka koje se teško otkrivaju a za posledicu imaju nepredvidivo ponašanje programa.

Na primer česta greška je korišćenje operatora = umesto operatora ==. Kompajler neće prijaviti grešku ako napišete sledeću liniju koda:

```
if (x = y)
```

a zapravo ste želeli da ispitajte da li je x jednako sa y tj.

```
if (x == y)
```

Slične greške se prave kod operatora

- `&` i `&&`
- `|` i `||`

te je neophodno biti posebno obazriv kod korišćenja operatora.

Često greške nastaju i pri korišćenju operatora inkrementa (`++`) i dekrementa (`--`) u prefiksnom odnosno u postfiksnom obliku tj. kao `++x` ili pak kao `x++`.

Napomena

Polaznici kursa koji su familijarni sa programskim jezicima C i C++ mogu da uoče da se po pitanju operatora programski jezik C# u principu ne razlikuje od ovih programskih jezika!!!

Ovu konstataciju ćemo moći da ponovimo posle mnogih delova lekcija koje predstoje što navodi na zaključak da poznavaoци programskih jezika C i C++ mogu veoma brzo da ovladaju programskim jezikom C#.

Prioritet operatora

Svi operatori imaju svoj prioritet. Kompajler izvršava neki izraz u kome postoji više operatora u redosledu koji se određuje na osnovu prioriteta operatora i mesta njihovog pojavljivanja u izrazu. Neki operatori imaju isti prioritet tj. svi operatori se mogu svrstati u kategorije. Prioritet ovih kategorija je sledeći (od najvišeg ka najnižem):

- osnovni operatori (pristup polju, poziv metode, pristup indeksu, postinkrementiranje, postekrementiranje, new, typeof, sizeof,)
- unarni operatori (i predekrementiranje i preinkrementiranje)
- aritmetički (množenje, deljenje, ostatak pri deljenju)
- sabiranje i oduzimanje
- pomeranje na nivou bita (pomeranje u levo i pomeranje u desno)
- relacioni operatori
- jednakost (jednako i različito)
- I na nivou bita
- ekskluzivno ILI na nivou bita
- ILI na nivou bita
- logičko I
- logičko ILI
- ternarni operator
- operatori dodeljivanja

Napomena

Ako niste sigurni u prioritet operatora tj. u redosled izvršavanja operatora u složenim izrazima koristite zagrade. Po pravilu najpre se izvršavaju zagrade i to ako imamo zagradu u okviru neke druge zagrade onda se najpre izvršava izraz u dubljoj zagradi pa tek onda izraz u spoljašnjoj zagradi.

Na primer, izraz **(a+(b%(c+d))** će se izvršiti na sledeći način:

- najpre će se sabrati **c** i **d**,
- zatim će se izvršiti operator **%**
- i tek na kraju će se na **a** dodati vrednost dobijena primenom operatora **%**.

Postavlja se pitanje da li će se efikasnost koda smanjiti upotrebom zagrada?

Odgovor je NE!!! Kod će biti poednako efikasan. Jedina razlika će biti u veličini koda ali par bajtova više nikada ne mogu da budu alternativa obezbeđivanju od eventualne pojave greške usled nepoznavanja prioriteta operatora!!!

Upravljačke strukture

Podrazumevano izvršavanje jednog programa je linija po linija u redosledu kako su one napisane u datoteci sa kodom tj od vrha ka dnu. Veoma često se nameće potreba da se linije koda izvršavaju u drugačijem redosledu ili pak da se neke linije koda izvršavaju više puta. Ovakav način izvršavanja se realizuje pomoću odgovarajućih upravljačkih struktura koje predstavljaju predmet ove lekcije. U savremenim programskim jezicima se, po mogućnostima koje nude, skupovi upravljačkih funkcija veoma malo međusobom razlikuju. Veoma često su čak i sintakse slične ili se pak poklapaju. Poznavanje sintakse upravljačkih struktura predstavlja osnovu u proučavanju jednog programskog jezika.

Blok

Blok se u C# definiše kao niz linija koda obuhvaćenih parom velikih zagrada tj

```
{  
    linija koda 1  
    ...  
    linija koda n  
}
```

Završetak linije koda

Simbol za završetak linije koda u C# je **tačka zarez** tj ;

Vrste upravljačkih struktura

U teoriji programiranja definisana su tri osnovna tipa programskih struktura:

- sekvenca,
- selekcija (ili grananje),
- iteracija (ili petlja).

Bilo koji algoritamski rešiv problem moguće je predstaviti korišćenjem osnovnih programskih struktura!!!

Znači programski jezik treba da obezbedi podršku za gore navedene programske strukture i onda može da se iskoristi za rešavanje bilo kog problema. Ako postoji više varijanti za pjedine strukture to će samo omogućiti jednostavnije rešavanje problema.

Sekvenca

Sekvenca je upravljačka struktura kod koje se linije koda koje je čine izvršavaju redom jedna za drugom po redosledu u kome se nalaze u bloku.

U programskom jeziku C# sekvenca se implementira korišćenjem bloka u kome se nalazi jedna ili više linija koda.

Primer: Upravljačka struktura Sekvenca

```
{  
    linija_koda_1  
    linija_koda_2  
    ...  
    linija_koda_n  
}
```

Redosled izvršavanja je: linija_koda_1 zatim linija_koda_2, ..., i kao poslednja linija_koda_n.

Konkretno:

```
{  
    x = x+1;  
    y=4+x;  
    z=y-x;  
}
```

Na početku bloka mogu da budu navedeni opisi promenljivih i konstanti koje su lokalne za blok.

Primer: Lokalne promenljive za blok

```
{  
double x, y, z;  
int i;  
x=x+1;  
y=4+x;  
z=y-x;  
i = 5;  
}
```

Selekcija

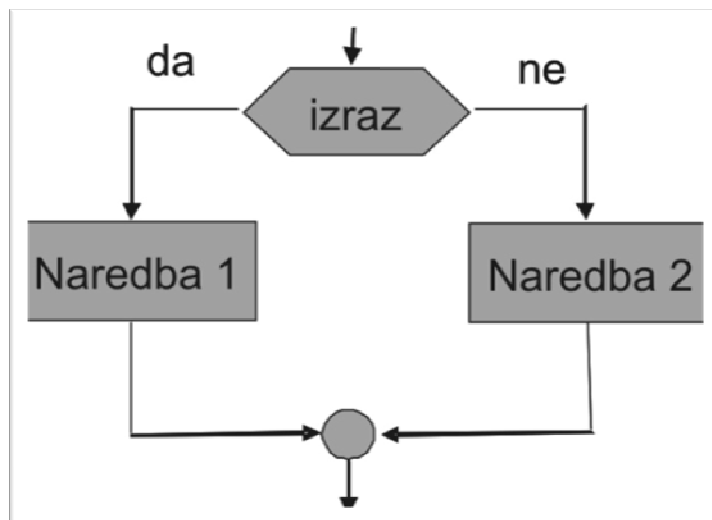
Upotrebom samo sekvence može da se reši samo ograničeni skup jednostavnih problema koje karakteriše fiksna struktura bez obzira na ulaze podatke ili pak akcije korisnika. U praksi je mnogo češći slučaj da se obrada ulaznih podataka značajno razlikuje zavisno od njihovih vrednosti. Selekcija ili grananje je upravljačka struktura koja pruža mogućnost izbora zavisno od nekog uslova. U slučaju da je uslov zadovoljen izvršava se jedan skup linija koda tj jedan blok odnosno ako uslov nije zadovoljen onda se izvršava drugi skup linija koda tj drugi blok.

Selekcija ili grananje se u C# realizuju naredbama:

- `if`
- `switch`

if-naredba

if-naredbom se implementira osnovi tip selekcije (grananja) kojom se vrši uslovno izvršenje jedne od dve moguće alternative. Standardni dijagram toka strukture grananja prikazan je na sledećoj slici



Treba uočiti da je jedna alternativa Naredba1 koja se izvršava kada logički izraz ima vrednost tačno (true) a druga alternativa je Naredba2 koja se izvršava kada izraz imavrednost netačno (false). Naredba1 i

Naredba2 mogu biti bilo koja izvršna naredba ili neka upravljačka struktura (sekvenca, selekcija ili iteracija).

Ekvivalentan kod u C#-u je:

```
if (izraz)
    Naredba1
else
    Naredba2
```

Dejstvo: Kada je vrednost izraza različita od nule (tj. true), izvršava se naredba <Naredba1>, u suprotnom, izvršava se naredba <Naredba2>.

Primer: if naredba - osnovni oblik

Deo koda kojim se određuje veći od dva broja predstavljena promenljivama x i y i dodeljuje trećoj promenljivoj veciBroj je:

```
if ( x > y )
    veciBroj = x;
else
    vecibroj = y;
```

Veoma često je potrebno da se neki deo koda izvrši ako je zadovoljen neki uslov a u slučaju da nije zadovoljen uslov nije potrebno izvršiti ni jednu liniju koda. U ovim slučajevima nije potrebno pisati else granu tj imamo takozvani "skraćeni oblik" if naredbe.

```
if ( izraz )

    Naredba1
```

Primer: if naredba - "skraćeni oblik"

Deo koda kojim se određuje apsolutna vrednost promenljive x i dodeljuje promenljivoj absx je:

```
absx = x;
if ( x < 0 )
    absx = -x;
```

Isti efekat bi postigli i korišćenjem if naredbe sa else granom :

```
if ( x < 0 )
    absx = -x;
else
    absx = x;
```

switch - naredba

Višestruko grananje je vrsta grananja kod koga postoji više alternativa koje mogu biti izvršene zavisno od toga koji uslovi su zadovoljeni. Broj uslova je dva ili više. Višestruko grananje se može realizovati višestrukom primenom if naredbi kao što je to pokazano na sledećem primeru.

Primer: višestruka if naredba

Kod kojim se određuje najveći od 3 broja predstavljena promenljivama x, y i z i upisuje u promenljivu najvećiBroj je:

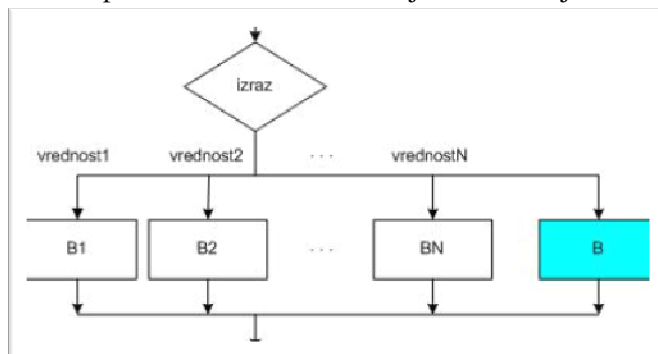
```
if ( x > y)
{
    if ( x > z )
        najvećiBroj = x;
    else
        najvećiBroj = z;
}
else
{
    if ( y > z )
        najvećiBroj = y;
    else
        najvećiBroj = z;
}
```

Postoje situacije gde su višestruka grananja takva da se izbor jedne od više mogućih alternativa vrši na osnovu neke vrednosti (vrednost neke promenljive ili pak izraza). Posmatrajmo kod koji treba za zadatu vrednost iz opsega (1 do 12) da prikazuje imena meseca u godini počev od zadatog do kraja godine (npr. ako je vrednost 10 program treba da ispiše oktobar, novembar, decembar). Ovaj problem je moguće rešiti višestrukom primenom if naredbi ali je mnogo elegantnije rešenje primenom switch naredbe koja ima opšti oblik:

```
switch ( izraz )
{
    case vrednost1: B1; break;
    case vrednost2: B2; break;
    ...
    case vrednostN: BN; break;
    default: B; break;
}
```

B1, B2, ..., BN i B su ili jednostavne naredbe ili neke upravljačke strukture ili blokovi.

Grafički prikaz switch naredbe dat je na sledećoj slici:



Može se uočiti da će se zavisno od vrednosti izraza izvršiti samo B ili pak neki od Bi gde važi da i uzima jednu od vrednosti 1, 2, ..., N. Ako izraz ne uzima ni jednu od vrednosti vrednost1, vrednost2, ..., vrednostN onda će se izvršiti samo B.

U nekim slučajevima ako izraz ne uzima ni jednu vrednost iz skupa vrednosti vrednost1, vrednost2, ..., vrednostN onda nije potrebno izvršiti nikakvu naredbu u tom slučaju će se koristiti switch naredba bez default case slučaja tj imaće oblik

```
switch ( izraz )
{
case vrednost1: B1; break;
case vrednost2: B2; break;
...
case vrednostN: BN; break;
}
```

Sledeći primer ilustruje primenu switch naredbe.

Primer: switch sa break

Ako se želi ispis samo imena tekućeg meseca onda switch naredba ima sledeći oblik:

```
switch( tekuci )
{
case 1: Text+="januar\n"; break;
case 2: Text+="februar\n"; break;
case 3: Text+="mart\n"; break;
case 4: Text+="april\n"; break;
case 5: Text+="maj\n"; break;
case 6: Text+="jun\n"; break;
case 7: Text+="jul\n"; break;
case 8: Text+="avgust\n"; break;
case 9: Text+="septembar\n"; break;
case 10: Text+="oktobar\n"; break;
case 11: Text+="novembar\n"; break;
case 12: Text+="decembar";
}
```

Iz prethodnih primera se može videti kako se višestruko grananje može realizovati pomoću switch naredbe. Međutim može se uočiti da broj case linija može biti veoma velik. U nekim slučajevima višestrukog grananja istu obradu treba izvršiti za više različitih vrednosti selektorskog izraza . U tim slučajevima se mogu spajati case linije kao što je to prikazano na sledećem primeru.

Primer: switch - više istih case

Kod koji u promenljivu brojDana upisuje broj dana tekućeg meseca odnosno 0 ako tekući mesec nije korektno unet je:

```

switch( tekuci )
{
case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    brojDana = 31; break;
case 2:
    brojDana = 28; break;
case 4: case 6: case 9: case 11:
    brojDana = 30; break;
default:
    brojDana = 0; break;
}

```

Napomena

Poznavaooci programskih jezika C i C++ će primetiti da se realizacija switch strukture u C# malo razlikuje. Naime, u ovim programskim jezicima osnovni oblik switch strukture je takav da se ne zahteva korišćenje break naredbe u svakoj case grani. Ova osobina pruža mogućnost da se relativno jednostavno realizuje program kojim se za dati ulaz tekući ispisuju imena svih meseci od tekućeg do kraja godine jer vrednost tekući samo određuje mesto gde se ulazi u niz case grana pri čemu se one izvršavaju do kraja switch strukture.

Zadatak: Proverite šta će se desiti ako se izostavi naredba break iz **case** grane u **switch** strukturi. Nakon toga napišite deo koda kojim se za dati mesec ispisuju imena svih meseci do kraja godine.

Iteracija

Veoma često se ukazuje potreba za ponavljanjem nekih naredbi više puta tj. za ponavljanjem nekog postupka u više **iteracija**. Iteracije se realizuju upotrebom upravljačkih struktura koje se nazivaju **petlje**. Postoji više vrsta petlji i one se mogu klasifikovati po više kriterijuma. Ako je kriterijum broj izvršavanja tela petlje onda postoje dve vrste petlji:

- a) petlje sa konstantnim brojem prolaza (brojačka petlja),
- b) petlje sa promenljivim brojem prolaza.

U programskom jeziku C# postoje dve vrste programskih petlji:

- petlje sa konstantnim brojem prolaza (unapred se zna koliko puta će se izvršiti telo petlje)
 1. for - petlja,
 2. foreach - petlja.
- petlje sa promenljivim brojem prolaza (broj izvršenja tela petlje se određuje u fazi izvršenja programa i zavisi od vrednosti za petlju relevantnih elemenata kao što su promenljive, elementi polja itd.)
 1. while - petlja,
 2. do-while - petlja.

for - petlja

for - petlja je petlja sa konstantnim brojem prolaza i naziva se još i brojačka petlja. Opšti oblik for petlje u programskom jeziku C# je:

```
for (<izraz1>; <izraz2>; <izraz3>)  
    <telo petlje>
```

gde:

<izraz1> - vrši inicijalizaciju promenljivih koje se koriste u petlji (što može da bude postavljanje početne vrednosti brojača petlje),

<izraz2> - predstavlja uslov na osnovu koga se odlučuje da li će se telo petlje još izvršavati ili se izvršavanje petlje prekida - petlja se izvršava dok je vrednost ovog izraza tačna (uslov za izvršenje tela petlje može da bude dok brojač petlje ne postigne svoju gornju granicu)

<izraz3> – definiše promenu vrednosti promenljivih koje se koriste u petlji. Navedena promena se vrši nakon svake iteracije (tu se može definisati kako se menja vrednost brojača petlje nakon svake iteracije).

<telo petlje> – telo petlje predstavlja jedna naredba, struktura ili blok.

Napomena: Bilo koji od ovih izraza može biti izostavljen, ali se znak ';' mora pisati.

foreach - petlja

Za objašnjenje foreach petlje potrebno je prethodno objasniti pojam skupa elemenata. Ukratko to je više elemenata koji su uređeni, na primer niz ili kolekcija o kojoj ćemo kasnije više govoriti.

Opšti oblik foreach petlje je:

```
foreach (tip promenljiva in izraz)
```

telo petlje

promenljiva je promenljiva koja se koristi u petlji za pristup svim elementima specificiranim u izrazu *izraz*.

Osnovno korišćenje petlje foreach biće ilustrovano sledećim primerom.

Primer

Neka je zadat celobrojni niz nekiNiz definisan kao:

```
int [ ] nekiNiz = { 1, 5, 2, 7, 3};
```

Sledećom foreach petljom se prolazi kroz sve elemente niza nekiNiz i oni se prikazuju na standardnom izlazu


```
strung izlaz = "";  
  
foreach (int indeks in nekiNiz)  
{  
  
    izlaz = izlaz + "element niza je = " + indeks + "\n";  
  
}  
  
MessageBox.Show(izlaz);
```

Rezultat rada ove petlje je prikaz elemenata niza nekiNiz tj

```
element niza je = 1  
element niza je = 5  
element niza je = 2  
element niza je = 7  
element niza je = 3
```

while - petlja

while petlja ima opšti oblik

```
while (<uslov>)  
  
    <telo petlje>
```

<uslov> - predstavlja logički izraz koji može biti tačan ili netačan. Ako je uslov zadovoljen onda se izvršava telo petlje. Nakon izvršenja poslednje naredbe u telu petlje opet se proverava uslov. while petlja se završava onog trenutka kada uslov više nije zadovoljen. To znači da uslov tj logički izraz sadrži elemente koji se menjaju u telu petlje. Na ovaj način se obezbeđuje da ne dodejmo u situaciju da je petlje beskonačna tj da se nikad ne izlazi iz petlje. Ovakve petlje se ne praktikuju mada u nekim specifičnim aplikacijama mogu da se pojave.

Primer

Sabiranje celih brojeva u intervalu od N do M moguće je obaviti i while petljom

```
int i = N;  
int suma = N;  
while ( i <= M)  
{  
    suma = suma + i;  
    i++;  
}
```

do while - petlja

do while petlja ima opšti oblik

```
do
  <telo petlje>
while (<uslov>)
```

<uslov> - predstavlja logički izraz koji može biti tačan ili netačan. Ova petlja funkcioniše tako što se izvršava telo petlje a zatim se proverava uslov. Ako je uslov ispunjen onda se opet izvršava telo petlje. U suprotnom se završava petlja.

Primer

Sabiranje celh brojeva u intervalu od N do M moguće je obaviti i petljom tipa do while na sledeći način:

```
int i = N;
int suma = 0;
do
{
    suma = suma + i;
    i++;
}
while (i <= M)
```

Naredbe skoka

Podrazumeva se da će se tela petlji izvršavati kompletno onoliko puta koliko je to zadato početnim uslovima i ostalim karakteristikama petlje. Međutim, u nekim situacijama će se želeći da se delovi tela petlje „preskoče“ tj. da se ne izvršavaju i da se nastavi sa sledećom iteracijom. Da bi se izmenio normalan tok izvršavanja neke upravljačke strukturei mogu se koristiti sledeće naredbe:

- goto
- break
- continue

goto naredba

goto naredba se još zove i naredbom bezuslovnog skoka. Ona omogućuje bezuslovni skok na neku labelu koja je definisana u programu. Labela se u programu dobija tako što se iza nekog identifikatora stavlja dvotačka npr. **ovoJeLabela**:

Naredba goto se koristi na sledeći način

```
goto labela;
```

tj u našem primeru

```
goto ovoJeLabela;
```

U sledećem primeru će biti ilustrovano korišćenje naredbe goto.

Primer

```
public static void Main()
{
    int ukupno = 0;
    int indeks = 0;
    string tekst = "";
    ovoJeLabela:
    indeks ++;
    ukupno += indeks;

    tekst = tekst + "indeks = " + indeks + "\n";

    if (indeks < 5)
    {

        tekst = tekst + "goto ovoJeLabela" + "\n";

        goto ovoJeLabela;
    }

    tekst = tekst + "ukupno = " + ukupno + "\n";
}
MessageBox.Show(tekst);
```

Rezultat izvršenja ovog programa je

```
indeks = 1
goto ovoJeLabela
indeks = 2
goto ovoJeLabela
indeks = 3
goto ovoJeLabela
indeks = 4
goto ovoJeLabela
indeks = 5
ukupno = 15
```

Napomena

Naredba goto se koristi samo u retkim situacijama i po pravilu je odraz loše projektovanog softvera te je kao takvu treba izbegavati jer veoma često dovodi do takozvanih nestrukturnih programa koji se u savremenom programiranju ne koriste.

break naredba

U nekim situacijama će biti potrebno prekinuti petlju tj njeno izvršavanje i prelazak na sledeću naredbu koja sledi iza petlje. Korišćenjem naredbe break može se prekinuti bilo koja vrsta petlje. U sledećem primeru će biti ilustrovano korišćenje naredbe break za prekid for petlje.

Primer

```
public static void Main()
{
    int ukupno = 0;
    string tekst = "";
    for (int indeks = 1; indeks <= 10; indeks++)
    {
        tekst = tekst + "indeks = " + indeks + "\n";
        ukupno += indeks;
        if (indeks == 5)
        {
            tekst = tekst + "prekid for petlje" + "\n";
            break;
        }
        tekst = tekst + "ukupno = " + ukupno + "\n";
    }
    MessageBox.Show(tekst);
}
```

Izvršenjem ovog programa dobiće se sledeći ispis:

```
indeks = 1
indeks = 2
indeks = 3
indeks = 4
indeks = 5
prekid for petlje
ukupno = 15
```

U slučaju da nije korišćena naredba break kojom je prekinuto dalje izvršavanje for petlje bili bi ispisani indeksi do 10 a vrednost promenljive ukupno bi bila 55.

Na isti način je moguće prekinuti bilo koju vrstu petlje.

continue naredba

Naredbom continue se ne prekida izvršavanje petlje već se samo prekida tekuća iteracija i nastavlja se sa sledećom iteracijom. Znači sve naredbe u telu petlje koje dolaze posle naredbe continue se preskaču i prelazi se na zaglavlje petlje tj proveravaju se uslovi za početak nove iteracije i ako su oni zadovoljeni ponovo se izvršava telo petlje od početka tj počinje nova iteracija. Korišćenje naredbe continue biće ilustrovano sledećim primerom.

Primer

```
public static void Main()
{
    int ukupno = 0;
    string tekst = "";
    for (int indeks = 1; indeks <= 10; indeks++)
    {
        if (indeks == 6)
        {
            tekst = tekst + "continue naredba" + "\n";
            continue;
        }
        tekst = tekst + "indeks = " + indeks + "\n";
        ukupno += indeks;
    }
    tekst = tekst + "ukupno = " + ukupno + "\n";
}
MessageBox.Show(tekst);
```

Rezultat izvršenja ovog programa je

```
indeks = 1
indeks = 2
indeks = 3
indeks = 4
indeks = 5
continue naredba
indeks = 7
indeks = 8
indeks = 9
indeks = 10
ukupno = 49
```

Primećuje se da je vrednost promenljive ukupno jednaka 49 tj. 6 manje od njene vrednosti koju bi imala u slučaju da u petlji nije korišćena naredba continue. U šestoj iteraciji je preskočena linija koda u kojoj se trenutna vrednost brojačke promenljive indeks dodaje na vrednost promenljive ukupno.

Intelli Sense

Obratite pažnju na pomoć koju vam pruža Visual Studio radno okruženje prilikom unosa koda. Ako ste počeli da unosite početna slova automatski dobijate prozor sa svim elementima koji su dostupni a počinu upravo tim slovima. Obratite pažnju da je na slici 2 prikazana jedna takva situacija: u liniji koda je ukucano samo **sa** i ispod se pojavio prozor koji nudi dve mogućnosti **satDrugIgrac** i **satPrviIgrac**. Dovoljno je podesiti strelicama fokus na željenu mogućnost i pritisnuti **Tab** taster i ceo tekst će se pojaviti kao deo vaše linije koda koju ste upravo počeli da pišete. Na ovaj način se značajno povećava brzina kucanja koda!!!

Kada ste ukucali neku promenljivu koja predstavlja zapravo neki objekat a zatim **tačku**, automatski se otvorio mali prozor koji vam nudi izbor iz liste svih svojstava i metoda koje poseduje klasa tog objekta. Na slici 1 je prikazan kod i prozor za objekat **satDrugiIgrac**. Vidi se da ovaj objekat ima 8 metoda koje su i prikazane u ovom prozoru.

Tehnologija koja pruža ovakvu (i još neke) vrste pomoći se zove **IntelliSense** i izuzetno je korisna jer skraćuje vreme unosa koda i eliminiše sintaksne greške. Primetićete da su svojstva prikazana plavom, a metode crvenom bojom. Takođe ćete videti metode koji niste sami napisali. To su metode koje je vaša klasa nasledila. O ovim metodama će biti reči kasnije.

```
private void btnPocetakIgre_Click(object sender, EventArgs e)
{
    brojPoteza = 0;
    InicijalizacijaMatrice();
    InicijalizacijaPolja();
    lblPobednik.Visible = false;
    satDrugiIgrac.

    satPrviIgrac.DodajMinut(1, 0);
    satDrugiIgrac.DodajSat(0, 0);
    lblVremePrviIgrac.Text = satPrviIgrac.ToString();
    lblVremeDrugiIgrac.Text = satDrugiIgrac.ToString();
}

private void btnZavrsetakIgre_Click(object sender, EventArgs e)
{
    satPrviIgrac.DodajSekundu(1, 0);
    lblVremePrviIgrac.Text = satPrviIgrac.ToString();
}
```

Slika 1: IntelliSense - izbor metode

```
private void btnPocetakIgre_Click(object sender, EventArgs e)
{
    brojPoteza = 0;
    InicijalizacijaMatrice();
    InicijalizacijaPolja();
    lblPobednik.Visible = false;
    sa

    RuntimeArgumentHandle
    RuntimeFieldHandle
    RuntimeMethodHandle
    RuntimeTypeHandle
    RunWorkerCompletedEventArgs
    RunWorkerCompletedEventHandler
    satDrugiIgrac
    satPrviIgrac
    SaveFileDialog
    SByte
    sbyte
    SByteConverter
    Scale
    ScaleChildren
    ScaleControl
```

Slika 2: IntelliSense

Nizovi

U rešenjima mnogih problema javlja se potreba za postojanjem većeg broja podataka istog tipa koje predstavljaju jednu celinu. Stoga se u programskim jezicima uvodi pojam niza ili u opštem slučaju pojam polja. Na primer, niz prostih brojeva manjih od 100. Nizovi mogu biti proizvoljnog tipa. Niz se deklarise na sledeći način:

```
tip[ ] imeNiza
```

gde tip može biti bilo koji tip. Uglaste zagrade iza oznake tipa ukazuju na to da je deklarisan niz a ne promenljiva.

Primer:

```
int[ ] celobrojniNiz;  
double[ ] alfaNiz;
```

Deklaracijom niza se samo kazuje da je neko ime niz a ne promenljiva. Za svaki niz je karakteristična i veličina niza tj. broj elemenata. To se radi na sledeći način:

```
celobrojniNiz = new int[20];
```

Operatorom new se kreira 20 elemenata tipa int koji su delovi niza celobrojniNiz. Ovaj niz (vektor) može imati maksimalno 20 elemenata tj. može imati i manje od 20 ali ne i više jer je u memoriji rezervisano tačno 20 mesta.

Moguće je objediniti i deklarisanje i definisanje niza kao

```
int[ ] celobrojniNiz = new int[20];
```

Svi elementi niza su inicijalizovani na podrazumevanu vrednost koja je za numeričke tipove nula. Inicijalizaciju možemo obaviti i sami navođenjem liste vrednosti u okviru para vitičastih zagrada međusobno odvojenih zarezima tj.

```
int[ ] celobrojniNiz = new int [5] { 1, 5, 8, 21, 3 }
```

Pristupanje elementima niza radi čitanja vrednosti ili upisa vrednosti vrši se pomoću operatora indeksa tj. srednjih zagrada kao

```
imeNiza[index]
```

Na primer

```
celobrojniVektor[5]
```

Indeksi nizova se kreću od 0. Znači, za niz celobrojniVektor prvi element ima indeks 0 a poslednji element ima indeks 19.

Važno je napomenuti da se nizovi zapravo realizuju kao objekti klase **System.Array** te je moguće koristiti svojstva i metode ove klase. Često korišćeno svojstvo je **Length** koje vraća broj elemenata niza.

Za prethodni slučaj celobrojniNiz.Length ima vrednost 20.

Primer: Suma nekog celobrojnog niza može se izračunati pomoću petlje

```
int[ ] celobrojniNiz = new int[20];  
//  
// deo koda za postavljanje elemenata niza  
//  
int suma = 0;  
for (int indeks = 0; indeks < celobrojniNiz.Length; indeks++)  
{  
    suma += celobrojniNiz[indeks];  
}
```

Primer

Ovaj primer ilustruje korišćenje jednodimenzionih polja (nizova, vektora).

```

string prikaz = "";
// polje celih brojeva
int[] celobrojnoPolje = new int[10];
int duzinaPolja = celobrojnoPolje.Length;
prikaz = prikaz + "duzina polja je " + duzinaPolja + "\n";
for (int brojac = 0; brojac < duzinaPolja; brojac++)
{
    celobrojnoPolje[brojac] = brojac;
    prikaz = prikaz + "celobrojnoPolje[" + brojac + "] = " +
        celobrojnoPolje[brojac] + "\n";
}

// polje znakova
char[] poljeZnakova = new char[6];
poljeZnakova[0] = 'z';
poljeZnakova[1] = 'd';
poljeZnakova[2] = 'r';
poljeZnakova[3] = 'a';
poljeZnakova[4] = 'v';
poljeZnakova[5] = '0';

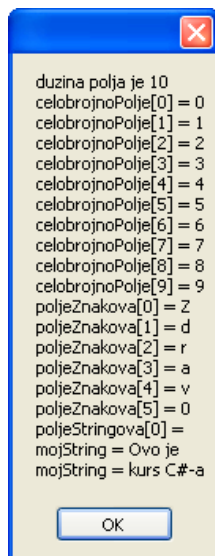
for (int brojac = 0; brojac < poljeZnakova.Length; brojac++)
{
    prikaz = prikaz + "poljeZnakova[" + brojac + "] = " +
        poljeZnakova[brojac] + "\n";
}

// polje stringova (niz znakova)
string[] poljeStringova = new string[2];
prikaz = prikaz + "poljeStringova[0] = " + poljeStringova[0] + "\n";
poljeStringova[0] = "Ovo je ";
poljeStringova[1] = "kurs C#-a";

foreach (string mojString in poljeStringova)
{
    prikaz = prikaz + "mojString = " + mojString + "\n";
}
MessageBox.Show(prikaz);

```

Startovanjem ovog programa dobija se izlaz prikazan na sledećoj slici.



Višedimenzionalni nizovi

Pored nizova koji imaju jednu dimenziju moguće je definisati nizove (polja) koji imaju dve ili više dimenzija. Veoma često se dvodimenzionalni nizovi poistovećuju sa pojmom matrice. Na primer, šahovska tabla se sastoji od crno belog polja dimenzija 8 puta 8. Višedimenzionalni nizovi mogu biti: pravougaoni i zupčasti (jagged).

Kod **pravougaonih nizova** broj elemenata u vrstama je isti dok se kod **zupčastih nizova** broj elemenata po vrstama može razlikovati. Pravougaono polje je na primer šahovska tabla jer svaka vrsta ima po 8 elementa. Ako na primer imamo dvodimenzionalno polje gde prva vrsta ima 3 elementa, druga vrsta 7 elemenata a treća vrsta recimo 5 elementa onda je to polje zupčasto.

Pravougaono polje se deklarise na sledeći način

```
tip[ , ] ime Polja
```

Primećuje se da u srednjim zagradam postoji znak , koji kazuje da je polje dvodimenzionalno. Trodimenzionalno polje će imati dva znaka , i tako dalje. Definisane šahovske table će biti

```
string[ , ] sahovskaTabla = new string[8, 8];
```

Za pristup nekom elementu polja potrebno nam je onoliko indeksa koliko dimenzija ima polje. Na primer sahovskaTabla[5,3] = "Beli kralj";

Napomena: Broj dimenzija nekog polja može se odrediti pomoću svojstva **Rank**.

Inicijalizacija višedimenzionalnih polja se može izvršiti po analogiji sa inicijalizacijom jednodimenzionalnog niza kao niz nizova vrednosti. Na primer:

```
string[ , ] imena = {  
    {"Pera", "Mika", "Laza", "Iva"}  
    {"Nena", "Ana", "Maja", "Zoran"}  
    {"Goran", "Dragan", "Jova", "Minja"}  
}
```

Zupčasto polje se deklarise kao

```
tip [ ] [ ] ... [ ] imePolja
```

Na primer,

```
string [ ] [ ] imena = new string[3][ ];
```

Dodela vrednosti može biti:

```
imena[0] = new string[4];  
imena[0][0] = "Pera";
```

```

imena[0][1]= "Mika";
imena[0][2]= "Laza";
imena[0][3]= "Iva";

imena[1]= new string[2];
imena[1][0]= "Nena";
imena[1][1]= "Ana";

imena[2]= new string[3];
imena[2][0]= "Goran";
imena[2][1]= "Dragan";
imena[2][2]= "Jova";

```

Primer: Ispis svih imena u polju imena može se izvršiti na sledeći način:

```

string tekst = "";
for (int vrsta = 0; vrsta < imena.Length; vrsta++)
{
    for ( int kolona = 0; kolona < imena[vrsta].Length; kolona++)
    {
        tekst = tekst + "imena[" + vrsta + "][" + kolona + "] = " + imena[vrsta][kolona] + "\n"
    }
}
MessageBox.Show();

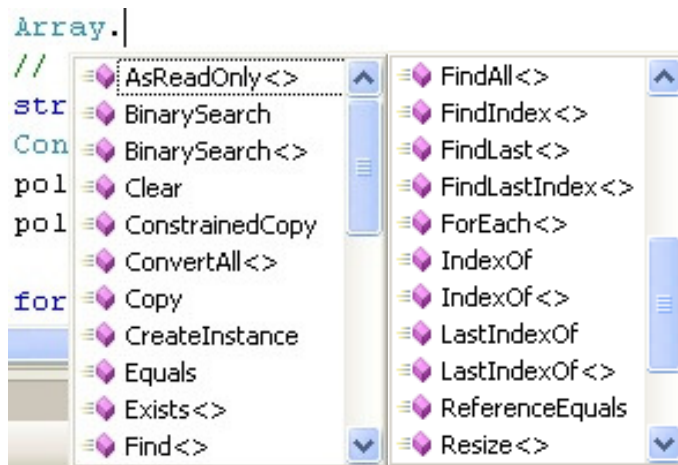
```

Često se kao zahtev nameće sortiranje nekog niza. Da bi obavili ovaj zadatak dovoljno je da iskoristimo činjenicu da je jedna od metoda klase System.Array i metoda **Sort**. Pozivom ove metode za odgovarajući niz taj niz se sortira. Na primer, sortiranje niza celobrojniNiz u rastućem redosledu izvršićemo sa

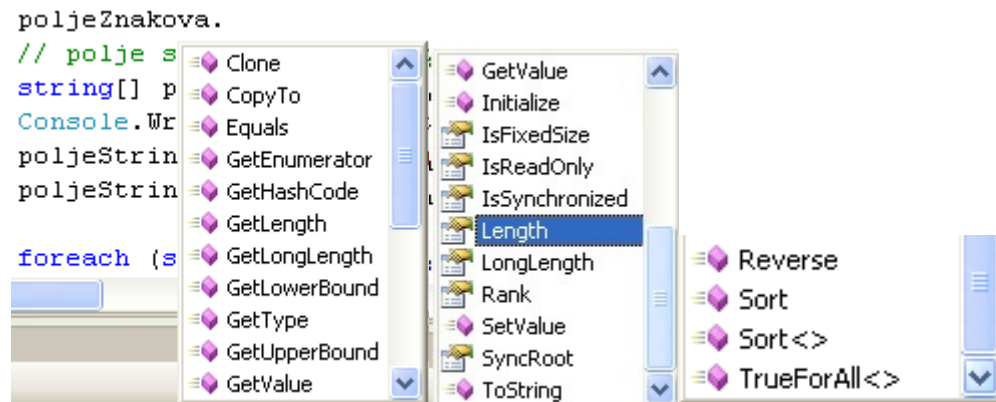
```
Array.Sort(celobrojniNiz);
```

Zadatak: Metoda Sort je preopterećena. Prouči sve oblike metode Sort.

Na sledećim slikama prikazane su metode klase Array:



Atributi polja su prikazani na sledećim slikama:



Objektno orijentisana paradigma

Bilo koji sistem se sastoji iz **objekata** koji međusobno komuniciraju. Svaki objekat ima neke karakteristike koje ga čine posebnim i razlikuju od drugih objekata. Veoma često neki objekti imaju iste vrste karakteristika ali različite vrednosti. Uzmimo, na primer, lopte za primere takvih objekata. Sve lopte imaju karakteristiku koja se zove boja čije vrednosti mogu biti različite od objekta do objekta. Tako vrednosti asocirane uz karakteristiku koja se zove boja mogu biti crvena, zelena, plava, itd. Objekat pored karakteristika koje se nazivaju i **atributima** ima u opštem slučaju i sposobnost nekakvog funkcionisanja tj. objekat se karakteriše i skupom funkcionalnosti koje se zadaju odgovarajućim funkcijama članicama tj. **metodama**. Tako na primer, neke lopte odskaču kada padnu na tlo više, a neke manje. Neke lopte mogu da plivaju (gumene), a neke ne mogu (lopte napravljene od olova tj. metalne kugle).

Programi se pišu sa ciljem da se reši neki problem tako što se svaki problem vezuje za neki sistem koji se opisuje (modeluje) na specifičan način. Stoga je najprirodniji način za rešavanje problema odgovarajućim programom, korišćenje objekata. Objekti predstavljaju osnovu objektno orijentisane (OO) paradigme. OO paradigma opisuje tj. modeluje sistem na prirodan način. Ako se ovako posmatraju stvari onda se grubo može reći da se programiranje korišćenjem OO paradigme sastoji od:

- Identifikovanja objekata.
- Identifikovanja atributa objekata kao i njihovih vrednosti.
- Identifikovanje funkcionalnosti objekata.
- Klasifikacija objekata prema atributima i funkcionalnostima u kategorije tj. **klase**. Rezultat klasifikacije je definicija klasa sa svim potrebnim svojstvima (atributima) i funkcionalnostima koje se nazivaju još i metode.
- Identifikovanje komunikacije između objekata (koji objekti komuniciraju, kada i na koji način, tj. koje poruke razmenjuju.).
- Generisanje klasa.
- Generisanje objekata i njihova komunikacija.
- Uništavanje objekata.

Klase i objekti

Klase su kao što smo rekli opisi koji definišu kako će se generisati objekti i koje osobine i funkcionalnosti ti objekti poseduju. Na primer, želimo da opišemo jednu školu kao sistem. Školu čine učenici i profesori. Svaki učenik se može okarakterisati velikim brojem osobina kao što su: ime, prezime, datum rođenja, mesto rođenja, itd. Takođe se i svaki profesor može okarakterisati osobinama kao što su ime, prezime, datum rođenja, mesto rođenja. Da li to znači da se učenici i profesori mogu posmatrati kao isti objekti u sistemu koji se zove škola? Očigledno ne, pošto i učenici i profesori imaju pored pobrojanih i neke specifične osobine. Tako učenik kao osobinu ima i razred koji pohađa, predmete i odgovarajuće ocene iz tih predmeta, itd. dok profesori imaju predmete koje drže, nivo školske spreme, datum zaposlenja, itd. Očigledno je da u modelu škole imamo dve osnovne klase: klasu **Učenik** i klasu **Profesor**. Instance ovih klasa tj. **objekti** su konkretni učenici i profesori škole.

Znači može se reći da klase predstavljaju opise tj. šablone za opis zajedničkih karakteristika grupe objekata. Klasom se vrši generalizacija grupe objekata tj. zanemaruju se karakteristike koje nisu jedinstvene za grupu objekata posmatranih u jednom kontekstu (na primer u kontekstu jedne škole nije bitno da li neki učenik ima plave ili crne oči ili pak da li neki učenik ima tetku u Americi). Klasa treba da sadrži samo one atribute i metode koji su bitni za sve objekte koje modeluje klasa u nekom kontekstu. Sve druge specifičnosti objekata se zanemaruju i ne ulaze u opis klase.

Sa druge strane ako posmatramo neki objekat on ima svoje karakteristike tj. atribute i ispoljava neko ponašanje opisano metodama. Za druge objekte koji komuniciraju sa nekim objektom nije bitno kako objekat realizuje neke funkcionalnosti već samo koje funkcionalnosti on poseduje.

Razmotrimo klasu Automobil. On ima kao atribute između ostalog: boju, broj vrata, dužinu, širinu, pogonsko gorivo (benzin, dizel, gas), marku, godinu proizvodnje, pređenu kilometražu, itd. U kontekstu vožnje automobila interesantne su nam funkcije koje nam omogućavaju pokretanje motora vozila (recimo pomoću ključa ili pak kartice ili pak samo pritiskom na dugme), pokretanje vozila unapred, pokretanje vozila unazad, povećanje brzine, smanjenje brzine, zaustavljanje vozila i zaustavljanje rada motora. Sve ovo nam je potrebno da bi vozili neki automobil. Toliki ljudi poznaju ove funkcije automobila a veoma mali broj ljudi zna kako zapravo funkcioniše motor sa unutrašnjim sagorevanjem !!! Ali, da i im je to uopšte potrebno sa stanovišta vožnje automobila? Zaista nije! Znači kada opisujemo funkcionalnosti objekata neke grupe objekata modelirane nekom klasom realizacija samih funkcionalnosti će biti važna samo samoj klasi ali ne i drugim objektima koji koriste objekte te klase. Jednom rečju, jedna od postavki objektno orijentisane paradigme je i sakrivanje detalja koji nisu bitni za korisnike objekata. Da li je to fer? Jeste, jer delovima motora neka se bave konstruktori a mene interesuje samo kako da upravljam mojim automobilom! Ako nastavimo u ovom pravcu doći ćemo do sledećeg pitanja: da li je bitno da li automobil koristi benzin ili gas ako automobil posmatram smao kao sredstvo koje treba da me preveze od mesta A do mesta B a cena me ne interesuje? Očigledno da ne jer u oba slučaja se automobil koristi na isti način.

Iz ove priče se može zaključiti da je pri definisanju klase od mnoštva karakteristika objekata koji se modeliraju klasom potrebno izdvojiti samo one atribute i metode koji su bitni za kontekst u kome se posmatraju dati objekti. Isto tako, sve one koji koriste neki objekat potrebno je

osloboditi poznavanja unutrašnjih detalja klase (objekta) koji mu ništa ne znače u kontekstu komunikacije sa objektom tj. korišćenja datog objekta.

Za razliku od proceduralnog programiranja gde su podaci bili razdvojeni od procedura za njihovu obradu u objektno orijentisanoj paradigmi se kroz pojam klase vrši objedinjavanje podataka i operacija koje mogu da se vrše nad njima. Ko može da pristupi kom podatku klase odnosno kojoj metodi klase zavisi od toga kako su im zadata prava pristupa odnosno kakav im je modifikator pristupa. U programskom jeziku C# postoje sledeći modifikatori pristupa:

- private
- public
- protected
- protected internal
- internal

Ako niko spolja ne treba da pristupi članu klase onda se taj član proglašava **privatnim** (private). Ako pak član klase treba da bude dostupan svima onda se on proglašava **javnim** (public). **Zaštićeni** (protected) član je dostupan samo unutar klase ili unutar izvedenih klasa koje su izvedene iz date klase (o pojmu izvedenih klasa će biti više reči u narednim lekcijama). **Protected internal** član je dostupan samo unutar klase, unutar izvedenih klasa koje su izvedene iz date klase ili klasa u istom programu. Internal član je dostupan samo unutar klase ili klasa u istom programu. Kako je pravo pristupa izuzetno bitno to se na početku deklaracije svakog člana klase navodi pravo pristupa. Ako se ono izostavi podrazumeva se da je privatno. Prava pristupa će kasnije opet biti razmatrana sa odgovarajućim primerima.

Nasleđivanje

Veoma često se za dva programa može reći da su veoma slična jer modeluju slične sisteme koji imaju iste mnoge karakteristike ali imaju i neke specifičnosti. Ako je nivo sličnosti velik a nivo specifičnosti mali postavlja se pitanje da li ćemo razvijati novu aplikaciju "od nule" ili ćemo pokušati da iskoristimo postojeću. Ako nam je kriterijum efikasnost tj. vreme onda ćemo sigurno izabrati ovaj drugi pristup. Ako uporedimo objekte koji se sreću u ovim aplikacijama primetićemo da je veliki broj objekata veoma sličan.

Objektno orijentisana paradigma uvodi pojam **nasleđivanja** tj. **izvođenja** koji predstavlja mogućnost da se jedna klasa definiše na osnovu neke druge klase pri čemu će ta nova klasa da ima sve osobine koje ima i postojeća klasa, neke osobine će biti promenjene a neke nove će biti dodate. To konkretno znači da ne moramo da prepisujemo postojeću klasu (poznati "copy-paste" pristup koji se često koristi u proceduralnom programiranju i predstavlja izvor mnogih grešaka) već samo da navedemo od koje klase polazimo tj. koja je **osnovna** ili **roditeljska** klasa i da modifikujemo ili dodamo neke članice klase.

Primer

Posmatrajmo klasu **Profesor**. U jednoj školi postoje i razredne starešine. Oni su takođe profesori ali imaju i dodatnu funkciju. Znači, možemo da definišemo klasu **RazredniStaresina** koja će biti izvedena iz klase Profesor. Dodati član klase RazredniStaresina je na primer odeljenje kome je dati profesor razredni starešina. Klasa RazredniStaresina će imati sve one članove koje ima i klasa Profesor. Ovo je trivijalan primer koji treba samo da ilustruje koncept izvođenja ili nasleđivanja. Kasnije će biti dato više detalja.

Definicija klase u programskom jeziku C#

Klasa se u programskom jeziku C# (nadalje nećemo naglašavati da se radi o programskom jeziku C# jer će se nadalje sve odnositi na ovaj programski jezik) definiše na dva načina zavisno od toga da li je osnovna klasa ili klasa koja je izvedena iz jedne ili više drugih klasa. Osnovna klasa se definiše na sledeći način:

```
class NazivKlase
{
/*
* telo klase sa svim potrebnim svojstvima,
* metodima i događajima
*/
}
```

Kreiranje objekata

Objekti (instance klase) se kreiraju pomoću operatora **new** kao u sledećem primeru.

```
Pravougaonik alfa = new Pravougaonik();
```

Ovom linijom koda se promenljiva (često se koristi i termin **varijabla**) alfa deklarise kao promenljiva tipa Pravougaonik a ujedno se i kreira objekat klase Pravougaonik i dodeljuje toj promenljivoj. Isti efekat se može postići i na sledeći način

```
Pravougaonik alfa;
alfa = new Pravougaonik();
```

tj. razdvajanjem deklaracije od kreiranja objekta.

Pristup članovima klase

Članovi klase su podaci članovi (atributi) i funkcije članice (metode). Kada se kreira jedan objekat neke klase pristupanje njegovim članovima vrši se primenom operatora tačka tj.

pristupanje podacima članovima

`imeObjektaKlase.ImeAtributa`

pristupanje funkcijama članicama

`imeObjektaKlase.NazivFunkcije(stvarni argumenti)`

Primer

```
Pravougaonik prav1 = new Pravougaonik(2.5, 3.0);
MessageBox.Show("Povrsina je " + prav1.Povrsina());
```

U prvoj liniji je kreiran objekat prav1 klase Pravougaonik a zatim je u drugoj liniji na standardni izlaz MessageBox ispisan tekst **Povrsina je** i površina objekta prav1 koja se dobija pozivom njegove metode Povrsina().

Podsetimo se da je neophodno deklarirati metodu Povrsina() kao **javnu** da bi moglo da joj se pristupi!

Napomena

Može se primetiti da u prethodnom primeru konstruktor klase Pravougaonik ima dva argumenta.

Konstruktori

Konstruktori su funkcije koje karakteriše: isto ime kao i ime klase, nemaju povratnu vrednost. Sve ostale karakteristike koje važe za bilo koju funkciju važe i za konstruktore. Svaki put kada se kreira objekat neke klase zapravo se poziva konstruktor te klase. Veoma često se u konstruktorima vrši inicijalizacija podataka članova klase.

Konstruktori mogu biti bez argumenata (tada se zovu **podrazumevani konstruktori** ili **default konstruktori**) ili sa argumentima. Jasno je da može postojati samo jedan podrazumevani konstruktor i više konstruktora sa parametrima. Ako ne definišemo podrazumevani konstruktor sam sistem će ga kreirati i telo konstruktora će biti prazno.

Znači, svaki put kada se kreira novi objekat neki konstruktor se poziva!

Primer

Definišimo za klasu Pravougaonik podrazumevani konstruktor koji će da postavlja vrednosti atributa duzina i sirina na nulu i konstruktor koji ima dva parametra kojima se postavljaju početne vrednosti parametara duzina i sirina.

```
public class Pravougaonik
{
    private float duzina;
    private float sirina;
    public Pravougaonik() // podrazumevani konstruktor
    {
        duzina = 0;
        sirina = 0;
    }
    // konstruktor sa parametrima
    public Pravougaonik( float duz, float sir)
    {
        duzina = duz;
        sirina = sir;
    }
}
```

Destruktori

Konstruktori se pozivaju svaki put kada se kreira neki objekat. Pri uništavanju objekata pozivaju se destruktori pre samog uništavanja objekta. Destruktori su funkcije koje imaju isto ime kao i ime klase tj. ime konstruktora sa dodatkom znaka ~ (tilda) kao prvog simbola imena. Za razliku od konstruktora destruktori nemaju definisan kvalifikator pristupa public. Takođe nemaju ni povratnu vrednost kao ni naredbu return u telu. Destruktori takođe nemaju ni parametre.

```
~Pravougaonik()
{
    // neki kod
}
```

Izvedene klase

Kao što je u uvodu rečeno jedna od osnova objektno orijentisane paradigme je mehanizam **nasleđivanja** tj. **izvođenja** (engl. inheritance). Jedna klasa može biti izvedena iz neke druge klase i kao rezultat nasleđivanja osobine osnovne tj. roditeljske klase se prenose i na izvedenu klasu (sreću se i termini klasa potomak, sin itd). Posmatrajmo klasu Zaposleni. Ova klasa kao atribut ima: ime zaposlenog i prezime zaposlenog. U konstruktoru se postavljaju ime i prezime zaposlenog.

```
public class Zaposleni
{
    public string ime;
    public string prezime;
    public Zaposleni(string ime, string prezime)
    {
        this.ime = ime;
        this.prezime = prezime;
    }
    public void Informacije()
    {
        MessageBox.Show("ime: " + ime + " prezime: " + prezime);
    }
}
```

Posmatrajmo sada klasu Rukovodilac. Rukovodilac je takođe i zaposleni u nekoj firmi i on ima i svoje ime i svoje prezime ali isto tako ima i rukovodeću poziciju. Znači, klasa Rukovodilac je samo jedan specijalni slučaj klase Zaposleni tj. ima iste osobine kao i klasa Zaposleni kao i svoju specifičnu osobinu (rukovodeću poziciju) koja je karakteriše. Stoga će klasa Rukovodilac biti izvedena iz klase Zaposleni. U definiciji izvedene klase se iza naziva klase navode dve tačke iza kojih sledi ime osnovne klase tj

```
public class Rukovodilac : Zaposleni
{
    public string pozicijaRukovodioca;
    public Rukovodilac(string ime, string prezime, string pozicija)
    : base(ime, prezime)
    {
        pozicijaRukovodioca = pozicija;
    }
}
```

Izvedena klasa ne nasleđuje konstruktor osnovne klase. Ako je definisan konstruktor u osnovnoj klasi potrebno je definisati i konstruktor u izvedenoj klasi. Da bi pozvali konstruktor osnovne klase u konstruktoru izvedene klase mora stajati eksplicitni poziv.

Primećujemo da konstruktor klase Rukovodilac ima 3 parametra od kojih se prva dva koriste kao parametri konstruktora osnovne klase tj. klase Zaposleni. Ovaj konstruktor se poziva korišćenjem službene reči base koja dolazi iza dve tačke i odnosi se na konstruktor osnovne klase. Konstruktor osnovne klase će izvršiti inicijalizaciju članova ime i prezime a u telu konstruktora izvedene klase će se izvršiti inicijalizacija člana pozicijaRukovodioca.

Napomena: Iz jedne osnovne klase moguće je izvesti više izvedenih klasa.

Kreiranje objekta izvedene klase je slično kreiranju objekta osnovne klase tj. navođenjem operatora new i pozivom konstruktora sa odgovarajućim parametrima. U našem primeru objekat klase Rukovodilac može se kreirati sa:


```
Rukovodilac sefRacunovodstva = new Rukovodilac("Petar", "Peric", "sef odeljenja finansija");
```

Kreirali smo objekat koji se zove sefRacunovodstva čije je ime Petar, prezime Peric i on je na poziciji šefa odeljenja finansija. Ako želimo da prikazemo informacije o šefu računovodstva koristićemo metodu Informacije() koja je definisana u osnovnoj klasi i njenim pozivom se prikazuju ime i prezime. Ako želimo da prikazemo i poziciju šefa eksplicitno koristimo atribut pozicijaRukovodioca tj.

```
sefRacunovodstva.Informacije();  
MessageBox.Show("na poziciji " +sefRacunovodstva.pozicijaRukovodioca);
```

Zgodno bi bilo da sve potrebne informacije dobijemo samo jednim pozivom metode Informacije. To je moguće tako što ćemo u izvedenoj klasi Rukovodilac predefinisati metodu Informacije i tada kažemo da je ova metoda „pregazila“ metodu iz osnovne klase. Da bi ovo bilo moguće u osnovnoj klasi metodu koju hoćemo da predefinišemo u izvedenoj klasi proglašavamo virtualnom tako što dodamo službenu reč **virtual** a u izvedenoj klasi istu tu metodu proglašavamo metodom koja prepisuje tj. „gazi“ virtualnu metodu iz osnovne klase tako što dodajemo službenu reč **override**. Sada klase izgledaju

```
public class Zaposleni
```

```
{  
    public string ime;  
    public string prezime;  
    public Zaposleni(string ime, string prezime)  
    {  
        this.ime = ime;  
        this.prezime = prezime;  
    }  
    public virtual void Informacije()  
    {  
        MessageBox.Show("ime: " + ime + " prezime: " + prezime);  
    }  
}
```

```
public class Rukovodilac : Zaposleni
```

```
{  
    public string pozicijaRukovodioca;  
    public Rukovodilac(string ime, string prezime, string pozicija)  
    : base(ime, prezime)  
    {  
        pozicijaRukovodioca = pozicija;  
    }  
    public override void Informacije()  
    {  
        MessageBox.Show("ime: " + ime + " prezime: " + prezime + " na poziciji: " + pozicijaRukovodioca);  
    }  
}
```

Ako sada pozovemo metodu Informacije za objekat tipa Rukovodilac prikazaće se i ime i prezime i pozicija.

U slučaju da je iz osnovne klase izvedena još neka klasa i u ovim klasama bi se mogla predefinisati ista metoda Informacija. Na ovaj način smo dobili da se jedna ista metoda ponaša na više različitih načina. Ova osobina se naziva **polimorfizam** (grčka reč poli znači više, a reč morf znači oblik).

Dodajmo sada našim klasama i osobine koje se odnose na način kako se računa plata. U osnovnu klasu dodajemo sledeće attribute i metode

```
protected int brojRadnihSati;  
protected float koeficijent;  
protected float zarada;  
public void Koeficijent(float koef)  
{  
    this.koeficijent = koef;  
}  
public void RadniSati(int sati)  
{  
    brojRadnihSati = sati;  
}  
public virtual float Plata()  
{  
    zarada = brojRadnihSati * koeficijent;  
    return zarada;  
}
```

Dodatni atributi su sa kvalifikatorom pristupa **protected** što znači da su dostupni samo unutar izvedene klase ali ne i izvan nje.

U izvedenu klasu dodajemo novi atribut dodatniKoeficijent jer rukovodilac ima dodatna primanja na račun svog radnog mesta tako da se plata rukovodioca računa na drugačiji način.

```
float dodatniKoeficijent;  
public void DodatniKoeficijent(float koef)  
{  
    dodatniKoeficijent = koef;  
}  
public new float Plata()  
{  
    zarada = base.Plata() + brojRadnihSati * dodatniKoeficijent;  
    return zarada;  
}
```

Sada metoda Plata() umesto **override** ima prefiks **new**. To znači da je metoda Plata() nova metoda. Ova metoda sakriva metodu iz osnovne klase tj. ako za neki objekat klase Rukovodilac pozovemo metodu Plata() izvršiće se nova metoda definisana u klasi Rukovodilac. To znači da i dalje postoji metoda osnovne klase koju možemo pozvati kao **base.Plata()**. Konkretno, u našem slučaju u telu nove metode Plata() pozvana je između ostalog i metoda Plata() iz osnovne klase.

Napomena: Ako se koristi prefiks `override` onda metoda osnovne klase nije dostupna tj. ona je „pregažena“.

Klasa `System.Object`

Klasa `System.Object` se ponaša kao osnovna klasa za sve klase tj. sve klase su izvedene iz ove klase. Ovo implicitno izvođenje je ugrađeno u sam jezik C# i nema potrebe da se explicitno naglašava da je neka klasa izvedena iz klase `System.Object`. To praktično znači da bilo koja klasa može da koristi metode klase `System.Object` kojih ima veći broj i neke od njih su izuzetno korisne i često primenjivane kao na primer:

- `string ToString()` - vraća string, čija je podrazumevana vrednost jednaka imenu objekta klase,
- `bool Equals(object)` - vraća true ako su dva objekta jednaka; ova metoda je metoda instance i preopterećena je metodom klase `Equals()` koja je opisana u sledećem redu.
- `static bool Equals(object, object)` - vraća true ako su dva objekta jednaka; ova metoda je metoda klase i preopterećena je metodom instance `Equals()` koja je opisana u prethodnom redu.
- `static bool ReferenceEquals(object, object)` - vraća true ako su dve reference jedanke tj. ako referenciraju isti objekat,
- `Type GetType()` - vraća klasu (ili tip) objekta (ili promenljive),
- `void Finalize()` - omogućava objektu čišćenje pre nego što skupljač đubreta (garbage collector) izbriše objekat iz memorije (u osnovi isto kao i destruktork),
- `object memberwiseClone()` - kreira kopiju objekta.

Napomena: Iz klase `System.Object` su izvedeni čak i tipovi kao što su na primer `int` i `char`. Pogledajmo sledeći primer:

```
int mojInteger = 10;
MessageBox.Show("mojInteger.ToString() = " + mojInteger.ToString()+"\n" +
                "mojInteger.GetType() = " + mojInteger.GetType());
```

kao rezultat ovog koda dobiće se sledeći ispis:

```
mojInteger.ToString() = 10
mojInteger.GetType() = System.Int32
```

Ovo može donekle da bude zbunjujuće! Kako se promenljiva može ponašati kao objekat? Odgovor leži u činjenici da se u toku izvršavanja programa promenljiva najpre implicitno konvertuje u objekat klase `System.Object`. I ovaj proces se naziva **boxing**.

Razmotrimo sledeći kod:

```
int mojInteger = 10;
object mojObjekat = mojInteger; // boxing promenljive mojInteger
```

U ovom slučaju je najpre odrađen boxing promenljive `mojInteger` a zatim je izvršeno dodeljivanje objektu **mojObjekat**.

Napomena: `System.Object` i `object` označavaju istu stvar. Oba naziva se odnose na `System.Object`. Zbog toga je u prethodnom kodu, `mojObjekat` objekat klase `System.Object`.

S druge strane pak objekat klase `System.Object` se može konvertovati u vrednosni tip i ovaj proces je poznat pod nazivom **unboxing**. U sledećem primeru je izvršen unboxing objekta **mojObjekat** u promenljivu tipa `int`:

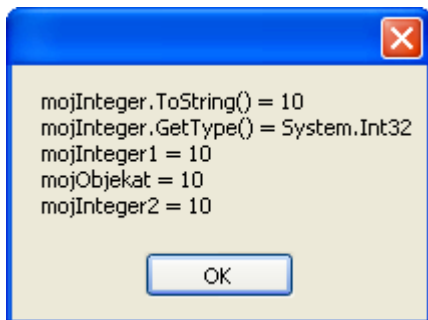
```
int mojInteger = (int) mojObjekat; // unboxing objekta mojObjekat
```

Primer: boxing i unboxing

Posmatrajmo sledeći kod:

```
string tekst = "";  
// implicitni boxing tipa int  
int mojInteger = 10;  
tekst = tekst + "mojInteger.ToString() = " + mojInteger.ToString() + "\n";  
tekst = tekst + "mojInteger.GetType() = " + mojInteger.GetType() + "\n";  
  
// eksplicitni boxing int u object  
int mojInteger1 = 10;  
object mojObjekat = mojInteger1; // boxing promenljive mojInteger1  
tekst = tekst + "mojInteger1 = " + mojInteger1 + "\n";  
tekst = tekst + "mojObjekat = " + mojObjekat + "\n";  
  
// eksplicitno unboxing object u int  
int mojInteger2 = (int)mojObjekat; // mojObjekat je unboxed  
tekst = tekst + "mojInteger2 = " + mojInteger2 + "\n";  
MessageBox.Show(tekst);
```

Kao rezultat izvršenja ovog koda dobija se izlaz kao na sledećoj slici:



Apstraktne klase

Posmatrajmo klasu Figura. Ova klasa ima atribut ime i metodu Povrsina.

```
abstract public class Figura  
{  
    public string ime;  
    public void postaviIme(string ime)  
    {  
        this.ime = ime;  
    }  
    abstract public float Povrsina();  
}
```

Iz ove klase moguće je izvesti dve nove klase DvodimenzioneFigure i TrodimenzioneFigure koje predstavljaju figure u ravni odnosno u prostoru i u skladu sa tim njihov položaj u ravni odnosno prostoru

određen je koordinatama gornjeg levog temena. Gornje levo teme ima svoje koordinate predstavljene atributima `float X_Koordinata`; `float Y_Koordinata`; i `float Z_Koordinata`;
Iz klase `DvodimenzionaFigure` izvedena je klasa `Pravougaonik` a iz klase `TrodimenzionaFigure` izvedena je klasa `Kvadar`. `Pravougaonik` je određen svojom dužinom i širinom a `kvadar` ima i visinu. Za figure u prostoru pored površine moguće je izračunati i zapreminu pa je dodatak metoda `float Zapremina()`.

```
abstract public class Figura
```

```
{
    public string ime;
    public void postaviIme(string ime)
    {
        this.ime = ime;
    }
    abstract public float Povrsina();
}
```

```
abstract public class DvodimenzionaFigura : Figura
```

```
{
    float X_Koordinata;
    float Y_Koordinata;
    public DvodimenzionaFigura(float X, float Y)
    {
        X_Koordinata = X;
        Y_Koordinata = Y;
    }
}
```

```
abstract public class TrodimenzionaFigura : Figura
```

```
{
    float X_Koordinata;
    float Y_Koordinata;
    float Z_Koordinata;
    public TrodimenzionaFigura(float X, float Y, float Z)
    {
        X_Koordinata = X;
        Y_Koordinata = Y;
        Z_Koordinata = Z;
    }
    abstract public float Zapremina();
}
```

```
public class Pravougaonik : DvodimenzionaFigura
```

```
{
    float duzina;
    float sirina;
    public Pravougaonik( float X_koo, float Y_koo, float duz, float sir) : base(X_koo, Y_koo)
    {
        duzina = duz;
        sirina = sir;
    }
}
```

```

    public override float Povrsina()
    {
        return duzina * sirina;
    }
}

public class Kvadar : TrodimenzionaFigura
{
    float duzina;
    float sirina;
    float visina;
    public Kvadar(float X_koo, float Y_koo, float Z_koo, float duz, float sir, float vis): base(X_koo,
Y_koo, Z_koo)
    {
        duzina = duz;
        sirina = sir;
        visina = vis;
    }

    public override float Povrsina()
    {
        return 2*(duzina * sirina + duzina * visina + sirina * visina);
    }

    public override float Zapremina()
    {
        return duzina * sirina * visina;
    }
}

```

Instanciranjem neke klase dobijaju se konkretni objekti koji imaju konkretne osobine. Posmatrajmo klasu Figura. Šta bi bila instanca ove klase tj. kakav bi to bio objekat? Očigledno je da je teško zamisliti takav objekat jer ga ne možemo tačno okarakterisati. Nasuprot tome ako kažemo da je neki objekat instanca klase Pravougaonik imamo tačnu predstavu o tom objektu tj gde se nalazi u ravni i koje su mu dimenzije. Može se zaključiti da neke klase nisu pogodne da bi kreirali instance tih klasa ali su zato pogodne za izvođenje novih klasa. Takav slučaj je sa klasom Figura. Ovakve klase se nazivaju apstraktnim klasama. Da je neka klasa apstraktna označava se službenom rečju abstract na početku deklaracije klase kao što je to slučaj sa klasom Figura ali isto tako i sa klasama DvodimenzionaFigura i TrodimenzionaFigura. Apstraktne klase mogu imati apstraktne metode koje isto tako imaju ključnu reč abstract u deklaraciji. Ove metode nemaju telo i one se definišu u izvedenim klasama.

Znači, apstraktne klase nemaju svoje instance tj objekte, koriste se za izvođenje drugih klasa, konstruktor apstraktne klase ne može biti apstraktan i apstraktne metode iz osnovne klase moraju se pregaziti metodama u izvedenim klasama.

Statički članovi klase

U okviru klase se definišu članovi: atributi i metode. Da bi pozvali neku metodu ili pak pristupili nekom atributu potrebno je znati za koju instancu klase tj. konkretni objekat se poziva metoda odnosno pristupa se atributu. Međutim, u nekim slučajevima je potrebno omogućiti pristup nekom članu a da se ne vezujemo ni za jedan konkretni objekat klase. Na primer, ako želimo da u svakom trenutku imamo evidenciju o broju kreiranih objekata neke klase potrebno je da imamo neku promenljivu čija će vrednost da se poveća za jedan svaki put kada se kreira novi objekat klase i da se smanji za jedan svaki put kada se uništi neki objekat klase. Ova promenljiva ne treba da pripada ni jednoj konkretnoj instanci klase već celoj klasi. Ovo se postiže pomoću takozvanih **statičkih članova klase**. Da bi označili da je neki član statički dodaje mu se kvalifikator **static**.

Slična situacija je i sa metodama koje takođe mogu biti statičke. Pristup statičkom članu klase se razlikuje od pristupa običnom članu klase jer se ne zahteva objekat već se navodi samo ime klase koje sledi operator tačka i ime statičkog člana klase. Na primeru klase Automobil ilustrovaćemo način definisanja i korišćenja statičkih članova klase.

```
public class Automobil
{
    private static int brojAutomobila = 0;
    public Automobil()
    {
        MessageBox.Show("kreiran objekat klase Automobil");
        brojAutomobila++;
    }

    ~Automobil()
    {
        MessageBox.Show("unisten objekat klase Automobil");
        brojAutomobila--;
    }

    public static int BrojObjekata()
    {
        return brojAutomobila;
    }
}
```

Klasa ima statički atribut brojAutomobila koji se povećava za jedan u telu konstruktora odnosno smanjuje za jedan u telu destruktora. Na taj način se vodi evidencija o trenutnom broju objekata klase Automobil. Statička metoda BrojObjekata() vraća vrednost promenljive brojAutomobila tj. informaciju o broju objekata. Da bi pozvali ovu metodu nije nam potreban ni jedan objekat već je dovoljno pozvati je sa imenom klase tj.

```
MessageBox.Show("Broj objekata klase automobil je " + Automobil.BrojObjekata());
```

Napomena: U telu statičke metode ne možemo koristiti pokazivač **this** jer se metoda ne odnosi ni na jedan konkretni objekat te je stoga this neodređeno.

Prava pristupa

Pristup elementima klase zavisi od modifikatora pristupa koji imaju. Kao što je već rečeno u programskom jeziku C# postoje sledeći modifikatori pristupa:

- private,
- public,
- protected,
- protected internal i
- internal.

Ponovimo njihovo značenje koje će sada biti mnogo jasnije jer sada znamo i šta su to izvedene klase tj. šta je to nasleđivanje odnosno izvođenje.

Ako niko spolja ne treba da pristupi članu klase onda se taj član proglašava **privatnim** (private). Ako pak član klase treba da bude dostupan svima onda se on proglašava **javnim** (public). **Zaštićeni** (protected) član je dostupan samo unutar klase ili unutar izvedenih klasa koje su izvedene iz date klase. **Protected internal** član je dostupan samo unutar klase, unutar izvedenih klasa koje su izvedene iz date klase ili klasa u istom programu. **Internal** član je dostupan samo unutar klase ili klasa u istom programu.

Kako je pravo pristupa izuzetno bitno to se na početku deklaracije svakog člana klase navodi pravo pristupa. Ako se ono izostavi **podrazumeva se da je privatno**.

Modifikatorom public se dozvoljava pristup članovima klase bez ikakvog ograničenja. To znači da vrednost atributa klase može da se menja i u klasi i van klase.

Primer: Definišimo klasu Ucenik koja ima dva javna atributa: Ime i Prezime.

```
class Ucenik
{
    public string Prezime;
    public string Ime;
}
```

Zaštićena svojstva

Oba svojstva u prethodnom primeru, korisnik klase može da postavi na bilo koju vrednost. U ovom slučaju to ne predstavlja problem. Ali ako želimo da u klasu **Ucenik** dodamo svojstvo **Razred**, pod pretpostavkom da je u pitanju učenik osnovne škole, validne vrednosti mogu biti samo brojevi od 1 do 8. Ako svojstvo **Razred** napravimo kao **javnu (public)** promenljivu, nemamo načina da kontrolišemo i proverimo vrednost koja je upisana u nju i da zabranimo pogrešne vrednosti.

Da bi ovo ostvarili, moramo posedovati neki mehanizam koji proverava vrednost pre nego što se dodeli promenljivoj. Ovo se postiže pomoću specijalnih **set** i **get** funkcija. Tehnika je sledeća: umesto da se pravi **javna (public)** promenljiva Razred, treba napraviti **privatnu (private)** promenljivu i radi bolje čitljivosti joj dati slično ime – na primer **prRazred (pr** od private).

U ovom momentu još uvek nemamo svojstvo (engl. Property) Razred, ali ga dodajemo pomoću **set** i **get** funkcija. Sintaksa je jednostavna, navodi se tip i naziv svojstva koje je sada **javno (public)** i u okviru njega upišu **set** i **get** funkcije. Izмениćemo postojeći kod kako bi bilo jasnije:


```

class Ucenik
{
    public string Ime;
    public string Prezime;
    private int prRazred;
    public int Razred
    {
        get
        {
            return prRazred;
        }
        set
        {
            prRazred = value;
        }
    }
}

```

Privatna promenljiva `prRazred` nije vidljiva van klase, i korisnici klase ne mogu niti da postavljaju niti da čitaju njenu vrednost. Ipak, ova promenljiva je ona koja je nama bitna i sa kojom radimo unutar svoje klase. Korisnik može da postavlja vrednost ove promenljive, ali na posredan način pomoću `public` svojstva deklarisanog u liniji `public int Razred`. Svojstvo `Razred` se sastoji od dva dela: **get** deo koji se automatski poziva kada korisnik čita vrednost svojstva `Razred` i **set** deo koji se automatski poziva kada korisnik postavlja vrednost svojstva `Razred`.

U **get** delu, se korisniku zapravo vraća vrednost privatne promenljive `prRazred`:

```
return prRazred;
```

U **set** delu se dodeljuje vrednost privatnoj promenljivoj `prRazred`.

```
prRazred = value;
```

Promenljiva **value** je rezervisana reč u jeziku C# i predstavlja vrednost koju je korisnik upotrebio prilikom dodele vrednosti svojstvu. I ovde se nalazi ključ za proveru vrednosti pre dodele promenljivoj `prRazred`. Pre linije `prRazred = value` treba proveriti da li je sadržaj promenljive **value** ispravan. Ako jeste dodeljujemo tu vrednost promenljivoj `prRazred`, u suprotnom najčešće generišemo grešku sa odgovarajućom porukom.

Znači možemo da rezimiramo: Ako želimo da napravimo zaštićena svojstva, kod kojih treba vršiti proveru pre dodele vrednosti procedura je sledeća:

- Napraviti privatnu promenljivu koja ima isto ime kao željeno svojstvo sa nekim prefiksom. (Ovu konvenciju imenovanja treba shvatiti kao preporuku, a ne kao čvrsto pravilo.)
- Napraviti javno (`public`) svojstvo sa željenim imenom i tipom istim kao privatna promenljiva
- Unutar javnog svojstva uneti **get** i **set** delove.
- U **get** delu korisniku pomoću **return** naredbe vratiti vrednost privatne promenljive.
- U **set** delu proveriti vrednost koju korisnik dodeljuje svojstvu (**value**) i ako je vrednost validna dodeliti je privatnoj promenljivoj. U suprotnom prikazati poruku o grešci.

Važno je primetiti da se ovde javno svojstvo koristi kao **interfejs** ka privatnom atributu objekta. Kao neka vrsta zaštite gde se mogu ispitati validne vrednosti pre dodele privatnoj promenljivoj. Ovo ispitivanje može biti jednostavno, ali i veoma složeno zavisno od potrebe. Dobro napravljena klasa treba sama sebe da "brani" od pogrešnih vrednosti. Nekada ćete je koristiti vi, a nekada drugi programeri i to stalno treba imati u vidu.

Nadalje će biti govora o najčešćem načinu za proveru, kako se pokreće i obrađuje greška u slučaju da vrednost nije validna.

Neki uslov može biti i složen kada se sastoji od dva ili više osnovna uslova. U tom slučaju neophodno je navesti i logičke operatore koji vezuju dva ili više uslova. Na primer, želimo da proverimo da li promenljiva **a** ima vrednost 5 i promenljiva **b** ima vrednost 3.

Kompletan uslov je tačan samo ako su tačni iskazi **a == 5** i **b == 3**. Logički operand koji na ovaj način spaja dva uslova se zove **I** (eng. and) i simbolički se obeležava se **&&**:

```
if (a == 5 && b == 3)
```

// izraz je tačan samo ako su oba uslova tačna.

Ako na primer, želimo da proverimo da li je promenljiva **a > 0** ili promenljiva **b < 10** koristi se logički operand **II** (eng. or) koji se simbolički obeležava sa dve vertikalne crte **||**:

```
if (a > 0 || b < 10)
```

// izraz je tačan ako je tačan ili prvi ili drugi ili oba uslova

Ako se sada vratimo na poslednji primer gde validna vrednost svojstva **Razred**, odnosno vrednost privatne promenljive **prRazred** treba da bude u opsegu od 1 do 4 uključujući, ispitivanje bi obavili na sledeći način:

```
if (value > 0 && value < 5)
```

U slučaju da je izraz tačan dodelićemo privatnoj promenljivoj **prRazred** vrednost **value**, a u slučaju da nije, potrebno je na neki način obavestiti korisnika klase da je napravio grešku. Dopišite sledeće linije koda:

```
public int Razred
{
    get
    {
        return prRazred;
    }
    set
    {
        if (value > 0 && value < 5)
        {
            prRazred = value;
        }
        else
        {
            // greska
        }
    }
}
```

Na ovaj način smo zaštitili privatnu promenljivu **prRazred** od pogrešnih vrednosti. Dodeljujemo joj vrednost samo ako je **value** veće od 0 i manje od 5, odnosno u intervalu od 1 do 4. Šta raditi ako korisnik klase unese pogrešnu vrednost? Taj slučaj se obrađuje u **else** delu uslova gde trenutno stoji samo komentar **// greska**.

Najčešći način je da se generiše greška u izvršavanju koja se prosleđuje korisniku klase. On je dalje odgovoran za obradu ove greške, a ako to ne uradi greška će zaustaviti izvršavanje programa.

Obrada izuzetaka - greške

U .NET okruženju greška je predstavljana klasom **Exception**. Tačnije rečeno, postoji više vrsta ovakvih klasa koji se vezuju za specifične greške kao na primer rad sa datotekama, rad sa bazama podataka i slično. Klasa **Exception** predstavlja generalnu klasu za predstavljanje greške koju ćemo mi i koristiti. Kao i kod svake klase, prvo je neophodno napraviti objekat pomoću naredbe **new**:

```
Exception greska = new Exception ("Pogrešna vrednost svojstva Razred");
```

U ovoj liniji koda smo napravili objekat na osnovu klase **Exception** i taj objekat je predstavljen promenljivom greska. Odmah prilikom kreiranja moguće je, kao ulazni parametar konstruktora, napisati proizvoljni tekst koji opisuje grešku.

U našem slučaju greška je opisana tekстом "Pogrešna vrednost svojstva Razred" i to je poruka koju će dobiti korisnik klase kada vrednost svojstva **Razred** postavi van validnog opsega.

Pokretanje ili podizanje ili bacanje ili izazivanje greške (eng. raise error) se izvršava naredbom **throw** gde je argument objekat koji predstavlja grešku:

```
throw greska;
```

Kada se izvrši **throw**, klasa prestaje sa radom i ovu grešku prenosi na korisnika klase, koji potom treba da je obradi na odgovarajući način.

Sada treba izmeniti primer kao što je prikazano:

```
static void Main(string[] args)
{
    Ucenik cu = new Ucenik();
    cu.Prezime = "Perić";
    cu.Ime = "Petar";
    cu.Razred = 3;
    MessageBox.Show (cu.Prezime + " " + cu.Ime + " " + cu.Razred);
}
i potom izmeniti set deo u klasi:
set
{
    if (value > 0 && value < 5)
    {
        prRazred = value;
    }
    else
    {
        Exception greska = new Exception("Pogrešna vrednost svojstva Razred");
        throw greska;
    }
}
```

Korisnik klase postavlja ispravnu vrednost za razred (**cu.Razred = 3;**) i klasa ne generiše nikakvu grešku. Uredno ispisuje dodeljenu vrednost u konzolnom prozoru

Obrada grešaka

Da bi sve funkcionisalo ispravno, korisnik klase ne treba da dozvoli da bilo koja greška nasilno završi njegovu aplikaciju. Ovo se odnosi na sve moguće greške koje mogu nastati u radu programa, a ne samo na naš primer.

Greške možemo podeliti u tri grupe:

- 1. Sintaksne greške** (eng. syntax errors) – rezultat pogrešno napisane naredbe, nedeklarisane promenljive i slično. Ove greške najčešće nisu problematične jer program koji ih sadrži ni ne može da se kompajlira.
- 2. Greške u vreme izvršavanja** (eng. runtime errors) – kao u našem primeru, greške koje se mogu ali i ne moraju dogoditi za vreme izvršavanja programa. Najčešće su rezultat neproverenih vrednosti promenljivih i pogrešnog unosa korisnika. Relativno se lako pronalaze i ispravljaju.

3. Logičke greške (eng. 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.

Obrada greške u C# se vrši pomoću naredbi **try** i **catch**. U osnovnom obliku sintaksa je sledeća:

```
try
{
    // linije koda koje mogu da generišu grešku
}
catch (Exception promenljiva)
{
    // obrada greške
}
finally
{
    // blok naredbi koje se izvršavaju bilo da se desio izuzetak ili ne
}
```

Upotreba je jednostavna, svaku liniju ili više njih koje mogu generisati grešku treba smestiti u **try** blok. U slučaju da se generiše greška, tok programa se automatski preusmerava na **catch** blok. U njemu pišemo kod koji se izvršava kada se pojavi greška tj. ispravlja grešku ako je to moguće ili pak obaveštava korisnik kakva je greška nastala. Opciono pomoću **Exception** promenljive možemo proveriti koja greška se dogodila, videti njen opis i zavisno od toga izvršiti odgovarajuću akciju. U našem primeru, greška se može dogoditi na liniji gde se dodeljuje vrednost svojstvu **Razred** – dakle ta linija se smešta u **try** blok. U **catch** bloku koji obrađuje grešku ćemo u konzolnom prozoru ispisati opis greške. Izmenite kod na sledeći način:

```
static void Main(string[] args)
{
    Ucenik cu = new Ucenik();
    cu.Prezime = "Perić";
    cu.Ime = "Petar";
    try
    {
        cu.Razred = 7;
    }
    catch (Exception ex)
    {
        MessageBox.Show ( ex.Message );
    }
    MessageBox.Show (cu.Prezime + " " + cu.Ime + " " + cu.Razred);
}
```

U obradi greške smo u konzoli ispisali tekst greške koristeći **Message** svojstvo klase **Exception**. Klasa **Exception** je u kodu predstavljena objektom tj. promenljivom **ex**. Kada pokrenemo program (obratite pažnju, vrednost svojstva **Razred** je postavljena na pogrešnu vrednost 7), dobija se sledeći rezultat tj. ispis:

Pogrešna vrednost svojstva Razred

Ovde je važno zapaziti da program nije nasilno zaustavljen zbog greške, već je ona obrađena i program je nastavio sa radom. Takođe, vrednost svojstva **Razred** je ostala nepromenjena, odnosno ostavljena na inicijalnu vrednost nula.

Namespace

U razvoju iole kompleksnijeg softvera po pravilu učestvuje više soba ili pak više različitih timova. U tom slučaju nije redak slučaj da će više osoba tj više timova u toku razvoja koda delova softvera koristiti za pojedine klase ista imena. Da bi izbegli eventualni konflikt i greške u korišćenju ovih klasa koristi se mehanizam koji se naziva namespace (prostor imena ili imenski prostor). Jedan namespace čini više deklaracija klasa koje čine jednu logičku celinu. Kada je potrebno koristiti klase iz nekog namespace-a potrebno je samo označiti o kom namespace-u se radi. Znači potrebno je samo voditi računa da ne dođe do konflikta u imenovanju samo u okviru namespace-a tjne mogu se javiti ista imena klasa u okviru jednog namespace-a. Do sada je prećutno korišćen System namespace tako što smo koristili klasu Console (odnosno njenu metodu WriteLine) koja je njegov deo.

Imenski prostor se deklarise korišćenjem ključne reči namespace iza koje sledi ime imenskog prostora koji delarišemo. Na primer,

```
namespace Grafika
{
    // deklaracija klase
    public class Figura
    {
        public string imeFigure;
    }
}
```

Ime Figura se može koristiti za imenovanje neke druge klase u nekom drugom namespace-u npr. namespace za klase za neki šahovski program.

```
namespace Sah
{
    // deklaracija klase
    public class Figura
    {
        public string imeFigure;
    }
}
```

Ako želimo da kreiramo objekat klase Figura iz namespace-a Grafika to ćemo uraditi tako što ćemo pored imena klase navesti i ime namespace-a tj

```
Grafika.Figura geometrijskaFigura = new Grafika.Figura();
```

Isto tako kreiranje objekta klase Figura iz namespace-a Sah se može obaviti na sledeći način

```
Sah.Figura sahovskaFiguraLovac = new Sah.Figura();
```

Imenski prostori mogu biti delovi nekog drugog imenskog prostora odnosno mogu imati svoje imenske prostore pa se kaže da imenski prostori mogu imati hijerarhijsku strukturu. Na primer, namespace Grafika može obuhvatati imenske prostore GrafikaURavni i GrafikaUProstoru tj

```
namespace Grafika
{
    // deklaracija imenskog prostora GrafikaURavni
    namespace GrafikaURavni
```

```

{
    // klase
}

// deklaracija imenskog prostora GrafikaUProstoru
namespace GrafikaUProstoru
{
    // klase
}

```

Napomena: Jedan imenski prostor nije ograničen samo na jedan fajl tj više fajlova mogu da sadrže opise klase koje su deo istog imenskog prostora.

Ako imamo duboku hijerarhiju imenskih prostora korišćenje klase u okviru najdublje ugnježenog imenskog prostora može biti poprilično nepraktično jer se ispred imena klase treba navesti kompletna hijerarhija imenskih prostora. Da bi izbegli ovakve stvari moguće je pre korišćenja klase iz imenskog prostora koristiti naredbu using kojom obaveštavamo kompajler da ćemo nadalje koristiti klase iz nekog imenskog prostora tako da nije potrebno navoditi kompletnu hijerarhiju već samo ime klase. Na primer

```

using Sah;
class SahovskaTabla
{
    Figura sahovskaFiguraLovac = new Figura();
    ...
}

```

Ako je potrebno koristiti klase koje imaju ista imena a nalaze se u različitim imenskim prostorima obavezno moramo uz njihovo ime koristiti i naziv imenskog prostora da bi kompajler znao o kojoj klasi se radi. Na primer

```

using Sah;
using Grafika;
class SahovskaTabla
{
    Sah.Figura sahovskaFiguraLovac = new Sah.Figura();
    Grafika.Figura slikaLovca = new Grafika.Figura();
    ...
}

```

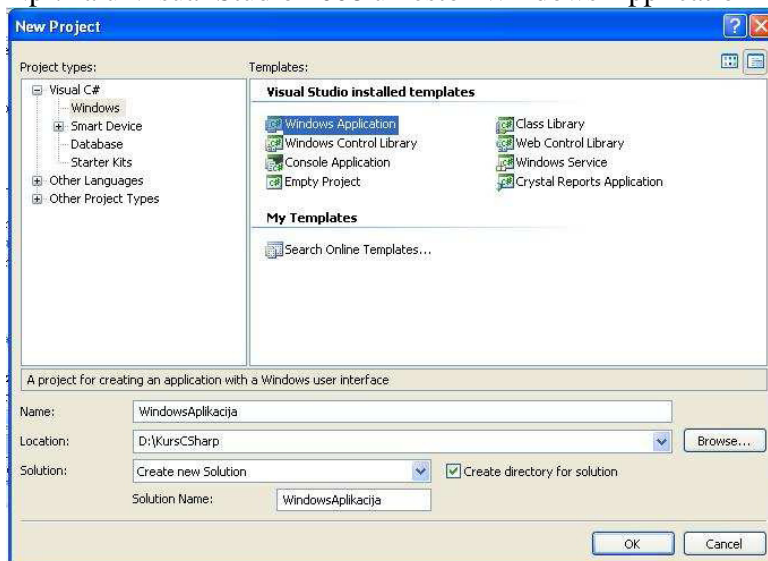
Razvoj Windows aplikacija u C#.NET

Karakteristika Windows aplikacija je bogat grafički korisnički interfejs (GUI). Windows aplikacije se sastoje od jedne ili više formi na kojima se nalaze kontrole (na primer, dugmad, labele(natpisi), combobox, listbox, edit polja, itd.). Svaka forma i kontrola ima sposobnost prihvatanja odgovarajućih događaja koje proizvodi korisnik (npr klik levim tasterom miša ili samo klik, dupli levi klik (dupli klik levim tasterom miša), desni klik, prevlačenje miša preko kontrole itd) ili pak sam sistem tako da se korišćenje jedne windows aplikacije svodi na komunikaciju sa korisnikom prepoznavanjem akcija korisnika i generisanjem odgovarajućeg odziva. Sve akcije se jednim imenom nazivaju **događajima** (engl. events) tako da se ovakav način programiranja sreće i pod nazivom programiranje upravljano događajima. Ovaj vid programiranja se zbog ove karakteristike odlikuje i drugačijim pristupom u razvoju aplikacije u odnosu na npr. proceduralno programiranje. Naime, najpre se generiše izgled korisničkog interfejsa a zatim se određuju događaji za pojedine forme i kontrole na koje će sistem da reaguje tj. da da neki odziv. Na kraju se za svaki događaj piše odgovarajuća funkcija kojom se definiše odziv aplikacije na neku akciju korisnika.

Pošto se u prvi plan stavlja interfejs tj. forme i kontrole koje su više manje standardizovane po pitanju funkcionalnosti, savremena okruženja za razvoj Windows aplikacija nude unapred definisane klase kojim su opisane forme i kontrole. Veoma često se ovakav razvoj Windows aplikacija naziva i vizualnim programiranjem. Takva situacija je i sa okruženjima VisualStudio koja nude bogat skup unapred definisanih biblioteka klasa. Kako je broj tih biblioteka odnosno klasa velik objasnimo samo one najčešće korišćene. Ako se ima u vidu da je princip koji je korišćen u realizaciji ovih biblioteka isti upoznavanje sa svojstvima ostalih ne bi trebao da predstavlja veći problem.

Kreiranje novog Windows projekta

Kreiranje projekta jedne Windows aplikacije je slično kreiranju projekta konzolne aplikacije s tom razlikom da se pri izboru vrste aplikacije odabere „Windows Application“ kao što je to prikazano na slici 1 (Slika 1 je iz okruženja Visual Studio 2005. Analogno je i za ostala Visual Studio okruženja. Npr. za u Visual Studio 2008 umesto "Windows Application" stoji "Windows Forms Application").



Slika 1: Kreiranje projekta Windows aplikacije (MS Visual Studio 2005)

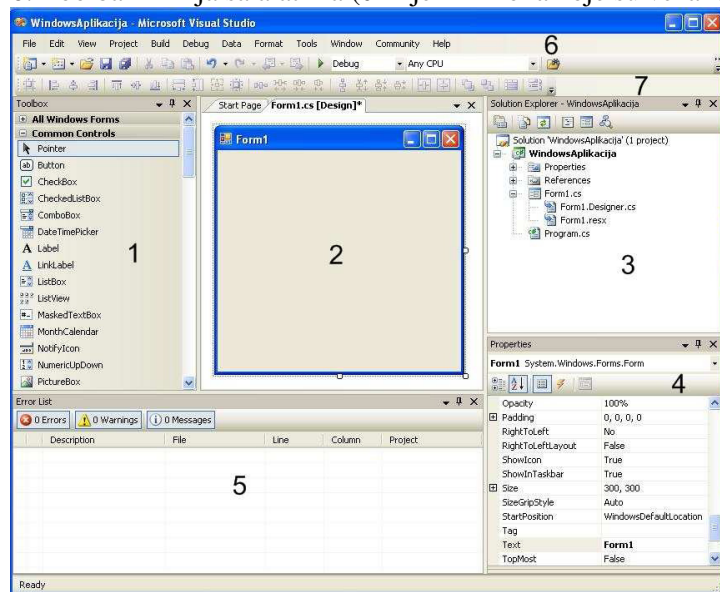
Po kreiranju novog projekta može se uočiti da se za razliku od slučaja kada smo kreirali konzolnu aplikaciju u centralnom delu pojavila prazna forma (slika 2) koja predstavlja početnu formu koja će se prikazati kada startujemo naš program.

Startujemo naš „program“ opet pritiskom na funkcijski taster F5. Kao rezultat dobićemo praznu formu tj. standardnu Windows formu na kojoj neće biti ni jedne kontrole (slika 3). Šta možemo da uradimo sa našim programom? Možemo da pomeramo formu (levi klik mišem na formu i povlačenje po desktopu), da je maksimizujemo odnosno minimizujemo i da je zatvorimo (levi klik na: kvadratić, liniju odnosno krstić, respektivno (u gornjem desnom uglu forme)). Očigledno je da nismo napisali ni jednu liniju koda a da smo dobili nekakvu Windows aplikaciju. Šta se zapravo desilo? Pokretanjem novog Windows projekta samo okruženje je odradilo sav posao umesto nas i kreiralo glavnu formu aplikacije koja predstavlja početnu tačku tj. osnovu u razvoju bilo koje Windows aplikacije. Znači, našu buduću aplikaciju ćemo da razvijamo tako što ćemo na osnovnu formu da dodamo neke elemente interfejsa a potom i da ispišemo kod za odgovarajuće događaje forme odnosno dodatih elemenata interfejsa (kontrola). Da bi efikasno odradili ove dve etape u razvoju Windows aplikacije razvojna okruženja Visual Studio su projektovana tako da su upravo prilagođena ovim aktivnostima. Na slici 2 je dat prikaz razvojnog okruženja Visual Studio 2005 i njegovih elemenata potrebnih za razvoj Windows aplikacija (Slično izgledaju i ostale verzije okruženja).

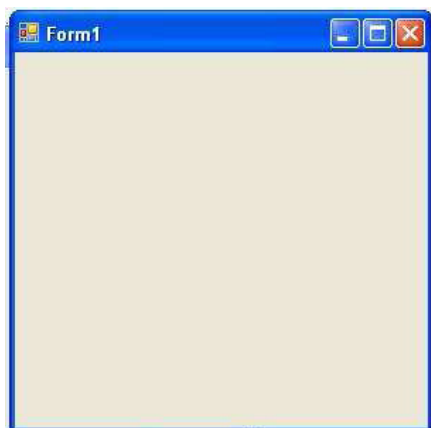
Elementi razvojnog okruženja prikazani na slici 2 su:

Toolbox (paleta sa alatima) - deo u kome se nalaze gotove komponente koje se mogu koristiti prevlačenjem na formu (drag-drop)

1. Forma - osnovna (i druge) forma koja čini osnovu interfejsa jedne Windows aplikacije (na formu se postavljaju komponente)
2. Solution Explorer - prozor u kome se prikazuju svi fajlovi jednog projekta odnosno Solution-a (Jedan Solution može da čini i više projekata kod ozbiljnijih softverskih proizvoda).
3. Properties window (Menadžer osobina) - prozor u kome se prikazuju sve osobine selektovane kontrole kao i svi mogući događaji kontrole.
4. Prozor za prikaz dodatnih informacija (Errors, Warnings, Messages) - prikaz različitih informacija, npr. rezultati kompajliranja itd.
5. Linija menija - Linija sa menijima razvojnog okruženja.
6. Toolbar - linija sa alatima (čini je niz ikona koje su vezane za pojedine funkcionalnosti okruženja).



Slika 2: Razvojno okruženje za razvoj Windows aplikacija

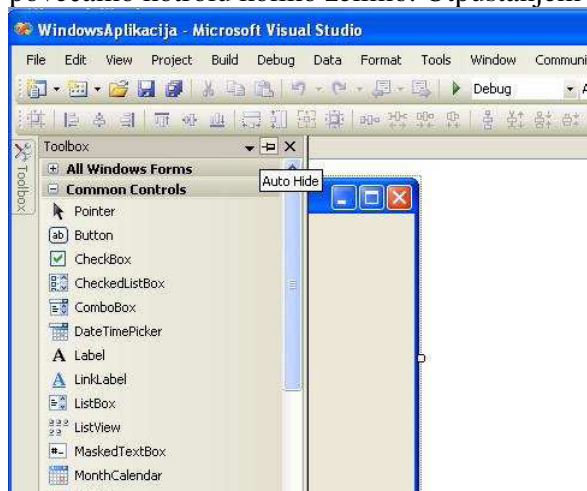


Slika 3: Prazna forma

Dodavanje elemenata interfejsa tj. **kontrola** (npr. dugme, labela, itd. (O kontrolama će kasnije biti više reči.)) na formu vrši se prevlačenjem iz **Toolboxa**(paleta sa alatima) u kome se nalaze kontrole grupisane u više grupa (paleta). Da bi se prikazao Toolbox treba pokazivač miša dovući na oznaku Toolboxa (leva strana okruženja ispod Toolbar linije).

Ako pomerimo miša van Toolboxa on će opet biti sakriven. Da se ovo ne bi dešavalo tj. da bi Toolbox stalno bio vidljiv potrebno je na vrhu Toolboxa kliknuti na simbol **AutoHide** (simbol čiode ili pritiskača) kao što je to prikazano na slici 4.

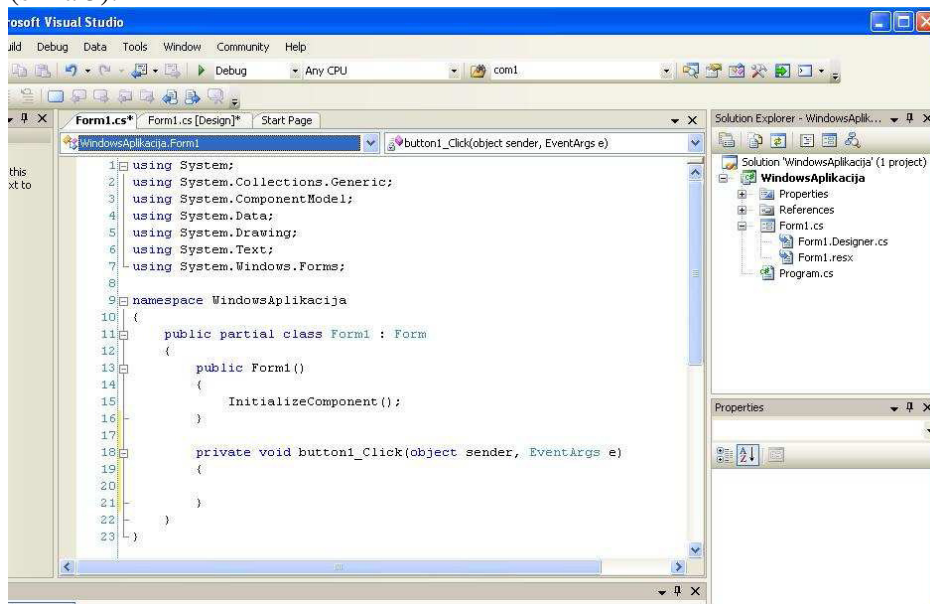
Iz Toolboxa možemo da izaberemo kontrolu i da je „dovučemo“ na glavnu formu. Ova akcija se obavlja tako što se dovede kursor miša iznad željene kontrole, pritisne levi taster miša i drži pritisnut uz istovremeno pomeranje pokazivača miša na mesto na formi. Kada se pokazivač miša pozicionira na željeno mesto otpusti se levi taster miša. Kao rezultat ove akcije koja se često naziva „dovlačenje“, „drag & drop“, itd. kontrola će biti prikazana sa nekom podrazumevanom veličinom. Ovu podrazumevanu veličinu možemo promeniti klikom na kontrolu (kontrola postaje selektovana - pojavljuju se kvadrati u uglovima i sredinama stranica) i menjanjem veličine u bilo kom od 8 mogućih smerova (slika 2) tako što se klikne na jedan od kvadratića selektovane kontrole i, uz držanje levog tastera miša pritisnutim, pomera miš u željenom pravcu i smeru. Drugi način za postavljanje kontrole je da kliknemo na kontrolu u Toolboxu a da zatim na mestu u formi na kom želimo da pozicioniramo kontrolu kliknemo i držeći levi taster miša pritisnut razvučemo tj. povećamo kontrolu koliko želimo. Otpuštanjem tastera miša kontrola se prikazuje.



Slika 4: AutoHide za Toolbox

Postavimo kontrolu **Button** (komandno dugme) na formu na jedan od prethodno opisana dva načina. Nakon toga kliknimo na kontrolu tako da ona dobije fokus (pojaviće se 8 belih kvadratića po obodima kontrole) tj. kako se to kaže selektujemo je, odnosno odaberimo je. Primetićemo da se sadržaj Menadžera osobina (Properties window) promeni (ili prozora za prikaz svojstava objekata). Naime, inicijalno on pokazuje osobine početne forme a sada osobine našeg dugmeta. Sve osobine dostupne u ovom prozoru možemo da promenimo pri čemu će one odmah biti vidljive. Promenimo osobinu **Text** u „Povecaj“ (originalna vrednost ove osobine je bila „button1“).

Osnovna namena komandnog dugmeta ili jednostavno dugmeta je izazivanje neke akcije nakon klika na njega. Ako npr. želimo da se svaki put kada kliknemo na dugme njegova veličina poveća moraćemo da napišemo odgovarajući kod koji će da obradi **dogadjaj** "klik na dugme". U tu svrhu treba dva puta brzo kliknuti na dugme. Kao rezultat prikazuje se prozor sa odgovarajućim kodom (slika 5).



Slika 5: Click metoda

Uočite funkciju

private void button1_Click(object sender, EventArgs e).

Ova funkcija će biti pozvana kao odgovor na korisnikov klik na dugme na formi. Dodajmo u telo ove funkcije kod kojim se povećavaju veličina forme i veličina dugmeta. Kontrole forma i dugme imaju osobinu **Size** koja se sastoji od osobina **Width** (širina) i **Height** (visina). Povećajmo širinu za 50 pixels sledećim kodom

```
private void button1_Click(object sender, EventArgs e)
{
    Width += 50;
    button1.Width += 50;
}
```

Width se odnosi na širinu forme a button1.Width na širinu dugmeta. Pokrenimo naš program pritiskom na F5. Rezultat je forma sa dugmetom. Svaki klik na dugme će povećati širinu i forme i dugmeta.

Kada se doda nova kontrola ona inicijalno ima svoje generičko ime kao u prethodnom slučaju button1. Sledeće dugme koje bi postavili imalo bi ime button2 itd. Ako je GUI (grafički korisnički interfejs aplikacije) bogat sa velikim brojem formi i kontrola veoma teško je snalaženje ako se zadrže generička imena. Stoga je uobičajeno da se imena kontrola promene tako da ukazuju na funkciju kontrole.

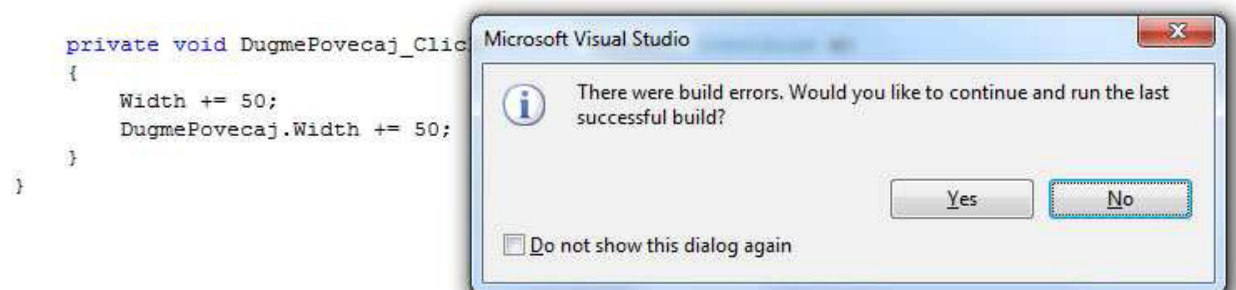
Promenimo ime glavne forme (osobina **Name**) u *GlavnaForma* a ime dugmeta u *DugmePovecaj*.

Sada kod izgleda

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace WindowsAplikacija
{
    public partial class GlavnaForma : Form
    {
        public GlavnaForma()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            Width += 50;
            DugmePovecaj.Width += 50;
        }
    }
}
```

Startovanjem programa vidimo da se ništa nije promenilo. Između ostalog i dalje na dugmetu piše button1. Treba praviti razliku između osobina **Name** i **Text**! Name predstavlja ime kontrole koje se koristi i pri pisanju koda dok je Text osobina koja se odnosi na tekst koji će biti ispisana na dugmetu. Promenimo tekst dugmeta u Povecaj a tekst forme Glavna Forma. Sada se i izgled naše aplikacije promenio.

Uočite da je ime funkcije koja opisuje događaj click na dugme ostalo nepromenjeno tj button1_Click!. Šta će se desiti ako ga promenimo? Kompajler će javiti grešku! Uočite da se ispod slova k u nazivu funkcije nalazi crvena linija.




Slika 6: Kompajler prijavljuje grešku u kodu aplikacije

```
private void DugmePovecaj_Click(object sender, EventArgs e)
{
    Width += 50;
    DugmePovecaj.Width += 50;
}
```

Ako kliknete na tu crvenu liniju dobija se dijalog za izbor akcije kao što je prikazano na sledećoj slici

```
private void DugmePovecaj_Click(object sender, EventArgs e)
{
    Width += 50;
    DugmePovecaj.Width += 50
}
```



Ako izaberemo „Rename button1_Click to DugmePovecaj_Click” ime funkcije će biti promenjeno i kompajler više neće javljati grešku.

Svaki put kada dodje do kolizije imena i okruženje prepozna tu koliziju pojaviće se podvučena linija na koju možemo da kliknemo i da vidimo šta nam okruženje nudi kao opcije za razrešenje problema.

NAPOMENA: Razvojna okruženja su praviljena sa ciljem da omoguće što jednostavniji razvoj aplikacija i uključuju mnoge funkcionalnosti (kao napred opisana) koje to i omogućavaju te ih treba obilato koristiti.

U prozoru za kod mogu se primetiti linije koje počinu sa **using** tj.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

Na ovaj način se u kod uključuju odgovarajuće biblioteke sa klasama koje one realizuju tj. odgovarajući **namespace**-ovi. Na primer, da bi mogli da koristimo Consolu za ispis poruka tj. rezultata obrade potrebno je uključiti **System**. Tako naredba za izlaz kao što smo videli ima oblik

```
Console.WriteLine("Neki izlaz"); ili pak
System.Console.WriteLine("Neki izlaz");
```

Programiranje vodjeno doganajima

Na osnovu prethodnog odeljka ove lekcije može se zaključiti da se programiranje Windows aplikacija znatno razlikuje od programiranja konzolnih aplikacija. Sintaksa i jezik su ostali isti ali se pristup promenio. Sada se u prvi plan stavlja izgled interfejsa tako da je **prva etapa** u kreiranju neke Windows aplikacije zapravo **kreiranje izgleda interfejsa** aplikacije. Nakon toga dolazi **druga etapa** gde se za svaki događaj (npr. klik levim dugmetom miša, dupli klik levim dugmetom miša, klik desnim dugmetom miša, pritisak na tab i slično) koji se vezuje za kontrole interfejsa formiranog u prvoj etapi generiše kod funkcija koje se vezuju za događaj tj. koje se izvršavaju kada se registruje događaj.

Znači programiranje se zapravo svodi na pisanje koda funkcija koje treba da se aktiviraju kada se desi neki događaj. Zato se ovaj pristup programiranju naziva i programiranje vodjeno ili upravljano događajima. Na prvi pogled reklo bi se da se sada programira "lokalno" jer programiramo funkcionalnosti koje treba da se dese kao odgovor na neki događaj (najčešće akciju korisnika

programa). Ovo je u principu mnogo jednostavniji pristup jer se globalno planiranje svodi na lokalno kreiranje funkcionalnosti.

Da li je ovakav način kreiranja aplikacije "prirodan"? Reklo bi se da jeste jer ako se posmatra korišćenje jedne aplikacije ono ne predstavlja ništa drugo do: pokretanja aplikacije i niza akcija korisnika iz kojih dolazi nekakav "odgovor" programa i tako sve dok se aplikacija ne završi. Jednom rečju korišćenje aplikacije je niz akcija korisnika koje očekuju adekvatan odziv aplikacije. Stoga se i samo kreiranje aplikacije svodi na: kreiranje vizuelnog izgleda aplikacije tj. grafičkog korisničkog interfejsa upotrebom opštepoznatih Windows kontrola i pisanja odgovarajućeg koda kojim se opisuje željeno ponašanje tj. "odgovor" aplikacije na pojedine akcije korisnika. Oba ova koraka su izuzetno važna i veoma često se oni u toku realizacije aplikacije prepliću tj. izgenerišu se neke forme sa nekim kontrolama a zatim se ispiše odgovarajući kod za evente kontrola a nakon toga isti postupak ponovi za neke nove forem i kontrole itd. tj. inkrementalno i iterativno. Dobra strana je što se od same osnovne forme pa nadalje uvek ima "neka verzija" aplikacije sa odgovarajućim korisničkim interfejsom.

Poseban akcenat se na ovaj način stavlja na projektovanje tj. dizajn interfejsa jer on određuje u krajnjoj meri kompletan način funkcionisanja cele aplikacije.

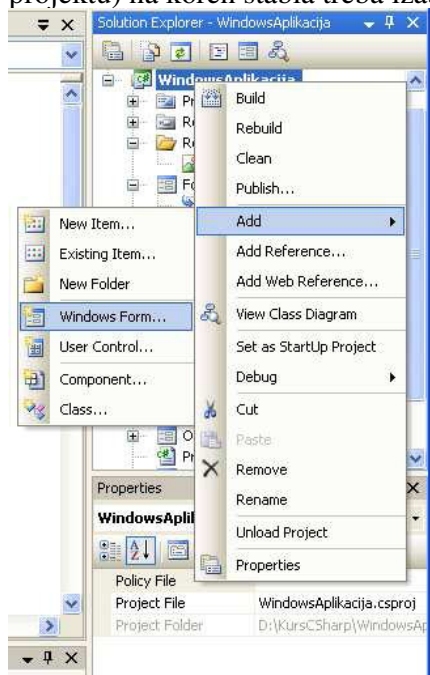
Znači, grubo se može reći da se razvoj jedne Windows aplikacije odvija takoi što se najpre isprojektuje interfejs koji obuhvata i popis svih događaja i akcija koje treba da slede te događaje.

Naon toga se u drugoj fazi piše kod za funkcije koje se pozivaju po aktiviranju događaja. Događaji se po pravilu vezuju za elemente grafičkog korisničkog interfejsa aplikacije.

Modalne i nemodalne forme

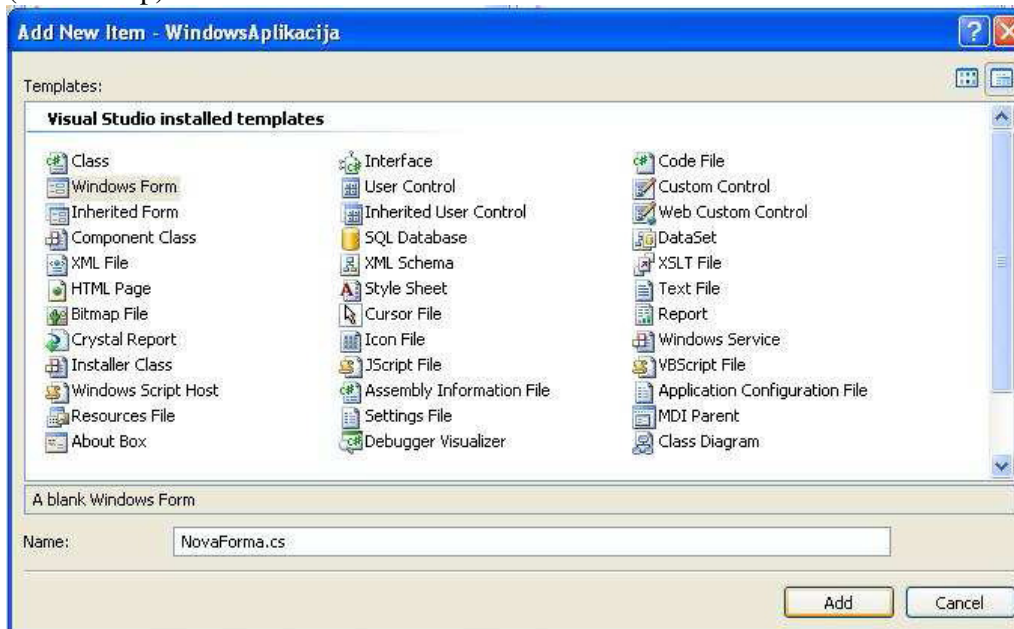
Pored osnovne tj. glavne ili ti početne forme mogu se kreirati dodatne forme koje će se prikazati kao odgovor na neku akciju korisnika. Dodajmo na osnovnu formu jedno dugme koje će nam poslužiti za prikazivanje nove forme tj. svaki klik na dugme rezultovaće u pojavljivanju nove forme.

Da bi uopšte mogla da se koristi nova forma mora najpre da se generiše. Generisanje dodatne forme se vrši na sledeći način: Desnim klikom u SolutionExploreru (prozor u kome se nalaze informacije o projektu) na koren stabla treba izabrati generisanje Windows forme kao što je to prikazano na slici 2.



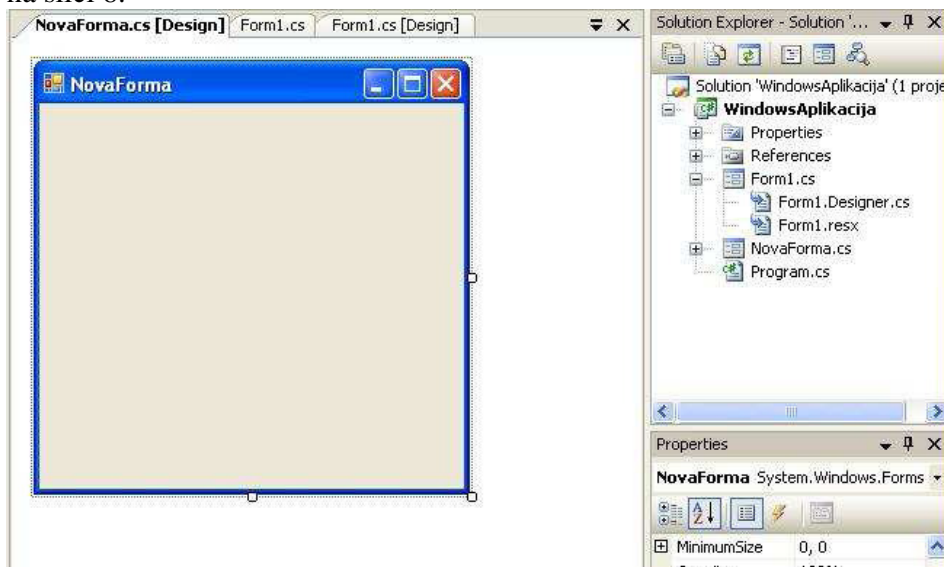
Slika 6: Dodavanje nove Forme.

Nakon toga se pojavljuje dijalog za unošenje imena forme kao što je to prikazano na slici 7. Unesimo ime NovaForma. Treba primetiti da svaku formu prati odgovarajući fajl sa kodom ekstenzijom cs (od C Sharp)



Slika 7: Dijalog za unošenje imena nove forme

Nakon unošenja imena i klika na Add dugme zatvara se dijalog, i pojavljuje se novi Tab koji se zove NovaForma.cs[Design] a u Solution Exploreru se pojavljuje novi fajl NovaForma.cs kao što se vidi na slici 8.



Slika 8: NovaForma u Solution Exploreru

Na novoj formi isto kao i na osnovnoj formi možemo dodavati nove kontrole. Za početak nećemo dodavati ni jednu kontrolu na novu formu već ćemo samo da je prikažemo kada se klikne na dugme Nova forma. Da bi to postigli treba uneti sledeći kod u telo funkcije koja obranjuje doganaj Click:


```
private void btnNovaForma_Click(object sender, EventArgs e)
{
    NovaForma frmNovaForma = new NovaForma();
    frmNovaForma.Show();
}
```

Po startovanju programa i klika na dugme "Nova forma" prikazuje se nova forma kao na slici 9.



Slika 9: Nova modalna forma.

Može se primetiti da fokus možemo prebacivati sa nove forme na osnovnu i obrnuto koliko puta hoćemo običnim klikom na odgovarajuću formu. Ovakva relacija izmenu dve forme se zove **"nemodalne forme"**.

Veoma često je u praksi potrebno drugačije ponašanje formi. Naime, nije dozvoljeno ovakvo preklapanje formi, već korisnik mora da zatvori novu formu da bi se vratio na prethodnu. Ovakva relacija izmenu formi se zove **"modalne forme"** (često je u upotrebi i termin **"Dialog forma"**). Ovakvo ponašanje je moguće postići ako za prikaz forme umesto metode **Show()** koristimo metodu **ShowDialog()**, takone bez argumenata tj.

```
private void btnNovaForma_Click(object sender, EventArgs e)
{
    NovaForma frmNovaForma = new NovaForma();
    frmNovaForma.ShowDialog();
}
```

Ako ponovo startujemo program videćemo da se ne može prebaciti fokus na osnovnu formu sve dok se ne zatvori nova forma (Zatvaranje forme se može obaviti kao i kod bilo koje druge Windows forme tj. aplikacije, klikom na krstić u gornjem desnom uglu forme. O ostalim načinima više reči će biti kasnije.). Modalni (dialog) način otvaranja forme se u praksi znatno češće koristi.

Zatvaranje forme

Otvorenu formu možemo uvek zatvoriti na standardni način klikom na dugme x u njenom gornjem desnom uglu. Menutim veoma često je potrebno programski zatvoriti (potrebno je napisati deo koda kao deo neke metode kojim se zatvara forma) formu nakon završetka odgovarajuće obrade koja je u vezi sa nekom formom tj. kontrolama na njoj. Da bi programski zatvorili formu moramo prethodno da objasnimo pokazivač **this**. **this** je promenljiva koja pokazuje na objekat u kome se nalazi – u našem slučaju na formu.

Kada bilo gde u kodu koji se nalazi u formi napišete **this** vi se zapravo referišete na sam Form objekat te su dostupne sve metode i svojstva ne samo forme, već i svih elemenata koji se nalaze na njoj.

Metoda koja zatvara formu je **Close()** bez argumenata. Kada bilo gde u kodu na formi napišemo:

```
this.Close();
```

ovo znači bezuslovno zatvaranje forme u kojoj se ovaj kod nalazi.

Kontrole

Postoji veliki broj različitih kontrola koje se mogu postaviti na formu i koje predstavljaju više manje standardne elemente bilo koje Windows aplikacije sa bogatim grafičim korisničkim interfejsom (GUI). Najčešće korišćene kontrole su:

- komandno dugme (Button)
- labela (Label)
- radio dugme (RadioButton)
- ček boks (CheckBox)
- tekst boks (TextBox)
- rič tekst boks (RichTextBox)
- list boks (ListBox)
- kombo boks (ComboBox)

U nastavku će ove kontrole biti detaljnije razmatrane.

Komandno dugme (Button)

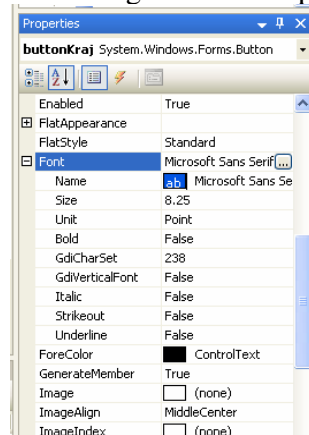
Komandno dugme (ili samo dugme) je verovatno najčešće korišćena kontrola koja je uz to i najjednostavnija i po funkciji verovatno najjasnija. Dugme omogućava korisniku da klikom miša na njega pokrene neku akciju – tj. neki segment programskog koda se izvršava.

Ključno svojstvo dugmeta je svojstvo **Text** kojim se postavlja njegov natpis. Generalno, svaka kontrola koja na sebi poseduje neki natpis ima svojstvo **Text** kao što je ranije napomenuto ovo svojstvo treba razlikovati od svojstva **Name** (preko svojstva Name se pristupa objektu u samom kodu).

Ovo svojstvo ima još jednu mogućnost: ako postavite znak **&** ispred bilo kog karaktera u **Text** svojstvu, taj karakter je podvučen i predstavlja skraćenicu sa tastature za klik akciju dugmeta.

Na primer, kada svojstvo **Text** dugmeta postavimo na **Nova &forma**, prikazuje se na dugmetu natpis **Nova forma** a skraćunica sa tastature je **Alt+f**. Znači istu akciju (prikazivanje nove forme) možemo da izazovemo i klikom na dugme kao i prečicom sa tastature. Jasno, na jednoj formi je potrebno imati jedinstvene skraćenice kako bi sve funkcionisalo kako je predviđeno tj kako bi sistem znao koju akciju da pokrene kao odgovor na unetu skraćenicu sa tastature.

Za dugme, kao i za većinu ostalih kontrola možemo postaviti font koji se koristi za ispis teksta na njemu kao i njegove atribute (podebljan(bold), ukošen(italic), podvučen(underline)). Svojstvo **Font** nam omogućava sve napred navedeno i na slici 10 je prikazano sa svim svojim atributima.



Slika 10: Osobine kontrole dugme i svojstvo Font.

Od doganaja, najčešće se koristi doganaj **Click**, koji se aktivira kada korisnik klikne na dugme.

Vežba

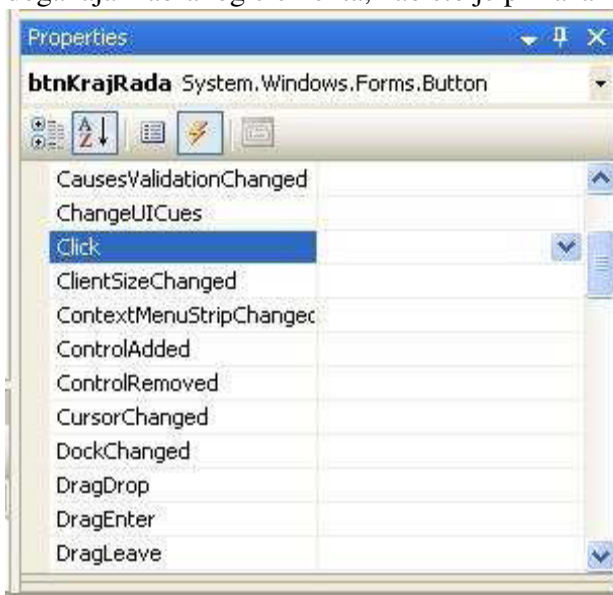
U postojeći projekat postavite novo komandno dugme na osnovnu formu.

Svojstvo **Name** postavite na **btnKrajRada**, a svojstvo **Text** na **&Kraj rada** (Slika 11).



Slika 11: Izgled forme

Kod za različite događaje kontrole (eng. control events) pišemo na sledeći način: mišem izaberite dugme i kliknite na ikonu (narandžasta munja) u Properties dijalogu koja daje listu svih raspoloživih događaja izabranog elementa, kao što je prikazano na slici 12 za kontrolu tipa Button.



Slika 12: Lista događaja (events) koje kontrola Button može da prihvati

Pronađite **Click** događaj i mišem uradite dvostruki klik na njega. Na ovaj način se otvara prozor za pisanje programskog koda koji se izvršava za izabranu akciju – **Click** u našem slučaju.

U ovoj vežbi, želimo da se klikom na dugme “Kraj rada” program završi. U telu funkcije koja obrađuje događaj klika na dugme treba upisati sledeći kod:

```
Application.Exit();
```

tj funkcija izgleda

```
private void btnKrajRada_Click(object sender, EventArgs e)
{
```

```
Application.Exit();  
}
```

Koristimo **Application** objekat koji predstavlja pokazivač na aplikaciju i njegov metod **Exit** koji bezuslovno završava rad aplikacije i oslobađa sve resurse koje ona koristi.

Pokrenite aplikaciju (**F5** funkcijski taster) i kliknite na dugme. Isti efekat (zatvaranje aplikacije) može se postići i skraćenicom sa tastature **Alt+K**, koja je postavljena pomoću **Text** svojstva dugmeta dodatkom znaka & ispred slova K.

Pored događaja Click dugme „prepozna“ veliki broj drugih događaja koje treba isprobati u samostalnom radu. Većina njih je orijentisana na grafički izgled dugmeta, pa posebno treba obratiti pažnju na svojstvo **Image** i njemu slična.

Labela

Kontrola Label (labela ili natpis) omogućuje prikaz teksta na formi koji može biti samostalan, ali se obično vezuje za neku drugu kontrolu i na taj način opisuje njeno značenje. Najčešće se postavlja uz **TextBox** kontrolu koja će biti kasnije opisana. Ključno svojstvo kontrole je **Text** koje takone poseduje mogućnost dodele skraćenice sa tastature kao što smo radili u slučaju kontrole Button (dugme). Uobičajeno je da se labeli postave svojstva koja se odnose na izgled i veličinu fonta u skladu sa veličinom forme na kojoj se nalazi, odnosno broju kontrola na formi (slika 13).



Slika 13: Podešavanje fonta

Postavimo labelu na našu drugu formu a njena svojstva na sledeće vrednosti:

- Text - Ovo je nova forma koja može biti modalna i nemodalna
- Font Name – Arial Black
- Font Size – 20
- ForeColor – Red

Ako pokrenemo našu aplikaciju i klikom na dugme "Nova forma" kao rezultat dobićemo formu sa ispisanim tekstom "ovo je nova forma" u crvenoj boji kao što je prikazano na slici 14.



Slika 14: Izmenjen font labela

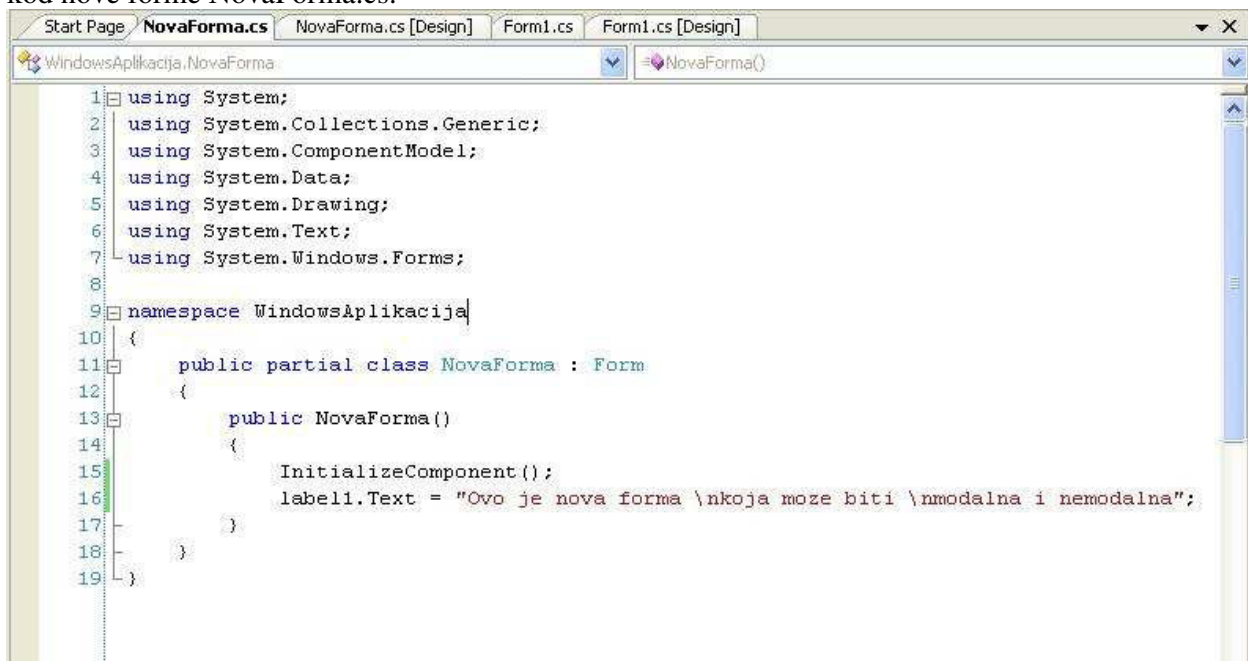
Na prvi pogled ovo je čudno jer smo osobinu text postavili na “Ovo je nova forma koja može biti modalna i nemodalna”. Gde se izgubio deo teksta? Tekst se nije izgubio već se ne prikazuje kompletno zbog veličine forme. Ako povećamo širinu ove forme (U desnom donjem uglu forme se nalazi 6 tačica u obliku trougla. Dovedi kursor miša u ovu oblast što će rezultovati promenom kursora miša u dvostranu strelicu, kliknuti i razvući prozor udesno sve dok se ne prikaže ceo tekst labela (slika 15).



Slika 15: Izmena veličine forme

Očigledno je da ovakvo rešenje nije odgovarajuće i da je bolje rešenje gde će tekst labela biti ispisan u dva ili više redova. Međutim, višelinjski text nije moguće obezbediti u fazi dizajna tj. kroz setovanje osobine Text. Ispisavanje labela u više linija se može izvesti postavljanjem osobine Text posmatrane labela u toku izvršenja programa tj. u runtime.

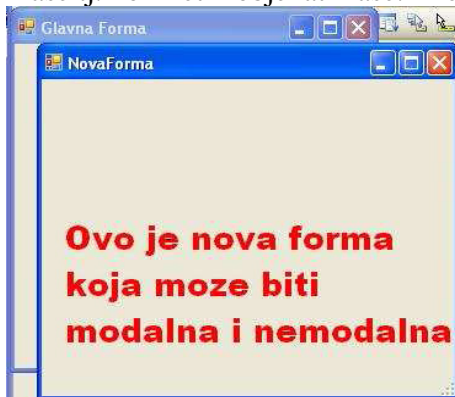
Kako ćemo to da uradimo? Unesimo odgovarajući kod u telo konstruktora nove forme kojim ćemo postaviti osobinu Text labela označene kao label1 pri čemu svaka linija osim poslednje treba da se završava sa \n što predstavlja oznaku za novi red (slika 16). Uočite da je naziv fajla u kome se nalazi kod nove forme NovaForma.cs.



Slika 16: Postavljanje Text osobine labela u konstrukturu forme.

Ako startujemo program i aktiviramo novu formu videćemo da se sada tekst labela ispisuje u 3 linije (slika 17). Šta se desilo? Odmah po kliku na dugme "Nova forma" pokrenuo se konstruktor koji je napravio objekat tipa Form i na njemu objekat label1 tipa label kome smo mi u konstrukturu postavili osobinu Text tako da se prikazuje u 3 linije.

Podsetimo se: Konstruktor je specijalna metoda klase koja se izvršava kada se kreira instanca neke klase tj. konkretni objekat klase. Ime konstruktora je isto kao i ime klase.



Slika 17: Labela sa tekстом u 3 linije.

TextBox

Linija za unos teksta (TextBox) je kontrola koja omogućava korisniku da unese podatke koji dalje mogu da se koriste u aplikaciji. Ovo je relativno složena kontrola koja se veoma često koristi upravo zbog činjenice da omogućuje unos podatka od strane korisnika. Unos teksta može biti ili u jednoj liniji ili u više linija. I TextBox kao i prethodno opisane kontrole ima veliki broj osobina te ćemo spomenuti samo najinteresantnije. Na primer, može se ograničiti mogući broj unetih karaktera, postaviti font koji se koristi itd.

Kao i kod kontrole Label i ovde je ključno svojstvo svojstvo **Text** koje se može i postavljati i "čitati" za vreme izvršavanja aplikacije i na taj način se vrši obrada unosa korisnika.

Svojstvo **Multiline** (koje ne postoji kod kontrole Label) označava da li je moguća jedna (**False** (netačno)) ili više linija teksta (**True** (tačno)). Ako je svojstvo **Multiline** postavljeno na **True**, obično se postavlja i svojstvo **ScrollBars** koje ovakvom višelinijskom tekstu dodaje mogućnost pomeranja po vertikali i horizontali u slučaju da je tekst dugačak a veličina kontrole nije dovoljna da bi se istovremeno prikazao kompletan tekst. ScrollBars se može setovati tako da se prikazuje vertikalni (**Vertical**), horizontalni (**Horizontal**), oba (**Both**) ili pak da se ne prikazuje ni jedan (**None**) bar.

Ako je potrebno ograničiti broj karaktera koje korisnik može uneti koristi se svojstvo **MaxLength** koje definiše maksimalni broj karaktera koji korisnik može uneti.

Ovo svojstvo je inicijalno postavljeno na 32767 (maksimalan ceo broj ako se koristi 16 bita za memorisanje celog broja tj. $(2^{16}-1)$), što je u praksi obično previše – uvek će biti znatno manje.

Kontrola TextBox može da prihvati veći broj doganaja. Jedan od najčešće korišćenih je događaj **TextChanged**. On se automatski pokreće svaki put kada korisnik unese ili obriše karakter u kontroli. Može se iskoristiti za proveru unetog sadržaja, on line pretragu itd.

Vežba

Napraviti formu u kojoj korisnik može da unese svoje ime i prezime.

Na formular postavite dve kontrole tipa Label, dve kontrole tipa TextBox i jednu tipa Button i postavite im sledeće osobine:

prva kontrola tipa **Label** :

Name: lblIme i **Text:** &Ime

prva kontrola tipa **TextBox** :

Name: txtIme i **MaxLength:** 15

druga kontrola tipa **Label** :

Name: lblPrezime i **Text:** &Prezime

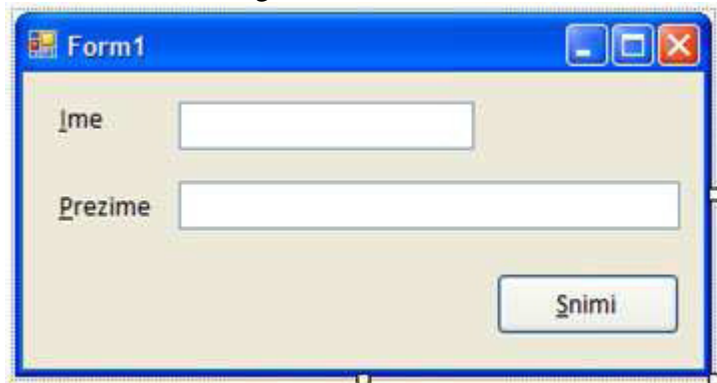
druga kontrola tipa **TextBox** :

Name: txtPrezime i **MaxLength:** 25

kontrola tipa dugme :

Name: btnSnimi i **Text:** &Snimi

Formular treba da izgleda kao na slici 18.



Slika 18: Forma za unos podataka o korisniku

U kompletnoj realnoj aplikaciji klikom na dugme **btnSnimi** bi unete podatke snimili u bazu podataka, ali kako se nismo upoznali sa bazama podataka uneti podaci će biti ispisani u okviru dijaloga za poruke (MessageBox). Da bi to postigli treba u funkciju koja opisuje događaj klik na dugme snimi uneti sledeći kod.

```
MessageBox.Show (txtIme.Text);
```

```
MessageBox.Show (txtPrezime.Text);
```

Objekat **MessageBox** predstavlja dijalog za poruke. Posедуje metodu **Show** koja ima jedan parametar – tekst koji treba prikazati. U ovom primeru prikazujemo sadržaj (uneti tekst) u kontrolama **txtIme** i **txtPrezime**.

Vidimo i sintaksu pristupa svojstvu bilo koje kontrole:

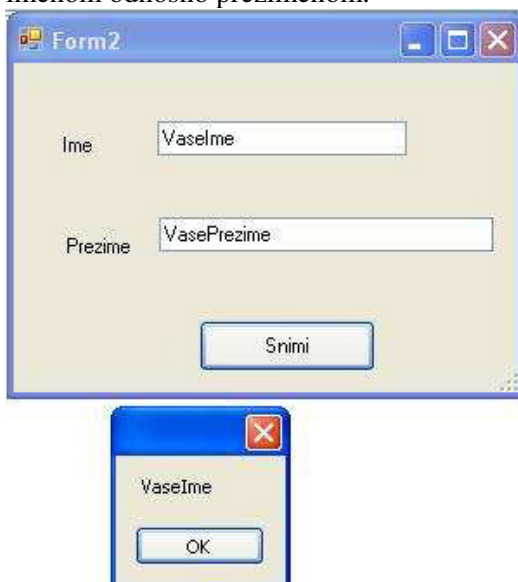
ImeKontrole.SvojstvoKontrole.

Na ovaj način možemo čitati ali i postavljati vrednosti svojstava kontrola.

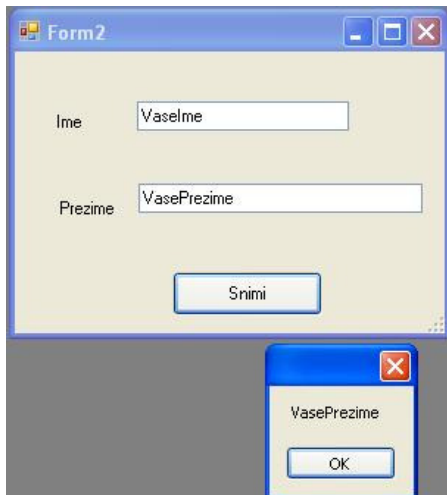
Napomena:

Postoji više varijanti metode **Show MessageBox** objekta, a ovde koristimo najjednostavniju.

Isprobajte aplikaciju tako što ćete uneti svoje ime i prezime u odgovarajuća polja za unos teksta i kliknete na dugme **Snimi**. Ako ste sve upisali kako treba, dobićete sledeća dva uzastopna dijaloga za poruke sa imenom odnosno prezimenom.



Slika 19: MessageBox sa unetim imenom.



Slika 20: MessageBox sa unetim prezimenom

Analiza vežbe

Na obe Label kontrole postavljene su odgovarajuće skraćenice i to kombinacija **Alt+I** treba da kurzor postavi na **TextBox** za unos imena, a **Alt+P** na **TextBox** za unos prezimena. **Alt+S** će pokrenuti Click doganaj dugmeta. Proverite ovo i uverite se da se naša forma ponaša na opisani način! Kako je ovo postignuto?

U svakoj Windows formi postoji pojam **TabOrder**. On definiše redosled kojim se vrši navigacija kada korisnik klikne na Tab ili Shift+Tab taster na tastaturi. Veoma je bitno postaviti logičan redosled kretanja po formularu na način na koji će korisnik najčešće da koristi aplikaciju. Svaka kontrola poseduje numeričko svojstvo **TabIndex** koje počinje od broja 0 pa na dalje. U našem primeru prva labela ima **TabIndex** svojstvo postavljeno na 0, prvi **TextBox** na 1, i tako redom. Drugi pojam koji treba znati je **Focus**. On predstavlja situaciju kada kontrola postaje aktivna. Na primer kada je kurzor postavljen u kontroli za unos prezimena, kažemo da je **TextBox txtPrezime** dobio fokus.

Napomena: Ne mogu sve kontrole da imaju fokus. Očigledno, **Label** kontrola ne može da dobije fokus.

Šta se dogana kada korisnik na tastaturi klikne **Alt+P**?

Labela lblPrezime ne može da dobije fokus, te ga automatski prenosi na prvu sledeću kontrolu koja to može – u našem primeru to je **TextBox txtPrezime**.

Prva sledeća kontrola se računa upravo po svojstvu **TabIndex**. Svaka kontrola koju postavimo na formu automatski dobija svoj **TabIndex** u skladu sa redosledom postavljanja.

Često se menutim u praksi postavlja potreba za ručnom izmenom redosleda kontrola jer kreiranje sadržaja forme nije uvek u skladu sa načinom korišćenja. To možemo uraditi tako što ćemo svakoj kontroli pojedinačno postaviti **TabIndex** svojstvo, što može biti poprilično zamoran posao podložan greškama naročito u slučajevima kada se na formi nalazi veći broj kontrola.

Drugi način za postavljanje Svojstva **TabIndex** je zgodniji i manje sklon greškama.

Zaustavite program i kliknite na formu za čije kontrole želimo da

uredimo **TabIndex** svojstvo. Iz menija **View** izaberite stavku **Tab Order**. Na ovaj način je prikazano **TabIndex** svojstvo svake kontrole na formularu (slika 21).



Slika 21: Redosled tabova kontrola na formi

NAPOMENA

Zašto je redosled po kome kontrole dobijaju fokus izuzetno važan? Važno je jer se i pored bogatog GUIa neke aplikacije u kome je moguće klikom miša prebacivati fokus sa jedne na bilo koju drugu kontrolu u mnogim primenama gde korisnik treba da popunjava neka polja tj. unosi neki tekst, pokazalo da je mnogo brži rad sa aplikacijom ako korisnik ne mora da pomera ruke sa tastature već su fokusi porenani u skladu sa najčešćim redosledom tj. načinom korišćenja aplikacije. Ovo dolazi do izražaja pogotovo kada korisnik dobro poznaje aplikaciju i sam proces koji podržava aplikacija. Iz istog razloga se uvode i skraćenice za pojedine kontrole, jer dobro obučen korisnik veoma efikasno koristi tastaturu pa je korišćenje miša svedeno na minimum. Da bi mu se omogućio ovakav rad potrebno je obratiti posebnu pažnju na redosled fokusa i korišćene skraćenice.

CheckBox

Ova kontrola omogućava korisniku da označi (markira, čekira) ili demarkira neku opciju.

Ako na formi postoji više CheckBox kontrola one su međusobno nezavisne tj. mogu se proizvoljno markirati ili demarkirati.

Ključno svojstvo ove kontrole je svojstvo **Checked** koje može imati dve vrednosti - **True** (tačno tj. kontrola je označena) i **False** (netačno tj. kontrola nije označena). Svojstvo Checked možemo postavljati i čitati kako u fazi dizajniranja tako i u toku izvršavanja aplikacije. U kodu se najčešće koristi kroz **IF** uslov koji vrši ispitivanje da li je CheckBox markiran ili ne i u zavisnosti od toga se vrši dalje grananje koda.

Opis značenja CheckBox kontrole se definiše kroz njeno **Text** svojstvo. Nije potrebno postavljati zasebnu Label kontrolu – ona je integrisana u CheckBox.

Vežba

Na formu za unos imena i prezimena dodajte jednu **CheckBox** kontrolu i svojstvo **Text** postavite na „Muškarac” pri čemu se znaci navoda ne unose (slika 12).



Slika 12: Forma sa CheckBox kontrolom nazvanom Muškarac

Prilikom snimanja pored imena i prezimena u realnoj aplikaciji bi trebali da upišemo i pol osobe. Da bi znali pol potrebno je da proverimo da li je CheckBox kontrola nazvana Muškarac markirana ili ne. U našem primeru u cilju provere pojaviće se MessageBox sa oznakom pola tj. ako je markirana kontrola biće ispisano "Muškarac" odnosno "Žena" ako nije markirana.

Da bi ovo uradili iskoristićemo if naredbu kojom ćemo da ispitamo da li je kontrola markirana ili ne. Kod ćemo da dodamo u telo funkcije koja opisuje događaj Click na dugme Snimi tako da će ona sada da izgleda :

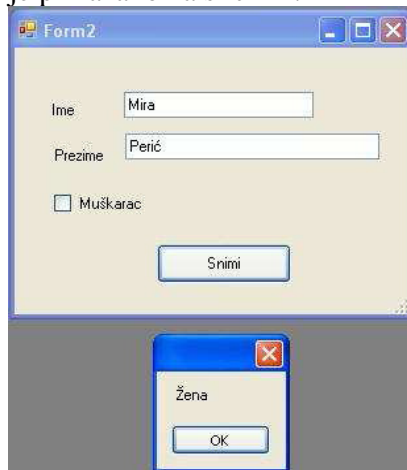
```
private void button1_Click(object sender, EventArgs e)
{
    if (checkBox1.Checked == true)
        MessageBox.Show("Muškarac");
    else
        MessageBox.Show("Žena");
    MessageBox.Show(txtIme.Text);
    MessageBox.Show(txtPrezime.Text);
}
```

Pokrenimo aplikaciju i markirajmo checkBox kontrolu Muškarac a zatim kliknimo na dugme Save. Kao rezultat dobija se messageBox sa tekstem "Muškarac" (Slika 13).



Slika 13: Poruka da je markiran CheckBox Muškarac

U slučaju da nije markirana CheckBox kontrola dobio bi se MessageBox sa natpisom "Žena" kao što je prikazano na slici 14.

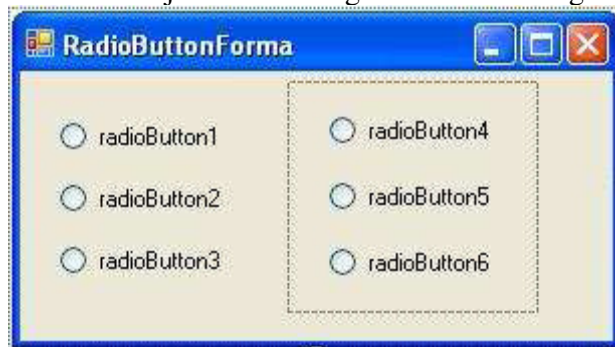


Slika 14: Poruka da nije markiran CheckBox Muškarac

Radio dugme (Radio Button)

Česte su situacije kada korisnik neke aplikacije može da izabere samo jednu od više ponudjenih opcija. Takvi slučajevi se rešavaju upotrebom kontrole koja se zove **Radio Button** kontrola. Ova kontrola je slična **CheckBox** kontroli s tom razlikom da imamo više mogućnosti za markiranje pri čemu možemo da markiramo samo jednu. Grubo rečeno ova kontrola je grupa checkBoxova pri čemu samo jedan može biti markiran, tj. kada markirate jedan, prethodni markirani postane demarkiran. Normalno može biti situacija kada nijedan nije markiran. Iz ovakvog ponašanja je i proisteklo ime ove kontrole "radio dugme", po analogiji sa starim radio aparatima gde se preko mehaničkih dugmadi birao frekventni opseg tj. pritiskom na bilo koje dugme, predhodno uključeno se isključuje tj. samo jedno može biti upaljeno.

Pod grupom radio dugmića se smatraju svi koji su postavljeni na isti **kontejner** (nosač) kontrola. Očigledni kontejner kontrola je forma, ali ih ima još nekoliko na raspolaganju. U našu aplikaciju dodajmo novu formu i nazovimo je **RadioButtonForma**. Takodje, na osnovnu formu dodajmo dugme **btnRadioGroup** koje će služiti za prikaz ove nove forme. Na **RadioButtonFormu** postavimo tri **Radio Button** kontrole. One su na istom kontejneru – formi, te su povezane i međusobno se isključuju. Zatim na formu postavimo kontrolu **Panel**(nova površina). Ona predstavlja drugi kontejner na koji ćemo staviti još tri radio dugmeta. Konačni izgled forme **RadioGroupForma** je prikazan na slici 15.



Slika 15: RadioButton kontrola

Ako startujemo aplikaciju i prikažemo formu **RadioButonForma** primetićemo da može da se markira samo jedno radio dugme u svakoj grupi tj. jedno od prvih tri za koje je kontejner forma

RadioButtonForma odnosno jedno od druga tri kojima je kontejner kontrola tipa **Panel**. U praksi se **Panel** kontrola često koristi za grupisanje Radio dugmadi.

Bitna svojstva **RadioButton** kontrole su identična kao kod **CheckBox** kontrole tj. **Checked** i **Text** sa istim vrednostima i funkcijom. Korišćenje je takone identično kao kod **CheckBox** kontrole. U kodu se proveravaju statusi radio kontrola i zavisno od toga se vrši grananje u aplikaciji.

ListBox

U mnogim aplikacijama se vrši izbor jedne od više mogućih stavki iz neke liste stavki. Kontrola koja pruža ovu mogućnost se naziva **List Box** kontrola. Ovo je složenija kontrola, koja se sastoji od više stavki. Na primer, može predstavljati listu predmeta u školi. Korisnik može izabrati predmet i na primer upisati ocene i slično. Liste se mogu "puniti" za vreme dizajna, ali je ipak najčešći slučaj punjenje u vreme izvršavanja i to iz neke baze podataka. Za sada ćemo je napuniti pomoću jednostavne petlje iz koda. Ako koristimo objektno orijentisanu terminologiju, možemo reći da je lista kolekcija stavki tipa **Object**. **Kolekcija** označava skup objekata istog tipa. Svaki član kolekcije ima svoj jedinstveni **indeks** koji počinje od broja 0 do ukupnog broja stavki u kolekciji umanjeno za jedan. Bitno je razumeti da kolekcija ima svoja specifična svojstva i metode, ali takone i svaki objekat u kolekciji. Na primer svaka kolekcija ima svojstvo **Count** koje je samo za čitanje i daje broj objekata u kolekciji. Kolekcije imaju metode kojima se može dodavati nov objekat u kolekciju (**Add**), obrisati pojedinačni objekat (**RemoveAt**) ili obrisati sve članove kolekcije (**Clear**). Svojstva i metode pojedinačnih objekata kolekcije zavise od tipa objekata.

Ako postoji kolekcija pod nazivom "**MojaKolekcija**", metodima i svojstvima te kolekcije pristupamo na sledeći način:

```
MojaKolekcija. Add (...)
```

```
MojeKolekcija.Clear ()
```

i slično

Pojedinačnim objektima kolekcije se pristupa preko indeksa objekta:

```
MojaKolekcija[5].Text
```

Indeksi objekata – članova kolekcije idu od nula do $\text{MojaKolekcija.Count} - 1$

U **List Box** kontroli ova kolekcija se zove **Items** i sastoji se od članova pod nazivom **Item** (stavka, element).

Sve kolekcije .NET Framework-a poštuju istu nomenklaturu: kolekcija je imenica u množini a svaki član kolekcije je ista imenica u jednini. Na primer kolekcija **Tables** se sastoji od objekata tipa **Table**.

Na glavnu formu dodajte novo dugme i imenujte ga sa **btnListBox** a **Text** osobinu postavite na

ListBox. Kreirajte zatim novu formu i imenujte je sa **ListBoxForm**. Na formu postavite **List Box** i

jednu **Button** kontrolu sa **Text** osobinom „Dodaj u listu”. Ostavićemo inicijalne nazive ovih kontrola

listBox1 i **button1**. **ListBoxForm** forma se prikazuje kao rezultat dogadjaja **Click** na dugme

btnListBox.

Želimo da klikom na dugme napunimo listu brojevima od 1 do 10. Koristimo metodu **Add** kolekcije **Items** gde je ulazni parametar generički tip **Object** što znači da možemo "podmetnuti" promenljive numeričkog ili string tipa – svedeno, sve su interno nasleđene od istog **Object** tipa. Duplim klikom na dugme otvorite prozor za pisanje kodai unesite sledeći kod:

```
private void button1_Click(object sender, EventArgs e)
{
    int i;
    for (i = 1; i <= 5; i++)
    {
        listBox1.Items.Add(i);
    }
}
```

Pokrenite primer i testirajte ga. Primetićete da svaki put kada pritisnete dugme dodajete još 5 stavki u listu. Nakon 3 klika na dugme "Dodaj u listu" lista će izgledati kao što je to prikazano naslici 16.



Slika 16: ListBox kontrola nakon 3 klika na dugme "Dodaj u listu"

Ako želimo da pre dodavanja obrišemo sve stavke (ako ih ima), dodaćemo još jednu liniju koda pre petlje koja briše sadržaj kolekcije **Items**:

```
private void button1_Click(object sender, EventArgs e)
{
    int i;
    listBox1.Items.Clear();
    for (i = 1; i <= 5; i++)
    {
        listBox1.Items.Add(i);
    }
}
```

Sada, pre punjenja liste uvek brišemo sve stavke.

Ako želimo da pročitamo vrednost izabrane stavke iz liste, postoje dva načina:

- jednostavno pročitamo svojstvo **Text**: **listBox1.Text**
- pomoću svojstva **SelectedItem** koja vraća **Object** prezentaciju izabrane stavke: **listBox1.SelectedItem** (uvek treba proveriti da li je **SelectedItem** različit od **null** da bi se izbegle eventualne greške!)

Brisanje pojedinačne stavke iz liste se vrši metodom **RemoveAt** kolekcije **Items**. Na primer ako želimo da obrišemo treću stavku u listi:

```
listBox1.Items.RemoveAt (2);
```

Napomena: Obratite pažnju da je indeks treće stavke = 2 jer se indeksi broje od nule. Isti tako **removeAt** ne može da se zove iz petlje.

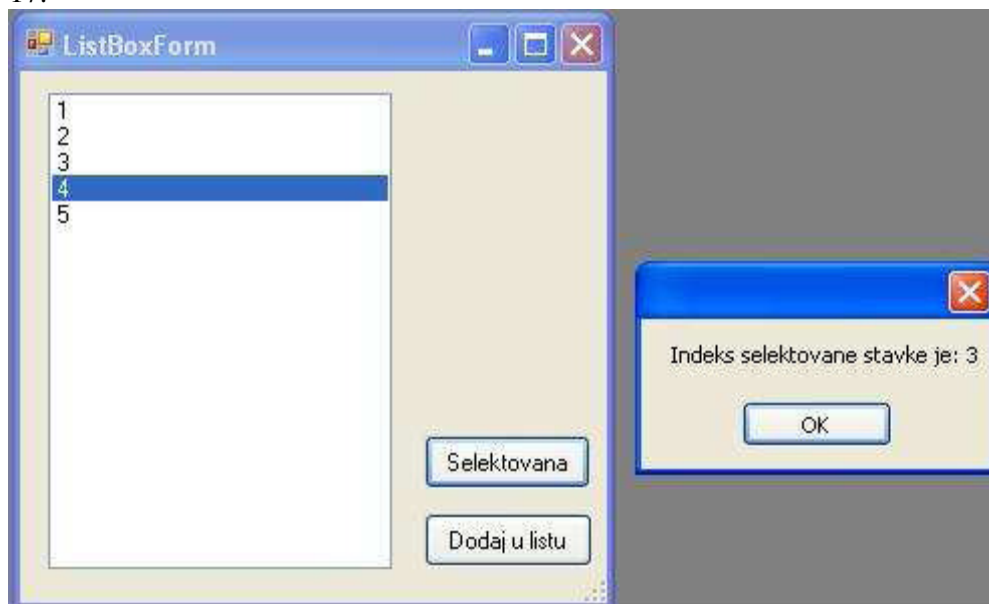
Svojstvo **SelectedIndex** pokazuje indeks izabrane stavke iz liste. U pitanju je tip **int**, a ako ni jedna stavka nije izabrana vrednost **SelectedIndex** svojstva je jednaka **-1**. Ovo se koristi kao indikacija da korisnik nije izabrao ni jednu stavku iz liste. **SelectedIndex** svojstvo je predvineno i za čitanje i za pisanje, tako da ako iz koda želimo da izaberemo na primer četvrtu stavku u listi:

```
listBox1.SelectedIndex = 3;
```

Dodajmo još jedno dugme na formu **ListBoxForm** i za događaj **Click** za ovo dugme dodajmo kod koji će da ispiše u **MessageBoxu** indeks selektovane stavke u **ListBox** kontroli (prethodno je potrebno napuniti listu klikom na dugme „Dodaj u listu”) tj

```
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("Indeks selektovane stavke je: " +
        listBox1.SelectedIndex);
}
```

Ako selektujemo stavku 4 prikazaće se MessageBox sa indeksom 3 kao što je to prikazano na slici 17.



Slika 17: Selektovana četvrta stavka

ComboBox

Kao što smo videli kontrola **TextBox** služi za unos nekog teksta. Sa druge strane kontrola **ListBox** kontrola služi za izbor stavke iz ponudjene liste stavki. Izbor stavke iz ponudjene liste možemo da izvršimo i uz pomoć kontrole koja se zove **ComboBox**. Ali isto tako ComboBox omogućuje i da unesemo tekst tj. ova kontrola u neku ruku predstavlja kombinaciju kontrola TextBox i ListBox. Postavljanje i očitavanje unetog teksta može se vršiti pomoću osobine **Text** kao kod TextBox kontrole. Generisanje liste stavki i manipulacija sa stavkama u listi tj. izbor stavke može se obavljati na isti način kao i kod kontrole ListBox.

Razmotrimo sledeći primer:

Izgenerišimo novu Windows aplikaciju tako što ćemo na glavnu formu da postavimo jedno dugme sa natpisom „Prikazi“, jedan ComboBox i jedan TextBox (slika 1).



Slika 1: Aplikacija sa ComboBox kontrolom

U listu stavki ComboBoxa dodajmo sledeća imena Misa, Pera, Laza i Goran.

Događajem "klik na dugme" treba omogućiti da se sadržaj ComboBoxa prikaže u textBoxu.

Ovu funkcionalnost ćete dobiti sledećim kodom:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            comboBox1.Items.Add("Misa");
            comboBox1.Items.Add("Pera");
            comboBox1.Items.Add("Laza");
            comboBox1.Items.Add("Goran");
        }
        private void button1_Click(object sender, EventArgs e)
        {
            textBox1.Text = comboBox1.Text;
        }
    }
}
```

Startujte program i proverite šta se dešava kada unesete neki tekst i pritisnete dugme Prikaži odnosno kada izaberete neku stavku iz liste u ComboBoxu i zatim opet pritisne dugmePrikaži. U oba slučaja u textBoxu

Broj događaja koje podržava kontrola ComboBox je poduža te je dalje nećemo objašnjavati.

Meniji

Gotovo svaka Windows aplikacija ima **meni** kao jedan od načina za interakciju sa korisnikom tj. kao način da korisnik aktivira pojedine funkcionalnosti aplikacije. Prednosti menija su mogućnost hijerarhijske organizacije koja odgovara konkretnoj aplikaciji uz minimalno zauzeće prostora. Meni se obično nalazi pri vrhu forme, ali to ne mora biti uvek slučaj. U pojedinim aplikacijama (ako za to zaista postoji opravdanost, jer je uobičajeni položaj pri vrhu te će to najviše odgovarati korisniku aplikacije) se meni može postaviti na dno ekrana, pa čak i po vertikali ako je to iz nekog razloga najbolji položaj. Dodatnu snagu menija predstavlja mogućnost dinamičkog dodavanja ili sakrivanja stavki čime meni prati korisnika zavisno od akcija koje se događaju u aplikaciji. Sakrivanjem stavki se onemogućuje korisniku aplikacije da proba da aktivira opciju koja nije podržana za dato stanje programa. Neaktivne opcije u meniju se razlikuju od aktivnih po tome što im je naziv ispisan svetlijom bojom (svetlo sivo). O tome da li je opcija aktivna ili neaktivna vodi računa programer tj. ne postoji automatski mehanizam koji to radi (znači, sve treba da se realizuje u kodu).

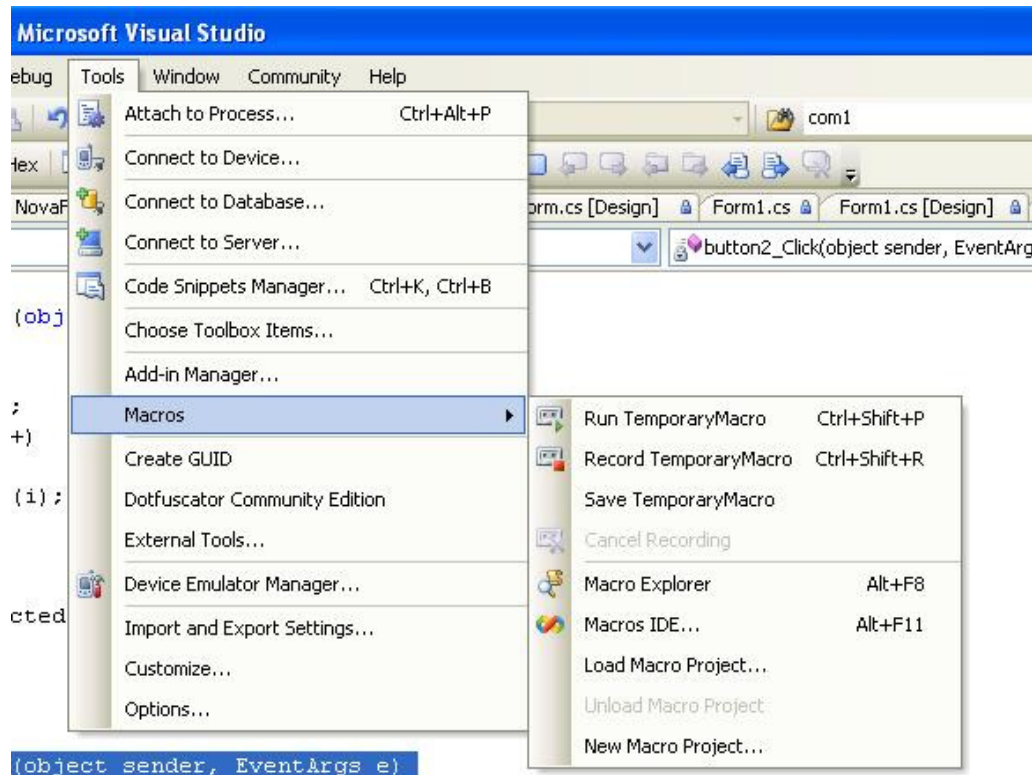
NAPOMENA: Pitanje dizajna GUIja i to pre svega opcija koje se pojavljuju u menijima u pojedinim fazama korišćenja aplikacije je izuzetno važno. Izbor da li će se neaktivna stavka u meniju prikazivati kao stavka u stanju "disable" ili se pak uopšte neće videti je jedna od odluka

koju treba da donese projektant softvera. Ako se stavka ne vidi onda su meniji kraći i pregledniji a opet sa druge strane u nekim menijim aje dobro da se ipak vidi šta sve od opcija tj. stavki postoji kao mogućnost ali neke trenutno nisu aktivne.

Ako GUI nije dobro odrađen i nisu konzistentno ispraćene sve situacije (scenarija korišćenja) u kojima pojedine stavke menija treba da budu aktivne/neaktivne veoma često će dolaziti do neispravno grada tj. do "pucanja" aplikacije ili zbunjujućih ponašanja jer će korisnik zahtevati neku akciju koja nije logična za dato stanje aplikacije a menijem mu takva akcija nije uskraćena. Stoga se ovom problemu u projektovanju softvera posvećuje dosta pažnje i to ne samo situacijama kada neka stavka menija treba da bude aktivna/neaktivna već i kako se projektuje skup komandi koji u pozadini zapravo predstavlja niz funkcija i njihovih poziva. Dodatno se komplikuje situacija ako GUI čine i Toolbarovi, kontekstni meniji, skraćenice itd. Vrhunac komplikacije u realizaciji GUIja i sistema komandi predstavlja realizacija mogućnosti tipa UNDO, REDO i CANCEL.

U Visual Studiju kreiranje menija je relativno jednostavno. Prilikom kreiranja menija razlikujemo horizontalnu liniju i vertikalne delove menija koji se prikazuju kada korisnik klikne na stavku u horizontalnoj liniji. Vertikalna linija može imati stavke na više nivoa hijerarhije.

Na slici 18 se vidi **Tools** meni i njegov podmeni **Macros** iz radnog okruženja Visual Studio.



Slika 18: Meni **Tools** i podmeni **Macros** u Visual Studio 2005 okruženju.

Osim teksta stavka menija može imati sličicu (eng. Icon) i definisanu skraćenicu sa tastature (eng. Shortcut) kao što to ima na primer stavka menija **“Attach to Process...”**. Njena skraćenica je **Ctrl+Alt+P**, što znači da se ova opcija može aktivirati istovremenim pritiskom na ove tastere.

Napomena: Kombinacija tastera može da se bira proizvoljno za skraćenice pri čemu treba da se vodi računa o preklapanju i konzistentnosti tj. ako u više menija u hijerarhiji menija imamo istu opciju onda bi i skraćenica trebala da bude ista.

Kreiranje menia

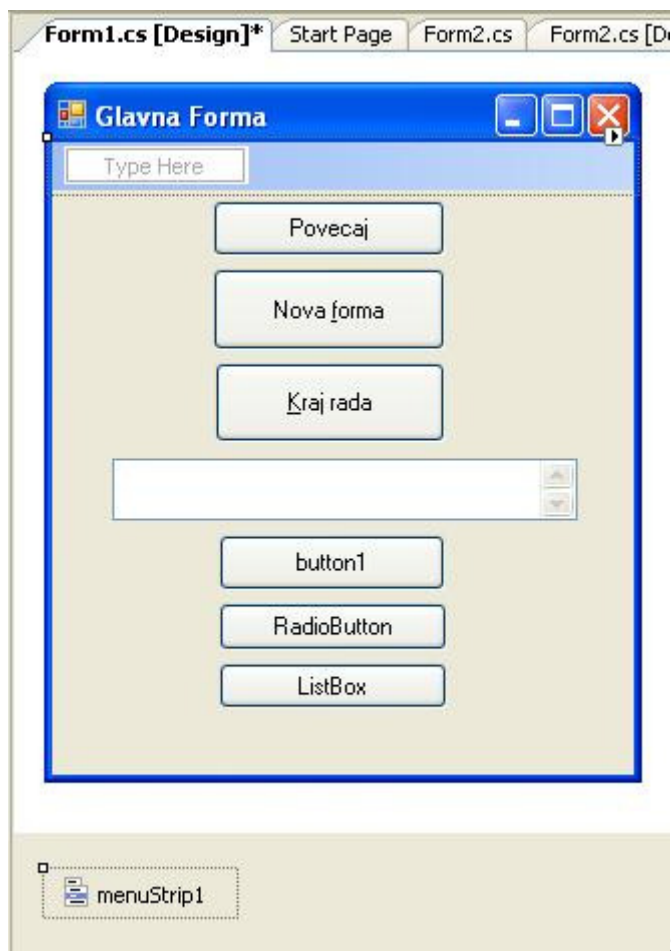
U prethodnim primerima kreiranje svih novih formi je izvršeno tako što smo dodavali novo dugme i pojavljivanje neke forme vezivali za događaj **Click** na odgovarajuće dugme. Kako će do kraja kursa broj formi da raste nije praktično da dodajemo nove dugmiće za pojavu svake nove forme jer ćemo morati da povećavamo veličinu Glavne forme. Postignimo istu funkcionalnost na mnogo elegantniji način tako što ćemo na našu Glavnu formu postaviti kontrolu **Menu**. Ova kontrola se postavlja na formu na sledeći način:

U paleti sa alatkama (eng. Toolbox) pronađite sekciju **„Menus & Toolbars“**. U okviru nje pronađite kontrolu **„MenuStrip“** koja daje potrebne funkcionalnosti menija na formi (slika 19).



Slika 19: Sekcija Menus & Toolbars i kontrola MenuStrip

Prevucite ovu kontrolu i pustite je na formu - nije važno gde, jer se meni automatski pozicionira na vrh forme, a sama kontrola u specijalnom prozoru ispod forme kao što je to prikazano na slici 20.



Slika 20: Pozicija Menija na formi.

Ako želite da kreirate meni, prvo morate da kliknete na kontrolu na dnu forme koja je dobila inicijalno ime „**menuStrip1**“. Tada se u dizajneru prikazuje meni na vrhu forme. Zapazite tekst „**Type Here**“ koji dovoljno govori sam za sebe tj. očekuje se unos nekog teksta. Napravićemo deo menija koji se odnosi na sve forme koje su kreirane iz Glavne forme. Tamo gde piše „Type Here“ unesite „**Forme**“ (bez znakova navoda).

Čim počnete da kucate odmah će se ispod i desno od ove stavke otvoriti mogućnost za dalji unos stavki menija, horizontalno i vertikalno (slika 21) što znači da je pravljenje hijerarhije menija veoma jednostavno i da se podrazumeva da su meniji u opštem slučaju hijerarhijski organizovani (što i jeste slučaj u iole složenijim aplikacijama):



Slika 21: Unošenje stavki menija

Verovatno ste primetili da svaki meni ima pridružen takozvani „hot key“ tj. skraćenicu. Slovo ili broj koji uz pritisnut „Alt“ specijalni taster predstavlja skraćenicu za izbor stavke u horizontalnom delu menija.

I ova funkcionalnost se jednostavno definiše – potrebno je ispred karaktera koji treba da bude „hot key“ ukucati karakter **&**. Ako želimo da „hot key“ za meni „Forme“ bude kombinacija **Alt+F** jednostavno treba ukucati tekst **&Forme**.

Karakter **&** se ne prikazuje, već je ispisano samo „Forme“. **Podvučeno slovo označava skraćenicu sa tastature.** Ovo se i vidi čim pritisnete **Enter** taster čime završavate unos teksta u stavci menija.

Nastavljamo sa kreiranjem menija „Forme“ tako što ćemo ispod njega napraviti nekoliko stavki. Ukucavamo redom jte jedno ispod drugog:

&Nova forma

&Forma Ime i prezime

&RadioButton forma

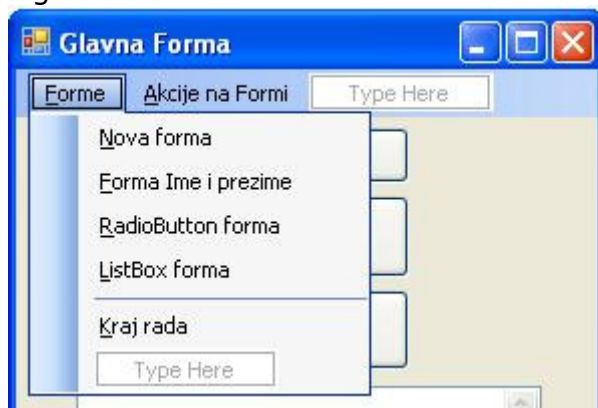
&ListBox forma

-

&Kraj rada

Ako za tekst stavke unesete samo znak – (minus), on ima specijalno značenje: u meniju se prikazuje horizontalni separator (horizontalna linija). Ovo je izuzetno korisno kada treba vizuelno razdvojiti različite grupe funkcionalnosti u okviru vertikalnog menija. Stoga je ova mogućnost ujedno i jedna od preporuka za kreiranje kvalitetnog GUI-a.

Na Glavnoj kontroli imamo i dugme „Povećaj“ kojim smo povećavali širinu forme. Dodajmo još jedan meni za akcije na formi (desno od menija Forme). Desno od menija Forme u polju gde piše „Type Here“ treba uneti tekst „&Akcije na formi“. Ispod ovog menija treba dodati stavku „&Povećaj“. Na kraju naš meni treba da izgleda kao na slikama 22 i 23.



Slika 22: Izgled menija „Forme“.



Slika 23: Izgled menija „Akcije na formi”

Dalje možemo da popunjavamo ostatak menija, zavisno od funkcionalnosti aplikacije koje naknadno dodajemo.

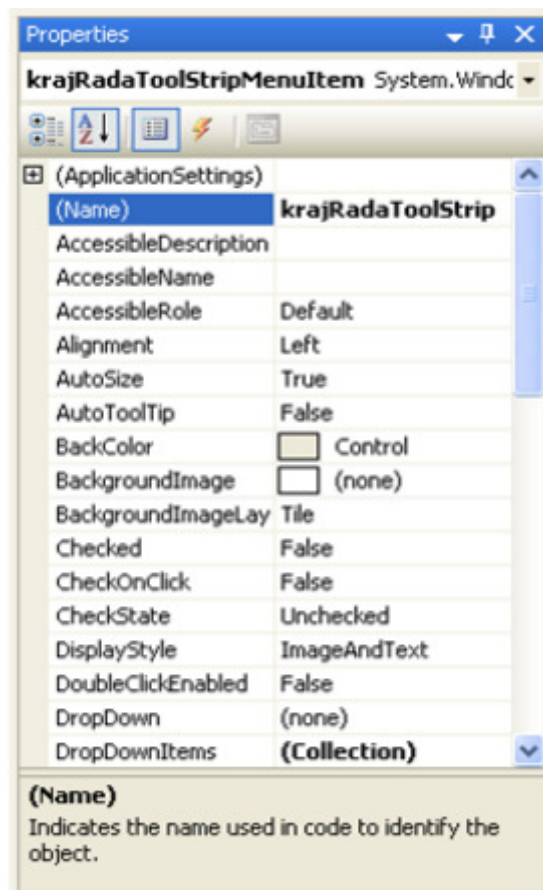
Tipičan hijerarhijski organizovan meni u velikom broju aplikacija se sastoji od menija **File**, **Edit**, **Tools** i **Help** kao što je to prikazano na slici 24. Više manje su i stavke ovih menija standardne (Open, Close, Save, Exit, itd) te ih treba postaviti u ovim menijima (naravno ako aplikacija realizuje takve funkcionalnosti) da bi korisnik koristio aplikaciju na uobičajeni način a samim tim i skratio vreme obuke za korišćenje aplikacije



Slika 24: Tipičan meni

Programiranje menija

Svaku stavku menija je najbolje "tretirati" kao komandno dugme. Programiranje se vrši tako što se zapravo programira **Click** događaj stavke menija. U primeru koji smo započeli kliknite samo jednom na stavku menija „**Kraj rada**”. Svaka stavka menija ima svoj skup svojstava koja su vidljiva u standardnom prozoru za prikaz svojstava objekta - „**Properties**” prozoru (slika 25):



Slika 25: Osobine stavke menija

Kao i kod svake druge kontrole ključno svojstvo je „**Name**“ koje je automatski generisano – u našem slučaju je to „**krajRadaToolStripMenuItem**“.

Svojstvo „**Text**“ definiše tekst stavke menija i njega smo podešavali iz dizajnera. Sada uradite dupli klik na stavku menija „**Kraj rada**“, čime se otvara kod **Click** događaja u prozoru za pisanje koda. Kada korisnik klikne na „Kraj rada“ želimo da zaustavimo izvršavanje programa. To se postiže pozivom metoda **Exit** objekta **Application** tj.:

```
private void krajRadaToolStripMenuItem1_Click(object sender, EventArgs e)
```

```
{
```

```
    Application.Exit();
```

```
}
```

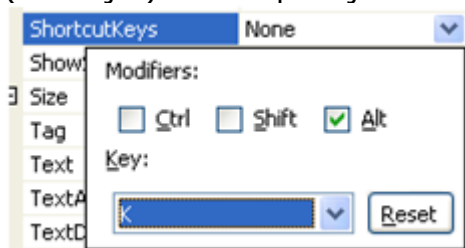
Application je referenca na aplikaciju koju kreiramo. Ovaj objekat poseduje metod **Exit** koji izvršava bezuslovni prekid rada aplikacije i oslobađa sve resurse koje je aplikacija zauzela (memoriju, datoteke, baze podataka i tako dalje...). Testirajmo to. Pokrenite aplikaciju i mišem kliknite na „**Kraj rada**“.

Pokušajte aktiviranje stavke i pomoću skraćenica sa tastature: prvo **Alt+F** kako bi otvorili meni „Forme“, a potom samo taster **K** kako bi aktivirali stavku „Kraj rada“. Medjutim, može se primetiti da se pritiskom na **Alt+F** otvara nova forma sa ispisanim tekstom „Ovo je nova forma koja može biti modalna i nemodalna“. Zašto? Ako bolje pogledamo dugme „Nova forma“ na Glavnoj formi, takodje ima skraćenicu **Alt+F**. Zbog toga promenimo skraćenicu ovog dugmeta na **Alt+N** tako što ćemo u Text osobini ovog dugmeta umesto teksta „Nova &forma“ uneti tekst „&Nova forma“. Pokrenimo iznova aplikaciju i proverimo šta će se sada desiti kada probamo skraćenicu **Alt+F**. Očigledno je da sada sve funkcioniše na način kako očekujemo tj. kako je prethodno opisano.

Skraćenice

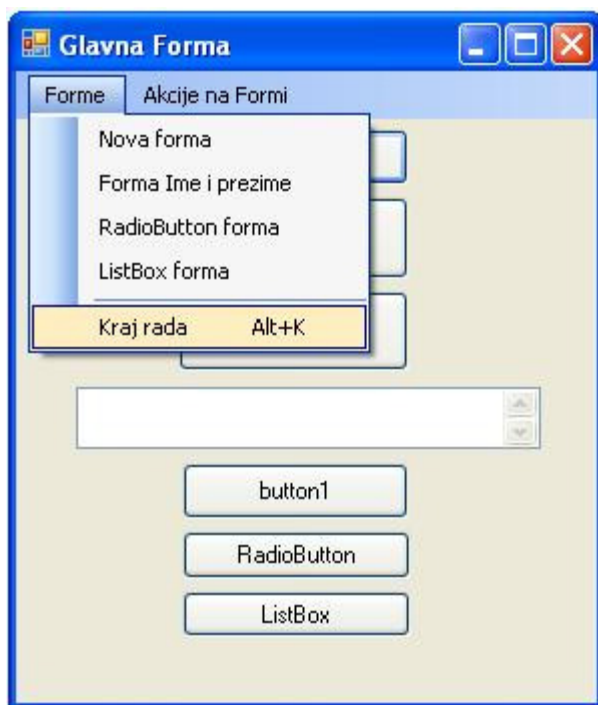
Postavlja se pitanje da li je moguće definisati skraćenicu sa tastature koja će, na primer, direktno aktivirati stavku „Kraj rada“?. Naravno da je moguće, čak se može definisati kombinacija bilo kog specijalnog tastera (**Control**, **Alt** ili **Shift**) i običnog karaktera. Takođe se mogu koristiti i funkcijski i drugi specijalni karakteri sa tastature.

U design modu ponovo jednim klikom miša izaberite u meniju Forme stavku „**Kraj rada**“. Potom u prozoru svojstava pronađite svojstvo „**ShortcutKeys**“. Otvorite padajuću listu pored ovog svojstva i trebalo bi odmah sve da bude jasno. Markirajte (čekirajte) **Alt** i iz padajuće liste izaberite slovo **K** kao što je prikazano na slici 26.



Slika 26: Podešavanje skraćenice za stavku menija

Sada kombinacija **Alt+K** završava našu aplikaciju (aktivira metodu Exit). Pokrenite aplikaciju i probajte. Primetite da se uz stavku menija prikazuje i njena skraćenica. Ovo se podešava svojstvom „**ShowShortcutKeys**“ koje može imati vrednosti **True** (tačno ili da) ili **False** (netačno ili ne).

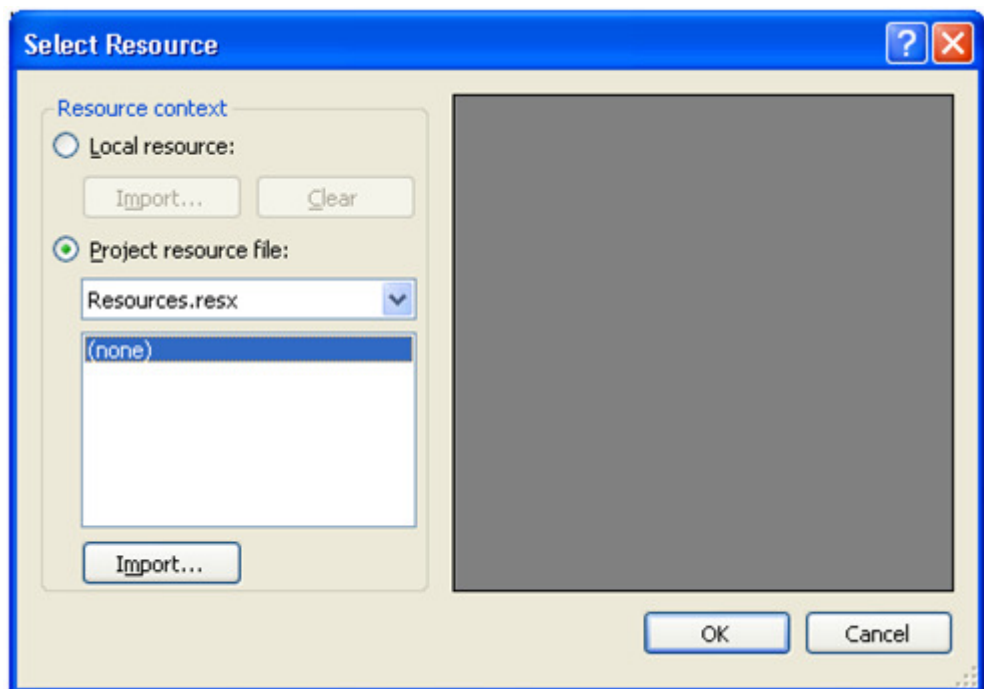


Slika 27: Skraćenica u stavki menija

Napomena: Pravite razliku izmedju "skraćenica": Shortcut i Hotkey!!!

Slike

Svaka stavka menija (uključujući i stavke u horizontalnoj liniji) može imati i sličicu tj. ikonu (eng. **Ikona**) koja doprinosi lepšem i razumljivijem korisničkom interfejsu (pod uslovom da je adekvatno izabrana i da je kvalitetno urađena što je opet "nauka za sebe"). Sličice koje se koriste za ovu namenu su najčešće u **bmp** ili **jpeg** formatu, dimenzija **16x16** piksela (mada može u druga veličina ali ne više od 32x32). Pogledajte na Internetu i naćićete veliki broj kolekcija sličica za besplatno preuzimanje. Ako ste vešti možete ih napraviti i sami korišćenjem nekog od programa za kreiranje ikona (preporuka je da probate da kreirate par ikona. Rad sa ovim programima je izuzetno jednostavan ali sam proces osmišljavanja tj. dizajniranja ikone je izuzetno težak!!! Izuzetno lako prevalimo preko usta rečenicu "Uf što su ružne ikone" ali je teško napraviti kvalitetne!). Pomoću svojstva „**Image**“ možete da izaberete sliku sa vašeg računara i dodelite je stavci vašeg menija. Kliknite na malo dugme pored ovog svojstva i dobićete dijalog kao što je prikazano na slici 28.



Slika 28: Dijalog za selekciju ikone za stavku menija.

U ovom dijalogu se dodaju resursi aplikaciji. Oni osim slika mogu biti i **zvučne datoteke**, **animacije** kao i druge binarne datoteke koje koristite u aplikaciji. Kliknite na dugme „**Import**“ i sa diska vašeg računara izaberite sliku koju ste prethodno pripremili. Na kraju kliknite na dugme „**OK**“. Ako je slika veća od 16x16 piksela, biće automatski skalirana na tu veličinu.

Uobičajeno je da se na računarima (verovatno i na vašem) u folderu “C:\Documents and Settings\All Users\Documents\My Pictures\Sample Pictures” nalazi i slika **Sunset.jpg** (ili pak neka druga). Ako ovu sliku navedemo kao resurs u dijalogu za dodavanje resursa za stavku menija „Kraj rada” naš meni će imati izgled kao na slici 29.

Nekada je korisna i mogućnost da se pored svake stavke menija nalazi znak za overu (eng. **Check**). Svojstvo **Checked** koje može imati vrednosti **True** ili **False** obezbeđuje upravo ovo.

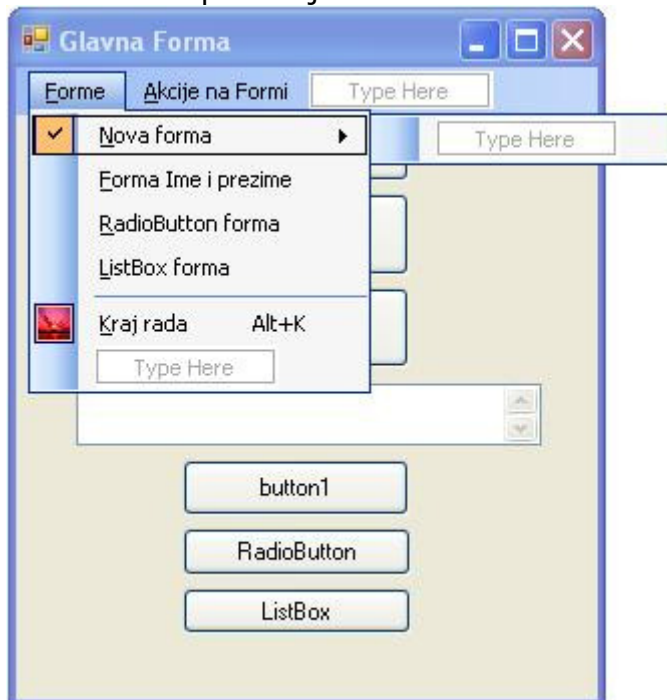
Izaberite stavku „**Nova forma**“, u svojstvima pronađite **Checked** i postavite ga na **True**. Sa leve strane ove stavke se pojavljuje znak za overu (slika 30). U praksi se znak za overu prikazuje ili ne zavisno od logike korišćenja aplikacije i akcija korisnika tj. trenutnog stanja aplikacije. Na primer, linija koda:

```
novaFormaToolStripMenuItem.Checked = true;
```

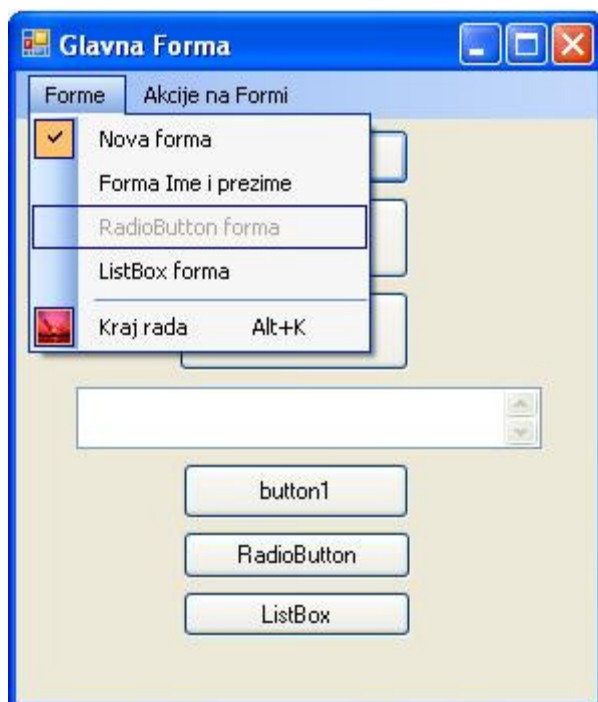
će za vreme izvršavanja programa postaviti znak za overu pored „Nova forma” stavke menija. Takođe, u praksi se često koristi dinamičko prikazivanje ili skrivanje stavke menija ili cele grupe, opet zavisno od logike aplikacije. Za ovo imamo dve mogućnosti, odnosno svojstva:

- Svojstvo **Enabled** koje kada je postavljeno na **False**, prikazuje stavku menija u sivoj boji i korisnik ne može da je aktivira (klikne na nju). Ponavljamo ovo je korisno ako želimo da korisnik zna da stavka menija postoji, ali trenutno nije dostupna (Na slici 30 je prikazan naš meni ako je za stavku „RadioButton forma” svojstvo **Enabled** postavljeno na **False**).
- Restriktivnije svojstvo **Visible** (koje imaju sve kontrole) postavljeno na **False** čini da se stavka menija uopšte ne vidi u situacijama kada ne treba da bude dostupna korisniku.

Napomena: Postavljanjem osobine Visible na false, kontrola i dalje postoji ali se trenutno ne vidi tj. ne prikazuje. Postavljanjem ove osobine na true, kontrola opet postaje vidljiva. Osobina Visible se veoma često eksploatiše da bi se postigli razni efekti (zavisno u kom kontekstu se koristi i sa kojom kombinacijom kontrola). Na primer, moguće je postaviti dve kontrole jednu preko druge i naizmeničnim postavljanjem njihovih osobina Visible na true odnosno false omogućiti da se naizmenično prikazuju.



Slika 29: Svojstvo Checked za stavku menija „Nova forma”

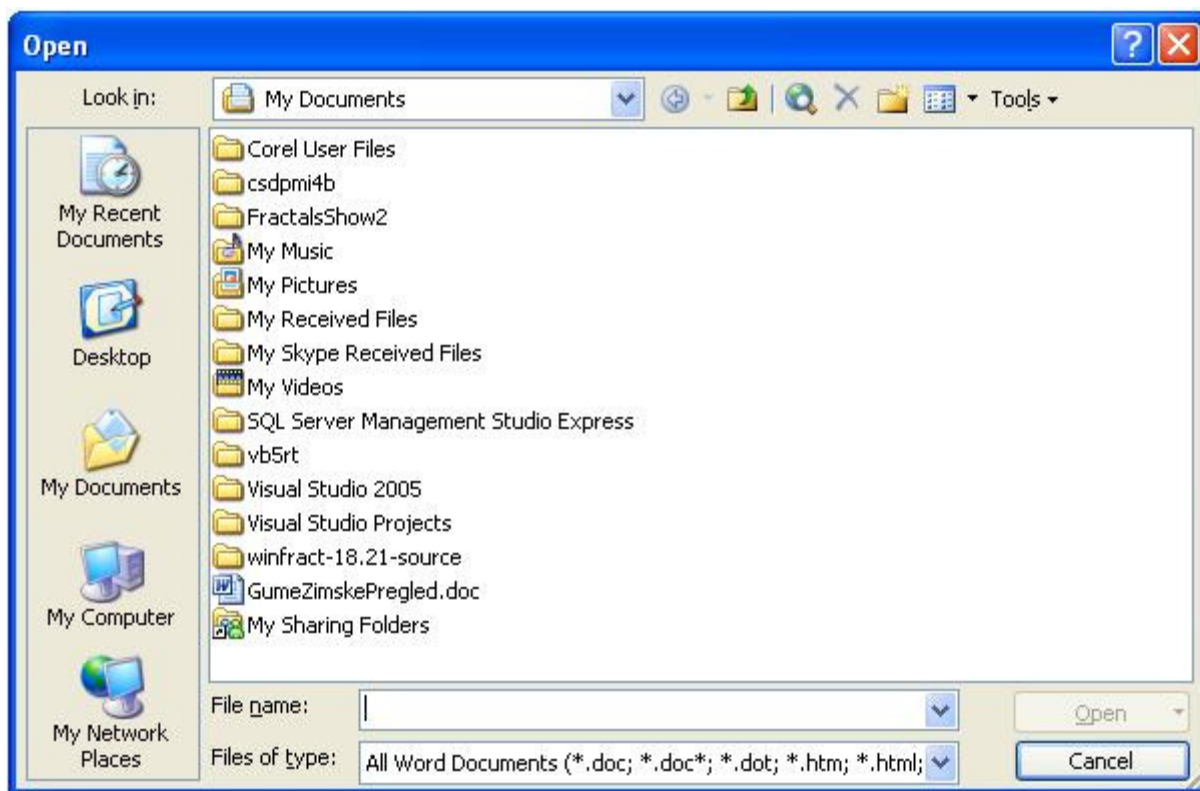


Slika 30: Rezultat postavljanja svojstvo Enabled na False za stavku „RadioGroup forma”

Rad sa standardnim dijalogima i datotekama

Većina Windows aplikacija ima potrebe da manipuliše različitim datotekama (fajlovima) (tekstualne datoteke, slike, itd.) u smislu čitanja i pisanja sadržaja.

Manipulacija datotekama obavezno uključuje i standardne dijaloge za izbor i čuvanje sadržaja. Oni se otvaraju izborom opcija **Snimi** (eng. Save) i **Otvori** (eng. Open). Na primer, u Microsoft Wordu 2003 dijalog za otvaranje dokumenta je prikazan na slici 31.



Slika 31: Open dijalog u aplikaciji Microsoft Word 2003

Bez obzira na vrstu dijaloga osnovne funkcionalnosti su iste: Moguće je izabrati datoteku, prikazane su samo datoteke koje zadovoljavaju određeni kriterijum (filter tj. tip datoteka) (All Word Documents na slici 31) uz mogućnost da se ručno unese ime datoteke - File name polje.

Ovaj dijalog je relativno složen i za njegovo programiranje treba uložiti dosta napora što se može zaključiti iz dosadašnjih lekcija. Upravo zbog toga su dijalozi, koji se često sreću u aplikacijama, deo mogućnosti razvojnog okruženja i nude se kao gotovi objekti. Znači nije ih potrebno ručno programirati! Oni su sistemski i postoje u svakoj verziji Windowsa. Čak, ako na primer kreirate aplikaciju u Windowsu XP sa ovim dijalozima i kopirate izvršnu datoteku na drugom OS npr. Windows Vistu, videćete Vistine dijaloge bez ikakve izmene aplikacije.

Korišćenje ovih dijaloga je veoma jednostavno, direktno i potpuno podržano u razvojnem okruženju VisualStudio. (u svim verzijama).

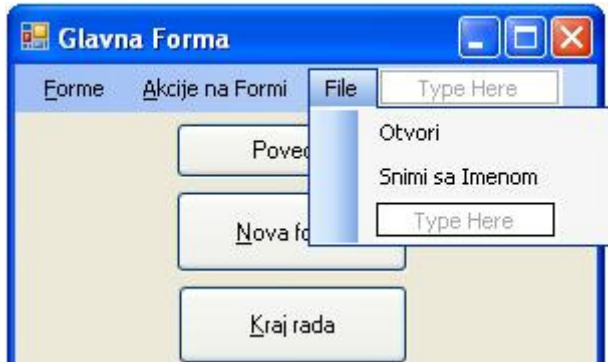
Korišćenje ovih dijaloga se može opisati sledećim koracima:

1. Postaviti odgovarajuću kontrolu na formu.
2. Postaviti potrebna svojstva kontrole (naslov, početni direktorijum, filter za ekstenzije datoteka, ...).
3. Otvaranje dijaloga

4. Po zatvaranju dijaloga proveriti da li je korisnik izabrao datoteku ili kliknuo na dugme Odustani (eng. Cancel)

5. Ako je izabrana datoteka, iz svojstva **FileName** čitamo putanju i ime datoteke i dalje radimo sa njom

Demonstriraćemo način na koji se koristi Open dijalog tako što ćemo u našu aplikaciju da dodamo meni “File” sa stavkama “Otvori” i “Snimi sa Imenom” (slika 32).



Slika 32: Meni “File”

Nadalje kreirajmo novu formu „OpenDialogForm” na način kako je to ranije objašnjeno. Ova forma treba da se pojavi kada se aktivira stavka „Otvori” u meniju „File”. Da bi to postigli za događaj **Click** za stavku „Otvori” unesimo sledeći kod

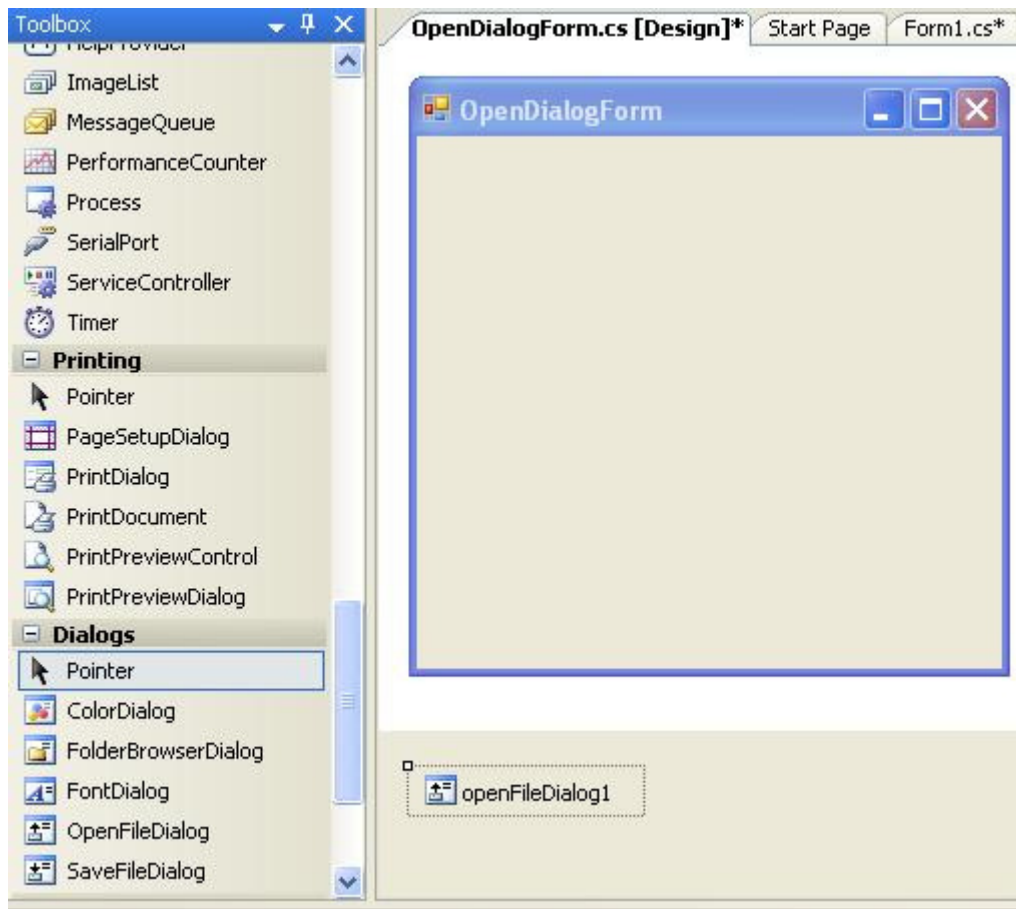
```
private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog odf = new OpenFileDialog();

    odf.ShowDialog();
}
```

Dalje radimo u skladu sa prethodno opisanim koracima:

1. Postaviti odgovarajuću kontrolu na formu

U paleti sa alatcima (sa leve strane razvojnog okruženja), pronadite sekciju “**Dialogs**” kao što je prikazano na slici 33. U ovoj sekciji se nalaze svi standardni Windows dijalozi. Osim pomenutih, tu su dijalozi za izbor boje, pregled direktorijuma i izbor fonta. Na formu „OpenDialogForm” postavite (prevuci i pusti (drag & drop) postupak) kontrolu **OpenFileDialog**. Kontrola se prikazuje na traci ispod forme i inicijalno je dobila naziv (svojstvo **Name**) **openFileDialog1**.



Slika 33: Dialogs sekcija u paleti sa alatima

2. Postaviti potrebna svojstva kontrole (naslov, početni direktorijum, filter za ekstenzije datoteka)

Kada pokrenemo dijalog za otvaranje datoteke, uobičajeno je da u njemu vidimo samo datoteke sa određenom ekstenzijom (tj. datoteke odgovarajućeg tipa) koju naša aplikacije prepoznaje. Obično se za svaki slučaj dodaje i opcija za prikaz svih datoteka - filter *.* sigurno ste do sada u ovakvim dijalogima videli opciju "All files" (sve datoteke).

Filter Open dijaloga se postavlja pomoću istoimenog svojstva **Filter**. Ovo svojstvo je tipa **string** i zahteva specifično formatiranje. Na primer, ako ovo svojstvo postavimo na:

Tekstualne datotekel*.txt|Sve datotekel*.*

napravili smo dva filtra. Prvi, sa nazivom "**Tekstualne datoteke**", koji prikazuje sve datoteke sa ekstenzijom **txt**, i drugi sa nazivom "**Sve datoteke**", koji prikazuje upravo to: sve datoteke sa bilo kojom ekstenzijom.

Iz ovoga se vidi da se svaki filter sastoji od **dva** dela: naziva filtra i kriterijuma za prikaz datoteka. Separator je **vertikalna crta**. Obratite pažnju da na kraju ne treba staviti vertikalnu crtu.

Dozvoljene su i sledeće kombinacije:

`Slike*.jpg;*.gif;*.bmp|Video*.mpg;*.avi;*.wmv`

U ovom slučaju takođe imamo dva filtra, **Slike** i **Video** ali svaki od njih će prikazati datoteke sa više ekstenzija. Separator između ekstenzija istog filtra je karakter **tačkazarez** (;).

U našem primeru, svojstvo **Filter openFileDialog1** kontrole postavite na:

`Tekstualne datoteke|*.txt|Sve datoteke|*.*`

Obično se postavlja i naslov dijaloga koji se definiše pomoću svojstva **Title**. Pošto u našem primeru treba otvoriti tekstualne datoteke postavite ovo svojstvo na **Otvori tekstualnu datoteku**.

Opciono možemo postaviti i naziv direktorijuma čiji će sadržaj biti prikazan odmah po otvaranju dijaloga. Ako želimo da se odmah prikaže koren diska C: u svojstvo **InitialDirectory** upišite **C:**.

Sada smo pripremili dijalog za otvaranje.

3. Otvaranje dijaloga

Dijalog poseduje metodu **ShowDialog** koja kao rezultat izvršavanja vraća akciju korisnika u dijalogu. Ova akcija predstavlja enumeraciju pod nazivom **DialogResult**. Nama su od interesa samo dve opcije:

- **DialogResult.OK** - korisnik je izabrao datoteku
- **DialogResult.Cancel** - korisnik je zatvorio dialog (kliknuo na Cancel dugme)

Proverom ove povratne vrednosti znamo da li je datoteka izabrana i dalje nastavljamo rad sa njom.

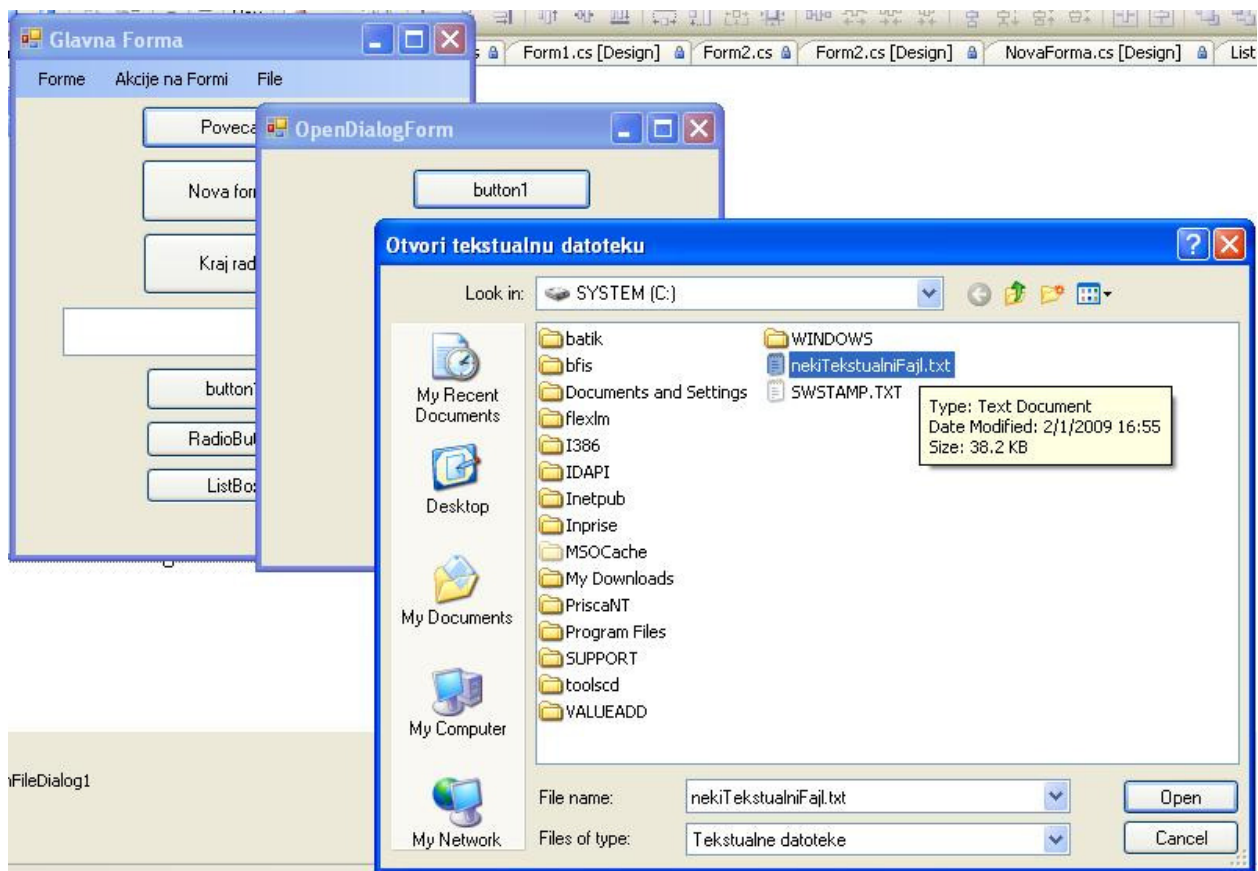
Na formu „OpenDialogForm” postavite jedno dugme i uradite dupli klik mišem na njega kako bi napisali sledeći kôd.

```
private void button1_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        this.Text = openFileDialog1.FileName;
    }
}
```

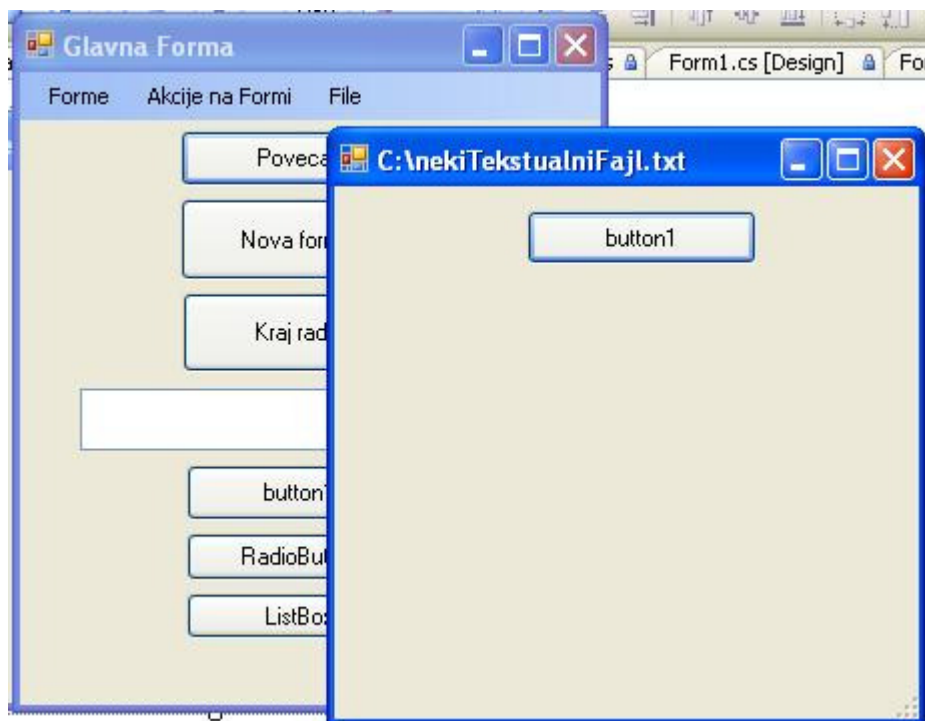
}

Pomoću metode **openFileDialog1.ShowDialog()** otvaramo dijalog i odmah proveravamo njegovu povratnu vrednost. Ako je ona jednaka **DialogResult.OK**, znači da je korisnik izabrao datoteku i kliknuo na dugme “**Open**” u dijalogu. U tom slučaju u svojstvu **FileName** se nalazi ime izabrane datoteke sa kompletnom putanjom. U naslovu formulara (**this.Text**) upisujemo putanju i ime izabrane datoteke. Dakle svojstvo **FileName** je ključno, jer definiše lokaciju izabrane datoteke u file sistemu.

Pokrenite program, aktivirajte stavku Otvori u meniju File a zatim kliknuti na dugme **Open**. Kao rezultat pojaviće se standardni dijalog za otvaranje fajla (slika 34). Obratite pažnju na dva filtra koja smo kreirali i proverite funkcionisanje oba. Na svom računaru pronađite odgovarajuću datoteku (u dijalogu prikazanom na slici 34 izabran je fajl nekiTekstualniFajl.txt) i kliknite na dugme “**Open**” na dijalogu. Nakon ovoga u naslovu forme će biti ispisana kompletna putanja sa nazivom izabrane datoteke (u našem slučaju „C: \nekiTekstualniFajl.txt”, vidi sliku 35.).



Slika 34: Standardni File Open dijalog.



Slika 35: Rezultat izbora fajla "nekiTekstualniFajl.txt".

Prikaz sadržaja datoteke

Otvaranje dijaloga i izbor datoteke, naravno ne otvara niti prikazuje njen sadržaj. Dijalogom smo samo selektovali neku postojeću datotetu. Sledeći logičan korak je da zaista i prikažemo sadržaj tekstualne datoteke. Za otvaranje, čitanje i pisanje po datotekama je zadužen skup klasa u imenovanom prostoru (eng. NameSpace) **System.IO**, te je neophodno da ga uključimo ([using System.IO;](#)) da bi mogli da ga koristimo.

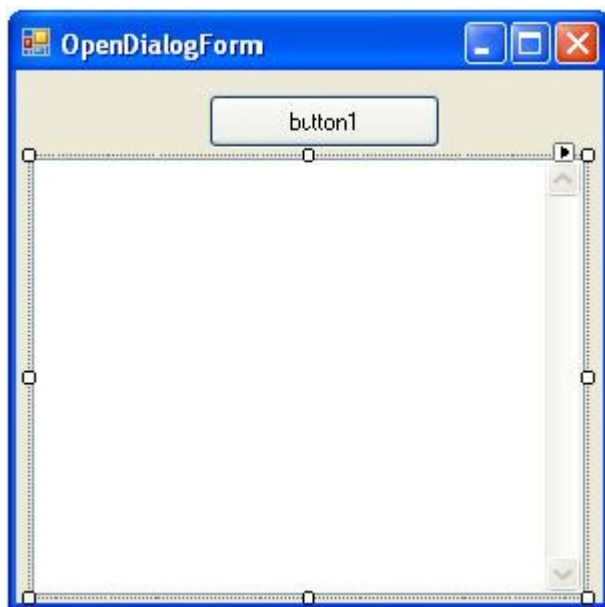
Za čitanje sadržaja datoteka koristi se klasa **StreamReader** kojoj se u konstruktoru zadaje putanja do konkretne datoteke. Putanju kao što smo videli već imamo u **FileName** svojstvu dijaloga. Metodom **ReadToEnd** StreamReader objekta čita se kompletan sadržaj zadate datoteke, i ova metoda vraća tip **string** koji ćemo prikazati u kontroli **TextBox**.

Zaustavite aplikaciju i na formu postavite TextBox kontrolu u kojoj će biti ispisan sadržaj tekstualne datoteke. Kontrola je inicijalno dobila ime (**Name** svojstvo) **textBox1**. Postavite joj sledeća svojstva na prikazane vrednosti:

Multiline: True

ScrollBars: Vertical

Forma "OpenDialog Form" sada treba da izgleda kao na slici 36.



Slika 36: Forma

Uradite dupli klik na dugme, kako bi modifikovali kôd na sledeći način:

```
private void button1_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        this.Text = openFileDialog1.FileName;

        System.IO.StreamReader sr = new StreamReader(openFileDialog1.FileName);

        textBox1.Text = sr.ReadToEnd();

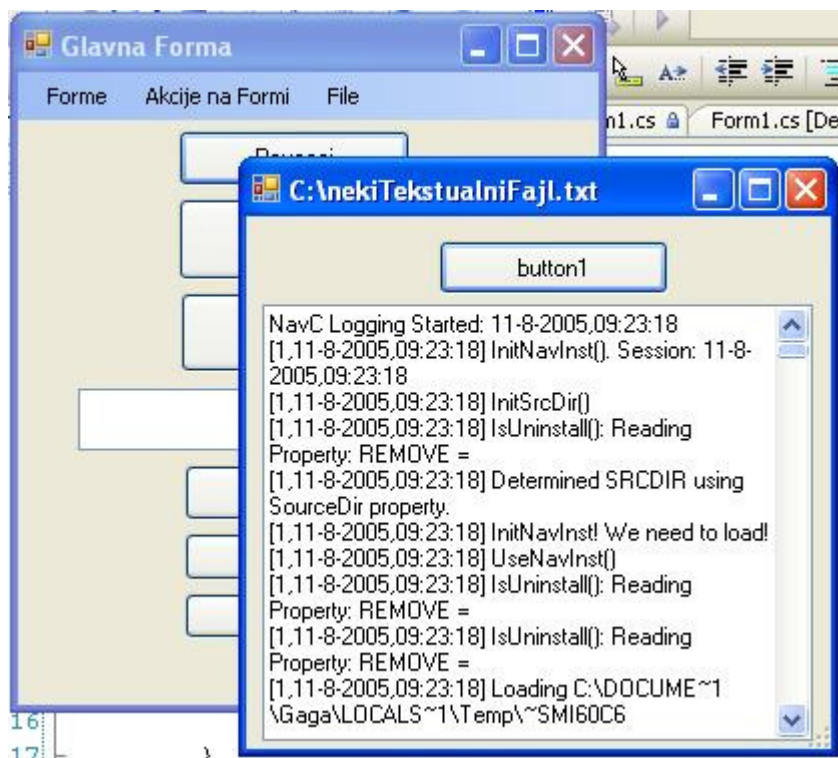
        sr.Close();
    }
}
```

Analizirajmo napisani kod. Inicijalizujemo objekat **sr** tipa `System.IO.StreamReader` i u konstruktoru postavljamo ime datoteke koju je korisnik izabrao u dijalogu. Metodom **ReadToEnd()** čitamo kompletni sadržaj datoteke i postavljamo ga u **Text** svojstvo kontrole **textBox1**.

Na kraju treba obavezno zatvoriti `StreamReader` objekat, što se radi metodom **Close()**. Ako startujemo aplikaciju i u dijalogu izaberemo tekstualni fajl njegov sadržaj će biti prikazan u `TextBoxu` kao što je to prikazano na slici 37.

UPOZORENJE:

Ako **StreamReader** ne zatvorite metodom **Close()**, datoteka koja se čita ostaje **zaključana** (eng. Locked) tako da ni jedna druga aplikacije ne može da je otvori. Kaže se da **StreamReader** datoteku otvara u **ekskluzivnom** režimu. Međutim, ako regularno zatvorite svoju aplikaciju, automatski se “otključavaju” sve otvorene datoteke. Znači obavezno je zatvaranje streamova!



Slika 37: Prikazan sadržaj datoteke nekiTekstualniFajl.txt.

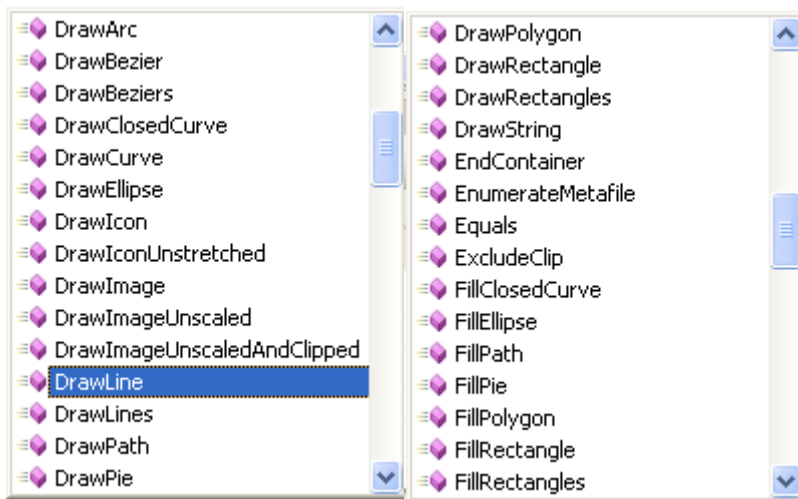
Klasa Graphics

Od velikog broja aplikacija se zahteva da pored teksta i brojeva imaju i odgovarajući grafički prikaz. Klasa koja omogućuje crtanje po bilo kojoj sistenskoj kontroli je klasa **Graphics**. Objekat ove klase predstavlja površinu po kojoj se crta. Znači prvi korak je kreiranje ovog objekta a zatim njegovo korišćenje u skladu sa slikom koju želimo da nacrtamo. Objekat klase Graphics za neku kontrolu se kreira pozivom metode **CreateGraphics()** tako da je sada površina za crtanje ta kontrola. Klasa Graphics omogućuje crtanje različitih oblika kao što su: *linija, pravougaonik, popunjeni pravougaoni, elipsa, popunjena elipsa, kružni luk, kriva linija na osnovu 4 tačke*, itd. Metode koje se najčešće koriste za ovakva crtanja su:

- **DrawLine** – crtanje prave linije,

- **DrawRectangle** –crtanje pravougaonika,
- **FillRectangle** – crtanje popunjenog pravougaonika,
- **DrawEllipse** – crtanje elipse,
- **FillEllipse** – crtanje popunjene elipse,
- **DrawArc** – crtanje kružnog luka,
- **DrawBezier** – crtanje krive (Bezierove krive) na osnovu 4 tačke.

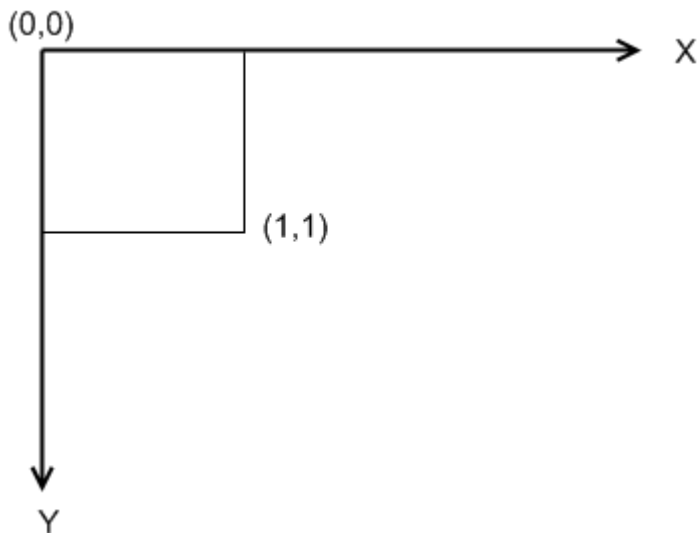
Pored ovih metoda postoje i druge metode iz čijih imena se može pretpostaviti način funkcionisanja tj. dejstvo. Neke se mogu videti na slici 1.



Slika 1: Metode za crtanje klase Graphics

Svaka površina po kojoj se crta ima koordinatni sistem koji malo odstupa od klasičnog Dekartovog koordinatnog sistema. Naime, u ovom koordinatnom sistemu se koordinatni početak (**x** koordinata jednaka **nuli** i **y** koordinata jednaka **nuli**) nalazi u gornjem levom uglu, pozitivni deo x ose ide od leve strane ka desnoj a pozitivni deo y ose od vrha ka dnu (pogledaj sliku 2).

Napomena: Ovakav načina definisanja koordinatnog sistema nije specifičnost programskog jezika C# već uobičajeni način za definisanje koordinatnog sistema tj. koristi se i u drugim programskim jezicima. Na prvi pogled ovakav način definisanja je malo čudan ali u toku rada sa grafičkim elementima postaće jasno zašto je ovaj način pogodniji od standardnog načina za predstavljanje koordinatnog sistema.



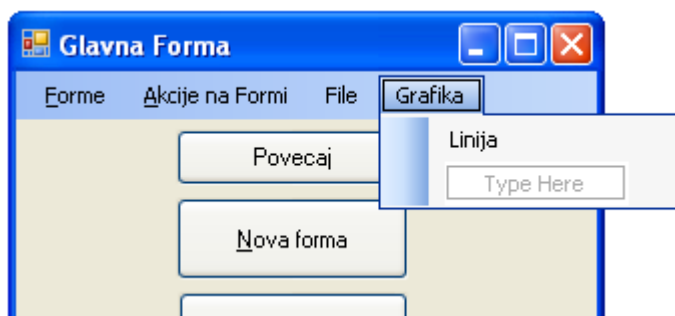
Slika 2: Koordinatni sistem površine za crtanje

Primer: Crtanje Linije

Kreirajmo novu formu koju ćemo nazvati **Grafika** na kojoj ćemo da nacrtamo jednu pravu liniju plave boje koja će biti usmerena od gornjeg levog ugla ka donjem desnom uglu forme. Prethodno dodajmo novi meni Grafika (na našu formu koja je korišćena u prethodnim primerima) sa podmenijem Linija kao što je to prikazano na slici 3. Za stavku Linija vežimo prikazivanje forme Grafika generisanjem koda za događaj Click na sledeći način.

```
private void linijaToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    Grafika grafika = new Grafika();
    grafika.ShowDialog();
}
```



Slika 3: Meni Grafika sa stavkom Linija

Pošto ćemo crtati po formi Grafika potrebno je da napišemo sledeći kod koji će da se veže za događaj **Paint** (događaj Paint je zadužen za samo iscrtavanje i on ima standardne argumente) tj.

```
private void Grafika_Paint(object sender, PaintEventArgs e)

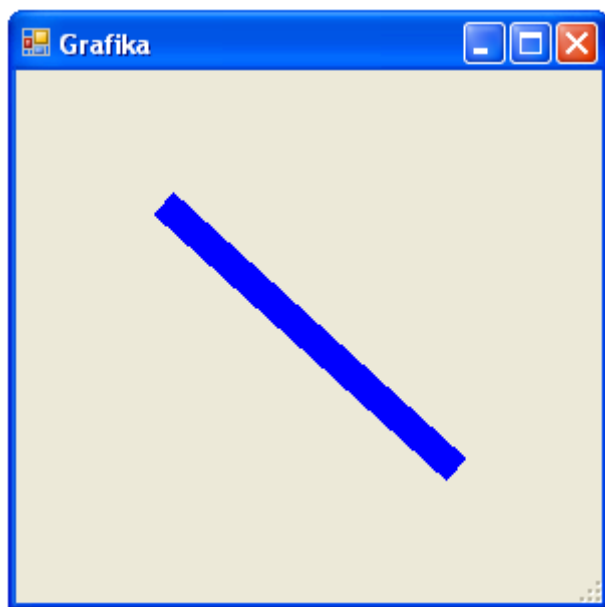
{

    Graphics graf = e.Graphics;
    Pen olovka = new Pen(Color.Blue, 15);
    int sirina = ClientRectangle.Width;
    int visina = ClientRectangle.Height;
    Point pocetakLinije, krajLinije;
    pocetakLinije = new Point(sirina / 4, visina / 4);
    krajLinije = new Point(3 * sirina / 4, 3 * visina / 4);
    graf.DrawLine(olovka, pocetakLinije, krajLinije);
    olovka.Dispose();
}
```

Kao rezultat izvršenja programa (izbor stavke Linija u meniju Grafika u glavnoj formi) dobija se plava linija kao što je to prikazano na slici 4.

Analizirajmo kod koji se izvršava za događaj Paint. Najpre je kreiran objekat klase Graphics. Zatim je kreiran objekat klase Pen koja definiše kako će izgledati linija kojom se crta (grubo rečeno kakvom olovkom ćemo crtati) tj. oblik linije, debljina linije, boja, itd.

Olovku smo podesili na plavu boju i debljinu linije od 15 pointa tako što su Color.Blue i 15 korišćeni kao argumenti konstruktora klase Pen. Linija koja se iscrtava primenom metode DrawLine spaja dve prethodno definisane tačke: pocetakLinije i krajLinije koje predstavljaju objekte klase Point. Ova klasa ima konstruktor čiji su argumenti celobrojne koordinatne X i Y izražene kao broj pixela. Interesantno je kako su određene ove dve tačke. Ako pokrenemo aplikaciju prikazaće se plava linija kao na slici 4.



Slika 4: Linija

Šta će se desiti ako promenimo veličinu prozora tj. forme Grafika (klik miša na donji desni ugao forme u oblasti 6 sivih tačkica u obliku trougla i pomeranje miša)? Primetićemo da se i linija menja u skladu sa promenom veličine forme. Kako je to postignuto? Primetićete da su za određivanje krajnjih tačaka linije korišćene dve pomoćne promenljive **sirina** i **visina** koje su postavljene na širinu odnosno visinu oblasti forme na po kojoj može da se crta (tj. koja je pod kontrolom korisnika) a predstavljena je osobinom **ClientRectangle** koja ima osobine **Width** i **Height**. Tačke smo odredili relativno u odnosu na **ClientRectangle** tj. kao jedna odnosno tri četvrtine širine odnosno visine. Na ovaj način će se svaki put kada promenimo veličinu forme Grafika promeniti i apsolutni položaji tačaka koji će relativno u odnosu na veličinu forme biti na istom mestu.

Napomena: Na osnovu prethodnog može se zaključiti da će se metoda za crtanje `Grafika_Paint()` tj. u opštem slučaju događaj `Paint` pozivati od strane sistema svaki put kada se nešto desi sa formom (u našem slučaju to je promena veličine forme).

Na kraju se pozivom metode `Dispose()` oslobađaju resursi zauzeti pri kreiranju objekta **olovka** klase `Pen`.

Crtanje Pravougaonika i Elipse

Crtanje pravougaonika i elipse je dosta slično crtanju linije tako da će kod za njihovo iscrtavanje biti dosta sličan kodu za iscrtavanje linije koji je ranije objašnjen.

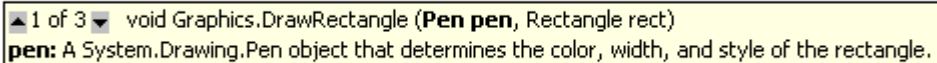
Pravougaonik se crta pozivom metode **DrawRectangle** koja se javlja u tri oblika a jedan od njih je i

DrawRectangle (Pen pen, int x, int y, int width, int height)

tj. argumenti su redom: objekat klase Pen, x koordinata gornjeg levog temena, y koordinata gornjeg levog temena, širina i visina pravougaonika.

Napomena: Ako se neka metoda pojavljuje u više pojava (overload) onog trenutka kada otkucamo otvorenu zagradu za unos argumenata pojaviće se pomoć (hint) koja će nam omogućiti da vidimo sve varijante metode i tipove odgovarajućih argumenata. Varijante biramo klikom na simbole strelica naniže ili naviše (slika 5). Sa slike 5 se vidi da metoda DrawRectangle ima 3 oblika i trenutno je prikazan drugi.

```
graf.DrawRectangle (
```



1 of 3 void Graphics.DrawRectangle (Pen pen, Rectangle rect)
pen: A System.Drawing.Pen object that determines the color, width, and style of the rectangle.

Slika 5: Pomoć za metode koje imaju više oblika

Elipsa se crta primenom metode **graf.DrawEllipse** koja takođe ima više pojava, tačnije četiri od kojih je jedan sličan prethodno pomenutom za crtanje pravougaonika tj.

graf.DrawEllipse (Pen pen, int x, int y, int width, int height)

gde su argumenti redom: objekat klase Pen, i pravougaonik opisan oko elipse koji je određen x koordinatom gornjeg levog temena, y koordinatom gornjeg levog temena, širinom i visinom.

Napomena: Ako su parametri za crtanje pravougaonika takvi da su mu stranice jednake dobija se kvadrat. Dok se pri crtanju elipse dobija kružnica ako su stranice pravougaonika u koji je upisana elipsa iste tj. predstavlja kvadrat.

Dopunimo metodu za crtanje kodom za crtanje jednog pravougaonika u plavoj boji, kvadrata u crvenoj boji, elipse u žutoj boji i kružnice u zelenoj boji. Pre toga promenimo debljinu olovke na 5. Sada metoda izgleda

```
private void Grafika_Paint(object sender, PaintEventArgs e)
```

```
{
```

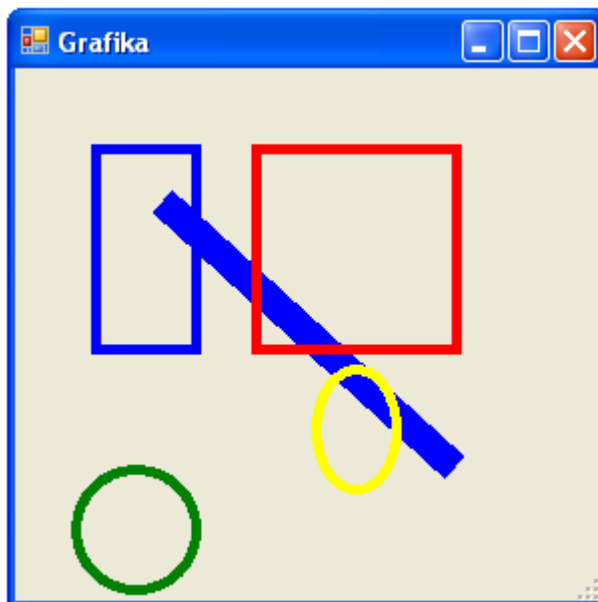
```
    Graphics graf = e.Graphics;  
    Pen olovka = new Pen(Color.Blue, 15);  
    int sirina = ClientRectangle.Width;  
    int visina = ClientRectangle.Height;  
    Point pocetakLinije, krajLinije;  
    pocetakLinije = new Point(sirina / 4, visina / 4);  
    krajLinije = new Point(3 * sirina / 4, 3 * visina / 4);  
    graf.DrawLine(olovka, pocetakLinije, krajLinije);
```

```

    olovka.Width = 5;
    graf.DrawRectangle(olovka, 40, 40, 50, 100);
    olovka.Color = Color.Red;
    graf.DrawRectangle(olovka, 120, 40, 100, 100);
    olovka.Color = Color.Yellow;
    graf.DrawEllipse(olovka, 150, 150, 40, 60);
    olovka.Color = Color.Green;
    graf.DrawEllipse(olovka, 30, 200, 60, 60);
    olovka.Dispose();
}

```

Pokretanjem aplikacije iscrtavaju se oblici kao na slici 6.



Slika 6: Oblici: linija, pravougaonik, kvadrat, elipsa i kružnica

Šta će se desiti ako sada promenimo veličinu prozora Grafika? Očigledno je da će plava linija promeniti položaj i veličinu a ostale figure ne menjaju ni položaj ni veličinu što je i razumljivo jer smo pri određivanju njihovih parametara koristili vrednosti koje ne zavise od veličine površine za crtanje kao što je to slučaj sa linijom (slika 7).

Ako želimo da oblici budu ispunjeni nekom bojom koristimo odgovarajuću funkciju za popunu tipa **FillOblik**. Da bi demonstrirali ove funkcije dodajmo telu metode Grafika_Paint sledeće linije koda čije je dejstvo iscrtavanje pravougaonika ispunjenog belom bojom i poligona sa 5 tačaka ispunjenog ljubičastom bojom (vidi sliku 8).

```
SolidBrush cetka = new SolidBrush(Color.White);
```

```
graf.FillRectangle(cetka, 60, 150, 40, 60);
```

```
cetka.Color = Color.Violet;
```

```
Point[] tackePoligona = {
```

```
    new Point(30, 60),
```

```
    new Point(140, 80),
```

```
    new Point(160, 100),
```

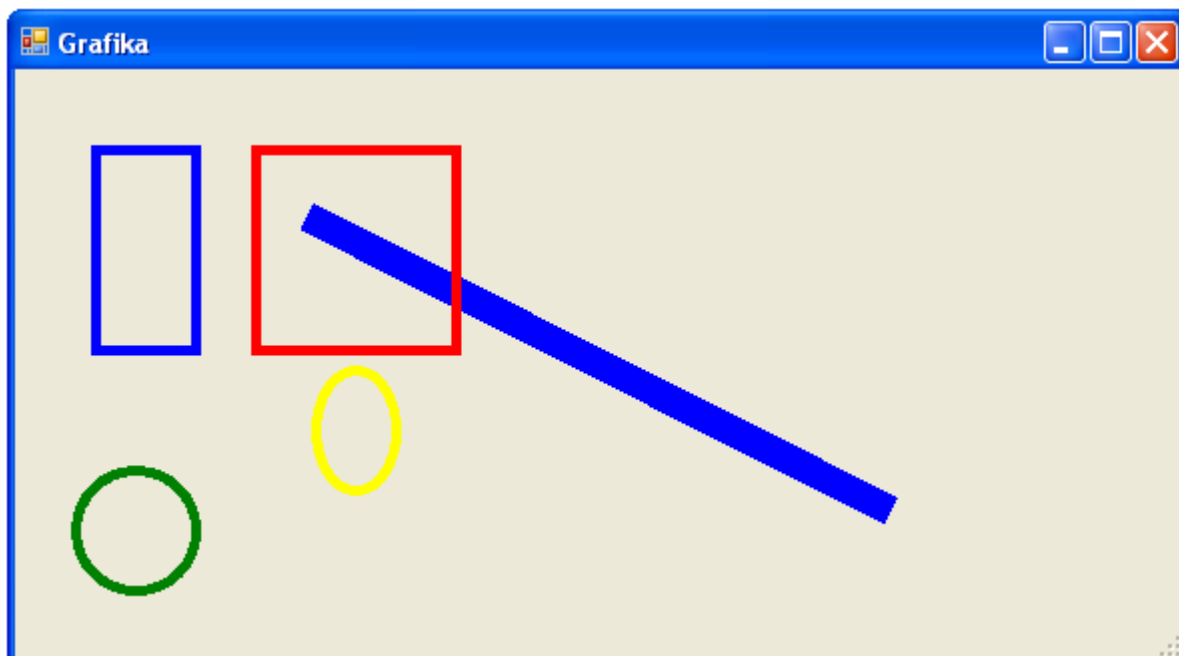
```
    new Point(90, 120),
```

```
    new Point(70, 90)
```

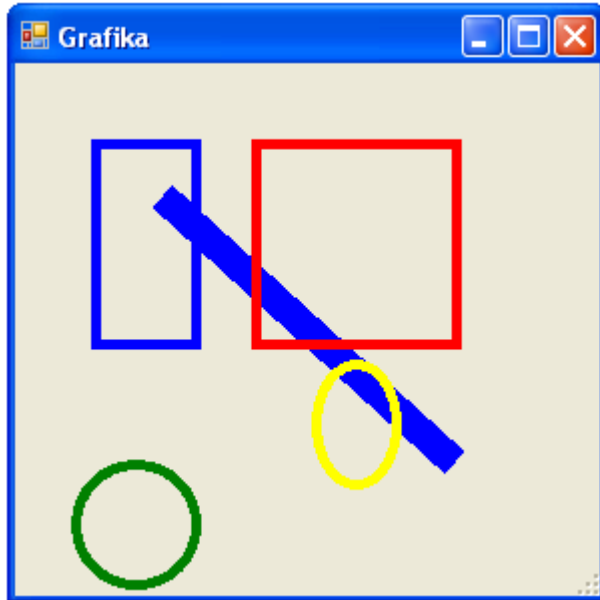
```
};
```

```
graf.FillPolygon(cetka, tackePoligona);
```

Da bi ispunili neki oblik nekom bojom potrebno je kreirati objekat tipa **SolidBrush** (u našem primeru objekat **cetka**) i podesiti mu osobinu **Color** na željenu boju. Za crtanje poligona prethodno smo generisali polje objekata tipa **Point**. Naš poligon ima 5 tačaka a može imati po želji više ili manje tačaka.



Slika 7: Povećan prozor za crtanje prikazan na slici 6

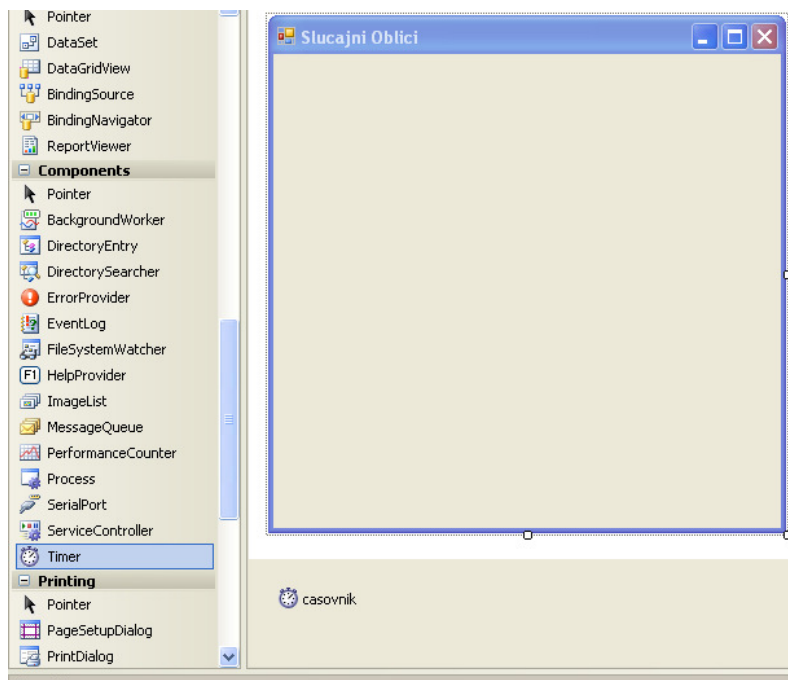


Slika 8: Ispunjeni oblici (pravougaonik i poligon)

Primer

Razmotrimo sada slučaj gde će se na formi iscrtavati pravougaonici koji će imati slučajan oblik, slučajnu boju i iscrtavaće se na slučajnim pozicijama na formi. Pravougaonici će se iscrtavati u pravilnim vremenskim razmacima. U titlebaru forme za iscrtavanje će stajati informacija o broju iscrtanih pravougaonika na formi u vidu teksta „Broj iscrtanih pravougaonika je ...“.

Generišimo novi tip forme i nazovimo je **SlučajniOblici** na način kao što je to rađeno i do sada. Iz Toolboxa dodajmo kontrolu **Timer** (nalazi se u grupi **Components**) na formu SlučajniOblici i nazovimo je **časovnik** (slika 9). Simbol časovnika će se pojaviti u dnu.



Slika 9: Komponenta Timer

Da bi timer bio aktivan podesimo osobinu **Enabled** na **True** a osobinu **Interval** na 1000 (slika 10). Povećanjem ili smanjenjem vrednosti za Interval usporavaćemo ili ubrzavati pojavljivanje novih pravougaonika tj. ova osobina određuje takt časovnika tj. vremenski interval. Nakon svakog intervala pojavljuje se jedan otkucaj tj. **Tick** događaj za koji ćemo vezati iscrtavanje novog pravougaonika. Podesimo metodu koja će se izvršavati na svaki događaj Tick na casovnik_Tick kao što je prikazano na slici 11. U metodi **casovnik_Tick** najpre generišemo objekat klase Graphics. Zatim korišćenjem objekta klase **Random** (generator slučajnih brojeva) koji je globalna promenljiva nazvana **SlučajniBroj** definišemo X i Y koordinatu mesta pojavljivanja pravougaonika vodeći računa o veličini prozora za iscrtavanje (koordinante moraju biti unutar oblasti za crtanje) kao i veličinu pravougaonika tj. širinu i visinu pravougaonika. Primećuje se da se svaki novi slučajni broj generiše pozivom metode **Next** koja može da ima dva celobrojna argumenta koja predstavljaju interval u kome treba da se nađe slučajni broj ili pak jedan argument koji predstavlja gornju granicu za generisani slučajni broj (donja granica je nula). Za generisanje boje je uzet slučajni broj sa ograničenjem 256 što znači da će pravougaonici imati jednu od 256 različitih boja. Na kraju se podešava **Text** osobina forme tako što se na fiksni tekst „Broj iscrtanih pravougaonika je“ dodaje vrednost globalne celobrojne promenljive **brojIscrtanihOblika** koja se povećava nakon svakog iscrtanog pravougaonika za jedan. Pošto je brojIscrtanihPravougaonika celobrojna promenljiva koristi se metoda **ToString()** kako bi se celobrojna vrednost promenljive konvertovala u string.

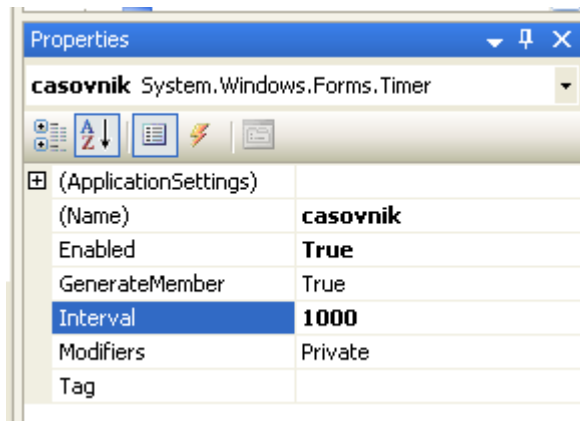
Kod metode **casovnik_Tick** je:

```

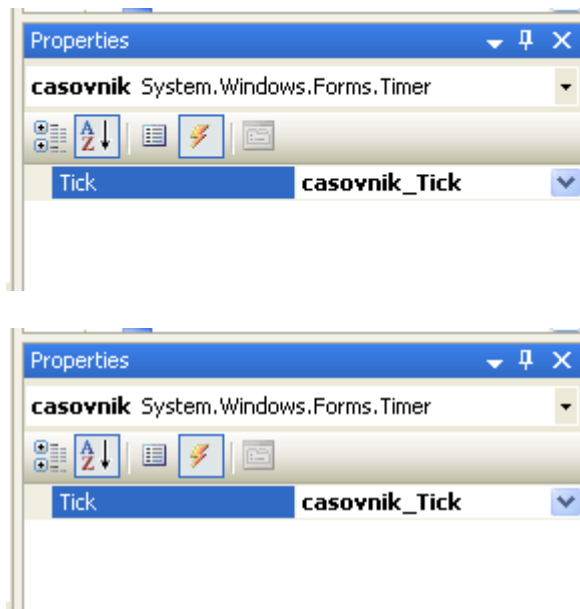
public partial class SlucajniOblici : Form
{
    Random SlucajniBroj = new Random();
    int brojIscrtanihOblika = 0;

    private void casovnik_Tick(object sender, System.EventArgs e)
    {
        Graphics graf = CreateGraphics();
        int sirina = SlucajniBroj.Next(20, 150);
        int visina = SlucajniBroj.Next(20, 150);
        int xKoordinata = SlucajniBroj.Next(0, ClientRectangle.Width - sirina);
        int yKoordinata = SlucajniBroj.Next(0, ClientRectangle.Height - visina);
        SolidBrush cetka = new SolidBrush(Color.FromArgb(SlucajniBroj.Next(256),
SlucajniBroj.Next(256), SlucajniBroj.Next(256)));
        graf.FillRectangle(cetka, xKoordinata, yKoordinata, sirina, visina);
        brojIscrtanihOblika++;
        Text = "Broj iscrtanih pravougaonika je " + brojIscrtanihOblika.ToString();
        cetka.Dispose();
    }
}

```



Slika 10: Osobine komponente Timer



Slika 11: Postavljanje metode za događaj Tick

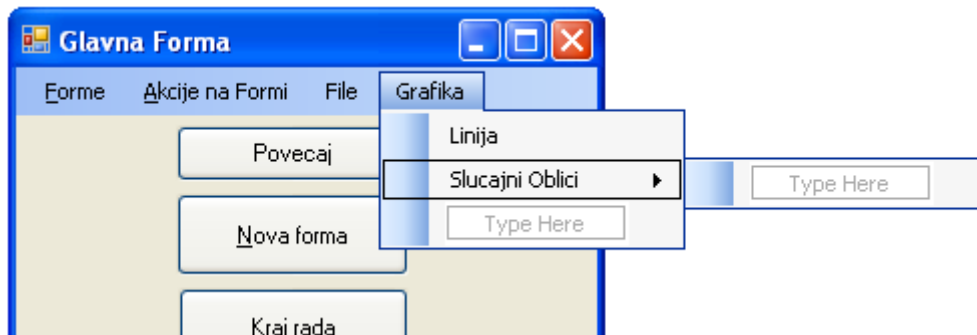
Da bi obezbedili iscrtavanje pravougaonika dodajmo novu stavku u meniju Grafika na glavnoj formi i nazovimo je **Slučajni oblici** (slika 12). Zatim za događaj Click vežimo kod koji obezbeđuje pojavljivanje forme SlucajniOblici tj.

```
private void slucajniObliciToolStripMenuItem_Click(object sender, EventArgs e)
{
    SlucajniOblici slucajniObliciForma = new SlucajniOblici();

    slucajniObliciForma.ShowDialog();
}
```

Po startovanju aplikacije i aktiviranja stavke **Slučajni oblici** u meniju **Grafika** pojaviće se novi prozor u kome će se iscrtavati pravougaonici na slučajan način a njihov broj će biti prikazan u naslovu prozora (slika 13).

Sve dok ne zatvorimo aplikaciju iscrtavaće se novi i novi pravougaonici a njihov broj će se prikazivati u naslovu prozora. Svaki novi pravougaonik će se pojavljivati u jednakom vremenskom intervalu tako da nam aplikacija može poslužiti i za "merenje vremena" odnosno prikaz vremena. Ovo nema daje ideju kako možemo da realizujemo analogni časovnik sa kazaljka. Uz prethodna znanja potrebno nam je i malo poznavanje trigonometrije.



Slika 12: Stavka Slučajni oblici u meniju Grafika



Slika 13: Slučajno iscrtavanje pravougaonika

Sistemsko vreme

Bez prikaza odnosno informacije o vremenu mnoge aplikacije jednostavno ne mogu da se zamisle. Rad sa vremenom koje obuhvata podatke o datumu i vremenu je omogućeno pomoću strukture **DateTime** koja je deo **System** prostora imena (namespace). Ova struktura sadrži sistemske vreme koje se dobija korišćenjem statičkog svojstva **Now**.

Ako se koriste metode **ToLongDateString()** i **ToLongTimeString()** dobija se datum odnosno vreme u obliku stringa. Datum je u formi dan, mesec, godina a vreme u formi sat, minuti i sekunde.

Kreirajmo novu formu koju ćemo da nazovemo **VremeForma** i obezbedimo prikazivanje sistemskog datuma i vremena. U tu svrhu na formu dodajmo četiri labela: za tekst „Datum“, za sam datum, za tekst „Vreme“ i za samo vreme. Imenujmo ih **tekstDatum**, **sistemskiDatum**, **textVreme** i **sistemskoVreme** redom i postavimo veličinu fonta na 24 a boju sistemskog vremena na crveno a boju sistemskog datuma na plavo kao što je to prikazano na slici 14. Može se primetiti da su inicijalne vrednosti za sistemsko vreme **00:00:00** a za datum **DD:MM:GGGG**.

Prikaz sistemskog vremena u trenutku kreiranja forme se postiže sledećim kodom

```
private void VremeForma_Load(object sender, EventArgs e)
{
    sistemskiDatum.Text = DateTime.Now.ToLongDateString();

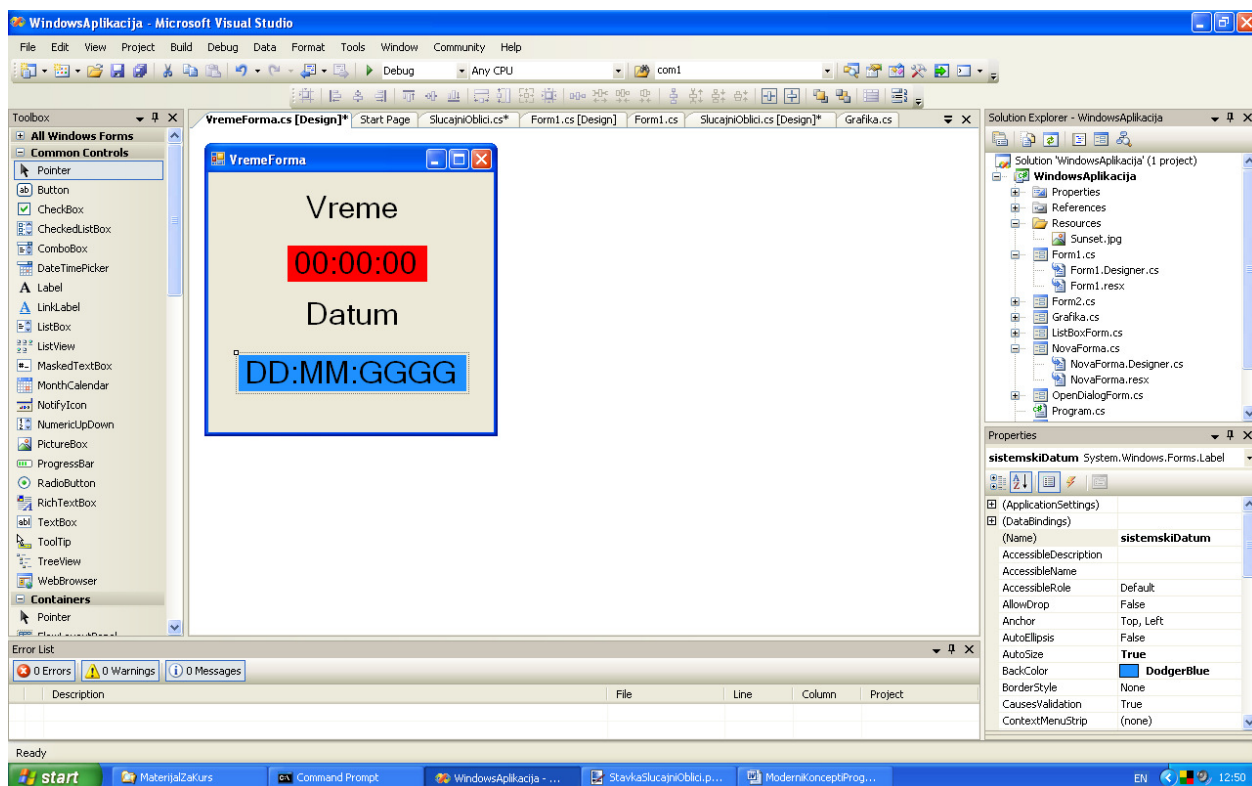
    sistemskoVreme.Text = DateTime.Now.ToLongTimeString();
}
```

Međutim, pored prikaza vremena u trenutku učitavanja forme želimo da forma prikazuje sistemsko vreme svo vreme dok je prikazana. Da bi to omogućili moramo da obezbedimo da se prikazano vreme „osvežava“ tako što će se u fiksним vremenskim intervalima uzimati nova vrednost sistemskog vremena i prikazivati. Očigledno je da nam je za to periodično očitavanje potrebna kontrola **Timer** za čiji događaj **Tick** ćemo vezati kod kojim se očitava sistemsko vreme i u skladu sa njim menjaju **Text** osobine labela **sistemskoVreme** i **sistemskiDatum**. Kako se očitavanje vremena i ispis vrše pri učitavanju forme vezaćemo za događaj **Tick** metodu **Load** za našu formu tj. metodu **VremeForma_Load**. Da bi se vreme očitavalo svake sekunde potrebno je da se osobina **Interval** kontrole **Timer** postavi na 1000, tako da će naš sat raditi sa tačnošću od jedne sekunde tj. menjaće se prikazano vreme svake sekunde.

Na glavnu formu dodajmo meni **Vreme** i stavku **Sistemsko vreme**. Za događaj Click vežimo prikazivanje forme **VremeForma** tj.

```
private void sistemskoVremeToolStripMenuItem_Click(object sender, EventArgs e)
{
    VremeForma formaZaVreme = new VremeForma();

    formaZaVreme.ShowDialog();
}
```



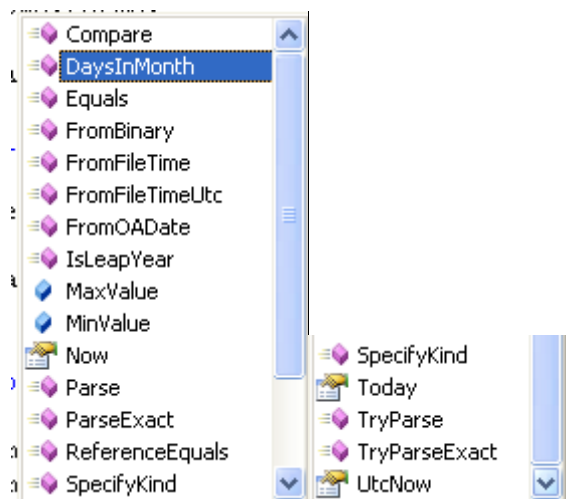
Slika 14: Forma Vreme.

Po startovanju aplikacije dobija se prozor kao na slici 15.

Pored metode Now klasa DateTime ima i druge korisne metode koje pružaju različite mogućnosti. Metode klase dateTime su prikazane na slici 16.



Slika 15: Forma za prikaz vremena i datuma



Slika 16: Metode klase DateTime.

U prethodnom primeru je ime meseca ispisano na engleskom jeziku. Kako bi mogli da modifikujemo naš primer pa da ispis meseca bude na srpskom jeziku? Šta treba uraditi da bi se datum ispisivao u obliku gde se mesec izražava brojem tj. 12.11.2009 ? Šta ako želimo da format datuma bude takav da se najpre prikazuje mesec pa dan pa godina?

Primer

Generišimo formu LevoDesnoForma na kojoj se nalazi 8 plavih kružnica i tri dugmeta kao što je to prikazano na slici 17. Kada se klikne na dugme Desno pojavljuje se crveni krug i počinje da se kreće u smeru kretanja kazaljke na satu. Pritisak na dugme Levo menja smer kretanja crvenog kruga i on postaje suprotan od smera kretanja kazaljke na satu. U meniju Grafika dodajmo stavku Levo Desno i povežimo je za prikazivanje forme LevoDesnoForma. Opisana funkcionalnost je postignuta sledećim kodom:

```
int pomeraj;  
float XCentar = 120;  
float YCentar = 100;  
int crveniKrug = 0;  
  
private void LevoDesnoForma_Paint(object sender, PaintEventArgs e)  
{  
  
    Graphics graf = CreateGraphics();  
    Pen olovka = new Pen(Color.Blue, 5);  
  
    graf.DrawEllipse(olovka, XCentar - 40, YCentar - 40, 40, 40);  
    graf.DrawEllipse(olovka, XCentar, YCentar - 40, 40, 40);
```

```

graf.DrawEllipse(olovka, XCentar + 40, YCentar - 40, 40, 40);
graf.DrawEllipse(olovka, XCentar + 40, YCentar, 40, 40);
graf.DrawEllipse(olovka, XCentar + 40, YCentar + 40, 40, 40);
graf.DrawEllipse(olovka, XCentar, YCentar + 40, 40, 40);
graf.DrawEllipse(olovka, XCentar - 40, YCentar, 40, 40);
graf.DrawEllipse(olovka, XCentar - 40, YCentar + 40, 40, 40);
olovka.Dispose();
}

```

```

private void buttonLevo_Click(object sender, EventArgs e)

```

```

{
    timer1.Enabled = true;
    pomeraaj = 7;
}

```

```

private void timer1_Tick(object sender, EventArgs e)

```

```

{
    Graphics graf = CreateGraphics();
    SolidBrush cetkaPozadina = new SolidBrush(this.BackColor);
    SolidBrush cetkaCrvena = new SolidBrush(Color.Red);
    Pen olovka = new Pen(Color.Blue, 5);
    // obrisi crveni krug a zatim iscrtaj kruznicu, a onda ofarbaj u crveno sledeci krug
    if (pomeraaj == 1)
    {
        // pomeranje u smeru kretanja kazaljke na satu

        switch (crveniKrug)
        {
            case 0:
                graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar - 40, 40, 40);
                graf.DrawEllipse(olovka, XCentar - 40, YCentar - 40, 40, 40);
                graf.FillEllipse(cetkaCrvena, XCentar, YCentar - 40, 40, 40);
                break;
            case 1:
                graf.FillEllipse(cetkaPozadina, XCentar, YCentar - 40, 40, 40);

```



```

    graf.DrawEllipse(olovka, XCentar, YCentar - 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar - 40, 40, 40);
    break;
case 2:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar - 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar - 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar, 40, 40);
    break;
case 3:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar + 40, 40, 40);
    break;
case 4:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar, YCentar + 40, 40, 40);
    break;
case 5:
    graf.FillEllipse(cetkaPozadina, XCentar, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar + 40, 40, 40);
    break;
case 6:
    graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar - 40, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar, 40, 40);
    break;
case 7:
    graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar, 40, 40);
    graf.DrawEllipse(olovka, XCentar - 40, YCentar, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar - 40, 40, 40);
    break;
}

}

else

{

    // pomeranje u smeru suprotnom od smera kretanja kazaljke na satu

    switch (crveniKrug)

    {

```

```

case 0:
    graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar - 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar - 40, YCentar - 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar, 40, 40);
    break;
case 1:
    graf.FillEllipse(cetkaPozadina, XCentar, YCentar - 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar, YCentar - 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar - 40, 40, 40);
    break;
case 2:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar - 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar - 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar, YCentar - 40, 40, 40);
    break;
case 3:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar - 40, 40, 40);
    break;
case 4:
    graf.FillEllipse(cetkaPozadina, XCentar + 40, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar + 40, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar, 40, 40);
    break;
case 5:
    graf.FillEllipse(cetkaPozadina, XCentar, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar + 40, YCentar + 40, 40, 40);
    break;
case 6:
    graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar + 40, 40, 40);
    graf.DrawEllipse(olovka, XCentar - 40, YCentar + 40, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar, YCentar + 40, 40, 40);
    break;
case 7:
    graf.FillEllipse(cetkaPozadina, XCentar - 40, YCentar, 40, 40);
    graf.DrawEllipse(olovka, XCentar - 40, YCentar, 40, 40);
    graf.FillEllipse(cetkaCrvena, XCentar - 40, YCentar + 40, 40, 40);
    break;
}

}

crveniKrug = (crveniKrug + pomeraj) % 8;
cetkaCrvena.Dispose();

```

```

        cetkaPozadina.Dispose();
        olovka.Dispose();
    }

    private void buttonDesno_Click(object sender, EventArgs e)
    {
        timer1.Enabled = true;

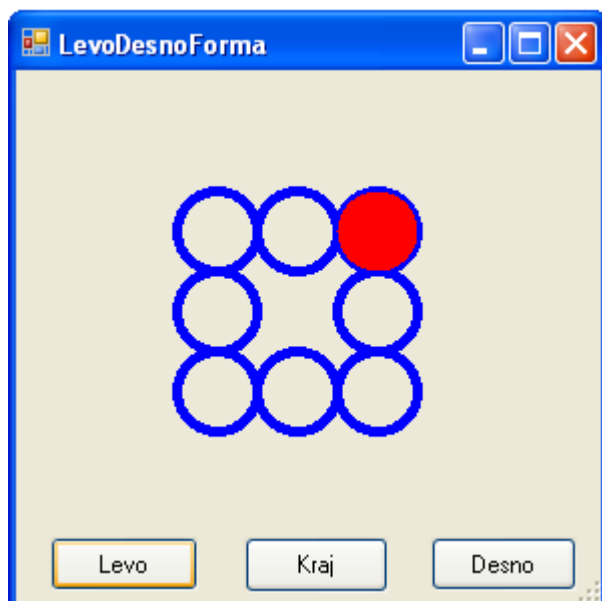
        pomeraj = 1;
    }

    private void buttonKraj_Click(object sender, EventArgs e)
    {
        this.Close();
    }

```

Analizirajmo kod. Da bi simulirali kretanje crvenog kruga potrebno je da u svakom koraku tj taktu časovnika koji postavljamo na formu izbrišemo trenutno crveni krug i ofarbamo sledeći u crveno. „Brisanje“ kruga ćemo izvesti zapravo farbanjem kruga u boju pozadine tj forme a nakon toga opet iscrtavamo plavu kružnicu. Smer kretanja crvenog kruga realizujemo pomoću promenljive pomeraj koja ima vrednost 1 za smer kretanja kazaljke na satu odnosno 7 za smer suprotan od smera kretanja kazaljke na satu. U svakom taktu se promenljiva crveniKrug menja za vrednost promenljive pomeraj po modulu 8 da bi obezbedili cikličnost.

Po startovanju programa u nekom koraku se dolazi u situaciju kao što je prikazano na slici 17.



Slika 17: Kretanje crvenog kruga

Kolekcije i njihova primena

Jedan od ozbiljnih problema u korišćenju nizova je njihova statičnost tj nakon kreiranja niza ne može se promeniti njegova veličina tako da je potrebno predvideti sve moguće buduće izmene kako bi se tačno odredila dimenzija niza što je veoma često nemoguće. Sledeći nedostatak je i nemogućnost direktnog ubacivanja odnosno brisanja elemenata. Elementima se pristupa samo preko numeričkog indeksa. Ovi problemi se rešavaju korišćenjem neke od postojećih klasa iz System.Collections koje omogućavaju kreiranje kolekcije elemenata čiji se kapacitet može promeniti i nakon njihovog kreiranja. Istovremeno pristup elementima kolekcije je fleksibilniji. Kako je pri kreiranju klasa kolekcija korišćena ista filozofija obradićemo samo neke. Ostale vrste kolekcija se po analogiji mogu veoma lako proučiti i koristiti. Kolekcije su:

- ArrayList (lista nizova)
- BitArray (niz bitova)
- Hashtable (heš tabela)
- SortedList (sortirana lista)
- Queue (red)
- Stack (magacin, stek)

ArrayList

Kolekcija ArrayList je veoma slična nizu s tim da ona ima dodatne mogućnosti od kojih je osnovna mogućnost automatskog proširenja pri dodavanju novih elemenata tj ona nema fiksnu veličinu kao niz. Bogat skup metoda ove klase omogućuje veoma jednostavnu

manipulaciju sa objektima ove klase. Moguće je na primer ubaciti element na bilo koju poziciju ili pak izbaciti element sa bilo koje pozicije.

Objekat klase ArrayList se kreira na sledeći način:

```
ArrayList lista = new ArrayList();
```

Očigledno je da se ovde nigde ne definiše koja je veličina niza tj koliko maksimalno elemenata može da ima niz. Dodavanje elementa u niz se postiže korišćenjem metode Add. Analizirajmo sledeći primer:

```
using System.Collections;

string IspisListe = "";
    ArrayList lista = new ArrayList();
    IspisListe = IspisListe + "kapacitet liste je " + lista.Capacity + " broj elemenata liste je " +
lista.Count;
    lista.Add("Jedan");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Drugi");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Treći");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Četvrti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Peti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Šesti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(2);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(3);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(4);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;

    MessageBox.Show(IspisListe);
```

Da kompajler ne bi prijavio grešku potrebno je uključiti biblioteku System.Collections. Najpre smo kreirali listu a potom su dodavani elementi. Nakon svakog dodavanja elementa u listu na standardnom izlazu je ispisivan kapacitet liste i broj elemenata u listi koji su dobijeni korišćenjem svojstava Capacity odnosno Count. Ako startujemo program dobija se sledeći izlaz:



Očigledno je da se kapacitet i broj elemenata u listi ne podudaraju uvek. Ono što je bitno je da ne postoji ograničenje za broj elemenata u listi.

Druga interesantna stvar je mogućnost dodavanja elemenata različitog tipa što je suprotno od mogućnosti koju pruža niz. Odakle ova mogućnost pravljenja nehomogenih kolekcija? Ako se ima u vidu da je kolekcija skup elemenata koji predstavljaju objekte klase System.Object i da su svi tipovi izvedeni iz ove klase jasno je zašto nema ograničenja po pitanju tipa elementa koji se dodaje ne samo ovoj kolekciji tipa ArrayList već i svim ostalim kolekcijama. Znači, kolekcije su zapravo skupovi elemenata tipa Object bez obzira kod tipa je element koji se dodaje kolekciji. Tako u našem primeru prvih šest elemenata koji su dodati u kolekciju su stringovi a preostala 3 su podaci tipa integer.

Dodajmo prethodnom kodu funkciju za prikaz elemenata liste

```
public static void PrikaziListu(string imeListe, ArrayList listaZaPrikaz)
{
    for (int indeks = 0; indeks < listaZaPrikaz.Count; indeks++)
    {
        Console.WriteLine(imeListe + "[" + indeks + "]= " + listaZaPrikaz[indeks]);
    }
}
```

i sledeće linije koje ilustruju korišćenje nekih metoda klase ArrayList:

```
string IspisListe = "";
```

```

private void btnArrayList_Click(object sender, EventArgs e)
{
    ArrayList lista = new ArrayList();
    IspisListe = IspisListe + "kapacitet liste je " + lista.Capacity + " broj elemenata liste je " +
lista.Count;
    lista.Add("Jedan");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Drugi");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Treći");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Četvrti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Peti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add("Šesti");
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(2);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(3);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;
    lista.Add(4);
    IspisListe = IspisListe + "\nkapacitet liste je " + lista.Capacity + " broj elemenata liste je "
+ lista.Count;

    PrikaziListu("lista", lista);
    ArrayList novaLista = new ArrayList(lista);
    PrikaziListu("Nova lista", novaLista);
    ArrayList listaBrojeva = lista.GetRange(6, 3);
    PrikaziListu("Lista brojeva", listaBrojeva);
    ArrayList listaStringova = lista.GetRange(0, 6);
    PrikaziListu("Lista stringova", listaStringova);

    novaLista.RemoveAt(5);
    PrikaziListu("Nova lista bez elementa sa indeksom 5", novaLista);
    novaLista.RemoveRange(2, 3);
    PrikaziListu("Nova lista bez elemenata sa indeksima 2,3 i 4 ", novaLista);
    PrikaziListu("Sortirana nova lista", novaLista);
}

```

```
    MessageBox.Show(IspisListe);
}
private string PrikaziListu(string imeListe, ArrayList listaZaPrikaz)
{
    for (int indeks = 0; indeks < listaZaPrikaz.Count; indeks++)
    {
        IspisListe = IspisListe + "\n" + imeListe + "[" + indeks + "]= " + listaZaPrikaz[indeks];
    }
    return IspisListe;
}
Console.WriteLine("'Drugi' se nalazi na poziciji " + novaLista.IndexOf("Drugi"));

novaLista.Sort();

PrikaziListu("Sortirana nova lista", novaLista);
```


kapacitet liste je 0 broj elemenata liste je 0
kapacitet liste je 4 broj elemenata liste je 1
kapacitet liste je 4 broj elemenata liste je 2
kapacitet liste je 4 broj elemenata liste je 3
kapacitet liste je 4 broj elemenata liste je 4
kapacitet liste je 8 broj elemenata liste je 5
kapacitet liste je 8 broj elemenata liste je 6
kapacitet liste je 8 broj elemenata liste je 7
kapacitet liste je 8 broj elemenata liste je 8
kapacitet liste je 16 broj elemenata liste je 9
lista[0]=Jedan
lista[1]=Drugi
lista[2]=Treći
lista[3]=Četvrti
lista[4]=Peti
lista[5]=Šesti
lista[6]=2
lista[7]=3
lista[8]=4
Nova lista[0]=Jedan
Nova lista[1]=Drugi
Nova lista[2]=Treći
Nova lista[3]=Četvrti
Nova lista[4]=Peti
Nova lista[5]=Šesti
Nova lista[6]=2
Nova lista[7]=3
Nova lista[8]=4
Lista brojeva[0]=2
Lista brojeva[1]=3
Lista brojeva[2]=4
Lista stringova[0]=Jedan
Lista stringova[1]=Drugi
Lista stringova[2]=Treći
Lista stringova[3]=Četvrti
Lista stringova[4]=Peti
Lista stringova[5]=Šesti
Nova lista bez elementa sa indeksom 5[0]=Jedan
Nova lista bez elementa sa indeksom 5[1]=Drugi
Nova lista bez elementa sa indeksom 5[2]=Treći
Nova lista bez elementa sa indeksom 5[3]=Četvrti
Nova lista bez elementa sa indeksom 5[4]=Peti
Nova lista bez elementa sa indeksom 5[5]=2
Nova lista bez elementa sa indeksom 5[6]=3
Nova lista bez elementa sa indeksom 5[7]=4
Nova lista bez elemenata sa indeksima 2,3 i 4 [0]=Jedan
Nova lista bez elemenata sa indeksima 2,3 i 4 [1]=Drugi
Nova lista bez elemenata sa indeksima 2,3 i 4 [2]=2
Nova lista bez elemenata sa indeksima 2,3 i 4 [3]=3
Nova lista bez elemenata sa indeksima 2,3 i 4 [4]=4
Sortirana nova lista[0]=Jedan
Sortirana nova lista[1]=Drugi
Sortirana nova lista[2]=2
Sortirana nova lista[3]=3
Sortirana nova lista[4]=4

OK

Hashtable

Kolekcija koja omogućuje pamćenje ključa i odgovarajuće vrednosti je kolekcija **Hashtable** (heš tabela). Razlika u odnosu na kolekciju ArrayList je u tome što se kod ArrayList element kolekcije određuje na osnovu mesta u listi tj. na osnovu indeksa elementa a kod Hashtable na osnovu ključa. Mehanizam je sličan kao kod rečnika nekog stranog jezika. Na osnovu strane reči u rečniku tražimo njeno značenje tj. ovde je reč ključ po kome se traži u rečniku a vrednost je samo značenje i opis reči.

Razmotrimo sledeći primer. Svaki veći grad ima svoju oznaku za registarske tablice. Na primer, za Niš je oznaka NI, za Beograd BG, itd. Sve gradove možemo da smestimo u kolekciju tipa Hashtable tako što ćemo za ključ da koristimo oznaku za registarske tablice a vrednost je naziv grada.

Da bi koristili kolekciju heš tabelu moramo da napravimo objekat klase Hashtable sa

```
Hashtable mojaHashtabela = new Hashtable();
```

Dodavanje elementa u haš tabelu može se izvršiti korišćenjem metode **Add**. Broj elemenata u heš tabeli može se dobiti pomoću atributa **Count**. Svojstvo **Keys** daje skup svih ključeva dok svojstvo **Values** daje skup svih vrednosti elemenata u haš tabeli.

Ako želimo da proverimo da li se element sa nekim ključem nalazi u heš tabeli napisaćemo

```
string vrednostElementa = (string) mojaHashtabela["nekiKljuc"];
```

Uočite da se dobijena vrednost konvertuje u string jer se pretraživanjem vraća objekat!

Vratimo se našem primeru sa registarskim tablicama i imenima gradova. Sledeći kod

```
string ispis = "";
```

```
private void btnHashTable_Click(object sender, EventArgs e)
{
    Hashtable mojaHashtabela = new Hashtable();
    mojaHashtabela.Add("NI", "Nis");
    mojaHashtabela.Add("BG", "Beograd");
    mojaHashtabela.Add("NS", "Novi Sad");
    mojaHashtabela.Add("SU", "Subotica");

    string mojGrad = (string)mojaHashtabela["NI"];
    ispis = ispis + "Moj grad je " + mojGrad;
    ispis = ispis + "\nBroj gradova u hash tabeli je " + mojaHashtabela.Count;
    if (mojaHashtabela.ContainsKey("NS"))
    {
```

```

        ispis = ispis + "\nhash sadrzi kljuc " + "NS";
    }

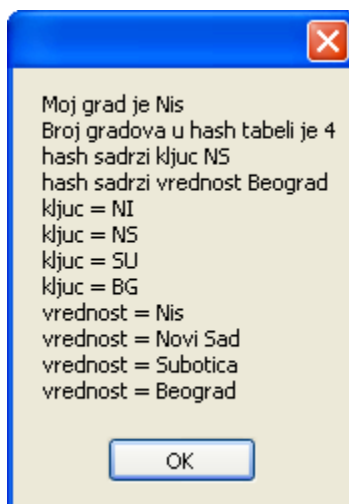
    if (mojaHashtabela.ContainsValue("Beograd"))
    {
        ispis = ispis + "\nhash sadrzi vrednost " + "Beograd";
    }

    foreach (string kljuc in mojaHashtabela.Keys)
    {
        ispis = ispis + "\nkljuc = " + kljuc;
    }

    foreach (string vrednost in mojaHashtabela.Values)
    {
        ispis = ispis + "\nvrednost = " + vrednost;
    }
    MessageBox.Show(ispis);
}

```

kao rezultat izvršenja daje izlaz prikazan na slici 2.



Slika 2: Haš tabela

Neke od metoda klase Hashtable su:

- **Clear** - uklanja element iz kolekcije
- **Clone** - kreira kopiju kolekcije
- **ContainsKey** - ispituje se da li se specificirani ključ nalazi u kolekciji
- **ContainsValue** - ispituje da li se specificirana vrednost nalazi u kolekciji
- **Remove** - uklanja iz kolekcije element sa specificiranim ključem