Saša Fajković

2014

Sadržaj:

1) Potrebno predznanje ... 5
2) OOP – Objektno Orijentirano Programiranje ... 6
3) Klase (razredi) i objekti (instance) ... 7
3. 1. Što su klase i objekti ... 7
3. 2. Kako se stvaraju klase ... 8
3. 3. Dodavanje članova klasi ... 10
3. 4. Dodavanje metoda klasi ... 11
3. 5. Nasljeđivanje klasa ... 11
4) Prava pristupa svojstvima i metodama: Internal, Public, Private, Protected, Protected Internal 13
4. 1. Teorija o pravu pristupa ... 13
4. 2. Internal pravo pristupa ... 13
4. 3. Public pravo pristupa ... 13
4. 4. Private pravo pristupa ... 13
4. 5. Protected pravo pristupa ... 14
4. 6. Protected Internal pravo pristupa ... 14
5) Konstruktor i Destruktor ... 15
5. 1. Općenito o konstrukteru ... 15
5. 2. Stvaranje konstruktora ... 15
5. 3. Pretpostavljeni konstruktor ... 16
5. 4. Preopterećenje konstruktora ... 16
5. 5. Destruktor ... 17
6) This - referenca na trenutnu instancu ... 18
7) Getters & Setters (Geteri i Seteri) ... 19
7. 1. Općenito i zašto ih koristiti ... 19
7. 2. Kako napraviti Getter i Setter svojstva ... 20
7. 3. Postavljanje „Read-Only“ svojstva ... 21
8) Windows Forms ... 22
8. 1. Što su Windows Forms ... 22
8. 2. Uređivanje svojstava ... 22
8. 3. Akcije/Događaji nad Windows Formama ... 23
8. 4. Brisanje metoda za neki događaj ... 23
9) Elementi na Windows Formi - uvod ... 24
9. 1. Što su elementi ... 24
9. 2. Dodavanje elemenata ... 25

9. 3. Svojstva elemenata 25
9. 4. Akcije/Događaji nad elementima 25
10) Svojstva i akcije elemenata na Windows Formi – nastavak (čitanje, pisanje i izmjena) 26
11) Iznimke u C# jeziku 27
11. 1. Što su iznimke 27
11. 2. Obrada iznimki (try / catch) 27
11. 3. Bacanje iznimki (throw) 29
12) Popis slika: 30

1) Potrebno predznanje

Prije proučavanja ovih materijala, pretpostavka je da ste dobro upoznati s radom i pojmovima poput varijable, funkcije, operatori, polja, petlje, grananje i kontrola toka. Također, očekuje se da imate osnovno znanje u radu s razvojnim sučeljima tj. da znate stvoriti novi projekt, otvoriti postojeći te da ste upoznati s osnovama rada na računalu (otvaranje mapa, dvostruki lijevi klik, desni klik, instalacija aplikacija i slično).

Poznavanje razreda String i pozivanja funkcija nad njim se također smatra usvojenim gradivom.

Ukoliko to nije slučaj, prvo utvrdite to gradivo, potom krenite na ove materijale jer se ovdje neće objašnjavati spomenuti pojmovi, već će se podrazumijevati da ste upoznati s njima te njihovom funkcionalnošću.

Svi primjeri će biti rađeni u Microsoft Visual Studio 2013 razvojnom sučelju. Ukoliko koristite neko drugo razvojno sučelje, princip rada je potpuno identičan i nećete imati problema.

Besplatne distribucije Microsoft Visual Studio razvojnog sučelja koje nose oznaku „***Express***“ su dostupne za preuzimanje sa službene Microsoft stranice:

http://www.visualstudio.com/downloads/download-visual-studio-vs

2) OOP – Objektno Orijentirano Programiranje

OOP je programska paradigma, odnosno način programiranja u kojem vlada koncept stvaranja objekata. Svaki objekt može imati svoje članove i funkcije koje možemo pozvati nad tim objektom. Funkcije koje pozivamo nad nekim objektom zovemo **metode**. Objekti nastaju kao **instance klase (razreda)**. Klasom odnosno razredom smatramo tipove podatka. Primjerice, String je razred, ali uz već postojeće tipove podataka, često imamo potrebu raditi vlastite tipove podataka.

Objektno orijentirani pristup omogućuje pisanje köda koji je lako izmjenjiv i nadogradiv te je köd moguće ponovno koristiti. Ovakav pristup programiranju donosi neka važna svojstva koja danas smatramo osnovnim načelima OOP-a. U ovom dijeli ćemo ih samo nabrojati, a naknadno i objasniti.

1. Nasljeđivanje (*engl. Inheritance*) – Iz već postojeće klase ili nove klase možemo stvoriti novu klasu sa svim članovima „nadklase", odnosno „izvedeni razred" nasljeđuje članove i metode iz „osnovnog razreda".
2. Enkapsulacija (*engl. Encapsulation*) – Skrivanje podataka odnosno zabrana pristupa podatcima jedne klase iz neke druge klase ili metode.
3. Polimorfizam (*engl. Polymorphism*) – U kratko rečeno, polimorfizam predstavlja značenje „jedno ime -> više oblika". To znači da možemo imati više funkcija s identičnim imenom no s manjom ili većom razlikom u funkcionalnosti.

3) Klase (razredi) i objekti (instance)

3. 1. Što su klase i objekti

Klasa odnosno razred predstavlja tip podatka. Navikli smo na tipove podataka poput *Integera* koji predstavljaju cijele brojeve, ili recimo *charactere* koji predstavljaju jedan znak. No što ako želimo raditi s kompleksnim brojevima[1]. U tom slučaju možemo definirati vlastite tipove podataka tako da napravimo klasu koja će imati svojstvo realne i imaginarne komponente. Do sada ste prilikom pisanja varijabli vjerojatno radili postupak da prvo upišete tip podatka varijable koju stvarate, zatim njeno ime te ste eventualno pridijelili neku vrijednost toj varijabli -> *Int broj = 5*.

Napravimo li klasu kompleksnih brojeva u kojoj definiramo da varijable koje će biti tipa „Kompleksni" sadrže realnu i imaginarnu komponentu mogli bi napraviti ovako nešto: *Kompleksni broj = 12+24i.* U ovom slučaju, imamo tip podatka koji zna prepoznati realne i imaginarne komponente kompleksnog broja. Također, takvim razredima možemo dodati funkcije za zbrajanje, oduzimanje, množenje i dijeljenje kompleksnih brojeva.

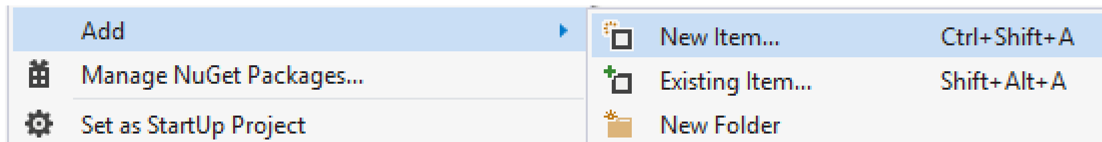
Kako bi lakše shvatili apstraktnost klase, pogledajmo na primjeru realnog svijeta. Zamislite da imate tip podatka „Vozilo". Za sada se ograničavamo samo na automobile. Svako vozilo može imati različite članove poput broja kotača, vrsta motora (benzin ili dizel), manualni ili automatski mjenjač, dužina, širina, visina, boja i ostalo. U programerskom svijetu, to znači da možemo stvoriti klasu „Vozilo" sa istim tim svojstvima. Zatim možemo napraviti varijablu koja je po tipu podataka „Vozilo". Primjer stvaranja takve varijable bi bio: „Vozilo autoBroj1;". Stvorili smo varijablu imena „autoBroj1" koji je po tipu „Vozilo". Kada stvorimo varijablu nekog takvog tipa, kažemo da smo stvorili **objekt** odnosno **instancu klase**.

Kako znamo da klasa „Vozilo" unutar sebe sadrži svojstva poput boje, dužine, širine i ostalih, ista ta svojstva se primjenjuju na varijablu „autoBroj1". Ako bi napisali „autoBroj1.boja = „crvena", tada bismo za tu varijablu definirali svojstvo „boja" i tom svojstvu dodijelili vrijednost „crvena". Isto tako, možemo stvoriti i drugi objekt „autoBroj2" koji će imati sva svojstva koja ima i klasa „Vozilo". Sada tom auto možemo definirati njegova svojstva. „autoBroj2.boja = „plava" će tom objektu definirati svojstvo „boja" s dodijeljenom vrijednosti „plava".

[1] Kompleksni broj se sastoji od realnog i imaginarnog dijela. Primjerice 3+2i ili 4-5i. Slovo „i" obilježava imaginarnu komponentu.

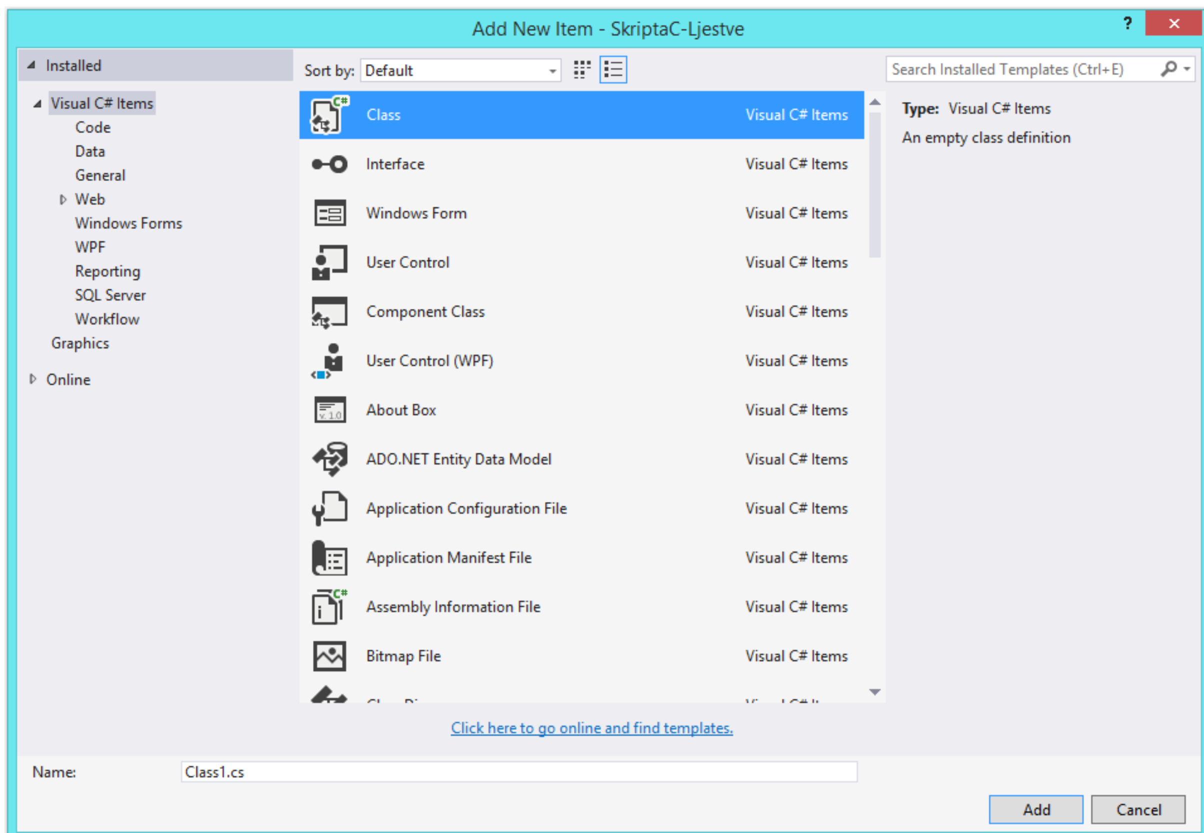
3. 2. Kako se stvaraju klase.

U Visual Studio razvojnom sučelju, unutar *Solutiona* na kojem radimo napravimo desni klik, odaberemo *Add* i zatim „*New Item*“.



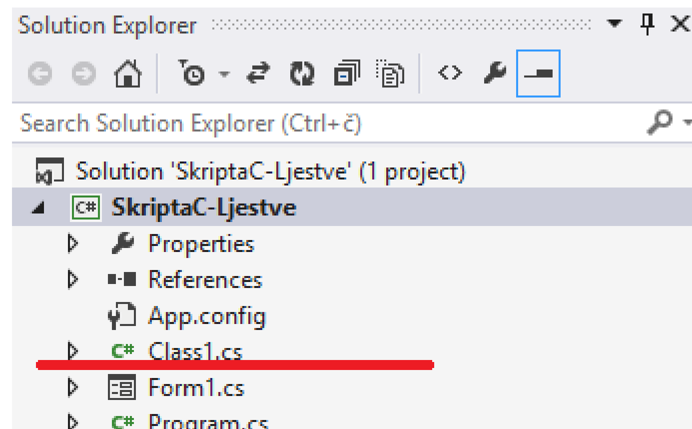
Slika 1 - Dodavanje nove klase

Otvara se izbornik za dodavanje datoteka.



Slika 2 - Izbornik za dodavanje klase

Obavezno mora biti odabrana datoteka „*Class*“. U donjem dijelu upisujemo ime klase koju želimo dodati. Nakon upisa imena, odabiremo tipku „*Add*“ te će se nova klasa pojaviti unutar *Solutiona.*



Slika 3 - Prikaz novo dodane klase u Solution Exploreru

Visual Studio će također odmah i otvoriti tu klasu za uređivanje.

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace SkriptaC_Ljestve
 8  {
        0 references
 9      class Class1
10      {
11      }
12  }
13
```

Slika 4 - Početni kôd nakon dodavanja nove klase

Pripazite da se vaša klasa nalazi unutar *Namespacea* s nazivom jednakim projektu (.cs ekstenzija) u kojem radite. Stvaranje klase započinjemo pisanjem ključne riječi „*class*“ nakon čega slijedi ime klase koju stvaramo. U programskom jeziku C#, konvencija je da se klase pišu velikim početnim slovom. Ukoliko se vaša klasa sastoji od više riječi, pišite ih sve spojeno s tim da svaku riječ započnete velikim slovom. Primjerice „**P**rva**T**estna**K**lasa“. Izbjegavajte pisanje imena klasa koja započinju brojevima i simbolima. Imajte na umu da će vam „*Intelli-Sense*“ uvijek nadopunjavati imena pa nije problem napisati dugačko ime klase.

3. 3. Dodavanje članova klasi

Kada stvaramo klasu, moramo imati na umu da stvaramo novi tip podataka. Kao takav, taj tip podataka je zapravo kompleksan tip podataka. Uostalom, da nam treba samo primitivan tip podataka (*Integer, Float, Double, Character,* ...) ne bismo niti radili novi tip podataka. Kompleksni tipovi podataka se sastoje od jednostavnih tipova podataka. Upravo te vodilje se držimo i prilikom izrade vlastitih klasa. Unutar vitičastih zagrada koje obilježavaju početak i kraj neke klase odnosno bloka naredbi, pišemo članove (i metode) te klase.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace SkriptaC_Ljestve
8  {
       2 references
9      public class Vozilo
10     {
11         public string boja;
12         public float visina;
13     }
14 }
15
```

Slika 5 - Dodavanje svojstava klasi

Vidimo kako su članovi napisani kao obične varijable nekog određenog primarnog tipa (izuzetak je *String* tip podataka koji sam po sebi nije primarni tip podataka, no sigurni smo da će *Compiler* znati raditi s takvim tipom podataka).

Sada smo stvorili klasu Vozilo koja ima članove „boja“ i „visina“. Napravimo li instancu te klase odnosno novi objekt koji je tipa „Vozilo“, imati ćemo na raspolaganju ta dva svojstva. Svojstvima (i metodama) pristupamo tako da iza objekta upišemo simbol točke nakon čega će se pojaviti dostupna svojstva i metode kojima zatim standardno pomoću operatora pridruživanja (znak jednakosti) pridjeljujemo neku vrijednost.

3. 4. Dodavanje metoda klasi

Na isti način kako smo dodavali i svojstva klasi, dodajemo i funkcije. Kada je neka funkcija dodana klasi, odnosno vezana za tu klasu, tada se to zove **metoda.** Metoda pozivamo nad objektima odnosno instancama neke klase.

```
7   namespace SkriptaC_Ljestve
8   {
        2 references
9       public class Vozilo
10      {
11          public string boja;
12          public float visina;
13
            1 reference
14          public string DohvatiBoju()
15          {
16              return boja;
17          }
18
19      }
20
```

Slika 6 - Dodavanje metoda klasi

Ovdje je napravljena metoda koja se zove „DohvatiBoju()“. Ta metoda ne prima niti jedan parametar i vraća ono što je pohranjeno u svojstvu „boja“ klase „Vozilo“.

3. 5. Nasljeđivanje klasa

Mogućnost nasljeđivanja klasa nam dopušta da stvorimo glavnu klasu iz koje izvodimo druge klase. Izvedene klase poprimaju svojstva glavne klase. Primjerice, imamo li klasu „Vozilo“ s elementom „brojKotaca“ koje obilježava broj kotača vozila i „duzina“ koji obilježava dužinu vozila, možemo napraviti dvije naslijeđene klase, jednu „Automobil“ i drugu „Motocikl“. Unutar svake od tih klasa ćemo definirati zasebna njihova svojstva koja su karakteristična samo za njih, no osnovna svojstva koja su zajednička motociklu i automobilu ćemo naslijediti iz klase Vozilo.

```
0 references
public class Automobil : Vozilo
 {
     int brojVrata;
 }
```

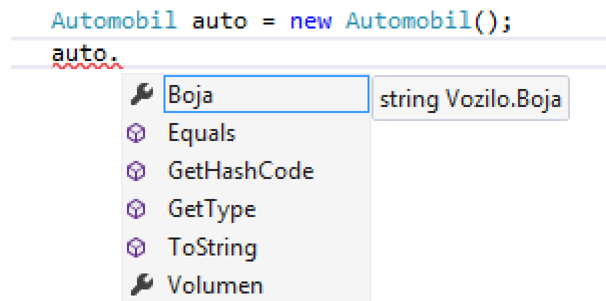
Slika 7 - Nasljeđivanje klasa

Sintaksa za nasljeđivanje je : (eventualno pravo pristupa) – ključna riječ „*class*“ - nova klasa koju stvaramo – simbol dvotočke (:) – ime klase koju nasljeđujemo.

```
public class Vozilo
 {
   0 references
   public Vozilo()[...]
    1 reference
    public Vozilo (string dodijeliBoju)[...]
    0 references
    public Vozilo (string dodijeliBoju, float dodijeliVisinu)[...]

     string boja;

     float visina;
     float duzina;
     float sirina;

     float volumen;
    0 references
    public float Volumen[...]

     2 references
     public string Boja[...]
 }
```

Slika 8 - Klasa Vozilo

Vidimo da unutar klase „Vozilo“ imamo elemente boja, visina, duzina, sirina i volumen. Svim svojstvima možemo pristupiti i unutar naslijeđene klase „Automobil“. Imajte na umu da možemo pristupiti samo svojstvima klase, u našem slučaju to su „Volumen“ i „Boja“.



Slika 9 - Pristup svojstvima nasljeđenje klase

4) Prava pristupa svojstvima i metodama: Internal, Public, Private, Protected, Protected Internal

U prethodnom ködu ste mogli uočiti ključnu riječ „*public*“, no nigdje nije bilo objašnjenja za nju. U ovom poglavlju naučit ćemo što je ta riječ te koje su još takve riječi i njihovu ulogu.

4. 1. Teorija o pravu pristupa

Klasama, njenim članovima i metodama možemo dodijeliti pravo pristupa koje se razlikuje kroz tri stupnja. Primjerice, želimo onemogućiti pristup nekim varijablama iz neke druge klase. S druge strane, možda ciljano želimo napraviti klasu čijim svojstvima i metodama želimo dati pravo pristupa iz svih ostalih klasa.

4. 2. Internal pravo pristupa

Klasu, svojstva i metode smo pisali s prefiksom „*public*“ kako bi joj mogli pristupiti iz glavne klase koju je Visual Studio generirao (kao i popratne datoteke). Pretpostavljeno (*engl.* ***Default***) pravo pristupa klasi koja se NE NALAZI unutar druge klase je ***Internal***.

Internal omogućava pristup klasi iz istog „*Assembly*“ ali ne i iz vanjskih. U našem slučaju, klasi možemo pristupiti iz ostalih klasa ukoliko se nalaze unutar istog *Solutiona*.

4. 3. Public pravo pristupa

Public (javno) pravo pristupa će omogućiti pristup klasi od bilo kuda. Takvoj klasi možemo pristupiti iz istog projekta, ali i iz drugih projekata, odnosno iz istog *Assembly-a*, ali i iz ostalih.

Pojednostavljeno – ovakvim klasama možemo pristupiti iz cijelog *Solutiona*.

4. 4. Private pravo pristupa

Private (privatno) pravo pristupa će dopustiti pristup klasi samo u ködu unutar te klase.

Pojednostavljeno – ovakvim klasama možemo pristupiti samo iz te same klase. Njenim metodama i članovima ne možemo pristupiti iz drugih klasa.

4. 5. Protected pravo pristupa

Protected (zaštićeno) pravo pristupa dopušta pristup klasi samo iz te same klase ili iz klase koja je stvorena iz klase koja ima zaštićeno pravo pristupa.

Pojednostavljeno – ovakvim klasama možemo pristupiti samo iz te same klase ili klase koju napravimo da nasljeđuje takvu klasu.

4. 6. Protected Internal pravo pristupa

Protected Internal (Zaštićeno privatno) pravo pristupa dopušta pristup klasi samo iz tog *Assembly*a i iz naslijeđenih klasa.

Pojednostavljeno – spoj prava pristupa *Protected* i *Internal*.

Klase i strukture mogu imati samo *Public* i *Internal* prava pristupa. Elementi unutar klase mogu imati sva od navedenih prava pristupa.

5) Konstruktor i Destruktor

5. 1. Općenito o konstrukteru

Konstruktor je metoda (funkcija unutar klase) koju pozivamo kada želimo stvoriti novu instancu te klase (novi objekt). Korištenjem konstruktora, prilikom stvaranja nove instance klase, možemo dodijeliti vrijednosti elementima klase koje će taj objekt poprimiti odmah pri stvaranju. Ukoliko ne navedemo nikakav konstruktor, koristit će se pretpostavljeni konstruktor. Naravno, sami možemo kreirati vlastite konstruktore.

Unutar jedne klase možemo imati više konstruktora!

5. 2. Stvaranje konstruktora

Konstruktor pišemo kao i bilo koju drugu metodu, unutar klase. Kako bi napravili konstruktor, ta metoda mora imati identično ime kao i sama klasa. Pravo pristupa koje dodjeljujemo konstruktoru je P*ublic*. Imajte cijelo vrijeme na umu, ako ne napravite svoj konstruktor, pozivat će se pretpostavljeni konstruktor.

Primijetite kako konstruktor, za razliku od svih ostalih funkcija nema specifikaciju povratnog tipa. Jednostavno prvo napišemo pravo pristupa (*Public*), zatim ime konstruktora te eventualne parametre unutar oblih zagrada (kao i kod ostalih metoda). Unutar vitičastih zagrada pišemo daljnji köd koji će se izvršiti prilikom stvaranja instance te klase.

Imajte na umu da pretpostavljeni konstruktor nema nikakvih parametara (unutar oblih zagrada)

```
public class Vozilo
 {

    1 reference
    public Vozilo()
     {
         boja = "crvena";
     }

      public string boja;
      public float visina;
```

Slika 10 - Stvaranje konstruktora

5. 3. Pretpostavljeni konstruktor

Prilikom stvaranja instanci klase, poziva se pretpostavljeni konstruktor ukoliko drugačije nije specificirano. Pretpostavljeni konstruktor će stvoriti elemente te klase koje potom možemo pozvati nad objektom te klase.

Savjet:

Za brže pisanje konstruktora možete upisati ključnu riječ „*ctor*“ i pritisnuti tipku „Tab“ dva puta te će Visual Studio sam generirati pretpostavljeni konstruktor.

5. 4. Preopterećenje konstruktora

Ukoliko prilikom stvaranja instance neke varijable želimo postaviti vrijednosti elemenata te klase na neke određene vrijednosti, tada pozivamo konstruktor namijenjen tome. Unutar jedne klase možemo imati više konstruktora. Svi konstruktori se „zovu“ isto, odnosno ime im mora biti identično kao i ime klase. Jedina razlika je u broju i/ili poretku parametara koje prima konstruktor.

```
5 references
public class Vozilo
 {
    0 references
    public Vozilo()
     {

     }

     1 reference
     public Vozilo (string dodijeliBoju)
    {
        boja = dodijeliBoju;
    }

     0 references
     public Vozilo (string dodijeliBoju, float dodijeliVisinu)
     {
         boja = dodijeliBoju;
         visina = dodijeliVisinu;
     }

      public string boja;
      public float visina;
```

Slika 11 - Preopterećenje konstruktora

Vidimo kako postoji prvi konstruktor „*Vozilo()*“ koji ne dodjeljuje nikakve vrijednosti elementima klase. Drugi konstruktor „*Vozilo (string dodijeliBoju)*“ će postaviti vrijednost elementa (varijable) „*boja*“ na vrijednost jednaku parametru „*dodijeliBoju*“. Treći konstruktor „*Vozilo(string dodijeliBoju, float dodijeliVisinu)*“ će dodijeliti vrijednosti za dva elementa klase. Element „*boja*“ će poprimiti vrijednost jednaku parametru „*dodijeluBoju*“, a element „*visina*“ će poprimiti vrijednost jednaku parametru „*dodijeliVisinu*“.

Kako se pozivaju konstruktori i stvaraju objekti ćemo upoznati u daljnjem tekstu

5. 5. Destruktor

Suprotno od konstruktora, destruktore koristimo kada želimo uništiti instancu neke klase. Pritom moramo pratiti neka pravila jezika C#.

- Destruktor ne smije biti definiran unutar struktura, već samo unutar klasa
- Svaka klasa smije imati najviše jedan destruktor
- Destruktori nemaju svojstvo nasljeđivanja niti preopterećivanja
- Destruktor nema pravo pristupa niti parametre

S obzirom da C# jezik ne zahtijeva kontrolu nad memorijom poput nekih drugih programskih jezika, konstruktorima se nećemo posvetiti. Detalje oko konstruktora možete pročitati ovdje:

http://msdn.microsoft.com/en-us/library/66x5fx1b.aspx

6) This - referenca na trenutnu instancu

Želimo li biti sigurni da pristupamo svojstvu instance nad kojom pozivamo neku metodu primjerice, koristimo ključnu riječ „*this*“ koja omogućava referenciranje na trenutnu instancu klase.

```
public class Vozilo
 {
    0 references
    public Vozilo()
     {

     }

     1 reference
     public Vozilo (string dodijeliBoju)
    {
        this.boja = dodijeliBoju;
    }

     0 references
     public Vozilo (string dodijeliBoju, float dodijeliVisinu)
     {
         this.boja = dodijeliBoju;
         this.visina = dodijeliVisinu;
     }
```

Slika 12 - Ključna riječ "this"

7) Getters & Setters (Geteri i Seteri)

7. 1. Općenito i zašto ih koristiti

Prilikom stvaranja elementa klase, praksa je ograničiti direktno pravo upisa odnosno izmjene vrijednosti tih elemenata. Ovo se pokazalo kao dobra ideja kako ne bi došlo do slučajnog mijenjanja tih vrijednosti „negdje kasnije“ u ködu, a također se štitimo od nevaljanog unosa vrijednosti prilikom samog stvaranja varijable.

Za to koristimo metode (funkcije) koje pozivamo nad svakim elementom. Unutar tih metoda navodimo želimo li imati pravo samo čitanja, samo pisanja/izmjene podataka ili oboje. Takvim načinom pisanja köda ne pristupamo direktno nekom elementu već prvo pozivamo metodu klase koja ima pravo pristupa tim elementima i preko te metode utječemo na vrijednosti elemenata.

Takve metode se popularno zovu **Getteri** i **Setteri**. Gettere koristimo za čitanje odnosno dohvaćanje vrijednosti nekog elementa, a *Settere* za postavljanje/izmjenu vrijednosti nekog elementa.

Također, ovakvim pristupom možemo zaštiti aplikaciju od rušenja izazivanjem iznimki prilikom nevaljanog unosa i obradom tih iznimaka. Iznimke ćemo obraditi naknadno.

7. 2. Kako napraviti Getter i Setter svojstva

```
public class Vozilo
 {
   0 references
   public Vozilo()[...]
    1 reference
    public Vozilo (string dodijeliBoju)[...]
    0 references
    public Vozilo (string dodijeliBoju, float dodijeliVisinu)[...]

     string boja;
     float visina;

     2 references
     public string Boja
     {
         get
         {
             return boja;
         }
         set
         {
             boja = value;
         }
     }
 }
```

Slika 13 - Izrada Getter i Setter svojstava

Vidimo da unutar klase „*Vozilo*“ imamo tri konstruktora s kojima smo se već upoznali. Ispod toga se nalaze dva elementa; „visina“ i „*visina*“. Pretpostavljeno pravo pristupa za elemente je kao i za klase – ***Internal*.** Pokušamo li pristupiti iz druge klase elementima instance proizašle iz klase „*Vozilo*“ to neće biti moguće. Na ovaj način smo se zaštitili od potencijalno neželjenog upisa i promjena vrijednosti tih elemenata.

Ispod deklaracije ta dva elementa nalazimo svojstvo „*Boja*“. Primijetite da je to svojstvo pisano **velikim početnim slovom**. Ovo je konvencija da se svojstvo za neki element nazove kao i taj element, samo početnim velikim slovom (elementi se naravno po konvenciji nazivaju s malim početnim slovom).

Unutar svojstva „*Boja*“ nalazimo dvije ključne riječi; „*get*“ i „set“. „*Get*“ svojstvo koristimo za dohvaćanje podataka, a to smo napravili korištenjem ključne riječi „*return*“ te imena elementa. Svojstvo „*set*“ koristimo za dodjelu ili izmjenu vrijednosti nekog elementa. Unutar „*set*“ dijela izvršili smo dodjelu vrijednosti elementu „*boja*“. Ovdje se pojavljuje nova ključna riječ „*value*“. „***Value***“ dohvaća vrijednost koju pridjeljujemo elementu

7. 3. Postavljanje „Read-Only“ svojstva

Read-only svojstvo omogućava zaštitu od upisa vrijednosti nekom elementu klase, odnosno ima samo *get* dio, ali ne i *set* dio. Samo *get* svojstvo postavljamo kada želimo onemogućiti upis u neku varijablu, a to su elementi čija se vrijednost izračunava iz nekih drugih elemenata.

Primjerice:

```
 float visina;
 float duzina;
 float sirina;

 float volumen;
0 references
public float Volumen
{
    get
    {
        return duzina * sirina * visina;
    }
}
```

Slika 14 - Read-only svojstvo

Vidimo da vrijednost za „volumen“ ne želimo upisivati već računati na osnovi drugih vrijednosti. U tom slučaju, za element „volumen“ ne želimo dopustiti pravo upisa već samo čitanja vrijednosti.

8) Windows Forms

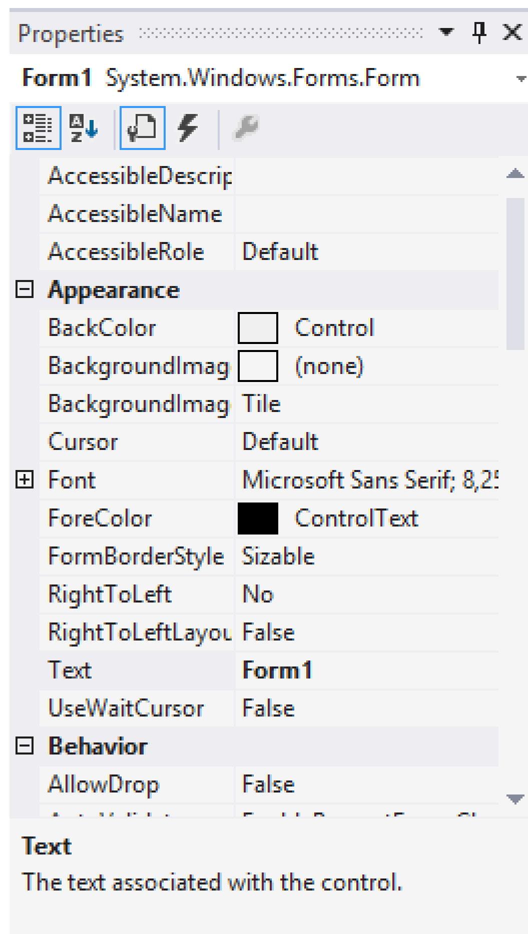
8. 1. Što su Windows Forms

Windows Forms (forme) je naziv za izradu grafički aplikacija čija je podrška sadržana unutar Microsoft .NET tehnologije. Korištenjem ove tehnologije (*frameworka*), omogućena je izrada nativnih (*native*) Microsoft aplikacija koje smo navikli viđati kroz Microsoft Windows operativni sustav.

Windows Forme su instance neke klase (konkretno *Form* klase) i kao takve podliježu pravilima OOP-a.

8. 2. Uređivanje svojstava

Svaka forma koju stvorimo ima svoja određena svojstva poput dimenzija, boje, naziva, teksta koji se ispisuje u zaglavlju forme itd. Svojstvima neke forme pristupamo preko *Properties* prozora.

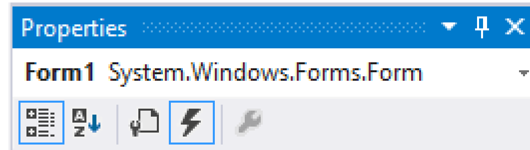


Slika 15 - Svojstva Windows Formi

Svako svojstvo nudi različite mogućnosti. Primjerice, svojstvo „*Texti*" se odnosi na natpis u zaglavlju forme, a svojstvo „(*Name*)" na sam naziv te forme. Ulaziti u detalje i opisivati svako pojedino svojstvo nema smisla te preporučam da ih unutar nekoliko minuta sami isprobate sve.

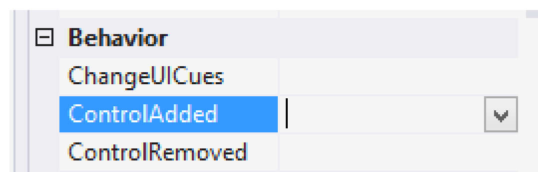
8. 3. Akcije/Događaji nad Windows Formama

Događaji (*events*) su radnje koje se mogu izvršiti nad nekim elementom, odnosno u ovom slučaju nad Windows Formom. Unutar *Properties* prozora se potrebno prebaciti na prikaz mogućih događaja.



Simbol za prikaz događaja je u obliku munje. Sada vidimo popis dostupnih događaja nad formom kojih je također izuzetno puno. Nema smisla ih učiti sve napamet jer većina događaja jako dobro svojim imenom opisuje na što se odnose. U konačnici, ukoliko ne možete pronaći neki događaj ili niste sigurno na što se odnosi, upišite na Google (ili neku Internet tražilicu) što vas zanima i odgovor će vrlo vjerojatno biti unutar prvih par ponuđenih linkova.

Kako bi pristupili nekom događaju, tj. kreirali köd koji se treba izvršiti kada dođe do nekog događaja, jednostavno napravimo „dvoklik" desno od imena događaja i Visual Studio će generirati metodu koja se poziva prilikom tog događaja. Unutar te metode pišemo köd koji se treba izvršiti za zadani događaj.



Slika 16 - Pristup događaju

```
1 reference
private void Form1_ControlAdded(object sender, ControlEventArgs e)
{
    
}
```

Slika 17 - Metoda koja se poziva prilikom događaja (ControlAdded)

8. 4. Brisanje metoda za neki događaj

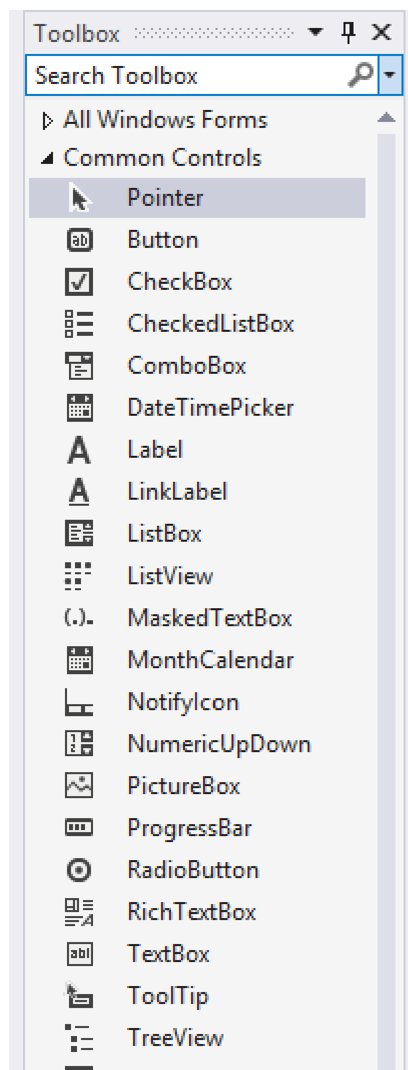
Metodu za neki događaj brišemo unutar köda gdje je metoda generirana, no pojavit će se poruka o pogrešci. Ovo je sasvim normalno. Nakon što se prikaže obavijest da imate greške u projektu, odaberete opciju „*No*" odnosno da ne želite nastaviti daljnje izvođenje te pogledajte pogreške koje se pojavljuju. Napravite „dvoklik" na pogrešku koja će vas odvesti u dio köda „dizajner" dijela köda za vašu formu te pobrišite svaku liniju u kojoj se nešto „crveni". To su linije koje se odnose na obrisane metode odnosno događaje, a pošto ih više nema, potrebno ih je i ukloniti iz „dizajnera".

9) Elementi na Windows Formi - uvod

9. 1. Što su elementi

Elementi forme (kontrole) su ono što postavimo na formu. To mogu biti polja za unos teksta, razni gumbi, kontrola za odabir datuma i još mnoštvo drugih. Kontrole se nalaze unutar prozora *Toolbox*. Ukoliko neki prozor nije vidljiv, potrebno je otići pod *VIEW* i upaliti željeni prozor.

Elementi su zapravo instance različitih klasa i kao takvi podliježu pravilima OOP-a.



Slika 18 - Toolbox

9. 2. Dodavanje elemenata

Elemente dodajemo na formu jednostavnim „*drag-n-drop*“ principom ili „dvoklikom“ na element. Elemente možemo razmještati po formi, prilagoditi im širinu i visinu, upisati neki tekst u njih i još mnoštvo toga.

9. 3. Svojstva elemenata

Svaki element ima svoja određena svojstva koja možemo mijenjati. Logično je da će prozor za sliku imati drugačija svojstva od primjerice polja za upis teksta. Svojstvima svakog elementa pristupamo preko *Properties* prozora, kao što smo vidjeli za Windows Forms objekt.

9. 4. Akcije/Događaji nad elementima

Kao i kod Windows Formsa, i nad elementima se može kontrolirati izvršavanje köda na neki događaj. Popisu događaja pristupamo preko *Properties* prozora, odaberemo prikaz događaja te napravimo „dvoklik“ desno od imena događaja za koji želimo napisati köd (identičan postupak kao i za Windows Forms objekte).

10) Svojstva i akcije elemenata na Windows Formi – nastavak (čitanje, pisanje i izmjena)

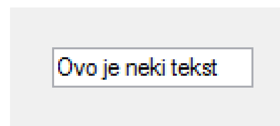
Često želimo svojstva nekog elementa mijenjati unutar samog köda. Primjerice, neku kontrolu želimo postaviti da bude nevidljiva kada se aplikacija pokrene, no u slučaju da se izvrši neka određena akcija, tu kontrolu možda želimo privremeno prikazati na formi.

Svojstvima nekog elementa pristupamo kao i kod ostalih instanci neke klase (elementi na formi su zapravo instance neke klase), upišemo simbol točke nakon imena kontrole i pojavit će se popis dostupnih svojstava.

```
txtVozilo.Text = "Ovo je neki tekst";
```

Slika 19 - Pristup svojstvima elementa unutar köda

U gornjem primjeru imamo Text Box kontrolu čije ime je „txtVozilo“. Svaka takva kontrola ima svojstvo „*Text*“ koje obilježava tekst upisan u tu kontrolu. S ovim ködom pristupamo tom svojstvu i postavljamo vrijednost svojstva na „Ovo je neki tekst“.



Slika 20 - Upis u Text svojstvo, kontrole textBox

Na isti način možemo dohvatiti vrijednost nekog svojstva.

```
string vrijednost = txtVozilo.Text;
```

Slika 21 - Dohvaćanje vrijednosti svojstva Text, kontrole textBox

11) Iznimke u C# jeziku

11. 1. Što su iznimke

Iznimke se događaju u slučaju pogrešaka. Najbanalniji primjer (ali na žalost učestao slučaj) je kada očekujemo da će korisnik u *textBox* kontrolu unijeti broj jer smo u *label* kontroli pored toga napisali „Unesi prvi broj“ i cijela aplikacija je zamišljena kao kalkulator. Naravno da će s vremenom doći do pogrešnog unosa podatka (namjerno ili slučajno).

Pokušamo li u tom slučaju pomnožiti tu vrijednost s nekim brojem, nailazimo na problem. Isto će se dogoditi pokušamo li takav podataka prebaciti u recimo *Integer* korištenjem metode *Parse()* i predati nekoj metodi. Aplikacije će se srušiti, a mi se želimo zaštiti od toga.

Ovdje nam pomažu iznimke (*Exception*) koje je potrebno „obraditi“.

Imajte na umu da postoji nekoliko glavnih kategorija iznimki te da različite metode stvaraju različite vrsti iznimki. Detaljne specifikacije možete pronaći na službenoj Microsoft stranici.

11. 2. Obrada iznimki (try / catch)

Kada imamo situaciju da može doći do pogrešnog prenošenja vrijednosti (primjerice predavanja teksta u metodu koja očekuje numerički podatak), koristimo „*try-catch*“. Unutar „*try*“ bloka pišemo köd koji će se „probati“ izvršiti. Ako se ipak taj köd neće moći izvršiti, onda će se automatski pozvati „*catch*“ blok i köd napisan u njemu.

```
try
{

}
catch (Exception)
{

    throw;
}
```

Slika 22 - Primjer praznog try-catch bloka

Kažemo da pogrešku **obrađujemo**.

Pogledajmo na primjeru:

```
int broj;

try
{
    broj = int.Parse(txtVozilo.Text);
}
catch (Exception greska)
{

    MessageBox.Show("Dogodila se pogreška: " + greska);
}
```

Slika 23 . Jednostavan primjer obrade pogreške

Unutar „*try*" bloka dohvaćamo vrijednost svojstva „*Text*" kontrole „*txtVozila*" i pokušavamo tu *string* vrijednost prebaciti u *Integer* tip podatka korištenjem funkcije *Parse()*. Ako taj postupak uspije, nastavit će se izvršavati köd iza „*try-catch*" dijela. Ukoliko postupak pretvorbe ne uspije (primjerice uneseno je neko slovo umjesto broja), pokreće se köd unutar *catch* bloka. U tom slučaju se i **poziva pogreške** (*call Exception*) koja je po svom tipu „*Exception*" i zove se „greska". Unutar „*catch*" bloka u kojem obrađujemo tu iznimku ispisujemo korisniku poruku korištenjem *MessageBox* elementa i metode *Show()* u kojoj navodimo tekst „Dogodila se pogreška" te dohvaćamo o kojoj vrsti pogreške se radi. Vrsta pogreške je pohranjena u varijablu „pogreska".

Mi možemo eksplicitno navesti detaljniju vrstu iznimke koja se poziva. Također, često možemo imati nekoliko „*catch"* blokova. Razlog tome je što želimo probati uhvatiti točno neku vrstu iznimke.

```
int broj;

try
{
    broj = int.Parse(txtVozilo.Text);
}
catch (FormatException)
{
    MessageBox.Show("Pogrešan format unosa. Lijepo ti piše da uneseš broj!");
}

catch (Exception greska)
{

    MessageBox.Show("Dogodila se pogreška: " + greska);
}
```

Slika 24 - Više catch blokova prilikom obrade iznimki

U ovom primjeru imamo dva *catch* bloka. Prvi obrađuje iznimku ukoliko je predana vrijednost u pogrešnom formatu. Ukoliko se dogodi neka druga vrsta pogreške nju će **uhvatiti** drugi *catch* blok.

11. 3. Bacanje iznimki (throw)

U slučaju neželjenih vrijednosti, često želimo namjerno izazvati pogrešku odnosno **baciti iznimku**. Za to koristimo ključnu riječ „***throw***“.

```
if (broj < 0)
    throw new ArgumentException("Broj mora biti veći od 0!");
else if (broj > 100)
    throw new ArgumentException("Broj mora biti manji od 100!");
```

Slika 25 - Namjerno izazivanje iznimki (throw)

U ovom primjeru zamišljamo da smo korisniku rekli da unese broj koji je veći od 0 (nula) i manji od 100 (sto). Vrijednost dohvaćamo iz neke *textBox* kontrole i tu vrijednost pohranimo u varijablu „broj“. Za potrebe objašnjavanja, smatramo da je korisnik zaista unio neku cjelobrojnu vrijednost. Sada nam preostaje provjeriti jeli unio dozvoljenu numeričku vrijednost odnosno nalazi li se broj koji je korisnik unio u raspnu od 0 do 100.

Prvim „*if*“ uvjetom provjeravamo jeli broj manji od nule. Ako je, **bacamo novu iznimku** (konkretno vrsta *ArgumentException*) kojoj kao parametar predajemo poruku koju želimo da se korisniku ispiše.

Pomoću „*else-if*“ testiramo jeli broj manji veći od 100 i ako je, želimo također baciti novu iznimku i reći korisniku da to nije dopušteno.

Kada se neka iznimka pozove, tada se izvršavanje aplikacije prekida i na taj način se osiguravamo da neće doći do njenog rušenja.

12) Popis slika:

Slika 1 - Dodavanje nove klase 8
Slika 2 - Izbornik za dodavanje klase 8
Slika 3 - Prikaz novo dodane klase u Solution Exploreru 9
Slika 4 - Početni köd nakon dodavanja nove klase 9
Slika 5 - Dodavanje svojstava klasi 10
Slika 6 - Dodavanje metoda klasi 11
Slika 7 - Nasljeđivanje klasa 11
Slika 8 - Klasa Vozilo 12
Slika 9 - Pristup svojstvima nasljeđenje klase 12
Slika 10 - Stvaranje konstruktora 15
Slika 11 - Preopterećenje konstruktora 16
Slika 12 - Ključna riječ "this" 18
Slika 13 - Izrada Getter i Setter svojstava 20
Slika 14 - Read-only svojstvo 21
Slika 15 - Svojstva Windows Formi 22
Slika 16 - Pristup događaju 23
Slika 17 - Metoda koja se poziva prilikom događaja (ControlAdded) 23
Slika 18 - Toolbox 24
Slika 19 - Pristup svojstvima elementa unutar köda 26
Slika 20 - Upis u Text svojstvo, kontrole textBox 26
Slika 21 - Dohvaćanje vrijednosti svojstva Text, kontrole textBox 26
Slika 22 - Primjer praznog try-catch bloka 27
Slika 23 . Jednostavan primjer obrade pogreške 28
Slika 24 - Više catch blokova prilikom obrade iznimki 28
Slika 25 - Namjerno izazivanje iznimki (throw) 29