Programski jezik C#

Osnove objektnega programiranja

Srečo Uranič in Matija Lokar

V 0.1

april 2009

Predgovor

Omenjeno gradivo pokriva osnove objektnega programiranja v C# .

Gradivo vsekakor ni dokončano in predstavlja delovno različico. V njem so zagotovo napake (upava, da čimmanj), za katere se vnaprej opravičujeva. Da bo lažje spremljati spremembe, obstaja razdelek Zgodovina sprememb, kamor bova vpisovala spremembe med eno in drugo različico. Tako bo nekomu, ki si je prenesel starejšo različico, lažje ugotoviti, kaj je bilo v novi različici spremenjeno.

Srečo Uranič in Matija Lokar

Kranj, april 2009

Zgodovina sprememb

05. 04. 2009: Različica V0.1 – Prva verzija

KAZALO

Objektno programiranje

Motivacija

Pri programiranju se je izkazalo, da je zelo koristno, če podatke in postopke nad njimi združujemo v celote, ki jih imenujemo **objekte**. Programiramo potem tako, da objekte ustvarimo in jim naročamo, da izvedejo določene postopke. Programskim jezikom, ki omogočajo tako programiranje, rečemo, da so objektno (ali z drugim izrazom predmetno) usmerjeni. Objektno usmerjeno programiranje nam omogoča, da na problem gledamo kot na množico objektov, ki med seboj sodelujejo in vplivajo drug na drugega. Na ta način lažje pišemo sodobne programe.

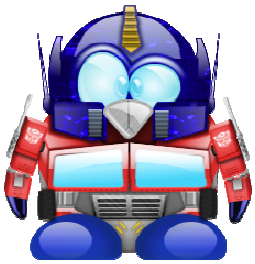
Objektno usmerjenih je danes večina sodobnih programskih jezikov. Pri njihovi uporabi hitro naletimo na pojme, kot so združevanje (**enkapsulacija**), večličnost (**polimorfizem**), dedovanje (**inheritence**), ki so vsaj na prvi pogled zelo zapleteni. Nas podrobnosti ne bodo zanimale in si bomo ogledali le osnove.

Objektno programiranje – kaj je to

Pri objektno usmerjenem programiranju se ukvarjamo z … objekti seveda. Objekt je nekakšna črna škatla, ki dobiva in pošilja sporočila. V tej črni škatli (objektu) se skriva tako koda (torej zaporedje programskih stavkov), kot tudi podatki (informacije nad katerimi se izvaja koda). Pri klasičnem programiranju imamo kodo in podatke ločene. Tudi mi smo do sedaj programirali več ali manj na klasičen način. Pisali smo metode, ki smo jim podatke posredovali preko parametrov. Parametri in metode niso bile prav nič tesno povezane. Če smo npr. napisali metodo, ki je na primer poiskala največji element v tabeli števil, tabela in metoda nista bili združeni – tabela nič ni vedela o tem, da naj bi kdo npr. po njej iskal največje število.

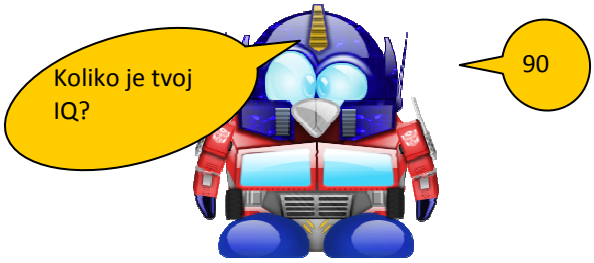
V objektno usmerjenem programiranju (OO) pa sta koda in podatki združeni v nedeljivo celoto – objekt. To prinaša določene prednosti. Ko uporabljamo objekte, nam nikoli ni potrebno pogledati v sam objekt. To je dejansko prvo pravilo OO programiranja – uporabniku ni nikoli potrebno vedeti, kaj se dogaja znotraj objekta. Gre dejansko zato, da nad objektom izvajamo različne metode. In za vsak objekt se točno ve, katere metode zanj obstajajo in kakšne rezultate vračajo. Recimo, da imamo določen objekt z imenom *robotek*. Vemo, da objekte take vrste, kot je *robotek,* lahko povprašamo o tem, koliko je njihov IQ. To storimo tako, da nad njimi izvedemo metodo

```
robotek.KolikoJeIQ();
```



Objekt robotek
Vir: http://commons.wikimedia.org/wiki/Crystal_Clear

Objekt odgovori s sporočilom (metoda vrne rezultat), ki je neko celo število.



Objekt sporoča
Vir: http://commons.wikimedia.org/wiki/Crystal_Clear

Tisto, kar je pomembno, je to, da nam, kot uporabnikom objekta, ni potrebno vedeti, kako objekt izračuna IQ (je to podatek, ki je zapisan nekje v objektu, je to rezultat nekega preračunavanja …?). Kot uporabnikom nam je važno le, da se z objektom lahko pogovarjamo o njegovem IQ. Vse "umazane podrobnosti" o tem, kaj je znotraj objekta, je prepuščeno tistim, ki napišejo kodo, s pomočjo katere se ustvari objekt.

Seveda se lahko zgodi, da se kasneje ugotovi, da je potrebno "preurediti notranjost objekta". In tu se izkaže prednost koncepta objektnega programiranja. V programih, kjer objekte uporabljamo, nič ne vemo o tem, kaj je "znotraj škatle", kaj se na primer dogaja, ko objekt povprašamo po IQ. Z objektom "komuniciramo" le preko izvajanja metod. Ker pa se klic metode ne spremeni, bodo programi navkljub spremembam v notranjosti objekta še vedno delovali.

Če torej na objekte gledamo kot na črne škatle (zakaj črne – zato, da se ne vidi, kaj je znotraj!) in z njimi ravnamo le preko sporočil (preko klicev metod), naši programi delujejo ne glede na morebitne spremembe v samih objektih.

Omogočanje dostopa do objekta samo preko sporočil in onemogočanje uporabnikom, da vidijo (in brskajo) po podrobnostih, se imenuje skrivanje informacij ali še bolj učeno **enkapsulacija**. In zakaj je to pomembno? Velja namreč, da se programi ves čas spreminjajo. Velika večina programov se dandanes ne napiše na novo, ampak se spremeni obstoječ program. In večina napak izhaja iz tega, da nekdo spremeni določen delček kode, ta pa potem povzroči, da drug del programa ne deluje več. Če pa so tisti delčki varno zapakirani v kapsule, se spremembe znotraj kapsule ostalega sistema prav nič ne tičejo.

Pravzaprav smo se že ves čas ukvarjali z objekti, saj smo v naših programih uporabljali različne objekte, ki so že pripravljeni v standardnih knjižnicah jezika C#. Oglejmo si dva primera objektov iz standardne knjižnice jezika:

- Objekt **System.Console** predstavlja *standardni izhod*. Kadar želimo kaj izpisati na zaslon, pokličemo metodo *Write* na objektu **System.Console**.

```
//uporaba metode WriteLine na objektu Console
Console.WriteLine();
```

- Objekt tipa **Random** predstavlja generator naključnih števil. Metoda *Next(a, b)*, ki jo izvedemo nad objektom tega tipa nam vrne neko naključno celo število med *a* in *b*.

```
//uporaba metode na objektu iz razreda Random
Random naklj = new Random();
int nakljucnoStevilo=naklj.Next(1, 100);
```

- Objekt tipa **StreamReader** predstavlja *vhodni kanal*. Kadar želimo brati s tipkovnice ali z datoteke, naredimo tak objekt in kličemo njegovo metodo **ReadLine** za branje ene vrstice.

```
//kreiranje objekta StreamReader
string datoteka = "Vaja.txt";
StreamReader branje = new StreamReader(datoteka);
string vrstica = branje.ReadLine();
```

Naučili smo se torej uporabnosti objektnega programiranja, nismo pa se še naučili izdelave svojih razredov. In če se vrnemo na razlago v uvodu – kot uporabniki čisto nič ne vemo (in nas pravzaprav ne zanima), kako metoda *Next* določi naključno število. In tudi, če se bo kasneje "škatla" *Random* spremenila in bo metoda *Next* delovala na drug način, to ne bo "prizadelo" programov, kjer smo objekte razreda *Random* uporabljali. Seveda, če bo sporočilni sistem ostal enak. V omenjenem primeru to pomeni, da se bo metoda še vedno imenovala *Next*, da bo imela dva parametra, ki bosta določala meje za naključna števila in da bo odgovor (povratno sporočilo) neko naključno število z želenega intervala.

Seveda pa nismo omejeni le na to, da bi le uporabljali te "črne škatle", kot jih pripravi kdo drug (npr. jih dobimo v standardni knjižnici jezika). Objekti so uporabni predvsem zato, ker lahko programer definira nove razrede in objekte, torej sam ustvarja te nove črne škatle, ki jih potem on sam in drugi uporablja.

Razred

Razred (**class**) je abstraktna definicija objekta (torej opis, kako je naša škatla videti znotraj). Veliko razredov je že vgrajenih (so v standardnih knjižnicah) v C# in njegovo okolje, pogosto pa je za naše aplikacije potrebno napisati tudi kak svoj razred. Z razredom opišemo, kako je neka vrsta objektov videti. Če na primer sestavljamo razred zajec, opisujemo, katere so tiste lastnosti, ki določajo vse zajce.

V razredu zapišemo lastnosti, stanja in obnašanje objektov: zapišemo torej katere **podatke** bomo hranili v objektih te vrste in katere **metode** oz. odzive objektov na sporočila. Stanje objekta torej opišemo s spremenljivkami (rečemo jim tudi **polja ali komponente**), njihovo obnašanje oz. odzive pa z **metodami**.

Če želimo nek razred uporabiti, mora običajno obstajati vsaj en **primerek** razreda. Primerek nekega razreda imenujemo **objekt**. Ustvarimo ga s ključno besedo **new**. Razred je tako v bistvu **šablona**, ki definira spremenljivke in metode skupne vsem objektom iste vrste, objekt pa je **primerek** (srečali boste tudi "čuden" izraz ***instanca***) nekega razreda.

```
imeRazreda primerek = new imeRazreda();
```

Objekt je računalniški model predmeta ali stvari iz vsakdanjega življenja (avto, oseba, datum…). Objekt je kakršenkoli skupek podatkov, s katerimi želimo upravljati. Osnovni pristop objektnega programiranja je torej:

objekt = podatki + metode za delo s podatki.

Objekt je "črna škatla“, ki sprejema in pošilja sporočila. Jedro objekta sestavljajo njegove spremenljivke, okrog katerih se nahajajo njegove metode.

Klasično/objektno programiranje

Pojasnimo sedaj še razliko med klasičnim in objektnim programiranjem. Vsak program je sestavljen iz zaporedja stavkov, v katerih uporabljamo spremenljivke in metode. Metode, ki jih uporabljamo, so bodisi naše lastne metode (napisali smo jih sami) ali pa že obstoječe metode (npr. metode »knjižnjice« *Math*, kot napr. *Sqrt*, *Pow*, *Round*, …).

Ko želimo kak podatek obdelati v klasičnem programiranju pokličemo ustrezno metodo z argumenti, ki naj jih ta metoda obdela. Če smo npr. napisali metodo *Vsota*, ki vrne celo število, ima pa dva celoštevilska parametra, bomo to metodo poklicali npr. takole:

```
int skupaj=Vsota(23,45);
```

Pri objektnem programiranju pa je zadeva nekoliko drugačna. Ker metode pripadajo objektom (ki jih izpeljemo oz. ustvarimo iz razredov), moramo pred imenom metode napisati še ime objekta, ki mu ta metoda pripada. Recimo, da smo napisali razred *Ulomek*, znotraj tega razreda pa smo napisali metodo *Vsota*. Iz razreda *Ulomek* smo nato ustvarili objekt *U1*. Metodo vsota objekta *U1* pokličemo sedaj takole:

```
int skupaj=U1.Vsota(23,45); //pred imenom metode zapišemo še ime objekta
```

Oglejmo si sedaj primer programa v C#, ki uporablja objekte, ki jih napišemo sami.

```
public class MojR
{
   private string mojNiz;

   public MojR(string nekNiz)
   {
      mojNiz = nekNiz;
   }
   public void Izpisi()
   {
      Console.WriteLine(mojNiz);
   }
}
public static void Main(string[] arg)
{
   MojR prvi;
   prvi= new MojR("Pozdravljen, moj prvi objekt v C#!");
   prvi.Izpisi();
}
```

Definicija razreda

oznaka (ime) objekta

kreiranje objekta

ukaz objektu

Na kratko razložimo, "kaj smo se šli". Pred glavnim programom smo definirali nov razred z imenom *MojR*. Ta je tak, da o vsakem objektu te vrste poznamo nek niz, ki ga hranimo v njem. Vse znanje, ki ga objekti tega razreda imajo je, da se znajo odzvati ne metodo *Izpis()*, ko izpišejo svojo vsebino (torej niz, ki ga hranimo v njem).

Glavni program *Main* (glavna metoda) je tisti del rešitve, ki opravi dejansko neko delo. Najprej naredimo primerek objekta iz razreda *MojR* (prvi) in vanj shranimo niz *"Pozdravljen, moj prvi objekt v C#!"*. Temu objektu nato naročimo, naj izpiše svojo vsebino.

Povejmo še nekaj o drugem načelu združevanja, o pojmu ***dostopnost*** (**enkapsulacija**). Potem, ko smo metode in polje združili znotraj razreda, smo pred temeljno odločitvijo, kaj naj bo javno, kaj pa zasebno. Vse, kar smo zapisali med zavita oklepaja v razredu, spada v notranjost razreda. Z besedicami **public, private** in **protected,** ki jih napišemo pred ime metode ali polja, lahko kontroliramo, katere metode in polja bodo dostopna tudi od zunaj:

- Metoda ali polje je privatno (**private**), kadar je dostopno le znotraj razreda.
- Metoda ali polje je javno (**public**), kadar je dostopno tako znotraj, kot tudi izven razreda.
- Metoda ali polje je zaščiteno (**protected**), kadar je vidno le znotraj razreda, ali pa v **podedovanih** (**izpeljanih**) razredih.

Obstajajo tudi druge dostopnosti, a se z njimi ne bomo ukvarjali. Prav tako ne bomo uporabljali zaščite *protected*. Omejili se bomo le na privatni (*private*) in javni dostop (*public*). Več o enkapsulaciji, torej pomenu besed *public*, *private* in *protected*, bo napisano v nadaljevanju.

Ustvarjanje objektov

V prejšnjem primeru smo iz razreda *MojR* tvorili objekt *prvi*. Pravimo, da smo deklarirali nov objekt razreda *mojR* z imenom *prvi*.

```
MojR prvi;  // prvi je NASLOV objekta
```

Nov objekt v pomnilniku ustvarimo z ukazom **new**. Brez besedice *new* objekta še **NI**. Naredimo sedaj npr. objekt tipa *MojR*, po pravilih, ki so določena z opisom razreda *MojR*. Novo ustvarjeni objekt se nahaja na naslovu *prvi*. Pravimo, da smo objekt *inicializirali*, oz. da smo objekt *definirali* (naslovu objekta smo priredili vrednost).

```
//z naslednjim ukazom bomo v pomnilniku naredili objekt tipa MojR, po pravilih, ki so določena z opisom razreda MojR. Novo ustvarjeni objekt se bo nahajal na naslovu prvi.
prvi = new MojR("Pozdravljen, moj prvi objekt v C#!");
```

Deklaracijo objekta in njegovo definicijo (prirejanje) lahko seveda združimo v en sam stavek takole:

```
MojR prvi = new MojR("Pozdravljen, moj prvi objekt v C#!");
```

Naslov objekta

Sicer vedno rečemo, da v spremenljivki *prvi* hranimo objekt, a zavedati se moramo, da to ni čisto res. V spremenljivki *prvi* je shranjen **naslov** objekta. Zato po

```
MojR prvi, drugi;
```

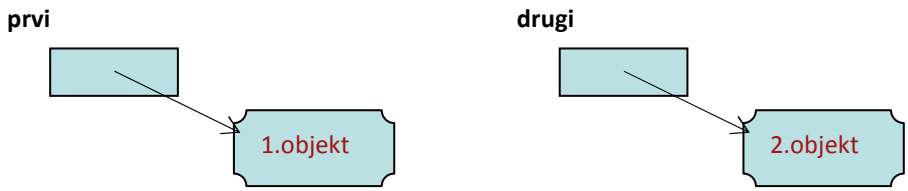
nimamo še nobenega objekta. Imamo le dve spremenljivki, v kateri lahko shranimo naslov, kjer bo operator *new* ustvaril nov objekt. Rečemo tudi, da sta spremenljivki *prvi* in *drugi* kazali na objekta tipa *MojR*. Če potem napišemo

```
prvi = new MojR("1.objekt");
```

smo v spremenljivko *prvi* shranili naslov, kjer je novo ustvarjeni objekt. Ustvarimo še en objekt in njegov naslov shranimo v spremenljivko *drugi*

```
drugi = new MojR("2. objekt");
```

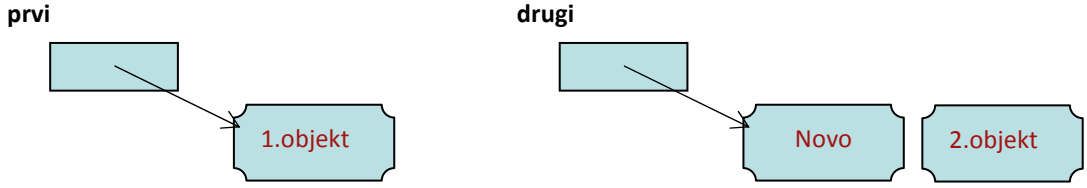
Poglejmo, kakšno je sedaj stanje v pomnilniku. S puščico v spremenljivkah *prvi* in *drugi* smo označili ustrezen naslov objektov. Dejansko je na tistem mestu napisano nekaj v stilu *h002a22* (torej naslov mesta v pomnilniku)



In če sedaj napišemo

```
drugi = new MojR("Novo");
```

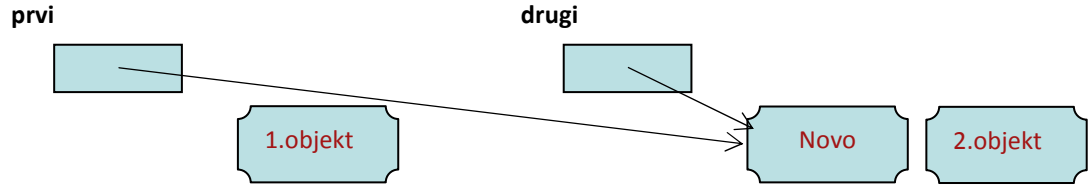
je vse v redu. Le do objekta z vsebino "2. objekt" ne moremo več! Stanje v pomnilniku je



in če naredimo še

```
prvi = drugi;
```

smo s tem izgubili še dostop do objekta, ki smo ga prvega ustvarili in pomnilnik bo videti takole.



Sedaj tako spremenljivka *prvi* in *drugi* vsebujeta naslov istega objekta. Do objektov z vsebino "1. Objekt" in "2. objekt" pa ne more nihče več. Na srečo bo kmalu prišel smetar in ju pometel proč. Smetar (**garbage collector**) je poseben program v C#, za katerega delovanje nam ni potrebno skrbeti in ki poskrbi, da se po pomnilniku ne nabere preveč objektov, katerih naslov ne pozna nihče več.

Primer:
Recimo, da smo napiali razred *Ulomek*. Z naslednjim stavkom bomo iz tega razreda izpeljali objekt *ime*.

```
Ulomek ime = new Ulomek(3, 4);
```

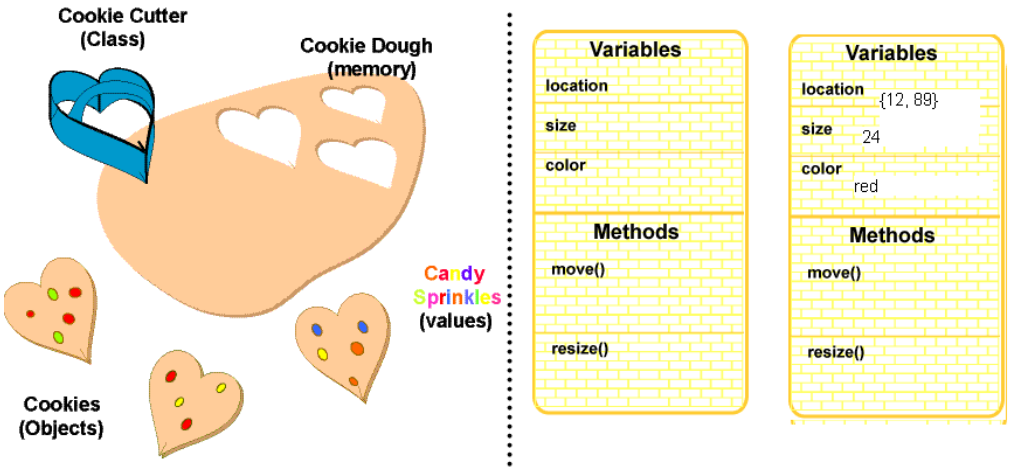
Rečemo: V objektu *ime* je shranjen ulomek ¾ oz. točneje: v spremenljivki *ime* je naslov nekega objekta tipa *Ulomek*, ki predstavlja ulomek ¾. Spremenljivka ime kaže na objekt tipa *Ulomek*, ki …

Sicer bomo vedno govorili kot: *... v objektu mojObjekt imamo niz, nad objektom zajecRjavko izvedemo metodo ...,* a vedeti moramo, da sta v spremenljivkah *mojObjekt* in *zajecRjavko* naslova in ne dejanska objekta.

Kako se torej lotiti načrtovanja rešitve s pomočjo objektnega programiranja?

- Z analizo ugotovimo, kakšne objekte potrebujemo za reševanje.
- Pregledamo, ali že imamo na voljo ustrezen razred (standardne knjižnice, druge knjižnice, naši stari razredi).
- Sestavimo ustrezen razred (določimo polja in metode našega razreda).
- Sestavimo "glavni" program, kjer s pomočjo objektov rešimo problem.

Pa še enkrat: razred (*class*) je opis vrste objekta (načrt, kako naj bo objekt videti) – opis ideje objekta. Primerek razreda (instanca) pa je konkretni objekt.



Zgledi

Ulomek

Denimo, da bi radi napisali program, ki bo prebral nek ulomek in mu prištel 1/2. "Klasično" se bomo tega lotili takole:

```
static void Main(string[] args)
{
    Console.Write("Števec ulomka: ");
    string beri = Console.ReadLine();
    int stevec = int.Parse(beri);
    Console.Write("Imenovalec ulomka: ");
    beri = Console.ReadLine();
    int imenovalec = int.Parse(beri);
    //ulomku prištejmo 1/2. Ulomka seveda seštejemo po pravilu za
    //seštevanje ulomkov: a/b + c/d =(a*d + c*b)/(b*d)
    stevec = stevec * 2 + imenovalec * 1;
    imenovalec = imenovalec * 2;
    // izpis
    Console.WriteLine("Nov ulomek je: " + stevec + " / " + imenovalec);
    Console.ReadKey();
}
```

Sedaj pa to naredimo še "objektno". Očitno bomo potrebovali razred *Ulomek*. Objekt tipa *Ulomek* hrani podatke o svojem števcu in imenovalcu. Njegovo znanje je, da "se zna" povečati za drug ulomek. To pomeni, da vsebuje metodo, ki ta ulomek poveča za drug ulomek. Celotno kodo razreda *Ulomek* seveda zaenkrat še ne bomo razumeli, a pomembno je to, da dobimo občutek, kaj pravzaprav pojem razred predstavlja.

```
class Ulomek
{   // razred Ulomek naj vsebuje dve polji: stevec in imenovalec
    public int stevec;
    public int imenovalec;

    public Ulomek(int st, int im)  // konstruktor
    {
        this.stevec = st;
        this.imenovalec = im;
    }

    public void Pristej(Ulomek a)//metoda, ki ulomek poveča za drug ulomek
    {
        this.stevec=this.stevec* a.imenovalec + this.imenovalec * a.stevec;
        this.imenovalec = this.imenovalec * a.imenovalec;
    }
}
static void Main(string[] args)
{
    Console.Write("Števec ulomka: ");
    string beri = Console.ReadLine();
    int stevec = int.Parse(beri);
    Console.Write("Imenovalec ulomka: ");
    beri = Console.ReadLine();
    int imenovalec = int.Parse(beri);
    Ulomek moj = new Ulomek(stevec, imenovalec); // NAREDIMO ulomek
    // "delo"
    Ulomek polovica = new Ulomek(1, 2);
    moj.Pristej(polovica); // UKAZUJEMO ulomku
    // izpis
```

```
    Console.WriteLine("Nov ulomek je: " + moj.stevec + " / " +
                      moj.imenovalec);
    Console.ReadKey();
}
```

Krog

Radi bi napisali program, ki bo prebral polmer kroga in izračunal ter izpisal njegovo ploščino. "Klasično" bi program napisali takole:

```
public static void Main(string[] arg)
{
    //  vnos podatka
    Console.Write("Polmer kroga: ");
    int r = int.Parse(Console.ReadLine());
    //  izračun
    double ploscina = Math.PI * r * r;
    //  izpis
    Console.WriteLine("Ploščina kroga: " + ploscina);
}
```

Sedaj pa to naredimo še "objektno". Očitno bomo potrebovali razred *Krog*. Objekt tipa *Krog* hrani podatke o svojem polmeru. Njegovo znanje pa je, da "zna" izračunati svojo ploščino.

```
class Krog
{
    public double polmer;   // polje razreda Krog
    public double Ploscina()    // metoda razreda krog
    {
        return Math.PI * polmer * polmer;
    }
}
public static void Main(string[] arg)
{
    Krog k = new Krog();  // naredimo nov objekt tipa krog
    Console.Write("Polmer: ");
    k.polmer = double.Parse(Console.ReadLine());  // polmer preberemo
    Console.WriteLine(k.Ploscina());   // izpis ploščine
}
```

Zgradba

Napišimo razred *Zgradba*, ki naj vsebuje dve polji: *kvadratura* in *stanovalcev*.

```
class Zgradba // deklaracija razreda Zgradba z dvema javnima poljema.
{
    public int kvadratura;
    public int stanovalcev;
}
```

Če sedaj ustvarimo objekt tipa *Zgradba*

```
Zgradba hiša = new Zgradba(); //  nov objekt razreda Zgradba
```

do polj *kvadratura* in *stanovalcev* dostopamo z

```
hiša.stanovalcev = 4;
```

oziroma

```
hiša.kvadratura = 2500;
```

Če imamo torej nek razred *ImeRazreda* in v njem polje *nekoPolje* in je objekt *mojObjekt* tipa *ImeRazreda*, z

```
mojObjekt.nekoPolje
```

dostopamo do te spremenljivke. Uporabljamo jo enako, kot vse spremenljivke tega tipa (če je *nekoPolje* tipa *double*, z *mojObjekt.nekoPolje* lahko počnemo vse tisto, kar pač lahko počnemo s spremenljivkami tipa *double*). V metodi *Main* kreirajmo dva objekta in za vsakega posebej ugotovimo, kolikšna stanovanjska površina pride na posameznika.

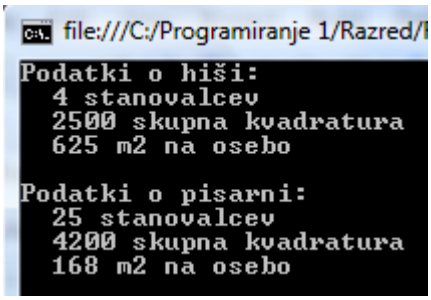
```
static void Main(string[] args)
{
    Zgradba hiša = new Zgradba();   // nov objekt razreda Zgradba
    Zgradba pisarna = new Zgradba();  // nov objekt razreda Zgradba
    int kvadraturaPP;  //  spremenljivka za izračun kvadrature na osebo
    // določimo začetne vrednosti prvega objekta
    hiša.stanovalcev = 4;
    hiša.kvadratura = 2500;
    // določimo začetne vrednosti drugega objekta
    pisarna.stanovalcev = 25;
    pisarna.kvadratura = 4200;
    // izračun kvadrature za prvi objekt
    kvadraturaPP = hiša.kvadratura / hiša.stanovalcev;

    Console.WriteLine("Podatki o hiši:\n " + hiša.stanovalcev + "
                      stanovalcev\n  " + hiša.kvadratura + " skupna
                      kvadratura\n  " + kvadraturaPP + " m2 na osebo");

    Console.WriteLine();
    // izračun kvadrature za drugi objekt
    kvadraturaPP = pisarna.kvadratura / pisarna.stanovalcev;

    Console.WriteLine("Podatki o pisarni:\n  " + pisarna.stanovalcev + "
                      stanovalcev\n  " + pisarna.kvadratura + " skupna
                      kvadratura\n  " + kvadraturaPP + " m2 na osebo");
}
```

Izpis, ki ga dobimo, je naslednji:



Razred *Zgradba* je sedaj nov tip podatkov, ki ga pozna C#. Uporabljamo ga tako, kot vse druge, v C# in njegove knjižnice vgrajene tipe. Torej lahko napišemo

```
Zgradba[] ulica;   // ulica bo tabela objektov tipa Zgradba
```

Pri tem pa moramo sedaj paziti na več stvari. Zato bomo o tabelah objektov spregovorili v posebnem razdelku.

Evidenca članov kluba

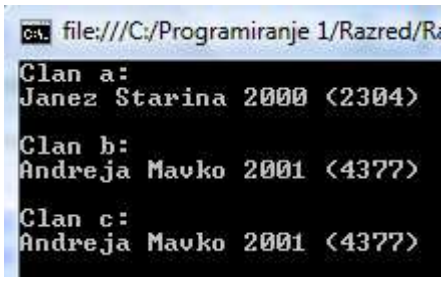
Napišimo program, ki vodi evidenco o članih športnega kluba. Podatki o članu obsegajo ime, priimek, letnico vpisa v klub in vpisno številke (seveda je to poenostavljen primer). Torej objekt, ki predstavlja člana kluba, vsebuje štiri podatke. Ustrezni razred je:

```
public class Clan
{
    public string ime;
    public string priimek;
    public int leto_vpisa;
    public string vpisna_st;
}
```

S tem smo povedali, da je vsak *objekt* tipa *Clan* sestavljen iz štirih komponent: *ime*, *priimek*, *leto_vpisa* in *vpisna_st*. Če je *a* objekt tipa *Clan*, potem lahko dostopamo do njegove komponente *ime* tako, da napišemo *a.ime*. Tvorimo nekaj objektov:

```
1: static void Main(string[] args)
2: {
3:     Clan a = new Clan();
4:     a.ime = "Janez";
5:     a.priimek = "Starina";
6:     a.leto_vpisa = 2000;
7:     a.vpisna_st = "2304";
8:     Clan b = new Clan();
9:     b.ime = "Mojca";
10:    b.priimek = "Mavko";
11:    b.leto_vpisa = 2001;
12:    b.vpisna_st = "4377";
13:    Clan c = b;
14:    c.ime = "Andreja";
15:    Console.WriteLine("Član a:\n" + a.ime + " " + a.priimek +
                         " " + a.leto_vpisa + " (" + a.vpisna_st + ")\n");
16:    Console.WriteLine("Član b:\n" + b.ime + " " + b.priimek +
                         " " + b.leto_vpisa + " (" + b.vpisna_st + ")\n");
17:    Console.WriteLine("Član c:\n" + c.ime + " " + c.priimek +
                         " " + c.leto_vpisa + " (" + c.vpisna_st + ")\n");
18: }
```

Ko program poženemo, dobimo naslednji izpis:



Hm, kako to, da sta dva zadnja izpisa enaka? V programu smo naredili dva nova objekta razreda *Clan*, ki sta shranjena v spremenljivkah *a* in *b*. A spomnimo se, da imamo v spremenljivkah *a* in *b* shranjena *naslova* objektov, ki smo ju naredili. Kot vedno, nove objekte naredimo z ukazom *new*. Spremenljivka *c* pa **kaže na isti objekt kot** *b*, se pravi, da se v 13. vrstici *ni* naredila nova kopija objekta *b*, ampak se spremenljivki *b* in *c sklicujeta* na isti objekt. In zato smo z vrstico 14 vplivali tudi na ime, shranjeno v objektu *b (*bolj točno, na ime, shranjeno v objektu, katerega naslov je shranjen v *b*)!

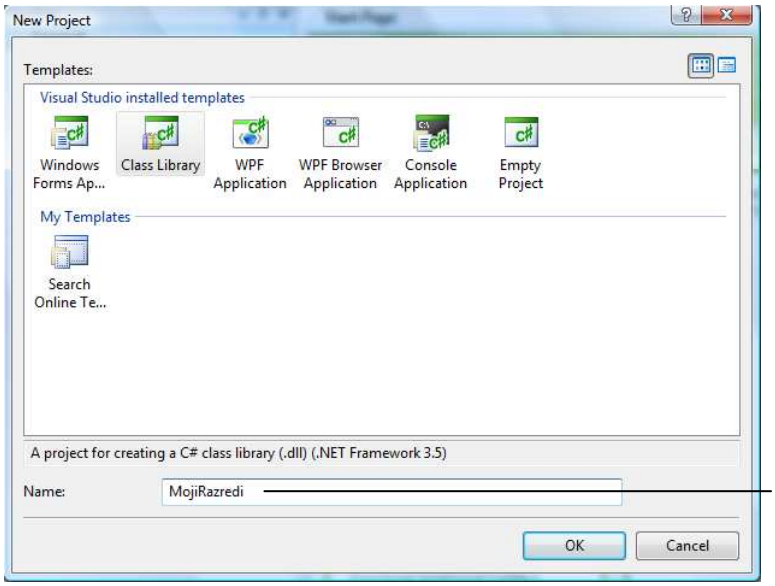
V vrsticah 4 – 7 smo naredili objekt *a* (naredili smo objekt in nanj pokazali z *a*) in nastavili vrednosti njegovih komponent. Takole nastavljanje je v praksi dokaj neprimerno, ker se zlahka zgodi, da kako komponento pozabimo nastaviti. Zato C# omogoča, da delamo nove objekte na bolj praktičen način s pomočjo **konstruktorjev**, o katerih pa bomo govorili nekolik kasneje.

Knjižnjice razredov in datoteka z razredi

V dosedanjih zgledih do smo razrede pisali le za "lokalno uporabo", znotraj določenega programa. Razred (ali pa več razredov) pa lahko zapišemo tudi v svojo datoteko, ki jo potem dodajamo k različnim projketom, ali pa celo zgradimo svojo **knjižnjico razredov**, ki jih bomo uporabljali v različnih programih. Na ta način bomo tudi bolj ločili tisti del programiranja, ko gradimo razrede in tisti del, ko uporabljamo objekte določenega razreda.

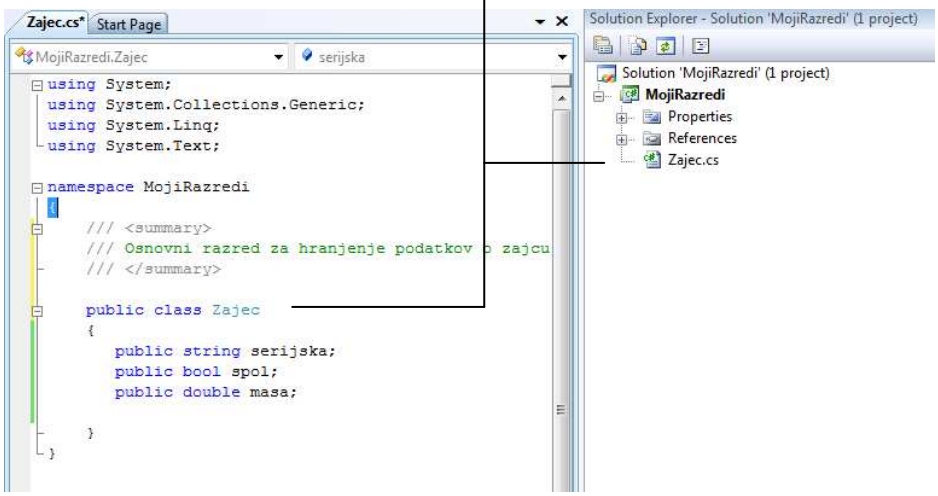
Ustvarjanje nove knjižnjice razredov

Naučimo se najprej, kako zgradimo svojo knjižnjico razredov: v okolju Visual C# tokrat ne izberemo Windows Applications/Console Application, ampak **Class Library**



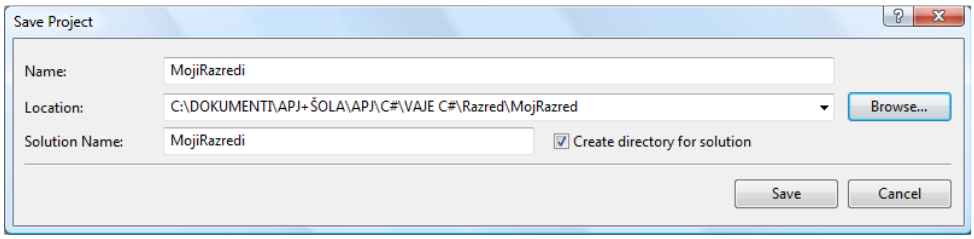
Kot ime knjižnjice bomo napisali npr. *MojiRazredi*. V to knjižnjico bomo kasneje lahko dodajali vse tiste razrede, ki jih bomo ustvarjali.

S klikom na gumb OK potrdimo našo izbiro. Vidimo, da je Visual C# za nas pripravil okolje in naredil tako imenovam imenski prostor *MojiRazredi*. Znotraj tega imenskega prostora bomo zlagali naše razrede. Ker nam ime *MyClass* ni všeč, bomo to spremenili v *Zajec*.



Dodajmo še komentar, ki naj se začne z ///. To so tako imenovani dokumentacijski komentarji. Ti služijo za to, da pripravimo dokumentacijo o razredu. Ta je zelo pomembna, saj nosi tisto informacijo, ki jo kasneje potrebujemo, če želimo razred uporabljati. Zaenkrat si zapomnimo le, da na ustrezno mesto napišemo kratek povzetek o tem, čemu je razred namenjen.
Napisali smo torej zelo poenostavljen opis, kako je določen poljuben zajec. Zavedati pa se moramo, da gre v bistvu le za načrt, kakšni so zajci, ne pa za konkretnega zajca. Opazimo tudi, da v knjižnjici, ki jo pišemo, ni metode *Main*. Knjižnjica namreč ni namenjena izvajanju oz. poganjanju tako kot smo bili navajeni pri dosedanjih programih, ampak je namenjena uporabi v drugih programih, ki jih bomo pisali kasneje.

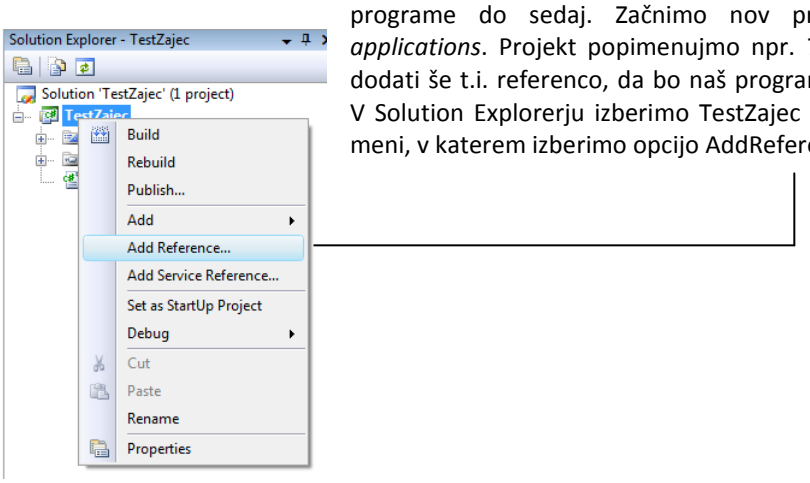
Preden nadaljujemo knjižnjico shranimo: v meniju *File* izberimo opcijo *Save All*. Odpre se pogovorno okno za shranjevanje: s klikom na gumb *Browse* izberimo še mapo, v katero bomo našo knjižnjico shranili (v našem primeru je ime mape *Moj Razred*)



Knjižnjico je sedaj potrebno še prevesti. V meniju *Build* izberimo opcijo *Build Solution* (ali pa stisnimo tipko *F6*) in naša knjižnjica je sedaj pripravljena za uporabo. Ustvarili smo namreč ustrezno prevedeno obliko te knjižnjice, ki je dobila ime *MojiRazredi.dll* (*dll* je kratica za *dinamic link library*). Knjižnjica zaenkrat vsebuje le definicijo razreda *Zajec*. Datoteka se nahaja v mapi ..\ *MojRazred**MojiRazredi**MojiRazredi**bin**Debug*.

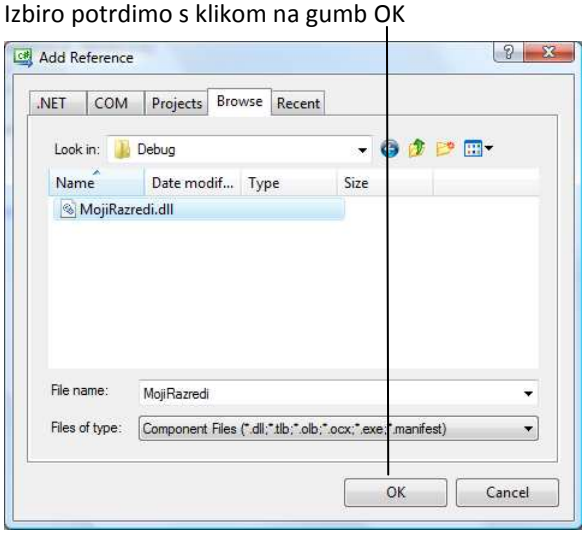
Uporaba knjižnjice razredov

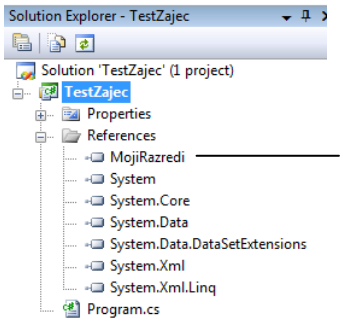
Denimo, da sedaj pišemo nek program, kjer bomo potrebovali Zajce. V ta namen sedaj sestavimo program v C#, ki bo med drugim uporabljal že obstoječo knjižnjico z imenom *MojiRazredi*. Obnašamo se tako kot smo pisali programe do sedaj. Začnimo nov projekt : *File/New Project/Console applications*. Projekt popimenujmo npr. *TestZajec*. V projekt moramo sedaj dodati še t.i. referenco, da bo naš program prepoznal naš razred *MojiRazredi*. V Solution Explorerju izberimo TestZajec in z desnim klikom odprimo lebdeči meni, v katerem izberimo opcijo AddReference



Odpre se okno za izbiro ustrezne reference: izberimo najprej jeziček *Browse* in nato poiščimo datoteko *MojiRazredi.dll*.

Izbiro potrdimo s klikom na gumb OK





Opazimo, da se v *Solution Explorerju* pojavi nova referenca *MojiRazredi*, kar pomeni, da imamo sedaj dostop do vseh razredov knjižnjice *MojiRazredi*.

Da bo dostop do naših razredov še lažji, na začetku datoteke dopišimo še en *using* stavek:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MojiRazredi; //NOVO dodana knjižnjica
```

Nov objekt razreda *Zajec*, ki je definiran znotraj knjižnjice *MojiRazredi* lahko sedaj naredimo takole:

```
Zajec rjavko = new Zajec();
```

V spremenljivki *rjavko* je v bistvu naslov, kje je novo ustvarjeni zajec (objekt), a kljub temu navadno rečemo, da smo ustvarili objekt *rjavko*, ki smo ga izpeljali iz razreda *Zajec* (ali še drugače: naredili smo novo ***instanco*** razreda *Zajec*). Ustvarili smo torej konkretnega zajca po navodilih za razred *Zajec*. Ta zajec (*rjavko*) ima torej tri podatke / lastnosti / komponente), to pa so spol, serijska številka in masa. Te lastnosti lahko sedaj določimo poljubno, npr. takole:

```
rjavko.spol = true;
rjavko.serijska = "BRGH_17_A";
rjavko.masa = 3.2;
```

S tako ustvarjenimi spremenljivkami lahko sedaj počnemo povsem enako kot z običajnimi spremnljivkami. Še primer celotne vsebine datoteke *TestZajec*:

```
...
using MojiRazredi;  //NOVO dodana knjižnjica
```

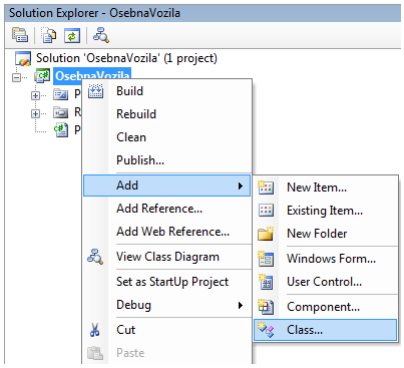
```
namespace TestZajec
{
  class Program
  {
    static void Main(string[] args)
    {
       Zajec rjavko = new Zajec();//ustvarimo novo instanco razreda Zajec
       rjavko.serijska = "1238-12-0";
       rjavko.spol = false;
       rjavko.masa = 0.12;
       rjavko.masa = rjavko.masa + 0.3; //maso zajca lahko  povečamo

       //posamezna polja objekta rjavko lahko izpisujemo
       Console.WriteLine("Zajec ima ser. št.:" + rjavko.serijska);
       Console.ReadKey();
    }
  }
}
```

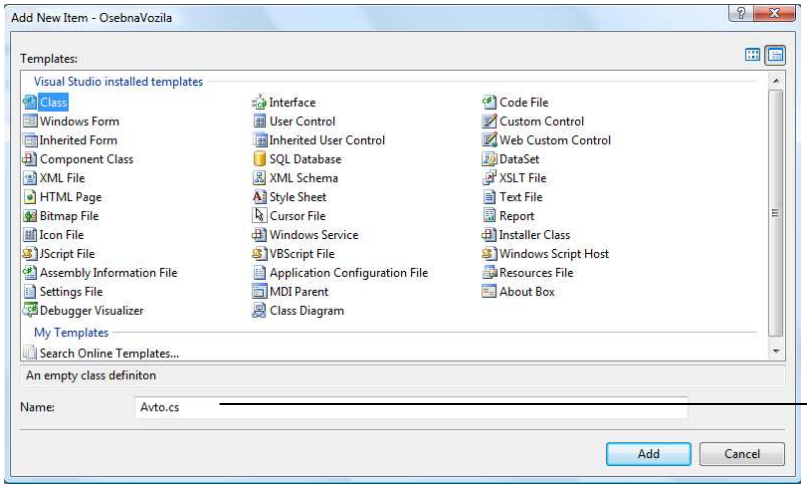
Datoteka z razredom (ali več razredi)

Namesto knjižnjice razredov pa lahko razred (ali pa skupino razredov) zapišemo tudi v posebno datoteko kar znotraj projekta, v bodočih projektih pa to datoteko samo dodamo v projekt.

Začnimo nov projekt (nova konzolna aplikacija) in ga poimenujmo *OsebnaVozila*. Kreirajmo sedaj razred *avto* s štirimi zasebnimi polji (*znamka*, *model*, *najvecjahitrost* in *teza*), ter javno metodo za izpis polj tega razreda. Tej metodi bomo dali ime *IzpisPodatkov*. Razred bomo zapisali v svojo datoteko, ki jo odpremo takole: v *SolutionExplorerju* izberimo vrstico *OsebnaVozila*, nato pa z desnim klikom miške odpremo lebdeči meni. Izberimo opcijo *Add* in nato *Class*.



V oknu, ki se odpre, se prepričajmo, če je izbrana opcija *Class* in vnesimo ime našega razreda – *Avto.cs*.



Vse skupaj potrdimo s klikom na gumb *Add*. V *Solution Explorerju* bomo opazili novo vrstico z imenom *Avto*, v urejevalniku pa bo prikazana začetna vsebina datoteke (razreda) *Avto.cs*. Vsebino dopolnimo, tako da je izgled celotne datoteke takle:

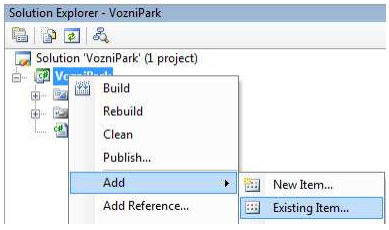
```
using System;
using System.Collections.Generic;
using System.Text;

namespace OsebnaVozila
{
    class Avto
    {
        //polja razreda Avto
        public string znamka;
        public string model;
        public int najvecjahitrost;
        public double teza;
        //javna metoda za izpis podatkov o konkretnem avtomobilu. Več o
        //metodah v razredih bo napisano v nadaljevanju
        public string IzpisPodatkov()
        {
            return ("\nIzpis podatkov o vozilu:\nZnamka: "
                                + znamka
                                + "\nModel: " + model
                                + "\nNajvečja hitrost: " + najvecjahitrost
                                + "\nteža vozila: " + teza);
        }
    }
}
```

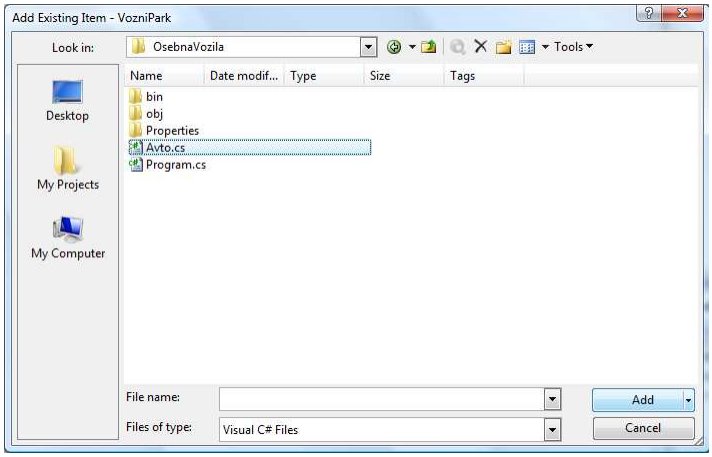
Datoteko shranimo in sedaj ponovno odprimo datoteko z "glavnim" programom (v *Solution Explorerju* dvoklinimo vrstico *Program.cs*). Deklarirajmo nov objekt *moj*, mu določimo začetne vrednosti in pokličimo metodo *IzpisPodatkov*:

```
static void Main(string[] args)
{
    Avto moj = new Avto();//deklaracija novega objekta moj
    //poskrbimo za inicializacijo
    moj.znamka = "Citroen";
    moj.model = "C4 Picasso";
    moj.najvecjahitrost = 185;
    moj.teza = 1850;
    //klic metode IzpisPodatkov, ki jo objekt moj seveda pozna, saj smo ga
    //izpeljali iz razreda Avto, v katerem je ta metoda definirana
    Console.WriteLine(moj.IzpisPodatkov());
}
```

Razred *Avto* smo torej napisali v svoji datoteki in ga nato uporabili v projektu, znotraj katerega smo ta razed tudi kreirali. Sedaj pa moramo še pokazati, kako bi ta razred uporabili v nekem drugem, novem proketu. Postopek je naslednji: ustvarimo nov projekt in ga poimenujmo npr. *VozniPark*. Ker želimo v tem projektu uporabiti razred, ki so ga v prejšnjem projektu zapisali v datoteko *Avto.cs*, ga lahko v naš novi projekt vključimo takole: v *Solution Explorerju* izberemo vrstico *VozniPark*, kliknemo desni miškin gumb in izberimo opcijo *Add*, ter nato *Existing Item*...



Odpre se pogovorno okno, s pomočjo katerega sedaj poiščimo datoteko *Avto.cs*, ki se nahaja znotraj prejšnjega projekta, ki smo ga poimenovali *OsebnaVozila*. Ko jo najdemo, izbiro potrdimo s klikom na gumb *Add*.



V *Solution Explorerju* se ojavi nova vrstica *Avto.cs* (razred *Avto* smo dodali v naš novi projekt). Ker smo razred Avto napisali v projektu Osebna vozila, se nanj sklicujemo takole:

```
static void Main(string[] args)
{
   OsebnaVozila.Avto sosedov = new OsebnaVozila.Avto();
   //...
}
```

Z objektom *sosedov* sedaj delamo popolnoma enakovredno kakor v prejšnjem projektu *OsebnaVozila*, v katerem smo razred *Avto* tudi definirali.

Do razreda *Avto* smo torej prišli preko imenskega prostora *OsebnaVozila*. Če pa ta imenski prostor z *using* stavkom dodamo na začeku programa tako, da napišemo

```
using System;
using System.Collections.Generic;
using System.Text;
using OsebnaVozila; //dodamo imenski prostor
```

potem pa lahko nov objekt razreda *Avto* napovemo takole:

```
static void Main(string[] args)
{
   Avto sosedov = new Avto();
   //...
}
```

Pokazali smo torej, kako kreiramo svojo lastno knjižnjico razredov (dll), ki jo potem dodamo kot referenco k vsem nadaljnjim projektom, prav tako pa smo pokazali kako ustvarimo datoteko z razredom in jo dodajamo k novim projektom. ***Samo zaradi enostavnosti in lažjega razumevanja, pa bomo v nadaljnji razlagi, primerih in zgledih, razrede pisali kar znotraj določenega programa***.

Povzetek

Najenostavnejši razred kreiramo tako, da določimo spremenljivke (polja), ki določajo objekte tega razreda. Te so lahko različnih podatkovnih tipov. Sintaksa najenostavnejše definicije razreda je naslednja:

```
public class ImeRazreda
{
   public podatkovni tip element1;
   public podatkovni tip element2;
      …
   public podatkovni tip elementn;
}
```

Če želimo tak razred uporabljati v nekem programu, moramo v programu dodati referenco na ustrezno prevedeno obliko (*dll*) tega razreda, ali pa datoteko s tem razredom dodati v program.

Objekt tega razreda ustvarimo z `new ImeRazreda()`. S tem dobimo naslov, kje ta objekt je in ga shranimo v spremenljivko tipa *ImeRazreda*:

```
ImeRazreda imeObjekta = new ImeRazreda();
```

Do posameznih spremenljivk (rečemo jim tudi komponente ali pa lastnosti), dostopamo z

```
imeObjekta.imeKomponente;
```

Objektne metode – funkcije zaredov

Svoj pravi pomen dobi sestavljanje razredov takrat, ko objektu pridružimo še metode, torej znanje nekega objekta. Kot že vemo iz uporabe v C# vgrajenih razredov, metode nad nekim objektom kličemo tako, da navedemo ime objekta, piko in ime metode. Tako če želimo izvesti metodo *WriteLine* nad objektom *System.Console* napišemo

```
System.Console.WriteLine("To naj se izpiše");
```

Če želimo izvesti metodo *Equals* nad nizom besedilo, napišemo

```
besedilo.Equals(primerjava);
```

in tako dalje.

Kako napišemo svojo lastno metodo znotraj razreda

Metodo znotraj poljubnega razreda napišemo tako kot vsako metodo: napišemo njeno glavo (dostopnost, tip rezultata, ime in parametre), ter telo metode.

Primer:

Razred *Clan*, ki smo ga napisali v prejšnjem poglavju, opremimo še z dvema metodama: metodo *Izpis*, ki izpiše podatke o članu, ter metodo *Opis*, ki vrne podatke o članu v obliki stringa.

```
public class Clan
{
    //Najprej lastnosti (polja) razreda Clan
    public string ime;
    public string priimek;
    public int leto_vpisa;
    public string vpisna_st;

    //Metoda, ki izpiše podatke o članu
```

```
    public void Izpis()
    {
        Console.WriteLine("Clan:\n" + this.ime + " " + priimek + " " +
                          leto_vpisa + " (" + vpisna_st + ")\n");
    }
    //Metoda, ki vrne podatke o članu v obliki stringa
    public string Opis()
    {
        return this.ime + " " + priimek + " " +  leto_vpisa + " (" +
        vpisna_st +")";
    }

}
public static void Main(string[] args)
{
  //Ustvarimo novega člana: nov objekt razreda Clan se imenuje c1
  Clan c1 = new Clan();
  //Članu c1 določimo polja
  c1.ime= "Janez";
  c1.priimek="Starina";
  c1.leto_vpisa=2000;
  c1.vpisna_st="2304";
  //S pomočjo metode Izpis objekta c1 izpišimo podatke o članu c1
  Console.WriteLine("Clan c1"); c1.Izpis();
  //V spremenljivo opisClana shranimo opis člana c1, ki ga dobimo s pomočjo
  //metode Opis objekta c1
  string opisClana = c1.Opis();
  Console.WriteLine("Clan c1:\n" + opisClana); //Izpis
  Console.ReadLine();
}
```

Seveda lahko v obstoječi razred kadarkoli dodajamo še nove metode, ali pa spremenimo (popravimo) obstoječe metode. Ko razred (ali pa projekt) ponovno prevedemo, imamo že vse pripravljeno za "pravilno" izvajanje vseh tistih programov, ki uporabljajo ta razred. Brez kakršnekoli spremembe v uporabniškem programu (in tudi brez ponovnega prevajanja tega programa), metoda sedaj deluje v skladu z v razredu narejenimi spremembami. Če torej opazimo, da je potrebno v razredu narediti kakšen popravek, le vsem našim uporabnikom pošljemo novo različico prevedenega razreda (ali knjižnice, ki vsebuje ta razred). Brez kakršnihkoli sprememb na strani uporabnika njihovi programi delujejo "popravljeno".

Gostota prebivalstva

Razred *Drzava* naj vsebuje polja za osnovne podatke o državi, ter metodo *Gostota* za izračun gostote prebivalstva. Kreirajmo objekt in pokažimo uporabo njegove metode *Gostota*.

```
class Drzava
{
  public string ime;
  public int povrsina;
  public int prebivalci;
  public double Gostota() // javna metoda za izračun gostote prebivalstva
  {
    return prebivalci/povrsina;  // vrnemo celo število!
  }
}
static void Main(string[] args)
{
  Drzava d1=new Drzava();// d1 je nov objekt razreda Drzava
  d1.ime = "Slovenija";
```

```
  d1.povrsina = 20000;
  d1.prebivalci = 1980000;
  // pokličimo metodo Gostota objekta d1
  Console.WriteLine("Gostota prebivalstva v " + d1.ime + " je " +
                    d1.Gostota()+ " preb/km2");
}
```

Polinom druge stopnje

Kreirajmo razred *Polinom*, ki hrani polinome do stopnje dva ($ax^2 + bx + c$). Polinom naj ima tri polja (koeficiente polinoma), pozna pa naj tudi tri metode: metodo za seštevanje dveh polinomov, metodo, ki vrne vrednost polinoma v poljubni točki in metodo *ToString*, ki ta polinom predstavi (vrne) v obliki stringa.

```
public class Polinom
{
   public double a, b, c;//koeficienti polinoma
   //Metoda za seštevanje dveh polinomov
   public Polinom Sestej(Polinom p)
   {
      Polinom zacasni=new Polinom();
      zacasni.a=a + p.a;
      zacasni.b=b + p.b;
      zacasni.c=c + p.c;
      return zacasni;
   }
   //metoda ki izračuna in vrne vrednost polinoma v točki x
   public double vrednostPolinoma(double x)
   {
       return a*x*x+b*x+c;
   }
   //metoda za izpis polinoma v obliki a*x2 + b*x + c
   public string ToString()
   {
       return a + "*x2 + " + b + "*x+ " + c;
   }
}
public static void Main(string[] args)
{
   Polinom P1=new Polinom(); //Prvi polinom
   P1.a = 2;
   P1.b = 3;
   P1.c = 5;
   Console.WriteLine("Prvi polinom: "+P1.ToString());//Izpis polinoma št.1

   Polinom P2 = new Polinom();//Drugi polinom
   P2.a = 1;
   P2.b = 5;
   P2.c = 7;
   Console.WriteLine("Drug polinom: "+P2.ToString());//Izpis polinoma št.2
   //S pomočjo metode Sestej objekta P1 polinomu P1 prištejmo polinom P2 in
   // rezultat izpišimo s pomočjo metode ToString
   Console.WriteLine("Vsota polinomov: " + P1.Sestej(P2).ToString());
   Console.ReadLine();
}
```

Kvader

Kreirajmo razred *Kvader*, ki naj vsebuje tri polja (robove kvadra) in metode za izračun telesne diagonale, površine in prostornine kvadra.

```
public class Kvader
{
    public double a, b, c;
    //Metoda izračuna in vrne telesno diagonala kovadra
    public double TelesnaDiagonala()
    {
       return Math.Sqrt(a*a+b*b+c*c);
    }
    //Metoda izračuna in vrne površino kvadra
    public double Povrsina()
    {
        return 2 * a * b + 2 * a * c + 2 * b * c;
    }
    //Metoda izračuna in vrne prostornino kvadra
    public double Prostornina()
    {
        return a * b * c;
    }
}
static void Main(string[] args)
{
    Kvader K1=new Kvader();//ustvarimo nov objekt K1 tipa Kvader in ga
    K1.a=1;
    K1.b=2;
    K1.c=3;
    //Klic objektne metode TelesnaDiagonala
    Console.WriteLine("Telesna diagonala kvadra: "+K1.TelesnaDiagonala());
    //Klic objektne metode Povrsina
    Console.WriteLine("Površina kvadra: " + K1.Povrsina());
    //Klic objektne metode Prostornina
    Console.WriteLine("Prostornina kvadra: " + K1.Prostornina());
    Console.ReadLine();
}
```

Razred *MojString*

Napišimo razred *MojString*, v katerem bomo napisali nekaj javnih metod za delo s stringi, ki naj pomenijo alternativo obstoječim metodam – npr. *Length*, *Replace*, *ToUpper*,…

```
class MojString
{
    private string stavek; //zasebno polje razreda
    public void DolociStavek(string st) //metoda za inicializacijo polja
                                        //stavek
    {
        stavek = st;
    }
    public int dolzina()//metoda ki vrne število znakov v stringu
    {
        return stavek.Length;
    }
    //metoda za zamenjavo določenih znakov v stringu z novimi znaki
    public void ZamenjajZnak(string stari, string novi)
```

```
    {
        stavek = stavek.Replace(stari, novi);
    }
    public string IzpisStavka()
    {
        return stavek;
    }
    public void VelikeCrke()
    {
        stavek = stavek.ToUpper();
    }
}

static void Main(string[] args)
{
    MojString st=new MojString ();
    st.DolociStavek("Razred MojString: metodam za delo s stringi smo
                    priredili slovenska imena!");
    Console.WriteLine("\nV stavku:\n\n"+st.IzpisStavka()+"\n\nje
                     "+st.dolzina()+" znakov!");
    st.ZamenjajZnak(" ", "X");//presledke nadomestimo z znakom X
    Console.WriteLine(st.IzpisStavka());
    st.ZamenjajZnak("X"," ");//vse znake "X" nadomestimo s presledki
    st.VelikeCrke();  //vse črke v stringu spremenimo v velike črke
    Console.WriteLine(st.IzpisStavka());
}
```

Podpisi metod

Pri pisanju metod ("navadnih" in objektnih) se moramo držati pravila, da se morajo razlikovati ne v imenu, ampak **podpisu**. Podpis metode je sestavljen iz imena metode in tipov vseh parametrov. Tip rezultata neke metode **NI** del podpisa.

Primeri različnih podpisov metod:

- public static int **NekaMetoda**(**double x**)
- public static int **NekaMetoda**()
- public static int **NekaMetoda**(**int x**)
- public static int **NekaMetoda**(**double x,int y**)
- public static double **NekaMetoda**(**int x, int y**)
- public static int **nekaDrugaMetoda**(**double y**)

Preobtežene metode

Tako "navadne", kot tudi objektne metode so lahko ***preobtežene*** (uporablja se tudi izraz ***preobložene***). Preobtežene metode so metode, ki imajo enako ime, razlikujejo pa se v številu ali pa tipu parametrov. Preobteženost torej označuje možnost, da imamo lahko več enako poimenovanih metod, ki pa sprejemajo parametre različnega tipa.

Zakaj je to uporabno? Denimo, da želimo napisati metodo *Ploscina*, ki kot parameter dobi objekt iz razreda *Kvadrat*, *Krog* ali *Pravokotnik*. Kako jo začeti?

```
public static double Ploscina(X lik)
```

Ker ne vemo, kaj bi napisali za tip parametra, smo napisali kar X. Če preobteževanje ne bi bilo možno, bi morali napisati metodo, ki bi sprejela parameter tipa X, kjer je X bodisi *Kvadrat*, *Krog* ali *Pravokotnik*. Seveda to ne gre, saj prevajalnik nima načina, da bi zagotovil, da je X oznaka bodisi za tip *Kvadrat*, tip *Krog* ali pa tip *Pravokotnik*.

Pa tudi če bi šlo – kako bi nadaljevali? V kodi bi morali reči ... če je lik tipa *Kvadrat*, uporabi to formulo, če je lik tipa *Krog* spet drugo ...

S preobteževanjem pa problem rešimo enostavno. Napišemo tri metode: ker gre v naslednjih primerih za "navadne" oz. "statične" metode smo dodali še besedico *static* (pravi pomen te besede bomo spoznali v poglavju o statičnih metodah).

```
public static double Ploscina(Kvadrat lik) { … }
public static double Ploscina(Krog lik) { … }
public static double Ploscina(Pravokotnik lik) { … }
```

Ker za vsak lik znotraj ustrezne metode točno vemo, kakšen je, tudi ni težav z uporabo pravilne forumule. Imamo torej tri metode, z enakim imenom, a različnimi podpisi. Seveda bi lahko problem rešili brez preobteževanja, recimo takole:

```
public static double PloscinaKvadrata(Kvadrat lik) { … }
public static double PloscinaKroga(Krog lik) { … }
public static double PloscinaPravokotnika(Pravokotnik lik) { … }
```

A s stališča uporabnika metod je uporaba preobteženih metod enostavnejša. Če v programu nastopa npr. nek lik *likX*, ki je bodisi tipa *Kvadrat*, tipa *Krog* ali pa tipa *Pravokotnik*, uporabnik metodo pokliče z *Ploscina(likX)*, ne da bi mu bilo potrebno razmišljati, kakšno je pravilno ime ustrezne metode ravno za ta lik. Prav tako je enostavneje, če dobimo še četrti tip objekta – v "glavno" kodo ni potrebno posegati – naredimo le novo metodo z istim imenom (*Ploscina*) in s parametrom katerega tip je novi objekt. Klic pa je še vedno *Ploscina(likX)*!

Konstruktor

V vseh dosedanjih primerih smo najprej ustvarili nov objekt, nato pa mu nastavili začetne vrednosti polj. Tako nastavljanje je v praksi dokaj neprimerno, ker se zlahka zgodi, da kako komponento pozabimo nastaviti. Zato C# omogoča, da delamo nove objekte na bolj praktičen način, s pomočjo **konstruktorjev**. **Konstruktor** je metoda, ki jo pokličemo ob tvorbi objekta z *new*. Je brez tipa rezultata. Ne smemo pa ga zamenjati z metodami, ki rezultata ne vračajo (te so tipa *void*). Konstruktor tipa rezultata sploh nima, tudi *void* ne. Prav tako smo omejeni pri izbiri imena te metode. Konstruktor mora imeti ime nujno enako, kot je ime razreda. Če konstruktorja ne napišemo, ga "naredi" prevajalnik sam (a metoda ne naredi nič). **Vendar pozor**: kakor hitro napišemo vsaj en svoj konstruktor, C# ne doda svojega.

Največja omejitev, ki loči konstruktorje od drugih metod je ta, da konstruktorja ne moremo klicati na poljubnem mestu (kot lahko ostale metode). Kličemo ga izključno skupaj z *new*, npr. *new Zajec()*. Uporabljamo jih za vzpostavitev začetnega stanja objekta.

Primer:

Napišimo konstruktor za razred *Zajec*:

```
public class Zajec
{
  public string serijska;
  public bool spol;
  public double masa;

  //konstruktor: NIMA tipa, njegovo ime pa mora biti enako kot je ime
  //razreda
```

```
  public Zajec()
  {
    this.spol = true; // vsem zajcem na začetku določimo moški spol
    this.masa = 1.0; // in tehtajo 1kg
    this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
  }
}
```

this

V zgornjem primeru smo v konstruktorju uporabili prijem, ki ga doslej še nismo srečali. Napisali smo *this*. Kaj to pomeni? **this** označuje objekt, ki ga "obdelujemo". V konstruktorju je to objekt, ki ga ustvarjamo. *this.ulica* se torej nanaša na lastnost/komponento objekta, ki se ustvarja (ki ga je ravno naredil *new*).

Denimo, da v nekem programu napišemo

```
Zajec rjavko = new Zajec();
Zajec belko = new Zajec();
```

Pri prvem klicu se ob uporabi konstruktorja *Zajec() this* nanašal na *rjavko*, pri drugem na *belko*. Na ta način (z *this*) se lahko sklicujemo na objekt vedno, kadar pišemo metodo, ki jo izvajamo nad nekim objektom. Denimo, da smo napisali

```
Random ng = new Random();
Random g2 = new Random();
Console.WriteLine(ng.Next(1, 10));
```

Kako so programerji, ki so sestavljali knjižnico in v njej razred *Random* lahko napisali kodo metode, da se je vedelo, da pri metodi *Next* mislimo na uporabo generatorja *ng* in ne na *g2*?

Denimo, da imamo razred *MojRazred* (v njem pa komponento *starost*) in v njem metodo *MetodaNeka*. V programu, kjer razred *MojRazred* uporabljamo, ustvarimo dva objekta tipa *MojRazred*. Naj bosta to *objA* in *objC*. Kako se znotraj metode *MetodaNeka* sklicati na ta objekt (objekt, nad katerim je izvajana metoda)? Če torej metodo *MetodaNeka* kličemo nad obema objektoma z *objA.MetodaNeka()* oziroma z *objC.MetodaNeka()*, kako v postopku za *metodaNeka* povedati, da gre:

- prvič za objekt z imenom *objA* in
- drugič za objekt z imenom *objC*

Če se moramo v kodi metode *metodaNeka* sklicati recimo na komponento *starost* (jo recimo izpisati na zaslon), kako povedati, da naj se ob klicu *objA.MetodaNeka()* uporabi starost objekta *objA*, ob klicu *objC.MetodaNeka()* pa starost objekta *objC*?

```
Console.WriteLine("Starost je: " + ?????.starost);
```

Ob prvem klicu so ???? *objA*, ob drugem pa *objC*. To "zamenjavo" dosežemo z *this*. Napišemo

```
Console.WriteLine("Starost je: " + this.starost);
```

Ob prvem klicu *this* pomeni *objA*, ob drugem pa *objC*. Torej v metodi na tistem mestu, kjer potrebujemo konkretni objekt, nad katerim metodo izvajamo, napišemo besedico *this*.

Uporaba konstruktorja:

Kot smo povedali že večkrat, je razred le načrt, kako so videti objkekti določenega razreda. Če potrebujemo konkreten primer, v našem primeru Zajca, ga "ustvarimo" z *new*:

```
Zajec rjavko = new Zajec();
```

S tem se je ustvaril konkreten zajec (torej tak, ki ima tri spremenjlivke za hranjenje podatkov), nad katerim smo takoj potem izvedli določeno akcijo po navodilih iz konstruktorja *Zajec()*. Kombinacija *new Zajec()* je torej poskrbela, da ima objekt tipa *Zajec* tri podatke z vrednostmi, kot je predpisano v konstruktorju.

Privzeti konstruktor

Če konstruktorja ne napišemo (kot ga npr. nismo v začetnih primerih poglavja o razredih in objektih), ga "naredi" prevajalnik sam (a metoda ne naredi nič). Konstruktor, ki ga avtomatično generira prevajalnik, je vedno public, nima nobenega tipa (niti void), nima argumentov, vrednosti numeričnih polj postavi na 0, polja tipa *bool* postavi na *false*, vsem referenčnim poljem (spremenljivkam) pa priredi vrednost *null*.

Dokler je bil naš razred *Zajec* takle:

```
public class Zajec
{
  public string serijska;
  public bool spol;
  public double masa;
}
```

je prevajalnik sam dodal privzeti kostruktor.

```
public Zajec()
{
}
```

Kakor hitro pa smo konstruktor *Zajec()* napisali sami, se seveda prevajalnik ni več "vmešaval":

Zgled - nepremičnine

Sestavimo razred, ki bo v svoja polja lahko shranil ulico, številko nepremičnine ter vrsto nepremičnine. Ustvarimo poljuben objekt, ga inicializirajmo in ustvarimo izpis, ki naj zgleda približno takole: Na naslovu Cankarjeva ulica 32, Kranj je blok.

```
class Nepremicnina
{
    public string ulica;
    public int stevilka;
    public string vrsta;
    // konstruktor
    public Nepremicnina(string kateraUlica,int hisnaStevilka, string vrsta)
    {
        this.ulica = kateraUlica;
        this.stevilka = hisnaStevilka;
        this.vrsta = vrsta;
    }
}
```

```
static void Main(string[] args)
{
   // za kreiranje novega objekta uporabimo konstruktor
   Nepremicnina nova = new Nepremicnina("Cankarjeva ulica 32, Kranj", 1234,
                                        "blok");
    // še izpis podatkov o nepremičnini
    Console.WriteLine("Na naslovu " + nova.ulica + " je " + nova.vrsta);
    Console.ReadKey();
}
```

V zgornjem konstruktorju smo uporabili besdico *this*. Ta je v konstruktorju obvezna le tedaj, kadar so imena parametrov enaka imenom polj razreda. Če pa se imena parametrov razlikujejo od imen polj, besedica *this* seveda ni potrebna. Zgornji konstruktor bi torej lahko napisali tudi takole:

```
public Nepremicnina(string kateraUlica,int hisnaStevilka, string vrsta)
{
    ulica = kateraUlica;
    stevilka = hisnaStevilka;
    vrsta = vrsta;
}
```

Zgled: razred Denarnica

Sestavimo razred *Denarnica*, ki bo omogočal naslednje operacije: dvig, vlogo in ugotavljanje stanja. Začetna vrednost se naj postavi s konstruktorjem.

```
public class Denarnica
{   // razred ima dve polji: ime osebe in stanje v denarnici
    public string ime;
    public double stanje;

    // konstruktor za nastavljanje začetnih vrednosti ima dva parametra
    public Denarnica(string ime, double znesek)
    {
        this.ime = ime;
        this.stanje = znesek;
    }
    public void dvig(double znesek)
    {
        stanje = stanje - znesek;
    }
    public void polog(double znesek)
    {
        stanje = stanje + znesek;
    }
}

static void Main(string[] args)
{
    // tvorimo dva objekta - uporabimo konstruktor
    Denarnica Mojca = new Denarnica("Mojca", 1000);
    Denarnica Peter = new Denarnica("Peter", 500);
    // izpis začetnega stanja v denarnici za oba objekta
    Console.WriteLine("Mojca - začetno stanje: " + Mojca.stanje);
    Console.WriteLine("Peter - začetno stanje: " + Peter.stanje);
    Mojca.dvig(25.55);  // Mojca zapravlja
    Peter.polog(70.34);  // Peter ja zaslužil
```

```
    // izpis novega stanja v denarnici za oba objekta
    Console.WriteLine("Mojca - po dvigu: " + Mojca.stanje);
    Console.WriteLine("Peter - po pologu: " + Peter.stanje);
}
```

Prodajalec

Prodajalcu napišimo program, ki mu bo pomagal pri vodenju evidence o mesečni in letni prodaji.

```
class Prodajalec
{
   public double[] zneski;// zasebna tabela ki hrani mesečne zneske prodaje
   public Prodajalec()    // konstruktor ki inicializira tabelo prodaje
   {
       zneski = new double[12]; // vsi elementi tabele dobijo vrednost 0
   }
   // metoda za izpis letne prodaje
   public void IzpisiLetnoProdajo()
   {
      Console.WriteLine("Skupna letna prodaja je : " + SkupnaLetnaProdaja()
                        + " EUR");
   }
   // metoda za izračun skupne letne prodaje
   public double SkupnaLetnaProdaja()
   {
       double vsota = 0.0;
       for (int i = 0; i < 12; i++)
           vsota += zneski[i];
       return vsota;
   }
}

static void Main(string[] args)
{
   //  tvorba objekta p tipa Prodajalec, obenem se izvede tudi konstruktor
   Prodajalec p = new Prodajalec();
   //  zanka za vnos prodaje po mesecih
   for (int i = 1; i <= 12; i++)
   {
      Console.Write("Vnesi znesek prodaje za mesec " + i + ": ");
      p.zneski[i - 1] = Convert.ToDouble(Console.ReadLine());
   }
   p.IzpisiLetnoProdajo(); // objekt p pozna metodo za izpis letne prodaje
}
```

V zgornjem primeru smo v razredu napovedali tabelo *zneski*. Za inicializacijo te tabele smo napisali konstruktor, seveda pa bi lahko tabelo inicializirali že pri deklaraciji s stavkom:

```
public double[] zneski = new double[12];//elementi tabele dobijo vrednost 0
```

Tabele objektov

Rekli smo že, da smo s tem, ko smo sestavili nov razred, sestavili dejansko nov tip podatkov. Ta je "enakovreden" v C# in njegovim knjižnicam vgrajenim tipom. Torej lahko naredimo tudi tabelo podatkov, kjer so ti podatki objekti.

Sedaj pišemo program, v katerem bomo potrebovali 100 objektov tipa *Zajec* (denimo, da pišemo program za upravljanje farme zajcev; razred Zajec poznamo že iz prejšnjih primerov). Ker bomo z vsemi zajci (z vsemi objekti tipa *Zajec*) počeli enake operacije, je smiselno, da uporabimo tabelo.

```
Zajec[] tabZajci;
```

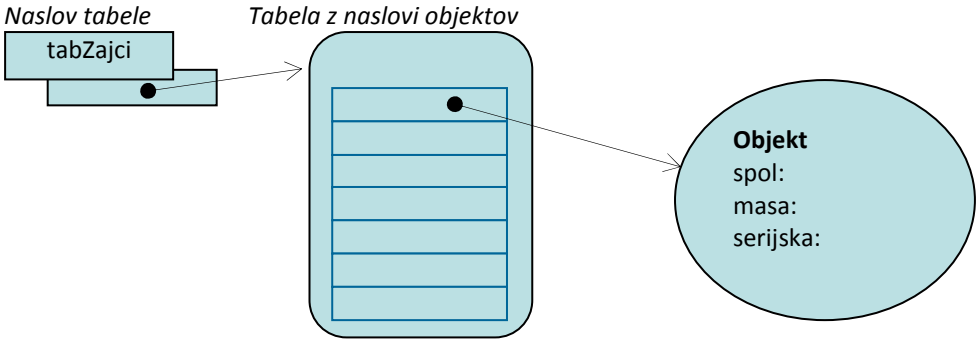
Spemenljivka *tabZajci* nam označuje torej tabelo, v kateri hranimo zajce. A bodimo tokrat še zadnjič pri izražanju zelo natančni. Natančno rečeno nam spremenljivka *tabZajci* označuje **naslov**, kjer bomo ustvarili tabelo, v kateri bomo hranili naslove objektov tipa *Zajec*. Ko torej napišemo

```
tabZajci = new Zajec[250];
```

smo s tem ustvarili tabelo velikosti 250. Kje ta tabela je, smo shranili v spremenljivko *tabZajci*. V tej tabeli lahko hranimo podatek o tem, kje je objekt tipa *Zajec*. V tem trenutku še nimamo nobenega objekta tipa *Zajec*, le prostor za njihove naslove. Šele ko napišemo

```
tabZajci[0] = new Zajec();
```

smo s tem ustvarili novega zajca (nov objekt tipa *Zajec*) in podatke o tem, kje ta novi objekt je (naslov), shranili v spremenljivko *tabZajci[0]*.



Seveda pa bomo še vedno govorili, da imamo zajca shranjenega v spremenljivki *tabZajci[0]*. Zapišimo še celoten program za našo farmo zajcev, skupaj za razredom *Zajec*:

```
public class Zajec
{
   public string serijska;
   public bool spol;
   public double masa;
}

public static void Main(string[] ar)
{
  Zajec[] zajci;
  zajci = new Zajec[250];  //  na farmi imamo do 250 zajecv
  int i = 0;
  int kolikoZajcev = 10;  //  trenutno imamo 10 zajcev
  while (i < kolikoZajcev)
  {
     zajci[i] = new Zajec();  //  "rodil" se je nov zajec
     zajci[i].serijska = "1238-12-" + i;
     zajci[i].spol = false;
     zajci[i].masa = 0.12;
     i++;
```

```
   }
}
```

Zgled - Zgoščenka

Pesem na zgoščenki je predstavljena z objektom razreda *Pesem*:

```
public class Pesem
{
    public string naslov;
    public int minute;
    public int sekunde;
    public Pesem(string nasl, int min, int sek)
    {
        naslov = nasl; minute = min; sekunde = sek;
    }
}
```

Objekt *new Pesem("Echoes",15,24)* predstavlja pesem "Echoes", ki traja 15 minut in 24 sekund. Sestavimo razred *Zgoscenka*, ki vsebuje naslov zgoščenke, ime izvajalca in tabelo pesmi na zgoščenki. Razredu *Zgoscenka* bomo dodali še objektno metodo *Dolzina()*, ki vrne skupno dolžino vseh pesmi na zgoščenki, izraženo v sekundah.

```
public class Zgoscenka
{
   public string avtor;
   public string naslov;
   public Pesem[] seznamPesmi;  // Napoved tabele pesmi (tabela objektov)
   // s konstruktorjem določimo avtorja, naslov in vse pesmi
   public Zgoscenka(string avtor, string naslov, Pesem[] seznamPesmi)
   {
     this.avtor = avtor;
     this.naslov = naslov;
     //inicializacija tabele pesmi
     this.seznamPesmi = new Pesem[seznamPesmi.Length];
     for (int i = 0; i < seznamPesmi.Length; i++)
     {
       // tabelo pesmi določimo s pomočjo parametra (tabele) seznamPesmi
       this.seznamPesmi[i] = new Pesem(seznamPesmi[i].naslov,
                  seznamPesmi[i].minute,seznamPesmi[i].sekunde);
     }
   }
   public int Dolzina() // metoda za izračun skupne dolžine vseh skladb
   {
     int skupaj = 0;
     for (int i = 0; i < seznamPesmi.Length; i++)
       skupaj = skupaj+seznamPesmi[i].minute * 60 + seznamPesmi[i].sekunde;
     return skupaj;
   }
}

static void Main(string[] args)
{
   Zgoscenka CD = new Zgoscenka("Abba", "Waterloo",
                       new Pesem[] { new Pesem("Waterloo", 3, 11),
                                     new Pesem("Honey Honey", 3, 4),
                                     new Pesem("Watch Out", 3, 22)
```

```
                                    });
   // Izpis albuma
   Console.WriteLine("Zgoščenka skupine: "+CD.avtor+"\n\nNaslov albuma:
                     "+CD.naslov);
   // izpiiemo celoten seznam (tabelo) pesmi
   for (int i = 0; i < CD.seznamPesmi.Length; i++)
      Console.WriteLine("Pesem št."+(i+1)+": "+CD.seznamPesmi[i].naslov);
  // pokličemo še metodo Dolzina objekta CD, ki vrne skupno dolžino vseh
  //skladb
  Console.WriteLine("Skupna dožina pesmi je "+CD.Dolzina()+" sekund!");
}
```

Zgled - Firma

Napišimo razred *Zaposleni*, ki zajema osnovne podatke o zaposlenem delavcu v neki firmi. Napišimo program, ki bo omogočal vnos podatkov v tabelo Firma (to ja tabela objektov, ki zajema vse zaposlene v podjetju), ter izpis te tabele.

```
public class Zaposleni
{
    public string imeInPriimek;
    public double dohodek;

    public Zaposleni(string ime, double letniDohodek) //konstruktur
    {
        imeInPriimek = ime;
        dohodek = letniDohodek;
    }
}

static void Main(string[] args)
{
    ////napoved tabele 5 objektov tipa Zaposleni
    Zaposleni[] Firma = new Zaposleni[5];
    for (int i = 0; i < Firma.Length; i++) //Vnos podatkov
    {
        Console.Write("Ime in priimek: ");
        string ime=Console.ReadLine();
        Console.Write("Dohodek: ");
        double dohodek=double.Parse(Console.ReadLine());
        Firma[i]=new Zaposleni(ime,dohodek); //ustvarimo novega zaposlenega
    }
    Console.WriteLine("Tabela zaposlenih in njihovih letnih dohodkov: ");
    //izpis podatkov iz tabele vseh zaposelnih
    for (int i = 0; i < Firma.Length; i++)
    {
        Console.WriteLine(Firma[i].imeInPriimek+": "+Firma[i].dohodek);
    }
    Console.ReadLine();
}
```

Več konstruktorjev

Za metode, tako "navadne" kot objektne, velja, da so lahko preobtežene. Popolnoma enako velja tudi za konstruktorje. Razred lahko vsebuje več konstruktorjev, ki pa se morajo razlikovati v parametrih (v podpisu).

Pri pisanju konstruktorjev pa je pomembno naslednje: če konstruktorja ne napišemo sami, potem ga C# avtomatično doda sam (privzeti konstruktor, brez parametrov). Kakor hitro pa napišemo poljubni konstruktor, se C# ne vmešava več in v tem primeru svojega konstruktorja NE doda. V ta namen si je dobro zapomniti, da v razredih vedno napišemo tudi privzeti konstruktor, ki je brez parametrov!

Kot primer razreda z več konstruktorji vzemimo razred *Zajec* iz prejšnjih poglavij. Uporabniki bi poleg privzetega zajca, radi še možnost, da bi takoj, ko zajca naredijo, le-temu določili še serijsko številko. Radi bi torej konstruktor

```
public Zajec(string serijskaSt)
```

Poleg tega pa včasih na farmo dobijo tudi pošiljko samic. Torej potrebujejo še konstruktor

```
public Zajec(bool spol)
```

Včasih pa so zajci "nestandardni"

```
public Zajec(string serijskaSt, bool spol, double teza)
```

Potrebovali bi torej razred *Zajec* , kim ima poleg osnovnega (oz. privzetega) še tri konstruktorje. A ker morajo imeti vsi konstruktorji ime točno tako, kot je ime razreda, to pomeni, da se morajo razlikovati v tipu oz. številu parametrov . Konstruktorje razreda *Zajec* bi tako napisali takole:

```
public class Zajec
{
   public string serijska;
   public bool spol;
   public double masa;
   public Zajec() //Osnovni konstruktor brez parametrov
   {
        this.spol = true; // vsem zajcem na začetku določimo moški spol
        this.masa = 1.0; // in tehtajo 1kg
        this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
   }
   //Preobteženi konstruktor s parametrom tipa string za nastavljanje
   //serijske številke
   public Zajec(string serijskaSt)
   {
       this.serijska = serijskaSt;
   }
   //Preobteženi konstruktor s parametrom tipa bool za nastavljanje spola
   public Zajec(bool spol)
   {
       this.spol = spol;
   }
   //Preobteženi konstruktor s tremi parametri za nastavljanje vseh treh
   //polj
   public Zajec(string serijskaSt, bool spol, double masa)
   {
       this.serijska = serijska;
       this.masa = masa;
       this.spol = spol;
   }
}
```

Ustvarimo sedaj štiri zajce, vsakič pa uporabimo drug konstruktor:

```
//ustvarimo novega zajca in mu s konstruktorjem določimo privzete vrednosti
//polj
Zajec prvi = new Zajec();
//ustvarimo novega zajca in mu s konstruktorjem določimo serijsko številko;
//spol je privzet (false), masa prav tako (0)
Zajec drugi = new Zajec("12001-01-1");
//ustvarimo novega zajca in mu s konstruktorjem določimo spol; serijska
//številka je privzeta (prazen string), masa prav tako (0)
Zajec tretji = new Zajec(true);
//ustvarimo novega zajca in mu s konstruktorjem določimo serijsko, spol in
//maso
Zajec cetrti = new Zajec("12001-01-2",true,1.55);
```

Pri pisanju konstruktorjev se lahko sklicujemo tudi na že napisani konstruktor (oz konstruktorje). Četrti konstruktor bi lahko tako zapisali tudi takole:

```
public Zajec(string serijskaSt, bool spol, double masa)
   :this (serijskaSt) //sklic na drugi konstruktor
{
   this.masa = masa;
   this.spol = spol;
}
```

Zgledi

Datum

Denimo, da v naših programih pogosto delamo z datumi. Zato bomo sestavili ustrezni razred. Najprej moramo pripraviti načrt razreda. Odločiti se moramo torej, kako bi hranili podatke o datumu. Podatki, ki sestavljajo datum, so dan (število), mesec (izbira: število ali pa niz) in leto (število). Denimo, da se odločimo, da bomo dan in leto hranili kot število, mesec pa kot niz (torej kot januar, februar, ...). Sedaj se moramo odločiti, katere metode bo poleg konstruktorjev poznal naš razred. Nekaj idej:

- Lepo izpiši
- Povečaj za 1 dan
- Je datum smiselen
- Je leto prestopno
- Nov datum za toliko in toliko dni pred/za danim datumom
- Dan v tednu
- ...

Potem, ko smo si pripravili načrt, se lotimo sestavljanja razreda in napišimo nekatere od prej naštetih metod:

```
public class Datum
{
    public int dan;
    public string mesec;
    public int leto;
    public Datum() //osnovni konstruktor
    {
      dan = 1;
      mesec = "januar";
      leto = 2000;
    } // privzeti datum je torej 1.1.2000
    //Dodajmo še dva konstruktorja
    public Datum(int leto) : this()
    {
      this.leto = leto; // this je nujen
```

```
    } // datum je torej 1.1.leto
    public Datum(int d,string m,int l):this(l)//upoštevamo prejšnji
                                              //konstruktor
    { // leto smo torej že nastavili
       this.mesec = m; // this ni nujen
       this.dan = d;
    } // datum je torej d.m.l
   //Sedaj se lotimo metod. Kot prvo, nas zanima, ali je leto prestopno
   public bool JePrestopno()
   {
      int leto = this.leto;
      if (leto % 4 != 0) return false;
      if (leto % 400 == 0) return true;
      if (leto % 100 == 0) return false;
      return true;
   }
}
```

Razred *Datum* smo uspešno naredili, sedaj pa ga še koristno uporabimo. Denimo, da želimo ugotoviti, če je letošnje leto prestopno. Dovolj bo, če le naredimo objekt, v katerega shranimo katerikoli letošnji datum in pokličemo pripravljeno metodo *JePrestopno*().

```
static void Main(string[] args)
{
    Datum danes = new Datum(30, "marec", 2009);
    if (danes.JePrestopno())
    {
        Console.WriteLine("Je prestopno leto");
    }
    else
    {
        Console.WriteLine("Ni prestopno leto");
    }
}
```

Kompleksna števila

Deklarirajmo razred *Complex*, ki naj predstavlja kompleksno število. Struktura naj ima dva konstruktorja (privzetega in še enega za nastavljanje realne in imaginarne komponente). Napišimo še metodo za izpis kompleksnega števila, nato pa kreirajmo tabelo 10 kompleksnih števil in jo inicializirajmo z naključnimi vrednostmi obeh komponent.

```
public class Complex
{
    public float realna;
    public float imaginarna;
    public Complex() // privzeti konstruktorr
    {
        realna = 0;
        imaginarna = 0;
    }
    //Konstruktor za nastavljanje vrednosti polj
    public Complex(float real, float imag)
    {
        realna = real;
        imaginarna = imag;
    }
```

```
    //Objektna metoda ki kompleksno število vrne kot string
    public string ToString(Complex C)
    {
        return (C.realna + " + ( " + C.imaginarna + " * i )");
    }
}

public static void Main()
{
    Complex[] tabK = new Complex[10]; //tabela 10 kompleksnih števil
    Random naklj = new Random();
    for (int i = 0; i < tabK.Length; i++)
    {
        tabK[i] = new Complex(naklj.Next(-10, +10), naklj.Next(-10, +10));
        Console.WriteLine("Kompleksno število št "+(i+1) + ": " + tabK[i].
           ToString (tabK[i]));
    }
    Console.ReadLine();
}
```

Padavine

Za vodenje evidence o padavinah v zadnjem letu potrebujemo naslednje podatke: ime kraja in podatke o količini padavin v zadnjih 12 mesecih (12 števil v polju/tabeli). Kreirajmo ustrezeno razred in metodi za vnos podatkov o padavinah za vseh 12 mesecev, ter metodo, ki vrne povprečno mesečno količino padavin ustreznega kraja!

```
public class padavine
{
    public string ime_kraja;
    public double[] kolicina; //deklaracija tabele padavin
    public padavine()//privzeti konstruktor
    {
        kolicina = new double[12]; //inicializacija tabele padavin
    }
    public padavine(string ime) //konstruktor ki nastavi ime kraja
    {
        ime_kraja = ime;
        kolicina = new double[12]; //inicializacija tabele padavin
    }
    public void vnos() //metoda za vnos količine padavin kraja
    {
        Console.WriteLine("\nVnos mesečne količine padavin za kraj: " +
                          ime_kraja);
        for (int i = 0; i < 12; i++)
        {
            Console.Write("Količina padavin v " + (i + 1) + ". mesecu: ");
            kolicina[i] = Convert.ToDouble(Console.ReadLine());
        }
    }
    //metoda za izračun povprečne količine padavin določenega kraja
    public double povp()
    {
        double vsota = 0;
        for (int i = 0; i < 12; i++)
            vsota += kolicina[i];
        //rezultat vrnemo zaokrožen na dve decimalki
        return (Math.Round(vsota / 12, 2));
```

```
    }
};

//glavni program
static void Main(string[] args)
{
    padavine kraj1 = new padavine("Kranj");
    kraj1.vnos(); //klic metode za vnos padavin kraja
    Console.WriteLine("Povprečna količina padavin v mestu " +
                  kraj1.ime_kraja + " je bila " + kraj1.povp());
    Console.ReadLine();
}
```

Točka

Napišimo razred *Tocka*, ki nja ima dve zasebni polji (koordinati točke), privzeti konstruktor, ki obe koordinati postavi na 0, ter konstruktor z dvema parametroma, s katerima inicializiramo koordinati nove točke. Napišimo še metodo, ki izračuna in vrne razdaljo točke od neke druge točke.

```
class Tocka
{
    public Tocka()//lasten privzeti konstruktor
    {
       x = 0;
       y = 0;
    }

    public Tocka(int initX, int initY) //Konstruktor z dvema parametroma
    {
       x = initX;
       y = initY;
    }
    //metoda za izračun razrdalje od poljubne točke
    public double RazdaljaOd(Tocka druga)
    {
       int xRazd = x - druga.x;
       int yRazd = y - druga.y;
       return Math.Sqrt(Math.Pow(xRazd,2) + Math.Pow(yRazd,2));
    }
    private int x, y;
}

static void Main(string[] args)
{
    Tocka A = new Tocka();  //Klic konstruktorja brez parametrov
    Tocka B = new Tocka(600, 800); //Klic konstruktorja z dvema parametroma
    double razdalja = A.RazdaljaOd(B); //Izračun razdalje obeh točk
    Console.WriteLine("Razdalja točke A od točke B je enaka " + razdalja+"
                      enot!");
}
```

Dostop do stanj objekta

Možnost, da neposredno dostopamo do stanj/lastnosti objekta **ni najboljši!** Glavni problem je v tem, da na ta način ne moremo izvajati nobene kontrole nad pravilnostjo podatkov o objektu! Tako lahko npr. za našega zajca *rjavka* v programu mirno napišemo

```
rjavko.masa = -3.2;
```

kar pa je seveda traparija. Ker ne moremo vedeti, ali so podatki pravilni, so vsi postopki (metode) po nepotrebnem bolj zapleteni, oziroma so naši programi bolj podvrženi napakam. Ideja je v tem, da naj objekt sam poskrbi, da bo v pravilnem stanju. Seveda potem moramo tak neposreden dostop do spremenljivk, ki opisujejo objekt, preprečiti.
Dodatna težava pri neposrednem dostopu do spremenljivk, ki opisujejo lastnosti objekta se pojavi , če moramo kasneje spremeniti način predstavitve podatkov o objektu. S tem bi "podrli" pravilno delovanje vseh starih programov, ki bi temeljili na prejšnji predstavitvi.

Objekt bomo zaradi tega velikokrat imeli za "črno škatlo" in njegove notranje zgradbe ne bomo "pokazali javnosti". Zakaj je to dobro? Če malo pomilsimo, uporabnika razreda pravzaprav ne zanima, kako so podatki predstavljeni. Če podamo analogijo z realnim svetom: ko med vožnjo avtomobila prestavimo iz tretje v četrto prestavo, nas ne zanima, kako so prestave realizirane. Zanima nas le to, da se stanje avtomobila (prestava) spremeni. Ko kupimo nov avto nam je načeloma vseeno, če ima ta drugačno vrsto menjalnika, z drugačno tehnologijo. Pomembno nam je le, da se način prestavljanja ni spremenil, da še vedno v takih in takih okoliščinah prestavimo iz tretje v četrto prestavo.

S "skrivanjem podrobnosti" omogočimo, da se v primeri ko pride do spremembe tehnologije, za uporabnika se stvar ne spremeni. V našem primeru programiranja v C# bo to pomenilo, da če spremenimo razred (popravimo knjižnico), se za uporabnike razreda ne bo nič spremenilo.

Denimo, da so v najnovejši verziji programskega jezika C# spremenili način hranjenja podatkov za določanje naključnih števi v objektih razreda *Random*. Ker dejansko nimamo vpogleda (neposrednega dostopa) v te spremenljivke, nas kot uporabnike razreda *Random* ta sprememba nič ne prizadane. Programe še vedno pišemo na enak način, kličemo iste metode. Skratka - ni potrebno spreminjati programov, ki razred *Random* uporabljajo. Še posebej je pomembno, da programi, ki so delovali prej, še vedno delujejo (morda malo bolje, hitreje, a bistveno je, da delujejo enako).

In še enkrat: s tem ko "ne dovolimo" vpogleda v zgradbo objekta, lahko poskrbimo za zaščito pred nedovoljenimi stanji. Do sedaj smo spremenljivke, ki opisujejo objekt, kazali navzven (imele so določilo **public**). To pomeni, da uporabnik lahko neposredno dostopa do teh spremenljivk in morebiti objekt spravi v nemogoče stanje, npr.

```
mojAvto.prestava = -10;
```

Dostopi do stanj

Do sedaj smo pisali programe tako, da smo naredili objekt. Ko smo želeli v objekt zapisati kak podatek, smo uporabili zapis:

```
ime_objekta.stanje = <izraz>;
```

Za primer vzemimo razred *Clan* in primer kreiranja novega člana

```
public class Clan
{
    public string ime;
    public string priimek;
    public int leto_vpisa;
    public string vpisna_st;
}
static void Main(string[] args)
{
    Clan novClan = new Clan();  //Nov objekt razreda Clan
```

```
    novClan.ime = "Katarina";
    novClan.leto_vpisa = 208;
}
```

Problem s takim pristopom nam kaže ravno vrstica, kjer smo se zatipkali in vnesli nemogoč podatek – za leto vpisa smo vnesli 208. Objekt *novClan* sedaj hrani napačen podatek. Radi pa bi, da take tipkarske napake "polovimo". Seveda bi lahko rekli – v redu, napisali bomo metodo, s katero bomo nastavljali leto vpisa in v metodi preverili, če je podatek smiselen. Takega pristopa smo se do sedaj že večkrat poslužili v zgledih in vajah.

Objektna metoda (znotraj razreda *Clan*) za nastavljanje leta vpisa bi izgledala npr. takole:

```
public bool SpremeniLetoVpisa(int leto)
{
  if ((2000 <= leto) && (leto <= 2020))
  {
    this.leto_vpisa = leto;
    return true;//leto je smiselno, popravimo stanje objekta in vrnemo true
  }
  return false; // leto ni smiselno, ne spremenimo nič in vrnemo false
}
```

In še zgled uporabe tega razreda

```
1:  Clan novClan = new Clan();
2:  novClan.ime = "Katarina";
3:  novClan.leto_vpisa = 2007;
4:  novClan.SpremeniLetoVpisa(208);
5:  novClan.SpremeniLetoVpisa(2008);
```

Sedaj v 4. vrstici podatek ne bo popravljen na napačnega, ker letnica ni ustrezna, v 5. vrstici pa bo sprememba uspešna. Če bi želeli preveriti, ali je sprememba uspela ali ne, bi lahko pogledali, kakšno vrednost je vrnila metoda. Po drugi strani pa že iz vrstice 5 vidimo, da je naša "zaščita" zelo pomanjkliva. Če bi napisali še

```
novClan.leto_vpisa = 20008;
```

bi se vrstica veselo izvedla in spet bi imeli v objektu napačen podatek.

Če bi torej preprečili nastavljanje polj (lasnosti) objekta na način

```
ime_objekta.stanje = <izraz>;
```

in bi se stanje spremenljivk, ki opisujejo objekt lahko spreminjalo le preko metod, bi lahko poskrbeli, da bi pred spremembo podatka izvedli ustrezne kontrole in v primeru poskusa nastavljanja napačnih podatkov ustrezno reagirali.

V ta namen imamo v C# možnost, da dostop do spremenljivk lahko nadziramo. V C# poznamo 4 načine dostopa:

- public
- private
- protected
- internal

Zadnjih dveh mi ne bomo uporabljali, zato si njihov pomen in uporabo oglejte kar sami. Dostop *public* smo že ves čas uporabljali, saj smo pisali na primer:

```
public string serijska;
```

```
public double masa;
```

Besedi *public* pomenita, da do teh dveh spremenljivk dostop ni omejen (je **javen** – *public*). Če tega ne želimo, *public* zamenjamo s *private*

```
private string serijska;
private int[] tab;
private Datum datumRojstva;
```

Včasih besedo, ki označuje način dostopa (javen ali privaten) spustimo in napišemo le

```
bool spol;
```

Kakšen dostop velja takrat, je odvisno od okoliščin. Običajno bo to kar *public*, ne pa vedno. Zato se opuščanju navedbe dostopa izogibajmo in ga pišimo vedno. Zakaj bi zgubljali čas s premišljevanjem o tem, v kakšnih okoliščinah je ta spremenljivka in kakšen bo potem dostop do nje.

Oglejmo si sedaj razliko med *public* in *private*. Zavedati pa se moramo, da znotraj razreda (torej ko sestavljamo razred in pišemo njegove metode**) ni omejitev**. Vedno (ne glede na način dostopa) je možen dostop do komponent – vse metode, napisane znotraj zareda, imajo neomejen dostop do vseh polj, ne glede na to, ali so polja javna (*public*) ali zasebna (*private*).

Omejitve pri načinu dostopa veljajo le, ko objekte določenega razreda uporabljamo – torej v drugih programih in razredih!

public

Če je način dostopa nastavljen na *public*, to pomeni, da do lastnosti (komponent, spremnljivk, polj ...) lahko dostopajo vsi, od kjerkoli (iz katerihkoli datotek (razredov)) in sicer z

```
ime_objekta.lastnost
```

Denimo, da smo v razredu `MojObjekt` napisali

```
public int javnaLastnost;
```

Kdorkoli naredi objekt vrste *MojObjekt*, npr. s stavkom:

```
MojObjekt x = new MojObjekt();
```

lahko dostopa do *javnaLastnost* neposredno:

```
x.javnaLastnost
```

Ta način smo uporabljali do sedaj.

private

Dostop *private* pomeni, da do lastnosti ne more dostopati nihče, razen metod znotraj razreda. Denimo, da smo v razredu `MojObjekt` napisali

```
private int privatnaLastnost;
```

Če sedaj naredimo objekt vrste *MojObjekt*:

```
MojObjekt x = new MojObjekt();
```

bo ob poskusu dostopa do *privatnaLastnost* na takle način:

```
x.privatnaLastnost
```

prevajalnik javil napako.

Zgled - Razred Zajec

```
public class Zajec
{
  public string serijska; // serijska stevilka zajca
  public bool spol; // true = moski, false = zenska
  private double masa; // masa zajca ob zadnjem pregledu
  public SpremeniTezo(double x) //Metoda za spremembo mase zajca
  {
     this.masa = x;//OK! Objektna metoda ima dostop tudi do zasebnega polja
  }
}

public static void Main(String[] ar)
{
  Zajec z1 = new Zajec();
  z1.serijska = "1238-12-0";
  z1.spol = false;
  z1.SpremeniTezo(0.11);
  z1.masa = 0.12;
  z1.masa = z1.masa + 0.3;
  System.Console.WriteLine("Zajec ima ser. št.:" + z1.serijska);
}
```

OK pri prevajanju!

Prevajalnik javi napako!

Dostop do stanj/lastnosti

Odločili smo se torej, da bomo iz objektov naredili črne škatle. Uporabnikom bomo z zasebnimi poli (*private*) preprečili, da bodo "kukali" v zgradbo objektov določenih razredov. Če bodo želeli dostopati do lastnosti (podatkov) objekta (zvedeti, kakšne podatke v objektu hranimo) ali pa te podatke spremeniti, bodo morali uporabljati metode. In v teh metodah bo sestavljalec razreda lahko poskrbel, da se s podatki "ne bo kaj čudnega počelo".

Potrebujemo torej:

- Metode za dostop do stanj (za dostop do podatkov v objektu)
- Metode za nastavljanje stanj (za spreminjanje podatkov o objektu)

Prvim pogosto rečemo tudi "*get*" metode (ker se v razredih, ki jih sestavljajo angleško usmerjeni pisci, te metode pogosto prično z *Get* (Daj)). Druge pa so t.i. "*set*" metode (set / nastavi)[1].

Ponovimo še enkrat, zakaj je pristop z *get* in *set* metodami boljši od uporabe javnih spremenljivk. Spotoma pa bomo navedli še kak razlog in (upamo) prepričali še zadnje dvomljvce, ki se tako težko poslovijo od javnih spremenljivk.

[1] Mimogrede, če bi šli v podrobnosti jezika C#, bi videli, da lahko določene komponente proglasimo za lastnosti (*Property*), ki zahtevajo, da jim napišemo metodi z imenom get in set in potem lahko uporabljamo notacijo *imeObjekta.imeLastnosti*. Na zunaj je videti, kot bi imeli javne spremenljivke. V resnici pa je to le "zamaskiran" klic *get* oziroma *set* metode. Mi bomo na začetku lastnosti spustili in bomo pisali "svoje" *get* in *set* metode.

Razlogi, zakaj je dostop do podatkov v objektu preko metod boljši kot neposredni dostop (preko javnih spremenljivk) so med drugim:

- Možnost kontrole pravilnosti!
- Možnost kasnejše spremembe načina predstavitve (hranjenja podatkov).
- Možnost oblikovanja pogleda na podatke:
- Podatke uporabniku posredujemo drugače, kot jih hranimo. Tako denimo v razredu *Datum* mesec hranimo kot število, proti uporabniku pa je zadeva videti, kot bi uporabljali besedni opis meseca.
- Dostop do določenih lastnosti lahko omejimo:
    - Npr. spol lahko nastavimo le, ko naredimo objekt (kasneje pa uporabnik spola sploh ne more spremeniti, saj se ne spreminja ... če odmislimo kakšne operacije, določene vrste živali ... seveda)
    - Hranimo lahko tudi določene podatke, ki jih uporabnik sploh ne potrebuje ..

Nastavitve podatkov (stanj)

Potrebujemo torej metode, ki bodo nadomestile prireditveni stavek. Za tiste podatke, za katere želimo, da jih uporabnik spreminja, napišemo ustrezno *set* metodo. Na primer:

```
zajcek.SpremeniTezo(2.5);
```

V bistvu smo s tem dosegli isto kot prej s prireditvenim stavkom

```
zajcek.masa = 2.5;
```

A metoda *spremeniTezo* lahko **PREVERI**, če je taka sprememba teže smiselna! Tako je zapis

```
zajcek.SpremeniTezo(-12.5);
```

načeloma nadomestek stavka

```
zajcek.masa = -12.5;
```

Vendar gre tu za bistveno razliko. Prireditveni stavek se bo zagotovo izvedel (in bo s tem zajec nevarno shujšal). Pri spreminjanju teže zajca z metodo, pa lahko **preprečimo** postavitev lastnosti objekta v napačno stanje! V metodi *SpremeniTezo* lahko izvedemo ustrezne kontrole in če sprememba ni smislena, je sploh ne izvedemo.

Najprej poskusimo takole:

```
public void SpremeniTezo(double novaTeza)
{
   // smiselna nova teza je le med 0 in 10 kg
   if ((0 < novaTeza) && (novaTeza <= 10))
      this.masa = novaTeza;
      // v nasprotnem primeru NE spremenimo teže
}

public bool SpremeniTezo(double novaTeza)
{
   // smislena nova teza je le med 0 in 10 kg
   if ((0 < novaTeza) && (novaTeza <= 10)){
   this.masa = novaTeza;
      return true; // sprememba uspela
   }
   // v nasprotnem primeru NE spremenimo teže
   // in javimo, da spremembe nismo naredili
   return false;
```

```
}
```

Ali imamo lahko obe metodi? Žal ne, saj imata obe enak podpis. Spomnimo se, da podpis sestavljajo ime metode in tipi parametrov. Tip rezultata pa ni del podpisa! Pri sestavljanju razreda se bomo torej odločili za eno teh dveh metod (odvisno od ocenjenih potreb).
Metoda je seveda lahko bolj kompleksna. Denimo da vemo, da se teža zajca med dvema tehtanjima ne more spremeniti bolj kot za 15%. Zato lahko z metodo "polovimo" morda napačne poskuse sprememb.

```
public bool SpremeniTezoVmejah(double novaTeza)
{
   // smislena nova teža je le med 0 in 10 kg
   // in če ni več kot 15% spremembe od zadnjič
   int sprememba = (int)(0.5 + (100 * Math.Abs(this.masa - novaTeza) /
                         this.masa));

   if ((0 < novaTeza) && (novaTeza <= 10) && (sprememba <= 15) )
   {
      masa = novaTeza; // this.masa ... Lahko pa this spustimo!
      return true; // sprememba uspela
   }
   // v nasprotnem primeru NE spremenimo teže
   // in javimo, da spremembe nismo naredili
   return false;
}
```

Kaj pa metodi za spreminjanje serijske številke in spola? Na zadnjo lahko mirno pozabimo. Zajec naj ima ves čas tak spol, kot mu ga določimo pri "stvarjenju" (torej v konstruktorju). Zato *set* metode za spremninjanje spola sploh ne bomo napisali. In ker je spremenljivka spol zaščitena s *private*, je uporabnik ne bo mogel spremeniti. Seveda, če bomo naš razred *Zajec* potrebovali za kakšen genetski laboratorij, kjer se gredo čudne stvari, pa bo morda metoda *SpremeniSpol* potrebna.

Tudi serijske številke verjetno ne bomo spreminjali. No, če malo bolje premislimo, pa nam konstruktor brez parametrov ustvari zajca, ki ima za vrednost serijske številke niz "NEDOLOČENO". Takim zajcem bomo verjetno želeli spremniti serijsko številko. Zato naj metoda *SpremeniSerijsko* pusti spreminjati le nedoločene serijske številke!

```
public bool SpremeniSerijsko(string seStev)
{
   // sprememba dopustna le, če serijske številke še ni
   if (this.serijska.Equals("NEDOLOČENO"))
   {
     this.serijska = seStev;
     return true;
   }
   return false; // ne smemo spremniti že obstoječe!
}
```

Opozorimo še na stvar, ki jo pri sestavljanju razreda pogosto pozabimo. Glavni razlog, da smo napisali te *set* metode je, da bi zagotovili, da so podatki pravilni. Zato je smiselno, da zagotovimo, da je teža ustrezna že ves čas! A zaenkrat smo na nekaj pozabili! Kaj pa začetno stanje, ki ga nastavimo v konstruktorju! Tudi v konstruktorju preverimo, če se uporabnik "obnaša lepo". Pogosto na to pozabimo in uporabnik lahko napiše npr.

```
Zajec neki = new Zajec("X532", true, 105);
```

105 kilogramskega zajca verjetno ni, a uporabnik je pozabil na decimalno piko v 1.05. Zato je kontrola smiselnosti podatkov potrebna tudi v konstruktorjih!

Dostop do stanj

Pogosto nam je vseeno, če uporabnik "vidi", kako hranimo podatke v objektu. A zaradi zagotavljanja pravilnosti in možnosti kontrole sprememb podatkov, smo dostop nastavili na *private*. S tem smo seveda onemogočili tudi enostaven način poizvedbe po vrednosti podatka v obliki *rjavko.masa*. Če bomo torej potrebovali težo zajca, bo potrebno pripraviti ustrezno metodo. Če bomo torej potrebovali težo zajca *zajcek*, bomo napisali

```
zajcek.PovejTezo()
```

Večina tovrstnih *get* metod je enostavnih in v telesu metode vsebujejo le ukaz *return*.

```
public double PovejTezo()
{
   return this.masa; // ali return masa
}
```

Zgled – razred Datum

Kot primer oblikovanja pogleda na podatke si oglejmo še razred *Datum*. Običajno ni smiselno, da podatke o mesecu hranimo kot niz, raje jih hranimo kot celo število. A uporabniku bi jih še vedno raje "servirali" kot *februar* in ne kot *2*. Zato bi metoda *KateriMesec()* bila lahko taka

```
public class Datum
{
  private int dan;
  private int mesec;
  private int leto;

  public Datum(int dan, int mesec, int leto) // konstruktor
  {
     this.dan = dan;
     this.mesec = mesec;
     this.leto = leto;
  }
  public string KateriMesec()
  {
     string[] imenaMesecev ={"januar", "februar", "marec", "april", "maj",
     "junij","julij","avgust","september","oktober","november","december"};

     return imenaMesecev[this.mesec-1]; // metoda vrne opis meseca
  }
}
static void Main(string[] args)
{
    Datum danes=new Datum(1,5,2009);
    //izpis tekočega meseca v opisni obliki
    Console.WriteLine(danes.KateriMesec()); //izpis "maj"
    Console.ReadLine();
}
```

Seveda so potrebne še spremembe v drugim metodah. Npr. uporabnik bo verjetno želel imeti metodi *NastaviMesec*, ki bosta sprejeli bodisi številčni ali pa opisni podatek.

Ostale metode

Poleg *get/set* metod in konstruktorjev v razredu napišemo tudi druge metode, saj objektov ne potrebujemo samo kot "hranilnikov" podatkov. S temi metodami določimo

- odzive objektov
- "znanje objektov"

Če se na primer spomnimo na objekte tipa *string*:

- Znajo povedati, kje se v njih začne nek podniz:
  `"niz".IndexOf("i")`
- Znajo vrniti spremiti črke v male in vrniti nov niz:
  `nekNiz.toLower()`
- Znajo povedati, če so enaki nekemu drugemu nizu:
  `mojPriimek.Equals("Lokar")`

Zato bomo tudi v "naših" razredih sprogramirali znanje objektov določenega razreda s tem, da bomo napisali ustrezne metode. Katere bodo te metode je pač odvisno od načrtovane uporabe naših razredov.

Metoda ToString

Poglejmo si naslednji program:

```
public class Zajec
{
   private string serijska; // serijska stevilka zajca
   private bool spol; // true = moski, false = zenska
   private double masa; // masa zajca ob zadnjem pregledu
   public Zajec() //Osnovni konstruktor brez parametrov
   {
        this.spol = true; // vsem zajcem na začetku določimo moški spol
        this.masa = 1.0; // in tehtajo 1kg
        this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
   }
}

public static void Main(String[] ar)
{
  Zajec zeko = new Zajec();
  Console.WriteLine("To je zajec: " + zeko );
  Console.ReadLine();
}
```

Če ga izvedemo, dobimo na zaslonu približno takle izpis:

Od kod, zakaj? Očitno objekte lahko tudi izpišemo. In če napišemo

```
string niz = "Zajec " + zeko + "!";
```

prevajalnik tudi nič ne protestira. Torej se tudi objekti "obnašajo" tako kot *int*, *double* ... in se torej po potrebi pretvorijo v niz.

Obnašanje, ki smo ga opazili (da se števila, objekti ...) pretvorijo v niz, omogoča metoda `ToString`. Metoda je nekaj posebnega, saj se lahko "pokliče kar sama". Namreč, če na določenem mestu potrebujemo *niz*, a naletimo na objekt, se metoda pokliče avtomatično. Torej

```
string niz = "Zajec " + zeko + "!";
```

dejansko pomeni

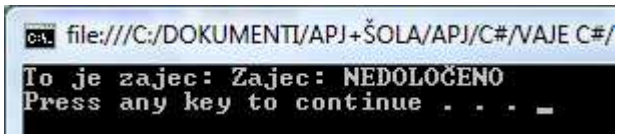
```
string niz = "Zajec " + zeko.ToString() + "!";
```

Od kje pride ta metoda, saj je v razredu *Zajec* nismo napisali. Gre spet za eno od tistih "čarovnij", ki jo počne prevajalnik, kot na primer pri konstruktorjih? Zadeva je malček drugačna in jo bomo *pojasnili* v nadaljevanju. Zaenkrat bodi dovolj le to, da ta metoda vedno obstaja, tudi če je ne napišemo in omogoča pretvorbo poljubnega tipa v niz.

A s pretvorbo nismo ravno zadovoljni. Radi bi kakšne bolj smiselne informacije. To storimo tako, da v razredu, ki ga definiramo (recimo *Zajec*), sami napišemo lastno metodo `ToString`.

```
public override string ToString()
{
    return "Zajec: " + this.PovejSerijsko();
}
```

Če to storimo in zaženemo isti program, dobimo izpis:



Opazimo novo neznano besedo – ***override***. Uporabiti jo moramo zaradi "dedovanja". Več o tem, kaj dedovanje je, kasneje, a povejmo, da smo z dedovanjem avtomatično pridobili metodo `ToString`. To je razlog, da je ta metoda vedno na voljo v vsakem razredu, tudi če je ne napišemo.

Če želimo napisati svojo metodo, ki se imenuje enako kot podedovana metoda, moramo napisati besedico `override`. S tem "povozimo " podedovano metodo.

Seveda bi lahko napisali tudi neko drugo metodo, na primer *Opis*, ki bi prav tako vrnila niz s smiselnim opisom objekta.

```
public string Opis()
{
    return "Zajec: " + this.PovejSerijsko();
}
```

A med metodama *ToString* in *Opis* je bistvena razlika: metodo *Opis* moramo obvezno poklicati sami, metoda *ToString* pa se kliče avtomatsko (če je potrebno). Če torej napišemo

```
string niz1 = "Zajec " + zeko.ToString() + "!";
string niz2 = "Zajec " + zeko + "!";
string niz3 = "Zajec " + zeko.Opis() + "!";
```

so vsi trije dobljeni nizi enaki!

Zapomnimo si torej, da metoda *ToString* obstaja tudi, če jo ne napišemo. A verjetno z vrnjenim nizom nismo najbolj zadovoljni, zato jo praktično vedno napišemo sami.

Zgled: celotni razred Zajec

Oglejmo si sedaj celotno kodo razreda *Zajec* s spremembami, ki smo jih naredili. Spomnimo se, kaj vse smo opravili:

- Pripravili smo nekaj konstruktorjev
- Zaščitili smo dostop do podatkov (*private*)
- Napisali smo ustrezne metode za spreminjanje podatkov o objektu
- Napisali smo metode za poizvedovanje po vrednosti podatkov
- Napisali smo metodo *ToString*, ki vrne opis objekta
- Napisali smo dve metodi, ki pomenita "znanje" objektov

```
public class Zajec
{
   private string serijska;
   private bool spol;
   private double masa;
   // Osnovni konstruktor
   public Zajec()
   {
      this.spol = true; // vsem zajcem na začetku določimo m. spol
      this.masa = 1.0; // in tehtajo 1kg
      this.serijska = "NEDOLOČENO";
   }
   //Dodatni konstruktor za nastavljanje serijske številke
   public Zajec(string serijskaStev)
        : this() // poklicali smo osnovni konstruktor Zajec()
   {
      this.serijska = serijskaStev;
   }
   //Dodatni konstruktor za nastavljanje vseh treh polj
   public Zajec(string serijskaStev, bool spol, double masa)
   {
      this.serijska = serijskaStev;
      this.SpremeniTezo(masa); // uporabimo metodo za sprem.
      this.spol = spol;
   }

   public double PovejTezo()
   {
      // težo bomo povedali le na 0.5 kg natančno
      int tezaKg = (int)this.masa;
      int decim = (int)((this.masa -tezaKg) * 10);
      if (decim < 3) return tezaKg + 0.0;
      if (decim < 8) return tezaKg + 0.5;
      return tezaKg + 1.0;
   }

   public bool SpremeniTezo(double novaTeza)
   {
     // smislena nova teža je le med 0 in 10 kg
     if ((0 < novaTeza) && (novaTeza <= 10)){
        masa = novaTeza;
        return true; // sprememba uspela
     }
```

```
      // v nasprotnem primeru NE spremenimo teže
      // in javimo, da spremembe nismo naredili
      return false;
    }

    public string PovejSerijsko()
    {
      return this.serijska;
    }

    public bool SpremeniSerijsko(string seStev)
    {
      // sprememba dopustna le, če serijska številka še ni določena
      if (this.serijska.Equals("NEDLOČENO"))
      {
        this.serijska = seStev;
        return true;
      }
      return false; // ne smemo spremeniti že obstoječe!
    }

    public bool JeSamec()
    {
      return this.spol;
    }

    // ker se spol naknadno NE spremeni, metode za
    // spreminjanje spola sploh ne ponudimo uporabniku!
    public double Vrednost(double cenaZaKg)
    {
      // pove vrednost zajca
      // zajce tehtamo na dogovorjeno natančnost
      return cenaZaKg * this.PovejTezo();
    }

    public bool MeNapojiti()
    {
      //ugotovimo, če ga je smiselno napojiti, da bomo "ujeli" naslednjih
      //pol kg! Dejanska teža je večja od "izmerjene"
      return (this.masa - this.PovejTezo() > 0);
    }

    public override string ToString()
    {
         return "Zajec: " + this.PovejSerijsko();
    }
}
```

Uporaba razreda Zajec

Poglejmo si, kako opravljene spremembe vplivale na uporabo rezreda Zajec v uporabniških programih. Pred spremembami so bili programski stavki približno takšni:

```
Zajec z1 = new Zajec();
z1.serijska = "1238-12-0";
z1.spol = false;
z1.masa = 0.12;
```

```
z1.masa = z1.masa + 0.3;
System.Console.WriteLine("Zajec ima ser. št.:" + z1.serijska);
```

Z novim razredom *Zajec* seveda se način uporabe povsem spremeni, saj ne moremo pričakovati, da bodo tako drastične spremembe, ki smo jih naredili, pustile uporabniške programe nespremenjene. Še enkrat povejmo, da razreda *Zajec* ne bi nikoli "zares" napisali tako, kot smo ga po starem (z javnimi spremenljivkami). Uporaba novega razreda *Zajec* bi bila npr. takale:

```
// novega zajca stvarimo s konstruktorjem
Zajec z1 = new Zajec("1238-12-0", false, 0.12);
// uporabimo nekatere izmed metod razreda Zajec
if (z1.SpremeniTezo(z1.PovejTezo() + 0.3))
    System.Console.WriteLine("Teža je spremenjena na " + z1.PovejTezo());
else System.Console.WriteLine("Teža je nespremenjena!"
     + " Prirastek je prevelik! Preveri!");
System.Console.WriteLine("Zajec ima ser. št.:" + z1.PovejSerijsko());
```

Ustvarjanje razredov in objektov: povzetek

Naredimo sedaj povzetek vsega povedanega.:

Konkretni objekt je primerek nekega razreda. Nastane s pomočjo operatorja *new*. Izjema so na prvi pogled objekti razreda *string*. A dejansko ne gre za izjemo, gre le drugačen zapis, ki "skrije" uporabo operatiorja *new*! Podobno je pri tabelah ko uporabimo inicializacijo začetnih vrednosti, naštetih v zavitih oklepajih.

Posamičnim lastnostim objekta vedno določimo privatni način dostopa (da so torej dostopni le metodam znotraj razreda). To povemo tako, da ob deklaraciji napišemo *private*. Razlog za to je v tem, da uporabnika ne zanima način hranjenja lastnosti objekta, ampak le smiselna uporaba objektov. S tem tudi omogočimo bolj kontrolirano delo s podatki v objektu. V ta namen pripravimo metode za delo s podatki/lastnosti kot so :

```
VrniPodatek, NastaviPodatek, SpremeniPodatek, ...
```

V razredu moramo obvezno imeti tudi konstruktorje. To so metode, ki povedo, kaj se zgodi ob kreaciji novega objekta. Prvenstveno poskrbijo, da so vsi podatki o objektu takoj nastavljeni na smislene vrednosti.

Ko pišemo metode, s katerimi nastavljamo vrednost (*set*), ne smemo pozabiti na preverjanje pravilnosti parametrov. Stanje objekta mora biti ves čas pravilno. Že pri pisanju konstruktorjev ne smemo pozabiti na ustrezno preverjanje parametrov. Kot smo omenili, morajo tudi ti poskrbeti, da je novoustvarjeni objek v "smiselnem stanju").

Pri metodah, s katerimi vračamo vrednost (t.i. *get* metode, večjih zapletov ne gre pričakovati. Seveda pa lahko metode izkoristimo, da podatke, ki jih v objektu hranimo na en način, vrnemo na drug način (npr. podatek o mesecu, ki je hranjen številčno vrnemo opisno).

Vsak naš razred naj bi imel tudi metodo *ToString*. Ta poskrbi, da se bodisi ob klicu, bodisi avtomatično (če se na tistem mestu namesto objekta pričakuje niz) objekt pretvori v niz. Uporabljamo jo za “predstavitev” objekta. Ker zaradi mehanizma dedovanja ta metoda avtomatično obstaja v vsakem razredu, moramo na začetek dodati še besedo *override*.

Poleg metod za nastavljanje/vračanje podatkov in metode *ToString* imamo v razredu običajno še več drugih metod. Tiste, katerih uporaba je namenjena uporabnikom, se začno z določilom *public*, “interne” (pomožne) metode pa so *private*.

Predvsem ori konstruktorjih smo uporabili tehniko preobteževanja (*overloading*). Gre za to, da imamo lahko več metod, ki se imenujeo enako, razlikujejo pa se v seznamu parametrov.

Lastnost (Property)

Polja so znotraj razreda običajno deklarirana kot zasebna (private). Vrednosti smo jim doslej prirejali tako, da smo zapisali enega ali več konstruktorjev, ali pa smo napisali posebno javno metodo (imenovali smo jo *get/set* metodo) za prirejanje vrednosti polj. Ostaja pa še tretji način, ki je namenjen izkušenim programerjem – za vsako polje lahko definiramo ustrezno lastnost (*property*), s pomočjo katere dostopamo do posameznega polja, ali pa z njeno pomočjo prirejamo (nastavljamo) vrednosti polja. Lastnosti v razredih torej uporabljamo za inicializacijo oziroma dostop do polj razreda (objektov). Lastnost (*property*) je nekakšen križanec med spremenljivko in metodo. Pomen lastnosti je v tem, da ko jo beremo, ali pa vanjo pišemo, se izvede koda, ki jo zapišemo pri tej lastnosti. Branje in izpis (dostop) vrednosti je znotraj lastnosti realizirana s pomočjo rezerviranih besed **get** in **set**: ***get*** mora vrniti vrednost, ki mora biti istega tipa kot lastnost (seveda pa mora biti lastnost istega tipa kot polje, kateremu je namenjena), v ***set*** pa s pomočjo implicitnega parametra **value** lastnosti priredimo (nastavimo) vrednost.

Navzven pa so lastnosti vidne kot spremenljivke, zato lastnosti v izrazih in prirejanjih uporabljamo kot običajne spremenljivke.

Zgled:

```
class Demo
{
    int polje;              //zasebno polje
    public Demo()    //konstruktor
    {
       polje = 0;
    }
    //deklaracija lastnosti (property) razreda Demo
    public int MojaLastnost
    {
        get  //metoda get za pridobivaje vrednosti lastnosti polje
        {
            return polje  ;
        }
        set  //metoda set za prirejanje vrednosti lastnosti polje
        {
            polje = value;
        }
    }
}

static void Main(string[] args)
{
    Demo ob = new Demo();//nov objekt razreda Demo
    Console.WriteLine("Originalna vrednost ob.MojaLastnost: " +
                      ob.MojaLastnost);

    ob.MojaLastnost = 100;
    Console.WriteLine("Vrednost ob.MojaLastnost: " + ob.MojaLastnost);

    Console.WriteLine("Prirejanje nove vrednosti -10 za ob.MojaLastnost");
    ob.MojaLastnost = -10;
    Console.WriteLine("Vrednost ob.MojaLastnost: " + ob.MojaLastnost);
}
```

Zgled Oseba

Deklarirajmo razred *Oseba* z dvema poljema (*ime* in *priimek*) ter javno metodo za nastavljanje vrednosti obeh polj. Razred naj ima tudi lastnost *PolnoIme*, za prirejanje in vračanje polnega imena osebe (imena in priimka). Ustvarimo nov objekt in demonstrirajmo uporabo lastnosti *PolnoIme*

```
class Oseba
{
    private string priimek; //zasebno polje razreda Oseba
    private string ime;     //zasebno polje razreda Oseba

    public void NastaviIme(string ime,string priimek)//javna metoda razreda
    {
        this.priimek = priimek;
        this.ime = ime;
    }
    public string PolnoIme   //javna lastnost razreda
    {
        get
        {
            return ime + " " + priimek;
        }
        set
        {
            string zacasna = value;
            //Celotno ime razdelimo na posamezne besede
            string[] imena = zacasna.Split(' ');
            //in jih spravimo  tabelo
            ime = imena[0];  //za ime vzamemo prvi string v tabeli
            //za priimek vzamemo drugi string v tabeli
            priimek = imena[imena.Length - 1];
        }
    }
}

static void Main(string[] args)
{
    Oseba politik = new Oseba();
    politik.NastaviIme("Nelson", "Mandela");
    //Izpis: Polno ime osebe je Neslon Mandela
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
    politik.PolnoIme = "Barack Obama";
    //Izpis: Polno ime osebe je Barack Obama
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
    politik.PolnoIme = "France Prešeren";
    //Izpis: Polno ime osebe je France Prešeren
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
}
```

Za posamezno polje lahko napišemo le eno lastnost, v kateri s pomočjo stavkov **get** ali **set** dostopamo oz. nastavljamo vrednosti posameznega polja.

Besedici **get** ali **set** lahko izpustimo in dobimo ***write-only*** ali ***read-only*** lastnosti.

```
class MojRazred
{
   double A = 3;  //zasebno polje
   double B = 4;  //zasebno polje
```

```
   //MojaVrednost je ReadOnly: vsebuje le get, ne pa tudi set
   public double MojaVrednost
   {
      get { return A * B; } //lastnost vrne vrednost produkta obeh polj
   }
}

static void Main(string[] args)
{
    MojRazred c = new MojRazred();   //nov objekt tipa MojRazred
    Console.WriteLine("MojaVrednost: "+c.MojaVrednost);
}
```

Statične metode

Za lažje razumevanje pojma **statična metoda**, si oglejmo metodo **Sqrt** razreda **Math**. Če pomislimo na to, kako smo jo v vseh dosedanjih primerih uporabili (poklicali), potem je v teh klicih nekaj čudnega. Metodo **Sqrt** smo namreč vselej poklicali tako, da smo pred njo navedli ime razreda (**Math.Sqrt**) in ne tako, da bi najprej naredili nov objekt tipa **Math**, pa potem nad njim poklicali metodo **Sqrt**. Kako je to možno?

Pogosto se bomo srečali s primeri, ko metode ne bodo pripadale objektom (instancam) nekega razreda. To so uporabne metode, ki so tako pomembne, da so neodvisne od kateregakoli objekta. Metoda **Sqrt** je tipičen primer take metode. Če bila metoda **Sqrt** običajna metoda objekta izpeljanega iz nekega razreda, potem bi za njeno uporabo morali najprej kreirati nov objekt tipa **Math**, npr takole:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

Tak način uporabe metode pa bi bil neroden. Vrednost, ki jo v tem primeru želimo izračunati in uporabiti, je namreč neodvisna od objekta. Podobno je pri ostalih metodah tega razreda (npr. Sin, Cos, Tan, Log, …). Razred **Math** prav tako vsebuje polje **PI** (iracionalno število Pi), za katerega uporabo bi potemtakem prav tako potrebovali nov objekt. Rešitev je v t.i. **statičnih poljih oz.metodah**.

V C# morajo biti vse metode deklarirane znotraj razreda. Kadar pa je metoda ali pa polje deklarirano kot **statično** (**static**), lahko tako metodo ali pa polje uporabimo tako, da pred imenom polja oz. metode navedemo ime razreda. Metoda **Sqrt** (in seveda tudi druge metode razreda **Math**) je tako znotraj razreda **Math** deklarirana kot statična, takole:

```
class Math
{
    public static double Sqrt(double d)
    {
        . . .
    }
}
```

Zapomnimo pa si, da statično metodo ne kličemo tako kot objekt. Kadar definiramo statično metodo, le-ta **nima** dostopa do kateregakoli polja definiranega za ta razred. Uporablja lahko le polja, ki so označena kot **static** (statična polja). Poleg tega, lahko statična metoda kliče le tiste metode razreda, ki so prav tako označene kot statične metode. Ne-statične metode lahko, kot vemo že od prej, uporabimo le tako, da najprej kreiramo nov objekt.

Napišimo razred točka in v njem statično metodo, katere naloga je le ta, da vrne string, v katerem so zapisane osnovni podatki o tem razredu

```
class Tocka
{
   public static string Navodila()//Statična metoda
   {
      string stavek="Vsaka točka ima dve koordinati/polji: x je abscisa, y
                     je ordinata!";
      return stavek;
   }
}

//Zato, da pokličemo statično metodo oz. statično polje NE POTREBUJEMO
//OBJEKTA!!!  Primer klica npr. v glavnem programu
Console.WriteLine(Tocka.Navodila());//Klic statične metode
```

Statična polja

Tako kot obstajajo statične metode, obstajajo tudi statična polja. Včasih je npr. je potrebno, da imajo vsi objekti določenega razreda dostop do istega polja. To lahko dosežemo le tako, da tako polje deklariramo kot statično.

```
class Test
{
    public const double Konstanta = 20;//Statično polje
}

//Zato, da pokličemo statično metodo/polje NE POTREBUJEMO OBJEKTA!!!
Console.WriteLine(Test.Konstanta);//klic statičnega polja
```

Primer:

```
public class Foo
{
   static public int x = 12; //x je STATIČNO polje (pripada razredu)
   public int y; //y je običajno, objektno polje (pripada objektom)

   public Foo(int z)
   {
      this.y = z;
   }

   public static int F(int a)
   {
      return x + a;
      //ali return Foo.x + a;
   }

   public int G(int a)
   {
      /*Znotraj metode G() označuje this objekt, na katerem je metoda
        poklicana. Objektna metoda G ima dostop do komponente this.y, kjer
        je this objekt, na katerem je metoda g klicana. Prav tako ima
        dostop do (statične) komponente x. */
        return this.y + Foo.x + a;
       //ali return y + Foo.x + a;
       //ali return  y + x + y;
       //ali return  ...
```

```
    }
}


static void Main(string[] args)
{
    /*Objektno metodo G()v razredu Foo lahko izvedemo, če imamo neki objekt
      obj1 razreda Foo, z ukazom obj1.G().*/

    Foo obj1 = new Foo(12);
    Foo obj2 = new Foo(30);
    /*Oba objekta (obj1 in obj2 ) vsebujeta svojo kopijo komponente y. Ker
    pa je komponenta x statična, vedno obstaja v eni sami kopiji, tudi če
    nimamo nobenih objektov razreda Foo*/

    int p = obj1.G(7);      // p == 12 + 7 == 19
    Console.WriteLine(p);  // Izpis 19
    /*Statična metoda F ima dostop do (statične) komponente x. Dostopa do
    komponente y nima,saj znotraj statične metode ne moremo pisati this.y*/
    Foo.x = -3;

    /*Statično metodo F() v razredu Foo lahko vedno izvedemo z ukazom
     Foo.F(). Znotraj statične metode objekt this ni definiran, ker se
     statična metode ne kliče na objektu.*/
    int q = Foo.F(5);      // q == -3 + 5 == 2
    Console.WriteLine(q); // Izpis 12
    Foo t = new Foo(100);
    Foo s = new Foo(200);
    int r = t.G(50);       // r == 100 + (-3) + 50 == 147
    Console.WriteLine(r); // Izpis 147
    Console.ReadKey();
}
```

Zgled

```
class Nekaj
{
  static int stevilo=0;  //Statično polje
  public Nekaj()         // Konstruktor
  {
     stevilo++;        //število objektov se je povečalo
  }

  public void Hello()
  {
     if (stevilo>3)
     {
        Console.WriteLine("Sedaj smo pa že več kot trije! Skupaj nas je že
                           "+stevilo);
     }
     switch (stevilo)
     {
        case 1 : Console.WriteLine("V tej vrstici  sem sam !!");
                 break;
        case 2 : Console.WriteLine("Aha, še nekdo je tukaj, v tej
                                    vrstici sva dva!!");
                 break;
        case 3 : Console.WriteLine("Opala, v tej vrstici smo že trije.");
                 break;
    }
```

```
  }
}

static void Main(string[] args)
{
   Nekaj a=new Nekaj();              //konstruktor za a
   a.Hello();
   {
      Nekaj b=new Nekaj();          //konstruktor za b
      b.Hello();
      {
         Nekaj c=new Nekaj();      //konstruktor za c
         c.Hello();
         {
            Nekaj d=new Nekaj(); //konstruktor za d
            d.Hello();
         }
      }
   }
}  //Konec programa-> Garbage Collector poskrbi za uničenje vseh objektov
```

Polje razreda je lahko statično a se njegova vrednost ne more spremeniti: pri deklaraciji takega statičnega polja zapišemo besedico **const** (const = Constant – konstanta). Besedica **static** pri deklaraciji konstantnega polja **NI** potrebna, pa je polje še vedno statično. Konstantno polje je torej avtomatično statično in je tako dostopno preko imena razreda in ne preko imena objekta! **Vrednost statičnega polja se NE da spremeniti**. Tako je npr. deklarirana konstanta **PI** razreda **Math**.

Primer:

```
class Test
{
    public static double kons1 = 3.14; //Statično polje
    public const double kons = 3.1416; //Konstantno polje je avtomatično
                                       //tudi statično
}

//Zato, da pokličemo statično polje NE POTREBUJEMO OBJEKTA!!!
Console.WriteLine(Test.kons1);//klic statičnega polja
Console.WriteLine(Test.kons);//klic konstantnega statičnega polja
Test.kons1= Test.kons1 + 1; //OK -> Statično polje LAHKO spremenimo
Test.kons= Test.kons + 1; //NAPAKA, konstantnega polja ne moremo spremeniti
```

Že iz prejšnjih primerov je razvidno, kdaj v razredih uporabljamo statična poja. Uporabljamo jih za različne konstante, za enolične identifikacije (ko naj ima npr. objekt svojo serijsko številko in so te številke zaporedne), vedno, ko je določen podatek skupen za vse objekte tega razreda

Kako vemo, ali naj bo komponenta (polje ali pa metoda) statična ali objektna? Statične komponente so tiste, ki so skupne za celoten razred – niso posebna lastnost enega (ali nekaj) primerkov objektov tega razreda. Lahko so zasebne (*private*) ali pa javne (*public*), odvisno pač od željenega načina dostopa. Če jih želimo skriti pred uporabniki, bo način dostopa *private*.

Zgledi

Kvader

Razred *Kvader* vsebuje tri zasebna polja (robove kvadra), statično polje *kolikoNasJe*, ki hrani podatek o številu živečih objektov razreda *Kvader*, dve statični *readonly* polji *MIN_DOL_STR* (minimalna dolžina roba) in *MAX_DOL_STR* (maksimalna dolžina roba). Za polje, ki je označeno kot *readonly* velja podobno kot za konstante in predstavlja vrednost, ki je kasneje ne moremo več spremeniti. Readonly polje lahko inicializiramo ob deklaraciji ali pa v statičnem konstruktorju (statični konstruktorji se uporabljajo le za inicializacijo statičnih polj). V razredu *Kvader* so štirje konstruktorji.

```
public class Kvader
{
    //polja so private:uporabnika ne zanima, kako imamo shranjene podatke *
    private double a;
    private double b;
    private double c;

    private int serijskaStevilka; // serijska števila objekta

    //skupno število ustvarjenih objektov
    private static int kolikoNasJe = 0;
    //minimalna dolžina roba kvadra
    public static readonly int MIN_DOL_STR = 1;
    //maksimalna dolžina roba kvadra
    public static readonly int MAX_DOL_STR = 100;
    /* Ker menimo, da je smiselno, da uporabnik neposredno vidi ti dve
       količini je dostop public. Spreminjati ju ne more zaradi readonly.
       Dostop: Kvader.MIN_DOL_STR
       Določilo static pomeni, da gre za razredno spremenljivko, torej je
       skupna za vse objekte
     */

    // vrni podatek
    public int PovejA()
    { // omogočimo uporabniku dostop do vrednosti zanj zanimivih podatkov *
        return (int)this.a;
    }
    public int PovejB()
    {
        return (int)this.b;
    }
    public int PovejC()
    {
        return (int)c;
    }

    private bool Kontrola(int x)
    { /* pomožna metoda, zato private */
        // preverimo, ce velja MIN_DOL_STR <= x <= MAX_DOL_STR
        return ((MIN_DOL_STR <= x) && (x <= MAX_DOL_STR));
    }

    // nastavi podatek
    public void StranicaA(int a)
    { /* omogočimo uporabniku spreminjanje podatkov */
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(a))
```

```
            this.a = a;  /* this je potreben, da ločimo med
                imenom podatka objekta in parametrom a */
        /* verjetno bi bilo bolj smiselno namesto
         a parameter poimenovati s ali kako drugače */
    }
    public void StranicaB(int a)
    {
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(a))

            this.b = a; /* this v nasprotju s prejšnjo metodo
               tu ni potreben. Še vedno pa velja, da bi bilo boljše,
               da bi parameter a poimenovali drugače - npr. s */
    }
    public void StranicaC(int s)
    {
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(s)) /* na ta način bi verjetno "zares"
                  sprogramirali tudi zgornji metodi */
            c = s;
    }

    // konstruktorji
    public Kvader()
    {
        a = b = c = 1;
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public Kvader(int s)
    {
        // kocka z robom s. Če podatek ni v redu - kocka z robom 1
        if (!Kontrola(s)) // če podatek ni v mejah,
            s = 1;        // ga postavimo na 1

        a = b = c = s;
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public Kvader(int s1, int s2, int s3)
    { // kvader s1 x s2 x s3
        // Če kak podatek ni v redu - ga postavimo na 1
        if (!Kontrola(s1)) s1 = 1; // če podatek ni v mejah,
          a = s1;  // ga postavimo na 1

        if (!Kontrola(s2)) s2 = 1; // če podatek ni v mejah,
          b = s2;  // ga postavimo na 1


        if (!Kontrola(s3)) s3 = 1; // če podatek ni v mejah,
          c = s3;  // ga postavimo na 1

        kolikoNasJe++;//povečamo število objektov
        serijskaStevilka = kolikoNasJe;
    }
```

```
    public Kvader(Kvader sk)
    {
        // Novi objekt je kopija obstoječega objekta sk
        this.a = sk.a; /* this ni potreben */

        /*tudi v razredu se splača uporabljati metode za delo s podatki
        čeprav imamo neposreden dostop do spremenljivk, saj nam ob
        morebitni spremembi predstavitve ni potrebno toliko popravljati!*/
        StranicaB(sk.PovejB());
        StranicaC(sk.PovejC());
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public static int PovejKolikoNasJe()
    {
        return kolikoNasJe;
    }

    public override string ToString()
    {  /* prekrivanje metode iz razreda Object */
       // Metoda vrne podatek v obliki Kvader: a x b x c
       return "Kvader: "+Kvader.kolikoNasJe+": "+ + PovejA() + " x " +
              PovejB() + " x " + PovejC();
    }

} // class Kvader
static void Main(string[] args)
{
    //Uporabimo privzeti konstruktor: vsi trije robovi bodo enaki 1
    Kvader K1 = new Kvader();
    Console.WriteLine(K1.ToString());

    //Uporabimo drugi konstruktor: vsi trije robovi bodo enaki 5
    Kvader K2 = new Kvader(5);
    Console.WriteLine(K2.ToString());

    //Uporabimo tretji konstruktor: robove nastavimo na 2, 4 in 6
    Kvader K3 = new Kvader(2, 4, 6);
    Console.WriteLine(K3.ToString());

    Kvader K4 = new Kvader(K3);//ustvarimo kopijo kvadra K3
    Console.WriteLine(K4.ToString());
    //Klic statične metode PovejKolikoNasJe, ki pove, koliko kvadrov smo
    //doslej že ustvarili
    Console.WriteLine("Skupno število živečih kvadrov: " +
                      Kvader.PovejKolikoNasJe());

    Kvader K5 = new Kvader();
    //velikosti robov petega kvadra nastavimo z našimi Set metodami
    K5.StranicaA(6);
    K5.StranicaB(8);
    K5.StranicaC(10);
    Console.WriteLine(K5.ToString());

    //Skušajmo ustvariti kvader z nemogočimi robovi
    //metoda Kontrola bo rob a in rob b spremenila na 1
    Kvader K6 = new Kvader(-2, 48888, 6);
    Console.WriteLine(K6.ToString());
    Console.WriteLine("Skupno število živečih kvadrov: " +
```

```
                        Kvader.PovejKolikoNasJe());
    Console.ReadKey();
}
```

Dedovanje (Inheritance) – izpeljani razredi

Kaj je to dedovanje

Dedovanje (Inheritance) je ključni koncept objektno orientiranega programiranja. Smisel in pomen dedovanja je v tem, da iz že zgrajenih razredov skušamo zgraditi bolj kompleksne, ki bodo znali narediti nekaj uporabnega. Dedovanje je torej orodje, s katerim se izognemo ponavljanju pri definiranju različnih razredov, ki pa imajo več ali manj značilnosti skupnih. Opredeljuje torej odnos med posameznimi razredi.

Vzemimo pojem sesalec iz biologije. Kot primer za sesalce vzemimo npr. konje in kite. Tako konji kot kiti počnejo vse kar počnejo sesalci nasploh (dihajo zrak, skotijo žive mladiče, …), a prav tako pa imajo nekatere svoje značilnosti (konji imajo npr. štiri noge, ki jih kiti nimajo, imajo kopita, …, obratno pa imajo npr. kiti plavuti, ki pa jih konji nimajo, ..). V Microsoft C# bi lahko za ta primer modelirali dva razreda: prvega bi poimenovali Sesalec, in drugega Konj, in obenem deklarirali, da je Konj podeduje (inherits) Sesalca. Na ta način bi med sesalci in konjem vzpostavili povezavo v tem smislu, da so vsi konji sesalci (obratno pa seveda ne velja!). Podobno lahko deklariramo razred z imenom Kit, ki je prav tako podedovan iz razreda Sesalec. Lastnosti, kot so npr, kopita ali pa plavuti pa lahko dodatno postavimo v razred Konj oz. razred Kit.

Bazični razredi in izpeljani razredi

Sintaksa, ki jo uporabimo za deklaracijo, s katero želimo povedati, da razred podeduje nek drug razred, je takale:

```
class IzpeljaniRazred : BazičniRazred
{
    . . .
}
```

Izpeljani razred deduje od bazičnega razreda. Za razliko od C++, lahko razred v C# deduje največ en razred in ni možno dedovanje dveh ali več razredov. Seveda pa je lahko razred, ki podeduje nek bazični razred, zopet podedovan v še bolj kompleksen razred.

Primer:

Radi bi napisali razred, s katerim bi lahko predstavili točko v dvodimenzionalnem koordinatnem sistemu. Razred poimenujmo Tocka:

```
class Tocka //bazični razred
{
    public Tocka(int x, int y)  //konstruktor
    {
          //telo konstruktorja
    }
    //telo razreda Tocka
}
```

Sedaj lahko definiramo razred za tridimenzionalno točko z imenom Tocka3D, s katerim bomo lahko delali objekte, ki bodo predstavljali točke v tridimenzionalnem koordinatnem sistemu in po potrebi dodamo še dodatne metode:

```
class Tocka3D : Tocka  //razred Tocka3D podeduje razred Tocka
{
   //telo razreda Tocka3D - tukaj zapišemo še dodatne metode tega razreda!
}
```

Klic konstruktorja bazičnega razreda

Vsi razredi imajo vsaj en konstruktor (če ga ne napišemo sami, nam prevajalnik zgenerira privzeti konstruktor). Izpeljani razred avtomatično vsebuje vsa polja bazičnega razreda, a ta polja je potrebno ob kreiranju novega objekta inicializirati. Zaradi tega mora konstruktor v izpeljanem razredu poklicati konstruktor svojega bazičnega razreda. V ta namen se uporablja rezervirana besedica base:

```
class Tocka3D : Tocka  ////razred Tocka3D podeduje razred Tocka
{
    public Toca3D(int z)
           :base(x,y)  //klic bazičnega konstruktorja Tocka(x,y)
    {
       //telo konstruktorja Tocka3D
    }
    //telo razreda Tocka3D
}
```

Če bazičnega konstruktorja v izpeljanem razredu ne kličemo eksplicitno (če vrstice :base(x,y) ne napišemo!), bo prevajalnik avtomatično zgeneriral privzeti konstruktor. Ker pa vsi razredi nimajo privzetega konstruktorja (v veliko primerih napišemo lastni konstruktor), moramo v konstruktorju znotraj izpeljanega razreda obvezno najprej klicati bazični konstruktor (rezervirana besedica base). Če klic izpustimo (ali ga pozabimo napisati) bo rezultat prevajanja compile-time error:

```
class Tocka3D : Tocka  //razred Tocka3D podeduje razred Tocka
{
    public Tocka3D(int z)
    //NAPAKA - POZABILI smo klicati bazični konstruktor razreda Tocka
    {
       //telo konstruktorja Tocka3D
    }
    //telo razreda Tocka3D
}
```

Določanje oz. prirejanje razredov

Poglejmo še, kako lahko kreiramo objekte iz izpeljanih razredov. Kot primer vzemimo zgornji razred *Tocka* in iz njega izpeljani razred *Tocka3D*.

```
class Tocka //bazični razred
{
   //telo razreda Tocka
}
```

```
class Tocka3D: Tocka //razred Tocka3D podeduje razred Tocka
{
    //telo razreda Tocka3D
}
```

Iz razreda *Tocka* izpeljimo še dodatni razred *Daljica*

```
class Daljica:Tocka  //razred Daljica podeduje razred Tocka
{
    //telo razreda Daljica
}
```

Kreirajmo sedaj nekaj objektov:

```
Tocka A = new Tocka();
Tocka3D B = A;  //NAPAKA - različni tipi


Tocka3D C = new Tocka3D ();
Tocka D = C; //OK!
```

Nove metode

Razredi lahko vsebujejo več ali manj metod in slej ko prej se lahko zgodi, da se pri dedovanju v izpeljanih razredih ime metode ponovi – v izpeljanem razredu torej napišemo metodo, katere ime, število in tipi parametrov se ujemajo z metodo bazičnega razreda. Pri prevajanju bomo zato o tem dobili ustrezno opozorilo - warning. Metoda v izpeljanem razredu namreč v tem primeru prekrije metodo bazičnega razreda. Če npr. napišemo razred *Tocka* in nato iz njega izpeljemo razred *Tocka3D*,

```
class Tocka   //bazni razred
{
     private int x, y;  //polji razreda Tocka

     public  void Izpis()  //metoda za izpis koordinat razreda Tocka
     {
          Console.WriteLine("Koordinate točke:\nx = " + x + "\ny = " + y);
     }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
     private int z;      //dodatno polje razreda Tocka3D

     //Metoda Izpis prepiše istoimensko metodo razreda Tocka
     public void Izpis()
     {
         Console.Write("z = " + z + "\n");
     }
}
```

nam bo prevajalnik zgeneriral opozorilo, s katerim nas obvesti, da metoda *Tocka3D.Izpis* prekrije metodo *Tocka.Izpis*:

'ConsoleApplication1.Program.Tocka3D.Izpis()' hides inherited member 'ConsoleApplication1.Program.Tocka.Izpis()'. Use the new keyword if hiding was intended. Program.cs

Program se bo sicer prevedel in tudi zagnal, a opozorilo moramo vzeti resno. Če namreč napišemo razred, ki bo podedoval razred *Tocka3D*, bo uporabnik morda pričakoval, da se bo pri klicu metode *Izpis* pognala metoda bazičnega razreda, a v našem primeru se bo zagnala metoda razreda *Tocka3D*. Problem seveda lahko rešimo tako, da metodo *Izpis* v izpeljanem razredu preimenujemo (npr. Izpis1), še boljša rešitev pa je ta, da v izpeljanem razredu eksplicitno povemo, da gre za NOVO metodo – to storimo z uporabo operatorja *new*.

```
class Tocka   //bazni razred
```

```
{
     private int x, y;  //polji razreda Tocka

     public  void Izpis()  //metoda za izpis koordinat razreda Tocka
     {
          Console.WriteLine("Koordinate točke:\nx = " + x + "\ny = " + y);
     }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
     private int z;      //dodatno polje razreda Tocka3D
     //Z operatorjem new napovemo, da ima razred Tocka3D SVOJO LASTNO
     //metodo Izpis
     new public void Izpis()
     {
         Console.Write("z = " + z + "\n");
     }
}
```

Zgled:

Napišimo razred *Krog* z zasebnim poljem *polmer* in lastnostmi *Premer*, *Obseg* in *Kvadratura*. Iz razreda nato izpeljimo razred *Krogla*, dodaj razredu lastno metodo *Kvadratura* in novo metodo *Volumen*.

```
class Krog  //Bazični razred
 {
     private double polmer;//zasebno polje razreda Krog

     public double Polmer  //lastnost za dostop do polja polmer
     {
        get
        {  if (polmer < 0)
             return 0.00;
           else
             return polmer;
        }
        set { polmer = value;}
        }
        public double Premer  //lastnost, ki vrne premer kroga
        {
            get {  return Polmer * 2;  }
        }
        public double Obseg  //lastnost, ki vrne obseg kroga
        {
            get  {  return Premer * Math.PI;  }
        }
        public double Kvadratura  //lastnost, ki vrne ploščino kroga
        {
            get  {  return Math.PI*Math.Pow(Polmer,2); }
        }
}
/*Kreirajmo sedaj razred Krogla ki naj podeduje razred Krog. Razred Krogla
bo podedoval polje polmer, podedoval bo lastnosti Premer in Obseg, imel bo
SVOJO lastnost za izračun kvadrature, poleg tega pa še novo lastnost za
izračun prostornine krogle */
class Krogla : Krog
{
     new public double Kvadratura  //z  operatorjem new smo označili, da
```

```
                        //ima razred krog SVOJO lastno lastnost kvadratura
    {
        get { return 4 * Math.PI*Math.Pow(Polmer,2); }
    }

    public double Volumen  //nova lastnost razreda Krogla
    {
        get {  return 4 * Math.PI * Math.Pow(Polmer,2) / 3; }
    }
}

//glavni program
static void Main()
{
     Krog c = new Krog();//nov objekt razreda Krog
     c.Polmer = 25.55;
     Console.WriteLine("Karakteristike kroga");
     Console.WriteLine("Polmer  : " + c.Polmer);
     Console.WriteLine("Premer  : " + c.Premer);
     Console.WriteLine("Obseg   : " + c.Obseg);
     Console.WriteLine("Ploščina: " + c.Kvadratura);

     Krogla s = new Krogla();//nov objekt razreda Krogla
     s.Polmer = 25.55;

     Console.WriteLine("\nKarakteristike krogle");
     Console.WriteLine("Polmer     : " + s.Polmer);
     Console.WriteLine("Premer     : " + s.Premer);
     Console.WriteLine("Obseg      : " + s.Obseg);
     Console.WriteLine("Površina   : " + s.Kvadratura);
     Console.WriteLine("Prostornina: " + s.Volumen);
}
```

Virtualne metode

Pogosto želimo metodo, ki smo je napisali v bazičnem razredu, v višjih (izpeljanih) razredih skriti in napisati novo metodo, ki pa bo imela enako ime in enake parametre. Eden izmed načinov je uporaba operatorja new za tako metodo, drug način pa je z uporabo rezervirane besede virtual. Metodo, za katero želimo že v bazičnem razredu označiti, da jo bomo lahko v nadrejenih razredih nadomestili z novo metodo (jo prekriti), označimo kot ***virtualno*** (*virtual*), npr.:

```
//virtualna metoda v bazičnem razredu - v izpeljanih razredih bo lahko
//prekrita(override)
public virtual void Koordinate()
{...}
```

V nadrejenem razredu moramo v takem primeru pri metodi z enakim imenom uporabiti rezervirano besedico override, s katero povemo, da bo ta metoda prekrila/prepisala bazično metodo z enakim imenom in enakimi parametri.

```
//izpeljani razred - besedica override pomeni, da smo s to metodo prekrili
//bazično metodo
public override void Koordinate()
{ ...}
```

Ločiti pa moramo razliko med tem, ali neka metoda prepiše bazično metodo (override) ali pa jo skrije. Prepisovanje metode (overriding) je mehanizem, kako izvesti drugo, novo implementacijo iste metode –

virtualne in override metode so si v tem primeru sorodne, saj se pričakuje, da bodo opravljale enako nalogo, a nad različnimi objekti (izpeljanimi iz bazičnih razredov, ali pa iz podedovanih razredov). Skrivanje metode (hiding) pa pomeni, da želimo neko metodo nadomestiti z drugo – metode v tem primeru niso povezane in lahko opravljajo povsem različne naloge.

Primer:

Naslednji primer prikazuje zapis virtualne metode *Koordinate*() v bazičnem razredu in *override* metode *Koordinate*() v izpeljanem razredu.

```
class Tocka   //bazični razred
{
    private int x, y;   //polji razreda Tocka

    //metoda Koordinate je virtualna, kar pomeni, da jo lahko prepišemo
    // (override)

    //metoda za izpis koordinat razreda Tocka
    public virtual void Koordinate()
    {
        Console.WriteLine( "Koordinate točke:\nx = "+x + "\ny = " + y);
    }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
    private int z;      //dodatno polje razreda Tocka3D

    //Metoda Koordinate je označena kot override - prepiše istoimensko
    //metodo razreda Tocka
    public override void Koordinate()
    {
       base.Koordinate();  //klic bazične metode Koordinate razreda Tocka
       Console.Write("z = "+z+"\n");
    }
}
```

Prekrivne (Override) metode

Kadar je bazičnem razredu neka metoda označena kot virtualna (*virtual*), jo torej lahko v nadrejenih razredih prekrijemo/povozimo (override). Pri deklaraciji takih metod (pravimo jim **polimorfne** metode) z uporabo rezerviranih besed *virtual* in *override*, pa se moramo držati nekaterih pomembnih pravil:

- Metoda tipa *virtual* oz. *override* NE more biti zasebna (ne more biti *private*);
- Obe metodi, tako *virtualna* kot *override* morata biti identični: imeti morata enako ime, enako število in tip parametrov in enak tip vrednosti, ki jo vračata;
- Obe metodi morata imeti enak dostop. Če je npr. virtualna metoda označena kot javna (*public*), mora biti javna tudi metoda *override*;
- Prepišemo (prekrijemo/povozimo) lahko le virtualno metodo. Če metoda ni označena kot virtualna in bomo v nadrejenem razredu skušali narediti *override*, bomo dobili obvestilo o napaki;
- Če v nadrejenem razredu ne bomo uporabili besedice *override*, bazična metoda ne bo prekrita. To pa hkrati pomeni, da se bo tudi v izpeljanem razredu izvajala metoda bazičnega razreda in ne tista, ki smo napisali v izpeljanem razredu;
- Če neko metodo označimo kot *override*, jo lahko v nadrejenih razredih ponovno prekrijemo z novo metodo.

Zgled: razred Tocka

Razred *Tocka* in razred *Tocka3D*, ki je izpeljan iz razreda *Tocka* sta implementirana v naslednjem primeru:

```
class Tocka   //bazni razred
{
    private int x, y;         //polji razreda Tocka

    public Tocka(int x,int y) //konstruktor
    {
        this.x = x;
        this.y = y;
    }
    //metoda Koordinate je virtualna, kar pomeni, da jo lahko preobtežimo
    //(naredimo override)
    public virtual void Koordinate()//metoda za izpis koord. razreda Tocka
    {
        Console.WriteLine( "Koordinate točke:\nx = "+x + "\ny = " + y);
    }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
    private int z;      //dodatno polje razreda Tocka3D
    //konstruktor - parametra x in y potrebujemo za dedovanje bazičnega
    //konstruktorja razreda Tocka
    public Tocka3D(int x,int y,int z)
         : base(x,y)
    {
       this.z = z;
    }
    //Metoda Koordinate prepiše istoimensko metodo razreda Tocka
    public override void Koordinate()
    {
       base.Koordinate();  //klic bazične metode Koordinate razreda Tocka
       Console.Write("z = "+z+"\n");
    }
}

static void Main(string[] args)
{
    Tocka A = new Tocka(1,1);   //Nov objekt razreda Tocka
    A.Koordinate();             //klic metode Koordinate razreda Tocka
    Tocka3D A3D = new Tocka3D(1, 2, 3); //Nov objekt razerda Tocka3D
    A3D.Koordinate();  //klic preobložene metode Koordinate razreda Tocka3D
}
```

Zgled: razred Kolobar deduje razred Krog

Napišimo razred *Krog* z zasebnim poljem polmer in virtualno metodo *ploscina*, nato pa še razred *Kolobar*. Razred *Kolobar* naj deduje razred K*rog*, dodano naj ima še eno zasebno polje *notranjiPolmer* in svojo lastno(override) metodo *ploščina*

```
class Krog   //bazni razred
{
    private int polmer;   //polje razreda Krog

    //metoda ploscina je virtualna, kar pomeni, da jo lahko preobtežimo
```

```
    public virtual double ploscina()
    {
        return Math.PI * polmer * polmer;
    }
   //lastnost razreda Krog, za dostop in inicializacijo polja polmer
    public int Polmer
    {
        get
        {
           return polmer;
        }
        set
        {
           polmer = value;
        }
    }
}

class Kolobar : Krog   //razred Kolobar podeduje razred Krog
{
    //ker Kolobar deduje Krog, že pozna polje polmer
    private int notranjiPolmer; //dodatno polje razreda Kolobar

    //metoda ploscina prepiše istoimensko metodo bazičnega razreda Krog
    public override double ploscina()
    {
        return Math.PI * (Polmer * Polmer - notranjiPolmer*notranjiPolmer);
    }

    //ker Kolobar deduje Krog, že pozna njegovo lastnost Polmer
    //dodatna lastnost (property) razreda Kolobar, za dostop in inic. Polja
    // notranjiPolmer
    public int NotranjiPolmer
    {
        get
        {
            return notranjiPolmer;
        }
        set
        {
            notranjiPolmer = value;
        }
    }
}

static void Main(string[] args)
{
    Random naklj = new Random();
    Krog k=new Krog();              //nov objekt razreda Krog
    k.Polmer = naklj.Next(1, 10);   //polje polmer inicializiramo preko
                                    //lastnosti Polmer

    Kolobar kol = new Kolobar();    //nov objekt razreda Kolobar
    //tudi polje polmer objekta kol inicializiramo preko lastnosti Polmer
    kol.Polmer = naklj.Next(1, 10);
    //polje notranjiPolmer inicializiramo preko lastnosti notranjiPolmer
    kol.NotranjiPolmer = naklj.Next(1, 10);

    //izpis ploščine kroga na 4 decimalke
    Console.WriteLine("Ploščina kroga: {0:F4}",k.ploscina());
```

```
    //izpis ploščine kolobarja na 4 decimalke
    Console.WriteLine("Ploščina kolobarja: {0:F4}", kol.ploscina());
}
```

Zgled: razred Kmetovalec deduje razred Oseba

Deklarirajmo razred *Oseba*, nato pa razred *Kmetovalec*, ki naj podeduje razred *Oseba*. Razred *Oseba* naj ima polje *ime*, razred *Kmetovalec* pa še dodatno polje *velikostPosesti*. Za oba razreda napišimo tudi konstruktor in virtualno metodo *ToString* za izpis podatkov o posameznem objektu!

```
public class Oseba             //bazični razred
{
    public string ime;         //bazično polje

    public Oseba(string ime)   //bazični konstruktor
    {
       this.ime = ime;
    }

    public virtual string ToString()  //virtualna metoda bazičnega polja
    {
        return "Ime=" + ime;
    }
}

public class Kmetovalec : Oseba  //razred Kmetovalec deduje razred Oseba
{
    int velikostPosesti;  //dodatno polje razreda Kmetovalec

    //konstruktor razreda Kmetovalec podeduje bazično polje ime
    public Kmetovalec(string ime, int velikostPosesti)
        : base(ime)
    {
        this.velikostPosesti = velikostPosesti;
    }
    //metoda ToString prekrije bazično metodo ToString
    public override string ToString()
    {
        return base.ToString() + "; Kvadratnih metorv = " +
        velikostPosesti.ToString();
    }
}

//glavni program
static void Main(string[] args)
{
    ArrayList osebe = new ArrayList();  //zbirka oseb

    Oseba mike = new Oseba("mike");
    osebe.Add(mike);
    Console.WriteLine("Osebe:");
    //izpišemo seznam vseh oseb
    foreach (Oseba p in osebe)
       Console.WriteLine(p.ToString());

    ArrayList kmetovalci = new ArrayList();  //zbirka kmetovalcev
    Kmetovalec john = new Kmetovalec("john", 10);
```

```
    Kmetovalec bob = new Kmetovalec("bob", 20);
    kmetovalci.Add(john);
    kmetovalci.Add(bob);
    Console.WriteLine();
    Console.WriteLine("Kmetovalci:");
    //izpišemo seznam vseh kmetovalcev
    foreach (Kmetovalec f in kmetovalci)
       Console.WriteLine(f.ToString());

    ArrayList vsiSkupaj = new ArrayList();  //zbirka vseh skupaj
    //napolnimo skupno zbirko vseh oseb
    foreach (Oseba o in osebe)
      vsiSkupaj.Add(o);
    foreach (Kmetovalec f in kmetovalci)
      vsiSkupaj.Add(f);
    Console.WriteLine();
    Console.WriteLine("Vsi skupaj:");
    //izpišemo seznam vseh oseb
    foreach (Oseba p in vsiSkupaj)
       Console.WriteLine(p.ToString());
}
```

Polimorfizem - mnogoličnost

V izpeljanih razredih se srečamo še z enim temeljnim pojmom objektno orientiranega programiranja – to je pojem **polimorfizem** oz. **mnogoličnost**. Pojem **polimorfizem** označuje princip, da lahko različni objekti razumejo isto sporočilo in se nanj odzovejo vsak na svoj način. Pomeni tudi, da je ista operacija lahko implementirana na več različnih načinov oz. zanjo obstaja več metod. Dedovanje in polimorfizem sta lastnosti, ki predstavljata osnovo objektnega programiranja in omogočata hitrejši razvoj in lažje vzdrževanje programske kode.

Kot primer za prikaz polimorfizma deklarirajmo razred *OsnovniRazred*, ki ima eno samo metodo z imenom *Slika*. Ker želimo, da bo vsak objekt, ki bo izpeljan iz tega razreda (tudi tisti v podedovanih – izpeljanih razredih) ohranil sebi lastno obnašanje ob klicu metode *Slika*, moramo uporabiti **polimorfno redefinicijo**. Polimorfno redefinicijo omogočimo z že znano definicijo *virtualne metode*. V telesu te metode bomo za vajo in zaradi enostavnosti prikazali le neko sporočilno okno!

```
public class OsnovniRazred   //temeljni razred
{
   //polimorfna redefinicija – omogočimo jo z virtualno metodo
   public virtual void Slika()
   {
        Console.WriteLine("Osnovni objekt!");
   }
}
```

Ker smo metodo *Slika* definirali kot *virtualno*, smo s tem napovedali, da bodo izpeljani objekti lahko uporabljali svojo (prekrivno oz. override) metodo *Slika*, ki pa bo imela enako ime.

Razred *OsnovniRazred* bo naš temeljni razred, iz katerega bomo tvorili izpeljane razrede in v njih tvorili nove objekte. Ker je metoda *Slika* virtualna to pomeni, da lahko v izpeljanih razredih to metodo prekrijemo (*override*) z metodo, ki bo imela enako ime a drugačen pomen (drugačno vsebino).

Napišimo sedaj še tri razrede, ki naj bodo izpeljani iz razreda *OsnovniRazred* in ki imajo svojo metodo *Slika*. Pred tipom takih metod mora stati besedice override, ki označuje, da se bodo objekti izpeljani iz teh razredov na to metodo odzivali vsak na svoj način. Osnovni pogoj pa je, da imajo take override metode enako raven

zaščite (npr. vse so public) , enako ime in enake parametre kot jih ima osnovna virtualna metoda v bazičnem razredu.

```
//Crta je razred, ki podeduje razred OsnovniRazred
public class Crta : OsnovniRazred
{
   public override void Slika()    //preobložena metoda razreda Crta
   {
      //Telo override metode je seveda lahko drugačno!!!
      Console.WriteLine("Črta.");
   }
}

//Tudi Krog je razred, ki podeduje razred OsnovniRazred
public class Krog : OsnovniRazred
{
   public override void Slika()
   {
      //Telo override metode je seveda lahko drugačno!!!
      Console.WriteLine ("Krog.");
   }
}

//Tudi Kvadrat je razred, ki podeduje razred OsnovniRazred
public class Kvadrat: OsnovniRazred
{
   //Telo override metode je seveda spet lahko drugačno!!!
   public override void Slika()
   {
       Console.WriteLine ("Kvadrat.");
   }
}
```

Poglejmo sedaj, kako bi te štiri razrede sedaj uporabili in na primeru izpeljanih objektov prikazali princip polimorfizma. V ta namen kreirajmo tabelo objektov. Ime tabele je *dObj*, tabela pa naj bo inicializirana tako, da so v njej lahko štiri objekti tipa *OsnovniRazred* .

```
OsnovniRazred[] dObj = new OsnovniRazred[4];//tabela objektov
```

Ker so razredi *Crta*, *Krog* in *Kvadrat* izpeljani iz bazičnega razreda *OsnovniRazred*, jih lahko priredimo isti tabeli *dObj*. Če te zmožnosti ne bi bilo, bi morali za vsak nov objekt, izpeljan iz kateregakoli od teh štirih razredov, kreirati svojo tabelo. Dedovanje pa nam omogoča, da se vsak od izpeljanih objektov obnaša tako kot njegov bazični razred.

```
dObj[0] = new Crta();          //konstruktor objekta dObj[0]
dObj[1] = new Krog();          //konstruktor objekta dObj[1]
dObj[2] = new Kvadrat();       //konstruktor objekta dObj[2]
dObj[3] = new OsnovniRazred();//konstruktor objekta dObj[3]

foreach (OsnovniRazred objektZaRisanje in dObj)
{
     objektZaRisanje.Slika();  //klic metode Slika ustreznega objekta
}
```

Ko je tabela inicializirana, lahko npr. s *foreach* zanko pregledamo vsakega od objektov v tabeli. Zaradi načela polimorfizma se v zagnanem programu vsak objekt obnaša po svoje, pač odvisno od tega, iz katerega razreda je bil izpeljan. Ker smo v izpeljanih razredih prepisali virtualno metodo *Slika* , se ta metoda v izpeljanih objektih

izvaja različno, pač glede na njeno definicijo v izpeljanih razredih. Pri vsakem prehodu zanke bomo tako dobili drugačno sporočilno okno.

Uporaba označevalca protected

Uporaba označevalcev **private** in **public** predstavlja obe skrajni možnosti dostopa do članov razreda (oz. objekta). Javna polja in metode so dostopne vsem, zasebna polja in metode pa so dostopne le znotraj razreda. Pogosto pa je koristno, da bazični razred dovoljuje izpeljanim razredom, da le-ti dostopajo do nekaterih bazičnih članov, obenem pa ne dovoljujejo dostopa razredom, ki niso del iste hierarhije. V takem primeru uporabimo za dostop označevalec ***protected***.

Nadrejeni razred torej lahko dostopa do člana bazičnega razreda, ki je označen kot *protected*, kar praktično pomeni, da bazični član, ki je označen kot *protected*, v izpeljanem razredu postane javen (*public*). Javen je tudi v vseh višjih razredih.

V primeru, ko pa nek razred ni izpeljani razred, pa nima dostopa do članov razreda, ki so označeni kot *protected* – znotraj razreda, ki ni izpeljani razred, je torej član razreda, ki je označen kot *protected* enak zasebenemu članu (*private*).

Vaje

- Dan je razred *Zival*, ki vsebuje tri zasebna polja in metode za nastavljanje in spreminjanje le-teh. Vsebuje tudi konstruktor brez parametrov, ki postavi spremenljivke na smiselno začetno vrednost.

```
public class Zival
{
    //objektne spremenljivke
    private int steviloNog;
    private string vrsta;
    private string ime;

    /*Konstruktor - ustvari objekt Zival s stirimi nogami vrste pes
        in imenom Pika.*/
    public Zival()
    {
        steviloNog = 4;
        vrsta = "pes";
        ime = "Pika";
    }

    //Metoda, ki vrne ime živali.
    public string VrniIme()
    {
        return this.ime;
    }

    // Metoda, ki vrne vrsto živali.
    public string VrniVrsto()
    {
        return this.vrsta;
    }
    /* Metoda, ki vrne število nog živali.
```

```
       Zagotovljeno je, da bo število nog vedno vecje od 0. */
    public int VrniSteviloNog()
    {
        return this.steviloNog;
    }
    /*Metoda, ki nastavi ime živali.*/
    public void NastaviIme(string vrednost)
    {
        if (vrednost != null)
        {
            this.ime = vrednost;
        }
    }
    /* Metoda, ki nastavi vrsto živali.*/
    public void NastaviVrsto(string vrednost)
    {
        if (vrednost != null)
        {
            this.vrsta = vrednost;
        }
    }
    /* Metoda, ki nastavi število nog živali.*/
    public void NastaviSteviloNog(int vrednost)
    {
        if (vrednost > 0)
        {
            this.steviloNog = vrednost;
        }
    }
}
```

Dopolnite razred z metodo *override public String ToString()*, ki naj izpiše podatke o objektih v obliki: *#Ime*: je vrste *#vrsta* in ima *#steviloNog* nog. Pri tem seveda zamenjajte *#vrsta* in *#steviloNog* z vrednostmi v istoimenskih spremenljivkah.

Nato napišite testni program, kjer ustvarite 5 živali in jim nastavite smiselno vrsto in imena ter število nog. Izpišite jih!

- Sestavi razred *Kolega*, ki ima tri komponente: ime, priimek in telefonska številka. Vse tri komponente naj bodo tipa string in javno dostopne. Napiši vsaj dva konstruktorja: prazen konstruktor, ki vse tri komponente nastavi na "NI PODATKOV" in konstruktor, ki sprejme vse tri podatke in ustrezno nastavi komponente. Napiši tudi metodo *public string ToString()*, ki vrne niz s smiselnim izpisom podatkov o objektu tipa Kolega (ime, priimek in telefonska številka). Sestavi testni program, v katerem ustvariš dva objekta tipa *Kolega*. En objekt ustvari s praznim konstruktorjem, drugega s konstruktorjem, ki sprejme tri parametre. Oba objekta tudi izpiši na zaslon. Napiši program, ki sestavi tabelo 10ih objektov tipa *Kolega*, prebere telefonsko številko in izpiše tiste objekte iz tabele, ki imajo tako telefonsko številko. Če takega objekta v tabeli ni, naj se izpiše: "Noben moj kolega nima take telefonske številke."

- V spodnji kodi je napaka. Poišči jo in jo odpravi (brez uporabe računalnika).

```
public class X
{
  public int a = 0;
  public int b = 2;
```

```
  override public ToString()
  {
     return "a = " + a + " b = " + b;
  }
}
```

Dana je koda:

```
public class Y
{
    private int x;
    private int y;

    public Y(int row, int col)
    {
        x = row;
        y = col;
    }
}
```

Kateri izmed spodnjih konstruktorjev je pravilen:

a) `y1 = new Y(2,3);`
b) `y2 = new Y(2);`
c) `y3 = new Y;`
d) `y4 = new Y();`
e) `y5 = new Y(10.0, 3);`
f) `y6 = new Y[5];`

Razredu Y dodamo še konstruktor

```
public Y(int row)
{
  x = row;
  y = 80;
}
```

Kateri izmed konstruktorjev pa so sedaj pravilni?

a) `y1 = new Y(2,3);`
b) `y2 = new Y(2);`
c) `y3 = new Y;`
d) `y4 = new Y();`
e) `y5 = new Y(10.0, 3);`
f) `y6 = new Y[5];`

Dana je koda:

```
public class Z
{
      private int a = 0;
      private int b =0;

      public Z(int aa, int bb)
      {
            a = aa;
            b = bb;
      }
}
```

Ustvarimo dva objekta tipa `Z`, in sicer `z1 = new Z(5, 4)` ter `z2 = new Z(5, 4)`.

- Kakšno vrednost nam vrne test z1 == z2 ? Trditev obrazloži.
- Kako »pravilno« preveriti, če sta dva objekta »enaka« ? Namig: metoda *equals*.
- Ali lahko metodo »*equals*« takoj uporabiš v zgornji kodi, ali moraš razred kaj spremeniti?

Napiši razred **Kosarka**, za spremljanje košarkaške tekme. Voditi moraš število prekrškov za vsakega tekmovalca (10 igralcev), število doseženih točk (posebej 1 točka, 2 točki in 3 točke), ter metodo za izpis statistike tekme. Doseganje košev in prekrškov realiziraj preko metode *zadelProstiMet*(), *zadelZa2Tocki*, *zadelZa3Tocke* in *prekrsek*.

Sestavi razred, ki bo v svoja polja lahko shranil ulico, številko nepremičnine ter vrsto nepremičnine. Ustvari poljuben objekt, ga inicializiraj in ustvari izpis, ki naj zgleda približno takole: Na naslovu Cankarjeva ulica 32, Kranj je blok.

Dana je koda razreda *Povecaj*:

```
public class PovecajR
{
	public void Povecaj(int a)
	{
		a++;
	}
	public void Povecaj(int[] a)
	{
		for(int i =0; i < a.Length; i++)
		a[i] = a[i] + 2;
	}
}
```

Ali je v kodi kakšna napaka? Če je, jo odpravi! Kaj izpiše sledeča koda?

```
PovecajR inc = new PovecajR();

int a = 5;
int[] b = {5};

inc.Povecaj (a);
inc.Povecaj (b);
Console.WriteLine("a = " + a);
Console.WriteLine("b = " + b[0]);
```

Sestavi razred *Datum*, ki predstavlja datum (dan, mesec in leto). Definiraj ustrezne komponente in konstruktorje. Definiraj metodo *override public string ToString()*, ki vrne datum predstavljen z nizom znakov.

Definiraj še metodo *public bool Equals(Datum d)*, ki vrne *True*, če je datum *this* enak datumu *d*. Razredu *Datum* dodaj metodo *public int CompareTo(Datum d)*, ki primerja *this* in *d* ter vrne celo število:

- < 0, če je *this* pred *d*
- 0, če je *this* enak *d*
- > 0, če je this za *d*

Sestavite razred Majica, ki hrani osnovne podatke o majici: velikost (število med 1 in 5), barvo (poljuben niz) in ali ima majica kratke ali dolge rokave. Vse spremenljivke razreda morajo biti privatne, razred pa mora tudi poznati metode za nastavljanje in branje teh vrednosti. Podpisi teh metod naj bodo:

- public int *VrniVelikost()* ter public void *SpremeniVelikost(int velikost)*
- public string *VrniBarvo()* ter public void *SpremeniBarvo(string barva)*
- public boolean *ImaKratkeRokave()* ter *public void NastaviKratkeRokave(boolean imaKratkeRokave)*

Metoda za nastavljanje velikosti majice naj poskrbi tudi za to, da bo velikost vedno ostala znotraj sprejemljivih meja! Če uporabnik poskusi določiti napačno velikost, naj se obdrži prejšnja vrednost.

Kandidat za direktorja banke: v banki je direktor uspel povzročiti politični škandal takšnih razsežnosti, da je moral odstopiti (seveda z odpravnino v višini 50 povprečnih plač) in zato sedaj iščejo novega direktorja. Za to službo se je prijavilo kar nekaj kandidatov, od katerih je vsak predložil sledeče podatke: ime, priimek, starost (v letih) in stopnjo izobrazbe - številka med 3 in 8 (vključno). Banka želi izbrati direktorja, ki ne bi bil starejši od 50 let (da bo lepo izgledal v medijih) in ima vsaj 6. stopnjo izobrazbe. Tvoja naloga je:sestaviti razred, ki bo hranil podatke o kandidatih za direktorja in dopolniti priložen testni program tako, da bo našel primerne kandidate za to mesto in jih izpisal.

Razred Kandidat mora poleg ustreznih spremenljivk (ki naj bodo privatne!) vsebovati vsaj javne metode:

- *public Kandidat ()* - konstruktor brez parametrov - ustvari kandidata s starostjo 20, stopnjo izobrazbe 6 in imenom in priimkom Neznani Neznanec.
- *public void NastaviImePriimek(string ime, string priimek)* - metoda, ki nastavi ime in priimek kandidata. Če je slučajno ime oz. priimek enak null, naj starega imena in priimka ne spremeni.
- *public string[] VrniImePriimek()* - metoda, ki vrne ime in priimek kandidata. Metoda naj vrne seznam nizov dolžine 2 - v prvem polju naj bo zapisano ime, v drugem pa priimek kandidata.
- *public void NastaviStarost(int starost)* - metoda, ki nastavi starost kandidata. Starost mora biti v mejah med 20 in 80 let - če ni, naj se stara starost ne spremeni.
- *public int VrniStarost()* - metoda, ki vrne starost kandidata.
- *public void NastaviIzobrazbo(int izobrazba)* - metoda, ki nastavi izobrazbo kandidata. Izobrazba mora biti v mejah med 3 in 8 - če ni, naj se stara izobrazba ne spremeni.
- *public int VrniIzobrazbo()* - metoda, ki vrne izobrazbo kandidata.
- public string ToString()

Dano je okostje programa, ki obravnava prijavljene kandidate.

```
public static Kandidat NajPrimernejši(Kandidat[] tabelaKandid)
{
      // tu poskrbi, da bodo v ime, priimek, ...
    // prišli podatki o najprimern. kandidatu
      string ime = "Moj";
      string priimek = "Prijatelj";
      int starost = 42;
      int izobrazba = 6;
      Kandidat nov = new Kandidat();
      nov.NastaviImePriimek(ime, priimek);
      nov.NastaviIzobrazbo(izobrazba);
      nov.NastaviStarost(starost);
      return nov;
}
public static void Main(string[] args)
{
    public string[] imena = {"Jana", "Jure", "Bernarda",
            "Anže", "Danijel", "Primož", "Anita", "Tina", "Matija"};
    public string[] priimki = {"Mali", "Ankimer", "Zupan",
            "Zevnik", "Bogataj", "Koci", "Vilis", "Jerše", "Lokar"};
    int kolikoKandidatov = 30;
    public Kandidat[] kandidati = new Kandidati[kolikoKandidatov];
    Random gen = new Random();

    for(int i=0; i < kolikoKandidatov; i++)
```

```
    {
            Kandidat tmp = new Kandidat();
            int randime = gen.Next(0, imena.Length - 1);
            int randpriimek = gen.Next(0,priimki.Length - 1);
            int randizobrazba = gen.Next(3,8);
            int randstarost = gen.Next(20,90);
            tmp.NastaviImePriimek(randime, randpriimek);
            tmp.NastaviIzobrazbo(randizobrazba);
            tmp.NastaviStarost(randstarost);
            kandidati[i] = tmp;
    }
    // tu poišči najprimernejšega kandidata in ga izpiši
    Console.WriteLine("Najprimernejsi je: " + NajPrimernejsi(kandidati));
    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
}
```

Spremeni metodo *NajPrimernejsi,* tako, da vrne kandidata, ki je najmlajši in ima zahtevano stopnjo izobrazbe (ali višjo). Recimo, da so si v banki premislili in bi radi namesto mladega neizkušenega direktorja zaposlili najstarejšega (ne glede na izobrazbo). Poišči ga!

Sestavi razred *Denarnica*, ki vsebuje celoštevilsko spremenljivko, ki predstavlja količino denarja v njej kot javno spremenljivko (v centih) in metodo public string *ToString()*, ki vrne niz "V denarnici je # centov". Ustvari tudi tabelo desetih denarnic z naključno mnogo denarja in jih izpiši.

- Ali lahko na ta način zagotoviš, da v denarnici nikoli ne bo negativno mnogo denarja?
- Seveda ne. To bi se na primer zgodilo, če bi programer s firme Nekaj pacamo d.o.o. po pomoti vstavil v spremenljivko, ki predstavlja količino denarja, negativno vrednost. Zato moramo razred popraviti! Namesto javne spremenljivke vzemimo privatno, poleg tega pa dopišimo še metodi dvigni in placaj, ki v denarnico dodata oz. iz nje odvzameta denar. Če želimo plačati več, kot imamo denarja v denarnici, naj metoda placaj vrže napako!

  Opomba: namesto teh dveh metod bi lahko napisali samo t.i. "set" metodo: *public void SetVrednost(int vrednost)*, ki količino denarja v denarnici nastavi na vrednost (in pri tem seveda pregleda, če je ta vrednost pozitivna). Ampak glede na naš razred je izbira dveh metod bolj naravna.
- Sestavi tudi testni program, kjer preizkusiš delovanje tojega razreda. V denarnico dodaj nekaj denarja, nekaj ga odvzemi in se prepričaj, da je v njej pravilna količina denarja. Preveri tudi plačilo prevelikega zneska in ustrezno odreagiraj!
- Ali prejšnjo nalogo sploh lahko v celoti izpolniš? Denar res lahko dodaš in ga odvzameš, ker pa je količina denarja privatna spremenljivka razreda, je od zunaj nikakor ne moreš prebrati! Ta problem se reši tako, da v razred dodaš javno metodo public int KolicinaDenarja(), ki nam vrne količino denarja v denarnici.
- Dopolni razred s konstruktorjem public Denarnica(int denar), ki ustvari denarnico z dano količino denarja. S pomočjo metode KolicinaDenarja preveri, če se je tvoja denarnica pravilno inicializirala. Ustvari tudi denarnico z negativno mnogo denarja in preveri, če metode razreda Denarnica v tem primeru vržejo napako. Ne pozabi je tudi ujeti v glavnem programu, sicer se ti bo program sesul!
- Denarnica ima tudi svojega lastnika. Dodaj privatno spremenljivko lastnik tipa string, ki vsebuje ime lastnika denarnice. Dopolni konstruktor tako, da poleg količine denarja sprejme tudi ime lastnika. Razredu dodaj tudi metodo VrniLastnik, ki vrne ime lastnika denarnice. Popravi tudi metodo ToString, tako da izpis vključuje tudi ime lastnika denarnice.
- Dopolni testni program tako, da ustvariš tabelo desetih denarnic z naključno količino denarja med 0 in 1000. Izpiši jih!
- Kdo izmed lastnikov si lahko privošči nakup banjice sladoleda, ki stane 800 centov?

Sestavite razred *PodatkovniNosilec*, ki predstavlja medij za shranjevanje podakov. Vsebuje naj informacije o kapaciteti medija (celo število, enota je bajt), izdelovalcu (niz), osnovni ceni (celo število, enota je evroCent) in stopnji obdavčitve (celo število, enota je procent). Vsebovati mora vsaj javne metode:

- *public PodatkovniNosilec()* - konstruktor, ki ustvari nosilec kapacitete 650Mb, proizvajalca "neznan", s ceno 100 centov in 20% davkom.
- *public PodatkovniNosilec(string ime, int kapaciteta, int cena, int davek)* - konstruktor s parametri.
- *public void NastaviKapaciteto(int bajti)* - nastavi kapaciteto medija v bajtih. Če je kapaciteta negativna, naj se ne spremeni.
- *public void NastaviIme(string ime)* - nastavi ime izdelovalca medija. Če je ime enako null, naj se ne spremeni.
- *public void NastaviCeno(int cena)* - nastavi ceno. Če je negativna, naj se ne spremeni.
- *public void NastaviObdavcitev(int pocent)* - nastavi obdavčitev v procentih. Če so procenti izven meja 0-100, naj se ne stara vrednost ne spremeni.
- *public int VrniOsnovnoCeno()* - vrne osnovno ceno medija (brez davka).
- *public double VrniDavek()* - koliko davka je potrebno plačati za ta medij (v evrih).
- *public double VrniProdajnoCeno()* - vrne osnovno ceno + davek (v evrih).
- *public string VrniIme()* - vrne ime proizvajalca medija.
- *public double VrniKapaciteto(char enota)* - vrni kapaciteto v enoti, ki jo določa znak enota:
  - 'b' - bajti
  - 'k' - kilobajti
  - 'M' - megabajti
  - 'G' - gigabajti

  Pazi na to, da je en kilobajt 1024 (in ne 1000) bajtov!
- *public override string ToString()* - vrne niz, ki smiselno opisuje razred.
- Sestavi tudi testni program, v katerem preveriš delovanje tvojega razreda.

Podana je koda razreda *Ulomek*. Dopolni manjkajoče metode ter v glavnem programu (metoda *Main*) ustvari dva ulomka, katerih števec in imenovalec naj bo naključno število med med 1 in 9 (vključno z mejama). Ulomka seštej in rezultat izpiši na zaslon.

```
public class Ulomek
{
  int stevec;
  int imenovalec;

  // Privzeti konstruktor ustvari ulomek 1/1
  // DOPOLNI!


  // Ustvari podani ulomek. Pazi tudi na pravilnost podatkov. Če so
  //napačni, ustvari enak ulomek kot v privzetem konstruktorju.
  public Ulomek(int stevec, int imenovalec)
  {
    // DOPOLNI!
  }

  // Metoda prišteje ulomek u k trenutnemu ulomku.
  public void Pristej(Ulomek u)
  {
    // DOPOLNI!
  }

  /*Metoda zmnoži trenutni ulomek z ulomkom u ter zmnožek vrne kot
  rezultat.Trenutni ulomek ter ulomek u se pri tem ne smeta spremeniti*/
```

```
  public DOPOLNI Zmnozi(Ulomek u)
  {
    // DOPOLNI!
  }
  // Metoda vrne vrednost ulomka kot realno število.
  public DOPOLNI Vrednost()
  {
    // DOPOLNI!
  }

  // Vrne niz oblike »stevec/imenovalec«
  DOPOLNI string ToString()
  {
    // DOPOLNI!
  }

  // Metoda okrajša dani ulomek
  public void Okrajsaj()
  {
    // DOPOLNI
  }

  // Obratna vrednost.
  public void Obrni()
  {
  // DOPOLNI
  }
}
```

Na osnovi kode spodaj podanega razreda *Oseba* sestavite razred *Stranka*, ki predstavlja bančno stranko tega komitenta. Poleg lastnosti, so enake kot v razredu *Oseba*, mora hraniti tudi podatke o tekočem računu (niz), številki EMŠO (niz) ter povprečnem mesečnem prilivu na tekoči račun (realno število).

```
public class Oseba
{
    private string ime;
    private string priimek;

    public Oseba(string ime, string priimek)
    {
        this.ime = ime;
        this.priimek = priimek;
    }
    public string VrniIme()
    {
        return this.ime;
    }
    public string VrniPriimek()
    {
        return this.priimek;
    }
}
```

Pri tem mora razred zadoščati naslednjim pogojem:

- vse spremenljivke v njem morajo biti privatne

- Poznati mora javni konstruktor, ki sprejme pet parametrov: ime, priimek, številko tekočega računa, številko EMŠO ter podatek o povprečnem mesečnem prilivu na račun. Pri tem mora biti zadnji podatek pozitivni+o realno število, prvi štirje pa neprazni nizi. Če to ni izpolnjeno, ustrezno reagiraj.
- Poznati mora metode za vračanje podatkov, ki jih hrani: imena, priimka, številke tekočega računa, številke EMŠO ter povprečnega priliva na račun.
- Metoto ToString, ki na smiselen način izpiše podatke o stranki.

Nazadnje sestavite testni program, kjer ustvarite tabelo desetih strank z izmišljenimi podatki, ter med njimi poščite stranko z najvišjim mesečnim prilivom na račun ter tistega z po leksikografski urejenosti prvim imenom (če jih je več z enakim največjim prilivom ali enakim prvim imenom poiščite kateregakoli) ter izpišite njune podatke na zaslon (z uporabo metode ToString).

Dalmatinci I (brez računalnika). Denimo, da smo želeli sestaviti razred Dalmatinec, ki ima lastnosti ime psa in število njegovih pik. Koda razreda je:

```
public class Dalmatinec
{
    public string ime;
    private int steviloPik;

    public Dalmatinec()
    {
        this.ime = "Reks";
        this.steviloPik = 0;
    }

    public void ImePsa(string ime)
    {
        ime = this.ime;
    }

    private void NastaviIme(string ime)
    {
        this.ime = ime;
    }

    public void NastaviSteviloPik(int steviloPik)
    {
        this.steviloPik = steviloPik;
    }
}
```

V glavnem programu smo ustvarili objekt *Dalmatinec* z imenom *d* in mu želimo nastaviti število pik na 100 in ime na *Pika*. Kateri način je pravilen? Pri nepravilnih povej, kaj in zakaj ni pravilno.

```
a) d.NastaviIme("Pika"); d.NastaviSteviloPik(100);
b) d.ime = "Pika"; d.steviloPik = 100;
c) d.ime = "Pika"; d.NastaviSteviloPik(100);
d) d.imePsa("Pika"); d.NastaviSteviloPik(100);
e) d.imePsa("Pika"); d.steviloPik = 100;
f) d.NastaviIme("Pika"); d.steviloPik = 100;
g) nobeden, ker tega sploh ne moremo storiti
```

Sedaj želimo našemu razredu Dalmatinec dodati tudi podatke o spolu psa. Ta podatek bomo hranili v spremenljivki spol. Interno (znotraj razreda) naj logična vrednost true pomeni ženski, false pa moški spol. Dopolnite razred tako, da bo zadoščal naslednjim trem pogojem:

- Spremenljivka spol naj ne bo dostopna izven razreda *Dalmatinec*

- obstaja naj metoda *KaksenSpol*, ki v primeru samca vrne 'm', v primeru samice pa 'f'.
- spol se nastavi le ob ustvarjanju objekta. Morali boste torej napisati konstruktor razreda *Dalmatinec*, ki sprejme kot parameter znak za spol. Ta naj bo kot zgoraj 'm' za samca in 'f' za samico. Predpostavite, da bo parameter zagotovo znak 'm' ali 'f'.

Sestavi še metodo *public static int SteviloSamcev(Dalmatinec[] dalmatinci)*, ki prešteje število samcev v tabeli dalmatincev.

- Potrebujemo razred, ki bo hranil podatke o objektih tipa *Instrukcije* z naslednjimi komponentami: vrsta inštrukcije, število opravljenih ur in ali je inštrukcija možna. Razred naj ima tudi metode in sicer: inštrukcija se opravlja, inštrukcija se ne opravlja. Z osnovnim konstruktorjem določi vrednost (poljubno) vsem komponentam razreda. Z dodatnim konstruktorjem poskrbi za možnost nastavitve začetne vrednosti za vrsto inštrukcije, opravljene ure ter ali je inštrukcija možna.
  Na osnovi razreda *Instrukcije* napiši še testni program, ki bo kreiral in izpisal dva objekta razreda *Instrukcije* in sicer tako, da bodo začetne vrednosti pri prvem objektu nastavljene z osnovnim konstruktorjem, pri drugem objektu pa z dodatnim konstruktorjem.

- Sestavi razred, ki predstavlja osnovo za izdelavo programa, s pomočjo katerega bomo pregledovali rezultate nekega športnega tekmovanja. Sestavi razred *Tekmovalec*, ki ima naslednje komponente: startno številko, ime, priimek in klub. Vsa polja so tipa string. Napiši vsaj dva konstruktorja in pripravi ustrezne *get/set* metode za vse te podatke. Z metodo *public String toString()* naj se izpišejo podatki o tekmovalcih (startna številka, ime, priimek, klub).

- Sestavi razred *Pacient*, ki ima tri komponente: *ime*, *priimek*, *krvna_skupina*. Komponenti *ime* in *priimek* naj bosta *public*, *krvna_skupina private*, vse tri pa tipa *string*. Napiši vsaj dva konstruktorja:

  - prazen konstruktor, ki vse tri komponente nastavi na "NI PODATKOV",
  - konstruktor, ki sprejme vse tri podatke in ustrezno nastavi komponente.

  Z metodo *public String toString()* naj se izpišejo podatki o pacientu (ime, priimek, krvna skupina).

- Sestavi razred *Piramida*, ki predstavlja pokončno piramido, ki ima za osnovno ploskev kvadrat. V razredu hrani podatke o dolžini stranice osnovne ploskve in višino piramide. Obe komponenti naj imata tip dostopa **private**. Višina in stranica nista nujno celi števili. Napiši vsaj dva konstruktorja:
  - prazen konstruktor, ki ustvari piramido višine 1 in s stranico osnovne ploskve dolžine 1
  - konstruktor, ki sprejme podatka o višini in dolžini stranice osnovne ploskve
  - Napiši program, ki ustvari tabelo 50 naključnih piramid in med njimi poišče piramido z največjo višino.

  Napiši **get** in **set** metode. V **set** metodah pazi na smiselnost podatkov (višina in dolžina stranice ne smeta biti negativni,...). Napiši tudi metodo *toString*, ki vrne niz s smiselnim izpisom podatkov o piramidi.

- Sestavi razred, kjer boš v objektih te vrste hranil ime države, njeno glavno mesto, površino (v km^2) in število prebivalcev. Pripravi ustrezne *get/set* metode za vse te podatke in vsaj dva konstruktorja. Ne pozabi tudi na smiselno metodo toString.

- Sestavi razred *Igrača*, ki ima tri privatne spremenljivke: tip igrače(tip *string*), število igrač na zalogi(tip *int*) ter ceno igrače(tip *double*).
  - sestavi prazen konstruktor;
  - sestavi konstruktor s tremi parametri;
  - napiši ustrezne *set* in *get* metode (pazi na smiselne vrednosti spremenljivk);
  - napiši metodo *toString*, ki naj smiselno izpiše, kateri tip igrače, koliko kosov je na zalogi in kakšna je cena;

- dodaj še metodo *zalogaIgrac*, ki sprejme pozitivno celo število, če se je število igrač povečalo (dobava novih) ter negativno celo število če se je število igrač zmanjšalo (prodane igrače). Temu primerno spremeni število igrač na zalogi in pri tem pazi, da število igrač ne pade pod nič;
- napiši metodo *znizanaCena*, ki sprejme celo število med 0 in 100 (%), to je za koliko procentov se bo znižala cena igrače, in nastavi novo ceno igrače;

napiši glavni program, ki bo ustvaril pet tipov igrač, tri izmed njih znižal za 10, 20 in 50% ter dvema spremenil zalogo.

V kemijskem laboratoriju večkrat preverjamo kislost snovi in jo definiramo s *pH* vrednostjo. Napiši razred *Snov*, ki bo imel tri komponente: *imeSnovi*, *ion* in *kislost*. Prvi dve sta tipa *string*, tretja pa *int*. Ker je kislost zelo pomemben podatek o snovi, zagotovi, da bo zagotovo pravilen. Zato bo ustrezna spremenljivka imela privaten dostop. Sestavi tudi ustrezni *get* in *set* metodi. Pazi, da boš tudi v konstruktorju s parametri poskrbel, da ne bo prišlo do napačne nastavitve. Če bo uporabnik podal napačno kislost, ustvari objekt, kjer za prva dva podatka napišeš, da sta neobstoječa, kislino pa pustiš tako, kot jo je vnesel uporabnik. Sestavi tudi konstruktor brez parametrov, ki naredi objekt, ki predstavlja vodo. Kreiraj tabelo snovi in napiši stavke za izpis vseh objektov.

Napiši razred *Kocka* tako, da bo izpeljan iz razreda *Kvadrat*. Razreda naj poleg svojih podatkov vsebujeta še privzeti konstruktor, metode za postavitev vrednosti podatkov in metode za izračun površine, obsega in volumna.