OBJEKTNO ORIENTIRANO PROGRAMIRANJE

mag. Boštjan Resinovič,
univ. dipl. inž. računalništva in informatike

Literatura:

- Programiranje 1; skripta, B2
- Lokar M., Uranič S.: Programiranje 1 (Ministrstvo za šolstvo in šport Republike Slovenije, 2008)
- Uranič S.: C# .NET
- Price, J., Gundelroy M.: Mastering Visual C# .NET (Sybex, 2002)
- Sharp, J., Jagger, J.: Microsoft Visual C# .NET Step by Step (Microsoft Press, 2003)
- Davis S, R, Sphar C.: C# for Dummies (Wiley, 2005)
- Bervar, G.: C++ na kolenih (Študentska založba, 2008)
- Žumer, V., Brest J.: Strukturirano programiranje v C++ (FERI, Inštitut za informatiko, 2001)
- Žumer, Viljem, Janez Brest: Objektno programiranje v C++ (FERI, Inštitut za informatiko, 2001)
- Bonačić D.: Kratek priročnik jezika C# in razlike z jezikom C++ ; zbrano gradivo za predavanja (FERI, Inštitut za informatiko, Maribor 2008) //dostopno na http://lisa.uni-mb.si/osebje/bonacic/predmeti/orodja_za_razvoj_aplikacij_in_vs3/CSharp/CSharp_zbrano_gradivo.pdf

Spletni viri:

- MSDN Library, http://msdn.microsoft.com/en-us/library/default.aspx
- C# Tutorial, http://www.csharp-station.com/Tutorial.aspx
- C# Tutorial, http://www.softsteel.co.uk/tutorials/cSharp/cIndex.html
- C# Practical Learning, http://www.functionx.com/csharp/
- C# Tutorial, http://www.ssw.uni-linz.ac.at/Teaching/Lectures/CSharp/Tutorial/
- Free On Line Programming Tutorials, http://www.programmingtutorials.com/csharp.aspx
- C# Examples, http://www.csharp-examples.net
- Orodja za razvoj aplikacij, http://lisa.uni-mb.si/osebje/bonacic/predmeti/orodja_za_razvoj_aplikacij_in_vs3/index.htm

Objektno programiranje /1

- Objektno programiranje temelji na dejstvu, da človek stvari zmeraj razvršča v nekakšne razrede.
  - Če vidimo bernardinca, takoj vemo, da je to pes, da je sesalec, da je toplokrvna žival... Vsak pes ima vse lastnosti sesalcev, poleg tega pa tudi svoje lastne.

- Pri objektnem programiranju je zato mogoče ustvariti hierarhijo razredov:
  - razred pes podeduje vse lastnosti razreda sesalec, razred bernardinec vse lastnosti razreda pes.
  - na ta način lahko enkrat sprogramirane razrede uporabimo za ustvarjanje novih in na ta način prihranimo na času

Objektno programiranje /2

- Lastnosti objektnega programiranja
  - kapsuliranje (skrivanje podatkov, oziroma omejevanje dostopa)
  - dedovanje (uporaba že ustvarjenih razredov za izpeljevanje novih)
  - polimorfizem (objekti različnih razredov se na isto sporočilo odzovejo vsak na različne načine)
- Več o posameznih lastnostih bomo spoznali kasneje.

Objektno programiranje – razredi in objekti /1

- Bernardinec, jazbečar, pudelj, pes, sesalec, itd. so razredi. **Razred si lahko predstavljamo kot tip**.

- Jazbečarja z imeni Fifi in Reksi pa sta objekta. **Objekti so primerki nekega razreda**.

- Ko programer definira razred, pravzaprav izdela svoj tip, ko ustvari spremenljivko tega svojega tipa, pa naredi primerek (instanco) razreda, torej objekt.

Objektno programiranje – razredi in objekti /2

- Razredi so v osnovi sestavljeni iz
  - **podatkov** (angl. fields) (**značilnosti** objektov, pri psu npr. barva dlake, starost, teža ...),
  - **metod** (angl. methods) (**aktivnosti**, ki jih objekti izvajajo sami ali pa se jih tičejo, pri psu npr. lajanje, tekanje, cepljenje ...)

- Podatkom in metodam nekega razreda rečemo tudi člani ali elementi razreda. V resnici jih je še več, omenimo vsaj še **dogodke** (angl. properties).

- C# pozna poleg podatkov in metod še eno osnovno kategorijo elementov, ki ji rečemo **lastnosti** (angl. properties). Več o lastnostih bomo spoznali kasneje.

Objektno programiranje – razredi in objekti /3

- V C# ustvarimo razred s ključno besedo struct ali s ključno besedo class.
  - struct je vrednostni tip in je napram class precej omejen
  - class je referenčni tip, omogoča dedovanje

```
class ime_razreda
{
    podatki
    metode
    …
 }
```

  - Vrstni red podatkov in metod ni pomemben.

Objektno programiranje – razredi in objekti /4

- primer razreda

```
class Vozilo
  {
     private string znamka;
     private string model;
     private int letoIzdelave;

     public void Izpisi()
     {
        Console.Write("znamka: " + znamka + "  ");
        Console.Write("model: " + model + "  ");
        Console.WriteLine("letnik: " + letoIzdelave);
        Console.WriteLine();
     }
  }
```

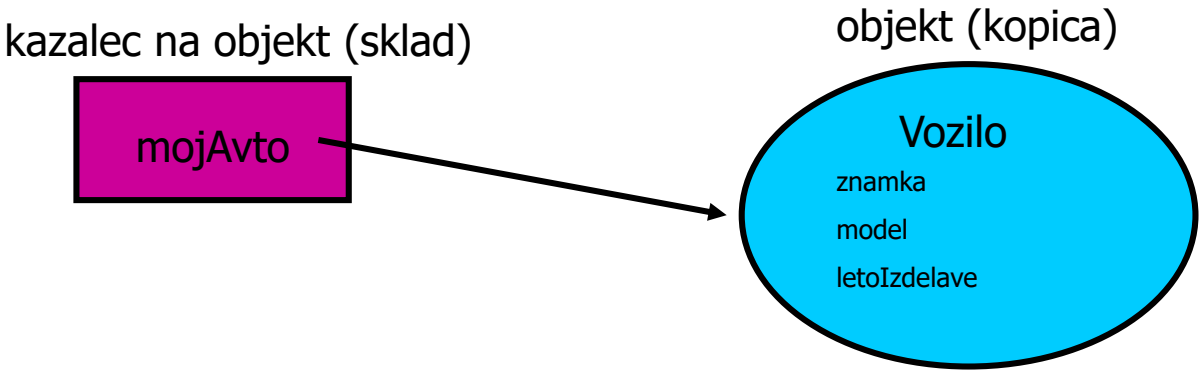
Program ObjektnoProgramiranje (RazrediInObjekti)

Objektno programiranje – razredi in objekti /5

- ustvarjanje objektov nekega razreda

```
Vozilo mojAvto;                  //ustvari referenco na vozilo na skladu
mojAvto = new Vozilo();          //dejansko ustvari primerek vozila na kopici

Vozilo tvojAvto = new Vozilo();  //ustvarjanje objekta v eni vrstici
```



Objektno programiranje – razredi in objekti /6

- Dostop do elementov(članov) razreda:
  - Podatke in metode sicer definiramo nad razredom, uporabljamo pa jih pri objektih.
  -
  - Do njih pridemo tako, da imenu objekta dodamo piko, nato pa navedemo element razreda npr:

```
mojAvto.Izpisi();
mojAvto.letoIzdelave = 2008;  //v našem primeru napaka, ker je private!!!
```

Program ObjektnoProgramiranje (RazrediInObjekti)

Objektno programiranje – kapsuliranje /1

- Vsak objekt je **kapsula**, ki dovoljuje le dostop do tistih elementov, ki niso skriti.
- Pri našem razredu Vozilo se na Izpisi() lahko skličemo npr. v glavnem programu na ostale element pa ne!
  - Izpisi() je javna metoda (public)
  - znamka, model in letoIzdelave pa so privatni podatki (private)
- C# pozna naslednje nivoje skrivanja oz. načine dostopa
  - **public** (element je dostopen od povsod)
  - **private** (**privzeto,** element je dostopen le v razredu, kjer je definiran)
  - **protected** (element je dostopen v razredu, kjer je definiran in v razredih, izpeljanih iz njega)
  - **internal** (element je dostopen v celem programu (natančneje assembly-ju))
  - **protected internal** (element je dostopen v celem programu (natančneje assembly-ju) in v razredih, izpeljanih iz razreda, kjer je definiran)

Objektno programiranje – kapsuliranje /2

- Privzet nivo skrivanja elementov je **private**.
- Elementi izven razreda privzeto torej niso vidni!
- **Podatke** največkrat definiramo kot private, **metode** pa kot internal ali public.
  - Do podatkov dostopamo posredno prek metod razreda.
  - C# pozna še eno možnost: podatke deklariramo kot lastnosti (property) razreda.
- Na ta način se izognemo morebitnim **napakam**, pa tudi potrebi, da bi uporabnik razreda nujno **poznal obliko**, v kateri so shranjeni podatki, npr.:
  - ne dovolimo ročnega spreminjanja neke lastnosti, ki se spreminja avtomatsko, npr. vrha sklada
  - datum vpisujemo kot tri cela števila, izpisujemo pa kot niz, pa sploh ne vemo, kako je shranjen (kot niz, kot objekt razreda MojDatum, kot objekt razreda DateTime, kot število ms od nekega trenutka …)
- Ko načrtujemo razred je **pomembno**, da predvidimo vse potrebne metode oz določimo vse potrebne lastnosti.
  - Če želimo določen podatek nastaviti samo za branje, metode za nastavljanje vrednosti pri zgornjih dveh načinih enostavno ne določimo.

Objektno programiranje – kapsuliranje /3

- Prikažimo uporabo get in set metode oz. lastnosti na primeru znamke in modela.
  Če sta različna od null, naj bo
  - vsebina shranjena v nizu iz malih črk dolžine 15 znakov (na koncu se vstavijo presledki)
  - ob vračanju
    - znamka prikazana v dejanski dolžini s samimi velikimi črkami
    - model prikazan v dejanski dolžini z veliko začetnico

- Primer get in set metode

```
public void SetZnamka(string znamka)
{
     //z this se skličemo na objekt, pri enakem imenu podatka in parametra je nujen
     //shranjujemo z malimi črkami kot niz 15 znakov oz. kot null
    if (znamka == null) this.znamka = null;
    else this.znamka = znamka.ToLower().PadRight(15);
}

public string GetZnamka()
{
     //vračamo z velikimi črkami v dejanski dolžini oz. kot null
    //tu this ne rabimo, lahko pa ga uporabimo (this.znamka namesto znamka)
    if (znamka == null) return null;
    else return znamka.ToUpper().Trim();
}
```

Objektno programiranje – kapsuliranje /4

- Primer lastnosti
  - lastnost običajno poimenujemo kot sam podatek razreda, le da z veliko začetnico

```
public string Model
 {
    get                 //vračamo z velikimo začetnico v dejanski dolžini oz. kot null
    {
         if (model == null) return null;

         string s, rez;
         s = model.Trim();
         rez = s.Substring(0,1).ToUpper();
         rez = rez + s.Substring(1, s.Length - 1);
         return rez;
    }

     set        //shranjujemo z malimi črkami kot niz 15 znakov oz. kot null
     {
      // z value se skličemo na vrednost, ki jo nastavljamo lastnosti
        if (value == null) model = null;
        else model = value.ToLower().PadRight(15);
    }
 }
```

Objektno programiranje – konstruktorji in destruktorji /1

- Za vsak razred obstajata dve posebni metodi:
  - privzeti **konstruktor**
  - privzeti **destruktor**
- Če ne naredimo svojih, se pri ustvarjanju objekta izvede privzeti konstruktor, pri uničenju objekta pa privzeti destruktor.
- Če naredimo svoje konstruktorje in destruktor, se izvedejo ti.
  - konstruktorjev je lahko več (govorimo o preoblaganju), destruktor je lahko en sam
  - konstruktorji so poimenovani z imenom razreda
  - destruktor je poimenovan z imenom razreda, ki ima spredaj znak ~
  - konstruktorji in destruktor nimajo tipa (niti void)
  - konstruktorje definiramo kot javne
  - destruktorji ne smejo imeti določenega nivoja skrivanja
  - **primer glave konstruktorja:** public MojRazred()
  - **primer glave destruktorja:** ~MojRazred()
- Konstruktorjev in destruktorjev ne kličemo direktno, izvedejo se avtomatično
  - konstruktorji ob ustvarjanju
  - destruktor, ko smetar (garbage collector) uniči objekt

Objektno programiranje – konstruktorji in destruktorji /2

- **Obstoječi** privzeti kostruktor ustvari objekt in nastavi podatke na privzete vrednosti:
  - na **0** za vrednostne tipe
  - na **null** za referenčne tipe
- Če želimo druge vrednosti, moramo narediti svoj privzeti ali drugačen konstruktor.
- Obstajajo tri glavne vrste konstruktorjev:
  - **privzeti**
    - nima argumentov
    - nastavi vrednosti podatkov na privzete, razen če ga preprogramiramo, kar je zelo pogosto
  - **pretvorbeni (parametrizirani)**
    - tvori nov objekt, tako da pretvori vrednosti, ki so skladne s tipi objektovih podatkov v objekt
    - te vrednosti so parametri konstruktorja
  - **kopirni**
    - tvori nov objekt iz že obstoječega
    - argument konstruktorja je že obstoječi objekt istega razreda
- Če naredimo kateri koli svoj konstruktor, **moramo** narediti tudi svoj privzeti konstruktor!

Objektno programiranje – konstruktorji in destruktorji /3

- Obstaja privzeti destruktor, ki
  - uniči objekt
  - sprosti ves pomnilnik, ki ga je neposredno ali posredno uporabljal objekt
- Svoj destruktor napisemo, če je potrebno sprostiti še druge vire
  - datoteke, ki jih je odprl objekt
  - dodatna okna,
  - mrežne vire …
- Destruktor se izvede, ko **smetar** uniči objekt
  - programer ne more zahtevati, kdaj naj se to zgodi
  - smetar sam ugotovi, kdaj je nek objekt "za v smeti"
  - tak objekt bo uničen ob naslednjem pobiranju smeti

Objektno programiranje – konstruktorji in destruktorji /4

PRIMER:

- Napišimo nov privzeti konstruktor, ki bo
  - letoIzdelave nastavil na letos,
  - znamko in model nastavil na null
- Napišimo še dva pretvorbena konstruktorja
  - prvi nastavi letoIzdelave na zahtevano leto, znamko in model pa na null
  - drugi nastavi vse tri na zahtevane vrednosti
- Dodajmo še kopirni konstruktor.
- Napišimo destruktor, ki bo napisal, kateri objekt se trenutno uničuje.
- Za vse podatke imejmo definirane ustrezne lastnosti.

Objektno programiranje – konstruktorji in destruktorji /5

```
//preprogramiran privzeti konstruktor
public Vozilo()
{
    //nastavimo letoIzdelave na letos, ostalo pustimo privzeto (torej se nastavi na null)
    LetoIzdelave = DateTime.Today.Year;
}

public Vozilo(int letoIzdelave) //pretvorbeni konstruktor 1
{
    //nastavimo letoIzdelave, ostalo pustimo privzeto (torej se nastavi na null)
    LetoIzdelave = letoIzdelave;
}

public Vozilo(string znamka, string model, int letoIzdelave) //pretvorbeni konstruktor 2
{
    //nastavimo vse podatke na vrednosti, ki so parametri konstruktorja
    Znamka = znamka;
    Model = model;
    LetoIzdelave = letoIzdelave;
}
```

Program ObjektnoProgramiranje (KonstrukturjiDestruktorji)

Objektno programiranje – konstruktorji in destruktorji /6

```
  //kopirni konstruktor
public Vozilo(Vozilo vozilo1)
{
    //nastavimo vse podatke novega objekta na podatke objekta vozilo1
    Znamka = vozilo1.Znamka;
    Model = vozilo1.Model;
    LetoIzdelave = vozilo1.LetoIzdelave;
}

~Vozilo()
{
     string z = "null", m = "null";
     if (Znamka != null) z = Znamka;
     if (Model != null) m = Model;
     Console.WriteLine("Vozilo {0} {1}, letnik {2} uničeno!", z, m, LetoIzdelave);
 }
```

Program ObjektnoProgramiranje (KonstrukturjiDestruktorji)

Objektno programiranje – konstruktorji in destruktorji /7

- **kopirni konstruktor vs. prirejanje**

```
Vozilo avto4 = new Vozilo(avto1);
//avto4 je samostojen objekt, ki ima enake vrednosti kot avto1
avto4.LetoIzdelave = 2003;
//spremenili smo samo avto4, ne pa tudi avto1

Vozilo avto5 = avto2;
//avto5 in avto2 sta en in isti objekt!!!
// (za avto5 sploh nismo klicali konstruktorja, še vedno imamo le 4 objekte)
//v resnici sta to dve različni referenci na skladu, ki se sklicujeta na isti
// objekt na kopici
avto5.LetoIzdelave = 2005;
//če spremenimo avto5 se bo spremenil tudi avto2 in obratno!!!
```

Program ObjektnoProgramiranje (KonstrukturjiDestruktorji)

Objektno programiranje – konstruktorji in destruktorji /8

- **kopirni konstruktor – plitka oz. globoka kopija**
  - če ima objekt podatke referenčnih tipov, se je treba odločiti, ali naj ima nov objekt svoje podatke (**globoka kopija**) ali pa imata oba objekta iste podatke (**plitka kopija**, če spremenimo podatke enega, se spremenijo tudi podatki drugega objekta, saj gre za ene in iste podatke)

  - recimo, da ima objekt podatek polje treh celih števil, v kopirnem konstruktorju pa kodo
    - plitka kopija

```
polje = vozilo1.polje;
```

    - globoka kopija

```
polje = new int[3];
for (int i = 0; i < polje.Length; i++)
{
   polje[i] = vozilo1.polje[i];
}
```

Objektno programiranje – primer

- Dokončajmo razred vozilo. Ima naj :
  - podatke: znamka, model, letoIzdelave
  - lastnosti za vse tri podatke
    - **Znamka**
      - shranjena v nizu iz malih črk dolžine 15 znakov (na koncu se vstavijo presledki)
      - prikazana v dejanski dolžini s samimi velikimi črkami
    - **Model**
      - shranjen v nizu iz malih črk dolžine 15 znakov (na koncu se vstavijo presledki)
      - prikazana v dejanski dolžini z veliko začetnico
    - **LetoIzdelave**
      - samo naredi podatek letoIzdelave javen
  - štiri konstruktorje
    - privzeti konstruktor, ki bo letoIzdelave nastavil na letos, znamko in model pa na null
    - dva pretvorbena konstruktorja
      - prvi nastavi letoIzdelave na zahtevano leto, znamko in model pa na null
      - drugi nastavi vse tri na zahtevane vrednosti
    - kopirni konstruktor
  - destruktor, ki bo napisal, kateri objekt se trenutno uničuje
  - dve metodi za izpis podatkov (objektno in statično)
  - metodi Vzgi() in Ugasni(), ki bosta napisali, da je vozilo prižgano oziroma ugasnjeno

Statične in objektne metode 1/2

- **objektne**
  - potrebujejo objekt
  - **klic**: **objekt**.Metoda()

- **statične**:
  - uporaba rezervirane besede static (pred navedbo tipa)
  - neke vrste razredne metode, ne potrebujejo objekta
  - direktno lahko dostopajo le do statičnih članov razreda (seveda pa lahko dostopajo tudi do nestatičnih podatkov nekega objekta, ki je bil v metodo prenesen kot parameter – kot v naslednjem primeru izpisa)
  - **klic v razredu**: Metoda()
  - **klic izven razreda**: **Razred.**Metoda()
  - primer:
    - metode v razredu Math, npr. Math.Abs(-5);
    - objektov razreda Math sploh nikoli ne ustvarjamo, vse metode tega razreda torej morajo biti statične

Statične in objektne metode 2/2

```
public void Izpisi()
{
    Console.Write("znamka: " + Znamka + "   ");
    …
}

public static void Izpisi(Vozilo vozilo)
{
    Console.Write("znamka: " + vozilo.Znamka + "   ");
    …
}
```

- **klica**: avto1.Izpisi(); Vozilo.Izpisi(avto1);

Program ObjektnoProgramiranje (StaticConstReadonly)

Statični podatki 1/3

- tudi statični podatki so "razredni"
- to pomeni, da **si vsi objekti statične podatke delijo** (v pomnilniku je le en podatek, ne glede na število objektov), medtem ko za nestatične velja, da ima vsak objekt svoje

- primer
  - **private static int stVozil;**

    - za evidenco, koliko vozil je trenutno ustvarjenih
    - podatek je avtomatično inicializiran na 0
    - v konstruktorjih ga povečamo za ena
    - v destruktorju ga zmanjšamo za ena

Statični podatki 2/3

```
class Vozilo
{
 private static int stVozil;
    …
 public Vozilo(string znamka, string model, int letoIzdelave)
   {
      //nastavimo letoIzdelave na letos
      Znamka = znamka;
      Model = model;
      LetoIzdelave = letoIzdelave;
      stVozil++;
   }
~Vozilo()
  {
     string z = "null", m = "null";
     if (Znamka != null) z = Znamka;
     if (Model != null) m = Model;
     stVozil--;
     Console.WriteLine("Vozilo {0} {1}, letnik {2} uničeno! Ostaja še {3} vozil.", z, m, LetoIzdelave, stVozil);
  }
…
}
```

Program ObjektnoProgramiranje (StaticConstReadonly)

Statični podatki 3/3

```
class Vozilo
{
    ...
 public static void Izpisi(Vozilo vozilo)
  {
      Console.Write("znamka: " + vozilo.Znamka + "  ");
      Console.Write("model: " + vozilo.Model + "  ");
      Console.Write("letnik: " + vozilo.LetoIzdelave + "  ");
     Console.WriteLine("vseh vozil: " + stVozil); //stVozil je statičen podatek, ne rabimo objekta
   }

 public void Izpisi()
  {
      Console.Write("znamka: " + Znamka + "  ");
      Console.Write("model: " + Model + "  ");
      Console.Write("letnik: " + LetoIzdelave + "  ");
     Console.WriteLine("vseh vozil: " + stVozil); //stVozil je statičen podatek, ne rabimo objekta,
                                                  //nanj se lahko skličemo tudi v objektni metodi
    }
}
```

- **Klic izven razreda**: Console.Write(" vseh vozil: " + **Vozilo**.stVozil);

Statične lastnosti

- Tako kot statične metode in statični podatki so možne tudi statične lastnosti:

```
class Vozilo
{
    …
   public static int StVozil
   {
       get { return stVozil; }
       set { stVozil = value; }
   }
}
```

- **Klic izven razreda**: Console.Write(" vseh vozil: " + **Vozilo**.StVozil);

Program ObjektnoProgramiranje (StaticConstReadonly)

Statični konstruktor

- ne da se ga klicati direktno
- se avtomatično izvede še preden je iz tega razreda izpeljan kateri koli objekt ali preden se prvič dostopa do katerega koli statičnega elementa
- obstaja lahko le en statični konstruktor za posamezen razred
- za vsak razred se statični konstruktor izvede le enkrat
- ne sme imeti parametrov
- pred statičnim konstruktorjem nikoli ne napišemo nivoja dostopnosti (besedic private, public, ..), ampak le besedico static
- ker je statičen, v njem ne moremo dostopati do nestatičnih elementov razreda, uporabimo ga torej za inicializacijo statičnih podatkov
- za ta namen lahko namesto statičnega konstruktorja uporabimo inicializacijo ob deklaraciji

Statični razredi

- imajo le statične elemente
- ne moremo jih klicati direktno, se pravi, da ni mogoče ustvariti nobenega objekta statičnega razreda
- statični razredi se naložijo ob zagonu programa avtomatično

- **uporaba**: za grupiranje metod in podatkov v razred, takrat ko primerki razreda niso potrebni
- **primer**: razred Math
  - statičen podatek PI
  - statične metode, npr, Abs, Sqrt, Sin, Cos …
  - nikoli ne ustvarimo primerka razreda Math, npr takole:
    - `Math x = new Math(0.88); double y = x.Sin();`
  - zmeraj kar takole (brez primerka):
    - `double x = 0.88; double y = Math.Sin(x);`

const

- public const double pi = 3.14; //statičen, skupen vsem objektom
- Rezervirana beseda const je veljavna le za podatke, za lastnosti (in seveda metode) pa ne.
- konstanten, **nespremenljiv** podatek, inicializirati ga je možno le ob deklaraciji
- konstantnih podatkov torej ni mogoče spreminjati nikjer, niti v konstruktorju
  - NAPAKA:

```
MojRazred(float Pi)
{
   pi=Pi; //pi-ja ne moremo spreminjati
}
```

- konstante so avtomatično statične
- rezervirane besede static pri const ne smemo uporabiti
- NAPAKA: pi=3.1415; //pi-ja ne moremo spreminjati
- Program ObjektnoProgramiranje (StaticConstReadonly)

readonly

- Rezervirana beseda readonly je veljavna le za podatke, za lastnosti in metode pa ne.
- readonly podatki so lahko statični ali objektni
  - public static readonly double pi = 3.14; //statičen, skupen vsem objektom
  - private readonly string ime; //vsak objekt ima svoje ime

- **samo bralen** podatek: inicializirati ga je možno ob deklaraciji **in v ustreznem konstruktorju**

```
MojRazred(float Pi, string imeObjekta)
{
    ime=imeObjekta; //OK, ni napaka
    pi=Pi; //NAPAKA, pi je statičen, inicializacija je možna le v staitčnem konstruktorju
}
static MojRazred()
{
   pi=Pi; //OK, ni napaka
   ime=imeObjekta; //NAPAKA, ime ni statičen element, inicializacija je možna v
                   //nestatičnem konstruktorju
}
```

- NAPAKA: ime="Lojzek"; //imena ne moremo spreminjati

- Program ObjektnoProgramiranje (StaticConstReadonly)

Objektno programiranje – Prekrivanje spremenljivk

- Kaj se zgodi, če sta neka lokalna spremenljivka ali formalni parameter metode poimenovana enako kot spremenljivka, deklarirana v razredu?
  - spremenljivka, ki je bila definirana v razredu, je **še zmeraj dosegljiva**
  - **ni pa več vidna**, saj jo prekrije lokalna spremenljivka oz. parameter metode
  - Da dosežemo prekrito (nevidno) spremenljivko, je treba uporabiti njeno **polno ime**,
    - pri **statičnih** podatkih ime razreda npr. **MojRazred.x** namesto x.
    - v **objektnih** metoda ključno besedo **this** npr. **this.y** namesto y
- V isti kategoriji lahko eno ime seveda uporabimo le enkrat (npr. dve spremenljivki v istem bloku ne moreta biti enako poimenovani).

Program PrekrivanjeSpremenljivk

this

- Uporaba
  - zmeraj, ko drugače ni jasno, na kaj se sklicujemo
  - dostikrat ga lahko spustimo

  - Tipičen primer:

```
MojRazred (int y)
{
    this.y = y;
}
```

Objektno orientirani programiranje – moduli razredov

- **Modul razredov** je samostojna datoteka, ki vsebuje enega ali več razredov.
- Za uporabo modula je treba v projekt vključiti izvorno kodo modula (.cs datoteko)

- Kreiranje modula
  - npr. v SolutionExplorerju **projektu** dodamo nov razred (Add/New/Class)
  - novi razred ima isto področje imen (namespace) kot projekt
    - lahko izbrišemo
    - lahko pustimo
    - lahko spremenimo
- Uporaba modula v projektu, kjer smo modul ustvarili
    - če pustimo isto imensko področje ali ga izbrišemo, je vse enako, kot če bi bil razred v isti datoteki kot projekt
      - Razred.ImeMetode(parametri);
    - če spremenimo področje imen, je treba sklice na metode prilagoditi
      - NovoPodročjeImen.Razred.ImeMetode(parametri)
      - če na vrhu dodamo using NovoPodročjeImen;
        lahko spet uporabimo Razred.ImeMetode(parametri)
- Uporaba modula v drugem projektu
  - npr. v SolutionExplorerju projektu dodamo **obstoječ** razred (Add/Existing Item)
  - uporaba je enaka kot v drugem primeru zgoraj

Objektno orientirano programiranje – knjižnice razredov

- **Knjižnica razredov** je shranjena v datoteki s končnico .dll je torej dinamično povezljiva knjižnica, ki vsebuje enega ali več razredov.
- Za uporabo modula je treba v projekt vključiti sklic na knjižnico (.dll datoteko)

- Kreiranje knjižnice
  - npr. v SolutionExplorerju **rešitvi** dodamo nov projekt tipa *Class Library* (Add/New Project/Class Library)
  - nastavimo področje imen na želeno vrednost
  - dodamo razrede, ki **morajo** biti **javni**
  - metode v razredih **morajo** biti **javne**
  - prevedemo (rezultat je .dll datoteka, ki se nahaja v podmapi projekta \bin\debug)
- Uporaba knjižnice v rešitvi, kjer smo knjižnico ustvarili
  - **dodamo sklic na knjižnico** (npr. v Solution Explorerju desni klik na projekt in Add Reference)
  - na zavihku Projects izberemo knjižnico
  - sklici na metode iz knjižnice
    - PodročjeImen.Razred.ImeMetode(parametri)
    - če na vrhu dodamo using PodročjeImen; lahko uporabimo Razred.ImeMetode(parametri)
- Uporaba knjižnice v drugi rešitvi
  - postopek je enak, samo da namesto zavihka Projects izberemo Browse in izberemo ustrezno .dll datoteko (ta način je splošen, uporabimo ga lahko tudi v rešitvi, kjer smo knjižnico kreirali)

Moduli in knjižnice - primer

- **Modul:**
  - razred trikotnik (stranice a, b, c; privzeti in pretvorbeni konstruktor, objektna metoda za obseg)
  - v osnovi del konzolskega projekta
  - uporaba v okenskem projektu
  - Program Moduli

- **Knjižnica:**
  - enak razred trikotnik kot zgoraj
  - uporaba v konzolskem projektu
  - še ena knjižnica za trikotnik (vse enako, samo da namesto treh stranic uporabimo polje dolžine tri)
  - zamenjava dll-ja in zagon programa brez ponovne kompilacije
  - opomba: knjižnice ne moremo kar preimenovati – potrebno je spremeniti ime za Assembly v Properties
  - Program Knjiznice

Uporaba poti do knjižnic 1/2

- Če želimo, lahko aplikaciji določimo pot do map, kjer so knjižnice.
- Pot je relativna glede na mapo z .exe datoteko.
- Knjižnice so lahko v mapi, kjer je .exe datoteka ali v mapah na poti.
- Načinov je več:

  - AppDomain.CurrentDomain.AppendPrivatePath("DLL-ji");
  - Ta način je v novejših verzijah Visual Studia sicer označen kot zastarel, vendar še deluje v okenskih projektih, ne pa tudi v konzolskih.

  - uporaba **konfiguracijske datoteke** z elementom <probing>

  - **GAC** (Global Assembly Cache)

Uporaba poti do knjižnic 2/2

- uporaba **konfiguracijske datoteke** z elementom <probing>
  - konfiguracijsko datoteko dodamo projektu z desnim klikom nanj v Solution Explorerju/Add/New Item
  - na ta način se ob prevajanju ustvari datoteka ImePrograma.exe.config, ki mora biti ravno s tem imenom v isti mapi kot program, da jo program ob izvajanju prebere
  - vanjo vpišemo

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="DLL1;DLL2"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

  - S tem smo določili, da so lahko DLL datoteke v mapah DLL1 in DLL2, ki sta podmapi mape, kjer je EXE datoteka.
  - Navedemo lahko tudi pot do globljih podmap, npr. DLLs\DLL1.
  - Dovoljene so samo podmape, ..\ ne deluje.

- Program Knjiznice

Preoblaganje operatorjev 1/5

- S preoblaganjem operatorjev **damo standardnemu operatorju nek drug pomen**.

- Za preoblaganje operatorjev uporabimo posebno metodo **operator**

rez = prvi * drugi;
si lahko predstavljamo kot
rez = operator * (prvi, drugi)

  - metoda operator mora biti
    - **static** (operator pripada razredu in ne objektu)
    - **public**
  - uporablja **le prenos po vrednosti** (ref in out nista dovoljena)

Preoblaganje operatorjev 2/5

- Shema preoblaganja:

  public static tip operator op (argumenti)

- kjer je
  - **op** operator, ki ga preoblagamo
  - vsaj eden od argumentov mora biti objekt razreda v katerem preoblagamo operator

- Primer
  - public static Ulomek operator - (Ulomek u1, int x)

```
{
  Ulomek nov = new Ulomek();
  nov.st = u1.st - x * u1.im;
  nov.im = u1.im;
  return nov;
}
```

Program PreoblaganjeOperatorjev

Preoblaganje operatorjev 3/5

- Preobložimo lahko **vse operatorje, razen**
  - **&& || = . ?: ?? -> => is as checked unchecked default delegate new sizeof typeof**
- **Primerjalni operatorji** morajo biti preobloženi paroma (če preobložimo enega moramo tudi drugega)
  - **==** in **!=**
  - **<** in **>**
  - **<=** in **>=**
- **Sestavljeni operatorji** so preobloženi **avtomatično**, če preobložimo osnovnega:
  - npr. +=, če preobložimo +
- **++** oz. **--** preobložimo **le enkrat**, **C# sam poskrbi** za **prefiksno** in **postfiksno** verzijo

Preoblaganje operatorjev 4/5

- POZOR:
  - ulomek - int ne reši primera int - ulomek
  - rabimo še eno preobložitev operatorja –
  - alternativa je implementacija operatorja za pretvorbo iz int v ulomek (potem rabimo le ulomek – ulomek)

- Pretvorbe
  - implicitne
  - eksplicitne
  - če napišemo implicitno, jo lahko uporabimo tudi kot eksplicitno

  - iz našega razreda
  - v naš razred

Preoblaganje operatorjev 5/5

- implicitna (avtomatična) iz Ulomek v double (ne navedemo tipa, ta je določen z operatorjem, tukaj double)
  - `public static implicit operator double(Ulomek u)`
    `{ return (double)u.st / u.im; }`

- eksplicitna (zahtevana) pretvorba iz Ulomek v int (pride lahko do izgube natančnosti)
- `public static explicit operator int(Ulomek u)`
  `{ return u.st / u.im; }`

- implicitna (avtomatična) pretvorba int v Ulomek
- `public static implicit operator Ulomek(int st)`
  `{ return new Ulomek(st, 1); }`

Program PreoblaganjeOperatorjev

Primer

- Nepremičninska agencija
- Program NepremicninskaAgencija

Dedovanje – osnove /1

- **Dedovanje** (*ang.* ***inheritance***) omogoča ustvarjanje hierarhičnih skupin razredov, ki si delijo podobne funkcionalnosti.

- Dedovanje omogoča, da se neki razred izpelje iz drugega. Izpeljani razred se imenuje *podrazred* ali *otrok*. Izpeljani razred lahko doda svoje metode in lastnosti in na ta način naredi razred bolj specializiran.

- S pomočjo dedovanja se lahko razredi grupirajo v drevesne hierarhične skupine. Nižje kot je razred v drevesu, bolj specializiran je.

- Pri dedovanju se vse metode, podatki in lastnosti osnovnega razreda (nadrazreda) **prenesejo** na podrazred. Tako ni potrebno podvajati programske kode.

- C# omogoča **le enostavno dedovanje**: razred je možno izpeljati samo iz enega nadrazreda.

Dedovanje – osnove /2

- Imejmo razred Vozilo in iz njega izpeljimo razred MotornoVozilo:

```
// razred Vozilo
class Vozilo
{
    private int hitrost;

    public int Hitrost
    {
        get
        {
            return hitrost;
        }
    }

    public void Pospesi(int kmh)
    {
        hitrost += kmh;
    }

    public void Upocasni(int kmh)
    {
        hitrost -= kmh;
    }
}
```

Program ObjektnoProgramiranje_Dedovanje

Dedovanje – osnove /3

- **Osnovna sintaksa za izpeljavo razreda:**

```
class podrazred : osnovniRazred
{
}
```

```
// razred MotornoVozilo
   class MotornoVozilo : Vozilo
   {
   }

// uporaba razreda MotornoVozilo v glavnem programu
   MotornoVozilo v = new MotornoVozilo();
   Console.WriteLine("Hitrost: {0} km/h", v.Hitrost);  // vrne 0
   v.Pospesi(25);
   Console.WriteLine("Hitrost: {0} km/h", v.Hitrost);  // vrne 25
```

Program ObjektnoProgramiranje_Dedovanje_OsnoveDedovanja

Dedovanje – dodajanje specialnosti /1

- Dodajanje specializiranih funkcionalnosti izpeljanemu razredu:

```
// razred Kolo
  class Kolo : Vozilo
  {
      public void Pozvoni()
      {
          Console.WriteLine("Cin cin!");
      }
  }
```

Program ObjektnoProgramiranje_Dedovanje_DodajanjeSpecialnosti

Dedovanje – dodajanje specialnosti /2

- Dodajanje specializiranih funkcionalnosti izpeljanemu razredu:

```
// razred MotornoVozilo
   class MotornoVozilo : Vozilo
   {
      private float seGoriva;
      private float velikostTanka;

      public float SeGoriva
      {
         get  { return seGoriva;}
         set { seGoriva = value;}
      }

      public float VelikostTanka
      {
         get { return velikostTanka;}
         set {velikostTanka = value;}
      }

      public float PotrebnoGoriva()
      {
         return velikostTanka - seGoriva;
      }
   }
```

Program ObjektnoProgramiranje_Dedovanje_DodajanjeSpecialnosti

Dedovanje – konstruktorji in destruktorji /1

- Konstruktorji in destruktorji **se ne dedujejo** iz baznih razredov.

- Če ne določimo drugače se pri ustvarjanju objekta izpeljanega razreda avtomatično izvede **privzeti konstruktor** osnovnega razreda. Na ta način osnovni konstruktor lahko “inicializira osnovni“ razred preden se zažene konstruktor podrazreda.

- Podobno velja za destruktorje, le da se ti poženejo v obratnem zaporedju kot konstruktorji (najprej torej destruktor izpeljanega in šele nato destruktor osnovnega razreda).

- Če želimo, lahko v konstruktorju izpeljanega razreda **eksplicitno** kličemo **katerikoli konstruktor osnovnega razreda**.
  - uporabimo **base** **(parametri izbranega konstruktorja)**

- **Eksplicitno** pa lahko kličemo tudi **katerikoli konstruktor izpeljanega razreda**.
  - uporabimo **this** **(parametri izbranega konstruktorja)**

Dedovanje – konstruktorji in destruktorji /2

- Konstruktorja v osnovnem razredu

```
public Vozilo()
{  Hitrost = 1; }

public Vozilo(int hitrost)
{ Hitrost = hitrost; }
```

- Konstruktorja v izpeljanem razredu

```
public MotornoVozilo()
{
   SeGoriva = 10;
   VelikostTanka = 40;
}

public MotornoVozilo(int kmh)
{
}

MotornoVozilo mv1 = new MotornoVozilo(20);
```

  - Najprej se avtomatično pokliče privzeti konstruktor nadrazreda.
    - hitrost ne bo 20, ampak bo 1
  - Privzeti konstruktor izpeljanega razreda se avtomatično ne pokliče, vrednosti se nastavijo na privzete vrednosti tipa (0 za vrednostne tipe in null za referenčne).
    - seGoriva bo 0 in ne 10, velikostTanka bo 0 in ne 40

Program ObjektnoProgramiranje_Dedovanje_KonstruktorjiDestruktorji

Dedovanje – konstruktorji in destruktorji /3

- Za pravilno delovanje bo treba uporabiti eksplicitni klic konstruktorja.
  - Nek konstruktor lahko **eksplicitno** pokličemo v **inicializatorju** (delu kode, ki se začne z : in se nahaja med glavo in telesom konstruktorja)
  - primer eksplicitnega klica izbranega konstruktorja tega razreda:

```
public MotornoVozilo(int kmh) : this ()
{
}
```

  - primer eksplicitnega klica izbranega konstruktorja osnovnega razreda:

```
public MotornoVozilo(int kmh) : base (kmh)
{
}
```

Program ObjektnoProgramiranje_Dedovanje_KonstruktorjiDestruktorji

Dedovanje – konstruktorji in destruktorji /4

- Popravimo konstruktor, da bo deloval, kot smo želeli:

```
public MotornoVozilo(int kmh)  :  this ()
{
        Hitrost = kmh;
}

public MotornoVozilo()
{
    SeGoriva = 10;
    VelikostTanka = 40;
}
```

  - Zaradi this() v inicializatorju pretvorbenega konstruktorja podrazreda se najprej pokliče privzeti konstruktor podrazreda.
  - Ta najprej implicitno kliče privzeti konstruktor osnovnega razreda:
    - hitrost bo 1
  - Nato se izvede telo privzetega konstruktorja podrazreda:
    - seGoriva bo 10
    - velikostTanka bo 40
  - Prek javne lastnosti Hitrost se nastavi vrednost podatku hitrost v telesu pretvorbenega konstruktorja podrazreda:
    - hitrost bo zdaj kmh

Program ObjektnoProgramiranje_Dedovanje_KonstruktorjiDestruktorji

Dedovanje – konstruktorji in destruktorji /5

- Še en primer (uporabimo eksplicitni klic pretvorbenega konstruktorja nadrazreda):

```
public MotornoVozilo(int kmh, float seGoriva, float velikostTanka) : base(kmh)
{
   SeGoriva = seGoriva;
   VelikostTanka = velikostTanka;
}
```

  - Zaradi base(kmh) v inicializatorju se **ne kliče privzeti** ampak **eksplicitno določeni konstruktor nadrazreda**
    - hitrost bo zdaj kmh
  - Nato nastavimo vrednost podatkov prek lastnosti:
    - SeGoriva(posledično this.seGoriva) bo seGoriva
    - VelikostTanka(posledično this.velikostTanka) bo velikostTanka
  - Če ne bi navedli base(kmh), bi se izvedel privzeti konstruktor osnovnega razreda (nadrazreda) in hitrost bi bila 1.

Program ObjektnoProgramiranje_Dedovanje_KonstruktorjiDestruktorji

Dedovanje – konstruktorji in destruktorji /6

- vsem konstruktorjem dodajmo ustrezen izpis
- ustvarimo destruktor za osnovni in izpeljani razred, oba naj imata ustrezen izpis
- v programu preizkusimo, če drži, kar smo povedali o vrstnem redu izvajanja konstruktorjev in destruktorjev
  - konstruktorji:
    - najprej nadrazred,
    - potem izpeljani razred
  - destruktorji:
    - najprej izpeljani razred,
    - potem nadrazred

Program ObjektnoProgramiranje_Dedovanje_KonstruktorjiDestruktorji

Dedovanje – dostop do elementov /1

- V izpeljanem razredu so **privatni** elementi sicer podedovani, a so **nedostopni.**

- Doslej smo uporabljali tudi javne (dostopni povsod) in interne (dostopni v okviru programa) elemente. Ta dva nivoja dostopov tipično uporabljamo za metode oz. lastnosti, redkeje za podatke.

- Obstajata pa še dva nivoja dostopa, oba povezana z dedovanjem:
  - **protected**: element je dostopen v razredu, kjer je definiran in v razredih, izpeljanih iz njega
  - **protected internal**: element je dostopen v celem programu in v razredih, izpeljanih iz razreda, kjer je definiran

Program ObjektnoProgramiranje_Dedovanje_PrivateDalmatinec

Dedovanje – dostop do elementov /2

- Problem se torej tipično pojavi pri podatkih, saj so ti največkrat privatni, medtem ko metode niso.

```
class Pes
   {
      private string ime;
      private int teza;
   }
class Dalmatinec : Pes
   {
      private int stPik;
   }
```

- Problem dostopa do podatkov, podedovanih iz nadrazreda rešujemo na tri načine:
  - v osnovnem razredu napišemo potrebne javne (ali interne) metode
  - za privatne podatke ustvarimo javne (ali interne) lastnosti
  - spremenimo deklaracijo osnovnega razreda in namesto privatnih uporabimo zaščitene podatke

Program ObjektnoProgramiranje_Dedovanje_PrivateDalmatinec

Dedovanje – dostop do elementov /3

- Uporaba javnih metod v osnovnem razredu:

```
public void Nastavi(string ime, int teza)
        {
            this.ime = ime;
            this.teza = teza;
        }
public void Izpisi()
        {
            Console.Write("ime: " + this.ime + " teža: " + this.teza);
}
```

- Uporaba javnih metod v izpeljanem razredu:

```
public void Nastavi(string ime, int teza, int stPik)
        {
// Metoda Nastavi iz nadrazreda je dostopna, ker je javna. Ker je del razreda Pes, lahko dostopa do imena in teže.
            base.Nastavi(ime, teza);
            this.stPik = stPik;
        }
new public void Izpisi()
        {
            // Metoda Izpisi iz nadrazreda je dostopna, ker je javna. Ker je del razreda Pes, lahko dostopa do imena in
    teže.
            base.Izpisi();
            Console.Write(" število pik: " + this.stPik);
        }
```

- Javne metode so seveda dostopne tudi povsod drugje, npr. v glavnem programu.

Program ObjektnoProgramiranje_Dedovanje_JavneMetode_Dalmatinec

Dedovanje – dostop do elementov /4

- **base** lahko uporabimo:
  - v inicializatorju v konstruktorju razreda za klic konstruktorja nadrazreda
    - base(parametri)
  - v telesu konstruktorja ali katere koli **objektne** metode za dostop do elementov nadrazreda
    - base.Metoda(parametri) //bolj pogosta raba
    - base.podatek, base.Lastnost //redkeje uporabljeno

- **base** lahko izpustimo, kadar s tem ne ustvarimo nobene nejasnosti
  - npr., ko v izpeljanem razredu ni metode z istim podpisom in je jasno, da kličemo metodo nadrazreda)

Program ObjektnoProgramiranje_Dedovanje_JavneMetode_Dalmatinec

Dedovanje – dostop do elementov /5

- V **statičnih** metodah **base** ne moremo uporabiti
  - namesto tega uporabimo **ime razreda**

```
public static void Izpisi(Dalmatinec dalmatinec)
        {
            Pes.Izpisi(dalmatinec);  //dalmatinec je pes, zato je tak klic povsem OK
            Console.Write(" število pik: " + dalmatinec.stPik);
        }
```

Program ObjektnoProgramiranje_Dedovanje_JavneMetode_Dalmatinec

Dedovanje – dostop do elementov /6

- new pri metodi izpis
  - metodi v osnovnem razredu in izpeljanem razredu imata enak podpis
  - metoda v izpeljanem **skrije** metodo v osnovnem razredu (prevajalnik javi opozorilo)
    - v izpeljanem razredu uporabimo new (enako obnašanje, kot če ne naredimo nič, le da ni več opozorila)
    - ali pa v osnovnem razredu uporabimo virtual v izpeljanem pa override
    - o razlikah med načinoma bomo več izvedeli pri polimorfizmu

Program ObjektnoProgramiranje_Dedovanje_JavneMetode_Dalmatinec

Dedovanje – dostop do elementov /7

- Uporaba javnih (ali internih) lastnosti:
  - lahko bi za javne (ali interne) naredili kar podatke same
  - slabo, ker to omogoči direkten dostop do podatkov od koderkoli iz programa in s tem poveča možnost napak
  - javne lastnosti glede tega niso problem: omogočajo kontrolo pravilnosti vpisa
  - set metodi lastnosti lahko še dodatno omejimo način dostopa, npr. namesto za javno jo lahko naredimo za zaščiteno
- V osnovnem razredu:

```
public int Teza
    {
        get
        {
            return teza;
        }
        protected set
        {
            if (value < 1) value = 1;
            teza = value;
        }
    }
```

- V osnovnem in izpeljanem razredu lahko beremo in nastavljamo težo, povsod drugod pa jo lahko le beremo.

Program ObjektnoProgramiranje_Dedovanje_JavneLastnosti_Dalmatineec

Dedovanje – dostop do elementov /8

- Uporaba zaščitenih podatkov:
  - deklaracijo osnovnega razreda spremenimo tako, da so podatki zaščiteni in ne privatni
  - v izpeljanih razredih bodo ti podatki zdaj dostopni
  - drugje, npr. v glavnem programu, pa ne bodo dostopni
- V osnovnem razredu:

```
protected string ime;
protected int teza;
public void Izpisi()
        {
            Console.Write("ime: " + this.ime + " teža: " + this.teza);
        }
```

- V izpeljanem razredu:

```
public void Izpisi()
        {
            //podatki se podedujejo, zato lahko uporabimo this ali base (ali pa nič));
            Console.Write("ime: " + this.ime + " teža: " + base.teza + " število pik: " + stPik);
        }
```

Program ObjektnoProgramiranje_Dedovanje_Protected_Dalmatineec

Dedovanje – primer za enostavno dedovanje

- Napišimo program za evidenco študentov:
  - določimo naštevni tip spol (seveda z elementoma moški / ženski)
  - definirajmo razrede
    - Datum
      - privatni podatki dan, mesec, leto (cela števila)
      - privzeti, pretvorbeni, kopirni konstruktor
      - objektni metodi za vpis in izpis
    - Oseba
      - privatni podatki ime, priimek, spol, datum rojstva
      - privzeti, pretvorbeni konstruktor, ki prejme ime, priimek, spol, datum ter pretvorbeni konstruktor, ki prejme ime, priimek, spol, dan, mesec, leto
      - objektni metodi za vpis in izpis in objektna metoda SePoroci, ki nastavi nov priimek
    - Student (izpeljan iz Oseba)
      - dodaten privatni podatek smer
      - ekvivalentni konstruktorji kot pri osebi, pozor na eksplicitni klic ustreznega konstruktorja nadrazreda Oseba
      - objektni metodi za vpis (omogoča le vpis smeri) in izpis (omogoča izpis podedovanih podatkov in smeri, za izpis podedovanih lastnosti kliče ustrezno metodo nadrazreda Oseba)
    - Preizkusi private / protected, ustvari Javno metodo / lastnost za ime in priimek.

Program ObjektnoProgramiranje_Dedovanje_EnostavnoDedovanje_OsebaStudent

Pretvorbe in prirejanja v hierarhiji razredov /1

- V hierarhiji razredov je mogoče objekte pretvarjati v drug razred oz. referencam na objekt prirediti objekt drugega razreda.

- Vsak objekt izpeljanega razreda je tudi objekt vseh nadrazredov v hierarhiji. Obratno NE velja.
- Za test uporabimo operator is.

```
//motorno vozilo je tako MotornoVozilo kot tudi Vozilo
if (mv1 is MotornoVozilo) Console.WriteLine("mv1 JE motorno vozilo, z
mv1.GetType() dobimo: {0}", mv1.GetType());
if (mv1 is Vozilo) Console.WriteLine("mv1 JE TUDI vozilo, čeprav z mv1.GetType()
dobimo: {0}", mv1.GetType());
```

- Operator is obakrat vrne true, GetType() pa obakrat„MotornoVozilo“

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /2

- **Pretvorbo (angl. casting)** lahko naredimo na dva načina
  - `(Vozilo)motornovozilo1`
  - `motornoVozilo1 as Vozilo`

  - Če pretvorba ne uspe, vrže prvi način izjemo, drugi pa ne. Neuspešnost as izrazi tako, da vrne null.

- Pri pretvorbi se objekt v pomnilniku dejansko **ne spremeni**, le obnaša se kot objekt razreda, v katerega pretvarjamo.

- V hierarhiji lahko pretvarjamo
  - **navzgor (angl. upcasting)**: bolj specialen objekt pretvarjamo v manj specialnega, torej v nekega **višje** v hierarhiji (npr. motorno vozilo v vozilo). Je **neproblematična**.
  - **navzdol (angl. downcasting)**: bolj splošen objekt pretvarjamo v bolj specialnega, torej v nekega **nižje** v hierarhiji (npr. vozilo v motorno vozilo). Je **prepovedana** (razen, ko je le navidezna).

Pretvorbe in prirejanja v hierarhiji razredov /3

- **Pretvorba navzgor (upcasting):**
  - Je neproblematično, saj ima objekt, ki ga pretvarjamo vse, kar imajo tudi objekti ciljnega razreda.

```
Console.WriteLine("mv1: " + (mv1 as Vozilo).GetType());
      //tudi če ga pretvorimo navzgor, se mv1 še zmeraj predstavlja kot
      //motorno vozilo, vendar pa lahko uporabi metode Vozil namesto
      // svojih
 (mv1 as Vozilo).NakaziSmer(Smer.desno); //mv1 pretvorimo v Vozilo in
                                         //pokličemo metodo NakaziSmer,
                                         //uporabi se metoda iz razreda
                                         //Vozila (razen, seveda, ko
                                         //imamo polimorfno obnašanje)
//((Vozilo)mv1).NakaziSmer(Smer.desno);
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /4

- **Pretvorba navzgor (upcasting):**
  - Uporabimo jo lahko tudi drugače, v kombinaciji s prirejanjem

```
//v1 = (Vozilo)mv1;    //v1 kaže na mv1, ki smo ga pretvorili navzgor v Vozilo,
                       //z v1 zdaj nimamo več dostopa do specialnosti
                       //motornih vozil, čeprav se v1 predstavlja kot MotornoVozilo
v1 = mv1 as Vozilo;    //pretvorbo lahko naredimo tudi z as, če gre za pretvorbo v
                       //hierarhiji dedovanja
Console.WriteLine("v1: " + v1.GetType());   //MotornoVozilo
v1.NakaziSmer(Smer.levo);    //izvede se metoda razreda Vozila
v1 = v2;   //zdaj je v1 spet Vozilo
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /5

- **Pretvorba navzdol (downcasting):**
  - Je načeloma odsvetovana in razen pri zgolj navidezni pretvorbi navzdol prepovedana.

```
((MotornoVozilo)v1).NakaziSmer(Smer.desno);  //NAPAKA: program se zaradi
                                             //izjeme pri pretvorbi sesuje
                                             //(InvalidCastException)
(v1 as MotornoVozilo).NakaziSmer(Smer.desno);  //NAPAKA: pretvorba sicer
                                               //ne povzroči sesutja,
                                               //a vrne null, ko poskusimo
                                               //z null nakazati smer, pa
                                               //se program sesuje
                                               //(NullReferenceException)
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /6

- **Pretvorba navzdol (downcasting):**
  - Pretvorba navzdol je dovoljena le, ko je v bistvu le **navidezna**, to pa je takrat, ko spremenljivka nadtipa v pomnilniku že kaže na objekt podtipa

```
Vozilo mv3 = new MotornoVozilo();    // v pomnilnik se namesti objekt razreda
                                     // MotornoVozilo, ki se tudi predstavlja kot
                                     // MotornoVozilo, pa kljub temu nima dostopa
                                     // do specialnosti MotornihVozil,
                                     // saj je mv3 Vozilo

//Console.WriteLine(mv3.SeGoriva);   //NAPAKA, mv3 sicer kaže na MotornoVozilo
                                     //(tako se tudi predstavi), a dostopa do
                                     //specialnosti nima

// navidezna pretvorba navzdol pa omogoči, da dostopamo do specialnosti
// MotornegaVozila prek mv3 (to deluje, ker mv3 kaže na objekt, ki je dejansko
// MotornoVozilo
Console.WriteLine("mv3: " + (mv3 as MotornoVozilo).SeGoriva);
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /7

- **Pretvorba navzdol (downcasting):**
  - Pretvorba navzdol je dovoljena le, ko je v bistvu le **navidezna**, to pa je takrat, ko spremenljivka nadtipa v pomnilniku že kaže na objekt podtipa

```
v1 = mv1;  //v1 se zdaj predstavlja kot MotornoVozilo
Console.WriteLine("v1: " + v1.GetType()); // MotornoVozilo
mv2 = v1 as MotornoVozilo; //dovoljeno, a pravzaprav ni pravi downcasting,
                           //saj v1 zdaj v pomnilniku kaže na objekt, ki je že
                           //sam po sebi motorno vozilo (mv1)

//mv2 ima dostop do specialnosti MotornihVozil, v1 pa jih nima
Console.WriteLine("mv2: " + mv2.SeGoriva);
// Console.WriteLine(v1.SeGoriva);  //NAPAKA, v1 sicer kaže na MotornoVozilo
                                    //(tako se tudi predstavi), a dostopa do
                                    // specialnosti nima

//navidezna pretvorba navzdol pa omogoči, da dostopamo do specialnosti
//MotornegaVozila prek v1 (to deluje, ker v1 kaže na objekt, ki je dejansko
//MotornoVozilo)
Console.WriteLine("v1: " + (v1 as MotornoVozilo).SeGoriva);
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /8

- **Operator is**
  - Pri (navidezni) pretvorbi navzdol, je priporočljivo z operatorjem **is** preveriti, ali je pretvorba mogoča.

```
Vozilo v1 = new MotornoVozilo(); Vozilo v2 = new Vozilo();

if (v1 is MotornoVozilo) Console.WriteLine("v1: " + (v1 as MotornoVozilo).SeGoriva);
else Console.WriteLine("Pretvorba v1 v MotornoVozilo ni mogoča!");

if (v2 is MotornoVozilo) Console.WriteLine("v2: " + (v2 as MotornoVozilo).SeGoriva);
else Console.WriteLine("Pretvorba v2 v MotornoVozilo ni mogoča!");
//pri v1 uspe, pri v2 ne

if (v1 is Vozilo) Console.WriteLine("v1 je tudi Vozilo " +);
else Console.WriteLine("v1 ni vozilo ni mogoča!");
//preverjanje uspe, v1 je tudi Vozilo, saj je vsako MotornoVozilo tudi Vozilo
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /9

- **Prirejanje navzdol je dovoljeno.**

```
v1 = mv1;
```

- Pri prirejanju navzdol gre v resnici za t.i. **upcasting**, saj pride do implicitne **pretvorbe navzgor** (iz podrazreda v nadrazred), kot bi napisali

```
v1=(Vozilo)mv1;
```

```
v1 = mv1;  //prirejanje navzdol deluje, v1 se predstavlja kot MotornoVozilo,
           //vendar ne more direktno dostopati do specialnih podatkov, lastnosti
           // ali metod Motornih vozil, npr. do SeGoriva, saj je prišlo do
           // implicitne pretvorbe navzgor v Vozilo

//Console.WriteLine(v1.SeGoriva);   //napaka, v1 nima dostopa do
                                    //specialnosti motornih vozil
Console.WriteLine("v1: " + v1.GetType());   //MotornoVozilo (saj je bil
             //objekt, na katerega kaže v1 ustvarjen kot MotornoVozilo)
Console.WriteLine((v1 as MotornoVozilo).SeGoriva);    //to pa deluje

v1.NakaziSmer(Smer.levo);    //izvede se metoda razreda Vozila
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /10

- **Prirejanje navzgor pa ni dovoljeno (razen, če je navidezno).**

- Pri prirejanju navzgor gre v resnici za t.i. **downcasting torej za pretvorbo navzdol** (iz nadrazreda v podrazred), vendar je **implicitna pretvorba prepovedana**

```
mv1 = v1; //SINTAKTIČNA NAPAKA: implicitno prirejanje navzgor
          //ni mogoče
```

- Pri **eksplicitni pa pride do izjeme**

```
mv1 = (MotornoVozilo)v1; //NAPAKA: prirejanje navzgor ni
                         //dovoljeno, saj gre za
                         //pretvorbo navzdol, pride do izjeme

mv1 = v1 as MotornoVozilo;

mv1.NakaziSmer(Smer.levo); //NAPAKA: prirejanje navzgor
                           //ni dovoljeno (downcasting),
                           //rezultat je null, ko naredimo
                           //nekaj z null, pride do izjeme
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Pretvorbe in prirejanja v hierarhiji razredov /11

- **Možno pa je navidezno prirejanje navzgor** (enako kot je možna navidezna pretvorba navzdol).

```
v1 = mv1;  //pravzaprav v1 = (Vozilo)mv1;
Console.WriteLine("v1:" + v1.GetType()); //objekt na katerega kaže v1,
                                         //je v resnici MotornoVozilo,
                                         //tako se zdaj predstavi tudi v1

v1 = v2;   //NI NAPAKA: v1 je pravzaprav v resnici vseskozi Vozilo
           //(ne glede na to, kako se je vmes predstavljalo), zato ne gre za
           //pravo prirejanje navzgor oz. pretvorbo navzdol (downcasting)
           //se pa v1 zdaj spet predstavlja kot Vozilo
Console.WriteLine("NI NAPAKA, v1: " + v1.GetType());   //Vozilo
```

Program ObjektnoProgramiranje_Dedovanje_PrirejanjaPretvorbe

Objektno programiranje – polimorfizem /1

- **Polimorfizem** (mnogoličnost, grško: poli = mnogo, morph = oblika;) je lastnost objektov, da se na isto navodilo odzovejo vsak na svoj način. To dosežemo tako, da vsakemu razredu za določeno metodo sprogramiramo lastno obnašanje:
  - v osnovnem razredu določimo le del obnašanja, ki je skupno vsem razredom in metodo določimo za **navidezno** (ključna beseda **virtual**)
    - Primer:
      - v razredu Vozila ustvarimo objektno metodo NakaziSmer()
      - v njej izpišimo v katero smer vozilo zavija
  - v izpeljanih razredih metodo **prekrijemo** (ključna beseda **override**)
    - obe metodi **morata** imeti enak podpis
    - prekrivna metoda tipično kliče prekrito metodo (ni pa to nujno) in doda še kakšno specialnost
    - Primer:
      - v MotornemVozilu pokličimo metodo nadrazreda za nakazovanje smeri in izpišimo, kateri smernik utripa
      - v Kolesu izpišimo, katero roko je kolesar dvignil in kam kolo zavija

Objektno programiranje – polimorfizem /2

- izven vseh razredov

```
//enumeracija za spremembo smeri
public enum Smer { levo, desno };
```

- v osnovnem razredu

```
virtual public void NakaziSmer(Smer smer)
        {
            if (smer == Smer.levo) Console.WriteLine("Zavijam na levo.");
            else Console.WriteLine("Zavijam na desno.");
        }
```

- v razredu MotornoVozilo

```
override public void NakaziSmer(Smer smer)
        {
            //najprej pokličemo ustrezno metodo nadrazreda
            base.NakaziSmer(smer);
            if (smer == Smer.levo) Console.WriteLine("Utripa levi smernik.");
            else Console.WriteLine("Utripa desni smernik.");
        }
```

Program ObjektnoProgramiranje_Dedovanje_Polimorfizem

Objektno programiranje – polimorfizem /3

- v razredu Kolo

```
override public void NakaziSmer(Smer smer)
        {
            //ustrezno metode nadrazreda ne kličemo (stvar odločitve programerja)
            Console.WriteLine("Kolesar ima dvignjeno {0} roko. Kolo zavija {0}.", smer);
        }
```

- v glavnem programu

```
vozilo.NakaziSmer(Smer.levo);   //izpiše Zavijam na levo.
motornoVozilo.NakaziSmer(Smer.levo);   //izpiše Zavijam na levo. Utripa levi smernik.
kolo.NakaziSmer(Smer.levo);   //izpiše Kolesar ima dvignjeno levo roko. Kolo zavija levo.
```

Program ObjektnoProgramiranje_Dedovanje_Polimorfizem

Objektno programiranje – polimorfizem /4

- Polimorfno obnašanje pa je še mnogo bolj zanimivo:

```
Vozilo vozilo = new Vozilo();
MotornoVozilo motornoVozilo = new MotornoVozilo();
Kolo kolo = new Kolo();

Vozilo kabrio = new MotornoVozilo();    //prevajalnik zaradi konstruktorja ve, da je kabrio MotornoVozilo
kabrio.NakaziSmer(Smer.desno);          //obnaša se kot MotornoVozilo

((Vozilo)kabrio).NakaziSmer(Smer.desno);   //pretvorba navzgor ne spremeni obnašanja,
                                           //kabrio se obnaša kot MotornoVozilo
                                           //uporabi se najbolj specializirana metoda, ki se lahko

Vozilo mountainBike = kolo;          //ne ustvari se nov objekt (nimamo new),
                                     //mountainBike in kolo sta isti objekt razreda Kolo)
 mountainBike.NakaziSmer(Smer.desno);  //obnaša se kot Kolo


 vozilo = mountainBike;  //eksplicitna pretvorba navzdol je prepovedana, prirejanje pa uspe
 vozilo.NakaziSmer(Smer.desno);    //uporabi se najbolj specializirana metoda, ki se lahko
                                   //obnaša se kot kolo
```

Program ObjektnoProgramiranje_Dedovanje_Polimorfizem

Objektno programiranje – polimorfizem /5
prekrivanje vs. skrivanje

- Kadar želimo polimorfno obnašanje, metodo osnovnega razreda v podrazredu **prekrijemo** (kombinacija virtual in override)
  - to je **mogoče le pri objektnih metodah**, pri **statičnih pa NE**
  - prevajalnik pri klicu metode uporabi metodo podrazreda (najbolj specializirano, ki jo lahko), čeprav npr. uporabimo pretvorbo navzgor v nadrazred

```
((Vozilo)motornoVozilo).NakaziSmer(Smer.desno); //obnaša se kot motorno vozilo
```

- Če polimorfnega obnašanja ne želimo, metodo osnovnega razreda samo **skrijemo** (uporabimo new, v nadrazredu lahko imamo virtual ali pa ne, to ni pomembno)
  - polimorfnega obnašanja ni
  - prevajalnik vodi metodi kot povsem nepovezani
  - zanj imata metodi isto ime povsem slučajno, obnašanje je tako, kot da bi bili to metodi z dvema različnima imenoma

```
((Vozilo)kolo).NakaziSmer(Smer.desno); //obnaša se kot vozilo
```

Program ObjektnoProgramiranje_Dedovanje_SkrivanjePrekrivanjeMetod

Objektno programiranje – polimorfizem /6

- V C# vsi razredi dedujejo od razreda **object**, ki je torej najvišji v hierarhiji razredov.
- Razred **object** implementira nekaj metod, ki se podedujejo in ki jih je možno prekriti. Taka je tudi metoda ToString().
- Ker imajo vsi razredi metodo ToString, je možno vsak objekt uporabiti npr. takole:
  - Console.WriteLine(" Jaz sem objekt " + objekt);
  - avtomatično se uporabi objekt.ToString(), ki vrne string, ki ga je mogoče zlepiti z drugim nizom
  - privzeto obnašanje metode pa pogosto ni zadovoljivo, zato metodo ToString() največkrat prekrijemo v vseh izpeljanih razredih

```
public override string ToString()
{
    return "Jaz sem kolo.";
}
```

```
Kolo kolo = new Kolo();
Console.WriteLine(kolo);      //izpiše Jaz sem kolo.
```

Objektno programiranje – polimorfizem /7

- Polimorfno obnašanje je še posebej koristno v zvezi s polji objektov (natančneje polji referenc na objekte)
  - ustvarimo lahko polje objektov osnovnega razreda
  - v polje dodajamo objekte osnovnega razreda in izpeljanih razredov
  - nad vsemi elementi uporabimo enak klic
  - prevajalnik ve, kašnega razreda so posamezni elementi polja, zaradi polimorfnega obnašanja objektov lahko vsakič uporabi ustrezno obnašanje (pri enaki kodi se posamezni elementi obnašajo različno)

```
Vozilo[] poljeVozil = new Vozilo[5];   //ustvarim polje petih vozil (natančneje polje petih referenc na vozila)
//polje napolnim z različnimi vozili: objekti naj bodo iz vseh treh razredov (Vozilo, MotornoVozilo, Kolo)
poljeVozil[0] = new Vozilo();
poljeVozil[1] = new MotornoVozilo();
poljeVozil[2] = new MotornoVozilo(200);
poljeVozil[3] = new Kolo();
poljeVozil[4] = new Kolo(25);
//elementi polja se predstavijo
foreach (Vozilo vozilo in poljeVozil) Console.WriteLine(vozilo);    //Jaz sem vozilo.
                                                                    //Jaz sem motorno vozilo.
                                                                    //Jaz sem motorno vozilo.
                                                                    //Jaz sem kolo.
                                                                    //Jaz sem kolo.
```

Program ObjektnoProgramiranje_Dedovanje_Polimorfizem_PoljeObjektov

Objektno programiranje – polimorfizem /8

- Če želimo dostopati do specialnih elementov podrazreda, nam polimorfno obnašanje objektov ne pomaga, saj imajo posamezni podrazredi različne in drugače poimenovane specialnosti.
- Lahko pa nekaj naredimo le s tistimi objekti polja, ki so primerki izbranega razreda.
- POZOR: moramo preveriti, ali je trenutni element v polju res pripadnik izbranega podrazreda.
- Če ni in če tega ne preverimo, pride do izjeme in program se sesuje.

```
//dostop do specialnih elementov podrazreda
//POZOR: nevarno!!! (če ne preverimo tipa in se skličemo na podatek napačnega tipa se program
//sesuje)
    foreach (Vozilo vozilo in poljeVozil)
    {
        Console.WriteLine(vozilo.GetType().ToString());
        //if(vozilo.GetType() == typeof(MotornoVozilo))
        //if (vozilo.GetType().ToString() == "Polimorfizem_PoljeObjektov.MotornoVozilo")
        if (vozilo is MotornoVozilo)
        {
            Console.WriteLine(((MotornoVozilo)vozilo).PotrebnoGoriva());
        }
    }
```

Program ObjektnoProgramiranje_Dedovanje_Polimorfizem_PoljeObjektov

Objektno programiranje – kompleksna hierarhija razredov 1/2

- V kompleksnejših aplikacijah je pogosta bogata hierarhija razredov. Značilnosti:
  - veliko nivojev, iz izpeljanih razredov izpeljujemo nove razrede, itd.
  - C# podpira le enostavno dedovanje
    - vsak razred je izpeljan le iz enega, večkratno dedovanje ni mogoče
  - z base lahko dostopamo le do "očeta" do deda in višje navzgor v hierarhiji pa ne
  - dedovanje lahko tudi prepovemo, razred zapečatimo:
    - sealed class Razred
  - razredi visoko v hierarhijo pogosto niso namenjeni ustvarjanju realnih objektov, ampak so abstraktni, namenjeni le določanju delovanja realnih razredov, ki so nižje v hierarhiji

Objektno programiranje – kompleksna hierarhija razredov 2/2

- Če želimo dostopati do metod višje v hierarhiji (npr. iz Sina do Deda), si z base ne moremo pomagati, lahko pa:
  - če nimamo polimorfizma

```
(this as Ded).Izpisi();
```

  - če imamo polimorfizem (virtual v zgornjem razredu, v izpeljanih pa override)

```
var ptrDed = typeof(Ded).GetMethod("PolimorfniIzpisi").MethodHandle.GetFunctionPointer();
var DedIzpisi = (Action)Activator.CreateInstance(typeof(Action), this, ptrDed);
DedIzpisi();
```

Program ObjektnoProgramiranje_Dedovanje_KompleksnaHierahijaRazredov

Objektno programiranje – abstraktni razredi /1

- **Abstraktni razredi**
  - so posebna vrsta razredov, ki se uporabljajo samo kot osnova za dedovanje
  - ne omogočajo ustvarjanja objektov
  - **samo definirajo elemente** (podatke, lastnosti, metode), ki se uporabljajo za dedovanje, ter na ta način določajo delovanje izpeljanih razredov.
  - elementi abstraktnih razredov so lahko
    - abstraktni: samo določajo, katere lastnosti in metode morajo implementirati izpeljani razredi, podatki ne morejo biti abstraktni
    - konkretni: imajo polno funkcionalnost (podatki, lastnosti, metode)

- **Abstraktne metode in lastnosti**
  - Ustvariti jih je mogoče samo znotraj abstraktnega razreda.
  - Vsaka abstraktna metoda oz. lastnost **mora biti prekrita** v izpeljanih razredih, ki niso abstraktni (uporabimo abstract in override).
  - Razlika med virtualno in abstraktno metodo/lastnostjo je v tem, da virtualne ni nujno prekriti, abstraktno pa je.
  - Konstruktorji in destruktorji ne morejo biti abstraktni
  - Podatki ne morejo biti abstraktni.

Objektno programiranje – abstraktni razredi /2

- Deklaracija (uporabimo ključno besedo abstract)

```
abstract class GeometrijskoTelo
 {
        public abstract double Povrsina { get; }
        public abstract double IzracunajVolumen();
 }

  static void Main(string[] args)
 {
     GeometrijskoTelo mojeTelo = new GeometrijskoTelo();    // vrne napako!!!
 }
```

- Zgornja lastnost in metoda nimata vgrajene nobene programske logike. Ta mora biti vgrajena v izpeljanih razredih.
- Gre za abstraktno geometrijsko telo, metoda je abstraktna, ker ni mogoče natančno določiti formule za izračun površine.
- Tudi izpeljani razredi so lahko abstraktni. V tem primeru tudi ni potrebno vgraditi programske logike.

Objektno programiranje – abstraktni razredi /3

- Abstraktni razredi smejo vsebovati tudi neabstraktne, lastnosti, metode in podatke:

```
abstract class GeometrijskoTelo
{
    public abstract double Povrsina { get;}
    public abstract double IzracunajVolumen();

    private double visina;

    public double Visina
    {
        get { return visina;}
        set {visina = value;}
    }

    public double IzracunajRazmerjeMedVolumnomInPovrsino()
      {
            return IzracunajVolumen() / Povrsina;
    }
}
```

- Metoda *IzracunajRazmerjeMedVolumnomInPovrsino* je neabstraktna (konkretna).
- Program ObjektnoProgramiranje_Dedovanje_AbstraktniRazrediMetodeLastnosti

Objektno programiranje – abstraktni razredi /4

- Dedovanje iz abstraktnih razredov je identično dedovanju iz navadnih razredov z razliko, da abstraktne lastnosti in abstraktne metode **morajo** biti **implementirane** in **prekrite** (override) v izpeljanem **neabstraktnem** razredu. Npr.:

```
class Kocka : GeometrijskoTelo
  {
     public Kocka()
     { Visina = 1; }

     public Kocka(double v)
     { Visina = v; }

     public override double Povrsina
     { get{return Math.Pow(Visina, 2) * 6;} }

     public override double IzracunajVolumen()
     { return Math.Pow(Visina, 3); }

     public override string ToString()
     { return "Kocka";}
  }
```

Program ObjektnoProgramiranje_Dedovanje_AbstraktniRazrediMetodeLastnosti

Objektno programiranje – abstraktni razredi /5

- Podobno naredimo tudi za kroglo.

- Ilustrirajmo polimorfno obnašanje geometrijskih teles:

```
static void Main(string[] args)
   {
        GeometrijskoTelo[] poljeTeles = new GeometrijskoTelo[3];
        poljeTeles[0] = new Kocka();
        poljeTeles[1] = new Kocka(10);
        poljeTeles[2] = new Krogla();

        foreach (GeometrijskoTelo telo in poljeTeles)
        {
            Console.WriteLine(telo + " - višina: " + telo.Visina + " površina: " + telo.Povrsina
                    + " volumen: " + telo.IzracunajVolumen()
                    + " razmerje volumen:površina: " + telo.IzracunajRazmerjeMedVolumnomInPovrsino());
        }
    }
```

Program ObjektnoProgramiranje_Dedovanje_AbstraktniRazrediMetodeLastnosti

Vmesniki /1

- Vmesniki definirajo elemente, ki jih morajo imeti razredi, ki implementirajo vmesnik. Vmesniki definirajo, **kaj razredi morajo delati**, ne pa kako.

- Razred lahko deduje samo iz enega nadrejenega razreda, lahko pa **implementira več vmesnikov**.

- Vmesniki in abstraktni razredi ponujajo podobno funkcionalnost. Kdaj uporabljati ene in kdaj druge?
  - **abstraktni razredi** se uporabljajo takrat, ko je funkcionalnost enega ali več elementov fiksna in ko se ne zahteva dednosti iz več razredov.
  - **vmesniki** se uporabljajo v primerih, ko je znano, da en razred potrebuje vsebino več vmesnikov (se obnaša na več načinov).

Vmesniki /2

- Vmesnik tipično deklariramo izven vseh razredov, tako da jih lahko uporabljajo vsi razredi. V takem primeru **sme biti interface public ali internal** (če ne napišemo ničesar, je public, drugi nivoji dostopa niso dovoljeni).

- Vmesniki smejo vsebovati **le metode in lastnosti**, podatkov ne.
- Vse metode in lastnosti vmesnika morajo biti **brez navedbe nivoja dostopa**.
- Metode in lastnosti v vmesniku **samo deklariramo**, implementirati jih v vmesniku ni mogoče.
- Vsi razredi, ki implementirajo vmesnik, **morajo implementirati vse** lastnosti in metode vmesnika in sicer kot **javne** (public).

Vmesniki /3

Sintaksa je naslednja:

interface ime_vmesnika
{
}

Primer vmesnika:

```
interface IPlen  //lahko napišemo public ali internal, privzeto je public
{
    int HitrostBega { get; set; }   // lastnost
    void Bezi();  // metoda
}

interface IPlenilec
{
    int HitrostNapada { get; set; }
    void Napada(IPlen plen);
}

Program NapredneTemeOOP_Vmesniki
```

Vmesniki /4

```
class Macka : IPlenilec   // razred implementira vmesnik IPlenilec
{
    private int hitrostNapada;
    public int HitrostNapada  //kar je implementirano iz vmesnika mora biti public
    {
            get { return hitrostNapada;  }
            set { hitrostNapada = value; }
    }

    public void Napada(IPlen plen)
  {
            plen.Bezi();
            if (hitrostNapada > plen.HitrostBega)
                Console.WriteLine("Plen ulovljen");
            else Console.WriteLine("Plen zbežal");
    }

    public void Prede()
    {
            Console.WriteLine("Macka prede");
    }
}
```

Program NapredneTemeOOP_Vmesniki

Vmesniki /5

```
class Riba : IPlen
{
    private int hitrostBega;

    public int HitrostBega
    {
        get { return hitrostBega;  }
        set { hitrostBega = value; }
    }

    public void Bezi()
    {
        Console.WriteLine("Riba bezi");
    }
}
```

Program NapredneTemeOOP_Vmesniki

Vmesniki /6

Uporaba v glavnem programu:

```
static void Main(string[] args)
{
    Macka muca = new Macka();
    muca.HitrostNapada = 10;

    Riba nemo = new Riba();
    nemo.HitrostBega = 12;

    muca.Prede();
    muca.Napada(nemo);

    muca.HitrostNapada = 15;
    muca.Napada(nemo);
}
// izhod:
// Mačka prede
// Riba beži,  Plen zbežal
// Riba beži,  Plen ulovljen
```

Program NapredneTemeOOP_Vmesniki

Vmesniki /7

Primer implementacije več vmesnikov:

```
class Riba : IPlen, IPlenilec
{
     //implementacija Ribe kot plena ostane enaka
     …

     //implementacija Ribe kot plenilca
     private int hitrostNapada;
     public int HitrostNapada
      {
              get {return hitrostNapada;}
              set {hitrostNapada = value;}
     }
     public void Napada(IPlen plen)
      {
            //implementacija metode Napada je lahko v različnih razredih različna
             Console.WriteLine("hitrost napada {0}   hitrost bega {1}.", hitrostNapada, plen.HitrostBega);

             if (hitrostNapada > plen.HitrostBega) Console.WriteLine("Plen ulovljen");
             else Console.WriteLine("Plen zbežal");
     }
}
```

Program NapredneTemeOOP_Vmesniki2

Vmesniki /8

Uporaba v glavnem programu:

```
static void Main(string[] args)
{
      IPlenilec morskiPes = new Riba();
      morskiPes.HitrostNapada = 30;
     //morskiPes.HitrostBega = 25;   NAPAKA: morski pes je le plenilec
      IPlen tuna = new Riba();
      tuna.HitrostBega = 15;
     //tuna.HitrostNapada = 25;   NAPAKA: tuna je le plen
      morskiPes.Napada(tuna);

     //Nemo je Riba, lahko je plen ali plenilec.
      Riba nemo = new Riba();
      nemo.HitrostBega = 35;
      nemo.HitrostNapada = 35;
      morskiPes.Napada(nemo);
      nemo.Napada(tuna);
      //Morski pes je sicer res plenilec, a ker je Riba, lahko postane tudi plen
     ((IPlen)morskiPes).HitrostBega = morskiPes.HitrostNapada;
      nemo.Napada((IPlen)morskiPes);
}
// Plen ulovljen – morski pes ulovi tuno
// Plen zbežal – nemo je hitrejši od morskega psa
// Plen ulovljen – nemo ulovi tuno (jasno!)
// Plen ulovljen – nemo ulovi morskega psa (nemo je res faca)
```

Vmesniki /9

V predhodnih primerih so bili vmesniki implementirani na **implicitni** način. Pri **eksplicitnem** se pred implementacijo metode ali lastnosti napiše še ime vmesnika. To je pomembno predvsem, ko ima neka metoda v dveh vmesnikih enak podpis. Pri implementaciji v tem primeru ne smemo napisati nivoja skrivanja!

Npr.:

```
void IPlenilec.Izpis()
{ … }
void IPlen.Izpis()
{ … }
```

Primeri uporabe:

```
IPlenilec morskiPes = new Riba();
morskiPes.Izpis();  //jasno je, da se kliče izpis iz IPlenilec

Riba nemo = new Riba();
((IPlenilec)nemo).Izpis();  //treba je navesti, kateri od obeh izpisov naj se uporabi
```

Program NapredneTemeOOP_Vmesniki2

Vmesniki /10

- Če bi bili zadovoljni z enako metodo Izpisi za oba vmesnika, lahko ponovno uporabimo „navadno" **implicitno** metodo, ki pokrije oba primera. Vendar pa mora biti taka metoda **javna**, drugače kljub pravilnemu podpisu (ime, parametri in tip, ki ga vrača), metoda ne izpolnjuje zahtev vmesnika.

- V primeru dveh **eksplicitno** implementiranih metod se je treba zavedati, da velja ena za IPLen, druga za IPlenilec, za „navadno" ribo pa nobena od njiju!

```
Riba nemo = new Riba();
((IPlen)nemo).Izpis(); //OK
((IPlenilec)nemo).Izpis(); //OK
nemo.Izpis(); //NAPAKA
```

- Če rabimo tudi izpis za ribo, ko nanjo ne gledamo ne kot na plen in ne kot na plenilca ampak kot na „navadno" ribo, rabimo torej še tretjo metodo. npr:

```
public void Izpis()
{
    Console.WriteLine("Riba");
}
```

Program NapredneTemeOOP_Vmesniki2

Vmesniki /11

```
IPlenilec morskiPes = new Riba();
IPlen tuna = new Riba();
Riba nemo = new Riba();



morskiPes.Izpis();  //uporabi se metoda IPlenilec.Izpis();

tuna.Izpis(); //uporabi se metoda IPlen.Izpis();

((IPlen)nemo).Izpis(); //uporabi se metoda IPlen.Izpis();
nemo.Izpis();  //deluje le, če imamo metodo, ki ni implementirana
               // eksplicitno za nek vmesnik

((Riba)morskiPes).Izpis(); //deluje le, če imamo metodo, ki ni
                           //implementirana eksplicitno za nek vmesnik
```

Program NapredneTemeOOP_Vmesniki2

Vmesniki /12

- Tudi vmesniki omogočajo dedovanje.
  - V izpeljanem vmesniku ni treba ponavljati že definiranih elementov. V končnem razredu, ki je izpeljan iz izpeljanega vmesnika pa je treba implementirati elemente vseh vmesnikov, ne glede na nivo.

Npr. že znana vmesnika **IPlen** in **IPlenilec** izpeljemo iz vmesnika **IZival**:

```
interface IZival
{
        string Ime { get; set; }
}

interface IPlen : IZival
{ ... }
interface IPlenilec : IZival
{ ... }



Program NapredneTemeOOP_VmesnikiZDedovanjem
```

Vmesniki /13

- V razredih Macka in Riba, ki implementirata vmesnika IPlen in IPlenilec **moramo implementirati** še lastnost **Ime** iz vmesnika IZival**.**

- Zdaj se lahko na objekte iz razreda **Macka** in **Riba** sklicujeta referenci tipa bodisi **IPlenilec** bodisi **IZival**.
-
- Če se uporablja kot IZival, se lahko uporabi samo lastnost **Ime.**
- Če se uporablja kot IPlenilec, se lahko uporabljajo tako elementi iz IPlenilec kot iz IZival, ne pa tudi elementi vmesnika IPlen**.**

- Če se uporablja kot Riba, se lahko uporablja kot IZival, IPlen in IPlenilec**.**

Vmesniki /14

```
Riba nemo = new Riba();
IZival zival = nemo;
IPlenilec plenilec = nemo;
IPlen plen = nemo;

zival.Ime = "Nemo";            //OK
//zival.HitrostNapada = 10;  //napaka, dostopno le Ime
plenilec.HitrostNapada = 10; //OK
plenilec.Ime = "Nemo";         //OK
//plenilec.HitrostBega = 10; //napaka, plenilec nima hitrosti bega
nemo.HitrostNapada = 15;       //OK
nemo.HitrostBega = 15;         //OK
nemo.Ime = "Nemo";             //OK
```

Program NapredneTemeOOP_VmesnikiZDedovanjem

Vmesniki /15

- **Razred** lahko hkrati implementira **nič ali več vmesnikov** in deduje po **nič ali enem razredu**.

- Če je lastnost ali metoda, ki jo zahteva vmesnik implementirana že v nadrazredu, je v izpeljanem razredu **ni potrebno implementirati**, saj je podedovana. Če želimo, pa jo seveda **smemo** implementirati:

  - Uporabimo lahko **skrivanje**. Če se v izpeljanem razredu ponovi ime elementa, ki je že določeno, je potrebno uporabiti ključno besedo **new** (drugače dobimo prevajalnikovo opozorilo).
  - Možno je tudi **prekrivanje** (uporabimo **virtual** in **override**).

Vmesniki /16

```
abstract class Zival
   {
      protected string ime;
      …
      public virtual string Ime
      {
         get { return ime; }
         set { ime = value; }
      }
      …
   }

class Macka: Zival,  IPlenilec   // razred deduje od Zival in implementira vmesnik IPlenilec (torej tudi IZival )
                                 //lastnosti Ime ni potrebno implementirati, ker je podedovana iz Zival
   { …
      //Vmesnik IZival zahteva implementacijo lastnosti Ime, vendar je ta lastnost podedovana
      //od razreda Zival, tako da je tu ni nujno treba implementirati
    }
```

Program NapredneTemeOOP_VmesnikiRazredZDedovanjemInVmesniki

Vmesniki /17

```
class Riba : Zival, IPlen, IPlenilec
// razred deduje od Zival (razred mora biti naveden prvi, pred vsemi vmesniki)
//in implementira vmesnika IPlen in IPlenilec (posredno tudi IZival )
//lastnosti Ime ni potrebno implementirati, ker je podedovana iz Zival, a se odločimo da jo
//bomo prekrili
   {
      ...
      public override string Ime
      {
          get { return "Ne znam povedat.";  }
          //set ne implementiramo, bo kar podedovan iz Zival, je pa neuporaben, ker get zmeraj
          //vrne isto
      }
   }
```


Program NapredneTemeOOP_VmesnikiRazredZDedovanjemInVmesniki

Vmesniki /18

```
Riba nemo = new Riba();
IZival zival = nemo;
IPlenilec plenilec = nemo;
IPlen plen = nemo;

Macka muca = new Macka();
 muca.Ime = "Muca";
 Console.WriteLine(muca.Ime);  //Uporabi se metoda, podedovana iz Zival, izpis: „Muca";
 Console.WriteLine("");

//Zaradi prekrivanja (polimorfizem!) se v vseh spodnjih primerih uporabi najbolj specializirana
//metoda, to je metoda iz Riba.
 zival.Ime = "Nemo";
 Console.WriteLine(zival.Ime);   //izpis: „Ne znam povedat"
 Console.WriteLine(nemo.Ime);  //izpis: „Ne znam povedat"
 Console.WriteLine(plen.Ime);    //izpis: „Ne znam povedat"
 Console.WriteLine(plenilec.Ime); //izpis: „Ne znam povedat"
Console.WriteLine(((Zival)nemo).Ime); //izpis: „Ne znam povedat" zaradi polimorfizma
 plenilec.HitrostNapada = 10;
```

Program NapredneTemeOOP_VmesnikiRazredZDedovanjemInVmesniki

Vmesniki /19

- Če nadrazred implementira nek vmesnik, izpeljani razred **deduje tudi to**, tako da pri izpeljanem razredu ni nujno navesti vmesnika

  - če je Riba IPlen in IPlenilec in
  - če je MorskiPes izpeljan iz Riba,
  - je MorskiPes avtomatično tudi IPlen in IPlenilec (posredno tudi IZival)

Vmesniki /20

Deklaracija razreda:

```
//ker deduje po razedu Riba, MorskiPes podeuje tudi implementacije vmesnikov, se pravi,
//da je ne le //Zival, ampak tudi IZival, IPlen in IPlenilec
    class MorskiPes: Riba
    {
        public bool ljudemNevaren;
    }
```

Uporaba:

```
//MorskiPes od Riba podeuje tudi vmesnike,
//je torej IPlen in IPlenilec (posredno tudi IZIval)
IPlenilec piki = new MorskiPes();
piki.Ime = "Piki";
Console.WriteLine(piki.Ime);

piki.Napada(plen);
nemo.Napada((IPlen)piki);
```

Program NapredneTemeOOP_VmesnikiRazredZDedovanjemInVmesniki

Sistemski vmesniki – sortiranje /1

- Če polje vsebuje podatke vgrajenih tipov, sortiranje z uporabo **Array.Sort(ime polja)** oz. **seznam.Sort()** deluje brez vsakršnih predpriprav.

- To je mogoče, ker jih zna med seboj primerjati z objektno metodo **CompareTo.**

- Če želimo **sortirati polje objektov našega lastnega razreda**, mora razred implementirati :
  - vmesnik **IComparable**
  - posledično tudi objektno metodo **CompareTo(object)**

- Program NapredneTemeOOP_VmesnikiSortiranje

Sistemski vmesniki – sortiranje /2

```
public class MojRazred : IComparable
  {
     public int stev;
     public string ime;

     public int CompareTo(Object o)  //definiramo privzeti način primerjanja
     {
        MojRazred drugi = o as MojRazred; //preverimo, ali je objekt primerek pravega razreda
        if (drugi != null)  //o je primerek razreda MojRazred
        {
           int rezultat = ime.CompareTo(drugi.ime);
           if (rezultat == 0) //imeni sta enaki, gledamo še številke
           {
              rezultat = this.stev - drugi.stev;
           }
           return rezultat;
        }
        else  //o ni primerek razreda MojRazred
           throw new ArgumentException("Objekt ni primerek razreda MojRazred.");
     }
  }
```

Uporaba: `Array.Sort(imePolja); za polje oz. seznam.Sort(); za List in ArrayList`

- Program NapredneTemeOOP_VmesnikiSortiranje

Sistemski vmesniki – sortiranje /3

- Lahko pa imamo več primerjalnikov za sortiranje po različnih kriterijih:
  - primerjalnik je samostojen razred, ki implementira vmesnik **IComparer**
    - vmesnik IComparer zahteva implementacijo metode
    - **Compare(Razred, Razred)**
  - primerjanje izvedemo s
    - **Array.Sort(ImePolja, new primerjalnik)**
    - **seznam.Sort(new primerjalnik)** za List (z ArrayList ne deluje)

```
public class PrimerjajPoStevilki : IComparer<MojRazred>
    {
        public int Compare(MojRazred o1, MojRazred o2)
        {
            //ni treba preveriti, ali sta objekta pravega razreda
            //to je določeno že z argumenti metode
            return o1.stev - o2.stev;
        }
    }
```

Uporaba: Array.Sort(ImePolja, new PrimerjajPoStevilki()); za polja
seznam.Sort(new PrimerjajPoStevilki()); za List

- Program NapredneTemeOOP_VmesnikiSortiranje

Strukture /1

- Strukture so podobne razredom s to razliko, da so primerki strukture spremenljivke ***vrednostnega tipa*** **(shranijo se na sklad).**

- Strukture so lahko sestavljene iz **podatkov**, **metod** in **lastnosti**. Vsebujejo lahko tudi
  - **konstruktorje** – možno je definirati več različnih konstruktorjev. Privzeti konstruktor brez parametrov ne more biti definiran. Privzeti konstruktor je impliciten za vse strukture in se ga ne more obiti.
  - **konstante** – konstante in naštevanja (enumerations) se lahko definirajo v strukturah.
  - **polja** – polja se lahko uporabljajo znotraj struktur vendar morajo biti inicializirana s konstruktorji.
  - **operatorje** – v strukturah je možno preobtežiti operatorje.
  - **dogodke** – strukture lahko vsebujejo dogodke ter jim na ta način omogočajo poročanje v primeru neke aktivnosti ali doseženega stanja.

Strukture /2

- Strukture **ne omogočajo dedovanja**! Elementi torej ne morejo biti tipa **virtual**, **protected** ali **abstract**.
- Strukture **lahko implementirajo enega ali več vmesnikov**.
- Strukture **ne uporabljajo destruktorjev**. Zakaj?
- Uporabljajo se, ker **so lahko bolj učinkovite**, (v primeru ko se uporablja struktura z majhnim številom lastnosti vendar v velikem številu). So pa lahko tudi **manj učinkovite** (pri prenosu velikih struktur v metode).
- V primeru objekta je potrebno:
  - rezervirati prostor v spominu za podatke in za referenco na te podatke. Pri strukturah to ni potrebno – prostor samo za podatke (**struktura bolj učinkovita**).
  - pri prenosu parametra v metodo prenesti le referenco. Pri strukturah se prenese kopija celotne strukture (**večja kot je struktura, bolj neučinkovit je prenos**).
- Te razlike so majhne vendar je lahko pri uporabi tisočih ali milijonov operacij kumulativen efekt znaten.

Strukture /3

Sintaksa:

struct ime_strukture {} oz.

struct ime_strukture : ime_vmesnika1, ime_vmesnika2, ... Ime_vmesnikaN {}

```
struct Tocka
    {
        public int x, y; // public le zato, da bomo lahko strukturo ustvarili brez new

       public  Tocka(int x, int y)
        {   this.x = x;    this.y = y;  }

        public int X
        {  get { return x; }     set { x = value; }   }


        public int Y
        {  get { return y; }     set { y = value;  }  }


        public double RazdaljaDoIzhodisca()
        {  return Math.Sqrt(x * x + y * y);  }
    }

Program NapredneTemeOOP_Strukture
```

Strukture /4

```
struct Daljica
   {
      private Tocka t1, t2;

      public Daljica(int x1, int y1, int x2, int y2)
      {
         t1 = new Tocka(x1, y1);
         t2 = new Tocka(x2, y2);
      }

     public Daljica(Tocka t1, Tocka t2)
     {
         //Če ustvarimo novo točko z new, deluje tudi, če so koordinate v
         // strukturi Točka privatne.
         this.t1 = new Tocka(t1.X, t1.Y);
         //Če ne uporabimo new, ne moremo do posameznih podatkov, dokler
         //niso inicializirani!
         //Ker so podatki znotraj točke javni, lahko dostopamo direktno do
         //njih npr. z this.t2.x Če bi uporabili this.t2.X namesto this.t2.x,
         //še zmeraj ne bi delovalo!
         this.t2.x = t2.X;
         this.t2.y = t2.Y;
         //lahko pa bi(tudi pri privatnih podatkih) nastavili podatke naenkrat
         //saj dostopamo do podatkov daljice (cele točke naenkrat)
          this.t2 = t2;
     }
```

Program NapredneTemeOOP_Strukture

Strukture /5

...nadaljevanje...

```
public void NastaviT1(int x, int y)
      {  t1.X = x;  t1.Y = y;     }

       public void IzpisiT1()
       {  Console.WriteLine("x1 = {0},  y1 = {1}", t1.X, t1.Y);   }

      // … (analogno za T2)

       public double Dolzina()
       {
          return Math.Sqrt((t1.X - t2.X) * (t1.X - t2.X) + (t1.Y - t2.Y) * (t1.Y - t2.Y));
       }
   }
```

Program NapredneTemeOOP_Strukture

Strukture /6

Spremenljivka se iz strukture lahko ustvari na dva načina:

- z uporabo **new** (uporabi se konstruktor, podatki so inicializirani)
- brez uporabe **new** (podatki niso inicializirani)

```
//brez uporabe new
Tocka t1;
//Napaka, ker t1.x in t1.y še nimata vrednosti
//t1.X = 5;  //Napaka
//Console.WriteLine("(X={0}, Y={1})", t1.X, t1.Y); //Napaka
//Console.WriteLine("Length: {0}", t1.RazdaljaDoIzhodisca()); //Napaka

//Če podatki ne bi  bili javni, jim sploh ne bi mogli nastaviti vrednosti
//in t1 bi bila neuporabna!

//Ko so VSI podatki inicializarani pa lahko uporabimo lastnosti in metode.
//POZOR: VSI podatki
t1.x = 3;  //Možno le, ker sta x in y javna podatka.
t1.y = 4;  //Možno le, ker sta x in y javna podatka.
Console.WriteLine("(X={0}, Y={1})", t1.X, t1.Y);
Console.WriteLine("Length: {0}", t1.RazdaljaDoIzhodisca());
```

Program NapredneTemeOOP_Strukture

Strukture /7

Z uporabo **new**:

```
//z uporabo new (uporabi se konstruktor, vrednosti so inicializirane)
Tocka t2 = new Tocka();
t2.X = 30;
Console.WriteLine("(X={0}, Y={1})", t2.X, t2.Y);
Console.WriteLine("Length: {0}", t2.RazdaljaDoIzhodisca());

Tocka t3 = new Tocka(1, 2);
Console.WriteLine("(X={0}, Y={1})", t3.X, t3.Y);
Console.WriteLine("Length: {0}", t3.RazdaljaDoIzhodisca());
Console.WriteLine();

 //še Daljica z new (brez new ne bo šlo, točki v daljici nista javni)
 Daljica d1 = new Daljica(1, 1, 3, 5);
 Daljica d2 = new Daljica(new Tocka(1,1), new Tocka(3,5));
 d1.IzpisiT1();
 d2.IzpisiT2();
 Console.WriteLine(d1.Dolzina());
```

Program NapredneTemeOOP_Strukture

Indekserji /1

- **Indekserji** (angl. indexers) so posebna vrsta lastnosti
  - omogočajo delo z razredi, kot da so polja
  - objekti lahko pri prikazovanju več vrednosti uporabljajo notacijo kot pri poljih

- Sintaksa:

```
public tip_podatka this[tip_indeksa ime _indeksa]
{
    get {...}
    set {...}
}
```

- *get* je obvezen – mora vračati vrednost tipa tip-podatka *set* ni obvezen.

Indekserji /2

```
class Dalmatinec
   {
      private string ime;
      private int stPik;
      private double teza;
      public const int stPodatkov = 3;

      public string this[int indeks]
      {
         get
         {
           if (indeks == 0) return ime;
           else if (indeks == 1) return stPik.ToString();
                else  return teza.ToString();
         }
        set
         {
           if (indeks == 0) ime = value;
           else if (indeks == 1)  stPik = int.Parse(value);
           else teza = double.Parse(value);
         }
      }
  }
```


Program NapredneTemeOOP_Indekserji

Indekserji /3

```
Dalmatinec d = new Dalmatinec();

for (int i = 0; i < Dalmatinec.stPodatkov; i++)
{
    Console.Write("d[{0}]: ", i);
    d[i] = Console.ReadLine();
}
for (int i = 0; i < Dalmatinec.stPodatkov; i++)
{
    Console.WriteLine(d[i]);
}
```

IZZIV: implementiraj potrebne vmesnike in kodo, da za izpis mogoče uporabit zanko foreach.

Program NapredneTemeOOP_IndekserjiPreoblagenje

Indekserji /4

- Indekserje pa v praksi bolj pogosto uporabimo, če se v razredu skriva neka zbirka. Do njenih podatkov lahko prek lastnosti dostopamo takole:

```
class MojSeznam
{
    private string[] podatki;
    private int dolzinaPolja;

    public string[] Podatki
    {
        get
        {
            return podatki;
        }
        set
        {
            podatki = value;
        }
    }
}
```


Program NapredneTemeOOP_IndekserjiPreoblagenje

Indekserji /5

- Lahko pa napišemo indekser:

```
class MojSeznam
{
        private string[] podatki;
        private int dolzinaPolja;

        public string this[int indeks]
        {
            get
            {
                return podatki[indeks];
            }
            set
            {
                podatki[indeks] = value;
            }
        }
}
```

Program NapredneTemeOOP_IndekserjiPreoblagenje

Indekserji /6

- Primer uporabe:

```
int velikost = 5;
MojSeznam mojInd = new MojSeznam(velikost);
//uporaba " navadne“ lastnosti
mojInd.Podatki[1] = "nekaj";
//uporaba "indekserja"
mojInd[3] = "še nekaj";
```

Program NapredneTemeOOP_IndekserjiPreoblaganje

Indekserji /7

- Nad nekim razredom lahko imamo več indekserjev (indekser preobložimo), prav tako ni nujno, da morajo biti indeksi indekserja cela števila. Napišimo še dva indekserja:

  - dodajmo razredu še dve polji: *imena* in *priimki*
  - preobložimo indekser tako, da bo imel še en argument (*ime* ali *priimek*), ki bo določal, ali se sklicujemo na polje imena ali priimki
  - če je ta argument napačen, vržimo izjemo

  - preobložimo indekser tako, da bo indeks tipa string
  - get bo izpisal kolikokrat se indeks pojavi v podatkih objekta
  - set bo vse pojavitve indeksa zamenjal z vrednostjo na desni strani enačaja

Program NapredneTemeOOP_IndekserjiPreoblaganje

Indekserji /8

```
public string this[string polje, int indeks]
{
   get
   {
      if (polje.ToLower() == "ime") return imena[indeks];
       else if (polje.ToLower() == "priimek") return priimki[indeks];
            else throw new ArgumentException ("Napaka pri prvem indeksu");
    }
    set
    {
       if (polje.ToLower() == "ime") imena[indeks] = value;
        else if (polje.ToLower() == "priimek") priimki[indeks] = value;
            else throw new ArgumentException("Napaka pri prvem indeksu");
    }
}
```

- Uporaba:

```
mojInd["ime", 1] = "Piki";
mojInd["priimek", 1] = "Jakob";
```

Program NapredneTemeOOP_IndekserjiPreoblaganje

Indekserji /9

```
public string this[string vnos]
    {
        get  //vrne število pojavitev za vnos
        {
           int count = 0;
           for (int i = 0; i < dolzinaPolja; i++)
           {
              if (podatki[i] == vnos) count++;
           }
           return count.ToString();
        }
        set  //vse pojavitve vnos zamenja z zahtevanim
        {
            for (int i = 0; i < dolzinaPolja; i++)
            {
                 if (podatki[i] == vnos) podatki[i] = value;
            }
         }
  }
```

- Uporaba:

```
mojInd["prazno"] = "nič"; //zamenja vse "prazno" z  "nič"
Console.WriteLine("Število pojavitev vrednosti \"nič\": {0}", mojInd["nič"]);
```

Program NapredneTemeOOP_IndekserjiPreoblaganje