Fakulteta za informacijske študije v Novem mestu

Borut Lužar

Programiranje

Pregled snovi

[Delovna verzija, 2. november 2024]

Novo mesto, november 2024

Kazalo

Predgovor

Pred vami je kratka zbirka vprašanj in nalog, ki naj vam služijo kot orientacija pri učenju za izpit pri predmetu Programiranje. Pri predmetu namreč (še) nimamo pripravljenih celovitih zapiskov, ki bi pokrivali vso snov, katero bomo obdelali na predavanjih. Dele zapiskov, ki so že in še bodo pripravljeni, bom sproti vključeval v nove verzije tega dokumenta. Snov na predavanjih sicer sledi kodi, ki jo imate v ta namen pripravljeno v GitHub repozitoriju, razlaga posameznih konceptov pa je tudi dobro opisana v dokumentaciji in drugi literaturi, ki jo uporabljamo pri predavanjih.

Predmet pokriva naprednejše koncepte v programiranju, v pregled snovi pa sem dodal tudi začetne tematike, da se lahko tisti, ki so malo programsko zarjaveli, dodatno pripravijo. Priporočam vam, da ob učenju dane naloge, ki jih ne bomo rešili na vajah, poskusite rešiti, prav tako pa odgovorite na zastavljena vprašanja.

Borut Lužar

1 Osnovni pojmi

Delamo z Visual Studio Community 2022 ter programskim jezikom C#.

1.1 Delo z razvojnim okoljem

1.1.1 Vprašanja

1. Kako ustvarimo nov projekt?
2. Katere tipe projektov imamo na voljo?
3. Kako dodamo nov projekt v trenutno rešitev?
4. Kako poiščemo definicijo metode, ki jo kličemo?
5. Kako program ustavimo na izbrani točki?
6. Kako poiščemo vir v dokumentaciji za izbrano rezervirano besedo ali operator?
7. Kako nastavimo pogojno ustavitveno točko (*conditional breakpoint*)?
8. Kako poženemo in ustavimo program?
9. Kako določimo, kateri projekt želimo pognati?
10. Kako preimenujemo izbrano spremenljivko?
11. Kako se hitro premikamo po kodi (skok v izbrano funkcijo in skok nazaj na prejšnjo točko)?

1.1.2 Naloge

Naloga 1.1.1. V Visual Studiu ustvarite nov projekt tipa Console Application in zaženite program HelloWorld!

Naloga 1.1.2. V rešitev iz prejšnje naloge dodajte še en projekt tipa Class Library, vanj dodajte razred z neko metodo. Nato novi projekt dodajte kot referenco v prvega ter v programu HelloWorld pokličite metodo iz drugega projekta.

Naloga 1.1.3. Na klic metode iz prejšnje naloge nastavite *breakpoint* in spremljajte izvajanje programa vrstico po vrstico, vključno s skokom v metodo iz razredne knjižnice. Uporabite bližnjice F10 (*step over*) in F11 (*step into*).

1.2 Izbirni stavki

1.2.1 Vprašanja

1. Sintaksa stavka `if`.
2. Sintaksa stavka `switch`.
3. Katere tipe vrednosti sprejema stavek `switch` v pogoju?
4. Katera implementacija je hitrejša v primerih večkratnih možnosti `if-else if` ali `switch`?
5. Kako v stavku `switch` učinkovito izvedemo enako kodo za več različnih možnosti (*cases*)?

1.2.2 Naloge

Naloga 1.2.1. Zapišite metodo, ki kot parameter dobi dan v tednu kot celo število, nato pa izpiše dan z ustreznim imenom. Funkcijo implementirajte na dva načina: z uporabo `if` ter `switch`.

Naloga 1.2.2. Zapišite funkcijo, ki uporablja stavek `if` za izpis uporabe primernega oblačila glede na zunanjo temperaturo. Uporabite vsaj štiri različne tipe oblačil. Razmislite, ali bi nalogo lahko rešili z uporabo stavka `switch`.

Naloga 1.2.3. Zapišite funkcijo, ki uporablja stavek `switch`, da se odloči za izpis števila dni v danem mesecu. Predpostavite lahko, da nismo v prestopnem letu. Možnosti, ki vrnejo enako vrednost, združite.

1.3 Zanke

1.3.1 Vprašanja

1. Katere tipe zank poznamo v C#?
2. Sintaksa zanke `for`.
3. Sintaksa zanke `foreach`.
4. Sintaksa zanke `while`.
5. Sintaksa zanke `do`.
6. V katerih primerih uporabimo stavek `break`?
7. V katerih primerih uporabimo stavek `continue`?
8. Na primeru opišite delovanje stavka `yield return`.

1.3.2 Naloge

Naloga 1.3.1. Zapišite metodo, ki izpisuje trenutni čas, dokler produkt ur, minut in sekund ni deljiv s 17. Uporabite lastnost `Now` razreda `DateTime`.

Naloga 1.3.2. Zapišite metodo, ki v danem seznamu realnih števil izbriše vse vrednosti, ki so manjše od začetnega povprečja. Naredite dve implementaciji: eno z zanko `for` in eno z zanko `foreach`. Na kaj morate paziti?

Naloga 1.3.3. Zapišite metodo, ki z uporabo zanke `do` izpisuje večkratnike danega naravnega števila, dokler vrednost večkratnika ne preseže danega parametra.

Naloga 1.3.4. Zapišite metodo, ki izpiše prvih k praštevil, kjer je k parameter. V funkciji uporabite vsaj en stavek `break` in en stavek `continue`.

Naloga 1.3.5. Na spletu poiščite postopek za branje izvorne kode html strani, podane z url naslovom, s C#. Nato zapišite metodo, ki sproti izpisuje hiper-povezave, ki se na izbrani strani pojavljajo, pri tem pa naj izvorno kodo bere druga metoda, ki za vračanje sprotnih hiper-povezav uporablja stavek `yield return`.

1.4 Metode

1.4.1 Vprašanja

1. Kako definiramo novo metodo?
2. Kaj je podpis (ang. *signature*) metode?
3. Kdaj metoda konča izvajanje?
4. Ali lahko uporabimo stavek `return` v metodah, ki ne vračajo ničesar (so `void`)?
5. Na katere načine lahko metoda vrne več različnih vrednosti? Naštejte vsaj tri možnosti.
6. V katerem primeru bi pri parametru metode uporabili določilo `ref`?
7. Kdaj je smiselno uporabiti določilo `ref` za parameter sklicnega tipa?

1.4.2 Naloge

Naloga 1.4.1. Zapišite metodo, ki kot parameter dobi dve celi števili in vrne njun najmanjši skupni večkratnik ter največji skupni delitelj.

Naloga 1.4.2. Zapišite metodo z enakim imenom kot v prejšnji nalogi, pri čemer naj ima še dodaten neobvezen parameter tipa `bool`, ki odloči, če želimo poiskati tudi največji skupni delitelj ali le najmanjši skupni večkratnik. Razmislite tudi, kaj naj metoda vrača.

Naloga 1.4.3. Zapišite metodo, ki kot sklicni (oz. `ref`) parameter prejme seznam realnih števil. Vrednosti v seznamu predstavljajo verjetnost padavin v zaporednih dneh, ki so bile izračunane in se v seznam zapisale iz zunanjega vira. Naloga metode je preveriti, če so vrednosti dopustne. V primeru, da je med njimi kakšna vrednost večja kot 1, metoda seznam izprazni oziroma mu priredi nov, prazen seznam. V nasprotnem primeru seznama ne spreminja.

1.5 Enumeracije

1.5.1 Vprašanja

1. Kaj je enumeracija oziroma enumeracijski tip? [Arh, Q15]
2. Kakšne tipe vrednosti sprejemajo konstante v enumeraciji?
3. Kako spremenimo tip vrednosti enumeracije?
4. Ali so izbrane vrednosti konstant lahko poljubne ali morajo biti v zaporedju?
5. Kako izpišemo vse konstante izbrane enumeracije?
6. Kako iz dane vrednosti dobimo ime konstante?
7. Kako iz dane konstante dobimo njeno vrednost?

1.5.2 Naloge

Naloga 1.5.1. Zapišite enumeracijo za imena mesecev, ki ima vrednosti tipa `byte`.

Naloga 1.5.2. Zapišite enumeracijo za imena ocen na fakulteti. Nato pripravite funkcijo, ki profesorja vpraša, kakšno oceno želi dati študentu na izpitu ter mu dajte na voljo ustrezne izbire.

Naloga 1.5.3. Zapišite enumeracijo za pet možnosti izbire pri študentskih anketah. Nekaj primerov vprašanj pripravite v seznamu, nato pa ob zagonu programa od uporabnika zahtevajte vnos ustreznega odgovora na vsako izmed vprašanj. Dodatno lahko posameznim vprašanjem določite tip možnega odgovora (ustvarite poseben objekt) in glede na tip ponudite odgovore oziroma primeren tip enumeracije za izbiro odgovora.

1.6 Vrednostni in sklicni tipi

1.6.1 Vprašanja

1. Kakšna je razlika med vrednostnimi in sklicnimi tipi? [Arh, Q11]
2. Kakšna je razlika med razredom (ang. `class`) in struktom (ang. `struct`)? [Arh, Q12]
3. Kaj je *boxing* in kaj *unboxing*? [Arh, Q13]

4. Zapišite primer *boxinga* za spremenljivko tipo `float`.

5. Kako se spremeni vrednost spremenljivke sklicnega tipa `refType`, ki kaže na spremenljivko vrednostnega tipa `valType`, če spremenljivki `valType` spremenimo vrednost?

1.6.2 Naloge

Naloga 1.6.1. Zamislite si primer razreda in primer strukta ter oba implementirajte.

Naloga 1.6.2. Definirajte tri spremenljivke vrednostnih tipov, na vsaki izmed njih izvedite *boxing*, spremenite vrednosti prvim trem spremenljivkam ter preverite vrednosti referenčnih spremenljivk.

Naloga 1.6.3. Spremenljivkam sklicnih tipov iz zgornje naloge priredite spremenljivke vrednostnih tipov, spremenite vrednosti prvim ter preverite vrednosti drugih.

1.7 Delo z datotekami

1.7.1 Vprašanja

1. Kateri objekt uporabljamo za pisanje v datoteko?

2. Kateri objekt uporabljamo za branje datoteke?

3. Zapišite preprosto metodo, ki ustvari novo datoteko in vanjo zapiše stavek, podan kot parameter metode.

4. Zapišite preprosto metodo, ki v datoteki, podani s potjo, prešteje število znakov.

5. Katero lastnost uporabimo za preverjanje, če smo datoteko že prebrali do konca?

6. S katero metodo zapišemo besedilo v datoteko, ne da bi jo zaprli?

1.7.2 Naloge

Naloga 1.7.1. Zapišite funkcijo, ki bo prebrala dano datoteko in v novo datoteko zapisala vse besede, ki se pojavijo v prvi, vendar v abecednem vrstnem redu.

Naloga 1.7.2. Zapišite funkcijo, ki v dani datoteki vse samoglasnike "zamakne" za ena, npr. 'a' gre v 'e', 'e' gre v 'i' in tako dalje.

Naloga 1.7.3. Zapišite funkcijo, ki beleži neke dogodke v datoteko vsako sekundo (uporabite funkcijo `Thread.Sleep(1000)`). Zapisuje jih naj vsaj minuto, ob vsakem zapisu pa naj se podatki že pojavijo v datoteki, tako da lahko že med izvajanjem programa preverite vsebino datoteke. Implementacijo naredite na dva načina - brez zapiranja datoteke in z zapiranjem datoteke. Na kaj morate biti pozorni v drugi verziji naloge?

1.8 Razred `Random`

1.8.1 Vprašanja

1. Na kakšen način pridobimo naključno izbrano število iz danega intervala?
2. Kako določimo, da so "naključno" izbrana vedno enaka števila?
3. Zakaj je uporaba semena (ang. *seed*) koristna?

1.8.2 Naloge

Naloga 1.8.1. Zapišite funkcijo, ki bo izžrebala številke za igro Loto. Vrne naj torej sedem naključno izbranih različnih števil med 1 in 39 ter dodatno število. Napišite še programček, ki za izbranih 7 števil funkcijo izvede tolikokrat, dokler vaša kombinacija ni izžrebana in izpišite število poskusov. Na kaj morate biti pozorni pri štetju poskusov?

Naloga 1.8.2. Zapišite funkcijo, ki za dano število poskusov simulira met igralne kocke (števila med 1 in 6). Beležite, kolikokrat padejo posamezna števila in ugotovite, če se njihova frekvenca s povečevanjem števila poskusov ujema oziroma približuje verjetnosti padca posameznega števila.

Naloga 1.8.3. Zapišite funkcijo, ki naključno izbira realna števila iz intervala med 0 in 1. Ugotovite povprečno vrednost števil, ki jih zadanete v milijonu poskusov.

2 Koncepti objektnega programiranja

V tem poglavju se bomo posvetili konceptom objektno orientiranega programiranja (OOP). Osnovni element OOP je *objekt* kot samostojna enota s svojimi lastnostmi in funkcionalnostmi. Objekte opišemo oziroma definiramo v *razredih*.

2.1 Osnovno o objektih

Do sedaj smo se ukvarjali s spremenljivkami, ki so večinoma nosile neke enostavne vrednosti. Srečali pa smo se tudi že s spremenljivkami, ki so svoje vrednosti lahko analizirale oziroma nam povedale nekaj več o njihovih lastnostih. Na primer, spremenljivka tipa `string` zna povedati, kako dolg je niz, ki ga ima za vrednost.

```
string cityName = "Amsterdam";
int lengthOfCityName = cityName.Length;
```

Prav tako lahko spremenljivka tipa `string` z metodo `IndexOf` preveri, če njena vrednost vsebuje dani znak.

```
char chrToSearch = 't';
if(cityName.IndexOf(chrToSearch) != -1)
{
  Console.WriteLine($"Niz {cityName} " +
      $"vsebuje znak '{chrToSearch}'!");
}
```

Objektno orientirani programski jeziki nam omogočijo, da si definiramo svoje tipe spremenljivk, pa tudi lastnosti in metode, ki jih za upravljanje z vrednostmi potrebujejo. Definicije za posamezni tip zapišemo v *razred*.

Razrede smo do sedaj že imeli definirane, saj morajo biti metode umeščene v razrede. Blok, ki oklepa definicijo razreda, ima v glavi rezervirano besedo `class` in *ime razreda*, ki bo obenem pomenil tudi naziv tipa spremenljivke, katerega bo razred definiral. Po konvenciji imena razredov vedno pišemo z veliko začetnico.

Definirajmo razred, ki bo opisoval objekt *miza*. Zanimajo nas lastnosti material, število nog in površina. Zapisali jih bomo kar v spremenljivke razreda (pravimo jim *polja*), vendar brez določila `static`, kot smo bili navajeni pri metodah.

```
public class Table
{
  public string material = "";
  public int numLegs;
  public double area;
}
```

K spremenljivkam in v glavo razreda smo dodali še določilo `public`, s katerim povemo, da dopuščamo dostop do njih tudi kodi, ki ni v razredu `Table`, saj bomo le tako lahko nastavili ustrezne vrednosti spremenljivkam, ki jih bomo kreirali po navodilih razreda `Table`.

Spremenljivko tipa `Table` kreiramo (v metodi `Main`) na enak način, kot smo kreirali na primer sezname.

```
Table kitchenTable = new Table();
```

Ukaz na desni strani `new Table()` je klic *konstruktorja*. Konstruktor je element, podoben metodi, ki ustvari *nov objekt* oziroma novo *instanco* razreda. Konstruktorje definiramo v razredu. Glava konstruktorja je podobna kot pri metodah:

[določilo zasebnosti] [ime razreda] ([parametri])

Razlika je v tem, da določila statičnosti nikoli ne uporabimo[1], prav tako ne zapisujemo tipa rezultata (saj konstruktor naredi objekt tipa razreda, v katerem je definiran) in ime konstruktorja je vedno enako imenu razreda (tako namreč ob klicu konstruktorja vemo, katero instanco objekta bo kreiral). Konstruktor lahko dobi tudi parametre, ki jih uporabimo v njegovi definiciji za nastavljanje lastnosti objekta. Privzeto ima vsak razred že definiran konstruktor brez parametrov.

Definirajmo razred `Book` s konstruktorjem, v katerem nastavimo obe lastnosti knjige.

```
public class Book
{
  public Book(int pages, int year)
  {
    numPages = pages;
    yearPublished = year;
  }

  public int numPages;
  public int yearPublished;
}
```

V metodi `Main` knjigo s 100 stranmi izdano leta 2000 kreiramo takole:

```
Book myBook = new Book(100, 2000);
```

Če ob kreiranju knjige ne želimo podati števila strani, si definiramo še en konstruktor, ki dobi le podatek o letu izdaje.

```
public class Book
{
  ...

  public Book(int year)
  {
    yearPublished = year;
  }

  ...
}
```

Tudi klic drugega konstruktorja je pričakovan:

```
Book newBook = new Book(2020);
```

[1] Razen v zelo posebnih primerih, ki ne bodo del teh zapiskov. Več o statičnih razredih si lahko preberete tukaj, več o statičnih konstruktorjih pa tukaj.

Lastnosti, ki jih ima knjiga, smo opisali v razredu, njihove vrednosti pa ima vsaka instanca razreda svoje, zato moramo vrednosti lastnosti za vsako instanco povedati posebej.

Funkcionalnosti objektov opišemo z metodami. Ponovili smo že *statične* metode, torej tiste, pri katerih v glavi zapišemo določilo `static`. Statične metode se izvedejo zgolj v odvisnosti od vhodnih podatkov in pokličemo jih le z njihovim imenom oziroma, ko jih želimo izvesti v drugem razredu, kot so bile definirane, to storimo s kombinacijo imena razreda, kjer so definirane, in njihovega imena ločenega s piko, npr.

```
RazredMetode.ImeMetode(vrednosti parametrov).
```

Pri definiciji *objektnih metod* v glavi izpustimo določilo `static`. Na ta način prevajalniku povemo, da bo metodo poklical objekt oziroma instanca nekega razreda, metoda pa bo ukaze v njenem jedru izvedla glede na lastnosti klicočega objekta.

Primere objektnih metod smo že spoznali. Na primer, za dano spremenljivko tipa `string` je `ToUpper` objektna metoda, ki vse črke v vrednosti spremenljivke spremeni v velike.

Oglejmo si primer metode za razred `Weather`, ki smo ga pripravili v prejšnjem razdelku. Metoda bo poskrbela za izpis lastnosti meritve.

```
1  public class Weather
2  {
3    ...
4
5    public void WriteProperties()
6    {
7      Console.WriteLine(
8        $"Vreme danes ob {this.MeasurementTime:HH:mm:ss} " +
9        "je takšno, da imamo: \n" +
10       $"{this.Temperature} stopinj celzija, \n" +
11       $"{this.AirPressure} mbar zračnega tlaka in \n" +
12       $"hitrost vetra {this.WindSpeed} m/s");
13   }
14 }
```

Metoda ne prejme nobenih parametrov, saj v tem primeru vse podatke dobi iz objekta, ki jo pokliče. Na klicoči objekt se nanaša rezervirana beseda `this`.[2] Klic metode bo na primer naslednji.

```
1  Weather currentWeather = new Weather();
2  currentWeather.WriteProperties();
```

Metodo `WriteProperties` smo poklicali z instanco `currentWeather`.

2.1.1 Vprašanja

1. Kako bi opisali razred in instanco razreda?

2. Kako ustvarimo novo instanco?

[2] V jedru metode v tem primeru eksplicitno navajanje `this` niti ni obvezno, dodali smo ga le za boljše razumevanje.

3. Kateri so osnovni elementi razreda?
4. Kako prepoznamo konstruktor?
5. Kako definiramo objektno metodo?
6. Kako pokličemo objektno metodo?
7. Kakšna je razlika med metodo z določilom `static` (*statično metodo*) in metodo brez tega določila (*objektno metodo*)? [Arh, Q32]

2.1.2 Naloge

Naloga 2.1.1. Ustvarite razred `ZapiskiPredavanj`, ki predstavlja opis instance zapiskov predavanj. Razredu dodajte prazen konstruktor, polje `seznamPoglavij` ter nekaj metod. Nato v metodi `Main` ustvarite instanco tega razreda, ji napolnite polje ter pokličite eno od metod.

Naloga 2.1.2. Razredu iz zgornje naloge dodajte še tri konstruktorje in v metodi `Main` ustvarite po eno instanco z vsakim izmed njih.

Naloga 2.1.3. Zgornji nalogi dopolnite z razredom `Poglavje`, ki nosi podatke o posameznem poglavju, npr. naslov, seznam podpoglavij in število strani. V razredu `ZapiskiPredavanj` napišite metodo, ki vrne skupno število strani v zapiskih.

2.2 Lastnosti

V prejšnjem razdelku smo že govorili o spremenljivkah razreda oziroma poljih. Definirali smo jih na nivoju razreda, torej izven vseh metod. Da smo do njih lahko dostopali tudi v drugih razredih, na primer tam, kjer smo kreirali instance razreda, smo jim morali dodati določilo `public`. Tega vedno nočemo storiti, saj v kakšnih primerih ne želimo, da ima razvijalec omogočen dostop do spreminjanja vrednosti neke spremenljivke. Pri tem gre lahko za varnostna vprašanja ali pa preprosto za omejevanje dostopa, da ne pride do lapsusa pri pisanju programa in se vrednost popravlja na mestu, ki za to ni predvideno. Popravljanje polja lahko onemogočimo samo z zmanjšanjem vidnosti, vendar na ta način zmanjšamo tudi dostop za branje, česar običajno nočemo.

Za upravljanje vrednosti lastnosti objektov zato uporabljamo posebne strukture, ki imajo povedno ime *lastnosti* (angl. *properties*). Njihovo jedro je sestavljeno iz dveh delov, in sicer iz opisa nastavljanja vrednosti (blok `set`) in opisa podajanja vrednosti (blok `get`).[3]

Najprej si oglejmo glavo lastnosti. Podobno kot metodi, ji določimo določilo zasebnosti, tip rezultata in ime, ne prejema pa nobenih parametrov. Po konvenciji imena lastnosti pišemo z veliko začetnico.

[določilo zasebnosti] [tip rezultata] [ime lastnosti]

Vrednost lastnosti v razredu sicer shranjujemo v spremenljivki, ki ima določilo zasebnosti nastavljeno na `private`, pa tudi zapisujemo jo vanjo. Zato bloka `get` in `set` delata z njo. Blok `get` vrača vrednost lastnosti, tako da jedro na prvi pogled spominja na metodo. Blok `set` spremenljivki, ki nosi vrednost, lastnosti priredi vrednost, ki jo lastnosti nastavimo, v jedru pa nastavljeno vrednost nosi rezervirana beseda `value`.

Oglejmo si primer definiranja lastnosti `Designer` v razredu `Dress`.

```
1  public class Dress
2  {
3    // Konstruktor
4    public Dress(string designedBy)
5    {
6      Designer = designedBy;
7    }
8
9    // Inicializacija spremenljivke,
10   // ki nosi vrednost za lastnost
11   private string designer = string.Empty;
12
13   // Definicija lastnosti
14   public string Designer
15   {
16     get
17     {
18       return designer;
19     }
20     set
21     {
```

[3] V angleščini ta dva bloka imenujemo *accessors*.

```
22          designer = value;
23        }
24      }
25    }
```

Jedri blokov `get` in `set` sta lahko kompleksnejši. Vanju lahko dodamo validacijo vrednosti, njeno oblikovanje ali poljubno logiko, ki je ne izvajamo pri pridobivanju vrednosti v uporabniškem vmesniku našega programa oziroma v delu kode, kjer nastavljamo oziroma beremo vrednost. Pri nastavitvi lahko na primer vrednost nastavimo na privzeto, če je podana vrednost neprimerna:

```
1     ...
2     // Definicija lastnosti
3     public string Designer
4     {
5       get
6       {
7         return designer;
8       }
9       set
10      {
11        if (value == null || value.Length == 0)
12          designer = "Not given";
13
14        designer = value;
15      }
16    }
17    ...
```

Če bi želeli omejiti popravljanje vrednosti lastnosti `Designer` na razred in izven razreda omogočiti le dostopanje do vrednosti, potem moramo v glavo stavka `set` dodati določilo `private`.

```
1     ...
2     // Definicija lastnosti
3     public string Designer
4     {
5       get
6       {
7         return designer;
8       }
9       private set
10      {
11        designer = value;
12      }
13    }
14    ...
```

Z omejitvijo vidnosti nastavljanja na `private` lahko vrednost lastnosti nastavimo samo v matičnem razredu, in sicer na treh mestih[4]:

[4]V resnici jo lahko nastavimo tudi v jedru kakšne druge lastnosti, vendar se nastavljanju vrednosti drugih lastnosti znotraj posamezne lastnosti izogibamo.

- neposredno za lastnostjo,
- v konstruktorju,
- v lokalni metodi.

Če želimo, da je lastnost na voljo samo za podajanje vrednosti, se pravi, da ji vrednosti ne moremo določiti niti v lokalni metodi, potem izpustimo blok `set`. Pravimo, da je lastnost na voljo *samo za branje* (ang. *read-only*).

Kadar v jedrih stavkov `get` in `set` nimamo posebne logike in zgolj beremo oziroma prirejamo vrednost spremenljivki, potem lahko uporabimo *samodejno implementacijo* lastnosti. V tem primeru prevajalnik za vrednost v ozadju pripravi polje, ki nosi podatek. Zgornji razred bi tako lahko prepisali v obliko:

```
public class Dress
{
  // Konstruktor
  public Dress(string designedBy)
  {
    Designer = designedBy;
  }

  // Samodejna implementacija lastnosti
  public string Designer { get; set; }
}
```

Zgled 2.2.1. Pripravimo razred `Weather`, ki bo definiral sklop vremenskih meritev. Poleg časa meritve definirajmo še lastnosti temperatura, zračni tlak in hitrost vetra. Dodajmo še prazen konstruktor in konstruktor, ki kot parameter prejme čas meritve. Pisanje v čas meritve nastavimo na zasebno vidnost, saj nočemo dovoliti njegovega spreminjanja, potem ko ga enkrat določimo (v tem primeru v konstruktorju).

```
  /// <summary>
  /// Razred za shranjevanje vremenskih podatkov
  /// za eno meritev
  /// </summary>
public class Weather
{
  // Prazen konstruktor
  public Weather() { }

  // Konstruktor z enim parametrom
  public Weather(DateTime time)
  {
    measurementTime = time;
  }

  // Zasebna spremenljivka za hranjenje
  // podatkov o času meritve
  private DateTime measurementTime;
  // Definicija lastnosti
```

```
20      public DateTime MeasurementTime
21      {
22        // Določimo lastnosti branja
23        get
24        {
25          return measurementTime;
26        }
27        // Določimo lastnosti pisanja
28        private set
29        {
30          measurementTime = value;
31        }
32      }
33
34      // Samodejna implementacija lastnosti
35      public double Temperature { get; set; }
36      public double AirPressure { get; set; }
37      public double WindSpeed { get; set; }
38    }
```

V razredu smo definirali tri lastnosti z uporabo samodejne implementacije, le lastnost `MeasurementTime` smo implementirali ročno. Ker niti pri njej v stavkih `get` in `set` ne potrebujemo posebne logike, bi samodejno implementacijo lahko uporabili tudi zanjo.

2.2.1 Vprašanja

1. Zakaj so lastnosti boljša izbira od polj (razrednih spremenljivk)? [Arh, Q34]
2. Kako definiramo lastnost?
3. Kako preprečimo popravljanje vrednosti lastnosti izven razreda?
4. Katera določila zasebnosti poznamo v C#? [Arh, Q23]
5. Kako definiramo samodejno implementirano lastnost, kje ji lahko nastavimo vrednosti, če je `set` nastavljen na `private`?
6. Kako zapišemo lastnost, ki je na voljo samo za branje? Kje lahko nastavljamo vrednosti njej?
7. V katerih primerih je smiselno implementirati lastnosti in v katerih namesto njih raje implementiramo metode?
8. Opišite uporabo rezervirane besede `value` v lastnostih.

2.2.2 Naloge

Naloga 2.2.1. Ustvarite razred `Avto` in mu dodajte nekaj ustreznih lastnosti, dve implementirajte v celoti, dve naj bosta samodejno implementirani in ena samodejno implementirana, vendar samo za branje.

Naloga 2.2.2. Lastnostim iz zgornjega razreda vrednosti nastavite v praznem konstruktorju, nato pa zapišite še metodo, ki uporabnika pozove k vnosu vrednosti za vsako izmed njih. Za lastnosti, ki sta implementirani v celoti, v bloku `set` preverite, če so vnešene vrednosti ustrezne.

Naloga 2.2.3. Pripravite razred `PrometnaIzkaznica` s pripadajočimi lastnostmi in konstruktorji ter jo dodajte kot lastnost v razred `Avto`.

2.3 Indekserji

V C# poznamo še eno vrsto elementov, ki je zelo podobna lastnostim - *indekserje*. Indekserji nam omogočajo *indeksiranje* neposredno na instancah razredov in tudi struktov, podobno kot smo tega navajeni iz seznamov in tabel. Vrednosti lahko določimo ali pridobimo neposredno na glavni instanci s podanim indeksom oziroma ključem, na mestu katerega leži iskana vrednost.

Oglejmo si primer določanja in branja vrednosti indekserja, definiranega v razredu `MyIndexerClass`.

```
// Ustvarimo nov objekt, ki ima definiran indekser.
MyIndexerClass myIndexerClass = new MyIndexerClass();

// Nastavimo dve vrednosti
myIndexerClass["Jabolko"] = 12.0;
myIndexerClass["Hruška"] = 17.0;

// Preberimo in izpišimo nastavljeni vrednosti
Console.WriteLine($"Cena jabolka={myIndexerClass["Jabolko"]}," +
  $" hruška={myIndexerClass["Hruška"]}");
```

Opazimo lahko, da v zgornjem primeru nismo nikjer določili tipa vrednosti, ki jo indekserju zapisujemo.

Pri tabelah in seznamih ta podatek vidimo neposredno iz njihove definicije, medtem ko pri instancah, ki imajo definiran indekser, to ni vidno. Tip vrednosti določimo ob definiciji indekserja. Struktura glave indekserja je podobna glavam lastnosti, z izjemo, da ne podajamo imena (saj se indeksiranje izvaja neposredno na instanci razreda), ampak namesto imena zapišemo `this`, dodatno pa podamo pa parameter (ki ima vlogo indeksa oziroma ključa), katerega uporabljamo pri določanju in branju vrednosti iz ustreznega polja. Oglejmo si primer, ko indekser predstavlja slovar.

```
// Definiramo polje, ki bo hranilo vrednosti,
// v našem primeru je to slovar
// (lahko pa bi uporabili tabelo, seznam itd.)
private Dictionary<string, double> dicPrice =
  new Dictionary<string, double>();

// Indekserji vzamejo vrednosti neposredno iz slovarja
// Parameter predstavlja indeks v slovarju.
// Vrača vrednost na danem indeksu.
public double? this[string article]
{
  get
  {
    return this.dicPrice.ContainsKey(article) ?
      (double?)this.dicPrice[article] : null;
  }
  set
  {
    // Popravljanje vnosa
    if (this.dicPrice.ContainsKey(article))
```

```
        this.dicPrice[article] = (double)value;
      // Dodajanje novega vnosa
      else
        this.dicPrice.Add(article, (double)value);
    }
  }
```

Podobno kot pri lastnostih, smo v razredu definirali zasebno spremenljivko `dicPrice`, ki skrbi za shranjevanje vrednosti indekserja. V našem primeru smo uporabili slovar in na ta način do vrednosti dostopamo s ključem tipa `string`, lahko pa bi uporabili tabelo (ali seznam), kjer bi kot ključ podali indeks v tabeli (ali seznamu).

Razmislimo še o primeru uporabe indekserja. Najbolj osnoven primer je razred, ki predstavlja nadgradnjo nekega slovarja, za katerega želimo v dodatnih lastnostih shraniti še dodatne metapodatke. S tem se lahko v instanci sprehajamo po vrednostih slovarja, obenem pa imamo dostop do nekaj splošnih lastnosti, ki veljajo za celoten slovar.

Zgled 2.3.1. Naredimo primer razreda, ki definira generacijo študentov (študente vpisane v istem študijskem letu). Vsaka generacija ima dve lastnosti - leto prvega vpisa in študijski program, indeksirani pa so študentje generacije.

```
public class StudentGeneration
{
  // Ime študijskega programa generacije
  public string ProgramName { get; set; }

  // Prvo leto vpisa
  public int FirstEnrolmentYear { get; set; }

  // Zasebni slovar, v katerem hranimo študente generacije,
  // za ključe pa uporabljamo njihove vpisne številke
  private Dictionary<int, Student> dicStudents =
    new Dictionary<int, Student>();

  // Definicija indekserja
  public Student this[int enrolmentNumber]
  {
    get
    {
      return dicStudents.ContainsKey(enrolmentNumber) ?
        dicStudents[enrolmentNumber] : null;
    }
    set
    {
      // Študenta dodamo samo, če še ne obstaja.
      if (dicStudents.ContainsKey(enrolmentNumber))
      {
        throw new Exception($"A student with enrolment number" +
          $" {enrolmentNumber} " +
          $" already exists! Use Update method for changing" +
          $" his properties.");
      }
      dicStudents[enrolmentNumber] = value;
```

```
        }
    }
}
```

2.3.1 Vprašanja

1. Kaj je indekser in kako ga definiramo?
2. Opišite razlike med lastnostmi in indekserji.
3. V katerih primerih je smiselno definirati indekser in kdaj preprosto delati s slovarjem? Navedite primer.

2.3.2 Naloge

Naloga 2.3.1. Za razred `Zapiski` iz Naloge 2.1.1 naredite indekser, ki bo vračal objekt `Poglavje` glede na dani naslov poglavja.

Naloga 2.3.2. Pripravite razred `Indeks`, ki bo vseboval indekser, ki bo vračal oceno glede na predmet, ki ga podate kot ključ. Predmeti naj bodo določeni v enumeraciji. Razredu dodajte še nekaj lastnosti, ki jih mora imeti vsak indeks.

Naloga 2.3.3. Napišite razred `StudentGeneration`, ki ima kot lastnosti *leto vpisa* in *študijski program*, kot indekser pa množico študentov vpisanih v ta program v danem letu. Študent naj identificira glede na njihovo vpisno številko, shranjeni pa naj bodo kot poseben objekt tipa `Student`.

2.4 Dedovanje

V objektno orientiranem programiranju sta dedovanje in polimorfnost dve ključni karakteristiki. Medtem ko prva omogoča nadgradnjo obstoječih razredov, druga izkorišča sorodnost objektov za poenostavitev pisanja kode po eni strani, po drugi strani pa omogoča večjo abstrakcijo in manipuliranje z različnimi objekti na enak način.

Z dedovanjem omogočimo izbranemu razredu uporabo lastnosti in metod razreda, iz katerega deduje. Razredu, ki deduje, pravimo *podrazred*, razredu, iz katerega deduje, pa *nadrazred*. Pri dedovanju ne gre le za uporabo lastnosti in metod, ampak jih lahko tudi dopolnjujemo, spreminjamo in dodajamo svoje.

Dedovanje uporabimo, ko v osnovnem razredu nimamo ustreznih funkcionalnosti, jih pa vanj ne želimo preprosto dodati, ker bodo dodatne funkcionalnosti imele samo nekatere (specifične) instance razreda. V takih primerih definiramo podrazred in mu dodamo manjkajoče elemente. Dedovanje nakažemo v glavi razreda tako, da za imenom razreda zapišemo dvopičje in ime razreda, iz katerega naj deduje.

Za primer vzemimo razred `Bicycle` in dva podrazreda `MountainBike` in `EBike`.

```
// Nadrazred za splošno kolo
public class Bicycle
{
  public int FrameSize { get; set; }

  public int NumGears { get; set; }

  public string Brand { get; set; }

  public int CurrentGear { get; set; }
}
```

```
// Nadrazred označimo za dvopičjem
public class MountainBike : Bicycle
{
  // Dodatna lastnost
  public double TireWidth { get; set; }
}
```

```
// Nadrazred označimo za dvopičjem
public class EBike : Bicycle
{
  // Dodatna lastnost
  public double BateryTime { get; set; }
}
```

Vsak razred lahko neposredno deduje iz natanko enega razreda, se pa vse lastnosti in metode dedujejo po liniji dedovanja. Privzeto je vsak razred podrazred razreda `Object`, ki med drugim vsebuje objektno metodo `ToString`, katera izpiše lastnosti izbrane instance razreda, in zato ima vsak razred oziroma njegova instanca na voljo metodo `ToString`.

Metoda `ToString` se na primer izvede, kadar objekt "prištevamo" nekemu nizu. Privzeti izpis metode `ToString` sicer običajno ni tisto, kar si od njega želimo, zato metodo

`ToString` v razredu "povozimo" oziroma prepišemo z metodo, ki pripravi izpis po naših željah. Pri tem uporabimo rezervirano besedo `override`. Oglejmo si primer za razred `Bicycle`.

```
// Nadrazred za splošno kolo
public class Bicycle
{
  ...

  public override string ToString()
  {
    return  $"Velikost okvirja: {FrameSize}\n" +
        $"Število prestav: {NumGears}\n" +
        $"Znamka: {Brand}";
  }
}
```

2.4.1 Prepisane metode

Da lahko neko metodo v podrazredu dopolnimo ali celo implementiramo povsem drugače, moramo to zanjo predvideti že v nadrazredu. To storimo z navedbo rezervirane besede `virtual`.[5]

Za primer pripravimo metodo `ChangeGear` v razredu `Bicycle`.

```
// Nadrazred za splošno kolo
public class Bicycle
{
  ...

  public virtual void ChangeGear(int increaseBy)
  {
    this.CurrentGear += increaseBy;
  }
}
```

To, da je metoda označena z `virtual`, pomeni, da se v tej obliki izvede za vsako instanco poljubnega podrazreda, razen če jo v podrazredu prepišemo.

```
public class EBike : Bicycle
{
  public double BateryTime { get; set; }

  public override string ToString()
  {
    // Z rezervirano besedo base povemo,
    // da se sklicujemo na nadrazred.
    return base.ToString() +
    $"Čas vzdržljivosti baterije: {BateryTime}";
  }
```

[5]Poleg `virtual` lahko uporabimo tudi rezervirano besedo `abstract`. Abstraktne razrede in metode si bomo ogledali v naslednjem razdelku.

```
12
13      // Rahlo popravimo metodo iz nadrazreda
14      public override void ChangeGear(int increaseBy)
15      {
16        this.CurrentGear += increaseBy;
17        Console.WriteLine("Zvišanje prestave lahko zmanjša "
18          + "vzdržljivost baterije!");
19      }
20    }
```

V metodi `ToString` smo uporabili klic `base.ToString()`, kar pomeni klic metode, ki je definirana v nadrazredu. Z rezervirano besedo `base` namreč označujemo sklice na nadrazred (tako kot s `this` klice na trenutni razred).

2.4.2 Nesorodne istoimenske metode

Včasih, vendar res zelo redko, želimo v podrazredu definirati metodo (ali lastnost) z enakim imenom, kot ga že ima neka metoda v liniji dedovanja navzgor (metoda nekega prednika), pri čemer pa ta metoda nima istega pomena (metodi nista sorodni) - s tem mislimo na to, da v določenih situacijah želimo uporabljati novo metodo, v drugih pa novo metodo z enakim imenom. Novo metodo v podrazredu definiramo tako, da na začetku glave metode dodamo rezervirano besedo `new`.[6]

Za primer v razred `Bicycle` in podrazred `EBike` dodajmo nesorodni metodi `Renew`. V nadrazredu bo imela metoda funkcijo obnovitve kolesa, v podrazredu pa obnovitve baterije.

```
1   public class Bicycle
2   {
3     ...
4
5     // Metoda v nadrazredu
6     public void Renew()
7     {
8       Console.WriteLine("Kolo se obnavlja.");
9     }
10  }
```

```
1   public class EBike : Bicycle
2   {
3     ...
4
5     // Istoimenska metoda v podrazredu, ki ima s tisto
6     // iz nadrazreda skupno zgolj ime.
7     new public void Renew()
8     {
9       Console.WriteLine("Baterija se obnavlja.");
10    }
11  }
```

[6] Rezervirano besedo `new` sicer lahko tudi izpustimo, razvojno okolje pa nas bo opozorilo, da z definiranjem metode z istim imenom skrijemo metodo, ki je že definirana v nekem predniku. Koda je preglednejša, če `new` vendarle uporabimo.

Omenimo še, da novo metodo lahko ustvarimo ne glede na to, če je metoda z istim imenom v nadrazredu označena z `virtual` ali `override`.

2.4.3 Kreiranje instance podrazreda

Ob kreiranju instance podrazreda se najprej izvedejo klici konstruktorjev po liniji nadrazredov. Omenili smo že, da se za razred, v katerem ne določimo konstruktorja eksplicitno, privzeto ustvari prazen konstruktor. Če imamo v nadrazredu definiran konstruktor, ki ni prazen, praznega pa ne, moramo v vsakem konstruktorju podrazreda eksplicitno poklicati tudi konstruktor nadrazreda in mu podati ustrezne parametre.

Za primer vzemimo razred `Coffee`.

```
1  public class Coffee
2  {
3    public Coffee(double quantity)
4    {
5      Quantity = quantity;
6    }
7
8    public double Quantity { get; set; }
9  }
```

Ima neprazen konstruktor in za njegov parameter moramo poskrbeti v vsakem konstruktorju podrazredov. Pri tem se sicer lahko odločimo, da za vrednost konstruktorja nadrazreda podamo neko privzeto vrednost.

```
1  public class Turkish : Coffee
2  {
3    // Ob klicu konstruktorja moramo poskrbeti še za
4    // parametre konstruktorja v nadrazredu
5    public Turkish(double quantity, int preparationTime)
6      : base(quantity)
7    {
8      PreparationTime = preparationTime;
9    }
10
11   public int PreparationTime { get; set; }
12 }
```

2.4.4 Zapečateni razredi

V posebnih primerih želimo za nek razred zaustaviti možnost njegovega dedovanja, se pravi, da želimo preprečiti ustvarjanje njegovih podrazredov. V takem primeru razred *zapečatimo* z uporabo določila `sealed` v glavi razreda.

```
1  public sealed class NonInheritableClass
2  {
3    ...
4  }
```

Določilo `sealed` lahko uporabimo tudi bolj omejeno, na primer da ga dodamo metodi ali lastnosti in s tem preprečimo njeno spreminjanje v podrazredih. Določilo je smiselno le, kadar ima metoda tudi določilo `override`. Če ima določilo `virtual`, potem želimo hkrati omogočiti njeno dedovanje in ga zapečatiti, kar je protislovno, če pa gre za novo metodo brez določila `virtual`, potem je v podrazredih tako in tako ne moremo povoziti.

```
public class ClassWithSealedMethod
{
  ...

  sealed public override SealedMethod() { ... }

  ...
}
```

2.4.5 Polimorfizmi

Polimorfizem pomeni "mnogo oblik". Koncept uporabljamo, kadar želimo združiti sorodne objekte in jih obravnavati na enak način, čeprav ima vsak od njih malo drugačno implementacijo metod z istim imenom.

Naredimo primer z liki. Glavni razred naj bo razred `Shape`.

```
public class Shape
{
  public Shape(int id)
  {
    ID = id;
  }

  public int ID { get; set; }

  /// <summary>
  ///  Pripravimo si metodo za izračun obsega,
  ///  ki jo bomo v podrazredih povozili.
  /// </summary>
  public virtual double Perimeter()
  {
    return 0;
  }
}
```

Virtualno metodo `Perimeter`, ki vrača vrednost `0`, pripravimo samo zato, da jo bomo lahko klicali tudi na instancah razreda `Shape`, čeprav takšnih instanc eksplicitno niti ne bomo kreirali.[7]

Nadalje pripravimo še dva podrazreda.

```
public class Circle : Shape
{
  public Circle(int id, double radius) : base(id)
```

[7] Kasneje se bomo naučili, da lahko razredom eksplicitno prepovemo izdelavo instanc tako, da jih označimo kot abstraktne.

```
    {
      Radius = radius;
    }

    public double Radius { get; set; }

    public override double Perimeter()
    {
      return 2 * Radius * Math.PI;
    }

    public double Area()
    {
      return Math.PI * Radius * Radius;
    }
  }

  public class Rectangle : Shape
  {
    public Rectangle(int id, double a, double b) : base(id)
    {
      A = a;
      B = b;
    }

    public double A { get; set; }

    public double B { get; set; }

    public override double Perimeter()
    {
      return 2 * A + 2 * B;
    }
  }
```

Ker so vse instance podrazredov obenem tudi instance nadrazreda, jih lahko zberemo v seznam, ki kot vrednosti prejme tip nadrazreda.

```
    List<Shape> lstShapes = new List<Shape>()
    {
      new Circle(1, 3.4),
      new Circle(2, 1),
      new Circle(3, 2),
      new Rectangle(4, 2, 6),
      new Rectangle(5, 3, 5.7)
    };
```

To pomeni, da se lahko po seznamu sprehodimo in za vsako instanco pokličemo na primer metodo `Perimeter`. Vse instance to metodo poznajo (tudi tiste tipa `Shape`), izvede pa se glede na razred, kateremu dejansko pripadajo. V primeru instance razreda `Circle` se tako izvede njena metoda.

```
    foreach(Shape shp in lstShapes)
    {
```

```
3         Console.WriteLine($"Obseg lika {shp.ID} "
4           + "je {shp.Perimeter()}");
5     }
```

Če želimo preveriti, ali imamo opravka s posebnim tipom, to preverimo z rezervirano besedo `is`. Če je instanca pravega tipa, jo moramo vanj še preoblikovati (pred njeno ime v oklepaju zapišemo iskani tip) in lahko pokličemo metodo, ki je na voljo le v tem tipu. Na primeru instanco preoblikujemo v `Circle` in pokličemo metodo `Area`.

```
1     foreach(Shape shp in lstShapes)
2     {
3       if (shp is Circle)
4       {
5         double area = ((Circle)shp).Area();
6       }
7     }
```

2.4.6 Vprašanja

1. Kaj je dedovanje v C#? [Arh, Q22]
2. Kako določimo, da je razred `RazredB` podrazred razreda `RazredA`?
3. Kako v podrazredu definiramo lastnost z enakim imenom kot v nadrazredu in kako v podrazredu definiramo metodo z enakim imenom kot v nadrazredu, ne da bi jo prepisali? [Arh, Q26]
4. Podrazred katerega razreda je privzeto vsak razred v C#?
5. Katere metode deduje vsak podrazred razreda `Object`?
6. Kako "povozimo" metodo `ToString`?
7. S katerim določilom preprečimo dedovanje razreda?
8. Kaj je polimorfizem? Navedite primer, kjer je koncept polimorfizma uporaben.

2.4.7 Naloge

Naloga 2.4.1. Definirajte nadrazred `Avto` in njegove podrazrede `Cabrio`, `SUV`, `Sedan`. Vsem določite primerne lastnosti.

Naloga 2.4.2. Za razred `Avto` določite metodo, ki bo uporabna v vseh podrazredih, v podrazredih pa določite metode, ki so zanje specifične.

Naloga 2.4.3. Za enega od podrazredov si zamislite lastnost z enakim imenom, kot je lastnost nadrazreda, vendar ima bistveno drugačno vlogo kot lastnost nadrazreda.

Naloga 2.4.4. Postali ste del razvojne ekipe večjega sistema. Vaša prva naloga je, da pripravite podrazred `NadzornaKomisija` razreda `Komisija`, ki bo vseboval metodo `PreveriClana`. Metoda z enakim imenom že obstaja v nadrazredu, vendar zaradi politike podjetja, tega razreda trenutno ne morete spreminjati, metoda pa tudi ni označena kot `virtual`, se pravi, da je ne morete povoziti. Ali lahko metodo z enakim imenom sploh dodate v podrazred? Če da, kako in kako se razlikuje njeno obnašanje glede na metode, ki jih povozimo z `override`?

Naloga 2.4.5. Definirajte razreda `Menu` in `Jed`. `Menu` naj predstavlja dnevni menu v restavraciji (glede na dan), ki ima kot lastnost tudi seznam jedi. Posamezna jed ima lastnosti naziv in cena.

Za razred `Jed` naredite podrazred `Sladica`, ki bo imel dodatno lastnost `Kalorije`.

V razredih `Jed` in `Sladica` povozite metodo `ToString`, da bo ustrezno vračala vse lastnosti instanc.

Metodo `ToString` povozite tudi v razredu `Menu`. Vrne naj niz z dnevom in vsemi jedmi, ki so na menuju, med seboj pa naj bodo ločene s prazno vrstico.

V razredu `Menu` napišite še metodo, ki bo izpisala skupno ceno menuja. Metoda naj ima vhodni parameter tipa bool, ki bo določal, ali želite ob ceni plačati še 10% napitnine ali ne. Če je vrednost parametra true, naj se skupna cena primerno izračuna.

Za vsaj dva dni v tednu pripravite instanci razreda `Menu`, ki bosta imeli na seznamu jedi vsaj po tri jedi, od tega vsak natanko eno jed tipa `Sladica`. Na koncu oba menuja tudi izpišite.

Naloga 2.4.6. Razvijate rešitev za spletno trgovino in pripraviti morate podatkovni model za nakupovalno košarico. Nakupovalna košarica lahko vsebuje množico različnih izdelkov. Napišite ustrezen razred za košarico, izdelek in vsaj pet podrazredov izdelka. Smiselno dodajte nekaj metod in lastnosti v vsakem od razredov, s čimer boste lahko na primeru prikazali, kako nam polimorfizmi pomagajo pri razvoju programskih rešitev.

2.5 Abstrakcija

Včasih v naših razrednih modelih zastavimo dedovanje na način, da je nadrazred zelo splošen. Celo tako splošen, da njegove neposredne instance nikoli ne želimo ustvariti, saj o njej nimamo dovolj podatkov.

Na primer, razred `ChessPiece`, ki predstavlja šahovsko figuro, lahko predpiše lastnost za vrednost figure in metodo, ki figuro premakne na izbrano polje, če je premik po pravilih. Za razred bi glede na tip figure definirali podrazrede in v vsakem implementirali metodo za premik, ki bi ustreznost premika preverjala za izbrani tip figure, medtem ko ta metoda v razredu `ChessPiece` nima nobenega podatka o dovoljenih premikih. Instanca razreda `ChessPiece` bi tako bila brez pomembne informacije o tipu figure in zato v naši rešitvi ustvarjanje takšne instance niti ne želimo dopustiti.

Programskemu jeziku preprečitev ustvarjanja instance danega razreda (in dopustitev ustvarjanja instanc podrazredov) sporočimo z rezervirano besedo `abstract`.

```
1  // Definicija abstraktnega razreda
2  public abstract class ChessPiece
3  {
4    ...
5  }
```

V abstraktnih razredih lahko definiramo tudi *abstraktne metode* in *lastnosti*. To so metode (oziroma lastnosti), ki v razredu nimajo dejanske implementacije, zgolj predpišemo, katere parametre metoda dobi in kakšnega tipa vrednosti vrača. Podobno pri lastnostih navedemo, katerega od elementov `get` in `set` naj lastnost implementira, ko jo prepišemo v podrazredu.

```
1  public abstract class ChessPiece
2  {
3    // Definicija abstraktne metode
4    public abstract bool Move(int line, int column);
5
6    // Definicija abstraktne lastnosti
7    public abstract string Label { get; }
8  }
```

Abstraktne metode in lastnosti moramo implementirati v vsakem od neabstraktnih razredov, ki dedujejo neposredno od abstraktnega razreda. Pri tem uporabimo rezervirano besedo `override`, da prevajalniku povemo, da popravljamo metodo, ki je definirana že višje po liniji dedovanja, in ne pripravljamo nove, nesorodne metode.

```
1  public class Queen : ChessPiece
2  {
3    // Implementacija metode v podrazredu
4    public override bool Move(int line, int column)
5    {
6      ...
7    }
8
9    // Implementacija lastnosti v podrazredu
10   public override string Label
```

```
    {
      get
      {
        return "Q";
      }
    }
  }
```

V podrazredu nekega razreda lahko torej prepisujemo metode, ki so v nadrazredu (oziroma višje po liniji dedovanja) označene z določili `virtual`, `abstract` ali `override`. Zadnje so metode, ki so bile označene z določiloma `virtual` ali `abstract` nekje višje po liniji dedovanja.

V abstraktnem razredu, kljub temu da jih neposredno ne bomo nikoli poklicali, lahko definiramo tudi konstruktorje. To storimo, kadar želimo na enem mestu nastaviti vrednost nekemu polju ali lastnosti za vsako instanco podrazreda. V našem primeru šahovskih figur bi to bila vrednost figure.

```
public abstract class ChessPiece
{
  // Konstruktor v abstraktnem razredu
  public ChessPiece(double weight)
  {
    this.Weight = weight;
  }

  public double Weight { get; }
}
```

```
public class Queen : ChessPiece
{
  // Konstruktor podrazreda, ki nastavi
  // vrednost lastnosti Weight v nadrazredu
  public Queen() : base(9.0) { }

  ...
}
```

Ustrezen konstruktor nadrazreda se bo izvedel ob kreiranju vsake instance kateregakoli podrazreda.

2.5.1 Vprašanja

1. Kaj je abstraktni razred?
2. Na katere načine lahko določimo, da smemo metodo prepisati v podrazredu?
3. Kakšna je razlika med določiloma `abstract` in `virtual` za metodo?
4. Kdaj je bolj smiselno imeti abstraktno metodo in kdaj virtualno?
5. Ali v abstraktnem razredu lahko definiramo konstruktor? Zakaj bi to želeli storiti, če instance abstraktnega razreda ne moremo ustvariti?

6. Ali lahko kako nastavimo/popravimo vrednost spremenljivke, ki je v abstraktnem razredu definirana z določiloma `private` in `readonly`, v podrazredu? Če da, kje in kako?

2.5.2 Naloge

Naloga 2.5.1. Zapišite abstraktni razred `Artikel` in mu smiselno določite dve abstraktni metodi ter eno virtualno. Nato pripravite še tri podrazrede, npr. `Oblačilo`, `Obutev`, `Pripomoček` in razmislite, če je smiselno, da so tudi ti razredi abstraktni in potrebujemo nadaljnjo vejitev. Če je odgovor da, naredite še vejitev na naslednji nivo.

Naloga 2.5.2. Razmislite o modelu razredov, kjer je smiselna uporaba abstraktnega razreda in nekaj podrazredov, pri čemer bo imel abstraktni razred vsaj en konstruktor, v katerem se nastavijo lastnosti, ki so skupne vsem podrazredom.

Naloga 2.5.3. V knjižnici so naročili izdelavo novega programskega orodja, ki bo omogočalo izposojo knjižničnega gradiva na več načinov: preko mobilne aplikacije, preko spletne aplikacije in preko namizne aplikacije, ki jo bo upravljal uslužbenec knjižnice. Pripravite razredni model z razredi, ki bodo predstavljali različna knjižnična gradiva. Napišite vsaj tri nivoje dedovanja, pri čemer naj bodo vsi razredi na prvih dveh nivojih abstraktni.

Naloga 2.5.4. Domača osnovna šola vam je naročila pripravo programa, ki bo znal upravljati s preprostimi geometrijskimi operacijami. Na primer, za dane like (trikotnik, kvadrat, pravokotnik, ...) in njihove velikosti (npr. dolžine stranic), bi moral program izračunati ploščino, obseg in druge lastnosti. Vaša naloga je, da pripravite modele (abstraktne razrede, podrazrede), ki bodo predvideli potrebne funkcije in lastnosti.

2.6 Vmesniki

2.6.1 Vprašanja

1. Opišite namen/uporabo vmesnikov.
2. Katere elemente lahko definiramo v vmesnikih?
3. Koliko vmesnikov lahko implementira en razred in kako se to razlikuje od dedovanja?
4. Na kakšen način implementiramo vmesnik v nek razred? Prikažite na primeru.
5. Kakšne so prednosti oziroma slabosti abstraktnih razredov glede na vmesnike? [Arh, Q24]
6. Ali lahko za implementiranje lastnosti, ki jo določa vmesnik, uporabimo samodejno implementacijo?
7. Kaj je eksplicitna implementacija metode vmesnika in kaj nam omogoča? [Arh, Q25]
8. Kako lahko pokličemo metodo vmesnika, ki je ekplicitno implementirana?
9. Ali lahko v vmesniku definiramo abstraktno metodo? Navedite primer situacije, ko je takšna rešitev smiselna.
10. Navedite primer, ko je za neko rešitev bolj smiselna implementacija abstraktnega razreda, in primer, ko je bolj smiselna implementacija vmesnika.

2.6.2 Naloge

Naloga 2.6.1. Definirajte razred `Porocilo` in vmesnik `Dokument`, ki ga `Porocilo` implementira. V vmesniku določite nujne (meta)lastnosti, ki jih običajno mora imeti vsak dokument, v razredu `Porocilo` pa jih ustrezno implementirajte.

Naloga 2.6.2. Definirajte še vmesnik `Workflow`, ki določa lastnosti in metode za pošiljanje poročila po delovnem toku, in ga implementirajte razredu `Porocilo` iz prejšnje naloge.

Naloga 2.6.3. Vmesniku `Workflow` definirajte še metodo `Sign` in jo eksplicitno implementirajte v razredu `Porocilo`, obenem pa v razred dodajte še metodo `Sign`, ki ni metoda vmesnika.

Naloga 2.6.4. Dopolnite Nalogo 2.5.3 tako, da pripravite vmesnik, ki bo ustrezno naslovil vse funkcionalnosti, ki bodo skupne vsem trem aplikacijam, da jih bodo razvijalski timi (vsako aplikacijo bo razvijala druga, specializirana skupina) lahko implementirali, ne da bi na katero pozabili. Na koncu vse razrede povežite v celoto. Naredite simulacijo namizne aplikacije s preprostim uporabniškim vmesnikom, ki bo sestavljen iz menuja v ukazni vrstici. Posameznih funkcionalnosti seveda (še) ni treba implementirati.

Naloga 2.6.5. Pripravili ste struct `Position`, ki vsebuje lastnosti `X` in `Y`, kateri prestavljata koordinati izbranega objekta. Dve instanci razreda želite med seboj primerjati in če imata enaki koordinati, ju razglasite za enaki. V ta namen povozite metodo `Equals`, ki jo predpisuje vmesnik `IEquatable<Position>`. Napišite struct in implementirajte vmesnik.

Naloga 2.6.6. Pripravite dva vmesnika. Prvi naj definira metode, ki jih potrebuje razred, kateri omogoča nakup oziroma rezervacijo vozne karte za poljuben tip prevoza. Drugi vmesnik naj vsebuje metode, ki omogočajo rezervacijo ustreznega sedišča na prevoznem sredstvu. Oba vmesnika naj vsebujeta metodo `Reserve`, ki ne dobi nobenega parametra. Nato implementirajte razred, ki bo poskrbel za funkcionalnosti, ki jih mora imeti sistem za rezervacijo vozne karte in ustreznega sedišča. Na koncu še kreirajte instanco tega objekta in pokličite vse njegove metode.

Naloga 2.6.7. V razvijalski ekipi smo zadolženi za pripravo abstraktnega nivoja razredov - vse realne podrazrede razvijajo ekipe študentov, mi pa zgolj pripravljamo ogrodje razrednega modela. Ker so ostale ekipe še neizkušene, jih ne bomo obremenjevali z implementacijami različnih vmesnikov, ampak bomo za to poskrbeli že na našem nivoju, njihovo delo bo zgolj ustrezna implementacija metod v podrazredih. Kako lahko to storimo? Naša naloga je priprava razrednega modela za program, ki bo skrbel za oddajo različnih poročil. Vsako poročilo mora biti pregledano, odobreno, izpolnjeno itd. Pripravite ustrezne abstraktne razrede in vmesnike, ki tem razredom pritičejo.

2.7 Razširitvene metode

2.7.1 Vprašanja

1. Kaj je razširitvena metoda? [Arh, Q33]
2. V kakšnem primeru nam pridejo prav?
3. Katere so posebnosti pri implementaciji razširitvenih metod?
4. Ali z razširitvijo lahko povozimo obstoječo metodo? Kako jo nato pokličemo?

2.7.2 Naloge

Naloga 2.7.1. Zapišite razširitveno metodo za "lep" izpis elementov seznama.

Naloga 2.7.2. Zapišite razširitveno metodo, ki prešteje število samoglasnikov v danem nizu.

Naloga 2.7.3. Zapišite razširitveno metodo z imenom `ToString`, ki vrača natanko obrnjen niz kot navadna metoda `ToString`. Obe metodi tudi pokličite in preverite pravilnost izpisa.

Naloga 2.7.4. Zapišite metodo za objekt, ki je definiran v razredu, do katerega nimate dostopa. Objekt predstavlja poročilo o poslovnem obisku, kar želimo, pa je enotna metoda, ki iz poročila samodejno izračuna dnevnice.

3 LINQ

3.1 Delegati in lambda izrazi

3.1.1 Vprašanja

1. Kaj je delegat oziroma kakšen je njihov namen?
2. Kaj nam naznani določilo `delegate`?
3. Opišite postopek definiranja novega tipa delegata in njegove uporabe kot parametra neke funkcije. Prikažite na primeru.
4. Kateri trije standardni tipi nadomestijo postopek definiranja novega tipa delegata?
5. Definirajte funkcijo kot objekt tipa `Func`.
6. Na katere tri načine lahko definiramo funkcijo? [Arh, Q57]
7. Kdaj uporabljamo enumeracijsko vrednost `MidpointRounding.AwayFromZero`?
8. Kakšna je razlika, če na objektu `IQueryable` uporabimo lambda izraz namesto npr. klica običajno definirane funkcije? [Arh, Q57]
9. Kako lahko skrajšamo sintakso definicije funkcije, če ta vsebuje zgolj stavek `return`?

3.1.2 Naloge

Naloga 3.1.1. Definirajte nov tip delegata, ki se sklicuje na funkcije, ki imajo dva vhodna parametra tipov `double` in `int`, vrnejo pa rezultat tipa `double`. Ustvarite instanco definiranega tipa (delegata) in mu podajte neko vrednost (funkcijo), nato pa izvedite klic delegata.

Naloga 3.1.2. Ustvarite funkcijo, ki kot parameter dobi seznam celih števil in dva različna tipa delegatov. Določite ji smiselno delovanje in jo izvedite.

Naloga 3.1.3. Definirajte funkcijo za iskanje največjega skupnega delitelja treh števil na dva načina.

Naloga 3.1.4. Ustvarite tri funkcije, pri čemer vsaka od njih kot parameter dobi seznam celih števil, kot drugi parameter pa naj ena dobi parameter tipa `Func<>`, druga `Action<>` in tretja `Predicate<>`. Določite jim smiselno delovanje in jih pokličite.

Naloga 3.1.5. S skrajšanim zapisom definirajte funkcijo, ki kot parameter dobi tri naravna števila, vrne pa njihovo povprečno vrednost.

3.2 Osnovna LINQ sintaksa

3.2.1 Vprašanja

1. Kakšna je osnovna struktura LINQ poizvedbe? [Arh, Q49]
2. Kakšna je sintaksa LINQ poizvedbe za urejanje in filtriranje elementov?
3. Kdaj je smiselno uporabiti anonimne tipe? [Arh, Q19]

3.2.2 Naloge

Naloga 3.2.1. Pripravite si svoj primer "podatkovne baze" (podobno kot imamo primer živali) študentov. Imajo naj lastnosti ime, priimek, datum rojstva, program in letnik študija ter seznam predmetov. Zadnji naj bo poseben objekt, ki poleg imena predmeta hrani še lastnost, če je predmet opravljen in s kakšno oceno.

Z LINQ poizvedbo pridobite študente, ki so se vpisali v lanskem letu, jih uredite padajoče po povprečni oceni ter izpišite samo njihova imena in povprečno oceno.

Naloga 3.2.2. V Nalogi 3.2.1 ste pri `select` naredili anonimni tip. V tej nalogi pripravite pomožen objekt, ki ima enake lastnosti kot anonimni, in ga uporabite pri `select` namesto anonimnega. Povozite tudi njegovo metodo `ToString`, da bo izpisovala enake vrednosti kot metoda anonimnega tipa.

3.3 LINQ s C# metodami

3.3.1 Vprašanja

1. Zapišite poizvedbo LINQ z urejanjem in filtriranjem s pomočjo razširitvenih metod. [Arh, Q50]
2. Ali je vrstni red metod pomemben za izvajanje? Kaj je lahko težava, če najprej pokličemo `Select` in šele nato `Where`?
3. Kako nam pomaga metoda `SelectMany`?
4. Razložite delovanje metode `Aggregate`.
5. Kakšna je razlika med metodama `All` in `Any`?

3.3.2 Naloge

Naloga 3.3.1. Napišite metodo, ki prebere vse vrstice izbrane tekstovne datoteke in v vsaki vrstici razbije zapis po presledkih oziroma tabulatorjih (uporabite metodo `Split`). Nato za dobljeno tabelo besed samo s pomočjo LINQ metod izberite tiste besede, ki vsebujejo črko 'a', jih uredite po abecedi in na koncu izpišite samo prve tri (oziroma največ tri, če so besede krajše) znake takih besed. Za izpis uporabite razširitveno funkcijo, ki izpisuje seznam nizov.

Naloga 3.3.2. Imamo podatke o uporabnikih knjižnice in knjigah, ki so si jih izposodili (za vsakega uporabnika seznam njegovih knjig oz. njihovih id-jev). Pripravite primer vhodne datoteke s temi podatki v obliki JSON, nato jo preberite in s pomočjo LINQ razširitvenih metod izpišite število različnih izposojenih knjig.

Naloga 3.3.3. Ustvarite si primer podatkovne tabele (samo v obliki seznama spremenljivk v seznamu kot na vajah) s podatki o gradivu v knjižnici, v kateri ima vsak zapis svoj enoličen identifikator, datum vnosa gradiva in datum spremembe zapisa, avtorja gradiva (samo njegov ID) ter celoštevilsko vrednost, ki podaja število izposoj gradiva. Ustvarite LINQ poizvedbe (uporabite njegove razširitvene metode) za naslednje naloge:

- Pridobite zapis, ki je bil nazadnje spremenjen.
- Pridobite gradivo, ki ga je avtor, ki je prvi po abecedi, najprej ustvaril.
- Pridobite vsa gradiva avtorja nazadnje dodanega gradiva.
- Pridobite število različnih avtorjev.
- Pridobite gradivo, katerega število izposoj je najbližje povprečni vrednosti vseh izposoj gradiv.

Naloga 3.3.4. Za podatke iz Naloge 3.3.2 s pomočjo metode `Aggregate` poiščite knjigo, ki je bila izposojena največkrat.

4 Načrtovalski vzorci

4.1 Splošno o načrtovalskih vzorcih (ang. *design patterns*)

Načrtovalski vzorci[8] so ponovno uporabljive rešitve podobnih problemov pri načrtovanju programskih rešitev. Gre za predloge oziroma opise načinov reševanja problemov. Za temeljno razdelitev se šteje 23 načrtovalskih vzorcev, ki jih je definirala "družba štirih" (The Gang of four - avtorji knjige Design Patterns: Elements of Reusable Object-Oriented Software), v grobem pa jih delijo na tri kategorije:

- načrtovalske vzorce za kreiranje (ang. *creational design patterns*)
- strukturne načrtovalske vzorce (ang. *structural design patterns*)
- vedenjske načrtovalske vzorce (ang. *behavioral design patterns*)

V teh zapiskih si bomo podrobneje ogledali dva iz prve in enega iz tretje kategorije.

4.1.1 Načrtovalski vzorci za kreiranje

Povzeto po:
https://dotnettutorials.net/lesson/creational-design-pattern/

Načrtovalski vzorci za kreiranje obravnavajo in omogočajo proces kreiranja instanc razredov na način, ki je najbolj primeren v trenutni situaciji. Običajno gre za omejevanje kreiranja instanc (na primer z omejevanjem dostopa do konstruktorjev in uporabo statičnih metod za klice konstruktorjev).

V praksi programske rešitve vsebujejo veliko razredov in posledično operiramo z mnogo instancami objektov na različnih mestih v kodi. Marsikdaj želimo na različnih mestih delati z isto instanco objekta, kar lahko dosežemo s shranjevanjem te instance v polje ali njenim podajanjem preko parametrov metod. Obstajajo pa boljše rešitve, ki nam pomagajo kodo bistveno poenostaviti, v veliko primerih pa celo pospešiti.

V osnovni razdelitvi najdemo naslednjih šest vzorcev za kreiranje (v angleščini):

- singleton,
- factory,
- abstract factory,
- builder,
- fluent interface,
- prototype.

[8] Slovensko izrazoslovje glede obravnavane tematike ni povsem poenoteno, zato v teh zapiskih uporabljam lastne prevode, spremljajo pa jih angleški izrazi.

4.1.2 Vprašanja

1. Kaj so načrtovalski vzorci (ang. *design patterns*)?
2. Na katere tri kategorije jih delimo?
3. Kaj omogočajo načrtovalski vzorci za kreiranje?
4. Naštejte vsaj tri tipe načrtovalskih vzorcev za kreiranje.

4.2 Vzorec singleton

Delno povzeto po:
https://csharpindepth.com/articles/singleton

Če želimo zagotoviti, da bomo imeli v našem programu zgolj oziroma največ eno instanco izbranega razreda, uporabimo vzorec singleton. Ideja singletona je, da ima izbrani razred zgolj privaten konstruktor, namesto konstruktorja pa razred izpostavi javno statično metodo, ki skrbi za instanco razreda.

Skupne lastnosti vzorca singleton so naslednje:

- Razred ima privaten konstruktor brez parametrov.
- Razred ima določilo `sealed`, kar pomeni, da se ga ne more dedovati.
- Ima statično spremenljivko, ki nosi referenco do morebitne edine instance razreda.
- Ima javno statično metodo (ali lastnost), ki to edino instanco posreduje klicatelju.

Spodnji razred *Singleton* je primer implementacije razreda, ki implementira vzorec singleton.

```
1  public sealed class Singleton
2  {
3    // Instanca, ki nadzoruje obstoj instance
4    private static Singleton uniqueInstance = null;
5
6    // Konstruktor, do katerega dostopamo samo znotraj razreda,
7    // brez parametrov
8    private Singleton()
9    {
10     // Vsaki instanci priredimo naključni ID
11     Random rnd = new Random();
12     this.RandomID = rnd.Next(1, 101);
13   }
14
15   // Javna lastnost vsake instance
16   public int RandomID { get; }
17
18   // Javna metoda, ki poskrbi za kreiranje instance,
19   // če še ne obstaja in jo vrne.
20   public static Singleton Instance()
21   {
22     // Če instanca še ni bila inicializirana,
23     // pokličemo konstruktor
24     if (uniqueInstance == null)
25       uniqueInstance = new Singleton();
26
27     return uniqueInstance;
28   }
29 }
```

V uporabniškem vmesniku preverimo, da res dobimo enako instanco na primer na naslednji način.

```
// Klic konstruktorja se ne prevede
//Singleton single = new Singleton();

// Pokličimo funkcijo enkrat
Singleton single1 = Singleton.Instance();
Console.WriteLine($"Naključni ID prve instance je: "
    + "{single1.RandomID}");

// In ponovno
Singleton single2 = Singleton.Instance();
Console.WriteLine($"Naključni ID druge instance je: "
    + "{single2.RandomID}");
```

In dobimo izpis:

```
Naključni ID prve instance je: 58
Naključni ID druge instance je: 58
```

Razred singleton lahko deduje druge razrede in implementira vmesnike, zato ima prednost pred statičnimi razredi. Njegova slabost se pokaže pri uporabi paralelizacije, ko hkrati dostopamo do edine instance. V takem primeru jo moramo ob dostopu zakleniti (`lock`).

Nekaj primerov uporabe:

- Razred za kontroliranje dnevniških (log) zapisov v datoteko.
- Povezava na podatkovno bazo.
- Pomnenje "fiksnih" vrednosti iz podatkovne baze, ki se ob izvajanju programa ne spreminjajo ("Caching").

Oglejmo si še primer implementacije razreda, ki skrbi za zapisovanje dogodkov v datoteko, ki ima v imenu čas zagona programa (oziroma čas prvega kreiranja instance zapisovalnika). Podoben primer je lokalno zapisovanje dnevnika za vsakega posameznega uporabnika, ki uporablja program.

```
public sealed class EventLog
{
  private static EventLog instance = null;

  private EventLog()
  {
    // Ime mape, kamor shranjujemo zapise,
    // pridobimo iz baze ali konfiguracijske datoteke
    string folderName = "";
    DateTime dtNow = DateTime.Now;

    // Pripravimo novo datoteko za zapisovanje
    this.LogFile =
      $"{folderName}EventLog-{dtNow:yyyy-MM-dd_HH-mm-ss}.txt";
  }
```

```
16
17    private string LogFile { get; }
18
19    public void WriteEvent(string evnt)
20    {
21      DateTime dtNow = DateTime.Now;
22      StreamWriter sw =
23        new StreamWriter(this.LogFile, true, Encoding.UTF8);
24      sw.WriteLine($"Event at {dtNow:HH-mm-ss}");
25      sw.WriteLine(evnt);
26      sw.WriteLine();
27      sw.Close();
28    }
29
30    public static EventLog Instance()
31    {
32      if (instance == null)
33      {
34        instance = new EventLog();
35      }
36
37      return instance;
38    }
39  }
```

Pokličimo funkcijo za zapisovanje v datoteko v uporabniškem vmesniku.

```
1  EventLog log = EventLog.Instance();
2  log.WriteEvent("Kreiramo nov dogodek.");
3  Thread.Sleep(1000);
4  log.WriteEvent("Kreiramo še enega.");
5  Thread.Sleep(1000);
6  log.WriteEvent("Počasi zaključujemo program.");
```

Tokrat ne dobimo izpisa v konzoli, temveč v datoteki.

```
Event at 00-55-26
Kreiramo nov dogodek.

Event at 00-55-27
Kreiramo še enega.

Event at 00-55-28
Počasi zaključujemo program.
```

4.2.1 Vprašanja

1. Kaj nam vzorec singleton omogoča?
2. Kako implementiramo vzorec singleton? Katere so njegove osnovne lastnosti? Prikažite na primeru.

3. V kakšnem primeru bi uporabili vzorec singleton?

4. Kakšne prednosti ima razred tipa singleton v primerjavi s statičnimi razredi, ki bi skrbeli za vračanje instance nekega razreda?

4.2.2 Naloge

Naloga 4.2.1. Napišite preprost program, ki ob zagonu uporabnika vpraša po nekaj fiksnih parametrih, ki si jih shrani v nek objekt, katerega v nadaljevanju ne moremo več spreminjati. Program nato v obliki zanke `while` uporabnika sprašuje o preprosta matematična vprašanja, npr. o vsoti ali produktu dveh naključno izbranih števil ter pričakuje njegov odgovor. Čas vsakega odgovora tudi izmeri. Spraševanje poteka, dokler se uporabnik dvakrat ne zmoti. Vsak odgovor uporabnika, čas in pravilnost, si program tudi zapomni v posebni datoteki. Pri implementaciji programa ustrezno uporabite vzorec singleton.

Naloga 4.2.2. Napišite razred, ki bo sledil vzorcu singleton in bo predstavljal osebni dokument (npr. potni list). Predpostavimo, da ima vsak posameznik lahko samo enega. Razred naj implementira vmesnik `IVerifiable`, ki zagotavlja metode za preverjanje pristnosti. Ustvarite še nek druga razred, npr. `EmploymentContract` (takih imamo več), ki ne sledi vzorcu singleton, vendar prav tako implementira vmesnik `IVerifiable`. Pripravite izvedbeno metodo, ki v nek seznam doda potni list in nekaj pogodb ter za vsak element seznama pokličite metodo za preverjanje pristnosti.

4.3 Vzorec factory

> Delno povzeto po:
> https://dotnettutorials.net/lesson/factory-design-pattern-csharp/

Z izrazom *factory* označujemo objekt/razred, ki ga uporabljamo za kreiranje drugih objektov. Podobno kot pri vzorcu singleton tudi pri vzorcu factory uporabniku onemogočimo dostop do konstruktorjev posameznih objektov, kreira jih lahko zgolj s pomočjo objekta factory. Gre za to, da lahko uporabnik med izvajanjem programa izbere, katerega od več sorodnih objektov želi kreirati.

Oglejmo si uporabo vzorca factory na primeru izpisa podatkov kreditnih kartic različnih tipov. Za potrebe primera si pripravimo vmesnik z lastnostmi, ki jih ima vsaka kreditna kartica ter tri razrede za različne tipe kreditnih kartic. Najprej si definirajmo enumeracijo, ki vsebuje vse možne tipe kartic.

```
1  public enum CreditCardType
2  {
3    Silver,
4    Gold,
5    Platinum
6  }
```

Nato pripravimo vmesnik.

```
1  public interface ICreditCard
2  {
3    CreditCardType CreditCardType { get; }
4    double Limit { get; }
5    double AnnualCharge { get; }
6  }
```

Sledijo trije razredi za vsak tip kartice.

```
1  class Silver : ICreditCard
2  {
3    public CreditCardType CreditCardType
4    {
5      get
6      {
7        return CreditCardType.Silver;
8      }
9    }
10
11   public double Limit
12   {
13     get
14     {
15       return 800;
16     }
17   }
18
```

```
  public double AnnualCharge
  {
    get
    {
      return 20;
    }
  }
}
```

```
class Gold : ICreditCard
{
  public CreditCardType CreditCardType
  {
    get
    {
      return CreditCardType.Gold;
    }
  }

  public double Limit
  {
    get
    {
      return 2000;
    }
  }

  public double AnnualCharge
  {
    get
    {
      return 50;
    }
  }
}
```

```
class Platinum : ICreditCard
{
  public CreditCardType CreditCardType
  {
    get
    {
      return CreditCardType.Platinum;
    }
  }

  public double Limit
  {
    get
    {
      return 5000;
    }
  }
```

```
18
19    public double AnnualCharge
20    {
21      get
22      {
23        return 100;
24      }
25    }
26  }
```

Če vzorca factory ne bi uporabili, bi v uporabniškem vmesniku kreirali instanco kreditne kartice na podlagi tipa, ki ga izbere uporabnik.

```
1  // Tip kartice se izbere v GUI-ju
2  CreditCardType type = CreditCardType.Silver;
3
4  // Pripravimo si novo spremenljivko
5  ICreditCard card = null;
6
7  // Ustvarimo instanco glede na izbrani tip
8  switch (type)
9  {
10   case CreditCardType.Silver:
11     card = new Silver();
12     break;
13   case CreditCardType.Gold:
14     card = new Gold();
15     break;
16   case CreditCardType.Platinum:
17     card = new Platinum();
18     break;
19 }
20
21 // Izpišemo podatke
22 Console.WriteLine("Podatki o kartici:");
23 Console.WriteLine($"  Tip: {card.CreditCardType}");
24 Console.WriteLine($"  Limit: {card.Limit}");
25 Console.WriteLine($"  Letni strošek: {card.AnnualCharge}");
```

Težava pri zgornjem načinu je, da so uporabniški vmesnik in razredi kreditnih kartic močno povezani. Če dodamo nov tip kartice, moramo to popraviti tudi v uporabniškem vmesniku, saj je treba dopolniti switch. Z uporabo vzorca factory pa bi logiko kreiranja prave instance prestavili v poseben razred *CreditCardFactory*.

```
1  static class CreditCardFactory
2  {
3    public static ICreditCard GetCreditCard(CreditCardType type)
4    {
5      // Pripravimo si novo spremenljivko
6      ICreditCard card = null;
7
8      // Ustvarimo instanco glede na izbrani tip
9      switch (type)
10     {
```

```
11          case CreditCardType.Silver:
12            card = new Silver();
13            break;
14          case CreditCardType.Gold:
15            card = new Gold();
16            break;
17          case CreditCardType.Platinum:
18            card = new Platinum();
19            break;
20        }
21        return card;
22    }
23  }
```

In tako bi koda v uporabniškem vmesniku preprosto izgledala takole.

```
1  // V uporabniškem vmesniku ohranimo samo logiko,
2  // ki se tiče uporabnika
3  CreditCardType type = CreditCardType.Silver;
4  ICreditCard card = CreditCardFactory.GetCreditCard(type);
5
6  // Izpišemo podatke
7  Console.WriteLine("Podatki o kartici:");
8  Console.WriteLine($"  Tip: {card.CreditCardType}");
9  Console.WriteLine($"  Limit: {card.Limit}");
10 Console.WriteLine($"  Letni strošek: {card.AnnualCharge}");
```

Tako v prvem kot v drugem primeru je izpis enak.

```
Podatki o kartici:
 Tip: Silver
 Limit: 800
 Letni strošek: 20
```

Vzorec factory je torej primerno uporabiti ko:

- imamo opravka z razredi, ki jim bomo dodajali podrazrede,
- nam uporabnik določi tip instance, ki jo moramo kreirati,
- lahko popravke objektov opravimo brez poseganja v uporabniški vmesnik.

4.3.1 Vprašanja

1. Kaj v programiranju označujemo z izrazom *factory*?
2. Kdaj je vzorec factory primerno uporabiti?
3. Opišite primer uporabe vzorca factory.
4. Na primeru na kratko opišite implementacijo vzorca factory.

4.3.2 Naloge

Naloga 4.3.1. Zapišite program, ki uporabniku omogoča izdelavo anketnih/izpitnih vprašanj. Zapise shranjujte v datoteke v obliki JSON zapisov s pomočjo knjižnic, ki omogočajo serializacijo in deserializacijo le-teh. Ena od možnosti programa naj bo tudi testno reševanje ankete. Pri implementaciji programa ustrezno uporabite vzorec factory.

Naloga 4.3.2. Pripravljamo aplikacijo za lokalni bar, kjer bo izbor koktejlov ponujen kar na tablici, na kateri se bo gost izbral napitek. Aplikacija bo imela preprost uporabniški vmesnik z vsemi napitki v ponudbi, pri čemer ga bomo razvili po navodilih lokalnega umetnika z veliko občutka za dizajn in zato uporabniškega v nadaljevanju vsaj nekaj časa ne bomo spreminjali. Za vse posodobitve ponudbe moramo poskrbeti v zalednem delu aplikacije. Ne smemo pozabiti, da bo aplikacijo uporabljal tudi barman, ki bo ob naročilu posameznega koktejla zraven dobil še recept za pripravo. Pripravite osnutek preproste verzije opisane aplikacije. Pri implementaciji ustrezno uporabite vzorec factory.

4.4 Strateški načrtovalski vzorec

Strateški načrtovalski vzorec (ang. *strategy design pattern*) nam pomaga pri gradnji, dopolnjevanju in popravkih razrednih modelov. Ko imamo neko razredno strukturo postavljeno, običajno dopolnitve od nas zahtevajo dodajanje enake funkcionalnosti samo nekaterim od razredov v strukturi, pri čemer želimo funkcionalnost implementirati le enkrat (npr. da se izognemo napakam pri popravljanju enake kode, ki se pojavi na več mestih) ter jo uporabiti v izbranih razredih.

Uporabo vzorca si oglejmo na primeru razredne strukture, kjer imamo kot temeljni razred abstraktni razred `Employee`.

```
1  public abstract class Employee
2  {
3    public Employee(string familyName, string givenName)
4    {
5      FamilyName = familyName;
6      GivenName = givenName;
7    }
8
9    public string FamilyName { get; set; }
10   public string GivenName { get; set; }
11
12   /// <summary>
13   /// Vsak zaposleni dobi plačo
14   /// </summary>
15   public void PaySalary(int amount, string bankAccount)
16   {
17     // Plača se nakaže na izbrani račun
18     Console.WriteLine($"Vaša plača (skupno {amount} eur) " +
19       "je bila izplačana danes, {DateTime.Now:d. M. yyyy}, " +
20       "na račun številka {bankAccount}");
21   }
22
23   /// <summary>
24   /// Vsak zaposleni govori tuj jezik
25   /// </summary>
26   public void SpeakForeignLanguage(string language)
27   {
28     Console.WriteLine($"Res je, govorim {language}");
29   }
30
31   public virtual void WorkDuties()
32   {
33     // Vsi zaposleni imajo neke skupne zadolžitve
34     Console.WriteLine($"Paziti moram, da ne zanetim požara!");
35   }
36 }
```

Vsak zaposleni ima ime in priimek, dobi izplačano plačo, govori tuj jezik in ima delovne zadolžitve.

Imamo tri podtipe zaposlenih, in sicer raziskovalce (`Researcher`), predavatelje (`Lecturer`) in osebe za stike z javnostmi (`PublicRelationsPerson`). Vsak od podtipov ima različne

delovne obveznosti in zato svojo implementacijo (virtualne) metode `WorkDuties`, ima pa tudi metode, ki so specifične le zanj. Na primer:

```
public class Researcher : Employee
{
  public Researcher(string familyName, string givenName)
    : base(familyName, givenName) { }

  public override void WorkDuties()
  {
    // Vsak poseben tip zaposlenega pa ima še posebne zadolžitve
    base.WorkDuties();
    Console.WriteLine($"Tudi raziskovati moram.");
  }

  /// <summary>
  /// Posebna metoda, ki jo ima le ta tip.
  /// </summary>
  public void WriteProjectApplication()
  {
    Console.WriteLine($"Pridno pišem raziskovalna vprašanja.");
  }
}
```

Zgornji razredni model je zadoščal zahtevam naročnika, *nato pa je prišla zahteva po dveh dodatnih funkcionalnostih.*

Najprej bi morali razredoma `Researcher` in `Lecturer` dodati lastnost `HIndex`. Gre torej za novo lastnost, ki ni skupna vsem podrazredom in zato ne spada v glavni nadrazred `Employee`.

Kot drugo moramo dodati nov tip zaposlenega, in sicer hišnika (`Janitor`), ki ima vse lastnosti kot ostali zaposleni, vendar ne govori tujega jezika. To pomeni, da moramo metodo v podrazredu prepisati oziroma jo dati na voljo samo razredom, ki jo potrebujejo.

Obe zahtevi bi lahko rešili tako, da bi pripravili vmesnika, ki bi zagotavljala ustrezne lastnosti in metode, vendar bi morali implementacije metod še vedno pripraviti v posameznih razredih. Tukaj uporabimo strateški načrtovalski vzorec.

Izbranim funkcionalnostim določimo skupne sklope in jih opišemo v vmesnikih. Na primer, funkcionalnosti, ki se tičejo znanja tujega jezika, lahko zberemo v enem vmesniku `IForeignLanguageSpeaker`.

```
public interface IForeignLanguageSpeaker
{
  void SpeakForeignLanguage(string language);

  void ReadForeignLanguage();
}
```

Nato za vsako "obnašanje" oziroma implementacijo funkcionalnosti uporabimo nov razred, ki implementira le te funkcionalnosti.

```
/// <summary>
/// Razred, ki implementira metodi za tekoče
/// govorjenje tujega jezika.
```

```
/// </summary>
public class SpeakForeignLanguageFluently
          : IForeignLanguageSpeaker
{
  public void SpeakForeignLanguage(string language)
  {
    Console.WriteLine($"Tuj jezik {language} govorim "
      + "tekoče dva dni skupaj!");
  }

  public void ReadForeignLanguage()
  {
    Console.WriteLine($"V tujem jeziku izvrstno berem!");
  }
}
```

```
/// <summary>
/// Razred, ki implementira metodi za manj tekoče govorjenje
/// tujega jezika.
/// </summary>
public class SpeakForeignLanguageSoSo : IForeignLanguageSpeaker
{
  public void SpeakForeignLanguage(string language)
  {
    Console.WriteLine($"Tuj jezik {language} govorim "
      + "bolj tako tako.");
  }

  public void ReadForeignLanguage()
  {
    Console.WriteLine($"V tujem jeziku berem bolje kot pišem!");
  }
}
```

```
/// <summary>
/// Razred, ki implementira metodi za negovorjenje tujega jezika.
/// </summary>
public class SpeakForeignLanguageNot : IForeignLanguageSpeaker
{
  public void SpeakForeignLanguage(string language)
  {
    Console.WriteLine($"Nič ne bo.");
  }

  public void ReadForeignLanguage()
  {
    Console.WriteLine($"V tujem jeziku berem, "
      + "ampak nič ne razumem!");
  }
}
```

V glavnem nadrazredu pripravimo instanco vmesnika `IForeignLanguageSpeaker` in metodo, ki pokliče ustrezno metodo instance.

```
public abstract class Employee
{
  .
  .
  .
  // Metodo SpeakForeignLanguage nadomestimo z instanco
  /// vmesnika IForeignLanguageSpeaker
  protected IForeignLanguageSpeaker SpeakForeign { get; set; }

  /// <summary>
  /// Pripravimo metodo, ki kliče metodo ustrezne
  /// instance IForeignLanguageSpeaker
  /// </summary>
  public void TrySpeakForeignLanguage(string language)
  {
    SpeakForeign.SpeakForeignLanguage(language);
  }
  .
  .
  .
}
```

Instanco ustreznega razreda za objekt `SpeakForeign` pa določimo v vsakem podrazredu posebej.

```
public class Researcher : Employee
{
  public Researcher(string familyName, string givenName)
    : base(familyName, givenName)
  {
    // Tukaj določimo ustrezno instanco
    // vmesnika IForeignLanguageSpeaker za ta razred
    SpeakForeign = new SpeakForeignLanguageFluently();
  }
  .
  .
  .
}
```

Na ta način vsakemu novemu podrazredu le dodamo ustrezno instanco objekta "obnašanja" z ustreznimi implementacijami. Če potrebujemo novo implementacijo, naredimo nov razred, ki implementira ustrezen vmesnik. Vsaka funkcionalnost bo tako implementirana le enkrat, izognemo se podvajanju kode in napakam.

4.4.1 Vprašanja

1. V kakšnih situacijah uporabljamo strateške načrtovalske vzorce?
2. Opišite zaporedje korakov, ki jim sledimo pri implementacijah po načelu strateškega načrtovalskega vzorca.
3. Podajte primer uporabe strateškega načrtovalskega vzorca.

4. Zakaj je določitev instance razreda za posamezno funkcionalnost (obnašanje) bolj primerna kot uporaba delegatov?

4.4.2 Naloge

Naloga 4.4.1. Pripravite razredni model, ki bo opisoval atlete. Atlet naj bo nadrazred, ki ima nekaj podrazredov (npr. metalec, šprinter, skakalec, deseterobojec). V glavnem razredu določite nekaj ustreznih funkcionalnosti (tek, met, skok), ki jih podrazredom ločeno implementirate glede na njihove specifike. Uporabite strateški načrtovalski vzorec.

Naloga 4.4.2. Pripravite abstrakten razred mobilna naprava, zanjo naredite nekaj podrazredov in implementirajte funkcionalnosti zanje. Npr. pošiljanje SMS-ov, telefoniranje, sprejemanje signala 4G ali celo 5G. Naprave naj bodo med seboj karseda različne, obenem pa naj model omogoča preprosto dopolnjevanje dodatnih funkcionalnosti in dodajanje novih podrazredov. Uporabite strateški načrtovalski vzorec.

5 Generiki

V mnogo primerih imamo opravka z objekti posebnih tipov, ki ne dedujejo nujno od istega razreda ali implementirajo istih vmesnikov, vseeno pa zanje napišemo metode, ki počnejo bolj ali manj iste stvari. Lep primer je "lep" izpis elementov iz seznama ali pa izpis vrednosti različnih enumeracij v uporabniškem vmesniku, ki ga uporabljamo na predavanjih. Še bolj pogosto smo uporabljali podatkovne strukture in njihove funkcije, katerim smo predpisali tipe elementov, ki jih vsebujejo. Da lahko kodo napišemo poenoteno, nam pomaga sintaksa generikov.

5.1 Osnovna sintaksa

Generiki (to so lahko razredi, strukture, vmesniki ali metode) nam omogočajo definiranje enakega obnašanja za različne tipe, pri čemer jim tipe podamo kot parametre (v kljukastih oklepajih).

Oglejmo si primer generičnega vmesnika, ki ga implementirajo razredi z enoličnim identifikatorjem.

```
1 public interface IUnique<T>
2 {
3   public T ID { get; set; }
4 }
```

Zgornji vmesnik implementiramo v dveh razredih. V prvem bo lastnost `ID` tipa `long`, v drugem pa tipa `int`.

```
1 public class Citizen : IUnique<long>
2 {
3   public Citizen(long emso)
4   {
5     ID = emso;
6   }
7
8   public long ID { get; set; }
9 }
```

```
1 public class Student : IUnique<int>
2 {
3   public Student(int enrolmentNumber)
4   {
5     ID = enrolmentNumber;
6   }
7
8   public int ID { get; set; }
9 }
```

Instanci obeh razredov kreiramo na naslednji način.

```
1 IUnique<long> obcan = new Citizen(010123500001);
2 IUnique<int> student = new Student(35230001);
```

Podobno lahko definiramo tudi generične razrede, če se na primer dani tip pojavi v več različnih metodah. Sintaksa je ves čas enaka - tip določimo v kljukastih oklepajih. Če imamo več parametrov za tipe, jih ločimo z vejico.

```
public class Article<T> : IUnique<T>
{
  public Article(T articleID)
  {
    ID = articleID;
  }

  public T ID { get; set; }

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

5.1.1 Vprašanja

1. Kako definiramo generični razred?
2. Kateri elementi jezika so lahko generični?
3. Navedite nekaj primerov, ko uporaba generikov poenostavi kodo.

5.1.2 Naloge

Naloga 5.1.1. Pripravite generično metodo, ki za dan seznam števil tipa `T` izračuna vsoto največjih k števil, pri čemer je k tudi parameter metode. V metodi uporabite le LINQ metode.

Naloga 5.1.2. Po dolgotrajnem študiju imate napisanih že precej knjižnic za obdelavo različnih podatkov. Težava pa je v tem, da v knjižnici oziroma programih nimate pravega reda oziroma strukture. Opazite sicer vzorec, da vhodne parametre vedno podate preko uporabniškega vmesnika, rezultate pa prav tako prikažete v njem. To pomeni, da za vsak tip obdelave podatkov oziroma problem uporabite poseben nabor parametrov (za vsak tip naj bodo zapisani v svojem razredu kot lastnosti), vaš program pa nato za izbrani problem prikaže možnosti vnosa parametrov ter izvede izračune. Potrebujete torej generični abstraktni razred `UserInterface`, za katerega bo tip parametra predstavljal razred parametrov izbranega problema. Razred `UserInterface` bo vseboval dve abstraktni metodi `ShowOptions` in `DoTask(T pars)`. Vsaka implementacija razreda, ki bo dedoval od razreda `UserInterface`, bo v `ShowOptions` implementirala pridobivanje oziroma določitev ustreznih parametrov, v metodi `DoTask` pa bo izvedla izračune in prikazala rezultate. Pripravite zgoraj opisane razrede in naredite vsaj eno realno implementacijo podrazreda.

6 Zbirke in podatkovne strukture

Dodatno branje:

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections
- Arh: Q43, Q45 in Q46

Pri upravljanju z velikimi količinami podatkov je način, na katerega jih shranjujemo, bistvenega pomena za učinkovitost naše programske opreme. V C# izbiramo med različnimi podatkovnimi strukturami, ki običajno že odgovarjajo našim potrebam, za naprednejše uporabnike pa je voljo tudi mnogo dodatnih NuGet knjižnic z bolj prilagojenimi strukturami.

V splošnem bomo objekte, ki omogočajo shranjevanje, branje in brisanje podatkov, imenovali *zbirke*.

6.1 Razlike med tabelami in zbirkami (ang. *collections*)

6.1.1 Vprašanja

1. Opišite bistvene razlike med tabelami in zbirkami ter v ta kontekst umestite še sezname tipa `List<T>`. [Arh, Q45]
2. Kakšna je razlika med `List` in `LinkedList`?
3. Opišite primer, ko je bolj smiselno uporabiti tabelo kot `List`.

6.2 Zbirke in njihovi vmesniki

Zbirke so dinamičen nadomestek za tabele, ki smo jih spoznali že pri osnovah. Medtem ko tabele upravljajo s točno določeno količino prostora, zbirke omogočajo dinamično povečevanje prostora, ko je trenutna kapaciteta zasedena.

Vsaka zbirka implementira vmesnik `ICollection<T>`, ki omogoča dodajanje (metoda `Add`), brisanje (metoda `Remove`) in čiščenje zbirke (metoda `Clear`). Vmesnik tudi zagotovi lastnost `Count`, s katero pridobimo število vsebovanih elementov.

```
ICollection<string> zbirka = new List<string>()
  { "Luka", "Jernej", "Dejan", "Denis",
    "Tilen", "Jernej", "Jakob", "Samo" };

Console.WriteLine($"Count pred Add: {zbirka.Count}");
zbirka.Add("Borut");
Console.WriteLine($"Count pred Remove: {zbirka.Count}");
zbirka.Remove("Borut");
Console.WriteLine($"Count pred Clear: {zbirka.Count}");
zbirka.Clear();
Console.WriteLine($"Count na koncu: {zbirka.Count}");
```

Dodatne specifike, ki jih posamezna zbirka (podatkovna struktura) omogoča, so zajete v dodatnih treh vmesnikih `IList<T>`, `ISet<T>` in `IDictionary<TKey, TValue>`. Pozor - vsi trije vmesniki podedujejo tudi vse člane vmesnika `ICollection<T>`!

Vmesnik `IList<T>` omogoča dostop do elementov zbirke z uporabo indeksov, vstavljanje elementa na dani indeks (metoda `Insert`) ter brisanje elementa z danega indeksa (metoda `RemoveAt`).

```
1 IList<string> seznam = (IList<string>)zbirka;
2
3 Console.WriteLine($"seznam[0]: {seznam[0]}");
4 seznam.Insert(0, "Borut");
5 Console.WriteLine($"seznam[0]: {seznam[0]}");
6 seznam.RemoveAt(1);
7 Console.WriteLine($"seznam[1]: {seznam[1]}");
```

Vmesnik `ISet<T>` zagotovi funkcionalnosti, ki jih pričakujemo od množice. Prva in najpomembnejša, na katero moramo paziti, je obstoj zgolj enega elementa z enako vrednostjo, pri čemer nas prevajalnik na to posebej ne opozori.

```
1 ISet<string> mnozica = new HashSet<string>()
2   { "Luka", "Jernej", "Dejan", "Denis",
3     "Tilen", "Jernej", "Jakob", "Samo" };
4
5 Console.WriteLine($"Število elementov: {mnozica.Count}");
```

Ob izpisu opazimo, da en element manjka.

```
Število elementov v množici: 7
```

Element "Jernej" smo vstavili dvakrat in drugi ni bil dodan.

Za takšne primere imamo na voljo novo metodo `Add`, ki ob izvedbi vrne vrednost `true` ali `false` glede na to, ali je bila uspešna ali ne.[9]

```
1 bool isInserted = mnozica.Add("Borut");
2 Console.WriteLine($"Element " +
3   $"{(isInserted ? "je bil dodan." : "ni bil dodan")}");
4 Console.WriteLine($"Število množici: {mnozica.Count}");
```

V tem primeru dobimo naslednji izpis.

```
Element je bil dodan.
Število elementov v množici: 8
```

Vmesnik `ISet<T>` implementira še nekaj drugih metod, ki omogočajo operacije nad množicami, na primer unijo (metoda `UnionWith`), presek (metoda `IntersectWith`) in razliko (metoda `ExceptWith`), ter metode, ki omogočajo preverjanje odnosov med množicami, na primer podmnožico (metoda `IsSubsetOf`) ali nadmnožico (metoda `IsSuperSetOf`).

Zadnji od vmesnikov, ki razširja vmesnik `ICollection<T>`, je vmesnik `IDictionary<TKey,TValue>`. Kot pove že ime, ga bomo uporabljali pri razredih, ki bodo implementirali lastnosti slovarjev. Opravka bomo imeli s podatki oblike ključ-vrednost. Ključ bomo uporabljali za dostop do vrednosti.

[9] Razmislite, kako pokličemo metodo `Add` iz vmesnika `ICollection<T>`, ki ne vrača vrednosti.

V slovar lahko elemente vstavljamo kar neposredno s pomočjo indekserja.

```
IDictionary<int, string> dnevi = new Dictionary<int, string>();

// Dodajmo nekaj vnosov
dnevi[1] = "ponedeljek";
dnevi[2] = "torek";
dnevi[5] = "petek";
```

Uporabimo lahko tudi metodo `Add`, kateri pa moramo podati tako ključ kot vrednost (ali pa ji podati instanco tipa `KeyValuePair`).

```
dnevi.Add(3, "sreda");
```

Če želimo pridobiti vrednost elementa z danim ključem, spet uporabimo funkcionalnost indekserja. V primeru, da ključ v slovarju ne obstaja, dobimo napako `KeyNotFoundException`.

```
// Vrednost dobimo s pomočjo indekserja,
// paziti pa moramo, da ključ v slovarju obstaja
Console.WriteLine($"3. dan v tednu: {dnevi[3]}");
Console.WriteLine($"4. dan v tednu: {dnevi[4]}"); // Vrže napako
```

Obstoj ključa zato vedno preverimo z metodo `ContainsKey`.

```
if(dnevi.ContainsKey(4))
  Console.WriteLine($"4. dan v tednu: {dnevi[4]}");
```

Elemente slovarjev lahko odstranimo zgolj z uporabo ključa, pri tem pa nam metoda `Remove` vrne vrednost `true` ali `false` glede na uspešnost operacije.

```
bool isRemoved = dnevi.Remove(1);
Console.WriteLine(
  $"Element {(isRemoved ? "je odstranjen" : "ni v slovarju")}");
```

Kot že rečeno, imajo elementi slovarja dva dela - ključ in vrednost. Posamezna seznama, zgolj s ključi oziroma vrednostmi, ki nastopajo v slovarju, sta nam na voljo preko lastnosti `Keys` oziroma `Values`.

```
var kljuci = dnevi.Keys;
var vrednosti = dnevi.Values;
```

6.2.1 Vprašanja

1. Kaj nam omogočata vmesnika `IEnumerable<T>` in `ICollection<T>`? [Arh, Q46]
2. Kateri trije vmesniki neposredno razširjajo vmesnik `ICollection<T>` in kakšne so razlike med njimi?
3. Naštejte dva razreda, ki implementirata `ISet<T>` in opišite njune značilnosti.
4. Opišite značilnosti vmesnika `IDictionary<TKey, TValue>` in opišite njegove značilnosti.
5. Kaj je lahko prednost in kaj slabost razreda `SortedDictionary<TKey, TValue>` v primerjavi z `Dictionary<TKey, TValue>`?
6. Opišite strukturi `Queue<T>` in `Stack<T>`. V kakšnem primeru je njuna uporaba smiselna?

6.2.2 Naloge

Naloga 6.2.1. Vrsta in sklad ne implementirata metode `Add`. Napišite razširitveno metodo zanju.

Naloga 6.2.2. Zapišite program, kjer od uporabnika zahtevate vnos besede s k črkami (k naj bo parameter, ki si ga izbere uporabnik), pri čemer mu naključno izberete prvo črko besede. Vnešene besede si zapisujte v skupno datoteko, da jo lahko po nekaj igrah uporabite za preverjanje vnosov (oziroma kot slovar).

Naloga 6.2.3. Na spletu poiščite seznam pošt in poštnih številk in pripravite podobno igro kot v prejšnji nalogi, kjer mora uporabnik glede na ime pošte ugotoviti njeno poštno številko. Katera podatkovna struktura bo najbolj primerna za implementacijo?

Naloga 6.2.4. Implementirajte svojo podatkovno strukturo `MyTree<T>`, ki predstavlja trojiško drevo, vedno pa vstavlja na prvo prosto mesto (se pravi, da gre za levo uravnoteženo trojiško drevo).

7 Hkratno programiranje

7.1 Osnovni pristopi k hkratnemu programiranju

7.1.1 Vprašanja

1. Kaj je hkratno programiranje (ang. *concurrent programming*), kaj večnitno (ang. *multithreaded programming*) in kaj asinhrono (ang. *asynchronous programming*)? [Arh, Q59]

2. Z uporabo razreda `Thread` na primeru pokažite, kako bi v istem programu zagnali še dva dodatna procesa.

3. Opišite funkcije `Start`, `Join` in `Sleep` v razredu `Thread`.

4. Z uporabo razreda `Task` na primeru pokažite, kako bi v istem programu zagnali še dve dodatni opravili.

5. Opišite funkciji `Run` in `Wait` v razredu `Task`.

6. Kako bi ustavili izvajanje opravila med izvajanjem? [Arh, Q63]

7. Kaj se zgodi, ko pokličemo lastnost `Result` nekega opravila?

7.1.2 Naloge

Naloga 7.1.1. Napišite program, ki hkrati izvaja dva procesa. Prvi proces vsakih 7 stotink sekunde poveča dani kot α za 1 radian, drugi proces pa vsakih 11 stotink sekunde poveča dani kot β za 4 radiane. Uporabniku se prikazujeta oba kota z modulom 2π, torej na nek način kot kota dveh urinih kazalcev. Uporabnik ima možnost prekinitve programa. Ko ga prekine, se mu izpiše razlika med kotoma. Cilj je, da gumb pritisne ob čim manjši razliki. Analizirajte, kako hitro se program odziva na prekinitev.

7.2 Ponavljajoč zagon procesa

7.2.1 Vprašanja

1. Za kakšne namene lahko uporabimo štoparice (ang. *Timers*)? [Arh, Q60]
2. Na primeru prikažite uporabo štoparice.

7.2.2 Naloge

Naloga 7.2.1. Napišite program, ki enkrat na dan zažene instanco brskalnika, ki odpre stran z novicami. (Pobrskajte, kako zaženemo nov proces (`Process`)).

7.3 Paralelizacija na zbirkah

7.3.1 Vprašanja

1. Kako izvajamo izračune na več objektih iz dane zbirke hkrati? Na kaj moramo biti pozorni? [Arh, Q62]
2. Kako določimo stopnjo paralelizacije?
3. Kaj dosežemo s stavkom `lock`? Prikažite njegovo uporabo na primeru.
4. V katerih primerih ne moremo uporabiti stavka `lock` in kako to rešimo pri popravkih vrednosti celih števil?
5. Kaj dosežemo z uporabo funkcije `AsOrdered`?
6. Kaj dosežemo s funkcijama `Skip` in `Take`?

7.3.2 Naloge

Naloga 7.3.1. Pripravite zbirko 1000 seznamov, kjer imate v vsakem od seznamov 10 000 naključno izbranih celih števil iz intervala od 1 do 100 000. Za vsakega izmed seznamov poiščite število praštevil, ki jih vsebuje, in izpišite maksimalno število praštevil v seznamih. Primerjajte čas izvajanja z zaporednim in vzporednim izračunom.

7.4 Asinhroni procesi

7.4.1 Vprašanja

1. Kaj dosežemo z uporabo stavka `await`? V kakšni metodi ga lahko uporabimo? [Arh, Q63]
2. Primerjajte izvajanje metod `Asynchronous.AsyncTest` iz kode predavanj. Kakšna je razlika med tremi primeri?
3. Kako prekinemo in kako spremljamo odstotek izvajanja opravila? Prikažite na primeru.
4. Kaj dosežemo z metodo `WhenAny`?

7.4.2 Naloge

Naloga 7.4.1. Napišite program, ki za danih pet spletnih strani (njihove naslove podajte v eni datoteki) prebere izvorno kodo, poišče vse vložene povezave (v atributih `href`) in odpre ter prebere spletne strani na teh povezavah. Za vsako od prebranih strani preveri, če katera od njih vsebujejo povezavo nazaj na začetno stran. Upoštevajte samo povezave na drugačni domeni. Za program smiselno uporabite pristope asinhronega programiranja.