

6. POGLAVLJE

Objektno orijentirano programiranje

U ovom poglavlju:

- Osnovni koncepti objektno orijentiranog programiranja
- Što je to učahurivanje, višeobličje, preopterećenje, prekoračenje i nasljeđivanje
- Određivanje prava pristupa članovima klasa
- Razlika između klasa, apstraktnih klasa i sučelja
- Kako nasljeđivati klase i nadograditi im funkcionalnost
- Kako implementirati sučelja

Koncept objektno orijentiranog programiranja najvažniji je koncept u .NET-u. Da biste mogli u potpunosti iskoristiti mogućnosti .NET-a, morat ćete naučiti objektno programirati. Ukoliko ste dosad programirali u jezicima kao što su C, Pascal ili Visual Basic, niste se još upoznali s tim konceptom – svu funkcionalnost programa sadržavale su posebne metode, a vi biste ih pozivali po potrebi. Koncept OOP-a je drukčiji i, kao što mu samo ime kaže, zasniva se na objektima i njihovim odnosima. U ovom poglavlju

II. DIO: OSNOVE PROGRAMIRANJA

upoznat ćete osnovne pojmove OOP-a koji će vam trebati i pri najosnovnijem .NET programiranju, stoga krenimo na posao.



Kad se govori o objektno orijentiranom programiranju u .NET-u, misli se na to da je svaki od .NET programskih jezika objektno orijentiran, a razlika postoji samo u sintaksi. Kako bi bilo preopširno detaljno obrađivanje mogućnosti svakog jezika, kao i u ostatku knjige, posvetit ćemo se C#-u.

Osnovni pojmovi OOP-a

Glavna stvar u objektno orijentiranom programiranju su *objekti*. Na objekte možete gledati kao na zasebne strukture koje sadržavaju neke povezane podatke i metode. Primjerice, radite li program koji simulira vožnju automobila, stvorit ćete poseban objekt za svaki auto. Taj objekt će sadržavati osnovne podatke, kao što su boja automobila, količina benzina u spremniku, njegova maksimalna brzina, broj osoba koje može prevoziti. Sadržavat će i neke metode: za ubrzavanje, usporavanje, zaustavljanje, skretanje i slično.

Ukoliko u svom programu želite simulirati neki automobil, jednostavno ćete stvoriti novi objekt te pozivati njegove metode za upravljanje vozilom, a nećete se morati brinuti o njihovoj funkcionalnosti, jer će ona biti u objektu. No ne mora kompletna funkcionalnost nekog objekta biti dostupna korištenju. Primjerice, skoro nikad nećete pri simuliranju vozila pozivati metodu za okretanje ključa u bravi, za prebacivanje u nultu brzinu, za paljenje svjetala i slično, već ćete to ostaviti metodi za pokretanje automobila. Ona će biti zadužena za pokretanje svih tih akcija, jer se to od nje i očekuje. Te druge akcije su, dakle, vama skrivene i koriste se samo interno unutar objekta.

Klase i objekti

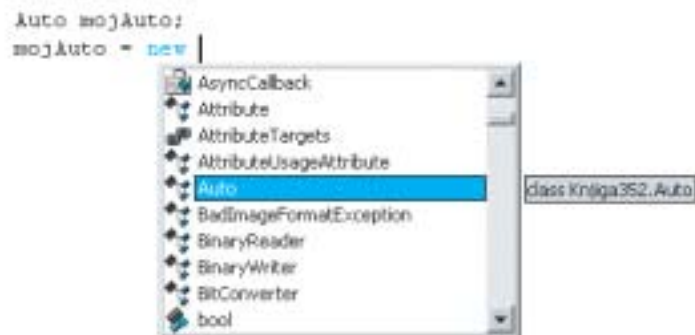
O klasama je već bilo riječi u prethodnim poglavljima, a one su ključne za razumijevanje i korištenje OOP-a u bilo kojem objektno orijentiranom jeziku, poput C++-a ili Delphija, a ne samo u .NET jezicima. Na klase se može gledati kao na predloške ili nacрте za objekte. U njima su točno definirani svi članovi objekta, njihova kompletna funkcionalnost, a postavljene su i inicijalne vrijednosti objekta potrebne za ispravan rad.

Pritom je vrlo važno upamtiti da se definiranjem klasa ne zauzima memorija, jer one služe samo kao nacrt. Tek se njihovim instanciranjem zauzima memorija, a ta stvorena instanca se naziva "objekt". Samo za podsjetnik, novi objekt se stvara na sljedeći način (pretpostavimo da je prije definirana klasa *Auto*):

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

```
// Deklaracija varijable tipa "Auto", ne zauzima se memorija
Auto mojAuto;

// Instanciranje novog objekta, zauzima se memorija
mojAuto = new Auto();
```



Slika 6-1:
Visual Studio automatski će vam pri stvaranju novog objekta ponuditi odgovarajuću klasu i tako olakšati pisanje kôda.

Svaki objekt je u potpunosti funkcionalna jedinica i sadrži sve potrebne podatke i varijable za svoj ispravan rad te otkriva svoju funkcionalnost *vanjskom svijetu*. Objekti su u stvarnom svijetu svuda oko nas – to su već spomenuti automobili, računala, instrumenti itd. Predstavljanje nekog objekta iz stvarnog svijeta objektom u programskom jeziku naziva se *apstrakcija*. Postupak pojednostavljenja objekata ključni je koncept programiranja.



Objektni modeli

Jednostavni objekti mogu sadržavati samo nekoliko varijabli, metoda ili događaja. Složeniji objekti mogu sadržavati čitav niz varijabli, gomile metoda, a može se pojaviti i potreba za podobjektima. Na primjer, automobil: on može sadržavati poseban objekt *Motor*, četiri objekta *Kotač* i slično. Kompozicija svih tih objekata čini objekt *Auto*, pa tako u slučaju da objekt *Motor* ima vrijednost atributa *Cilindri* 4, takav automobil će imati drugačije performanse nego objekt čiji podobjekt *Motor* ima za vrijednost atributa *Cilindri* 8. Tu cijela stvar ne prestaje – podobjekti mogu imati svoje podobjekte i tako u nedogled.

Hijerarhija takvih objekata naziva se *objektni model* i predstavlja strukturu i međusobne odnose povezanih objekata.

II. DIO: OSNOVE PROGRAMIRANJA



U slučaju da neki jezik podržava rad s objektima, nemojte da vas to prevari – to još uvijek ne znači da on podržava objektno orijentirano programiranje. Evo i ukratko mogućnosti koje jezik mora podržavati da bi bio objektno orijentiran:

- Mora podržavati objekte koji predstavljaju apstrakciju stvarnog svijeta. Oni moraju imati mogućnost spremanja i skrivanja svog stanja te sučelje (izložene metode) koje definira operacije nad objektom.
- Svi objekti moraju pripadati nekoj klasi.
- Treba biti podržano nasljeđivanje između klasa.

Učahurivanje ili enkapsulacija

Jedan od glavnih principa OOP-a je učahurivanje (engl. *encapsulation*, enkapsulacija). Da bismo ga objasnili, potrebno je vratiti se na opis neke klase. On se sastoji od dva dijela: sučelja (predstavlja pogled na neku klasu iz *vanjskog svijeta*) i implementacije (ugradnja mehanizama pomoću kojih se ispunjava funkcionalnost objekata opisana u sučelju). Koncept učahurivanja govori da je implementacija objekta potpuno neovisna o njegovu sučelju.

Aplikacija koja koristi vaš objekt komunicira s njime preko vidljivog joj sučelja. Sjetimo se primjera automobila – klasa *Auto* ima sučelje koje se sastoji od metoda za ubrzavanje, usporavanje, zaustavljanje i skretanje, a implementacija se nalazi u skrivenim objektima kao što su oni za okretanje ključa u bravi, paljenje svjetala i slično. Sve dok je sučelje isto, aplikacija može ispravno komunicirati i upravljati objektom, koliko god se implementacija mijenjala. Tako naknadno možete potpuno promijeniti funkcionalnost metode za paljenje svjetala ili je čak izbrisati, a aplikacija će dalje ispravno raditi, jer će pozivati metodu za pokretanje automobila koja i dalje postoji u sučelju.

Učahurivanje je, dakle, koncept koji kaže da ništa ne smije ovisiti o unutrašnjim detaljima nekog objekta. Objekti trebaju međusobno komunicirati isključivo preko javno dostupnih i vidljivih metoda i svojstava. Kasnije će u poglavlju biti detaljnije objašnjeno kako se postiže učahurivanje odnosno kako se skriva implementacija od *vanjskog svijeta*.

Višeobličje ili polimorfizam

Višeobličje (engl. *polymorphism*, polimorfizam) jest mogućnost da klase odnosno objekti na različite načine implementiraju ista sučelja. Drugim riječima, višeobličje omogućava pozivanje istih metoda i korištenje istih svojstava bez obzira na to o kojoj se implementaciji sučelja radi. Objasnimo sve na konkretnom primjeru, naravno, s automobilima. Recimo da vaša aplikacija komunicira s objektom *Auto* koji ima već opisano sučelje (ubrzavanje, usporavanje, zaustavljanje, skretanje). Ukoliko po-

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

stoji još objekata, kao što su *Kamion* ili *Brod*, koji imaju identično sučelje, s njima možete komunicirati na isti način, bez pridavanja posebne pozornosti tome o kojem se objektu radi. Naravno, kasnije u poglavlju će sve biti objašnjeno na primjerima.

Višeobličje sučelja

Ukoliko klasa u sebi ne sadrži implementaciju i konkretan kôd, već samo definiciju sučelja kao opis njenih mogućnosti, tada ju jednostavno nazivamo sučeljem (engl. *interface*). Sučelje, dakle, definira kakvu funkcionalnost treba imati neka klasa, no ne petlja se u njenu implementaciju.

Neki objekt može podržavati više sučelja, što znači da ima višestruku funkcionalnost. S njime je moguće komunicirati preko bilo kojeg od podržanih sučelja. Isto tako, više različitih objekata može podržavati isto sučelje, baš kao u prethodnom primjeru s automobilima.

Dakle, postoji sučelje *IVozilo* (prefiks *I* dolazi od *interface*; prema konvenciji, njime se uvijek označava sučelje) koje definira metode za ubrzavanje, usporavanje, zaustavljanje i skretanje. Ono ne implementira te metode, već isključivo navodi da ih podržava. Naša klasa *Auto* implementira sučelje *IVozilo* tako da ugrađuje funkcionalnost u obavezne metode. No isto tako ima smisla da sučelje *IVozilo* implementiraju i klase *Kamion* i *Brod*. S njima je, dakle, moguće komunicirati na isti način i nije uopće bitno o kojoj se klasi radi – kako sve one implementiraju isto sučelje, moguće je nad njima koristiti bilo koju metodu tog sučelja.

Nasljeđivanje

Koncept nasljeđivanja (engl. *inheritance*) dopušta da ugradite svu funkcionalnost već postojeće klase u neku novu klasu. Nasljeđivanjem nova klasa preuzima svu funkcionalnost od one koju je naslijedila – sve njene metode, varijable i druge članove. No novu klasu možete mijenjati i dodati joj novu funkcionalnost, a isto tako, što je mnogo važnije, možete *prekoračiti* postojeće nasljeđene metode i napisati potpuno nove s drugačijom funkcionalnošću.

Vratimo se opet na primjer automobila. Možete tako napraviti novu klasu *SportskiAuto* koja glavnu funkcionalnost nasljeđuje od klase *Auto* – sve njene metode i ostale članove. Na vama je samo da implementirate nove metode, primjerice za spuštanje pomičnog krova, a možete promijeniti i postojeće, primjerice napisati drugačiju metodu za zaustavljanje da više koristi kočnice i tako naglo koči.

Klasa u .NET-u može nasljeđivati od samo jedne klase, koja se tad naziva *baznom* klasom. No primijetite da zato klasa može nasljeđivati od više sučelja (naravno, pritom ima obavezu implementirati sve metode svih naslijeđenih sučelja).



II. DIO: OSNOVE PROGRAMIRANJA

Pristup članovima klase

Za izvedbu učajurivanja podataka i metode klasa te za otkrivanje funkcionalnosti nekog objekta *vanjskom svijetu* ključno je određivanje prava pristupa pojedinim članovima klase. U tablici 6-1 nalaze se ključne riječi za određivanje prava pristupa.

Tablica 6-1:
Ključne riječi za određi-
vanje prava pristupa
članovima klasa u C#-u

Ključna riječ	Utjecaj na člana
<code>public</code>	Član je dostupan svima
<code>private</code>	Član je dostupan samo unutar klase
<code>internal</code>	Članu se može pristupiti iz svih klasa unutar istog <i>assemblyja</i>
<code>protected</code>	Članu se može pristupiti samo iz klase koja ga definira i iz klasa koje od nje nasljeđuju
<code>protected internal</code>	Predstavlja uniju <i>protected</i> i <i>internal</i> ključnih riječi – članu je moguće pristupiti iz svih klasa unutar istog <i>assemblyja</i> te iz svih klasa koje nasljeđuju u klasu u kojoj je član definiran

Član deklariran s *public* ključnom riječi vidljiv je kompletnom kôdu izvan klase. Tako je moguće pozivati sve *public* metode neke klase i mijenjati sve *public* varijable od bilo kuda iz kôda. Dakle, sve metode deklarirane s *public* su dio sučelja i pomoću njih je moguće komunicirati s objektom. Članovi deklarirani s *private* vidljivi su samo unutar te klase i najčešće predstavljaju implementaciju, nevidljivu vanjskom svijetu. Vrlo je važna i *protected* ključna riječ, kojom dozvoljavamo pristup članu samo iz klase u kojoj je definiran i iz svih klasa koje ju nasljeđuju.



Pod *članovima* neke klase podrazumijevaju se sve njene metode, varijable i događaji koji pripadaju toj klasi.

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Tablica 6-2:
Pojednostavljena tablica 6-1, u kojoj je zorno prikazano u kojem je dijelu kôda dostupan član označen nekom ključnom riječi

Ključna riječ	U klasi u kojoj je definiran	U klasama koje ga nasljeđuju	U svim klasama istog assemblyja	Kodu izvan istog assemblyja
public	✓	✓	✓	✓
private	✓			
internal	✓		✓	
protected	✓	✓		
protected internal	✓	✓	✓	

Evo i konkretnog primjera kako se definiraju prava pristupa članovima neke klase.

```
public class Auto
{
    // Varijabli je dozvoljen bezuvjetan pristup
    public int BrojPrijedjenihKilometara;

    // Metodu mogu pozivati sve metode iz ove klase
    // i assemblyja, ali ne i vanjskog koda
    internal void NatociGorivo()
    {
    }

    // Varijabli mogu pristupati samo članovi ove klase
    private int TrenutnaBrzina;

    // Metodu mogu pozivati sve metode ove klase
    // i svih klasa koje ju nasljeđuju u
    protected void UpaliSvjetlo()
    {
    }
}
```

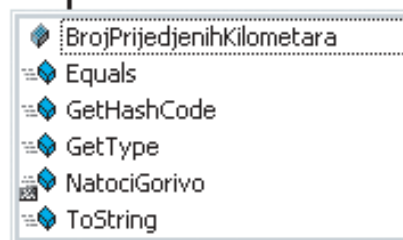
II. DIO: OSNOVE PROGRAMIRANJA



Slika 6-2:

Pri pisanju kôda automatski ćete dobiti popis svih članova kojima možete pristupiti – na popisu je vidljiva varijabla *BrojPrijedjenihKilometara* (*public*) i *NatociGorivo* (*internal*).

```
Auto mojAuto;  
mojAuto = new Auto();  
  
mojAuto.
```



Ukoliko u C#-u ne navedete ključnu riječ pristupa pri deklaraciji člana, njemu će pristup biti dozvoljen pod *private* uvjetima odnosno moći će mu se pristupiti samo unutar njegove klase.

I klasama pristupaju, zar ne?

Vjerojatno ste primijetili da u prethodnom primjeru i klasa ima oznaku prava pristupa. Kod njih je situacija vrlo slična kao i kod samih članova klase, samo na drugoj razini. Primjerice, *public* klase mogu se instancirati iz

bilo kojeg objekta u aplikaciji (što je i *defaultno* ponašanje, ukoliko se ne navede pravo pristupa), dok su *internal* klase dostupne samo unutar istog *assemblyja*.

Zajednički (*static*) članovi

Dosad smo se susreli samo s *običnim* članovima klase, tj. onima koji su jedinstveni za svaku instancu objekta. Primjerice, koristi li klasa varijablu *Broj*, svaki objekt te klase moći će imati spemljenu drugu vrijednost u tu varijablu.

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

```
PrimjerKlasa objekt1 = new PrimjerKlasa();
PrimjerKlasa objekt2 = new PrimjerKlasa();
objekt1.Broj = 10;
objekt2.Broj = 73;
```

Istovremeno su spremljene dvije vrijednosti odnosno svaki objekt ima spremljenu vlastitu vrijednost varijable *Broj*.

No moguće je i drugačije ponašanje – definiranje posebnih, zajedničkih varijabli. Te varijable će, dakle, biti zajedničke za sve objekte neke klase. Primjerice, da smo definirali da klasa ima zajedničku varijablu *Broj*, postojala bi samo jedna njena vrijednost za sve instance (a ne dvije, kao u gornjem primjeru, ili više).

Zajednička varijabla se definira korištenjem *static* ključne riječi:

```
public class PrimjerKlasa
{
    public static int Broj;

    // ...
}
```

Varijablu *Broj* i dalje je moguće koristiti u svim metodama klase, no postojat će samo jedna njena vrijednost, zajednička za sve instance.

Primijetite da *static* ključna riječ ne određuje prava pristupa, već samo je li član zajednički za cijelu klasu. Svi *static* članovi i dalje mogu biti *public*, *private* itd.



No da biste pristupili varijablama koje su zajedničke za cijelu klasu, morat ćete malo promijeniti sintaksu. Umjesto pristupa varijabli preko instance objekta, kao u prethodnim primjerima, za pristup ćete morati koristiti ime klase.

```
PrimjerKlasa objekt1 = new PrimjerKlasa();
PrimjerKlasa.Broj = 10;
```

Netočna sintaksa bi bila da pokušate promijeniti sadržaj zajedničke varijable naredbom `objekt1.Broj = ...`.

Primijetite i da metode mogu biti zajedničke za cijelu klasu, tj. za sve njene objekte. No kako one u tom slučaju ne pripadaju nekom objektu, ne mogu koristiti njegove varijable – u zajedničkim

II. DIO: OSNOVE PROGRAMIRANJA

metodama možete koristiti samo zajedničke varijable i, naravno, varijable prenesene metodama preko parametara ili deklarirane unutar same metode.



Kako zajedničke varijable pripadaju klasi, a ne nekom njenom objektu, njima je moguće pristupiti čak i prije nego što instancirate neki objekt te klase. Konkretno, u gornjem primjeru niste trebali stvoriti objekt *objekt1* prije no što ste upisali 10 u varijablu *Broj*. Ukoliko pak neki objekt klase sa zajedničkom varijablom stvorite nakon što već toj varijabli pridelite vrijednost, iz tog objekta ćete joj svedjedno moći pristupiti, jer je ona vezana za klasu.

Preopterećenje

Preopterećenje (engl. *overloading*) jedan je od ključnih koncepata objektno orijentiranog programiranja. Naime, iza te pomalo nejasne riječi skriva se vrlo korisna mogućnost – u .NET-u možete stvarati više članova neke klase koji imaju isto ime, no koriste se za različite stvari. Preopterećenje se najčešće koristi pri pisanju metoda. Primjerice, možete stvoriti dvije metode imena *Ispis*, od kojih jedna može imati samo jedan parametar, i to tekst koji treba ispisati, a druga metoda može imati dva parametra, s tim da drugi određuje koliko puta tekst treba ispisati.

Ključ je, dakle, u različitim parametrima koje primaju te istoimene funkcije. Evo kako biste izveli prethodni primjer:

```
public void Ispis(string tekst)
{
    Console.WriteLine(tekst);
}

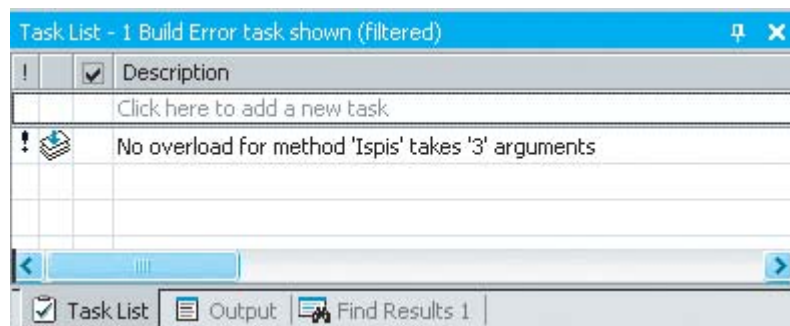
public void Ispis(string tekst, int broj)
{
    for (int i = 0; i < broj; i++)
        Console.WriteLine(tekst);
}
```

Dakle, preopterećene metode su metode s istim imenom, no s drugačijim parametrima. Parametri se mogu razlikovati u broju (konkretno, naš primjer ima jednu metodu s jednim parametrom, a drugu metodu s dva parametra) ili u svom tipu (primjerice, mogu obje metode imati samo jedan

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

parametar, no njegov se tip može razlikovati od metode do metode, pa tako može postojati metoda koja radi isključivo s brojevima, dok druga radi isključivo s tekстом).

Slika 6-3:
Ukoliko metodu *Ispis* pokušate pozvati s 3 parametra, javit će se pogreška, jer takva preopterećena metoda ne postoji.



Iako, dakle, preopterećene metode moraju imati drugačiji *potpis* (parametre), ne moraju sve vraćati istu vrijednost ili pak imati ista prava pristupa (jedna može biti *public*, druga *private* itd.). Pri izvršavanju programa i pozivu neke preopterećene metode provjeravaju se tipovi podataka koji su proslijeđeni pozvanoj metodi. Ukoliko se pak ne pronađe metoda koja prima poslone tipove, javlja se greška.

```
// Poziva se prva metoda
Ispis("Dobar dan!");

// Poziva se druga metoda
Ispis("Dobar dan!", 10);
```

Kao što je spomenuto u tekstu, preopterećenja metoda idu dalje od istoimenih metoda s različitim brojem parametara – češće je preopterećenje metoda s istim brojem parametara, no različitih tipova. To se spominje jer VB.NET omogućava definiranje *opcionalnih* parametara čije navođenje nije obavezno. Da smo naš primjer pisali u VB.NET-u, ne biste se morali brinuti preopterećenjem metoda, već biste jednostavno drugi parametar metode učinili opcionalnim i, ukoliko on ne bi bio naveden pri pozivu funkcije, smatralo bi se da je broj ispisivanja jednak 1.



II. DIO: OSNOVE PROGRAMIRANJA

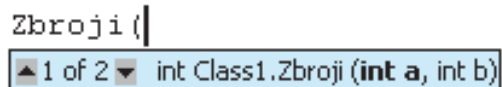
Dakle, vrlo je koristan i primjer u kojem postoje preopterećene metode s istim brojem parametara, no različitih tipova. Evo i primjera:

```
public int Zbroji(int a, int b)
{
    return a + b;
}

public string Zbroji(string a, string b)
{
    return a + " " + b;
}
```

Slika 6-4:

Korištenje preopterećenih metoda pri pisanju kôda – Visual Studio će vam ponuditi popis svih preopterećenih metoda napisanog imena, s tipovima njihovih parametara, a kroz njih se možete kretati korištenjem kursorских strelica prema gore i dolje.



```
Zbroji(|
1 of 2 int Class1.Zbroji (int a, int b)
```

Kao što uočavate, radi se o metodi za zbrajanje. Ukoliko joj se proslijede brojevi, ona će vratiti njihov zbroj. No ukoliko joj se proslijede znakovni nizovi, primjerice “Dobar” i “dan!”, ona će ih konkatenerati i između njih umetnuti razmak, što bi rezultiralo s “Dobar dan!”. Ovo je sasvim jednostavan primjer, no dovoljan za shvaćanje koncepta.

Mnoge često korištene metode iz .NET-a su preopterećene. Primjerice, pišete li aplikacije za konzolu, tj. bez grafičkog sučelja, često se susrećete s *Console.WriteLine* metodom koja ispisuje liniju teksta. Ona pak ima čak 19 preopterećenih metoda, koje se pozivaju ovisno o proslijeđenim parametrima (posebna metoda se poziva ukoliko želite ispisati broj, posebna ukoliko želite ispisati tekst itd.). Na slici 6-5 prikazana je pomoć pri pisanju te parametri za jednu od njih.

Slika 6-5:

Popis preopterećenih metoda za *Console.WriteLine*



```
Console.WriteLine(|
3 of 19 void Console.WriteLine (string format, object arg0, object arg1, object arg2)
format: The format string.
```

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Izgubljeni u metodama

Razvojna okolina Visual Studio predstavlja idealnu pomoć u pisanju kôda. Zahvaljujući IntelliSense tehnologiji, pri pisanju metoda nudi vam se pomoć u obliku opisivanja ulaznih parametara, opisa funkcionalnosti metoda i slično. Vi i za svoje metode možete pisati pomoć koja će se pojaviti pri pisanju njihovih poziva. Pozicionirajte se u redak iznad njihove definicije i napišite *///* (tri *slasha*) i pojavit će se mali XML kôd u koji možete upisati opis metode, opis svih parametara koje prima te opis rezultata koji vraća.

```
/// <summary>
/// Opis metode.
/// </summary>
/// <param name="a">Opis prvog
parametra.</param>
```

```
/// <param name="b">Opis drugog
parametra.</param>
/// <returns>Opis rezultata
metode.</returns>
```

Tako upisani tekst poslužit će vam u trenutku kad trebate opisanu metodu pozvati iz kôda. Naime, kao na slici 6-6, pojavit će se opis svih parametara koji će se mijenjati kako budete prelazili na drugi parametar. Isto tako, primaknete li miša već gotovom pozivu u kodu, pojavit će vam se *tooltip* koji opisuje njenu funkcionalnost. Želite li pisati uredan kôd, koji mogu i drugi koristiti, svakako pišite komentare uz svoje metode – samo iznad njihove definicije napišite spomenuta tri *slasha* (*///*) i pojavit će se predložak koji trebate ispuniti svojim komentarima.

```
Zbroji(15, 2
```

```
▲ 1 of 2 ▼ int Class1.Zbroji(int a, int b)
b: Drugi broj
```

Slika 6-6:

Pomoć pri pisanju poziva metoda izvući će se iz vaših komentara uz svaku metodu.

Preopterećenje operatora

U nekim specifičnim situacijama vrlo vam dobro može doći i mogućnost preopterećenja operatora. Možda poželite za neki tip podataka definirati drugačije ponašanje za standardne operatore, kao što su zbrajanje, množenje, usporedbe (veće, manje, jednako) ili logičke operacije. Uzmimo za primjer da radite s jednostavnom klasom koja prati ime i prezime te plaću svakog zaposlenika.

```
public class Zaposlenik
{
```

II. DIO: OSNOVE PROGRAMIRANJA

```
string ImePrezime;
int Placa;
}
```

Što ako želite na jednostavan način zbrojiti njihove plaće? To možete učiniti vrlo jednostavno – trebate samo preopteretiti operator zbrajanja. No prvo pogledajmo kako izgleda sintaksa preopterećenja operatora:

```
static public tip_podataka operator op (Parametar1 [, Parametar2])
{
    implementacija
}
```

Italik stilom označeni su dijelovi koje trebate sami promijeniti. Pod *tip_podataka* treba staviti tip koji vraća metoda. *Op* je operator, primjerice +, -, >, <, != itd. Primijetite da preopterećeni operator mora biti definiran kao *static*, što je i logično, jer je to zajednički operator za sve objekte tog tipa.



Svaki preopterećeni operator mora biti definiran kao *public* i *static*.

Dakle, preopterećeni operator mora biti definiran unutar same klase. Evo kako bi izgledao operator koji bi pri zbrajanju dvaju zaposlenika vratio zbroj njihovih plaća:

```
public class Zaposlenik
{
    public string ImePrezime;
    public int Placa;

    static public int operator + (Zaposlenik a, Zaposlenik b)
    {
        return a.Placa + b.Placa;
    }
}
```

Preopterećeni operator “+” vraća cjelobrojnu vrijednost, a kao parametre prima dva objekta, tj. dva zaposlenika. Njegova funkcionalnost je jednostavna – samo izračunava i vraća zbroj plaća dvaju

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

zaposlenika. U samom kodu više se ne trebate brinuti o zbrajanju – ukoliko kao parametre pošaljete objekte zaposlenika, tu će funkcionalnost obaviti netom definirani operator.

```
Zaposlenik a = new Zaposlenik();

a.ImePrezime = "Marko Marić";
a.Placa = 5700;

Zaposlenik b = new Zaposlenik();
b.ImePrezime = "Pero Perić";
b.Placa = 7100;

Console.WriteLine(a + b); // Ispisat će "12800"
```

Nasljeđivanje

Nasljeđivanje (*inheritance*) je koncept koji omogućava preuzimanje metoda, varijabli i ostalih članova od drugih klasa. Zahvaljujući nasljeđivanju, imate mogućnost stvaranja klasa koje implementiraju jednostavnu i svima zajedničku funkcionalnost, a zatim stvaranja novih specifičnih klasa koje nasljeđuju sve mogućnosti od tih jednostavnih, te ih nadograđuju novom funkcionalnošću.

Vratimo se na naš primjer s automobilima. Recimo da imate definiranu klasu *Automobil* koja u sebi ima definirane metode za pokretanje, zaustavljanje, skretanje te varijablu u kojoj je spremjena količina benzina u spremniku. Želite li pak napraviti klasu *Kamion*, i u njoj ćete morati definirati metode za pokretanje, zaustavljanje i skretanje te varijablu za količinu spremnika u benzinu. Mnogo je jednostavnije to izvesti nasljeđivanjem – klasa *Kamion* sve će članove naslijediti od klase *Automobil*, a dodat će i neke svoje nove članove.

Pogledajmo prvo kako izgleda klasa *Automobil*:

```
class Automobil
{
    public int Benzin;

    public void Kreni()
    {
        // implementacija
    }

    public void Stani()
    {
```

II. DIO: OSNOVE PROGRAMIRANJA

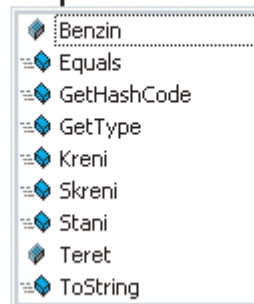
```
// implementacija
}

public void Skreni(int smjer)
{
    // implementacija
}
}
```

Naša nova klasa *Kamion* ima vrlo sličnu funkcionalnost kao i *Automobil*, no uz dodatak jednog novog člana, varijable zadužene za praćenje tereta u kamionu. Stoga će ona naslijediti klasu *Automobil*. Nasljeđivanje se vrši tako da se uz ime nove klase stavi dvotočka (":") i navede ime klase od koje se nasljeđuje.

```
class Kamion : Automobil
{
    public int Teret;
}
```

```
Kamion mojKamion = new Kamion();
mojKamion.
```



Slika 6-7:

Uočite da će objekt klase *Kamion* imati na raspolaganju i sve javne metode klase *Automobil* od koje nasljeđuje.



Klasa može nasljeđivati samo od jedne nad-klase (koja se u tom slučaju naziva baznom klasom), tj. u našem primjeru *Kamion* može nasljeđivati samo klasu *Automobil*. U prijevodu, neka klasa ne može svoju funkcionalnost naslijediti od više klasa, primjerice od klase *Automobil* i od klase *MotornoVozilo*.

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Iako vam se na prvi pogled može učiniti da klasa *Kamion* ima samo jedan član, to nije istina – ona je naslijedila svu funkcionalnost klase *Automobil*, baš kao što je vidljivo i na slici 6-7. Korištenjem nasljeđivanja vrlo lako možete nadograđivati funkcionalnost klasa.

S druge strane, jedna klasa može istovremeno nasljeđivati od više sučelja, jer sučelja ne implementiraju funkcionalnost, već to ostavljaju samim klasama, no to će biti objašnjeno kasnije u poglavlju.

No tu cijela priča ne prestaje. Klasu *Kamion* može naslijediti neka nova klasa *KamionPrikolica* koja će naslijediti svu njegovu funkcionalnost (dakle, uključujući i funkcionalnost klase *Automobil*) te dodati neku novu.

Glavni objekt

Svi objekti dostupni u .NET Frameworku i nastali iz vaših klasa nasljeđuju u od glavnog objekta, *System.Object* bazne klase. Tako ste vjerojatno na slici 6-7 uočili neke članove koji ne pripadaju vašoj klasi, što znači da ih je ona od nekog naslijedila. *System.Object* definira nekoliko metoda – *Equals* uspoređuje jednakost između dvije instance odnosno radi li se o istom objektu. Vrlo je važna i već spominjana *ToString*

metoda koja će ispisati tekst koji opisuje objekt. Primjerice, ispišete li *mojKamion.ToString()* za rezultat ćete dobiti *namespace* klase i njeno ime (primjerice, *mojeKlase.Kamion*). Ukoliko pak *ToString()* metodu pozovete nad nekim cijelim brojem, dobit ćete tekst koji sadrži njegovu vrijednost. Njega pak možete proslijediti nekoj metodi koja prima samo tekstovne parametre.

Iako po *defaultu* svaku klasu možete naslijediti u nekoj drugoj klasi, možete definirati i klase koje ne dozvoljavaju nasljeđivanje. Primjerice, možete u svojem programu koji se bavi automobilima definirati klasu *Letjelica* i onemogućiti njeno nasljeđivanje, jer ionako ne postoji niti jedna druga klasa (vozilo) koja bi mogla iskoristiti njene mogućnosti. Tada ćete napisati:

```
sealed class Letjelica
{
    // implementacija
}
```

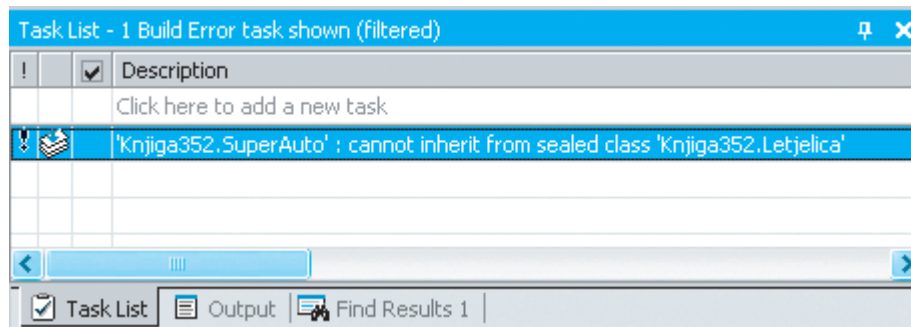
Među klasama u .NET biblioteci ima dosta *sealed* klasa, tj. onih koje ne dozvoljavaju nasljeđivanje. Stoga se nemojte čuditi ukoliko poželite nadograditi njihovu funkcionalnost, a kompajler

II. DIO: OSNOVE PROGRAMIRANJA

vam ne dozvoli. U kasnijim poglavljima ove knjige prijeći ćemo na naprednije teme i pokazivati kako nadograđivati funkcionalnost klasa .NET Frameworka, te je zato važno znati prepoznati klase koje to ne dozvoljavaju.

Slika 6-8:

Pokušate li naslijediti od klase koja to ne dozvoljava, kompajler će vam javiti grešku.



Prekoračenje članova klasa

Dakle, već smo naučili da klasa koja nasljeđuje neku drugu klasu preuzima svu njenu funkcionalnost. No uz puko preuzimanje njenih članova, nova klasa može svakog od njih prekoračiti (engl. *override*) i dati mu novu implementaciju. Tako možete zamijeniti metode i izraditi ih na drugi način.

Konkretno, klasa *Automobil* ima metodu *Kreni*, koja pokreće automobil. No klasa *Kamion* može definirati drugačije ponašanje za metodu *Kreni*. Iako ju je ona naslijedila, može je redefinirati i pridijeliti joj drukčiju funkcionalnost.

Prvo i najvažnije – u definiciji klase *Automobil* morate naznačiti da sve klase koje ju nasljeđuju mogu redefinirati odnosno prekoračiti metodu *Kreni*. To postizete ključnom riječju *virtual*.

```
class Automobil
{
    public virtual void Kreni()
    {
        // implementacija
    }

    // ...
}
```

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Sada u klasi *Kamion* samo trebate definirati novu metodu istog imena s istim parametrima i označiti je ključnom riječi *override*.

```
class Kamion : Automobil
{
    public override void Kreni()
    {
        // nova implementacija
    }

    // ...
}
```

```
class Kamion : Automobil
{
    public override
    {
    }
}
```

Slika 6-9:

Čim pri definiranju nove klase napišete ključnu riječ *override*, *Visual Studio* će vam ponuditi popis klasa koje možete prekoračiti.



Oprez, oprez, slijedi važna napomena! Iako bi se moglo zaključiti da metode koje nisu označene s *virtual* ne možete prekoračiti u novim klasama, to nije u potpunosti istina. Naime, mnoge metode iz baznih .NET klasa neće biti *virtual*, no vi ih svejedno možete prekoračiti tako da pri njihovoj deklaraciji u novim klasama navedete ključnu riječ *new*, primjerice:

```
internal new void Kreni() { // itd.
```

Tako zapravo stvarate novu implementaciju neke metode i potpuno ste neovisni o originalu iz bazne klase.

No samo metode koje su definirane s ključnom riječi *virtual* moći će se ponašati po konceptima višeobličja, što je objašnjeno kasnije u poglavlju. Ukoliko neku nevirtualnu metodu prekoračite ključnom riječi *new*, za nju neće vrijediti pravila višeobličja.

II. DIO: OSNOVE PROGRAMIRANJA

Ponekad ćete ipak poželjeti pozvati funkcionalnost bazne klase. Primjerice, ukoliko se metode *Kreni* u klasi *Automobil* i klasi *Kamion* razlikuju samo u tome što se u klasi *Kamion* provjerava je li teret ukrcan, jednostavnije će biti u prekoračenoj klasi dodati tu provjeru, a ostatak prepustiti baznoj klasi, tj. onoj u metodi *Automobil* koja je zadužena za paljenje svjetala, pokretanje vozila, prebacivanje u prvu brzinu itd.

Baznoj klasi se pristupa korištenjem ključne riječi *base*. Evo kako bi izgledala metoda *Kreni* u klasi *Kamion*:

```
class Kamion : Automobil
{
    public override void Kreni()
    {
        // provjeri teret

        base.Kreni();
    }

    // ...
}
```

Dakle, korištenjem ključne riječi *base* pozvala se metoda *Kreni* u baznoj klasi, tj. klasi *Automobil*. U njoj je definirana ostala funkcionalnost pokretanja vozila, a u klasi *Kamion* samo smo još dodali provjeru tereta (unutar komentara).



Tek kad razumijete nasljeđivanje možete u potpunosti razumjeti sve ključne riječi koje određuju prava pristupa metodama. Konkretno, možda je ostalo nejasno kakve su to *protected* klase. Primjerice, da ste u klasi *Automobil* metodu *Stani* definirali kao *private*, nju ne biste naslijedili u klasi *Kamion*, jer je ona, po definiciji, dostupna samo unutar klase u kojoj je definirana (dakle *Automobil*). No da ste je definirali kao *protected*, mogli biste joj pristupiti iz klase *Kamion*. Ipak i dalje to ne biste mogli iz glavnog programa, jer, po definiciji, *protected* članovima je dozvoljen pristup samo iz klasa u kojima su definirani te iz klase koje ih nasljeđuju.

Sučelja

Već prije smo objasnili što su *sučelja* (engl. *interfaces*) u OOP-u, no nije naodmet ponoviti. Dakle, radi se o točno definiranim članovima koji služe za komuniciranje sa samim objektom. Sučelje

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

samo definira članove koji će biti dostupni njegovom implementacijom, a samu implementaciju prepušta klasama koje ga nasljeđuju. Tako možete s različitim objektima koji implementiraju isto sučelje komunicirati na isti način.

Objasnimo sve na najpoznatijem primjeru iz literature. Dakle, pretpostavimo da postoji sučelje imena *IOblik* koje u sebi sadrži metodu za računanje površine. To sučelje mogu implementirati različite klase, primjerice klasa *Kvadrat* i klasa *Krug*. To u praksi znači da obje klase moraju sadržavati metodu za računanje površine, no svaka će je klasa izračunati na svoj način.

Iako biste taj primjer na prvi pogled, primjenjujući netom naučeno, mogli izvesti korištenjem klase, tj. tako da postoji klasa *Oblik* u kojoj je definirana metoda za računanje površine te da nju nasljeđuju klase *Kvadrat* i *Krug* i prekoračuju metodu za računanje površine, to ipak nije ispravno. Prvi razlog je u tome što klasa *Oblik* mora imati implementiranu metodu za računanje površine, što ne možemo očekivati, jer ne znamo o kakvom se obliku radi. Drugi razlog je što se nasljeđivanjem od klase *Oblik* novim klasama nikako ne može nametnuti obaveza da implementiraju svoju metodu za računanje površine, jer mogu koristiti i onu od klase *Oblik*, koju pak ne znamo implementirati bez informacije o kakvom se obliku radi. Sučelja rješavaju oba problema – ona samo definiraju metode koje klase koje ih nasljeđuju moraju imati, a pritom ih ne implementiraju (što nam treba za metodu za računanje površine) te svim klasama koje ih nasljeđuju nameću obavezu implementacije svih metoda definiranih u sučelju.

Definiranje sučelja

Sučelje se definira ključnom riječju *interface*. Za razliku od klasa, svi članovi moraju biti definirani bez ključne riječi koja određuje prava pristupa, poput *public*, *private* itd. Prava pristupa određuje samo ključna riječ navedena uz definiciju sučelja – napišete li *public interface*, svi će njegovi članovi biti *public*.

```
public interface IVozilo
{
    void Kreni();
    void Stani();
    void Skreni(int smjer);
}
```

Kao što vidite, definirano je sučelje *IVozilo*, no bez implementacije samih metoda. Sve su metode definirane svojim tipom podataka koji vraćaju (u našem slučaju, niti jedna ne vraća ništa i sve su *void*) te parametrima koje prima.

Klase se obavezuju implementirati sučelje tako da ga jednostavno naslijede. Sintaksa je ista:

```
class Automobil : IVozilo
{
```

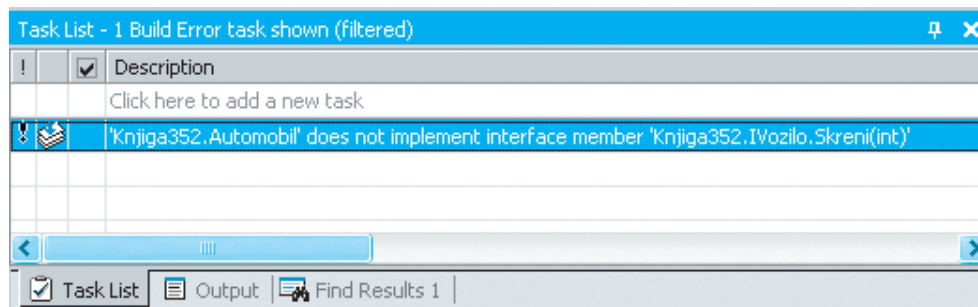
II. DIO: OSNOVE PROGRAMIRANJA

```
public void Kreni() {
    // implementacija
}

// obavezno implementirati sve metode iz sučelja IVozilo
}
```

Sve što trebate napraviti da bi klasa bila ispravna jest implementirati sve metode definirane u sučelju koje je naslijeđeno. Ukoliko to ne učinite, tj. ne napišete metode *Kreni*, *Stani* i *Skreni* istih parametara i povratnih tipova varijabli, kompajler će vam javiti grešku.

Slika 6-10:
Kompajler je javio grešku jer nismo implementirali metodu *Skreni*.



Primijetite da pri implementiranju sučelja ne trebate koristiti ključnu riječ *override*, jer ne prekoračujete postojeće metode. Vi ih samo implementirate i dajete im funkcionalnost, pa sve metode sučelja definirate na standardan način, baš kao što biste činili i da se ne nasljeđuje od sučelja.

No, za razliku od nasljeđivanja klasa, pri čemu možete naslijediti samo jednu klasu, sučelja su mnogo slobodnija – dozvoljeno vam je naslijediti koliko god sučelja želite. Naravno, tada morate implementirati svako od njih. Sučelja koja se nasljeđuju moraju se odvojiti zarezom pri definiciji.

```
class Kamion : IVozilo, ITeretnjak
{
```

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

```
// obavezno implementirati sve metode iz
// sučelja IVozilo i ITeretnjak
}
```

Ne možete stvoriti objekt iz sučelja. Sučelje u sebi, za razliku od klasa, ne sadrži implementaciju, pa ako pokušate instancirati novi objekt iz nekog sučelja (primjerice “`IVozilo`” `mojeVozilo = new IVozilo()`), dobit ćete informaciju o greški “`Cannot create an instance of the abstract class or interface`”.



Implementacija varijabli sučelja

Osim metoda, kao u prethodnim primjerima, sučelja mogu definirati i varijable koje svaka klasa mora implementirati. U tom je slučaju postupak malo složeniji nego pri jednostavnom deklariranju nove varijable. U sučelju možete definirati hoće li neka varijabla biti samo za čitanje, samo za pisanje ili oboje. Evo i kako:

```
public interface IVozilo
{
    int Benzin
    {
        get;
        set;
    }
}
```

Kao što vidite, u sučelju smo definirali *integer* varijablu *Benzin* koja je i za čitanje (tome služi *get*) i za pisanje (tome služi *set*). Ukoliko biste željeli definirati da se varijabla *Benzin* u klasama koje nasljeđuju sučelje ne može mijenjati iz ostatka programa, maknuli biste *set*, a ukoliko želite spriječiti njeno čitanje, maknut ćete *get*.

No tu nije kraj. Sad morate u svakoj klasi koja naslijedi to sučelje implementirati metode *get* i *set* za postavljanje vrijednosti varijabli. To ipak nije složeno kao što se čini.

```
class Automobil : IVozilo
{
    private int b;
```

II. DIO: OSNOVE PROGRAMIRANJA

```
public int Benzin
{
    get
    {
        return b;
    }
    set
    {
        b = value;
    }
}
```

Ključna je varijabla *b* koja je označena s *private* pravom pristupa. U nju će se interno spremati vrijednost stanja benzina. To se omogućava uz pomoć *get* i *set* metoda definiranih unutar vitičastih zagrada varijable *Benzin*. *Get* metoda je jednostavna i njen je zadatak pročitati varijablu *b* u kojoj je interno spremljena vrijednost benzina i vratiti je. Metoda *set* pak postavlja varijablu *b* na vrijednost *value* – radi se o vrijednosti koja se iz vanjskog programa želi upisati u varijablu *Benzin*.

Evo i primjera:

```
Automobil mojAuto = new Automobil();
mojAuto.Benzin = 15;
Console.WriteLine(mojAuto.Benzin);
```

Prvo se instancira objekt tipa *Automobil*. Zatim se upisuje vrijednost *Benzin* varijable. Iza scene se zapravo u tom trenutku poziva *set* metoda, koja u varijablu *b* upisuje proslijeđenu vrijednost odnosno 15. Uočite da je varijabla *b* označena s *private*, tj. ne može joj se pristupiti iz vanjskog programa i cijela komunikacija s njom ide preko varijable *Benzin*.

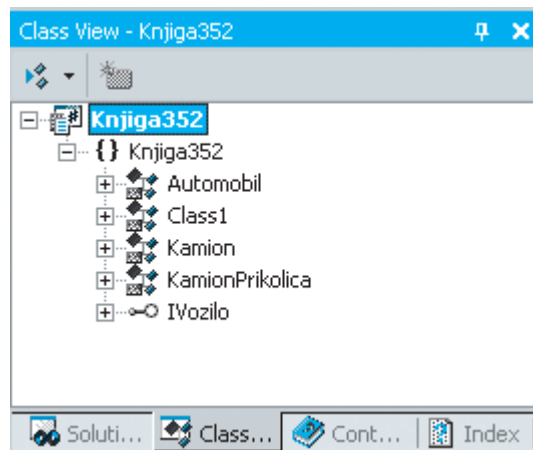
Dok ispisujemo vrijednost varijable *Benzin* poziva se njena *get* metoda koja vraća vrijednost varijable *b*. Naravno, vi možete cijelu stvar malo zakomplicirati te dodati još poneku funkcionalnost u *get* i *set* metode, no ovo će u većini slučajeva biti dosta.

Višeobličje

Evo nas napokon i na prvim primjerima objektno orijentiranog programiranja. Ono što slijedi najbolji je pokazatelj naprednih mogućnosti objektno orijentiranih jezika. Dakle, radi se o već prije objašnjenom konceptu, a sad ćemo ga iskušati i u praksi.

Već smo spominjali da koristeći metode nekog sučelja možemo komunicirati sa svim klasama koje ga implementiraju. Sljedeći primjer to najbolje pokazuje. Naime, napraviti ćemo metodu koja će

6. POGLAVLJE: OBJEKTNO ORIJENTIRANO PROGRAMIRANJE



Slika 6-11:
*Class View omogućava vam pregled svih
klasa i sučelja u vašem programu.*

Pogled svisoka

Osim ručnog pisanja kôda i stvaranja klasa i njihovih članova, možete se poslužiti i *Class Viewom* smještenim u desnom stupcu Visual Studia (ukoliko ga ne vidite, odaberite opciju *View – Class View* ili pritisnete tipkovničku kraticu **Ctrl + Shift + C**). Primjerice, da biste dodali novu varijablu u sučelje, kliknite na sučelje u *Class Viewu* desnim gumbom miša i odaberite opciju *Add – Add Property*. Pojavit će vam se prozor kao na slici 6-12, a u njemu možete definirati novu varijablu te odabrati je li samo za čitanje (*get*), za pisanje (*set*) ili oboje (*get/set*).



Naravno, *Class View* omogućava vam i druge korisne stvari. Želite li stvoriti novu klasu, kliknite desnim gumbom miša na najviši element u popisu koji odražava ime vašeg programa te odaberite opciju *Add – Add Class*. Otvorit će vam se prozor u kojem možete definirati sve postavke nove klase: prava pristupa, tip, odrediti koju klasu ili sučelje nasljeđuje itd.

Poigrajte se s *Class Viewom* jer skriva dosta korisnih opcija, koje mogu uvelike ubrzati stvaranje novih klasa i njihovih članova. Postupak je jednostavan – kliknite desnim gumbom miša na bilo koji element i proučite što vam se sve nudi.

Slika 6-12:
*Automatsko dodavanje varijable u
sučelje*

II. DIO: OSNOVE PROGRAMIRANJA

iskoristiti funkcionalnost nekog sučelja, a potpuno je svejedno objekt koje klase joj ga proslijediti, sve dok te klase implementiraju to sučelje.

Za početak pretpostavimo da imamo dvije klase, *Automobil* i *Kamion*, a obje implementiraju sučelje *IVozilo*. Pogledajmo metodu koja može komunicirati s obje klase:

```
void Voznja(IVozilo vozilo)
{
    if (vozilo.Benzin > 10)
    {
        vozilo.Kreni();
        vozilo.Skreni(1);
        vozilo.Stani();
    }
}
```

Radi se o metodi koja simulira neku vožnju. Pogledate li njenu implementaciju, uočit ćete da koristi sve metode sučelja *IVozilo*. Njoj možemo proslijediti objekt tipa *Automobil* i objekt tipa *Kamion*, jer oba implementiraju sučelje *IVozilo*, te stoga neće biti problema – vozilom će se normalno upravljati odnosno pozivati sve metode.

```
Automobil mojAuto = new Automobil();
Kamion mojKamion = new Kamion();

Voznja(mojAuto);
Voznja(mojKamion);
```

Objekti proslijeđeni metodi *Voznja* automatski će biti pretvoreni (*cast*) u objekt tipa *IVozilo*. Kako sučelje nameće obavezu implementacije svih njegovih članova, možete biti sigurni da sve metode i varijable korištene u metodi *Voznja* postoje te da će ona biti ispravno izvršena.

Dakle, to je pravo *višeobličje* – s različitim objektima koji implementiraju isto sučelje može se komunicirati na isti način koristeći metode tog sučelja.

Iako je prije bilo spomenuto da ne možete instancirati objekt iz sučelja (koristeći *new* operator), možete ipak deklarirati objekt tog tipa. Pogledajte primjer:

```
Automobil mojAuto = new Automobil();
IVozilo vozilo;

vozilo = (IVozilo) mojAuto;
vozilo.Kreni();
```

Apstraktne klase

I ako vam se može učiniti da su klase i sučelja sve što vam ikad može zatrebati pri programiranju, to nije sve! Po funkcionalnosti negdje između klasa i sučelja nalaze se *apstraktne klase*.

Možda ćete ponekad željeti stvoriti klasu koja implementira neke metode, no isto tako ostavlja implementaciju drugih metoda klasama koje ju nasljeđuju. Kao što vidite, radi se o kombinaciji običnih klasa (zato jer i same implementiraju neku funkcionalnost) i sučelja (jer prepuštaju implementaciju druge funkcionalnosti klasama koje ih nasljeđuju). Rezultat toga su, dakle, apstraktne klase, koje – poput sučelja – moraju biti naslijeđene i ne mogu se koristiti same za sebe (baš zbog tih neimplementiranih, ali definiranih metoda).

```
public abstract class
MojaApstraktnaKlasa
{
    public abstract void
ApstraktnaMetoda();
    public void
FunkcionalnaMetoda() {
        // implementacija
    }
}
```

Primijetite ključnu riječ *abstract* – njome se određuje apstraktna klasa. Ona također služi i za definiranje apstraktnih metoda. U primje-

ru je definirana *ApstraktnaMetoda* koju moraju implementirati sve klase koje nasljeđuju apstraktnu klasu (kao i kod svih metoda u sučeljima). No možete i definirati neku metodu s implementacijom (u našem primjeru *FunkcionalnaMetoda*) – njih možete označiti i s *virtual*, čime dozvoljavate da sve klase koje nasljeđuju apstraktnu klasu implementiraju svoju funkcionalnost tih metoda.

Pri nasljeđivanju i implementaciji apstraktne klase sve apstraktne metode morate prekoračiti korištenjem ključne riječi *override*. Tu se one razlikuju od sučelja, čije članove ne trebate prekoračiti korištenjem *override*, već samo implementirati na standardan način. Dakle, želite li naslijediti prethodnu klasu, evo kôda:

```
public class NovaKlasa :
MojaApstraktnaKlasa
{
    public override void
ApstraktnaMetoda()
    {
        // implementacija apstrakt-
        ne metode
    }
}
```

Primijetite da niste trebali naslijediti i implementirati metodu *FunkcionalnaMetoda*, jer njena implementacija već postoji u apstraktnoj klasi i jednostavno je od nje naslijeđena.

II. DIO: OSNOVE PROGRAMIRANJA

Objekt *mojAuto* eksplicitno je pretvoren (*cast*) u objekt tipa *IVozilo* (naravno, uvjet za ovo je da klasa *Automobil* implementira sučelje *IVozilo*). Objektu tipa *vozilo* dostupne su sve metode sučelja *IVozilo*, što je i očekivano, no uočavate li potencijalan problem? U sučelju *IVozilo* ne postoje njihove implementacije, ali to ne smeta – koristit će se od one klase iz koje je objekt pretvoren u objekt tipa *IVozilo*. Konkretno, koristit će se metode klase *Automobil*, jer je objekt *vozilo* prije pretvaranja bio tipa *Automobil*.

Slične mogućnosti vam se pružaju i pri korištenju klasa. Vratimo se na stariji primjer u kojem imamo klasu *Automobil* te klasu *Kamion* koji je nasljeđuje i nadopunjava novom funkcionalnošću. Dakle, klasa *Automobil* je bazna klasa. Sve klase koje ju nasljeđuju mogu se, zahvaljujući višeeobličju, ponašati kao ona.

Tako možete stvoriti sličnu metodu *Voznja*, koja će za parametar primiti objekt tipa *Automobil*. Ukoliko je pak pozovete prosljeđivši joj objekt tipa *Kamion*, on će biti automatski pretvoren u svoju baznu klasu. Naravno, sve dodatne metode koje je implementirala klasa *Kamion* postat će nedostupne, a vidljivi će biti samo članovi bazne klase.

Evo nas i na kraju poglavlja koje se bavi objektno orijentiranim programiranjem. Ono je bilo posve nužno za razumijevanje rada .NET-a jer, kao što ćete vidjeti u kasnijim poglavljima, gotovo svaki dio .NET-a, svaka njegova klasa i sve vaše želje za nadograđivanjem njihovih mogućnosti, rezultirat će potrebom za korištenjem koncepata OOP-a.

Primjerice, želite li napraviti vlastito tekstualno polje za Windows aplikaciju koje će, upišete li u njega tekst “tajna” promijeniti boju, trebate samo napraviti novu klasu koja nasljeđuje klasu za tekstualno polje i dodati joj tu sitnu funkcionalnost. To je, naravno, samo vrh ledenog brijega, a vašoj mašti je prepušteno istraživanje drugih mogućnosti.