DEO I

Osnove jezika C#

U OVOM DELU

POGLAVLJE 1

Osnove tipova

U OVOM POGLAVLJU

- Izrada klase
- Definisanje polja, svojstava i metoda klase
- Definisanje statičkih članova klase
- Dodavanje konstruktora
- Inicijalizovanje svojstava u konstruktoru
- Upotreba modifikatora `const` i `readonly`
- Višekratna upotreba istog koda u različitim konstruktorima
- Izvođenje nove klase od postojeće
- Pozivanje konstruktora osnovne klase
- Redefinisanje metode ili svojstva osnovne klase
- Deklarisanje interfejsa
- Realizovanje interfejsa
- Izrada strukture
- Izrada anonimnog tipa
- Zabrana instanciranja pomoću apstraktne osnovne klase
- Interfejs ili apstraktna osnovna klasa?

Ovo poglavlje objašnjava osnove izrade tipova u jeziku C#. Ako već poznajete C#, veći deo ovog poglavlja biće vam samo podsetnik.

Pošto obradimo elemente klasa kao što su polja, svojstva i metode, saznaćete šta su konstruktori, kako se definišu i realizuju (implementiraju) interfejsi i kada se koriste strukture. Informacije predstavljene u ovom poglavlju jesu osnovne, ali i ključne. Kao i u tolikim drugim oblastima, ako ne savladate osnove kako treba, ništa drugo se neće uklopiti.

Izrada klase

Zadatak: Treba da deklarišete klasu.

Rešenje: Počećemo deklaracijom jednostavne klase koja će sadržati 3D koordinate tačaka.

```
//standardni imenski prostori za uvoženje, koje Visual Studio
//umeće u svaku datoteku
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassSample

{
    //public - da bi klasa bila vidljiva izvan svog sklopa
    public class Vertex3d
    {
    }
}
```

Klasa koju smo definisali zasad je prazna, ali popunićemo je tokom ovog poglavlja.

Klasa je definisana kao javna (engl. *public*), što znači da je vidljiva u svakom drugom tipu koji referencira sklop u kojem se klasa nalazi. C# defininiše nekoliko modifikatora vidljivosti, koji su nabrojani u tabeli 1.1.

TABELA 1.1 **Modifikatori vidljivosti**

Vidljivost	Primenjivo na	Opis
Public	Tipove, članove tipova	Vidljivo svuda, čak i izvan sklopa
Private	Tipove, članove tipova	Vidljivo samo u kodu koji pripada istom tipu
Internal	Tipove, članove tipova	Vidljivo samo u kodu koji se nalazi u istom sklopu
Protected	Članove tipova	Vidljivo samo u kodu koji pripada istom ili izvedenom tipu
Protected internal	Članove tipova	Vidljivo samo u kodu koji se nalazi u istom sklopu ili u izvedenoj klasi u drugom sklopu

Ako vrsta vidljivosti klase nije izričato zadata, podrazumeva se `internal`.

Definisanje polja, svojstava i metoda

Zadatak: Definiciji klase treba da dodate polja, svojstva i metode.

Rešenje: Sada ćemo klasi `Vertex3d` dodati nešto korisno.

```
public class Vertex3d
{
    //polja
    private double _x;
    private double _y;
    private double _z;
    //svojstva
    public double X
    {
        get { return _x; }
        set { _x = value; }
    }
    public double Y
    {
        get { return _y; }
        set { _y = value; }
    }
    public double Z
    {
        get { return _z; }
        set { _z = value; }
    }
    //metoda
    public void SetToOrigin()
    {
        X = Y = Z = 0.0;
    }
}
```

Nekoliko napomena u vezi s prethodnim kodom:

- Sva polja su deklarisana kao `private` (privatna), što je uglavnom dobra praksa.
- Svojstva su deklarisana kao `public` (javna) ali im možete zadati i vidljivost tipa `private`, `protected` ili `protected internal`, kako vam odgovara.
- Svojstva mogu imati naredbu `get`, `set` ili obe.
- U svojstvima, `value` je argument koji se podrazumeva (u kodu metode).

U sledećem primeru, svojstvu X se dodeljuje vrednost 13 pomoću argumenta `value`:

```
Vertex3d v = new Vertex3d();
v.X = 13.0;
```

Automatski definisana svojstva

Često ćete nailaziti na sledeći šablon:

```
class MyClass
{
    private int _field = 0;
    public int Field { get { return _field; } set { _field = value; } }
}
```

C# ima skraćenu sintaksu za takve slučajeve:

```
class MyClass
{
    public int Field {get; set;}

    //moramo inicijalizovati vrednost u konstruktoru klase
    public MyClass()
    {
        this.Field = 0;
    }
}
```

NAPOMENA Ne možete imati automatski definisano svojstvo koje ima samo naredbu `get` (ako ne postoji pozadinsko polje, kako biste mu zadali vrednost?), ali naredba `set` može biti privatna:

```
Public int Field { get ; private set; }
```

Definisanje statičkih članova klase

Zadatak: Treba da definišete podatke ili metode koji su dostupni iz samog tipa, a ne isključivo iz pojedinačnih instanci klase. Statičke metode se često koriste i kao metode koje deluju na sve instance istog tipa.

Rešenje: Dodajte rezervisanu reč `static`, kao u sledećoj metodi koja sabira dva objekta klase `Vertex3d`:

```
public class Vertex3d
{
    ...
    public static Vertex3d Add(Vertex3d a, Vertex3d b)
    {
        Vertex3d result = new Vertex3d();
        result.X = a.X + b.X;
        result.Y = a.Y + b.Y;
        result.Z = a.Z + b.Z;
        return result;
    }
}
```

Statička metoda se poziva na način naveden u sledećem primeru:

```
Vertex3d a = new Vertex3d(0,0,1);
Vertex3d b = new Vertex3d(1,0,1);
Vertex3d sum = Vertex3d.Add(a, b);
```

Dodavanje konstruktora

Zadatak: Treba da automatski inicijalizujete nove objekte klase.

Rešenje: Definišite specijalnu metodu, koja se zove konstruktor i koja ima isto ime kao klasa, a nema povratni tip. Kôd konstruktora se izvršava kada pravite instancu klase – ali se nikad ne poziva direktno.

Slede dva konstruktora klase `Vertex3d` – jedan koji ima argumente, dok ih drugi nema, ali inicijalizuje objekte podrazumevanim vrednostima.

```
class Vertex3d
{
    public Vertex3d()
    {
        _x = _y = _z = 0.0;
    }

    public Vertex3d(double x, double y, double z)
    {
        this._x =x;
        this._y = y;
        this._z = z;
    }
}
```

NAPOMENA Konstruktori ne moraju uvek biti javni. Na primer, možete definisati konstruktor tipa `protected` koji je vidljiv samo u nasleđenim klasama. Možete definisati čak i privatni konstruktor koji sprečava instanciranje (za klase koje služe kao pomoćne ili nezavisne) ili koji je vidljiv samo u metodama iste klase (na primer, za statičke metode koje realizuju tzv. „model fabrike“).

Dodavanje statičkog konstruktora i inicijalizovanje objekta

Zadatak: Klasa sadrži statičke podatke koje treba inicijalizovati.

Rešenje: Statička polja možete inicijalizovati na dva načina. Jedan je pomoću statičkog konstruktora, koji je sličan standardnom konstruktoru, ali nema modifikatore vidljivosti, niti argumente:

```
public class Vertex3d
{
    private static int _numInstances;
    static Vertex3d()
    {
        _numInstances = 0;
    }
    ...
}
```

Međutim, zbog boljih preformansi, preporučljivo je da, kad god je to moguće, statička polja inicijalizujete u samoj deklaraciji polja, kao u sledećem primeru:

```
public class Vertex3d
{
    private static int _numInstances = 0;
    ...
}
```

Inicijalizovanje svojstava bez konstruktora

Zadatak: Vrednosti svojstava klase želite da inicijalizujete u trenutku deklarisanja promenljive, uprkos tome što konstruktor klase nema argumente koji to omogućavaju.

Rešenje: Upotrebite sintaksu za inicijalizovanje objekata, kao u sledećem primeru:

```
   class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
}

...
Person p = new Person()
                  { Id = 1, Name = "Ben", Address = "Redmond, WA" };
```

Upotreba modifikatora `const` i `readonly`

Zadatak: U klasi treba da definišete polja čije se vrednosti ne mogu menjati tokom izvršavanja koda

Rešenje: Oba modifikatora, `const` i `readonly`, mogu se upotrebiti za definisanje podataka koji se ne menjaju, ali postoje važne razlike između njih: poljima definisanim kao `const` morate zadati vrednosti u deklaraciji klase. Iz tog razloga, kao i zbog činjenice da su im vrednosti nepromenljive, ona se ponašaju kao statička polja. S druge strane, vrednosti

polja definisanih kao `readonly` mogu se zadati u deklaraciji klase ili u konstruktoru, ali ni na jednom drugom mestu.

```
   public class Vertex3d
{
    private const string Name = "Vertex";
    private readonly int ver;

    public Vertex3d()
    {
        ver = Config.MyVersionNumber;//U redu
    }

    public void SomeFunction()
    {
        ver = 13;//Greška!
    }
}
```

Višekratna upotreba istog koda u različitim konstruktorima

Zadatak: Imate više konstruktora, ali pošto im je deo funkcionalnosti zajednički, želite da izbegnete dupliranje koda. U C++ i nekim starijim jezicima, često se dešavalo da različiti konstruktori pozivaju isti blok inicijalizacionog koda. U takvim slučajevima, najčešće se zajednički kôd izdvajao u funkciju koju je svaki konstruktor pozivao.

```
//primer na jeziku C++
class MyCppClass
{
public:
    MyCppClass() { Init(); }
    MyCppClass(int arg) { Init(); }
private:
    void Init() { /* ovde dolazi zajednički kôd za inicijalizovanje*/ };
}
```

Rešenje: U jeziku C#, možete pozivati druge konstruktore iste klase pomoću rezervisane reči `this`, kao u sledećem primeru:

```
public class Vertex3d
{
    public Vertex3d(double x, double y, double z)
    {
        this._x = x;
        this._y = y;
        this._z = z;
    }
```

```
    public Vertex3d(System.Drawing.Point point)
        :this(point.X, point.Y, 0)
    {
    }
    ...
}
```

Izvođenje nove klase od postojeće

Zadatak: Treba da napravite uže specijalizovanu klasu tako što ćete postojećoj klasi dodati novo i/ili drugačije ponašanje.

Rešenje: Primenite nasleđivanje da biste iskoristili postojeću osnovnu klasu i dodajte joj novu funkcionalnost.

```
public class BaseClass
{
    private int _a;
    protected int _b;
    public int _c;
}

public class DerivedClass : BaseClass
{
    public DerivedClass()
    {
        _a = 1;//nije dozvoljeno! _a je privatan član klase BaseClass
        _b = 2;//ok
        _c = 3;//ok
    }

    public void DoSomething()
    {
        _c = _b = 99;
    }
}
```

Izvedena (nasleđena) klasa omogućava pristup javnim i zaštićenim (engl. *protected*) članovima svoje osnovne klase, ali ne i njenim privatnim članovima.

Pozivanje konstruktora osnovne klase

Zadatak: Imate nasleđenu klasu i želite da njen konstruktor poziva određeni konstruktor osnovne klase.

Rešenje: Slično pozivanju drugih konstruktora iz konstruktora iste klase, iz konstruktora nasleđene klase možete pozvati i određeni konstruktor osnovne klase. Ukoliko ne zadate nijedan konstruktor, poziva se podrazumevani konstruktor osnovne klase. Ako podrazumevani konstruktor osnovne klase zahteva argumente, morate zadati vrednosti za njih.

```
public class BaseClass
{
    public BaseClass(int x, int y, int z)
    { ... }
}

public class DerivedClass : BaseClass
{
    public DerivedClass()
    : base(1, 2, 3)
    {

    }
}
```

Redefinisanje metode ili svojstva osnovne klase

Zadatak: U nasleđenoj klasi želite da izmenite ponašanje osnovne klase.

Rešenje: Metodu ili svojstvo osnovne klase morate deklarisati kao `virtual` i oni moraju biti vidljivi u izvedenoj klasi. U izvedenoj klasi upotrebite rezervisanu reč `override`.

```
public class Base
{
    Int32 _x;
    public virtual Int32 MyProperty
    {
        get
        {
            return _x;
        }
    }

    public virtual void DoSomething()
    {
        _x = 13;
    }
}
```

```
public class Derived : Base
{
    public override Int32 MyProperty
    {
        get
        {
            return _x * 2;
        }
    }

    public override void DoSomething()
    {
        _x = 14;
    }
}
```

Objekti osnovne klase mogu da referenciraju instance osnovne klase ili instance svake izvedene klase. Na primer, sledeći blok koda prikazuje rezultat „28“ a ne „13“.

```
Base d = new Derived();
d.DoSomething();
Console.WriteLine(d.MyProperty().ToString());
```

Funkcije osnovne klase možete pozivati iz izvedene klase i pomoću rezervisane reči *base*.

```
public class Base
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base.DoSomething");
    }
}
public class Derived
{
    public virtual void DoSomething()
    {
        base.DoSomething();
        Console.WriteLine("Derived.DoSomething");
    }
}
```

Pozivanje metode Derived.DoSomething daje sledeće rezultate:

```
Base.DoSomething
Derived.DoSomething
```

Redefinisanje metoda i svojstava koja nisu deklarisana kao virtual

Zadatak: Funkcionalnost osnovne klase nije deklarisana s rezervisanom rečju `virtual`, ali vi želite da je ipak redefinišete u izvedenoj klasi. U nekim slučajevima može vam zatrebati da izvedete klasu iz biblioteke koja potiče iz nekog spoljnog izvora i želite da redefinišete određenu metodu, ali u osnovnoj klasi ta metoda nije deklarisana kao `virtual`.

Rešenje: Možete je ipak redefinisati, ali uz sledeće ograničenje: redefinisanu metodu pozivaćete isključivo preko reference na izvedenu klasu. Da biste to uradili, upotrebite rezervisanu reč new (u kontekstu drugačijem od onog na koji ste verovatno navikli).

```
class Base
{
    public virtual void DoSomethingVirtual()
    {
        Console.WriteLine("Base.DoSomethingVirtual");
    }

    public void DoSomethingNonVirtual()
    {
        Console.WriteLine("Base.DoSomethingNonVirtual");
    }
}
class Derived : Base
{
    public override void DoSomethingVirtual()
    {
        Console.WriteLine("Derived.DoSomethingVirtual");
    }
    public new void DoSomethingNonVirtual()
    {
        Console.WriteLine("Derived.DoSomethingNonVirtual");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Derived via Base reference:");

        Base baseRef = new Derived();
        baseRef.DoSomethingVirtual();
        baseRef.DoSomethingNonVirtual();

        Console.WriteLine();
        Console.WriteLine("Derived via Derived reference:");

        Derived derivedRef = new Derived();
        derivedRef.DoSomethingVirtual();
        derivedRef.DoSomethingNonVirtual();
    }
}
```

Rezultati prethodnog bloka koda izgledaju ovako:

```
Derived via Base reference:
Derived.DoSomethingVirtual
Base.DoSomethingNonVirtual

Derived via Derived reference:
Derived.DoSomethingVirtual
Derived.DoSomethingNonVirtual
```

Pokušajte da shvatite zašto su rezultati baš takvi kakvi jesu.

Deklarisanje interfejsa

Zadatak: Treba vam apstraktan skup funkcionalnosti, za koji nije definisano kako je realizovan i koji se može primeniti za više tipova.

Rešenje: Sledi primer jednostavnog interfejsa za objekat koji može da reprodukuje nešto snimljeno – na primer, audio ili video datoteku, pa čak i generički tok podataka. Interfejs ne opisuje šta je to nešto, nego više kako bi to trebalo da se ponaša.

```
public interface IPlayable
{
    void Play();
    void Pause();
    void Stop();
    double CurrentTime { get; }
}
```

Zapamtite da interfejsi mogu sadržati i metode i svojstva. (Mogu sadržati i događaje, što ćemo obraditi u poglavlju 15, „Delegati, događaji i anonimne metode“.) U interfejsima ne zadajete vrste vidljivosti za članove interfejsa, zato što su, po definiciji, oni svi javni.

Realizovanje interfejsa

Zadatak: Funkcionalnost definisanu u interfejsu treba da realizujete u obliku klase.

Rešenje: Da biste realizovali dati interfejs, potrebno je da u klasi deklarišete svaku metodu i svako svojstvo iz interfejsa, da ih označite kao javne i napišete kôd za njih.

```
public class AudioFile : IPlayable
{
    private IntAudioStream _stream;
    ...
    public void Play()
    {
```

```
        //detalji realizacije
        _stream.Play();
    }
    public void Pause()
    {
        _stream.Stop();
    }
    public void Stop()
    {
        _stream.Stop();
        _stream.Reset();
    }
}
```

NAPOMENA Visual Studio može da vam pomogne u ovom slučaju. Kada iza imena klase dodate `": IPlayable"`, preko toga se prikazuje Smart Tag. Ako Smart Tag pritisnete mišem, Visual Studio vam nudi opciju da u klasi generiše prazne članove interfejsa.

Realizovanje više interfejsa u istoj klasi

Zadatak: Ista klasa treba da realizuje više interfejsa, pri čemu postoji mogućnost dupliranja imena metoda.

Rešenje: Klasa može da realizuje više interfejsa, koje treba navesti razdvojene zarezima:

```
public class AudioFile : IPlayable, Irecordable
{
...
}
```

Međutim, može se dogoditi da je ista metoda definisana u dva (ili više) interfejsa. U našem malom primeru, pretpostavimo da metoda `Stop()` postoji i u interfejsu `IPlayable` i u interfejsu `IRecordable`. U tom slučaju, jedan od interfejsa morate izričito navesti.

```
public class AudioFile : IPlayable, Irecordable
{
    void Stop()
    {
        //iz interfejsa IPlayable
    }

    void IRecordable.Stop()
    {
        //iz interfejsa IRecordable
    }
}
```

Te dve metode pozivali biste na sledeći način:

```
AudioFile file = new AudioFile();
file.Stop();//poziva verziju iz intefejsa IPlayable
((IRecordable)file).Stop();//poziva verziju iz interfejsa IRecordable
```

Imajte u vidu da smo proizvoljno odlučili da metoda `Stop()` iz interfejsa `IRecordable` treba da bude eksplicitno navedena – mogli ste isto tako odlučiti i da metoda `Stop()` interfejsa `IPlayable` bude ona koju ćete eksplicitno navesti.

Izrada strukture

Zadatak: Potreban vam je tip s malom količinom podataka i bez režijskog opterećenja koje donosi klasa.

Za razliku od jezika C++, gde su strukture i klase funkcionalno identične, u jeziku C# postoje važne i suštinske razlike:

- Strukture su vrednosni tipovi, nasuprot referentnim tipovima, što znači da se one čuvaju na steku. Zauzimaju manje memorije i prikladne su za predstavljanje manjih struktura podataka. Osim toga, ne morate ih deklarisati s operatorom new.
- Od struktura ne možete da pravite izvedene tipove. One su po definiciji nenasledive (videti poglavlje 2, „Izrada svestranih tipova“).
- Strukture ne mogu imati konstruktor bez parametara. On već postoji implicitno i inicijalizuje na nulu sva polja u strukturi.
- Svaki konstruktor strukture mora inicijalizovati sva polja strukture.

Pri pozivanju metoda, strukture se prosleđuju po vrednosti, isto kao i svaki drugi vrednosni tip. Budite oprezni s opsežnim strukturama.

Rešenje: Struktura se definiše slično kao klasa:

```
public struct Point
{
    private Int32 _x;
    private Int32 _y;

    public Int32 X
    {
        get { return _x; }
        set { _x = value; }
    }

    public Int32 Y
    {
        get { return _y; }
        set { _y = value; }
    }
```

```
    public Point(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public Point() {} //Ovo nije dozvoljeno!
    //Ni ovo nije dozvoljeno! Ne inicijalizujete _y
    public Point(int x) { this._x = x; }
   }
```

Strukture se mogu koristiti u sledećem obliku:

```
Point p;//definiše, ali ne inicijalizuje
p.X = 13;//ok
p.Y = 14;

Point p2 = new Point();//inicijalizuje p2
int x = p2.X;//x će biti nula
```

Definisanje anonimnog tipa

Zadatak: Želite da definišete jedinstven tip za privremenu upotrebu, kojem ne treba ime.

Rešenje: Rezervisana reč `var` omogućava deklarisanje anonimnih tipova čija svojstva definišete neposredno iza tipa.

```
class Program
{
    static void Main(string[] args)
    {
        var part = new { ID = 1, Name = "Part01", Weight = 2.5 };
        Console.WriteLine("var Part, Weight: {0}", part.Weight);
        Console.WriteLine("var Part, ToString(): {0}", part.ToString());
        Console.WriteLine("var Part, Type: {0}", part.GetType());
    }
}
```

Program prikazuje sledeće rezultate:

```
var Part, Weight: 2.5
var Part, ToString(): { ID = 1, Name = Part01, Weight = 2.5 }
var Part, Type:
➥ <>f__AnonymousType0`3[System.Int32,System.String,System.Double]
```

U poglavlju 3, „Opšte tehnike kodiranja“, potražite više informacija o upotrebi rezervisane reči `var`.

NAPOMENA Na prvi pogled, izgleda kao da `var` nema definisan tip, ali je to pogrešan utisak. Kompajler će generisati strogo tipiziran objekat. Pogledajte sledeći primer koda:

```
var type1 = new { ID = 1, Name = "A" };
var type1Same = new { ID = 2, Name = "B" };

var type2 = new { ID = 3, Name = "C", Age = 13 };

type1 = type1Same; //Ok
type1 = type2; //Nije ok
```

Za ovaj poslednji red, prevodilac će javiti sledeću grešku:

```
Cannot implicitly convert type 'AnonymousType#2' to 'AnonymousType#1'
```

Zabrana instanciranja klase pomoću apstraktne osnovne klase

Zadatak: Treba vam osnovna klasa koja će sadržati zajedničku funkcionalnost, ali ne želite da se ta klasa direktno instancira.

Rešenje: Označite klasu kao apstraktnu.

```
public abstract MyClass
{
    ...
}
public MyDerivedClass : MyClass
{
    ...
}
MyClass myClass = new MyClass(); //ovo nije dozvoljeno!
MyClass myClass = new MyDerivedClass(); //ovo je u redu
```

Pojedine metode unutar apstraktne klase možete označiti kao apstraktne kako ne biste morali da pišete za njih podrazumevani kôd, kao u sledećem slučaju:

```
public abstract MyClass
{
public abstract void DoSomething();
}
MyClass myClass = new MyClass();//ovo nije dozvoljeno!
```

Interfejs ili apstraktna osnovna klasa?

Kada projektujete hijerarhije klasa, često je potrebno da odlučite da li klase koje se nalaze na nivou korena hijerarhije (roditeljske klase) treba da budu apstraktne osnovne klase, ili da umesto konkretnih klasa definišete interfejse.

Sledećih nekoliko smernica pomoćiće vam da donesete odluku.

U prilog interfejsima:

- Da li će biti potrebno da klase nasleđuju istovremeno više osnovnih klasa? To nije moguće u jeziku C#, ali je moguće da jedna klasa realizuje više interfejsa istovremeno.
- Da li ste u razmišljanju došli do tačke kada dobro shvatate razliku između onoga što klasa *predstavlja* i onoga što klasa *radi*? Interfejsi često definišu šta klasa radi, dok osnovna klasa može da definiše šta ona predstavlja.
- Interfejsi često ne zavise od toga šta klasa predstavlja i mogu se koristiti u mnogim situacijama. Mogu se dodati klasi a da se ne vodi računa o tome šta klasa predstavlja.
- Interfejsi često omogućavaju dizajn s veoma labavo spregnutim elementima.
- Izvođenje prevelikog broja klasa iz jedne osnovne klase može dovesti do toga da budete „preplavljeni“ s previše usitnjenom funkcionalnošću.

U prilog osnovnim klasama:

- Da li postoji razumna količina zajedničke funkcionalnosti ili podataka, upotrebljiva u svim izvedenim klasama? U tom slučaju, apstraktna osnovna klasa može biti dobro rešenje.
- Realizovanje istog interfejsa u većem broju različitih tipova može biti uzrok čestog ponavljanja koda, dok apstraktna osnovna klasa može da grupiše zajednički kôd na jedno mesto.
- Apstraktna osnovna klasa može da definiše podrazumevani oblik realizovanja.
- Apstraktne osnovne klase nameću strogu strukturu koda, što u nekim slučajevima može biti poželjno.
- Ako ustanovite da ugrađujete previše funkcionalnosti u apstraktne osnovne klase, druga mogućnost je da pokušate da pojedine funkcionalne oblasti realizujete u obliku komponenata. Kad steknete više iskustva, utvrdićete šta je najbolje rešenje u pojedinim situacijama. Često je najbolja određena kombinacija obe mogućnosti.