

# Tehnike programiranja

(napredniji koncepti iz OOP-a)

2016/17.05

## ➤ Nestrukturirano

- Programi koji se sastoje od slijeda naredbi i djeluju nad zajedničkim skupom podataka (globalnim varijablama)
- Ponavljanje nekog posla znači i kopiranje naredbi.

## ➤ Proceduralno

- Izdvajanjem naredbi u procedure,
- Program postaje slijed poziva procedura.

## ➤ Modularno

- Procedure zajedničke funkcionalnosti grupiraju se u module
- Dijelovi komuniciraju pozivima, moduli mogu imati lokalne podatke

## ➤ Objektno

- Objekti komuniciraju slanjem poruka

# Dijelovi programa

3

## ➤ programska cjelina (program unit)

- skup programskih naredbi koje obavljaju jedan zadatak ili jedan dio zadatka, npr. glavni program, potprogrami (procedure, funkcije)

## ➤ programski modul

- skup logički povezanih programskih cjelina → modularno programiranje
- npr. C datoteka sa statičkim varijablama

## ➤ komponenta

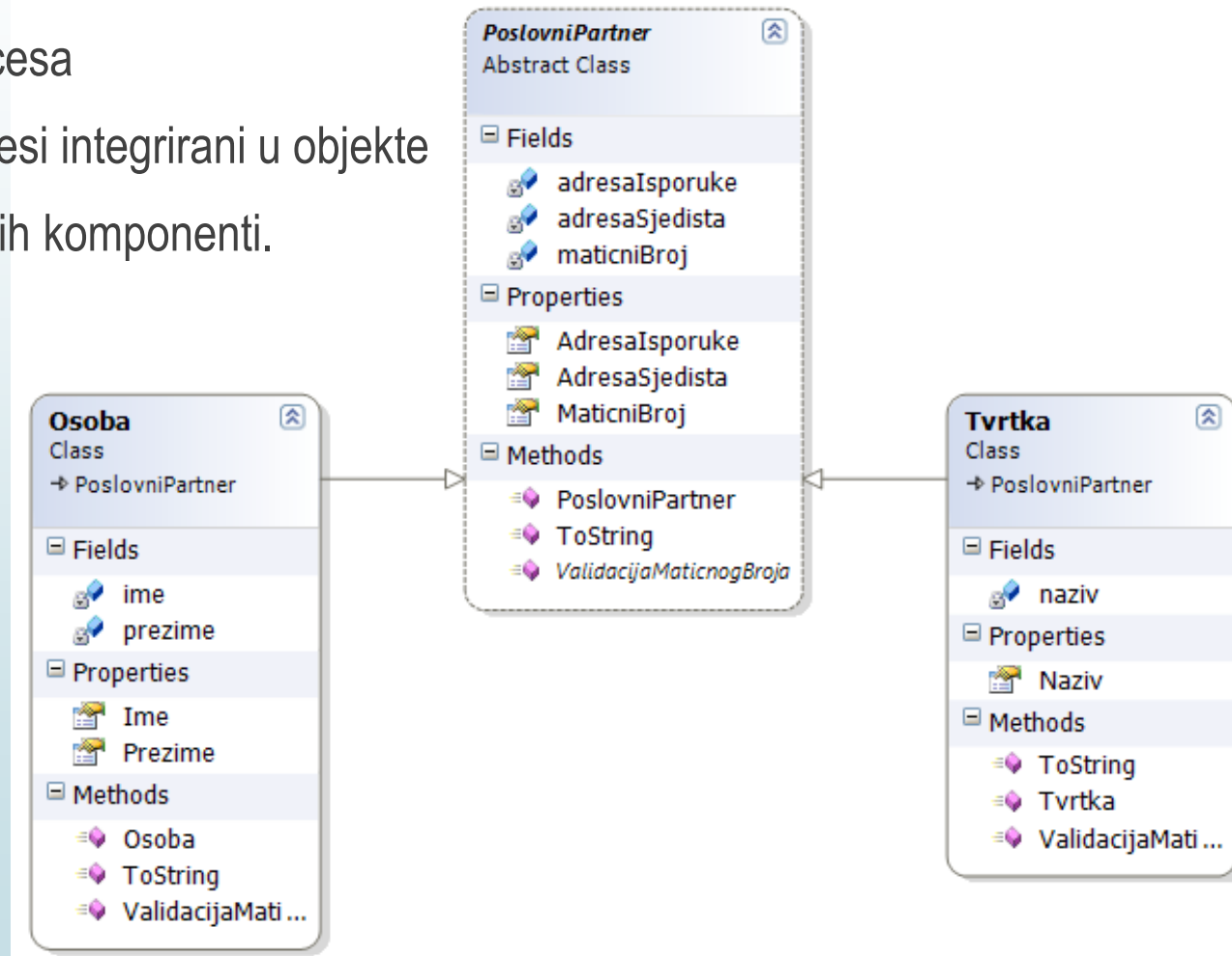
- bilo koji sastavni dio softvera, uobičajeno podrazumijeva fizičke cjeline
- npr. assembly (skup, sklop)
  - DLL ili EXE (izvršna) datoteka uključujući pripadajuće resursne (datoteka.resx) ili sigurnosne elemente

# Osnove objektno orijentiranog programiranja

4

## Objektno usmjerena paradigma

- entiteti iz stvarnog svijeta opisuju se apstraktnim objektima (razredima, klasama objekata)
- integracija oblikovanja podataka i procesa
- tokovi podataka, entiteti i veze te procesi integrirani u objekte
- složeni sustavi grade se iz pojedinačnih komponenti.



# Nasljeđivanje

5

- Omogućuje stvaranje novog razreda na temelju postojećeg razreda

```
class DerivedClass : BaseClass { }
```

- Generalizacija (i pripadna specijalizacija)

- osnovni razred – bazni razred (base class), superrazred (superclass), roditelj
- izvedeni razred – derivirani (derived), podrazred (subclass), dijete

- Dijete nasljeđuje članove roditelja i definira vlastite članove

- Višeobličje, polimorfizam (polymorphism)

- Postupak deklariran u osnovnom razredu biva nadjačana jednako deklariranim postupkom u naslijeđenom razredu.
- Podtip se ponaša drukčije odnosu na nadtip ili neki drugi podtip.

- Zamjenjivost (substitutability)

- Razred je podtip osnovnog razreda ako je u aplikaciji moguće zamijeniti osnovni razred podtipom, a da aplikacija normalno nastavi s radom.
- Drugim riječima, ako postoji kôd koji se odnosi na nadtip `Kupac`, umjesto njega može se supstituirati bilo koji njegov podtip

# Nasljeđivanje u programskom jeziku C#

6

- U programskom jeziku C# moguće je naslijediti samo jedan razred, a implementirati više sučelja.

```
class Osoba : Partner, IAdresa, IRazred { ... }
```


- Nasljeđuju se *public* i *protected* varijable i postupci
- privatne varijable i postupci ne mogu biti naslijeđeni
- *sealed* razred ne može biti naslijeđen

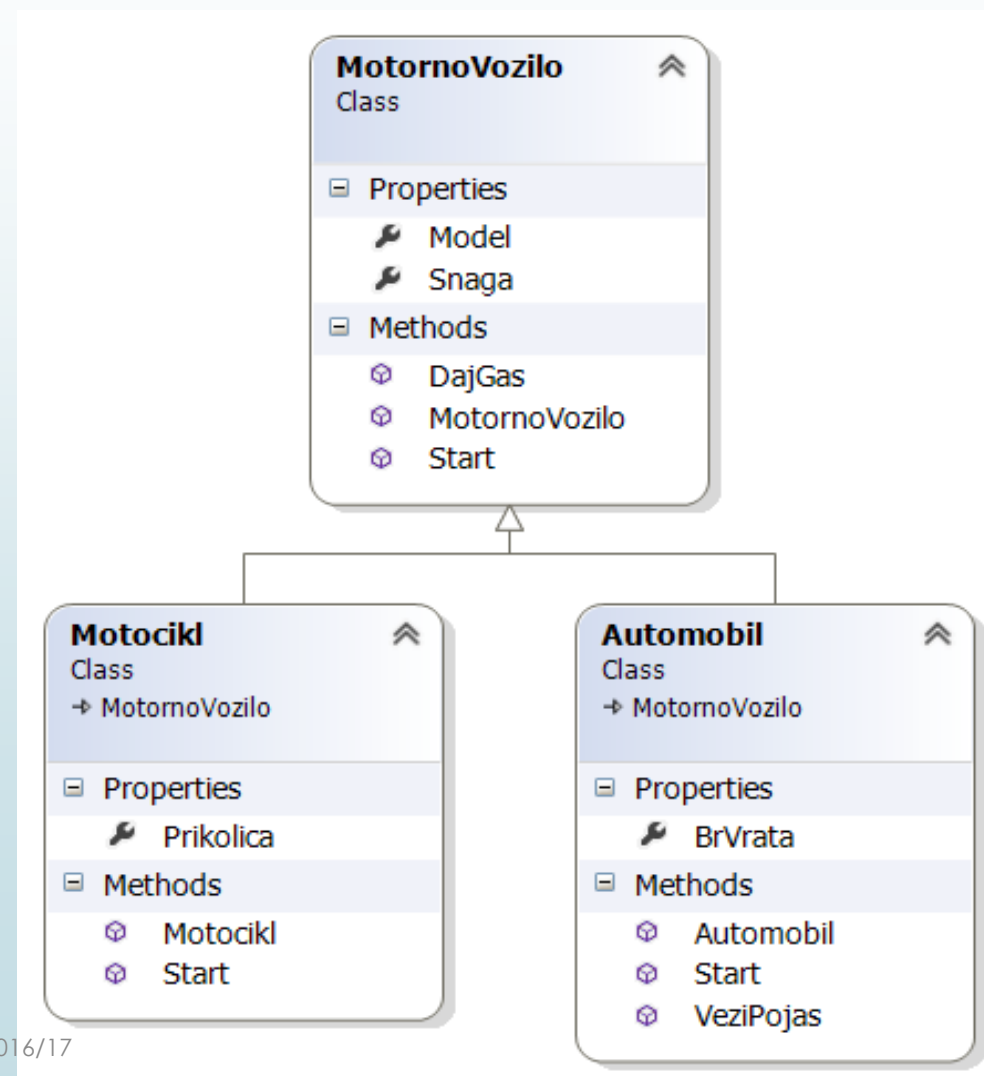
```
sealed class SealedPoint { ... }  
class MyPoint : SealedPoint // pogreška prevođenja
```

- base.member pristup nadjačanim članovima
  - Npr. base.F();
- Instanciranje objekta izvedenog razreda izaziva poziv konstruktora osnovnog razreda
- Pri uništenju objekta prvo se poziva finalizator izvedenog razreda, a zatim finalizator osnovnog razreda

# Nadjačavanje postupaka (method overriding)

7

- `virtual` deklarira virtualni postupak roditelja koji može biti nadjačan
- `override` deklarira postupak djeteta koji nadjačava, a može koristiti nadjačani postupak
- Primjer  Razredi \ MotornoVozilo



# Primjer nasljeđivanja – osnovni razred

8

➡ Primjer:  Razredi \ MotornoVozilo

```
class MotornoVozilo
{
    // atributi
    public string Model{ get; set; } // svojstvo
    ...

    // konstruktor
    public MotornoVozilo(string model, double snaga) { ... }


    // postupak (zajednicki svim vozilima)
    public void DajGas() { ... }

    // postupak koji izvedeni razredi implementiraju zasebno
    public virtual void Start() { ... }
}
```



# Primjer nasljeđivanja – izvedeni razred

9

- Primjer:  Razredi \ MotornoVozilo
- Tko što nasljeđuje?
- Može li se „sakriti” bazno svojstvo ili metoda (*public* pretvoriti u *private*) ?
- Što bi bilo da postupak *Start* nije virtualan?

```
class Automobil : MotornoVozilo {  
    // dodatni atribut  
    public int BrVrata { get; set; }  
    // konstruktor  
    public Automobil(string model, double snaga, int brVrata)  
        : base(model, snaga) // poziv osnovnog konstruktora  
    { ... }  
    // nadjačavanje baznog postupka Start()  
    public override void Start() { ... }  
    // dodatni postupak za Automobil  
    public void VeziPojas() { ... }  
}
```

# Modifikatori tipova i članova

10

## ➤ Modifikatori pristupa razredima i članovima

- `public` – pristup nije ograničen
- `private` – pristup ograničen na razred u kojem je član definiran
- `protected` – pristup ograničen na razred i naslijeđene razrede
- `internal` – pristup ograničen na program u kojem je razred definiran
- `protected internal` – pristup dozvoljen naslijeđenim razredima (bez obzira gdje su definirani) i svima iz programa u kojem je razred definiran

## ➤ Neki od značajnijih modifikatora

- `abstract` – razred može biti samo osnovni razred koji će drugi nasljeđivati
- `const` – atribut (polja) ili lokalna varijabla je konstanta
- `new` – modifikator koji skriva naslijeđenog člana od člana osnovnog razreda
- `readonly` – polje poprima vrijednost samo u deklaraciji ili pri instanciranju
- `sealed` – razred ne može biti naslijeđen
- `static` – jedini, zajednički član svih instanci razreda (ne kopija nastala s instancom)
- `virtual` – postupak ili dostupni član koji može biti nadjačan u naslijeđenom razredu – (prilikom nadjačavanja dodaje se modifikator `override`)

# Apstraktni razredi

11

## ➤ `abstract` – apstraktni razred

➤ Osnovni razred je nedovršen

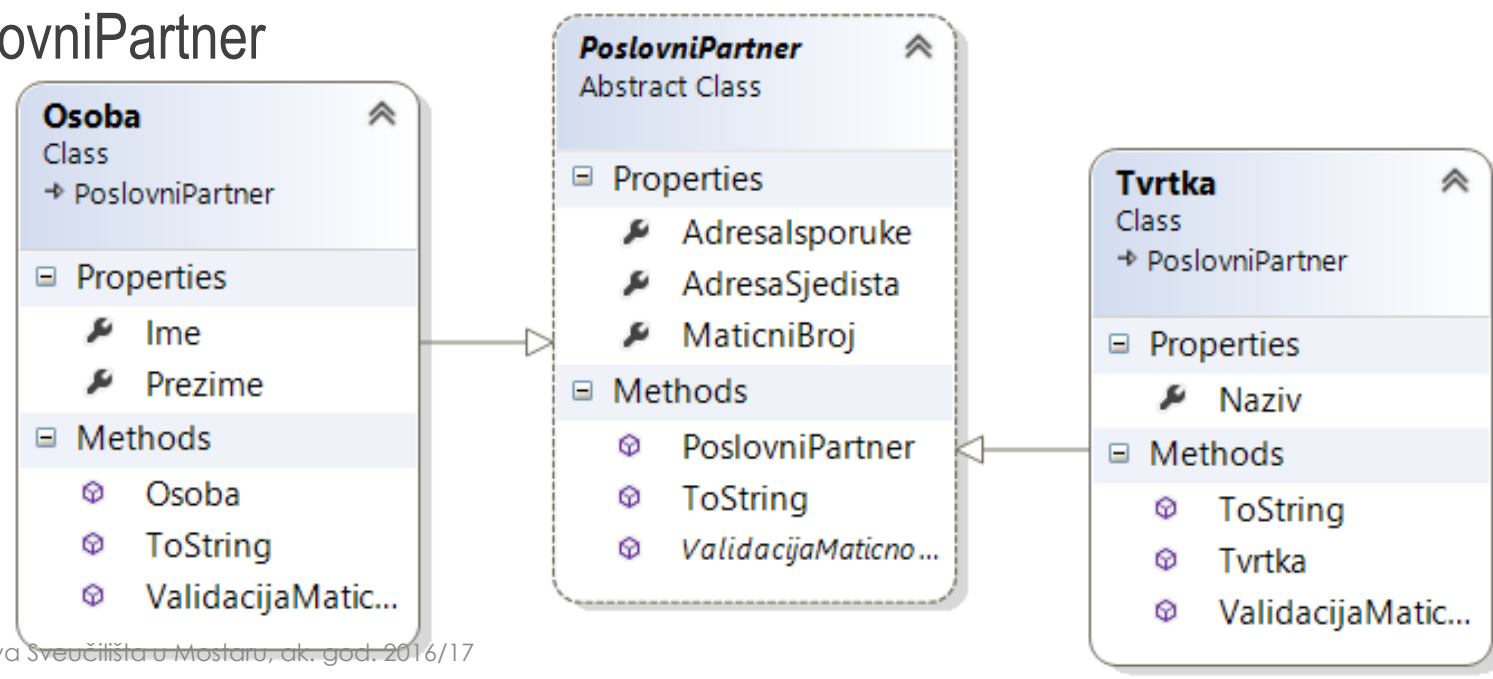
➤ Apstraktni postupci nisu ugrađeni

➤ Izvedeni razred **mora** u potpunosti ugraditi nedovršene postupke

➤ osim ako i sam nije apstraktan

➤ Ne može se instancirati objekt apstraktnog razreda, ali apstraktni razred može imati konstruktor!

## ➤ Primjer Razredi\PoslovniPartner



```
abstract class A {
    protected A(){...}
    public abstract int F();
}

class B: A {
    public override int F() { ... }
}
```

# Primjer nasljeđivanja apstraktnog razreda

12

➡ Razred Osoba nasljeđuje razred PoslovniPartner

➡ Konstruktor

```
public Osoba(string maticniBroj, string adresaSjedista,  
            string adresaIsporuke, string ime, string prezime)  
    : base(maticniBroj, adresaSjedista, adresaIsporuke) {  
    this.Ime = ime; this.Prezime = prezime;  
}
```

➡ Implementacija apstraktnog postupka

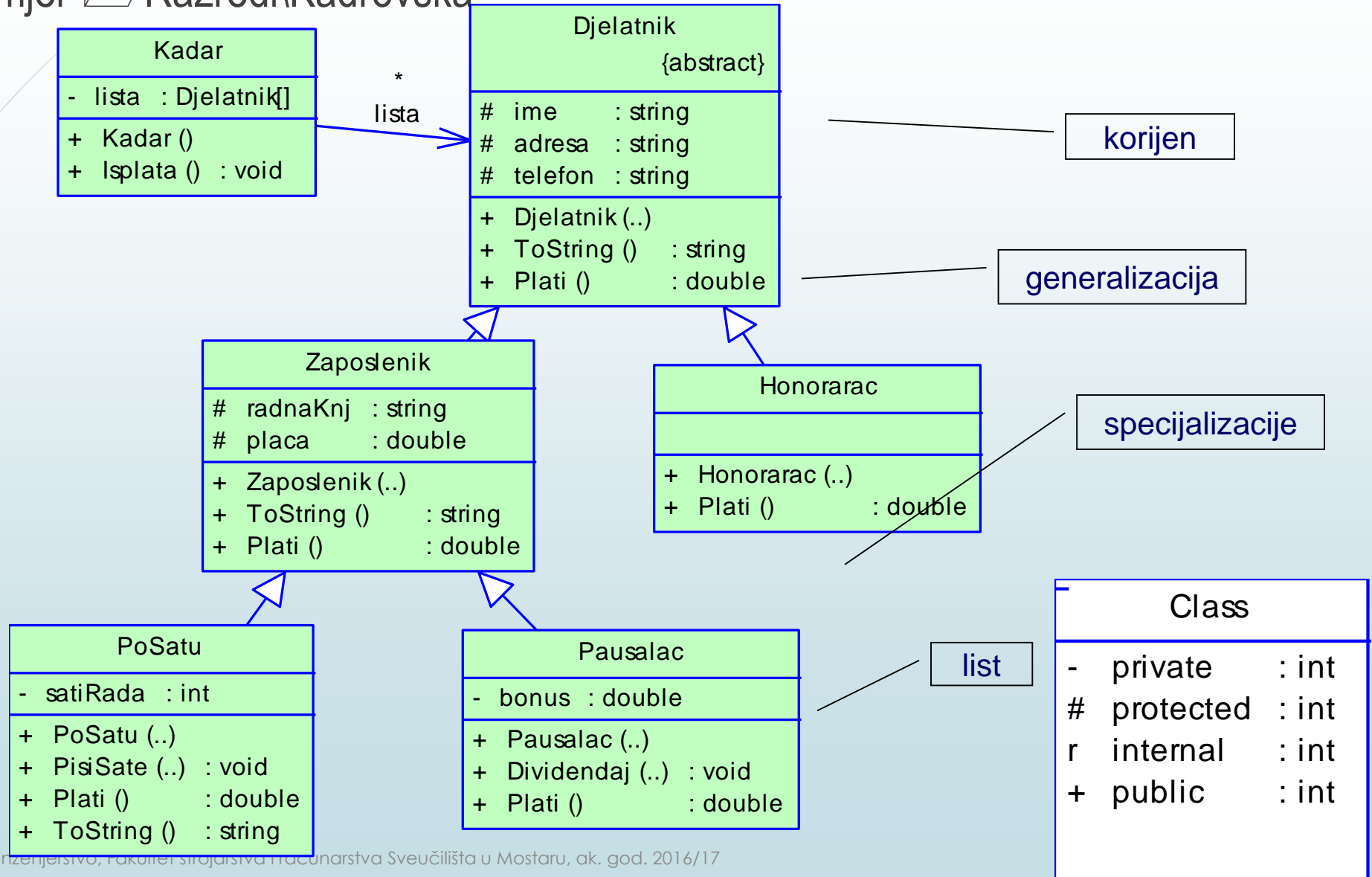
```
abstract class PoslovniPartner{  
    public abstract bool ValidacijaMaticnogBroja();  
    // apstraktni postupak je potrebno implementirati u izvedenom razredu  
}
```

```
class Osoba : PoslovniPartner{  
    public override bool ValidacijaMaticnogBroja()  
    { ...implementacija u izvedenom razredu }  
}
```

# Primjer hijerarhije nasljeđivanja

13

► Primjer  Razredi\Kadrovski




## ➤ Primjer dinamičkog povezivanja Razredi\Kadrovski

- referenca može pokazivati na različite objekte
- tip trenutnog objekta određuje postupak koji se obavlja

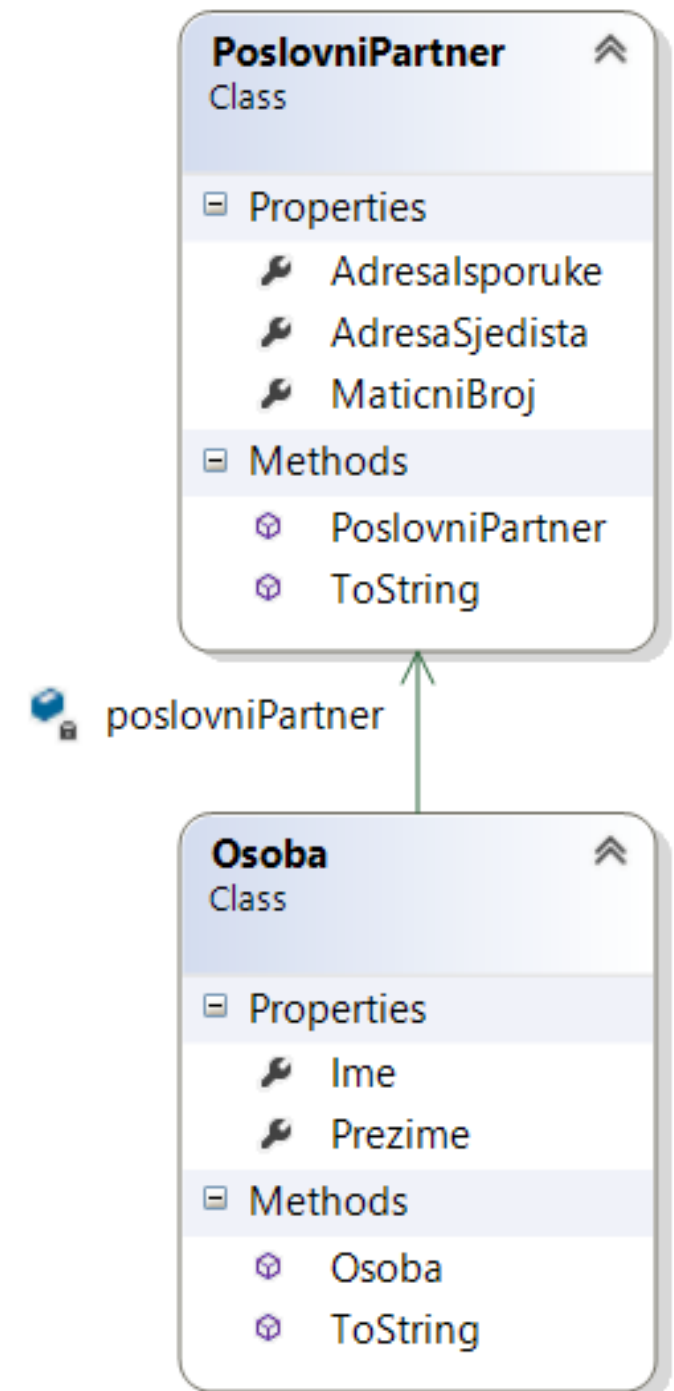
```
public class Kadar {  
    private Djelatnik[] lista;  
    public Kadar () {  
        lista = new Djelatnik[6];  
        lista[0] = new Pausalac ("Bobi", "Bobinje bb",  
                                "555-0469", "123-45-6789", 2423.07);  
        lista[1] = new Zaposlenik("Rudi", "Rudinje 1",  
                                "555-0101", "987-65-4321", 1246.15);  
        ...  
        ((Pausalac)lista[0]).Dividendaj (500.00); // cast  
        ((PoSatu)lista[3]).PisiSate (40); // cast  
    }  
    public void Isplata() {  
        double iznos;  
        for ( int count=0; count < lista.Length; count++ ) {  
            Console.WriteLine( lista[count] );  
            iznos += lista[count].Plati(); // polimorfizam  
        }  
    }  
}
```

# Kompozicija umjesto nasljeđivanja

15

- Umjesto nasljeđivanja, dijete može "učahuriti roditelja"
- Primjer  Razredi\Kompozicija

```
class Osoba
{
    PoslovniPartner poslovniPartner;
    private string ime;
    private string prezime;
    ...
}
```



# Sučelje (interface)

16

- Sučelje (interface) - specifikacija članova razreda bez implementacije
  - Sadrži samo deklaracije operacija
  - Može sadržavati postupke, svojstva, indeksere i događaje
  - Svaki postupak je apstraktan (nema implementaciju)
- Notacija
  - Standard za naziv: INaziv
  - Proširena i skraćena (*lollipop*) notacija
- Sučelje definira obvezu ugradnje
  - Razred koji nasljeđuje sučelje, mora implementirati sve postupke
  - Sučelje može implementirati i apstraktni razred
- Sučelje može biti argument nekog postupka čime se postiže veća općenitost

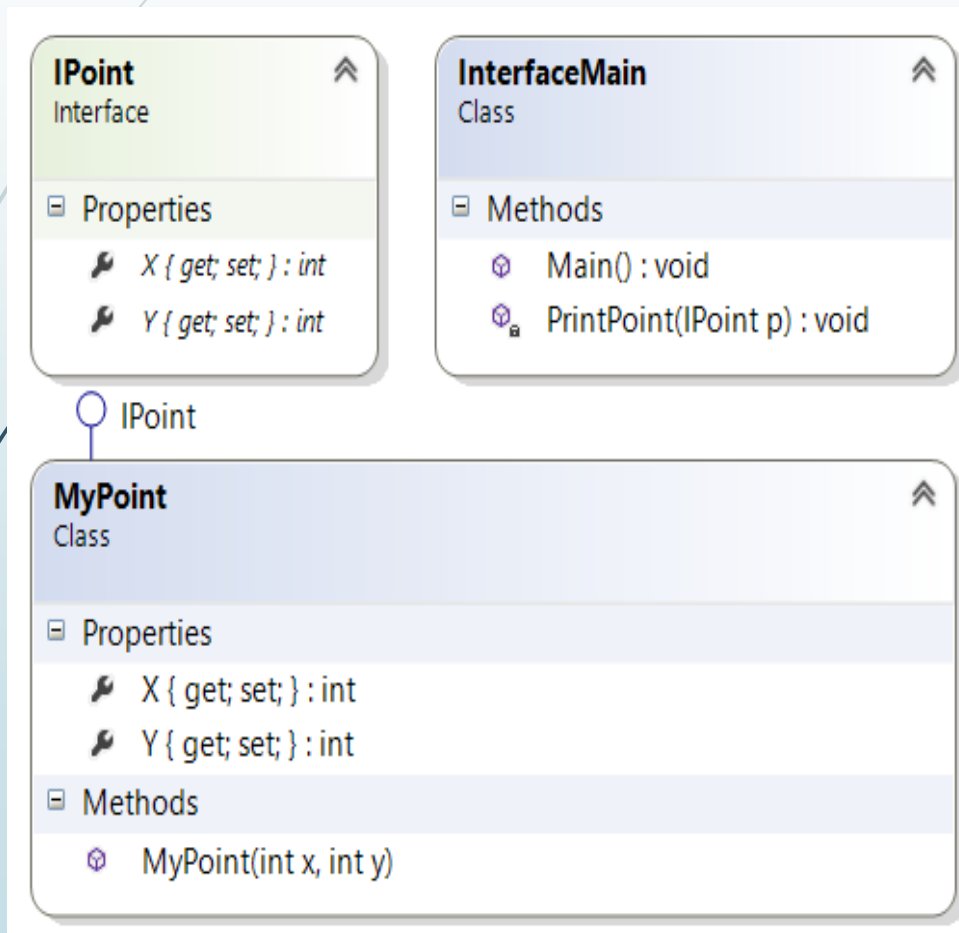


# Primjer sučelja i realizacije

17

## ➤ Primjer Razredi\Sucelje

- postupak za ispis prima bilo koju implementaciju sučelja IPoint




```
interface IPoint {  
    // Property signatures:  
    int X { get; set; }  
    int Y { get; set; }  
}
```

```
class MyPoint : IPoint {  
    //automatsko svojstvo  
    public int X{  
        get ; set ;  
    }  
    ...  
}
```

```
class InterfaceMain {  
    ... void PrintPoint(IPoint p) {  
    ...
```

- Prostor imena *System.Collections* sadrži sučelja i razrede za rad s kolekcijama
  - Kolekcija je skup povezanih objekata.
- Sučelja: *ICollection*, *IEnumerator*, *IEnumerable*, *IDictionary* i  *IList*
  - Određuju osnovne funkcionalnosti kolekcija
  - Razred koji implementira jedno ili više tih sučelja naziva se kolekcija.
- Razredi: *ArrayList*, *BitArray*, *Hashtable*, *Queue*, *SortedList*, *Stack*, ...
  - Mogu pohranjivati različite tipove objekata unutar iste kolekcije
  - Rjeđe se koriste, jer postoje slične generičke varijante

- Sadrži sučelja i razrede za generičke kolekcije
  - Bolja tipska sigurnost (*type safety*) i bolje performanse od ne-generičkih
  - Funkcionalnosti ne-generičkih kolekcija
- Neki razredi prostora imena *System.Collections.Generic*
  - **public class List<T>** : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
  - **public class Queue<T>** : IEnumerable<T>, ICollection, IEnumerable
  - **public class Stack<T>** : IEnumerable<T>, ICollection, IEnumerable
  - **public class Dictionary<TKey,TValue>** : IDictionary<TKey,TValue>, ICollection<KeyValuePair<TKey,TValue>>, IEnumerable<KeyValuePair<TKey,TValue>>, IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback

- `List<T>` – parametrizirana kolekcija objekata kojima se može pristupati preko indeksa u listi
  - npr `List<int>`, `List<string>`, `List<Osoba>`
  - Funkcionalnost razreda `ArrayList`, pri čemu u svi elementi istog tipa i nije moguće ubaciti element nekog drugog tipa (provjera prilikom kompilacije)
  - Primjer  Razredi \ Generics

```
List<string> listaString = new List<string>();  
listaString.Add("jedan");  
...  
string trazen = ...  
if (listaString.Contains(trazen))  
    Console.WriteLine("Element postoji u listi na " +  
        listaString.IndexOf(trazen) + ". mjestu.");  
else  
    Console.WriteLine("Element ne postoji u listi!");
```

# Prednosti generičkih kolekcija

21

- Negeneričke kolekcije objekte pohranjuju kao `System.Object`
  - Takva pohrana je prednost sa stanovišta fleksibilnosti
  - Nedostatak je obavljanje *boxinga* prilikom dodavanja `value` tipa podataka (dobivanje reference na `value` podatak da bi se s tim podatkom moglo postupati kao sa `System.Object`)
    - pad performansi za veće količine podataka
    - potrebna konverzija (*unboxing*) da se dobije smisleni podatak
- Generičke kolekcije su tipizirane
  - Bolje performanse – ne mora se obavljati *boxing/unboxing*
  - Tipska sigurnost (*type safety*) – zaštita od unosa podataka nepoželjnog tipa i pogrešaka koje pritom mogu nastati (npr. pokušaj izvršavanja naredbe neprikladne za dani tip podataka)

# Primjer izrade parametriziranog razreda

22

➡ Primjer  Razredi \ Generics

```
using System.Collections.Generic;

public class Stog<T>
{
    T[] elementi;
    public void Stavi(T element) {...}
    public T Skini() {...}
}

Stog<int> stogInt = new Stog<int>();
Stog<string> stogString = new Stog<string>();
Stog<Automobil> stogAutomobil = new Stog<Automobil>();
```

# Ograničenja na tip parametriziranog razreda

23

- Kontekstualna ključna riječ *where*
- Moguća ograničenja nekog tipa T:
  - *where T:struct* – tip T mora biti *value type* (*Nullable* također isključen)
  - *where T:class* – tip T mora biti *reference type*
  - *where T:new()* – tip T mora imati prazni konstruktor
  - *where T:naziv baznog razreda* – tip T mora biti navedeni razred ili razred koji nasljeđuje taj razred
  - *where T:naziv sučelja* – tip T mora implementirati navedeno sučelje
  - *where T:U* – tip T mora tip U ili izveden iz tipa U pri čemu je U drugi tip po kojem se vrši parametrizacija

## ➤ *Primjer:*

```
public class GenRazred<T, U>
    where T:Stog<U> where U:IPoint, new() {
    . . .
```

- U navedenom primjeru *GenRazred* je određen s dva tipa pri čemu prvi tip mora biti stog iz prethodnog primjera određen tipom koji implementira sučelja *IPoint* i ima prazni konstruktor

# Tablice s raspršenim adresiranjem i rječnici

24

## ➡ Dictionary<TKey, TValue>

- ➡ System.Collection.Generics
- ➡ Parovi ključ, vrijednosti
- ➡ Ključ i vrijednost su tipizirani
- ➡ Razlikovati postupak Add i indeksir [ključ]

```
Dictionary<int, string> dict = new Dictionary<int, string>(10);  
  
dict.Add(100, "Jedan");  
dict[200] = "Dva";  
  
foreach (KeyValuePair<int, string> entry in dict) {  
    Console.WriteLine(entry.Value);  
}
```



# Proširenja (eng. extensions)

25

➡ Razred za koji se piše proširenje je naveden kao prvi parametar statičke metode u statičkom razredu, prefiksiran s *this*

➡ Pozivaju se kao da se radi o postupku unutar tog razreda (iako to nije)

➡ Primjer  Razredi\Generics

```
public static class Extensions{  
    public static V DohvatiIliStvori<K, V> (this Dictionary<K, V> dict, K key)  
        where V : new() {  
  
        if (!dict.ContainsKey(key)) {  
            V val = new V();  
            dict[key] = val;  
        }  
        return dict[key];  
    }  
}
```

```
var dict = new Dictionary<int, Stog<string>>();
```

```
var stog = dict.DohvatiIliStvori(1);
```

# Odnosi razreda u složenijim tipovima

26

- Razred `Automobil` je podrazred razreda `MotornoVozilo`.
- U kojem su odnosu `List<Automobil>` i `List<MotornoVozilo>` ?
  - Nisu hijerarhijski povezani
- U kojem su odnosu `IEnumerable<Automobil>` i `IEnumerable<MotornoVozilo>`  
`Comparer<Automobil>` i `Comparer<MotornoVozilo>`
  - Odgovor nije očit (jednostavan) !!

# Invarijantnost, kovarijantnost i kontravarijantnost

27

- Invarijantnost (engl. invariance)
  - Mora se koristiti samo specificirani tip
- Kovarijantnost (engl. covariance)
  - Mogućnost korištenja nekog općenitijeg tipa umjesto originalno navedenog
  - Neki tip je kovarijantan ako zadržava postojeće odnose među tipovima
- Kontravarijantnost (engl. contravariance)
  - Mogućnost korištenja nekog podtipa umjesto originalno navedenog
  - Neki tip je kontravarijantan ako stvara suprotan odnos među postojećim tipovima

## ► Primjer Razredi \ CovarianceContravariance

```
void IspisiVozila(IEnumerable<MotornoVozilo> vozila) {  
    for(var vozilo in vozila)  
        Console.WriteLine(vozilo.Model) ;  
}
```

- Kao argument moguće je poslati `List<MotornoVozilo>`, jer `List<T>` implementira sučelje `IEnumerable<T>`, ali moguće je poslati i `List<Automobil>`

- `List<Automobil>` se može pretvoriti u `IEnumerable<Automobil>`, a sučelje `IEnumerable<T>` je kovarijantno

```
public interface IEnumerable<out T>
```

```
IEnumerable<Automobil> auti = new List<Automobil>();  
...  
IspisiVozila(auti)
```

## ► Primjer Razredi \ CovarianceContravariance

```
void IspisiBoljiMotor(Motocikl a, Motocikl b,  
                    IComparer<Motocikl> comparer) {  
    Console.WriteLine(comparer(a, b) < 0 ? a : b);  
}
```

- Kao komparator moguće je poslati objekt tipa `IComparer<MotornoVozilo>`, jer je sučelje `IComparer<T>` **kontavarijantno**

```
public interface IComparer<in T>
```

- Delegati su objekti koje sadrže reference na postupke
  - Omogućavaju metodama da budu argumenti neke druge metode
  - Nalik pokazivačima na funkcije u C-u
- Delegati koji sadrže reference na više postupaka nazivaju se *MultiCastDelegate*
  - Pozivom delegata redom se pozivaju referencirane metode
- Delegat se definira kao varijabla određenog tipa delegata
- Tip delegata definira se sljedećom sintaksom

```
public delegate PovratniTip NazivTipaDelegata (argumenti)
```

  - Npr. `public delegate double Tip(int a, string b)` bi definirao tip delegata koji bi omogućio pohranu referenci na sve postupke kojima imaju dva argumenta tipa `int` i `string`, a vraćaju `double`
  - Interno se stvara novi razred *NazivTipaDelegata* koji nasljeđuje razred *MultiCastDelegate*
  - Nakon toga bi se mogao definirati delegat na sljedeći način `Tip nazivdelegata;`
  - Delegati imaju definirane operacije `=`, `+=`, `-=`

# Primjer tipa delegata

31

➤ Primjer  Razredi \ Delegati \ Program.cs

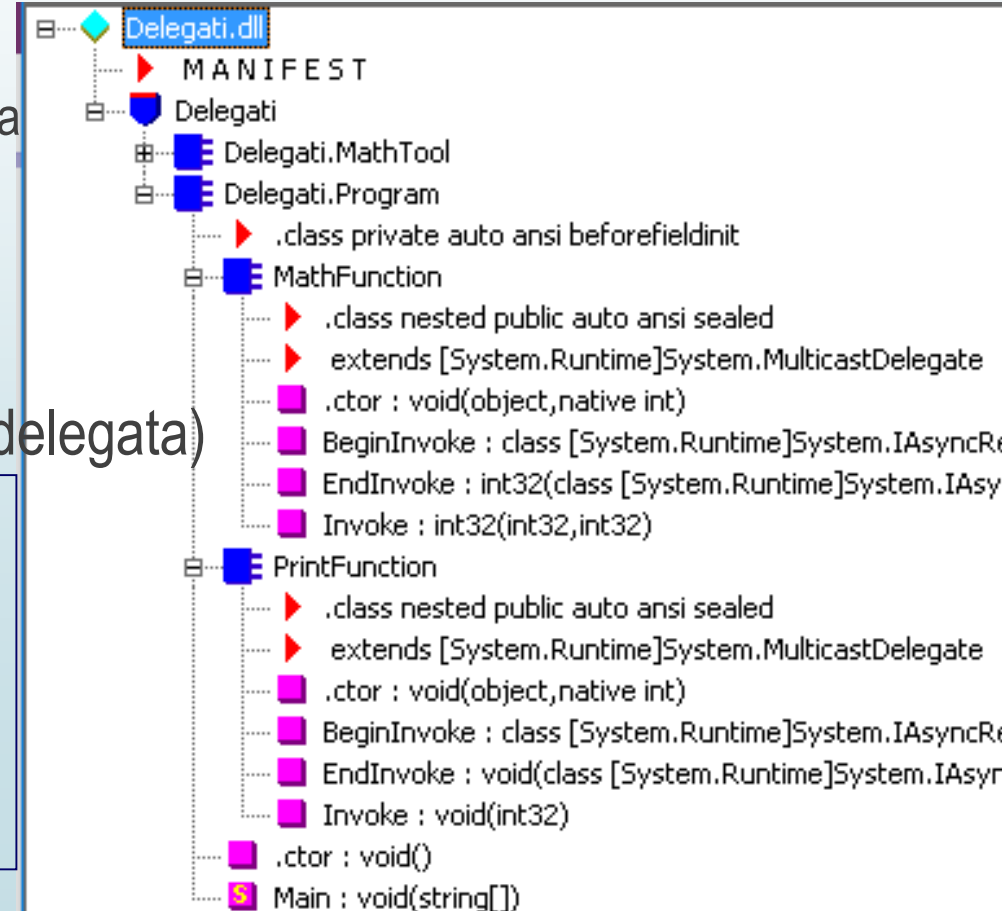
```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
}
```

➤ Definirana su 2 tipa delegata

- MathFunction za postupke koji primaju dva cjelobrojna argumenta i vraćaju cijeli broj
- PrintFunction za postupke koji primaju cijeli broj i ne vraćaju ništa

➤ U glavnom programu definirane dvije varijable (delegata)

```
class Program {  
    static void Main(string[] args)  
    {  
        MathFunction mf = ...  
        PrintFunction pf = ...  
    }  
}
```



# Primjer pridruživanja postupka delegatu (1)

32

➡ Primjer  Razredi \ Delegati \ MathTool.cs

➡ Vlastiti razred MathTool sadrži nekoliko postupaka koji svojim potpisom odgovaraju tipova delegata

```
public class MathTool {  
    public static int sum(int x, int y) { return x + y; }  
    public static int diff(int x, int y) { return x - y; }  
    public static void printSquare(int x) {  
        Console.WriteLine("x^2 = " + x * x);  
    }  
    public static void printSquareRoot(int x) {  
        Console.WriteLine("sqrt(x) = " + Math.Sqrt(x));  
    }  
}
```



# Primjer pridruživanja postupka delegatu (2)

33

➡ Primjer  Razredi \ Delegati \ Program.cs

➡ Varijabla mf je delegat koji sadrži reference na postupke koje primaju 2 cijela broja (kao što je npr. postupak sum iz razreda MathTool)

```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
    static void Main(string[] args)  
    {  
        int x = 16, y = 2;  
        MathFunction mf = MathTool.sum;  
        Console.WriteLine("mf({0}, {1}) = {2}", x, y, mf(x, y));  
        mf = MathTool.diff;  
        Console.WriteLine("mf({0}, {1}) = {2}", x, y, mf(x, y));  
    }  
}
```

# Primjer pridruživanja postupka delegatu (3)

34

➤ Primjer  Razredi \ Delegati \ Program.cs

➤ Delegatu se može pridružiti više postupaka (mogu se naknadno ukloniti)

➤ Obično ima smisla za postupke koje ne vraćaju nikakvu vrijednost

```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
    static void Main(string[] args) {  
        int x = 16, y = 2;  
        ...  
        PrintFunction pf = MathTool.printSquare;  
        pf += MathTool.printSquareRoot;  
        pf(x);  
        pf -= MathTool.printSquare;  
        Console.WriteLine();  
        pf(y);  
    }  
}
```

# Primjeri postojećih tipova delegata

35

- Func i Action kao dva najpoznatija tipa delegata
- `Action<in T1>`, `Action<in T1, in T2>`,  
`Action<in T1, in T2, in T3>`, ..., `Action<in T1,..., in T16>`
  - Referenca na postupke koji ne vraćaju ništa, a primaju 1, 2, 3, ..., ili 16 argumenata
  - Argumenti su kontravarijantni (vidi sljedeći slajd)
- `Func<in T1, out TResult>`, `Function <in T1, in T2, out TResult>`, ..., `Function<in T1, in T2, ... in T16, out TResult>`
  - Referenca na postupke primaju do 16 argumenata i vraća vrijednost tipa TResult
  - TResult je kovarijantan, a ostali su kontravarijantni (vidi sljedeći slajd)
- U prethodnim primjerima se umjesto *MathFunction* mogao koristiti *Function<int, int, int>*, a umjesto *PrintFunction* *Action<int>*

# Function i varijantnost

36

- Povratni tip je kovarijantan što znači da postupak koji se pridružuje delegatu može vraćati izvedeni tip od onog koji je predviđen pri parametrizaciji
- Ulazni tipovi su kontravarijantni što znači da postupak koji se pridružuje za ulazne argumente može imati traženi tip ili njemu nadređene.
- Npr. Ako je definiran delegat tipa *Func<Automobil, Zaposlenik>* (za neki automobil vrati zaposlenika) tada se delegatu tog tipa mogu pridružiti reference na postupke

- `Zaposlenik Test(Automobil a) { ... }`

- `PoSatu Test(Automobil a) { ... }`

- `Pausalac Test(MotornoVozilo a) { ... }`

ali ne i npr.

- `Djelatnik Test(Automobil a) { ... }`

- `Zaposlenik Test(ElektricniAutomobil a) { ... }`

- Navedeno ima smisla, jer ako je *Func<Automobil, Zaposlenik> f = nešto od navedenog* tada negdje kasnije u programu slijedi *Zaposlenik z = f(neki automobil)* pa je potpuno svejedno da li je povratna vrijednost *Zaposlenik* ili nešto izvedeno iz njega, odnosno prima li pridruženi postupak *Automobil* ili nešto općenitije.

- Objects, Classes, and Structs (C# Programming Guide)
  - <http://msdn2.microsoft.com/en-us/library/ms173109.aspx>
- Generics (C# Programming Guide)
  - <http://msdn.microsoft.com/en-us/library/0x6a29h6.aspx>
- Kovarijantnost i kontravarijantnost
  - <https://msdn.microsoft.com/en-us/library/mt654055.aspx>
  - <https://msdn.microsoft.com/en-us/library/dd799517.aspx>
  - <http://tomasp.net/blog/variance-explained.aspx/>