OOP

C#

Sta je to tip podataka, a sta struktura?

Tip podataka je odredjen (konacnim) **skupom vrednosti** i nekim **paketom operacija i relacija** nad elementima tog skupa.
Pr. ({-32768, ..., -1, 0, 1, ..., 32767}, +, -, * div, mod, =, <)

Struktura podataka je konglomerat podataka. Ona se formira od drugih (jednostavnijih) struktura i od tipova. Jedine operacije koje se nad strukturama mogu obavljati su **operacije selekcije**.

Pr. var **a**: **array** [1 .. 100] of real;

struktura **metod struktuiranja**

Metod struktuiranja ima sopstvene operacije selekcije.
Pr. **array** indeksiranje (a[i]), a **record** projektovanje (odabir polja sloga, a.ime) - PASCAL

Pascal:
eksplicitni metodi struktuiranja (array, record, record-case)
implicitni (preko pokazivaca).

Sta je to tip podataka, a sta struktura?

Razlika između tipa i strukture je u tome sto je nad elementima tipa moguće vršiti nekakve operacije i smeštati ih u nekakve relacije, dok je nad strukturom moguće obavljati samo selekciju komponente.

Ako uradimo sledeće

```
type STR = packed array [1 .. 100] of char;

procedure Concat(s1, s2 : STR; var res : STR); ...
procedure Substr(s : STR; from, to : integer; var res : STR); ...
function  Compare(s1, s2 : STR) : integer; ...
```

STR ima skup vrednosti i operacije na elementima tog skupa (Concat, Substr, Compare) dakle STR je sada tip.

Zasto je važno praviti nove tipove?

Za uspešan rad u velikim timovima i na velikim projektima bitno pisati apstraktan kood.

NPR. Za projekat u kome je potrebno raditi nesto sa matricama: napraviti tip MATRIX (struktura MATRIX snabdevena operacijama za rad sa matricama), pa kada dodje do rešavanja konkretnog problema, samo pozivati gotove (i testirane) procedure.

```
C := Inv(A) * Transpose(B) + Det(Y) * Adj(X)
```

```
Inv(A, A1);
Transpose(B, B1);
MatMul(A1, B1, P);
Adj(X, X1);
ScalMul(Det(Y), Q);
MatAdd(P, Q, C);
```

```
for i := 1 to n do
      for j := 1 to n do
         (* rutina koja invertuje matricu A u A1 *);
   for i := 1 to n do
      for j := 1 to n do
         B1[i,j] := B[j, i];
   for i := 1 to n do
      for j := 1 to n do
         (* rutina koja mnozi A1 i B1 *);
   (* itd *)
```

Dakle, dizajnirati odgovarajuce strukture podataka, napisati procedure za manipulaciju tim strukturama podataka.

Tako dolazimo do jednog viseg nivoa apstrakcije u programima. U trenutku kada je potrebno primeniti neke operacije na nekim strukturama, ne interesuje nas *kako* procedure rade, vec *sta* rade.

Upotreba tipa

```
var a, b: MATRIX;
begin
   NewMatrix(a); NewMatrix(b);
   ReadMatrix(a);
   Transpose(a, b);
   WriteMatrix(b);
   DisposeMatrix(a);
   DisposeMatrix(b)
end;
```

Implementacija

```
type MATRIX = array [1 .. X, 1 .. Y] of real;
```

```
type MATRIX = ^MatrixEntry;
     MatrixEntry =  record
                      i, j: integer;
                    entry: real;
                    right, down:
  MATRIX
                    end;
```

NIJE BITNA IMPLEMENTACIJA, VEĆ MANIFESTACIJE (TJ. OSOBINE) OPERACIJA DATOG TIPA PODATAKA.

Osnovna ideja OOP-a - Abstract data type - ADT

Struktura podataka → Tip podatka data abstraction
ADT → Apstakcija + Skrivanje podataka information hiding

Apstraktni tip podataka je tip podataka čiju implementaciju ne znamo (primer: ne znamo da li su matrice predstavljene sa array ili preko pokazivaca), ali znamo kako se ponašaju operacije nad vrednostima tog tipa, tako da pišemo program koristeci samo osobine operacija.
Dakle, apstrahovana je jedna dimenzija problema: implementacija tipa (apstrahovati znaci izbaciti iz posmatranja ono sto u tom trenutku nije bitno).

Za pojam ADT neraskidivo vezan pojam sakrivanja informacije (information hiding). Programeru se da ime tipa i paket procedura. Implementacija tipa se SAKRIJE da je on ne vidi.

Primer

TIP

KLASA

Promenljiva

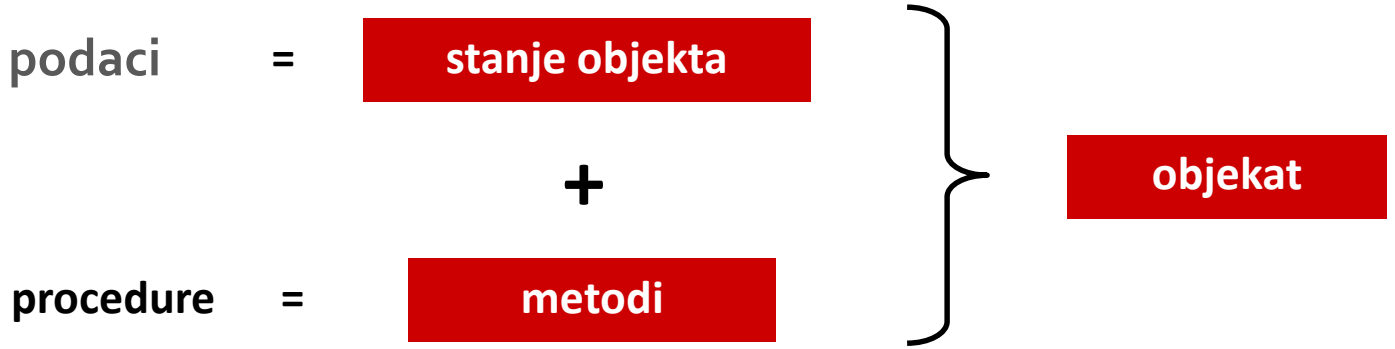
OBJEKAT

```
static void Main(string[] args)
    {
      Random rnd = new Random();
      int[] niz = new int[10];
      for (int i = 0; i < 10; i++)
        niz[i] = rnd.Next(1, 100);
      Console.WriteLine("Generisan je niz");
      for (int i = 0; i < 10; i++)
        Console.Write("{0} ",niz[i]);
      Console.ReadKey();
    }
```

KLASA je TIP.

OBJEKAT je PRIMERAK (instanca) tipa.

Klase i objekti



- Klasom se opisuju objekti sa istim
  - karakteristikama (podaci članovi)
  - ponašanjem (funkcionalnostima – metode)

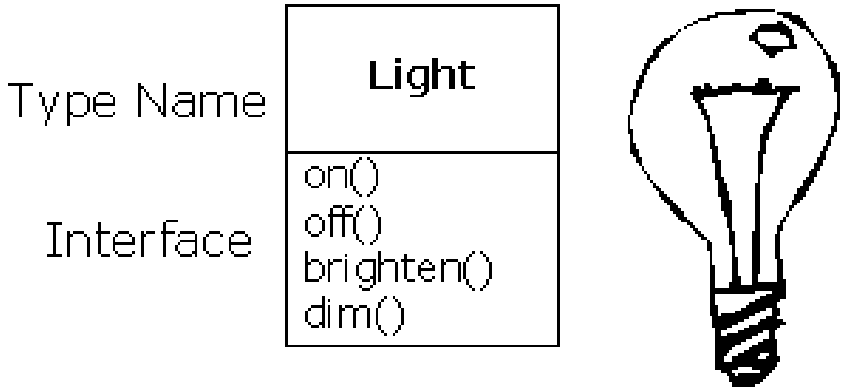
- Podaci članovi (atributi)
  - svaki objekat ima sopstvene vrednosti podataka članova
  - trenutne vrednosti podataka objekta čine trenutno **stanje objekta**
- Funkcije članice (metodi)
  - njima su definisana **ponašanja objekta**
  - poziv metoda jednog objekta – **slanje poruke**
  - obrada zahteva tj. **odgovaranje na poruku**

- slanje **poruke** objektu = pozivanje metoda nekog objekta

```
rnd.Next(1, 100);
```

- podaci unutar objekta moraju biti zaštićeni, može im se (eventualno) pristupiti (čitanje, promena stanja i sl.) slanjem poruke.

skrivanje podataka se naziva **učaurivanjem (encaptulation) - information hiding**



PRIMER.
`lt` - objekat klase `Light`
`lt.on();` - slanje poruke, tj. poziv metoda on objekta `lt`

Objekti i klase u C#

C#

Definicija klase

```
<modifikator> class <className> {

    <modifikator> <tip> <imepromenljive1>;
    <modifikator> <tip> <imepromenljive2>;
    ...

  <modifikator> <povratni tip> <imemetoda1> (<tip> <arg1>,...) {
        ... //implementacija metoda 1
  }
  <modifikator> <povratni tip> <imemetoda2> (<tip> <arg1>,...) {
        ... //implementacija metoda 2
  }
  ...

}  // kraj definicije klase
```

podaci članovi

funkcije članice - metodi

Primer klase

```
class Circle
{
        double Area()
        {
                return Math.PI * radius * radius;//Math je statička klasa
        }
        int radius;
}
```

Korišćenje klase

```
class Program
  {
    static void Main(string[] args)
    {
      Circle k = new Circle();
      k.radius = 3;
      Console.WriteLine("Povrsina kruga je " + k.Area());
      Console.ReadKey();
    }
  }
  class Circle
  {
    public double Area()
    {
      return Math.PI * radius * radius;//Math je statička klasa
    }
    public void setRadius(int r)
    {
      radius = r;
    }
    public int radius;
  }
```

Primer

- Definisati klasu `Robot` na sledeći način

Robot – bez modifikatora
`rbr` – tipa integer, bez modifikatora
`setrbr(int br)` – metod koji postavlja vrednost promenljive rbr `sayHello()` – metod koji ispisuje poruku *Hello, I am robot no. ___*

Definisati aplikaciju `UseRobot` (preciznije `public` klasu `UseRobot`) koja u svom main metodu:

- kreira objekat klase `Robot`,
- postavlja vrednost `rbr` na `1`,
- šalje poruku kreiranom objektu da se javi (`sayHello`).

Preporučeno imenovanje klase i elemenata

- Po pravilu se koriste mala slova za imenovanje identifikatora, dok se korišćenje velikih slova posebno naglašava
- Ime klase treba da počinje sa velikim slovom – klasa Circle
- Imena javnih – public članova klase treba da počinju sa velikim slovom – metod Area()
- Imena privatnih članova treba da imaju sva mala slova – podatak radius tipa int

Inicijalizatori objekata

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

Delete the second two lines, and the semicolon after "Guy()" and add a right curly bracket.

```
joe = new Guy() {
```

Press space. As soon as you do, the IDE pops up an IntelliSense window that shows you all of the fields that you're able to initialize.

```
joe = new Guy() {
```

Press tab to tell it to add the Cash field. Then set it equal to 50.

```
joe = new Guy() { Cash = 50
```

Type in a comma. As soon as you do, the other field shows up.

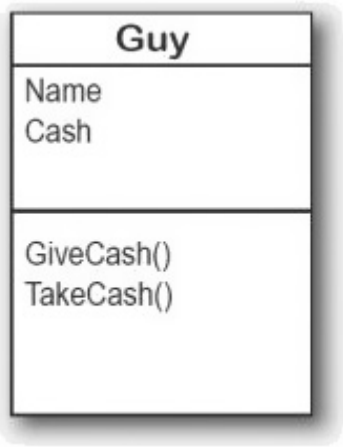
```
joe = new Guy() { Cash = 50,
```

Finish the object initializer. Now you've saved yourself two lines of code!

```
joe = new Guy() { Cash = 50, Name = "Joe" };
```

This new declaration does exactly the same thing as the three lines of code you wrote originally. It's just shorter and easier to read.

Object intializers save you time and make your code more compact and easier to read... and the IDE helps you write them.




```
Circle k = new Circle();
k.radius = 3;
```

```
Circle k = new Circle() { radius = 3};
```

O objektima i njihovim referencama

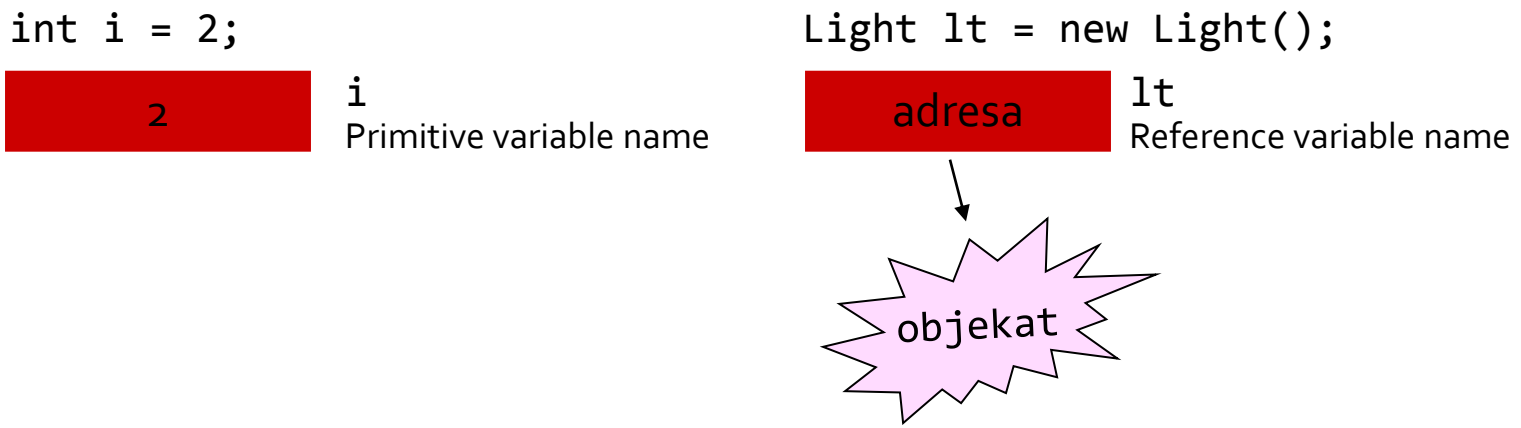
- U OO jezicima sve čime se manipuliše je objekat neke klase♀ , odnosno **referenca na objekat** .
  (čak i sama aplikacija, naravno ne računajući primitivne tipove)

  Objekat – TV

  Referenca – daljinski

- Sa sobom nosite daljinski, a ne televizor. Možete kupiti daljinski i bez televizora.

```
Light lt; // kreirana je samo referenca
Light lt = new Light(); // kreiran je i objekat
```



O objektima i njihovim referencama

References are like labels for your object

Using an int

1. Write a statement to declare the integer.

```
int myInt;
```

2. Assign a value to the new variable.

```
myInt = 3761;
```

3. Use the integer in your code.

```
while (i < myInt) {
```

Using an object

1. Write a statement to declare the object.

```
Dog spot;
```

When you have a class like Dog, you use it as the type in a variable declaration statement.

2. Assign a value to the object.

```
spot = new Dog();
```

3. Check one of the object's fields.

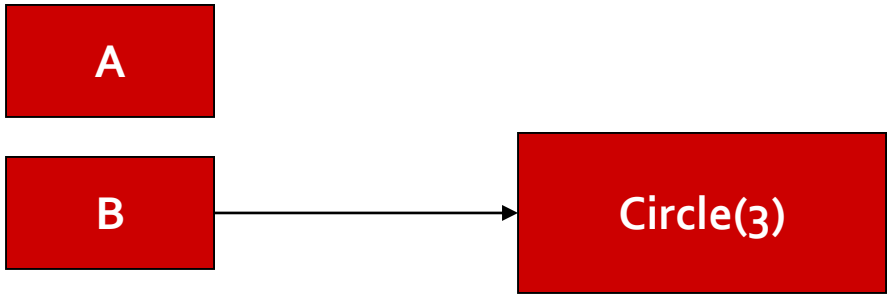
```
while (spot.Happy) {
```

This is the reference type variable named `spot`, and it will refernce an object of type `Dog`

This is the object that `spot` refers to.

O objektima i njihovim referencama

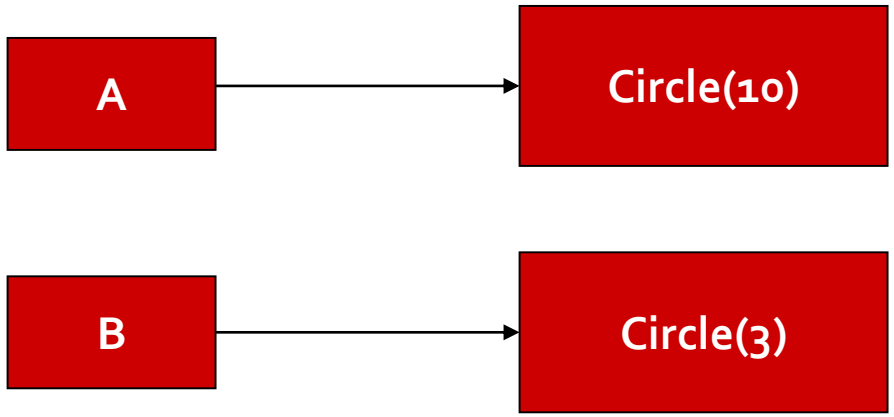
```
int x;
Circle A;
Circle B = new Circle();
B.radius = 3;
```



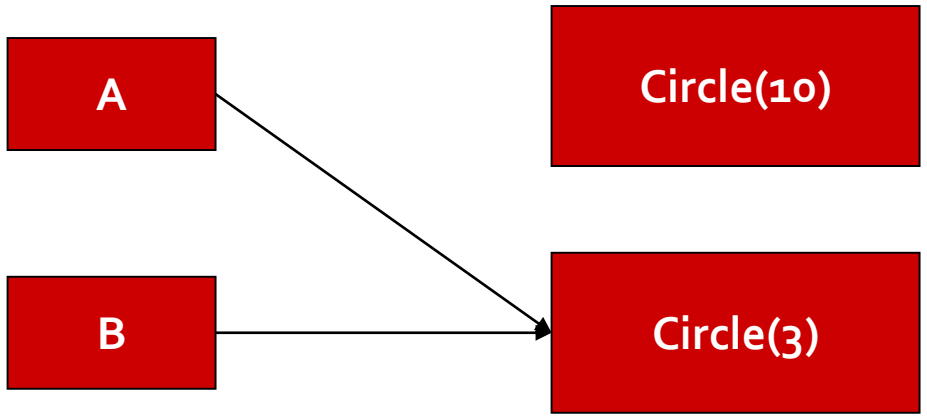

```
double z = A.Area();  // greska, objekat ne postoji
double w = B.Area(); // OK
```

O objektima i njihovim referencama

```
Circle A = new Circle();
A.radius = 10;
```



A = B; - ne kreira se novi objekat vec A postaje referenca na vec postojeci objekat



Metodi (1)

C#

Definicija metoda

- C-like

```
returnType methodName ( parameterList )      // potpis
{
      // telo metoda
}
```

```
int addValues(int leftHandSide, int rightHandSide)
{
   // ...
  return leftHandSide + rightHandSide;
}
```

```
void showResult(int answer)
{
  // display the answer
  ...
  return;
}
```

- Metodi se definišu **samo kao deo klase**. Pozivi pogrešnih metoda za neki objekat se registruju pri kompajliranju.

```
int x = a.f(); // a je objekat odgovarajuce klase
```

Prenos referenci i vrednosti

- Prenos parametara po vrednosti je podrazumevan način prenosa.
- Prenos po referenci se obazbeđuje ključnom rečju ref.

```
class MyClass
{ public int Val = 20; }                    // Initialize field to 20.

class Program          ref modifier         ref modifier
{                          ↓                    ↓
   static void MyMethod(ref MyClass f1, ref int f2)
   {
      f1.Val = f1.Val + 5;                  // Add 5 to field of f1 param.
      f2 = f2 + 5;                          // Add 5 to second param.
   }

   static void Main()
   {
      MyClass A1 = new MyClass();
      int A2 = 10;

      MyMethod(ref A1, ref A2);             // Call the method.
   }           ↑       ↑
}              ref modifiers
```

Prekalapanje metoda (Overloading)

- **OVERLOADING**
  U jednoj klasi (ili pri nasleđivanju) se može definisati više metoda sa istim imenom ali se oni **moraju razlikovati po broju ili bar po tipu argumenata**. Dozvoljeno je i da vraćaju različite tipove (ali pod uslovom da se razlikuju po argumentima).

```
class A
{
    long AddValues( int a, int b) { return a + b; }
    long AddValues( int a, int b, int c) { return a + b + c; }
    long AddValues( float a, float b) { return a + b; }
    long AddValues( long a, long b) { return a + b; }
}
```

```
class B
{
    long AddValues( long a, long b) { return a+b; }
    int AddValues( long c, long d) { return c+d; }
}
```

Illegal !

Konstruktori

Primer 1.

```
class Tacka{
      private double x;
      private double y;
      public Tacka() { x=0.0; y=0.0; }
      public Tacka(double a, double b) { x=a; y=b; }
      public double getX() { return x; }
      public double getY() { return y; }
}
class Test {
 static void Main(String[] args){
  Tacka a = new Tacka();
  Tacka b = new Tacka(1,1);
  System.out.println("A(" + a.getX() + "," + a.getY() + ")\n");
  System.out.println("B(" + b.getX() + "," + b.getY() + ")\n");
}
}
```

konstruktori

- Konstruktor je metod ima isto ime kao i klasa, pozivase isključivo pri instanciranju objekata (dakle, operatorom **new**),
- **nema povratne vrednosti**
- Klasa može imati više konstruktora (overloading na delu)
- Konstruktor bez parametara je **default konstruktor** i on postoji kada klasa nema posebno implementiran (naveden) ni jedan konstuktor videti Primer 1., u suprotnom neće postojati.

Oblast važenja

C#

Oblast važenja

- Oblast važenja (scope) podrazumeva vidljivost i životni vek ‘imena’

```
{ int x = 12;
        /* samo je x dostupno*/
        { int q = 96;
                /* x i q su dostupni */
         }
/* samo x je dostupno a q ‘ne postoji’ */
}
```

- Životni vek objekata nije isti životnom veku primitivnih tipova. Nakon kreiranja objekat postoji i posle }, jer je iz oblasti važenja ‘izašla‘ samo referenca

```
{
        Tacka s = new Tacka();
}
```

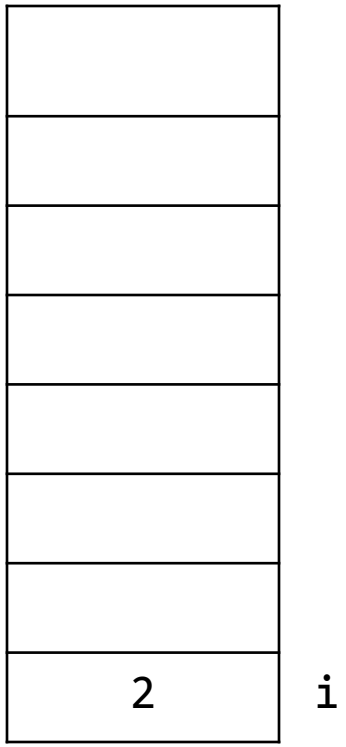
- Kada objekat više nije potreban, tj. nije referenciran ni jednom referencom onda biva automatski oslobođen garbage collector-om

- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```

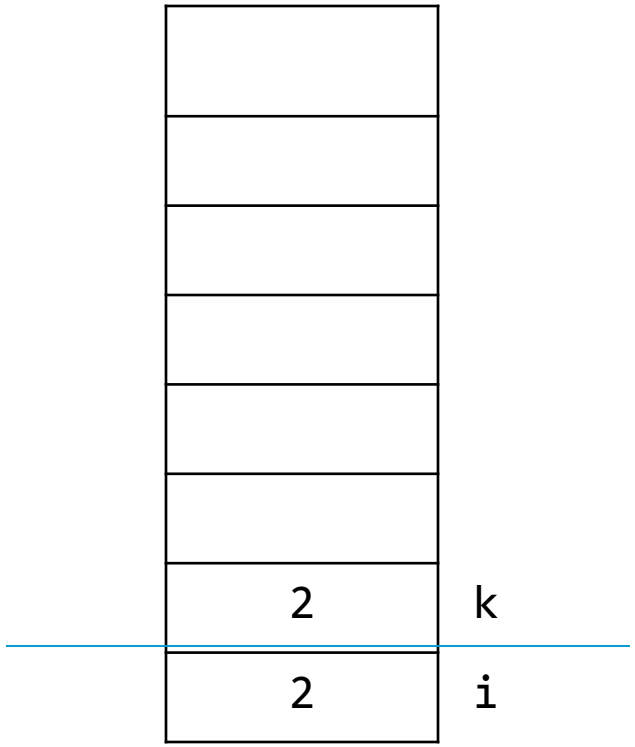
- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```



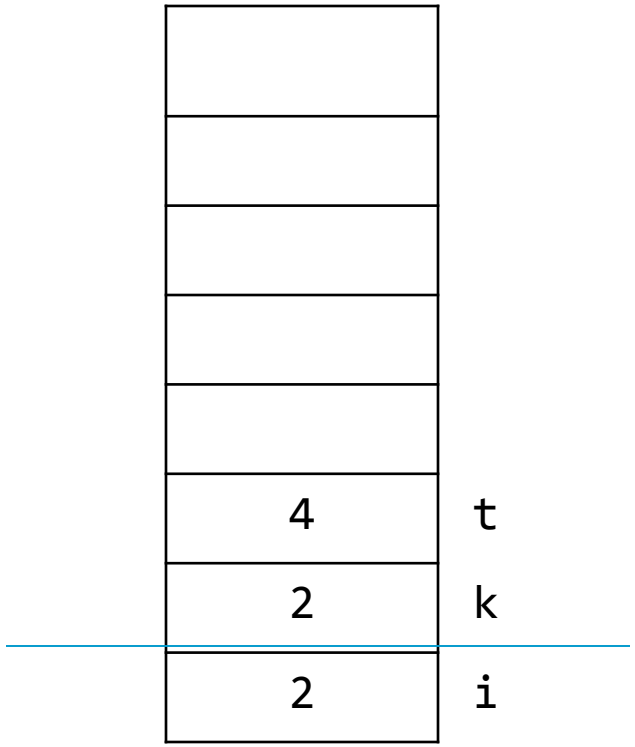
- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```



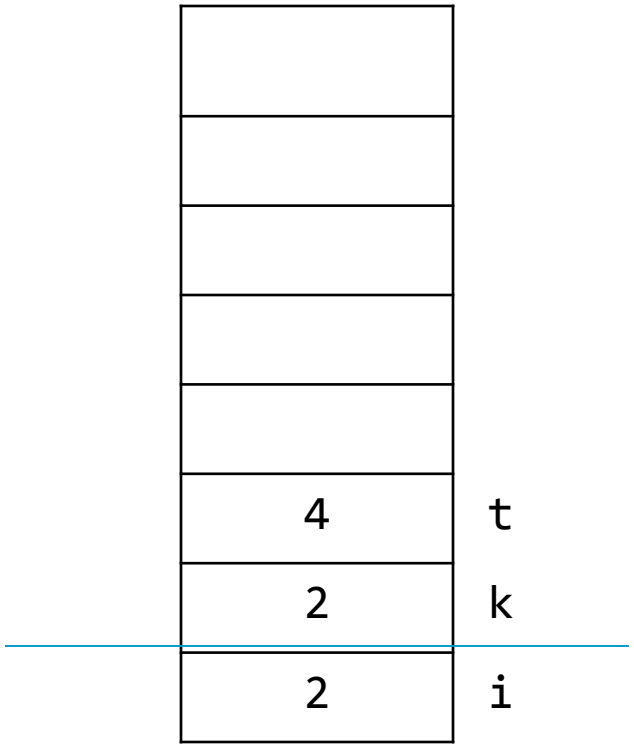
- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```



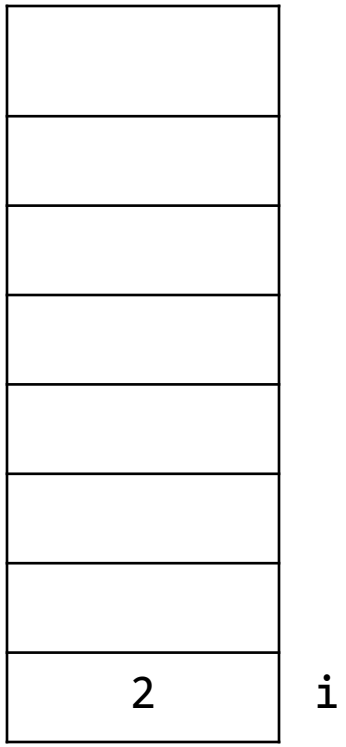
- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```



- **STACK (stek)** je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

```
main(…)
{
  int i = 2;
  func(i);
  . . .
  return;
}

void func(int k)
{
  int t = k * k;
  . . .
  return;
}
```

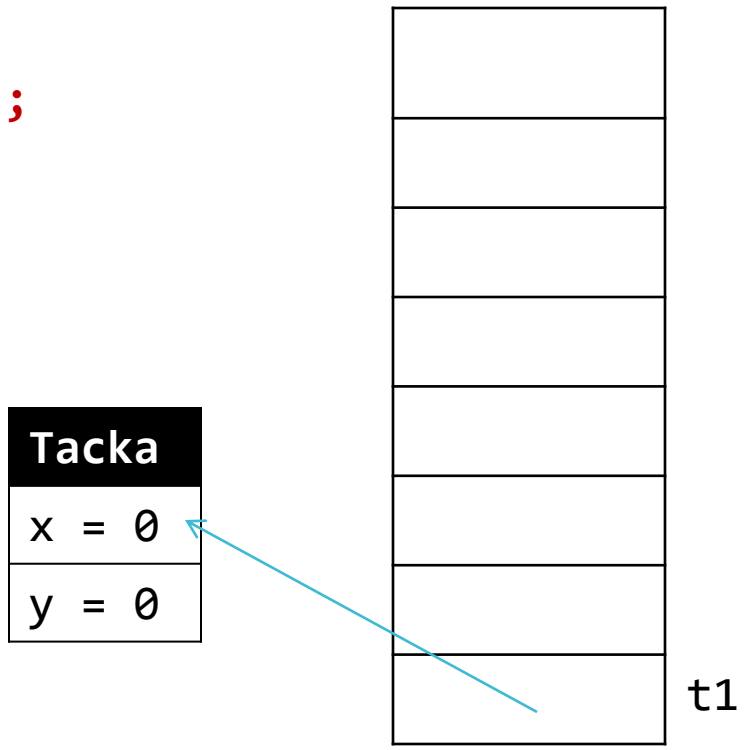


- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```

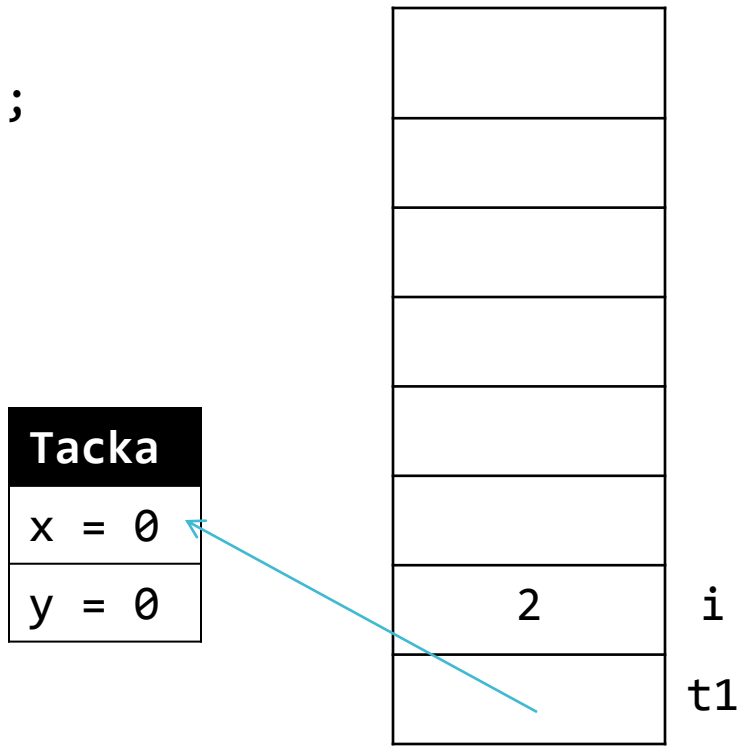
- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```



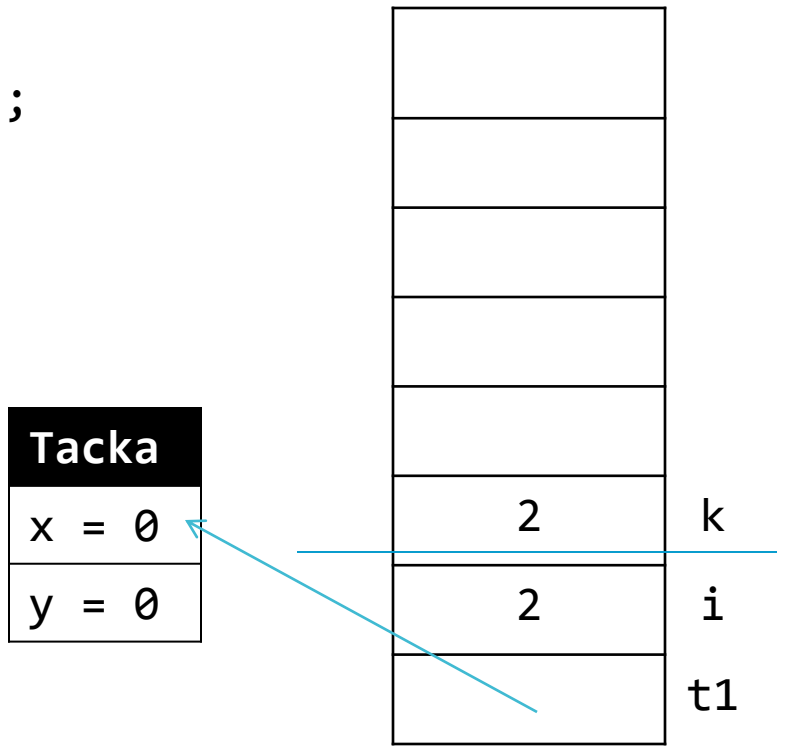
- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```



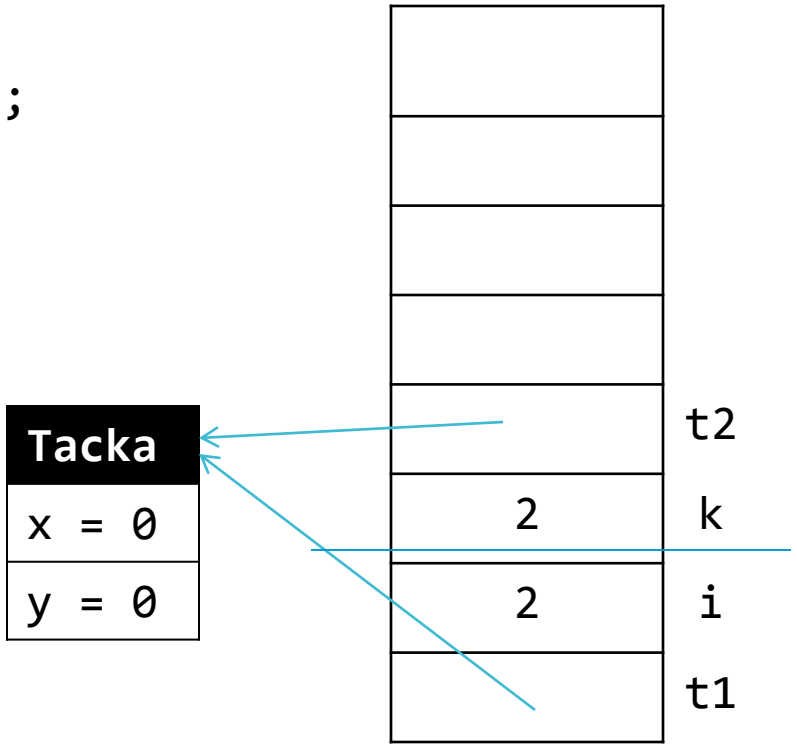
- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```



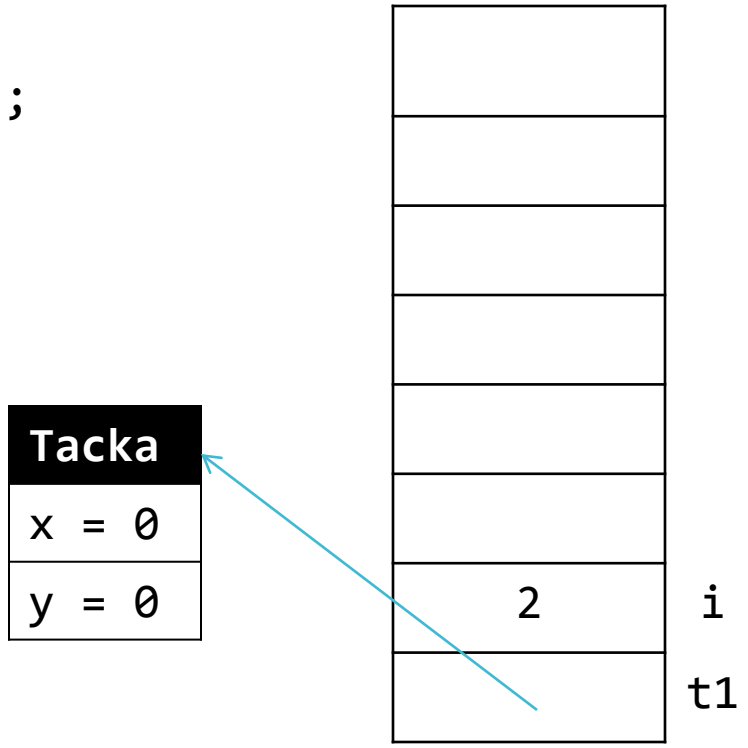
- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```

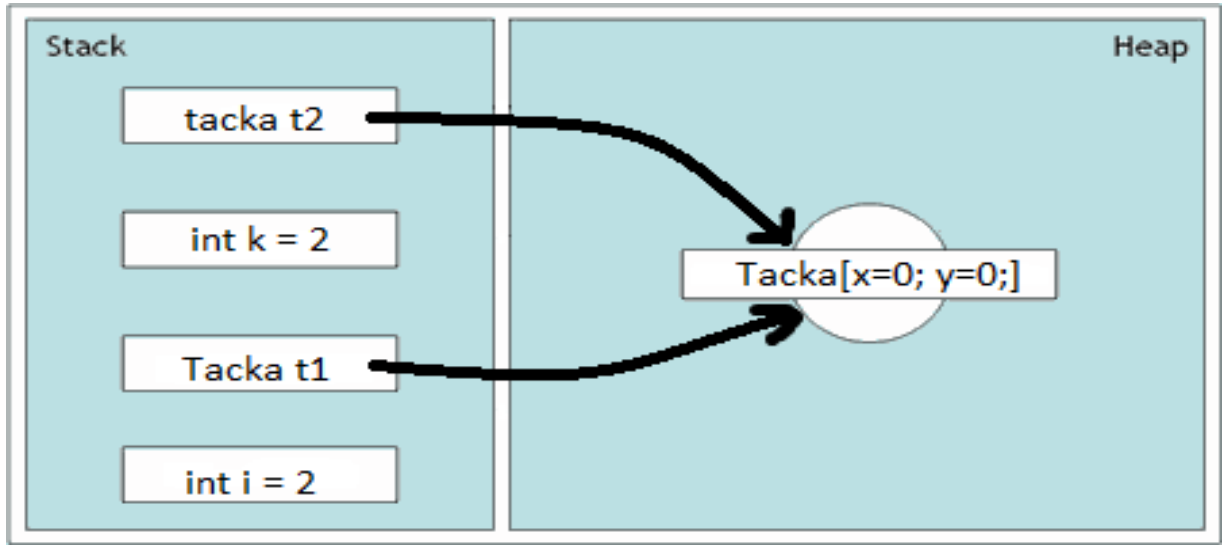


- **STACK (stek)** nije zadužen za čuvanje objekata. Na stek se pakuju samo reference.

```
main(…)
{
  Tacka t1 = new Tacka();
  int i = 2;

  if(i  > 1 )
  {
    k = 2;
    Tacka t2 = t1;
  }

  return;
}
```



- Objekti se čuvaju u delu memorije koji se naziva **HEAP (hrpa)**.
- **HEAP (hrpa)** je deo memorije koji je rezervisan za dinamičke promenljive, tj. za promenljive koje se stvaraju u toku izvršavanja programa.
- Pristup dinamički kreiranim promenljivama se ostvaruje pomoću pokazivača (referenci). **Na jednu istu dinamičku promenljivu može pokazivati i više pokazivača.**



- Kada prestane oblast važenja reference briše se samo referenca sa steka, ali ne i objekat. **Objekat ostaje u memoriji na heapu**.

- JVM vodi računa o broju referenci na objekat, te stoga objekat ne sme biti obrisan iz memorije ukoliko je obrisana jedna referenca. Objekat sme biti obrisan samo kada više ne postoje reference na isti.

- Postoji posebna nit koja vodi računa i uklanja objekte koji nisu referencirani - **Garbage Collector**. Ona se izvršava kada to dozvoljava procesorsko vreme ili kada je neophodno osloboditi memoriju.

Članovi objekata / klasa

C#

static – članovi klase

- Static označava nešto zajedničko svim instancama date klase.

Statički podaci i metodi	podacima/metodama članovima klase
Nestatički podaci i metodi	podacima/metodama članovima objekta

Za **static int x;** kao varijablu članicu klase runtime environment pri učitavanju definicije klase instancu date klase alocira potrebnu memoriju za tu varijablu koju kasnije dele sve druge instance ove klase.

- Metode deklarisane sa static se tretiraju slično.

```
public static void main(String[] args)
```

- Da bismo koristili statičku metodu neke klase nije potrebno imati instancu te klase. Statičkim metodama se pristupa sa:

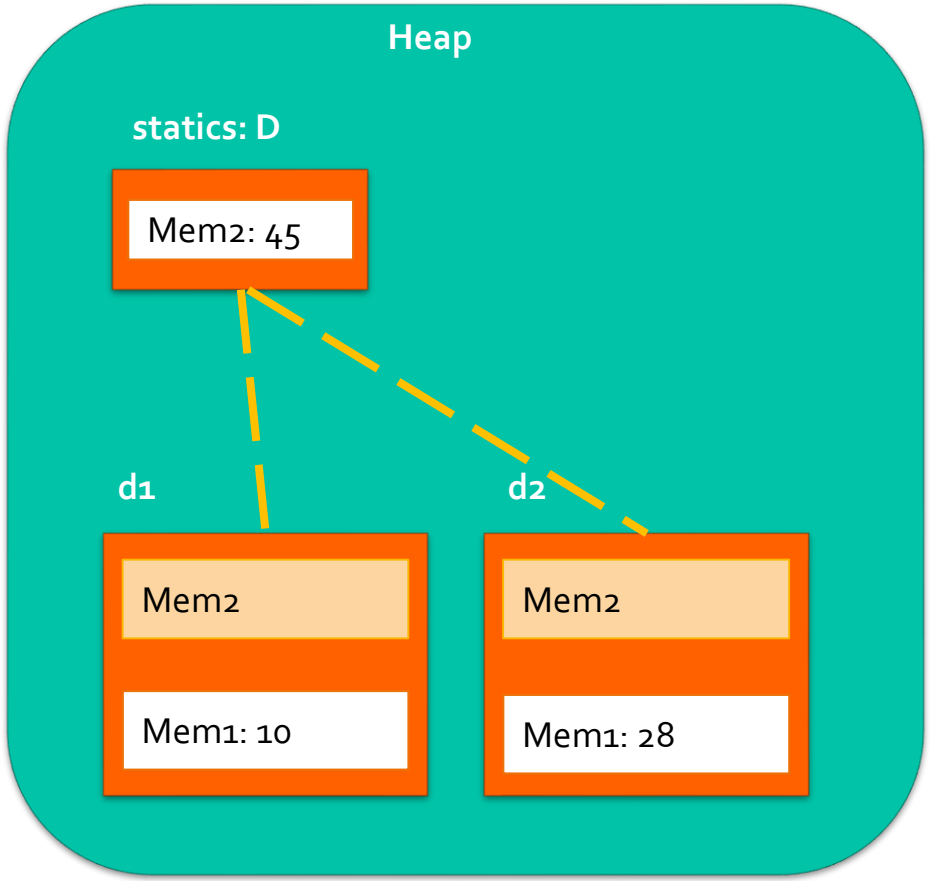
```
(ime_klase).(ime_statičke_metode)
```

npr. `double d = Convert.ToDouble("3.4");`

Na isti način se koriste i statički podaci.

static podaci

```
class D
{
  int Mem1;
  static int Mem2;
  …
}

static void Main()
{
  D d1 = new D();
  D d2 = new D();
  …
}
```



static metodi

- Za statičke metode je važno napomenuti da one **ne mogu koristiti nestatičke** metode iste klase direktno, niti pristupati nestatičkim varijablama.

```
class Program
   {
       static void Main(string[] args)
       {

           Test t = new Test();
           t.y = 10;                // error
           t.print();
           t.print_static();        // error

           Test.x = 3;              // error
         Test.y = 5;
           Test.print();              // error
           Test.print_static();
       }
   }

   class Test {
       public int x;                       // nestatička varijabla
       public static int y;                // statička varijabla

       public void print() {               // nestatička metoda
         Console.WriteLine("Hello, World");
        }
       public static void print_static() { // statička metoda
         x++; //error
           y++; //ok
           Console.WriteLine("Hello, World");
       }
   }
```

Primer

Napisati definiciju tipa Predmet koji ima:

- jedinstven, automatski generisan identifikacioni broj
- jednoslovnu oznaku vrste predmeta
- metode kojim se dobija tip predmeta, kao i njegov ID.

Primer

```
class Predmet{
   char oznaka;
   int id;
   static nextID;
   Predmet() {
        id = nextID;
        nextID++;
   }
   int getId() { return id;}
   char getOznaka { return oznaka;}
}
```

static klasa

- Čitava klasa takođe može da bude statička što znači da može da ima samo statičke članove – podatke i metode

```
public static class Math
{
public static double Sin(double x) {...}
public static double Cos(double x) {...}
public static double Sqrt(double x) {...}
...
}
```

this

C#

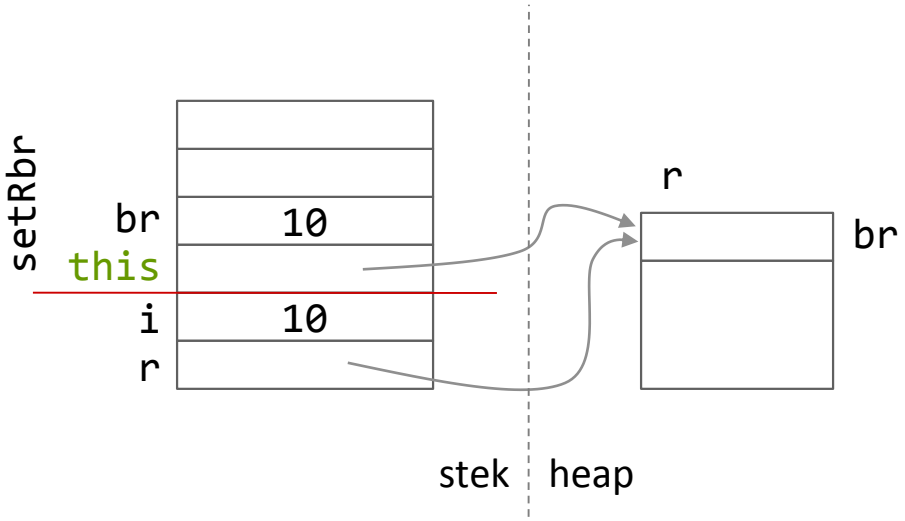
Gde je kod, a gde podaci?

- Objekti su nosioci aktivnosti, medjutim kako može postojati više objekata iste klase nema potrebe na više mesta čuvati jedan te isti kod. **Kod je izdvojen u posebno parče memorije**.

- Objekti čuvaju samo podatke, ne čuvaju kod metoda, već samo adresu gde se nalazi kod u memoriji
- Kako onda računar zna sa kojim objektom radi?

this

```
class Robot {
    int rbr; …
    void setRbr(int br) { rbr = br; } …
}
class Test {
    public static void main(){
      Robot r = new Robot();
      int i = 10;
      r.setRbr(i);
    }
}
```



Metod **setRbr** pored argumenta navednog u definiciji poseduje i implicitni parametar **this** koji sadrži adresu objekta čiji je metod pozvan, tj. kome je prosleđena poruka.

```
void setRbr(int br) { rbr = br; }
```

je isto što i

```
void setRbr(int br) { this.rbr = br; }
```

Dakle, potpis metoda `setRbr`

`void setRbr(int br)` je od strane komplajlera proširen

`void setRbr(Robot this,int br)`

Statički metodi ne dobijaju `this`!

Metod **setRbr** pored argumenta navednog u definiciji poseduje i implicitni parametar **this** koji sadrži adresu objekta čiji je metod pozvan, tj. kome je prosleđena poruka.

```
void setRbr(int br) { rbr = br; }
```

je isto što i

```
void setRbr(int br) { this.rbr = br; }
```

Dakle, potpis metoda `setRbr`

`void setRbr(int br)` je od strane komplajlera proširen

`void setRbr(Robot this,int br)`

Primer

Napisati tip

- kojim se opisuju **Muzicare** i to:
  - imenom,
  - prezimenom,
  - vrstom muzike koju izvodi i
  - cenom jednog sata izvodjenja

  Tip Muzicar treba da sadrži metode kojim se postavljaju i vraćaju vrednosti atributa, sadrži i:
  - Metod `prijavaNaFestival` koja dobija objekat Festivala na koji se Muzicar prijavljuje
  - Metod koji vraća string reprezentaciju muzicara u formi
    ime prezime – vrsta muzike
- kojim se opisuje **Festival** i to:
  - nazivom
  - Programom koji predstavlja listu parova (učesnik,dužina nastupa), gde je dužina nastupa izražena u minutima
  - Danom, mesecom i godinom održavanja (podrazumeva se da je festival jednodnevni)

  Tip Festival treba da sadrži metode kojim se postavljaju i vraćaju vrednosti atributa, sadrži i:
  - Metod upisiIzvodjaca koja dobija Muzičara i dodaje ga na spisak sa podrazumevanom dužinom nastupa 60 minuta
  - Metod upisiIzvodjaca koja dobija Muzičara i vreme trajanja nastupa i dodaje ga na spisak
  - Metod promeniTrajanje koji dobija Muzičara, nalazi ga u listi i menja njegovo vreme
  - Metod koji vraća ukupnu sumu novca koju treba isplatiti
  - Metod koji dobija ime izvođača i izbacuje ga iz programa
  - Metod koji vraća string reprezentaciju festivala u formi
    Naziv i godina
    ime prezime – vrsta muzike
    ime prezime – vrsta muzike
    prijavljenih izvođača

Za definisanje tipa izvođača definisati enumeraciju.
Za realizaciju liste koristiti:

a. Dva niza jedan sa imenima, jedan sa vremenima
b. Dictionary kolekciju

Izvođači neće imati redne brojeve već će nastupati onim redom kojim su se prijavili

Primer

Napisati testnu klasu u kojoj se:

- Kreiraju objekti muzičara Bruno Mars (pop,5000), Soni Mur (elektro,3500), Eminem (rep,4500)
- Kreira objekat Exit festivala sa vremenom održavanja...
- Kreirane muzičare prijaviti na Exit
- Brunu promeniti vreme trajanja nastupa na 90
- Ispisati string preprezentaciju festivala
- Za svakog prijavljenog ispisati njegovo ime i trajanje nastupa

Nasleđivanje

C#

Nasleđivanje

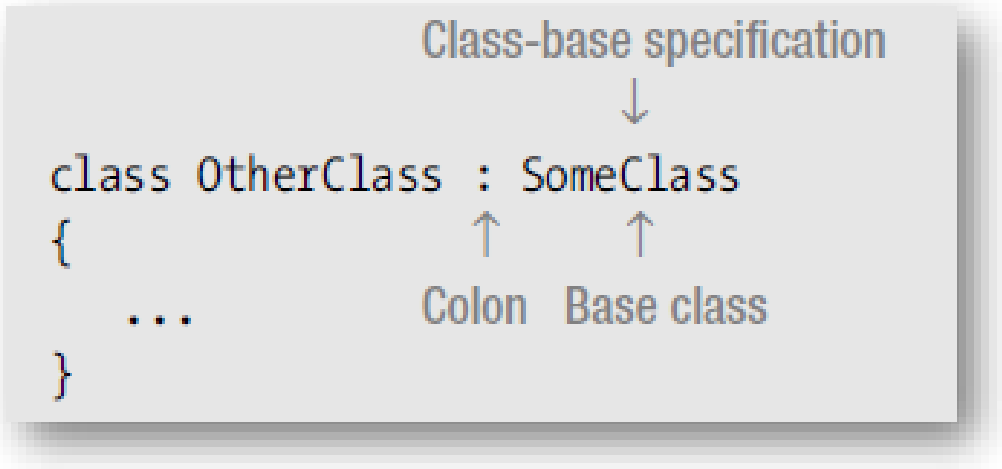
Nasleđivanje (inheritance) je jedan od najvažnijih koncepata OOP-a i predstavlja **mogućnost uvođenja novih tipova/klasa proširivanjem osobina i ponašanja postojećih tipova/klasa**.

Klasa koja nasleđuje (proširuje) se naziva
IZVEDENA, PODKLASA ili DETE-KLASA i
predstavlja uži tip podatka u odnosu na klasu iz koje je izvedena.

Klasa koja biva proširena, tj. čije osobine i ponašanja nasleđuje dete-klasa se naziva
SUPERKLASOM, NADKLASOM ili RODITELJSKOM KLASOM i
predstavlja širi tip podatka u odnosu na klasu iz koje je izvedena.

Nasleđivanje

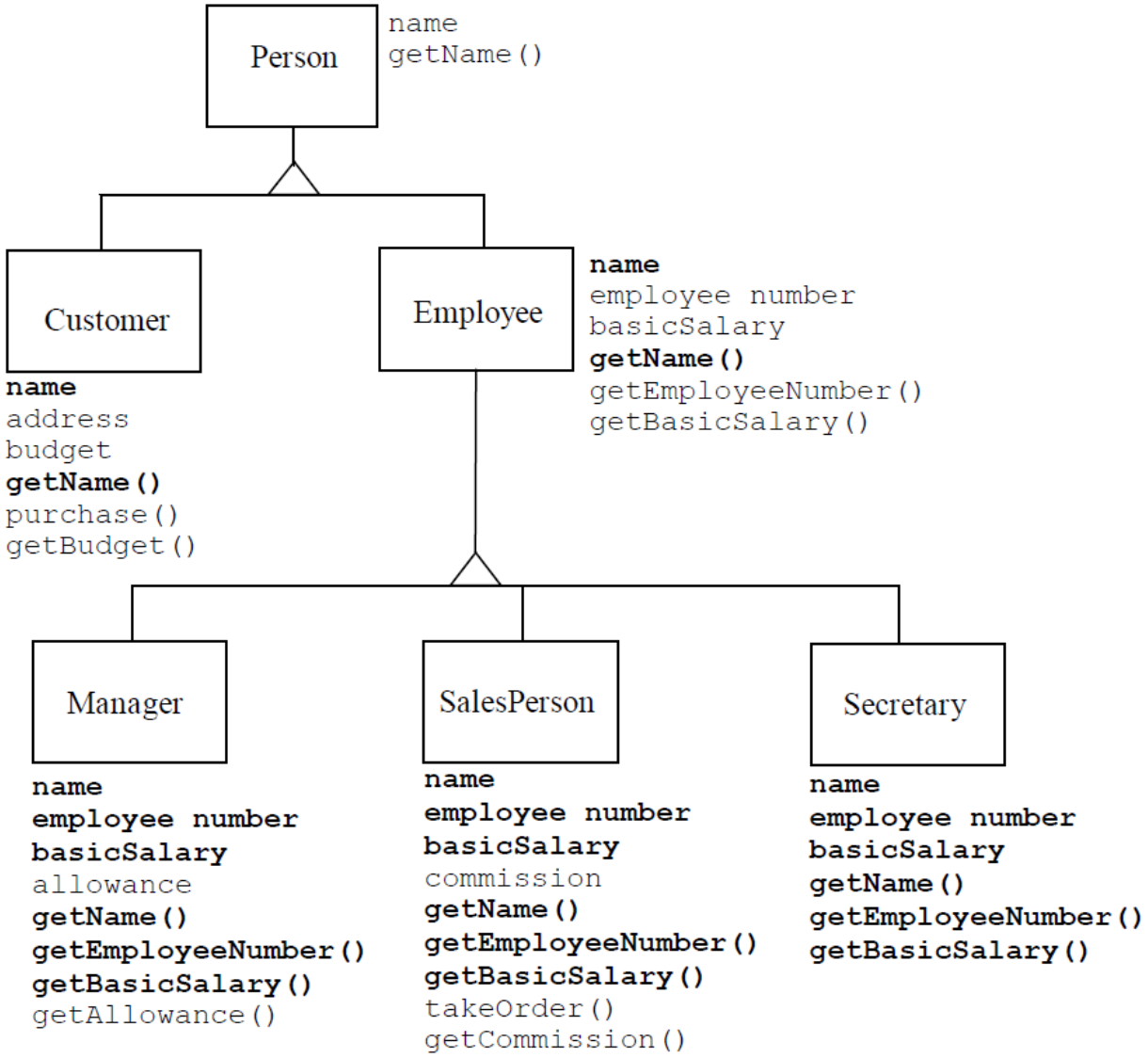
- Svaka klasa može da ima neograničen broj podklasa.
- Podklase nisu ograničene na promenljive, konstruktore i metode klase koje nasleđuju od svoje roditeljske klase.
- Podklase mogu dodati i neke druge promenljive i metode ili predefinisati stare metode.
- U deklaraciji podklase navode se razlike između nje i njene superklase.



- Kažemo da klasa B nasleđuje klasu A, ako:
  - su objekti klase B jedna vrsta objekata klase A, odnosno
  - objekti klase B imaju sve osobine A (i još neke sebi svojstvene).

Dakle, objekti klase B su vrsta objekata klase A, a za vezu klase B sa klasom A kažemo da je tipa a-kind-of .

Primer


Person
name
getName()
Customer
name
address
budget
getName()
purchase()
getBudget()
Employee
name
employee number
basicSalary
getName()
getEmployeeNumber()
getBasicSalary()
Manager
name
employee number
basicSalary
allowance
getName()
getEmployeeNumber()
getBasicSalary()
getAllowance()
SalesPerson
name
employee number
basicSalary
commission
getName()
getEmployeeNumber()
getBasicSalary()
takeOrder()
getCommission()
Secretary
name
employee number
basicSalary
getName()
getEmployeeNumber()
getBasicSalary()

Jednostruko nasleđivanje

- C# podržava **jednostruko nasleđivanje, tj. Svaka klasa može imati samo jednu direktnu nadklasu**.
- U C#, sve klase (i sistemske i naše) su direktno ili indirektno izvedene iz klase **Object**, jer ako se eksplicitno ne navede nadklasa neke klase, onda je ona implicitno izvedena iz **Object**.

Npr. `HelloWorld` se, zapravo, prevodi kao:

```
class HelloWorld : Object {


        }
```

Primer

Napisati na jeziku C# sledeće tipove:

Zaposleni koji sadrži:

- Privatni atribut imeRadnika (String).
- Javni atribut plata koji predstavlja realan broj.
- Privatni atribut bonus koji predstavlja indicator da li zaposleni ima bonus ili ne. Ovaj indicator ima vrednost TRUE ako zaposleni ima bonus a FALSE ako nema.
- Konstruktor koji prima ime radnika i bonus.
- Javnu metodu izracunajPlatu koja ne vraća ništa, ali ima ulazni argument koji predstavlja broj sati koji je zaposleni radio. Vrednost plate se postavlja po formuli broj_sati*osnovni_koeficijent, gde osnovni_koeficijent iznosi 200din.
- getter-e i setter-e za sve promenljive.
- Metod toString koji ispisuje podatke o zaposlenom u obliku imeRadnika:Tip

KancelarijskiRadnik je Zaposleni koji sadrži:

- Privatni atribut straniJezik (String).
- Konstruktor koji prima ime radnika i strani jezik i zna se da kancelarijski radnik nema bonus.
- Javnu metodu izracunajPlatu koja ne vraća ništa, ali ima ulazni argument koji predstavlja broj sati
- koji je zaposleni radio. Vrednost plate se računa tako što se na osnovnu platu koja mu pripada kao
- zaposlenom dodaje još broj_sati*koeficijent_kancelarijskog_radnika, gde
- koeficijent_kancelarijskog_radnika iznosi 120din.
- Metod toString koji ispisuje podatke o kancelarijskom radniku u obliku imeRanika: (plata, nema bonus)

Primer

Tip **Menadzer** je Zaposleni koji sadrži:

- Privatni atribut teren koji predstavlja indicator da li menadzer izlazi na teren ili ne. Ovaj indicator ima vrednost TRUE ako izlazi na teren FALSE ako ne izlazi.
- Konstruktor koji prima ime radnika i informaciju o terenu , kao i informacija o bonusu.
- Javnu metodu izracunajPlatu koja ne vraća ništa, ali ima ulazni argument koji predstavlja broj sati koji je zaposleni radio. Vrednost plate se računa tako što se na osnovnu platu koja mu pripada kao zaposlenom dodaje još broj_sati*koeficijent_menadzera, gde koeficijent_menadzera iznosi 150din. Ukoliko menadzer ima bonus na platu se dodaje 1000din a ukoliko ide na teren dodaje se još 500din.
- Metod toString koji ispisuje podatke o kancelarijskom radniku u obliku imeRanika: (plata, informacija o bonusu), gde informacija o bonusu podrazumeva ispisano DA ili NE.

Napraviti **Testnu** klasu i u njoj:

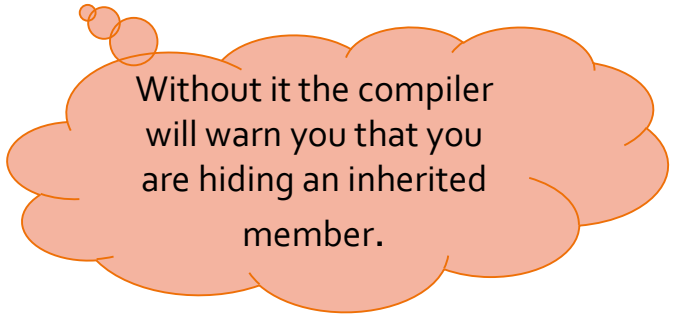
- Kreirati jedan objekat klase KancelarijskiRadnik koji se zove Marko koji zna Nemacki jezik.
- Zatim kreirati drugi objekat klase Menadzer koji se zove Nikola, ne ide na teren i nema bonus.
- Ukoliko je Marko radio 20 sati a Nikola 15, odštamati podatke o onom zaposlenom koji ima veću platu.
- Zatim kreirati niz od 4 zaposlena o Kancelarijski radnik Misa koji zna Francuski o Menadzer Sale koji ne ide na teren ali ima bonus o Kancelarijski radnik Mimi koji zna Engleski o Menadzer Miki koji ide na teren i nema bonus
- Izračunati plate svim zaposlenima ako su svi radili po 30 sati i ispisati podatke o svim zaposlenima.
- Ispisati koliko je novca ukupno isplaćeno za plate zaposlenima.
- Ispisati koliko je novca isplaceno za plate za kancelarijske radnike a koliko za menadzere.

C#

Manipulisanje nasleđenim elementima

Sakrivanje nasleđenih članova

- Izvedena klasa **ne može izgubiti/izbrisati** nijedan od nasleđeniih članova, ali ih **može sakriti / prekriti**.
- Prekrivanje se vrši definisanjem novog člana sa istim tipom i imenom kada su u pitanju podaci, a sa istim potpisom kada su u pitanju metodi.
- Pri prekrivanju je potrebno ispred člana kojim se prekriva navesti ključnu reč **new**.
- Mogiće je sakriti i statičke članove.



Sakrivanje nasleđenih članova

```
class SomeClass                                          // Base class
{
   public string Field1 = "SomeClass Field1";
   public void   Method1(string value)
       { Console.WriteLine("SomeClass.Method1:  {0}", value); }
}


class OtherClass : SomeClass                            // Derived class
{ Keyword
    ↓
   new public string Field1 = "OtherClass Field1";  // Mask the base member.
   new public void   Method1(string value)          // Mask the base member.
    ↑   { Console.WriteLine("OtherClass.Method1:  {0}", value); }
} Keyword


class Program
{
   static void Main()
   {
      OtherClass oc = new OtherClass();       // Use the masking member.
      oc.Method1(oc.Field1);                  // Use the masking member.
   }
}
```

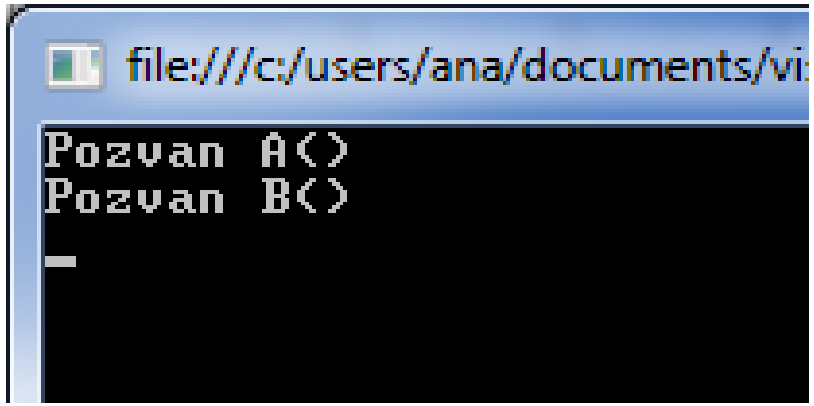
Konstrukcija objekata

C#

Konstrukcija objekata izvedene klase

- Konstrukcija objekata izvedene klase teče tako što prvo biva pozvan konstruktor nadklase, a zatim biva izvšeno telo kontruktora izvedene klase.

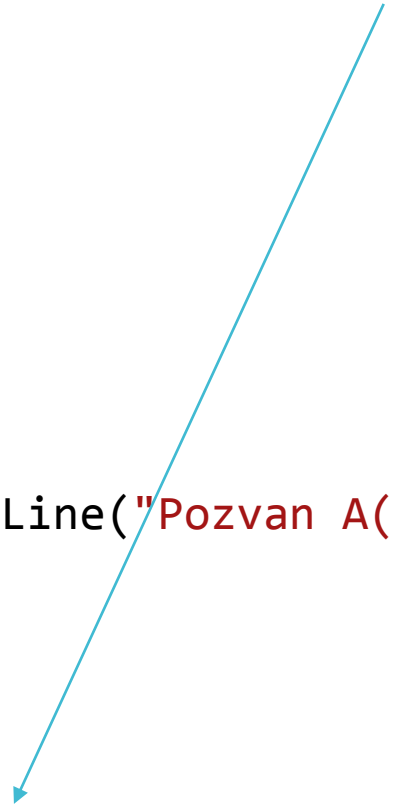
```
static void Main(string[] args)
        {
            B ojekatB = new B();
        }

        public class A
        {
            public A() {
                Console.WriteLine("Pozvan A()");
            }
        }

        public class B : A
        {
            public B() {
                Console.WriteLine("Pozvan B()");
            }
        }
```



Ekspilicitni poziv konstruktora nadklase

- Poziv specifičnog (nepodrazumevanog) konstruktora se obavlja navođenjem ključne reči **base**.

```
static void Main(string[] args)
        {
            B ojekatB = new B();
        }

                        public class A
                               {
                                   public A(int x) {
                                       Console.WriteLine("Pozvan A({0})",x);
                                   }
                               }

                               public class B : A
                               {
                                   public B() : base(10) {
                                       Console.WriteLine("Pozvan B()");
                                   }
```



Kompatibilnost tipova

C#

Kompatibilnost tipova

- C# je strogo tipiziran jezik

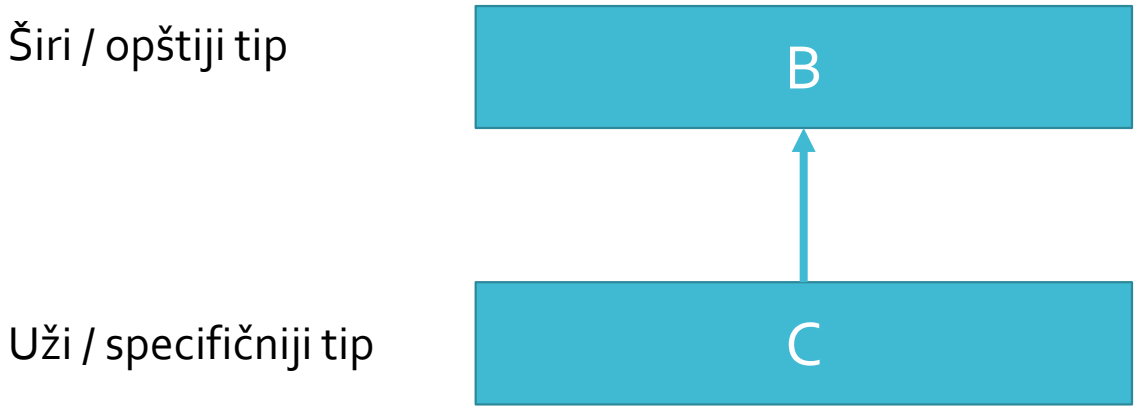
  kompatibilnost tipova se proverava tokom prevodjenja

- Pri operaciji dodele, inicijalizaciji, prenosu parametara i vraćanju rezultata metoda tip izraza mora biti kompatibilan tipu promenljive kojoj se izraz dodeljuje.
- Dve osnovne grupe tipova su:
  - Vrednosni tipovi – primitivni tipovi (float, double, int, ...) i korisnički definisani tipovi (upotrebom `struct` konstrukcije)
  - Referencni tipovi – klase, interfejsi, delegati
- Tip reference koja je rezultat izraza mora biti
  - Tip promenljive kojoj se dodeljuje
  - Njegov podtip
  - ili null

Konverzija

- Nadtipovi su manje posebni, smatraju se širim i nalaze se više u hijerahiji tipova
- Konverzija proširivanja (naviše, nagore)
  - Podtip se konvertuje u nadtip
  - Automatski
- Konverzija sužavanja (naniže, nadole)
  - Nadtip se konvertuje u podtip
  - Eksplicitno (cast)

```
class D : B { ... }

...

D d1 = new D();

B b1=new B();

B b = d1;        // naviše

D d = (D)b1;  // naniže, obavezan cast
```



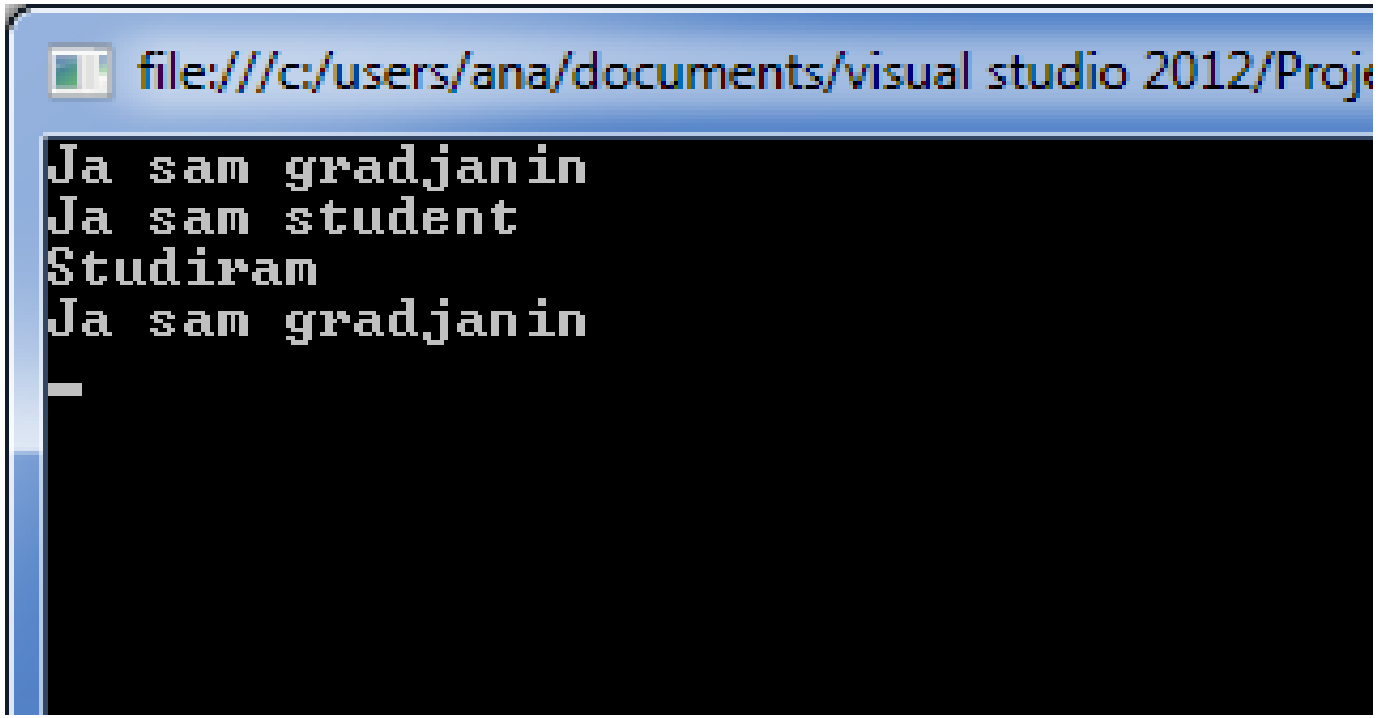
```
class Gradjanin {                          6
  public string ime;
  public void jaSam() {
     Console.WriteLine("Ja sam
           gradjanin");
  }
}

class Student : Gradjanin {
  public string fakultet;
  public void jaSam() {
     Console.WriteLine("Ja sam
                  student");
  }
  public void studiram() {
     Console.WriteLine("Studiram ");
  }
}
```

Unable to cast object of type 'Gradjanin' to type 'Student'

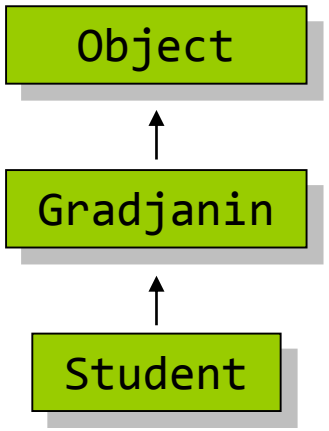
```
Gradjanin g1 = new Gradjanin();
Student s1 = new Student();
g1.jaSam();
s1.jaSam();
s1.studiram();

Gradjanin g2 = s1;
g2.jaSam();


Console.ReadKey();
```

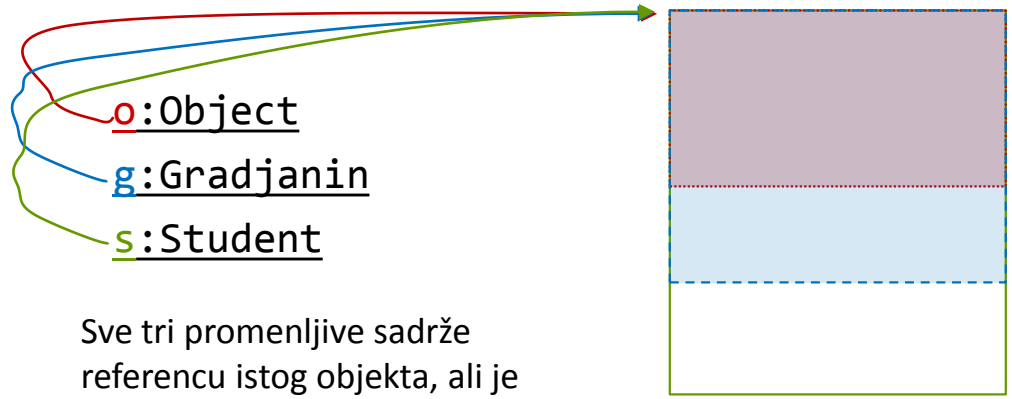
Ponašanje u skladu sa tipom reference, a ne objektom koji se nalazi iza.

Rano vezivanje poziva metoda sa kodom koji treba da se izvrši.



Konverzija sužavanja - CAST





Sve tri promenljive sadrže referencu istog objekta, ali je dostupnost podataka i metoda objekta na koji ukazuju različita.

- g se može koristiti **SAMO** za poziv metoda koji su definisani i u klasi Gradjanin.
- ako je preko reference g potrebno pozvati metod specifičan za Student klasu koristeći referencu g, NEOPHODAN je cast g na Student.

```
((Student)g).studiram();
```

Virtual i override

C#

Prekrivanje/prepisivanje nasleđenih metoda

- Polimorfno prepisivanje nasleđenih metoda

  Vrši se kada je potrebno da dete-klasa izmeni ponašanje opisano nasleđenim metodom.

- Prepisivanje se u C# izvodi:
  - Metod u izvedenoj i osnovnoj klasi imaju isti potpis i povratni tip.
  - Metod osnovne klase je označen sa **virual**.
  - Metod u izvedenoj klasi je označen sa **override**.

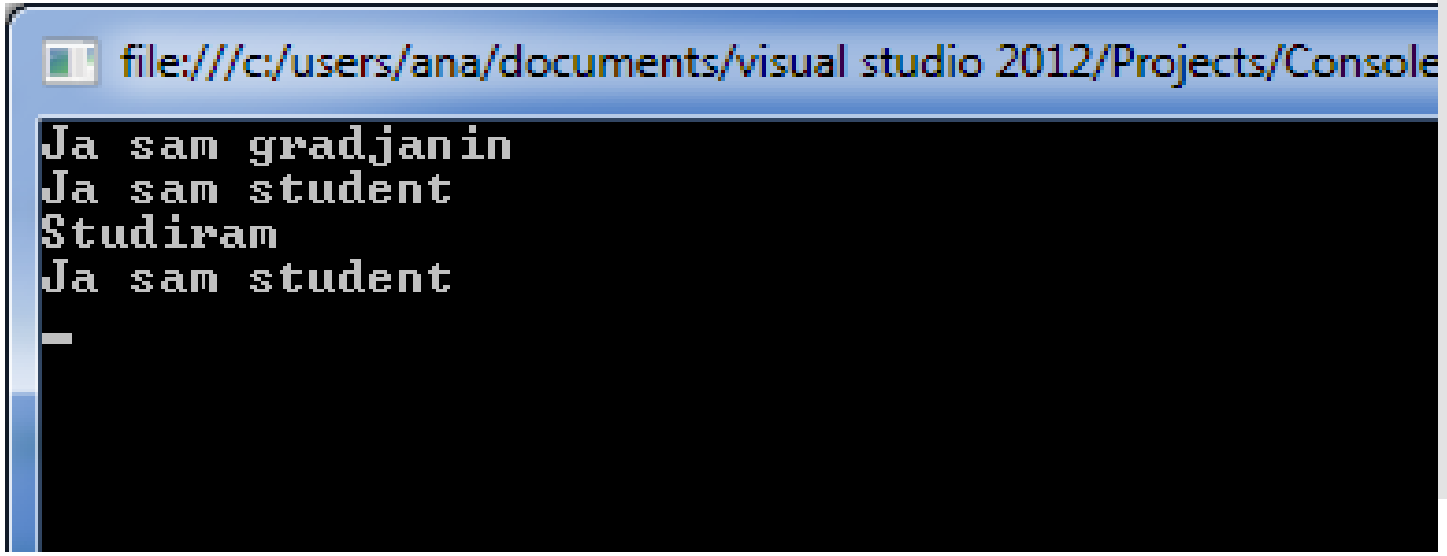
```
public class Ptica
{
    public virtual void Leti()
    {
        //neki kod
    }
    ….
}

public class Pingvin : Ptica
{
    public override void Leti
    {
        Messagebox.Show("Pingvini ne lete");
    }
}
```

```
class Gradjanin {
  public string ime;
  public vitrual void jaSam() {
     Console.WriteLine("Ja sam gradjanin");
  }
}


class Student : Gradjanin {
  public string fakultet;
  public override void jaSam() {
     Console.WriteLine("Ja sam student");
  }
  public void studiram() {
     Console.WriteLine("Studiram ");
  }
}
```

```
Gradjanin g1 = new Gradjanin();
Student s1 = new Student();
g1.jaSam();
s1.jaSam();
s1.studiram();

Gradjanin g2 = s1;
g2.jaSam();


Console.ReadKey();
```

Ponašanje specifično za objekat na koji je referenca usmerena.

Kasno vezivanje poziva metoda sa kodom koji treba da se izvrši.



- mogućnost da varijablom određenog tipa referenciramo objekte različitih tipova i da automatski pozivamo metode koje su specifične za tip objekta na koji varijabla referencira
- uslovi
  - Poziv metoda podklase kroz varijablu bazne klase
  - Pozvana metoda mora biti i član bazne klase
  - Signatura metode i povratni tip moraju biti isti i u baznoj i u izvedenoj klasi
  - Metodi moraju imati istu dostupnost
  - Polimorfizam se odnosi samo na metode – reference baznog tipa mogu se koristiti samo za pristup podacima članovima baznog tipa
- Ne mogu se prepisati static (tj. statički metodi ne mogu biti virtuelni), niti ne-virtuelni metodi.

Enkapsulacija (učaurivanje)

C#

- Učaurivanje (enkapsulacija) je mogućnost skrivanja podataka i metoda koji pripadaju objektu/klasi.
- To znači da je moguće kontrolisati pristup/upotrebu klase, metoda ili varijable.
- Pod pristupom nekoj klasi se podrazumeva mogućnost da se u drugoj klasi (npr. testnoj) može:
  - Kreirati instanca date klase
  - Data klasa može proširiti
  - Pristupi određenom metodu ili varijabli te klase ili njenog objekta

Kontrola pristupa (Access control) omogućava enkapsulaciju obezbeđivanjem kontrole vidljivosti klase, metoda ili varijabli.

Modifikatori pristupa članovima C#	Access
public	Nema ograničenja. Članovi su dostupni svim metodima bilo koje klase.
protected	Članovi klase A označeni sa protected su dostupni metodima klase A i metodima klasa izvedenih iz klase A.
internal	Članovi klase A koji su označeni sa internal su dostupni metodama klasa u istom *projektu* (assembly).
protected internal	Članovi klase A koji su označeni sa protected internal su dostupni metodima klase A, metodima klasa izvedenim iz klase A i bilo koje klase u istom assembly-ju (protected OR internal)
Private (podrazumevana vidljivost)	Članovi klase A označeni sa private su dostupni samo metodima date klase.

```
SuperChef mySuperChef = new SuperChef();
int loyalCustomerOrderAmount = 94;

mySuperChef.cookieRecipe = "get 3 eggs, 2 1/2 cup flour, 1 tsp
salt, 1 tsp vanilla and 1.5 cups sugar and mix them together.
Bake for 10 minutes at 250. Yum!";

string recipe = mySuperChef.GetRecipe(56);
```

```
public class SuperChef
{
        public string cookieRecipe;
        private string secretIngredient;
        private const int loyalCustomerOrderAmount = 60;
        public int Temperature;
        private string ingredientSupplier;

        public string GetRecipe (int orderAmount)
        {
                if (orderAmount >= loyalCustomerOrderAmount)
                {
                        return cookieRecipe + " " + secretIngredient;
                }
                else
                {
                        return cookieRecipe;
                }
        }
}
```

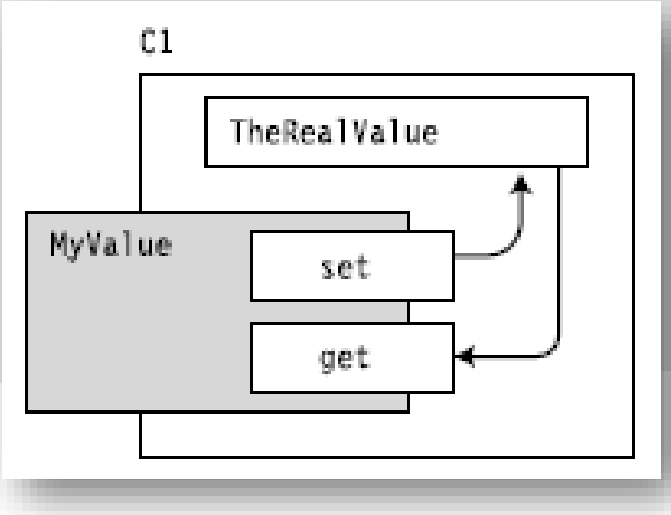
PROPERTIES

Property je imenovani skup dva uparena metoda nazvana accessors:

- **set** accessor se koristi za dodelu vrednosti privatnoj varijabli koja pamti vrednost.
- **get** accessor koji se vraća vrednost pripadajuće implicitne varijable

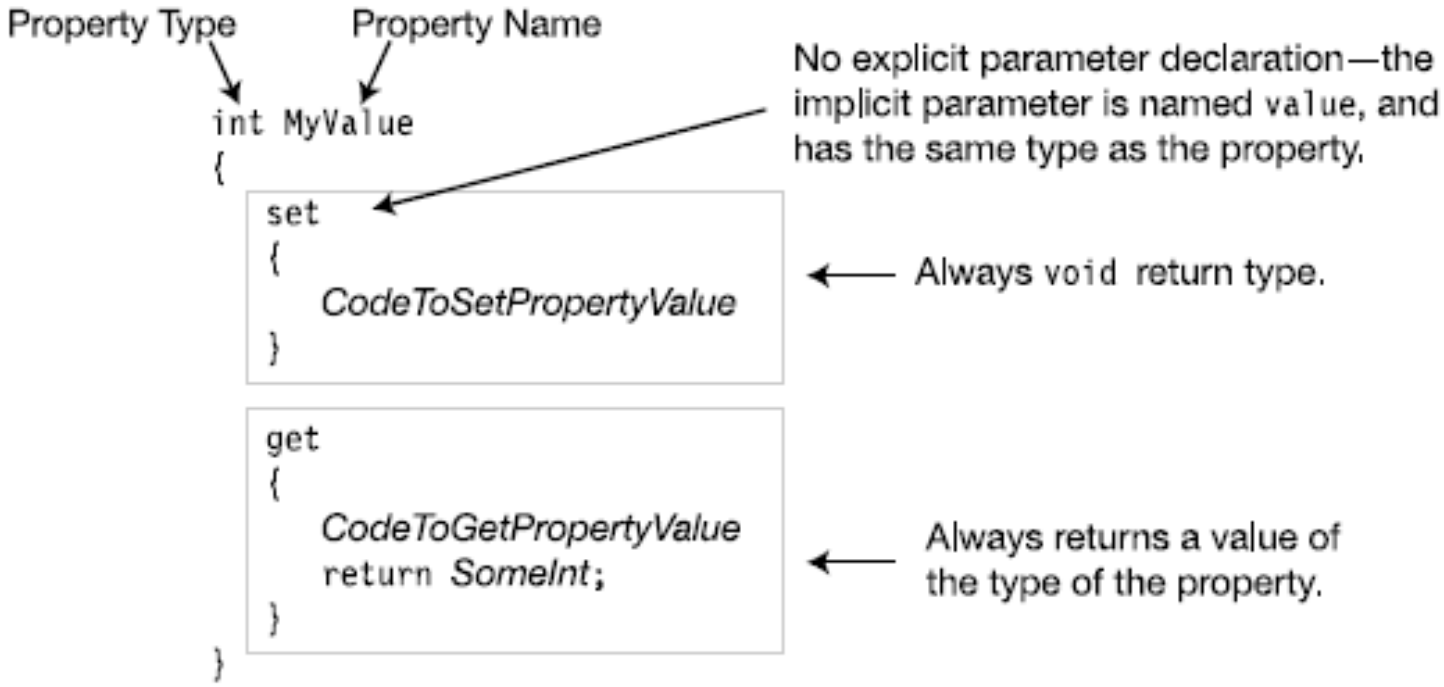
Dozvoljava klijentu da čita/menja stanje objekta kao da pristupa podacima članovima, pri čemu se zaista izvršavaju metodi.

```
class C1
{
   private int TheRealValue;              // Field: memory allocated

   public int MyValue                     // Property: no memory allocated
   {
      set
      {
         TheRealValue = value;
      }

      get
      {
         return TheRealValue;
      }
   }
}
```



PROPERTIES

```
class Program
    {
        static void Main(string[] args)
        {
            Figura f = new Figura("<3");
            f.Oznaka = "<:(";
            Console.WriteLine(f.Oznaka);
            Console.ReadKey();
        }
    }

    class Figura
    {
        string oznaka;

        public string Oznaka
        {
            get { return oznaka; }
            set { oznaka = value; }
        }

        public Figura(string oznaka)
        {
            this.oznaka = oznaka;
        }

    }
```



ACCESSORS

set accessor

- Jedan implicitni parameter value, istog tipa kao i property.
- void povratni tip

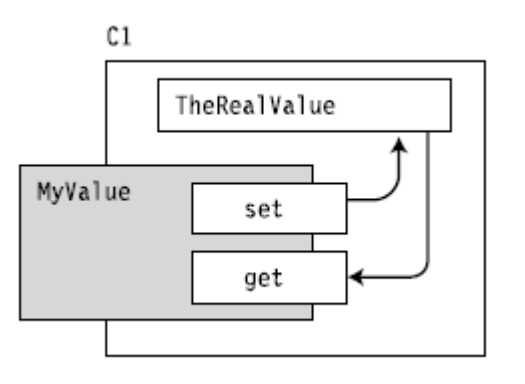
get accessor

- Nema paramatre
- Povratni tip je isti kao i property tip.
- Mora da ima return po svakoj liniji izvršavanja u razgranatom scenariju.

Raspored set i get accessor-a je nebitan.
Implicitni parametar value se tretira kao isvaka druga promenljiva.
Upotreba property-a je identična upotrebi atributa (iako suštinski nisu isto).

ACCESSORS

```
class C1
{
   private int TheRealValue;              // Field: memory allocated

   public int MyValue                     // Property: no memory allocated
   {
      set
      {
         TheRealValue = value;
      }

      get
      {
         return TheRealValue;
      }
   }
}
```



Property name

```
MyValue = 5;            // Assignment: the set method is implicitly called
z = MyValue;            // Expression: the get method is implicitly called
```

Property name

```
y = MyValue.get();      // Error! Can't explicitly call get accessor.
MyValue.set(5);         // Error! Can't explicitly call set accessor.
```

Properties and Associated Fields

- property je obično povezan sa konkretnim poljem u klasi.
- Polje za koje je vezan neki propery se naziva *backing field* or *backing store.*
- Uobičajeno je da se polje deklariše kao private, a da se za njega veže public property preko kojeg je moguće spolja manipulisati vrednostima u datom polju.

```
class C1
{
   private int TheRealValue = 10;    // Backing Field: memory allocated
   public  int MyValue               // Property: no memory allocated
   {
      set{ TheRealValue = value; }  // Sets the value of field TheRealValue
      get{ return TheRealValue; }   // Gets the value of the field
   }
}

class Program
{
   static void Main()
   {
                                  Read from the property as if it were a field
      C1 c = new C1();                          ↓
      Console.WriteLine("MyValue:  {0}", c.MyValue);

      c.MyValue = 20;     ←   Use assignment to set the value of a property
      Console.WriteLine("MyValue:  {0}", c.MyValue);
   }
}
```

Operacije u accessor-ima

- Accessor-i nisu ograničeni samo na dodelu ili preuzimanje vrednosti iz nekog polja.

```
int TheRealValue = 10;                          // The field
int MyValue                                     // The property
{
   set {                                        // Sets the value of the field
      TheRealValue = value > 100                // but makes sure it's not > 100
                     ? 100
                     : value;
   }
   get {                                        // Gets the value of the field
      return TheRealValue;
   }
}
```

Read-Only and Write-Only Properties

```
class RightTriangle
{
   public double A = 3;
   public double B = 4;
   public double Hypotenuse                    // Read-only property
   {
      get{ return Math.Sqrt((A*A)+(B*B)); }    // Calculate return value
   }
}

class Program
{
   static void Main()
   {
      RightTriangle c = new RightTriangle();
      Console.WriteLine("Hypotenuse:  {0}", c.Hypotenuse);
   }
}
```

Automatska implementacija Property-a

- Moguće je deklariasti property bez deklarisanja vazanog polja (Auto-implemented properties)
- Nisu dozvoljeni read-only i write-only auto-implemented properties.

```
class C1
{                ← No declared backing field
   public int MyValue                          // Allocates memory
   {
      set; get;
   }     ↑    ↑
}    The bodies of the accessors are declared as semicolons.

class Program
{
   static void Main()
   {                          Use auto-implemented properties as regular properties.
      C1 c = new C1();                                    ↓
      Console.WriteLine("MyValue:  {0}", c.MyValue);

      c.MyValue = 20;
      Console.WriteLine("MyValue:  {0}", c.MyValue);
   }
}
```

```
MyValue:  0
MyValue:  20
```

Modifikatori klasa

C#

Modifier	Function
public	C# classes marked *public* are available to everyone.
internal	C# classes marked internal are available only within the current project.
protected	Nested C# classes marked protected are available to the containing class or from types derived from the containing class.
protected internal	Nested C# classes marked protected internal are available to the assembly or types derived from the containing class.
private	Nested C# classes marked private are available to be used only on the containing class.
sealed	Classes marked sealed can not be subclassed.
abstract	C# classes marked abstract cannot be instantiated. Subclasses may be instantiated if they implement all the abstract methods (if any).
static	Contain only static members and cannot be used to instantiate objects (such as Console). A static class can contain only static members and can't have instance constructors, since by implication it can never be instantiated. Static classes can, however, have a static constructor. (statički konstruktori - specijalan slučaj, pozivaju se bez kontrole korisnika, jednom pre intanciranja, ukoliko je ono moguće, nemaju modifikator public/private ..., niti imaju parametre) You cannot inherit from static classes—they're sealed.

Class must be marked static

```
static public class MyMath
{
   public static float PI = 3.14f;
   public static bool IsOdd(int x)
                     { return x % 2 == 1; }
   public static int Times2(int x)
                    { return 2 * x; }
}

class Program
{
   static void Main( )
   {
      int val = 3;
      Console.WriteLine("{0} is odd is {1}.", val, MyMath.IsOdd(val));
      Console.WriteLine("{0} * 2 = {1}.",     val, MyMath.Times2(val));
   }
}
```

Members must be static

Use class name and member name.

Apstraktne klase

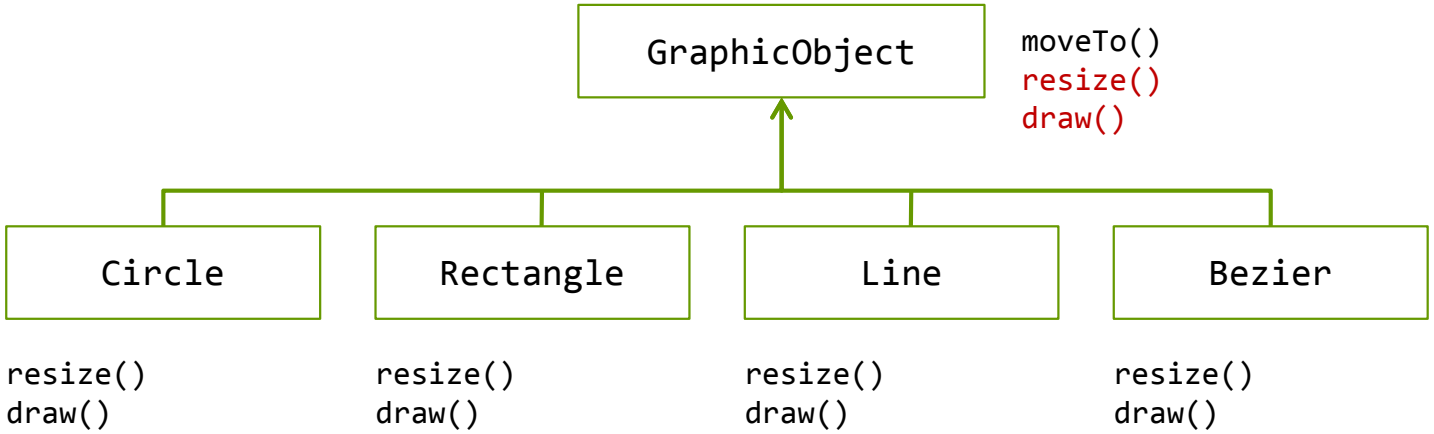
C#

Motivacija

- Pretpostavite da pred sobom imate jednu oop aplikaciju za vektorsko crtanje. Neki od standardnih **grafičkih objekata** koje bi aplikacija trebala da nudi su: krugovi, pravougaonici, linije, Bezijerove krive.
  Ovi objekti imaju neka zajednička:
  - **stanja** određena pozicijom, orjentacijom, bojom konture, bojom unutrašnjeg dela objekta itd. i
  - **ponašanja**, kao što su pomeranje na novu poziciju (moveTo()), rotacija, resize(), iscrtavanje (draw())

  Neka od navedenih stanja i ponašanja su
  - **ista** za sve grafičke objekte – pozicija, boja konture i moveTo(), a neka zahtevaju
  - potpuno **različite implementacije**, kao što su resize() i draw() .

  Za svaki grafički objekat koji želimo da koristimo u aplikaciji, moramo da imamo definisane metode resize() i draw(), ali se ne može dati opšta implementacija ova dva metoda.
- Dakle, bilo bi logično definisati jednu roditeljsku klasu GraphicObject, a zatim sve različite tipove grafičkih objekata izvesti iz nje



Apstraktni metodi

Problem je u tome, što bi klasa `GraphicObject` morala da ima predviđene metode `resize()` i `draw()`, ali nije moguće definisati implementaciju tih metoda (bar ne na jednostavan, a opšti način, koji važi za sve vrste grafičkih objekata).

Rešenje: U objektnim jezicima je moguće metode ostaviti i bez implementacije. Takvi metodi se nazivaju apstraktnim i u Javi se deklarišu na sledeći način

```
abstract void metod_name(arguments);
```

apstraktni metodi nemaju implementaciju, tj. definisano telo

Apstraktne i konkretne klase

- Klase u kojima postoje metodi koji su apstraktni se nazivaju **apstraktnim klasama** i pri njihovom deklarisanju je neophodno, kao i kod metoda, potrebno apstraktnost naglasiti.

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

- **Apstraktna klasa se NE MOŽE INSTANCIRATI**.
- Klase koje se mogu instancirati se nazivaju **konkretnim**.

```
public abstract class absLine {
   int i;
   public void a(){}
   public abstract int b();
}
```

Apstraktna klasa - karakteristike

- objedinjuje zajedničke karakteristike većeg broja klasa
- deklariše se upotrebom ključne reči abstract
- ne može biti instancirana, ali se može proširivati
- ako u nekoj klasi postoji apstraktna metoda tada i sama klasa mora da bude definisana kao apstraktna, obrnuto ne mora da važi, tj.

  klasa se može definisati kao apstraktna, a da pri tome nema ni jedan apstraktan metod
- ukoliko podklasa neke apstraktne klase nema implementirane sve nasleđene apstraktne metode onda i sama podklasa mora biti proglašena apstraktnom
- Sama apstraktna klasa se ne može instancirati, ali:
  - se mogu instancirati konkretne deca-klase
  - mogu postojati **reference tipa apstaktne klase** koje ukazuju na objekte konkretnih klasa koje su u izvedene iz apstraktne

```
GraphicObject g = new GraphicObject();
Circle c = new Circle();
GraphicObject g = c;   ili   GraphicObject g = new Rectangle();
```

Primer

```
abstract class WashingMachine
{
  public WashingMachine()
  {
    // Code to initialize the class goes here.
  }

  abstract public void Wash();
  abstract public void Rinse(int loadSize);
  abstract public long Spin(int speed);
}
```

```
class MyWashingMachine : WashingMachine
{
  public MyWashingMachine()  {
    // Initialization code goes here.
  }
  override public void Wash()  {
    // Wash code goes here.
  }
  override public void Rinse(int loadSize) {
    // Rinse code goes here.
  }
  override public long Spin(int speed)
  {
    // Spin code goes here.
  }
}
```

Interfejsi

C#

Interfejs je apstraktan referencni tip:

- koji sadrži metode koji su implicitno (podrazumevano) `public` i `abstract`,

```
public interface Merljivo{
        int jeJednako(Merljivo u);
        double velicina();
}
```

- Koji ne može da sadrži polja/atribute i konstante
- koji se ne može instancirati
- može biti proširen – tačnije moguće je jedan interfejs izvesti iz drugog

```
public interface MerljivoExt : Merljivo{
        double obim();
}
```

Upotreba

- Klase implementiraju interfejse
  Nakon navodjenja imena neke podklase i eveltualnog definisanja da je izvedena iz neke nadklase, mogu se opciono implemetirati (implements) **jedan ili više interfejsa** (simulacija višestrukog nasleđivanja)

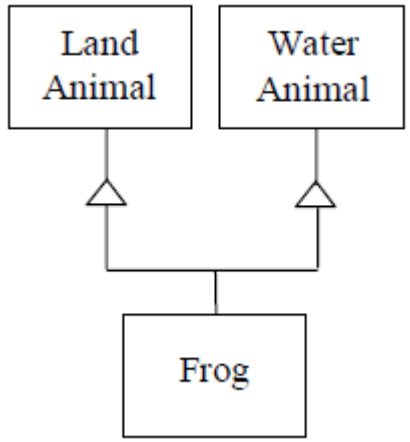
```
class Circle extends GraphicObject : MerljivoExt {
    // implemetacija nasledjenih apstraktnih metoda
    void draw() { ... }
    void resize() { ... }
    // implemetacija metoda interfesa
    public int jeJednako() { ... }
    public double velicina() { ... }
    public double obim() { ... }
}
```

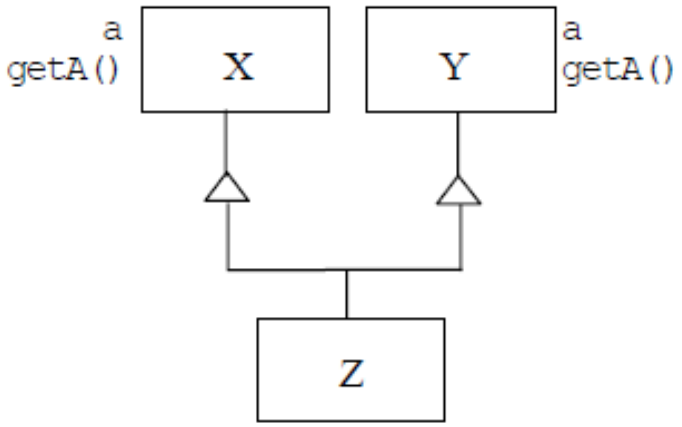
Za razliku od apstraktnih metoda nasledjenih iz apstraktnih klasa, ne mora se navoditi override.

- Kada se za neku **konkretnu** klasu definiše da implemetira neki interfejs tada se u njoj moraju implementirati i sve metode interfejsa. Zašto?

Višestruko nasleđivanje

- Nasleđivanje u OO jezicima može biti:
  - **Jednostruko** – klasa neposredno može da nasledi samo jendu klasu
  - **Višestruko** – klasa može imati dva ili više neposrednih roditeljskih klasa

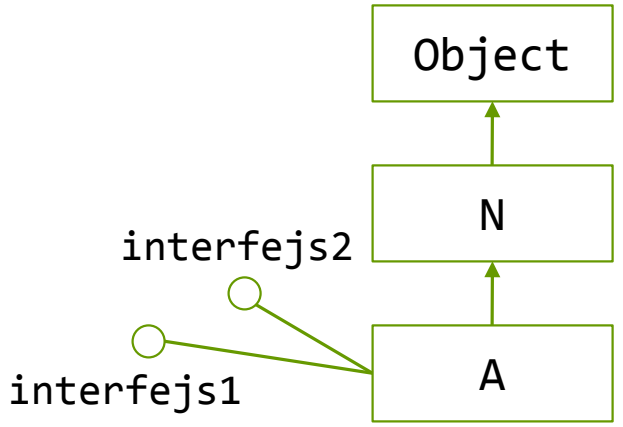




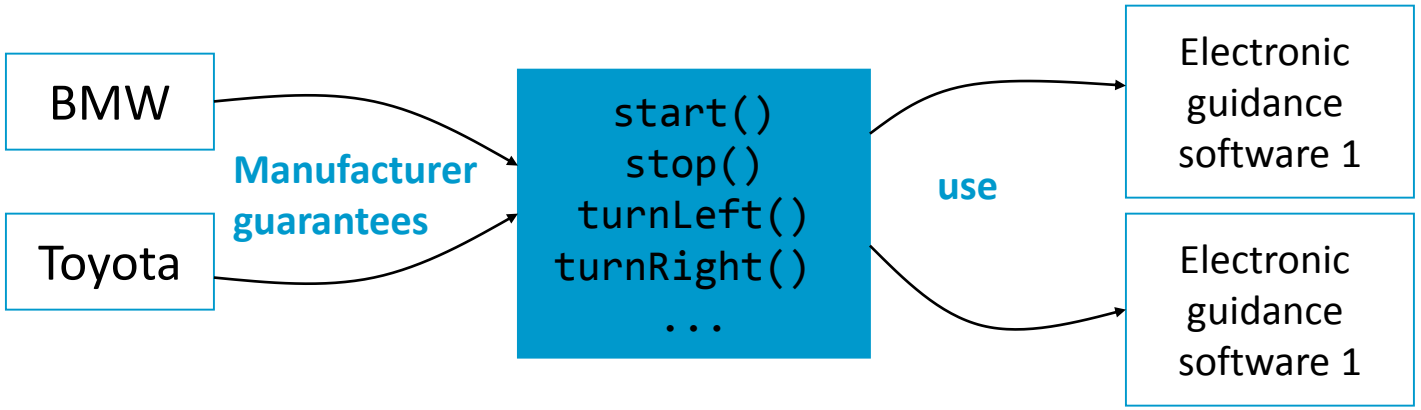
- Problemi višestrukog nasleđivanja potiču od višestrukog nasleđivanja implementacije.

Višestruko nasleđivanje

```
class A extends N : interfejs1, interfejs2 {
    . . .
}

A a = new A();
N n = a;
Object o = a;
interfejs1 i1 = a;
interfejs1 i2 = a;
```



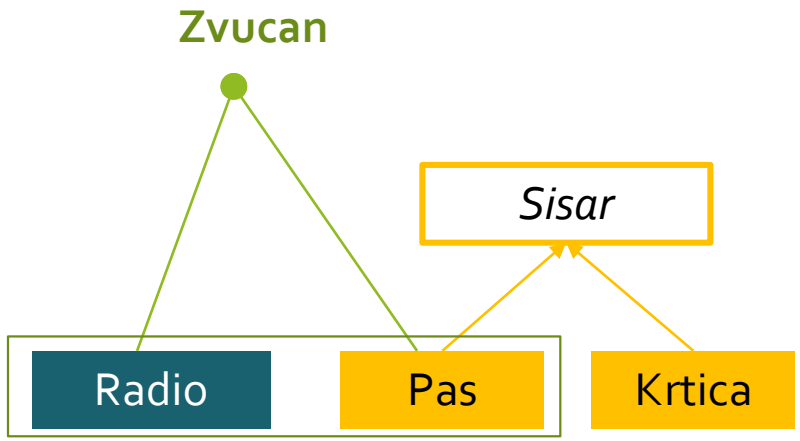
Interfejsi kao ugovori



Zahvaljujući interfejsima, više međusobno nepovezanih klasa može biti referencirano referencom istog tipa.

Primer

```
public interface IZvucan { void javljamSe(); }

abstract public class Sisar {
      public string Vrsta { set; get; }
      public Sisar(string vrsta) { Vrsta = vrsta;}
      public abstract bool plivac();
      override public string ToString() { return Vrsta; }
}
public class Krtica : Sisar{
      public Krtica():base("krtica"){ }
      override public bool plivac() {return false;}
}

public class Pas : Sisar, IZvucan {
      public string Rasa { set; get; }
      public Pas(string rasa) : base("pas") { Rasa = rasa; }
      public void javljamSe() { Console.WriteLine("Av! Av!"); }
      override public bool plivac() { return true; }
}
public class Radio : IZvucan{
      public void javljamSe() { Console.WriteLine("Slusate radio S2"); }
}
```



Primer

```
class Program
    {
        static void Main(string[] args)
        {
            //Sisari
            IZvucan[] pricalice = new IZvucan[]{new Pas("koton"), new Radio()};
            Sisar[] sisari = new Sisar[] { (Pas)pricalice[0], new Krtica() };

            pricalice[0].javljamSe();
            pricalice[1].javljamSe();

            Console.WriteLine(((Object)sisari[0]).ToString());
            Console.WriteLine(sisari[1].ToString());

            Console.ReadKey();
        }
}
```

Kojoj klasi pripada objekat na koji referenca ukazuje?

```
Object[] kontejner = new Object[5];
kontejner[0] = pricalice[0]; kontejner[1] = pricalice[1]; kontejner[2] = sisari[0];
kontejner[3] = sisari[1]; kontejner[4] = "Ja sam string";


foreach (var objekat in kontejner)
{
 if (objekat is IZvucan) ((IZvucan)objekat).javljamSe();
   else
     if (objekat is Sisar) Console.WriteLine("Ja sam plivac : {0}",
                                              ((Sisar)objekat).plivac());
     else Console.WriteLine("Ne umem da se javim, a nisam ni sisar");

 }
```

```
List<IZvucan> pricalice = new List<IZvucan>();
pricalice.Add(new Pas("koton")); pricalice.Add(new Radio());

List<Sisar> sisari = new List<Sisar>();
sisari.Add((Pas)pricalice[0]); sisari.Add(new Krtica());

foreach(IZvucan p in pricalice) p.javljamSe();
Console.WriteLine(((Object)sisari[0]).ToString());
Console.WriteLine(sisari[1].ToString());

List<Object> kontejner = new List<Object>();

foreach(IZvucan p in pricalice) kontejner.Add(p);
foreach (Sisar s in sisari) kontejner.Add(s);
kontejner.Add("Ja sam string");

foreach (var objekat in kontejner) {
                if (objekat is IZvucan) ((IZvucan)objekat).javljamSe();
                else
  if (objekat is Sisar) Console.WriteLine("Ja sam plivac : {0}",
((Sisar)objekat).plivac());
  else Console.WriteLine("Ne umem da se javim, a nisam ni sisar");
            }
```

UML