6

Objektno orijentirano programiranje

- Drugačiji pristup programiranju
- Klasa i objekt
- Kreiranje objekta iz klase
- Nasljeđivanje klasa

Drugačiji pristup programiranju

O važnosti i brzini razvoja informatičke tehnologije te potrebe za informatikom, besmisleno je uopće raspravljati. Da bi informatičari, ponajviše programeri mogli držati korak sa zahtjevnim potrebama tržišta, bilo je potrebno zamijeniti neke starije tradicionalne pristupe programiranju. Osnovna ideja, ovakvog novog programerskog pristupa je: što je više moguće koristiti već napisani kôd u svojim programima. Kao što je poznato većina današnjih aplikacija ima gumbe, izbornike,... svaki puta iz početka pisati sav kod za rad s takvim elementima programskog sučelja, uistinu može biti naporno. Dakle, ideja je da ili koristimo kôd za takve elemente koji je već netko definirao ili smo ga mi sami definirali, a onda mi samo mijenjamo npr. tekst na gumbu, veličinu gumba, akciju koja će se dogoditi klikom na gumb,... Ovakav pristup nam omogućava da nakon što jednom napišemo npr. funkciju koja će provjeravati je li broj prost, više nikada ne moramo tu funkciju pisati, čak ju nećemo morati niti kopirati u svoje nove programe, već ćemo ju "zapakirati" i samo pozivati u svojim programima. Ovakav pristup programiranju je osobito prigodan za ispravljanje grešaka ili bilo kakve izmjene. Npr. ako smo primijetili da naša funkcija koja provjerava je li broj prost ne radi dobro, nećemo morati mijenjati kod svih programa u kojima smo koristili tu funkciju, jer smo ju tamo samo pozivali. Dovoljno će biti samo promijeniti definiciju te funkcije na jednom mjestu i automatski će ona ispravno raditi i u svim ostalim programima. Slično je i s gumbima, ukoliko u jednom trenutku dođu u modu okrugli gumbi a ne više pravokutni kao što su danas, neće biti potrebno mijenjati sve programe koje u sebi imaju pravokutne gumbe već ćemo jednostavno izmijeniti gotovi kod koji smo koristili za kreiranje gumba i automatski će svi gumbi u svim programima, u kojima smo pozivali taj kod biti okrugli.
Ovakav pristup programiranju znatno je olakšao rad velikim programerskim kućama. Naime, većina današnjih programa kreirana je od nezavisnih dijelova koda koji se pozivaju u "glavnom programu". Ukoliko se ustanovi greška u programu, ona se ispravlja u točno određenoj komponenti koda. Ispravljena komponenta koda pošalje se korisniku, koji staru komponentu zamijeni novom i program radi ispravno.
Upravo gore opisani način programiranja osnova je objektno-orijentiranog programiranja.

Klasa i objekt

Motiv za objektno-orijentirano programiranje naveli smo u prošlom naslovu, sama ideja objektno-orijentiranog programiranja potekla je iz našeg, realnog svijeta. Okrenemo se oko sebe vidjet ćemo neke objekte, npr. automobile, avione, zgrade, kuće, motocikle, televizore,... Prilikom programiranja nerijetko ćemo u svojim programima koristiti upravo njih ili neke druge imaginarne objekte, kao što su trokut, krug, jednadžba, broj,...
Kažemo li npr. automobil, avion ili trokut, mislimo na čitavu **klasu** automobila, aviona ili trokuta, tj. mislimo na sve automobile, avione ili trokute. Međutim kažemo li npr. crveni A3, izbor će biti već malo smanjen, znat ćemo da su to samo crveni automobili marke Audi tip A3. Znamo li npr. još i registarsku oznaku znat ćemo točno koji je to automobil, a ako radimo u policiji, znat ćemo sve i o vlasniku automobila, jačinu motora, godinu proizvodnje,... Nakon što znamo sve te detalje, više ne govorimo o čitavoj klasi automobila već o konkretnom automobilu tj. o **objektu** automobil.
Upravo tako je i u programiranju. **Klasa** će biti svojevrstan predložak za objekt, ona sadrži općenita svojstva. Nakon što neku klasu detaljno opišemo ona postaje **objekt** odnosno **instanca**. **Instanca** je samo drugi naziv za **objekt**. Dakle **objekt** odnosno **instanca** su konkretizacija **klase**. Takva konkretizacija klase, tj. definiranje osnovnih svojstava klase nazivamo **kreiranje** (instanciranje**) objekta**.

Primjer 6 – 1:
Neka osnovna svojstva automobila bila bi: marka, tip, snaga, maksimalna brzina, snaga, boja, broj šasije, registracijska oznaka. Ukoliko ne znamo sva navedena svojstva ili bar većinu njih, govorimo o klasi automobil. Međutim ako kažemo to je Mazda, 121, snaga je 38 kw, maksimalna brzina je 180 km/h plave je boje, klasu smo uvelike smanjili. Znamo li još i broj šasije ili registracijsku oznaku znat ćemo točno o kojem se automobilu na ulici radi i tada govorimo o tom automobilu kao objektu.

Osnovna svojstva trokuta bila bi duljine stranica i mjere kutova. Kažemo li trokut mislim na bilo koji trokut koji postoji i u tom slučaju je trokut klasa. Međutim, kažemo li trokut čije su duljine stranica 3, 4 i 5 cm, znamo točno o kojem se trokutu radi i znamo sve o njemu te govorimo o objektu trokut čije su stranice 3, 4 i 5 cm.

Do sada smo rekli da klase općenito imaju neka svojstva. Kod automobila su to bili marka, tip,... Osim svojstava, nad klasama je moguće izvršavati određene radnje. Neke radnje kod automobila bi bile: upali motor, ubaci u prvu brzinu dodaj gas, pritisni spojku,... Slično je i u programiranju, što znači da će klase moći imati neka **svojstva** i neke radnje koje će se nad klasama moći izvršavati tzv. **metode**.
Svojstva će u stvari biti neke globalne varijable, dok će **metode** biti funkcije (metode) definirane unutar neke klase. Općenito ćemo svojstva i metode zvati **elementima klase**.
Općenito je cilj klase da njenim svojstvima i metodama možemo pristupati iz drugih klasa. Npr. definiramo li klasu koja će između ostalih sadržavati metodu koja će provjeravati je li neki broj prost, tada bismo htjeli toj metodi pristupati iz drugih klasa. Općenito, da bismo svojstvu ili metodi neke klase mogli pristupati iz druge klase, one trebaju biti deklarirane **public** (javne). Osim **public**, klasa može imati i elemente (svojstva ili metode) koje će biti samo pomoćne za definiciju **public** metoda i ne trebaju se moći pozivati u drugim klasama. Za takve ćemo metode reći da su **private**. Npr. u klasi u kojoj se nalazi metoda koja provjerava je li neki broj prost, može postojati metoda koja će brojati koliko neki broj ima djelitelja, ta metoda može biti pomoćna za metodu koja provjerava je li neki broj prost, no nju ne moramo moći pozivati iz drugih klasa, stoga ona može biti **private** metoda.

Sljedeći važan pojam kada govorimo o klasama je **konstruktor** klase. **Konstruktor** klase je metoda unutar klase, koja ima isto ime kao klasa a ne vraća nikakav tip. Konstruktor se izvršava prilikom kreiranja objekta. On ne vraća nikakvu vrijednost, a namjena mu je inicijaliziranje svojstava objekta, tj. postavljanje svojstava na neku početnu vrijednost. Jedna klasa može imati jedan ili više konstruktora. Već smo rekli da konstruktor ima isto ima kao klasa. Sama po sebi se nameću pitanje, koja je namjena više konstruktora? Kako je moguće da dvije ili više metoda imaju isto ime? Kako će klasa znati koju metodu treba izvršiti? U stvari se radi o tome da ako klasa ima dva ista konstruktora, onda oni moraju imati različiti broj parametara. Već smo rekli da je namjena konstruktora inicijalizacija svojstava objekta. Mi objekt možemo kreirati bez da znamo vrijednosti svih svojstava, u tom slučaju će se svojstva postaviti na neke inicijalne vrijednosti. U tom slučaju ćemo vrijednosti svojstava definirati kada ih saznamo. Npr. ako imamo klasu *auto*, nije nužno da prilikom kreiranja objekta tipa *auto* znamo sva njegova svojstva. U tom slučaju ćemo svojstva inicijalizirati na neke početne vrijednosti, npr. za maksimalna brzina će biti 100 km/h ako drugačije nije rečeno,...
Ukoliko prilikom kreiranja objekta znamo vrijednosti svih svojstava klase, možemo kreirati objekt s tim svojstvima.
Upravo je to namjena više konstruktora. U prvom slučaju ćemo pozvati konstruktor bez parametara, koji će svojstva inicijalizirati na neku početnu vrijednost, najčešće 0 ako se radi o numeričkim vrijednostima odnosno prazan string ako se radi o tekstualnim vrijednostima, to definira sam programer klase. U drugom slučaju ćemo pozvati konstruktor sa parametrima, koji će vrijednosti svojstava postaviti na vrijednosti koje smo proslijedili preko parametara.
Općenito je slučaj da klasa može imati dvije ili više metode koje se jednako zovu. U tom slučaju one moraju imati različiti broj parametara ili parametri moraju biti različitog tipa, a za takve metode kažemo da su **preopterećene (overloadane)** metode.

Nakon što smo se upoznali s pojmom klase i svim važnijim pojmovima koji dolaze s klasom, vrijeme je da naučimo kako kreirati klasu.
Klasu u Javi ćemo općenito kreirati na sljedeći način:

```
public class ime_klase
{
    svojstva_i_metode;
}
```

Primjer 6 – 2:
Definirajmo klasu **Trokut**, čija će svojstva biti duljine stranica trokuta (**a**, **b** i **c**) a metode će biti **Opseg()** i **Povrsina()** koje će vraćati opseg i površinu trokuta.

Rješenje:
Ovdje ćemo očito imati tri svojstva (duljine stranica **a**, **b** i **c**), koja će biti **public** svojstva jer im trebamo moći pristupiti iz drugih klasa. Nadalje ćemo imati dvije isto tako **public** metode **Opseg ()** i **Povrsina ()** za računanje opsega i površine trokuta. Osim toga imat ćemo jednu private metodu **Poluopseg ()** koja će vraćati poluopseg trokuta, a koja nam je potrebna za računanje površine trokuta Heronovom formulom: $P = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$, pri čemu je *s* poluopseg: ($\frac{a+b+c}{2}$). Ovdje je logično imati najmanje dva konstruktora:
prvi konstruktor bez parametara, koji će vrijednosti parametara **a**, **b** i **c** postaviti na 0. Drugi konstruktor će imati tri parametra i inicijalizirati svojstva **a**, **b** i **c** na vrijednosti parametara. Ovdje

čak ima smisla staviti i konstruktor koji će imati jedan parametar a odnosit će se na jednakostranične trokute i on će vrijednosti parametara **a**, **b** i **c** postaviti na vrijednost proslijeđenog parametra. Konstruktor s dva parametra nema previše smisla, on bi se odnosio na jednakokračne trokute, no on nije spretan jer bi moglo doći do zbrke koji je parametar duljina osnovice jednakokračnog trokuta a koji duljina kraka trokuta.

```
public class Trokut
{
    public int a, b, c;
    public Trokut ()
    {
        a = 0;
        b = 0;
        c = 0;
    }

    public Trokut(int x, int y, int z)
    {
        a = x;
        b = y;
        c = z;
    }

    public Trokut (int x)
    {
        a = x;
        b = x;
        c = x;
    }

    private double Poluopseg ()
    {
        return (double)(a + b + c) / 2;
    }

    public double Povrsina ()
    {
        double s = Poluopseg ();
        return Math.sqrt (s * (s - a) * (s - b) * (s - c));
    }

    public int Opseg ()
    {
        return a + b + c;
    }
}
```

Primijetimo da smo kod metode **Poluopseg ()** pisali `return (double) (a + b + c) / 2`. Da nismo pisali (double) poluopseg ne bi bio točan. Naime, kako su a, b i c cijeli brojevi rezultat bi bio cijeli broj pa bi npr. poluopseg trokuta s duljinama stranica 3, 3 i 3 bio 4 a ne 4.5. Stoga smo najprije zbroj pretvorili u *double* pa smo tek onda dijeliti i u tom slučaju će poluopseg biti *double*.

Napomena:
Da smo zaglavlje drugog konstruktora:
`public Trokut(int x, int y, int z)`
pisali kao
`public Trokut(int a, int b, int c)`

došlo bi do problema jer se svojstva klase i parametri metode zovu jednako. U tim slučajevima bi definicija ove metode trebala imati oblik:

```
public Trokut(int a, int b, int c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
```

this u ovom slučaju znači da je u nastavku navedeno svojstvo ili metoda klase. Dakle, interpretacija ovoga bi bila svojstvu `a` klase u kojoj se nalazim (*Trokut*) pridruži vrijednost parametra `a` i tu ne bi bilo nikakvih problema. **this** možemo pisati općenito ispred elemenata klase, no ako nemamo ovakvog miješanja parametara metoda s elemenata klase, to nije potrebno.

Kreiranje objekta iz klase

Kao što smo već rekli, osnovna namjena klase je koristiti je u drugim klasama, odnosno svojim programima. Nakon što smo naučili kreirati vlastitu klasu, vrijeme je da pokažemo kako tu klasu pozivati u drugim klasama, te kako pozivati definirati njena svojstva i pozivati njene metode.
Na klasu možemo gledati kao na jedan novi tip podataka, koji će imati neke vrijednosti i nad kojim ćemo moći izvršavati neke operacije (pozivati metode).
Kao i svaki drugi tip podataka, klasu trebamo deklarirati. Opći oblik deklaracije objekta čiji će tip biti neka klasa je:

```
ime_klase ime_objekta;
```

Na ovaj način smo u stvari samo najavili da ćemo koristiti objekt *ime_objekta*, koji će biti tipa *ime_klase*. Taj objekt još zapravo ne postoji. S takvim objektom ne možemo još ništa raditi. Da bismo mogli nešto raditi nad tim objektom (npr. izvršavati metode koje smo definirali u pripadnoj klasi) moramo ga **instancirati**, tj. zaista ga stvoriti. Objekt ćemo instancirati na sljedeći način:

```
ime_objekta = new ime_klase ([parametri]);
```

Prisjetimo se, kada smo govorili o klasama, rekli smo da klasa može imati jedan ili više konstruktora i to smo nazivali **preopterećenje** konstruktora, preopterećeni konstruktori su se razlikovali po broju parametara, odnosnu tipu parametara. Prilikom instanciranja klase, mi u stvari izvršavamo određeni konstruktor i to onaj čiji broj i tip parametara odgovaraju broju i tipu parametara koje smo naveli prilikom instanciranja klase.

Često ćemo posljednje dvije naredbe ujediniti te ćemo istovremeno deklarirati i kreirati objekt. To ćemo napraviti sljedećom naredbom:

```
ime_klase ime_objekta = new ime_klase ([parametri]);
```

Primjer 6 – 3:
Kolike će biti duljine stranica trokuta nakon sljedećih naredbi:

a) `Trokut t = new Trokut ();`
b) `Trokut t = new Trokut (3, 4, 5);`
c) `Trokut t = new Trokut (2);`

ako je *Trokut* klasa koju smo kreirali u *primjeru 4 – 2*.

Rješenje:

Očito smo u svakom od primjera kreiranjem objekta *t* koji je tipa *Trokut* pozivali drugi konstruktor klase *Trokut*. Stoga će duljine stranica biti:

a) **a = 0, b = 0, c = 0**; u ovom slučaju se izvršava prvi konstruktor (bez parametara), koji duljine stranica postavlja na 0.

b) **a = 3, b = 4, c = 5**; izvršava se drugi konstruktor (s tri parametra), prvi proslijeđeni parametar bit će duljina stranice *a*, drugi duljina stranice *b*, dok će treći proslijeđeni parametar biti duljina stranice *c*.
c) **a = 2, b = 2, c = 2**; izvršava se treći konstruktor, koji će duljine svih stranica postaviti na jedinu proslijeđenu vrijednost, koja je u ovom slučaju 2.

Pristup elementima klase

Upravo smo naučili kako kreirati objekt iz neke klase. Što ćemo sada s tim objektom?
Već prije smo prije rekli da svaka klasa ima svoja svojstva i metode tj. neke radnje koje se nad njom mogu izvršavati. Pa jedino što ćemo mi raditi s objektima je mijenjati im svojstva i izvršavati radnje nad njima (pozivati metode).
To znači da npr. kada smo jednom definirali klasu *Trokut*, čija su svojstva duljine stranica trokuta a metode računaju opseg i površinu, tada više nikada nećemo morati u svojim programima pisati formulu za računanje površine trokuta. Jednostavno ćemo kreirati objekt iz klase *Trokut*, postaviti mu duljine stranica i jednostavno pozivati metodu *Povrsina ()*, koja nam vraća površinu tog trokuta. Dakle, slobodno možemo zaboraviti na Heronovu formulu Ukoliko ikada itko zaključi da Heronova formula za računanje površine trokuta nije točna i pojavi se neka nova formula za računanje površine trokuta (što nije vjerojatno), jednostavno ćemo izmijeniti svoju metodu *Povrsina ()* u klasi *Trokut* i svi programi će automatski računati površinu trokuta prema novoj formuli.

Sada kada smo uvidjeli smisao objekta, postavlja se pitanje kako pristupiti elementima objekta (svojstvima i metodama)? pa pristupat ćemo im prema sljedećem principu:

```
ime_objekta.element;
```

Primjer 6 – 4:
Kreirajmo objekt *t* koji će biti instanca klase *Trokut* koju smo kreirali u *primjeru 4 – 2*, pri čemu će duljine stranica biti postavljene na 3, 4 i 5, a zatim u varijablu *p* spremimo površinu tog trokuta.

Rješenje:
Najprije kreirajmo objekt *t*, koji će biti instanca klase *Trokut* i čije će duljine stranica biti 3, 4 i 5.
Prvi način na koji možemo ovo riješiti je da duljine stranica proslijedimo konstruktoru odmah prilikom kreiranja objekta:

```
Trokut t = new Trokut (3, 4, 5);
```

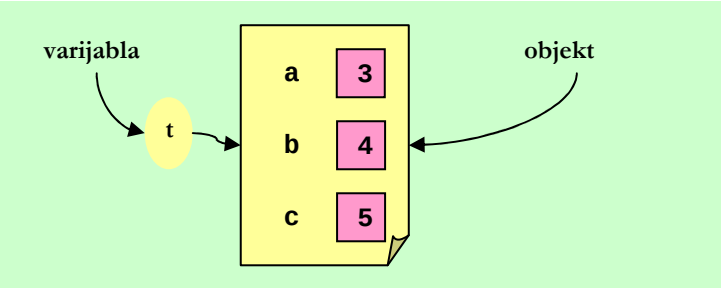
Isto ovo mogli smo napraviti tako da kreiramo objekt *t* iz klase *Trokut*, ali ovaj puta bez parametara. U tom će slučaju vrijednosti svojstava **a**, **b** i **c** biti postavljene na 0, a zatim postavimo vrijednosti svojstava na vrijednosti 3, 4 i 5, što bi izgledalo ovako:

```
Trokut t = new Trokut ();
t.a = 3;
t.b = 4;
t.c = 5;
```

U prvom će se slučaju prilikom kreiranja objekta *t* izvršavati drugi konstruktor, dok će se u drugom slučaju izvršavati prvi konstruktor.
Nakon što smo kreirali objekt tipa *Trokut*, trebamo još u varijablu *p* spremiti površinu tog trokuta. To ćemo napraviti pozivanjem metode *Povrsina ()* nad objektom *t*, na sljedeći način:

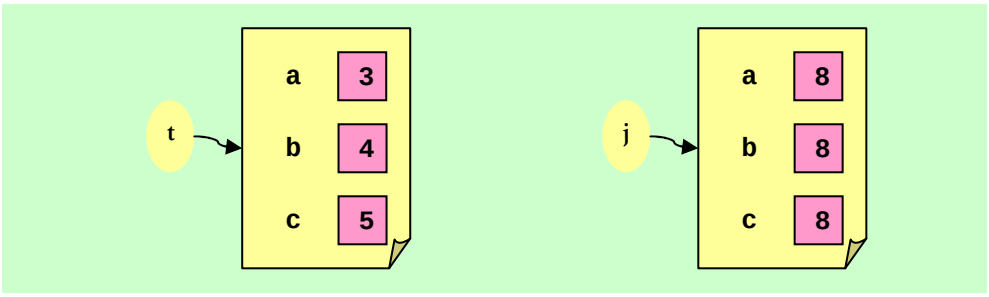
```
double p = t.Povrsina ();
```

Napišemo li npr.: `Trokut t = new Trokut (3, 4, 5);`
U memoriji će se alocirati prostor za objekt *Trokut* te će se vrijednosti svojstava (stranice **a**, **b** i **c** postaviti na vrijednosti 3, 4 i 5) a u varijablu *t* će biti pohranjena adresa objekta u memoriji.

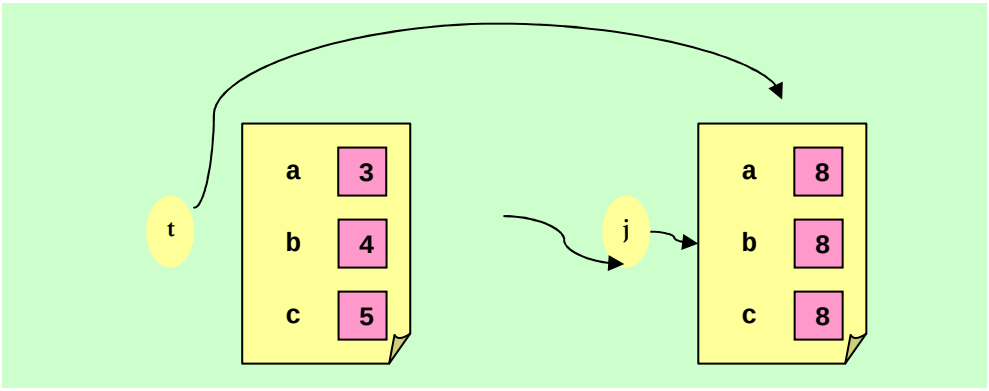


Općenito nad objektima nije moguće izvršavati standardne operacije npr. +, -, ==,... Razmislimo li malo, to uistinu ima smisla. Što bi značilo `t + j` ako su *t* i *j* dvije instance klase *Trokut*? Što bi značilo zbrojiti dva trokuta?

Operator pridruživanja (=) nad objektima je moguće pozivati, samo on ne radi uvijek ono što bismo možda očekivali. Pretpostavimo da smo kreirali dvije instance klase *Trokut*, *t* i *j*. Kao na sljedećoj slici:



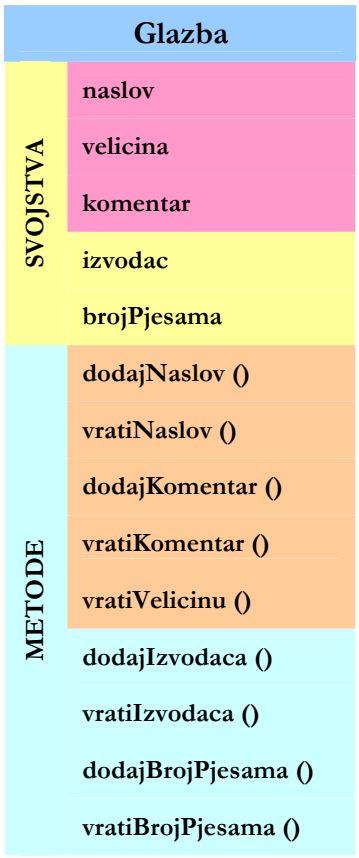
Izvršavanjem izraza: `t = j`, dogodit će se sljedeće:

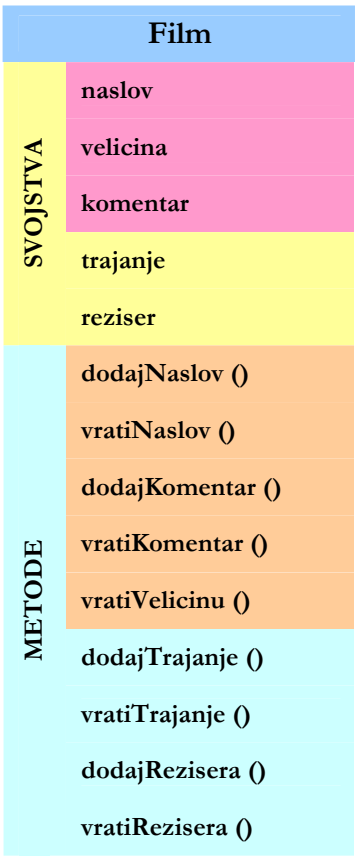


To znači da smo samo promijenili referencu i sada *t* i *j* pokazuju na isti objekt. Izvršavanjem izraza: `t.a = 4`
svojstvo `a` (duljina stranice *a*) objekta *j* će također imati vrijednost 4.

Nasljeđivanje klasa

Nasljeđivanje je vrlo čest pojam u objektno orijentiranom programiranju. Kao i kod samog objektno orijentiranog programiranja, osnovna ideja nasljeđivanja je izbjeći višestruko pisanje istih dijelova koda. Ideja je vrlo jednostavna: imamo jednu osnovnu klasu s osnovnim svojstvima i metodama koje ju opisuju, a onda po potrebi kreiramo nove klase koje ju "nadopunjuju" s nekim dodatnim svojstvima i metodama. Za takve klase koje "nadopunjuju" osnovnu klasu reći ćemo da ju **nasljeđuju**.
Ilustrirajmo nasljeđivanje na jednom jednostavnom primjeru:
pretpostavimo da želimo organizirati svoju kolekciju DVD-a i želimo napraviti program koji će nam to omogućiti. Očito ćemo imati dvije vrste DVD-a: s filmovima i s glazbom. Dakle kreirat ćemo dvije klase s pripadnim svojstvima i metodama:

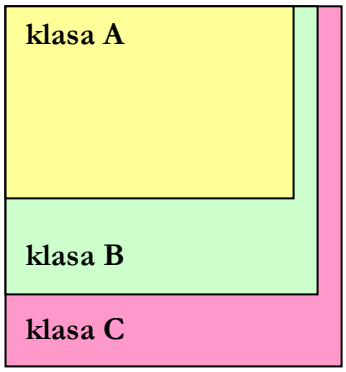




Kao što možemo primijetiti, neka svojstva i metode se nalaze u obje klase i stoga i moramo ih dva puta definirati.
Problem možemo riješiti tako da elemente koji se pojavljuju u obje klase definiramo u jednoj posebnoj klasi *DVD*:

Sad kada imamo definiranu klasu *DVD*, možemo kreirati klase *Film* i *Glazba* koje nasljeđuju klasu *DVD*. Nasljeđivanjem klase *DVD* klase *Film* i *Glazba* automatski nasljeđuju i sva svojstva i metode klase *DVD*. Dakle sva ta svojstva i metode ne trebamo još jednom definirati.
Općenito pretpostavimo da imamo neku klasu **A**, koja ima svoja svojstva i metode. Klasu **A** može **naslijediti** klasa **B**, koja će u tom slučaju imati sva svojstva i metode kao i klasa **A**, ali može imati i još neka svoja dodatna svojstva i metode. U tom slučaju je **A nadklasa** klase **B**, dok ćemo za **B** reći da je **podklasa** od **A**.

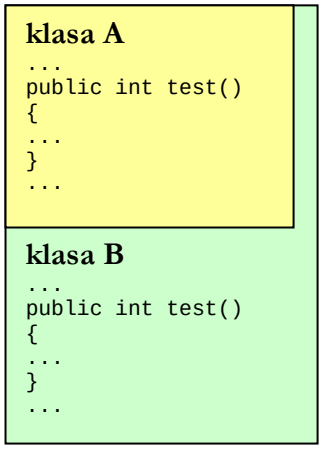
Kreiranjem instance klase **B** moći ćemo pozivati sve koje smo definirali u klasi **B**, ali i one definirane u klasi **A**, dok ćemo kreiranjem instance klase **A** moći pozivati samo elemente klase **A**. Podklasa tako također može biti nadklasa nekoj drugoj klasi i na taj način je moguće izgraditi **hijerarhiju** klasa.



Klasu **B** koja će naslijediti klasu **A** kreirat ćemo na sljedeći način:

```
class B extends A
{
    definicija_klase;
{
```

Kod ovakve hijerarhije može se dogoditi da je metoda s istim imenom kreirana u dvije klase koje su na različitoj razini u hijerarhiji. Npr. pretpostavimo da je u klasi **A** definirana metoda **test ()** i metoda s istim imenom je definirana i u klasi **B**, koja nasljeđuje klasu **A**.

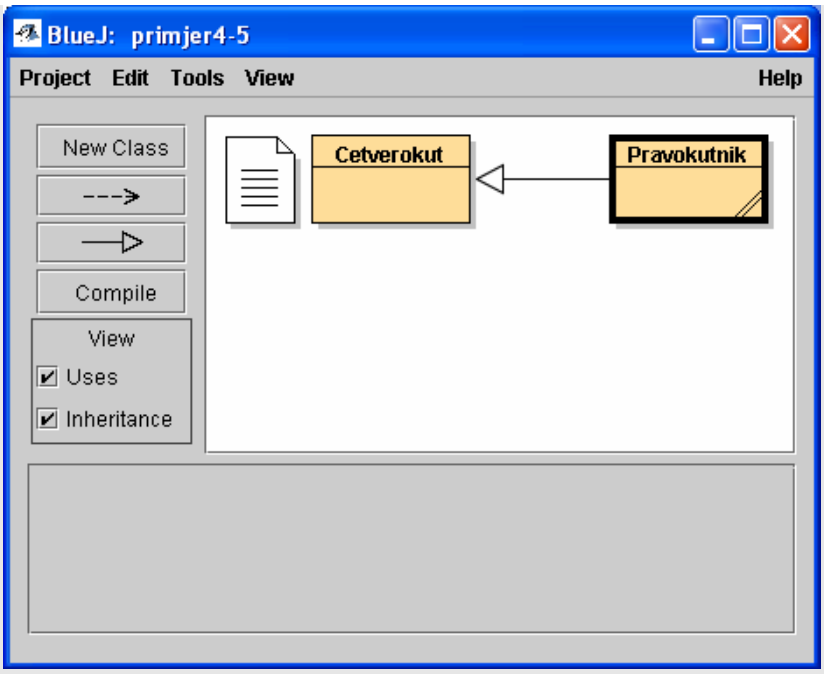


Kako klasa B nasljeđuje klasu A ona nasljeđuje i njenu metodu *test ()*, a isto tako ona ima i svoju metodu *test ()*. Kreiramo instancu klase **B** te pozovemo metodu *test ()*, nije sasvim jasno koja metoda *test ()* će se izvršiti.
U ovakvim slučajevima izvršit će se metoda *test ()* koja je definirana u klasi B i to se zove **prekoračenje** (override) metoda.

Općenito smo mogli imati klasu **C**, koja nasljeđuje klasu **B** (klasa C nema metodu *test ()*). Pozivanjem metode *test ()* nad instancom klase **C** izvršit će se metoda *test ()* iz klase **B**. Dakle, pravilo je takvo da se izvršava metoda iz prve nadklase.

Primjer 6 – 5:
Kreirajmo klasu **Cetverokut**, čija će svojstva biti duljine stranica (a, b, c, d) četverokuta, dok će jedina metoda biti **Opseg ()**, koja će vraćati opseg četverokuta. Nadalje kreirajmo klasu **Pravokutnik** koja nasljeđuje klasu **Cetverokut** te ima još metode **Povrsina ()**, koja računa površinu pravokutnika i **Dijagonala ()**, koja vraća duljinu dijagonale pravokutnika.

Rješenje:
```
public class Cetverokut
{
    public int a, b, c, d;
    public Cetverokut()
    {
        a = b = c = d = 0;
    }

    public Cetverokut (int a, int b, int c, int d)
    {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
    }

    public int Opseg ()
    {
        return a + b + c + d;
    }
}

public class Pravokutnik extends Cetverokut
{
    public Pravokutnik ()
    {
        a = b = c = d = 0;
    }

    public Pravokutnik (int a, int b)
    {
        this.a = a;
        this.b = b;
        this.c = a;
        this.d = b;
    }

    public int Povrsina ()
    {
        return a * b;
    }

    public double Dijagonala ()
    {
        return Math.sqrt (a * a + b * b);
    }
}
```



Napomena:
Svaka klasa može nasljeđivati samo jednu klasu, tj. klasa može imati samo jednu nadklasu. S druge strane neku klasu može nasljeđivati bilo koja klasa, tj. klasa može biti podklasa.