# Klase i objekti

## Objekti

Sposobnost prepoznavanja fizičkih objekata je osobina koju ljudi razvijaju u vrlo ranoj fazi života. Još je Dekart primetio da ljudi svet posmatraju u objektno orijentisanom smislu. Ljudski mozak želi da razmišlja o objektima. Naše misli i memorija su organizovani u vidu objekata i njihovih veza.

Jedna od ideja koja leži u osnovi objektno orijentisanog softvera je da se softver organizuje na način koji odgovara načinu razmišljanja objektno orijentisanog mozga. Umesto mašinskih instrukcija koje menjaju bitove u glavnoj memoriji, mi želimo "stvari" koje nešto "rade". Naravno da se na mašinskom nivou ništa nije promenili, obrasci bitova se i dalje menjaju preko mašinskih instrukcija. Promena je u tome što mi ne moramo da razmišljamo na taj način.

Pitanje: Navedite nekoliko objekata.

Olovka, tastatura, cipele, sto

Pitanje: Šta je to što karakteriše neki objekat.

- Objekat je napravljen od opipljivog materijala
- Objekat se ponaša kao celina, sastavljena od delova
- Objekat ima osobine
- Objekat može da čini neke stvari i na njemu mogu da se čine neke radnje.

Prva stavka iz ove liste je suviše restriktivna. Mogu postojati i objekti koji nisu opipljivi. Prisetite se svog računa u banci. Premda to nije materijalna stvar, račun ima svoje osobine (stanje, vlasnika) i vi sa njim možete da činite neke stvari (da podižete novac, ugasite ga). U skladu sa ovim može se reći da jedan objekat:

- ima identitet (ponaša se kao celina)
- ima stanje (ima osobine koje se mogu menjati)
- ima ponašanje (može da čini neke stvari i nad njim se mogu vršiti neke radnje)

Grady Booch, jedan od pionira objektno orijentisanog programiranja koristi sledeću definiciju objekta:

*Objekat ima stanje, ponašanje i identitet. Struktura i ponašanje sličnih objekata su definisani njihovom zajedničkom klasom.*

### *Stanje*

Pogledajmo automat za prodaju sokova. Uobičajeno ponašanje ovakvih objekata je da kada neko ubaci novčić i pritisne dugme, dobije sok. Šta će se desiti ako korisnik prvo izabere piće, pa tek onda ubaci novčić? Većina mašina neće uraditi ništa, pošto je korisnik narušio osnovnu pretpostavku njenog rada. Slično ovom, šta ako korisnik ubaci više novaca nego što je potrebno. Većina mašina će sa zahvalnošću progutati dodatni novac.

U svakom od prethodnih slučajeva smo videli da na ponašanje objekta utiče njegova istorija, odnosno videli smo da je bitan redosled po kojem objekat radi. Razlog za ovakvo ponašanje, koje zavisi od vremena i događaja je da se objekat nalazi u nekom stanju. Esencijalno stanje vezano za naš automat za sokove je količina novca koju je korisnik ubacio, ali da još uvek nije pritisnuo dugme. Na osnovu ovog se može izvesti sledeća definicija:

*Stanje nekog objekta obuhvata sve statičke (nepromenljive) osobine objekta, zajedno sa trenutnim vrednostima dinamičkih osobina.*

Na primeru našeg automata, statička osobina je da ona može da prihvati ubačen novac. Dinamička osobina bi bila trenutno ubačeni novac.

Činjenica da svaki objekat ima stanje implicira da svaki objekat zauzima neki prostor, bilo da je reč o stvarnom svetu, ili o memoriji računara.

### *Ponašanje*

Nijedan objekat ne postoji samostalno. Objekti komuniciraju sa drugim objektima. U skladu sa tim možemo reći da:

*Ponašanje pokazuje kako se objekti ponašaju i reaguju, u smislu promene njihovog stanja i razmene poruka.*

Drugim rečima ponašanje objekta predstavlja njegove vidljive aktivnosti.

Operacija predstavlja neku akciju koju jedan objekat obavlja da bi od drugog izmamio reakciju. Klijent na primer, može pozvati operaciju length, koja vraća vrednost koja označava dužinu nekog objekta. Kod čistih objektno orijentisanih jezika se kaže da je jedan objekat uputio drugom objektu poruku. Neki jezici će to nazvati metodom, neki funkcijom, neki operacijom. U svakom slučaju misli se na istu stvar.

### *Identitet*

Copeland i Khoshafian daju sledeću definiciju:

*Identitet je osobina objekta koja ga razlikuje od svih drugih objekata.*

U programiranju posebnu pažnju treba obratiti na razliku između imena objekta i samog objekta. Mogu postojati (često i postoje) više imena za isti objekat, ali to ne znači da su u pitanju različiti objekti.

### Softverski objekti

Mnogi programi se pišu da rade stvari koje se tiču realnog sveta. U tom smislu je pogodno imati "sotfverske objekte" koji su slični sa objektima iz realnog sveta. Na taj način je lakše razumeti program. Softverski objekti imaju stanje, ponašanje i identitet, isto kao realni objekti. Softverski objekti postoje samo u okviru računarskog sistema i nemaju nikakvog uticaja na stvarne objekte.

Verovatno je preterano reći da je memorija sve što postoji unutar računara. Sa druge strane, sva ostala elektronika, procesor, magistrale, tastatura, video kartica itd., postoje samo da bi radili sa memorijom i prikazali ono što se u njoj nalazi. Šta bi onda softverski objekat bio nego parče memorije?

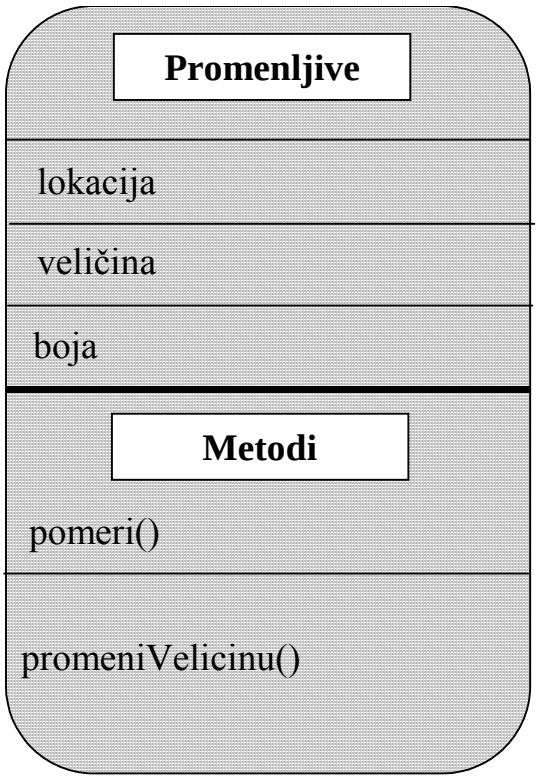
I realni i softverski objekti imaju identitet, stanje i ponašanje.

Softverski objekti imaju identitet pošto zauzimaju poseban deo memorije. Softverski objekat je poseban, čak i ako izgleda isto kao neki drugi objekat.

Softverski objekti imaju stanje. Deo memorije koju zauzima softverski objekat se koristi za promenljive koje sadrže vrednosti.

Softverski objekti imaju ponašanje. Deo memorije koju oni zauzimaju se troši za skladištenje metoda (programa) koji objektu omogućavaju da "nešto radi". Objekat nešto radi, kada se izvršava neki od njegovih metoda.

Na sledećoj slici je prikazana slika jednog objekta u memoriji.




Softverski objekat se sastoji od promenljivih i metoda. Cigle na slici predstavljaju bajtove u memoriji, koju objekat zauzima. Ovaj objekat ima neke promenljive, pozicija, boja i veličina, kao i neke metode koji kontrolišu njegovo ponašanje.

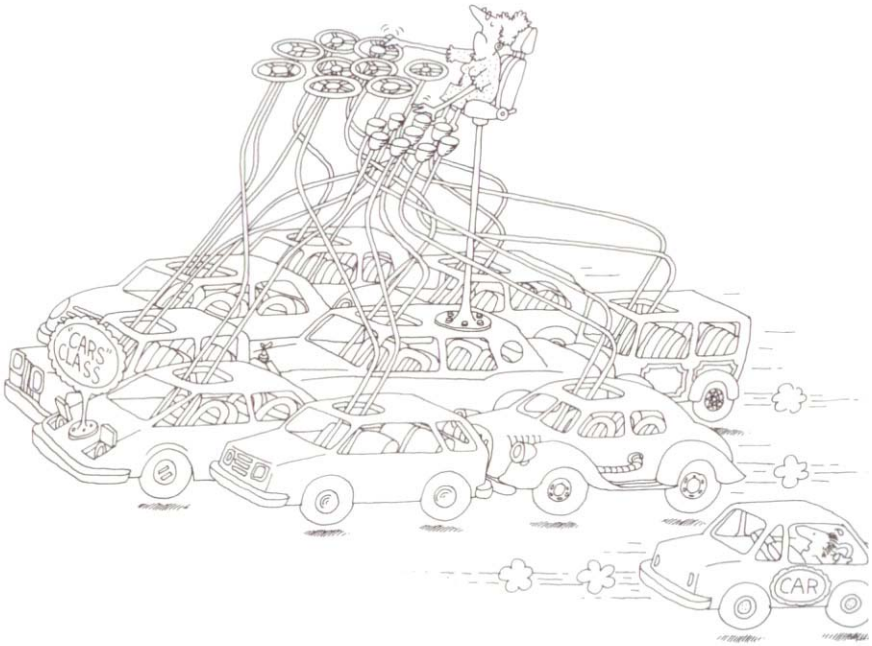
U objektno orijentisanom programiranju, programer, za opisivanje objekata, koristi programski jezik (kao što je Java). Kada se program pokrene, kreiraju se objekti, koji počinju da rade "stvari" preko svojih metoda.

## Klase

Koncept objekta i klase se međusobno stalno prepliću, tako da ne možemo govoriti o nekom objektu, a da ne pomenemo njegovu klasu. Između ova dva termina ipak, postoje bitne razlike. Dok objekat predstavlja konkretan entitet, koji postoji u vremenu i prostoru, klasa je samo apstrakcija, suština objekta. Grady Booch daje sledeću definiciju klase:

*Klasa je skup objekata koji imaju zajedničku strukturu i ponašanje.*

Jedan objekat predstavlja primerak neke klase.



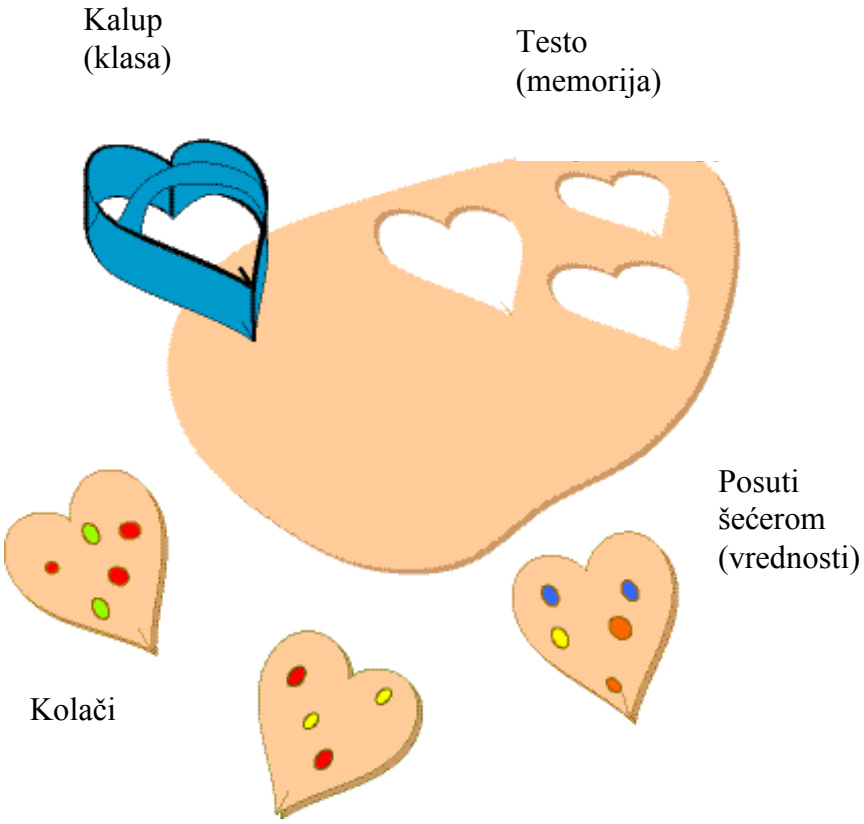

Pogledajte primerak neke knjige. Taj primerak ima svoje osobine. Može biti sasvim nov, ili pohaban, možda ste na prvoj stranici napisali svoje ime, možda pripada biblioteci, pa ima svoju jedinstvenu šifru.

Sa druge strane, osnovni elementi knjige, kao što su naslov, izdavač, autor i sadržaj su određeni opisom koji je primenljiv na svaki zaseban primerak. Naslov knjige je taj i taj, izdavač je taj i taj itd. Taj skup osobina ne definiše objekat, već klasu objekata (ponekad se koristi i termin tip objekata).

Klasa je ta koja definiše kalup. Objekti sagrađeni pomoću tog kalupa su instance ili primerci te klase.

Da napravimo analogiju klase i kalupa za pravljenje kolača. Postoji samo jedan kalup za kolače, koji se može upotebiti za pravljenje više kolača. Kolači su objekti i svaki od njih ima svoju individualnost, pošto je napravljen od različitog dela testa. Različiti kolači imaju različite karakteirstike, iako su napravljeni na osnovu istog kalupa (mogu se dalje ukrašavati različitim ukrasima, ili se mogu peći različito vreme.

Kolači mogu da se naprave, a mogu i da se unište (da ih pojedete). To što ste možda pojeli kolače ne utiče na kalup za pravljenje kolača. On se može upotrebiti ponovo, sve dok ima dovoljno testa.




Klasa je softverski tekst i ona je statička, što znači da postoji nezavisno od bilo kakvog izvršavanja. Objekat napravljen na osnovu te klase je dinamička struktura podataka, koja postoji samo u memoriji računara, tokom izvršenja programa.

### *Klasa kao modul i tip*

Da biste shvatili objektno orijentisani pristup vrlo je bitno shvatiti da klase istovremeno igraju dve uloge. Ove dve uloge su pre pojave objektno orijentisanog programiranja uvek posmatrane odvojeno. Klasa je istovremeno i modul i tip.

Programski jezici (savremeni) uvek uključuju mogućnost građenja modula i neki sistem tipova.

Modul je jedinica dekompozicije softvera. Postoje razni oblici modula, kao što su na primer, procedure i funkcije, potprogrami i sl. Proces dekomponovanja softvera na module je vođen principima logike i rukovođenja i održavanja projekta, a ne nekom suštinskom neophodnošću.

Tipovi su na prvi pogled različit pojam. Tip je statički opis dinamičkih objekata, odnosno raznih podataka koji će se koristiti tokom izvršenja programa. Svaki jezik ima unapred definisane tipove, ali postoje i tipovi koje definišu programeri.

Pojam modula je stvar sintakse jezika, dok je tip stvar semantike, jer direktno utičena izvršavanje softverskog sistema, putem definisanja oblika objekata koje će sistem stvoriti i kojima će rukovati.

U pristupu koji nije objektno orijentisan su koncepti tipa i modula jasno odvojeni. Najvažnije svojstvo klase je da ona obuhvata oba koncepta i spaja ih u celinu. Klasa je modul, ili jedinica dekompozicije softvera. Klasa je, međutim, i tip.

Stanje neke klase se opisuje preko njenih atributa. Ponašanje klase se opisuje njenim metodima.

### *Klase u Javi*

Klasa se u Javi definiše preko ključne reči class, iza koje sledi ime klase. Ime klase podleže svim pravilima koja se odnose na imena promenljivih u Javi.

class Primer{……}

Prethodni iskaz definiše klasu Primer. Sve što pripada klasi mora biti definisano između otvorene i zatvorene velike zagrade.

Klasa je samo plan za moguće objekte. Ona sama po sebi ne kreira objekte. Kada programer želi da u Javi kreira novi objekat, koristiće operator new. Taj proces se naziva instanciranjem objekta.

U sledećem primeru je pokazano kako se kreira i koristi klasa String. U pitanju je klasa koja je unapred pripremljena u Javi i koja se koristi za manipulaciju tekstualnim objektima.

```
class StringTester{
  public static void main ( String[] args ){
    String str1;   // str1 je promenljiva koja ukazuje na objekat
                  // ali objekat još uvek ne postoji
    int    len;    // len je promenljiva tipa int
    str1 = new String("Probni tekst"); //kreira se objekat tipa String

    len  = str1.length();  // poziva se metod objekta length()
    System.out.println("Dužina stringa je " + len);
  }
}
```

Kada se ovaj program izvrši u redu

```
str1 = new String("Probni tekst");
```

se kreira novi objekat. To se radi na osnovu opisa u klasi String. Klasa String je definisana u biblioteci java.lang, koja dolazi sa instalacijom Jave. Računar će pronaći deo memorije za ovaj objekat, napraviti raspored prema planu u klasi String i u tu memoriju postaviti metode i podatke.

Podatak u ovoj klasi će biti tekst "Probni tekst", dok će metodi biti oni koji su definisani u ovoj klasi. Jedan od tih metoda je length(). Promenljiva str1 se koristi za pristup (referenciranje) tom objektu. Drugim rečima, str1 je ime tog objekta.

To što promenljiva može da ukazuje na neki objekat, ne znači da ona uvek ukazuje na objekat. Ako pogledate prethodni primer, promenljiva str1 ukazuje na objekat tek posle primene operatora new, koji kreira novi objekat. Pre poziva operatora new, programu je samo naznačeno da će promenljiva str1 biti upotrebljena za ukazivanje na objekat tipa string.

### *Sintaksa pristupanja članovima klase*

Nakon što ste dobili referencu na objekat, možete pristupati atributima i metodima tog objekta. Red str1.length(), u prethodnom primeru označava poziv metoda objekta klase String.

U objektno orijentisanom programiranju se za označavanje stanja i ponašanja objekata koriste različiti termini. Za promenljive, koje označavaju stanje objekta, se najčešće koristi termin atributi. Funkcije koje ukazuju na ponašanje objekta se nazivaju metodima. I jedni i drugi zajedno predstavljaju članove klase. Članovima klase se u Javi pristupa preko operatora . (tačka).

```
referencaNaObjekat.članObjekta
```

Ovo ste videli u prethodnom primeru, kada ste želeli da pristupite metodu lentgh(), objekta str1. Sintaktički se pristupanje metodu razlikuje od pristupanja atributu klase, po tome što iza imena metoda stoje male zagrade. Između tih zagrada se mogu naći dodatni argumenti, ali i ne moraju.

### *Statički članovi klase*

Klase se uglavnom koriste za kreiranje objekata, kao što se kalup za kolače koristio za pravljenje kolača. Ipak i klasa ima svoju materijalnost, kao što pored kolača postoji i kalup za kolače. Klasa i njeni objekti su različite stvari, kao što su kalup i sami kolači različiti.

Kalup za kolače ima različite stvari, koje nisu zajedničke sa kolačima koji su pomoću njega napravljeni. Kalup je napravljen od metala i ima oštre ivice. Nakon što se kalup upotrebi za kolače, biće puno kolača, ali će i dalje postojati samo jedan predmet od čelika, sa oštrim ivicama (kalup).

Karakteristike definicije klase, koje su zajedničke za sve objekte, odnosno koje pripadaju samoj klasi, se u Javi nazivaju statičkim. Ove karakteristike se označavaju preko ključne reči static. Pošto one pripadaju klasi, a u programu postoji samo jedna klasa (kao tip) to znači da u trenutku izvršavanja programa, ako je nešto deklarisano kao static, postoji samo jedan primerak.

To se može izraziti i na sledeći način: bez obzira na to koliko se objekata istog tipa napravi, ako je nešto definisano kao static, postojaće samo jedan primerak.

Svi članovi klase mogu biti statički. Ovde se pod članovima misli i na atribute klase i na njene metode. Statičke atribute smo već objasnili. To su promenljive, koje za sve objekte istog tipa, koji se naprave u programu imaju istu vrednost. U sledećem primeru je pokazano kako to izgleda u praksi:

```
class Primer{
   static int brojac = 0;
}
```

Pored atributa i metodi klase mogu biti deklarisani kao statički. Vrlo je bitno da kod statičkih metoda, na umu imate sledeće:

Program može da izvrši statički metod klase, a da pre toga nije napravljen nijedan objekat. Svi drugi metodi (koji nisu statički) predstavljaju deo objekta, pa se prema tome, ne mogu izvršiti preko nego što se napravi objekat.

Pogledajmo ponovo primer od ranije:

```
class StringTester{
  public static void main ( String[] args ){
    String str1;   // str1 je promenljiva koja ukazuje na objekat
                   // ali objekat još uvek ne postoji
    int    len;    // len je promenljiva tipa int
    str1 = new String("Probni tekst"); //kreira se objekat tipa String

    len  = str1.length();  // poziva se metod objekta length()
    System.out.println("Dužina stringa je " + len);
  }
}
```

Sada o ovom primeru možete da kažete mnogo više detalja.

U ovoj datoteci je definisana klasa StringTester.

- Klasa ima statički metod po imenu main().

- Pošto je main statički metod, on je karakteristika klase. Objekti klase nemaju svoj, poseban metod main.
    - Pošto je main statički metod, postojaće samo jedan metod main.
- Posle kompajliranja ovde datoteke, kod koji se dobija će izvršiti virtuelna mašina.
- Virtuelna mašina će potražiti bajtkod verziju metoda main.
    - U tom trenutku ne postoji nijedan objekat.
- Metod počinje da se izvršava, preko operatora new se kreira objekat, poziva se metod length() itd.

Prisetite se osnovne ideje objektno orijentisanog programiranja: aplikacija se sastoji od niza softverskih objekata, čiji se metodi izvršavaju određenim redosledom. Koraci koje smo opisali pokazuju kako se pokreće taj niz objekata (u ovom slučaju je napravljen samo jedan objekat).

### *Konstruktori*

Objekat je deo memorije koji sadrži promenljive i metode. Klasa je opis objekta. Prilikom kreiranja objekta se koristi njegov opis. Objekti se kreiraju preko operatora new. Kada se primeni operator new, poziva se specijalan metod klase, tzv. konstruktor.

```
class StringTester{
  public static void main ( String[] args )  {
    String str1;   // str1 je referenca na objekat
    int    len;
    str1 = new String("Probni tekst"); // kreira se objekat tipa String
    len  = str1.length();  // poziva se metod length()
    System.out.println("Duzina stringa je " + len );
  }
}
```

Konstrktor ima isto ime kao klasa. U redu

```
    str1 = new String("Probni tekst"); // kreira se objekat tipa String
```

se kreira novi objekat tipa String. Tom prilikom se poziva konstruktor klase String, koji je unapred definisan. Konstruktori mogu imati parametre (vrednosti koje se šalju u metod), kao što je bio slučaj sa konstruktorom koji smo koristili u prethodnom primeru ("Probni tekst").

U klasi može postojati više konstruktora. Svaki od njih može (i treba) da ima različite ulazne argumente. Više o tome kasnije.

Konstruktori se pozivaju prilikom kreiranja novih objekata i služe za inicijalizaciju atributa tog objekta. Inicijalizacija se može obaviti i prilikom deklarisanja promenljivih kao što smo već pokazali.

```
class A{
   int c = 3;
}
```

Ako se prilikom deklaracije ne vrši i inicijalizacija, atributi klase će dobiti podrazumevane vrednost. To znači da će, na primer, brojevi biti incijalizovani na 0, promenljive tipa boolean na false, dok će objekti biti inicijalizovani na null.

Dobra programerska praksa nalaže da se ne oslanjate na podrazumevane vrednosti, već da se promenljive ili inicijalizuju, ili da im se vrednost dodeljuje u okviru konstruktora.

U klasi može da postoji i tzv. podrazumevani konstruktor. To je konstruktor koji nema argumente. Programer može sam da napravi podrazumevani konstruktor, ali i ne mora. Ako to nije učinio programer, kompajler će sam napraviti podrazumevani konstruktor. Taj podrazumevani konstruktor neće imati argumente, a telo će mu biti prazno.

## Tipovi podataka

U memoriji računara se skladište proizvoljni obrasci sa bitovima. Ono što ovim obrascima daje neki smisao su tipovi podataka.

Pogledajte sledeću sliku. To je pocepani list papira, na kome stoji neki tekst.


Šta mislite da znači tekst koji se ovde nalazi? Da li je to deo logoa neke firme. Da li je to ime neke osobe. Da li je to deo neke druge reči?

Naravno da na ova pitanja nije lako odgovoriti. Potrebno je imati dodatne podatke. Da biste tačno odgovorili, potrebno je da vidite ostatak lista, ili da znate odakle papir dolazi. Ako ne znate kontekst u kome se koristi, teško je reći šta konkretno znači.

Već smo pomenuli da se u memoriji računara nalaze proizvoljni nizovi bitova. Kao i kod ovih slova, značenje bitova zavisi od toga kako ćete ih upotrebiti. Konkretna šema koja se koristi za konkretan niz bitova se naziva tipom podatka.

Za tip podatka se može reći sledeće:

- To je šema za upotrebu bitova tako da predstave neke vrednosti.
- Vrednosti ne moraju biti brojevi, već mogu biti bilo koja vrsta podataka, sa kojom računar može da radi.
- Sve vrednosti u računaru su predstavljene preko nekog od tipova podataka.

Pogledajmo na primer, sledeći niz bitova:

```
0000000001100111
```

Ovo je niz od 16 bitova na koji biste mogli da naiđete negde u memoriji računara. Šta on predstavlja.

Ako ne znamo ništa više o obrascu koji se koristi, onda je nemoguće reći šta ovi bitovi predstavljaju. Ako je tip podatka (obrazac) short, onda je ovo vrednost 103 (broj). Tip short je jedan od primitivnih tipova podataka iz Jave.

Šta bi značio niz bitova 0000000000000000? Možda biste rekli da je to "nula". Ne mora da znači. Čak i ovako očigledni obrasci nemaju automatski neko značenje. Ako bismo Vam rekli da gornji obrazac koristi tip short, onda biste bili u pravu, kao tip short, taj obrazac predstavlja ceo broj 0.

Ako bi određeni obrazac uvek imao isto značenje, onda bi se moglo predstaviti samo nekoliko tipova podataka. To bi bilo suviše restriktivno. Zbog toga je programerima dozvoljeno da, tokom pisanja programa, uvode svoje tipove podataka (ne u svim programskim jezicima).

## Primitivni tipovi podataka

Bilo bi velikih problema, ako biste svaki put kada nameravate da koristite neke podatke, pisali svoju šemu predstavljanja pomoću bitova. Postoje neki tipovi podataka koji su tako fundamentalni, da su načini na koji se oni predstavljaju ugrađeni u programske jezike. Takvi tipovi se nazivaju primitivnim tipovima podataka.

U Javi postoji osam primitivnih tipova podataka. To su:

| byte | short | int | long | float | double | char | boolean |
|---|---|---|---|---|---|---|---|

Velika i mala slova kod imena ovih osam tipova su vrlo bitna. To znači da je "byte" ime primitivnog tipa podatka, ali "Byte" nije. Programski jezici kod kojih se kod kojih se pravi razlika između malih i velikih slova se označavaju kao "case sensitive". Neki jezici ne prave razliku između malih i velikih slova. Ovo se posebno odnosi na stare jezike koji su nastali kada i niste mogli uneti mala slova.

U frazi primitivni tipovi podataka reč primitivni znači "osnovna komponenta koja se koristi za kreiranje drugih, većih delova". Da biste rešili veliki problem, vi najpre pronalazite primitivne operacije, koje su potrebne, a onda ih koristite da dobijete rešenje.

## Objekti

U Javi postoji puno ugrađenih tipova, a vi (kao programer) možete da napravite koliko god želite novih tipova. Ipak, svi podaci koji postoje u Javi spadaju u jednu od dve kategorije. To su primitivni tipovi podataka i objekti. Postoji samo osam primitivnih tipova podataka. Bilo koji tip podatka koji vi sami uvedete u program je tipa objekat.

| Primitivni podaci | objekti |
|---|---|

Svi podaci

Evo nekih napomena vezanih za objekte i primitivne tipove podataka:

- Za primitivne tipove podataka se koristi mali, fiksiran, broj bajtova.
- U Javi postoji samo osam primitivnih tipova podataka.
- Programer ne može da napravi nove primitivne tipove podataka.
- Objekat je veliki blok podataka. Objekat može da koristi više bajtova u memoriji.
- Objekat se obično sastoji od više manjih delova.
- Tip podatka nekog objekta je njegova klasa.

Da biste bolje shvatili primitivne tipove podataka možete ih smatrati vijcima, navrtkama, dok je objekat cela mašina.

## Numerički primitivni tipovi podataka

Brojevi su u programiranju tako bitni, da je u Javi šest od osam primitivnih tipova namenjeno predstavljanju brojeva. Postoje primitivni tipovi za cele brojeve, kao i za realne brojeve. Celi brojevi (integer) nemaju decimalni delovi, dok realni imaju. Ovako je na papiru. U memoriji računara, međutim, ne postoje decimalni brojevi. Čak i realni brojevi, sa decimalnim delom, su predstavljeni preko niza bitova. Sa druge strane postoji razlika u metodu koji se koristi za predstavljanje celih brojeva, u odnosu na onaj koji se koristi za predstavljanje realnih brojeva.

Tipovi podataka za cele brojeve

| Ime | Veličina | Opseg |
|---|---|---|
| byte | 8 bitova | -128 do +127 |
| short | 16 bitova | -32,768 do +32,767 |
| int | 32 bita | (oko)-2 milijarde to +2 milijarde |
| long | 64 bita | (oko)-10E18 to +10E18 |

Svaki primitivni tip koristi, određen, nepromenljiv, broj bitova. to znači da će za isti tip podatka, biti upotrebljen isti broj bitova, bez obzira na to koji broj predstavljate. To na primer, znači da će sve vrednosti tipa short koristiti 16 bitova, odnosno vrednost trideset hiljada će biti predstavljena preko 16 bitova.

Sve vrednosti tipa long će biti predstavljene sa 64 bita. Vrednost nula (tipa long) će biti predstavljena sa 64 bita, vrednost trideset hiljada, takođe.

Vrednosti koje su veće, moraju biti predstavljene sa više bitova. Ovo je slično sa zapisivanjem brojeva na papiru. Za zapisivanje većih brojeva, vam je potrebno više cifara. Ako je za predstavljanje neke vrednosti potrebno više bitova, od broja koji se koristi za taj tip podatka, onda se ta vrednost ne može predstaviti preko tog tipa podatka.

Primitivni tipovi za predstavljanje realnih brojeva

| Tip | Veličina | Opseg |
|---|---|---|
| float | 32 bita | -3.4E+38 do +3.4E+38 |
| double | 64 bita | -1.7E+308 do 1.7E+308 |

Veći opseg kod numeričke vrednosti traži više bitova. Različite veličine kod tipova za predstavljanje celih brojeva, omogućavaju da izaberete pravi tip podatka, sa kojim ćete raditi. Obično birate tip podatka čiji je opseg mnogo veći od brojeva sa kojima očekujete da ćete realno raditi. Ako program sadrži samo nekoliko promenljivih, onda će on biti podjednako brz i neće zauzimati mnogo memorije, bez obzira na tip koji koristite za svoje promenljive.

Zašto biste onda, uopšte koristili tipove podataka manje veličine? Odgovor je u činjenici da većina programa koji se stvarno koriste rade sa ogromnim količinama podataka (milijarde stavki), tako da upotreba manjih tipova može da dovede do značajne uštede prostora i vremena. I pored toga programeri ne rade u skladu sa ovim. Uglavnom se za predstavljanje celih brojeva koristi tip int, a za predstavljanje realnih brojeva tip double.

Kada u programu želite da napišete neki broj, ne morate znati kako da ga predstavite preko bitova. Broj možete otkucati isto kao što biste uradili na pisaćoj mašini. Ovakav broj se naziva literalom. Primeri za literale koji predstavljaju cele brojeve su:

```
125          -32          16          0          -123987
```

Svi prethodni primeri su 32-bitni literali tipa int. 64-bitni literal tipa long ima veliko slovo "L" ili malo slovo "l" na kraju. Iako možete koristiti malo slovo l, nemojte, jer se ono vrlo lako može zameniti brojem 1. Uvek koristite veliko slovo L.

```
125L          -32L          16L          0l          -123987l
```

Poslednja dva primera su ispravna, ali mogu da dovedu do zabune.

### *Realni brojevi*

Ako u programu koristite vrednost 197.0, decimalna tačka govori kompajleru da ovu vrednost treba da predstavi preko nekog od tipova za realne brojeve. Obrazac bitova koji se koristi za broj 197.0 se puno razlikuje od obrasca koji se koristi za ceo broj 197.

Brojevi tipa float se ponekad nazivaju brojevima sa jednostrukom preciznošću. Takvi izrazi potiču iz programskog jezika FORTRAN, koji je jedno vreme bio dominantan programski jezik. Tip podatka double ima dva puta više bitova, pa se ponekad brojevi tog tipa nazivaju brojevima sa dvostrukom preciznošću.

Literali tipa realnih brojeva, u programima, koriste decimalnu tačku, a ne decimalni zarez.

```
123.0        -123.5         -198234.234       0.00000381
```

Literali koje su napisani kao u prethodnom primeru, automatski postaju tipa double. U principu, osim ako nemate neki specijalan razlog, u programima za realne brojeve, treba da koristite tip double. Tada će literali kao što su ovi odgovarati tipovima podataka vašin promenljivih. (Jedan od razloga primene tipa float bi mogao biti obrada datoteke u kojoj su 32-bitni realni brojevi.)

Možda bi neko pomislio da će promenljiva tipa float zeuzeti dva puta manje memorije u odnosu na promenljivu tipa double. To nije tačno. Ako u programima koristite promenljive tipa double, dobićete program koji je samo nekoliko bajtova duži. Ako se uzme u obzir da je veličina tih programa više hiljada bajtova, onda to nije puno.

Ponekad je potrebno da eksplicitno zatražite da literal bude tipa float. Ovo možete uraditi ako na kraju literala dodate slovo "F" ili "f".

```
123.0f        -123.5F         -198234.234f       0.00000381F
```

Ako je potrebno da eksplicitno zadate da je neki literal tipa double, možete na kraj dodati slovo "D" ili "d".

```
123.0d        -123.5D         -198234.234d       0.00000381D
```

Ako na kraju literala koji predstavlja realan broj, nema nikakvog slova, on automatski postaje tipa double.

Oba sledeća iskaza su ispravna:

```
double rats = 8912 ;
double rats = 8912.0 ;
```

Prvi iskaz će raditi, ali to nije baš najbolji kod. Pre inicijalizacije promenljive, literal tipa integer, mora da se konvertuje u tip double. U drugom iskazu je promenljiva inicijalizovana literalom tipa double.

Ponekad ćete koristiti i naučnu notaciju. Svi naredni literali su tipa double:

```
1.23E+02     -1.235E+02      -1.98234234E+05    3.81E-06
```

Veliko slovo E znači "puta 10 na stepen ". Ceo broj koji sledi pokazuje koji je to stepen na koji treba podići broj 10.

Ceo broj koji sledi iza slova E se može objasniti i da drugi način. To je broj koji govori u kom smeru i za koliko mesta treba pomeriti decimalnu tačku. Pozitivni brojevi ukazuju na pomeranje u desno, a negativni na pomeranje u levo.

### *Preciznost realnih brojeva*

Pogledajmo vrednost 1/3 u decimalnoj notaciji:

```
0.333333333333333333
```

Rezultat nije konačan. Nema granica koliko brojeva 3 dolazi. Kod podataka tipa float, gde na raspolaganju imamo samo 32 bita, nema dovoljno bitova za predstavljanje neograničenog broja trojki.

Podatak tipa float ima 23 bita za preciznost. (Preostali bitovi se koriste za definisanje veličine broja.) Ovo je ekvivalentno sa otprilike 7 decimalnih mesta.

Broj mesta za preciznost podataka tipa float, je isti bez obzira na veličinu broja. Ovim tipom podatka možete predstaviti brojeve koji su otprilite do 3.4E+38. Preciznost ovih velikih brojeva takođe je ograničena samo na sedam decimalnih mesta.

Za skladištenje tipa double se koristi 64 bita, a i opseg je mnogo veći i iznosi -1.7E+308 do +1.7E+308. Ovde je i preciznost mnogo veća. Ovde iznosi 15 značajnih cifara.

### *Podaci tipa char*

Karakteri se kod računara mnogo koriste. U Javi postoji poseban tip podatka, koji se koristi za predstavljanje karaktera. to je tip char. Promenljiva tipa char u memoriji zauzima 16 bitova. U mnogim programskim jezicima se za predstavljanje karaktera koristi samo 8 bitova. U Javi se koristi 16 bitova, pošto treba predtaviti i karaktere, koji dolaze iz drugih jezika, a ne samo iz engleskog. Takav način predstavljanja se naziva Unicode.

Pogledajte, na primer, pogledate obrazac

```
000000000110011
```

Ako znate da je ovih 16 bitova tipa char, onda možete da pogledate tabelu i pronađete da ovi bitovi predstavljaju karakter 'g'. Ako imate zaista dobro pamćenje, setićete se da je isti niz od 16 bitova predstavljao ceo broj 103, ako se posmatrao kao tip short. Da biste znali šta neki niz bitova predstavlja morate znati tip podatka.

Velika i mala slova se predstavljaju različitim nizovima bitova. Znaci interpunkcije i specijalni karakteri su takođe tipa char. Pod specijalnim karakterima se misli na "belinu" i slične karaktere.

Postoje i kontrolni karakteri koji označavaju kraj reda ili mesto početka nove strane.

Primitivni tipa char predstavlja JEDAN karakter. On ne sadrži nikakve informacije o fontu. Kada u istom trenutku radite sa više karaktera (što je skoro uvek), onda morate da koristite objekte koji su sastavljeni od više podataka tipa char.

Literali tipa karakter se u programu uokviruju apostrofom:

```
'm'          'y'                'A'
```

Kontrolni karakteri se u programima predstavljaju sa nekoliko karaktera koji se nalaze unutar apostrofa:

```
'\n'          '\t'           '\377'
```

Svaki od prethodnih primera predstavlja po jedan karakter. Prvi je 16-bitni karakter koji označava novi red, drugi je tabulator, a treći je karakter za brisanje.

### *Podaci tipa boolean*

U Javi postoji i primitivni tip podatka boolean. On se koristi da predstavi vrednost tačno ili netačno. Promenljiva tipa boolean može imati samo jednu od dve vrednosti:

tačno (true) ili netačno (false)

U programu pisanom u Javi reči true i false uvek označavaju vrednosti tipa boolean. Tip podatka je inače, dobio ime po matematičaru Džodžu Bulu.

### Klase omotači za primitivne tipove podataka

Sećate da je rečeno da postoji razlika između primitivnih tipova podataka i objekata. Taj jaz ponekad treba da se premosti. U skladu sa tim za svaki primitivni tip podatka postoji odgovarajuća klasa omotač. Ove klase se mogu koristiti za konverziju primitivnog tipa podatka u objekat, a i za konverziju nekih tipova objekata u primitivni tip. U sledećoj tabeli su prikazani tipovi podataka i njihove klase omotači:

| Primitivni tip | Klasa omotač |
|---|---|
| byte | Byte |
| short | Short |
| int | Int |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Kao primer možemo pogledati broj 103, koji ako se predstavi primitivnim tipom long u memoriji zuzima 16 bitova. Ista vrednost se može naći i u objektu tipa Long. Objekti koriste mnogo više od 16 bitova.

## Promenljive i operatori dodele

### Promenljive

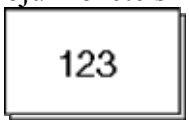
U svakom, pa i najmanjem programu, se stalno izračunavaju vrednosti, zapisuju vrednosti, program se vraća na prethodno izračunate vrednosti i sl. Ove vrednosti se čuvaju u malim delovima glavne memorije, koje nazivamo promenljivim.

Za skladištenje mašinskih intrukcija i podataka se koriste bajtovi u glavnoj memoriji. Elektronska kola glavne memorije (i ostalih tipova memorije) ne prave razliku između programa i podataka. Prilikom rada programa se neke memorijske lokacije koriste za mašinske instrukcije, a druge za podatke. Ideja o upotrebi iste memorije i za instrukcije i za podatke je potekla od pionira računarstva, Džona Njumana.

Da bi podatke mogao da stavi u memoriju i da ih kasnije odatle preuzme, program mora da ima ime za svaki deo memorije koji koristi.

Promenljiva je ime lokacije u glavnoj memoriji, koja koristi konkretan tip podatka, za skladištenje određene vrednosti.

Sećate se da je tip podatka konkretna šema za upotrebu obrasca bitova u glanoj memoriji, za predstavljanje podataka. Promenljivu možete shvatiti kao kutiju napravljenu od jednog ili više bajtova, u koju možete smestiti vrednost konkretnog tipa.




```
cena;
tip je "long"
```

Promenljive imaju ime, kao što je cena u prethodnom primeru.

### *Deklaracija promenljive*

Pogledajmo sledeći kod:

```
class example {
  public static void main ( String[] args ){
    long plata = 123;     //deklaracija promenljive
    System.out.println("Vrednost promenljive je: " + plata );
  }
}
```

Iskaz long plata = 123;  predstavlja deklaraciju promenljive. Deklaracija promenljive je mesto na kome program saopštava da će biti potrebna promenljiva.

Deklaracija promenljivoj daje ime i definiše njen tip podatka. Prilikom deklaracije se može i zadati konkretna vrednost koja se u tu promenljivu smešta. Kod jezika višeg nivoa (kao što je Java), programer ne treba da brine o tome šta računar stvarno radi prilikom deklaracije. Ako zatražite promenlivu tipa long, dobićete je. Detalji o bajtovima i memorijskim adresama su ostavljeni kompajleru.

Deklaracija u našem programu traži deo memorije od 64 bita, koji će se koristiti za primitivni tip podatka long i čije će ime biti plata. Inicijalno će se u ovu promenljivu smestiti vrednost 123. Kompajler (i kasnije interpreter) su tu da osiguraju da će se to zaista i desiti.

Promenljiva se u programu ne može koristiti ako nije deklarisana. Promenljiva se može deklarisati samo jednom. (Ovo važi za Javu, ali ne i za sve programske jezike).

### *Sintaksa deklaracije promenljive*

Sećate se da sintaksa označava gramatiku programskog jezika. Sintaksa se može odnositi i na pojedine delove programa, pa tako možemo govoriti i sintaksi deklaracije promenljive.

Deklaracija promenljive se može ostvariti na nekoliko načina:

- `tipPodatka imePromenljive;`
  - Ovo je prvi način za deklaraciju promenljive. Njime se deklariše samo zadaje tip podatka i daje ime promenljive.
- `tipPodatka imePromenljive = inicijalnaVrednost;`
  - Ovim iskazom se deklariše promenljive, zadaje se njen tip podatka, rezerviše se memorija za nju i u tu memoriju se stavlja vrednost. Inicijalna vrednost mora po tipu biti odgovarajuća.
- `tipPodatka imePrvePromenljive, imeDrugePromenljive;`
  - Ovim se deklarišu dve promenljive istog tipa, ali se tip promenljivim ne daje nikakva vrednost. Ako želite, ovo možete da uradite i za više od dve promenljive.
- `tipPodatka imePrvePromenljive = incijalnaVrednostJedan, imedrugePromenljive = inicijalnaVrednostDva ;`
  - Ovim se deklarišu dve promenljive. Obe su istog tipa i u obe se stavljaju inicijalne vrednosti. Na isti način možete deklarisati i više promenljivih.

### *Imena promenljivih*

Programer je taj koji bira ime promenljive koju će koristiti u programu. U programima postoji nekoliko stvari kojima treba zadati ime, a jedna od njih su promenljive. Ime koje programer izabere se naziva identifikatorom. Prilikom izbora identifikatora treba voditi računa o nekim pravilima:

- Koristite samo karaktere od 'a' do 'z', 'A' do 'Z', brojeve od '0' do '9', karakter '_' i karakter '$'. U imenu promenljive ne sme da se nađe belina (blanko karakter).
- Imena nemojte počinjati brojevima
- Ime može imati proizvoljnu dužinu.
- Velika i mala slova se smatraju različitim karakterima.
- Ime ne može biti rezervisana reč.
- Ime ne sme biti već upotrebljeno u nekom delu programa.

Rezervisana reč je reč koja u Javi već ima neko značenje. Na primer, int, double, true ili import su rezervisane reči. Umesto da pamtite kompletnu listu rezervisanih reči, upamtite da ne smete da ih koristite kod zadavanja imena, tako da to možete da lako promenite, ako slučajno pogrešite.

Sledeći primer sadrži nekoliko deklaracija promenljivih:

```
class example{
  public static void main ( String[] args )  {
    long   radniSati = 40;
    double plata     = 10.0, poreskaStopa = 0.10;

    System.out.println("Radni sati: " + radniSati);
    System.out.println("Plata : " + (radniSati * plata) );
    System.out.println("Porez  : " + (radniSati * plata * poreskaStopa)
 }
}
```

Karakter * označava množenje, pa prema tome (radniSati * plata) znači da treba pomnožiti brojeve koji se nalaze u ovim promenljivim.

Kada se znak + nađe iza stringa, on označava spajanje stringova. Prema tome "Radni sati:" + radniSati pravi novi string, koji počinje sa "Radni sati", a završava sa karakterima koji čine vrednost promenljive radniSati. Ovaj program će odštampati sledeći izlaz:

```
Radni sati: 40
Plata  : 400.0
Porez  : 40.0
```

Ovaj program ilustruje vrlo važnu ideju: Da biste koristili vrednost koja se nalazi u nekoj promenljivoj, potrebno je samo da upotrebite ime promenljive.

Pogledajte, na primer, prvi iskaz System.out.println, u kome smo koristili promenljivu radniSati. Ovaj iskaz kaže "pronađi vrednost promenljiev radniSati i upotrebi je ovde".

### *Pisanje iskaza u više redova*

Iskazi se mogu pisati u više redova. Svuda gde je dozvoljeno staviti blanko karakter, možete podeliti iskaz. To znači da iskaz ne možete podeliti na sredini imena, niti između navodnika string literala. Sledeći primer pokazuje kako neke iskaze možete podeliti u više redova:

```
class example{
```

```
  public static void main ( String[] args )  {
    long   radniSati = 40;
    double plata     = 10.0,
           poreskaStopa = 0.10;

    System.out.println("Radni sati: " +
radniSati);
    System.out.println("Plata  : "
+ (radniSati * plata) );
    System.out.println("Porez  : " + (radniSati
* plata * poreskaStopa) );
  }
}
```

Iako je ovo ispravno, podela iskaza na ovaj način može da dovede do zabune. U sledećem primeru je prikazan isti primer, ali sa poboljšanim stilom podele:

```
class example{
  public static void main ( String[] args )  {
    long   radniSati = 40;
    double plata     = 10.0,
           poreskaStopa = 0.10;

    System.out.println("Radni sati: " +
        radniSati );
    System.out.println("Plata  : " +
       (radniSati * plata) );
    System.out.println("Porez  : " +
       (radniSati * plata * poreskaStopa) );
  }
}
```

Trebalo bi da uvek drugi deo podeljenog iskaza uvlačite, tako da se lakše uoči da je u pitanju nastavak prethodnog reda.

## Iskazi dodele

Do sada smo uvek koristili vrednosti koje smo incijalno smeštali u promenljive, odnosno nismo menjali te vrednosti. Kako i samo ime ukazuje od promenljivih se očekuje da se njihove vrednosti menjaju. Promenu vrednosti promenljive ćete ostvariti preko iskaza dodele. Evo jednog primera u kome se koristi iskaz dodele:

```
class primer3{
  public static void main ( String[] args )  {
    long plata ;  //deklaracija bez inicijalne vrednosti

    plata = 123;  //iskaz dodele
    System.out.println("Vrednost promenljive je: " + plata );
  }
}
```

Iskaz dodele u promenljivu stavlja vrednost 123. To znači da će tokom izvršenja programa postojati blok memorije od 64 bita u kome će se nalaziti vrednost 123.

### Sinataksa iskaza dodele

Ovaj program je odštampao isti izlaz kao prvi primer. Ovaj program, međutim, nije inicijalizovao promenljivu, tako da je kasnije morao da u nju smesti neku vrednost.

Iskaz dodele ima sledeću sintaksu:

```
ime promenljive = izraz;
```

- Znak jednakosti "=" znači "dodeliti".
- imePromenljive je ime promenljive koja je deklarisana negde u programu.
- izraz je vrednost koju treba dodeliti.

Iskaz dodele računaru nalaže da obavi sledeća dva koraka:

1. Izračuna vrednost izraza
2. Da tu vrednost smesti u promenljivu.

Na primer, iskaz

```
sum = 32 + 8 ;
```

radi dve stvari:

1. izračunava vrednost izraza - izračunava se 32 + 8, što daje 40
2. izračunatu vrednost stavlja u promenljivu, odnosno sum dobija vrednost 40.

### Izrazi

Uobičajeno je da se kaže:

iskaz u promenljivu sum postavlja vrednost 30.

Ako je izraz sa desne strane komplikovan (nije običan broj, kao što je 30), onda tu postoje dva koraka koja treba uzeti u obzir.

Izraz je kombinacija literala, operatora, promenljivih i zagrada. Svi ovi elementi se koriste za izračunavanje vrednosti.

Ova definicija traži i neka objašnjenja:

- literali su karakteri koji direktno znače neku vrednost. Na primer: 3.456 je literal.
- operator je simbol, koji označava neku aritmetičku operaciju. Na primer +, * (množenje) i sl.
- promenljiva je deo memorije koji sadrži neku vrednost.
- Pod zagradama se ovde misli na otvorenu i zatvorenu malu zagradu.

Izrazi moraju imati odgovarajuću sintaksu, ali je najbolje pravilo da oni treba da izgledaju isto kao izrazi koje koristite u algebri.

### *Beline (razmaci) kod pisanja izraza*

Izraz se može napisati bez ikakvih razmaka. Operatori i zagrade su dovoljni za odvajanje delova izraza. Ipak, vi možete u izrazu upotrebiti razmake, da biste vizuelno odvojili delove koji imaju određeno značenje. Sledeći iskaz je, na primer, sasvim ispravan:

```
(radniSati*plata)-odbici
```

Sledeći izraz ima potpuno isto značenje:

```
(radniSati * plata) - odbici
```

Razmake treba koristiti pametno, da biste drugim ljudima (a i sebi) objasnili šta izraz znači. Razmaci se ne mogu postaviti u sredini identifikatora. Sledeći iskaz nije ispravan:

```
(radni Sati * plata) - odbici
```

Razmaci mogu biti sintaktički ispravno postavljeni, ali da ipak nisu upotrebljeni na pravi način:

```
12-4   /   2+2
```

Ovde izgleda kao da 4 treba da se oduzme od 12, a da se nakon toga rezultat podeli sa 4. Pošto se razmaci ne uzimaju u obzir, izraz je isti kao:

```
12   -   4/2  +  2
```

Ovako postavljeni razmaci imaju mnogo više smisla, jer je jasnije šta izraz u stvari znači.

### *Aritmetički operatori*

Aritmetički operator je simbol koji označava da treba da se uradi neka aritmetička operacija. Ako se u istom izrazu koristi više operatora, onda mora postojati redosled po kojem se operacije odvijaju. U tom smislu se može govoriti o prioritetu operatora. Prvo se izvršavaju operatori sa višim prioritetom. U Javi postoji puno operatora. Neki od njih su dati u sledećoj tabeli:

| Operator | Značenje | Prioritet |
|---|---|---|
| - | unarni minus | visok |
| + | unarni plus | visok |
| * | množenje | srednji |
| / | deljenje | srednji |
| % | deljenje sa ostatkom | srednji |
| + | sabiranje | nizak |
| - | oduzimanje | nizak |

Operatori iz iste grupe u ovoj tabeli, imaju isti prioritet. Tako na primer, operatori plus i minus imaju isti prioritet.

Izračunavanje izraza ide sa leva udesno.

U izrazu učestvuju operatori i operandi. Operatore smo već pomenuli, a operandi su su vrednosti na koje se operator primenjuje. Na primer, u iskazu

13 - 5,

13 i 5 su operandi, a - je operator.

Operacija u kojoj učestvuju samo celi brojevi se uvek odvija sa 32 bita, ili više. Ako je jedan od operanada 64-bitni (tip long), onda se i operacija radi sa 64 bita. U suprotnom se operacija uvek odvija sa 32 bita, čak i ako su oba operanda manja.

Ako se u izrazu koristi promenljiva tipa short (16 bitova), procesor tu vrednosti ubacuje u svoju 32-bitnu aritmetičku jedinicu za cele brojeve. Veličina promenljive ne mora da odgovara broju bitova koje procesor koristi za aritmetiku. Na primer:

```
short x = 12;         // 16 bitni short
int   rezultat;         // 32 bitni int

rezultat = x / 3;       // aritmetika se radi sa 32 bita
```

Kod izraza x/3 računar će podeliti 32-bitni broj 12 sa 32-bitnim brojem 3 i sve to staviti u 32-bitnu promenljivu rezultat. Literal 3 automatski predstavlja 32-bitnu vrednost. Evo još jednog primera:

```
short x = 12;
short y = 3;
short rezultat;

rezultat = x / y;
```

Kod izraza x/y će računar podeliti 32-bitni broj 12, sa 32-bitnim brojem 3, čak i ako promenljive x i y imaju po 16 bitova. Rezultat će biti skraćen na 16 bitova i onda postavljen u promenljivu rezultat.

Operator deljenja "/" znači celobrojno deljenje, ako su oba operanda celi brojevi. Ako je jedan od operanada realan broj, onda je u pitanju deljenje realnih brojeva. Rezultat celobrojnog deljenja je uvek ceo broj. Ako je rezultat deljenja realan broj, decimalni deo se odbacuje (ne zaokružuje se).

U tom smislu postoji razlika između onog što radi Java i što bi uradio digitron. Digitron će za izraz

```
7/2
```

uraditi deljenje realnih brojeva.

Java će na isti izraz primeniti aritmetiku celih brojeva, tako da će rezultat ovog izraza biti 3. Iako se ovo može učiniti čudnim, ipak većina programskih jezika, pa i Java funkcionišu na ovaj način.

Ako pogledamo operatore koje smo do sada pomenuli, videćemo da neki od njih rade sa jednim, a neki sa dva operanda. U tom smislu se operatori mogu podeliti na unarne (sa jednim operandom) i binarne (sa dva operanda). Binarni operatori uvek moraju imaju dva operanda, iako se ponekad može učiniti da ih ima više. U takvim slučajevima je jedan ili oba operanda podizraz. Na primer:

```
(12.0 * 31) / 12
```

Prvi operand za operator deljenja je u zagradama i predstavlja poseban izraz.

Ako su kod binarnih operatora oba operanda celi brojevi, onda je reč o aritmetici celih brojeva. Ako je makar jedan od operanada realan broj, onda je u pitanju operacija sa realnim brojevima.

Ako su a i b celobrojne promenljive, onda sledeći izrazi predstavljaju celobrojno deljenje:

```
12 * b
a - 2
56%a
```

Ako su a i b celi brojevi, a x i y realni brojevi, onda sledeći izrazi predstavljaju deljenje realnih brojeva:

```
x * b
(a - 2.0)
56*y
```

Pravilo o aritmetici celih brojeva ili aritmetici realnih brojeva se primenjuje i na složenije izraze, korak po korak. Pogledajmo sledeći izraz:

```
( 1/2 + 3.5 ) / 2.0
```

Kakav je rezultat? Primenićemo pravilo, prvo unutrašnje zagrade, a onda operatori sa najvišim operatorima.

```
( 1/2 + 3.5 ) / 2.0
 ---
 prvo ovo
```

Pošto su oba operanda celi brojevi, operacija je celobrojno deljenje, što daje rezultat:

```
( 0 + 3.5 ) / 2.0
```

Sada nastavljamo sa izrazom unutar zagrada. Operator plus je sabiranje realnih brojeva, pošto je jedan od operanada realan broj (3.5).

```
3.5 / 2.0
```

Na kraju se obavlja i poslednja operacija:

```
1.75
```

### *Operator deljenja po modulu*

Simbol za operaciju deljenja po modulu je kao što ste videli oznaka procenta %. Tako je na primer, rezultat sledećeg izraza:

13 % 5

broj 2. To je ostatak posle celobrojnog deljenja ova dva broja.

Pogledajte sledeći program:

```
class deljenjePoModulu{
  public static void main ( String[] args )  {
    int količnik, ostatak;

    količnik =  17 / 3;
    ostatak =  17 % 3;
    System.out.println("Količnik je: " + količnik);
    System.out.println("Ostatak je: " + ostatak);
    System.out.println("Original je: " +
        (količnik *3 + ostatak) );
  }
}
```

Iz ovog primera se vidi kako se radi sa ovim operatorom.

Ovaj operator može se koristiti i sa negativnim brojevima. U tom slučaju važe sledeća pravila:

- Ako su oba operanda pozitivni operacija se obavlja onako kako je opisano.
- Ako je levi operand negativan, onda je i rezultat negativan.
- Ako je levi operand pozitivan, onda je i rezultat pozitivan.
- Znak desnog operanda se zanemaruje u svim slučajevima.

Evo kako to izgleda na nekoliko primera:

```
17 %  3 == 2     -17 %  3 == -2
17 % -3 == 2     -17 % -3 == -2
```

### *Operatori inkrementiranja i dekrementiranja*

Operacija koja se u programima najčešće izvodi je najverovatnije povećavanje vrednosti promenljive za 1. Većina programa u sebi ima petlje. Većina petlji se kontroliše preko promenljive koja broji prolaze kroz petlju. Svaki put kada se izvrši petlja, promenljivoj se dodaje jedan.

Postoje i druge situacije kada se promenljivoj dodaje jedinica. Istraživanja su pokazala da je dodavanje jedinice nekoj promenljivoj operacija koja se najčešće izvršava. Sa druge strane i procesori su tako napravljeni da ovu operaciju izvršavaju vrlo brzo.

Već znamo kako se promenljivoj dodaje 1:

```
brojac = brojac + 1 ;  // dodavanje jedinice brojacu
```

Ovo je prilično jednostavno, ali postoji i kraći način da se to isto uradi:

```
brojac++ ;  // dodaje jedan brojacu
```

U ovom iskazu se koristi operator inkrementiranja ++. On vrednosti promenljive brojac dodaje 1.

Operator inkrementiranja ++ dodaje jedinicu promenljivoj. Obično je primenljiva celobrojnog tipa (byte, short, int, long), ali može biti i realan broj (float, double). Između dva znaka plus ne sme da se nađe nijedan karakter. Obično se znaci plus stavljaju odmah iza promenljive, premda to nije obavezno.

Operator inkrementiranja se može koristiti i kao deo aritmetičkog izraza.

```
int sum = 0;
int brojac = 10;
sum = brojac ++ ;
System.out.println("sum: "+ sum " + brojac: " + brojac);
```

Kako radi iskaz sum = counter++;

U ovom iskazu se vrednost promenljive brojac povećava nakon što se promenljiva upotrebi.

Iskaz dodele se izvršava na sledeći način:

- Izračunava se vrednost izraza sa desne strane znaka =
    - Vrednost je 10 (pošto se brojač još uvek nije povećao)
- Vrednost se dodeljuje promenljivoj sa leve strane znaka =
    - promenljiva sum dobija vrednost 10.
- Sada se primenjuje operator ++: brojac se povećava na 11
- U poslednjem iskazu se štampa: `sum: 10 brojac: 11`

Operator inkrementiranja treba koristiti pažljivo i samo kada je to potrebno. Ponekad izraz može biti duži, ali i jasniji bez ovog operatora.

Operator inkrementiranja koji smo do sada koristili se stavljao iza promenljive. On se može staviti i ispred promenljive. U tom smislu razlikujemo prefiks i postfiks operatore inkrementiranja. U oba slučaja se vrednost promenljive povećava za 1, ali postoji razlika u tome kako se to radi:

```
++ brojac - znači povećanje pre upotrebe
brojac ++ - znači povećanje nakon upotrebe
```

Ako se ovi operatori koriste samostalno, onda nije bitno da li se koristi prefiks ili postfiks verzija. Sa druge strane, ako je ovaj operator deo nekog složenijeg izraza, onda se mora voditi računa. Pogledajmo prethodni primer:

```
int sum = 0;
int brojac = 10;
sum = ++brojac ;
System.out.println("sum: "+ sum " + brojac: " + brojac);
```

Operator ++ je sada prefiks operator.

Sada se vrednost promenljive brojac povećava pre nego što se u širem izrazu upotrebi njena vrednost.

Iskaz dodele se izvršava u sledećim koracima:

- izračunava se izraz sa desne strane znaka =
    - vrednost je sada 11 (pošto se brojac povećava pre upotrebe)
- Promenljivoj sa leve strane znaka = se dodeljuje vrednost.
    - sum dobija vrednost 11
- Poslednji iskaz štampa `sum: 11 brojac: 11`

Pored operatora inkrementiranja postoje i operatori dekrementiranja. To je operator --. Ovaj operator smanjuje vrednost promenljive na koju se primeni za 1. Takođe postoje prefiks i postfiks verzija.

| Izraz | Operacija | Primer | Rezultat |
|---|---|---|---|
| x++ | vrednost se upotrebi pa se doda 1 | int x = 10;<br>int y;<br>y = x++ ; | x je 11; y je10 |
| ++x | dodaje 1, pa onda koristi vrednost | int x = 10;<br>int y;<br>y = ++x ; | x je 11; y je 11 |
| x-- | koristi vrednost, pa onda oduzima 1 | int x = 10;<br>int y;<br>y = x-- ; | x je 9; y je 10 |
| --x | oduzima se 1, pa se onda koristi vrednost | int x = 10;<br>int y;<br>y = --x ; | x je 9; y je 9 |

Sledeći kod:

```
int x = 99;
int y = 10;
y = --x ;
System.out.println("x: " + x + "  y: " + y );
```

Štampa sledeći izlaz:

```
x: 98 y: 98
```

### *Još neki operatori dodele*

Operatori +, -, *, / (i drugi) se mogu koristiti zajedno sa znakom jednakosti, čime se dobijaju složeni operatori. Na primer, sledeći iskaz će promenljivoj sum dodati 5.

```
sum += 5;
```

Ovaj iskaz ima isti efekat kao:

```
sum = sum + 5;    // add 5 to sum
```

| Operator | Operacija | Primer | Efekat |
|---|---|---|---|
| = | dodela | `sum = 5;` | `sum = 5;` |
| += | sabiranje sa dodelom | `sum += 5;` | `sum = sum + 5;` |
| -= | oduzimanje sa dodelom | `sum -= 5;` | `sum = sum - 5;` |
| *= | množenje sa dodelom | `sum *= 5;` | `sum = sum * 5;` |
| /= | deljenje sa dodelom | `sum /= 5;` | `sum = sum/5;` |

### *Konstante*

U programu će se često dešavati da je potrebno da konstantnoj vrednosti date ime. Možete na primer na taj način definisati poresku stopu od 18% i poresku stopu od 8%, koje trenutno kod nas postoje. Ovo su konstante zato što njihova vrednost tokom izvršenja programa ne bi smela da se menja. Ovo se može uraditi na sledeći način:

```
class RacunanjePoreza{
  public static void main ( String[] arg )  {
    final double visaStopa = 0.18;
    final double nizaStopa = 0.08;

    . . . . . .
  }
}
```

U programu možete uočiti ključnu reč final. Ova rezervisana reč govori kompajleru da se vrednost neće menjati. Ovo je poput ugovora između vas i kompajlera. Vi se obavezujete da tu vrednost nećete menjati, a kompajler je tu da proverava da li se tog obećanja držite.

Imena konstanti podležu istim pravilima kao i imena promenljivih. (Programeri ponekad za imena konstanti koriste velika slova, ali je to stvar stila, a ne deo specifikacije jezika.) Konstante dalje mogu da se koriste u izrazima tipa:

```
porez  = ukupanIznos * visaStopa ;
```

Sa druge strane, sledeći iskaz sadrži sintaktičku grešku:

```
visaStopa = 0.20;    // pokušaj promene poreske stope (neuspešan)
```

Zašto se koriste konstante? Dva su osnovna razloga:

One povećavaju čitljivost programa i olakšavaju proveru njegove ispravnosti.
Ako konstanta treba da se promeni (na primer, novi poreski zakon promeni poreske stope), onda jedino treba da promenite deklaraciju. Ne morate da pretražujete ceo program i da menjate svako pojavljivanje