**Core Java Programming**

**Kompajler** (engl. compiler) je kompjuterski program čija je osnovna svrha generisanje drugih kompjuterskih programa. Kompajler prevodi izvorni kod programa napisanih jezicima višeg nivoa u mašinski, kako bi takav kod mogao kompjuter da izvrši. Posao koji kompajler obavlja drugačije se naziva kompajliranje, a jezici koji koriste kompajler - kompajlirani jezici. Posao kompajliranja obavlja se odjednom, pre nego što je potrebno da se određeni program napisan višim jezikom izvrši na nekom računaru. Rezultat kompajliranja jeste izvršni fajl sa sadržajem napisanim mašinskim kodom.

**Interpreter** obavlja identičan posao kao i kompajler, ali na nešto drugačiji način. Za razliku od kompajlera, interpreter obavlja prevođenje manjih delova izvornog koda, i to u toku njegovog izvršavanja. Tako korišćenjem interpretera nije potrebno prevesti kompletan program, već se prevođenje obavlja u delovima, prilikom izvršavanja.

**Algoritam** se definiše kao ispravno opisana procedura za rešavanje nekog problema, najčešće problema u matematici ili kompjuterskom programiranju.

Da bi se algoritam prezentovao, potrebno je iskoristiti neki od načina koji se u te svrhe koriste. Četiri najčešće korišćena načina za predstavljanje algoritama su:

1. predstavljanje rečima, govorom, engleskim ili nekim drugim jezikom
2. predstavljanje crtanjem dijagrama toka (engl. flowchart)
3. predstavljanje pseudokodom i
4. predstavljanje kodom nekog programskog jezika.

Kompjuterski algoritmi mogu se predstaviti korišćenjem tri osnovne kontrolne strukture:

1. **sekvenca,**
2. **selekcija i**
3. **ponavljanje**

**Sekvencijalna** **struktura** je konstrukcija kod koje se koraci algoritma izvršavaju jedan za drugim.

**Selekcija** je jedna od struktura izvršavanja programske logike, koja omogućava da se određeni koraci uopšte ne izvrše. Tako selekcija omogućava modelovanje logike koja treba da sadrži neku odluku.

**Struktura ponavljanja** omogućava da se određeni koraci algoritma, samim tim, i kompjuterskog programa izvrše više puta, sve dok za tim ima potrebe.

**Pseudokod** je još jedan od načina na koji je moguće artikulisati algoritme, i to korišćenjem izjava napisanih prirodnim jezikom, koje mogu sadržati određene reči engleskog jezika.   
Jednostavno, svi programski jezici poseduju svoj skup pravila za pisanje koda, koja se drugačije nazivaju **sintaksa**. Algoritam napisan korišćenjem pseudokoda je univerzalan, tako da je njega sa lakoćom moguće implementirati korišćenjem bilo kog programskog jezika.

**Java virtuelna mašina** predstavlja jednu vrstu softverskog procesora, čija je uloga da bytecode prevede u varijantu mašinskog jezika koja odgovara platformi na kojoj će Java program biti izvršen. Zbog postojanja bytecodea i virtuelne mašine, može se reći da Java izvorni kod prolazi kroz dva koraka prevođenja pre nego što ga hardver kompjutera izvrši:

* prvi korak prevođenja jeste kompajliranje izvornog koda u bytecode;
* drugi korak prevođenja obavlja Java virtuelna mašina, čiji je sastavni deo Just-In-Time kompajler (**JIT**), koji prevođenje bytecodea u mašinski kod obavlja kombinovanjem kompajliranja i interpretiranja.

Zbog dva upravo spomenuta koraka prevođenja, za Javu se može reći da je kompajlirani, ali i intrerpretirani jezik.

Pored virtuelne mašine, svaka Java platforma poseduje i veliki skup gotovih funkcionalnosti koje su na raspolaganju programerima prilikom pisanja Java programa. Gotove funkcionalnosti koje su sastavni deo Java platformi drugačije se nazivaju aplikativni programski interfejs (Application Programming Interface – API).

Svaka od četiri nešto ranije pomenute Java platforme poseduje jasno definisano mesto primene, koje diktira i osobine virtuelne mašine, ali i skup funkcionalnosti:

* **Java SE** – sastoji se od skupa funkcionalnosti koji čini osnovu Java jezika; reč je o svim osnovnim funkcionalnostima koje čine Java jezik, a kojima će biti posvećen kompletan kurs koji je pred vama;
* **Java EE** – predstavlja nadogradnju Java SE platforme funkcionalnostima za razvoj kompleksnih, višeslojnih, pouzdanih mrežnih aplikacija;
* **Java ME** – platforma čiji su virtuelna mašina i skup pripadajućih funkcionalnosti posebno optimizovani za uređaje sa ograničenim hardverskim mogućnostima, kao što su mikrokontroleri, senzori, štampači, digitalni foto-aparati i slično; poseduje redukovan set Java SE funkcionalnosti;
* **JavaFX** – platforma koja je posebno namenjena za razvoj Java aplikacija sa grafičkim korisničkim okruženjem, primarno za desktop, ali i za web i mobilne platforme.

**Java Runtime Environment (JRE) –**softverska distribucija koja poseduje Java izvršno okruženje; izvršno okruženje je drugi naziv za Java virtuelnu mašinu; reč je o distribuciji koja omogućava da se na nekom kompjuteru izvršavaju Java programi;

**Java Development Kit (JDK) –**distribucija koja se sastoji iz izvršnog okruženja, odnosno virtuelne mašine, odgovarajućeg skupa funkcionalnosti i različitih alata koji su namenjeni razvoju Java programa.

**JRE** je namenjen isključivo za to da omogući izvršavanje Java programa, dok **JDK**, pored izvršavanja, omogućava i njihov razvoj.

Za pisanje Java koda, mozemo koristiti razlicita razvojna okruzenja a neka od najpoznatijih su IntelliJ IDEA i Visual Studio Code.

Kompletan izvorni kod Java programa grupisan je u nešto što se u Java jeziku naziva **klasa**. Svaki Java kod postoji isključivo unutar neke klase. Drugim rečima, nijedan Java kod ne može postojati izvan klase.

Java kompajler je zapravo fajl koji nosi naziv **javac**. Da bismo provjerili dostupnost java kompajlera, unutar CMD terminal, pokrenuti cemo sledecu komandu:

javac -version

U vecini razvojnih okruzenja, nakon kreiranja java projekta, automatski se generise kod klase koji izgleda ovako:  
  
**public** **class** MyFirstProgram {

}

**Metoda main**

Metode su još jedan način za grupisanje koda u programskom jeziku Java. Ipak, za razliku od klasa, unutar metoda se grupiše izvršni kod, odnosno kod koji obavlja neki posao. Metode se smeštaju unutar klasa, a jedna klasa može imati proizvoljan broj metoda.

Metoda main je specijalna metoda koja predstavlja ulaznu tačku Java programa. Svaki Java program mora imati main metodu. To je metoda koja se automatski poziva kada se pokrene Java program.

**Uvod u sintaksu Java jezika  
  
Šta je sintaksa nekog programskog jezika?**

Sintaksa programskih – ali i bilo kog govornog jezika – predstavlja skup pravila koja učesnik u komunikaciji mora da ispoštuje kako bi ostali korisnici tog jezika mogli da ga razumeju

Kod programskih jezika, postojanje i najmanje sintaksne greške u komunikaciji između programera i kompjutera znači nemogućnost prevođenja izvornog koda. Upravo zbog toga je striktno pridržavanje sintaksnih pravila prilikom pisanja programskog koda nekog jezika veoma važno.

**Struktura Java programa**

Programski jezik Java, kao i većina drugih modernih jezika, najveći deo sintakse preuzima iz jezika C++, odnosno C. Naime, jezik C++ je u trenutku nastanka Jave bio najrasprostranjeniji programski jezik, pa takav odabir tvoraca Jave uopšte ne čudi.

Osnovni elementi od kojih se može sastojati izvorni kod Java programa su sledeći:

izjave (naredbe);

blokovi;

rezervisane (ključne) reči;

identifikatori;

literali;

komentari;

prazni prostori.

Za svaki od upravo nabrojanih elemenata Java programa postoje jasna pravila za njegovo kreiranje.

**Izjave (naredbe)**

Izjava (naredba) predstavlja najmanji samostalni deo jednog programa. Izjava se može uporediti i sa jednom rečenicom u svakodnevnim govornim/pisanim jezicima. U izvornom kodu našeg prvog Java programa postoji sledeća izjava:

System.out.println("Hello World");

U programskom jeziku Java, svaka izjava se mora završiti karakterom tačka i zapeta.

Nepostojanje karaktera tačka i zapeta na kraju izjave u programskom jeziku Java se smatra sintaksnom greškom.

**Sintaksne greške**

Bilo koje nepoštovanje sintaksnih pravila jezika Java proizvodi sintaksnu grešku. Izvorni kod sa sintaksnom greškom nije validan i ne može se kompajlirati u bytecode. Stoga je preduslov za kompajliranje izvorni kod bez ijedne sintaksne greške.

**Izjave i linije**

Veoma često se može čuti da se veličina nekog programa meri u linijama, pa se tako može naići npr. na 100 linija koda, 1245 linija koda ili na 128. liniji koda se nalazi sintaksna greška.

U praksi se zbog preglednosti najčešće pribegava smeštanju svake naredbe u zasebnu liniju.

**Blokovi**

Blok koda predstavlja način za grupisanje izjava, odnosno naredbi u jednu logičnu celinu. U primeru prve Java aplikacije prikazane na početku ove lekcije postoje dva bloka koda. Njih je veoma lako prepoznati po otvorenim i zatvorenim vitičastim zagradama  – {}.

Prvi blok iz naše Java aplikacije izgleda ovako:

public class MyFirstProgram {

}

Ovo je blok koda koji se koristi za kreiranje jedne klase.

Drugi blok iz naše Java aplikacije kojim se kreira jedna metoda, je sledeći:

public static void main(String[] args) {

System.out.println("Hello World");

}

Na kraj bloka ne stavlja se karakter tačka zapeta.

**Vitičaste zagrade**

U programskom jeziku Java, blokovi se kreiraju upotrebom vitičastih zagrada – {}. { označava početak bloka, a } završetak bloka. **Svaki blok koji se otvori mora se i zatvoriti.** Stoga, svaka otvorena vitičasta zagrada mora posedovati i pripadajuću zatvorenu vitičastu zagradu.

**Ključne reči**

Izvorni kod našeg prvog Java programa sa početka ove lekcije poseduje izvestan broj specijalnih reči – public, class, void... Reč je o elementima jezika koji se nazivaju ključne reči.

Ključne reči su reči koje su rezervisane od strane jezika, pa tako za Java kompajler imaju posebno značenje. Tako, na primer, kada kompajliranje izvornog koda dođe do reči class, kompajler zna da je naša namera da kreiramo novu Java klasu.

Java jezik poseduje veliki broj ključnih reči, mozemo ih vidjeti u sledecoj tabeli:

A white background with black text

Description automatically generated

**Identifikatori**

Prilikom pisanja Java koda, programer ima mogućnost i da samostalno formira nazive određenih jezičkih elemenata. To se postiže korišćenjem identifikatora. Jednostavno, određeni jezički elementi, kao što su klase i metode, moraju da budu imenovani. Upravo zbog toga u programskom jeziku Java postoje identifikatori.

U primeru našeg prvog programa postoji nekoliko identifikatora. Za početak, naziv klase je jedan primer identifikatora:

public class MyFirstProgram {

}

Takođe, naziv metode je još jedan primer identifikatora:

public static void main(String[] args) {

System.out.println("Hello World")

}

Sada je primer identifikatora reč main.

**Pravila za definisanje identifikatora:**

Bitno je razumeti da identifikatori nisu nešto što je unapred definisano jezikom. Drugim rečima, programer sam smišlja njihove nazive. Zato je veoma bitno znati da je prilikom definisanja identifikatora potrebno poštovati nekoliko pravila za njihovo pisanje:

identifikatori mogu biti sastavljeni od slova, brojeva i karaktera donja crta i dolar;

identifikatori moraju sadržati makar jedan karakter;

identifikatori ne smeju početi brojem;

identifikatori su osetljivi na velika i mala slova (case sensitive), tako da myClass i MyClass nije isto;

ključne reči se ne mogu koristiti kao identifikatori.

Jedan primer validnog identifikatora, koji predstavlja naziv klase, može da izgleda ovako:

class JavaClass

{

}

Sa druge strane, primer sintaksno netačnog identifikatora može da izgleda ovako:

class 5Class

{

}

**Literali**

Pored identifikatora, programer ima mogućnost da proizvoljno, odnosno samostalno definiše još jedan jezički element – reč je o literalima, odnosno vrednostima.

Literali predstavljaju način za predstavljanje vrednosti unutar programa. U primeru našeg prvog Java programa postoji jedan literal. Reč je o tekstu koji naš program ispisuje kao svoj rezultat:

System.out.println("Hello World");

Unutar ovakve izjave, odnosno naredbe, tekst "Hello World" je primer literala. Ovim literalom se u programu predstavlja jedna tekstualna vrednost koju program ispisuje na izlazu. Pored tekstualnih vrednosti, prilikom razvoja Java programa moguće je kreirati i razne druge tipove vrednosti, pa se tako literali koriste za predstavljanje brojeva, teksta, logičkih vrednosti...

**Komentari**

Prilikom pisanja Java koda moguće je napisati i tekst koji nema nikakvo značenje za kompajler. Na taj način je moguće napisati neku dodatnu informaciju o kodu koja će poslužiti kao uputstvo ili naznaka za lakše snalaženje programera. Takve dodatne informacije se nazivaju komentari.

A white background with black text

Description automatically generated

**Javadoc**

Java poseduje ugrađen mehanizam za generisanje dokumentacije o kodu koji se piše. Ovaj mehanizam se naziva Javadoc. Javadoc komentari počinju kosom crtom, nakon čega slede dve zvezdice (/\*\*), a završavaju se zvezdicom i kosom crtom (\*/). Iz ovoga se može zaključiti da je razlika u odnosu na standardne višelinijske komentare samo u jednom karakteru zvezdica više prilikom otvaranja Javadoc komentara:

/\*\*

\* this is example

\* of Javadoc comment

\*/

Jedan realan primer Javadoc komentara bi mogao da izgleda ovako:

/\*\*

\* This is my test class.

\* @author John Lord <john.lord@example.com>

\* @version 1.8

\* @since 1.4

\*/

public class Test {

// program logic

}

Prikazanim Javadoc komentarom definišu se neke osnovne informacije o programu, koje primarno pomažu drugim programerima da razumeju čemu je namenjen kod koji se nalazi unutar Test klase.

Bitno je da primetite i da svaka linija unutar Javadoc komentara započinje karakterom zvezdica. Tako nešto nije pravilo, već samo konvencija, odnosno preporuka, zbog bolje preglednosti i čitljivosti.

**Prazna mesta**

Analizom izvornog koda našeg prvog Java programa može se primetiti da je u njegovom formiranju iskorišćeno specifično formatiranje koda. Kod je smešten u određeni broj novih redova (linija), neki elementi su razdvojeni praznim mestima, dok su određene linije uvučene u odnosu na ostatak koda. Sve ovo je urađeno kao bi se dobio pregledniji i čitljiviji kod. Prazna mesta (white spaces) koja su u te svrhe korišćena su:

* razmaci;
* tabulatori;
* prelasci u novi red.

Primenom specifičnog formatiranja koje poboljšava čitljivost koda, dobijaju se linije. Veoma je bitno znati da za Java kompajler pojam linije, baš kao ni pojam praznih mesta, nema nikakvo značenje. Naime, potpuno je legitimno napisati kod kompletnog programa u jednoj liniji ali se ne praktikuje zbog otezavanja citanja, kao i razumijevanja koda.

**Tipovi podataka**

Srž svakog programskog jezika jeste mogućnost obrade podataka. Podaci koji se obrađuju mogu biti različitog tipa: brojevi, tekst, [logičke vrednosti](https://www.link-elearning.com/linkdl/opisPojma.php?id=160438)... U primeru našeg prvog Java programa iz prethodnih lekcija, definisali smo samo jedan podatak:

public class MyFirstProgram {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

**Šta su tipovi podataka?**

Tipovi podataka omogućavaju da se unutar programa predstave različite vrste vrednosti – karakteri, tekst, razne vrste brojeva, logičke vrednosti, sve su to tipovi podataka. Tipovi podataka omogućavaju da se, prilikom definisanja neke vrednosti, Java virtuelnoj mašini naglasi kojeg će tipa biti ta vrednost.

A black background with arrows pointing to a message

Description automatically generated

String je tip podataka kojim se u Java jeziku predstavlja tekst; message je primer identifikatora pomoću koga će tekstualna vrednost biti dostupna u nastavku koda.

Karakter jednako (=) je primer jednog operatora. Tekst Hello World koji se nalazi ispod navodnika je vrednost, odnosno literal.

Na kraju, kompletna prikazana naredba jeste način na koji se u Java jeziku kreira jedna promenljiva.

**Tipovi podataka u Javi**

Praktično svi programski jezici podrazumevaju generalnu podelu tipova podataka na proste (primitivne) i složene tipove. Takva je situacija i kod programskog jezika Java, gde postoje primitivni tipovi i referentni tipovi; može se čuti i termin složeni ili objektni tip, a reč je o sinonimima za termin referentni tip. Stoga se na kraju može zaključiti da u Javi postoje dve različite grupe tipova:

prosti, odnosno primitivni tipovi;

složeni, referentni, odnosno objektni tipovi.

**Primitivni tipovi**

A close-up of a computer screen

Description automatically generated

U tabeli iznad, prikazani su svi primitivni tipovi jezika Java. Kolona Tip podatka oslikava naziv tipa, ali i ključnu reč jezika koja se koristi za dobijanje konkretnog tipa prilikom pisanja programskog koda. U tabeli je takođe prikazan i prostor koji podaci različitih tipova zauzimaju u kompjuterskoj memoriji, kao i opsezi vrednosti koje oni mogu imati.

**Bit i bajt i opsezi prostih tipova**

Svaki podatak zauzima određeni prostor unutar kompjuterske memorije. Memorijski prostor koji podaci zauzimaju u memoriji izražava se korišćenjem jedinice bit. **Bit** je najmanja jedinica za predstavljanje digitalnih podataka. Jedan bit može imati samo dve vrednosti – 0 ili 1. U tabeli možete videti da boolean tip podatka u memoriji zauzima samo jedan bit, i upravo zbog toga podaci takvog tipa mogu imati samo dve vrednosti.

Ostali prosti tipovi u Javi zauzimaju veću količinu memorije, a sve u zavisnosti od njihovog opsega. Tako tip byte u kompjuterskoj memoriji zauzima 1 bajt. Bajt je jedinica koja predstavlja 8 bita. S obzirom na to da se sastoji od 8 bita, opseg ovog tipa podataka se računa vrlo lako: 28 = 256 (dva na osmi jednako je 256). S obzirom na to da se tipom byte može predstaviti 256 različitih vrednosti, njegov opseg je od -128 do 127, što na kraju daje 256 jedinstvenih vrednosti. Po identičnom principu se računaju i opsezi ostalih prostih tipova u Javi.

U Javi postoji nekoliko tipova za predstavljanje brojeva i po jedan za predstavljanje karaktera i logičkih vrednosti. Može se, zapravo, reći da u Javi postoje tri osnovne grupe primitivnih tipova:

* celobrojni tipovi – byte, short, int, long;
* tipovi za predstavljanje brojeva sa decimalom – float, double;
* tip za predstavljanje karaktera – char;
* logički tip - boolean.

**Celobrojni tipovi**

Java definiše četiri vrste celobrojnih podataka: byte, short, int i long.

Najčešće korišćeni tip podataka u Javi je **int**, zato što njegov opseg u većini slučajeva zadovoljava potrebe predstavljanja celobrojnih vrednosti. Stoga je, kako bi se u Java programu definisala jedna celobrojna vrednost, moguće napisati sledeće:

int x = 10;

Svaki Java program zahteva postojanje main metode. Ova metoda se mora naći unutar neke klase, pa je tako postojanje jedne klase i main metode apsolutni minimum koda koji moramo imati kako bismo mogli da dobijemo neki efekat. Prilikom pokretanja programa od strane Java virtuelne mašine, aktivira se kod koji se nalazi unutar main metode, pa će upravo to biti mesto unutar koga ćemo i mi pisati kod u ovoj lekciji:

public class MyFirstProgram {

public static void main(String[] args) {

int x = 10;

System.out.println(x);

{  
}

Najveci broj koji je moguće predstaviti korišćenjem int tipa je: 2147483647. Evo šta se događa ukoliko se kao int tip pokuša definisati vrednost koja je veća od upravo navedene:

A computer screen shot of a program

Description automatically generated

Na slici možete videti da razvojno okruženje jasno signalizira da u kodu postoji sintaksna greška i da je definisani broj isuviše veliki kako bi bio definisan kao int tip. Rešenje se ogleda u korišćenju celobrojnog tipa proširenog opsega. Takav tip se u Javi zove **long**:

long x = 2147483648L;

Sada je tip podatka promenjen u long, što omogućava definisanje celobrojnih vrednosti znatno šireg opsega. Takođe, i samu vrednost je bilo potrebno modifikovati i na njen kraj dodati veliko slovo L. Nakon takvih izmena, naš kod više ne sadrži sintaksu grešku.

Po identičnom principu funkcionišu i byte i short celobrojni tipovi. Razlika je samo u opsegu vrednosti koje se mogu definisati. Vrednosti u opsegu od -128 do 127 moguće je predstavljati tipom **byte**:

byte x = 10;

Kada je potrebno predstaviti vrednost veću od 127, ali manju od 32767, može se koristiti tip **short**:

short x = 322;

Tipovi byte i short koriste se kada ušteda memorije igra važnu ulogu i kada znamo da celobrojne vrednosti neće biti veće od opsega ovih tipova. U realnim okolnostima, prilikom razvoja modernog softvera za web, desktop ili pametne telefone, upotreba byte i short tipova je vrlo retka, pa su najkorišćeniji tipovi int i long.

Bez obzira na celobrojni tip koji se koristi, vrednost je moguće definisati u nekoliko različitih oblika:

* dekadni (decimalni);
* [oktalni](https://www.link-elearning.com/linkdl/opisPojma.php?id=160440);
* [heksadecimalni](https://www.link-elearning.com/linkdl/opisPojma.php?id=160441);
* Binarni

Dekadni, odnosno decimalni oblik je podrazumevan. Ostali oblici se definišu korišćenjem specifičnih prefiksa koji se dodaju na vrednosti:

int decVal = 26;

int octVal = 032; // 2 \* 8^0 + 3 \* 8^1 = 2 + 24 = 26

int hexVal = 0x1a; // 10 \* 16^0 + 1 \* 16^1 = 10 + 16 = 26

int binVal = 0b11010; // 2^1 + 2^3 + 2^4 = 2 + 8 + 16 = 26

Kod ilustruje definisanje četiri pojedinačne celobrojne vrednosti korišćenjem različitih oblika, i to po redosledu definisanja: dekadni, oktalni, heksadecimalni i binarni. Svakim oblikom se zapravo definiše identična vrednost (26), a u komentarima koji su definisani u nastavku naredbi možete videti matematiku koja se koristi za pretvaranje konkretne vrednosti u dekadni oblik.

**Decimalni tipovi**

U javi je značajna i mogućnost predstavljanja brojeva koji nisu celi, već poseduju i određenu vrednost nakon decimalne tačke. Takvi tipovi se veoma često nazivaju i tipovi sa pomičnim (pokretnim) zarezom (floating point) ili jednostavno decimalni tipovi. U Java jeziku postoje dva takva tipa: float i double. Razlika je samo u preciznosti, odnosno opsegu, pri čemu double obezbeđuje duplo veći opseg.

Kako bi se, na primer, u Java programu predstavila konstanta Pi, neophodno je iskoristiti jedan od dva upravo spomenuta tipa:

double PI = 3.14;

Na float vrednost je neophodno dodati veliko slovo **F**, kako bi prevodilac znao da želimo da koristimo tip duplo manje preciznosti:

float PI = 3.14F;

**Karakteri**

Kada je u Java programu potrebno predstaviti neki karakter, koristi se primitivni tip **char**. Evo kako može izgledati definisanje karaktera koji predstavlja veliko slovo X:

char someChar = 'X';

Bitno je da odmah primetite da se prilikom definisanja vrednosti karaktera koriste apostrofi, odnosno da se karakter postavlja između jednostrukih navodnika.

U tabeli možete videti da char tip u Javi zauzima 16 bita (2 bajta), te da je njegov opseg vrednosti od 0 do 65535.

Napomena:

Tip char koristi se isključivo za predstavljanje pojedinačnih karaktera. To praktično znači da nije moguće postaviti više od jednog karaktera ispod jednostrukih navodnika prilikom definisanja char vrednosti.

**Logički tip – boolean**

U osnovi svakog programskog jezika jeste i logički tip podataka, koji omogućava definisanje podataka koji mogu imati dva oblika: tačno/netačno, uključeno/isključeno, 0/1... Reč je o tipu koji se u Java jeziku naziva boolean.

Tip boolean u memoriji zauzima 1 bit, što praktično znači da može imati samo dve vrednosti:  0 i 1. U programskom jeziku Java se ove dve vrednosti predstavljaju vrednostima **true**ili **false**(tačno ili netačno):

boolean isTrue = true;

ili

boolean isTrue = false;

**Složeni (referentni) tipovi**

Složeni tipovi podataka su naziv dobili na osnovu osobine koja im omogućava da unutar sebe mogu da objedine veći broj vrednosti, ali i operacija, odnosno ponašanja.

Složeni tipovi podataka u ovom trenutku značajno prevazilaze naše poznavanje Java jezika. Kako bismo se na pravi način upoznali sa složenim tipovima podataka, neophodno je da obradimo pojmove klasa i objekata. Tim pojmovima biće posvećen naredni modul ovog kursa. Ipak, sa nekoliko složenih, odnosno referentnih tipova možemo se upoznati i sada:

**Tekstualni podaci**

U programskom jeziku Java, za rukovanje tekstualnim vrednostima koristi se tip **String**. Stoga, kada je potrebno definisati neki tekst, pribegava se sledećem:

String myWord = "Hi!";

Ovo je način da se u Java jeziku definiše tekstualni podatak. Potrebno je obratiti pažnju na sledeće:

* ključna reč String ima veliko početno slovo; veliko slovo izdvaja ovaj tip od skupa primitivnih tipova koji se pišu malim slovom; naime, String je složeni tip, ali se može kreirati kao i primitivni tipovi; zbog toga se String veoma često naziva specijalnim tipom;
* vrednosti tipa String pišu se između klasičnih, dvostrukih navodnika, za razliku od char tipa, čije se vrednosti smeštaju između jednostrukih navodnika.

Kao što možete videti, upotreba String tipa se ni po čemu ne razlikuje od korišćenja prostih tipova prikazanih ranije u ovoj lekciji. Mi ćemo i u nastavku koristiti String kao da je reč o prostom tipu, a kada se bolje upoznamo sa pojmom složenih tipova, iskoristićemo i sve blagodeti koje String poseduje kao složeni tip podatka.

**Specijalni karakteri unutar Stringa**

Upravo ste mogli da vidite da se tekstualni podaci u Java jeziku smeštaju ispod navodnika.

Ukoliko u vrijednosti naseg stringa, zelimo da koristimo navodnike, pojaviti ce se sintaksna greska. Kako bi se rešio ovakav problem, pribegava se upotrebi karaktera obrnuta kosa crta (backslash):

String message = "This is quotation mark - \"";

Ispred karaktera navodnik sada je postavljen karakter backslash, čime se dobija sintaksno ispravan kod. Jednostavno, karakter backslash signalizira kompajleru da je unutar tekstualne vrednosti potrebno da se pojavi navodnik.

Ovo nije jedina situacija u kojoj se karakter backslash koristi ispred određenih karaktera prilikom definisanja String vrednosti. Korišćenje karaktera backslash čini da u nekim situacijama karakter koji sledi dobije posebno značenje i da ga kompajler tretira drugačije. Takve situacije predstavljene su tabelom ispod:



System.out.println("Hello\tWorld");

System.out.println("HelloWorld\b\b\b\b\b");

System.out.println("Hello\nWorld");

System.out.println("Hello\rWorld");

System.out.println("\'HelloWorld\'");

System.out.println("\"HelloWorld\"");

System.out.println("This is backslash - \\");

Ovakav kod će na izlazu da proizvede ispise kao na slici ispod:

A screenshot of a computer

Description automatically generated

**Omotači primitivnih tipova**

U programskom jeziku Java, za svaki od 8 primitivnih tipova postoji po jedan složeni referentni tip. Takvi tipovi dodatno proširuju mogućnosti prostih tipova, njihovim kombinovanjem sa određenim funkcionalnostima. Zato što se sastoje iz odgovarajućeg prostog tipa i skupa pripadajućih funkcionalnosti, ovi tipovi se veoma često nazivaju i omotači prostih tipova.

A screenshot of a computer

Description automatically generated

Nazivi složenih tipova započinju velikim slovom.

Na prvi pogled, razlika između int i Integer tipova je samo u ključnoj reči. Ipak, već više puta je rečeno da složeni tipovi nisu ograničeni isključivo na vrednost, već da pored vrednosti mogu da poseduju i različita ponašanja. U to se možemo uveriti korišćenjem razvojnog okruženja, na način ilustrovan slikom ispod:

A screenshot of a computer program

Description automatically generated

Slozeni tipovi imaju ugradjene funkcionalnosti, kao sto prikazuje slika iznad.

Omotače prostih tipova je preporučljivo koristiti samo onda kada za tim stvarno ima potrebe, odnosno onda kada je neophodno koristiti neku od ugrađenih funkcionalnosti koje ovi tipovi imaju. U svim ostalim situacijama, najbolje je koristiti proste tipove, jer oni obezbeđuju neuporedivo bolje performanse.

**Promenljive**

Promenljive su kontejneri u koje se skladište vrednosti. Drugim rečima, nakon što se neka vrednost uskladišti u memoriji, ona biva reprezentovana kroz promenljivu. Jedna promenljiva tokom izvršavanja Java programa može da ima različite vrednosti i upravo se zbog toga ona i naziva promenljiva.

Osnovni oblik u kome se može javiti jedna promenljiva u Java jeziku izgleda ovako:

String message;

Na osnovu ove naredbe možemo da zaključimo da se promenljiva minimalno sastoji iz:

* tipa podatka koji će promenljiva moći da prihvati (u prikazanom primeru je to tip String, što znači da će promenljiva moći da prihvati podatke tekstualnog tipa);
* identifikatora koji predstavlja naziv promenljive (u primeru je to message).

**Deklarisanje i inicijalizovanje**

Da bi neka promenljiva postojala, mora se kreirati. Proces kreiranja promenljive naziva se **deklarisanje**. Deklaracija je zapravo sledeća naredba:

String message;

Deklaracija se sastoji iz dva dela. Prvi označava tip podatka koji će promenljiva predstavljati, a drugi predstavlja naziv promenljive.

Proces dodeljivanja vrednosti nekoj promenljivoj drugačije se naziva inicijalizovanje:

message = "Hello World";

Stoga prva naredba ilustruje deklaraciju, a druga inicijalizaciju:

String message; //declaration

message = "Hello World"; //initialization

Deklaracija i inicijalizacija se mogu obaviti i objedinjeno, odnosno u jednoj naredbi i to na sledeći način:

String message = "Hello World";

Promenljive su dobile naziv zbog mogućnosti redefinisanja vrednosti. To praktično znači da je vrednost jedne promenljive u svakom trenutku moguće promeniti:

String message = "Hello World";

message = "This is new message";

Ovo su sada dve naredbe. U prvoj naredbi se obavlja deklaracija i inicijalizacija promenljive sa nazivom message i vrednošću Hello World. U drugoj naredbi se obavlja promena vrednosti promenljive. Stoga nakon druge naredbe promenljiva message ima novu vrednost – This is new message.

Promenu vrednosti jedne promenljive je moguće obaviti proizvoljan broj puta, sve dok se poštuje njen tip. To znači da prilikom promene vrednosti String promenljive njoj nije moguće dodeliti vrednost nekog drugog tipa.

Unutar bloka u kom pišemo može postajati samo jedna promenljiva sa jednim nazivom.

**Imenovanje promenljivih**

Naziv promenljive je ono što se navodi nakon ključne reči kojom se definiše njen tip. Programer ima mogućnost da samostalno definiše naziv promenljive. Ipak, naziv promenljive je zapravo jezički element koji se naziva **identifikator**.

Pored neophodnih pravila, prilikom imenovanja promenljivih dobro je pridržavati se i nekih ustaljenih dobrih praksi, koje su formulisane kroz konvencije imenovanja. Jedna od najpopularnijih je CamelCase notacija, koja podrazumeva veliko slovo na početku svake reči u promenljivoj, osim eventualno prve:

MyVariable ili myFirstVariable

Kao što možete videti, sve reči od kojih se sastoji naziv promenljive pišu se sastavljeno, sa početnim velikom slovom, s tim što prvo slovo naziva promenljive može biti malo ili veliko.

**Ispis vrednosti promenljivih**

U skupu ugrađenih funkcionalnosti Java jezika nalazi se i nekoliko metoda koje omogućavaju ispis poruka na izlazu:

* println
* print
* printf

Do sada smo se već upoznali sa jednim od načina za ispis vrednosti iz Java programa. Naravno, reč je o metodi **println**. Evo kako se ona koristi:

int x = 5;

System.out.println(x);

Na izlazu dobijamo ispisanu vrednost promenljive x, a to je 5. Metoda println ispisuje poruku na izlazu i prelazi u novi red. U ovo se možemo uveriti ukoliko napišemo sledeći kod:

int x = 5;

int y = 10;

System.out.println(x);

System.out.println(y);

Vrednosti promenljivih će biti ispisane jedna ispod druge:

5  
10

Ukoliko je potrebno ispisati vrednosti u istom redu, može se koristiti metoda **print**:

int x = 5;

int y = 10;

System.out.print(x);

System.out.print(y);

Za rezultat daje:

510

Vrednosti su napisane jedna nakon druge, bez razmaka i prelaska u novi red. U tome je razlika između metoda print i println.

Još jedan način za ispis poruka na izlazu jeste korišćenje metode **printf**. Metoda printf omogućava kombinovanje predefinisanog teksta i vrednosti promenljivih i koristi se na sledeći način:

int x = 5;

System.out.printf("The value of variable x is %d", x);

Izvršavanjem ovakvog koda, dobija se sledeći ispis:

The value of variable x is 5

Evo kako je dobijen ovakav ispis. U okviru tekstualnog ispisa (koji se nalazi pod znacima navoda), bitno je da primetite sekvencu karaktera **%d**. U okviru finalnog ispisa, ova sekvenca karaktera je zamenjena vrednošću promenljive x. To je upravo i svrha ovakvog načina prikaza promenljivih na izlazu. Sekvenca karaktera %d označava da će na njenom mestu u okviru ispisa stajati celobrojna vrednost promenljive, koja je navedena nakon navodnika.

**Gde se nalaze metode za ispis vrednosti?**

U prethodnim redovima podrobnije smo se upoznali sa tri ugrađene funkcionalnosti Java jezika za ispis poruka unutar konzole/terminala. Za takve funkcionalnosti se kaže da su ugrađene, s obzirom na to da su one sastavni deo samog jezika, odnosno Java izvršnog okruženja. Sve one se nalaze unutar klase PrintStream koja objedinjuje funkcionalnosti za ispis podataka. Ova klasa sastavni je deo ugrađene klase System.

**Konverzija vrednosti**

Nešto ranije ste imali prilike da vidite da je vrednost jedne promenljive moguće menjati proizvoljan broj puta, sve dok se ostaje u okvirima tipa koji je definisan. U narednim redovima, moći ćete da vidite da pravilo striktnog poštovanja tipa nije u potpunosti isključivo i da je u nekim situacijama vrednost promenljive jednog tipa moguće dodeljivati promenljivoj koja je drugog tipa. Java virtuelna mašina u takvim situacijama samostalno obavlja konverziju tipova. Kako biste lakše razumeli o čemu je reč, analiziraćemo sledeću situaciju:

public class Main {

public static void main(String[] args) {

int x = 10;

double y = x;

System.out.println(y);

}

}

U prikazanom primeru, prvo je deklarisana i inicijalizovana jedna promenljiva tipa int. Njena vrednost je postavljena na 10. Zatim je deklarisana još jedna promenljiva (y) i njoj je dodeljena vrednost promenljive x. Iako su vrednosti različitih tipova, JVM u pozadini za nas obavlja konverziju. Upravo zbog toga se ovakva vrsta konverzije naziva **implicitna konverzija**.

**Implicitnu konverziju** je moguće obaviti isključivo između kompatibilnih tipova, odnosno kada se promenljivoj čiji je tip šireg opsega dodaje vrednost promenljive čiji je tip užeg opsega. U prikazanom primeru je upravo to i obavljeno. Tip int u memoriji zauzima 4 bajta, a tip double 8 bajtova. Upravo zbog toga, JVM može da pretvori vrednost int tipa u tip double bez ikakvih gubitaka. Obrnuti slučajevi implicitne konverzije nisu mogući osim uz neophodno naglasavanje.

**Eksplicitna konverzija** se obezbeđuje navođenjem tipa u koji želimo da izvršimo konverziju. Tip se navodi u zagradama ispred naziva promenljive čiji tip vrednosti konvertujemo:

double x = 10.5;

int y = (int)x;

Sada je ispred naziva promenljive x, unutar zagrada, postavljen tip int. Na taj način se postiže eksplicitna konverzija. Eksplicitna konverzija se često drugačija naziva kastovanje.

S obzirom na to da se u prikazanom primeru konvertuje vrednost koja u memoriji zauzima 8 bajtova (double) u vrednost koja u memoriji zauzima 4 bajta (int), očekivano je da dolazi do gubitka određenih informacija. U to se lako možemo uveriti ispisom vrednosti promenljive y, koja je sledeća:

10

Deo iza decimalne tačke je izgubljen, pa se ovakva vrsta konverzije često naziva i konverzija sa gubicima (lossy conversion).

Upravo prikazane konverzije je moguće obavljati samo između kompatibilnih vrednosti. Tako, na primer, nije moguće konvertovati boolean u double.

**Promenljive bez eksplicitno definisanog tipa**

U programskom jeziku Java moguće je obaviti kreiranje promenljive bez eksplicitnog definisanja njenog tipa. Za obavljanje takvog posla, umesto konkretnog tipa, koristi se posebna rezervisana reč – **var**.

var name = "Ben";

Ovo je primer deklaracije i inicijalizacije jedne promenljive bez eksplicitno definisanog tipa. U ovakvim situacijama, kompajleru se prepušta da samostalno utvrdi kojeg tipa je definisana vrednost. Upravo zbog toga se var može koristiti samo prilikom objedinjene deklaracije i inicijalizacije. Drugim rečima, ovako nešto ne bi bilo validno:

var name;

S obzirom na to da se Java kompajler oslanja na vrednost kako bi pogodio tip promenljive, upotreba reči var u naredbama bez inicijalizacije nije moguća.

Upotreba reči var posebno je korisna prilikom baratanja složenim tipovima podataka, kada može značajno da pojednostavi i ubrza proces pisanja izvornog koda.

**Napomena**

Reč var zapravo nije ključna reč u jeziku Java, već samo rezervisan naziv tipa podatka. Stoga je reč var moguće koristiti i kao identifikator za imenovanje promenljivih.

**Operatori**

Osnovna namena svakog kompjuterskog programa jeste da na neki način izmeni podatke koji se koriste tokom izvršavanja programa. Takva operacija se drugačije naziva obrada podataka, a osnovni mehanizmi obrade podataka zasnivaju se na upotrebi operatora.

**Šta su operatori?**

Operatori su posebni leksički elementi jezika, koji omogućavaju izvršavanje operacija nad vrednostima. Iako to možda niste znali, unutar primera iz prethodnih lekcija intenzivno je korišćen jedan operator:

int x = 10;

Ovo je primer deklaracije i inicijalizacije jedne promenljive. Za obavljanje inicijalizacije koristi se karakter jednako (=). Reč je o operatoru dodeljivanja. Operator dodeljivanja, kao što i samo ime sugeriše, vrši dodeljivanje vrednosti promenljivoj.

**Koji operatori postoje u Java jeziku?**

Java jezik poznaje nekoliko vrsta operatora:

* aritmetički operatori;
* operatori dodeljivanja;
* operatori poređenja;
* logički operatori.

**Aritmetički operatori**

Aritmetički operatori su oni koji izvršavaju aritmetičke operacije nad vrednostima.

A close-up of a person's face

Description automatically generated

**Sabiranje, oduzimanje, množenje i deljenje**

Četiri operatora standardnih aritmetičkih operacija, baš kao i u matematici, obavljaju sabiranje, oduzimanje, množenje i deljenje. Primer njihovog korišćenja može da izgleda ovako:

int x = 10;

int y = 5;

System.out.println(x + y);

System.out.println(x - y);

System.out.println(x \* y);

System.out.println(x / y);

U primeru je prvo obavljeno deklarisanje i inicijalizovanje dve celobrojne promenljive sa vrednostima 10 i 5. Zatim su nad njima upotrebljeni aritmetički operatori za sabiranje, oduzimanje, množenje i deljenje. Kod proizvodi sledeći izlaz:

15  
5  
50  
2  
Prilikom izvođenja aritmetičkih operacija, treba voditi računa o redosledu njihovog izvršavanja. Aritmetičke operacije u Javi imaju isti prioritet izvršavanja kao i u matematici. Množenje i deljenje imaju prednost nad sabiranjem i oduzimanjem. Redosled izvršavanja operacija može da se promeni upotrebom zagrada.

**Modulo (ostatak)**

Operator % koristi se za dobijanje celobrojnog ostataka pri deljenju dva broja:

int modulo = 10 % 3; // 10 = 3 \* 3 + 1

System.out.println(modulo);

Ovakav kod proizvodi sledeći rezultat:

1

Do ovakvog rezultata došlo se na sledeći način: najveći ceo broj koji se može dobiti deljenjem broja 10 brojem 3 jeste broj 3; 3 puta 3 je 9, što znači da do vrednosti 10 ostaje 1; 1 je ostatak, što je i rezultat koji se dobija od prikazanog koda.

**Operatori za uvećavanje i umanjenje vrednosti**

Posebna vrsta aritmetičkih operatora u Java jeziku jesu operatori koji se koriste za uvećavanje i umanjenje vrednosti. Njih karakteriše upotreba nad samo jednim [operandom](https://www.link-elearning.com/linkdl/opisPojma.php?id=160446), te se stoga često drugačije nazivaju unarni operatori.

A white background with black text

Description automatically generated

int x = 10;

x++;

Nakon izvršenja prikazanog koda, promenljiva x će imati vrednost 11.

Operator za uvećavanje je ekvivalent korišćenju aritmetičkog operatora sabiranja, na sledeći način:

int x = 10;

x = x + 1;

Operatori za uvećavanje i umanjivanje mogu se postavljati nakon, ali i ispred vrednosti, odnosno operanda. Tako nešto posebno je značajno kada se operatori uvećanja i umanjenja koriste unutar naredbe za dodeljivanje vrednosti. Evo pogledajte o čemu je reč:

int x = 10;

int y = x++;

Možda očekujete da će promenljiva y da dobije vrednost 11 (10+1), ali tako nešto neće biti slučaj. Promenljiva y će nakon izvršavanja prikazanog koda imati vrednost 10. Jednostavno, operator dodeljivanja ima prioritet, pa se dodeljivanje obavlja pre uvećavanja. Ukoliko želimo da se prvo obavi uvećanje, pa tek onda i dodela, potrebno je napisati sledeće:

int x = 10;

int y = ++x;

Operator uvećavanja sada je premešten pre promenljive čija vrednost se uvećava. Tako će uvećavanje biti obavljeno pre dodele i na kraju će promenljiva y imati vrednost 11.

**Operatori dodeljivanja**

Operatorima dodeljivanja obavlja se dodeljivanje neke vrednosti promenljivoj. Osnovni operator dodeljivanja jeste karakter jednako (=). Ipak, Java poznaje i nekoliko drugih operatora ovog tipa, koji predstavljaju zapravo kombinaciju operatora dodeljivanja i aritmetičkih operatora.

A close-up of a white background

Description automatically generated

**Operatori poređenja**

Java poseduje nekoliko operatora koje je moguće koristiti za poređenje dve ili više vrednosti. Poređenjem dolazi do stvaranja nove vrednosti, koja ukazuje na rezultat poređenja. Takva vrednost je logičkog tipa (boolean), što je jedan od prostih Java tipova.

Operatori poređenja koriste se u logičkim izjavama kako bi utvrdili istinitost nekog izraza.

A close-up of a pink box

Description automatically generated

Osnovni primer korišćenja operatora poređenja biće prikazan nad sledećim promenljivama:

int value1 = 1;

int value2 = 2;

Ovo su dve celobrojne promenljive (int tipa) sa vrednostima 1 i 2. Ukoliko bismo želeli da ispitamo njihovu jednakost, dovoljno bi bilo napisati nešto ovako:

System.out.println(value1 == value2);

U ovakvoj situaciji, na izlazu se dobija sledeći ispis:

false

Vrednosti 1 i 2 nisu jednake, te se zbog toga kao rezultat ovakvog poređenja dobija vrednost false.

Napomena:

Pazite da ne pomešate operatore **=** i **==**. Prvi operator je operator dodele, dok je drugi operator – operator poređenja. Greška uzrokovana pogrešnom upotrebom ova dva operatora može drastično izmeniti tok programa. Zato je veoma važno da do nje ne dođe.

Po identičnom principu se mogu koristiti i ostali operatori poređenja:

System.out.println(value1 != value2);

Ovo je sada primer u kome se ispituje nejednakost, pa se na izlazu dobija:

true

Vrednosti 1 i 2 nisu jednake, pa otuda vrednost true (istinito). Izraz bi mogao da se pročita ovako: Da li su vrednosti promenljivih value1 i value2 različite? Odgovor je, naravno, potvrdan, pa otuda i kao rezultat vrednost true.

Upotreba operatora < i > poznata je iz matematike:

System.out.println(value1 < value2);

System.out.println(value1 > value2);

Ovakav kod proizvodi sledeći efekat:

true  
false

Vrednost promenljive value1 jeste manja od vrednosti promenljive value2, pa je otuda prva vrednost **true**, a druga **false**.

Pored operatora veće i manje, u Javi, baš kao i u matematici, postoje operatori veće-jednako i manje-jednako:

int value1 = 2;

int value2 = 2;

System.out.println(value1 <= value2);

System.out.println(value1 >= value2);

Na izlazu se dobijaju sledeći rezultati:

true  
true

**Logički operatori**

Logički operatori se koriste kako bi utvrdili logičku povezanost između promenljivih, odnosno vrednosti. Najčešće se koriste u kombinaciji sa upravo prikazanim operatorima poređenja.

Logičkim operatorima se mogu kreirati različiti složeni logički izrazi, koji kao svoju finalnu vrednost uvek moraju imati true ili false.

A group of small dots

Description automatically generated with medium confidence

AND:

Analizom primera upotrebe logičkih operatora može se mnogo toga zaključiti. Operator AND zahteva ispunjenje svih navedenih uslova kako bi proizveo vrednost true. U protivnom, proizvodi vrednost false.

A close-up of a pink rectangle

Description automatically generated

OR:

Sa druge strane, operator OR zahteva ispunjenje samo jednog od uslova kako bi kao svoju finalnu vrednost proizveo true.

A white rectangular object with blue text

Description automatically generated

NOT:

Na kraju, operator NOT je operator negacije, koji true vrednost pretvara u false, a false u true.

A close-up of a pink rectangle

Description automatically generated

**Napomena:**

Logički operatori posebno važnu upotrebnu vrednost imaju prilikom formiranja uslova za kontrolu toka izvršavanja koda.

Evo jednog primera upotrebe logičkih operatora:

int value1 = 1;

int value2 = 2;

System.out.println((value1 == 1) && (value2 == 2));

System.out.println((value1 == 1) || (value2 == 2));

Deklarisane su dve promenljive i obavljena je njihova inicijalizacija vrednostima 1 i 2. Zatim su formulisana dva logička izraza. Rezultat je sledeći:

true  
true

Naravno, svaki od zasebnih uslova proizvešće vrednost true (pošto je 1=1 i 2=2), pa će i celokupni izrazi imati vrednosti true.

**Ternarni operator**

Ternarni operator je obavezni operator svakog popularnijeg jezika. Ni Java nije izuzetak. Ovaj operator je nešto teži za razumevanje, ali je u osnovi veoma jednostavan i posebno koristan kada se jednom ovlada njegovom sintaksom.

Ternarni operator se koristi za dodelu vrednosti promenljivoj, ali na osnovu nekog predefinisanog uslova. Sintaksa ternarnog operatora je sledeća:

myVariable = (condition) ? value1 : value2;

Napisano prostim jezikom, značenje navedene sintakse bi bilo: ukoliko je condition ispunjen, promenjivoj myVariable dodeli value1, a u protivnom value2.

Ternarni operator se gradi korišćenjem karaktera ? i :. Karakter upitnik (?) razdvaja uslov od mogućih vrednosti, a karakter dve tačke (:) razdvaja ponuđene vrednosti.

Primer upotrebe ternarnog operatora je sledeći:

public class Main {

public static void main(String[] args) {

int age = 15;

String message = (age < 18) ? "You can’t enter" : "Welcome";

System.out.println(message);

}

}

Ukoliko je vrednost promenljive age manja od 18, promenljiva message dobija vrednost You can’t enter. U protivnom, promenljiva message dobija vrednost Welcome.

Dakle, u okviru radnog okruženja možete promeniti vrednost promenljive age i uočiti kako se vrednost promenljive message menja na osnovu uslova koji je ispunjen.

**Nadovezivanje stringova**

Nešto ranije prikazani aritmetički operator za sabiranje (+) u jeziku Java ima još jednu funkciju. Ova funkcija se ogleda u mogućnosti [nadovezivanja](https://www.link-elearning.com/linkdl/opisPojma.php?id=160457) tekstualnih vrednosti ili konkatenacije stringova. Pogledajte primer:

String s1 = "String";

String s2 = "Value";

System.out.println(s1 + s2);

U primeru su definisane dve tekstualne vrednosti, a zatim su ove vrednosti sabrane kao da se radi o brojevima. Na izlazu se dobija:

  StringValue

Konkatenacija je mogla da se izvede i na druge načine, a da se dobije isti izlaz:

String s1 = "String";

System.out.println(s1 + "Value");

Ili:

System.out.println("String" + "Value");

Može se zaključiti da u kombinaciji sa String vrednostima operator sabiranja obavlja dodavanje jedne tekstualne vrednosti drugoj, čime se dobija nova tekstualna vrednost.

**If, else if, else**

Programi koje smo do sada kreirali nisu imali veliku mogućnost kontrole toka. To nas je i sputavalo u izvođenju nekih ozbiljnijih operacija.

**Šta je kontrola toka?**

Izvršavanje Java koda obavlja se sleva nadesno, pri čemu se napisane izjave (naredbe) izvršavaju redom, jedna za drugom:

int x = 1;

int y;

y = x;

U prikazanom primeru definisane su tri Java naredbe. One će se izvršiti redom kojim su napisane, pa će tako prvo biti deklarisana i inicijalizovana promenljiva x, zatim će biti deklarisana promenljiva y i na kraju će vrednost promenljive x biti dodeljena promenljivoj y.

Java omogućava različite načine za kontrolu toka:

* grananje (uslovno izvršavanje);
* petlje (ponavljanje).

**Grananje (uslovno izvršavanje)**

Grananje omogućava da se određene naredbe izvrše samo u slučaju zadovoljenja nekog uslova. Zbog toga je grananje usko povezano sa logičkim tipovima podataka i logičkim operatorima. Naime, Java omogućava da se kreira više mogućih grana izvršavanja, a da sam odabir grane izvršavanja bude obavljen u zavisnosti od neke kontrolne logičke vrednosti.

Java poznaje dve vrste kondicionalnih (uslovnih) naredbi:

* if..else grupa naredbi;
* switch.

**if naredba**

* Osnovna naredba grananja u Java jeziku jeste naredba if. Ova naredba postoji u praktično svim programskim jezicima, a svrha joj je oslikana u samom nazivu: ako.
* Reč if (ako), sama za sebe, nema baš mnogo logike u retorici, a tako je i u programiranju. Naime, pored ove ključne reči potrebno je postaviti i neki uslov od koga će zavisiti ishod ove naredbe. Tako opšta sintaksa if naredbe izgleda ovako:

if(condition) {

//code to execute if condition is met

}

Za formiranje if naredbe, koristi se ključna reč if, nakon koje se u zagradama navodi uslov. U slučaju ispunjenja uslova, izvršava se blok koda if naredbe.

A diagram of a code

Description automatically generated

Na slici iznad, izvršavanje programa dolazi do dela nazvanog condition (uslov). U ovom delu se proverava definisani uslov i ukoliko je on ispunjen, izvršava se kod if bloka. U protivnom, if blok koda se preskače, a izvršavanje se nastavlja prvom sledećom naredbom. Primer koji ilustruje upotrebu if naredbe je sledeći:

int speed = 9;

if (speed < 10) {

System.out.println("Too slow...");

}

Na početku je deklarisana i inicijalizovana promenljiva sa nazivom speed i vrednošću 9. Nakon ove linije, sledi if naredba. Uslov je definisan tako da se proverava da li je vrednost promenljive speed manja od 10. Ukoliko je ovakav uslov ispunjen, izvršava se blok koda između vitičastih zagrada, koji u primeru poseduje jednu naredbu.

Da je kojim slučajem vrednost promenljive speed bila veća ili jednaka broju 10, blok koda iz primera se ne bi izvršio.

**Logički uslovi i logičke vrednosti**

Nešto ranije je rečeno da se unutar zagrada nakon ključne reči if navodi uslov koji diktira da li će if blok biti izvršen ili ne. Ipak, ukoliko želimo da budemo potpuno precizni, može se reći da se unutar zagrada nakon ključne reči if navodi neka logička vrednost (true ili false):

if (true) {

System.out.println("Too slow...");

}

Naravno, ovakav blok koda uvek će se izvršiti, zato što je unutar zagrada navedena vrednost true. Tako se ovakav blok i ne može nazvati uslovnim, jer je verovatnoća njegovog izvršavanja unapred poznata. Stoga se direktno navođenje logičke vrednosti kao uslova grananja vrlo retko koristi.

Ipak, bitno je znati da se grananje u svojoj biti obavlja na osnovu logičkih vrednosti true i false, pa je i nešto ovako potpuno legitimno napisati. Identično važi i za sve ostale naredbe grananja, koje će biti prikazane u nastavku.

**Blokovi uslovnih naredbi**

U upravo prikazanom primeru mogli ste da vidite da se if naredba sastoji iz uslova i bloka. Blok se formira kao i svaki drugi blok u jeziku Java, korišćenjem vitičastih zagrada:

int speed = 9;

if (speed < 10) {

System.out.println("Too slow...");

}

**if...else naredba**

U ovom trenutku se može postaviti jedno pitanje:

Šta ukoliko je potrebno izvršiti određeni blok koda kada if uslov nije zadovoljen?

Odgovor na ovo pitanje krije se u upotrebi ključne reči else, pa tako naredba if postaje naredba if...else.

if...else je naredba za kontrolu toka koja omogućava definisanje logike koja će se izvršiti  ukoliko je uslov ispunjen, ali i logike koja će se izvršiti ukoliko uslov nije ispunjen. Sintaksa if...else naredbe je sledeća:

if (condition) {

statement\_1;

} else {

statement\_2;

}

U zagradama nakon ključne reči if navodi se uslov (condition). U slučaju ispunjenja ovakvog uslova, izvršava se prva izjava (statement\_1), a u protivnom druga (statement\_2).

A diagram of a code

Description automatically generated

U zavisnosti od ispunjenja uslova, izvršava se jedan od ova dva bloka koda. Bitno je razumeti da će jedan blok koda morati da se izvrši. Koji će to blok biti – zavisi od ishoda uslova.

Primer upotrebe if...else naredbe:

if (1 == 1) {

System.out.println("true");

} else {

System.out.println("false");

}

U primeru je definisan uslov 1==1. Potpuno je jasno da će uslov biti ispunjen, te se stoga izvršava if blok, a na izlazu ispisuje:

True

Veoma česta greška koja se javlja prilikom kreiranja uslovnih blokova je korišćenje operatora dodeljivanja (=) umesto operatora poređenja (==).

**if...else if...else naredba**

Šta se dešava ukoliko je potrebno definisati dodatne blokove koda sa sopstvenim uslovima?

Odgovor na ovo pitanje krije se u upotrebi ključnih reči else if.

else if omogućava da se prilikom kreiranja if...else naredbe definišu dodatni uslovi, pa tako naredba if...else postaje if...else if...else naredba.

Sintaksa ovakve naredbe je sledeća:

if (condition\_1) {

statement\_1;

} else if (condition\_2) {

statement\_2;

} else if (condition\_n) {

statement\_n;

} else {

statement\_last;

}

U ovom primeru, pored if i else blokova, postoje i dva else if bloka. Ukoliko je prvi uslov (condition\_1) ispunjen, izvršiće se izjava statement\_1. Ukoliko je uslov condition\_2 ispunjen, izvršiće se izjava statement\_2. Ukoliko je uslov condition\_n ispunjen, izvršiće se naredba statement\_n.

Na kraju, ukoliko nijedan uslov nije ispunjen, izvršiće se izjava statement\_last. Bitno je razumeti da broj else if blokova može biti proizvoljan. U primeru ih ima dva, dok je moguće imati bilo koji broj takvih blokova (jedan, dva, tri, četiri...).

A diagram of a computer code

Description automatically generated

Tok izvršavanja koda sa slike je sledeći:

* izvršavanje koda dolazi do prvog uslova (condition1);
* ukoliko je uslov condition1 ispunjen, izvršava se block1 i nakon toga se izlazi iz if…else if…else konstrukcije;
* ukoliko prvi uslov (condition1) nije ispunjen, ispituje se tačnost drugog uslova (condition2);
* ukoliko je uslov condition2 ispunjen, izvršava se block2;
* ukoliko nijedan od uslova nije ispunjen, izvršava se else blok koda.

**Ternarni operator**

O ternarnom operatoru već je bilo reči u lekciji o operatorima.  Tada je rečeno da je ternarni operator veoma koristan u situacijama kada je potrebno ostvariti kontrolu toka. Da budemo precizniji, korišćenjem ternarnog operatora moguće je promenljivoj dodeliti vrednost na osnovu nekog uslova:

String message = (age < 18) ? "You can't enter" : "Welcome";

A close-up of words

Description automatically generated

S obzirom na to da je ternarni operator već obrađen, njegov način funkcionisanja je jasan. Uslov je age < 18, a potencijalne vrednosti koje bi mogle da budu dodeljene promenljivoj message su You can’t enter i Welcome. Ukoliko je vrednost promenljive age manja od 18, message dobija vrednost You can’t enter.

Ukoliko je vrednost promenljive age 18 ili više, message dobija vrednost Welcome.

Ternarni operator se uvek može transformisati u if...else naredbu a to bi izgledalo ovako:

String message = "";

int age = 19;

if (age < 18) {

message = "You can't enter";

} else {

message = "Welcome";

}

System.out.println(message);

Rezultat prikazanog primera biće ispis vrednosti Welcome, zato što je vrednost promenljive age veća od 18.

**Switch**

switch je još jedna naredba za kontrolisanje toka koja postoji unutar Java jezika. Ova naredba omogućava kontrolisanje toka, baš kao što je to bio slučaj i sa if, else if i else naredbama. Ipak, switch to obavlja na nešto drugačiji način:

switch (expression) {

case label\_1:

statements\_1

break;

case label\_2:

statements\_2

break;

default:

statements\_def

break;

}

switch naredba poredi expression vrednost sa vrednostima case klauzula. U prikazanom primeru, vrednosti case klauzula su obeležene sa label\_1, label\_2 itd.

Kada se utvrdi podudaranje expression vrednosti sa nekom od case klauzula, izvršavanje programa se preusmerava na taj blok. U slučaju da se ne pronađu podudaranja, izvršava se opcioni default blok.

A diagram of a computer program

Description automatically generated

**Tip kontrolne vrednosti kod switcha**

U upravo prikazanom primeru ste mogli da vidite da je kontrolna vrednost koju switch dobija int tipa (reč je o vrednosti koja se navodi u zagradama nakon ključne reči switch i na osnovu koje se odlučuje koji će slučaj biti aktiviran). Nije obavezno da kontrolna vrednost bude int tipa. Tako je dozvoljeno korišćenje sledećih tipova: char, byte, short, int, Character, Byte, Short, Integer, String, enum. Svi navedeni tipovi su nam poznati, osim poslednjeg, enum tipa. O njemu će biti reči tokom izlaganja o naprednim osobinama Java jezika.

**Napomena**

Default blok koda se, po konvenciji, postavlja na kraj switch naredbe, ali to ne mora biti pravilo. Ovaj deo koda je moguće postaviti bilo gde unutar switch-a.

Svaki od switch slučajeva se mora završiti break naredbom.

Uloga ključne reči break je da izvede izvršavanje koda izvan switch naredbe. Kada izvršno okruženje naiđe na ovu naredbu, završava se izvršavanje switch naredbe. Ukoliko zaboravite da navedete ovu naredbu na kraju slučajeva switch konstrukcije, switch neće funkcionisati ispravno:

int x = 0;

switch (x) {

case 0:

System.out.println("zero");

case 1:

System.out.println("one");

break;

default:

System.out.println("unknown value");

break;

}

U ovakvoj situaciji, s obzirom na to da je vrednost promenljive x jednaka 0, očekivano je da se na izlazu dobije ispis zero. Ipak, ispis je sledeći:

zero  
one

Iako prikazani kod ne poseduje sintaksnu grešku, već se prevodi i izvršava, rezultat nije očekivan. Neočekivani rezultat smo dobili zato što smo zaboravili da na kraju prvog slučaja postavimo break naredbu. Zato Java izvršno okruženje, i pored toga što je vrednost x jednaka 0, ulazi u naredni slučaj i izvršava njegovu logiku (ispis poruke one). S obzirom na to da drugi slučaj na svom kraju poseduje break naredbu, izvršavanje se tu zaustavlja. Zbog ovakve osobine switch konstrukcije kaže se da svaki case na svom kraju mora imati break naredbu.

**Višestruko poklapanje kod switch naredbe**

Sta ukoliko je potrebno da se neki blok koda izvrši ukoliko x ima bilo koju od vrednosti 1 ili 2? U takvoj situaciji mora se pribeći definisanju višestrukih uslova.

Postavka primera će izgledati ovako:

Potrebno je napraviti switch strukturu koja će testirati vrednost promenljive x. Ako x ima vrednost 1 ili 2, na izlazu je potrebno ispisati „YES”, dok za svaku drugu vrednost promenljive x treba ispisati „NO”.

int x = 1;

switch (x) {

case 1:

case 2:

System.out.println("YES");

break;

default:

System.out.println("NO");

break;

}

Na ovaj način, prvi blok koda će se izvršiti kada promenljiva x ima vrednost 1 ili 2. Za sve ostale vrednosti, izvršava se default blok koda.

**Unapređeni switch**

Java jezik dodatno unapređuje switch naredbu, pa je nju moguće koristiti i na nešto drugačiji, kompaktniji način. Takav switch se naziva unapređeni switch.

Za početak ćemo videti kako se prvi primer switch konstrukcije iz ove lekcije može transformisati korišćenjem unapređenog switch-a. Takav primer je izgledao ovako:

int x = 1;

switch (x) {

case 0:

System.out.println("zero");

break;

case 1:

System.out.println("one");

break;

default:

System.out.println("unknown value");

break;

}

Takav kod se može unaprediti na sledeći način:

int x = 1;

switch (x) {

case 0 -> System.out.println("zero");

case 1 -> System.out.println("one");

default -> System.out.println("unknown value");

}

Napomena:

Unapređeni switch neće raditi u razvojnom okruženju, jer je to jedna od najnovijih pogodnosti Jave.

Čak i na prvi pogled, jasno je da je upravo prikazani kod znatno kompaktniji od onog iz prethodnog primera. Karakteri dve tačke (:), koji su razdvajali kontrolne vrednosti slučajeva od njihove logike, zamenjeni su karakterima **->**. Takođe, unapređeni switch nas oslobađa potrebe za definisanjem break naredbe na kraju svakog slučaja, što je ogromna prednost. Jednostavno, nešto ranije ste mogli da vidite da izostavljanje break naredbe može proizvesti velike probleme. Zbog toga je ovaj unapređeni način kreiranja switch-a velika pogodnost Java jezika.

**For, while, do while**

Petlje imaju nešto drugačije osobine nego uslovno izvršavanje (grananje). Da bi se njihove osobine na najbolji način razumele, biće razmotreno sledeće pitanje:

Na koji način se jedna ista naredba Java koda može izvršiti više puta?

Na primer, potrebno je pet puta ispisati jednu istu poruku. Najjednostavnije rešenje za postizanje opisanog ponašanja jeste navođenje pet sukcesivnih naredbi za ispis:

System.out.println("Hello World");

System.out.println("Hello World");

System.out.println("Hello World");

System.out.println("Hello World");

System.out.println("Hello World");

Željeni rezultat je postignut, ali šta da je bilo potrebno Hello World poruku ispisati 1000 puta? Navođenje 1000 identičnih naredbi je vrlo nepraktično i zamorno. Ono što je potrebno uraditi kako bi se nastali problem razrešio jeste iznaći način da se izvršnom okruženju kaže: izvrši naredbu za ispis poruke tačno 1000 puta. Upravo su petlje način da se jedna takva logika formuliše u programski kod Java jezika.

A screen shot of a black background

Description automatically generated

Sa slike se može zaključiti da u svetu programiranja petlje omogućavaju da se određeni deo koda izvrši više puta tako što se tok izvršavanja ciklično vraća unazad.

U osnovi svake petlje jeste proces ponavljanja. Svako ponavljanje se drugačije naziva **iteracija**. Tako su petlje zapravo sačinjene iz iteracija. Petlja započinje prvom iteracijom, a završava se nakon poslednje iteracije. Broj iteracija jedne petlje može biti proizvoljan, od nula do beskonačno. Petlja sa beskonačnim brojem iteracija nema svoj kraj, pa se drugačije naziva **mrtva petlja**.

**Petlje u Java jeziku**

Java poznaje nekoliko naredbi koje omogućavaju kreiranje petlji:

* for
* while
* do…while

Takođe, Java poseduje i dve naredbe za kontrolu izvršavanja petlji:

* break
* continue

**For petlja**

For petlja omogućava da se određeni blok koda izvrši tačan broj puta. U uvodnom delu ove lekcije postavljeno je pitanje:

Kako ispisati jednu istu poruku određeni broj puta?

Odgovor na ovo pitanje leži upravo u korišćenju for petlje:

for (int i = 0; i < 5; i++) {

System.out.println("Hello World");

}

Da bismo razumeli kako je ovako nešto postignuto, neophodno je upoznati se sa sintaksom for petlje:

for (initialization; condition; final-expression)

statement

A diagram of a code

Description automatically generated

For petlja kreira se korišćenjem ključne reči for. Nakon ove ključne reči, u zagradama se navode tri izjave (naredbe):

1. **initialization (inicijalizacija)** – početna naredba koja se izvršava samo jednom, na početku izvršavanja petlje; koristi se za inicijalizaciju jednog ili više brojača petlje; u prikazanom primeru to je int i = 0;
2. **condition (uslov)**– naredba kojom se utvrđuje istinitost uslova; ukoliko je uslov ispunjen, prelazi se na izvršavanje tela petlje; ovo je naredba koja se izvršava pre svake iteracije; u prikazanom primeru je to i < 5;
3. **final-expression** – naredba koja se izvršava nakon svake iteracije; uglavnom se koristi za uvećanje ili smanjenje vrednosti brojača; u primeru je to i++

Nakon zagrada u kojima se nalaze tri upravo opisane naredbe, sledi telo petlje koje započinje otvorenom, a završava se zatvorenom vitičastom zagradom. Unutar bloka petlje može se naći proizvoljan broj naredbi koje se ponavljaju svakom iteracijom.

Baš kao i kod uslovnih blokova, vitičaste zagrade za formiranje bloka for petlje je moguće izostaviti ukoliko telo petlje poseduje samo jednu naredbu.

A computer code with text and symbols

Description automatically generated

**Promenljiva koja predstavlja brojač postoji samo unutar for petlje**

Svaka promenljiva u Java jeziku poseduje nešto što se naziva oblast vidljivosti. Ovaj pojam se odnosi na sva ona mesta u programskom kodu sa kojih je jednoj promenljivoj moguće pristupiti. Blokovi, kao sintaksni elementi jezika, uvek predstavljaju jednu oblast vidljivosti.

To praktično znači da se promenljive deklarisane unutar for petlje mogu koristiti samo unutar bloka petlje, a ne i izvan takvog bloka. To pravilo važi i za promenljivu koja predstavlja brojač, zato što je i ona primer promenljive koja se deklariše unutar bloka for petlje:

for (int i = 0; i < 5; i++) {

System.out.println("Hello World");

}

System.out.println(i);

Takođe, nije neophodno da se brojač uvećava. Njega je u svakoj iteraciji moguće i umanjivati:

for (int i = 5; i > 0; i--) {

System.out.println("Hello World");

}

**Petlja unutar petlje**

Jednu for petlju je moguće smestiti unutar druge for petlje i na taj način dobiti jednu zanimljivu strukturu. Takav pristup se veoma često koristi u kompleksnijim programima.

Evo kako izgleda jedna struktura koja podrazumeva petlju unutar petlje:

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 3; j++) {

System.out.println(j);

}

}

Pogledajte primjer ispod:

public class Main {

public static void main(String[] args) {

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 3; j++) {

System.out.println(j);

}

}

}

}

Najlakše je da ovakvu strukturu posmatrate iznutra, odnosno počevši od unutrašnje petlje. Ona ima brojač imenovan nazivom j i unutar tela petlje se obavlja ispis vrednosti brojača. Ova unutrašnja petlja će se izvršiti tri puta, što znači da će na izlaz ispisati vrednosti 0, 1, 2. Kada ne bi bilo spoljašnje for petlje, to bi bio i jedini ispis koji bismo dobili od ovakvog programa. Ipak, spoljašnja petlja čini da se izvršavanje unutrašnje petlje ponovi četiri puta, pa se na kraju na izlazu dobija sledeće:

0  
1  
2  
0  
1  
2  
0  
1  
2  
0  
1  
2

Bitno je da primetite da svaka petlja poseduje brojač sa jedinstvenim nazivom. S obzirom na to da unutrašnja petlja pripada bloku spoljašnje petlje, neophodno je da se za brojač koristi promenljiva jedinstvenog naziva.

**while petlja**

Pored for petlje, Java poznaje još nekoliko načina da se određeni kod izvrši više puta. Jedan od njih jeste while petlja.

while petlja izvršava određeni blok koda sve dok je ispunjen uslov za izvršavanje petlje. To praktično znači da ova vrsta petlje nema unapred definisan broj iteracija. Sintaksa while petlje je sledeća:

while (condition) {

statement

}

Sve dok je uslov (condition) ispunjen, izvršava se blok koda, odnosno telo petlje.

A diagram of a code block

Description automatically generated

Primer upotrebe while petlje je sledeći:

public class Main {

public static void main(String[] args) {

int x = 0;

while (x < 5) {

System.out.println("Hello World");

x = x + 1;

}

}

}

Na početku se deklariše i inicijalizuje promenljiva x, sa vrednošću 0. while petlja se kreira korišćenjem ključne reči while, a nakon nje, unutar zagrada se definiše uslov:

x<5

Ovo praktično znači da će se petlja izvršavati sve dok je vrednost promenljive x manja od 5.

Telo petlje definisano je vitičastim zagradama. Unutar tela petlje vrši se ispisivanje poruke Hello World, ali i uvećavanje vrednosti promenljive x za jedan. Da kojim slučajem ova inkrementacija nije navedena, uslov za izvršavanje ove petlje uvek bi bio ispunjen. Na taj način bi se stvorila jedna **beskonačna petlja**, odnosno petlja koja nikada ne bi prestala da se izvršava.

Upravo prikazanim primerom while petlje dobija se identičan efekat kao i nešto ranije, kada smo koristili for petlju:

Hello World  
Hello World  
Hello World  
Hello World  
Hello World

**do…while petlja**

Upravo opisana while petlja ni na koji način ne garantuje da će se njeno telo uopšte i izvršiti. Jednostavno, ukoliko uslov nije ispunjen, telo petlje se neće izvršiti nijednom:

int x = 6;

while (x < 5) {

System.out.println("Hello World");

x = x + 1;

}

U primeru, vrednost promenljive x je veća od 5, tako da uslov za izvršavanje while petlje nije ispunjen, te se kod unutar tela while petlje neće izvršiti nijednom. Ali šta ukoliko je potrebno osigurati da se petlja izvrši makar jednom, bez obzira na ispunjenje uslova?

Tako nešto moglo bi se postići ukoliko bi se provera ispunjenosti uslova smestila nakon bloka koda koji je potrebno izvršiti. Upravo to je osnovna osobina petlje do…while.

Sintaksa do…while petlje je sledeća:

do {

// statements

} while (condition);

Na početak se postavlja ključna reč **do**, a nakon nje blok koda koji je oivičen vitičastim zagradama. Nakon bloka, koristi se ključna reč **while** za definisanje uslova izvršenja petlje.

A diagram of a code

Description automatically generated

Sada se prethodni primer može transformisati na sledeći način:

int x = 6;

do {

System.out.println("Hello World");

x = x + 1;

} while (x < 5);

Rezultat je sledeći:

Hello World

Zaključak je da se telo petlje izvršilo jednom i pored toga što uslov nije ispunjen. Provera ispunjenosti uslova se obavlja nakon izvršavanja svake iteracije, te stoga do...while petlja uvek poseduju makar jednu iteraciju.

**Naredbe break i continue**

Ponekad je potrebno kontrolisati izvršavanje petlje na osnovu nekog uslova i to unutar tela petlje. U takvim situacijama je moguće koristiti naredbe break i continue.

Naredba continue prekida aktuelnu iteraciju i prelazi na sledeću.

Naredba break napušta petlju.

Sledeći primer ilustruje upotrebu naredbe continue:

int i;

for (i = 0; i < 100; i++) {

if (i > 50) {

continue;

}

System.out.println(i);

}

System.out.println(i);

U primeru je kreirana jedna for petlja, koja će imati 100 iteracija (od 0 do 99). Ipak, unutar for petlje je definisan jedan uslov, kojim se proverava da li je brojač veći od 50. Ukoliko je brojač veći od 50, poziva se naredba continue. S obzirom na to da ova naredba odmah prelazi na izvršavanje sledeće iteracije, linija za ispis vrednosti brojača se neće izvršiti u slučaju da je brojač veći od 50. Sve ovo proizvodi sledeći rezultat:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
100

Analizom izlaza može se utvrditi da se linija za ispis vrednosti brojača unutar tela petlje izvršava do vrednosti 50. Petlja nastavlja da se izvršava do kraja, ali ne ispisuje vrednosti, u šta se možemo uveriti poslednjim ispisanim brojem. To je broj 100, koji je ispisan od strane linije koja se nalazi izvan petlje.

Ovo praktično znači da je vrednost brojača nakon završetka petlje 100, što dokazuje da se petlja izvršila u celosti.

Za razliku od naredbe continue, koja prekida trenutnu i prelazi na sledeću iteraciju, naredba break kao svoj efekat ima izlazak iz petlje. Drugim rečima, naredba break završava petlju. Stoga pogledajte šta će se dogoditi ukoliko ključnu reč continue zamenimo ključnom rečju break:

int i;

for (i = 0; i < 100; i++) {

if (i > 50) {

break;

}

System.out.println(i);

}

System.out.println(i);

Ovoga puta, kod proizvodi sledeći izlaz:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
51

Jasno se može videti razlika između naredbi continue i break. Kada brojač dostigne vrednost 51, aktivira se if uslovni blok unutar koga se nalazi naredba break. Izvršavanjem naredbe break dolazi do trenutnog izlaska iz petlje, te se prelazi na izvršavanje sledeće naredbe. Zbog toga je poslednja vrednost koja se štampa unutar for petlje 50. Nakon izvršavanja for petlje, štampa se vrednost 51.

**Napomena**

U primerima efekata naredbi break i continue, brojači su namerno deklarisani izvan for petlji, kako bi se njihova vrednost očuvala i nakon završetka petlji. U realnim okolnostima, kao što je već rečeno nešto ranije, ovo se smatra lošom praksom.

**Klase i objekti**

Java je objektno orijentisan programski jezik. To praktično znači da su klase i objekti njegova centralna figura.

**Šta su klase i objekti?**

Objekti su način da se unutar Java programa predstave podaci po uzoru na pojmove iz realnog sveta. Na primer, ukoliko unutar Java programa želimo da predstavimo podatke studenata, svaki student bi predstavljao jedan objekat u našem programu. Po istom principu, ukoliko bismo želeli unutar našeg programa da predstavimo podatke različitih automobila, svaki pojedinačni automobil bi bio jedan objekat. Stoga se može reći i to da je objekat instanca, odnosno jedan primerak nečega.

A screenshot of a computer

Description automatically generated

Objekti se kreiraju na osnovu određenog šablona, odnosno modela. U objektno orijentisanom programiranju takvi šabloni se drugačije nazivaju **klase**.

Klasa je šablon, odnosno model na osnovu koga nastaju objekti. Na osnovu jedne klase se može stvoriti proizvoljan broj objekata, pri čemu svaki objekat može imati različite osobine. Tako su svi objekti sa slike automobili, ali automobili različitih osobina. Jedan se zove Renault 4L i plave je boje, drugi se zove Geely TX4 i ima žutu boju, dok je treći crni Ford Crown Victoria.

**Kreiranje klasa**

U objektno orijentisanom programiranju, klasa se može doživeti kao kalup, a objekti kao kolači koji se na osnovu takvog kalupa izlivaju. Kao što nije moguće napraviti takav kolač bez kalupa, ni objekat se ne može kreirati bez klase.

Klasa se kreira na sledeći način:

class Car {

}

U programskom jeziku Java, klase se kreiraju korišćenjem ključne reči **class**, nakon koje sledi naziv klase. U prikazanom primeru, naziv klase je **Car**. Ključna reč class i naziv klase zajednički se nazivaju **zaglavlje klase**. Nakon zaglavlja, sledi **telo klase**, koje je oivičeno vitičastim zagradama. Tako je telo Java klase jedna vrsta bloka koda u sintaksnoj hijerarhiji jezika.

A black background with text and arrows

Description automatically generated

**Klase u dosadašnjem toku ovog kursa**

Prilikom kreiranja našeg prvog Java programa, bili smo u obavezi da kreiramo jednu Java klasu. Takođe, u prethodnim lekcijama koristili smo i neke klase koje su ugrađene u sam Java jezik – String, Integer...

**Osnovni klasni članovi**

Telo upravo kreirane klase je prazno, zato što između otvorene i zatvorene vitičaste zagrade nema ničega. Unutar tela klase, može se naći izvorni kod kojim se kreiraju dva osnovna tipa njenih članova:

* svojstva (polja);
* metode

**Svojstva** su zapravo promenljive koje se deklarišu direktno unutar klasa:

class Car {

String make;

String model;

}

Unutar klase Car sada smo definisali dva svojstva. To su zapravo dve promenljive tipa String, unutar kojih će se čuvati informacije o proizvođaču i modelu automobila.

Pored svojstava, unutar klasa se mogu naći i metode. Metodama se definišu funkcionalnosti koje su vezane za namenu klase:

class Car {

String make;

String model;

void startEngine(){

System.out.println("Engine started...");

}

}

Unutar klase Car sada je dodata i jedna metoda. Iz sintaksnog ugla jezika, reč je o jednom bloku, unutar koga se objedinjuje kod koji treba da obavi neku operaciju. Stoga je upravo kreirana metoda slikovita – ona treba da pokrene motor jednog automobila.

**Kreiranje objekata**

Sada kada imamo klasu koju smo samostalno definisali, nju možemo iskoristiti za kreiranje objekata. Jedan objekat na osnovu klase Car se može kreirati na sledeći način:

Car myCar = new Car();

Upravo prikazanu naredbu je potrebno da postavite unutar main metode Java programa. Korišćenjem ovakve naredbe obavlja se kreiranje objekta. Objekat se kreira upotrebom ključne reči **new**, nakon koje se navode naziv klase i zagrade.

Prikazanom naredbom zapravo se obavlja deklarisanje i inicijalizovanje jedne promenljive. Stoga je prikazanu naredbu bilo moguće razdvojiti na dva dela:

Car myCar;

myCar = new Car();

Ovo je sada primer koji podrazumeva odvojenu deklaraciju i inicijalizaciju jedne promenljive preko koje će u programu biti dostupan kreirani objekat.

**Korisnički definisani tipovi podataka**

Bitno je da primetite razliku između upravo prikazanog primera deklaracije promenljive i onih iz prethodnih lekcija ovog kursa. Naime, na mestu na kome se definiše tip promenljive stoji reč Car. Iz ovoga možemo da zaključimo da je tip promenljive myCar – Car. To na kraju znači da smo kreiranjem klase Car prvi put obavili kreiranje jednog korisnički definisanog tipa. Kada se za neki tip kaže da je korisnički definisan, to znači da on ne postoji kao deo Java jezika. Drugim rečima, mi smo ga samostalno kreirali. Tako se može reći da u Javi postoje dve vrste tipova:

* ugrađeni (int, float, String, Integer...);
* korisnički (programer ima mogućnost da samostalno kreira proizvoljan broj sopstvenih tipova).

Proces kreiranja objekata drugačije se naziva **instanciranje klase**. Instanciranjem klase kreira se jedna njena instanca, odnosno objekat, koji u kodu postaje dostupan kroz promenljivu. Na osnovu jedne klase moguće je napraviti proizvoljan broj objekata:

Car myCar = new Car();

Car myCar2 = new Car();

Ovo su sada dva objekta kreirana instanciranjem jedne iste klase. Na identičan način je moguće kreirati proizvoljan broj objekata. Zajedničko za sve kreirane objekte je da unutar njih postoje svojstva i metode koji su definisani unutar klase. Kreiranim svojstvima i metodama je moguće pristupiti navođenjem naziva promenljive i karaktera tačka.

S obzirom na to da su svojstva zapravo promenljive koje se definišu unutar klasa, njima je moguće operisati korišćenjem pristupa koji su već ilustrovani:

Car myCar = new Car();

myCar.make = "Honda";

myCar.model = "Accord";

Ovakvim kodom definisane su vrednosti svojstava make i model objekta koji je kreiran na osnovu klase Car. Na ovaj način, objekat jednog automobila u našem programu počinje da dobija svoj oblik.

Drugi objekat koji je kreiran na osnovu klase Car može se konfigurisati na sledeći način:

Car myCar2 = new Car();

myCar2.make = "Lexus";

myCar2.model = "LS500";

Naš drugi objekat kreiran na osnovu klase Car poseduje drugačije vrednosti svojstava make i model. Na ovaj način smo dobili dva objekta koji imaju isti tip (Car), ali se koriste da u našem programu predstave dva različita automobila.

**Složeni tipovi podataka**

Svaka klasa je primer složenog tipa podataka, zato što unutar sebe može da objedini veliki broj pojedinačnih vrednosti (svojstava) i ponašanja (metoda).

**Nasleđivanje**

Jedna od osnovnih osobina objektno orijentisanog programiranja jeste i nasleđivanje. Nasleđivanje je pojam koji omogućava da jedna klasa nasledi osobine od neke druge klase. Upravo je nasleđivanje „krivac” zbog koga je pomoću promenljive. moguće pristupiti određenim klasnim elementima koje nismo kreirali u klasi Car. Svi takvi elementi potiču iz osnovne klase Java jezika. Reč je o klasi Object.

Sve klase u Java programskom jeziku, bez obzira na to da li je reč o ugrađenim klasama ili klasama koje mi samostalno kreiramo, nasleđuju klasu Object. Upravo to je razlog zbog koga objekti naše klase Car poseduju elemente koje mi nismo eksplicitno definisali u samoj klasi.

**Metode**

Pored svojstava, metode su takodjer osnovni klasni članovi. Metode su način za grupisanje određene funkcionalnosti u jednu celinu. Stoga se metode u drugim programskim jezicima često nazivaju i funkcije. Ipak, zbog objektne orijentacije Java jezika, funkcije imaju naziv metode, što označava da se one nalaze unutar klasa. Drugim rečima, u Javi nije ni moguće napraviti funkciju/metodu koja se nalazi izvan neke klase. U jezicima u kojima je to moguće, sve funkcionalnosti su funkcije, a one unutar klasa se nazivaju metode.

Metoda je specijalni blok koda, zadužen za izvršenje određene logike. Metoda prilikom izvršavanja logike može da koristi određene vrednosti, koje se nazivaju **ulazni parametri**, a može i emitovati svoj rezultat kao **povratnu vrednost**.

A green rectangular object with black text

Description automatically generated

Na slici je slovom x označen ulazni parametar, odnosno vrednost koju metoda dobija na obradu. Nakon obrade, metoda emituje rezultat izvršavanja. Ipak, nije obavezno da metoda ima ulazne i uzlazne parametre.

U nešto ranije kreiranoj Java klasi, prvoj koju smo kreirali, nalazila se i jedna metoda:

void startEngine() {

System.out.println("Engine started...");

}

Ovo je primer metode koja nema ulaznih ni izlaznih parametara. Sve što ona obavlja jeste ispis jedne poruke na izlazu. Ključna reč **void** označava da metoda nema povratnu vrednost, a na osnovu praznih zagrada nakon njenog naziva može se zaključiti da ona nema ulaznih parametara.

A computer screen shot of words

Description automatically generated

Sa slike možete da vidite osnovnu strukturu metoda u Java jeziku. Metode se sastoje iz potpisa i tela. Potpis obuhvata:

* tip povratne vrednosti (u primeru void);
* naziv metode (u primeru startEngine);
* eventualne ulazne parametre koji se navode unutar zagrada, nakon naziva metoda (u primeru ()).

Telo metode se definiše upotrebom vitičastih zagrada. Unutar njih se definiše logika metode.

Kako bi se aktivirao kod koji se nalazi unutar neke metode, nju je potrebno pozvati. **Poziv metode** obavlja se navođenjem njenog naziva i zagrada.

Evo kako može izgledati jedna metoda koja poseduje i ulazne i izlazne parametre:

double kwToHp(int kw) {

double result = kw / 0.745699872;

return result;

}

A computer screen with text and arrows

Description automatically generated

Ovo je metoda koja je namenjena konverziji snage motora iz kilovata u konjske snage. Metoda prihvata jedan parametar tipa int (kw), što je zapravo snaga motora u kilovatima. Ulazni parametar možete videti unutar zagrada, nakon naziva metode. Unutar tela metode, obavlja se računanje vrednosti, koja se postavlja za vrednost promenljive result. Na kraju se vrednost takve promenljive emituje kao povratna. Emitovanje povratne vrednosti se postiže upotrebom ključne reči return, dok je tip povratne vrednosti definisan unutar potpisa metode.

Ovakva metoda je mogla biti jednostavnije formulisana na sledeći način:

double kwToHp(int kw){

return kw / 0.745699872;

}

Nad promenljivom kojom se predstavlja jedan objekat automobila pozvana je metoda kwToHp i tom prilikom joj je prosleđena vrednost 147. Može se reći da je 147 ulazni parametar ove metode. Ovakav kod će se izvršiti i metoda kwToHp će svoj rezultat isporučiti kao povratnu vrednost.

**Konstruktori**

Nešto ranije ste videli na koji način se obavlja instaciranje klase:

Car myCar = new Car();

Do sada je rečeno da instanciranje podrazumeva upotrebu ključne reči new, ali se do sada nismo posebno bavili onim što dolazi nakon ove ključne reči. Nakon ključne reči možete videti nešto što podseća na poziv metode čije je ime Car. Zapravo, deo nakon ključne reči new i jeste poziv metode, ali jedne specijalne metode koja se naziva konstruktor.

A black background with text and arrows

Description automatically generated

Konstruktor je specijalna metoda unutar klase, koja se poziva prilikom kreiranja objekata. Svaka klasa može imati jedan ili više konstruktora. Drugim rečima, ne postoji klasa bez konstruktora. Možda se u ovom trenutku pitate zbog čega onda u klasi koju smo kreirali ne postoji konstruktor.

Odgovor je vrlo jednostavan: kada se konstruktor ne napiše eksplicitno, Java kompajler kreira konstruktor koji se naziva podrazumevani. Reč je zapravo o metodi koja ima isto ime kao i klasa, nema ulaznih ni izlaznih parametara, a telo joj je prazno. Takav konstruktor izgleda ovako:

class Car {

Car() {

}

}

Ukoliko objekte kreiramo na način na koji je to obavljeno u jednom od prethodnih poglavlja, nema nikakve potrebe da konstruktor samostalno definišemo. Ipak, ukoliko želimo da na neki način prilagodimo proces kreiranja objekata, moguće je samostalno definisati konstruktor:

class Car {

String make;

String model;

Car(String \_make, String \_model) {

make = \_make;

model = \_model;

}

}

Klasa Car sada poseduje dva konstruktora. Jedan je bez parametara i bilo kakve logike, dok drugi, kao i u prethodnom primeru, prihvata dva parametra. U ovakvim situacijama je moguće birati način za obavljanje instanciranja:

Car myCar = new Car("Honda", "Accord");

Car myCar2 = new Car();

Obe naredbe za kreiranje objekata su validne, zato što klasa Car sada poseduje dva konstruktora.

**Overloading**

Upravo prikazani slučaj postojanja dva konstruktora sa različitim ulaznim parametrima primer je osobine Java jezika koja se naziva overloading. Overloading je zapravo mogućnost preklapanja metoda, odnosno postojanja metoda sa istim nazivom, ali različitim sklopom ulaznih parametara.

**This**

Nešto ranije, prilikom samostalnog definisanja konstruktora sa parametrima, napisali smo ovako nešto:

class Car {

String make;

String model;

Car(String \_make, String \_model) {

make = \_make;

model = \_model;

}

}

Obratite pažnju na to da smo parametre koje prihvata konstruktor imenovali sa početnim karakterom donja crta, kako bi se njihova imena razlikovala od svojstava definisanih unutar klase. Svakako, bilo bi najbolje da i oni imaju identičan naziv:

Car(String make, String model) {

make = make;

model = model;

}

Iako ovakav kod ne bi proizveo sintaksu grešku, efekat ne bi bio očekivan. Na ovaj način uopšte i ne bi bile postavljene vrednosti klasnih svojstava make i model, zato što se u ovakvoj situaciji nazivi make i model odnose na parametre konstruktora. Kako bi kod obavio ono što želimo bez izmene naziva parametara, neophodno je da iskoristimo jednu do sada nepoznatu mogućnost jezika. To je ključna reč **this**.

Ključna reč this omogućava pristup objektu unutar metoda i konstruktora klase. Izvan klase, kao što ste već mogli da vidite, nakon njenog instanciranja, objekat se reprezentuje kroz promenljivu. Ipak, ukoliko je to isto potrebno uraditi unutar klase, pribegava se korišćenju ključne reči this.

Ključna reč this nam omogućava i da učinimo da kod iz našeg konstruktora obavlja ono što smo zamislili:

Car(String make, String model) {

this.make = make;

this.model = model;

}

Promenljive sa istim nazivima sada se razlikuju pomoću ključne reči this. Svojstvima make i model pristupili smo pomoću ključne reči this, na identičan način kao i korišćenjem promenljive izvan klase.

**Prosti i složeni tipovi i čuvanje unutar memorije**

S obzirom na to da u ovom trenutku znamo što su to prosti, ali i složeni tipovi u Javi, možemo da govorimo o načinu na koji se oni smeštaju u memoriju. Naime, u jednoj od prethodnih lekcija u kojima je bilo reči o tipovima podataka, rečeno je da se složeni tipovi drugačije nazivaju i **referentni**. Kako biste razumeli odakle dolazi ovaj pojam referentni, neophodno je da se pozabavimo načinom na kojim se prosti i složeni tipovi smeštaju u memoriju.

Svaki podatak, bez obzira na to da li je prost ili složen, zauzima određeno mesto u memoriji. Ipak, prosti i složeni tipovi se smeštaju u različitim delovima radne memorije. Prosti tipovi se smeštaju na statički deo memorije, koji se naziva **Stack**, gde zauzimaju svoj prostor prilikom aktivacije programa. Sa druge strane, složeni tipovi se smeštaju u dinamičkom memorijskom delu, koji se naziva **Heap.** Unutar Stacka se čuva samo adresa koja pokazuje na njihovu stvarnu vrednost u dinamičkoj memoriji.

A diagram of a stack

Description automatically generated

Slika ilustruje uprošćen prikaz memorijske strukture programa unutar koga su definisane tri vrednosti. Dve su objekti, dok je treća celobrojna numerička vrednost. U Java programu, takve vrednosti bi mogle da izgledaju ovako:

Car myCar = new Car();

int myNumber = 16;

Car myCar2 = new Car();

Sa slike možete videti da je prosta vrednost (16) direktno smeštena unutar statičkog dela memorije (Stack). Kod složenih tipova, na statičkom delu memorije smešta se samo referenca koja pokazuje na stvarnu vrednost koja je uskladištena unutar dinamičkog dela memorije, koji se naziva Heap. Upravo zbog ove činjenice se složeni tipovi nazivaju referentni, a za promenljive kaže da čuvaju samo referencu do objekta a ne stvarnu vrednost.

Razlog ovoga leži u različitoj veličini memorije koju primitivni i referentni tipovi mogu da zauzmu. Svaki primitivni tip ima unapred definisanu količinu memorije koju zauzima u memoriji. To ste, uostalom, mogli da vidite u jednoj od prethodnih lekcija, u kojoj je bilo reči o prostim tipovima. Ipak, tako nešto nije slučaj kod složenih tipova. S obzirom na to da oni mogu biti sačinjeni iz proizvoljnog broja prostih vrednosti i ponašanja, količina potrebne memorije za njihovo čuvanje nije unapred poznata.

Upravo zbog toga se pribegava čuvanju složenih tipova unutar dinamičkog dela memorije. Java virtuelna mašina poseduje kompleksne mehanizme za rukovanje takvom vrstom memorije, što na kraju omogućava čuvanje vrednosti čija veličina nije unapred poznata – i to sve bez ikakve intervencije programera. Zbog toga se kaže da Java jezik poseduje ugrađene mehanizme za rukovanje memorijom koji programera oslobađaju potrebe za samostalnom [alokacijom](https://www.link-elearning.com/linkdl/opisPojma.php?id=160488) i oslobađanjem memorije.

**Paketi**

Paketi se u Java jeziku koriste za grupisanje većeg broja srodnih klasa, odnosno za uspostavljanje određene vrste hijerarhijske strukture. Uopšteno govoreći, hijerarhijska struktura se koristi kako bi se elementi organizovali u skladu sa relacijama i vezama koje postoje između takvih elemenata. Dobar primer takve hijerarhijske strukture je [fajl sistem](https://www.link-elearning.com/linkdl/opisPojma.php?id=160461).

Fajl sistem predstavlja hijerarhijsku organizaciju foldera u okviru nekog skladišta podataka. Na taj način se olakšava snalaženje u okviru velike količine podataka. Na isti način se može razmišljati i o paketima u Javi. Oni predstavljaju sistem za tipiziranu klasifikaciju klasa.

**Podrazumevani paket**

Svaka Java klasa mora pripadati nekom paketu. U skladu sa tim, podrazumevani paket se koristi za klase koje nisu eksplicitno dodate unutar nijednog paketa, baš kao što je to slučaj u primeru iz prethodne lekcije. Sve klase koje se nalaze direktno unutar src foldera pripadaju podrazumevanom paketu.

Iako je potpuno legitimno imati klase unutar podrazumevanog paketa, savetuje se kreiranje sopstvenog, pa čak i onda kada nam je potreban samo jedan paket i kada naš projekat ima mali broj klasa.

**Kreiranje paketa**

Paketi nisu ništa drugo do folderi na fajl sistemu.

Korišćenjem IntelliJ IDEA razvojnog okruženja, paketi se kreiraju slično kao i klase. Iz kontekstnog menija koji se dobija desnim klikom na folder src, potrebno je odabrati opciju **New -> package**

Paketi se, po nepisanom pravilu, imenuju korišćenjem malih slova, bez ikakvog karaktera za razdvajanje reči.

**Dodavanje klase u paket**

Paket koji smo upravo kreirali je prazan. Kako bi se unutar njega dodala neka klasa, potrebno je uraditi sledeće:

* klasu je potrebno smestiti unutar foldera koji predstavlja paket;
* na početak fajla u kome se nalazi klasa je potrebno dodati jednu posebnu naredbu za smeštanje klase u paket.

Stoga, ukoliko bismo želeli da našu klasu Car dodamo u upravo kreirani paket, bilo bi potrebno da je prebacimo u folder koji predstavlja paket i na njen sam početak postavimo sledeću liniju koda:

package mypackage;

public class Car {  
}

Naredba za smeštanje klase unutar paketa definiše se korišćenjem ključne reči **package** nakon koje se navodi naziv paketa.

**Korišćenje klase iz nekog drugog paketa**

Sve klase koje se nalaze u jednom paketu uzajamno su vidljive bez eksplicitnog naglašavanja. Ipak, klasa iz jednog paketa nije u stanju da vidi klase i njihove članove iz nekih drugih paketa. Takvu situaciju sada imamo i unutar našeg primera:

A screen shot of a computer program

Description automatically generated

Naša glavna klasa MyFirstProgram nalazi se unutar podrazumevanog paketa. Unutar main metode pokušavamo da pristupimo klasi Car, koja je sada deo posebnog paketa – mypackage. Razvojno okruženje jasno signalizira da ne može da pronađe klasu Car.

U praksi se najčešće pribegava rjesenju koji se naziva **uvoz ili importovanje** (import) klase iz jednog paketa u klasu koja se nalazi u nekom drugom paketu. Posao importovanja se obavlja korišćenjem ključne reči **import**.

Ponekad će se javiti potreba za importovanjem većeg broja klasa iz jednog paketa:

import mypackage.MyClass;

import mypackage.MyClass1;

import mypackage.MyClass2;

Ukoliko je potrebno importovati sve klase koje nalaze u jednom paketu, nakon naziva paketa navodi se karakter tačka i nakon njega zvezdica:

import mypackage.\*;

**Više klasa sa identičnim nazivom**

Paketi, između ostalog, omogućavaju da unutar jednog Java programa postoji veći broj klasa sa identičnim nazivom. Unutar jednog paketa, sve klase moraju imati jedinstvene nazive, ali dve klase koje se nalaze u različitim paketima mogu imati identična imena.

**Paketi unutar paketa**

Pored klasa, sadržaj paketa mogu biti i drugi paketi. Tako je moguće napraviti paketnu strukturu dublju od jednog nivoa. Paket unutar paketa se može kreirati na već prikazan način, aktiviranjem opcije New -> Package. Ipak, kontekstni meni je potrebno aktivirati nad već kreiranim paketom.

Unutar IntelliJ IDEA razvojnog okruženja, različiti nivoi paketa se odvajaju karakterom tačka. Kada pokrenemo opciju za kreiranje potpaketa, unutar prozora za unos njegovog naziva već dobijemo pripremljen prefiks subpackage.

Iako se možda čini da se na ovaj način automatski importuju i sve klase potpaketa subpackage, to nije slučaj. Sadržaj takvog paketa moramo eksplicitno importovati:

import mypackage.\*;

import mypackage.subpackage.\*;

**Ugrađeni paketi u jeziku Java**

Java posjeduje ugradjene pakete, kao sto mozete vidjeti na slici ispod:

A diagram of a program

Description automatically generated

Na slici možete videti da su koreni ugrađeni paketi **java** i **javax**. Neko opšte pravilo je da su java paketi bazni, a javax paketi nadogradnje. Takođe, ugrađeni paketi imaju i svoje potpakete. Na primer, potpaket paketa java je awt, dok awt ima svoj potpaket event.

Neki od najznačajnijih ugrađenih Java paketa i njihovi kratki opisi prikazani su u tabeli ispod:

A screenshot of a computer

Description automatically generated

Jedini ugrađeni paket koji ne zahteva prethodno importovanje je **java.lang**. Ovaj paket se automatski importuje u svaki Java program. Svi ostali paketi zahtevaju importovanje, odnosno referenciranje u okviru klase u kojoj se koriste. Na primer, ukoliko bismo želeli da iskoristimo klasu Rectangle koja se nalazi u paketu java.awt, ne bi bilo dovoljno da napišemo samo ovako:

Rectangle rectangle = new Rectangle();

Pod uslovom da u našem programu nema klase sa ovim nazivom koju smo samostalno kreirali, kompajler neće moći da zna gde je potrebno da potraži klasu sa nazivom Rectangle. Stoga mu mi to moramo reći:

import java.awt.\*;

Na ovaj način smo obavili importovanje svih klasa koje se nalaze unutar paketa java.awt. Klasu Rectangle smo mogli importovati i pojedinačno:

import java.awt.Rectangle;

IntelliJ IDEA razvojno okruženje poseduje nekoliko načina na koje samostalno, umesto nas, može da obavi dodavanje neophodne naredbe za importovanje klasa koje se nalaze u nekim drugim paketima.

Prvi način je da prilikom pisanja naredbe za instanciranje klase odaberete odgovarajuću klasu iz prozora sa ponuđenim tipovima.

Kada unesete naziv Rectangle, razvojno okruženje će vam ponuditi sve klase koje u svom nazivu poseduju unetu reč. Potrebno je da se pozicionirate na Rectangle klasu iz paketa java.awt i pritisnete Enter. IntelliJ IDEA će automatski na početak dokumenta dodati naredbu za import:

import java.awt.\*;

Drugi način za automatsko dodavanje import naredbe jeste odabir opcije Import class, koja se može naći u panelu koji se otvara kada pokazivač miša pozicionirate preko klase koja još nije importovana.

A screen shot of a computer program

Description automatically generated

Pod uslovom da IntelliJ IDEA može da detektuje gde se takva klasa nalazi, obaviće automatsko importovanje.

**Svojstva i metode**

**Modifikatori pristupa**

Java jezik poseduje funkcionalnost za kontrolu dostupnosti klasa i njihovih članova. Korišćenjem takve funkcionalnosti možemo da kontrolišemo pristup klasama i njihovim članovima, odnosno da odredimo iz kojih delova našeg programa će njima moći da se pristupi. Takva funkcionalnost se drugačije naziva kontrola pristupa; realizuje se upotrebom posebnih ključnih reči koje se nazivaju modifikatori pristupa.

U Java jeziku postoje četiri modifikatora pristupa:

**private**

Ovo je modifikator pristupa koji označava da će članovi deklarisani ovom ključnom rečju biti vidljivi samo u okviru klase u kojoj se nalaze. Dobija se korišćenjem ključne reči private i moguće ga je koristiti samo nad klasnim članovima.

**package-private**

Modifikator pristupa koji označava da svi drugi elementi istog paketa imaju pristup klasi ili članu klase sa ovakvim modifikatorom. Ovaj modifikator pristupa predstavlja podrazumevani modifikator i jedini je koji ne poseduje ključnu reč. Stoga se dobija kada se ključna reč izostavi.

**protected**

Modifikator pristupa koji ukazuje na to da se članovima može pristupiti iz same klase gde su definisani i iz [izvedenih klasa](https://www.link-elearning.com/linkdl/opisPojma.php?id=160485). Ovaj modifikator se može koristiti samo nad klasnim članovima, a dobija se upotrebom ključne reči protected.

**public**

Modifikator pristupa kojim se kreiraju javno dostupne klase i njeni članovi. Takve klase i članovi su dostupni iz svih delova jednog Java programa. Dobija se korišćenjem ključne reči public.

A group of colorful dots

Description automatically generated with medium confidence

Sada je jasno da ključna reč public prilikom definisanja klase i main metode u kodu našeg prvog Java programa označava da će takvi elementi našeg programa biti javno dostupni. main metoda, kao ulazna tačka programa, mora biti javno dostupna. U protivnom dolazi do greške:

private static void main(String[] args) {

}

main metodu smo sada učinili privatnom i prilikom prevođenja programa sa ovakvom metodom se dobija poruka:

Error: Main method not found in class MyFirstProgram, please define the main method as:   
    public static void main(String[] args)  
    or a JavaFX application class must extend javafx.application.Application

Modifikatore pristupa možemo još bolje razumeti na primeru prve Java klase koju smo u prethodnoj lekciji samostalno kreirali:

class Car {

String make;

String model;

Car() {

}

Car(String make, String model) {

this.make = make;

this.model = model;

}

void startEngine() {

System.out.println("Engine started...");

}

}

Unutar naše klase Car, ni na jednom elementu nema ključne reči koja definiše modifikator pristupa. To praktično znači da sama klasa i svi njeni članovi poseduju podrazumevani modifikator pristupa – package-private. Pogledajte šta bi se dogodilo kada bismo, na primer, metodu za startovanje motora učinili privatnom:

private void startEngine() {

System.out.println("Engine started...");

}

Metoda startEngine() je sada privatna i izvan klase se uopšte i ne može pozvati.

A screen shot of a computer program

Description automatically generated

Razvojno okruženje IntelliJ IDEA jasno signalizira da metoda startEngine() ima privatni modifikator pristupa. Stoga ovakav kod poseduje grešku zbog koje neće biti moguće njegovo prevođenje.

**Modifikatori pristupa klasa**

Kao što ste nešto ranije mogli da pročitate, direktno nad klasama se mogu definisati samo dva modifikatora pristupa: public i package-private. Nad klasnim članovima se mogu primeniti i svi ostali modifikatori.

**Statički elementi klasa**

Ulazna tačka Java programa, odnosno main metoda poseduje još jednu ključnu reč koja je za nas nepoznanica:

public static void main(String[] args) {

}

To je ključna reč **static** koju možete videti u potpisu main metode. Korišćenje ključne reči static govori da određeni element klase pripada isključivo klasi, a ne nekom pojedinačnom objektu koji se korišćenjem takve klase kreira. To na kraju omogućava da se statičkom elementu može pristupiti bez prethodnog instanciranja klase.

Metoda main po jezičkoj specifikaciji mora biti statička, kako bi virtuelna mašina prilikom pokretanja aplikacije mogla da je pozove bez prethodnog kreiranja objekta klase u kojoj se main metoda nalazi.

I mi smo u prethodnoj lekciji kreirali jednu metodu koja je idealni kandidat da postane statička. Reč je o metodi klase Car, koja je namenjena za pretvaranje snage motora iz kilovata u konjske snage:

double kwToHp(int kw){

return kw / 0.745699872;

}

S obzirom na to da svaki automobil mora imati motor, metoda za računanje jačine motora u konjskim snagama karakteristična je za sve automobile. Stoga bi bilo dobro kada bismo takvu metodu mogli da koristimo bez prethodnog instanciranja. Tako nešto se može postići njenim pretvaranjem u statičku metodu:

static double kwToHp(int kw){

return kw / 0.745699872;

}

Po identičnom principu, mogu se kreirati i statička polja:

class Car {

String make;

String model;

static int noCarInTheShowroom = 16;

}

Klasa Car sada poseduje još jedno svojstvo (polje) tipa int, sa nazivom noCarInTheShowroom. Njegova vrednost je direktno unutar klase inicijalizovana na 16. Ovo svojstvo možete da doživite kao način da se u programu vodi evidencija o ukupnom broju vozila koja se nalaze u nekom auto-salonu. Svakako to nije informacija koja je vezana za neku konkretnu instancu klase Car, pa je upravo zbog toga ovo polje i proglašeno statičkim, definisanjem ključne reči static ispred njegovog tipa.

Ovakvom polju se sada može pristupiti na sledeći način:

System.out.println(Car.noCarInTheShowroom);

Ovakva naredba će na izlazu da ispiše vrednost 16.

Zanimljivo je da se statičkim elementima klase može pristupiti i nad objektima:

Car myCar = new Car();

System.out.println(myCar.noCarInTheShowroom);

Rezultat će biti identičan kao i u prethodnom primeru. Ipak, pristup statičkim članovima klase preko njihovih instanci predstavlja lošu programersku praksu. Ovde smo to uradili kako bismo ilustrovali jednu veoma važnu osobinu statičkih svojstava:

Car myCar = new Car();

Car myCar2 = new Car();

myCar.noCarInTheShowroom = 20;

System.out.println(myCar2.noCarInTheShowroom);

Iz ovoga se može zaključiti da, iako izgleda kao da noCarInTheShowroom pripada pojedinačnim objektima, to zapravo nije tako. Kao što je već rečeno, statička polja pripadaju klasi, a ne pojedinačnim instancama.

**Konstante**

Logičan nastavak priče o statičkim poljima jeste definisanje pojma konstanti. Konstante su polja čije se vrednosti nakon inicijalizacije ne mogu menjati. Konstanta se može kreirati na sledeći način:

public static final double PI = 3.14;

Prikazanom naredbom definiše se jedna konstanta. Konstanta je zapravo matematička vrednost PI. Bitno je da primetite da se u Javi konstante kreiraju kombinacijom ključnih reči static i final. Značenje ključne reči static smo već obradili, dok ključna reč **final** osigurava da se vrednost ovakvog polja neće moći menjati.

A screen shot of a computer program

Description automatically generated

Konstante se po konvenciji imenuju velikim slovima, baš kao što je to obavljeno u prethodnom primeru (PI). Ukoliko se naziv konstante sastoji iz više reči, pojedinačne reči se odvajaju donjom crtom:

public static final String APPLICATION\_PATH = "c:/myApplication/";

**Tipovi promenljivih u zavisnosti od mesta deklaracije**

S obzirom na to da smo se u dosadašnjem toku lekcije upoznali sa klasama i metodama, možemo da razmotrimo sva ona mesta na kojima se mogu pojaviti promenljive u jednom Java program.

A screenshot of a computer screen

Description automatically generated

Slika prikazuje kod naše Car klase. Klasa poseduje tri svojstva, dva konstruktora i jednu metodu. Na primeru ovakvog koda, možemo konstatovati da promenljive u Javi mogu imati nekoliko različitih uloga, na osnovu kojih se može reći da postoje sledeće vrste promenljivih:

* objektne promenljive (svojstva);
* klasne (statičke) promenljive;
* lokalne promenljive;
* parametarske promenljive.

**Objektne promenljive** ili, kako se često nazivaju, instance jesu one promenljive koje predstavljaju svojstva instance neke klase. Ovo su promenljive koje se pojavljuju samo u kontekstu objekta i nije moguće pristupiti im bez prethodnog instanciranja klase. U prikazanom primeru, to su svojstva make i model (ali ne i ulazni parametri konstruktora make i model; oni su nešto drugo):

Car.make = "Honda";

Ovakva naredba proizvodi grešku, zato što se svojstvu make ne može pristupiti bez objekta.

**Klasne (statičke) promenljive** postoje samo u domenu klase, što znači da nije potrebno obaviti instanciranje kako bi im se pristupilo.

U primeru sa slike klasna promenljiva je noCarInTheShowroom:

Car.noCarInTheShowroom = 20;

**Lokalne promenljive** su one koje su deklarisane u okviru nekog bloka koda. U našem primeru, takva je promenljiva result unutar metode kwToHp. Reč je o promenljivama koje postoje samo u okviru metode u kojoj su definisane, pa se upravo zbog toga i zovu lokalne. To praktično znači da njima nije moguće pristupiti izvan metode u kojoj se nalaze.

A screen shot of a computer program

Description automatically generated

Primer podrazumeva jednu klasu sa dve metode. Jedna je osnovna main metoda, a druga metoda sa nazivom testMethod unutar koje je deklarisana jedna lokalna promenljiva. Unutar main metode se pokušava pristupiti promenljivoj myVariable – što, naravno, rezultuje neuspehom.

Pojam lokalnih promenljivih nije ograničen na metode. Drugim rečima, lokalne promenljive mogu da postoje i unutar drugih vrsta blokova sa kojima smo se već susretali. Reč je o uslovnim blokovima i blokovima petlji. Promenljive deklarisane unutar takvih blokova takođe su lokalne:

if (true) {

String myValue = "Hello World";

}

System.out.println(myValue);

Ovde možemo da vidimo jednu promenljivu (myValue) koja je deklarisana unutar uslovnog if bloka. Uslov je formulisan tako da će se njegov kod uvek izvršiti. Ipak, i pored toga, pokušaj ispisa vrednosti promenljive myValue izvan bloka u kome je ona deklarisana stvara grešku:

java: cannot find symbol  
    symbol:   variable myValue

**Parametarske promenljive** su zapravo jedna vrsta lokalnih promenljivih, ali promenljivih koje omogućavaju da metode dobiju ulazne ili emituju izlazne vrednosti. U primeru sa slike, takve promenljive su make i model, kao ulazni parametri konstruktora, i kw kao ulazni parametar metode kwToHp. Bitno je da razumete da će takve promenljive prilikom pozivanja metode biti popunjene prosleđenim vrednostima i da se od tog trenutka one ponašaju kao upravo ilustrovane lokalne promenljive:

static double kwToHp(int kw) {

kw = 10;

double result = kw / 0.745699872;

return result;

}

Evo sada smo obavili i pristup takvoj lokalnoj promenljivoj i čak izvršili i promenu njene vrednosti. U ovakvoj situaciji, šta god da se ovoj metodi prosledi kao ulazni parametar, ona će kao vrednost kilovata uvek koristiti broj 10:

System.out.println(Car.kwToHp(150)); // 13.410220888438078

**Primitivni/referentni parametri**

Kada smo već kod ulaznih parametara metoda, dobro je poznavati još jednu njihovu osobinu koja je veoma važna. Naime, u programskom jeziku Java, metodama se mogu prosleđivati primitivne vrednosti, ali i parametri složenih tipova, odnosno objekti.

Pogledajmo šta se dešava kada se metodi prosleđuje parametar primitivnog tipa:

A screenshot of a computer program

Description automatically generated

U prikazanom kodu definisana je jedna klasa sa nazivom TestClass. Ona sadrži standardnu main metodu za pokretanje programa i još jednu metodu sa nazivom passMethod, koja će poslužiti kao metoda na kojoj ćemo obavljati ogled. Ova metoda prihvata jedan ulazni parametar tipa int, postavlja njegovu vrednost na 10 i na kraju na izlazu ispisuje vrednost promenljive x.

U okviru main metode deklarisana je promenljiva x i njoj je dodeljena vrednost 5.

**Lokalne promenljive istog naziva**

Obratite pažnju na to da unutar obe metode (main i passMethod) postoje promenljive identičnog naziva (x). Ovako nešto je potpuno legitimno, s obzirom na to da je reč o lokalnim promenljivama, koje postoje samo unutar bloka u kome su deklarisane. Stoga je svaka od ovih promenljivih potpuno nezavisna i one ne utiču jedna na drugu.

Unutar main metode zatim je obavljen poziv metode passMethod i njoj je prosleđena vrednost lokalne promenljive x. Pre i posle poziva metode, dodate su naredbe za ispis vrednosti lokalne promenljive x. Na kraju izvršavanja programa, na izlazu se dobija sledeće:

This is initial value: 5  
This is the value from method: 10  
This is the value after the completion of the method: 5

Možemo da se uverimo da metoda nije izvršila promenu vrednosti promenljive x. Razlog je vrlo jednostavan. Primitivne promenljive se prosleđuju po vrednosti, odnosno, metode dobijaju samo kopije vrednosti promenljivih. Na taj način, promenljive izvan metoda ostaju nepromenjene.

Referentni tipovi podataka se u Javi takođe prosleđuju po vrednosti. U Javi ne postoji nešto što se naziva prosleđivanje po referenci. Kada se metodi prosledi neki objekat, njoj se, u suštini, prosleđuje pokazivač na objekat, odnosno vrednost memorijske lokacije objekta. Ipak, vrednosti polja jednog objekta se mogu izmeniti u okviru metode ukoliko imaju modifikator pristupa koji to omogućava. To će ilustrovati naredni primer, u kome ćemo koristiti jedan referentni tip koji ćemo samostalno kreirati:

public class ReferenceType {

public int x = 5;

}

Kao što vidite, reč je o klasi ReferenceType unutar koje se nalazi samo jedno polje sa imenom x i vrednošću 5.

**Podrazumevane vrednosti polja**

Već više puta do sada ste mogli da vidite da je promenljivu moguće deklarisati bez inicijalizacije, odnosno bez dodeljivanja početne vrednosti. Vrlo je bitno znati šta će se u takvim situacijama dogoditi sa takvim promenljivama, odnosno da li one poseduju neku podrazumevanu vrednost ili ne.

Prilikom rada sa lokalnim promenljivama (promenljive unutar metoda), situacija je vrlo jasna. Sve promenljive moraju biti inicijalizovane ili dolazi do pojave greške,

A screen shot of a computer program

Description automatically generated

Promenljiva sa nazivom value nije inicijalizovana i prilikom pokušaja da joj se pristupi dolazi do pojave greške.

Ipak, drugačija je situacija sa promenljivama koje se deklarišu unutar klase, bilo da je reč o klasnim ili o objektnim promenljivama:

public class TestClass {

int variableA;

String variableB;

boolean variableC;

char variableD;

}

Ovo je klasa sa četiri polja, odnosno četiri promenljive koje su deklarisane, ali nisu inicijalizovane. Pogledajte sada šta se dešava ukoliko napravimo jedan objekat na osnovu ove klase:

TestClass myObj = new TestClass();

System.out.println(myObj.variableA);

System.out.println(myObj.variableB);

System.out.println(myObj.variableC);

System.out.println(myObj.variableD);

A black screen with white text

Description automatically generated

Na osnovu ovakvog izlaza, može se zaključiti da objektne promenljive, odnosno svojstva, dobijaju određene podrazumevane vrednosti kada ih ne inicijalizujemo eksplicitno. Podrazumevana vrednost se razlikuje u zavisnosti od tipa promenljive:

* numeričke promenljive dobijaju podrazumevanu vrednost 0 ili 0 u zavisnosti od tipa;
* promenljive tipa String kao podrazumevanu vrednost dobijaju null;
* promenljive tipa boolean imaju podrazumevanu vrednost false;
* promenljive tipa char imaju podrazumevanu vrednost karaktera koji u [Unicode](https://www.link-elearning.com/linkdl/opisPojma.php?id=160466) setu karaktera ima kod \u000; reč je o karakteru koji se zove null character.

**Šta je to null?**

Upravo ste mogli da vidite da je podrazumeva vrednost String promenljivih null. To je zapravo rezervisana vrednost koju mogu imati objektne promenljive. Ona predstavlja referencu koja ne pokazuje ni na jedan objekat. Tako null predstavlja neku vrstu pandana vrednosti 0, koja se može dodeliti prostim tipovima, s tim što se null može koristiti samo nad objektima.

O stacku i heapu već je bilo reči u jednoj od prethodnih lekcija, tako da znamo da se referentni tipovi smeštaju na heap, dok se na stacku nalazi samo pokazivač, odnosno memorijska lokacija objekta. Kada neki objekat ima vrednost null (odnosno nema vrednost), to znači da kreirana objektna promenljiva ne pokazuje ni na jednu memorijsku lokaciju.

Pošto je moguće razdvojiti kreiranje i instanciranje jednog objekta, u periodu između ove dve operacije vrednost deklarisane promenljive biće null.

**Postulati OOP-a**

Za kompletno razumevanje objektno orijentisanog programiranja neophodno je da ovladamo još nekim veoma važnim principima. Stoga će lekcija pred vama biti posvećena osnovnim postulatima objektno orijentisanog programiranja. Oni su sledeći:

* apstrakcija;
* enkapsulacija;
* nasleđivanje;
* polimorfizam.

**Apstrakcija**

Kreiranje objekata koji predstavljaju entitete iz realnog sveta drugačije se naziva modelovanje. Modelovanjem se omogućava rukovanje entitetima u programskom kodu Java programa. Tako modelovanje omogućava da korišćenjem klasa u programskom jeziku Java stvorimo tipove koji su našem programu potrebni, a inicijalno ne postoje unutar same Java platforme:

A computer code on a white background

Description automatically generated

Ovo je primer Car klase za modelovanje automobila. Klasa nam je poznata iz prethodne lekcije, a u ovoj lekciji je ona blago modifikovana, tako da sadrži:

* četiri objektna svojstva – proizvođač, model, težina i boja;
* jedno klasno svojstvo, koje služi za čuvanje ukupnog broja vozila;
* dva konstruktora, pri čemu prvi nema parametara, a drugi prihvata parametre za inicijalizaciju svojstava;
* objektnu metodu startEngine(), koja formira ispis da je motor određenog automobila pokrenut.

U upravo prikazanom primeru ste mogli da vidite da se u procesu modelovanja veoma kompleksni pojmovi iz realnog sveta mogu predstavljati znatno jednostavnije, što se često i radi. Tako smo i mi prilikom kreiranja klase automobila modelovali samo nekoliko osobina i ponašanja takvog realnog pojma.

Drugim rečima, jedan automobil se, pored proizvođača, modela, težine i boje, može definisati i brojnim drugim osobinama – tip menjača, emisiona klasa, broj brzina, tip pogona itd. Jasno je da smo mi prilikom modelovanja pojma automobila napravili jedan uprošćeni, pojednostavljeni model. Takav proces, kojim se postiže kreiranje jednostavnih modela znatno kompleksnijih entiteta iz realnog sveta, u objektno orijentisanom programiranju se drugačije naziva apstrakcija.

**Enkapsulacija**

Objekat predstavlja zaokruženu celinu, zato što u potpunosti opisuje jedan pojam iz realnog sveta. Unutar objekata objedinjuju se informacije i funkcionalnosti koje se tiču entiteta koji se modeluje. Objedinjavanje takvih informacija i funkcionalnosti drugačije se naziva enkapsulacija.

Enkapsulacija se najbolje može razumeti na već prikazanim primerima kreiranja objekata automobila.

A close-up of a pill

Description automatically generated

Slika dočarava pojam enkapsulacije na primeru jednog Java objekta. Ovdje nije slučajno prikazana prava kapsula – svojstva i metode objedinjuju se unutar objekta baš kao i različiti sastojci unutar jedne farmaceutske kapsule. Enkapsulacijom se različita svojstva i metode jednog objekta prezentuju kao celina, kojom se rukuje korišćenjem naziva promenljive koja čuva referencu na objekat.

Veoma bitan scenario kojim se u Java jeziku postiže enkapsulacija, jesu modifikatori pristupa o kojima je već bilo reči. Naime, korišćenjem modifikatora pristupa, mi imamo mogućnost da veoma precizno definišemo kako se objektima i članovima naših klasa može rukovati. Drugim rečima, korišćenjem modifikatora pristupa imamo mogućnost da kontrolišemo vidljivost određenih objektnih članova i da na taj način postignemo još bolji efekat enkapsulacije.

Enkapsulacija se veoma često demonstrira na primeru objektnih svojstava, tako što se ona kreiraju kao privatna polja. Kako bi moglo da im se pristupi i izvan klase u kojoj su definisana, pribegava se definisanju specijalnih metoda koje se koriste za upis i čitanje vrednosti privatnih svojstava. Takve metode su poznate pod nazivom **get i set metode**. Evo kako to izgleda na praktičnom primeru:

A computer code on a white background

Description automatically generated

Ovo je isečak klase Car, unutar koje je svojstvo weight privatno, pa mu se ne može pristupiti izvan klase. Kako bi njegovom vrednošću moglo da se rukuje, kreirane su dve metode – getWeight i setWeight. Unutar tela ovih metoda obavlja se čitanje i upisivanje vrednosti privatnog weight polja, upravo tim redosledom.

Postojanje get i set metoda omogućava kontrolisanu interakciju sa privatnim poljem weight. To praktično znači da na ovaj način možemo da definišemo neku posebnu logiku koja će se aktivirati prilikom čitanja i upisivanja vrednosti ovog svojstva. Na primer, nešto ovako:

A computer code with numbers and symbols

Description automatically generated with medium confidence

Ovo je metoda za postavljanje vrednosti weight svojstva. Možete da vidite da unutar ovakve metode, pored osnovne logike postavljanja vrednosti, postoji i kod koji proverava prosleđenu vrednost. Ukoliko je ona manja od nule, svojstvo dobija vrednost 0, a ukoliko je prosleđena vrednost veća od 3000, svojstvo dobija vrednost 3000. Drugim rečima, definisana logika ne dozvoljava da svojstvo weight ima vrednost izvan opsega 0–3000.

Korišćenje get i set metoda može da učini da neko ko koristi objekte naše klase, a nema pristup izvornom kodu, ne može ni da zna da li neko svojstvo postoji ili ne. Evo jednog takvog primera:

A screenshot of a computer code

Description automatically generated

Primer opet podrazumeva isečak klase Car, sa akcentom na make i model svojstva, koja su privatna. Za postavljanje njihovih vrednosti postoji po jedna set metoda. Ipak, direktno čitanje vrednosti ovih svojstava izvan klase nije moguće, s obzirom na to da nisu napravljene pojedinačne get metode. Umesto njih, u klasi postoji jedna get metoda – getName. Reč je o metodi koja isporučuje kompletno ime automobila, koje je sačinjeno iz proizvođača i modela. Ovaj primer ilustruje dve bitne osobine:

* get i set metode omogućavaju da se naprave polja koje se mogu samo čitati ili samo upisivati;
* get i set metode omogućavaju da se javna vrednost formira na osnovu više privatnih vrednosti (u prikazanom primeru, korisnik klase ne može ni da zna da u klasi ne postoji polje name).

**Nasleđivanje**

Nasleđivanje omogućava kreiranje klasa cije osobine i ponašanja nasleđuju od neke druge klase.

A car and truck with arrows

Description automatically generated with medium confidence

Slika ilustruje jedan primer nasleđivanja u objektno orijentisanom svetu. Klase Convertible i Pickup nasleđuju osnovni skup svojstava i ponašanja od klase Car. Unutar klase Car objedinjene su osnovne karakteristike koje poseduju svi automobili. Klase Convertible i Pickup nasleđuju takav skup osnovnih osobina, ali pored njih poseduju i određene osobine koje su karakteristične samo za njih.

Pređimo sada na prvi primer nasleđivanja. U nastavku ćemo kreirati klasu Convertible za predstavljanje posebne vrste automobila – kabrioleta:

public class Convertible extends Car {

// ...

}

Ključna reč kojom se obavlja nasleđivanje između klasa u Java jeziku jeste **extends**. Ova ključna reč navodi se između naziva klase koja se nasleđuje i nasleđene klase. Sledeći korak jeste definisanje klasnih članova naše nove klase. Klasa Convertible imaće sve članove koje ima i klasa Car, uz dodatak jednog svojstva koje će čuvati vrednost koja oslikava tip krova kabrioleta. S obzirom na to da smo obavili nasleđivanje, ne moramo da sve one članove iz klase Car ponovo kreiramo unutar klase Convertible, već je potrebno da kreiramo isključivo članove koji se razlikuju:

public class Convertible extends Car {

String roofType;

}

Kao što je rečeno, osobenost klase Convertible jeste svojstvo za čuvanje tipa krova, te smo unutar nje deklarisali samo njega.

Sledeći korak je da definišemo konstruktor:

public class Convertible extends Car {

String roofType;

public Convertible(String make, String model, int weight, String color, String roofType) {

super(make, model, weight, color);

this.roofType = roofType;

}

}

Konstruktor Convertible klase prihvata sve parametre kao i onaj iz klase Car, uz dodatak parametra koji je karakterističan za nju. Unutar konstruktora koristi se jedna posebna ključna reč – **super**. To je ključna reč koja omogućava da se obavi pozivanje konstruktora ili pristupanje svojstvima i metodama roditeljske klase. Ključa reč super najčešće se koristi za pozivanje konstruktora roditeljske klase, što je učinjeno i u primeru.

Prilikom takvog poziva se prosleđuju parametri koje prihvata konstruktor roditeljske klase. Tako će roditeljska klasa da obavi njihovu obradu, a nama samo preostaje da inicijalizujemo svojstvo roofType, što je u primeru i učinjeno. Klasa Convertible kreirana u prethodnim redovima nasledila je sve osobine klase Car.

**Pravila prilikom nasleđivanja u Javi**

Prilikom nasleđivanja postoji par pravila koja treba da budu ispoštovana i od kojih zavisi tretman nasleđenih elemenata. Prvo pravilo glasi da klasa koja nasleđuje može imati samo jednu roditeljsku klasu.

Drugo pravilo se odnosi na modifikatore pristupa. Naime, direktno se nasleđuju samo  public, protected i package-private klasni članovi. Klasnim članovima koji su obeleženi private modifikatorima pristupa ne može se direktno pristupiti iz klasa koje nasleđuju. Kako bi im se pristupilo, takvi elementi moraju imati protected modifikator ili odgovarajuće get i set javne metode.

**Zabrana nasleđivanja**

U programskom jeziku Java moguće je i onemogućiti da se neka klasa nasledi. Sve što je potrebno uraditi, jeste postaviti ključnu reč final u deklaraciju klase:

public final class Car {

//...

}

Sada je definisano da se klasa Car neće moći naslediti, pa u ovakvoj situaciji kod koji smo do sada pisali u ovoj lekciji neće funkcionisati.

**Polimorfizam**

Prilikom realizovanja nasleđivanja u objektno orijentisanom programiranju, veoma često dolazi do potrebe za redefinisanjem nekih nasleđenih ponašanja. Tako roditeljski objekat može definisati neku funkcionalnost koja ne zadovoljava u potpunosti njegove naslednike. Ipak, naslednici imaju mogućnost da nasleđeno ponašanje prilagode, dopune ili u potpunosti izmene. Na taj način se stvaraju različite verzije jednog istog ponašanja, odnosno verzije koje odgovaraju svakom pojedinačnom tipu objekta. Takva osobina objektno orijentisanog programiranja drugačije se naziva polimorfizam, odnosno višestruko značenje jednog istog pojma.

Upravo opisanu situaciju imamo u prethodnom primeru. Pogledajte o čemu je reč:

Convertible convertible1 = new Convertible("Honda", "S2000", 1274, "silver", "Vinyl, soft-top");

System.out.println(convertible1.getInfo());

U prikazanom primeru kreiran je jedan objekat tipa Convertible. Zatim je nad njim pozvana metoda getInfo koja je definisana unutar roditeljske klase. Evo kako izgleda ispis:

Make: Honda  
Model: S2000  
Weight: 1274  
Color: silver

Ukoliko analizirate dobijeni ispis, moći ćete da zaključite da u njemu nedostaje informacija o tipu krova. Ovo je potpuno očekivano, pošto se metoda getInfo nalazi u roditeljskoj klasi, a roditeljska klasa uopšte ne poseduje polje za čuvanje informacije o tipu krova. Da bi se ovakav problem prevazišao, neophodno je i unutar klase Convertible definisati metodu getInfo. Takav postupak se u objektno orijentisanom programiranju naziva redefinisanje metoda (**overriding**):

A computer screen shot of text

Description automatically generated

Sada i unutar klase Convertible postoji metoda getInfo. Ona obavlja isto što i metoda iz roditeljske klase, uz dodatak ispisa informacije o tipu krova. Kako bi se naznačilo da je reč o redefinisanoj metodi, odnosno kako bi se naglasilo da metoda sa istim nazivom postoji i unutar roditeljske klase, pre potpisa metode postavljena je oznaka **@Override**.

S obzirom na to da je najveći deo upravo prikazane metode identičan onom u roditeljskoj klasi, dobro bi bilo kada bismo na neki način mogli da iskoristimo logiku koja je već definisana unutar roditeljske klase. Tako nešto zapravo i možemo da uradimo i to korišćenjem jedne ključne reči sa kojom smo se već susretali. U pitanju je ključna reč super. Ona će nam omogućiti da pristupimo getInfo metodi iz roditeljske klase:

A computer code with black text

Description automatically generated

Ključna reč super omogućava pristup objektu roditeljske klase. Nju smo iskoristili kako bismo pozvali metodu getInfo iz Car klase. Od nje dobijamo informacije o proizvođaču, modelu, težini i boji, pa na kraju na takve informacije dodajemo još samo podatak o tipu krova. Ovakav pristup je, kao što možete videti, znatno kompaktniji i predstavlja vrhunac redefinisanja metoda.

**Overloading**

Još jedan pojam u Javi koji oslikava klasičan primer polimorfizma jeste overloading. Sa ovim pojmom smo se već sreli u jednoj od prethodnih lekcija, kada je bilo reči o definisanju konstruktora. Naime, overloading se ogleda u mogućnosti da u okviru jedne iste klase postoji veći broj metoda istog imena, ali različitih ulaznih i izlaznih parametara. Prilikom poziva metode, u zavisnosti od prosleđenih parametara, poziva se odgovarajuća metoda. I u primeru iz ove lekcije postoji overloading. Reč je o klasi Car, u kojoj postoje dva konstruktora. Oni, naravno, imaju iste nazive, a različite ulazne parametre, što je primer overloadinga.

**Nizovi**

U jednoj od prethodnih lekcija naučili smo kako se barata promenljivama. Iako se vrednosti promenljivih mogu menjati, one u jednom trenutku mogu imati samo jednu vrednost. Na primer:

String cars = "Honda";

Ovo je jedna String promenljiva koja čuva vrednost koja predstavlja jednog proizvođača automobila. Ipak, šta ukoliko bismo unutar promenljive cars poželeli da sačuvano imena još nekih proizvođača automobila? Za obavljanje takvog posla bili bi nam potrebni nizovi.

**Šta su nizovi?**

Osnovni sastojak gotovo svih programskih jezika su nizovi. Nizovi omogućavaju da se unutar jedne promenljive smesti veći broj vrednosti. Kako biste ovu rečenicu bolje razumeli, evo kako izgleda struktura jednog Java niza:

A screen shot of a number

Description automatically generated

Plavi kvadrati oslikavaju različite vrednosti koje postoje unutar jednog niza. Može se videti da ih ukupno ima devet. Svaka vrednost unutar niza se drugačija naziva **element niza**. Svaki element niza poseduje svoju poziciju, koja se izražava korišćenjem **indeksa**. Brojanje indeksa nizova započinje od nule, pa tako prvi element niza ima indeks 0, a peti element niza indeks 4.

Unutar jednog Java niza moguće je smeštati podatke istog tipa. Tako se unutar jednog niza može naći skup brojeva, unutar drugog skup tekstualnih podataka, a unutar trećeg niza skup logičkih vrednosti.

**Deklarisanje niza**

Kreiranje niza u Java jeziku započinje njegovom deklaracijom. Deklaracija niza je zapravo deklaracija promenljive, preko koje će vrednosti niza biti dostupne u programu. Evo kako može da izgleda deklaracija jednog niza:

**Int**[] array;

Baš kao i prilikom deklaracije regularnih promenljivih, deklaracija niza se sastoji iz dva dela:

* definisanja tipa niza;
* definisanja naziva niza.

U upravo prikazanom primeru tip niza je **int[]**, što znači da kreiramo niz celobrojnih vrednosti tipa **int**. Uglaste zagrade predstavljaju specijalne karaktere koji govore da će ova promenljiva moći da prihvati niz vrednosti, a ne samo jednu int vrednost.

Drugi deo ove deklaracije predstavlja naziv niza i on je u našem slučaju array.

Kao i kod promenljivih drugih tipova, deklarisanjem se ne kreira automatski i sam niz. Deklaracija govori kompajleru da će kreirana promenljiva sadržati vrednost niza naznačenog tipa.

**Kreiranje niza**

Kreiranje niza obavlja se korišćenjem ključne reči **new**:

int[] array = new int[5];

Ovakvom naredbom kreira se jedan niz celobrojnih vrednosti koji će alocirati memoriju potrebnu za smeštanje pet celobrojnih vrednosti tipa int. Drugim rečima, nakon ove naredbe promenljiva x imaće referencu na niz sa pet elemenata od kojih će svaki predstavljati prostor za jednu celobrojnu vrednost.

**Nizovi u Javi imaju fiksnu veličinu**

Odmah je potrebno primetiti jednu vrlo važnu osobinu nizova u Java jeziku. Njihova veličina se mora unapred utvrditi, odnosno, prilikom kreiranja je neophodno navesti njihovu veličinu. To je učinjeno i u prikazanoj naredbi, gde broj 5 unutar uglastih zagrada predstavlja broj elemenata koje će niz imati. Nakon kreiranja niza u Java jeziku, veličinu niza **nije** moguće menjati.

**Nizovi su objekti**

U programskom jeziku Java, nizovi su objekti. To možete videti i na osnovu upotrebe ključne reči new koja se koristi za njihovo kreiranje. U pozadini, za svaki tip niza, Java jezik poseduje odgovarajuću klasu koju koristi prilikom kreiranja niza.

Nakon izvršavanja upravo prikazane naredbe za kreiranje niza, svi njegovi članovi će biti automatski inicijalizovani i imaće podrazumevane vrednosti 0.

**Ispis vrednosti niza i klasa Arrays**

Kako bismo se uverili da je ovo što smo upravo rekli tačno, odnosno da kreirani niz poseduje pet članova čije vrednosti su trenutno podrazumevane, u narednim redovima ćemo obaviti ispisivanje elemenata niza na izlazu. Za obavljanje takvog posla, iskoristićemo jednu ugrađenu klasu Java platforme. Reč je o klasi Arrays.

Klasa Arrays poseduje skup različitih funkcionalnosti koje su dostupne za manipulaciju nizovima. Jedna od takvih funkcionalnosti jeste i ona koja omogućava da se svi elementi niza objedine unutar jedne String vrednosti:

int[] array = new int[5];

System.out.println(Arrays.toString(array));

Kao što možete videti, klasa Arrays i njena statička metoda toString omogućavaju nam da dobijemo sve elemente niza, objedinjene u jednu String vrednost, koju možemo da ispišemo na izlazu:

[0, 0, 0, 0, 0]

Izlaz koji se dobija potvrđuje ono što je rečeno nešto ranije – niz koji smo kreirali ima pet članova i svaki od njih je inicijalizovan na podrazumevanu vrednost 0.

**Napomena**

Korišćenje klase Arrays zahteva da se ona prethodno uveze, pa je na početak dokumenta potrebno dodati sledeću naredbu:

**import** java.util.Arrays;

**Dodela vrednosti članovima niza**

U prethodnim redovima ste mogli da vidite da odmah nakon kreiranja niza njegovi članovi dobijaju podrazumevane vrednosti, koje naravno zavise od njihovog tipa. Kod numeričkih tipova, podrazumevane vrednosti su 0 ili 0.0, kod logičkih false, a kod referentnih tipova null.

Da bismo samostalno definisali vrednosti članova niza, neophodno je uraditi nešto ovako:

array[0] = 23;

Na ovaj način je obavljeno postavljanje vrednosti prvog elementa niza. Već je rečeno da se elementi nizova numerišu korišćenjem indeksa, a u prikazanoj naredbi je iskorišćen upravo indeks kako bi se pristupilo elementu niza. S obzirom na to da je unutar uglastih zagrada navedena vrednost 0, obavlja se pristup prvom elementu niza.

Nakon ovakve naredbe možemo da proverimo da li je vrednost prvog elementa našeg niza zaista promenjena:

int[] array = new int[5];

array[0] = 23;

System.out.println(Arrays.toString(array));

Kod će proizvesti ovakav efekat:

[23, 0, 0, 0, 0]

Na osnovu dobijenog ispisa, možemo da vidimo da je vrednost prvog člana niza uistinu promenjena.

Članovi niza se ponašaju kao i bilo koje druge promenljive, pa je tako njihovu vrednost moguće menjati proizvoljan broj puta, koristiti unutar izraza, obrađivati operatorima i slično.

**Čitanje vrednosti elemenata niza**

Na identičan način je moguće obaviti i čitanje vrednosti pojedinačnih elemenata niza. Opet je potrebno iskoristiti naziv niza i u uglastim zagradama navesti indeks elementa čiju vrednost želimo da dobijemo:

int[] array = new int[5];

array[0] = 23;

System.out.println(array[0]);

U ovom primeru, na izlazu se ne ispisuju svi elementi niza, već samo onaj koji ima indeks 0. Drugim rečima, na izlazu se dobija vrednost prvog člana niza:

23

**Objedinjena deklaracija i inicijalizacija**

Niz je u Javi moguće deklarisati i inicijalizovati u jednoj naredbi, ukoliko unapred znamo vrednosti njegovih elemenata:

int[] array = new int[]{3, 5, 2, 6, 1, 6};

Odmah nakon kreiranja niza, sada su navedene vitičaste zagrade i unutar njih vrednosti elemenata niza.

Prilikom objedinjene deklaracije i inicijalizacije, moguće je otići i korak dalje i potpuno izostaviti logiku za kreiranje novog objekta:

int[] array = {3, 5, 2, 6, 1, 6};

U programskom jeziku Java, ovo je najkompaktniji način za kreiranje nizova, pod uslovom da unapred znamo elemente koje je niz potrebno da ima.

**Nizovi u Javi prihvataju elemente samo jednog tipa**

Niz može da prihvati isključivo vrednosti istog tipa. To znači da nije moguće napraviti niz celobrojnih vrednosti u kome će se naći i vrednost sa pokretnim zarezom (double ili float).

 U Java jeziku postoji način da se i ovako nešto prevaziđe:

Object[] myArray = {3, 6.5, 2, 6, 1, 6};

Napravljena je samo jedna mala intervencija na već prikazanoj naredbi i greške više nema. Naime, tip članova niza je iz int promenjen u Object. Object je u Javi osnovni tip, odnosno tip na kome se baziraju svi ostali tipovi, pa je ovako nešto potpuno legitimno. Drugim rečima, s obzirom na to da je osnova svih tipova, Object se može koristiti za predstavljanje bilo kojih vrednosti u Javi. Tako možemo da odemo i korak dalje, pa da napišemo ovako nešto:

Object[] myArray = {3, 6.5, 2, 6, 1, 6, "Hello"};

Sada je na kraj niza postavljen još jedan element koji je tipa String. To praktično znači da u ovom trenutku naš niz poseduje vrednosti tri različita tipa. Zapravo, ukoliko želimo da budemo u potpunosti precizni, svi elementi upravo prikazanog niza imaju isti tip – Object.

Pre nego što se upišu u niz, vrednosti koje vidite su različitog tipa. Ali, prilikom upisivanja u niz, okruženje obavlja njihovo prevođenje u Object tip, tako da unutar niza svi članovi poseduju identičan tip. Prilikom čitanja vrednosti iz niza, obavlja se obrnut postupak, odnosno, vrednosti iz Object tipa konvertuju se nazad u svoj izvorni tip. Takve konverzije se drugačije nazivaju boxing i unboxing i obavljaju se automatski od strane Java okruženja.

**Utvrđivanje dužine niza**

Dužina niza se veoma lako može dobiti svojstvom **length**.

Npr: array.lenght;

**Indeks i broj elemenata**

Potrebno je obratiti pažnju na to da indeks niza nije isto što i broj elemenata niza, što često ume da zbuni. U ovom slučaju, broj elemenata niza (length) je šest, iako je poslednji indeks niza zapravo 5, a sve zbog računanja indeksa nizova od nule.

U upravo prikazanom primeru možete videti još jedan detalj koji potvrđuje da su nizovi u Javi objekti. Naime, nad promenljivom koja čuva referencu na niz pristupa se svojstvu length, što nesumnjivo potvrđuje da je reč o objektu. Prosti tipovi ne mogu imati svojstva.

**Prolazak kroz niz**

Kada se govori o nizovima u programiranju, veoma često se može čuti fraza prolazak kroz niz. Reč je zapravo o operaciji koja podrazumeva sukcesivno čitanje svakog elementa pojedinačno, i to uglavnom redom.

Prolazak kroz niz se najčešće obavlja kada je potrebno:

* sprovesti određene intervencije nad svakim od elemenata niza;
* pojedinačno ispisati svaki od elemenata jednog niza.

Osnovni pristup za postizanje prolaska kroz niz jeste upotreba for petlje, koja je obrađena u jednom od prethodnih modula. Evo kako može izgledati primer prolaska kroz niz korišćenjem for petlje:

A screen shot of a computer program

Description automatically generated

U prikazanom primeru, for petlja je iskorišćena za ispis vrednosti svakog elementa niza, pa se na izlazu dobija ovakav ispis:

3  
5  
2  
6  
1  
6

U okviru radnog okruženja, sada možete datom nizu celih brojeva array menjati dužinu i elemente, kako biste uvežbali korišćenje **for** petlje za prolazak kroz niz i ispis njegovih elemenata.

Pored for petlje, Java jezik poseduje još jednu petlju, koja je specijalno namenjena za prolazak kroz nizove. Reč je o petlji koja se naziva unapređeni for (enhanced for):

A black and green text

Description automatically generated with medium confidence

Unapređeni for poseduje sintaksu ilustrovanu slikom ispod:

A black background with text and arrows

Description automatically generated

S obzirom na to da je specijalno namenjen za prolazak kroz nizove, unapređeni for ne poseduje delove za deklarisanje brojača, definisanje uslova i naredbu koja se izvršava nakon svake iteracije. Umesto takvih delova koji karakterišu standardnu for petlju, unapređeni for unutar zagrada poseduje samo dva dela:

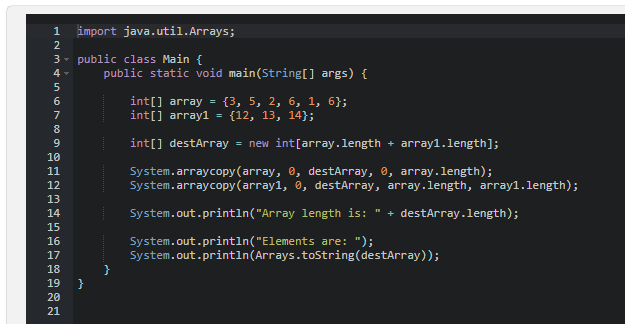
* deklaraciju promenljive koja će u svakoj iteraciji da dobija vrednost narednog elementa niza (u primeru je to int value); naziv promenljive je moguće samostalno definisati, dok tip zavisi od tipa elemenata niza;
* referencu na niz (u primeru je to array, što je naziv promenljive koja čuva referencu na niz kroz koji je potrebno proći).

Dva upravo opisana segmenta unapređene for petlje odvajaju se karakterom dve tačke.

Unapređeni for se često drugačije naziva i **for each**.

**Kopiranje niza**

Već je rečeno da nakon kreiranja  niza njegovu veličinu nije moguće menjati. To praktično znači da nizu nakon kreiranja nije moguće dodeliti nove članove. Ukoliko se takva potreba ipak javi, rešenje može da predstavlja kreiranje novog niza koji bi imao sve elemente starog i nekoliko novih elemenata koje želimo da dodamo.



Primer započinje deklaracijom i inicijalizacijom dva niza – array i array1. Zatim se obavlja kreiranje još jednog, trećeg niza (destArray), čija se veličina postavlja na zbir prva dva niza. Reč je o nizu unutar koga će biti objedinjeni svi elementi prva dva niza, pa se upravo zbog toga njegova dužina postavlja na zbir prva dva niza.

Kako bi se novi niz (destArray) popunio vrednostima koje se nalaze u prva dva niza, koristi se jedna ugrađena metoda Java platforme – arraycopy.

**Metoda arraycopy**

arraycopy je statička metoda klase System, koja se može koristiti za kopiranje vrednosti iz jednog niza u neki drugi. Reč je o metodi koja poseduje ovakav potpis:

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

Metoda arraycopy prihvata pet parametara:

* Object src – izvorišni niz;
* int srcPos – početni indeks izvorišnog niza;
* Object dest – odredišni niz;
* int destPos – početni indeks u odredišnom nizu, gde će početi smeštanje elemenata iz izvorišnog niza;
* int length – broj članova koji će biti kopirani.

U prikazanom kodu, metoda arraycopy koristi se dva puta, odnosno po jednom za kopiranje svakog od početnih nizova u odredišni destArray niz. Na kraju se može reći da se na ovaj način obavlja neka vrsta sabiranja, odnosno nadovezivanja nizova.

**Sortiranje niza**

Sortiranje je još jedna od veoma čestih operacija koje se sprovode nad nizovima. U nastavku će biti prikazan primer sortiranja sledećeg niza:

int[] array = {3, 5, 2, 6, 1, 6};

Sortiranje ovog niza može se obaviti na sledeći način:

int[] array = {3, 5, 2, 6, 1, 6};

Arrays.sort(array);

System.out.println(Arrays.toString(array));

Nakon izvršavanja prikazanog koda, na izlazu se dobija:

[1, 2, 3, 5, 6, 6]

Za obavljanje sortiranja u prikazanom primeru se koristi metoda sort ugrađene klase Arrays.

U okviru radnog okruženja, sada možete testirati i sortiranje niza. Datom nizu celih brojeva array možete menjati dužinu i elemente, kako biste dobili različite rezultate ispisa.

**Pretraga niza**

U okviru ugrađene klase Arrays postoji i metoda za pretragu nizova. Reč je o metodi koja pretražuje određeni niz u potrazi za definisanim elementom. Evo kako izgleda primer pretrage niza:

int[] a = {5, 1, 3, 4, 2};

Arrays.sort(a);

System.out.println(Arrays.binarySearch(a, 2));

System.out.println(Arrays.binarySearch(a, 10));

Prvo je bitno da primetite da se nakon deklaracije i inicijalizacije niza obavlja njegovo sortiranje, korišćenjem već prikazane metode sort. Jednostavno, metoda za pretragu koja je iskorišćena u nastavku zahteva da niz bude sortiran pre nego što se obavi pretraga. Za obavljanje pretrage, u prikazanom primeru se koristi metoda binarySearch.

**Metoda binarySearch**

Metoda binarySearch nalazi se unutar klase Arrays. Reč je o statičkoj metodi, koja poseduje sledeći potpis:

public static int binarySearch(int[] a, int key)

Metoda binarySearch prihvata dva parametra:

* prvi parametar se odnosi na niz koji je potrebno pretražiti; nije neophodno da to bude niz čiji su članovi int tipa – metoda binarySearch može da pretraži nizove bilo kog tipa;
* drugi parametar predstavlja ključ pretrage, odnosno vrednost koju je potrebno potražiti unutar niza.

Ukoliko metoda binarySearch unutar niza pronađe traženu vrednost, kao svoj rezultat ona isporučuje poziciju (indeks) na kojoj se nalazi tražena vrednost. Ukoliko vrednost ne postoji unutar niza, isporučuje se negativna vrednost. Takva povratna vrednost se dobija kada se dužina niza, uvećana za jedan, pretvori u negativan broj.

Nakon upoznavanja sintakse metode binarySearch, moći ćemo da razumemo povratne vrednosti koje dobijamo:

1  
-6

Vrednost 1 znači da prva tražena vrednost (2) postoji unutar niza i da ima indeks 1. Vrednost -6 znači da druga tražena vrednost (10) ne postoji unutar niza. Na izlazu se dobija -6 zato što je 6 dužina koju bi niz imao da unutar njega postoji i traženi element. Pretvaranjem takvog broja u negativni broj, dobija se -6.

**Višedimenzionalni nizovi**

Nizovi u Java jeziku nisu ograničeni samo na proste tipove, već je unutar njih moguće smestiti i referentne, odnosno složene vrednosti. To sve na kraju znači da je unutar jednog niza moguće smestiti više drugih nizova. U takvoj situaciji se dobijaju nizovi sa više dimenzija.

A screen shot of a computer screen

Description automatically generated

Na slici 18.4. je prikazana struktura jednog višedimenzionalnog niza sa četiri elementa. Pri tome je svaki od elemenata zaseban niz (svi oni su predstavljeni plavom bojom). Tako se, na primer, može reći da je prvi element niza, na poziciji 0, zapravo još jedan niz. S obzirom na to da su i sami elementi nizovi, i oni poseduju svoje indekse.

A screenshot of a computer

Description automatically generated

Sada su pored indeksa glavnog niza prikazani i indeksi pojedinačnih elemenata. Na taj način, četvrti element drugog elementa glavnog niza jednoznačno je određen indeksima [1, 3].

Ovakvi višedimenzionalni nizovi sačinjeni su iz dve dimenzije, pa se zbog toga drugačije nazivaju dvodimenzionalni nizovi. Ipak, višedimenzionalni nizovi mogu imati proizvoljan broj dimenzija (dve, tri, četiri...).

A screenshot of a computer game

Description automatically generated

Na slici je prikazan jedan višedimenzionalni niz sa tri dimenzije. On ima tri elementa (0, 1 i 2), od kojih je svaki po jedan dvodimenzionalni niz.

Kreiranje višedimenzionalnih nizova veoma je lako i ne zahteva korišćenje bilo kakve specijalne sintakse. U nastavku će biti prikazano nekoliko takvih primera:

int[][] matrix

= {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

};

Ovo je primer jednog dvodimenzionalnog niza. Kako bi Java izvršno okruženje znalo da je reč o nizu sa dve dimenzije, nakon tipa niza, postavljaju se dva para uglastih zagrada – **[][]**.

Kako bi se pristupilo elementu jednog ovakvog niza, koristi se sledeći pristup:

System.out.println(matrix[0][0]);

**Odvojena deklaracija i inicijalizacija višedimenzionalnih nizova**

U upravo prikazanom primeru, niz je odmah inicijalizovan vrednostima, koje su navedene unutar vitičastih zagrada. Višedimenzionalni niz se može kreirati i bez inicijalizacije:

int[][] arr = new int[3][4];

arr[0][0] = 1;

arr[0][1] = 2;

arr[1][0] = 3;

arr[1][1] = 4;

System.out.println(arr[0][1]); // 2

U okviru radnog okruženja, sada možete testirati kreiranje višedimenzionih nizova i ispis njihovih članova.

Kao što je nešto ranije rečeno, višedimenzionalni nizovi nisu ograničeni na dve dimenzije. To znači da je vrlo lako moguće uvesti novu dimenziju:

A white background with black and green text

Description automatically generated

Ovo je sada niz sa tri dimenzije, koji ima dva člana. To su zapravo dva dvodimenzionalna niza. Svaki dvodimenzionalni niz poseduje po četiri pojedinačna jednodimenzionalna niza.

Na kraju, prilikom kreiranja višedimenzionalnih nizova, broj dimenzija se ne mora zaustaviti na tri. Moguće je napraviti i nizove sa četiri i više dimenzija. Ipak, ovakvi nizovi su teški za razumevanje, a njihova upotreba retka.

**Brojevi**

Odabir odgovarajućeg tipa broja zavisi od dva faktora:

* veličine broja koji je potrebno prikazati;
* tipa broja, odnosno toga da li je potrebno rukovati celim brojem ili brojem sa decimalom.

A white background with black text

Description automatically generated

**Korišćenje karaktera donja crta prilikom definisanja numeričkih vrednosti**

Za razdvajanje grupa cifara u okviru jedne brojčane vrednosti može se koristiti karakter donja crta. Tako nešto može poboljšati preglednost i čitljivost koda. Donja crta neće biti tretirana od strane prevodioca, te će broj imati istu vrednost koju je imao i bez donjih crta. Evo primera upotrebe donje crte u brojčanim vrednostima:

A black and white text

Description automatically generated

Donja crta se može postaviti samo između cifara. Na sledeće pozicije se ne može postaviti donja crta:

* na početak ili kraj broja;
* pored decimalne tačke;
* pre **F** ili **L** sufiksa.

**Složeni numerički tipovi**

Pored prostih tipova, u Java programima se brojevi mogu predstaviti i korišćenjem složenih tipova, odnosno objekata nekoliko ugrađenih klasa. Takve klase se veoma često nazivaju i omotači primitivnih tipova, a u Javi ih ima šest, odnosno za svaki prosti tip po jedna:

* Integer
* Short
* Byte
* Float
* Long
* Double

Svi složeni numerički tipovi imaju identičnu roditeljsku klasu. Reč je o klasi Number.

A diagram of a number

Description automatically generated

Unutar klase Number objedinjuju se osnovne funkcionalnosti koje su zajedničke za sve upravo prikazane klase numeričkih tipova. U jednoj od prethodnih lekcija, omotači prostih tipova su nam pomogli da razumemo osnovne osobine složenih tipova podataka, a u nastavku ove lekcije upoznaćemo se sa njihovim najznačajnijim osobinama, koje ih izdvajaju od primitivnih numeričkih tipova.

**Boxing i unboxing**

Numerička vrednost se korišćenjem ugrađenih složenih tipova može definisati baš kao da je reč o vrednosti prostog tipa:

Integer myNumber = 16;

Ipak, Integer je jedan od složenih tipova u Java jeziku. Drugim rečima, Integer je klasa, a iz prethodnih lekcija znate da se objekti klasa kreiraju korišćenjem ključne reči new, nakon koje se navodi poziv konstruktorske metode. Upravo ovako:

Integer myNumber = new Integer(16);

Funkcionalnost jezika koja omogućava da se promenljivama Integer tipa vrednosti dodeljuju kao da je reč o prostim tipovima naziva se **Boxing**:

Integer myNumber = 16;

Ovo je naredba koja ilustruje boxing. Boxing je zapravo proces kojim se vrednost prostog tipa konvertuje u odgovarajući tip složenog omotača. Kada Java kompajler dođe do ovakve naredbe, on u pozadini automatski poziva konstruktor Integer klase i njemu prosleđuje primitivnu vrednost.

Potpuno očekivano, suprotan proces od upravo predstavljenog boxinga naziva se **unboxing**. Unboxing se odnosi na konvertovanje složenog tipa omotača nazad u primitivni tip. Drugim rečima, unboxing se odnosi na proces izvlačenja primitivne vrednosti iz objektnog tipa omotača.

Unboxing se takođe obavlja automatski od strane Java kompajlera i to u sledećim situacijama:

* kada se objekat omotača prosleđuje nekoj metodi koja zahteva vrednost prostog tipa;
* kada se objekat omotača koristi za postavljanje vrednosti neke promenljive primitivnog tipa.

Evo najprostijeg primera unboxinga:

Integer myNumber = 16;

int myPrimitiveNumber = myNumber;

U ovom slučaju, Java kompajler samostalno izvlači prostu vrednost iz myNumber objekta i nju dodeljuje promenljivoj myPrimitiveNumber.

**Najznačajnije funkcionalnosti numeričkih omotača**

Osnovna prednost numeričkih omotača jeste postojanje određenih funkcionalnosti koje je moguće koristiti nad numeričkim vrednostima. Najznačajnije takve funkcionalnosti koje su dostupne nad instancama prikazane su tabelom ispod:

A close up of a text

Description automatically generated

Pored ovih metoda koje se mogu pozvati nad vrednostima, složeni tipovi omotača poseduju i određen broj statičkih metoda koje su dostupne nad svim nešto ranije spomenutim klasama (Integer, Double, Float...). Takve funkcionalnosti ilustrovane su tabelom ispod:

A close up of text

Description automatically generated

Primeri korišćenja upravo prikazanih statičkih metoda izgledaju ovako:

System.out.println(Integer.compare(17, 16));

System.out.println(Integer.sum(17, 16));

System.out.println(Integer.max(17, 16));

System.out.println(Integer.min(17, 16));

Rezultat ovakvog koda je sledeći:

1  
33  
17  
16

**Klasa Math**

Pored funkcionalnosti koje postoje unutar ugrađenih klasa koje predstavljaju omotače primitivnih tipova, za rad sa brojevima je moguće koristiti i metode koje se nalaze u jednoj specijalnoj ugrađenoj klasi. Reč je o klasi Math.

A screenshot of a computer

Description automatically generated

Evo kako funkcioniše metoda round za zaokruživanje vrednosti:

System.out.println(Math.round(5.4));

System.out.println(Math.round(5.5));

System.out.println(Math.round(5.6));

Na početku, prikazane su tri naredbe u kojima se koristi metoda round. U svakoj naredbi, ovoj metodi se prosleđuje nešto drugačija vrednost. Metoda round zaokružuje vrednost na najbliži ceo broj. Stoga su vrednosti koje se dobijaju sledeće:

5

6

6

Naredna metoda za zaokruživanje sa kojom ćemo se upoznati je metoda floor:

System.out.println(Math.floor(5.4));

System.out.println(Math.floor(5.5));

System.out.println(Math.floor(5.6));

Na ovaj način se dobija sledeće:

5.0  
5.0  
5.0

S obzirom na to da zaokružuje broj na prvi manji, metoda floor u svim situacijama proizvodi vrednost 5.0.

Nešto drugačije funkcioniše metoda ceil koja zaokruzuje vrednost na prvu vecu:

System.out.println(Math.ceil(5.4));

System.out.println(Math.ceil(5.5));

System.out.println(Math.ceil(5.6));

Ovoga puta se na izlazu dobija:

6.0  
6.0  
6.0

**Tekst**

Prilikom kreiranja Java programa, podjednako važni kao i brojevi su podaci u tekstualnom obliku. Sa njima smo se već susretali u dosadašnjem toku ovog kursa, a u lekciji pred vama dodatno ćemo proširiti naše znanje o rukovanju tekstualnim podacima.

**String**

U programskom jeziku Java, tekstualni podaci se predstavljaju korišćenjem tipa String. String je jedan od složenih, odnosno referentnih tipova podataka, koji je predstavljen klasom String. Ipak, baš kao i složeni tipovi koji predstavljaju omotače prostih tipova, i String se može koristiti kao da je reč o prostom tipu.

Objekti tipa String zapravo su sekvence karaktera, pa tako predstavljaju nadgradnju primitivnog tipa char. Jednostavno, uvek je zgodnije raditi sa kolekcijom karaktera koji se ponašaju kao jedna celina nego sa svakim karakterom pojedinačno.

**Kreiranje stringova**

Stringovi se mogu kreirati kao da su vrednosti prostog tipa:

String message = "Hi!";

Ovakva naredba ilustruje deklarisanje promenljive tipa String sa nazivom message i vrednošću Hi!. Iako tip String nije prost tip, u jeziku Java je dozvoljeno promenljivoj tipa String vrednost dodeljivati kao da je reč o prostom tipu. Ovakvo kreiranje String vrednosti se naziva **implicitno kreiranje**.

Kao što je slučaj i sa bilo kojim drugim objektom, String se može kreirati i upotrebom ključne reč new nakon koje sledi poziv konstruktora:

String word = **new** String("Hi!");

U ovoj naredbi obavlja se kreiranje objekta tipa String upotrebom ključne reči new i prosleđivanjem vrednosti konstruktoru. Ovakvo kreiranje String vrednosti naziva se **eksplicitno kreiranje**.

**Čuvanje String vrednosti unutar memorije**

Postoji bitna razlika između dva upravo prikazana načina kreiranja String vrednosti. Kod implicitnog kreiranja, koje je prvo predstavljeno, Java koristi specijalni mehanizam za čuvanje tekstualnih vrednosti. Tekstualne vrednosti definisane implicitno (bez konstruktora) čuvaju se u specijalnom memorijskom odeljku koji se naziva **String Pool**.

String Pool je poseban odeljak unutar dinamičkog dela memorije (Heap), koji je specijalno rezervisan za String vrednosti. Java virtuelna mašina prvo proverava da li unutar String Poola već postoji identična tekstualna vrednost. Ukoliko postoji, u novu promenljivu se upisuje referenca na postojeću vrednost. Ukoliko vrednost ne postoji unutar String Poola, ona se unutar njega upisuje i promenljiva dobija referencu, odnosno njenu adresu. Najznačajnija osobina ovakvog mehanizma ogleda se u tome da se identične tekstualne vrednosti dele između promenljivih, kako bi se uštedeo memorijski prostor. Drugim rečima, ukoliko su definisane dve promenljive tipa String sa istom vrednošću, one će deliti jedan isti sadržaj u okviru String Poola.

Sa druge strane, objekti tipa String koji se kreiraju upotrebom ključne reči new i konstruktora smeštaju se direktno unutar dinamičkog dela memorije (Heap), baš kao i svaki drugi objekat. Za njih ne postoji String Pool, kao što je to slučaj sa String-ovima koji se kreiraju kao prosti tipovi.

A diagram of a conversation

Description automatically generated

Primer sa slike 20.1. ilustruje čuvanje pet String promenljivih unutar memorije. Pritom, svaka od takvih 5 String promenljivih poseduje identičnu tekstualnu vrednost – Hello. Prve tri promenljive kreirane su po ugledu na proste tipove, odnosno direktnom inicijalizacijom. Promenljive s4 i s5 ilustruju eksplicitno kreiranje String vrednosti, zato što podrazumevaju upotrebu ključne reči new i konstruktorske metode.

Zbog implicitnog načina kreiranja, promenljive s1, s2 i s3 čuvaju referencu na identičnu vrednost koja se nalazi unutar String Poola. Sa druge strane, iako imaju identičnu tekstualnu vrednosti, promenljive s4 i s5 referenciraju različite memorijske lokacije unutar dinamičkog (Heap) dela memorije.

Zbog ovakve osobine String vrednosti, one se nikada ne porede korišćenjem standardnog operatora jednakosti:

String s4 = new String("Hello");

String s5 = new String("Hello");

System.out.println(s4 == s5);

Na prvi pogled potpuno neočekivano, ovakav kod će proizvesti vrednost false, što znači da tekstualne vrednosti predstavljene promenljivama s4 i s5 nisu jednake. Možemo da vidimo da one jesu identične, ali zbog upravo prikazanog načina na koji se String vrednosti smeštaju unutar memorije, ovakve promenljive pokazuju na dva podatka koja unutar dinamičkog dela memorije imaju različite adrese.

**Nadovezivanje stringova**

Pojam nadovezivanja stringova već je spominjan u prethodnim lekcijama ovog kursa. Reč je o mogućnosti dodavanja teksta na postojeću String vrednost. Do sada smo se upoznali sa nadovezivanjem koje se postiže upotrebom operatora za sabiranje (+):

String myString = "My ";

String myString1 = "string";

System.out.println(myString + myString1);

Drugi način za nadovezivanje String-ova predstavlja korišćenje jedne od ugrađenih metoda String klase. Reč je o metodi concat:

String myString = "My ";

String myString1 = "string";

System.out.println(myString.concat(myString1));

Oba pristupa proizvode identičan efekat, ali se ne smatraju posebno efikasnim rešenjima. Naime, korišćenjem oba pristupa, nadovezivanjem se u pozadini obavlja kreiranje potpuno novog String objekta, čiji sadržaj čine dva nadovezana String-a. To na kraju praktično znači da se čestim nadovezivanjem u memoriji gomila velika količina String objekata, što predstavlja lošu praksu.

**Stringovi su nepromenljivi**

Upravo opisano ponašanje direktno je uslovljeno jednom veoma važnom osobinom String vrednosti. String vrednosti su u Java jeziku nepromenljive (***immutable***). To znači da se nakon kreiranja String objekta njegovo stanje više ne može menjati:

String myString = "My ";

myString.concat("string");

System.out.println(myString);

U ovom primeru, prvo je kreirana jedna implicitna String promenljiva. Zatim je na njenu vrednost, korišćenjem metode concat, nadovezana još jedna String vrednost. Na kraju je vrednost promenljive myString ispisana na izlazu:

My

Jasno je da se sa vrednošću promenljive myString nije dogodilo ništa. Jednostavno, metoda concat kao svoj rezultat emituje novi objekat tipa String, dok objekat na koji pokazuje promenljiva myString ostaje nepromenjen. Naravno, razlog je nepromenljivost String-ova u Javi.

Znatno efikasniji pristup za nadovezivanje stringova jeste korišćenje jedne specijalne klase, koja se naziva **StringBuilder**. Njenim korišćenjem drastično se smanjuje potrošnja memorije prilikom nadovezivanja stringova i povećava preglednost koda. Naredni primer daje rezultat identičan prethodnim, samo uz upotrebu StringBuilder klase:

StringBuilder sb = new StringBuilder();

sb.append("My ");

sb.append("string");

System.out.println(sb);

**Metode klase String**

Već smo rekli da tip podataka String u programskom jeziku Java predstavlja složeni tip podataka, odnosno objektni tip. Ovo omogućava da se nad promenljivama ovoga tipa mogu pozivati razne metode za obradu vrednosti ovih promenljivih. U narednom delu ove lekcije biće prikazane neke od najčešće korišćenih metoda za obradu tekstualnih podataka:

* **toCharArray() –**pretvara String u niz karaktera;
* **length() –**vraća dužinu String-a u karakterima;
* **toUpperCase() –**pretvara sva slova String-a u velika;
* **toLowerCase() –**pretvara sva slova String-a u mala;
* **trim() –**uklanja prazne karaktere sa početka i kraja String vrednosti;
* **indexOf() –**pronalazi indeks na kome se nalazi neki karakter ili tekst;
* **charAt() –**vraća karakter koji se nalazi na prosleđenoj poziciji;
* **substring() –**kreira isečak String-a;
* **equals() –**poredi dve String vrednosti;
* **startsWith() –**proverava da li String započinje određenim skupom karaktera;
* **endsWith() –**proverava da li se String završava određenim skupom karaktera;
* **replace() –**vršu zamenu određenih karaktera u String-u;
* **split() –**deli String na više delova.

**Softverske greške i izuzeci**

Greške su sastavni deo gotovo svakog softverskog proizvoda. Vrlo je teško pronaći složeniji program bez greške koja se u nekim situacijama može ispoljiti. Takođe, tokom razvoja i testiranja softverskih proizvoda vrlo često se detektuje i otkloni veliki broj softverskih grešaka. Sa druge strane, neke greške nikada ne budu detektovane i otklonjene.

Greške mogu prouzrokovati brojne probleme prilikom izvršavanja nekog programa. Simptomi mogu varirati od neočekivanog rezultata izvršavanja, preko korumpiranih podataka, pa sve do potpunog otkaza i nemogućnosti korišćenja kompletnog sistema. Stoga je razumevanje pojma softverskih grešaka i načina za njihovo otklanjanje od presudnog značaja prilikom razvoja softvera.

**Softverske greške**

Greška prilikom izvršavanja nekog programa drugačije se naziva **bug**.

Softverska greška jeste bilo koji nedostatak ili otkaz u kompjuterskom programu koji proizvodi netačno ili neočekivano ponašanje. Iz ugla modernih programskih jezika, pa samim tim i Java programiranja, softverske greške se mogu podeliti u tri grupe:

* **sintaksne greške** (*syntax errors*) – greške koje nastaju usled nepoštovanja sintaksnih pravila jezika;
* **greške tokom izvršavanja programa** (*runtime errors*) – greške koje kompajler ne detektuje, a koje se mogu ispoljiti tek tokom izvršavanja programa;
* **logičke greške** (*logic errors*) – greške do kojih dolazi tokom izvršavanja programa, ali okruženje ni na koji način ne signalizira njihovo postojanje; stoga je ovo najkompleksniji tip softverskih grešaka.

**Sintaksne greške**

Sintaksne greške nastaju usled nepoštovanja sintaksnih pravila jezika koji se koristi. Ovakvu vrstu grešaka je uglavnom najjednostavnije detektovati i otkloniti, pošto je program sa sintaksnom greškom potpuno nefunkcionalan i ne može se prevesti. To praktično znači da prevođenje Java programa sa sintaksnom greškom neće biti moguće. Tokom prevođenja, kompajler detektuje sve sintaksne greške i na jasan način nas obaveštava o njihovom postojanju. Evo jednog takvog primera:

A computer code with black text

Description automatically generated

Ovo je primer jednog vrlo jednostavnog Java programa, koji na izlazu ispisuje poruku Hello World. Ipak, prikazani program poseduje jednu sintaksnu grešku, zbog koje ne može da bude preveden. Možete li da je detektujete?

Unutar prikazanog programa, naredba za ispis poruke na izlazu nije završena karakterom za završetak naredbe – tačkom i zapetom. Stoga, prikazani kod nije sintaksno ispravan i prevođenje neće biti moguće. Prilikom prevođenja, od Java kompajlera se dobija sledeća poruka:

C:\java\_programs\JavaProgram\src\JavaProgram.java:7:42  
java: ';' expected

Evo upravo sada možete da vidite zbog čega se kaže da su sintaksne greške najjednostavnije za detekciju i otklanjanje. Program nije moguće prevesti, pa samim tim ni pokrenuti, a Java kompajler signalizira u čemu je problem. Tako dobijamo poruku da unutar klase JavaProgram postoji sintaksna greška na 42. karakteru 7. linije koda. Takođe, dobija se i dodatna poruka da se na navedenoj poziciji unutar našeg izvornog koda očekuje karakter tačka i zapeta.

**Greške tokom izvršavanja programa**

Ukoliko Java program ne poseduje sintaksne greška, njegovo prevođenje i pokretanje se obavlja bez problema. Ipak, to ne znači da tokom izvršavanja programa ne može doći do pojave grešaka. Evo primera jednog Java programa koji poseduje grešku koja će do izražaja doći tek tokom izvršavanja:

A white background with black and green text

Description automatically generated

U prikazanom programu, obavlja se deklarisanje jednog niza sa četiri celobrojne vrednosti. Zatim se u narednoj naredbi postavlja vrednost osmog člana niza (člana sa indeksom 7). S obzirom na to da u Java jeziku nizovi imaju fiksnu veličinu, ovako nešto proizvodi grešku tokom izvršavanja programa. Jednostavno, kompajler ne poseduje ugrađene mehanizme za detektovanje ovakve vrste grešaka. Zbog toga se ovakav kod prevodi i započinje izvršavanje od strane Java virtuelne mašine. Onoga trenutka kada izvršavanje dođe do naredbe za dodelu vrednosti osmom članu niza, JVM na izlazu emituje sledeću poruku:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 4  
at JavaProgram.main(JavaProgram.java:8)

**Greške prilikom izvršavanja programa čine da program odmah završi svoje izvršavanje**

Veoma bitna osobina grešaka do kojih dolazi tokom izvršavanja programa je da one čine da program momentalno zaustavlja svoje izvršavanje. Drugim rečima, dolazi do otkaza programa. To znači da će sve one naredbe koje se nalaze nakon naredbe sa greškom ostati neizvršene. U ovu konstataciju se veoma lako možemo uveriti. Dovoljno je da postavimo jednu naredbu za ispis jednostavne poruke odmah nakon naredbe koja može da proizvede grešku:

numberC = numberA / numberB;

System.out.println("Hello World");

Prilikom pojave greške, moći ćete da vidite da se naredba za ispis poruke Hello World uopšte neće izvršiti.

**Pojam izuzetka**

Pojava greške tokom izvršavanja Java programa propraćena je stvaranjem jednog veoma bitnog pojma. Reč je o pojmu izuzetka (exception). Drugim rečima, prilikom pojave greške tokom izvršavanja programa, Java izvršno okruženje kreira jedan objekat koji se naziva izuzetak i koji se koristi da reprezentuje grešku do koje je došlo.

S obzirom na to da su izuzeci reprezentacija grešaka do kojih dolazi tokom izvršavanja programa, Java platforma poseduje veliki broj ugrađenih klasa koje se koriste za predstavljanje različitih vrsta grešaka do kojih može doći prilikom izvršavanja programa.

A diagram of a system

Description automatically generated

**Zbog čega su nam potrebni izuzeci?**

Već je rečeno da su izuzeci objekti koje kreira okruženje kako bi moglo da artikuliše greške do kojih dolazi tokom izvršavanja programa. Ipak, osim poruke izuzetka, koja nam govori do kakve greške je došlo tokom izvršavanja programa, mi se još uvek nismo upoznali sa najznačajnijom osobinom izuzetaka. Reč je o mogućnosti njihove obrade.

Obrada izuzetaka podrazumeva adekvatno rukovanje objektima izuzetaka, čime se sprečava prekid izvršavanja Java programa. Proces obrade izuzetaka podrazumeva korišćenje tri specijalne naredbe Java jezika:

* try
* catch
* finally

Svaka od tri upravo prikazane naredbe poseduje svoje specifično zaduženje. Ipak, prikazane naredbe se nikada ne koriste samostalno, već se pribegava njihovom kombinovanju, pa se tako dobijaju:

* try..catch
* try..catch...finally

**try...catch**

Naredba try...catch sačinjena je iz pojedinačnih naredbi try i catch. Svaka od ovih naredbi ujedno je i zaseban blok koda. Tako naredba try...catch ima sledeći oblik:

try {

//block of code to try

}

catch (Exception err) {

//block of code to handle exceptions

}

Unutar bloka koji se definiše try naredbom navodi se kod koji može da proizvede izuzetak. Ukoliko tokom izvršavanja koda unutar try bloka dođe do izuzetka, aktivira se blok koda definisan catch naredbom.

Uvidom u sintaksu try...catch naredbe, može se videti da catch blok prihvata jedan parametar koji je u primeru imenovan kao err. Ovo je promenljiva koja se od strane izvršnog okruženja popunjava objektom izuzetka kada dođe do pojave greške. Naziv err je proizvoljan i može se menjati.

Evo kako se try...catch može upotrebiti na nešto ranije prikazanom primeru inicijalizacije nepostojećeg člana niza:

int[] nums = new int[4];

try {

nums[7] = 10;

} catch (Exception exc) {

System.out.println("There is an error.");

}

Naredba koja je proizvodila grešku, pa samim tim i izuzetak, sada je smeštena unutar try bloka. Unutar catch bloka je definisan kod koji će se aktivirati ukoliko do izuzetka dođe. Na kraju, efekat programa će biti ovakav:

There is an error.

Na izlazu dobijamo poruku da je došlo do greške prilikom izvršavanja programa. Ipak, što je najznačajnije, program ne prekida svoje izvršavanje kada dođe do izuzetka. U to se vrlo lako možemo uveriti:

int[] nums = new int[4];

try {

nums[7] = 10;

} catch (Exception exc) {

System.out.println("There is an error.");

}

System.out.println("Hello World");

Nakon postojećeg koda, dodali smo još jednu naredbu za ispis poruke na izlaz:

There is an error.  
Hello World

Iako u programu dolazi do pojave greške, on bez problema nastavlja svoje izvršavanje, sve do kraja. Ovo je najznačajnija osobina obrade izuzetaka.

**Čitanje informacija izuzetka**

U prikazanom primeru, obavljena je najosnovnija obrada izuzetka, koja je sprečila zaustavljanje programa. Ipak, obrada izuzetaka omogućava mnogo više, pa je tako moguće doći do različitih informacija o samom izuzetku:

int[] nums = new int[4];

try {

nums[7] = 10;

} catch (Exception exc) {

System.out.println("Exception class: " + exc.getClass());

System.out.println("Exception message: " + exc.getMessage());

}

Sada smo, umesto jednostavne poruke koju smo samostalno formirali, zapravo prvi put iskoristili objekat izuzetka kako bismo kroz njega došli do nekih informacija o grešci. Na izlazu smo ispisali naziv klase i poruku izuzetka:

Exception class: class java.lang.ArrayIndexOutOfBoundsException  
Exception message: Index 7 out of bounds for length 4

**Različiti tipovi izuzetaka**

U upravo prikazanom primeru, prvi put smo iskoristili objekat izuzetka koji se našem programu isporučuje kao promenljiva koja postoji unutar catch bloka. Ipak, bitno je da primetite da je ta promenljiva trenutno tipa Exception. Nešto ranije ste mogli da vidite da je Exception jedna od osnovnih klasa svih izuzetaka. To na kraju znači da je njome moguće predstaviti veliki broj različitih izuzetaka do kojih može doći prilikom izvršavanja Java programa. Ipak, kada tačno znamo do kog izuzetka može doći prilikom izvršavanja neke funkcionalnosti, moguće je definisati i specijalizovani tip izuzetka:

try {

nums[7] = 10;

} catch (ArrayIndexOutOfBoundsException exc) {

System.out.println("Exception class: " + exc.getClass());

System.out.println("Exception message: " + exc.getMessage());

}

Tip promenljive koja predstavlja izuzetak sada je promenjen na ArrayIndexOutOfBoundsException, što je specijalizovani tip izuzetka koji i dobijamo u upravo prikazanom primeru. Stoga će efekat biti identičan kao u prethodnom primeru.

Prilikom samostalnog definisanja tipa izuzetka, potrebno je voditi računa. Naime, ukoliko navedemo specijalizovani tip izuzetka, a prilikom izvršavanja programa dođe do emitovanja izuzetka nekog drugog tipa, izuzetak neće biti obrađen:

int[] nums = new int[4];

try {

nums[7] = 10;

} catch (ArithmeticException exc) {

System.out.println("Exception class: " + exc.getClass());

System.out.println("Exception message: " + exc.getMessage());

}

U ovom primeru, obrađujemo izuzetak tipa ArithmeticException. Međutim, u programu dolazi do pojave izuzetka drugog tipa, pa je efekat kao da obrada izuzetka nije ni definisana.

Prilikom obrade izuzetaka, moguće je definisati i veći broj tipova izuzetaka koji će biti obrađeni:

try {

nums[7] = 10;

} catch (ArithmeticException | ArrayIndexOutOfBoundsException exc) {

System.out.println("Exception class: " + exc.getClass());

System.out.println("Exception message: " + exc.getMessage());

}

Sada su prilikom kreiranja catch bloka definisana dva tipa izuzetaka koji će moći da budu obrađeni. Klase izuzetaka se razdvajaju karakterom uspravna crta.

Sada u okviru radnog okruženja možete testirati i prikazane primere iz ovog odeljka lekcije.

**try...catch...finally**

Pored try i catch naredbi, nešto ranije je spomenuta još jedna naredba koja se može koristiti u postupku obrade izuzetaka. Reč je o naredbi finally:

try {

block of code to try

}

catch (Exception err) {

block of code to handle errors

}

finally {

block of code to always execute

}

Kod definisan finally blokom uvek se izvršava, bez obzira na to da li je došlo do greške ili ne.

**Samostalno emitovanje izuzetaka**

Da sada smo videli nekoliko primera obrade izuzetaka koje Java okruženje automatski kreira prilikom pojave greške tokom izvršavanja Java programa. Ipak, moguće je i samostalno vršiti njihovo emitovanje i to korišćenjem naredbe throw. Sintaksa ove naredbe je:

throw expression;

Naredba throw formira se korišćenjem ključne reči throw, nakon koje se navodi objekat izuzetka koji je potrebno emitovati. Evo kako može da izgleda jedna metoda koja emituje izuzetak:

static void errorMethod() {

Exception exc = new Exception("There is an error");

throw exc;

}

Unutar tela metode errorMethod obavlja se kreiranje jednog objekta tipa Exception. Konstruktoru klase Exception prosleđuje se tekst greške. Ovakav izuzetak se zatim emituje korišćenjem naredbe throw. Ipak, kako bi ovakav primer bio u potpunosti funkcionalan, neophodno je obaviti još jednu intervenciju nad našom metodom errorMethod. Naime, sve metode koje emituju izuzetke moraju biti obeležene korišćenjem specijalne ključne reči **throws**:

static void errorMethod() throws Exception {

Exception exc = new Exception();

throw exc;

}

Ovo je sada potpuno validna i sintaksno ispravna metoda koja emituje izuzetak iz svog tela. S obzirom na to da iz svog tela emituje izuzetak, ona se sada ne može pozvati bez koda za obradu izuzetka:

try {

errorMethod();

} catch (Exception e) {

System.out.println(e.getMessage());

}

S obzirom na to da proizvodi izuzetak, metodu errorMethod je neophodno pozvati unutar try bloka. Nakon emitovanja izuzetka, on biva uhvaćen od strane catch bloka. Do poruke izuzetka koju smo nešto ranije kreirali dolazi se korišćenjem metode getMessage, pa se na izlazu dobija:

There is an error

**Proveravani i neproveravani izuzeci**

U dosadašnjem toku lekcije spominjali smo samo klasu Exception, kao jednu od korenih klasa svih ostalih klasa izuzetaka u Java jeziku. Ipak, pored ove klase postoji jedan još generalniji tip izuzetaka. Reč je o klasi Throwable.

A diagram of a diagram

Description automatically generated

Klasa Throwable osnovna je klasa svih ostalih klasa izuzetaka. Kao što vidite, nju direktno nasleđuju dve klase:

* Error
* Exception

Klasa Error koristi se za reprezentaciju ozbiljnih problema tokom izvršavanja programa. Stoga nije predviđeno da izuzeci koji nasleđuju ovu klasu budu obrađeni, jer se smatra da sistem ne može da nastavi svoj rad nakon pojave takve vrste greške. Neke od najpoznatijih klasa kojima se predstavljaju takve ozbiljne greške u sistemu su OutOfMemoryError, StackOverflowError, InternalError...

Klasa Exception, kao što ste mogli da vidite u dosadašnjem toku ove lekcije, koristi se za predstavljanje grešaka koje ne ugrožavaju izvršavanje programa, pa je stoga predviđeno da budu obrađene od strane programera, kako bi program mogao da nastavi svoje izvršavanje.

Izuzeci tipa Error drugačije se nazivaju neproveravani, a izuzeci tip Exception proveravani izuzeci.

Na kraju, sve ovo znači da bi se nasleđivanjem klase Error stvorio jedan izuzetak koga ne bi bilo potrebe eksplicitno obrađivati korišćenjem try...catch blokova.

**Samostalno kreiranje klasa izuzetaka**

Prethodni primer ilustrovao je samostalno emitovanje izuzetka koji je kreiran na osnovu osnovne ugrađene klase Exception. U realnim okolnostima najčešće se pribegava samostalnom kreiranju klasa izuzetaka. Evo kako može izgledati jedna klasa izuzetka koji mi samostalno kreiramo:

public class OutOfRangeException extends Exception {

@Override

public String getMessage() {

return "The number is out of range!";

}

}

Klasa koju smo kreirali ima naziv OutOfRangeException. Ona nasleđuje osnovnu klasu proveravanih izuzetaka – Exception. Takođe, unutar klase OutOfRangeException obavili smo redefinisanje metode getMessage koja se nalazi unutar roditeljske klase. Na ovaj način smo definisali poruku koja će moći da se dobije kada se nad objektom izuzetka pozove metoda getMessage.

Ovako kreiran izuzetak sada se može upotrebiti na sledeći način:

static void calculate(int num) throws OutOfRangeException {

if (num > 100 || num < 0) {

throw new OutOfRangeException();

}

//method logic

}

Izuzetak koji smo samostalno kreirali upotrebili smo unutar metode calculate. To je metoda koja prihvata jedan celobrojni parametar i koja na osnovu takvog parametra treba da obavi određenu kalkulaciju. Ipak, zamislite da za potrebe kalkulacije prosleđena vrednost treba da bude u opsegu od 0 do 100. Upravo zbog toga je unutar metode dodata logika za proveru opsega vrednosti.

Ukoliko vrednost nije u spomenutom opsegu, obavlja se emitovanje izuzetka koji smo upravo kreirali. Logika metode calculate je izostavljena kako bi metoda bila što jednostavnija i kako bismo se mogli usredsrediti na logiku za emitovanje izuzetka.

Na kraju, ovakva metoda se može pozvati na sledeći način:

A computer screen shot of a program

Description automatically generated

S obzirom na to da je prosleđena vrednost izvan definisanog opsega, na izlazu se dobija:

The number is out of range!

Ovo je poruka koju smo nešto ranije samostalno definisali unutar metode getMessage naše klase OutOfRangeException.

**Debug**

**Šta su logičke greške?**

Logičke greške su najkompleksnija vrsta grešaka do kojih može doći prilikom izvršavanja softvera. Za njih se kaže da su najkompleksnije zato što ih je najteže uočiti. Ova vrsta grešaka nastaje u situacijama u kojima program ne proizvodi očekivani rezultat, a izvršno okruženje ne prijavljuje bilo kakav izuzetak ili neregularnost. Drugim rečima, kod sa logičkom greškom ne obavlja ono što se od njega očekuje. Zbog nepostojanja bilo kakve poruke od izvršnog okruženja, detekcija i uklanjanje logičkih grešaka zahteva pažljivu analizu koda i poznavanje korišćenja nekog od alata za *debug*.

Na sreću, sva moderna razvojna okruženja poseduju takve alate, pa ni IntelliJ IDEA nije izuzetak. Stoga će veliki deo ove lekcije biti posvećen korišćenju *debug* alata koji je sastavni deo IntelliJ IDEA okruženja. Ipak, za početak, pogledajmo primer jednog Java programa koji poseduje logičku grešku.

A computer screen shot of a program code

Description automatically generated

Ovo je Java program koji od korisnika prihvata dva broja. Brojevi koje korisnik unese smeštaju se unutar promenjivih numberA i numberB. Program zatim utvrđuje koji od dva uneta broja je veći i obaveštenje prikazuje korisniku.

Kada ovakav program prevedete i pokrenete, neće se dogoditi ništa neuobičajeno. Možete probati da unesete različite ulazne podatke i program će na izlaz emitovati poruke različitog sadržaja. Ipak, ukoliko se malo bolje udubite u poruke koje se dobijaju od programa, može se uočiti nekonzistentnost. Pogledajte o čemu je reč. U nastavku će biti ilustrovani različiti slučajevi korišćenja prikazanog programa, kako bi se detektovala logička greška.

**Slučaj 1**

Ulazni podaci: 13, 15

Poruka: Number 15 is greater than 13

**Slučaj 2**

Ulazni podaci: 15, 13

Poruka: Number 15 is greater than 13

**Slučaj 3**

Ulazni podaci: 5, 5

Poruka: Number 5 is greater than 5

Analizom navedenih slučajeva korišćenja može se dosta toga zaključiti o unutrašnjem izvršavanju programa.  U prvom slučaju, uneti su brojevi 13 i 15. Program obaveštava da je broj 15 veći od broja 13, na osnovu čega se može zaključiti da program uspešno funkcioniše. Drugi slučaj je sličan – ulazni podaci su identični, ali su uneti obrnutim redom. Program emituje identičnu poruku, pa se može zaključiti da i na osnovu *Slučaja 2* program funkcioniše korektno.

*Slučaj 3*pokazuje evidentan nedostatak programa. Uneti brojevi su 5 i 5, a program obaveštava da je broj 5 veći od broja 5.

Na ovom primeru se može videti baš ono što je rečeno na početku ove lekcije. Logičke greške ne proizvode izuzetak, ali čine da program ne emituje očekivani rezultat. I upravo je to ono što se događa u ovom primeru. Program nas obaveštava je broj 5 veći od broja 5, što svakako nije odgovarajući izlaz.

Upravo prikazana logička greška je jedna od jednostavnijih reprezentacija takve vrste grešaka i svakako je mogla da bude detektovana i pažljivijom opservacijom koda. Ipak, ukoliko još uvek ne uviđate u čemu je problem u našem programu, iskoristićemo jedan specijalizovani alat razvojnog okruženja koji će nam pomoći u detekciji logičke greške.

**Debugger**

Gotovo svako moderno razvojno okruženje unutar sebe poseduje i specijalizovani alat koji se naziva debugger. Na osnovu ovog naziva možete da naslutite da je reč o alatu koji pomaže u uklanjanju grešaka iz izvornog koda programa. Reč je zapravo o alatu koji omogućava pauziranje i kontrolisanje izvršavanja koda.

Unutar debuggera postoji nekoliko veoma važnih opcija:

* **Rerun** – ponovno pokreće program u debug režimu;
* **Resume** – ukoliko je program pauziran, nastavlja izvršavanje koda;
* **Pause** – zaustavlja izvršavanje koda neposredno pre naredbe koja treba da se izvrši sledeća;
* **Stop** – zaustavlja izvršavanje programa i napušta debug režim; okruženje se vraća u stanje u kome je bilo pre pokretanja programa u debug režimu.

Svakako, najznačajnija opcija je ona koja omogućava da se izvršavanje programa pauzira, čime se ulazi u poseban debug režim, koji omogućava da se izvršavanje koda eksplicitno kontroliše.

**Kontrola izvršavanja koda**

Nakon što se obavi pauziranje koda, postaje dostupno nekoliko veoma važnih opcija, kojima je moguće kontrolisati izvršavanje programa.

A screenshot of a computer

Description automatically generated

Na slici možete da vidite grupu opcija koje se koriste za kontrolisanje izvršavanja koda nakon što se izvršavanje pauzira. Opcije su obeležene sa *Step Actions*, zato što omogućavaju da se kroz izvorni kod prođe korak po korak (*step*), odnosno naredbu po naredbu.

A screen shot of a computer

Description automatically generated

* **Step Over** – prelazi na izvršavanje prve sledeće linije; ipak, ukoliko je sledeća linija poziv neke metode, takva metoda se izvršava u celosti, a izvršavanje se ponovo pauzira na prvoj naredbi nakon poziva metode;
* **Step Into** – prelazi na izvršavanje sledeće linije, ali ukoliko je sledeća linija poziv metode, takva metoda se ne izvršava u celosti, već se izvršavanje pauzira na prvoj naredbi unutar metode;
* **Force Step Into** – omogućava ispitivanje funkcionalnosti koje ne pripadaju direktno našem programu; drugim rečima, naš program može da koristi funkcionalnosti koje se nalaze u nekim drugim kompajliranim bibliotekama; ova opcija omogućava da se uđe u kod takvih funkcionalnosti kao da je reč o onima koje smo samostalno pisali;
* **Step Out** – ukoliko se izvršavanje nalazi unutar metode, izvršava se kompletan ostatak metode, a izvršavanje se pauzira na prvoj naredbi nakon poziva takve metode;
* **Drop Frame** – omogućava da se prilikom izvršavanja koda vratimo unazad, odbacivanjem izvršavanja metode koja se trenutno izvršava; ova opcija omogućava da ponovo ispitamo izvršavanje metode u kojoj smo već bili;
* **Run to Cursor** – izvršava sve naredbe sve dok ne stigne do pozicije na kojoj se unutar editora trenutno nalazi kursor.

**Tačke prekida**

Upravo prikazani primer korišćenja IntelliJ IDEA debuggera imao je jednu olakšavajuću okolnost. Naime, jedna od prvih naredbi programa je i ona koja blokira izvršavanje programa sve dok korisnik ne unese neke vrednosti. Stoga smo imali praktično neograničeno vreme na raspolaganju da izvršavanje programa pauziramo i upoznamo se sa svim opcijama za kontrolisanje izvršavanja. Ipak, u programima koji nemaju ovakvih naredbi, odnosno naredbi koje blokiraju izvršavanje programa, pokretanje programa u debug modu ne bi se mnogo razlikovalo od regularnog pokretanja.

U to se možete uveriti ukoliko naredbe za preuzimanje podataka od korisnika zamenite direktnom inicijalizacijom promenljivih. Nakon toga, pokrenite program u debug modu i moći ćete da vidite da unutar Debug konzole dobijamo rezultat izvršavanja i da praktično nemamo vremena da stignemo da pauziramo izvršavanje programa.

Kako bi se i u ovakvim situacijama omogućilo pauziranje programa, pribegava se korišćenju tačaka prekida. Tako su tačke prekida još jedan način da se pauzira izvršavanje programa.

Tačka prekida se postavlja jednostavnim klikom na prostor koji se nalazi na levoj strani editora, odnosno između editora i brojeva koji predstavljaju linije koda. Kao potvrda da je tačka prekida uspešno postavljena, dobija se karakteristična crvena oznaka.

A screenshot of a computer program

Description automatically generated

Nakon postavljanja tačke prekida u primeru iz ove lekcije, potrebno je program pokrenuti u debug modu, isto kao i u prethodnom primeru. Ipak, ovoga puta će se izvršavanje zaustaviti u definisanoj tački prekida, a mi ćemo izvršavanje dalje moći da kontrolišemo korišćenjem već opisanih komandi.

Korišćenjem upravo prikazanih funkcionalnosti IntelliJ IDEA debuggera veoma lako možemo da utvrdimo u čemu je problem našeg programa. Jednostavno, prilikom pisanja programa nismo predvideli slučaj po kome bi ulazni podaci bili jednaki. S obzirom na to da korisnik ima mogućnost da unese dve identične vrednosti, neophodno je i da program poseduje odgovarajuću logiku za obradu takvih ulaznih podataka. Drugim rečima, program mora za bilo koji unos da proizvede odgovarajući izlaz:

A computer screen shot of a program code

Description automatically generated