



Distance Learning System

# Core Java Programming

Kontrola toka programa

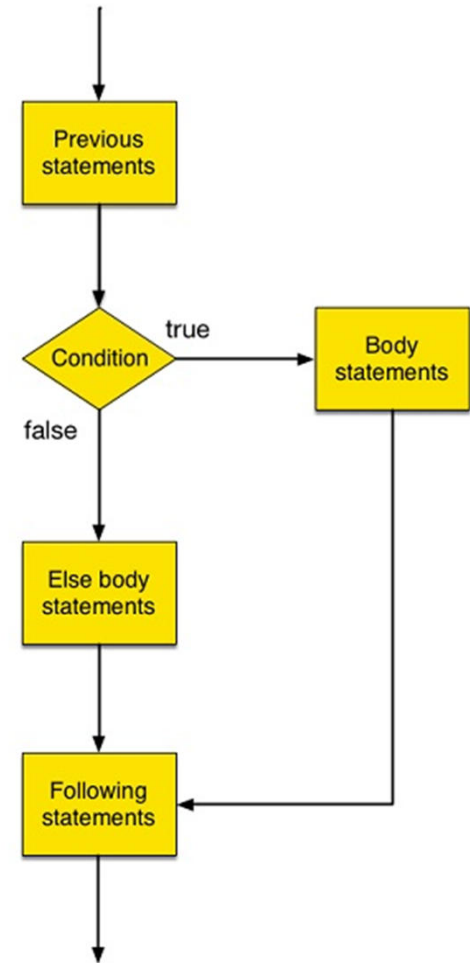
# Kontrola toka programa (jcex092014)

---

- Ni jedan program ne bi imao smisao ukoliko ne bi mogli da unutar njega donosimo neke odluke
- Donošenje odluka u programu utiče na njegov tok, te se ovaj proces naziva kontrolom toka programa
- U osnovne načine da se kontroliše tok jednog programa spadaju **grananje** i **petlje**
- Eventualno, u kontrolu toka mogu biti svrstane i strukture za kontrolu izuzetaka (try/catch blokovi)

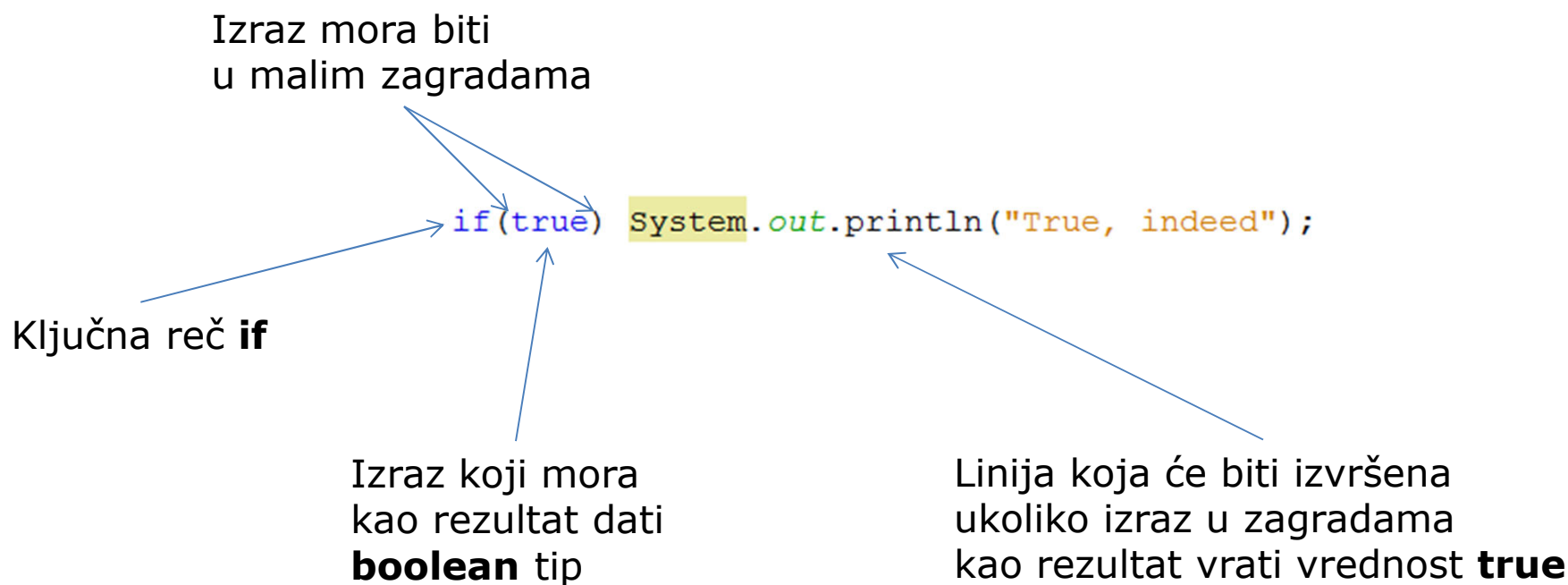
# If else struktura

- Osnovni način za kontrolu toka je **selekcija**
- Osnovna način za realizaciju selekcije u Javi je **if else** struktura
- Ova struktura funkcioniše tako što na osnovu rezultata izraza izvrši jedan ili drugi deo programa
- Za realizaciju se koriste naredbe **if** i **else** u kombinaciji sa vitičastim zagradama



# Implementacija if strukture

- If struktura, se implementira na sledeći način:



# Uslovljeni blokovi

---

- Prethodni primer daje mogućnost uslovnog izvršavanja jedne linije koda
- Linija može biti u nastavku if uslova, ili u sledećem redu, pa su tako sve tri varijante koje slede, ispravne:

```
if(true) System.out.println("True, indeed");
```

```
if(true)  
    System.out.println("True, indeed");
```

```
if(true)  
  
    System.out.println("True, indeed");
```

# Uslovljeni blokovi

- Šta da uradimo ako želimo da se izvrši više uslovljenih linija
- Ako pokušamo da izvršimo sledeći kod, dobićemo naizgled neočekivan rezultat

```
if(false)
```

```
    System.out.println("This is false");  
    System.out.println("Indeed");
```

The branch is never used  
----  
(Alt-Enter shows hints)

- Umesto da budu izvršene obe linije (što bi možda očekivali), izvršena je samo ona druga

```
run:
```

```
Indeed
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Ipak, ovaj epilog potpuno se uklapa sa prethodno objašnjenim ponašanjem if uslova: **uslov se odnosi samo na prvu sledeću liniju**

# Uslovljeni blokovi

- Ako hoćemo da uslovno izvršimo više od jedne linije, moramo upotrebiti **blok** koda (oivičiti ga vitičastim zagradama), koji želimo da izvršimo uslovno:

```
if(false) {  
    System.out.println("This is false");  
    System.out.println("Indeed");  
}
```

Kod je u bloku

- Ovako napisan uslov, biće uspešno izvršen, i ni jedna poruka neće otići na izlaz
- Okruženje će nas tokom pisanja ovog koda upozoriti na beskorisnost ovog koda (jednostavno if(false) znači da blok nikada neće biti izvršen
- Ako se podsetimo **bočnih efekata** iz prethodnih lekcija, ovo može biti važno upozorenje za nas, ali često, ovo je odličan način da neki deo koda brzo onemogućimo ili omogućimo u svrhu testiranja

The branch is never used  
----  
(Alt-Enter shows hints)

# Šta sve možemo testirati

- Glavni akteri u if uslovima su **operatori poređenja** i njihova kombinacija sa **logičkim operatorima**
- Znamo da operatori poređenja kao rezultat daju boolean tip, a ovo je upravo jedini dozvoljeni tip u if uslovu

```
int x = 10;
if(x==10){
    System.out.println("x is 10");
}
```

- U uslovu takođe možemo kombinovati operatore poređenja sa logičkim operatorima:

```
double health = 20;
boolean dead = false;
health-=25;
if(health<20&&!dead){
    dead = true;
}
```

```
String entered_username = "peter";
String entered_pass = "123";
String stored_username = "peter";
String stored_pass = "123";
if(entered_username==stored_username&&entered_pass==stored_pass){
    System.out.println("user is valid");
}
```

Upotreba operatora == nije baš sjajno rešenje za poređenje Stringova. Radije treba koristiti metod equals.



# Spajanje uslova ključnom rečju else

- Često ćemo hteti da izvršimo evaluaciju nekog stanja i da na osnovu te evaluacije uradimo nešto, a ponekad i promenimo samo stanje. Na primer, recimo da hoćemo da proverimo da li je naš igrač mrtav, i ako jeste, da ga oživimo i prikažemo poruku, a ako nije, da samo prikažemo poruku.
- Na osnovu onoga što do sada znamo, mogli bi napisati sledeći kod:

```
boolean hero_is_dead = true;
if(hero_is_dead){
    System.out.println("Hero is dead. Reviving");
    hero_is_dead = false;
}
if(!hero_is_dead){
    System.out.println("Hero is not dead. Nothing to do.");
}
```

- Unesite kod, analizirajte ga i objasnite zbog čega nije ispravan

# Spajanje uslova ključnom rečju else

- Kod sa prethodnog slajda nije ispravan, jer u svakom slučaju vrši obe provere, a obzirom da se u prvom uslovnom bloku menja status promenljive, oba uslova bivaju ispunjena, što ne bi smelo da se dogodi
- Ako hoćemo da prekinemo evaluaciju promenljive nakon prvog ispunjenog uslova, koristimo kombinaciju **if else**

```
boolean is_true = true;
if(is_true){
    System.out.println("it is true");
} else {
    System.out.println("it is not true");
}
```

Uslov je ispunjen  
i ništa se više  
ne proverava

Do ovog dela  
se ne stiže

Ovde je kraj

# Spajanje uslova ključnom rečju else

---

- Ako bi po istom principu modifikovali prvobitni kod, rešili bi problem

```
boolean hero_is_dead = true;
if(hero_is_dead){
    System.out.println("Hero is dead. Reviving");
    hero_is_dead = false;
} else {
    System.out.println("Hero is not dead. Nothing to do.");
}
```

- Problem kod ovog primera je što heroj može imati samo jedno od dva stanja, pa možemo da se zadovoljimo odgovorom na pitanje „da ili ne?“

# Ternarni operator

---

- Iako se naziva operatorom, ternarni operator nije to u bukvalnom smislu reči
- Ternarni operator je skraćenica za **if else** strukturu opisanu u prethodnom slajdu
- Ternarni operator uvek prosleđuje jednu od dve vrednosti na levu stranu i to po sledećem principu:

**USLOV ? VREDNOST UKOLIKO JE USLOV TACAN : VREDNOST UKOLIKO JE USLOV NETACAN**

- Implementacija ternarnog operatora izgleda ovako:

```
System.out.println(  
    hero_is_dead?"Hero is dead. Reviving":"Hero is not dead. Nothing to do."  
);
```

# Spajanje više uslova ključnom rečju else

- Šta bi se dogodilo ukoliko bi heroj mogao imati jedno od tri stanja: full\_health, critical\_health i dead. Recimo da su ta stanja označena brojevima 3, 2 i 1 respektivno.
- Činjenica da nakon **else** možemo izvršiti bilo koju liniju ili blok koda, znači da možemo izvršiti i samu novu if else strukturu

```
byte hero_state = 1;
if(hero_state==1){
    System.out.println("I am dead. Well... I am dead");
    hero_state = 3;
    System.out.println("Alive again");
} else
if(hero_state==2) {
    System.out.println("I am in critical condition. Not superhappy");
} else
if(hero_state==3){
    System.out.println("I am full, feel perfect");
}
```

Nakon ispunjenja uslova  
Evaluacija se prekida  
Izvršava se uslovljeni blok  
i program nastavlja odavde

# Gnježđenje if uslova

---

- If uslove možemo gnjezditi
- Iako ne postoji ograničenje u dubini gnježđenja, ako je ona previše velika, znači da nešto konceptualno nije u redu sa kodom

```
int x = 20;  
if(x<30)  
    if(x>10)  
        if(x<25)  
            if(x>15)  
                System.out.println("X is exclusively between 15 and 25");
```

- Uvek je bolja praksa što ranije usmeravanje određenog posla na odgovarajući deo sistema, nego korišćenje jednog dela sistema za više namena

# Vežba - kalkulator

---

- Potrebno je kreirati program koji će prikazati rezultat računske operacije na osnovu tri informacije koje u njega ulaze
  - String operator
  - double operand1
  - double operand 2)
- Rešenje: jcex092014 Calculator

# Vežba - Kartice

---

- Potrebno je kreirati program za naplatu kreditnih kartica
- U programu postoji metod koji prihvata objekat klase **Card**.
- Klasa **Card** ima **konstruktor** koji kao parametar prihvata inicijalno stanje na računu
- Klasa **Card** ima polja: **balance**, i metod **charge**.
- Metod charge proverava da li na kartici ima dovoljno novca (balance), i ukoliko ima, izvršava naplatu (smanjuje balance). Ukoliko nema, prikazuje korisniku da ne postoji dovoljno sredstava na računu, i otkazuje naplatu



# Switch / case struktura

---

- Kada se vrši evalulacija jedne promenljive, što je više puta rađeno u prethodnim slajdovima, if else nije najbolje rešenje. Mnogo češće, u ovu svrhu koristimo **switch / case** konstrukciju
- Switch case ne radi ništa što već ne možemo uraditi sa if else, već samo obrađuje specifičnu situaciju tretiranja jedne iste promenljive kroz više uslova
- Ključni elementi switch / case strukture su **switch** i **case**, ali takođe bitnu ulogu imaju i ključne reči (elementi) **break** i **default**.

# Gde možemo sresti switch / case

- Switch case se često sreće u proverama stanja

Gotovo svaka Windows aplikacija ima u sebi jednu switch case strukturu poput sledeće

```
switch( message )
{
    case WM_CREATE:
        return 0;
    case WM_PAINT:
    {
        HDC hdc;
        PAINTSTRUCT ps;
        hdc = BeginPaint( hwnd, &ps );
        EndPaint( hwnd, &ps );
    }
    return 0;
    case WM_KEYDOWN:
        switch( wParam )
        {
            case VK_ESCAPE:
                PostQuitMessage( 0 );
                break;
            default:
                break;
        }
        return 0;
    case WM_DESTROY:
        PostQuitMessage( 0 );
        return 0;
    break;
}
```

Web aplikacije često koriste switch case prilikom autentifikacije korisnika

```
switch ($?errno) {
    case 0:
        break;
    case 1:
        echo 'The username or password you entered is incorrect';
        break;
    default:
        echo 'Unknown error';
};
```



User Name

Password

Remember me ☐

Odabir između Always i Just once opcija na Androidu

Complete action using

	Chrome
	Google+

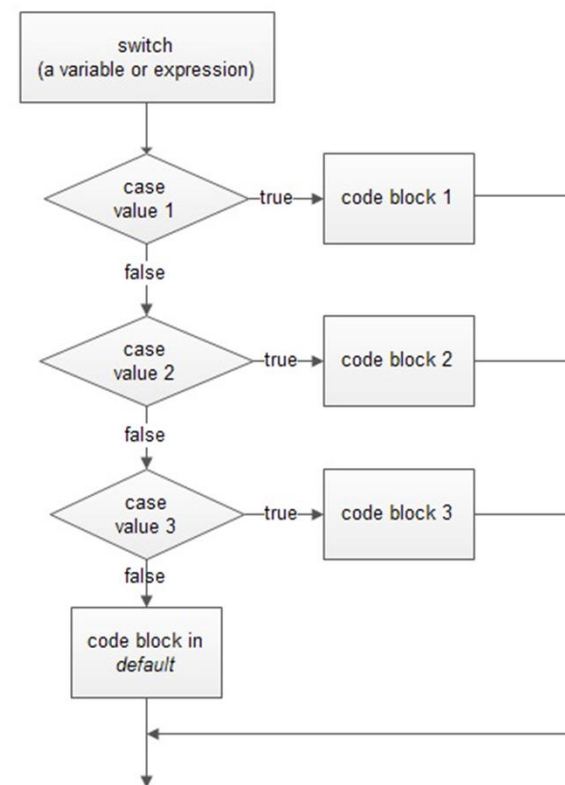
**Kontrola toka programa**

**LINKgroup**

# Switch / case sintaksa

- Sintaksa switch case strukture je:

```
switch (condition){  
  case value1:  
    some code  
  break;  
  case value2:  
    some code  
  break;  
  default:  
    some code  
  break;  
}
```



# Switch / case postavka

---

- Evaluacija promenljive hero\_state pomoću if else strukture

```
byte hero_state = 1;
if(hero_state==1){ System.out.println("Hero is dead"); }
else if(hero_state==2){ System.out.println("Hero is low on health"); }
else if(hero_state==3){ System.out.println("Hero is fully charged"); }
```

- Rešenje istog problema pomoću switch case strukture

```
switch(hero_state){
    case 1:
        System.out.println("Hero is dead");
        break;
    case 2:
        System.out.println("Hero is low on health");
        break;
    case 3:
        System.out.println("Hero is fully charged");
        break;
}
```

# Switch / case blokovi

---

- Kod switch case-a postoje važne osobenosti koje se moraju znati:
  - Vitičaste zagrade u switch case-u nisu opcione, već obavezne  
`switch(hero_state){ }` ← **Obavezno**
  - Switch može da radi sa celobrojnim vrednostima, karakterima i stringovima, ali ne i sa decimalnim brojevima, niti kompleksnim tipovima (osim wrapper-a za proste tipove (Integer, Byte...)). Takođe, String je moguće evaluirati tek od nedavno, pa, ukoliko radite sa verzijom Jave manjom od 7, evaluacija Stringa neće biti moguća
  - Break nije obavezan u strukturu, ali će, ukoliko ga ne postavimo, automatski biti izvršena i sledeća grana uslova

```
switch(hero_state){  
    case 1:  
        System.out.println("I will be executed");  
    case 2:  
        System.out.println("I will also be executed");  
}
```



# Switch / case blokovi

- Redosled case-ova nije bitan

```
hero_state = 5;
switch(hero_state){
    default: System.out.println("Default value"); break;
    case 2: System.out.println("I will also be executed"); break;
    case 1: System.out.println("I will be executed"); break;
}
```

- U switch strukturi, sve od oznake slučaja (case #: ) pa do ključne reči break, smatra se blokom. Moguće je i eksplicitno ovo označiti vitičastim zagradama, ali i bez njih, pomenuta zona biće tretirana kao blok:

**Oba koda daju isti rezultat**

```
switch(hero_state){
    case 1:
        System.out.println("Hero is dead");
        System.out.println("Really dead");
        System.out.println("Dead like...you know");
        break;
```

```
switch(hero_state){
    case 1: {
        System.out.println("Hero is dead");
        System.out.println("Really dead");
        System.out.println("Dead like...you know");
        break;
    }
```

# Vežba 1

---

- Pokušajte da izmenite prethodno kreirani primer kalkulatora tako da umesto if else, koristi switch case strukturu
- Rešenje: **jcex092014 SwitchCaseCalculator**

# Vežba 2

---

- U program ulaze sledeći podaci:

```
int x = 523134;  
int y = 325423;
```

- Potrebno je izvršiti proveru ostatka deljenja x i y. Ukoliko ne postoji ostatak, prikazati poruku da ostatak ne postoji, u suprotnom, proveriti da li je ostatak veći ili manji od hiljadu i dati odgovarajuću poruku.
- Rešenje **jcex092014 NoRemainderDivision**



# Vežba 3

---

- Dati su sledeći ulazni podaci za jedno vozilo:

```
String carMake = "Ford";  
int doors = 4;
```

- Potrebno je napraviti uslovnu konstrukciju koja će da proverava da li je proizvođač automobila Ford i ukoliko jeste, da u zavisnosti od broja vrata prikazuje odgovarajuću poruku.
- Rešenje **jcex092014 CheckCarConstructor**



Distance Learning System

# Core Java Programming

Petlje

# Petlje

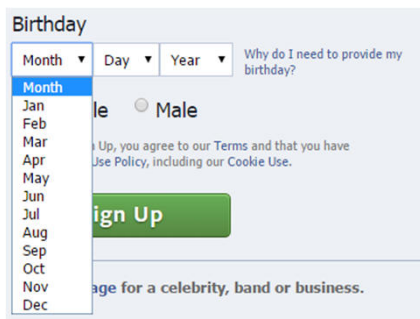
---

- Petlje su sistemi u programima koji nam omogućavaju da jednu stvar uradimo više puta
- Možemo je postići ugrađenim tehnologijama (**for**, **while**, **foreach**), a takođe i ručno, **rekurzijom**
- Ugrađene tehnologije za realizaciju petlje u Javi su:
  - **for** (kada znamo koliko puta hoćemo da ponovimo kod)
  - **while** (kada ne znamo koliko puta hoćemo da ponovimo kod)
  - **do while** (kada ne znamo koliko puta hoćemo da, ali moramo ga izvršiti bar jednom)
- Svaki put kada dođe do izvršavanja bloka koda u petlji, kaže se da je izvršena **iteracija** petlje

# Gde se sve koristi for petlja

- Kao i u slučaju if i switch uslova, ovo pitanje je toliko dvosmisleno da možemo jednostavno odgovoriti: **svugde**.

Padajući meni sa mesecima je kreiran pomoću petlje



Birthdays

Month Day Year Why do I need to provide my birthday?

Month

- Jan
- Feb
- Mar
- Apr
- May
- Jun
- Jul
- Aug
- Sep
- Oct
- Nov
- Dec

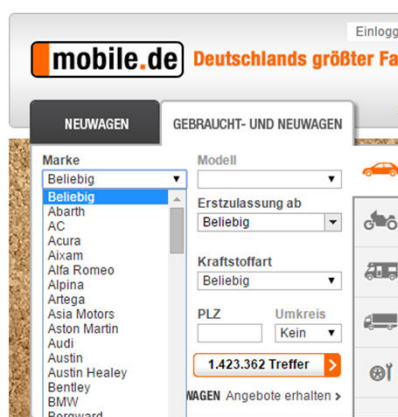
Sign Up

age for a celebrity, band or business.

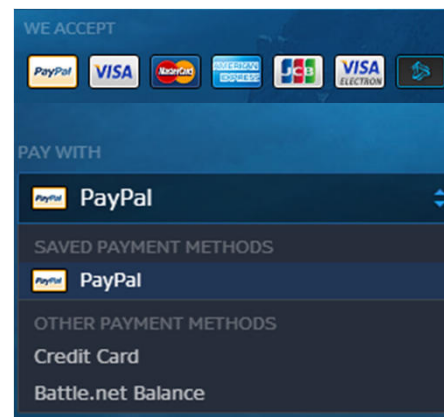
Lista quest-ova u igri je kreirana pomoću for petlje



Lista proizvođača je kreirana pomoću for petlje



Lista dostupnih payment provajdera kreirana je pomoću for petlje



For petlja

LINKgroup

# Gde se sve koristi for petlja

- Svi akteri na sceni, iscrtavaju se pomoću for petlje



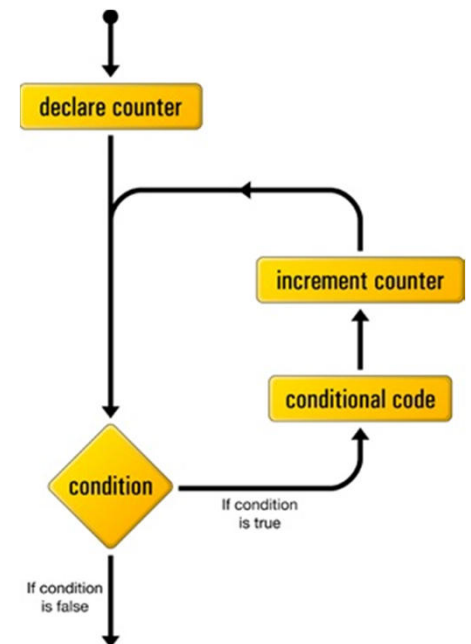
**For petlja**

**LINKgroup**

# For petlja

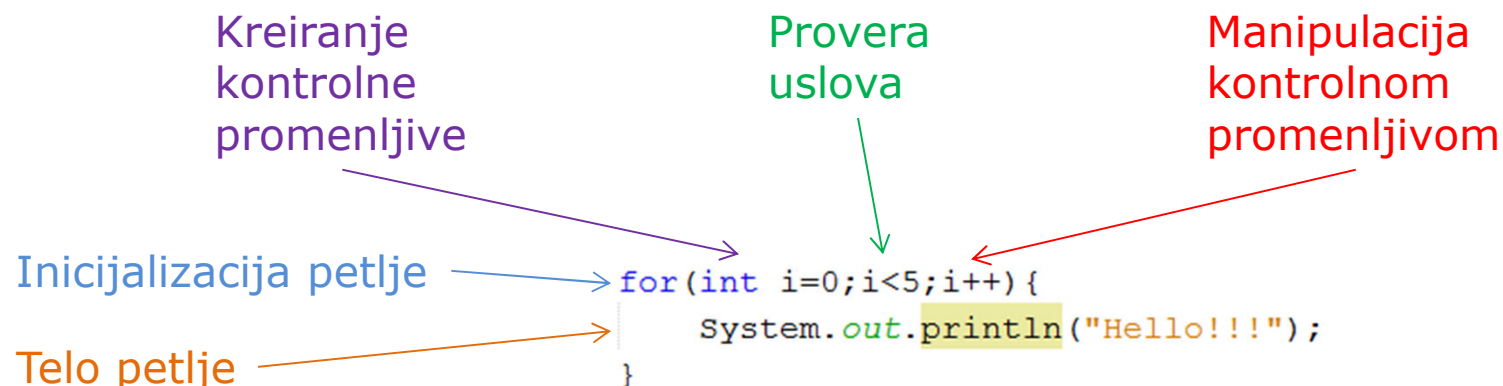
- For petlja je osnovna petlja u gotovo svim programskim jezicima
- Ona se smatra simbolom petlji
- For petlja se realizuje pomoću ključne reči **for**
- Izvršava se sve dok je određeni uslov ispunjen
- For petlja ima sledeću strukturu:

```
for(inicijalizacija;provera uslova;modifikacija){  
    kod petlje (sve ono što će se izvršiti više puta)  
}
```



# Implementacija for petlje

- For petlja se implementira na sledeći način:



- Primećujemo da se struktura deli na dve logičke celine: **inicijalizaciju** i **telo**

# Implementacija for petlje

- Za razliku od switch / case strukture, for petlja ne zahteva korišćenje bloka, pri čemu se kao i kod if else blokova, tada petlja odnosi samo na prvu sledeću liniju

```
for(int i=0;i<5;i++){  
    System.out.println("Hello!!!");  
}  
  
for(int i=0;i<5;i++){  
    System.out.println("Hello!!!");  
}  
  
for(int i=0;i<5;i++){  
    System.out.println("Hello!!!");  
    System.out.println("I am actually not part of loop");  
}
```

Biće tretirano u petlji

Neće biti tretirano u petlji



# Inicijalni deo for petlje

---

- Inicijalni deo for petlje se deli na tri celine odvojene oznakom ;
- Sadržaj svake od celina nije naročito bitan, sve dok poštujemo sintaksna pravila
- Na primer, svaka celina može čak biti i prazna, ali tada ćemo imati mrtvu petlju, koja će se večno izvršavati, kao u sledećem primeru:

```
for(;;){  
    System.out.println("Hello!!!");  
}
```

- Druga krajnost je da svaki deo inicijalizacije sadrži više izjava koje se tiču petlje

```
for(int i=0,j=10,u=25;i<20&&j<30&&u>15;i+=2,j+=4,u--){  
    System.out.println("i:"+i+" j:"+j+" u:"+u);  
}
```

- Ipak, u najvećem broju slučajeva, for petlja se koristi sa jednom kontrolnom vrednošću

```
for(int i=0;i<5;i++){  
    System.out.println("Hello!!!");  
}
```

# Život kontrolne promenljive for petlje

- Kontrolna promenljiva for petlje tiče se samo petlje u kojoj je definisana, tako da je moguće uraditi ovako nešto:

```
for(int i=0;i<5;i++) {  
    System.out.println("i is " + i);  
}  
for(int i=0;i<5;i++) {  
    System.out.println("i is " + i);  
}
```

- U obe petlje definiše se i, i ono se odnosi samo na tu petlju. Nakon završetka petlje, i nije vidljivo za spoljašnji svet.
- Ako hoćemo da se u inicijalizaciji petlje tretira promenljiva koja je vidljiva i u svetu izvan petlje, treba i da je definišemo izvan petlje (naravno sa posledicama u oba slučaja)

Ovo ne može

```
for(int i=0;i<5;i++) { }  
System.out.println("i is " + i);
```

Ovo može

```
int i;  
for(i=0;i<5;i++) { }  
System.out.println("i is " + i);
```

# Vežba

---

- Kreirati for petlju koja će prikazati tablicu množenja sa 1, 2 i 3 za brojeve od 1 do 10

```
*****
#      1      2      3
*****
1      1      2      3
2      2      4      6
3      3      6      9
4      4      8     12
5      5     10     15
6      6     12     18
7      7     14     21
8      8     16     24
9      9     18     27
10     10     20     30
```

# Gnježđenje petlje

---

- U petlji možemo kreirati bilo kakav kod, pa čak i samu petlju
- Ovo omogućava kreiranje takozvane petlje u petlji, odnosno, ugnježdene petlje

```
for(int i=0;i<10;i++){  
    for(int j=0;j<20;j++){  
        System.out.println("I is " + i + " and j is " + j);  
    }  
}
```

# Vežba

---

- Ako su date promenljive `w` i `h`

```
int w = 10;
int h = 20;
```
- Potrebno je kreirati program koji na izlazu kreira sledeću sliku,  
tako da je njena širina jednaka vrednosti promenljive `w`, a visina jednaka vrednosti promenljive `h`

```
#####
#000000000000000000#
#000000000000000000#
#000000000000000000#
#000000000000000000#
#000000000000000000#
#000000000000000000#
#000000000000000000#
#000000000000000000#
#####
```

# Foreach petlja

---

- Foreach petlja je varijacija for petlje, koja se koristi za rad sa iterabilnim objektima (nizovima ili listama objekata).
- Foreach u Javi ne postoji kao zasebna komanda, već se realizuje kroz specijalan oblik for petlje (**enchanced for**):

```
int[] numbers = {1,5,2,3,7};  
for(int val : numbers){  
    System.out.println(val);  
}
```

# ForEach petlja

---

- Da bi razumeli pun potencijal foreach i for petlje, neophodno je da budemo upoznati sa nizovima
- Realizacija foreach petlje bez nizova zahteva upoznavanje Iteratora, što je nešto komplikovanija varijanta, pa je najbolje proučavati je nakon savladavanja nizova
- Primer foreach petlje bez upotrebe kolekcija i nizova, dostupan je putem git-a pod nazivom: `jcx092014 ForeachLoop`

# While petlja (jcex092014 WhileLoop)

---

- While petlja je karakteristična po tome što je koristimo obično kada ne znamo koliko puta treba da se izvrši kod unutar nje
- While petlja nema sistem inicijalizacije kontrolnih promenljivih niti manipulacije njime. Onda jednostavno startuje neki kod sve dok je određeni uslov ispunjen. Ovo je opasna osobina, jer vrlo lako može dovesti do mrtve petlje. Ali sa druge strane, while vrlo često upravo i koristimo kako bi ostvarili mrtvu petlju



# Gde se sve koristi while / do while petlja

---

- While petlja ima veoma širok dijapazon upotrebe
- Sva grafička okruženja koja rade u realnom vremenu, sadrže po jednu ovakvu petlju
  - Igre
  - GUI operativnih sistema
- Takođe, u pozadini while petlju imaju: serveri, programi za striming (video, audio)...

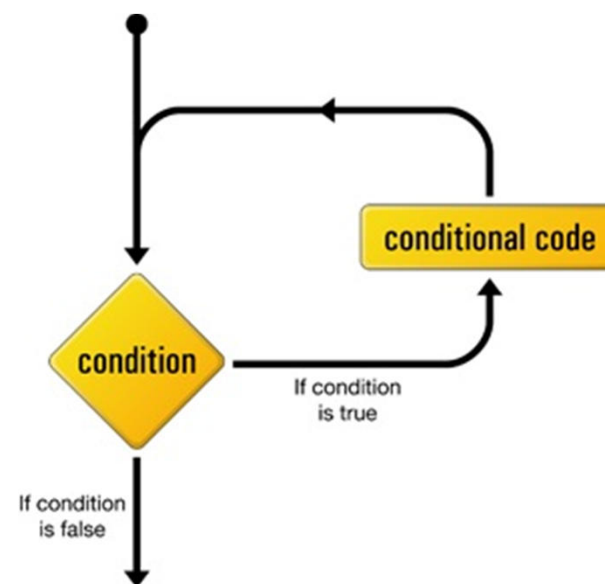
# Struktura while petlje

- Struktura while petlje sastoji se od ključne reči while, uslova u malim zagradama i uslovljenog koda

```
while(condition) {  
    conditional code  
}
```

- Kao i kod for petlje i if else strukture, vitičaste zagrade nisu obavezne

```
while(condition)  
    conditional code
```



# Kontrola while petlje

---

- While petlju moramo kontrolisati ručno
- Sami moramo kreirati kontrolnu promenljivu i sami moramo manipulaciju tom promenljivom tokom izvršavanja petlje

```
int number = 5;
while (number >= 2) {
    System.out.println(number);
    number--;
}
```

- Ovde treba biti izuzetno pažljiv, jer vrlo lako možemo izgubiti kontrolu nad petljom

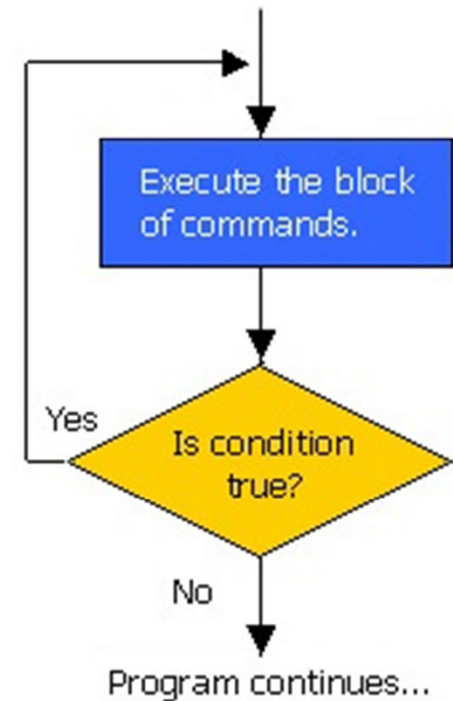
```
int number = 5;
while (number >= 2) {
    System.out.println(number);
}
```

# Do while

---

- Do while je samo varijacija while petlje
- Ona garantuje da će se telo petlje izvršiti bar jednom
- Struktura do while petlje je slična strukturi while petlje:

```
do {  
    code block  
} while(condition);
```



# Do while

---

- Sledeći kod zbog neispunjenja uslova, ignoriše telo while petlje:

```
number = 5;
while(number>5) {
    System.out.println("Number is bigger than 5");
}
```

- U sledećem kodu, poruka će biti prikazana, jer je prvo došlo do aktivacije tela, a tek onda do provere uslova:

```
do {
    System.out.println("Number is bigger than 5");
} while (number>5);
```

# Ključne reči **break** i **continue**

---

- Kada ručno kontrolišemo petlju, postoje dve stvari koje ćemo hteti da uradimo:
  - Da preskočimo iteraciju
  - Da prekinemo kompletnu petlju
    - Druga stavka ima i varijaciju – napuštanje spoljašnje petlje
- U ovu svrhu, koriste se dve ključne reči:
  - **break**
    - Momentarno napušta petlju prekidajući izvršavanje tela
  - **continue**
    - Momentarno prelazi na sledeću iteraciju petlje i prekida izvršavanje tela u aktuelnoj iteraciji

# Ključna reč break

---

- U trenutku kada program naiđe na ključnu reč break unutar while (ili for) petlje, prekinuće tu petlju
- Sledeći primer prekida while petlju ukoliko promenljiva number dobije vrednost 3

```
number = 0;
while (number<5) {
    if (number==3) {
        break;
    }
    System.out.println(number);
    number++;
}
```

# Prekidanje spoljašnje petlje

- Ako hoćemo da napustimo i spoljašnju petlju, tada moramo upotrebiti koncept imenovane petlje (labela)
- Ovaj koncept podrazumeva da imenujemo petlju oznakom IME: (myloop:,loopstart:,deadloop:,sometloop:...), a zatim da se vratimo na taj deo koda kada se za tim ukaže potreba, pomoću naredbe: break IME.
- Na primer:
  - break myloop;
  - break loopstart;
  - break deadloop;
  - break sometloop;

```
outerloop:
while(true){
    while(number<5){
        if(number==3){
            break outerloop;
        }
        System.out.println("Number: " + number);
        number++;
    }
}
System.out.println("Program continue");
```



# Ključna reč continue

---

- U trenutku kada program naiđe na ključnu reč continue unutar while (ili for) petlje, prekinuće izvršavanje tog bloka i preći na sledeću iteraciju
- Treba biti izuzetno pažljiv prilikom upotrebe continue jer lako može dovesti do pojave mrtve petlje

```
number = 0;
while (number < 5) {
    if (number == 3) {
        number++;
        continue;
    }
    System.out.println("Number: " + number);
    number++;
}
```

# Vežba

---

- Program zamišlja broj i pita nas koji je broj zamislio, zatim nam nudi unos sa tastature. Ukoliko pogodimo zamišljeni broj, računar nam potvrđuje, zamišlja novi broj i ponavlja proceduru
- Ukoliko ne pogodimo, program nas obaveštava da nismo pogodili i nudi mogućnost unosa novog broja:
- Pomoć:
- Da bi od računara dobili slučajan broj, možemo koristiti klasu Random:

```
Random rand = new Random();  
zamisljeniBroj = rand.nextInt(3);
```

 **Broj od 0 do 2**

```
Koji broj sam zamislio?  
Unesi broj: 1  
Nije  
Unesi broj: 6  
Nije  
Unesi broj: 2  
Nije  
Unesi broj: 3  
Jeste!  
Koji broj sam zamislio?  
Unesi broj:
```

# Vežba

---

- Treba napraviti program koji će iscrtavati oznaku \* u konzoli
- Oznaka \* treba da se kreće levo desno određenom brzinom menjajući smer kada dođe do određene tačke
- Za realizaciju treba upotrebiti while i for petlju
- Pomoć:
  - Da bi mogli da prepíšemo sadržaj u jednom redu konzole, koristimo konzolnu oznaku `\r` na kraju reda (`System.out.print("\r");`)
  - Ako hoćemo da program „malo odmori“ možemo napisati: `Thread.sleep(BROJ MILISEKUNDI);`
- Rešenje: **jcex092014 SimpleAnimation**
- Pokušajte da modifikujete aplikaciju tako da radi sa dve oznake, koje se različito ponašaju

# Rekurzija

---

- Rekurzija je specifičan način za ponovljeno izvršavanje jednog koda
- Ovo je veoma moćan alat, koji se veoma lako može oteti kontroli, te je potrebno veoma pažljivo rukovati njime
- Osnovni koncept rekurzije je pozivanje funkcije/metode od strane same sebe
- Struktura rekurzije izgleda ovako:

```
f() {  
    f();  
}
```

# Implementacija rekurzije

---

- Da bi implemenirali rekurziju, potrebno je da iz jedne metode pozovemo istu tu metodu

```
static void recursiveFunction() {  
    System.out.println("Hello from myself");  
    recursiveFunction();  
}  
  
public static void main(String[] args) {  
    recursiveFunction();  
}
```

- Treba biti pažljiv u samom startu. Java ima ograničenje u broju mogućih rekurzija, te će kod iznad izvesno rezultirati beskonačnom rekurzijom (infinite recursion), što će dalje dovesti do prepunjenja stack-a programa, i konačno, prekida njegovog izvršavanja

# Kontrola rekurzije

---

- Iako smo videli koliko može biti opasna, rekurzija je izuzetno moćna u kontrolisanim uslovima
- Rekurziju možemo (moramo) kontrolisati na sličan način kao i while petlju – pomoću kontrolnih promenljivih
- Da bi prekinuli rekurzivnu petlju, jednostavno treba da **ne** pozovemo metod
- Primer desno kontroliše rekurziju promenljivom ***recursive\_repeats*** i ne dozvoljava joj da se izvrši više od 10 puta.

```
static int recursive_repeats = 1;
static void recursiveFunction() {
    System.out.println("Hello from myself " + recursive_repeats);
    if(recursive_repeats++<10)
        recursiveFunction();
}
public static void main(String[] args) {
    recursiveFunction();
}
```

# Kontrola rekurzije

- Na prethodnom slajdu bila je korišćena globalna kontrolna promenljiva, ali često ćemo kontrolnu promenljivu ubacivati u pozvani metod kao parametar

```
static void recursiveFunction(int counter) {  
    System.out.println("Hello from myself " + counter);  
    if(counter-->1)  
        recursiveFunction(counter);  
}  
  
public static void main(String[] args) {  
    recursiveFunction(10);  
}
```

- Ovde takođe treba biti pažljiv, jer je funkcija ta koja ne zna šta se dešava u svetu oko nje, i možemo dosta pogrešiti prosleđivanjem neispravnog podatka

## Ispravno rešenje

```
static void recursiveFunction(int counter) {  
    System.out.println("Hello from myself " + counter);  
    if(counter>1)  
        recursiveFunction(--counter);  
}  
  
public static void main(String[] args) {  
    recursiveFunction(10);  
}
```

## Neispravno rešenje – „ubija“ aplikaciju

```
static void recursiveFunction(int counter) {  
    System.out.println("Hello from myself " + counter);  
    if(counter>1)  
        recursiveFunction(counter++);  
}  
  
public static void main(String[] args) {  
    recursiveFunction(10);  
}
```

# Vežba 1

- Pokušajte da ustanovite koliko puta će biti pozvan metod counter u sledećem primeru:

Pomoćna klasa

```
1 public class RecursionControl {  
    public int value;  
}
```

Metod

```
2 public static void counter(int start, RecursionControl stop)  
{  
    System.out.println("Hey, I am called!");  
    if(stop.value > start) {  
        return;  
    }  
    ++stop.value;  
    counter(start--, stop);  
}
```

Prvi poziv

```
3 RecursionControl stop = new RecursionControl();  
int start = 5;  
stop.value = 2;  
counter(start, stop);
```



# Vežba 2

---

Pokušajte bez kucanja programa, da odgonetnete koji će biti rezultat sledećeg programa na izlazu

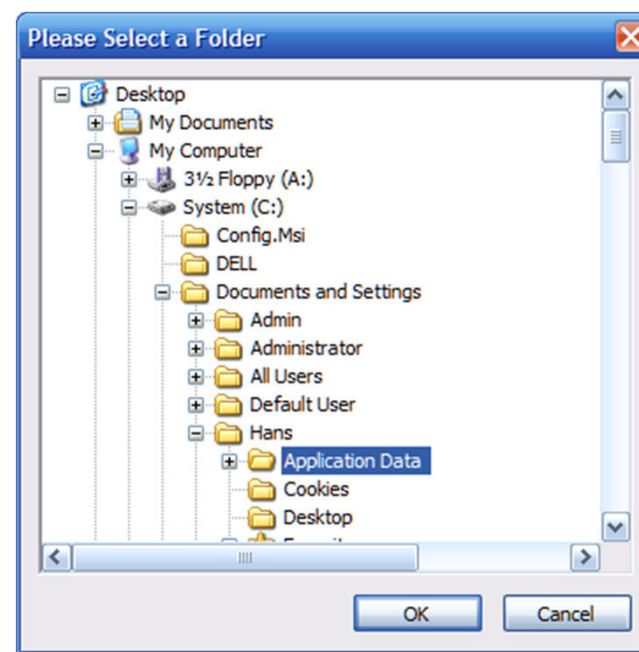
```
public static int happening(int num) {  
    if (num==1) {  
        return 1;  
    } else {  
        return happening(num-1)+num;  
    }  
}  
  
public static void main(String[] args) {  
    System.out.println(happening(4));  
}
```

# Gde sve možemo sresti rekurziju?

**Pathfinding** je najčešće baziran na rekurziji



**Struktura fajl sistema** se formira pomoću rekurzije

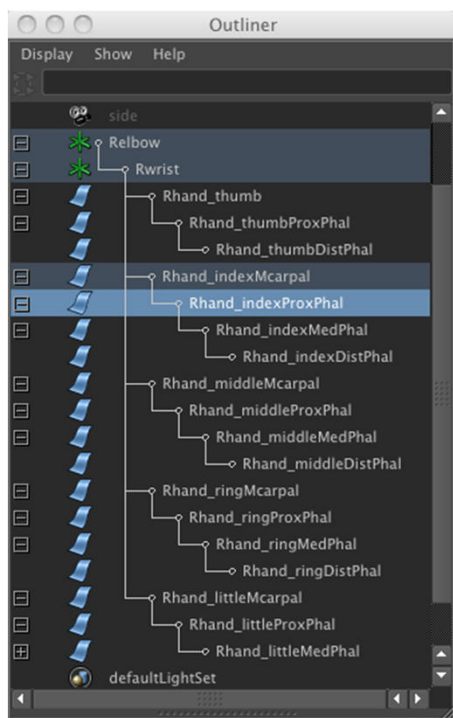


**Rekurzija**

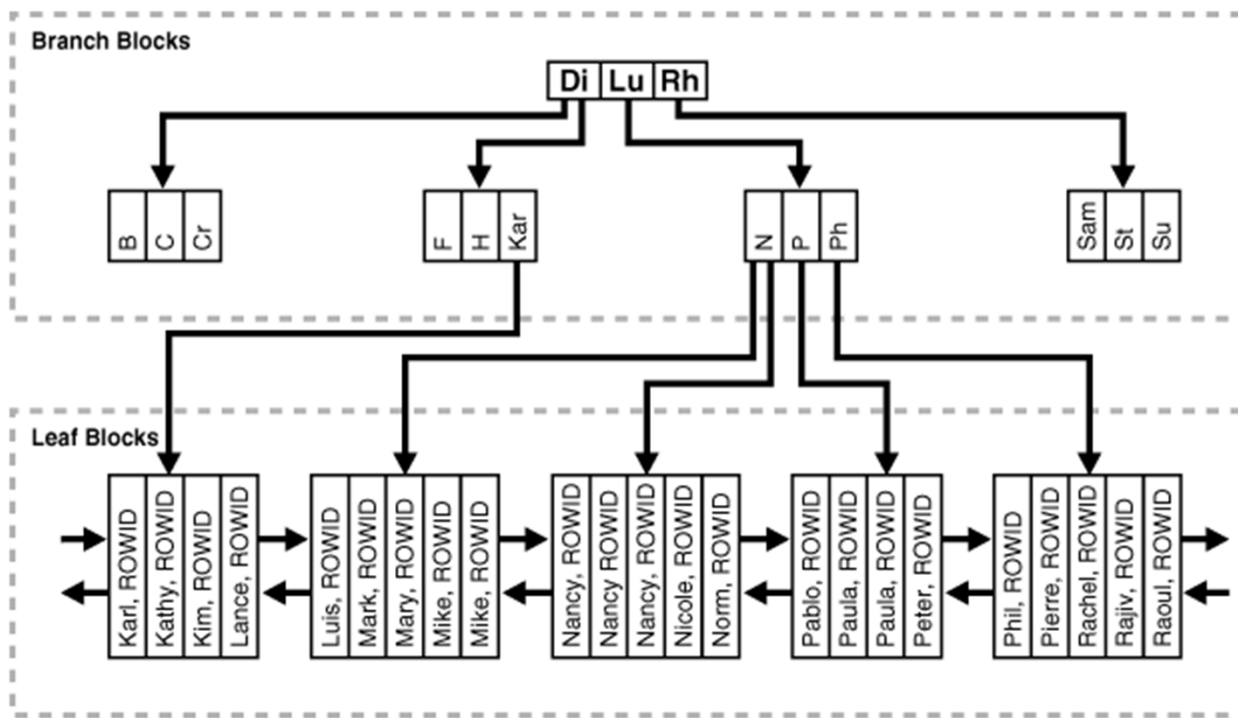
**LINKgroup**

# Gde sve možemo sresti rekurziju?

## Stabla scena



## Indeksi u bazi podataka



**Rekurzija**

**LINKgroup**