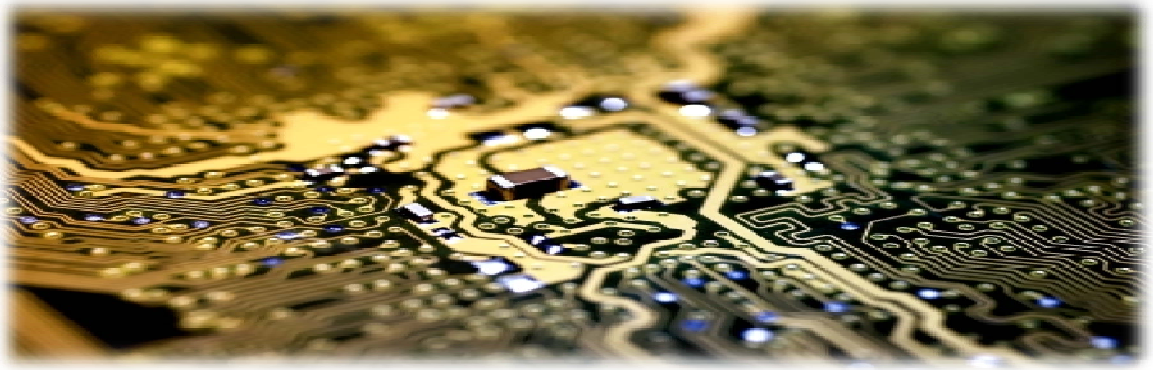
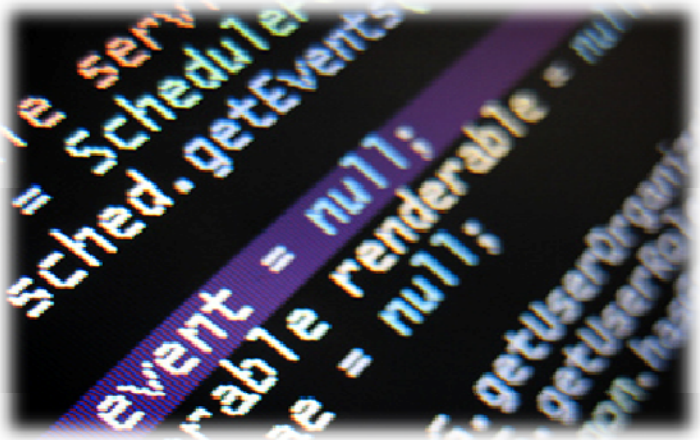
T-5 Osnovni principi izrade kvalitetnog programskog koda

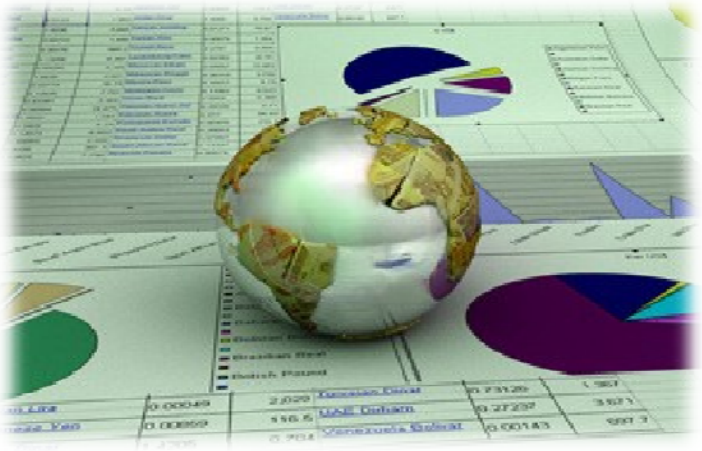


Sadržaj

- Šta je kvalitetan programski kod?
- Imenovanje identifikatora
- Formatiranje koda
- Kvalitetne klase
- Kvalitetne metode
- Ispravno korišćenje promenljivih, iskaza, konstanti, petlji i uslovnih iskaza
- Defanzivno programiranje
- Komentari i dokumentacija

Šta je kvalitetan programski kod?





Zbog čega je bitan kvalitet koda?

```
public static void main(String[
]args)                              {
  int value=   010,i  =5,    w;
switch   (   value){                                  case
10:w=5;System.out.println(w);break;case 9:i=0;break;
case           8:System.out. println("8 "   );break;
default        :System.out.println("def ")         ;{
System       .out.println("hoho ")                 ;}
for(     int k =0  ;k < i ;k++,System.              out
.  println(k - 'f'));break;}                      {System
.                 out.println("loop!");              }}
```

Šta radi ovaj kod? Da li je on ispravan?

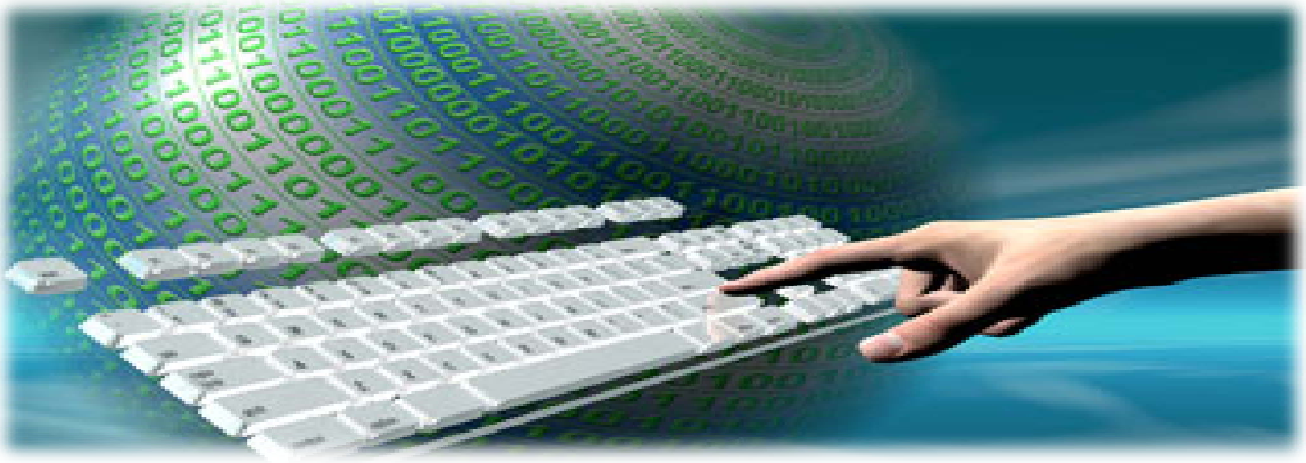
Zbog čega je bitan kvalitet koda? (2)

```
public static void main(String[] args) {
  int value = 010, i = 5, w;
  switch (value) {
    case 10:
      w = 5; System.out.println(w); break;
    case 9:
      i = 0; break;
    case 8:
      System.out.println("8 "); break;
    default:
      System.out.println("def ");
      System.out.println("hoho ");
      for (int k = 0; k < i; k++, System.out.println(k - 'f'));
      break;
  }
  System.out.println("loop!");
}
```

Sada je kod formatiran, ali je još uvek nejasan.

Šta je programski kod visokog kvaliteta?

- Visoko kvalitetan programski kod:
  - Jednostavan za čitanje i razumevanje
    - Jednostavan za modifikovanje i održavanje
  - Uvek iskazuje ispravno ponašanje
    - Dobro je testiran
  - Dobra arhitektura i dizajn
  - Dobra dokumentacija
    - Kod je samo sadržaja
  - Dobro formatiran



Imenovanje identifikatora

Imenovanje klasa, interfejsa, enumeracija, promenljivih i konstanti

Koristite imena sa značenjem

- Uvek koristite imena sa značenjem
  - Imena treba da odgovore na sledeća pitanja:
    - *Šta ova klasa radi? Koja je namena ove promenljive? Zašta se koristi ova promenljiva / klasa?*
  - Dobri primeri:
    - `FactorialCalculator`, `studentsCount`, `Math.PI`, `configFileName`, `createReport`
  - Loši primeri:
    - `k`, `k2`, `k3`, `junk`, `f33`, `KJJ`, `button1`, `variable`, `temp`, `tmp`, `temp_var`, `something`, `someValue`

Osnovna uputstva za imenovanje

- Uvek koristite engleski jezik
  - Kako bi se osećali da čitate kod koji su pisali vijetnamci, sa imenima promenljivih na vijetnamskom jeziku?
  - Engleski jezik je jedini jezik koji svi programeri poznaju
- Izbegavajte skraćenice
  - Primer: `scrpCnt` ili `scriptsCount`
- Izbegavajte imena koja se teško izgovar

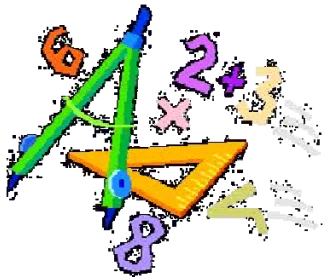
  - Primer: `dtbgRegExPtrn` ili `dateTimeBulgarianRegExPattern`

Dužina imena

- Koliko dugačka treba da budu imena klasa / interfejsa / metoda ?
  - Imena treba da budu dugačka onoliko koliko je potrebno
  - Nemojte da skraćujete imena uloliko će ona postati nejasna
  - Vaše razvojno okruženje ima autocomplete, zar ne?
- Dobri primeri: `FileNotFoundException`, `CustomerSupportNotificationService`
- Loši primeri: `FNFException`, `CustSuppNotifSrvc`

Imenovanje metoda

- Imena metoda moraju imati značenje
- Treba da odgovore na pitanje:
  - Šta radi ova metoda?

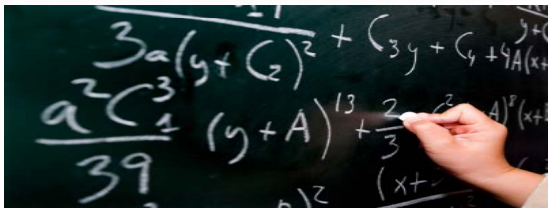


- Ukoliko ne možete da nađete odgovarajuće ime, razmislite da li ta metoda ima jasnu namenu.
- Dobri primeri: `findStudent`, `loadReport`, `sinus`
- Loši primeri: `method1`, `doSomething`, `handleStuff`, `sampleMethod`, `dirtyHack`

Single Purpose of All Methods

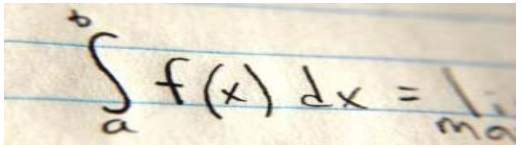
- Metode uvek treba da imaju jedinstvenu svrhu!
  - U suprotnom teško ih je ispravno imenovati
  - Kako imenovati metodu koja kreira godišnji izveštaj o prihodu, downloaduje update sa interneta ili skenira računarske viruse?
  - `createAnnualIncomesReportDownloadUpdatestAndScanForViruses`
- Metode koje imaju višestruku namenu (slaba kohezija) je teško imenovati
  - Potrebno je refaktorisati umesto imenovanja

Imenovanje promenljivih



- Imena promenljivih
  - Trebaju biti u camelCase formatu
  - Preporučena forma: [Imenica] ili [Pridev] + [Imenica]
  - Treba da objašnjavaju namenu promenljive
    - Ukoliko ne možete naći dobro ime za promenljivu proverite da li ima jedinstvenu namenu
    - Izuzetak: promenljive sa veoma malim opsegom, npr. indeks u kratkoj for petlji
  - Imena moraju biti konzistentna u projektu

Imenovanje promenljivih - primeri

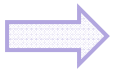


- Dobri primeri:
  - `firstName`, `report`, `usersList` , `fontSize`, `maxSpeed`, `font`, `startIndex`, `endIndex`, `charsCount`, `configSettingsXml`, `config`, `dbConnection`, `createUserSqlCommand`
- Loši primeri:
  - `foo`, `bar`, `p`, `p1`, `p2`, `populate`, `LastName`, `last_name`, `LAST_NAME`, `no_convertImage`, `MAXSpeed`, `_firstName`, `temp`, `temp2`, `_temp`, `firstNameMiddleNameAndLastName`

Privremene promenljive

- Da li privremene promenljive zaista postoje?
  - Sve promenljive u programu su privremene jer postoje samo u toku izvršenja programa?
- Privremene promenljive se uvek mogu imenovati bolje od `temp` ili `tmp`:

```
// Swap a[i] and a[j]
int temp = a[i];
a[i] = a[j];
a[j] = temp;
```



```
// Swap a[i] and a[j]
int oldValue = a[i];
a[i] = a[j];
a[j] = oldValue;
```

Dužina imena promenljivih

- Koliko duga trebaju biti imena promenljivih?
  - Zavisi od opsega i životnog veka
  - Promenljive koje su "poznatije" trebaju imati duža i samo sadržajna imena
- Primeri prihvatljivih imena:

```
for (int i=0; i<users.length; i++)
  if (i % 2 == 0)
    sum += users[i].getWeight();
```

```
class Student {
  public String lastName;
}
```

- Primeri neprihvatljivih imena:

```
class PairOfLists {
  private int count;
}
```

```
class Student {
  private int i;
}
```

Imenovanje konstanti

- Koristite CAPITAL_LETTERS za `final` polja
- Koristite samosadržajna imena koja opisuju njihove vrednosti
- Dobri primeri:

```
private static final int READ_BUFFER_SIZE = 8192;
public static final PageSize DEFAULT_PAGE_SIZE = PageSize.A4;
private static final int FONT_SIZE_IN_POINTS = 16;
```

- Loši primeri:

```
public static final int MAX = 512; // Max what? Apples or Oranges?
public static final int BUF256 = 256; // What about BUF256 = 1024?
public static final String GREATER = "&gt;"; //GREATER_HTML_ENTITY
public static final int FONT_SIZE = 16; // 16pt or 16px?
public static final PageSize PAGE = PageSize.A4; // PAGE_SIZE
```

Formatiranje koda

Zbog čega je potrebno formatiranje koda?

```
import java.io.FileInputStream;import java .io.
FileOutputStream;import java.io.IOException;import java
.io.InputStream;import java.io . OutputStream;public
class Test{public static void copyFileWithBuffer(
String sourceFileName ,String destFileName )throws
IOException{InputStream inputStream=new FileInputStream
(sourceFileName );try{ OutputStream outputStream
= new FileOutputStream(destFileName);try {byte [
] buffer = new byte[64* 1024] ;int bytesRead ;while
((bytesRead=inputStream .read( buffer)) !=-1
){outputStream.write(buffer,0,bytesRead ) ;}}finally {
outputStream.close();}}finally {inputStream.close();}}}
```

Osnove formatiranja koda

- Ciljevi dobrog formatiranja
  - Da se poboljša čitljivost koda
  - Da se olakša održavanje koda

- Osnovni princip formatiranja koda:

Format programskog koda treba da oslika njegovu logičku strukturu.

  - Bilo koji stil formatiranja koji poštuje ovaj princip je dobar
    - Svaki drugi stil je loš

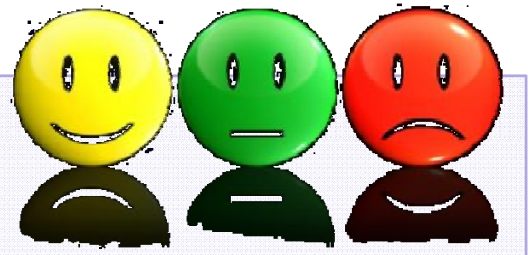
Uvlačenje metoda i blokova

- Metode treba da budu uvučene za jedan [Tab] od tela klase
- Telo metode takođe treba da bude uvučeno za jedan [Tab]

```
public class IndentationExample {
    private int Zero() {
        return 0;
    }
}
```

Dobro i loše formatiranje

- Dobar primer:



```
for (int i=0; i<10; i++) {
  System.out.println("i=" + i);
}
```

- Loši primeri:

```
for (int i=0; i<10; i++)
  System.out.println("i=" + i);
```

```
for (int i=0; i<10; i++) System.out.println("i=" + i);
```

```
for (int i=0; i<10; i++)
{
  System.out.println("i=" + i);
}
```

Prelom dugačkih linija

- Prelomi dugu liniju nakon interpunkcije
- Uvucite narednu liniju dvostrukim [Tab]
- Nemojte dodatno uvlačiti treću liniju
- Primeri:

```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
    matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
    matrix[x, y+1] == 0) {
  // Code comes here indented by a sinlge [Tab] …
```

```
DictionaryEntry<K, V> newEntry =
  new DictionaryEntry<K, V>(
  oldEntry.Key, oldEntry.Value);
```

Neispravan prelom dugih linija

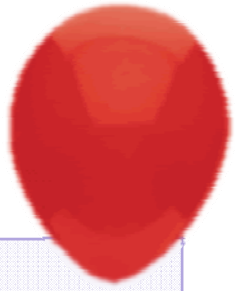
```
if (matrix[x, y] == 0 || matrix[x-1, y] ==
    0 || matrix[x+1, y] == 0 || matrix[x,
    y-1] == 0 || matrix[x, y+1] == 0) {
  …
```


```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
  matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
  matrix[x, y+1] == 0) {
  …
```

```
DictionaryEntry<K, V> newEntry
  = new DictionaryEntry<K, V>(oldEntry
  .Key, oldEntry.Value);
```

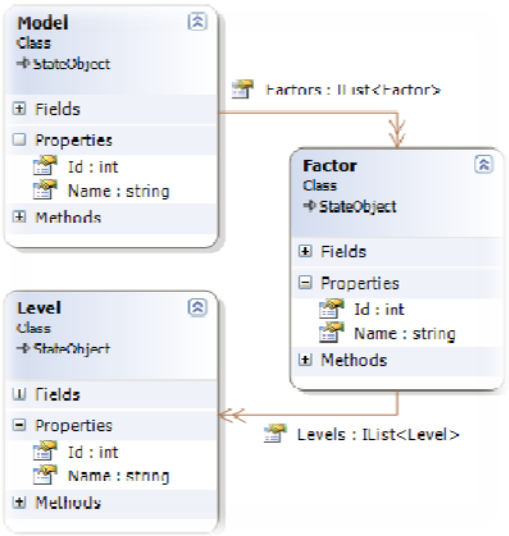
Poravnjavanje koda

- Bilo kakva vrsta poravnavanja se smatra štetnim
  - Ravnanja je teško održavati!
- Loši primeri:



```
Date           date     = new java.util.Date();
int            count    = 0;
Student        student  = new Student();
List<Student> students = new ArrayList<Student>();
```

```
matrix[x, y]                 == 0;
matrix[x + 1, y + 1]         == 0;
matrix[2 * x + y, 2 * y + x] == 0;
matrix[x * y, x * y]         == 0;
```



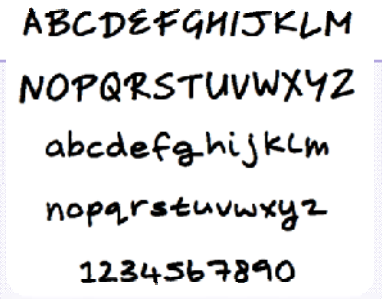
Klase visokog kvaliteta

Kako dizajnirati klase visokog kvaliteta?
Abstrakcija, kohezija i povezivanje

Klase visokog kvaliteta: abstrakcija

- Prezentujte konzistentan nivo **abstrakcije** u deklaraciji klase (javno vidljivi članovi)
  - Koji stepen abstrakcije klasa implementira?
  - Da li ona predstavlja samo jednu stvar?
  - Da li ime klase adekvatno opisuje njenu namenu?
  - Da li klasa definiše jasan i razumljiv javni interfejs?
  - Da li klasa sakriva sve svoje detalje implementacije?

Primer dobre abstrakcije



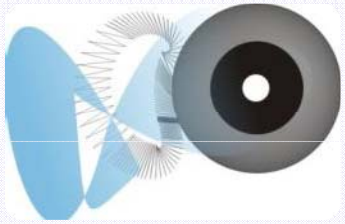

```
public class Font {
  public String name;
  public float sizeInPoints;
  public FontStyle style;

  public Font(String name, float sizeInPoints, FontStyle style) {
    this.setName(name);
    this.setSizeInPoints(sizeInPoints);
    this.setStyle(style);
  }

  …

  public void drawString(DrawingSurface surface,
    String str, int x, int y) { … }

  public Size measureString(String str) { … }
}
```

Primer loše abstrakcije

```
public class Program {
  public String title;
  public int size;
  public Color color;
  public void initializeCommandStack();
  public void pushCommand(Command command);
  public Command popCommand();
  public void shutdownCommandStack();
  public void initializeReportFormatting();
  public void formatReport(Report report);
  public void printReport(Report report);
  public void initializeGlobalData();
  public void shutdownGlobalData();
}
```

Da li ova klasa zaista predstavlja "program"? Da li je njeno ime dobro?

Da li ova klasa ima samo jednu namenu?



Enkapsulacija

- Minimizirajte vidljivost klasa i članova
  - Počnike od `private` i pomerajte se na `package`, `protected` i `public` ukoliko je potrebno
- Klase treba da sakriju svoje detalje implementacije
  - Princip enkapsulacije u OOP
  - Sve što nije deo javnog interfejsa klase treba da bude deklarisano kao `private`
- Nikada ne deklarišite polja klase kao public (izuzev konstanti)
  - Koristite metode i svojstva da pristupate poljima.

Metode visokog kvaliteta

Kako dizajnirati i implementirati metode visokog kvaliteta? Razumevanje kohezije i vezivanja

Zbog čega su nam potrebne metode?

- Metode su značajne u razvoju softvera
  - Smanjuju kompleksnost
    - Podeli i savladaj: složeni problemi se dele u nekoliko jednostavnijih
  - Povećavaju čitljivost koda
    - Male metode sa dobrim imenima čine kod samo-dokumentujućim
  - Smanjuju mogućnost dupliranja koda
    - Duplirani kod je teško održavati

Osnove korišćenja metoda

- Osnovni princip ispravne upotrebe metoda:

Metoda treba da radi ono što kaže njeno ili ili treba da ispaljuje izuzetak. Bilo koje drugo ponašanje je neispravno!

- Metode treba da rade ono što kažu njihova imena
  - Ništa manje
  - Ništa više

- U slučaju neispravnog ulaza ili preduslova, trebalo bi da se aktivira izuzetak

Primeri ispravnih metoda

```
long sum(int[] elements) {
  long sum = 0;
  for (int element : elements) {
    sum = sum + element;
  }
  return sum;
}
```

```
double calcTriangleArea(double a, double b, double c)
{
  if (a <= 0 || b <= 0 || c <= 0) {
    throw new IllegalArgumentException(
      "Sides should be positive.");
  }
  double s = (a + b + c) / 2;
  double area = Math.sqrt(s * (s - a) * (s - b) * (s - c));
  return area;
}
```

Primeri loših metoda

```
long sum(int[] elements) {
  long sum = 0;
  for (int i = 0; i < elements.length; i++) {
    sum = sum + elements[i];
    elements[i] = 0;
  }
  return sum;
}
```

Skriveni sporedni efekat. Nemojte ovo raditi!

```
double calcTriangleArea(double a, double b, double c) {
  if (a <= 0 || b <= 0 || c <= 0) {
    return 0;
  }
  double s = (a + b + c) / 2;
  double area = Math.sqrt(s * (s - a
  return area;
}
```

Neispravan rezultat. Umesto ovoga ispalite izuzetak!

Dužine metoda

- Koliko dugačke trebaju biti metode?

  - Ne postoji određeno ograničenje
  - Izbegavajte metode koje su duže od jednog ekrana
    - Jedan ekran ≈ 30-40 linija
  - Kohezija i povezanost su bitnije od dužine metoda!
  - Dugačke metode nisu uvek loše
    - Budite sigurni da imate dobar razlog za njihovu dužinu

Defanzivno programiranje

Korektno upravljanje nekoretnim ulazima

Principi defanzivnog programiranja

- Osnovni princip defanzivnog programiranja

Bilo koja javna metoda mora da proverava ulazne podatke, preduslove i rezultate

- Defanzivno programiranje znači:
  - Da se očekuje nekorektan ulaz i on se ispravno obrađuje
  - Da se ne razmišlja samo o uobičajenom toku izvršenja, već i da se razmatraju neuobičajene situacije
  - Da se obezbedi da neispravan ulaz generiše izuzetak, a ne neispravan izlaz.

Primer defanzivnog programiranja

```
public String Substring(String str, int startIndex, int count)
{
  if (str == null) {
    throw new NullPointerException(“str is null.");
  }
  if (startIndex >= str.length()) {
    throw new IllegalArgumentException(
      "Invalid startIndex:" + startIndex);
  }
  if (startIndex + count > str.length()) {
    throw new IllegalArgumentException(
      "Invalid length:" + count);
  }
  …
  Debug.assert(result.length() == count);
}
```

Provera ulaza i preduslova.

Izvršavanje glavne logike.

Provera rezultata.

Izuzeci – dobra praksa

- Izaberite dobro ime za vašu klasu izuzetka
  - Loš primer:

```
throw new Exception("File error!");
```

  - Dobar primer:

```
throw new FileNotFoundException("Cannot find file " + fileName);
```

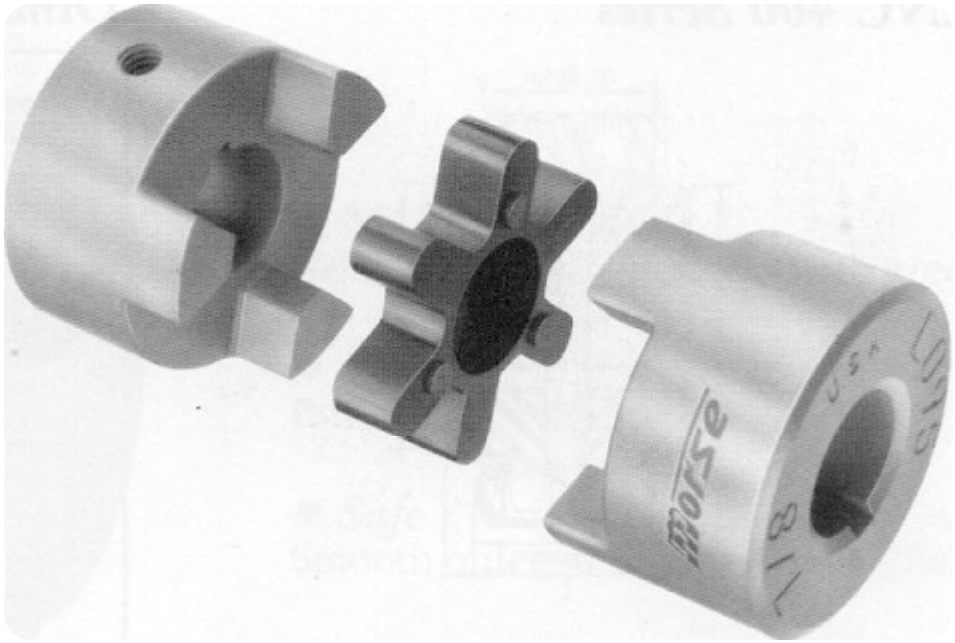
- Koristite deskriptivne poruke o grešci
  - Loš primer:

```
throw new Exception("Error!");
```

  - Dobar primer:

```
throw new IllegalArgumentException("The speed should be " +
  "between " + MIN_SPEED + " and " + MAX_SPEED + ".");
```

Kohezija i vezivanje



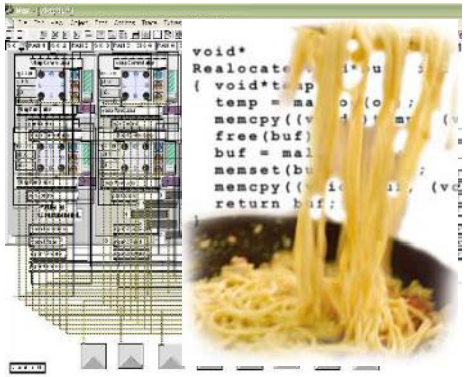
Jaka kohezija

- Metode treba da imaju jaku koheziju
  - Treba da adresiraju jedan zadatak i da ga adresiraju na zadovoljavajući način
  - Treba da imaju jasnu namenu
- Metode koje adresiraju nekoliko zadataka u isto vreme je teško imenovati
- Jaka kohezija se koristi u inženjerstvu
  - U računarstvu bilo koja PC komponenta izvršava jedan zadatak, npr. hard disk izvršava skladištenje podataka

Strong and Weak Cohesion

- Jaka kohezija: hard disk, CD-ROM, floppy

- Slaba kohezija: špageti kod



Jaka kohezija

- Primer jake kohezije:
  - Klasa `Math` koja ima metode i atribute:
    - `sin()`, `cos()`, `sqrt()`, `pow()`, `exp()`
    - `Math.PI`, `Math.E`

```
double sideA = 40, sideB = 69;
double angleAB = Math.PI / 3;

double sideC =
  Math.pow(sideA, 2) + Math.pow(sideB, 2) -
  2 * sideA * sideB * Math.cos(angleAB);

double sidesSqrtSum = Math.sqrt(sideA) +
  Math.sqrt(sideB) + Math.sqrt(sideC);
```

Slaba kohezija

- Primer slabe kohezije
  - Klasa `Magic` koja ima sve ove metode:

```
public void printDocument(Document d);
public void sendEmail(String recipient, String
  subject, String text);
public void calculateDistanceBetweenPoints(int x1,
  int y1, int x2, int y2)
```

- Još jedan primer:

```
MagicClass.makePizza("Fat Pepperoni");
MagicClass.withdrawMoney("999e6");
MagicClass.openDBConnection();
```

Slabo vezivanje

- Šta je **slabo vezivanje**?
  - Minimalna zavisnost metoda od drugih delova koda
  - Minimalna zavisnost članova klasa od eksternih klasa i njenih članova
  - Nema sporednih efekata
  - Ako je vezivanje slabo, veoma lako možemo ponovo upotrebiti metodu ili grupu metoda u novom projektu
- Jako vezivanje→ špageti kod

Slabo i jako vezivanje

- Slabo vezivanje:
  - Lako je zameniti stari HDD
  - Lako je smestiti ovaj HDD na drugu matičnu ploču
- Jako vezivanje:
  - Gde je video kartica?
  - Možemo li zameniti video karticu?

Primer slabog vezivanja

```
class Report {
  public bool loadFromFile(String fileName) {…}
  public bool saveToFile(String fileName) {…}
}

class Printer {
  public static int print(Report report) {…}
}

class LooseCouplingExample
{
  public static void main(String[] args) {
    Report myReport = new Report();
    myReport.loadFromFile("C:\\DailyReport.rep");
    Printer.print(myReport);
  }
}
```

Primer jakog vezivanja

- Prosleđivanje parametara kroz atribute klase
  - Tipičan primer jakog vezivanja
  - Nemojte ovo raditi ukoliko nemate dobar razlog!

```
class Sumator
{
  public int a, b;
  private int sum() {
    return a + b;
  }
  public static void main(String[] args) {
    Sumator sumator = new Sumator();
    sumator.a = 3;
    sumator.b = 5;
    System.out.println(sumator.sum());
  }
}
```

Zašto ne bi prosledili brojeve kao parametre metode?

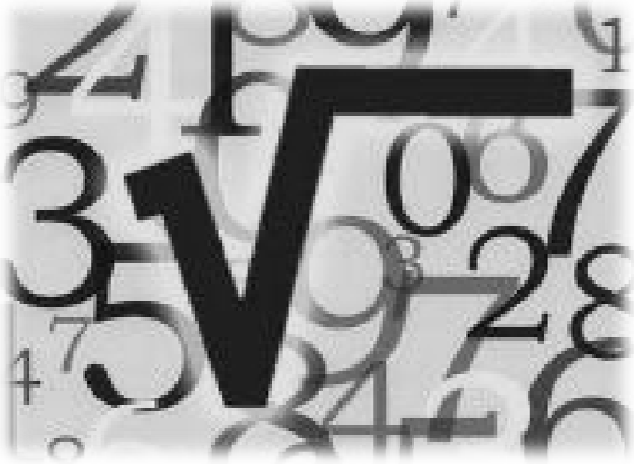
Špageti kod

- Kombinacija slabe kohezije i jakog vezivanja (špageti kod):

```
class Report {
  public void print() {…}
  public void initPrinter() {…}
  public void loadPrinterDriver(string fileName) {…}
  public bool saveReport(string fileName) {…}
  public void setPrinter(string printer) {…}
}

class Printer {
  public void setFileName() {…}
  public static boolean loadReport() {…}
  public static boolean checkReport() {…}
}
```

Korišćenje promenljivih

Dobra praksa



Jedna namena

- Promenljive treba da imaju jednu namenu
  - Nikada ne koristite jednu promenljivu za više namena!
  - Ekonomisanje memorijom nije opravdanje
- Možete li izabrati dobro ime za promenljivu koja ima višestruku namenu?
  - Primer: korišćenje promenljive koja čuva broj studenata za čuvanje proseka njihovih ocena
  - Predloženo ime: `studentsCountOrAvgGrade`

Vraćanje rezultata iz metode

- Uvek smestite rezultat metode u neku promenljivu pre nego što ga vratite
  - Poboljšava se čitljivost koda
    - Povratna vrednost ima samodokumentujuće ime
  - Jednostavnije debagovanjeng
  - Primer:

Namena ove formule je očigledna.

```
int salary = days * hoursPerDay * ratePerHour;
return salary;
```

Na ovoj liniji možemo postaviti prekid i proveriti vrednost.

  - Loš primer:

```
return days * hoursPerDay * ratePerHour;
```

Opseg promenljivih

- Opseg promenljive definiše koliko je neka promenljiva "poznata" u programu
  - Statičke promenljive su mnogo "poznatije" nego promenljive instance , a one su "poznatije" od localnih promenljivih
  - Vidljivost promenljivih je direktno vezana sa njihovim opsegom
    - `public`, `protected`, `package`, `private`
- Uvek se trudite da smanjite opseg promenljiv
  - Ovo smanjuje potencijalno vezivanje
  - Izbegavajte public atribute (izuzetak: konstante)
  - Pristupajte svim atributima kroz svojstva/metode

Primer proširenog opsega

```
public class Globals {
  public static int state = 0;
}

public class Genious {
  public static void printSomething() {
    if (Globals.state == 0) {
      System.out.println("Hello.");
    }
    else {
      System.out.println("Good bye.");
    }
  }
}
```

Variable Span and Lifetime

- Variable **span**
  - The average number of lines of code (LOC) between variable usages
- Variable **lifetime**
  - The number of lines of code (LOC) between the first and the last variable usage in a block
- Keep variable span and lifetime as low as possible

Always define and initialize variables just before their first use and never before it!

Nepotrebno veliki raspon i životni vek promenljive

```
int count;
int[] nums = new int[100];
for (int i = 0; i < nums.length; i++) {
  nums[i] = i;
}
count = 0;
for (int i = 0; i < nums.length / 2; i++) {
  nums[i] = nums[i] * nums[i];
}
for (int i = 0; i < nums.length; i++) {
  if (nums[i] % 3 == 0) {
    count++;
  }
}
System.out.println(count);
```

Životni vek ("count") = 15

raspon= 15 / 4 = 3·75

Smanjeni raspon i životni vek promenljive

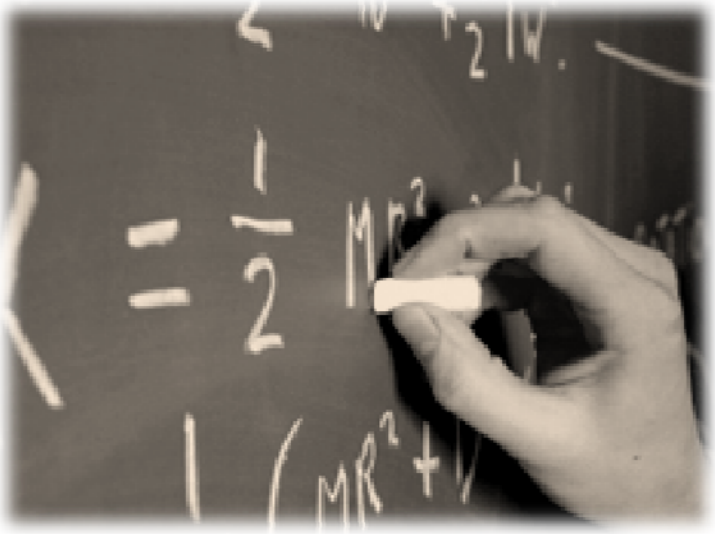
```
int[] nums = new int[100];
for (int i = 0; i < nums.length; i++) {
  nums[i] = i;
}
for (int i = 0; i < nums.length / 2; i++) {
  nums[i] = nums[i] * nums[i];
}
int count = 0;
for (int i = 0; i < nums.length; i++) {
  if (nums[i] % 3 == 0) {
    count++;
  }
}
System.out.println(count);
```

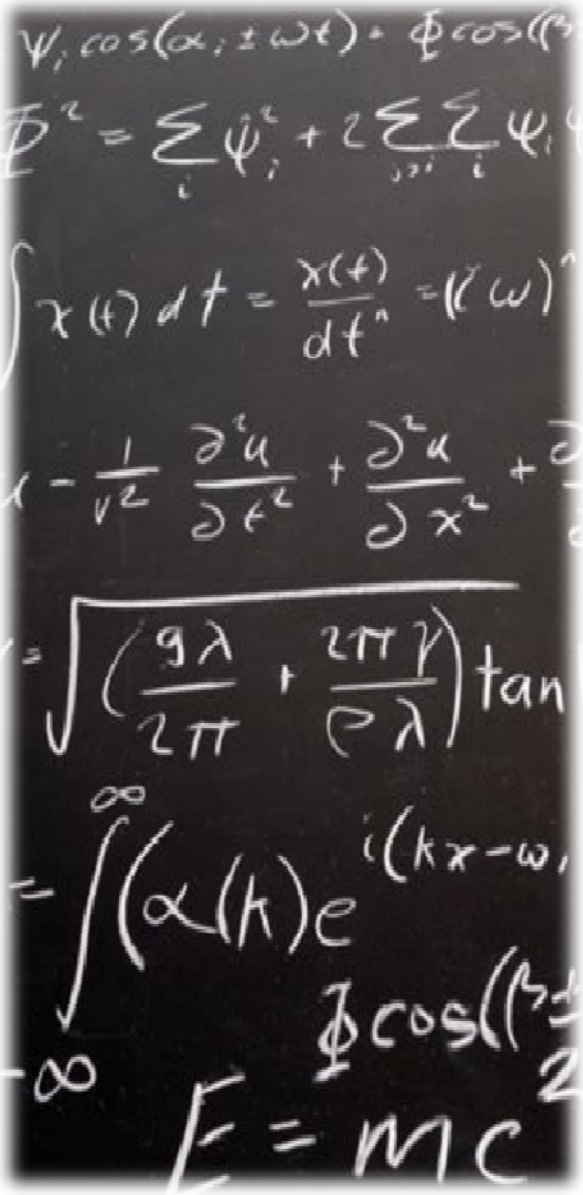
Životni vek = 7

raspon=
7 / 3 = 2.33

Korišćenje iskaza

Dobra praksa





Izbegavajte složene iskaze

- Nikada ne koristite složene iskaze u kodu!
  - Loš primer:

```
for (int i=0; i<xCoords.length; i++) {
  for (int j=0; j<yCoords.length; j++) {
    matrix[i][j] =
      matrix[xCoords[findMax(i)+1]][yCoords[findMin(j)-1]] *
      matrix[yCoords[findMax(j)+1]][xCoords[findMin(i)-1]];
  }
}
```

Šta da radimo ako u ovoj liniji dobijemo `IndexOutOfBoundsException`?

Postoji 10 potencijalnih izvora `IndexOutOfBoundsException` izuzetka u ovom iskazu!

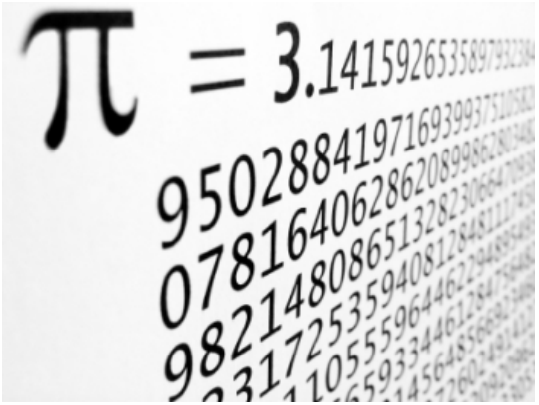
- Složeni iskazi su loši zbog toga što:
  - Čine kod komplikovanim za čitanje i razumevanje, teškim za debagovanje, modifikovanje i održavanje

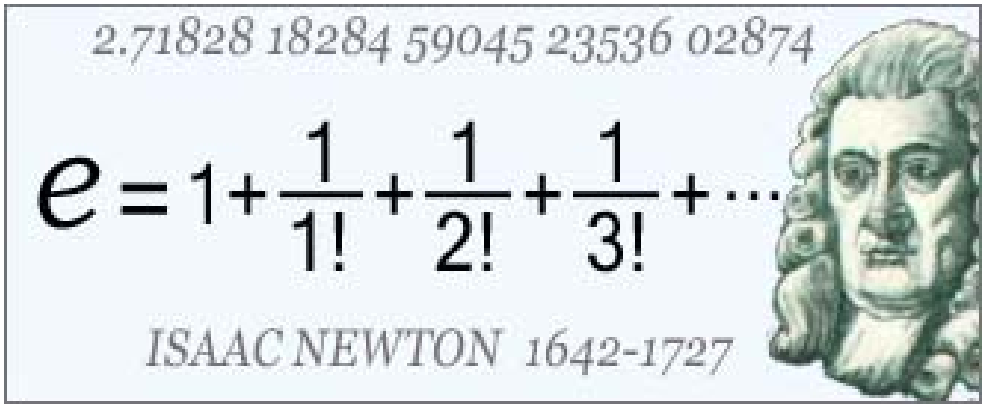
Uprošćavanje složenog iskaza

```
for (int i = 0; i < xCoords.length; i++) {
  for (int j = 0; j < yCoords.length; j++) {
    int maxStartIndex = findMax(i) + 1;
    int minStartIndex = findMin(i) - 1;
    int minXcoord = xCoords[minStartIndex];
    int maxXcoord = xCoords[maxStartIndex];
    int minYcoord = yCoords[minStartIndex];
    int maxYcoord = yCoords[maxStartIndex];
    int newValue =
      matrix[maxXcoord][minYcoord] *
      matrix[maxYcoord][minXcoord];
    matrix[i][j] = newValue;
  }
}
```

Korišćenje konstanti

Kada i kako koristiti konstante?





Izbegavajte magične brojeve i stringove

- Šta je **magični broj** ili **vrednost**?
  - Magični brojevi/ vrednosti su literali koji se razlikuju od `0`, `1`, `-1`, `null` i `""` (prazan string)
- Izbegavajte korišćenje magičnih brojeva/ vrednosti

  - Teški su za održavanje
    - Kada dođe do promene, morate modifikovati sve pojave magičnog broja/konstante
  - Njihovo značenje nije tako očigledno
    - Primer: šta znači broj `1024` ?

Zli magični brojevi

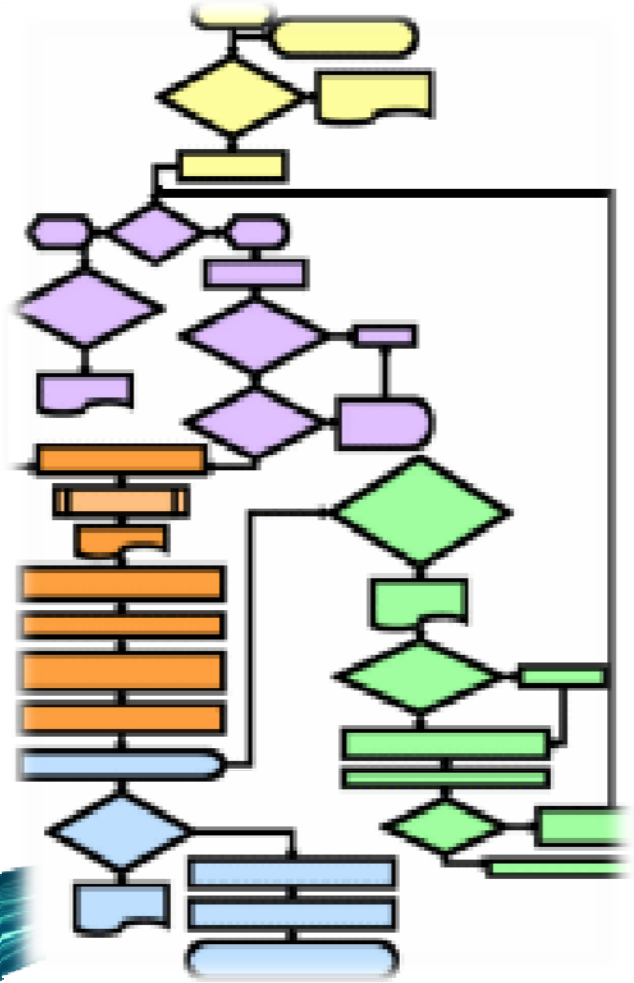
```
public class GeometryUtils {
  public static double calcCircleArea(double radius) {
    double area = 3.14159206 * radius * radius;
    return area;
  }

  public static double calcCirclePerimeter(double radius) {
    double perimeter = 6.28318412 * radius;
    return perimeter;
  }

  public static double calcElipseArea(
      double axis1, double axis2) {
    double area = 3.14159206 * axis1 * axis2;
    return area;
  }
}
```

Pretvaranje magičnih brojeva u konstante

```
public class GeometryUtils {
  public static final double PI = 3.14159206;
  public static double calcCircleArea(double radius) {
    double area = PI * radius * radius;
    return area;
  }
  public static double calcCirclePerimeter(double radius) {
    double perimeter = 2 * PI * radius;
    return perimeter;
  }
  public static double calcElipseArea(
      double axis1, double axis2) {
    double area = PI * axis1 * axis2;
    return area;
  }
}
```

Korišćenje kontrolnih struktura

Ispravno korišćenje uslovnih iskaza i petlji



Korišćenje uslovnih iskaza

- Uvek koristite { i } za telo uslovnih iskaza, čak i kada je u pitanju samo jedna linija:

```
if (condition) {
  doSometing();
}
```

- Zbog čega izostavljanje zagrada može biti štetno?

```
if (condition)
  doSomething();
  doAnotherThing();
doDifferentThing();
```

  - Ovo je nerazumljiv kod + nerazumljivo formatiranje

Koristite jednostavne uslove

- Ne koristite složene `if` uslove
  - Uvek ih možete uprostiti uvođenjem boolean promenljivih ili boolean metoda
  - Loš primer:

```
if (x > 0 && y > 0 && x < width-1 && y < height-1 &&
  matrix[x, y] == 0 && matrix[x-1, y] == 0 &&
  matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&
  matrix[x, y+1] == 0 && !visited[x, y]) { … }
```

  - Složeni boolean iskazi su štetni
  - Kako ćete naći problem ako dobijete `IndexOutOfBoundseException` izuzetak?

Uprostite boolean iskaze

- Prethodni primer se lako može refaktorisati u samoopisujući kod:

```
boolean inRange =
  x > 0 && y > 0 && x < width-1 && y < height-1;
boolean emptyCellAndNeighbours =
  matrix[x, y] == 0 && matrix[x-1, y] == 0 &&
  matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&
  matrix[x, y+1] == 0;
if (inRange && emptyCellAndNeighbours && !visited[x, y])
```

- Sada je kod:
  - Jednostavan za razumevanje – logika uslova je jasna
  - Jednostavan za debugovanje - breakpoint može giti unutar `if`

Izbegavajte duboko ugnježđavanje blokova

- Duboko ugnježđavanje uslovnih iskaza i petlji čini kod nejasnim
  - Duboko ugnježđavanje ≈ 3-4 ili više nivoa ugnježđavanja
  - Duboko ugnježđeni kod je složen i težak za čitanje i razumevanje
  - Obično možete izvući delvoe koda u odvojene metode
    - Ovo pojednostavljuje logiku koda
    - Korišćenje dobrih imena metode čini kod samo-dokumentujućim

Primer dubokog ugnježđavanja

```
if (maxElem != Integer.MAX_VALUE) {
  if (arr[i] < arr[i + 1]) {
    if (arr[i + 1] < arr[i + 2]) {
      if (arr[i + 2] < arr[i + 3]) {
        maxElem = arr[i + 3];
      }
      else {
        maxElem = arr[i + 2];
      }
    }
    else {
      if (arr[i + 1] < arr[i + 3]) {
        maxElem = arr[i + 3];
      }
      else {
        maxElem = arr[i + 1];
      }
    }
  }
```

(continues on the next slide)

Primer dubokog ugnježđavanja(2)

```
  else {
    if (arr[i] < arr[i + 2]) {
      if (arr[i + 2] < arr[i + 3]) {
        maxElem = arr[i + 3];
      }
      else {
        maxElem = arr[i + 2];
      }
    }
    else {
      if (arr[i] < arr[i + 3]) {
        maxElem = arr[i + 3];
      }
      else {
        maxElem = arr[i];
      }
    }
  }
}
```

Primer izbegavanja dubokog ugnježdavanja

```
private static int max(int i, int j) {
  if (i < j) {
    return j;
  }
  else {
    return i;
  }
}

private static int max(int i, int j, int k) {
  if (i < j) {
    int maxElem = max(j, k);
    return maxElem;
  }
  else {
    int maxElem = max(i, k);
    return maxElem;
  }
}
```

(continues on the next slide)

Primer izbegavanja dubokog ugnježdavanja

```
private static int findMax(int[] arr, int i) {
  if (arr[i] < arr[i + 1]) {
    int maxElem =
      max(arr[i + 1], arr[i + 2], arr[i + 3]);
    return maxElem;
  }
  else {
    int maxElem = Max(arr[i], arr[i + 2], arr[i + 3]);
    return maxElem;
  }
}

…

if (maxElem != Integer.MAX_VALUE) {
  maxElem = FindMax(arr, i);
}
```

Komentari i dokumentovanje koda

Koncept samo-dokumentovanog koda

Samo-dokumentovani kod

- Efikasni komentari ne ponavljaju kod
  - Oni ga opisuju na višem nivou i otkrivaju ne tako očigledne detalje
- Osnovni principi samo-dokumentujućeg koda

Najbolja dokumentacija je sam kod.

Napravite kod samo-razumljivim i samo-dokumentujućim, jednostavnim za čitanje i razumevanje.

Ne dokumentujte loš kod, isprogramirajte ponovo!

Primer loših komentara

```
public static List<Integer> findPrimes(int start, int end) {
  // Create new list of integers
  List<Integer> primesList = new ArrayList<Integer>();

  // Perform a loop from start to end
  for (int num = start; num <= end; num++) {
    // Declare boolean variable, initially true
    boolean prime = true;

    // Perform loop from 2 to sqrt(num)
    for (int div = 2; div <= Math.sqrt(num); div++) {
      // Check if div divides num with no remainder
      if (num % div == 0) {
        // We found a divider -> the number is not prime
        prime = false;

        // Exit from the loop
        break;
      }

      // Continue with the next loop value
    }
```

(continues on the next slide)

Primer loših komentara(2)

```
    // Check if the number is prime
    if (prime) {
      // Add the number to the list of primes
      primesList.add(num);
    }
  }

  // Return the list of primes
  return primesList;
}
```

Primer samo-dokumentujućeg koda

```
public static List<Integer> findPrimes(
    int start, int end) {
  List<Integer> primesList = new ArrayList<Integer>();
  for (int num = start; num <= end; num++) {
    boolean isPrime = isPrime(num);
    if (isPrime) {
      primesList.add(num);
    }
  }
  return primesList;
}
```

Dobrom kodu nisu potrebni komentari. On je samo-objašnjavajući.

(continues on the next slide)

Primer samo-dokumentujućeg koda(2)

```
private static boolean isPrime(int num) {
  boolean isPrime = true;
  int maxDivider = (int) Math.sqrt(num);
  for (int div = 2; div <= maxDivider; div++) {
    if (num % div == 0) {
      // Found a divider -> the number is not prime
      isPrime = false;
      break;
    }
  }
  return isPrime;
}
```

Ovaj komentar objašnjava ne tako očigledan detalj. On ne ponavlja kod.

Dobre metode imaju dobra imena i lako ih je pročitati i razumeti.