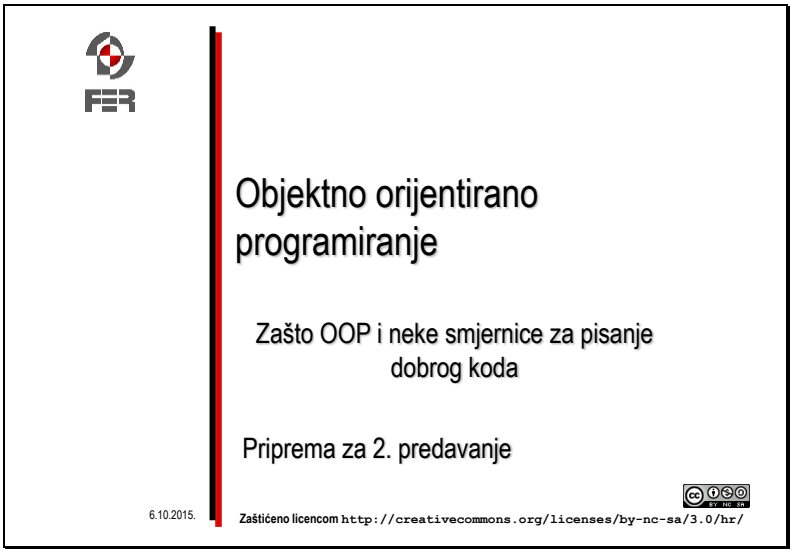
Slide 1



Poštovani studenti, ove bilješke će vam ukratko pojasniti neke pojmove iz prezentacije. Molimo da ih pažljivo proučite uz čitanje samih slajdova.

Slide 2

Programske paradigme i jezici (gdje se uklapa OOP?)

- Tri najvažnije programske paradigme danas
  - Proceduralna programska paradigma
    - Programiranje u C-u kakvo se radi na uvodnim kolegijima FER-a
  - Objektno-orijentirana programska paradigma (OOP)
    - Predmet interesa ovog kolegija (npr. programski jezik Java)
  - Deklarativna programska paradigma
    - Funkcijski jezici u kojima se sve radi pozivima funkcija i rekurzijama (npr. Haskell)
    - XML i njegove varijante (npr. Microsoftov XAML za izradu vizualno/grafički zahtjevnih programa)

Zašto su paradigme važne? U svakoj paradigmi postoji niz programskih jezika. Neke paradigme su popularnije od drugih, a isto vrijedi i za programske jezike. Slaba popularnost nikako ne znači da je neka paradigma ili jezik loša ili loš. To samo znači da se zbog povijesnih razloga krenulo nekim smjerom i da je većina ljudi prihvatila takve procese. Primjerice, objektno-orijentirana paradigma je danas de facto standard u računarstvu te će većina poduzeća koja se bave proizvodnjom programske potpore (softvera) koristiti programske jezike iz ove paradigme (Java, C#, C++ i sl). S druge strane jezici funkcijske paradigme poput Haskella danas se intenzivno koriste u znanstveno-istraživačkom svijetu i nezamjenjiv su alat znanstvenicima. Baš zbog ovog dualiteta sve više jezika postaje višeparadigmatskima te u svojoj sintaksi kombiniraju više paradigmi (npr. C# je od verzije 3.0 uveo lambda-izraze, te se time u objektni jezik uvelo elemente funkcijske paradigme).

Slide 3

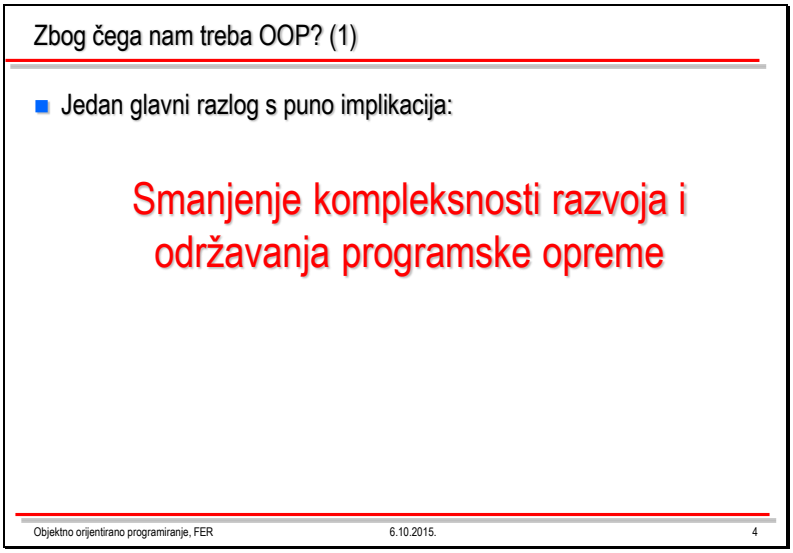
Počeci objektno orijentiranih jezika

- Simula (1967.)
  - prvi programski jezik sa svojstvima objektno-orijentirane paradigme
  - namijenjen izgradnji sustava za simulaciju
  - uveden pojam klase / razreda
- Smalltalk (1972.)
  - prvi "pravi" (čisti) objektno-orijentiran programski jezik ("sve je objekt")
  - razvijen u laboratoriju Xerox PARC
  - Smalltalk-80 je najkorištenija verzija
- C++
  - "hibridni" objektno-orijentirani jezik nastao iz C-a - ispočetka se zvao "C s razredima" ("C with Classes")
  - razvio ga je Bjarne Stroustrup (1983.) u Bell Labs
  - početna ANSI standardizacija dovršena je (tek) 1998., a 2003. je izdana standardna verzija s ispravljenim pogreškama
  - C++ je "predak" danas široko korištenih jezika - Java, C# i VB.NET

Objektno orijentirano programiranje, FER 6.10.2015. 3

Uz navedene programske jezike danas su vrlo popularni objektni jezici Java i C#. Za razliku od programskog C++ koji je nastao evolucijom iz C-a, C# i Java se smatraju čistim objektnim jezicima. Autorima tih jezika je bilo stalo da se korisnika oslobodi razmišljanja o tzv. „implementacijskim detaljima", tj. programiranju koje se oslanja na bitove i bajtove. Iza jezika kao što su Java i C# stoje tzv. programski okviri (eng. framework) čija je zadaća programeru ponuditi puno gotovih funkcionalnosti koje programer koristi kao da ih je sam programirao. Iako se čini da je ovo „posuđivanje" loše, zapravo to nije tako. Cijelo se računarstvo temelji na razinama ponovne iskoristivosti, te ideji da se programira jednom, a koristi višestruko. Na taj način programeri dobivaju vremena za fokus na rješavanje problema koji se vezuju uz namjenu same aplikacije (npr. Razvoj kvalitetnijeg sučelja, dizajn pametnih algoritama koji će korisniku olakšati ili unaprijediti rad sa programom i sl).

Slide 4



Između ostalih važnih razloga, objektno orijentirana paradigma služi smanjenju kompleksnosti razvoja programske opreme i njenom lakšem održavanju. Nije u pitanju samo ponovna iskoristivost već gotovog programskih komponenti koje su redovite napravljene izrazito kvalitetno i adekvatno testirane tako da pružaju sigurnu podršku programeru, već se radi i o samom sistemu kojeg OOP nameće. OOP i jezici koju ju podržavaju na neki način tjeraju i obvezuju programera na rad u okvirima standarda. To ne znači da programer gubi slobodi, već samo da ga se usmjerava na načine na koje može postići tu slobodu. Pošto su načini jasno i dobro definirani, svi programeri koji programiranju u objektnoj paradigmi mogu „računati" da postoje neki standard rada te se zbog toga mogu osloniti na neke zajedničke dobro znane elemente. Ova tema će se produbiti kada predavanja dođu do teme sučelja.

Slide 5

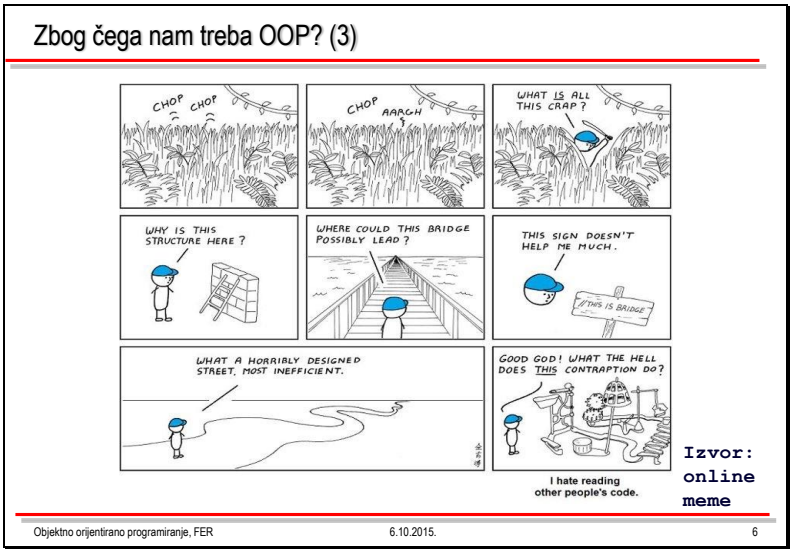
Zbog čega nam treba OOP? (2)

- složenost današnjih računalnih sustava
  - banke, osiguranja, trgovački lanci
  - državna uprava
  - poslovni sustavi zračnih luka, sustavi u avionima
- ograničenost ljudskog mozga
  - Miller (1956)
    - čovjek ne može paralelno misliti o više od 5 – 7 mentalnih entiteta
  - Edsger Dijkstra (prvi računarac u Nizozemskoj; autor preteče članka "A Case against the GO TO Statement")
    - *"The competent programmer is fully aware of the strictly limited size of his own scull; therefore, he approaches the programming task in full humility"*

Objektno orijentirano programiranje, FER 6.10.2015. 5

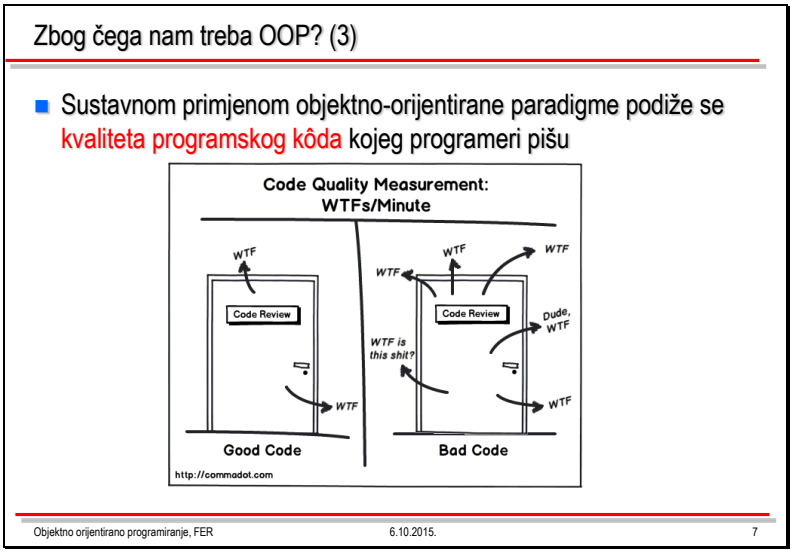
Jedan od prvih „računaraca" je Edsger Dijkstra, čovjek koji je tvorac mnogih vrijednih ideja u računarstvu se i dan danas koriste, ali u ponešto izmijenjenom obliku, prilagođenom modernom svijetu i novim tehnologijama. No, neki njegovi radovi su unatoč svemu prilično univerzalni i dotiču se izazove koje su imali prvi programeri, ali koje imaju i današnji programeri. Dijkstra je u slajdu navedenim člankom nastojao problematizirati odnos programera prema vlastitom radu. S obzirom da su često suočeni s kratkim rokovima, nedefiniranosti problema i sl. programeri imaju tendenciju programirati neuredno (npr. Imenovati varijable kako im se dopadne – xxxx, fido, jamagarac i sl) što se u konačnici reflektira na kvalitetu programa, koji postaju teško čitljivo. U današnjem svijetu mnogu programi traju i desetljeća, te je vrlo važno da budu čitljivi kako bi ih se što lakše nadograđivalo (održavalo). Neke studije ističu da se više vremena troši na čitanje programa nego na njihovo pisanje.

Slide 6



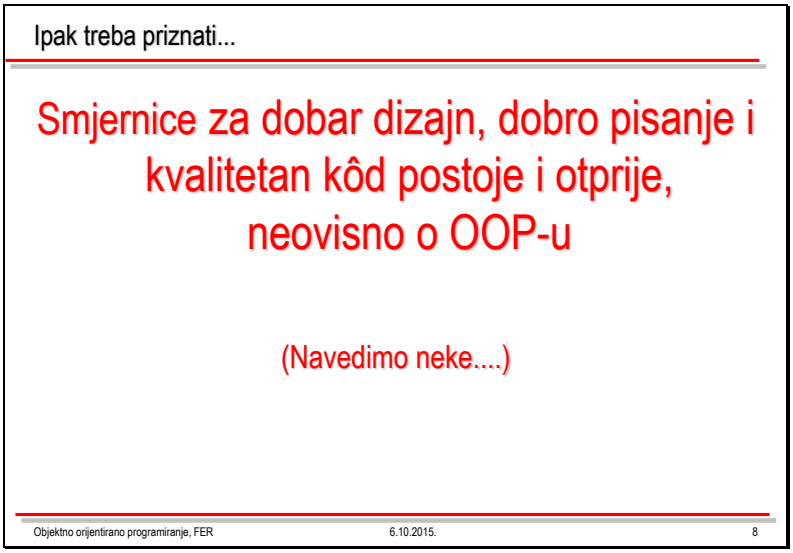
Ova zanimljiva ilustracija govori o tome kako je teško čitati tuđi programski kôd. Kao što je ilustrirano, kôd ponekad sadrži programske elemente koji su nepotrebni i koji su zaostali od prethodnih revizija programa (npr. Varijabla sum se koristila za zbrajanje varijabli i i j, ali budući da su te varijable postale nepotrebne, sum više nema smisla te treba i njega ukloniti – neki programeri u žurbi zaborave ukloniti takve varijable). Takvi programi i dalje rade i to izvrsno, no sam kôd temeljem kojih su izrađeni se sastoji i od nepotrebnih elemenata koji otežavaju čitanje. Također, koliko god da se savjetuje lijepo komentirati programe, pisanje komentara samo da bi ih imali u kôdu nema smisla. Proučite gornju ilustraciju, komentar koji se spominje u biti ne govori ništa novo i ne pomaže programeru u snalaženju u kôdu, te je time suvišan i nepotrebno povećava kôd.

Slide 7



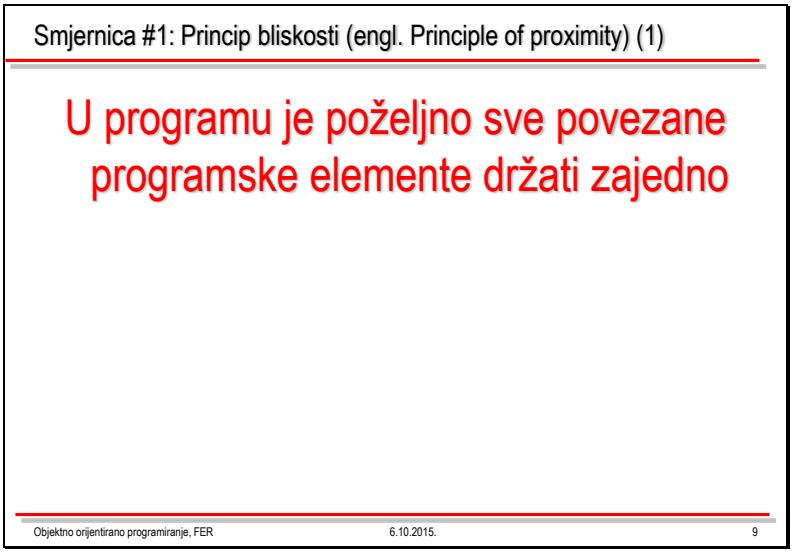
Kao što je prije i navedeno, OOP ne rješava sve probleme pisanja lošeg kôda već samo usmjerava programere na određene standardne načine pisanja kôda. Time programeri pišu „čišći” kôd kojeg je lakše mijenjati i ponovno koristiti (eng. resuable code). Upravo je potonje vrlo važno u modernom razvoju programske potpore jer se unutar jednog tima ili firme određene programske komponente mogu ponovno koristiti čime se smanjuje ukupno vrijeme razvoja programa i povećava njegova vrijednosti. Također, što više ljudi koristi neku programsku cjelinu, to je veća šansa da će se pronaći i riješiti greške (tzv. bugove). S vremenom će ta komponenta sve brže i brže postajati bliža savršenoj.

Slide 8



Ne postoje matematičke formule za pisanje dobrog kôda. Neke je dijelove pisanja programa moguće jasno specificirati, no većina razvoja programa ovisi o dosjetljivosti i kreativnosti programera. Neke vrlo velike i neefikasne programe bi vješti programeri napisali kao kratke i vrlo efikasne, koristeći znanje s naprednih predmeta na fakultetu, iskustvo, savjete kolega te vlastitu sposobnost i kreativnost. Neovisno o objektnoj paradigmi, postoje savjeti za pisanje kvalitetnih programa, čini će kôd biti dovoljno dobar za lako održavanje i kasnije snalaženje.

Slide 9



Princip bliskosti je jedan od temeljnih principa kojeg bi se svaki programer trebao pridržavati. On govori da bi se programske elemente (npr varijable) trebalo tako organizirati da ne budu razbacane. Dakle treba težiti da sva pojavljivanja neke varijable (npr. int i) budu na istom mjestu, tj. Što bliže jedno drugom. Programski jezik Java to itekako podržava i kao jedan od primjera se može navesti deklaracija varijabli koju je, za razliku od programskog jezika C, moguće napraviti bilo gdje u kôdu, a ne sam na početku bloka ili funkcije. Ako se varijabla i koristi tek negdje na sredini programa, tamo ju je potrebno i deklarirati.

Slide 10

Smjernica #1: Princip bliskosti (engl. Principle of proximity) (2)

- Prostor ranjivosti varijabli (engl. window of vulnerability)
  - Kôd koji se nalazi između mjesta uporabe iste varijable
  - Razlozi zbog kojih bi prostor ranjivosti trebao bi biti što manji
    - S vremenom se u njega može dodati novi kôd koji nenamjerno mijenja vrijednost varijable
    - Osoba koja mijenja kôd mogla bi zaboraviti koju vrijednost varijabla sadržava

Slide 11

Smjernica #2: Minimizacija dosega (engl. scope) (1)

Doseg programskih elemenata je potrebno minimizirati

Slide 12

Smjernica #2: Minimizacija dosega (engl. scope) (2)

- Doseg (engl. scope ili engl. visibility)
  - Područje programa u kojem je varijabla vidljiva i u kojem se može koristiti
  - Širok doseg = vidljivost na mnogo mjesta u programu
    - Npr. niz struktura s podacima o zaposlenicima koji se koristi kroz cijeli program
  - Ograničen doseg = vidljivost u malom dijelu programa
    - Npr. varijabla – brojač u for petlji (npr. u C++)
- Smjernice za smanjenje dosega varijable
  - Inicijalizirati varijable koje se koriste u petlji neposredno prije petlje radije nego na početku postupka
  - Inicijalizirati varijablu neposredno prije uporabe
  - Grupirati povezane naredbe (one koje koriste iste varijable)
  - Razlomiti grupe povezanih naredbi u posebne postupke
  - Pri programiranju krenuti sa što manjim dosegom te ga po potrebi proširiti

Slide 13

Smjernica #3: Korištenje varijabli (1)

Jednu se varijablu smije koristiti samo u jednu svrhu

Slide 14

Smjernica #3: Korištenje varijabli (2)

- Varijabla `temp` se u istome programu upotrebljava u dvije različite svrhe
  - Za pohranu vrijednosti diskriminante
  - Kao privremena vrijednost kod zamjene vrijednosti
- Loša praksa!
  - Čini kôd manje čitljivim
- Potrebne dvije varijable: `discriminant` i `oldRoot`
- Pitanje: što ne valja s imenima varijabli (vidjeti smjernicu #4)?

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
// swap the roots
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

C++

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

Slide 15

Smjernica #4: Imenovanje varijabli (1)

Varijablu je potrebno imenovati na način da joj ime oslikava svrhu u koju se koristi

Slide 16

Smjernica #4: Imenovanje varijabli (2)

Loše

```
x = x - xx;
xxx = fido +  salesTax(fido);
x = x + lateFee( x1, x ) + xxx;
x = x + interest( x1, x );
```

Dobro

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + salesTax( newPurchases );
balance = balance + lateFee( customerID, balance ) +
    monthlyTotal;
balance = balance + interest( customerID, balance );
```

Java

- Oba odsječka kôda rade istu stvar. Koji je jasniji? Koji je čitljiviji?
- Što radi odsječak kôda?
- Kada se nađete u situaciji u kojoj potrošite puno vremena na shvaćanje odsječka kôda, razmislite o preimenovanju varijabli

Slide 17

Smjernica #5: Magični brojevi (1)

Brojeve direktno upisane u programski kôd potrebno je uglavnom izbjegavati. Kao alternativu koristiti konstante ili pobrojani tip (enum).

Slide 18

Smjernica #5: Magični brojevi (2)

- Brojevi koji se pojavljuju direktno u kôdu
- Zamijeniti ih imenovanim konstantama
  - Npr. C: #define MAX_ENTRIES = 17
    Java: final int MAX_ENTRIES = 17
- Ukoliko ih izbjegavamo:
  - Izmjene programa su sigurne
    - Primjer: zamjena samo onih pojava broja 100 koje se odnose na jednu te istu stvar
  - Izmjene programa su jednostavnije
  - Kôd postaje čitljiviji

```
for i = 0 to 99 do ...

ILI

for i = 0 to MAX_ENTRIES-1 do ...
```

Pseudokôd

Slide 19

Smjernica #6: Funkcionalna kohezija (1)

Svaki postupak (funkcija) u programu mora obavljati jedan jasno definiran zadatak. To svojstvo postupka nazivamo funkcionalnom kohezijom.

Slide 20

Smjernica #6: Funkcionalna kohezija (2)

- Koliko usko su operacije unutar postupka povezane
- Vrlo važna kada razmatramo KAKO napraviti postupak
- Primjeri (pretpostavka da ime postupka precizno označava što postupak radi):
  - Funkcija `Cosine()` ima dobru koheziju jer je cijeli postupak posvećen jednom zadatku
  - Funkcija `CosineAndTan()` ima lošiju koheziju jer pokušava raditi više od jednog zadatka
- Cilj:Jedan postupak za jedan i samo jedan zadatak

Slide 21

Primjer 1. Prema kvalitetnijem kôdu (još uvijek bez OOP) (1)

- Postupak (metoda, funkcija) je jedan od najvažnijih “izuma” u računarstvu
- Čim je to moguće, “slobodne” programske linije potrebno je upakirati u postupak
  - Na taj način se poboljšava ponovna iskoristivost kôda
  - Kod postaje pregledniji te ga je lakše čitati i održavati

Slide 22

Primjer 1. Prema kvalitetnijem kôdu (još uvijek bez OOP) (2)

- Loše

```
int main_stari(){
float radijus1, radijus2, stranica_kvadrata;
float povrsina_kruznica1, povrsina_kruznica2,
povrsina_kvadrata;
float min;
printf("Unesite radijus1, radijus2, stranicu kvadrata ");
scanf("%f %f %f", &radijus1, &radijus2, &stranica_kvadrata);
povrsina_kruznica1 = radijus1 * radijus1 * 3.14;
povrsina_kruznica2 = radijus2 * radijus1 * 3.14;
povrsina_kvadrata = stranica_kvadrata * stranica_kvadrata;
printf("Kruznica1: %f, kruznica2: %f, kvadrat: %f\n",
povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata);
if (povrsina_kruznica1 < povrsina_kruznica2){
   min = povrsina_kruznica1;
}
else{
   min = povrsina_kruznica2;
}
if (povrsina_kvadrata < min){
   min = povrsina_kvadrata;
}
printf("Najmanja povrsina: %f", min);
return 0;
}
```

C

- Učitati radijus dvije kružnice
- Učitati stranicu kvadrata
- Odrediti koji je od tri učitana lika površinom najmanji i ispisati tu površinu

Slide 23

Primjer 1. Prema kvalitetnijem kôdu (još uvijek bez OOP) (3)

- Bolje – izdvojene funkcije koje se mogu ponovno koristiti (engl. reusability)

```c
#define PI 3.14

float izracunaj_povrsinu_kruznice(float radijus){
   return radijus * radijus * PI;
}

float izracunaj_povrsinu_kvadrata(float stranica){
   return stranica * stranica;
}

float najmanji(float x, float y, float z){
float min;
if (x < y){
   min = x;
}
else{
   min = y;
}
if (z < min){
   min = z;
}
return min;
}
```

C

```c
int main(){
//Učitati radijus dvije kružnice
//Učitati stranicu kvadrata
//Odrediti koji je od tri učitana lika površinom najmanji i
ispisati tu površinu
float radijus1, radijus2, stranica_kvadrata;
float povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata;
float min;
printf("Unesite radijus1, radijus2, stranicu kvadrata ");
scanf("%f %f %f", &radijus1, &radijus2, &stranica_kvadrata);
povrsina_kruznica1 = izracunaj_povrsinu_kruznice(radijus1);
povrsina_kruznica2 = izracunaj_povrsinu_kruznice(radijus2);
povrsina_kvadrata =
   izracunaj_povrsinu_kvadrata(stranica_kvadrata);
printf("Kruznica1: %f, kruznica2: %f, kvadrat: %f\n",
povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata);
printf("Najmanja povrsina: %f",
najmanji(povrsina_kruznica1, povrsina_kruznica2,
   povrsina_kvadrata));
return 0;
}
```

Slide 24

Primjer 2. Prema kvalitetnijem kôdu (na pola puta do OOP) (1)

- Organizacija “srodnog” koda u C-u u module s .h ekstenzijom

Kruznica.h

```
#define PI 3.14

float izracunaj_povrsinu_kruznice(float radijus){
   return radijus * radijus * PI;
}
```

Kvadrat.h

```
float izracunaj_povrsinu_kvadrata(float stranica){
   return stranica * stranica;
}
```

Pomocne_funkcije.h

```
float najmanji(float x, float y, float z){
float min;
if (x < y){
   min = x;
}
else{
   min = y;
}
if (z < min){
   min = z;
}
return min;
}
```

```
#include "Kruznica.h"
#include "Kvadrat.h"
#include "Pomocne_funkcije.h"

int main(){
//Učitati radijus dvije kružnice
//Učitati stranicu kvadrata
//Odrediti koji je od tri učitana lika površinom najmanji i
ispisati tu površinu
float radijus1, radijus2, stranica_kvadrata;
float povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata;
float min;
printf("Unesite radijus1, radijus2, stranicu kvadrata ");
scanf("%f %f %f", &radijus1, &radijus2, &stranica_kvadrata);
povrsina_kruznica1 = izracunaj_povrsinu_kruznice(radijus1);
povrsina_kruznica2 = izracunaj_povrsinu_kruznice(radijus2);
povrsina_kvadrata =
   izracunaj_povrsinu_kvadrata(stranica_kvadrata);
printf("Kruznica1: %f, kruznica2: %f, kvadrat: %f\n",
povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata);
printf("Najmanja povrsina: %f",
najmanji(povrsina_kruznica1, povrsina_kruznica2,
   povrsina_kvadrata));
return 0;
}
```

Slide 25

Primjer 3. Prema kvalitetnijem kôdu (još ¼ puta do OOP) (2)

- Organizacija “srodnog” koda u C-u u strukture koje zajedno sa srodnim funkcijama čine .h module

Kruznica.h

```
#define PI 3.14

typedef struct{
   double radijus;
} Kruznica;

float izracunaj_povrsinu_kruznice(Kruznica k){
   return k.radijus * k.radijus * PI;
}
```

Kvadrat.h

```
typedef struct {
   double stranica;
} Kvadrat;

float izracunaj_povrsinu_kvadrata(Kvadrat kv){
   return kv.stranica * kv.stranica;
}
```

Pomocne_funkcije.h

```
float najmanji(float x, float y, float z){
float min;
if (x < y){
   min = x;
}
else{
   min = y;
}
if (z < min){
   min = z;
}
return min;
}
```

Slide 26

Primjer 3. Prema kvalitetnijem kôdu (još ¼ puta do OOP) (3)

- U glavnom se programu koriste strukture definirane u .h modulima, na način da se šalju funkcijama iz modula koje s njima rade

```
#include "Kruznica.h"
#include "Kvadrat.h"
#include "Pomocne_funkcije.h"

int main(){
//Učitati radijus dvije kružnice
//Učitati stranicu kvadrata
//Odrediti koji je od tri učitana lika površinom najmanji i ispisati tu površinu
Kruznica k1, k2;
Kvadrat kv;
float povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata;
float min;
printf("Unesite radijus1, radijus2, stranicu kvadrata ");
scanf("%f %f %f", &k1.radijus, &k2.radijus, &kv.stranica);
povrsina_kruznica1 = izracunaj_povrsinu_kruznice(k1);
povrsina_kruznica2 = izracunaj_povrsinu_kruznice(k2);
povrsina_kvadrata = izracunaj_povrsinu_kvadrata(kv);
printf("Kruznica1: %f, kruznica2: %f, kvadrat: %f\n",
povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata);
printf("Najmanja povrsina: %f", najmanji(povrsina_kruznica1, povrsina_kruznica2, povrsina_kvadrata));
return 0;
}
```

Slide 27

Modeliranjem do objektno-orijentiranog kôda

Uz pomoć modeliranja probleme iz stvarnog svijeta prebacujemo u računalni. Pisanje svakog objektno-orijentiranog program počet će fazom modeliranja.

Slide 28

Modeliranje

- modeliranje (u računarstvu)
  - proces kojim komadić stvarnog svijeta pojednostavljujemo prema vlastitim potrebama
  - identificiramo ključne atribute i ponašanja

MODELIRANJE

ATRIBUTI:

nazivMarke
nazivModela
serijskiBroj
kapacitet

PerilicaRublja

PONAŠANJE:

prihvatiOdjeću()
prihvatiDeterdžent()
uključi()
isključi()

Slide 29

Preciznost modela

- ovisno o zahtjevima po našu aplikaciju, model može biti manje ili više precizan

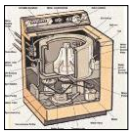
PerilicaRublja 1

ATRIBUTI:

nazivMarke
nazivModela
serijskiBroj
kapacitet

PONAŠANJE:

prihvatiOdjeću()
prihvatiDeterdžent()
uključi()
isključi()



PerilicaRublja 2

ATRIBUTI:

nazivMarke
nazivModela
serijskiBroj
kapacitet
volumenBubnja
motor
vrstaMotora

PONAŠANJE:

prihvatiOdjeću()
prihvatiDeterdžent()
uključi()
isključi()
prihvatiOmekšivač()
pokreniCentrifugiranje()

Slide 30

Primjer 4. Modeliranje prostorije u sustavu za rezervaciju dvorana

- Potrebno je modelirati prostoriju u sustavu za rezervaciju dvorana na FER-u. Za prostoriju je potrebno sačuvati informaciju o njenom nazivu, kapacitetu te je li zavodska ili javna. Prostoriji se može mijenjati kapacitet i status zavodska/javna.

MODELIRANJE

```
enum status_prostorije{
   Zavodska, Javna
};

typedef struct {
   char naziv[100];
   int kapacitet;
   status_prostorije status;
} Prostorija;

Prostorija postavi_status_prostorije(Prostorija p,
   status_prostorije status){
   p.status = status;
   return p;
}

Prostorija postavi_kapacitet(Prostorija p,
   int kapacitet){
   p.kapacitet = kapacitet;
   return p;
}
```

Slide 31

Zadaci za vježbu

1. Pronaći programski odsječak (po vlastitom izboru) koji ste pisali na prethodnim godinama studija i obrazložiti da li je u skladu sa smjernicama za oblikovanje kôda #1-#5.
2. Proširiti model iz Primjera 4. na način da se modeliraju i ostali entiteti u sustavu: učenici (JMBAG, ime, prezime), nastavnici (šifra, ime, prezime) i predmeti (naziv). Kako (i gdje) bi se modeliralo ponašanje kojim se uz predmet vežu nastavnici koji ga izvode? Za sve navedeno napisati kôd u C-u.

Slide 32

Pravila pisanja čitljivog koda

Slide 33

Sadržaj predavanja

- Pravila pisanja čitljivog koda
- Komentari u kodu
- Korištenje Eclipsea:
  - generiranje *Javadoca*
  - oznake u kodu
  - predlošci
  - *debugging*
  - Refaktoriranje

Slide 34

Čitljivost koda

- **elegancija** se isplati
- pretežno **mala** slova (*lowercase*)
- **praznine** se koriste zbog čitljivosti
  - **uvlačenje** ocrtava ustroj cijele datoteke koda
  - **razmaci** određuju ustroj izraza i izjava
  - **prazne linije** odvajaju blokove koda
- Standardi kodiranja:
  - svako pravilo se može **prekršiti** ako imate dobar razlog
  - pravila se primjenjuju **konzistentno**
  - standard: priklonite se **uobičajenom** korištenju
  - svi članovi **tima** ga se moraju **strogo pridržavati**

Slide 35

Identifikatori i "čarobni brojevi"

- na **engleskom** jeziku
- Identifikatori
  - klase i sučelja počinju **velikim**, svi ostali **malim** slovom
  - **logičke** varijable i metode: `isValid, canOpen`
  - dohvaćanje vrijednosti objekta: **get** → `getArea, getRealPart`
  - postavljanje vrijednost objekta: **set** → `setRealPart`
- **neimenovane konstante u kodu**
  - primjer: duljina polja u zadatku je 10
  - **nejasan** kod ("Otkud tu 10?")
  - teškoće u **promjeni** konstante
    - ponekad "10" znači nešto **drugo**
    - negdje piše "od 0 do 9"
  - **definirati** i **smisleno nazvati** konstantu:

```
private static final int ARRAY_SIZE = 10;
```

Slide 36

Klase i metode

- Klase
  - predstavlja **predmet**, imenicu
  - što je moguće jednostavnija
  - samo **jedna** namjena

```
public class ClassName
   extends SuperClass
   implements Interface1,
   Interface2
{
  // tijelo uvučeno za
  // jedan korak
}
```

- Metode
  - predstavlja djelovanje, glagol
  - poželjno jedan ekran, najdulje A4 **stranica**
  - pazite na dug niz argumenata
    - skupite **srodne** argumente u jednu klasu
    - suzite **funkcionalnost** metode

```
public void methodName(
     SomeClass argument
)
throws SomethingThrowable
{
  // tijelo uvučeno za
  // jedan korak
}
```


(!) dobar primjer realloc() funkcije u C-u iz Writing Solid Code

Slide 37

Softverska dokumentacija

- **dokumentacija** je informacija o proizvodu namijenjena korisniku
- **unutrašnja** i **vanjska**
- programer dokumentira (opisuje) svoj rad za:
  - samog **sebe (unutrašnja)**
  - (poznate) **članove tima (unutrašnja)**
  - (nepoznate) **buduće programere (unutrašnja)**
  - ljude koji će samo **koristiti (vanjska)**

Slide 38

Samodokumentirajući kod

- upute računalu ujedno razumljive ljudima
  - identifikator razumljiv po imenu
  - objekti domene rješenja odgovaraju točno pojmovima domene problema
  - kodne apstrakcije prate ljudske
- komentari:
  - samo najnužniji
  - jezgrovite, ali pune rečenice
  - usput, ne naknadno

Slide 39

Komentari (1)

- **pozicioniranje**: **prije** koda
- ponekad na kraju linije koda (*inline*) – samo ako je komentar **vrlo kratak** i takvim će ostati
- odaberite stil oblikovanja
  - minimalni trud pri pisanju
  - lako se može proširiti ili prepraviti
- jednolinijski komentar: `// komentar`
- vrste komentara:
  - izvedbeni komentari
    - **ponavljaju** kod, **objašnjavaju** kod, **sažetak** koda, opisuju **namjeru**
  - **javadoc** komentari
  - **iskomentirani** kod
  - **oznake** u kodu

Slide 40

Komentari (2)

- Komentari koji **ponavljaju** kod

```
// z je korijen iz zbroja kvadrata
z = Math.sqrt( x*x + y*y );
```

  - ne valja:
    - nema novih informacija
    - zakrčuje kod
- Komentari koji **objašnjavaju** kod
  - kad je kod **prekompliciran**
  - obično je bolje **popraviti** kod nego ga **objašnjavati**
  - često na osnovu pretpostavke da čitatelj ne zna programski jezik ili njegov dio
  - objašnjenja su nužna kod **optimiziranog** koda

Slide 41

Komentari (3)

- Komentari kao **sažetak** koda
  - **veći** dio koda se **sažeto opisuje**
  - na taj način se **brže** prolazi kroz kod
  - u jeziku domene **rješenja**
  - dobar postupak:
    - napisati sučelje metode
    - tijelo popuniti pseudokodom na prirodnom jeziku
    - zamijeniti svaku liniju pseudokoda sa stvarnim kodom
    - pseudokod pretvoriti u komentare
- Komentari koji opisuju **namjeru** autora
  - ako je iz koda jasno **što** se radi
  - treba odgovoriti **zašto** to autor radi
  - u jeziku domene **problema**

Slide 42

Javadoc komentari

- mehanizam generiranja vanjske **API dokumentacije** na osnovu komentara u kodu
- počinju s `/**`, završavaju s `*/`
- pišu se prije deklaracije klase, varijable ili metode
- može se formatirati u HTML-u
- posebne oznake u komentaru počinju s "`@`"
  - za klase: `author`, `version`...
  - za metode: `param`, `return`, `throws`, `deprecated`...

Slide 43

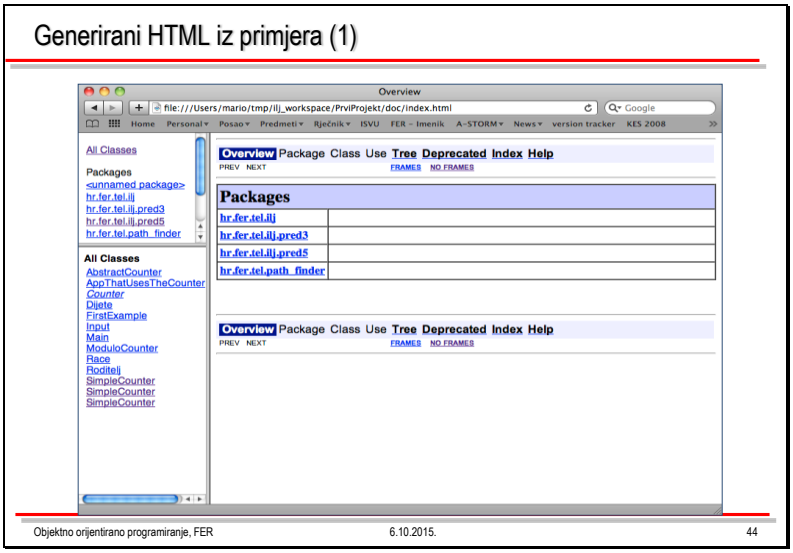
Primjer Javadoc izvornog koda

```
/**
 * This is Counter interface that is used for all Counters.
 *
 * @author Mario Kušek
 */
public interface Counter {

   /**
    * Increments counter by one.
    */
   void increment();

   /**
    * Returs the state of the counter.
    *
    * @return value of the counter
    */
   int getState();
}
```

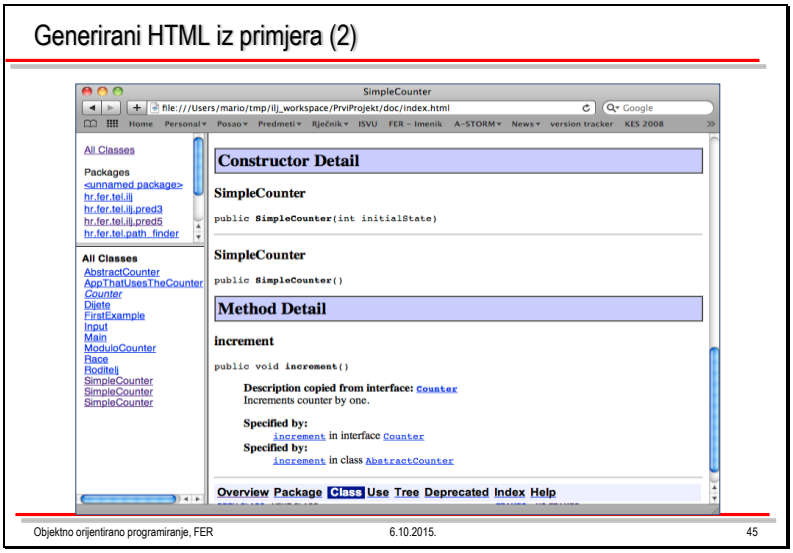
Slide 44



Generiranje Javadoca iz Eclipsa:
-Project -> Generate Javadoc…
-Odaberemo dostupnost Package
-Odaberemo Finish

U direktoriju doc unutar projekt će se generirati Javadoc.

Slide 45



Generiranje Javadoca iz Eclipsa:
-Project -> Generate Javadoc…
-Odaberemo dostupnost Package
-Odaberemo Finish

U direktoriju doc unutar projekt će se generirati Javadoc.

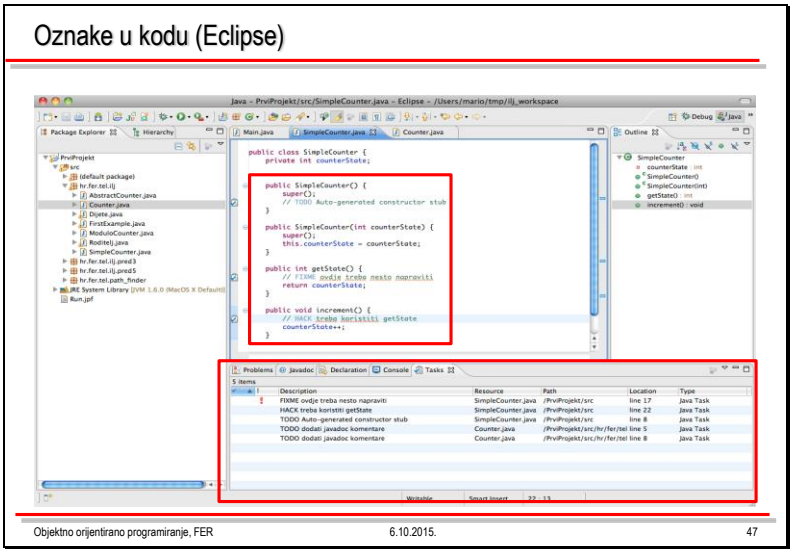
Slide 46

Komentari (4)

- **iskomentirani** kod
  - privremeno izbačen dio napisanog koda
  - jednolinijski – bez razmaka
    - `//System.println( "i = " + i );`
  - višelinijski sa **/*** i ***/**
- komentari koji služe kao **oznake** u kodu
  - oznake samom sebi ili drugim članovima tima
    - dio koda nije dovršen
    - ne nalazi se u konačnoj verziji koda
    - alati podržavaju brzo pronalaženje takvog koda
  - primjer:

```
public SimpleCounter() {
      super();
      // TODO Auto-generated constructor stub
}
```

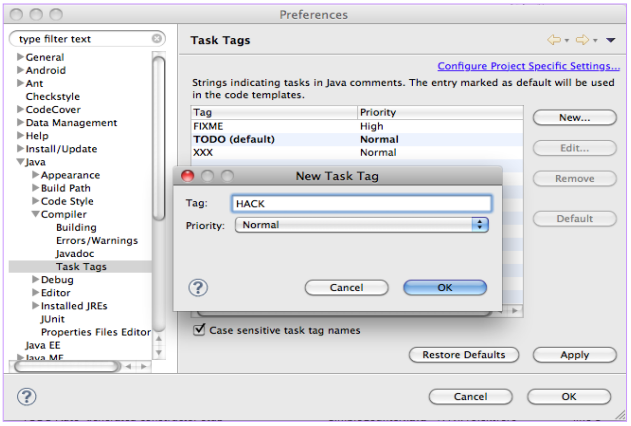
Slide 47



Za otvaranje pogleda (“view”) Tasks koristite sljedeće: Window -> Show view -> Other… -> General -> Tasks

Slide 48

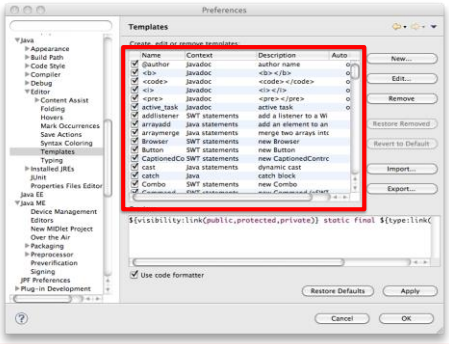
Dodavanje novih oznaka u kodu (Eclipse)



Slide 49

Nadopunjavanje koda (predlošci – *templates*) (Eclipse)

- Pregledavanje postojećih:
  - Eclipse → Preferences → Java → Editor → Templetes



Getting started with Eclipse code templates - http://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-code-templates/index.html

Slide 50

Najčešće korišteni predlošci koda (Eclipse)

- for – petlja po polju ili kolekciji
- new – stvaranje nove varijable ili atributa
- sysout – ispis na standardni izlaz
- try – obrada pogrešaka (blok `try`/`catch`)
- catch – obrada pogrešaka (blok `catch`)
- cast – pretvaranje is jedne vrste u drugu
- main – stvaranje metode `main`

Slide 51

Pronalazak sintaksnih pogrešaka (Eclipse)

- u kodu

AppThatUsesTheCounter.java

```
public class AppThatUsesTheCounter {

    public static void main(String[] args) {
        SimpleCounter c123 = new SimpleCounter();

        System.out.println("Prvi poziv:" + c123.getCounterState());

        c123.increment()
        System.out.println("Drugi poziv: + c123.getCounterState());

    }
}
```

- u pogledu Problems

Problems | Javadoc | Declaration | Console

5 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (5 items)				
String literal is not properly closed by a double-quote	AppThatUsesTheCounter.java	/Ilj/src	line 10	Java Problem
String literal is not properly closed by a double-quote	AppThatUsesTheCounter.java	/Ilj/src	line 10	Java Problem
Syntax error, insert ";" to complete Statement	AppThatUsesTheCounter.java	/Ilj/src	line 9	Java Problem
Syntax error, insert ";" to complete Statement	AppThatUsesTheCounter.java	/Ilj/src	line 10	Java Problem
Syntax error, insert ")" to complete Expression	AppThatUsesTheCounter.java	/Ilj/src	line 10	Java Problem

Slide 52

Ispravljanje sintaksnih pogrešaka (Eclipse)

AppThatUsesTheCounter.java

```
public class AppThatUsesTheCounter {

    public static void main(String[] args) {
        SimpleCounter c123 = new SimpleCounter();

        System.out.println("Prvi poziv:" + c123.getCounterState());

        c123.increment()
        System.out.println("Drugi poziv: + c123.getCounterState());

    }
}
```

- klik na crveni X - reći će koja je greška
- doći na podvučeni dio i pritisnuti: CTRL+1 ili kliknuti na

```
System.out.println("Drugi poziv: + c123.getCounterState());
```

Insert missing quote

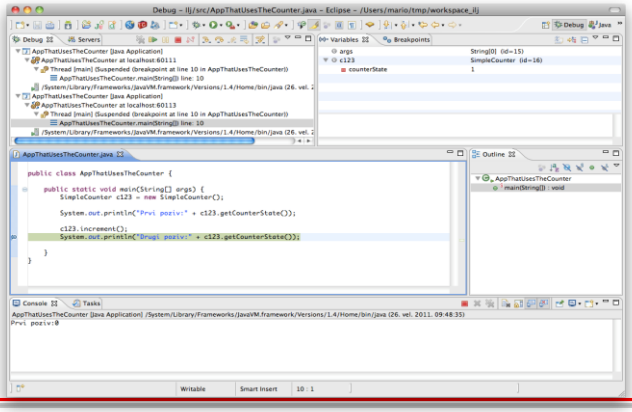
```
...
c123.increment()
System.out.println("Drugi poziv: +
 c123.getCounterState());"
...
}
```

Press 'Tab' from proposal table or click for focus

Slide 53

Debugging – ispravljanje grešaka (Eclipse)

- pokrenuti program u modu Debug: 



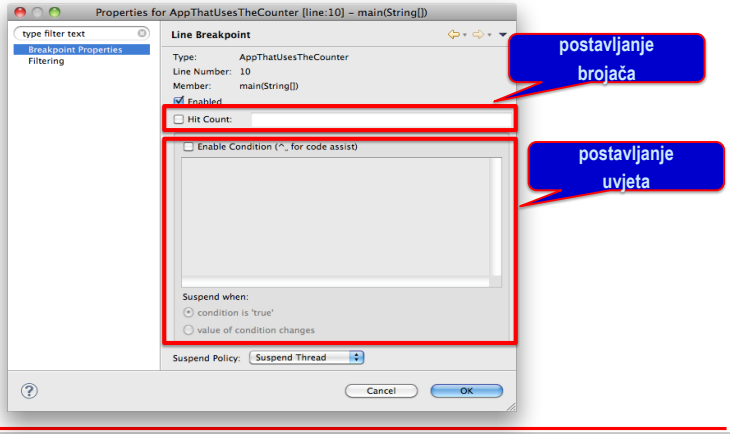
Slide 54

Debugging – ispravljanje grešaka (Eclipse)

- u tom modu možemo:
  - pregledavati i mijenjati varijable/atribute
  - prolaziti kroz kod
    - ulazak u metodu (F5)
    - izvršavanje naredbe(F6)
    - izlazak iz metode (F7)
    - nastaviti program izvođenjem (F8)
  - koristiti točku zaustavljanja (*breakpoint*)
    - postavljanje *breakpointa* (CTRL+SHIFT+B, dvostuki klik na početak retka)
    - *breakpoint* s brojačem – zaustavi se kada se x puta dođe na taj redak
    - *breakpoint* s uvjetom – zaustavi se kada je uvjet ispunjen

Slide 55

Postavke *breakpointa* (Eclipse)



Slide 56

Refaktoriranje (*refactoring*)

- postupak promjene koda uz zadržavanje iste funkcionalnosti
- ciljevi:
  - uklanjanje duplog koda
  - pojednostavljenje složene logike
  - objašnjavanje nejasnog koda
  - poboljšanje dizajna
- Eclipse ima ugrađene osnovne mehanizme refaktoriranja

Slide 57

Refaktoriranje u Eclipseu

- promjena naziva:
  - varijabli
  - metoda
- izvlačanje lokalne varijable (*extract local variable*)
- pretvaranje lokalne varijable u atribut (*convert local variable to field*)
- izvlačenje metode (*extract method*)
- izvlačenje sučelja (*extract interface*)
- izvlačenje nadklase (*extract superclass*)

Slide 58

Primjer refaktoriranja (ovo će biti jasno kasnije)

- napravimo klasu `SimpleCounter`
- napravimo klasu `ModuloCounter`
- izvlačimo nadklasu `AbstractCounter`
  - iz obje klase izvučemo atribut `counterState`, metodu `getState` i apstraktnu metodu `increment`
- izvlačimo sučelje `Counter`
- u klasu `Main` dodajemo petlju za provjeru brojenja do 15
- izvlačimo petlju u novu metodu
- promijenimo naziv metode
- promijenimo naziv parametra