

Java – Teoretske cake



1. Prvi karakter u identifikatoru(nazivi klasa, promijenjivih, metoda) ne smije biti broj

2. Ne može se konvertovati siri tip na uži

```
long a = 5L;  
int b = a;
```

3. Prethodno se može izvršiti kaštovanjem

```
long a = 5L;  
int b = (int)a;
```

4. Moguće je vršiti konverziju prema gore, odnosno iz podtipa u supertip što se vrši implicitno, a kod konverzije prema dole (suzavanje), iz supertipa u podtip potrebno je vršiti kaštovanje
5. **Lokalne promijenjive (koje nisu parametri klase, već se nalaze kao pomoćne promijenjive u metodama, konstruktorima i blokovima) se ne inicijalizuju implicitno prilikom kreiranja, tako da ih je potrebno ručno (eksplicitno) inicijalizovati inace kompajler prijavljuje gresku. Isto važi za lokalne reference samo što je njih moguće inicijalizovati null referencom, ali onda može doći do greske prilikom izvršavanja** 
6. Kod operatora "=", ako su operandi reference, kopira se samo sadržaj reference a ne kompletni objekti na koje ukazuju
7. Ako je jedan od operandi kod operatora "=" klase String, cio izraz je String
8. Break – prekida tijelo tekuće ciklične strukture i izlazi iz nje, a continue prekida tijelo tekuće ciklične strukture i otpočinje sledeću iteraciju petlje
9. Moguće je deklarirati metodu koja ima naziv identičan nazivu konstruktora (i nazivu klase) 
10. Moguće je da se u jednoj datoteci nađe više definicija različitih klasa
11. Naziv datoteke treba da bude identičan nazivu klase, i sa ekstenzijom .java
12. Klasa koja ima public vidljivost mora biti definisana u datoteci sa nazivom koji je identičan nazivu klase i ekstenzijom .java (ovo pravilo određuje da izvorna






datoteka ne može sadržavati više od jedne public klase, tj. ako se u jednoj izvornoj datoteci nalaze definicije više klasa, onda samo jedna može biti public i izvorna datoteka će imati naziv koji odgovara nazivu public klase (sa ekstenzijom .java)). Svaka definicija klase iz iste izvorne datoteke kompajlira se u zasebnu datoteku sa .class ekstenzijom







13. može postojati samo jedna deklaracija paketa u datoteci sa izvornim kodom, tj. jedna klasa se ne može nalaziti u više paketa
14. korištenje tipova (klasa i interfejsa) moguće je i bez uvoza, ali se u tom slučaju navodi puno kvalifikovano ime tipa
15. deklaracija uvoza u datoteci sa izvornim kodom se mora nalaziti iza deklaracije paketa
16. identifikatori klase mogu maskirati statičke članove koji se uvoze, u ovim slučajevima jedini način pristupa određenoj statičkoj metodi ili statičkom atributu jeste navođenjem njenog/njegovog punog kvalifikovanog imena
17. iz statičkog konteksta može se pristupiti samo statičkim članovima
18. kako se statički kod ne izvršava u kontekstu objekta, tako i reference this i super nisu dostupne iz statičkog konteksta
19. iz nestatičkog konteksta su uvijek dostupni i statički članovi
20. lokalne promjenljive metode obuhvataju formalne parametre metode, kao i promjenljive koje su deklarirane unutar tijela metode i kreiraju se iznova, pri svakom pozivu metode
21. bitno je napomenuti da objekti nemaju isti opseg vidljivosti kao primitivni tipovi i reference, jer objekat će postojati na heap-u sve dok ne bude uklonjen od strane garbage collector-a
22. modifikator pristupa public može se koristiti za deklaraciju top-level tipova (klasa, interfejsa i enumeracija) u paketima, kako bi one mogle biti dostupne kako iz svog paketa, tako i iz drugih

Public – dostupan iz svih paketa
Podrazumijevani – dostupan samo iz svog paketa

23. modifikator abstract se koristi u deklaraciji klase kako bi označio da klasa ne može biti instancirana, tj. kako bi označio da je klasa apstraktna

24. svaka klasa može biti deklarirana kao apstraktna klasa, ali klase koje imaju jednu ili više apstraktnih metoda (metode deklarirane ključnom riječju `abstract` i koje nemaju tijelo) moraju biti deklarirane kao apstraktne klase. Ovakve klase se mogu koristiti kako bi specificirale određeno ponašanje, ali se klasama nasljednicama ostavlja da obezbijede odgovarajuću implementaciju. Klase nasljednice, da bi mogle biti instancirane, moraju obezbijediti implementaciju svih apstraktnih metoda nasljeđenih iz apstraktne klase – u suprotnom, i klase nasljednice moraju biti deklarirane kao apstraktne
25. interfejsi specificiraju samo apstraktne metode, bez implementacije bilo koje metode – interfejsi su po svojoj prirodi implicitno apstraktni i ne mogu biti instancirani
26. interfejs je moguće deklarirati kao apstraktan, korištenjem ključne riječi `abstract`, ali je to neoptrebno i predstavlja redundansu, **dok enum tipovi ne mogu biti deklarirani kao apstraktni**
27. modifikator `final` se koristi u deklaraciji klase kako bi označio da klasa ne može biti nasljeđena, tj. kako bi označio da ne mogu postojati klase nasljednice date klase
28. samo klase koje su u potpunosti definisane, tj. kod kojih su sve metode implementirane, mogu biti deklarirane kao `final` klase - prema tome, klasa ne može u isto vrijeme biti deklarirana i kao `abstract` i kao `final`
29. navođenjem modifikatora pristupa članova, klasa kontroliše koji od njih su dostupni drugim klasama (klijentima), modifikatori pristupa članova su: `public`, `protected`, `private` i nespecificirani (prijateljski, `friendly`) pristup
30. bitno je napomenuti da modifikatori pristupa članova klase imaju značenje samo ako je klasa ili klasa nasljednica dostupna klijentima (drugim klasama); ako klasa nije deklarirana kao `public` klasa, onda `public` modifikator pristupa članova date klase nema nikakvo značenje za klase iz drugih paketa
31. specijalni karakteri `[] { } () , . * + ? $ " ' \ ^`
32. u interfejsu iako metoda nema modifikator u klasi nasljednici mora biti napisan `public` inace će kompajler prijaviti gresku, jer je podrazumijevani `friendly` što znači smanjili smo prava pristupa

33. metoda join blokira izvršavanje niti odakle je pozvana (u većini slučajeva main koja kreira sve ostale niti) sve tok metoda nad kojom je pozvana ne završi sa radom
34. ako se kod stringova za poredjenje koristi operator = poredi se da li ta dva stringa referenciraju isti objekat, a ako se koristi equals poredi se da li su im vrijednosti objekata iste 
35. ako imamo String s="abc"; -svi stringovi npr s2,s3 kojima je dodjeljen abc referenciraju isti objekat samo ima jedan stvarni objekat "abc"
36. ako klasa implementira interfejs serializable a ima referencu koja referencira klasu koja ne implementira serializable pri serijalizaciji prijave gresku 
37. Klasa koja nasledjuje Externalizable mora biti deklarirana kao public inace se nece moci deserijalizovati prijave gresku-prvo se izvršava podrazumijevani konstruktor pa se onda ocitaju vrijednosti-iako smo neke promjenljive deklarirali kao tranzientne pri DESerijalizaciji ce postojati jer se ponovo konstruise objekat pa im normalno mozemo pristupiti
38. Reference na steku, objekti na hipu 
39. Nizovi u javi predstavljeni kao objekti i mogu imati primitivne tipove kao i objekte
40. Ime klase mora odgovarati imenu datoteke
41. U okviru istog paketa samo jedna klasa moze biti deklarirana kao public
42. Java ne podrzava literale u binarnom brojnom sistemu
43. U jednoj datoteci moze biti samo jedna naredba package....tj jedna datoteka ne moze biti u dva paketa
44. Deklaracija uvoza u datoteci mora biti iza deklaracije paketa tj
package p;
import java.io.*; nikako obratno
45. Nad boolean tipom se moze primjeniti & | ^ (tj svi osim ~ negacije) 
46. etfbl.net net.etfbl
47. etf-bl.net net.etf_bl
48. student.etfbl.int int_.etfbl.student
49. paket.24sata.info info._24sata.paket 
50. svaka klasa moze biti deklarirana kao apstraktna, ali klasa koja ima bar jednu apstraktnu metodu obavezno mora biti deklarirana kao apstraktna

51. ako klasa naslednica nasledjuje apstraktnu klasu I ako ne implementira sve njene metode I ona mora biti deklarirana kao apstraktna.
52. Interfejsi su apstraktni I ne mogu se instancirati
53. Klasa kod koje je modifikator final njene metode se ne mogu redefinisati prijavljena ce biti greska I ta klasa mora imati implementirane sve metode sto znaci klasa u isto vrijeme ne moze biti I final I abstract
54. Jedan clan klase moze imati samo jedan modifikator pristupa inace greska
55. Enum tipovi ne mogu biti apstraktni
56. Modifikator pristupa apstraktne metode u nekoj klasi ne moze biti private jer se klasa ne moze redefinisati 
57. Kako staticke metode ne mogu biti redefinisane tako one ne mogu biti deklarirane kao apstraktne 
58. Final metode ne mogu biti nekompletne tj ne implementirane
59. Error I runtime izuzeci ne moraju ispuniti catch ili specifični zahtjev jer su necekirani izuzeci
60. Catch blok ne mora postojati, ali mora biti finally
61. Catch ili finally se ne mogu pojaviti prije try bloka I ne mogu se pojaviti bez try bloka
62. Kada redefinisemo metodu mozemo baciti sve, manje, ni jedan ili podklase izuzetaka 
63. Ako se u try catch bloku hvata izuzetak roditeljske klase pa klase naslednice prijave se greska 
64. Konstruktori I inicijalni blokovi se ne mogu naslediti 
65. U drugim paketima se nasledjuju samo varijable koje imaju public I protected dostupnost 
66. Klasa koja nije definisana kao public nece biti dostupna izvan svog paketa

67. REDEFINISANJE METODE

Metoda mora imati isti naziv, tip, broj I redosled parametara

Povratni tip moze biti kovarijantan tip


Nova metoda ne moze smanjiti dostupnost, ali je moze povecati


Moze baciti sve, ni jedan, ili podskup provjerenih izuzetaka koji su specificirani u throws klauzuli metode koju redefinisuje

SAMO U KLASI NASLEDNICI

68. Staticni članovi žive na nivou klase

69. U klasi nasljednici se ne može redefinisati metoda roditeljske klase bice greska

70. Staticka metoda u klasi nasljednici može sakriti staticku metodu roditeljske klase 

71. Apstraktna klasa može imati staticka polja I metode 

72. PREKLAPANJE


Metode imaju isti naziv, ali razlicitu listu parametara

Lista parametara se može razlikovati kako po tipu tako I po broju


Nije dovoljno da se metode razlikuju samo po povratnom tipu da bi bile preklopljene

I metode instance I staticke metode mogu biti preklopljene kako u osnovnoj tako I u klasi nasljednici


U ISTOJ KLASI ILI U KLASI NASLJEDNICI


73. Polje osnovne klase ne može biti redefinisano u klasi nasljednici, ali može biti maskirano 


74. Polje u klasi nasljednici može maskirati staticko polje osnovne klase

75. Tip polja nije bitan vazan je samo naziv 


76. Staticka metoda klase nasljednice može maskirati staticku metodu roditeljske klase ako vaze isti uslovi kao I za redefinisanje metoda instanci


77. Ako su im potpisi isti a zahtjevi kao throws klauzule ili tipa nisu isti prijave se greska 

78. Ako su potpisi metoda razliciti onda je rijec o preklapanju, a ne o maskiranju 

79. Staticka metoda u klasi nasljednici ne može maskirati metodu instance osnovne klase 

80. Prva linija u polju konstruktora ili this ili super ne može oboje 


81. Kod preklapanja metoda NEMA OGRANICENJA ZA POVRATNI TIP, ZA IZUZETKE I ZA DOSTUPNOST 

82. Ako klasa nema konstruktor kompajler će jos ugraditi podrazumijevani BEZ PARAMETARA 

83. Interfejsi mogu biti I bez tijela

84. Sve metode interfejsa su implicitno abstract I public

85. Sve konstante u interfejsu su public static final 

86. Metode interfejsa koje implementira klasa moraju imati public dostupnost 

Klasa ne može specificirati novi izuzetak ako ga nema u metodi interfejsa 🐞

87. Kriterijumi za redefinisanje metode vaze i pri implementaciji interfejsa

88. Metode interfejsa uvijek moraju biti implementirane kao metode instance a ne kao staticke metode 🐞

89. Metode interfejsa ne mogu biti ni staticke ni final ni private ni protected

Podrazumijevano u interfejsu su public abstract

Podinterfejs može redefinisati metode svojih superinterfejsa pri tome redefinisane metode nisu nasledjene 🐞

90. Deklaracije apstraktnih metoda mogu biti i preklopljene analogno preklapanju metoda kod klasa 🐞

91. KONSTANTE U INTERFEJSIMA MORAJU BITI INICIJALIZOVANE. 🐞

92. KONSTANTE U KLASAMA MORAJU BITI INICIJALIZOVANE PRIJE KORISTENJA 🐞

93. Metoda toString vraća NazivKlase@hes vrijednost objekta

94. Klasa StringBuffer je sinhronizovana

95. Instance klase FILE su nepromjenljive

96. SERIJALIZACIJA

Serijalizujemo objekte, a ne klase

Tranzijentne i staticke članove ne možemo serijalizovati implicitno 🐞

97. KONKURENTNO PROGRAMIRANJE

98. Metodu sleep staviti u try catch blok 🐞

99. Interferencija niti

100. Greške usljed nekonzistentnosti memorije

101. Happens before relacija

Sinhronizacija, volatile promjenljive start i join

Monitor može biti u posjedu samo jedne niti u jednom vremenskom periodu

102. Sinhronizaciju možemo izvršiti preko sinhronizacionih metoda i preko sinhronizacionih blokova

103. Synchronized se koristi samo za metode ne može za polja

104. Sinhronizacija konstruktora nije moguća 🐞

105. REENTRANT sinhronizacija

Omogućava nitima da više puta dodju u posjed monitora

Tamo gdje sinhronizacioni kod poziva metodu koja također ima sinhronizacioni kod i oba segmenta koriste isti monitor

Bez ovoga bi se tesko izbjeglo sopstveno blokiranje niti
Postoje tri vrste blokiranje mrtvo,zivo I izgladnjivanje
Objekat je immutable ako njegovo stanje posle kreiranja ne moze biti
promjenjeno

Za postizanje immutable:

Ne smiju biti seteri

Polja moraju biti private I final

Deklarisati klasu kao final a moze I privatni konstruktor

GENERICKI TIPOVI

- 106. Za rjesavanje compile time bug-ova
- 107. Run time bug-ova
- 108. Daju odredjenu stabilnost
- 109. Detekcija odredjenih gresaka
- 110. Tipskoj promjenljivoj T se ne moze pristupiti iz statickog konteksta 🚫
- 111. Genericki tip koji nije deklarisan kao final moze biti nasljedjen 🚫
- 112. Kod mape I kljucevi I vrijednosti moraju biti objekti 🚫
- 113. modifikator public – označava da je član klase vidljiv za sve klase u bilo
kojem paketu programa – ovo važi i za članove instance i za statičke članove 🚫
- 114. modifikator protected – označava da je član klase vidljiv samo za klase u
istom paketu i za sve klase nasljednice u bilo kojem paketu – sve ostale klase
ne mogu pristupiti protected članovima 🚫
- 115. modifikator private – označava da je član klase vidljiv samo unutar svoje
klase – ovo važi i za klase nasljednice, bez obzira da li se one nalaze u istom ili
drugom paketu 🚫
- 116. paketski ili prijateljski (friendly) pristup – ako modifikator pristupa člana nije
specificiran – podrazumijevani modifikator označava da je član klase vidljiv
samo od strane klasa u istom paketu (podrazumijevani modifikator je
restriktivniji od protected modifikatora) 🚫
- 117. modifikator static – statički članovi pripadaju klasi u kojoj su deklarisani i
nisu dio niti jedne instance klase
- 118. modifikator final – promjenljiva u čijoj deklaraciji se nalazi modifikator final
je konstanta. Njena vrijednost ne može biti promjenjena nakon inicijalizacije,
tj. vrijednost promjenljive primitivnog tipa ne može biti promjenjena, kao ni

vrijednost reference (final reference uvijek pokazuju na isti objekat). Ključna riječ final ne može uticati na to da stanje objekta na koji ukazuje final referenca ne bude izmijenjeno 🚫

119. promjenljive koje su static i final se najčešće koriste za definisanje konstanti
120. promjenljive definisane unutar interfejsa su implicitno final promjenljive 🚫
121. final promjenljiva ne mora biti inicijalizovana pri deklaraciji, ali mora biti inicijalizovana prije nego što bude korištena 🚫
122. final promjenljive obezbjeđuju da vrijednost ne može biti promjenjena, dok final metode obezbjeđuju da ponašanje ne može biti promjenjeno 🚫
123. modifikator pristupa apstraktne metode ne može biti private, jer u tom slučaju ne bi bilo moguće da klasa koja nasljeđuje apstraktnu klasu redefiniše apstraktnu metodu i obezbijedi njenu implementaciju
124. samo metode instance mogu biti deklarisanе kao apstraktne –kako statička metoda ne može biti redefinisana, nije dozvoljeno ni njeno deklarisanje kao apstraktne metode 🤝
125. final metoda ne može biti apstraktna, jer ne može biti nekompletna
126. metode specificirane u interfejsu su implicitno apstraktne metode, tako da modifikator abstract nije potrebno ni navoditi u deklaraciji metode interfejsa
127. modifikator synchronized može biti korišten pri deklaraciji metoda, ne i promjenljivih 🚫
128. native metode su metode čija implementacija nije definisana u Java programskom jeziku, već u nekom drugom programskom jeziku (npr., C, C++, ...)
129. modifikator transient može biti korišten pri deklaraciji promjenljivih, ne i metoda, ove promjenljive se koriste kada je potrebno sačuvati stanje objekta koristeći serijalizaciju 🚫
130. modifikator volatile može biti korišten pri deklaraciji promjenljivih, ne i metoda; ovaj modifikator se koristi kada je potrebno upozoriti kompajler da ne vrši optimizacije nad poljem koje je deklarisan kao volatile, a koje bi mogle dovesti do nepredvidivih rezultata kada se polju pristupa od strane više niti
131. Klasa Throwable je roditeljska klasa svih grešaka (klasa Error i njene klase nasljednice) i svih izuzetaka (klasa Exception i njene klase nasljednice)

132. Samo objekti koji su instance ove klase ili neke od njenih klasa nasljednica mogu biti bačeni od strane JVM ili putem throw naredbe. Slično, samo ova klasa ili neka od njenih klasa nasljednica može biti tip argumenta catch klauzule
133. nakon try bloka, mora doći barem jedan catch blok ili finally blok
134. moguće je da catch blok ne postoji, ali u toj situaciji finally blok mora biti specificiran
135. blokovi catch i finally moraju da se pojave nakon try bloka i ne mogu da se pojave bez try bloka
136. nakon završetka catch bloka, izvršavanje se nastavlja u finally bloku, ako je ovaj blok specificiran, i to bez obzira na to da li je novi izuzetak bačen u catch bloku. 🚫
137. situacija u kojoj catch blok ima argument čiji je tip u hijerarhiji iznad tipa argumenta nekog od narednih catch blokova nije dozvoljena – ovakvu situaciju prijavio je kompajler, jer naredni catch blok ne bi nikad mogao da se izvrši. 🚫
138. finally blok – uvijek se izvršava, bez obzira na to da li se desio izuzetak u try bloku i da li je izvršen catch blok
139. Nasljeđivanje članova zavisi od njihove dostupnosti. Ako je članu osnovne klase moguće pristupiti navođenjem naziva tog člana, bez bilo kakve dodatne sintakse (poput ključne riječi super), onda se taj član smatra nasljeđenim. Iz tog razloga privatni, redefinisani i sakriveni članovi osnovne klase nisu nasljeđeni. 🚫
140. izvedena klasa nasljeđuje sve public i protected članove roditeljske klase, bez obzira u kojem paketu je izvedena klasa. Ako je izvedena klasa u istom paketu kao i roditeljska, nasljeđuje i friendly članove; članovi koji imaju podrazumijevanu dostupnost u osnovnoj klasi ne mogu biti nasljeđeni od strane klasa u drugim paketima 🚫
141. Konstruktori i inicijalizacioni blokovi ne mogu biti nasljeđeni, jer nisu članovi klase 🚫
142. U Java programskom jeziku nije dozvoljeno višestruko nasljeđivanje, tj. svaka klasa može imati samo jednu roditeljsku klasu –jednostruko nasljeđivanje

143. nasljeđivanje definiše relaciju „je vrsta“ između osnovne i klase nasljednice
– ova relacija podrazumijeva da se vrijednost reference objekta klase nasljednice može dodijeliti referenci objekta osnovne klase, jer objekat klase nasljednice može da se pojavi gdje god se očekuje objekat osnovne klase - ova dodjela podrazumijeva proširivanje reference (eng. widening reference conversion)

144. referenca kalkulator1 se može koristiti za poziv metoda nad objektom klase ProsireniKalkulator koje su nasljeđene iz klase Kalkulator

```
Kalkulator kalkulator1 = new ProsireniKalkulator(1,2); // 1
int zbir = kalkulator1.zbir(); // 2
// int proizvod = kalkulator1.proizvod(); // 3
```

145. bitno je primijetiti da kompajler u vrijeme prevođenja programa ima saznanje o deklarisanom tipu reference

– na osnovu toga što je tip reference kalkulator1 Kalkulator, kompajler ograničava da samo metode iz klase Kalkulator mogu biti pozivane putem ove reference 🚫

– za vrijeme izvršavanja referenca kalkulator1 referenciraće objekat klase ProsireniKalkulator – iako referenca kalkulator1 referencira objekat klase ProsireniKalkulator nije moguće pozvati metodu proizvod, jer kompajler u vrijeme prevođenja nema saznanje o tome koji objekat će referenca kalkulator1 referencirati, tj. kompajler ima saznanje samo o deklarisanom tipu reference (poziv metode proizvod u ovom slučaju rezultiraće greškom pri kompajliranju) 🚫

146. klasa nasljednica može redefinisati metode osnovne klase - redefinisane metode omogućava da klasa nasljednica obezbijedi vlastitu implementaciju metode koju bi inače naslijedila iz osnovne klase

147. redefinisana metoda osnovne klase se u ovoj situaciji ne nasljeđuje, a nova metoda u klasi nasljednici mora da ispunjava sljedeće:

- mora da ima isti potpis (naziv metode, tip, broj i redoslijed parametara),
- povratni tip nove metode može biti podtip povratnog tipa redefinisane metode (kovarijantni povratni tip),
- nova metoda ne može smanjiti dostupnost metode, ali je može povećati,

– nova metoda može baciti sve, nijedan ili podskup provjerenih izuzetaka (uključujući i njihove podklase) koji su specificirani u throws klauzuli redefinisane metode

148. Metoda instance u klasi nasljednici ne može redefinirati statičku metodu osnovne klase – pokušaj kompajliranja dovešće do greške - statička metoda u klasi nasljednici može sakriti statičku metodu osnovne klase 🚫

149. Metode u čijoj deklaraciji se nalazi modifikator final ne mogu biti redefinirane – pokušaj redefiniranja ovakve metode dovešće do greške pri kompajliranju 🚫

150. Apstraktne metode u klasama nasljednicama koje nisu apstraktne moraju biti redefinirane, tj. implementacija ovih metoda mora biti obezbjeđena

151. Modifikator private u deklaraciji metode označava da metoda nije dostupna za klase nasljednice, pa iz tog razloga ova metoda ne može biti redefinirana

152. ako klasa ima samo apstraktne metode – onda treba biti interfejs, a ne apstraktna klasa

153. klase koje su deklarirane kao abstract mogu, ali ne moraju, imati apstraktne metode 🚫

154. apstraktna klasa može imati statička polja i statičke metode i može implementirati interface

155. povratni tip nove metode može biti podtip povratnog tipa redefinirane metode – ovaj povratni tip naziva se kovarijantnim povratnim tipom

156. klasa nasljednica mora koristiti ključnu riječ super kako bi pozvala redefiniranu metodu roditeljske klase

157. preklapljene metode su metode koje imaju isti naziv, ali različitu listu parametara - lista parametara se može razlikovati po tipu, redoslijedu i/ili broju parametara 🚫

158. kako povratni tip nije dio potpisa metode, nije dovoljno da se metode razlikuju samo po povratnom tipu da bi bile preklapljene

159. i metode instance i statičke metode mogu biti preklapljene u osnovnoj ili klasi nasljednici 🚫

160. ako klasa posjeduje više konstruktora oni su uvijek preklapljeni, jer svi konstruktori imaju isti naziv (koji odgovara nazivu klase), pa se njihovi potpisi

razlikuju samo po listi parametara

	redefinisanje	preklapanje
naziv metoda	mora biti identičan	mora biti identičan
lista parametara	mora biti identična	ne smije biti identična
povratni tip	identičan ili kovarijantan	nema ograničenja
izuzeci	svi, nijedan ili podskup izuzetaka redefinisane metode	nema ograničenja
dostupnost	ne može se smanjiti	nema ograničenja
lokacija	samo u klasi nasljednici	u istoj ili u klasi nasljednici
izvršavanje metode određuje	tip objekta referenciran u vrijeme izvršavanja	deklarisani tip reference

6

161. polja osnovne klase ne mogu biti redefinisana u klasi nasljednici, ali mogu biti maskirana
162. polja se maskiraju tako što se u klasi nasljednici definiše polje sa istim nazivom kao u osnovnoj klasi – u ovakvom slučaju poljima osnovne klase se ne može pristupiti direktno, pa je jasno da se ona ne nasljeđuju – pristup poljima osnovne klase koja se ne nasljeđuju, uključujući i maskirana polja, može se ostvariti korištenjem ključne riječi super
163. za razliku od redefinisanja metoda, gdje metoda instance ne može redefinisati statičku metodu, kod maskiranja polja ne postoji takvo ograničenje – polje instance u klasi nasljednici može maskirati statičko polje osnovne klase
164. tip maskiranih polja nije bitan, važan je jedino naziv polja
165. statička metoda u klasi nasljednici može maskirati statičku metodu osnovne klase, ako su uslovi identični uslovima za redefinisanje metoda instance
166. ako su potpisi metoda identični, a drugi zahtjevi poput povratnog tipa, throws klauzule i dostupnosti nisu ispunjeni, kompajler će prijaviti grešku
167. ako su potpisi metoda različiti, onda je riječ o preklapanju metoda, a ne o maskiranju
168. maskirana statička metoda uvijek može biti pozvana preko imena osnovne klase u kodu klase nasljednice, kako iz statičkog tako i iz nestatičkog konteksta – moguće je koristiti ključnu riječ super za poziv maskirane statičke metode, u nestatičkom kontekstu u kodu klase nasljednice

- 169. statička metoda u klasi nasljednici ne može maskirati metodu instance osnovne klase – pokušaj ovakvog maskiranja dovešće do greške pri kompajliranju** 🚫
170. ključna riječ `super` označava referencu koju je moguće koristiti u nestatičkom kontekstu, ali samo u klasi nasljednici, radi pristupa poljima i pozivanju metoda osnovne klase 🚫
171. pri pozivu metoda korištenjem ključne riječi `super`, metoda osnovne klase se poziva bez obzira na stvarni tip objekta i bez obzira na to da li je metoda redefinisana u klasi nasljednici 🚫
172. konstruktori ne mogu biti nasljeđeni niti redefinisani, ali mogu biti preklopljeni unutar iste klase
173. ulančavanje: korištenjem `this` i korištenjem `super`
174. konstrukcija `this()` sa odgovarajućom listom parametara rezultira pozivom odgovarajućeg konstruktora
175. obavezno je da poziv konstruktora korištenjem konstrukcije `this()` bude prva naredba u tijelu konstruktora
176. konstrukcija `super()` se koristi u konstruktoru klase nasljednice radi poziva odgovarajućeg konstruktora neposredne roditeljske klase – koji konstruktor roditeljske klase će biti izvršen zavisi od liste parametara `super()` konstrukcije – preporuka je da se konstrukcija `super()` koristi za inicijalizaciju nasljeđenog stanja • konstrukcija `super()` mora biti prva naredba u konstruktoru i moguće ju je koristiti samo u konstruktoru
177. kako moraju biti prve naredbe u konstruktoru, jasno je da se konstrukcije `this()` i `super()` ne mogu pojaviti u istom konstruktoru 🚫
178. ako se u konstruktoru klase nasljednice ne nalazi eksplicitan poziv konstrukcije `super()`, kompajler će implicitno ubaciti poziv ove konstrukcije (bez argumenata) kako bi pozvao podrazumijevani konstruktor roditeljske klase 🚫 ■
- 179. ako roditeljska klasa ne posjeduje podrazumijevani konstruktor, tj. ako roditeljska klasa posjeduje samo konstruktore sa parametrima, onda ubacivanje poziva konstrukcije `super()` od strane kompajlera u konstruktore klase nasljednica neće imati efekta – u ovakvim situacijama desiće se greška pri kompajliranju – da bi se ova greška izbjegla u konstruktorima klase**

nasljednica potrebno je eksplicitno pozvati konstruktor roditeljske klase, korištenjem konstrukcije super() sa odgovarajućim argumentima

- 180. interfejse implementiraju klase, pa iz tog razloga članovi interfejsa implicitno imaju public dostupnost – iz tog razloga modifikator public može biti izostavljen u deklaraciji članova interfejsa
- 181. da bi klasa implementirala interfejs, mora da implementira sve njegove metode
- 182. sve konstante definisane u interfejsu su implicitno public, static i final 🖱️
- 183. metode interfejsa moraju imati public dostupnost pri implementaciji u klasi (ili njenim klasama nasljednicama)
- 184. klasa ne može smanjiti dostupnost metode interfejsa, niti može specificirati novi izuzetak u throws klauzuli metode, jer će to dovesti do narušavanja ugovora interfejsa – što nije legalno
- 185. klasa može implementirati samo neke od metoda interfejsa, tj. klasa može obezbijediti parcijalnu implementaciju interfejsa – u ovom slučaju klasa mora biti deklarirana kao apstraktna 🖱️
- 186. metode interfejsa uvijek moraju biti implementirane kao metode instance, a ne kao statičke metode 🖱️
- 187. bez obzira koliko interfejsa klasa implementira, direktno ili indirektno, ona posjeduje samo jednu implementaciju metode koji može biti deklarirana u više interfejsa 👍
- 188. jedan interfejs može da naslijedi jedan ili više drugih interfejsa, tj. kod interfejsa postoji višestruko nasljeđivanje – u deklaraciji interfejsa koji nasljeđuje jedan ili više drugih interfejsa nazivi interfejsa koji se nasljeđuju se specificiraju u zaglavlju interfejsa, nakon ključne riječi extends, i razdvajaju se znakom „ , “
- 189. podinterfejs nasljeđuje sve metode svojih superinterfejsa, jer su metode interfejsa javne (public) 🖱️
- 190. podinterfejs može redefinisati deklaracije apstraktnih metoda nasljeđene od svojih superinterfejsa – pri tome, redefinisane metode nisu nasljeđene 🖱️
- 191. iako interfejsi ne mogu biti instancirani, moguće je deklarirati referencu nekog tipa interfejsa

192. u interfejsu se mogu deklarirati konstante koje su implicitno public, static i final - ove konstante moraju biti inicijalizovane ➡
193. konstantama deklariranim u interfejsu može pristupiti bilo koji klijent (klasa ili interfejs) putem punog kvalifikovanog imena ➡
194. ako klijent (klasa ili interfejs) implementiraju, odnosno nasljeđuju interfejs, onda je konstanti moguće pristupiti navođenjem njenog jednostavnog imena ➡
195. polimorfizam omogućava da reference referenciraju objekte različitih tipova u različito vrijeme tokom izvršavanja
196. poziv privatne metode instance nije i ne može biti polimorfan, jer se takav poziv može javiti samo unutar klase i odnosi se na tačno određenu metodu, što može biti utvrđeno za vrijeme kompajliranja
197. polimorfizam se može ostvariti kako putem nasljeđivanja, tako i putem implementacije interfejsa ➡
198. unutrašnja klasa ima pristup svim varijablama okružujuće klase čak i privatnim samo nema pristup poljima u mainu. ➡
199. Ako se dva puta pozove start za istu nit desice se greska/izuzetak ➡
200. Friendly modifikator pristupa je jaci/rigorozniji od protected
201. Integer extends Number tako da nije dozvoljeno Integer I = (Number)o;
202. Short,int,char,byte,enum mogu u switch
203. Hes tabela ne dozvoljava ni ključevi ni vrijednosti da budu null
204. Kada imamo join metodu moramo baciti izuzetak ili imati try catch blok specificiran ➡
205. U hesh setu je potrebno redefinisati metodu equals ➡
206. Unutrašnja neimenovana klasa može naslediti tačno jednu klasu ili implementirati tačno jedan interfejs ➡
207. Metoda lokalne unutrašnje klase može biti apstraktna
- 208.
209. Klasa StringBuffer ne redefiniše equals i hesh f-ju
210. Apstraktna klasa može imati konstruktor i main ➡
211. Lokalna varijabla nema modifikator pristupa ➡
212. List je interfejs tako da ne možemo imati List<String> l=new List<String>(); ➡
213. Redefinisanje metode run sa public static void run je greska
- Modifikator synchronized može ➡

214. U hes mapi se trebaju redefinisati equals I hes metode
215. Enum kolekcija moze imati konstruktor I varijable 📌
216. Ako pristupamo iz unutrashnje klase lokalnoj promjenljivoj ona mora biti final 📌
217. Unutrashnja klasa se mora prvo definisati pa instancirati
218. Klasa number sadrzi integer
219. Hes set treba da redefinisuje equals
- 220.
221. **Static nested class** nema pristup nestatickim clanovima okruzjuce klase
222. .
- ```
223. class Foo
224. {
225. class Bar{ }
226. }
227. class Test
228. {
229. public static void main (String [] args)
230. {
231. Foo f = new Foo();
232. /* Line 10: Missing statement ? */
233. }
234. }
```
235. Foo.Bar b = f.new Bar();
236. Instancirali smo Bar
237. *private* makes a member accessible only from within its own class
238. **protected** makes a member accessible only to classes in the same package or subclass of the class
239. **default (friendly)** access is very similar to protected (make sure you spot the difference) default access makes a member accessible only to classes in the same package.
240. **public** means that all other classes regardless of the package that they belong to, can access the member (assuming the class itself is visible)

241. *final* makes it impossible to extend a class, when applied to a method it prevents a method from being overridden in a subclass, when applied to a variable it makes it impossible to reinitialise a variable once it has been initialised
242. *abstract* declares a method that has not been implemented.
243. *transient* indicates that a variable is not part of the persistent state of an object.
244. *volatile* indicates that a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

```
245. public class Outer
246. {
247. public void someOuterMethod()
248. {
249. //Line 5
250. }
251. public class Inner { }
252.
253. public static void main(String[] argv)
254. {
255. Outer ot = new Outer();
256. //Line 10
257. }
258. } new Inner(); //At line 5 tacan odgovor
259. Da smo isto ovo napisali u liniji 10 prijavilo bi
 gresku jer ne mozemo pristupiti nestatickoj
 promjenljivoj iz statickog konteksta
260.
```

```
261. public class ReturnIt
262. {
263. returnType methodA(byte x, double y) /* Line 3
 */
264. {
265. return (long)x / y * 2;
```

266.        }

267.   }

268.   Povratni tip je double

From a byte to a short, an int, a long, a float, or a double.

- From a short, an int, a long, a float, or a double.

- From a char to an int, a long, a float, or a double.

- From an int to a long, a float, or a double.

- From a long to a float, or a double.

- From a float to a double.

269.   protected abstract void m1();

270.   static final void m1(){}

271.   synchronized public final void m1() {}

272.   private native void m1();

273.   ispravne deklaracije metoda

```
interface Base2 implements Base {}
```

```
abstract class Class2 extends Base
{ public boolean m1(){ return true; }}
```

```
abstract class Class2 implements Base {}
```

```
abstract class Class2 implements Base
{ public boolean m1(){ return (7 > 4); }}
```

```
abstract class Class2 implements Base
{ protected boolean m1(){ return (5 > 7) }}
tacni odgovori 3 I 4
```

*Koja je legalna deklaracija nestatickih klasa I interfejsa???*

1. `final abstract class Test {}`
2. `public static interface Test {}`
3. `final public class Test {}`
4. `protected abstract class Test {}`
5. `protected interface Test {}`
6. `abstract public class Test {}`



*3 i 6 su tacni odgovori*

☐ `float[ ] f = new float(3);`  
☐ `float f2[ ] = new float[ ];`  
Greske prilikom inicijalizacije nizova

```
try
{
 int x = 0;
 int y = 5 / x;
}
catch (Exception e)
{
 System.out.println("Exception");
}
catch (ArithmeticException ae)
{
 System.out.println(" Arithmetic Exception");
}
System.out.println("finished");
```

Kod nece biti kompajliran jer ce `ArithmeticException` biti nedostizan zato sto je podklasa klase `Exception`

```
public class X
{
 public static void main(String [] args)
 {
 try
 {
 badMethod();
 System.out.print("A");
 }
 catch (Exception ex)
 {
 System.out.print("B");
 }
 finally
 {
 System.out.print("C");
 }
 System.out.print("D");
 }
 public static void badMethod()
 {
 throw new Error(); /* Line 22 */
 }
}
```

Ispisace C i izaci ce na ekranu izuzetak, jer klasa Error je necekirani Exception

Finally blok se uvijek izvrsava

```
public class X
{
 public static void main(String [] args)
 {
 try
 {
 badMethod();
 System.out.print("A");
 }
 catch (Throwable ex)
 {
 System.out.print("B");
 }
 finally
 {
 System.out.print("C");
 }
 System.out.print("D");
 }
 public static void badMethod()
 {
 throw new Error(); /* Line 22 */
 }
}
```

```
}
```

U ovom slucaju ispisace BCD, jer je klasa Throwable  
roditeljska klasa svih izuzetaka

```
public class RTExcept
{
 public static void throwit ()
 {
 System.out.print("throwit ");
 throw new RuntimeException();
 }
 public static void main(String [] args)
 {
 try
 {
 System.out.print("hello ");
 throwit();
 }
 catch (Exception re)
 {
 System.out.print("caught ");
 }
 finally
 {
 System.out.print("finally ");
 }
 System.out.println("after ");
 }
}
```

Ispisace

hello throwit caught finally after



```

public class Test
{
 public static void aMethod() throws Exception
 {
 try /* Line 5 */
 {
 throw new Exception(); /* Line 7 */
 }
 finally /* Line 9 */
 {
 System.out.print("finally "); /* Line 11 */
 }
 }
 public static void main(String args[])
 {
 try
 {
 aMethod();
 }
 catch (Exception e) /* Line 20 */
 {
 System.out.print("exception ");
 }
 System.out.print("finished"); /* Line 24 */
 }
}

```

Ispisace

finally exception finished

Which collection class allows you to associate its elements with key values, and allows you to retrieve objects in FIFO (first-in, first-out) sequence?

[A.java.util.ArrayList](#)

[C.java.util.HashMap](#)

[B.java.util.LinkedHashMap](#)

[D.java.util.TreeMap](#)



Which collection class allows you to access its elements by associating a key with an element's value, and provides synchronization?

[A.java.util.SortedMap](#)

[C.java.util.TreeSet](#)

[B.java.util.TreeMap](#)

[D.java.util.Hashtable](#)



Klase mogu biti zapisane:

```
public final class ImeKlase
```

```
final public class ImeKlase
```

```
abstract public class ImeKlase
```

```
public abstract class ImeKlase
```

za metode interfejsa :

```
public abstract void metoda();
```

```
abstract public void metoda();
```

```
public void metoda();
```

```
abstract void metoda();
```

```
void metoda();
```

za polja interfejsa:

```
public static final int=5;
```

validan je bilo koji raspored prva tri, a ne moramo nist napisat od njih podrazumijeva se samo polje mora biti inicijalizovano

metoda instance moze biti redefinisana u klasi  
nasljednici u apstraktnu metodu vase isti uslovi kao za  
redefinisanje u obicnu metodu instance

```
class Game {
 static String s = "-";
 String s2 = "s2";
 Game(String arg) { s += arg; }
}
public class Go extends Game {
 Go() { super(s2); } //greska ne moze
//pozvati sa parametrom s2 nadklase Game prvo se mora
//ona konstruisati
 { s += "i "; }
 public static void main(String[] args) {
 new Go();
 System.out.println(s);
 }
 static { s += "sb "; }
}
```

```
public class Internet {
 private int y = 8;
 public static void main(String[] args) {
 new Internet().go();
 }
 void go() {
 int x = 7;
```

TCPIP ip = new TCPIP(); //prvo deklarisanje pa  
instanciranje obratno greska

```
class TCPIP {
 void doit() { System.out.println(y + x); }
 //pristupili smo x iz unutranje klase moramo je
 //proglasiti final
 }
 ip.doit();
 }
 }
```

```
import java.util.*;
public class Piles {
 public static void main(String[] args) {

 TreeMap<String, String> tm = new TreeMap<String,
String>();
 //sortira po kljucevima u rastucem nizu prioritet
 //brojevi

 TreeSet<String> ts = new TreeSet<String>();
```

```
//sortira u prirodnom poretku
//prvo brojevi pa slova u rastucem tj prirodnom poretku
String[] k = {"1", "b", "4", "3", "2", "d"};
String[] v = {"a", "d", "3", "b", "2", "e"};
for(int i=0; i<6; i++) {
 tm.put(k[i], v[i]);
 ts.add(v[i]);
}
System.out.print(tm.values() + " ");
Iterator it2 = ts.iterator();
while(it2.hasNext()) System.out.print(it2.next() + "-");
} }
```

```
import java.util.*;
class Radio {
String getFreq() { return "97.3"; }
static String getF() { return "97.3"; }
}
public class Ham extends Radio {
String getFreq() { return "50.1"; }
static String getF() { return "50.1"; }
public static void main(String[] args) {
List<Radio> radios = new ArrayList<Radio>();
radios.add(new Radio());
radios.add(new Ham());
for(Radio r: radios)
System.out.print(r.getFreq() + " " + r.getF() + " ");
} }
//97.3 97.3 50.1 97.3
```

```
class Bonds {
Bonds force() { return new Bonds(); }
}
public class Covalent extends Bonds {
Covalent force() { return new Covalent(); }
public static void main(String[] args) {
new Covalent().go(new Covalent());
}
void go(Covalent c) {
```

```

go2(new Bonds().force(), c.force());
}
void go2(Bonds b, Covalent c) {
 Covalent c2 =(Covalent)b; //greska ne moze se
 kastovati Bonds u Covalent /prolazi kompajliranje
 Bonds b2 = (Bonds)c;
} }

import java.util.*;
public class Bucket {
 public static void main(String[] args) {
 Set<String> hs = new HashSet<String>();
 Set<String> lh = new LinkedHashSet<String>();
 Set<String> ts = new TreeSet<String>();
 List<String> al = new ArrayList<String>();
 String[] v = {"1", "3", "1","4","2","b","a","c"};
 for(int i=0; i< v.length; i++) {
 hs.add(v[i]); lh.add(v[i]); ts.add(v[i]);
 al.add(v[i]);
 }
 Iterator it = hs.iterator();
 while(it.hasNext())System.out.print(it.next() + " ");
 //hes set nema redosleda ubacivanja elemenata 3 2 1 b c
 //a 4

 System.out.println("hes set");
 Iterator it2 = lh.iterator();
 while(it2.hasNext())System.out.print(it2.next()+" ");

 // linked hes set ispisuje kako unosimo bez duplih
 //elemenata 1 3 4 2 b a c

 System.out.println("linked hes set");
 Iterator it3 = ts.iterator();
 while(it3.hasNext())System.out.print(it3.next()+ " ");

 // treeset sortira u prirodnom poretku prvo brojevi pa
 //slova 1 2 3 4 a b c

 System.out.println("tree set");
 }
}

```

```

Iterator it5 = al.iterator();
while(it5.hasNext())System.out.print(it5.next()+ " ");

// arraylist poredak kako smo unosili elemente podrzava
//duplikate 1 3 1 4 2 b a c

 System.out.println("array");
} }

```

```

class Bird {
public static String s = "";
 public static void fly() { s += "fly "; }
}
 public class Hummingbird extends Bird {
 public static void fly() { s += "hover "; }
 public static void main(String[] args) {
 Bird b1 = new Bird();
 Bird b2 = new Hummingbird();
 Bird b3 = (Hummingbird)b2;
 Hummingbird b4 = (Hummingbird)b2;

 b1.fly(); b2.fly(); b3.fly(); b4.fly();
 System.out.println(s);
 } }
//fly fly fly hover

```

```

class One {
void go1() { System.out.print("1 "); }
 final void go2() { System.out.print("2 "); }
 private void go3() { System.out.print("3 "); }
}
 public class OneB extends One {
 void go1() { System.out.print("1b "); }
 void go3() { System.out.print("3b "); }

 public static void main(String[] args) {
 new OneB().go1();

```

```
new One().go1();
new OneB().go2();
new OneB().go3();
new One().go3(); //greska metoda ima privatni pristup
samo joj se moze pristupiti iz klase
} }
```

```
public class Simulacija{

 public static void main(String args[]){

int y, count = 0;
 for(int x = 3; x < 6; x++) {
 try {
 switch(x) {
 case 3: count++;

 case 4: count++;
```



```

 case 7: count++;

 case 9: { y = 7 / (x - 4); count += 10; } //kada x
bude 4 NullPointerException i samo se u catch bloku
uveca count i kraj

 }
 } catch (Exception ex) { count++;}
 }
 System.out.println(count);
 }
}

//16

```

```

public class Simulacija2{
 public static void main(String args[]){
int count = 0;
 outer:
 for(int x = 0; x < 5; x++) {
 middle:
 for(int y = 0; y < 5; y++) {
 System.out.println("u middle"+" "+x+" "+y+"
"+count);
 if(y == 1) continue middle; //nastavlja izvorsavanje
sledece iteracije for petlje u middle
 }
 }
}

```

```

 System.out.println("iza continue u middle"+" "+x+"
"+y+" "+count);
 if(y == 3) break middle; //iskace iz for petlje i
labele middle i nastavlja u for petlju outera
 count++;
 System.out.println("iza break u middle"+" "+x+"
"+y+" "+count);
 }
 if(x > 2) continue outer; //kad x bude vece od 2 nece
izlaziti iz petlje for
 count = count + 10;
 System.out.println("u outeru iza middla"+" "+x+"
"+count);
}
System.out.println("count: " + count);
}
}
/*
u middle 0 0 0
iza continue u middle 0 0 0
iza break u middle 0 0 1
u middle 0 1 1
u middle 0 2 1
iza continue u middle 0 2 1
iza break u middle 0 2 2
u middle 0 3 2
iza continue u middle 0 3 2
u outeru iza middla 0 12
u middle 1 0 12
iza continue u middle 1 0 12
iza break u middle 1 0 13
u middle 1 1 13
u middle 1 2 13
iza continue u middle 1 2 13
iza break u middle 1 2 14
u middle 1 3 14
iza continue u middle 1 3 14
u outeru iza middla 1 24
u middle 2 0 24
iza continue u middle 2 0 24

```

```

iza break u middle 2 0 25
u middle 2 1 25
u middle 2 2 25
iza continue u middle 2 2 25
iza break u middle 2 2 26
u middle 2 3 26
iza continue u middle 2 3 26
u outeru iza middla 2 36
u middle 3 0 36
iza continue u middle 3 0 36
iza break u middle 3 0 37
u middle 3 1 37
u middle 3 2 37
iza continue u middle 3 2 37
iza break u middle 3 2 38
u middle 3 3 38
iza continue u middle 3 3 38
u middle 4 0 38
iza continue u middle 4 0 38
iza break u middle 4 0 39
u middle 4 1 39
u middle 4 2 39
iza continue u middle 4 2 39
iza break u middle 4 2 40
u middle 4 3 40
iza continue u middle 4 3 40
count: 40
**/

```

```

import java.util.*;
public class Elway {
public static void main(String[] args) {
ArrayList[] ls = new ArrayList[3];
for(int i = 0; i < 3; i++) {
ls[i] = new ArrayList();
ls[i].add("a" + i);
}
Object o = ls;
do3(ls);

```

```

for(int i = 0; i < 3; i++) {
 // insert code here

 //System.out.print(o[i] + " "); //greska trazi se
 array a pronadjen je objekt
 //System.out.print((ArrayList[])[i] + " "); //greska
 System.out.print(((Object[])o)[i] + " "); //prolazi
 System.out.print(((ArrayList[])o)[i] + " "); //prolazi
 isti ispis imaju
}
}
static Object do3(ArrayList[] a) {
 for(int i = 0; i < 3; i++) a[i].add("e");
 return a;
} }
// [a0,e] [a1,e] [a2,e]

```

```

class Wanderer implements Runnable {
 public void run() {
 for(int i = 0; i < 2; i++)
 System.out.print(Thread.currentThread().getName() + "
");
 }
}

public class Wander {
 public static void main(String[] args) {
 Wanderer w = new Wanderer();
 Thread t1 = new Thread();
 }
}

```

```
Thread t2 = new Thread(w);
Thread t3 = new Thread(w, "fred");
 t1.start(); t2.start(); t3.start(); //prvi i drugi
nist ne ispisuju
 } }
```

```
//thread 1 fred thread 1 fred
```

```
class Pancake { }
class BlueberryPancake extends Pancake { }
public class SourdoughBlueberryPancake2 extends
BlueberryPancake {
 public static void main(String[] args) {

 Pancake p4 = new SourdoughBlueberryPancake2();

 // insert code here

 ////Pancake p5 = p4; //prolazi
```

```
//Pancake p6 = (BlueberryPancake)p4; //prolazi

// BlueberryPancake b2 = (BlueberryPancake)p4;
//prolazi

// BlueberryPancake b3 =
(SourdoughBlueberryPancake2)p4; //prolazi

//SourdoughBlueberryPancake2 s1 = (BlueberryPancake)p4;
//greska nekompatibilni tipovi

//SourdoughBlueberryPancake2 s2 =
(SourdoughBlueberryPancake2)p4; //prolazi
```

```
public class Simulacija3{

 public static void main(String args[]){
 go();
 }

 static void go() {
 int cows = 0;
 int[] twisters = {1,2,3};
 for(int i = 0; i < 4; i++)
 switch(twisters[i]) {
```

```

 case 2: cows++;
 case 1: cows += 10;
 case 0: go();
 }
 System.out.println(cows);
}

}

//StackOverflowError

public class Simulacija2{

 public static void main (String args[]){

 int x = 3;
 for(int i = 0; i < 3; i++) {
 if(i == 1) x = x++;
 if(i % 2 == 0 && x % 2 == 0) System.out.print(".");
 if(i % 2 == 0 && x % 2 == 1) System.out.print("-");
 if(i == 2 ^ x == 4) System.out.print(","); // x se
 nece povecat sve dok i ne bude 2 tj dok se ne bude
 koristio operator poredjenja
 }
 System.out.println("<");
 }
}
// --,<

public class Self extends Thread {
 public static void main(String[] args) {
 try {
 Thread t = new Thread(new Self());
 t.start();
 t.start();
 } catch (Exception e) { System.out.print("e "); }
 //desice se izuzetak zbog dva puta startanja istog
 treda
 }
 public void run() {

```

```
for(int i = 0; i < 2; i++)
System.out.print(Thread.currentThread().getName() + "
");
} }
```

```
//e tred-4 tred-4
```

```
class RainCatcher {
 static StringBuffer s;
 public static void main(String[] args) {
 Integer i = new Integer(42);
 for(int j = 40; j < i; i--)
 switch(i) {
 case 41: s.append("41 ");
 default: s.append("def ");
 case 42: s.append("42 "); //posle ove linije puca kod
 }
 System.out.println(s);
 } }
```

```
//nije napravljen objekat StringBuffer greska
```

```
//da je napravljen izlaz bi bio 42 41 def 42
```

```
import java.io.*;
class ElectronicDevice { ElectronicDevice() {
System.out.print("ed "); }}
class Mp3player extends ElectronicDevice implements
Serializable {
 Mp3player() { System.out.print("mp "); }
}
public class MiniPlayer extends Mp3player {
 MiniPlayer() { System.out.print("mini "); }
 public static void main(String[] args) {
 MiniPlayer m = new MiniPlayer();
```



```

try {
 FileOutputStream fos = new
FileOutputStream("dev.txt");
 ObjectOutputStream os = new ObjectOutputStream(fos);
 os.writeObject(m); os.close();
 FileInputStream fis = new FileInputStream("dev.txt");
 ObjectInputStream is = new ObjectInputStream(fis);
 MiniPlayer m2 = (MiniPlayer) is.readObject();
 is.close();
} catch (Exception x) { System.out.print("x "); }
} }
// ed mp mini ed

```

//izvrsice se samo konstruktor one klase koja ne  
implementira serijalizabile

```

public class Limits {
 private int x = 2;
 protected int y = 3;
 private static int m1 = 4;
 protected static int m2 = 5;
 public static void main(String[] args) {
int x = 6; int y = 7;
 int m1 = 8; int m2 = 9;
 new Limits().new Secret().go();
 }
 class Secret {

```

```
void go() { System.out.println(x + " " + y + " " + m1
+ " " + m2); }
} }
```

```
//2 3 4 5
```

```
public static void main(String[] args) {
 List<Integer> x = new ArrayList<Integer>();
 x.add(new Integer(3));
 doStuff(x);
 for(Integer i: x)
 System.out.print(i + " ");
 }
 static void doStuff(List y) {
 y.add(new Integer(4));
 y.add(new Float(3.14f));
 }
 //ispisace 3 4 i bacice izuzetak, jer u vrijeme
 kompajliranja prolazi, ali metoda doStuff ne zna da je y
 genericka
```

```
//ClassCastException
```

```
class Grandfather {
 String name = "gf "; //bili staticki ili ne isto je
 String doStuff() { return "grandf "; }
}
class Father extends Grandfather {
 String name = "fa ";
 String doStuff() { return "father "; }
}
public class Child extends Father {
 String name = "ch ";
 String doStuff() { return "child "; }
```

```

public static void main(String[] args) {

 Father f = new Father();
 System.out.print(((Grandfather)f).name
+ ((Grandfather)f).doStuff()) ;
 Child c = new Child();
 System.out.println(((Grandfather)c).name
+ ((Grandfather)c).doStuff() + ((Father)c).doStuff());
} }
//uvjek se poziva ime roditelja jer nije maskirano, ali
metode su redefinisane tako da poziva iz svoje klase
//gf father gf child child

```

```

public class BackHanded {
 int state = 0;
 BackHanded(int s) { state = s; }
 public static void main(String... hi) {
 BackHanded b1 = new BackHanded(1);
 BackHanded b2 = new BackHanded(2);
 System.out.println(b1.go(b1) + " " + b2.go(b2));
 }
}

```

```
int go(BackHanded b) {
 if(this.state == 2) {
 b.state = 5;
 go(this); //sto znaci izvrsite se metoda do kraja i jos jedan njen poziv
 }
 return ++this.state;
}}
```

```
package com.wickedlysmart;
import com.wickedlysmart2.*;
//da je import glasio import com.wickedlysmart2.Utills; klasa bi bila vidljiva
public class Launcher {
 public static void main(String[] args) {
 Utills u = new Utills();
 u.do1();
 u.do2();
 u.do3();
 }
}
```

```
}}
package com.wickedlysmart2;
class Utils {
void do1() { System.out.print("do1 "); }
protected void do2() { System.out.print("do2 "); }
public void do3() { System.out.print("do3 "); }
}
```

//klasa Utils je sa podrazumijevanim privilegijama sto znaci nije vidljiva  
//izvan svog paketa jer nije public