

DODATNO POGLEDATI - 79, 80, 81, 87, 226, 232, prezentacija-ugnjezdjeni(8, 9)

- ako se stek može dinamički proširivati a nema dovoljno memorije za to ili nema dovoljno memorije pri kreiranju niti – `OutOfMemoryError`
- ako neko izračunavanje zahtjeva veći heap nego što je to moguće – `OutOfMemoryError`
- class loader povezuje prevedene datoteke
- identifikatori se sastoje od slova, brojeva, donje crte, broj ne može biti na početku
- sve ključne riječi se pišu malim slovima i njihovo nepravilno korištenje rezultuje greskama pri kompajliranju
- literali u oktalnom obliku(0) i u heksadecimalnom obliku(0X) kao prefix, binarni literali ne postoje
- karakter literal pod jednostrukim navodnicima
- primitivni tipovi se čuvaju na steku
- svaki primitivni tip ima odgovarajuću okružujuću klasu(wrapper)
- reference na steku, dok su sami objekti na heap-u
- dvije vrste inicijalizacionih blokova: statički i nestatički
- statički inicijalizacioni blok se izvršava JEDANPUT, prilikom učitavanja klase. Ovaj blok je označen ključnom riječi `static` prije separatora `{`, a u njemu se mogu inicijalizovati samo statički atributi klase
- nestatički inicijalizacioni blok se izvršava za svaki objekat koji se kreira i inicijalizuje nestatičke attribute klase(moguće je inicijalizovati i statičke attribute ali se ne preporučuje)
- lokalne promjenjive se NE inicijalizuju podrazumjevanom vrijednošću prilikom prilikom kreiranja kod poziva metode tj. kada započne izvršavanje metode, prema tome kompajler će prijaviti kao grešku svaki slučaj korištenja neinicijalizovane promjenjive
- za lokalne reference važi isto pravilo kao i za lokalne promjenjive(ako je inicijalizujemo sa null može se desiti izuzetak `NullPointerException`, tako da je najbolje da svaka referenca pokazuje na konkretan objekat)
- promjenjive instance su dostupne sve dok se objekat kojem pripadaju koristi, Statičke promjenjive su dostupne sve dok je klasa kojoj pripadaju dostupna, Lokalne promjenjive se kreiraju pri svakom izvršavanju metode, konstruktora ili bloka i nakon okončavanja više nisu dostupne
- pri pohranjivanju većeg u manji primitivni tip kompajler javlja grešku
- `byte` -> `short` -> `int` -> `long` -> `float` -> `double`
- ako je neophodno izvršiti konverziju s tipa u koji to se mora eksplicitno uraditi (`int i = (int)5.5`) i desava se gubljenje informacija
- u slučaju da pokušaj kastovanja nije legalan, desava se greška pri kompajliranju
- konverzija proširivanja reference nikad neće baciti izuzetak, dok kod konverzije sužavanja reference može doći do bacanja `ClassCastException` izuzetka

- Boxing je konverzija primitivnog tipa u okružujući tip, a unboxing je konverzija iz okružujućeg tipa u primitivni tip
- pokušaj boxing konverzije u neodgovarajući okružujući tip rezultuje greskom pri kompajliranju, a unboxing reference okružujućeg tipa koja ima null vrijednost rezultuje `NullPointerException`
- rezultat aritmetičkog izraza će stalno biti veći tip, postoji jedan izuzetak kada se sabiraju byte i short vrijednosti rezultat je tipa int jer nikada rezultat ne može biti tipa byte ili short
- naziv metode i lista parametara metoda čine potpis metode, pri čemu je redoslijed parametara u listi bitan
- bitno je napomenuti da se nazivi klase i nazivi metoda nalaze u različitim prostorima imena. Iz tog razloga, ne postoji konflikt naziva između naziva klase i naziva metode, prema tome, moguće je deklarirati metodu koja ima naziv identičan nazivu konstruktora (i nazivu klase)
- podrazumijevani konstruktor postavlja vrijednosti atributa na podrazumijevane vrijednosti
- ako u klasi postoji samo konstruktor koji nije podrazumijevani, onda je obavezno programera da implementira i podrazumijevani konstruktor, ako je on i potreban. u slučaju da to nije urađeno svaki pokušaj korištenja podrazumijevanog konstruktora dovodi do greške pri kompajliranju
- greška pri kompajliranju ako se poziva neka metoda koja ne postoji u klasi ili se pristupa privatnim atributima
- statičkim članovima se pristupa preko imena klase
- reference na nizove se čuvaju na steku, a elementi se čuvaju na heap-u
- `int[]` niz ili `int niz[]`
- kreiranjem niza alokirano je odgovarajuće memorijsko prostora i inicijalizovan podrazumijevanim vrijednostima
- deklaracija paketa (`package ...`) onda ide `import`, i na kraju deklaracija tipova (klase, interfejsi, enumeracija)
- uvozom nekog paketa se ne uvoze i njegovi podpaketi
- kada se uvezu dva tipa sa istim imenom jedino rješenje je korištenje njihovog punog kvalifikovanog imena
- `{ kalkulator k = new Kalkulator(); }` nakon izlaska iz datog bloka referenca k više neće biti dostupna, ali objekat klase Kalkulator će još zauzimati prostora na heap-u sve dok se ne ukloni od strane garbage collector-a
- ako ne navedemo ni jedan modifikator pristupa podrazumijeva se `package`, i ta klasa ili atribut je vidljiva u okviru svog paketa
- nije moguće instancirati objekat apstraktno klase ali je moguće imati reference na apstraktnu klasu pri čemu ta referenca referencira objekat klase nasljednice
- enum tipovi ne mogu biti apstraktni
- enum tipovi su implicitno `final` i ne mogu biti definisani kao `final` pomoću ključne riječi `final`

- modifikator final u kontekstu klasa znaci da klasa ne moze biti nasljedjena, a u kontekstu atributa final znaci da vrijednost tog atributa ne moze biti promjenjena nakon inicijalizacije
- promjenjive koje su definisane kao static I final najcesce se koriste za definisanje konstanti
- final promjenjive ne moraju biti inicijalizovane pri deklaraciji ali moraju biti inicijalizovane prije nego sto budu koristene
- abstraktne metode mogu samo biti metode instance I to povlaci da I ta klasa mora biti abstraktna, a staticke metode ne mogu biti abstract
- ako metoda zavrшава bacanjem izuzetka postanje return naredbe nije neophodno
- svi izuzeci osim RUNTIMEEXCEPTION, ERROR I njihovih klasa nasljednica nazivaju se provjereni izuzeci. provjereni izuzetak se mora obraditi
- neprovjereni izuzeci su ERROR I RUNTIMEEXCEPTION I sve njihove klase nasljednice I oni ne zahtjevaju provjeru, kada se desi ERROR izuzetak, oporavak uglavnom nije moguc, a RUNTIMEEXCEPTION se desava kao rezultatprogramerskih gresaka
- CLASSNOTFOUNDEXCEPTION bice bacen kada aplikacija pokusa da ucita klasu na bazi datog naziva, CLONENOTSUPPORTEDEXCEPTION bice bacen kada se klonira objekat a ta klasa ne implementira interface Cloneable, INSTANTIONEXCEPTION bice bacen pri pokusaju nepravilnog kreiranja objekta(npr. instanciranje abstraktne klase ili intefejsa), INTERRUPTEDEXCEPTION bice bacen kada jedna nit prekina drugu nit, NOSUCHFIELDEXCEPTION bice bacen kada neka klasa ne posjeduje zahtjevano polje, a NOSUCHMETHODEXCEPTION bice bacen kada neka klasa ne posjeduje zahtijevanu metodu, PARSEEXCEPTION bice bacen kada se desi greska pri parsiranju
- finally blok se mora stalno izvršiti
- ako imamo vise catch blokova I ako je prvi blok na koji ce izuzetak naici u hijerarhiji izndad tipa argumenta nekog od narednih catch blokova tada ce se desiti greska pri kompajliranju jer izuzetak nikada nece moci da doce u nize catch blokove
- vrijednost vracena return naredbom u finally bloku je uvijek primarna u odnosu na vrijednost vracenu return naredbom u try bloku
- ako neka metoda samo prosljedjuje izuzetak dalje na tom put ce biti izvršeni svi finally blokovi
- ako se ne nadje ni jedan catch blok koji moze da obradi baceni izuzetak program staje sa izvršavanjem
- kada neka klasa naslijedi neku drugu klasu I redefinisuje neku metodu, ona moze da: baci manje izuzetaka ili izostavi throws klauzulu, baci iste izuzetke, baci izuzetke koji su podklase onih koji su baceni u metodi koja je redefinisana
- kalkulator k = new ProsireniKalkulator(1,2); ako se nad k pozove neka metoda iz prosirenog kalkulatora to ce dovesti do greske pri kompajliranju jer kompajler nema saznanja u vrijeme kompajliranja
- ako hocemo da redefinisemo neku metodu u podklasi onda ta metoda mora da ima isti potpis, povratni tip moze da bude podtip povratnog tipa redefinisane metode, nova metoda ne moze smanjiti dostupnost metode, ali je moze povecati, nova metoda moze baciti sve, nijedan ili podskup provjerenih izuzetaka iz redefinisane metode

- ako redefinisemo neku metodu I bacimo neki Exception koji nije definisan u osnovnoj klasi desice se greska pri kompajliranju
- metoda instance u klasi nasljednici ne moze redefinisati staticku metodu osnovne klase, greska pri kompajliranju
- staticka metoda u klasi nasljednici moze sakriti staticku metodu osnovne klase
- metode gdje se nalazi modifikator final ne mogu biti redefinisane, dovodi do greske pri kompajliranju
- ako metoda u nasljednici ima isti potpis kao I u osnovnoj klasi, to znaci da je redefinisana
- ako metoda u nasljednici ima isti naziv ali razlicitu listu parametara, ona se razlikuje po tipu, redoslijedu ili broju parametara onda to znaci da je ta metoda preklopljena
- iako metode instance u klasi nasljednici ne mogu redefinisati staticku metodu, kada su u pitanju polja, polje instance u klasi moze maskirati staticko polje osnovne klase
- this je samo dostupna u nestatickom kontekstu
- ako koristimo poziv konstruktora koristenjem konstrukcije this() ili super() to mora biti prva naredba u tijelu konstruktora
- this() I super() se ne mogu naci u istoj klasi
- ako ne postoji super() konstruktor to podrazumjevano ubaci kao prvu liniju u konstruktoru, ako u osnovnoj klasi nije definisan nijedan konstruktor nece biti problema jer ce biti podrazumjevano deinisani podrazumjevani konstruktor, ali ako ima neki konstruktor sa parametrima a nema podrazumjevani dolazi do greske pri kompajliranju
- metode interfejsa moraju imati public dostupnost I klasa ne smije smanjiti njihovu dostupnost
- metode interfejsa moraju stalno da budu implementirane kao metode instance, a ne kao staticke metode
- klasa posjeduje samo jednu implementaciju metode koja moze biti deklarirana u vise interfejsa
- kod interfejsa postoji visestruko nasljedjivanje
- u interfejsima se mogu deklarirati konstante koje su implicitno public, static I final, I te konstante moraju biti inicijalizovane
- operator instanceof vraca false ako je s lijeve strane null, a ako ne postoji veza izmedju ova dva tipa desava se greska pri kompajliranju
- polimorfizam se moze ostvariti nasljedjivanjem i implementacijom
- metoda clone – protected Object clone() throws CloneNotSupportedException ovu metodu moze da redefinisuje svaka klasa I dobra praksa je da se stavlja na public....ako se clone metoda oslanja na clone metodu Object ta klasa mora da impleментira interface Cloneable ili ce baciti CLONENOTSUPPORTEDEXCEPTION
- metoda equals – public Boolean equals(Object obj) vraca istinitu vrijednost ako I samo ako dvije reference koje se porede referenciraju isti objekat
- metoda hashCode – public int hashCode() ova metoda vraca memorijsku adresu objekta kao podrazumjevanu hash vrijednost objekta, ako se redefinisuje equals metoda, potrebno je redefinisati I hashCode metodu da bi equals raditi kako treba

- metoda `toString` – `public String toString()` vraća tekstualnu reprezentaciju objekta, ako se ne redefiniše on će vratiti `NazivKlase@hashVrijednostObjekta`
- metoda `getClass` – `public final Class<?> getClass()` vraća klasu objekta koja je predstavljena objektom klase `java.lang.Class`
- metoda `finalize` – `protected void finalize() throws Throwable` poziva se nad objektom od strane garbage collector-a kada on otkrije da ne postoji niti jedna referenca na taj objekat
- sve okružujuće klase imaju dva konstruktora, jedan za primitivne tipove, a drugi za `String` i taj sa `Stringom` može baciti `NumberFormatException`, isti slučaj je i sa `valueOf()` metodom koja je statička u okružujućim klasama
- okružujuće klase `Integer` i `Long` imaju statičke metode za konverziju u odgovarajuću `String` reprezentaciju (binarna, oktalna, heksadecimalna) `toBinary/Octal/HexString(int/long)`
- `String` objekti su nepromjenjivi, to znači da se objekat klase `String` ne može promijeniti. Sve metode klase `String` vraćaju novi objekat, dok originalni objekat uvijek ostaje netaknut
- kompajler optimizuje rukovanje string literalima. Svi string literali sa identičnom sekvencom karaktera dijele samo jedan objekat klase `String`
- objekti kreirani pomoću konstruktora klase `String` su uvijek novi objekti, nezavisno od toga koju sekvencu karaktera sadrže
- klase `StringBuilder` i `StringBuffer` implementiraju promjenjivu sekvencu karaktera, pored mogućnosti promjene karaktera može se promijeniti i kapacitet
- `StringBuilder` i `StringBuffer` nisu u nikakvoj vezi sa klasom `String` i ne mogu se čak ni eksplicitno kastovati
- `String` i `StringBuffer` su sinhronizovane klase, a `StringBuilder` nije, to je jedina razlika između `StringBuffer` i `StringBuilder` klase, `StringBuilder` je brži
- `StringBuilder` i `StringBuffer` klase ne redefinišu `equals()` i `compareTo()` metode i zbog ovoga se ne mogu porediti...jedini način da se uporede su da se pretvore u `String` i onda da se uporede
- `StringBuilder` i `StringBuffer` imaju metode `insert()`, `append()`, `delete()` sa svim mogućim kombinacijama
- `StringBuilder` i `StringBuffer` redefinišu metodu `toString()` i tako se najlakše može konvertovati u `String`
- statičke promjenjive i tranzijentne promjenjive neće biti upisane u tok pri procesu serijalizacije
- ako neki objekat sadrži reference na neki drugi objekat, da bi se izvršila serijalizacija i taj drugi objekat mora biti serijalizabilan ili će se desiti `NOTSERIALIZABLEEXCEPTION`
- objekti klase koje nasljeđuju serijalizabilnu klasu uvijek su serijalizabilni
- postoje i alternativni načini za serijalizaciju objekata, npr. da implementiramo dvije metode `writeObject()` i `readObject()` – `private void writeObject(ObjectOutputStream oos) throws IOException` i `private void readObject(ObjectInputStream ois) throws Exception`

- drugi alternativni način je korištenjem Externalizable interface i implementacijom dvije metode writeExternal() i readExternal() – public void writeExternal(ObjectOutput out) throws IOException i public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException, u ovom slučaju je obavezno postojanje podrazumjevanog konstruktora u klasi čiji se objekat serijalizuje, u slučaju da ovaj konstruktor ne postoji desava se INVALIDCLASSEXCEPTION
- kod generika se prevodi nezavisno, postoji samo jedna klasa, bez obzira na broj poziva generika, `Stek<Disk> stekDiskova` i `Stek<String> stekStringova`, kada bi napisali `stekDiskova.getClass() == stekStringova.getClass()` dobija se true, što nam pokazuje da su ovo sve iste klase
- sirovi tip za klasu `Stek<String>` je `Stek`, dozvoljeno je napisati `Stek stekStringova = new Stek()` i ovo predstavlja sirovi tip, ovo je omogućeno zbog kompatibilnosti sa starom Javom, jer tamo nije bilo generickih tipova...kada se koriste sirovi tipovi objekat `Stek` vraća objekat tipa `Object`
- formalni parameter generika se ne može koristiti kao tip statickog polja genericke klase, u statickim metodama (povratni tip, tip parametra metode, tip lokalne promjenjive), u statickim inicijalizacionim blokovima date klase, a razlog za ovo je zato što prevodilac prevodi sve ovo u jednu klasu (sirovi tip)
- statickim metodama u generickoj klasi se ne može pristupiti sa `Stek<String>.m()`, mora ici sa `Stek.m()`;
- u generickoj klasi `A<T>` nije dozvoljeno kreirati objekat tipa `T` ili niz takvih objekata
- argumenti generika mogu biti klase, interfejsi i nizovi (čak i nizovi primitivnih tipova), ali ne mogu biti samo primitivni tipovi
- zbog jednostavnijeg pisanja moguće je i napisati `Stek<String> stekStringova = new Stek<>()` i ovo će preuzeti tip koji je sa lijeve strane znaka =
- ako je `D` podtip od `B`, to ne znači da je `G<D>` podtip od `G<B>`
- genericka klasa ne može biti podklasa `Throwable` klase
- i ugnjezdena klasa može biti genericka
- parameter generika se može ograniciti sa gornje strane `class G<T extends B & I1 & I2>`, ovo znači da `T` mora da naslijedi `B`, implementira `I1` i `I2`, a ako se ništa ne navede ograničenje je klasa `Object`
- u generickim tipovima, promjenljivoj `T` se ne može pristupiti iz statickog konteksta, ona se isto ponasa kao bilo koja druga promjenjiva
- kada se zamijene tipske promjenjive sa stvarnim tipovima nastaju različiti parametrizovani tipovi koji nemaju veze jedan sa drugim tj. ne postoji nikakva veza između njih
- genericki tip koji nije označen kao final može biti naslijeđen
- `GenericSub<T> extends GenericBase<T>` ovo nam obezbjeđuje da će tipske promjenjive biti zamijenjene istim stvarnim tipom u obe klase
- `GenericBase<Integer> sub2 = new GenericSub<String>()` ovo će dovesti do greske pri kompajliranju jer ne postoji veza između supertip-podtip
- moguće je da genericki tip naslijedi negenericki, također je moguće i da konkretan tip naslijedi parametrizovani tip - `IntegerHolder extends GenericHolder<Integer>`

- JVM ne zna za genericke tipove I kompajler kada prevodi genericku klasu on informacije o tipskim promjenjivim brise I ubacuje kostovanje u konkretne promjenjive
- kompajler ce prijaviti unchecked upozorenje ako koristenje genericke klase bez parametrizacije moze rezultirati potencijalnim problemom za vrijeme izrsavanja npr. `GenericHolder holder = new GenericHolder<Integer>()`
- problem je sto `G<D>` nije podtip tipa `G<T>`, gdje je `D` podtip tipa `T` I zato se uvode dzoker znakovi, kada bi gledali nizove `D[]` je podtip od `T[]`
- ako stavimo `G<?>` kao formalni parametar neke metode, onda se toj metodi moze poslati bilo koji tip, npr. `G<String>`, `G<Integer>`, `G<Object>`
- postoje dva tipa ogranicenja sa gornje I sa donje strane, sa gornje strane se moze ograniciti sa `G<? extends E>` to znaci da je `E` najvisi tip u hijerarhiji sa kojim postoji kompatibilnost, sa donje strane se moze ograniciti sa `G<? super E>` to znaci da je `E` najnizi tip u hijerarhiji sa kojim postoji kopatibilnost...nije moguće istovremeno dzoker ograniciti sa donjom I gornjom granicom, takodje se ne moze granica sastojati od vise tipova
- mozemo da koristimo I dzoker znak ? koja oznacava bilo koji tip, tako je `GenericHolder<?>` nastala od genericke klase `GenericHolder` kojem je kao parameter stvarnog tipa prosljedjen dzoker znak ?
- `GenericHolder<? super Integer>` oznacava familiju tipova koji nastaju parametrizacijom generickog tipa `GenericHolder<T>` gdje je tipska promjenjiva `T` mijenja stvarnim tipom `Integer` ili nekom supertipom `Integer`-a
- `GenericHolder<? extends Number>` oznacava familiju tipova koji nastaju parametrizacijom generickog tipa `GenericHolder<T>` gdje je tipska promjenjiva `T` mijenja stvarnim tipom `Number` ili nekim podtipom `Number`-a
- Dzoker znakovi se ne mogu koristiti u izrazima za kreiranje instanci, npr. `GenericBase<? extends Long> gb1 = new GenericBase<?>` ce dovesti do greske pri kompajliranju a ovo se desava jer ? nije konkretan tip, ista ova situacija se desava I pri pokusaju instanciranja interfejsa
- ova situacija je dozvoljena - `GenericBase<? extends Long> gb1 = new GenericBase<Long>`
- `List test = new ArrayList<String>()` I ovo je raw tip, nema veze sto je inicijalizovan sa desne strane kada je referenca raw tip
- ako je raw tip proce kompajliranje ali ce se desiti `CLASSCASTEXCEPTION` ako je konkrentni objekat koji se kastuje drugog tipa nego u taj koji se kastuje, a ako je parametrizovan tip onda ce se desiti greska pri kompajliranju ako se pokusa dodati neki drugi tip u datu kolekciju
- ako klasa niti nasljedjuje neku drugu klasu onda je nemoguće da ta klasa naslijedi I klasu `Thread` tada je potrebno da ta klasa implementira interface `Runnable`
- `BLOCKED` – ceka da dobije monitor zeljenog objekta, `WAITING` – ceka na notify ili join, `TIMED_WAITING` - nit spava, ceka na notify ili join, `WAITING` – nit moze u ovom stanju da bude neograniceni vremenski period
- prioritet niti moze se moze definisati `MAX_PRIORITY`, `NORM_PRIORITY`, `MIN_PRIORITY`
- ako nit spava, I ako se desi interrupt desava se `InterruptedException`

- nit prelazi u stanje terminacije kada se okonca metoda run ili kada se baci neki izuzetak u metodi run
- nit koja je usla u sinhronizovanu metodu moze pozivati druge sinhronizovane metode jer da to nije omoguceno nit bi mogla da blokira sama sebe
- ako se sinhronizuje staticka metoda, nit dolazi u posjed java.lang.Class objekta povezanog s klasom
- sinhronizacija se ne moze izvesti nad null vbrijednosti, u tom slucaju ce biti bacen NULLPOINTEREXCEPTION
- ako postoji synchronized blok u statickoj metodi, taj blok mora biti sinhronizovan nad objektom java.lang.Class
- ugnjezdena klasa moze da prosiruje proizvoljnu klasu, da implementira proizvoljan interfejs, da bude osnova za prosirivanje, da se deklarise kao final I abstract
- ime ugnjezdeno tipa dostupno je direktno u okruzujucem tipu, a izvan mu se pristupa navodjenjem punog imena OkruzujuciTip.UgnjezdeniTip
- ugnjezdena klasa ima pristup svim clanovima okruzujuce klase cak I ako su privatni
- ugnjezdedene klase mogu biti I private, protected I public
- okruzujuca klasa takodje ima pristup svim clanovima ugnjezdene klase
- tipovi ugnjezdeni u interfejsu su uvijek (podrazumjevano) javni
- klasa koja nasljedjuje ugnjezdenu klasu ne nasljedjuje prava pristupa
- ugnjezdena klasa moze biti staticka kao I svi drugi clanovi klase, postoje staticke ugnjezdedene klase I untrasnje ugnjezdedene klase (nestaticke)
- staticka ugnjezdena klasa ne moze da direktno pristupa nestatickim poljima ili metodama obuhvatajuce klase, moze samo preko refernce tog objekta
- klasa ugnjezdena u interfejsu je stalno javna I staticka I ugnjezdeni interfejs je uvijek staticki
- untrasnja klasa ne moze da sadrzi staticka polja osim final static koja se odmah inicijalizuju, a staticka ugnjezdena klasa moze imati polja static
- ako hocemo da u ugnjezdenoj klasi referenciramo neko polje iz okruzujuce mozemo to da uradimo preko punog imena Okruzujuca.this.polje
- lokalne klase se mogu definisati unutar metoda, unutar konstruktora, unutar inicijalizacionih blokova I oni nisu clanovi okruzujuce klase, one su pristupacne samo u tom bloku gdje su definisane
- reference na objekte lokalne klase se mogu prenositi kao argumenti I vracati kao rezultati metoda
- anonimna klasa ne moze da ima extends I implements klauzulu
- ako ima vise istih default metoda iz vise interfejsa, onda klasa mora da redefinisuje tu metodu I ako ne redefinisuje desava se greska pri kompajliranju, a ako postoji samo jedna default ne mora se redefinisati
- naziv klasa moze poceti malim slovom
- ILEGALARGUMENTEXCEPTION je RUNTIMEEXCEPTION I ne mora se baciti u metodi
- samo final konstante mogu biti u switch
- ako se u catch baca novi izuzetak ne mora se nigdje hvatati I ne mora biti u throws klauzuli, a prije tog bacanja se izvrši finally ako postoji



- try with resources je ista stvar kao i bez ikakvog try bloka
- iako se hvata Exception u metodi može se bacati izuzetak u metodi
- ako jedan interfejs nasljedjuje drugi interfejs i ako se pojavi metoda sa istim potpisom ali razlicitim povratnim tipom desice se greska pri kompajliranju jer je duplira metoda
- paziti na instanceof da li su u ikakvoj vezi, mogu biti dvije odvojene grane jedne nadklase
- kod enuma ako postoji podrazumjevani konstruktor onda se pri kreiranju tog tipa poziva konstruktor za svaki izraz u enumu, i enum se ne može instancirati, enum se ponasa kao primitivni tip, ako se ne inicijalizuje u METODI desava se greska pri kompajliranju, ako se ne inicijalizuje u KLASI onda je null
- konstruktor u enumu mora biti private
- iako se nista ne kreira enum se kreira prilikom ucitavanja tog enuma tj. ponasa se kao staticki kontekst
- bilo kakva veza sa Object clone mora implementirati Cloneable
- kada se klonira objekat dobija se novi objekat, ali ako postoji referenca na neki drugi objekat i ako se ona promijeni mijenja se i u clone(String je izuzetak jer je on IMMUTABLE)
- String string = "string"; String s6 = ("test " + string).intern() kaze da se provjeri to za vrijeme kompajliranja i onda to predstavlja isti objekat kao i String s1 = "test string"
- ako postoje dvije metode u klasi sa istim potpisom a razlicitim povratnim tipom, kompajler javlja gresku
- klasa nasljednica naslijedi i staticke metode osnovne klase
- Spoljasnja.Unutrasnja u = s.new Spoljasnja.Unutrasnja(); ovo je greska pri kompajliranju...Spoljasnja.Unutrasnja u = s.new Unutrasnja(); ovo je pravilno gdje je s referenca na spoljasnju klasu, mora se upotrijebiti objekat spoljasnje klase jer se unutrasnja klasa veze za spoljasnju
- anonimna klasa bukvalno predstavlja klasu koja je izvedena iz neke poznate klase i prosirena sa nekim dodatnim funkcijama, kada se izvrsava operator new anonimne klase poziva se konstruktor nad klase, može i podrazumjevani, a može i sa parametrima
- Ako definisete anonimnu unutrasnju klasu u kojoj želite da koristite objekat definisan izvan nje, prevodilac zahtjeva da taj objekat bude finalan. Zbog toga je argument metode odr() finalan. Ukoliko to zaboravite, dobija se greška pri kompajliranju, a ako se on prosljedjuje kroz konstruktor i nista se ne koristi u anonimnoj klasi onda ne mora da bude finalan, on se može i koristiti u metodi ako je effective final, a to znaci da mu je samo jednom mijenjana vrijednost prije koristenja, a ako je vise puta onda se desi greska pri kompajliranju
- blok za inicijalizaciju instance predtavlja konstruktor anonimne klase
- ako neka klasa nasljedjuje neku unutrasnju klasu ona ne može da ima podrazumjevani konstruktor i to ce dovesti do greske pri kompajliranju, mora da primi na reference spoljasnje ciju unutrasnju nasljedjuje i u tom konstrukturu mora biti ReferencaNaSpoljnuKlasu.super();
- kada se implementira metoda iz interfejsa ona mora imati isti povratni tip ili kovarijantni povratni tip, ili ce doci do greske pri kompajliranju jer ta metoda nije

redefinisana, a ako se onda napravi jos jedna metoda sa istim potpisom ali razlicitim povratnim tipom, doce do greske pri kompajliranju jer metoda sa takvim potpisom vec postoji

- toString() metoda throwable klase je ImeKlase : poruka , ili samo Ime klase ako poruka nije podesena
- paziti na kastovanje u privatnu unutrasnju klasu, to nece proci ako main nije u toj okruzjujucoj klasi koja sadrzi tu unutrasnju
- kod generickih tipova se ne moze: pozvati konstruktor paramtra, ne mogu nizovi, ne moze instanceof zato sto je to sve njemu isto
- jedini put kada se javlja greska pri kompajliranju kod raw tipova je kod kod unboxinga, npr imamo raw listu I ubacimo 5 kao broj I onda hocemo da izvorsimo int a = lista.get(0) I desava se unboxing I Object je I greska pri kompajliranju
- imamo primjer List<? extends Bird> lista = new ArrayList<Bird>(); u ovakvu kolekciju ne moze se nista dodavati NIGDJE
- imamo primjer List<? super Bird> lista = new ArrayList<Bird>(); u ovakvu kolekciju moze se dodavati jer u ovom slucaju znamo cijelu strukturu I nece se desiti CLASSCASTEXCEPTION
- List<X super B> ovo ne moze jer super samo moze ici sa dzoker znakom
- public class G<?> pri definiciji klase ovo ne moze
- metoda instanceof se ne moze redefinisati statickom metodom u klasi nasljednici
- u intefejsu staticke I default metode moraju biti definisane ili se desava greska pri kompajliranju
- staticka unutrasnja klasa moze se instancirati bez objekta okruzjujuce klase
- instanceof interfejsa stalno radi, nikada se nece desiti greska pri kompajliranju
- Outer.Inner I = new Outer().new Inner(); ovo prolazi
- poslije definicije enum vrijednosti ako postoje dodatni parametri I metode mora postojati ;
- ako je u pitanju lokalna unutrasnja klasa(klasa unutar metode) ili anonimna klasa onda ona NE MOZE DA MIJENJA SAMO LOKALNE PROMJENJIVE, POLJA MOZE
- Byte – 1B ; Short – 2B ; Int – 4B ; Long – 8B ; Float – 4B ; Double – 8B ; Char – 2B ; Boolean – 1bit