JAVA RAZUMEVANJE POJMOVA

KLASE I OBJEKTI, REFERENCE i ISTANCE

--- Klasa je kao neki plan za gradnju, znači kao neki blueprint po kome mogu više kuća na primer da se sagrade.

--- Klasu čine skup objekata iliti ga istanci koju je definišu bolje određuju itd.

--- Gledajuću tako, svaka kuća ima svoju lokaciju, tako i klase imaju svoju lokaciju iliti ga referencu.

--- Lokaciju ili referencu možemo proslediti konstruktoru ili metodama kao parametre za dalji rad.

THIS vs SUPER

--- This i super su keywordovi a this() i super () su metode.

--- Super keyword se koristi za pristup ili pozivanje roditeljske klase tj njenih promenljivih i metoda.

--- This keyword se koristi za pozivanje promeljivih i metoda u okviru svoje klase. Pozivanje this keyword je

neophodno kada imamo parametar sa istim imenom kao field (promenljiva u okviru neke klase). 95 % field je

uvek private, uvek ima private acces modifier.

--- Oba keyword možemo koristiti bilo gde osim u static bloku (static bloku ili static metoda).

Korišćenjem u ovom delu dovešće do compile-time error-a.

--- This keyword se obično koristi u konstruktorima i setterima, i ponekad u getterima ali više

početnici koriste ovaj deo.

--- Super keyword se obično koristi u overriding metodama, kada pozvani metod ima isto ime

u roditeljskoj klasi. Da ne stavimo keyword super, metod bi pozvao sam sebe zauvek

ili dok se ne popuni memorija.

--- U Javi imamo this() i super() što ih možemo svrstati kao pozivi.

--- This() se koristi za poziv konstruktora iz drugog konstruktora u istoj klasi.

--- Poziv this() može se samo koristiti u okviru konstruktora, i mora biti prva linija koda u konstruktoru. Koristi se za prilikom umnožavanja konstruktora kada jedan konstruktor

zove drugi zbog smanjena dupliranja koda.

--- Super() je jedini način za poziv konstruktora iz roditeljske klase.

--- Default se uvek stavlja super() u konstruktor bez argumenata, to stavlja kompajler.

--- Poziv super() uvek mora da bude prva linija koda u konstruktoru.

--- Čak i apstraktne klase imaju konstruktore, ali ih nikada ne možemo pozvati koristeći nove

keywordove.

--- Abstraktna klasa je i dalje super klasa, tako da je to poziv konstruktora kada neko instancira

Trenutnu klasu.

--- Konstuktor može da pozove super() i this() ali nikada obe u isto vreme.

METHOD OVERLOADING VS METHOD OVERRIDING

--- Metod overloading znači da imamo 2 ili više odvojenih metoda u klasi sa istim imenom ali

Imaju različite parametre.

--- Motod vraćanja može ali i ne mora da bude različito i to nam omogućava da koristimo

metode sa istim imenom više puta.

--- Overloading je dobar način da se kod koji pišemo ne duplira, i samim tim ne moramo da

pamtimo različita imena motoda.

--- Overloading nema nikakve veze sa polimorfizmom, ali većina Java programera overloading smatra kao Compile Time Polimorfizam.

--- Drugim rečima, kompajler odljičuje koja će motada biti pozvana po kriterijumu imena,

povratnog tipa i liste argumenata.

--- Overloadovati se mogu static metode i instance metoda.

--- Overloadovanje se obično dešava u jednoj klasi, ali metod se može isto tretirati

overloadovanim u podklasi te klase. Razlog toga je to što podklasa preuzima

(inherits) jednu verziju od metode u roditeljskoj klasi, i onda podklasa iliti

dečija klasa može imati drugu overloadovanu verziju od te metode.

--- Motode će se smatrati overloadovane ako budu ispunile oba uslova:

* Metode moraju da imaju isto ime
* Metode moraju da imaju različite parametre

--- Ako metode ispunjavaju uslove od gore navedenih, onda mogu imati ali ne moraju:

* Imati različit return type
* Imati različit kontrolni pristup (public, private, default)
* Imati različite čekirane ili ne čekirane izuzetke

METOD OVERRIDING

--- Metod overriding znači definisanje motoda u dečijoj klasi, koja već postoji u roditeljskoj

klasi potpuno ista(isto ime, isti argumenti).

--- Proširivanjem roditeljske klase, dečija klasa dobija sve metode definisane u roditeljskoj

klasi(te metode su poznate kao derivati metode).

--- Metoda overriding je i poznata kao Runtime Polimorfizam i Dynamic Method Dispatch, zato

što metoda koja će biti pozvana odlučuje JVM u runtim-u.

--- Kada overridujemo, preporučljivo je da odmah iznad metode stavimo @Override.

Ovo je naznaka da kompajler pročita i da nam pokaže error ako ne poštujemo pravila

Overrida.

--- Ne možemo da overridujemo statične metode, samo istance.

--- Pravila za uspešno overridovanje:

* Mora da ima isto ime i argumente
* Return type može da bude podklasa return type u roditeljskoj klasi
* Ne može da ima niži po značenju acces modifier
* Ako je metod u roditeljskoj protected mi ne smemo da koristimo private u dečijoj, ali smemo da koristimo public

--- Važne stvari za overriding metoda

* Samo inherited (nasleđene) metode mogu biti overridovane, tačnije overriding može samo u dečijim klasama
* Konstruktori i private metode ne mogu biti overridovane
* Ne mogu ni metode koje su finalne ili final metoda
* Poklasa može da koristi super.methodName() da pozove super klass verziju overidovane metode

STATIC METODE VS ISTANCE METODA

--- Static metode su definisane statičkim modifikatorom pristupa.

--- Static metode ne mogu da pristupaju instance metodama i instanci promenljivih, direktno.

--- Static metode se obično koriste za operacije koje ne zahtevaju nikakve podatke iz instance

klasa(this).

--- Podsetnik, this keyword je trenutna instanca klase.

--- Znači u static metodi NE MOŽEMO da koristimo this keyword.

--- Kad god se vidi metod koji ne koristi instance promenljivih, taj metod bi trebao da bude

static metod.

--- Na primer, main metod je static metod i njega poziva JVM.

INSTANCE METODE

--- Instance metode pripadaju instanci klase.

--- Da bi koristili instance metode, prvo moramo da napravimo instancu klase tako što

koristimo keyword new.

--- Instance metode mogu da pristupe instance metodama i instance promenljivama direktno.

--- Instance metode mogu da pristupe direktno i static metodama i static promenljivama.

STATIC VS INSTANCE VARIABLES

STATIC PROMENLJIVA

--- Static promenljive se deklarišu sa keywordom static.

--- Static promenljive su još poznatije kao i static member promenljive.

--- Svaka instanca klase deli istu static promenljivu.

--- Ako je promena nastala na toj promenljivoj, sve ostale instance će videti promenu.

--- Static promenljive se ne koriste često, ali nekad imaju svoju primenu. Na primer, kada

čitamo unos korisnika koristeći Skener, mi ćemo tada Skener deklarisati kao statičnu

promenljivu. Tako mogu static metode da im pristupe direktno.

INSTANCE PROMENLJIVA

--- Ne koriste keyword static.

--- Instance promenljive su i poznate kao fields tj polja ili kao member promenljiva.

--- Instance promenljive pripadaju instanci klase.

--- Svaka instanca ima svoju kopiju instance promenljive. Svaka instanca može imati drugačiju

vrednost tj. stanje. Instanca promenljivih predstavlja stanje same instance.

KOMPOZICIJA

--- Kompozicija se od inheritensa ilitiga od nasleđivanja razlikuje tako što, kod nasleđivanja

dečija klasa nasleđuje od roditeljske klase stvari tj instance, i možemo reći da su

jako slične. Kao vozilo i tipa Golf. Golf jeste vozilo i vozilo jeste Golf. Takva veza.

--- Kompozicija je kao što sama reč kaže samo komponenta veće slike, tipa kao procesor,

matična ploča itd naspram računara. To sve čini računar zajedno ali posebno tj svaka

za sebe ne čine računar.

NIZOVI

--- Nizovi su struktura podataka koja omoguaćava da skladištimo više vrednosti u jednu

promenljivu.

--- Difoltna vrednost u brojčanom nizu elemenata setovana je na nulu.

--- Nizovi su indexirani od 0. I računa se od 0 do n-1 to jest od 10 elemenata idex kreće od 0

I završava se sa 9.

--- Ako pokušamo da pristupimo indexu van opsega, Java će da vrati

ArrayIndexOutOfBoundsException, što govori da je index van opsega.

--- Da pristupimo elementima niza koristimo [ ] zagrade, prepoznatljive kao i array access

operator.

--- Kao što smo rekli, difoltna vrednost brojnom nizu je nula, boolean-u je false a Stringu

je null.

--- Niz se može odmah i odrediti pomoću { }, čime odmah možemo staviti elemente u niz.

Tipa int[ ] array = { 1, 2,3}; čiji je logičan sadržaj, 1,2 i 3.

ABSTRAKTNE KLASE VS INTERFEJSI

--- Abstraktne klase su slične interfejsima ali abstraktne klase ne možemo da instanciramo, i one mogu da imaju mix deklarisanih metoda koje su ili deklarisane ili nisu. Što znači da ne

ne moraju da imaju deklarisane metode.

--- Kod abstraktnih klasa, mi možemo da definišemo field ili polja koja nisu statična i finalna,

I možemo staviti public, protected ili private metode.

--- Abstraktna klasa može da proširi ili extenduje samo jednu roditeljsku klasu ali

može da implementira više interfejsa.

--- Kada je abstraktna klasa podklasa, podklasa obično daje implementaciju za sve abstraktne

metode u roditeljskoj klasi.

--- Kako god, ako se to ne desi, onda i podklasa ili dečija klasa mora da bude abstraktna to jest

da se nazove.

KORISTI ABSTRAKTNE KLASE KADA:

* Kada želiš da deliš code sa nekoliko jako bliskih klasa koje imaju nešto zajedničko
* Kao na primer: Animal – field name, age
* Očekuješ da klase koje extenduju od tvoje abstraktne klase imaju puno istih metoda ili

polja ili zahtevaju access modifaje da budu drugi od publica(protected ili private)

* Želiš da staviš non static ili non final field iliti ga polje(primer: name, age). Ovo ti omogućava da definišeš motode koje mogu da pristupe i izmene objekat(getName, setName).
* Kad imaš zahtev za svoju trenutnu klasu da implementiraš neku metodu ali da druge metode budu otvorene za override metode od dečije klase
* Zaključak: Potreba abstrakte klase je da da klasičnu definiciju normalne klase od koje mogu više različitih klasa da dele.

INTERFEJS:

--- Interfejs je samo skup metoda klase, nije implementacija. Znači samo skup metoda

--- U interfejsu mi određujemo kakvu će operaciju objekat da izvrši. Ove operacije su definisane

klasama koje implementiraju interfejse.

--- Interfejs formira ugovor ili vezu između klasa i spoljašnjeg sveta, i ova veza je učvršćena u

build vremenu od kompajlera.

--- Ne možemo stvoriti instancu na njima, i mogu da sadrže mix metoda određenim sa ili bez

Implementacije. Sve metode u interfejsu su automatski public i abstract.

--- Interfejs može da extenduje drugi interfejs.

--- Interfejsi su više fleksibilni i mogu da podnesu mnogo više stresa u dizajniranju programa

nego implementacije.

--- Uvođenjem interfejsa u tvoj program, ti stvarno uvodiš tačke promenljivih u koje ti možeš

da staviš druge implementacije za taj interfejs. Pravi cilj interfejsa je abstrakcija,

razdvajajući šta od kako ?

--- Od Jave 8 interfejsi mogu da zadrže default metode, drugim rečima metode sa

Implementacijom. Glavna reč default se koristi (najviše za kompaktibilnost unazad),

i statične metode.

--- Od Jave 9 interfejsi mogu još da sadrže private motede(najčešće korišćene kada dve default

metode u interfejsu dele često korišćen code).

KORISTI INTERFEJSE KADA:

* Smatraš da će nepovezane klase koristiti tvoj interfejs(nemaju nešto toliko zajedničko).

Kao na primer interfejsi kao Comparable i Clonable se često koriste od strane

ne povezanih tj različitih klasa.

* Želiš da preciziraš ponašanje specifičnog data type-a, ali te ne zanima ko implementira

ponašanje.

* Želiš da razdvojiš različita ponašanja.
* The Collections API je super primer, imamo List interfejs i implementaciju

ArrayListe i LinkListe.

* The JDBC API je još jedan super primer. Postoji od skoro samo interfejsa.

Implementacije su pokazane kao JDBC drivers. Ovo nam omogućava

da pišemo sav JDBC code odvojeno od baze podataka DB vendor.