Java mini-konspektas

Klasės ir objektai

Tarkime turim:

Class Point{

int a;

static int z;

public Point() { this.a = 5;}

private Point(int tmp) { a = tmp; } [\\Privatus](file:///\\Privatus) konstruktorius **WTF?**

public static Point createWithVal(int tmp) { return new Point(tmp); } [\\Pvz](file:///\\Pvz)

public static void print1 (){ sout(a); } [\\EROR](file:///\\EROR), negalima **statiniam** metode naudoti **nestatinių** kint.

public static void print2 (){ sout(z); } [\\Viskas](file:///\\Viskas) tvarkoj

}

Point p1 = new Point(); Point p2 = new Point();

Point p3 = p1;

Su **New** sukuriam naują unikalų objektą ir mum yra gražinamas reference(adresas/nuoroda).

* (p1 == p2)? -> False, nors ir p1.a = p2.a, tai yra atsikiri objektai
* (p3 == p1)? -> True, nes abu rodo į tą patį objektą
* p1.z = 5; (p1.z == p2.z)? -> True, nes statiniai kint yra bendri visos klasės obj.
* p1.print2() ir Point.print2() abu galimi variantai naudoti static metodą

**Kam naudojami privatūs konstruktoriai?**

1. Jei turim klasę su statineis metodais ir nenorim, kad būtų galima sukurti jos obj.
2. Singletonui
3. Jei turim klasę, kur kintamieji yra tik konstantos
4. Jei norim, kad konstruktorius būtų labiau self-documenting

**Singleton –** Programavimo principas, kai klasėje **Singl**(Čia yra kaip pavyzdinis pavadinimas) yra konstruktorius padaramas privatus, pačioje klasėseje sukurimas **Singl** objektas ir aprašytas metodas getInstance(); kuris grąžina reference į jau sukurtą **Singl** objektą. Panaudojimas:

* Kai norim užtikrinti, kad egzistuotų tik vienas klasės obj.
* Reikia kreiptis į jį iš skirtingų vietų appse
* Kai norim išsaugoti informaciją, net kai į ją neturime rodyklės

**Kompozicija –** Kai vienas obj. Susideda iš kitų

**Paveldėjimas –** Kai vienas obj. Paveldi savybes iš kito su žodžiu extendens arba Implements. Su **Super()** galima patikslinti, kurį tėvinės klasės konstr. Kviesti

Pagal nutylėjimą JAVA visi metodai yra „virtual“, bet nėra virtual konstruktorių

Skirtumas tarp abstrakčių klasių ir interfeisų

|  |  |
| --- | --- |
| Abstrakčios klasės | Interfeisai |
| Galima extendint tik vieną klasę | Galima implementinti bet kokį skaičių interfeisų |
| Gali extendinti kitą klasę (Abstrakščią arba paprastą) | Interfeisai gali extendinti tik interfeisus |
| Gali turėti abstrakčius ir konkrečius metodus | Gali turėti tik abstrakčius metodus |
| Kad metodas būtų abstraktus, reikia deklaruoti/parašyti “abstract” | Kad metodas būtų abstraktus, nebūtina parašyti “abstract” |
| Gali turėti protected ir public metodus | Gali turėti tik public metodus |
| Gali turėti **static, final, static final,** kintamuosius | Gali turėti tik **public static final,** kintamajį |

**Delegation –** Klasė, kurios darbas yra nukreipti/adresuoti objektą toje klasėje, kad jis atliktų savo funkciją. Su delegacija yra galimybė apeiti JAVA hujovą idėją, kad galima paveldėti tik vieną klasę.

**Proxy –** Peradresuoja klientą į tikrajį objektą, veikia kaip tarpininkas. Panaudojimas:

1. Reikia valdyti prieigą prie objekto
2. Norime paslėpti objekto veikimo sudėtingumą

Objektų Klonavimas

Klonas yra tiesiog kopija, kurio nors objekto. JAVA tai yra tiesiog objektas, kuris yra panašioje būsenoje, kaip ir orginalas. Klonavimui, reikia implementuoti **Cloneable** ir overridinti **clone().**

Jei klasė turi tik primityvius kintamuosiuis(int, double, etc…) tai padaromas pilnasklonavimas ir gauname naują obj. nepriklausomą nuo orginalaus obj.

Jei klasė turi primityvius kint. ir obj. Tai primityvius kintamuosiuis nukopijuoja ir grąžina reference(adresą) kitų klasės obj.

Klonavimas yra dviejų tipų:

* **Shallow** **–** Paprasta kopija, kur klasėje esantys objektai nekopijuojami, o tiesiog nukupijuojamas reference(adresas). Iškyla problema, kad jei pakeisime klonuoto objekto viduje esantį objektą, pasikeis ir orginalas
* **Deep -** Klonas yra visiškai nepriklausomas nuo savo orginalo, tai yra padarom, truputį pamodifikavus **clone()** metodą.

Kopijos Konstruktorius

Tarkim turim dvi klases

Public class PointOne

{

Private int x;

Private int y;

Public PointOne(PointOne point) {this.x = point.x; this.y = point.y;}

public PointOne copyPoint(PointOne point) throws CloneNotSupportedException{

if(!(point instanceof Cloneable)) {

throw new CloneNotSupportedException("Invalid cloning"

}

return new PointOne(point.x, point.y);}

}

Public class PointTwo extends PointOne

{

Private int z;

Public PointTwo(PointTwo point){super(point); this.z = point.z;}

}

Jeigu męs bandytume klonuoti pirmą arba antrą klasę, mum visada objektas gautūsi **PointOne** tipo, kad taip nebūtų reikia apsirašyti papildomą metodą **PointOne** klasėje(pažymėtas raudonai)

**Package –** Tai realiai tas pats, kas namespace C++, naudojami, tam kad nebūtų kintamųjų vardų kolizijos ir kad būtų patogiau grupuoti failus.

Exceptionai

Pagrindinei keywords: try-catch-finally-throw

Veikimo principas: Vygdoma funkcija **try** blocke -> Jei įvyksta klaida funkcijoje ir yra **throwinamas erroras**, likusios funkcijos dalies nevygdo -> error sugauna pirmas **catch‘as** kuris gali jį gaudyti -> **catchas** susitvarko su error, kad programa nelūžtų -> tesemas darbas toliau **try** block -> jeigu yra **finally** blokas jis yra įvygdomas(Jis yra įvygdomas, bet kuriuo atvėju, rastas error ar ne).

Visada, dėti pirma savo apsirašytus excetptionus catch blokuose, o poto tik bendrus, nes jei bus įdėtas benras, tai iki sukonkretinto exceptiono gali nedaeiti.

Jeigu iš **finally** blocko išeinama su **break/return/throw** sakiniu pirminė priežąstis(throwintas erroras) dėl kurios buvo išeita iš try blocko yra pamirštama.

UML diagramos

Jos yra naudojamos, kad būtų lengviau atvaizduote diagramų hierarchiją