Konspektas atliktas remiantis Vilniaus universiteto **Rimanto Vaicekausko** objektinio programavimo paskaitų medžiaga.

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

Reference tipai

* **Strong reference –** Standartinis reference Point a = new Point();
* **Weak reference –** Naudojimas **WeakReference**<Point> weakR = new **WeakReference**<Point>(a); **JVM** sutikęs tokį obj. pažymės trinimui, dažniausiai naudojami DBConection‘ui
* **Soft Reference –** Naudojimas **SoftReference**<Point> softR = new **SoftReference**<Point>(a) skiriasi nuo **weak reference**, tik tuo, kad jei **JVM** pažymės obj. trinimui, jį **GC** ištrins tik tada, kai trūks atminties
* **Phantom Reference –** Naudojimas **PhantomReference**<Point> phantomR = new **PhantomReference**<Point>(a); skiriasi nuo **weak reference**, tik tuo, kad jei **JVM** pažymi obj. trinimui, jį **GC** įdės į ‚reference queue‘

**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. Statiniai elementai nėra paveldimi, jie yra bendri visom klasėm. Su **Super()** galima patikslinti, kurį tėvinės klasės konstr. Kviesti

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

**Asocijacija –** Kai du atskiri objektai yra susija, pvz turime klasę bank ir consumer,abu šie objektai gali būti susija, bet nebūtinai bank turi turėti consumer ir atvirkščiai.(Kad aiškiau būtų, atsiverskit Association.java)

**Aggregation –** Kai vienas klasės objektas turi kitą, bet nėra priklausomas nuo jo, pvz: consumer klasė turi bank obj. consumer klasė gali egzistuoti ir be bank ojb. Tai pat bank klasė gal egzistuoti nebūdama consumer dalimi(Kad aiškiau būtų, atsiverskit Aggregation.java)

**Kompozicija –** Kai klasė sudaryti iš kito/ų obj. be kurių ji negali funkcionuoti. Pvz: turim klasę library ir toje klasėje turim listą obj. book, jei sunaikinsim library dings ir book listas, taip pat negalime turėti library be book listo(Kad aiškiau būtų, atsiverskit Composition.java)

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 pavaizduoti klasių hierarchija, dažniausiai naudojama:

1. projektavimui,
2. konstravimui,
3. dokumentavimui

Yra dviejų tipų: **Struktūrinės**(klasės, objektai, Paketai, Komponentai) ir **Elgsenos**(Veiklos, Būsenų, Panaudojimo)

I/O Srautai

Srautų tipai:

* **InputStream** ir **OutputStream –** operuoja baitais(byte tipas)
* **Reader** ir **Writer –** operuoja simboliais(char tipas), leidžia išvengti simbolių kodavimo ir dekodavimo problemų(Galima operuoti simboliais virš baitų srauto)

**Specializuotos I/O klasės –** Įgalina atlikti specializuotas išvestis ir įvestis, parūpina papildomą funkcionalumą.

**Filtrai –** prideda papildomų funkcinių galimybių, pvz: **BufferedInputStream() –** moka buferizuot baitus, o **LineNumberOutputStream –** moka suskaičiuoti nuskaitytas eilutes. Abu apjungus galime gauti **getLineNumber()**

**Serializacija –** Kur Java obj. Gali būti paverstas baitais, įskaitant jo visus fieldus ir duomenis ir taip pat pačio objekto tipą. Šita dalis nepriklauso nuo JVM(tai galima padaryti ant bet kokios platformos). Kad naudotis **Serializacija** reikia klasėje implementuoti java.ioSerializable.

Gijos

**Gijos –** tai „lengvi“ lygiagretūs procesai programos viduje. Juose nepatartina naudoti ciklų. Jas panaudoti yra du būdai:

1. **Implement Runnable –** Klasė implementuoja **Runnable** interfeisą ir realizuoja metodą **run()**. Tada yra sukuriamas naujas **Thread** objektas ir į jo parametrus paduodama klasės obj ir iškviečiama komanda **start()**
2. **Extend Thread –** Klasė extendina **Thread** klasę ir užklojamas **run()** metodas. Tada sukuriamas tos klasės objektas ir išvkičiamas metodas **start()**

**Giją** galima sustabdyti, kažkuriam laiko tarpui su komanda **sleep(**milisekundės**)**, o su **interupt()** komanda jos veikimą galima pratęsti

**Gijų sinchronizavimas –** Dažniausiai nodojami yra monitoriai, tai yra **synchronized(**objektas**)** metodas, tas leidžia duomenis skaityti ir modifikuoti korektiškai.

Masyvai. Objektų talpyklos

**Java Masyvai –** Gali būti klonuojami(Shallow) ir serializuojami

**Java collectinai:**

1. **List** – Elementų seka
2. **Set** – Rinkinys be pasikartojančių elementų
3. **Map** – Poros <raktas, reikšmė> aibė
4. **Collection** – objektų elementų rinkinys

Java nuo 5 versijos

1. Parametrizuoti tipai(**generics**) – Realiai veikia kaip C++ Template, taip pat sumažina klaidų kiekį, nes tikrinimas vyksta kompiliavimo metu. Negalima turėti statinių **generics** kintamųjų
2. For ciklo iteravimas – Seniau reikėjo naudoti iteratorius, o dabar yra for( obj : c)
3. Collection<**?**> - Visų kolekcijų supertipas.
4. Collections< ? extends **class**> - Leidžia naudoti betkokius **class** vaikus arba pačia **class**, kaip tipą collection‘ui
5. **ENUM –** Enumeracijos, tai realiai klasė, kurios objektai, konstantiniai elementai.
6. **Lambda –** Supaprastina kodą ir palengvina funkcinį programavimą.

Garbage Collection

**Nepasiekiamas objektas –** Tarkim mes sukuriam Integer i = new Integer(4), mūsų reikšmė 4 yra pasiekiama tik per referenc‘ą i, tai jeigu mes parašome i = null, mūsų reikšmė „tiesiog plaukioja“ atmintyje ir yra nebepasiekiama.

**GC**(Garbage collector) – atlaisvina memory hepą, ištrindamas nepasiekiamą antmintį. Jis automatiškai, kas kiek laiko prabėga pro atmintį.

Pagrinde yra keturi būdai sudaryti sąlygas, kad **GC** surinktų objektą:

1. Reference(adresą) prilyginti null
2. Reference(adresą) prilyginti kitai reikšmei
3. Objektas išeina iš galiojimo srities, pvz metode sukurti obj, automatiškai ištrinami, kai baigiam vygdyti metodą
4. [Island of Isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/)

**GC** gali kviesti ir pats developeris:

* Naudoti **System.gc()** metodą
* Naudoti **Runtime.getRuntime().gc()** metodą

**Daemon thread** – Gija, su mažu prioritetu, kur background‘e gali rinkti šiukšles. Jeigu norim Giją padaryti **Daemon**, tai kai sukuriam obj. parašom obj.setDaemon(true)