Tuletamine ja pärimine

• Tähistab üldistusseost

– objekt on millegi alamliik (tüüp), ingl.k "is-a"

– uus klass e. alamklass (*derived class*) on sarnane olemasolevaga

– saame kaasa kõik ülemklassi (*base class*) meetodid ja väljad

• Süntaks

class SubClass extends SuperClass {}

class Detergent extends Cleanser {}

Koodinäide: **Detergent.java**

– Vaikimisi on iga asi objekt

class Cleanser {}

class Cleanser extends Object {}

• Käitumine tuletatud klassis

– Pärandamine

• alamklass saab kõik ülemklassi meetodid

– Lisamine

• kirjutame uue meetodi - makeBubbles()

– Muutmine

• asendame (*override*) päritud meetodi scrub()

• objekt sisaldab ülemklassi objekti - viit võtmesõnaga super

Alamklassi meetodite ülekate

Nimede peitmine

• Alamklassis võib üle katta (*overload*) ülemklassi meetodeid

vrdl. C++ga kus Barn klassist kaob ära meetod add(String)

class Storage {

String add(String a) {}

int add(int a) {}

}

class Barn extends Storage {

// C++-s peidaks see ülemklassi add() meetodid

// Javas lisab täiendava add() meetodi

Object add(Object l) {}

}

• Meetodi asendamisel (*override*) kasutage @Override

annotatsiooni

class Cellar extends Storage {

@Override String add(String a) {}

}

Hierarhia sisene ligipääs

Protected

• Hea tava laiendatava klassi kavandamisel

– Väljad private, meetodid public

• Private võib piirata laiendusvõimalusi

• väli/meetod

protected – on maailma jaoks kättesaamatu

– nähtav kõikidele alamklassidele / sama paketi

klassidele

– väljad on soovitav jätta private, ligipääsuks luua

Protectedmeetod

Hierarhia sisene tüübiteisendus

(*upcasting*)

• Pärimise eesmärk - alamklassil sama liides, mis ülemklassil

class Fruit {

void eat() {

System.out.println("Nam nam nam");

}

}

– Alamklass oskab reageerida ülemklassi meetoditele

{()}

class Orange extends Fruit{ }

...

Orange o = new Orange();

o.eat();

– Alamklassi saab kasutada ülemklassina - ülesteisendus (*upcasting*)

Fruit f = o; // automaatne ülesteisendus

// Sama, mis Fruit f = (Fruit) o;

f.eat();

• Ülesteisendusel kaotab klass meetodeid (väiksem ühisosa)

Muutumatud suurused

*final*

• Final on midagi, mis ei muutu

• Tähendus sõltub võtmesõna final asukohast

– Välja ees - väärtust ei saa üle kirjutada (konstant)

• Objektmuutujate korral ei saa viita ümber tõsta, objekti sisu on muudetav!

• Algväärtust ei pea deklareerima, määratakse konstruktoris (*blank final*)

– Argumendi tüübi ees - meetodi sees ei saa argumenti viita muuta

– Meetodi ees - pärimisel ei saa meetodit alistada (*overriding*)

• Kaitseb kriitilist funktsionaalsust tahtmatu rikkumise eest

– Klassi ees - klassi ei ole võimalik pärimisega laiendada

• Järeldused

– final meetod ei ole asendatav

– Meetodite/klasside final-iks märkimine on ohtlik - kaotame klassi laiendusvõimaluse

Kokkuvõte

Kumb valida: kompositsioon või pärimine?

• Kompositsioon

– Kõige levinum taaskasutuse võte

– Kirjeldab "koosneb"-seost (*has-a*)

– Vajame teise klassi funktsionaalsust, aga mitte liidest

– Komponendid deklareeritakse reeglina private ligipääsuga

• Pärimine

– Kasutatakse harvem

– Teeme olemasolevast klassist uue (parema?) versiooni

– Klassi funktsionaalsuse kohandamine vastavalt vajadusele

– Kirjeldab "on"-seost (*is-a*)

– Säilib klassi üldisem versioon - baasklassi liides

– Kas vajame ülesteisendust (*upcasting*)?

• Apelsinid ja Virsikud on ühed Puuviljad kõik!

Mitmene pärimine

*multiple inheritance*

• Klass laiendab mitut klassi

– Konn on korraga veeloom ja

maismaaloom

• Probleem: sama meetod asub

mitmes klassis

– Kuidas kombineerida realisatsioonid?

– Mis järjekorras väärtustada objektid

(konstruktorite väljakutse)?

• Java lahendus:

– Spetsiaalne klassi liik, millel puudub

realisatsioon

– Kui puudub realisatsioon, kaovad ka

loetletud probleemid

Liides

interface

• Abstraktse klassi eriliik

interface TerrestrialAnimal

– Omab kahte ligipääsu: public või package Access

• Ei sisalda realisatsiooni

– Määrab vormi, aga jätab sisu lahtiseks

– Kõik meetodid on ainult deklaratsioonid, puudub keha

void run();

– Liidese meetodid on alati public

• Liidese laiendamiseks tuleb see realiseerida

– Realiseerimisel anname meetoditele sisu

class Frog implements TerrestrialAnimal {

public void run() {

// boing-boing-boing

}

– Realisatsioon ei tohi vähendada nähtavust - meetod määrata kui public

• Üks klass võib laiendada mitut liidest

class Frog implements TerrestrialAnimal, AquaticAnimal

• Liides võib laiendada teist liidest

interface EvilMonster extends Monster

• Realiseerimisel anname sisu

kõikidele meetoditele

class Dragon implements EvilMonster {

void haunt() {}

void doHarm() {}

}

Liidese eripärad

• Liidese väljad

– On alati staatilised ja final

– Sobivad konstantide defineerimiseks

– Saab väärtustada staatilise meetodiga

• Vältige nimede kollisiooni

– Klass ei saa realiseerida liideseid, mille samanimelistel meetoditel on erinevad tagastustüübid

interface Jumping {

void jump();

}

interface Leaping {

int jump();

}

class HappyFrog implements Jumping, Leaping {

// kumba jump() meetodit peaksime realiseerima

}

Java standardliidesed

java.util.Collection

• Olulisemad meetodid:

– add(Object o) - elemendi lisamine

– clear() - tühjendamine

– contains(Object o) - elemendi sisalduvuse otsing

– iterator() - iteraatori loomine-tagastamine

– remove(Object o) - elemendi eemaldamine

– size() - suuruse küsimine

• Olulisemad alam-liidesed:

– java.util.List - järjestatud loend

• Elementidel on positsioon, indekseerimine läbi positsiooni: get(i)

• Lisamine, kustutamine positsioonipõhiselt: remove(i)

– java.util.Set - unikaalsete elementide (järjestamata) hulk

• Järjekord puudub, positsioonipõhine indekseerimine ei toimi

• Kasutamine:

– Programmeerime liidese, mitte konkreetse realisatsiooni vastu

– Kui vajame ainult elementide kogumit, positsioon ei ole oluline:

Collection<Frog> frogs = new ArrayList<Frog>();

– Kui vajame positsioonipõhist ligipääsu:

List<Frog> frogs = new LinkedList<Frog>();

Liides Comparable

– Defineerib klassis objektide *loomuliku* järjekorra

– Täielik järjestus – õnnestub leida minimaalne element

– Meetod compareTo(Object other)

• võrdleb jooksvat objekti (this) etteantud objektiga other

– Realiseerimine:

x.compareTo(y)

• Liides Comparator

– Kasutada, kui klassil puudub *loomulik* objektide järjekord

– Meetod compare(Object one, Object other)

• võrdleb, kas objekt one on suurem objektist other

– Realiseerimisel sama tulemus, mis one.compareTo(other)

• Kasutuskoht - objektide sortimine: Collections.sort(),

Arrays.sort()

Objektist koopia loomine

java.lang.Cloneable

• Idee: igal objektil clone() meetod

– Koopia saamiseks Dog copy = (Dog) dog.clone();

– Java keeles halvasti disainitud, raske korrektselt

kasutada [2]

– Kasutada ainult siis, kui tarvis massiivist

“pinnapealset” koopiat

– Kloonimise asemel loo parem *Copy Constructor*

ning vajadusel meetodid shallowCopy() /

deepCopy()

• Objekti kopeerimine

– Pinnapealne koopia (shallow copy)

• Kopeerime objekti väljad bitt-haaval

• Viitmuutujad näitavad endisesse kohta

• Tehakse tavaliselt Copy Constructori abil

– Sügav koopia (deep copy)

• Koopia objektist ja rekursiivselt igast tema väljast

Polümorfism praktikas

• Tarkvaraarenduse püsiväärtus on:

**Muudatus**

• Programmi disain peab soodustama muudatusi

– Kui tarkvara ei kohandu, siis seda ei kasutata

**Leiame programmi muutuvad aspektid ning eraldame need püsivatest**

– Kapseldame muutuva osa (tõstame eraldi meetodisse / klassi)

– Muudetav osa on kasutatav iseseisva “moodulina”, mida saab täiendada nii, et see ei mõjutaks programmi püsivat osa

Sisemised klassid

inner classes

• Klassi sees saab defineerida uue klassi:

class Parcel {

class Content{

}

void dispatch() {

Content c = new Content();

}

}

• Aitab koodi grupeerida ja peita

– Klass Content ei ole universaalselt kasutatav, sõltub konkreetsest Parcel1 klassist

– Kasutamine - nt. sisemiste (ajutiste) vahetulemuste hoidmine

• Väljaspool klassi deklareerime kui Välimine.Sisemine

Parcel.Content c

– Sisemisest klassist objekti loomisel eeldame, et välimine objekt (p) on olemas

Parcel p = new Parcel();

// uus Parcel1.Content läbi objekti p

Parcel.Content pc = p.new Content();

Anonüümne sisemine klass (*anonymous inner class*)

– Ühekordselt kasutatav klass - loome uue objekti ilma klassile nime andmata

– Kasutatakse objektide kokku sidumisel:

• Sortimisel Comparator liidese ühekordne realiseerimine

• Isetehtud andmestruktuurile Iterable liidese

realiseerimine

• GUI sündmusekuularite *(event handler)* loomine (ActionListener)