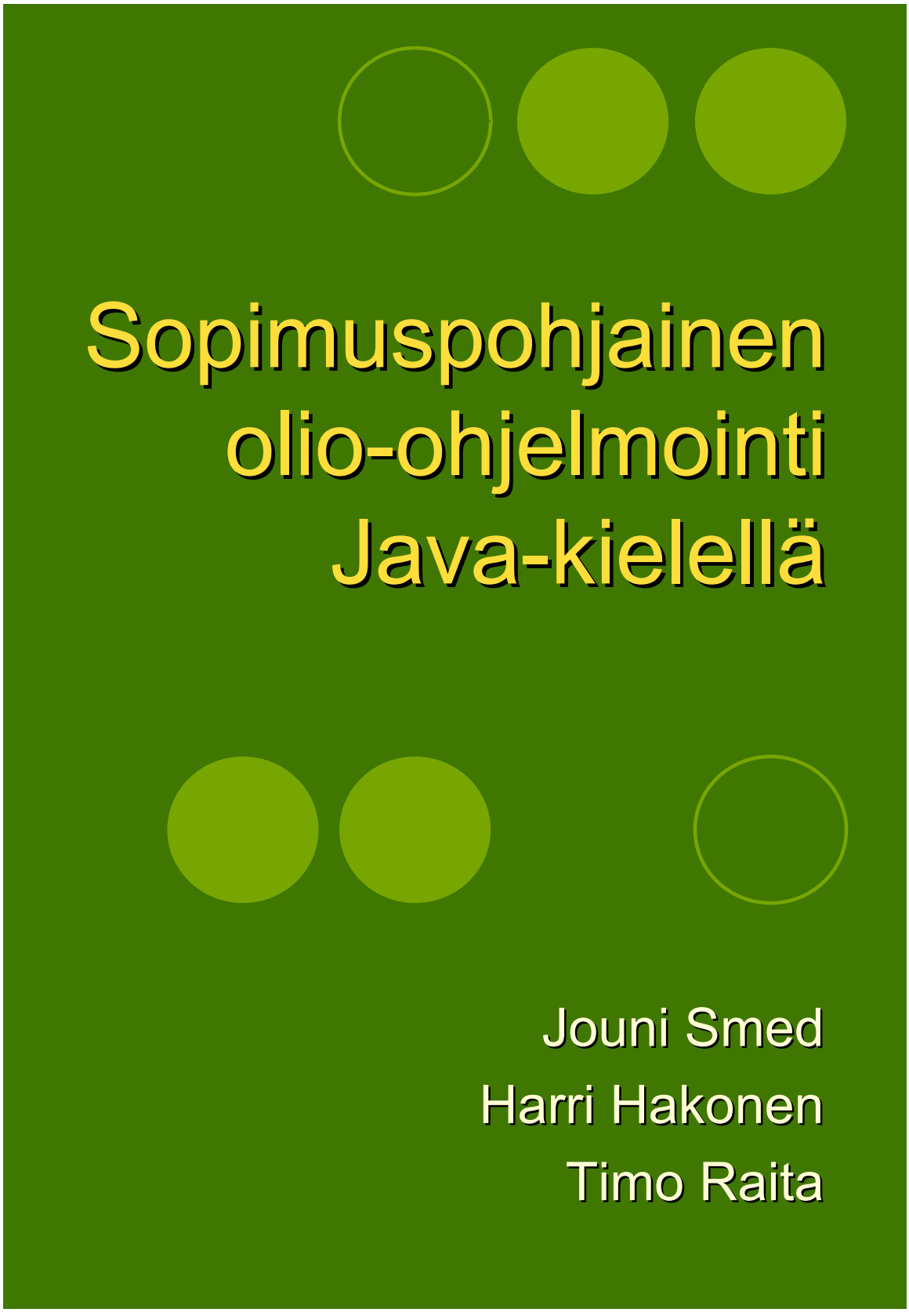
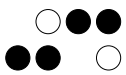

Sopimuspohjainen
olio-ohjelmointi
Java-kielellä
Jouni Smed
Harri Hakonen
Timo Raita

Sopimuspohjainen olio-ohjelmointi Java-kielellä

Jouni Smed *Harri Hakonen* *Timo Raita*

Turun yliopisto, Informaatioteknologian laitos



ISBN 978-952-92-1775-5 (nid.)
ISBN 978-952-92-1776-2 (PDF)

Sisältö

Taulukot

Kuvat

Listaukset

Esipuhe

Kädessäsi oleva oppimateriaali on alunperin syntynyt tukemaan ja täydentämään Turun yliopiston Informaatioteknologian laitoksen olio-ohjelmointikursseja. Ajan saatossa olemme huomanneet, että on olemassa laajempikin tarve suomenkieliselle ohjelmoinnin perusongelmiin keskittyvälle monografialle. Tämän vuoksi julkaisemme olio-ohjelmoinnin liittyvän taitotietomme tänä elektronisena kirjana kaikkien luettavaksi. Koska nykyisen ohjelmistotuotannon ongelmat ovat aivan muualla kuin varsinaisessa ohjelmoinnissa, tässä kirjassa käsitellyt asiat kuuluvat olio-ohjelmoijan perusammattitaitoon ja siten hänen yleissivistykseensä.

Tässä kirjassa keskitytään nimenomaan ohjelmointiasioihin ottamatta kantaa siihen millainen ohjelmistonkehitysprosessi on käytössä. Esitettyjä periaatteita voidaan siis soveltaa niin agile- kuin suunnittelupainotteisessa ohjelmistotuotannossa. Pohjatietona oletetaan Java-kielen syntaksin hallinta sekä ohjelmoinnin peruskäsitteiden teoreettinen ja käytännön tuntemus, joita tämä kirja pyrkii syventämään. Johtavana teemana on, miten kirjoitetaan hyviä ohjelmia; sellaisia, jotka on jäsennelty hyvin, helppo ymmärtää, korjata ja uudelleenkäyttää sekä joita ennen kaikkea voidaan ylläpitää. On tunnettua, että ohjelmointityön kustannuksista noin 70–80% aiheutuu ylläpidosta (virheiden korjaamisesta ja uusien piirteiden kirjoittamisesta ohjelmistoon) ja tämä on omiaan korostamaan sitä, että ennustettavien muutosten tekemisen tarve tulisi ottaa huomioon jo etukäteen, ts. tällaiset muutospaikat tulisi olla helposti löydettävissä ja korjausten tekeminen yksinkertaista. Yksinkertaisuus tarkoittaa tässä yhteydessä sitä, että koodia on helppo ymmärtää ja muutokset lokalisoituvat pienelle alueelle, eivätkä generoi lisäkorjauksia. Tätä päämäärää voidaan tavoitella käyttämällä kaikkia niitä keinoja, joilla ohjelmistokomponentit saadaan riippumattomiksi toisistaan. Erityisesti sopimuspohjainen ohjelmointi ja luokan sisäisen esitysmuodon kapselointi ovat mekanismeja, jotka lisäävät ohjelman osien välistä riippumattomuutta. Riippumattomuushan on ehdoton edellytys mm. järjestelmän testaamiselle ja se mahdollistaa kehitystyön rinnakkaistamisen.

On syytä muistaa, että tässä kirjassa esitetyt asiat liittyvät suurten ohjelmistojen (*programming-in-the-large*) rakentamisessa hyväksi havaittuihin tekniikoihin. Kokonaisuus, joka koostuu yhdestä tai kahdesta luokasta (vrt. harjoitustyöt), on yleensä helposti hallittavissa eikä niissä välttämättä tule esiin ongelmia, jotka ovat ominaisia suurille systeemeille. Erityisesti on huomattava, että pienet sovellukset

(*programming-in-the-small*) voi tehdä hyvinkin pieni ryhmä (yksi tai kaksi henkilöä), joka hallitsee teknisen ja ongelma-alueen kokonaisuuden. Käytännön työt eivät ole pieniä, joten ohjelmoijille on sovittava omat vastuualueensa, heidän pitää pystyä kommunikoimaan (sekä ohjelma- että mentaalitasolla) keskenään ja heidän on ilmaistava kirjallisesti ne periaatteet, joita he ovat ohjelman muodostamisessa soveltaneet (ts. dokumentointi eri muodoissaan). Sen lisäksi että suuri ohjelma pitää jakaa erillisiin osiin, epäjatkuvuus on myös ajallista: ne, jotka kirjoittivat ohjelman, eivät todennäköisesti tule ylläpitämään sitä. Ohjelmoijan suurin haaste onkin tehdä ylläpitäjien työ helpoksi.

Annetuissa esimerkeissä pyritään havainnollistamaan käyttökelpoisten luokkakokonaisuuksien rakentamiseen liittyviä ongelmia ja niiden ratkaisuja. Tarkoituksena on siis tarjota eväitä myöhemmälle oppimiselle, joka sisältää vielä suurempien kokonaisuuksien — suunnittelumallien (*design patterns*), komponenttien (*components*) ja sovellusrunkojen (*frameworks*) — rakentamista ja käyttöä.

Kädessäsi oleva oppimateriaali nojautuu joulukuussa 2006 julkaistuun standardiin "Java 2 Platform Standard Edition Version 6.0". Sitä edeltänyt, syyskuussa 2004 julkaistu 5.0-versio toi suuria muutoksia Java-kieleen, mikä on otettu huomioon sekä tekstissä että esimerkeissä. Kielen uudistumisen lisäksi muutosta on edesauttanut kirjoittajien näkemyksen tarkentuminen (tai muuttuminen) vuosien kuluessa. On silti ollut mielenkiintoista huomata, kuinka ajan kuluessa Java-kieleen on vähitellen otettu mukaan yhä enemmän piirteitä, joista puhuttiin jo tämän opuksen ensimmäisissä laitoksissa.

Historia ja kiitokset

Tämä oppimateriaali on kokenut vuosien varrella laajoja(kin) uudistuksia. Alkuperäisen *Ohjelmointi II Java-kielellä* -luentomonisteen kirjoittaja Timo Raita poistui keskuudestamme keväällä 2002 jättäen jälkeensä osittain keskeneräisen käsikirjoituksen. Uutta ja nykyistä opetusohjelmaa paremmin vastaava nimeä kantava *Sopimuspohjainen olio-ohjelmointi Java-kielellä* sisältää Timon alkuperäistä tekstiä niiltä osin (eli valtaosin) kun se on yhä ajankohtaista. Olemme koettaneet säilyttää Timon esitystyylin uusissa ja uudistetuissa osioissa, mutta tämä opus esittelee asiat meidän haluamallamme tavalla — ja Timo saattaisi hyvinkin pudistella päätään muutamille tekemillemme valinnoille tai painotuksille.

Koko joukko Turun yliopiston Informaatioteknologian laitoksen henkilökunnan edustajia on aktiivisesti vaikuttanut omalta osaltaan tämän opuksen syntyyn ja kehitykseen. Heistä mainittakoon erityisesti Timo Knuutila, Ville Leppänen, Jukka Teuhola ja Antero Järvi. Harjoitustehtävistä kiitokset kuuluvat myös innokkaille demonstraattoreille, joista tässä mainittakoon (kronologisessa järjestyksessä) Kai Nikulainen, Kari Salonen, Miikka Åsten, Tomi Tervonen, Sami Krappe, Olli Luoma, Juha Kivioja, Sanna Tuohimaa ja Heidi Vähämaa.

Koska kirjoittaminen on inhimillistä toimintaa, tässä laitoksessa epäämättä lymyää muokkaamista vaativia kohtia. Siispä pyydämme kaikkia lukijoita harrastamaan lähdekritiikkiä eikä vain pureksimatta nielemään lukemaansa.

Turussa 23.2.2007,

Jouni Smed Harri Hakonen

Luku 1

Johdanto

Tämä oppimateriaali keskittyy ideoihin, joilla ohjelmistoista saadaan luotettavia ja ylläpidettäviä. Perusajatus sekä rutiini- että luokkatasolla on, että tarkasteltava kokonaisuus on itsenäinen, ts. se sisältää kaiken oleellisen tiedon sitä käyttävälle asiakkaalle; asiakkaan ei tarvitse tutkia muita kokonaisuuteen sidoksissa olevia osia ymmärtääkseen sen toiminnan ja käyttötarkoituksen.

Ensimmäisessä osassa lähdetään liikkeelle olion käyttäytymisen perusyksiköstä, *rutiinista* (luku 2). On ensiarvoisen tärkeää, että yksittäisten piirteiden, olipa kyseessä sitten rutiini tai attribuutti, käyttö on selkeää ja että piirteen merkitys on ymmärrettävissä luokan dokumentoinnista. Rutiinin vastuiden ja oikeuksien jäsentämisessä avustaa *sopimuspohjainen suunnittelu* (*design by contract*), johon kuuluu olennaisena osana rutiinin määrittely. Määrittely kertoo tyhjentävästi, mihin tarkoitukseen rutiini on kirjoitettu ja miten sitä kutsutaan.

Luvussa 3 näkökulmaa laajennetaan *luokkakokonaisuuteen*. Tällöin mietitään mitä asioita tulee ottaa huomioon luokkaa rakennettaessa: *julkisen liitännän* (*public interface*) selkeys ja kattavuus, liitäntään kuuluvien piirteiden saumaton yhteistoiminta, sisäisen esitysmuodon eheys sekä vaikutukset perijäluokkien toimintaan (ja toteutukseen). Yleiskäyttöisen luokan kirjoittaminen on haastava tehtävä; ohjelmoijan on aina pyrittävä tekemään rutiineista ja luokista uudelleenkäytettäviä, mikä edellyttää sitä, että pyritään analysoimaan kaikkia niitä vaatimuksia, joita potentiaaliset asiakkaat voivat esittää.

Luvussa 4 paneudutaan *kokonaisen luokkasysteemin* tarkasteluun. Esimerkkien avulla valaistaan erilaisia suunnitteluprosessiin liittyviä valintoja ja niiden vaikutusta lopputulokseen. Samalla nähdään, kuinka korostunut GUI-ajattelu (*graphical user interface*) johdattelee helposti vääränlaisiin ohjelmistoratkaisuihin. Tämä esimerkki on tärkeä siksi, että Java on suuntautunut voimakkaasti s/t-ohjelmointiin ja erityisesti graafisten käyttöliittymien tekoon. Samassa yhteydessä tulee luon-

nollisella tavalla esiin *suunnittelumallin* (*design pattern*) käsite, josta annetaan esimerkki.

Toisessa osassa tarkastellaan laveammin OO-ohjelmointia. Luku 5 keskittyy syvällisesti *Javan periytymismekanismiin*. Aluksi selvitetään, miten periytyviä piirteitä voidaan muokata ja minkälainen liitäntä periviin luokkiin päin kannattaa kirjoittaa, jotta muutokset perityssä luokassa eivät aiheuttaisi epähaluttuja muutoksia perijöissä. Myös rajapintaluokkien ja abstraktien luokkien merkitystä OO-ohjelmoinnissa ja luokkahierarkian rakentamisessa selvitetään. Kun polymorfisuuden idea on esitetty, keskitytään dynaamiseen sidontaan, erityisesti niihin hieman erikoisiinkin sääntöihin, joilla se Javassa toimii.

Luvussa 6 tarkastellaan Java-olioiden perustoimintoja. Samalla kartoitetaan tilanteita, joissa edellisessä luvussa esiteltyjä periytymismekanismia käytetään (monet näistä tavoista ovat hyvinkin kaukana periytymisen alkuperäisestä ajatuksesta, alityypityksestä).

Luvussa 7 esitellään Javan versiossa 5.0 mukaan tulleita geneerisiä luokkia. Koska kyseessä on kieleen jälkikäteen tehty lisäys, geneeriset tyypit on toteutettu *tyyppitypistyksen* (*type erasure*) avulla, joka poikkeaa tietyin osin muiden ohjelmointikielen käyttämistä toteutuksista.

Luvussa 8 keskitytään Javan *kokoelmaluokkien* (*collections*) muodostaman perintähierarkian esittelyyn. Samassa yhteydessä tutkitaan iteraattoreiden toimintaa sekä kokoelmaluokkien suhdetta geneerisyyteen. Lisäksi tarkastellaan esimerkinomaisesti, miten oma kokoelmatyyppinen luokka kannattaa rakentaa.

Luku 9 syventää *periytymis- ja asiakasrelaatioon liittyvää tietämystä* sekä reaalimaailman mallintamisen että ohjelmasysteemin rakenteen kannalta.

Tästä lyhyestä sisältökuvauksesta lukija lienee jo vetänyt selkeän johtopäätöksen: jotta voisi kutsua itseään ohjelmoijaksi (ammattimielessä), pitää ymmärtää että ohjelmistoja myydään toisilla tekijöillä kuin millä niitä toteutetaan. Järjestelmän käyttäjä näkee pelkästään systeemin ulkoiset tekijät, mutta ne kuvautuvat hyvin harvoin suoraan systeemin sisäisiksi rakenneosiksi. Tämän vuoksi ulkoisten tekijöiden (kuten käyttöliittymän) ei pitäisi antaa johtaa sisäisten rakenneosien (kuten luokkarajapintojen) toteutusta. Esiteltävien ohjelmanrakentamisperiaatteiden päämääränä on saada aikaan oikein toimiva ohjelma, jota on helppo ylläpitää kun mallinnettava reaalimaailma muuttuu — niin kuin aina väistämättä käy...

1.1 Java-kielestä

Vaikka tässä opuksessa esiteltävät asiat sopivat pääosin mihin tahansa oliokieleen, lienee paikallaan tehdä pieni katsaus opetuskielenä käytettävän Javan historiaan. Keväällä 1995 tapahtuneen ensijulkaisun jälkeen Java on kulkenut ohjelmointikielenä eteenpäin. Kielen kehittäjän, Sun Microsystemsin nykyisenä julkaisupolitiikana

versio	julkaistu	luokkia	pakkauksia	erityistä
1.0a	maaliskuu 1995			alfa-versio
1.0	toukokuu 1996	212	8	
1.1	helmikuu 1997	504	23	sisäluokat
1.2	joulukuu 1998	1520	59	Swing, Collections
1.3	toukokuu 2000	1842	76	
1.4	helmikuu 2002	2991	135	**`assert`**
5.0	syyskuu 2004	3279	166	geneerisyys
6.0	joulukuu 2006	3777	202	

Taulukko 1.1: Javan versiohistoria

on julkaista uusi versio (*major version*) noin puolentoista vuoden välein. Versioiden välillä julkaistaan tarpeen mukaan revisioita (*minor version*), joissa paikataan ja täydennetään varsinaisen version virheitä ja puutteita.

Taulukkoon 1.1 on koottu Java-versioiden julkaisuajankohdat, luokkien ja pakkausten lukumäärät ja tärkeimmät muutokset. Tätä kirjoittaessa Javan tuoreimman version täydellinen nimi on ”Java 2 Platform Standard Edition 6.0”. Nimen alkuosa ”Java 2 Platform” otettiin käyttöön versiosta 1.2 lähtien. Keskiosan ”Standard Edition” erottaa perusversion yrityskäyttöön tarkoitetusta järeämmästä ”Enterprise Edition” -versiosta ja mobiililaitteille suunnatusta kevyemmästä ”Micro Edition” -versiosta. Loppuosan versionumeroon 6.0 lienee vaikuttaneet kaupalliset syyt, sillä sisäisesti se tunnetaan versionumerona 1.6.0.

Ennen nykyistä versiota suurimmat muutokset kielen tasolla Java on kokenut, kun versiossa 1.1 esiteltiin sisäluokat (ks. kohta 3.3) ja versiossa 1.4 **`assert`**-mekanismi (ks. kohta 2.2.3). Version 5.0 mukana Java-kieleen on tullut seuraavat uudet piirteet:

- geneeriset tyypit (ks. luku 7)
- paluutyypin kovarianssi (ks. kohta 5.4.1)
- literaalityyppi **`enum`** (ks. kohta 3.3.4)
- kääntäjä tulkitsee `@`-merkillä alkavat sanat annotaatioiksi (*annotations*), joilla ilmaistaan meta-ominaisuuksia ja joita voidaan käsitellä ajoaikaisen reflektoinnin avulla
- primitiivityyppien automaattinen kuorrutus ja kuorinta (*autoboxing/unboxing*), ts. primitiivityyppejä (esim. **`int`** ja **`byte`**) voidaan käyttää vastaavien kuoriluokkien (*wrapper class*) kanssa (esim. `Integer` ja `Byte`); esimerkiksi asetuslauseet `Integer x = 42` ja **`byte`** `b = new Byte("14")` ovat sallittuja

- iteraattorisilmukka (ns. *foreach*-lause); jokaista `Iterable`-rajapinnan mukaisesti käyttäytyvää oliota ja taulukkoja (jotka eivät silti jostain syystä toteuta `Iterable`-rajapintaa...) voidaan läpikäydä **`for`**-silmukassa esimerkiksi

```
public static void main(String[] komentorivi)
  {
  for (String parametri : komentorivi)
    System.out.println(parametri);
  }
```

  jolloin **`for`**-silmukka voitaisiin ajatella luettavaksi "*for each* `parametri` *in* `komentorivi`"

- vaihtelevanmittaiset parametrilistat; metodin viimeisen parametrin tyypin perään kirjoitetut kolme pistettä (esim. **`void`** `järjestä(Integer... luvut)`) tulkitaan taulukoksi (esim. **`void`** `järjestä(Integer[] luvut)`) mutta metodin kutsuja voi luetella parametrit pilkulla erotettuna (esim. `järjestä(4, 2, 6, 1)`)

- avainsanayhdistelmällä **`static import`** voidaan tuoda yksittäisiä luokkapiirteitä (kuten **`import`**-avainsanalla kokonaisia luokkia) luokan käyttöön

Kuten listasta nähdään vain ensimmäiset neljä muutosta tuovat jotain uutta Javakieleen ja loput uudistuksista ovat pelkästään ohjelmoijan elämää (toivottavasti) helpottavaa syntaktista sokeria.

Javan jatkokehitys on tätä kirjoitettaessa mielenkiintoisessa tienhaarassa, sillä Sun julkaisi marraskuussa 2006 sekä Standard Edition- että Micro Edition -implementoinnit GNU General Public -lisenssin (GPL) alla. Vaikka Sun siis yhä hallitsee Java-tuotenimeä ja määrittelee virallisen version kehityssuunnan, saattaa vastaisuudessa olla liikkeellä useita erilaisia (ja mahdollisesti keskenään yhteensopimattomia) Javan kehitysversioita.

1.1.1 Kääntäminen ja ajaminen

Oletuksena Java-kääntäjä toimii version 6.0 mukaisesti, jolloin komentorivi voisi näyttää seuraavalta:

```
javac MunKoodi.java
```

Joissakin tapauksissa kääntäjä saattaa ilmoittaa tarkistamattomista varoituksista (*unchecked warnings*) ja kehoittaa käyttää `-Xlint`-vipua niiden esiinsaamiseksi. Tällöin komentorivi on seuraava:

```
javac -Xlint MunKoodi.java
```

Ohjelmaa ajaessa kannattaa kytkeä **`assert`**-ilmoitukset päälle:

```
java -enableassertions MunKoodi
```

Tämä voidaan lyhentää muotoon:

```
java -ea MunKoodi
```

1.1.2 Javadoc

Java-ohjelmien tekemisen yhteydessä on hyvä samanaikaisesti tutustua ohjelmien tekniseen dokumentointiin tarkoitettuun työkaluun nimeltä Javadoc. Kommenttimerkkien `/**` ja `*/` väliin kirjoitettujen *täkyjen* (*tags*) avulla voidaan dokumentoida luokkien, jäsenmuuttujien ja metodien toimintaa. Taulukon 1.2 yläosassa on kertauksena yleisimmät täkyt[1].

Liittessä A esitellyt tiedostot `common.jd` ja `project.jd` auttavat dokumentoinnissa. Edellinen sisältää yleisiä täkymäärityksiä (kuten `@.pre` ja `@.post`, ks. taulukko 1.2) ja jälkimmäistä voi muokata ja uudelleennimetä käsillä olevan projektiin sopivaksi lisäämällä sinne ne luokkien lähdekooditiedostot, jotka halutaan mukaan Javadoc-dokumentin. Tämän jälkeen Javadoc-dokumentit generoidaan komentorivillä

```
javadoc @common.jd @project.jd
```

1.2 Käytetyistä merkinnöistä

1.2.1 Määrittelymerkinnät

Määrittelyissä käytetään pääsääntöisesti Javan loogisia operaattoreita, jotka on koottu taulukkoon 1.3. Oikosulkevia operaattoreita käytetään, kun halutaan korostaa operaattoreiden käsittelyjärjestystä, ts. merkintä p `&&` q mahdollistaa sen, että mikäli ehto p on epätosi, ehtoa q ei tarvitse evaluoida. Esimerkiksi ehdon

```
joukko != null && joukko.onTyhjä()
```

jälkimmäistä osaa ei voi evaludoida mikäli ensimmäinen ei ole voimassa, mutta ne voidaan silti kirjoittaa samaan lauseeseen oikosulkevaa operaattorin avulla.

Koska Java-kielen loogiset ilmaisut eivät riitä aivan kaikkeen, mitä tällä kurssilla tarvitaan, laajennetaan niitä seuraavalla kuudella apumerkinnällä:

Implikaatio Implikaatiota `==>` käytetään ilmaisemaan riittävää tai välttämätöntä edellytystä, ts. lauseen p `==>` q ehto p on ehdon q riittävä edellytys ja ehto q

[1] Lisätietoa löytyy sivulta ⟨http://java.sun.com/javase/6/docs/technotes/guides/javadoc/⟩.

`@author`	tekijä
`@version`	versio
`@since`	mukana versiosta lähtien
`@throws`	poikkeuksen esittely
`@param`	parametrin esittely
`@return`	paluuarvon esittely
`@see`	ristiviittaus
`@.pre`	alkuehto
`@.post`	loppuehto
`@.postProtected`	perijälle suunnattu loppuehto
`@.postPrivate`	privaatti loppuehto
`@.classInvariant`	luokkainvariantti
`@.classInvariantProtected`	perijälle suunnattu luokkainvariantti
`@.classInvariantPrivate`	privaatti luokkainvariantti
`@.abstractionFunction`	abstraktiofunktio
`@.time`	aikakompleksisuus
`@.space`	tilakompleksisuus
`@.correspondence`	tekijän yhteystiedot
`@.download`	lähdekoodilinkki
`@.toDo`	keskeneräinen

Taulukko 1.2: Javadocin perustäkyt sekä kirjoittajien `common.jd`-tiedostossa esittelemät lisätäkyt.

operaatio	merkintä
negaatio	`!`
konjunktio	`&`
disjunktio	`|`
poissulkeva disjunktio	`^`
oikosulkeva konjuktio	`&&`
oikosulkeva disjuktio	`||`

Taulukko 1.3: Javan loogiset operaattorit.

on ehdon p välttämätön edellytys. Implikaatio voidaan kirjoittaa auki negaation ja disjunktion avulla

$$p \texttt{ ==> } q =_{\text{def}} (\texttt{!}p) \texttt{ | } q$$

mutta implikaation käyttö on useimmiten selvempää.

Ekvivalenssi Ekvivalenssi `<==>` on tosi, jos ja vain jos ehtojen totuusarvot ovat samat. Toisin sanoen ekvivalenssi voitaisiin ilmaista seuraavasti käyttäen negaatiota ja poissulkevaa disjunktiota:

$$p \texttt{ <==> } q =_{\text{def}} \texttt{!}(p \texttt{ ^ } q).$$

Universaalikvanttori Kun halutaan ilmaista että kaikille taulukon tai kokoelmaluokan alkioille tai tietyille muuttujan arvoille pätee jokin totuusarvoinen lauseke, käytetään universaalikvanttoria `FORALL`. Yleinen muoto on

`FORALL(`*alkio* `:` *kokoelma*`;` *totuusarvolauseke*`)`

tai

`FORALL(`*muuttuja* `:` *muuttujan totuusehto*`;` *totuusarvolauseke*`)`

Esimerkiksi alkuehto

```
/**
 *  @.pre  FORALL(mj : lauma; mj.equals("Cow"))
 *  @.post true
 */
public void muut(String[] lauma)
```

vaatii että kaikkien rutiinin parametrina annetun taulukon merkkijonoalkioiden arvona on `"Cow"`. Vaihtoehtoisesti sama voitaisiin ilmaista viittaamalla taulukon indekseihin:

```
/**
 *  @.pre  FORALL(i : 0 <= i < lauma.length; lauma[i].equals("Cow"))
 *  @.post true
 */
public void muut(String[] lauma)
```

Kuten esimerkeistä nähdään, muuttujan tyyppimäärittely voidaan jättää pois, mikäli se on ilmeinen käyttökontekstista. Samoin esimerkiksi indeksirajojen määrittelyä voidaan suoraaviivaistaa (ts. edellisessä esimerkissä rajat olisi voitu kirjoittaa ”ohjelmakoodimaisemmin” `0 <= i & i < lauma.length`).

Kannattaa huomata, että jos kokoelma on tyhjä tai muuttujan totuusehto ei ole voimassa, `FORALL` tulkitaan todeksi.

Eksistenssikvanttori Kun halutaan ilmaista että taulukossa tai kokoelmaluokassa on alkio tai tietyllä muuttujalla on arvo, jolle pätee jokin totuusarvoinen lauseke, käytetään eksistenssikvanttoria `EXISTS`. Yleinen muoto on

`EXISTS(`*alkio* `:` *kokoelma*`;` *totuusarvolauseke*`)`

tai

`EXISTS(`*muuttuja* `:` *muuttujan totuusehto*`;` *totuusarvolauseke*`)`

Esimerkiksi alkuehto

```
/**
 *  @.pre  EXISTS(mj : parvi; mj.equals("Chicken"))
 *  @.post true
 */
public void kotkot(String[] parvi)
```

vaatii että rutiinin parametrina annetussa taulukossa on (ainakin) yksi merkkijonoalkio, jonka arvo on `"Chicken"`. Vaihtoehtoisesti sama voitaisiin ilmaista viittaamalla taulukon indekseihin:

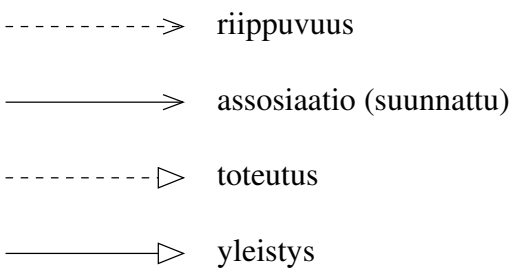
```
/**
 *  @.pre  EXISTS(i : 0 <= i < parvi.length;
 *                            parvi[i].equals("Chicken"))
 *  @.post true
 */
public void kotkot(String[] parvi)
```

Kannattaa huomata, että jos kokoelma on tyhjä tai muuttujan totuusehto ei ole voimassa, `EXISTS` tulkitaan epätodeksi.

Arvo ennen rutiinikutsua Kun halutaan viitata parametrin tai luokkamuuttujan alkuperäiseen arvoon ennen rutiinikutsua, käytetään funktiota `OLD`. Käyttöyhteydestä riippuen arvo voi tarkoittaa ko. muuttujan arvoa tai joskus myös viittauksen päässä olevan objektin arvoa. Esimerkiksi loppuehto

```
/**
 *  @.pre  true
 *  @.post this.equals(OLD(this))
 */
public void konservoi()
```

vaatii että rutiini ei muuta **`this`**-olion arvoa (ts. ei aiheuta arvoon vaikuttavia sivuvaikutuksia).



Kuva 1.1: Relaatioiden piirrosnotaatiot: riippuvuus (*dependency*), assosiaatio (*association*), toteutus (*realization*) ja yleistys (*generalization*).

Rutiinin paluuarvo Kun halutaan viitata rutiinin paluuarvoon, käytetään ilmausta `RESULT`. Esimerkiksi loppuehto

```
/**
 *  @.pre  t != null
 *  @.post RESULT.length == t.length &
 *         FORALL(i : 0 <= i < t.length;
 *                           RESULT[i] == t[(t.length - 1) - i])
 */
public int[] käännä(int[] t)
```

vaatii että paluuarvona annettava taulukko on yhtä pitkä kuin parametrina annettu taulukko ja että paluuarvotaulukossa alkiot ovat käänteisessä järjestyksessä.

1.2.2 Piirrosmerkinnät

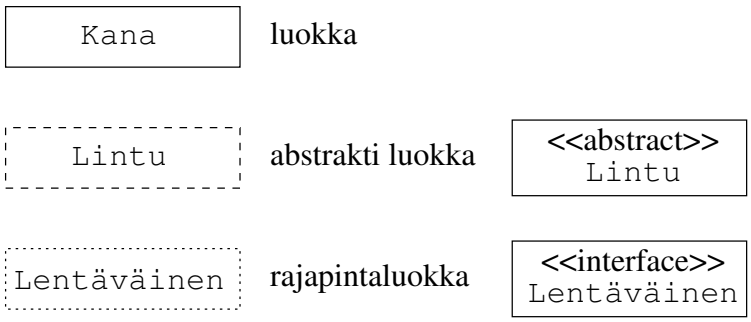
Luokkien ja niiden välisten suhteiden havainnollistamisessa käytetään pääosin Unified Modeling Language -mallinnuskielen (UML) mukaisia perusmerkintöjä. Relaatioihin liittyvät piirrosnotaatiot on koottu kuvaan 1.1 ja luokkiin liittyvät piirrosnotaatiot kuvaan 1.2.

Tehtäviä

1-1 Varmista että löydät tuoreimman version Javan API-dokumentaatiosta. Etsi sieltä pakkaukset `java.lang` ja `java.util` ja tutustu niissä oleviin luokkiin.

1-2 Laadi ohjelma, joka tulosta luvun n kertotaulun kertojille $[1, 10]$. Kerrottava n annetaan komentoriviparametrina. Käännä ja aja ohjelma.

1-3 Tutustu tiedostoihin `common.jd` ja `project.jd`. Kommentoi tehtävässä 1-2 laatimasi ohjelma käyttäen sopivia Javadoc-täkyjä. Generoi dokumentaatio Javadoc-työkalun kanssa.



Kuva 1.2: Luokkien ja rajapintojen piirrosnotaatiot. Vasemmalla on tässä materiaalissa käytetyt lyhennysmerkinnät ja oikealla UML-standardin mukaiset piirrosnotaatiot abstraktille luokalle ja rajapintaluokalle.

1-4 Miten ilmaisisit seuraavat väitteet käyttäen kvanttoreita `FORALL` ja `EXISTS`?

(a) Kaikki kokonaislukutaulukon `kiva` alkiot ovat positiivisia.

(b) Merkkijonossa `syöte` esiintyy merkki `'k'`.

(c) Kokonaislukutaulukon `lottorivi` minimialkiolla ei ole duplikaatteja (so. se esiintyy taulukossa vain kerran).

Osa I

Luokkapohjaisuus

Luku 2

Rutiinin muodostaminen

Tässä luvussa esitellään käytännön ideoita hyvien rutiinien kirjoittamiseen. Keskeisenä teemana on ns. *sopimuspohjainen ohjelmointi*, joka on käytännössä osoittautunut erittäin toimivaksi konseptiksi virheiden välttämisessä, havaitsemisessa, paikallistamisessa ja korjaamisessa. Tarkoituksena on antaa selkeitä ja konkreettisia ohjeita siitä, miten kukin voi omalta osaltaan pyrkiä parantamaan kirjoittamiensa ohjelmien laatua. Esiteltävät tekniikat eivät ole kieliriippuvia, joten niitä voi ainakin jonkinasteisesti soveltaa kaikkiin ohjelmointikieliin. Jos siis käytät jotain muuta kieltä kuin Javaa, ei syytä huoleen, sillä itse asiassa on tärkeämpää sisäistää (ja soveltaa) esiteltävät periaatteet kuin kirjoittaa niitä heijastavaa koodia. Jo pelkkä ajattelutavan muutos parantaa ohjelmiston rakennetta.

Perusedellytys ohjelmistosysteemin laadukkuulle on, että ohjelmistokokonaisuus mietitään ja analysoidaan ensin kaikessa rauhassa. Tätä jatketaan, kunnes ohjelma *näyttäisi* toimivan oikein. Vain noviisit käyttävät tekniikkaa, jossa ohjelmaa aloitetaan kirjoittaa suoraa päätä, minkä jälkeen se käännetään, korjataan käännösvirheet ja suoritetaan, havaitaan loogiset virheet, korjataan ja kokeillaan taas. Vaikka gurut saattavat käyttää samaa tekniikkaa, heillä on etuna se, että he tuntevat sovellusalueensa vähintään 15 vuoden ajalta ja osaavat sen takia varoa pahimpia karikoita ohjelmiston toteutuksessa. Mutta gurukin tietää, että tilanne pitää analysoida huolellisesti ennen toteutusta, kun siirrytään tutulta sovellusalueelta oudompaan maastoon.

Javan ja itse asiassa koko oliosuuntautuneen ohjelmoinnin ideologiana on, että ohjelmoijaa ei päästetä näin helpolla, vaan häneltä edellytetään ammattitaitoisempaa otetta työhönsä: ennen koodin kirjoittamista hän miettii tarkkaan luokkien ja niissä olevien rutiinien keskinäiset roolit, niiden rajaamisen, yleisyyden ja helppokäyttöisyyden. Tämän jäsennystyön tulos tulisi, tavalla tai toisella, *ilmaista eksplisiittisesti ohjelmakoodissa*. Tämän monisteen kantavana perusajatuksena on-

kin eri ohjelmakokonaisuuksien systemaattinen dokumentointi tavalla, joka tukee ohjelmointityötä kaikissa tilanteissa.

Hyvän ohjelman tunnistaa paitsi siitä, että se toimii oikein ja on ymmärrettävä, myös siitä, että se on *vankka* (*robust*) siinä mielessä, että jos eteen tulee odottamaton tilanne, ohjelma pystyy paremmin ilmoittamaan, missä ongelma on tapahtunut ja mahdollisesti myös itse toipumaan siitä. Erikoistilanteet hoidetaan tyypillisesti niin, että kutsuttu rutiini ilmaisee asian kutsujalle normaalista poikkeavalla tavalla, kutsuja tulkitsee tilanteen ja suorittaa tarvittavat toimenpiteet. Erilaisia tapoja ja niiden etuja/haittoja tarkastellaan luvun lopussa. Samassa yhteydessä tutkitaan Javan poikkeustenkäsittelymekanismia.

2.1 Rutiinin määrittely

Määrittelyllä (*specification*) tarkoitetaan sellaista ohjelmaan kirjoitettua tekstiä, joka pyrkii abstraktisti ja kompaktisti kuvaamaan jonkin ohjelmakomponentin toimintaa ja merkitystä. Määrittelyistä saatavat hyödyt ovat moninaiset — ja niistä lisää kohta — mutta määrittelyjen ensisijainen tarkoitus on parantaa ohjelmiston luotettavuutta. Maailmalta löytyy nimittäin paljon esimerkkejä, joissa iso projekti on kaatunut vain sen takia, että systeemiin osana kuulunut ohjelmisto ei ole toiminut toivotulla tavalla.

Esimerkki 2.1 ESA (European Space Agency) kertoi vuonna 1996 kantoraketista Ariane 5, joka jouduttiin tuhoamaan vain 40 sekuntia maasta lähdön jälkeen; välittömät kustannukset olivat noin 500 miljoonaa euroa, suunnittelukustannuksineen vahinko nousi 7 miljardiin euroon. Syynä oli ohjelmistopoikkeus, jota ei käsitelty (koska ohjelmoija oli ollut sitä mieltä, että käsittelijät hidastavat ohjelman suoritusta!). Ongelma olisi havaittu ajoissa, jos toteutuksessa olisi käytetty sopimuspohjaista suunnittelua.

Esimerkki 2.2 Marsin ilmastoa tutkimaan lähetetty Mars Climate Orbiter -luotain tuhoutui vuonna 1999 aiheuttaen noin 350 miljoonan euron menetykset. Syynä oli sekaannus mittayksiköiden muunnoksessa, sillä rakettien työntövoiman muunnos newtoneiksi sisälsi virheen, minkä vuoksi luotain lensi liian lähelle Marsin pintaa. Taustalla oli siis riittävän dokumentoinnin puuttuminen ohjelmasta.

Vastaavanlaisia suuria vahinkoja löytyy myös ”maanläheisemmistä” paikoista kuten esimerkiksi pankeista (joskaan niistä ei yleensä puhuta suureen ääneen).

Jotta ohjelmavirheiltä vältyttäisiin (tai ainakin niitä voitaisiin vähentää), pitää ohjelmaan kirjoittaa informaatiota, joka auttaa ylläpitäjää työssään, erityisesti ymmärtämään, mitä muut ovat aiemmin tehneet. Tämän informaation pitää olla yksikäsitteistä, ymmärrettävää sekä mielellään vakiomuotoista ja sen käsittelyyn tulee olla apuvälineitä (esimerkiksi Javadoc on tällainen työkalu). Määrittelyjen käyttö ja niistä saatava hyöty pohjautuu tutkimukseen, jota on alunperin tehty

kehitettäessä menetelmiä ohjelmien oikeaksi todistamiseen. Oikeaksi todistaminen on vielä melko harvinaista käytännössä, mutta tarkasteltaessa ohjelmointikielten kehitystä on selvästi nähtävissä, että tutkimuksen kautta saadut ohjenuorat hyvien ohjelmien tekemiseen integroituvat vähitellen myös toteutuskieliin. Tämä ilmenee parhaiten ehkä siinä, että ohjelmoija joutuu kirjaamaan ylös yhä enemmän niistä ajatuksista, joihin toteutus nojautuu, myös itse ohjelmakoodiin.

2.1.1 Signatuuri ja toiminnan kuvaus

Rutiinin määrittely on ohjelman ehkäpä tärkein sanallinen kuvaus, joten se kannattaa tehdä huolellisesti. Rutiinin määrittelyn merkitys korostuu siksi, että ohjelmoijalla on tapana kadottaa informaatiota työtä tehdessään. Työtehtävän kuvaus voidaan antaa suullisesti tai se saattaa olla kirjoitettu tarveanalyysin ja toteutuksen suunnittelun pohjalta systeemiä kuvaavaan dokumenttiin. Tämän nojalla ohjelmoija valitsee rutiinille ratkaisun ja toteuttaa sen. Tehtävänantoa ei usein kuitenkaan dokumentoida rutiinin yhteyteen, vaikka toteutukseen kirjoitettaisiinkin joitakin toimintaa kuvaavia kommentteja. Mitä tämän tiedon kadottamisesta seuraa? Ohjelmaa myöhemmin korjaavan tai täydentävän henkilön täytyy lukea ja ymmärtää toteutus ja vetää siitä johtopäätökset rutiinin alkuperäisestä tehtävästä. Tämä voi olla hankalaa, sillä annetulle tehtävälle on tavallisesti lukuisa joukko erilaisia mahdollisia toteutuksia (tehtävän ja ratkaisun välillä on yksi-moneen -relaatio). Jotta ohjelmiston ylläpitäjä tai luokan uudelleenkäyttäjä ei joutuisi näin kohtuuttoman tehtävän eteen, kannattaa rutiinin yhteyteen kirjoittaa aina sen abstrakti kuvaus.

Rutiinin määrittely koostuu: *signatuurista* (*signature*) ja *toiminnan kuvauksesta*:

Signatuuri Kertoo rutiinin nimen, nostettavat poikkeukset sekä parametrien ja palautettavien tietojen määrän, järjestyksen ja tyypit.[1] Rutiinin abstraktissa kuvauksessa signatuurin tärkein informaatio on parametrien ja tulostietojen tyypit ja merkitys, sillä ne ovat kutsujan kannalta oleellisimpia.

Toiminnan kuvaus Tapa, jolla lukijalle halutaan antaa abstrakti (toteutuksesta riippumaton) esitys siitä, mitä rutiini tekee. Usein käytäntönä on liittää otsikkoon lyhyt yleisluonteinen kommentti, joka kertoo selkeästi ja kompaktisti, mihin tarkoitukseen rutiini on kirjoitettu. Toiminta voidaan kuvata tarkemmin kirjoittamalla *alkuehto* (*precondition*) ja *loppuehto* (*postcondition*), jotka ovat totuusarvoisia lausekkeita. Alkuehto kertoo, minkälaisten ehtojen on oltava voimassa, jotta rutiinia voi kutsua. Loppuehto taas ilmaisee ne

[1] Tämä on yleinen määritelmä, Javassa palautustiedon tyyppi ei kuulu signatuuriin ja poikkeustenkin rooli on epämääräinen.

muutokset, jotka rutiinin suoritus saa aikaan (itse asiassa se on siis hyvin lähellä em. yleiskommenttia, joskaan ei yleensä identtinen).

Esimerkki 2.3 Neliöjuurifunktion määrittely voisi olla muotoa

```
/**
 *  Palauttaa x:n neliöjuuren.
 *  @.pre  x >= 0
 *  @.post Math.abs(RESULT * RESULT - x) < 1.0e-10.0 &
 *         RESULT >= 0.0
 */
public static double neliöjuuri(double x)
```

missä otsakerivi kertoo rutiinin signatuurin ja kommentti sisältää toiminnan kuvauksen.

2.1.2 Merkitys ohjelmointiprosessin kannalta

Määrittelyn esitysmuoto vaikuttaa suuresti työn onnistumiseen ja siitä saatavien hyötyjen arviointiin. Esityksen tulisi (a) käyttää notaatiota, jota sekä asiakas että toimittaja ymmärtävät ja (b) olla yksikäsitteinen. Edellisestä johtuen kuvaus kirjoitetaan yleensä luonnollisella kielellä, jolloin jälkimmäinen kohta ei ole voimassa. Tämä tietysti antaa projektin vastuuhenkilölle mahdollisuuden väistää velvollisuuksia kertomalla, että ”näin minä sen ymmärsin”. Käyttämällä formaalia määrittelykieltä (Z, VDM, Larch, CSP, Petri-verkot jne.) voidaan toiminnat esittää eksaktisti ja ilman tulkintaongelmia. Kaupalliset ohjelmistotalot ovat kuitenkin vältelleet tätä lähestymistapaa, vaikka sen on todettu antavan joissakin tarkkuutta vaativissa (erityisesti reaaliaikaiseen ohjelmointiin liittyvissä) projekteissa erinomaisia tuloksia. Syinä vastustukseen ovat mm. seuraavat seikat [9]:

- Projektijohtajat ovat yleensä melko konservatiivisia ja haluttomia ottamaan käyttöön tekniikoita, joista saatava hyöty (= säästetty raha) ei ole ilmeistä lyhyellä tähtäimellä.
- Ohjelmoijat eivät ole saaneet koulutusta määrittelykielen käyttöön. Koulutus maksaa ja vie ohjelmoijat pois tuottavasta (?) työstä. Formaalin kielen oppiminen edellyttää hyvää diskreetin matematiikan ja logiikan tuntemusta. Joten: ”koska yhtiömme ohjelmoijilla ei ole riittävää, aiemmin hankittua peruskoulutusta tähän, ei siihen ryhdytä nytkään (!)”. Määrittelykieli on kuitenkin vain formaali väline ilmaista jokin toiminto, *aivan kuin ohjelmointikielikin*, joten molempien oppiminen on tarkalleen yhtä vaikeaa. Tosin joistakin määrittelykielistä puuttuvat temporaaliset aspektit, minkä takia asiat on helpompi esittää niiden avulla.
- Ohjelmiston tilaaja ei tunne formaaleja tekniikoita ja on siksi haluton rahoittamaan asioita, joista ei tiedä riittävästi.

- Joitakin ohjelmisto-osia, esimerkiksi s/t-toimintoja, rinnakkaisprosesseja ja ajoitusvaatimuksia voi olla vaikea kuvata joillakin määrittelykielillä.
- Määrittelykieliä ja niiden käytöstä saatavia etuja ei tunneta hyvin yritysten IT-osastoilla.
- Määrittelykieliä tukevia ohjelmistotyökaluja ei ole vielä kehitetty riittävästi.

Formaalit määrittelykielet ja ohjelmaan kirjoitettavat sanalliset kommentit edustavat ääripäitä määrittelyjen esittämisessä, ainakin mitä määrittelyjen yksikäsitteisyyteen tulee. Eräänlaisen kompromissin tarjoaa käytetty ohjelmointikieli ja sen tarjoama tuki määrittelyjen esittämiseen. Yksinkertaisimmillaan alku- ja loppuehdot voidaan kirjoittaa perussyntaksilla (esim. ehtolauseena), jolla tarkistetaan ehtojen paikkansapitävyys. Joissakin kielissä (esim. Eiffel [8]) alku- ja loppuehtojen kirjoittamiseen on oma syntaksinsa ja ehtojen tarkistamisia voidaan kontrolloida hyvin monipuolisesti.

Kattavien ja selkeiden määrittelyjen käyttö parantaa kiistatta ohjelmiston laatua. Parhaiten sen merkitys näkyy seuraavissa ohjelmointiprosessin kannalta kriittisissä asioissa:

Kommunikointi Määrittely toimii *kommunikointivälineenä rutiinin implementoijan ja sen käyttäjän välillä*:

$$\textit{Rutiinin käyttäjä} \longleftrightarrow \boxed{\text{Määrittely}} \longleftrightarrow \textit{Rutiinin toteuttaja}$$

Näin *kutsuja ja kutsuttava saadaan riippumattomiksi toisistaan.* Tämän nojalla:

- *Toteutus on muunneltavissa.* Rutiinin toteutusta voidaan mennä muuttamaan vapaasti milloin tahansa (esim. kirjoittaa se eri kielellä) ilman, että kutsuja huomaa mitään. Tämä tietysti edellyttää, että rutiinin uusikin toteutus täyttää määrittelyn.[2]
- *Kutsujan ei tarvitse tuntea salattua tietoa.* Kuka tahansa voi kutsua rutiinia vapaasti, koska kaikki käytettävissä oleva informaatio on määrittelyssä: ei ole olemassa mitään ”salaista”, esimerkiksi implementointialgoritmiin tai suorituksen jälkeisen tilan muistipaikkojen arvoihin liittyvää tietoa, joka kutsujan on tunnettava. Juuri näinhän asian pitääkin olla, koska kutsujan kannalta se *mitä* tehdään, on relevanttia ja se *miten* kyseinen toiminto saadaan aikaan on irrelevanttia.

[2]Tarkasti ottaen uusi toteutus voi tehdä enemmänkin kuin vanha määrittely lupaa; kutsujalla ei kuitenkaan voi olla mitään tätä vastaan (edellyttäen, että ko. lisätoiminto ei ole ristiriidassa vanhan toteutuksen kanssa). Asiasta lisää kohdassa 2.2.

Asiakkaan ja toimittajan erottaminen toisistaan määrittelyn avulla on ehdoton edellytys rutiinien uudelleenkäytölle. Tällöin myös ohjelmaan tehtävien muutosten ja ylläpidon vaikutukset pystytään rajaamaan lokaalimmaksi. Tämän lisäksi implementointia voidaan testata, se voidaan ymmärtää ja kirjoittaa ilman, että tiedetään (toteutukseen liittyviä) yksityiskohtia sen käyttämistä tai sitä käyttävistä rutiineista. Ohjelmistosysteemin jäsentäminen käy myös helpommin, koska kokonaisuus voidaan käydä läpi abstraktio kerrallaan. Lokaalisuudesta johtuen ohjelmistosysteemin toteutus voidaan antaa usealle itsenäiselle, samanaikaisesti toimivalle työryhmälle, mikä on normaali käytäntö nykyaikaisissa ohjelmistoprojekteissa.

Suunnittelu ja dokumentointi Määrittely on oiva *apuväline suunnittelussa ja dokumentoinnissa*. Rutiinien määrittelyt voidaan kirjoittaa luokkiin jo suunnitteluvaiheessa ennen varsinaisia toteutuksia. Luokat ja niiden väliset relaatiot voidaan miettiä valmiiksi pelkkiä määrittelyjä käyttäen. Vasta myöhemmin, kun ohjelmistosysteemi ja sen eri osille asetetut vaatimukset ymmärretään paremmin, valitaan eri olioiden sisäiset esitystavat. Tämän päätöksen viivyttäminen lisää todennäköisyyttä, että kokonaissysteemistä tulee toimiva ja tehokkuusvaatimukset täyttävä. Lisäksi suunnittelija ja toteuttaja — jotka ovat luonnollisestikin eri henkilöitä, koska kyseessä on iso projekti — käyttävät keskenään samaa formalismia (kuvauskieltä), joten ohjelmointiprosessista saadaan *saumaton* (*seamless*). Tämä on erittäin tärkeä näkökohta, sillä tällöin ei synny eri kuvausmenetelmien välisiä epäjatkuvuuksia.

Odotukset Kun rutiinin määrittely kirjoitetaan ennen sen toteutusta esimerkiksi suunnitteluprosessin tuloksena, niin rutiinin *toteuttaja joutuu miettimään, mihin hän sitoutuu*. Menettely pakottaa ottamaan kantaa toteutuksessa mahdollisesti ilmeneviin ristiriitaisuuksiin, erikoistapauksiin ja monimielisyyksiin. Ohjelmoijan on annettava näille kaikille jotkin kutsujan kannalta selkeät ja ilmeiset ratkaisut. Koska ohjelmoija joutuu selvittämään rutiinien käyttäjien vaatimuksia, hän saa paremman kuvan rutiinille asetetuista odotuksista ja osaa siten tehdä toteutuksesta yleiskäyttöisen. Tuloksena saadaan selkeitä ja oikein toimivia ohjelmistokomponentteja, jotka tukevat nykyisiä ja tulevia ohjelmistoprojekteja.

Toimivuus Se, että rutiinin toiminta voidaan kuvata abstraktisti ottamatta kantaa sen toteutukseen, antaa myös mahdollisuuden *osoittaa, että toteutus vastaa määrittelyä*. Ei ole nimittäin mitään mieltä kysyä, toimiiko ohjelma tai sen osa oikein, jollei ole referenssiä, jonka suhteen asiaa tutkitaan. Tämä on tärkeä aspekti esimerkiksi tehtäessä sopimuksia ohjelmiston toimittajan (ohjelmistotalo tms.) ja tilaajan (teollisuusyritys tms.) välillä. Tilaajan ja toimittajan yhteistyönä tehty määrittely on pohja sille, että sopimukseen

kirjattujen ehtojen voidaan todeta tulleen täytetyiksi ohjelman luovutus- ja käyttöönottovaiheessa.

Virheet *Määrittelyt auttavat virheiden paikallistamisessa.* Alku- tai loppuehdon rikkominen paljastaa virheen yleensä melko välittömästi sen tapahduttua, joten virhepaikka on helppo löytää. On selvää, että jos alkuehto ei ole voimassa, kutsuja ei ole tarkistanut sitä (ja siten olettanut alkuehdon olevan seurausta jostain aiemmasta toiminnasta), joten virhe on kutsujassa. Jos sen sijaan loppuehto ei tule voimaan, virhe on rutiinissa itsessään. Kun virhe on tunnistettu, määrittely antaa ”rajat”, jotka on huomioitava korjausta tehtäessä. Käytännön ohjelmointityössä varsinkin alkuehtojen ajoaikainen tarkistaminen on osoittautunut hyvin hyödylliseksi, sillä iäkkäämpi ohjelmakoodi on usein luotettavampaa kuin työn alla oleva.

Luokkakokonaisuus Rutiinien määrittelyt ohjaavat luokkakokonaisuuteen kuuluvien *piirteiden valintaa ja keskinäisiä rooleja.* Määrittelyjen avulla voidaan todeta toisiinsa liittyvien piirteiden saumaton yhteistyö. Täten kokonaisuudesta tulee laadukas: se ei ole vain satunnainen joukko yhteen kerättyjä piirteitä, vaan kullakin on oma hyvin määritelty tehtävänsä kokonaisuudessa. Näin piirrekokoelmasta saadaan suppea mutta kattava.

Periytymismekanismi Alku- ja loppuehtojen (sekä luokkainvarianttien) avulla *periytyminen ja erityisesti rutiinien korvaaminen* (*overriding*) voidaan hoitaa oikein. Tämä on seikka, jota ei juurikaan mainita alan oppikirjoissa, mutta joka on ehdoton edellytys sille, että polymorfismi ja dynaaminen sidonta toimivat halutusti. Perijäluokan korvaavan toteutuksen on noudatettava perityn luokan vastaavan rutiinin määrittelyä, koska muuten korvaavaan rutiiniin nojautuvien operaatioiden toiminta menee totaalisesti sekaisin ja koko ajatus hienosta periytymismekanismista murenee.

2.2 Sopimuspohjainen ohjelmointi

Sopimuspohjainen ohjelmointi/suunnittelu (*programming/design by contract*) [8] perustuu siihen ajatukseen, että määrittely muodostaa samanlaisen ”kontrahdin”, joita jokainen tekee normaalielämässäkin toistuvasti.

Esimerkki 2.4 Kun viet kirjeen postiin, oletat, että posti kuljettaa kirjeen perille. Asiakkaana velvollisuutesi on kirjoittaa kuoren päälle osoite ja liimata postimerkki yläkulmaan (kirjekuljetuksen alkuehto). Jos näin on, posti lupaa kuljettaa kirjeen vastaanottajalle (kirjekuljetuksen loppuehto kertoo saamasi hyödyn). Kun asiakas on huolehtinut velvoitteestaan, postin on toimitettava kirje perille (postin velvoite). Toisaalta postihenkilökunta tietää, että jos kirjeessä ei ole postimerkkiä, kirjeelle ei tarvitse tehdä mitään,

	Velvoitteet	*Hyödyt*
Asiakas	Rutiinin alkuehdon on oltava voimassa ennen kutsua.	Rutiinin suorituksen päättyessä loppuehto on voimassa.
Toteuttaja	Toteutuksen on saatettava loppuehto voimaan.	Alkuehdon karsimia tilanteita ei tarvitse käsitellä rutiinin rungossa.

Taulukko 2.1: Sopimuspohjaisuus asiakkaan ja toimittajan kannalta.

vaan periaatteessa sen voi heittää vaikka roskiin (postin hyöty): kontrahdin rikkovaan tapaukseen voi suhtautua haluamallaan tavalla. Muut ehdot, kuten se että posteljooni tuntee katujen nimet, liittyvät postilaitoksen sisäiseen toteutukseen, joten niitä ei mainita sopimuksessa.[3]

2.2.1 Asiakkaan ja toteuttajan velvoitteet

Ohjelmointiin sovellettuna velvoitteet ja hyödyt ovat taulukon 2.1 mukaisia. Asiakkaan velvoite liittyy alkuehtoon, toimittajan loppuehtoon. Vastaavasti asiakkaan hyöty löytyy loppuehdosta, toimittajan alkuehdosta. Sopimus kertoo seuraavat asiat:

- Alkuehto sanoo asiakkaalle *kuinka paljon tulee tehdä valmistelua*, jotta tietty tulos (loppuehto) saadaan aikaan. Alkuehdon täyttäminen pitää asiakkaan kannalta olla riittävä velvoite, sillä mitään sopimuksen ulkopuolista tietoa ei voida edellyttää tarvittavan.

- Loppuehto kertoo toimittajalle *kuinka vähän* tulee tehdä, jotta se on hyväksyttävää sopimuksen kannalta. Jos alkuehto ei ole voimassa, rutiinin toiminta voi olla epämääräinen. Se voi palauttaa minkä tahansa arvon, suoritus voi jäädä silmukkaan tms.

Sopimuksen tarkoituksena on tehdä selvä jako asiakkaan ja toimittajan vastuualueista. Tuloksena on yksinkertaisempaa koodia, koska samoja järjestelmän tilaehtoja ei tarkisteta useaan kertaan.

Esimerkki 2.5 Aiemmin esimerkissä 2.3 esitetyn neliöjuurifunktion määrittelystä nähdään, että rutiini ei ole valmistautunut käsittelemään negatiivisia argumentteja (selviää siitäkin, että palautettava arvo ei ole kompleksiluku). Rutiinin asiakkaan kannalta tämä tarkoittaa sitä, että todellisen argumentin arvo pitää tarkistaa ennen kutsua:

[3]Ohjelmassa tällaiset yleiset ehdot lausutaan luokan luokkainvariantissa, joista tarkemmin luvussa 3.

```
if (arvo >= 0.0) juuri = neliöjuuri(arvo);
else // ... tee tarvittava muu toiminto
```

Määrittelyn mukaan `neliöjuuri`-rutiini olettaa asiakkaan hoitavan velvoitteensa *eikä tarkista argumenttia*. Toteutus

```
public static double neliöjuuri(double x)
  {
  if (x < 0.0) // Hoida erikoistilanne...
  else // Käytä neliöjuuren laskevaa algoritmia...
  }
```

on siis vastoin määrittelyä, sillä sopimuksen alkuehdossahan on juuri tehty selväksi, että rutiini ei ota velvollisuudekseen hoitaa negatiivisia argumentteja. Sen sijaan toteutus

```
/**
 *  Palauttaa x:n neliöjuuren.
 *  @.pre  true
 *  @.post Jos x < 0.0 nostaa poikkeuksen IllegalArgumentException
 *         muutoin Math.abs(RESULT * RESULT - x) < 1.0e-10.0 &
 *         RESULT >= 0.0
 */
public static double neliöjuuri(double x)
                                    throws IllegalArgumentException
  {
  if (x < 0.0)
    throw new IllegalArgumentException("neliöjuuri: x = " + x);
  else // Käytä neliöjuuren laskevaa algoritmia...
  }
```

on taas jo järkevä. Ero näiden kahden määrittelyn välillä voi tuntua pieneltä, mutta kutsujan kannalta se ei sitä ole. Jälkimmäisessä tapauksessa alkuehto on **true**, joten asiakas voi kutsua rutiinia tekemättä mitään tarkistuksia. Aiemmin alkuehtona esitetty tilanne onkin nyt siirretty loppuehtoon ja on siten osa rutiinin normaalia käyttäytymistä; kyse on siis kokonaan eri määrittelystä kuin aiemmin. Äkkiseltään tuntuisi, että jälkimmäinen määrittely olisi ”käyttäjäystävällisempi” kuin aiempi, onhan edellisessä tehtävä aina tarkistus ennen kutsua. Asiakas ei tosin pääse jälkimmäisen kutsun yhteydessä yhtään vähemmällä työllä, sillä sen täytyy varautua ottamaan kiinni `IllegalArgumentException`-poikkeus (poikkeusten käyttöä tarkastellaan lähemmin kohdassa 2.4.3).

Alkuehtoa ei ole syytä yrittää saada poistettua kokonaan, vaan se kannattaa valita siten, että se on järkevä ja luonteva rutiinin abstraktin merkityksen kannalta. Esimerkkitapauksemme vaatimus argumentin ei-negatiivisuudesta on varsin kohtuullinen asiakkaalle eritoten siksi, että useissa tilanteissa todellisen argumentin tiedetään täyttävän kyseisen ehdon edeltävän laskennan perusteella. Tällöin tarkistusta ei tarvitse tehdä.

Esimerkki 2.6 Laskettaessa n:n luvun x_i $(i = 1, \ldots, n)$ otoskeskihajontaa kaavalla $\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2/(n-1)}$, missä $\overline{x}$ on lukujen keskiarvo, riittää kirjoittaa

```
suhde = summalauseke / (n - 1);
hajonta = neliöjuuri(suhde);
```

tai jos halutaan muistuttaa lukijalle, että neliöjuurifunktion alkuehdon tiedetään olevan voimassa, kirjoitetaan

```
suhde = summalauseke / (n - 1);
assert suhde >= 0.0 : "Negatiivinen suhde.";
hajonta = neliöjuuri(suhde);
```

Ylläolevaan koodiosaan olisi tietysti mielekästä kirjoittaa myös toinen tarkistus:

```
assert n >= 2 : "Liian pieni otos.";
suhde = summalauseke / (n - 1);
assert suhde >= 0.0 : "Negatiivinen suhde.";
hajonta = neliöjuuri(suhde);
```

Mikäli **assert**-lauseiden ehdot eivät ole voimassa, ohjelman suoritus päättyy poikkeukseen `AssertionError`.

Ohjelman mielivaltaiseen kohtaan sijoitetun väittämän tarkoituksena on selventää koodia. Väittämä toimii ohjelman kehitysvaiheessa tarkistuspisteenä, jolla taataan ehtolausekkeen paikkansapitävyys. Myöhemmin väittämä toimii dokumentointiapuna: ohjelmaa ylläpitävän henkilön ei tarvitse välttämättä lukea edellistä koodiosaa ymmärtääkseen, mitkä muuttujat sisältävät suorituksen kannalta oleellista tilatietoa ja mitkä ehdot sitovat niiden arvoja ko. koodikohdassa. Erityisesti ennen rutiinin kutsua kirjoitettu **assert**-tarkistus kertoo koodin lukijalle: ”alkuehdon testaamatta jättäminen ei ole vahinko, vaan tässä kohtaa koodia alkuehdon pitäisi olla aina voimassa”.

2.2.2 Alisopimus

Muutama sana vielä siitä, mihin systeemin suunnittelija sitoutuu kiinnittäessään rutiinin määrittelyn. Suuri osa määrittelyn avulla saavutetuista hyödyistähän perustuu siihen, että määrittely on koodia abstraktimmalla tasolla oleva kuvaus, joka ei muutu vaikka itse toteutus muuttuisikin. Tilanne ei ole kuitenkaan aivan näin vakava, sillä määrittelyä voi myöhemminkin muuttaa ilman, että rutiinin asiakkaille olisi siitä haittaa (tai että ne edes huomaisivat sitä). Muutos täytyy kuitenkin tehdä kontrolloidusti.

Olkoon `P` rutiinin alku- ja `Q` sen loppuehto. Tällöin rutiinia kuvaa pari ⟨`P`, `Q`⟩. Sanotaan, että määrittely ⟨`P'`, `Q'`⟩ on edellisen *määrittelyn alisopimus* (*subspecification*) silloin ja vain silloin, kun

$$\texttt{(P ==> P')} \ \&\ \texttt{(Q' ==> Q)}.$$

Jos määrittely ⟨`P`, `Q`⟩ korvataan nyt alisopimuksella ⟨`P'`, `Q'`⟩, asiakas ei koe tulleensa mitenkään petetyksi, sillä:

- Asiakas on aina aiemmin tarkistanut, että `P` on voimassa ennen kutsua, joten heikompana myös ehto `P'` on voimassa ja siten uuden rutiinin suoritus on turvallista aloittaa.[4]

- Edut, jotka aiempi toteutus on taannut, ovat edelleen voimassa, koska vahvemmasta ehdosta `Q'` seuraa ehto `Q`.[5]

Esimerkki 2.7 Sivulla 16 esitetyn neliöjuurirutiinin loppuehtoa voidaan myöhemmin muuttaa vaikkapa muotoon

```
/**
 *  Palauttaa x:n neliöjuuren.
 *  @.pre  x >= 0
 *  @.post Math.abs(RESULT * RESULT - x) < 1.0e-30.0 &
 *         RESULT >= 0.0
 */
```

Aiemmilla asiakkailla ei ole mitään sitä vastaan, että rutiinin antama tulos on tarkempi.

Alisopimuksen käsite on tärkeä myös sen vuoksi, että rutiinin korvauksen (*overriding*) tulee perijäluokassa kunnioittaa yliluokassa annetun vastaavan rutiinin määrittelyä. Tämän vaatimuksen nojalla (ja ottaen huomioon, että asiakas voi dynaamisen sidonnan vuoksi kutsua perijäluokan rutiinia tietämättään) perijäluokan määrittelyn pitää olla uudelleentoteutettavaa rutiinia vastaavan määrittelyn alisopimus (tästä lisää kohdassa 5.5.1).

2.2.3 Sopimuspohjainen ohjelmointi Javalla

Java-kieli tukee määrittelyjen kirjoittamista melko huonosti. Kuten edellä on nähty, määrittely on yleensä selkeintä kirjoittaa kommentiksi käyttäen Javadoc-työkalun täkyjä. Sekä määrittely että toteutus muodostavat tässä esitystavassa omat selkeät lohkonsa. Erillinen määrittelylohko korostaa myös osien rooleja: määrittelykuvauksen on tarkoitus abstrahoida (olla korkeammalla tasolla kuin) itse koodi. Esitystavan ongelmana on kuitenkin se, että määrittely voi (normaalin kommentin tavoin) unohtua päivittämättä, kun rutiinin merkitys muuttuu.

Varsinkin ohjelman kehitysvaiheessa olisi tärkeää, että alku- ja loppuehdot voitaisiin myös tarkistaa ajon aikana esimerkiksi **`assert`**-mekanismin avulla. Tämä voidaan tehdä kirjoittamalla väittämät rutiinin runkoon.

Esimerkki 2.8 Tutun ja turvallisen neliöjuurirutiinimme toteutus voitaisiin kirjoittaa seuraavaan muotoon:

[4] Uudet ehdot voidaan ajatella liitetyn rutiinin aiempiin alkuehtoihin `||`-operaatiolla.
[5] Uudet ehdot voidaan ajatella liitetyn rutiinin aiempiin loppuehtoihin `&&`-operaatiolla.

```
/**
 *  Palauttaa x:n neliöjuuren.
 *  @.pre  x >= 0
 *  @.post Math.abs(RESULT * RESULT - x) < 1.0e-10.0 &
 *         RESULT >= 0.0
 */
public static double neliöjuuri(double x)
  {
  //-- Alkuehto
  assert x >= 0.0 : "Alkuehtorikkomus";
  //-- Totetus
  double tulos;
  // Lasketaan neliöjuuren arvo muuttujaan tulos...
  //-- Loppuehto
  assert tulos >= 0.0 : "Loppuehtorikkomus";
  assert Math.abs(tulos * tulos - x) < 1.0e-10.0 :
         "Loppuehtorikkomus";
  //-- Paluuarvo
  return tulos;
  }
```

Koska **assert**-väittämän rikkomus nostaa `AssertionError`-virheen, se aiheuttaa käsittelemättömänä ohjelman suorituksen päättymisen. Tämä on järkevää, koska kyseessä on yleensä virhetilanne, josta ohjelma ei pysty omin avuin toipumaan.

Vaikka **assert**-tarkistukset voidaankin kytkeä päälle ja pois päältä ajoympäristölle annettavilla optioilla, on mekanismi joustamaton siinä mielessä, että se kohtelee kaikkia väittämiä samanarvoisina. Usein on kuitenkin mielekästä valita tarkistettavat ehdot selektiivisesti väittämätyypin mukaan (tarkistetaan vain esim. alkuehdot). Tähän päästään ohjelmoijien jo kauan tuntemalla tempulla: kirjoitetaan kukin väittämä ehtolausekkeeseen, jota kontrolloi globaali totuusarvoinen muuttuja.

Esimerkki 2.9 Vartioituja ehtolausekkeita käyttäen esimerkin 2.8 toteutus voitaisiin uudelleenkirjoittaa muotoon

```
public static double neliöjuuri(double x) throws
                     AlkuehtorikkomusVirhe, LoppuehtorikkomusVirhe
  {
  //-- Alkuehto
  if (Väittämäkontrolli.ALKUEHTO)
    if (x < 0.0) throw new AlkuehtorikkomusVirhe();
  //-- Toteutus
  double tulos;
  // Lasketaan neliöjuuren arvo muuttujaan tulos...
```

```
//-- Loppuehto
if (Väittämäkontrolli.LOPPUEHTO)
  {
  if (tulos < 0.0) throw new LoppuehtorikkomusVirhe();
  if (Math.abs(tulos * tulos - x) >= 1.0e-10.0)
    throw new LoppuehtorikkomusVirhe();
  }
}
```

`AlkuehtorikkomusVirhe` ja `LoppuehtorikkomusVirhe` periytyvät ajoaikaisen suorituskoneiston virhettä edustavasta `Error`-luokasta. Rajapinnalla `Väittämäkontrolli` on hallitaan sitä, mitkä väittämät tarkistetaan:

```
public interface Väittämäkontrolli
  {
  public static final boolean
    ALKUEHTO            = true,
    LOPPUEHTO           = false,
    TARKISTUSVÄITTÄMÄ   = true,
    SILMUKKAINVARIANTTI = false,
    LUOKKAINVARIANTTI   = true;
  }
```

Valinta sen suhteen, mitkä väittämät tarkistetaan ajoaikana ja mitkä ei, on syytä tehdä huolellisesti. Mahdollinen (ja käytännössä ehkä usein tavattu) skenaario on, että ohjelman rakentamisvaiheessa kaikki tarkistukset ovat päällä, mutta käyttöönottovaiheessa ne poistetaan, lähinnä tehokkuussyihin vedoten. Tämä on kuitenkin harkitsematonta; eihän lentokapteenikaan poista kaikkia mittareita koneesta opittuaan lentämään. Koska olioajattelu ohjaa (ainakin osittain) rakentamaan ohjelmia ja ohjelmakirjastoja kokoavaan (*bottom-up*) tyyliin, missä aluksi rakennetaan peruspalikat, sitten niiden päälle uutta jne. abstraktiotaso kerrallaan, voidaan alimpien moduulien ajatella toimivan jo niin luotettavasti, että niiden loppuehtoja ei enää ajoaikana tarkisteta. Alkuehdot on kuitenkin syytä tarkistaa aina, sillä uusia asiakkaita mahdollisine virheineen tulee ajan myötä aina lisää.

2.3 Määrittelyjen kirjoittaminen

Määrittelystä saadaan täysi hyöty vain, jos se on *täydellinen* (*complete*) siinä mielessä, että

- rutiinin asiakas ymmärtää sen perusteella mihin tarkoitukseen rutiini on tehty, ja
- se kertoo minkä ehtojen on oltava voimassa, jotta rutiinia voi kutsua.

Aloittelijan on erityisesti syytä kiinnittää huomiota ensimmäiseen kohtaan. Ohjelmoijan käytössä olevaa luokkakirjastoa (esim. Javan API-luokkakirjasto, C++:n STL-kirjasto tai Eiffelin Base-luokat) kannattaa selata ja tutkia ko. kielelle ominaista tapaa antaa valmiiden luokkien määrittelyt. Valitettavasti kuvaukset ovat usein heikkotasoisia, joten niihin on syytä suhtautua kriittisesti. Esimerkiksi Java-kirjastosta löytyy rutiineja, joiden merkitys ei selviä API-dokumentoinnista, vaan asiakkaan on mentävä tutkimaan lähdekoodia. Ja juuri tätä koko määrittelyllä pyritään välttämään! Toteutusta tutkiessaan asiakas tulee helposti tehneeksi päätelmiä, jotka ovat rutiinin tehtävän kannalta epärelevantteja ja käyttäneeksi niitä hyväksi omassa ohjelmassaan. Aiempaan postiesimerkkiin palataksemme, kirjeen lähettäjän ei tarvitse käydä kurkistamassa postin lajittelukeskukseen tai seurata askel askeleelta posteljoonin toimintaa, sillä ne voivat muuttua tai antaa väärän mielikuvan siitä, mikä kirjeen perilleviemisessä on olennaista. Jos asiakkaan ja toimittajan välille syntyy toteutuksen yksityiskohtiin nojautuva sidos, seuraukset näkyvät ikävästi silloin, kun toimittajarutiinin toteutusta joudutaan muuttamaan.

Määrittelyt antavat lukijalle paljon informaatiota rutiinien ja siten myös koko luokan käyttäytymisestä. Listauksessa 2.1 esitelty `Pino` on rajapinta, jossa on esitelty operaatioiden otsikot ja niihin liittyvät määrittelyt. Rajapinnan antama kuvaus on abstrakti siinä mielessä, että se ei ota kantaa varsinaiseen toteutukseen (so. pinon sisäiseen esitysmuotoon). Tästä syystä rajapintaa voidaan käyttää apuna esimerkiksi ohjelmistosysteemin suunnittelussa. Näin tehty suunnitelma voidaan tarkistaa Java-kääntäjällä. Määrittelyt sisältävä rajapinta on myös oiva dokumentti ohjelmiston ylläpitäjälle.

2.3.1 Alkuehto

Rutiinin sanotaan olevan *osittainen* (*partial*), jos sen alkuehdossa on rajoituksia. Muussa tapauksessa rutiini on *totaalinen* (*total*), ts. sen alkuehto on **`true`**. Peruskysymys alkuehtoa valittaessa on, kuinka vahva alkuehdon tulisi olla. Eli tulisiko alkuehtoa pyrkiä heikentämään aina niin pitkälle, että rutiineista tulisi totaalisia? Mietitäänpä tätä asiaa sekä rutiinin toteuttajan että käyttäjän kannalta:

Toteuttajan näkökulma On selvää, että mitä vahvempi alkuehto valitaan, sitä vähemmällä rutiinin implementoija pääsee, koska rajoitusten nojalla hänen pitää käsitellä vain huolella valitut tapaukset. Tiukat rajoitukset mahdollistavat usein myös tehokkaan toteutuksen (vrt. haku lajittelemattomasta tai lajitellusta taulukosta). Alkuehtoa ei saa kuitenkaan tehdä liian tiukaksi, koska se rajoittaa voimakkaasti potentiaalisten asiakkaiden määrää. Erityisen tiukka alkuehto johtaa siihen, että rutiinia on mahdollista kutsua vain siinä ohjelmistosysteemissä, johon se on alunperin kirjoitettu. Tämä ei tietenkään ole mielekästä, koska uudelleenkäyttö on koko OO-ohjelmoinnin kulmakiviä.

Listaus 2.1: Rajapinta `Pino`.

```
/**  Pino on alkiokokoelma, joka noudattaa LIFO-käyttäytymistä. */
public interface Pino<T>
  {
  //-- Havainnointioperaatiot
  /**
   *  Palauttaa alkioiden määrän.
   *  @.pre   true
   *  @.post  RESULT == (pinossa olevien alkioiden lukumäärä)
   */
  public int annaKoko();

  /**
   *  Palauttaa pinon päällimmäisen alkion.
   *  @.pre   !onTyhjä()
   *  @.post  RESULT == (pinon päällimmäinen alkio)
   */
  public T päällimmäinen();

  /**
   *  Palauttaa pinon alimmaiset alkiot.
   *  @.pre   !onTyhjä()
   *  @.post  RESULT == (pinon alimmaiset alkiot pinona)
   */
  public Pino<T> alimmaiset();

  /**
   *  Tarkistaa onko pino tyhjä.
   *  @.pre   true
   *  @.post  RESULT == (annaKoko() == 0)
   */
  public boolean onTyhjä();

  /**
   *  Tarkistaa onko pino täysi.
   *  @.pre   true
   *  @.post  RESULT == (pinoon mahtuu vielä alkioita)
   */
  public boolean onTäysi();
```

Listaus 2.1 (jatkoa): Rajapinta `Pino`.

```
//-- Muutosoperaatiot
/**
 * Lisää alkion pinon päälle.
 * @.pre   !onTäysi()
 * @.post  OLD(this).equals(this.alimmaiset()) &
 *         (päällimmäinen() == alkio)
 */
public void lisää(T alkio);

/**
 * Poistaa pinon päällimmäisen alkion.
 * @.pre   !onTyhjä()
 * @.post  this.equals(OLD(this).alimmaiset())
 */
public void poista();

/**
 * Tyhjentää pinon.
 * @.pre   true
 * @.post  onTyhjä()
 */
public void tyhjennä();
}
```

Siksi rutiinia on syytä tarkastella aina muussakin kuin vain alkuperäisessä, tarkasti rajatussa ympäristössä.

Asiakkaan näkökulma Asiakkaalle totaalinen rutiini on miellyttävä, koska sitä on turvallista kutsua tilanteessa kuin tilanteessa. Mitä tiukempi alkuehto valitaan, sitä enemmän kutsujalla on töitä. Alkuehdon pitää kuitenkin aina olla perusteltavissa itse määrittelyyn nähden niin, että asiakas kokee sen mielekkääksi (vrt. esim. `Pino`-luokan `poista`-rutiinin alkuehto `!onTyhjä()`).

Alkuehdon voimakkuudesta ei voida siis sanoa mitään definitiivistä. Rajoitettuun käyttöön (ts. kun asiakkaat tunnetaan hyvin) tulevat rutiinit muodostuvat usein osittaisiksi, kun taas yleiskäyttöisten kirjastorutiinien käyttöä helpottavat suhteellisen heikot alkuehdot. On kuitenkin osoittautunut, että jälkimmäisessäkin tapauksessa asiakkaalta kannattaa vaatia melko paljon. Tällä tähdätään siihen, että rutiini pyrkisi tekemään vain oman, *tarkasti rajatun* toimensa mahdollisimman hyvin

sen sijaan, että yrittäisi tehdä siitä kaiken kattavan ja miellyttää kaikkia asiakkaita. Jos kuitenkin käy niin, että yksikään asiakas ei pysty kutsumaan rutiinia tekemättä alkuehtotarkistusta (ts. sen voivat vain harvat, jos kukaan, päätellä aiemmasta toiminnastaan), pitäisi ehto poistaa ja siirtää ko. tilanteiden hoito kutsuttavalle rutiinille.

On vielä syytä huomata, että (a) alkuehtoon tehtävät muutokset implikoivat muutoksia myös loppuehtoon, ja (b) se, että kutsuja vapautetaan ehtojen tarkistamiselta, ei silti välttämättä yksinkertaista rutiinia tai sen käyttöä. Toimittajarutiinin toteuttajan on nimittäin ratkaistava ne tavat, joilla asiakkaalle kerrotaan erilaisista erikois- ja ongelmatilanteista. Tästä syystä nostettavien poikkeusten sekä palautettavien arvojen määrä ja tulkinta kasvaa. Se heijastuu kutsun yhteyteen kirjoitettavan koodin lisääntymisenä. Samalla operaation käyttö saattaa muuttua epäselvemmäksi.

Tarkastellaan seuraavaksi minkälaisia rajoituksia alkuehtoon tyypillisesti kirjoitetaan. Ainakin seuraavat ovat hyvin yleisiä:

- *Argumentteihin liittyvät rajoitukset.* Edellä esitellyllä neliöjuurifunktiolla rajoitettiin argumentin `x` arvoaluetta. Mainittakoon erityisesti, että alkuehtoon ei kirjoiteta mitään argumenttien tyyppeihin liittyvää (esim. että funktion `neliöjuuri` todellisen argumentin tulisi olla reaalinen), kääntäjä pitää huolen tyyppiyhteensopivuuksista. Alkuehdon tarkoituksena on ainoastaan rajoittaa tyyppien sallimien tietojen arvoaluetta; `neliöjuuri`-funktion tapauksessa kerrotaan, mitkä **`double`**-tyypin arvot ovat sallittuja.

- *Operaation käyttöön liittyvät rajoitukset.* Pinoluokan `poista`-operaation alkuehtona on `!onTyhjä()`. Ehto liittyy olion tilaan, ei syöttöargumentteihin.

- *Rutiinin toteutukseen liittyvät ehdot.* Tätä ei pidä käsittää niin, että toteutus on tehty hyvin suoraviivaisesti (ohjelmoijan vaivoja säästäen) ja siksi sitä voi kutsua vain tiettyjen ehtojen ollessa voimassa. *Kaikkien alkuehtojen tulee aina olla järkeviä asiakkaan näkökulmasta* eikä toteutus saa vaikuttaa niihin. On kuitenkin tilanteita, joissa toteutus saadaan tehokkaaksi vain, kun argumentit ovat "oikeanmuotoisia". Esimerkiksi haku taulukosta voidaan tehdä nopeasti, jos taulukko on lajiteltu (binäärihaku, interpolaatiohaku tms.). Asiakkaan on helppo ymmärtää tämä vaatimus; toisaalta, rutiinin toteuttajalla on edelleen melko paljon vapautta valitessaan toteutuksen.

Huomaa erityisesti, että *alkuehtoon ei saa kirjoittaa rajoituksia, joita kutsuja ei pysty tarkistamaan.* Postiesimerkissä ehtona ei voi olla, että kirje viedään perille, jos postinkuljetusauto on kunnossa, postinkantaja ei ole flunssassa tai postinkantaja tuntee osoitteen, sillä eihän kirjeen lähettäjä voi tietää postitoimiston sisäisistä asioista mitään. Ohjelmassa tämä tarkoittaa mm. sitä, että alkuehtoon ei voi kirjoittaa olion sisäiseen esitysmuotoon liittyviä ehtoja, jotka vaatisivat esimerkiksi

private-piirteiden käyttöä olion ulkopuolelta. Oikeastaan tämä on hiukan monimutkaisempi asia johon palataan vielä uudestaan. On myös virheellistä pyytää asiakasta (luokassa `A`) tarkistamaan sellaista rutiinin alkuehtoa (luokassa `B`), joka edellyttää sellaisen luokan `C` käyttöä, johon luokalla `B` on pääsy, mutta luokalla `A` ei.

2.3.2 Loppuehto

Ohjelmistosysteemin toteutusta varten tehdään ensin suunnitelma, jossa eri ohjelmistokomponenttien roolit tulevat esille. Suunnitelma dokumentoidaan ja sen pohjalta toteuttajat aloittavat työskentelynsä. Ohjelmoijalla saattaa olla siis melko hyvä kuva rutiinien loppuehdoista tai ne on ainakin johdettavissa kohtuullisella vaivalla ko. dokumentista. Rutiinien nimet, jotka selvästi liittyvät loppuehtoihin, on myös voitu antaa jo aiemmin. Mikäli näin ei ole, ohjelmoijan tehtävänä on määrätä itse luokan julkiseen liitäntään tulevat piirteet ja niiden rajaukset. Erittäin tärkeää on jakaa toiminnot *perustehtäviin* niin, että kukin rutiini tekee vain yhden tarkasti rajatun toimenpiteen.[6]

Esimerkki 2.10 `Tili`-luokkaan ei ole syytä kirjoittaa rutiinia `tilinkäsittely`, jolla hoidetaan sekä otot että panot. Ongelman tunnistaa siitäkin, että rutiinille pitää välittää kontrollitietoa, joka kertoo kummasta tapahtumasta on kyse. Tämän tiedon perusteella rutiinin logiikka jakautuu kahteen erilliseen haaraan. Parempi on tietysti tehdä erilliset rutiinit `otto` ja `pano`, jolloin kutsustakin näkee suoraan kummasta tapahtumasta on kyse. Vältä siis tilanteita, joissa rutiinille välitetään tämäntyyppistä kontrollitietoa.

Loppuehto kuvaa miten rutiini käyttäytyy laillisten kutsujen yhteydessä. Se kertoo tulostietojen merkityksen sekä viittaustyyppisten argumenttien kautta tapahtuvat muutokset (perustyyppiä oleviin argumentteihin ei kutsujalle näkyviä muutoksia voi tehdäkään). Jos rutiinissa voi tapahtua virhetilanteita, joiden seurauksena nostetaan esimerkiksi poikkeus, on kaikki erilaiset lopetustilanteet kerrottava loppuehdossa. Jos rutiini haluaa muuttaa argumenttina annettua oliota eikä se ole kutsujan kannalta mielekästä, on mietittävä kannattaisiko olio kopioida ja päättää sitten annetaanko kopiointi asiakkaan vai toimittajan huoleksi. Tämäkin jo osoittaa sen, että paitsi tapahtuvista muutoksista, usein on yhtä tärkeää kertoa myös *mitä tietoja ei muuteta*. Tätä varten jatkossa otetaan käyttöön seuraava sääntö: *tunnisteisiin, joita ei ole mainittu loppuehdossa, ei ole tehty muutoksia.*

Rutiinin asiakkaan on syytä lukea huolella, mitä loppuehdossa sanotaan ja miettiä pilkun tarkasti sen merkitystä, eikä tehdä mitään ylimääräisiä, loppuehdossa mainitsemattomia johtopäätelmiä rutiinin toiminnasta esimerkiksi piirteen nimen perusteella. Erityisen hyvä testi on asettautua implementoijan asemaan ja

[6] Jos rutiinin nimen keksimisessä on ongelmia, se saattaa olla merkki siitä, että rutiini yrittää tehdä useita perustoimintoja (tai osia niistä).

ajatella minkälaisella toteutuksella pääsisi vähimmällä työllä, sillä niin implementoija vääjäämättä toimii.

Alkuehtojen tavoin myös loppuehtojen vahvuuden merkitystä on syytä tarkastella kriittisesti. Luonnollista kieltä käytettäessä määrittelystä tulee helposti *toiminnallinen* (*operational*), jolloin loppuehto kertoo suoraan rutiinin toteutustavan:

```
/**
 *  @.pre  Tieto alkio esiintyy taulukossa taulu ainakin kerran.
 *  @.post Rutiini tutkii paikkoja taulu[0], taulu[1],...
 *         ja palauttaa ensimmäisen indeksin i, jolle
 *         alkio.equals(taulu[i]).
 */
public static int paikka(Object[] taulu, Object alkio)
```

Kokeneen ohjelmoijan on yleensä helppo tehdä toiminnallinen määritys, mutta ongelmana on, että tällainen määrittely on (a) yleensä melko pitkä (teepä lyhyt ja selkeä toiminnallinen kuvaus neliöjuuren laskevalle ohjelmalle!), ja (b) liian vahva sitoen toteuttajan kädet lähes täysin sekä (c) siihen voi jäädä — luonnollisesta kielestä johtuen — tulkinnanvaraisia ilmauksia. Toiminnallisia kuvauksia kannattaa siis yrittää välttää. Yleensäkin, jos loppuehtoa joudutaan vahvistamaan, se on tehtävä harkiten, koska ehdon vahveneminen on tavallisesti merkki siitä, että rutiinin toteutusvaihtoehtojen määrä pienenee (tuloksena helposti ns. *ylimäärittely*, engl. *overspecification*). Tämän takia on tärkeää tarkkailla, ettei vahventaminen ole johtanut siihen, että rutiinin ratkaisumenetelmä tulee tarkasti kiinnitettyä, vaan että määrittely antaa edelleen mahdollisuuden tehokkaampaan ja/tai elegantimpaan totetukseen. Loppuehdon tiukentaminen tarkoittaa myös lisätyötä toteuttajalle, koska rutiini lupaa tehdä enemmän kuin aiemmin. Jos esimerkiksi lajittelurutiinille annetaan lisävaatimus tehdä hommansa stabiilisti (yhtä suuret alkiot eivät saa muuttaa keskinäisiä paikkojaan alkuperäiseen tilanteeseen verrattuna), on selvää, että työmäärä lisääntyy.

Keskitie on kultainen loppuehdonkin tapauksessa, sillä myös liian heikko (eli *vajaamääritelty*, engl. *underdetermined*) loppuehto on ongelmallinen. Jos rutiinin käyttäytymiseen liittyvistä yksityiskohdista jätetään osa määrittelemättä, jotkin alkuehdon täyttävät kutsut saattavat johtaa useaan valinnaiseen (mutta oikeaan) tulostietoon. Implementointi takaa tällöin vain sen, että palautettava arvo kuuluu kyseiseen tulostietojoukkoon. Esimerkiksi hakurutiini

```
/**
 *  @.pre  taulu != null && EXISTS(a : taulu; a.equals(alkio))
 *  @.post taulu[RESULT].equals(alkio)
 */
public static int paikka(Object[] taulu, Object alkio)
```

voi olla asiakkaan kannalta liian heikko, koska se ei kerro, mikä indeksiarvo palautetaan, jos haettava tieto esiintyy taulukossa useammin kuin kerran. Toisaalta on muistettava, että tarpeetonta loppuehdon tiukentamista on vältettävä: määrittelyn tulee olla *riittävän yleinen* — onhan hakurutiinissa esimerkiksi riippumattomuus taulukon koosta tärkeää — mutta vain, jos se lisää rutiinin käytettävyyttä.

Loppuehtoon kirjoitetaan vain normaaliin toimintaan liittyviä kuvauksia. Erikoistapauksien hoidosta kirjoitetaan oma kuvauksensa, sillä tällöin on kyse jollakin tavoin poikkeuksellisesta tilanteesta:

```
/**
 *  @.pre  taulu != null
 *  @.post taulu[RESULT].equals(alkio) &
 *         FORALL(j : 0 <= j < RESULT; !taulu[i].equals(alkio))
 *  @throws AlkioEiLöydyPoikkeus
 *          Nostetaan jos !EXISTS(a : taulu; a.equals(alkio)).
 */
public static int paikka(Object[] taulu, Object alkio)
                                          throws AlkioEiLöydyException
```

`AlkioEiLöydyPoikkeus` on tarkistettava poikkeus ja kutsujan on valmistauduttava käsittelemään se, jos haettua tietoa ei löydy taulukosta.

Esimerkki 2.11 Hieman erikoisempi loppuehto on `TaulukkoPino`-luokan luontioperaatiolla (oletetaan että luokka perii abstraktin luokan `RajoitettuPino`, ks. listaus 5.1, s. 134):

```
/**
 *  @.pre  maksimikoko >= 0
 *  @.post annaKoko() == 0 &
 *         annaKapasiteetti() == maksimikoko
 *  @.postPrivate pino != null
 */
public TaulukkoPino(int maksimikoko)
  {
  super(maksimikoko);
  pino = new Object[maksimikoko];
  }
```

Loppuehto on jaettu kahteen osaan: Ensimmäinen on suunnattu asiakkaalle ja kertoo abstraktin käsitteen ”pino” ominaisuuksista. Toinen ehto on privaatti ja se on tarkoitettu luokan toteuttajalle ja ylläpitäjille. Tämä loppuehto liittyy pinon sisäiseen toteutukseen eikä näin muodoin ole mitenkään oleellinen asiakkaalle (tällaisessa väittämässä on tyypillisesti **private**-piirteitä, joten siinäkin mielessä se on asiakkaalle irrelevantti). Väittämä on kuitenkin hyvä tarkistaa, jotta pino-olion tiedettäisiin olevan luonnin jälkeen sisäisesti eheä.

Loppuehdossa on tyypillisesti siis kahdentyyppisiä väittämiä:

- *Rutiinin toiminnan vaikutukset asiakkaalle.* Tämä kertoo mihin tarkoitukseen rutiini on tehty ja miten lopputulos näkyy asiakkaalle.[7] Tässä yhteydessä on kerrottava myös mahdollisten erikoistilanteiden syntymis- ja hoitotavoista. Haluttaessa korostaa jonkin tiedon — esimerkiksi viittaustyyppisen argumentin — mutatoitumattomuutta, mainitaan tämä osana loppuehtoa.

- *Olion sisäiseen esitysmuotoon liittyvät ehdot.* Loppuehto voi sisältää osia, joilla ei ole asiakkaalle mitään merkitystä, koska ne liittyvät luokan sisäiseen toteutukseen. Yleensä tällaiset ehdot kirjoitetaan luokkainvarianttiin (ks. kohta 3.2.2), mutta jotkin yksittäiseen rutiiniin liittyvät osat on kirjoitettava myös loppuehtoon. Tästä ei ole asiakkaalle haittaa, muttei sillä ole toisaalta mitään informaatioarvoakaan.

Ja vielä lyhyesti: *hyvän määrittelyn tunnistaa siitä, että rutiinin merkitys on ilmaistu yksinkertaisesti ja lyhyesti ja että se kuvaa tarkasti rajatun ja hyvin määritellyn toimenpiteen, jonka ymmärtäminen ei riipu rutiinin käyttöyhteydestä.* Merkityksen kuvaamisessa on luonnollista käyttää apuna luokan muita julkiseen liitäntään kuuluvia piirteitä, koska luokka määrittelee abstraktin käsitteen ja väittämät liittyvät ko. abstraktin käsitteen ominaisuuksiin. Kytkemällä julkisia piirteitä näin toisiinsa asiakas ymmärtää paremmin rutiinin nivoutumisen muuhun kokonaisuuteen.

2.3.3 Väittämä

Alku- ja loppuehtojen kaltaisia *väittämiä* (*assertion*) voidaan kirjoittaa mihin kohtaa ohjelmaa tahansa.[8] Väittämät kertovat aina jotakin ohjelmassa olevien tunnisteiden ja vakioiden välisistä suhteista eli ohjelman tilasta (muistin sisällöstä) saavuttaessa suorituksessa ko. ohjelmakohtaan. Väittämää voi siis pitää tavallisena totuusarvolausekkeena, jonka tulisi olla voimassa (**`true`**) siinä kohtaa ohjelmaa, mihin se on kirjoitettu. Yleinen tapa on se, että väittämään kirjoitetaan vain sellaisia ehtoja, jotka koskettavat toimintaa; muutoin järjestelmän tila oletetaan muuttumattomaksi.

Kahden muistipaikan sisällön vaihtaminen voitaisiin kuvata kirjoittamalla kunkin käskyn edelle ja jälkeen aina väittämä, joka kertoo missä tilassa ohjelma on sillä hetkellä:

```
/* Alkuehto P: (x == OLD(x)) & (y == OLD(y)) */
```

[7] Rutiinin käyttäytymiseen voivat periaatteessa vaikuttaa, paitsi argumenttien kautta saatavat tiedot, myös sellaiset globaalit tiedot, joihin rutiini pääsee käsiksi. Globaalista tiedosta riippuvaa toimintaa pitäisi kuitenkin ehdottomasti välttää ja välttämättömissä tapauksissa se on dokumentoitava selkeästi.

[8] Matemaattisesti tulkiten väittämät ovat *ensimmäisen asteen predikaattilogiikan* (*first-order predicate logic*) peruselementtejä, *relaatioita*.

```
apu = x;
/* (x == OLD(x)) & (y == OLD(y)) & (apu == OLD(x)) */
x = y;
/* (x == OLD(y)) & (y == OLD(y)) & (apu == OLD(x)) */
y = apu;
/* Loppuehto Q: (x == OLD(y)) & (y == OLD(x)) */
```

Tunnisteiden `x` ja `y` alkuarvoiksi on nimetty `OLD(x)` ja `OLD(y)`. Väittämät on kirjoitettu kommenteiksi, ja ensimmäiselle ja viimeiselle väittämälle on annettu nimet (`P` ja `Q`). Antamalla käskylohkon muodostamalle kokonaisuudelle nimi `swap`, voidaan sen toiminnasta antaa kuvaus[9]

```
/* P */ swap(x, y); /* Q */.
```

Tämä kuvaus kertoo, että jos ehto `P` on tosi rutiinin `swap` alkaessa, rutiinin suoritus johtaa tilaan `Q`. Pari $\langle$`P`, `Q`$\rangle$ ilmaisee abstraktilla tasolla, että rutiini vaihtaa muistipaikkojen `x` ja `y` sisällöt keskenään. Käskylohko `swap(x, y)` on eräs tämän abstraktin kuvauksen täyttävistä toteutuksista. On hyvä huomata, että kolmikko `/* P */ swap(x, y); /* Q */` on myös itsessään väittämä; se on tosi, jos toteutus vastaa rutiinin määrittelyä. Määrittely antaa siis referenssin, johon tukeudutaan silloin, kun halutaan osoittaa, että rutiinin toteutus toimii oikein.

Väittämiä voidaan käyttää myös varmistamaan silmukoiden toiminnan oikeellisuus. Tällainen ns. silmukkainvariantti on väittämä, jonka tulee olla voimassa silmukan jokaisen kierroksen alussa (ja lopussa). Silmukkainvarianteista kerrotaan enemmän liitteessä B.

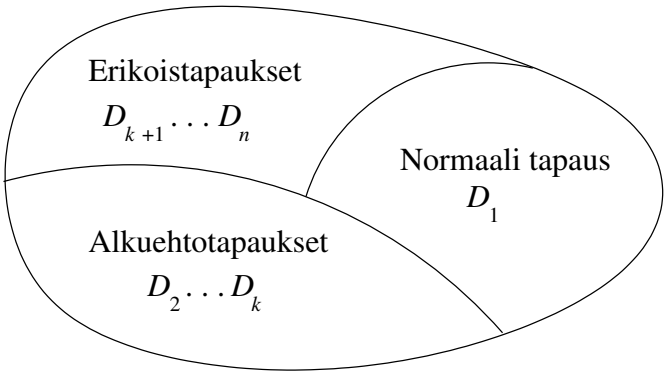
2.4 Erikoistilanteiden hallinta

Tarkastellaan vielä kokonaisvaltaisesti rutiinin käyttäytymistä, lähinnä sitä, miten rutiini kommunikoi asiakkaan kanssa erilaisissa tilanteissa. Kriittinen tarkastelu on tärkeää jo yksistään siitä syystä, että moniin ohjelmointikieliin on juurtunut erilaisia kulttuureja, jotka ohjelmoija omaksuu asettamatta niitä useinkaan kyseenalaiseksi. Kysymys on asiakkaan ja toimittajan välisestä sopimuksesta; on tärkeää miettiä huolellisesti ne tilanteet ja tavat, joilla kutsujan ja kutsuttavan välinen kommunikointi tapahtuu, jotta liitännästä saataisiin järkevä ja selkeä.

2.4.1 Rutiinin määrittely- ja arvojoukko

Rutiinia voidaan abstraktisti tarkastella matemaattisena funktiona, jolla on oma määrittelyjoukkonsa (*domain*) D ja arvojoukkonsa R (*range*). Matemaatikot puhuvat tässä yhteydessä lisäksi myös maalijoukosta C (*codomain*), jonka osajoukko

[9]Tämä ei ole Java-kieltä vaan kyse on koodiosan yleisestä nimeämisestä.



Kuva 2.1: Määrittelyjoukon jako eri osiin.

R on: $R \subseteq C$. Rutiini on siis kuvaus $f : D \to R$. Joukko D määräytyy rutiinille annettavien parametrien tyypeistä.[10] Esimerkiksi itseisarvon laskevalle rutiinille se on $\mathbb{R}$ (kaikkien reaalilukujen joukko).[11] Yleisemmin se on kuitenkin usean parametrityypin karteesinen tulo. Esimerkiksi kokonaislukujakolaskussa se on $\mathbb{Z} \times \mathbb{Z}$ ($\mathbb{Z}$ edustaa kaikkien kokonaislukujen muodostamaa joukkoa), ja `paikka`-rutiinissa (ks. sivu 31) `Object[]` $\times$ `Object`. Rutiinin arvojoukko muodostuu puolestaan normaalitapauksen ja poikkeustilanteiden palauttamien tietotyyppien karteesisesta tulosta.

Vahvasti tyypitetyssä kielessä (kuten Java) käännösaikainen tyyppitarkistus takaa sen, että kutsussa esiintyvät todelliset parametrit ovat rutiinin määrittelyjoukon alkioita ja rutiinin palauttamat arvot sen arvojoukon alkioita. Rutiinin tehtävänä on partitioida D eli jakaa se erillisiin osajoukkoihin $D_1, D_2, \ldots, D_n$ siten, että

$$D = D_1 \cup D_2 \cup \cdots \cup D_n$$

missä kunkin D_i-joukon alkiot käyttäytyvät aina samalla tavalla ja kahden eri joukon alkiot eri tavoin. Kiinnitetään D_1 normaalitapaukseksi, muut ovat siitä poikkeavia.[12] Joukot $D_2, \ldots, D_k$ liittyvät alkuehtojen poissulkemiin tapauksiin ja $D_{k+1}, \ldots, D_n$ erikoistilanteisiin (ks. kuva 2.1). Edelliset hoitaa kutsuja, jälkimmäiset kutsuttava.

Kuhunkin rutiiniin liittyy aina D_1, muut ovat valinnaisia. Jos $k = 1$, rutiinilla ei ole alkuehtoja, joten se on totaalinen. Toisaalta jos $k = n$, rutiini ei hoida erikoistapauksia (niitä ei ole tai ne on siirretty alkuehtoon). Joissakin tilanteissa alkuehtotapaus D_i $(i = 2, \ldots, k)$ voidaan joutua muuttamaan erikoistapaukseksi

[10] Huomaa, että *kaiken* rutiinille välitettävän tiedon voidaan ajatella kulkevan parametrien kautta.

[11] Lukujen esitystavasta johtuva epätarkkuus ja lukualueen äärellisyys jätetään tässä huomiotta.

[12] Huomaa, että jonkin tapauksen valitseminen ”normaalitapaukseksi” saattaa johtua vain näkökulmasta, josta asiaa tarkastellaan; joku muu voisi pitää jotain toista tapausta normaalina.

$D_{k+1}, \ldots, D_n$ siitä yksinkertaisesta syystä, että alkuehdon tarkistaminen on epäkäytännöllistä (tehotonta), jolloin ratkaisua yritetään ensin ja vasta jälkeenpäin tarkistetaan mitä tapahtui. Joskus taas alkuehdon tarkistaminen on mahdotonta siksi, että operaation soveltuvuutta voidaan testata vain kokeilemalla (esim. lukeminen tiedostosta). Tätä menettelyä voidaan soveltaa tietysti vain, mikäli epäonnistunut yritys ei johda vakavampiin seurauksiin.

2.4.2 Erikoistilanteesta tiedottaminen

Alkuehdon mukaiset tapaukset osataan jo käsitellä sen mukaisesti mitä aiemmin on kerrottu, joten keskitymme seuraavassa vain normaali- ja erikoistilanteiden hoitoon, lähinnä siihen, miten ne erotetaan toisistaan. Ennen toteutusvaihtoehdon valintaa on kuitenkin syytä analysoida erikoistilanteet huolellisesti eli (a) varmistua siitä, että kaikki erikoistilanteet on katettu ja (b) miettiä kuhunkin tapaukseen oma hoitotapansa. Kohta (a) tuntuu itsestäänselvyydeltä, mutta käytäntö on osoittanut, että puuttuvien, ehkäpä marginaalisiltakin tuntuvien tapausten käsittely jää usein tekemättä ja se on syypää hyvin moniin ohjelmointivirheisiin. Esimerkiksi solmun lisäys linkettyyn listaan voi toimia oikein yleisessä tapauksessa, mutta epäonnistuu, jos lista on tyhjä; lajittelualgoritmi toimii muuten oikein, mutta ei silloin, jos lajiteltavana on vain yksi alkio jne.

Ensimmäinen sääntö erikoistapausten hoitoon on, että ne yritetään välttää kokonaan antamalla kyseiset tapaukset suoraan kutsujan hoidettavaksi. Kutsuja on korkeammalla abstraktiotasolla kuin kutsuttava, joten se osaa hoitaa erikoistapaukset luultavasti paremmin kuin kutsuttava. Erikoistapaus jää suoraan kutsujan hoidettavaksi tietysti silloin, kun se kirjataan rutiinin alkuehdoksi.

Jos erikoistilannetta ei haluta siirtää alkuehtoon, mietitään muita vaihtoehtoja. Rutiini voi erikoistilanteessa välittää tietoa kutsujalle viidellä perustavalla:

(a) nostamalla poikkeuksen,

(b) viittaustyyppisen parametrin kautta,

(c) toimittajaluokan attribuuttien kautta,

(d) globaalia tietoa sisältävän olion kautta tai

(e) palauttamalla tietoa normaalitapauksen tapaan **`return`**-lauseella.

Riippumatta valitusta vaihtoehdosta, *kutsuja joutuu jokaisen kutsun jälkeen aina tarkistamaan mitä kutsuttavassa tapahtui.* Tämä aiheuttaa helposti sekavuutta kutsujan koodiin.

Yleissääntönä voidaan pitää sitä, että *erikoistapauksista tulee välittää tieto kutsujalle aina tarkistettavien poikkeusten avulla.* Silloin kutsuja joutuu ottamaan

kantaa erikoistilanteisiin (kirjoittamaan **catch**-osan), sillä kääntäjä huolehtii, että niiltä ei voi ummistaa silmiään. Parametrien kautta välitettävää tietoa tulisi välttää, koska se ei näy kutsujalle syntaktisesti mitenkään ja aiheuttaa herkästi tilanteita, joihin kutsuja ei ole varautunut (tai halua sitoutua).

Yhdistelmä, missä funktio välittää tietoa (paitsi **return**-lauseen myös) parametrien tai luokassa olevien attribuuttien kautta, on erittäin ongelmallinen, koska se tekee epätoivottavan sivuvaikutuksen (tästä lisää luvussa 3). Tällaisessa tilanteessa kaikki tiedot kannattaa välittää kutsujalle **return**-lauseen kautta esimerkiksi `Object[]`-tyyppisessä taulukossa. Tämä ratkaisu vaatii kuitenkin palautusarvojen tulkintaa eli kutsujalta tarkkaa tietoa palautettavien olioiden järjestyksestä ja merkityksestä. Parempi vaihtoehto olisi tehdä kutakin palautettavien arvojen kombinaatiota kohden aina oma luokkansa ja antaa asiakkaan käyttää sen operaatioita, jolloin tilanne on asiakkaan kannalta mielekkäämpi.

Esimerkki 2.12 Erityisesti Javan geneerisyys (ks. luku 7) auttaa tällaisten aliohjelmakohtaisten paluuolioiden kuvauksessa. Voimme nimittäin määritellä palautettavalle tiedolle ”vartijoijan”, jonka tehtävänä on pitää kirjaa funktion palauttaman tuloksen laillisuudesta:

```
public class Vartijoitu<T>
  {
  private boolean onLaiton;
  private T arvo;

  protected Vartijoitu(boolean onLaiton, T arvo)
    {
    this.onLaiton = onLaiton;   this.arvo = arvo;
    }

  public boolean onLaillinen() { return !onLaiton; }

  /** @.pre  onLaillinen() */
  public T annaArvo() { return arvo; }

  public static <T> Vartijoitu<T> luoLaillinen(T arvo)
    {
    return new Vartijoitu<T>(false, arvo);
    }

  public static <T> Vartijoitu<T> luoLaiton()
    {
    return new Vartijoitu<T>(true, null);
    }
  }
```

Jos esimerkiksi emme halua rajoittaa `Pino`-rajapinnan (ks. listaus 2.1) `päällimmäinen`-metodin käyttöä mitenkään, voimme määritellä sen myös seuraavasti:

```
/**
 *  Palauttaa pinon päällimäisen alkion.
 *  @.pre  true
 *  @.post RESULT.onLaillinen() ==> (RESULT.annaArvo() ==
 *                                   (pinon päällimmäinen alkio))
 */
public Vartijoitu<T> päällimmäinen();
```

Vaihtoehto (c) on myös ongelmallinen, sillä asiakkaan on helppo ”unohtaa” tarkistaa, millä tavoin operaation suoritus on päättynyt. Menettely epäonnistuu myös silloin, kun useat eri säikeet voivat synkronoimatta muuttaa kohdeolion attribuuttien arvoja.

Vaihtoehtoa (d) eli globaalien olioiden käyttöä tiedonvälitykseen pitäisi myös välttää, sillä kyseinen menettely luo vahvoja sidoksia eri rutiinien välille. Ja kuten tiedämme *kaikki ohjelmakomponenttien väliset kiinteät sidokset ovat haitallisia*, silloin kun ohjelmaa muutetaan ja ylläpidetään.

Vaihtoehto (e) on tuttu monille C-ohjelmoijille (ja valitettavasti myös Java-ohjelmoijille). Kieleen on voimakkaasti juurtunut tapa, jossa erikoistilanne *koodataan palauttamalla erityinen tulostieto*, joka ei tehtävän luonteen huomioon ottaen voi olla normaalitulos:

```
/**
 *  @.pre  true
 *  @.post RESULT == (n:n kertoma jos n >= 0; muuten 0)
 */
public static int kertoma(int n)
```

Kuten vaihtoehdoissa (c) ja (d), myös tämän ratkaisun haittana on, että erikoistapaus saattaa jäädä vahingossa huomioimatta, koska siitä ilmoitetaan kutsujalle tarkalleen samalla menettelyllä kuin normaalituloksestakin. Vastuu on siis kutsujalla. Sitä paitsi on paljon tilanteita, joissa rutiinin arvojoukkoa ei voida jakaa normaali- ja erikoisalueisiin.[13]

Esimerkki 2.13 `Pino`-luokan rutiini `päällimmäinen()` ei voi käyttää **`null`**-arvoa erikoistilanteen ilmaisemiseen, koska kyseinen arvo saattaa olla joissakin tilanteissa ihan käypä tieto pinoon talletettavaksi. Myöskään

```
/**
```

[13] Javan automaattinen kuorrutus ja kuorinta tosin mahdollistaisi primiitivityyppien arvoalueen laajentamisen **`null`**-viittauksella, mutta se, onko tällainen ratkaisu järkevä, onkin sitten aivan toinen asia.

```
 * @.pre  true
 * @.post RESULT == (pinon päällimmäinen alkio)
 */
public T päällimmäinen()
  {
  if (onTyhjä()) System.out.println("Pino on tyhjä!");
  else return pino[koko - 1];
  }
```

ei ole järkevä, sillä yleiskäyttöinen `päällimmäinen()` *ei saa kirjoittaa mitään oletustulosvirtaan.*

Rutiinin ei siis pidä itse tulostaa virheilmoitusta, sillä (i) rutiinin kutsuja tietää paremmin mitä erikoistilanteen sattuessa kannattaa tehdä ja (ii) rutiini tekee tällöin sivuvaikutuksen: se tekee määrittelyssä esitellyn toiminnan lisäksi jotain ylimääräistä, josta voi koitua asiakkaalle haittaa. *Tällaisia sivuvaikutuksia pitäisi välttää kaikin tavoin.* Joissakin erittäin harvinaisissa erikoistilanteissa yo. toteutuksen kaltainen menettely voidaan katsoa järkeväksi; esimerkiksi jos syöttötiedoissa on virheellinen tietue, se kannattaa jättää prosessoimatta, tulostaa tieto tapahtuneesta *virhevirtaan* (`System.err`) ja lopettaa ohjelman suoritus hallitusti (esim. komennolla `System.exit(int)`). Toinen mahdollinen paikka, missä erikoistilanteeseen reagoidaan tulostamalla on silloin, kun käytetään applettia, jolla ei ole konsoli-ikkunaa virheilmoituksia varten. Tällöin tilanteesta voidaan ilmoittaa `showStatus(String)`-metodilla selaimen tilariville.

Yhteenvetona tästä tarkastelusta saadaan seuraava nyrkkisääntö erikoistilanteiden käsittelyyn:

1. Siirrä erikoistilanne alkuehtoon, mikäli se on asiakkaan näkemän abstraktion — siis rutiinin ja sitä ympäröivän luokan — kannalta selkeää ja ymmärrettävää.

2. Erikoistilanteet, joita ei ole siirretty alkuehtoon, tulisi pääsääntöisesti hoitaa tarkistettavilla poikkeuksilla, jolloin kutsuja pakotetaan ottamaan kantaa erikoistapauksiin. Muodosta tarvittaessa tilannetta vastaava poikkeusluokka ja mieti mitä tietoa asiakas saattaisi haluta poikkeusolion kautta välitettävän. Kirjoita luokkaan piirteet tämän analyysin pohjalta.

3. Jos edelliset säännöt eivät jostain kumman syystä johda luonnollisiin ratkaisuihin, välitä tieto erikoistilanteen laadusta räätälöidyn vartijaolion, parametrien, toimittajaluokan havainnointioperaatioiden tai erikoisarvojen avulla asiakkaalle. Käytä tätä menettelyä kuitenkin vain aivan äärimmäisenä vaihtoehtona.

Tämän mukaan **`return`** on varattu vain normaalitapausta varten. Toisaalta, *normaalitapauskin* välitetään usein argumenttien kautta silloin, kun se vaikkapa rutii-

nin aika- tai tilatehokkuuden kannalta on puolusteltavissa. Esimerkiksi lajittelurutiini ei normaalisti tee kopiota taulukosta, jonka sisältö lajitellaan, vaan palauttaa tuloksen samassa taulukossa (katsopa `Arrays.sort()`-metodien määrittelyä). On kuitenkin selvää, että rutiinista saadaan yleiskäyttöisempi, jos se pitää syöttönä saadun taulukon sisällön alkuperäisenä ja palauttaa lajitellun tiedon **`return`**-lauseen kautta toisessa taulukossa.

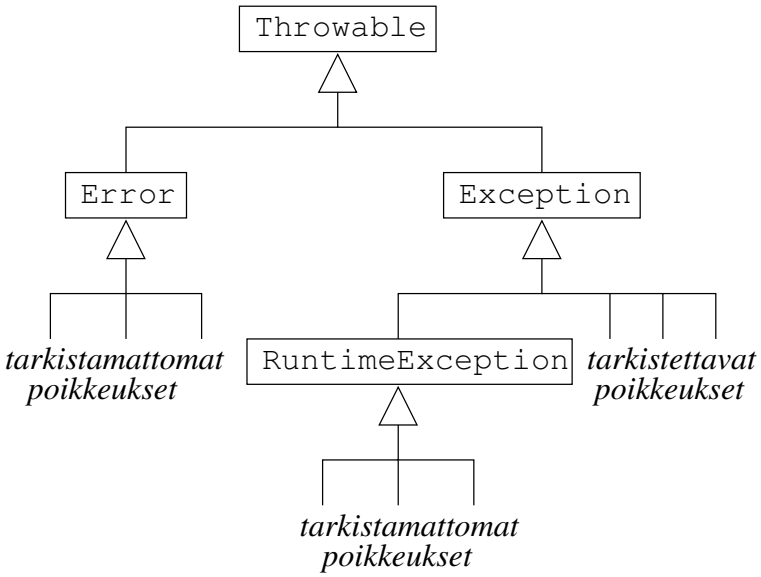
2.4.3 Javan poikkeusmekanismi

Rutiini päättyy useimmiten normaalisti, jolloin ohjelman kontrolli siirtyy **`return`**-lauseen tai rungon viimeisen käskyn suorituksen jälkeen asiakkaaseen, tarkemmin sanottuna kutsua seuraavaan käskyyn. Mikäli rutiini päättyy epänormaalisti ja nostaa poikkeuksen, kontrolli siirtyykin aivan muualle, nimittäin poikkeuksen hoitavaan käsittelijään, joka voi sijaita hyvinkin kaukana kutsupaikasta. Mikäli asiakas haluaa itse varautua sekä normaaliin että poikkeustapaukseen, tulee kutsu ympäröidä **`try`**–**`catch`** -osalla. Poikkeustenkäsittelijässä päätetään tehtävistä toimenpiteistä sekä siitä, poiketaanko normaalista kontrollivuosta jatkossakin.

Poikkeuksen nostoon liittyy aina jokin poikkeusolio (kuinkas muuten), jonka avulla on mahdollista välittää mitä tahansa tietoa poikkeuksen synnyttäneestä ohjelmakohdasta käsittelijään. Tässä mielessä poikkeus muistuttaa normaalipaluuta rutiinista, jonka suoritus päättyy **`return`**-lauseeseen. Poikkeusolion kautta välitetyn tiedon avulla käsittelijä pääsee ”syvemmälle” poikkeuksen aiheuttajaan. On kuitenkin varottava antamasta käsittelijälle sellaista tietoa, jonka avulla käsittelijä pääsee muuttamaan poikkeuksen aiheuttaneen olion sisäistä esitysmuotoa (informaation piilottamisperiaate murtuu).

Abstraktein poikkeuksia käsittelevistä luokista on `Throwable`, joka on juuriluokan `Object` perijä (ks. kuva 2.2). Kaikkien Java-ohjelmassa nostettavien poikkeusten pitää olla `Throwable`-yhteensopivia (eli esitelty jossakin sen aliluokassa). Luokalla `Throwable` on kaksi konstruktoria: toinen on argumentiton ja toisella on `String`-tyyppinen argumentti, jota käytetään virheilmoituksen antoon. Tämä merkkijono saadaan halutessa käsiin luokan `Throwable` rutiinilla `getMessage()`. Luokalla on mm. rutiini `printStackTrace()`, jolla voidaan tulostaa virhevirtaan se rutiinien kutsuketju, joka johti virheeseen (sekä virheen oletettu syy).

Javan poikkeukset jaetaan niiden semantiikan mukaan kahteen eri luokkaan: *tarkistettaviin poikkeuksiin* (*checked exceptions*) ja *tarkistamattomiin poikkeuksiin* (*unchecked exceptions*). Tarkistettavat edustavat ”normaalipoikkeuksia”, joita nostetaan esimerkiksi kun rutiinissa kohdataan erikoistilanne. Tarkistamattomat ovat luonteeltaan varsin vakavia poikkeuksia. Ajatuksena on, että tarkistamaton poikkeus edustaa virhettä, josta ei pystytä ohjelmallisesti toipumaan, joten sille ei (yleensä) ole mahdollista edes kirjoittaa käsittelyosaa. Esimerkiksi `Throwable`-luokan välitön perijä `Error` edustaa tarkistamattomia poikkeuksia, jotka liittyvät Ja-



Kuva 2.2: Poikkeusluokkien päähaarat.

va-systeemin ajoaikaisessa toiminnassa tapahtuneisiin virheisiin (virhe virtuaalikoneessa, linkitysvirhe, väittämärikkomus, jne.).

Tarkistamattomat poikkeukset

Tarkistamattomat poikkeukset eroavat melko selvästi tarkistetuista Java-syntaksin suhteen:

(a) Poikkeuksen nimeä ei tarvitse kirjoittaa rutiinin otsikkoon, vaikkei se ole kuitenkaan kiellettyä. On käytännössä hyvin suositeltavaa että tällainen poikkeus dokumentoidaan liittymäkuvaukseen, koska silloin kutsuja näkee kaikki ne mahdolliset tavat, joilla kutsuttava voi lopettaa toimintansa.

(b) Tarkistamattomia poikkeuksia ei tavallisesti yritetä hoitaa poikkeustenkäsittelijän avulla vaan annetaan edetä kutsuketjussa aina ylimmälle tasolle (aina `main`-metodin kutsujalle) asti, jolloin ohjelman suoritus pysähtyy.

(c) Rutiinin korvausta ei ole perijäluokassa millään tavalla sidottu yliluokan vastaavan rutiinin otsikossa mainittuihin tarkistamattomiin poikkeuksiin. Erityisesti rutiinin korvauksen kovarianssisääntö ei ole voimassa (ks. kohta 5.4.1).

Luokka `Error` on omistettu Java-systeemin omille vakaville virheille. Jos ohjelmoija haluaa esitellä omia tarkistamattomia poikkeuksia, suositellaan ne kirjoitettavaksi luokan `RuntimeException` perijöiksi. `RuntimeException` edustaa nimensä mukaisesti

tyypillisiä ajoaikaisia virhetilanteita. Näitä ovat mm. `NullPointerException` (yritys kutsua rutiinia `x.f()` kun `x == null`), `IndexOutOfBoundsException` (taulukon laiton indeksi), `IllegalArgumentException` (rutiinille on annettu vääränlainen argumentti) ja `ArithmeticException` (nollalla jakaminen).

Sopimuspohjaisessa ohjelmoinnissa tarvittavat poikkeukset ovat luonteeltaan tarkistamattomia, koska ne ilmentävät ohjelman loogisia virheitä, jotka on syytä korjata. Väittämän rikkomisesta ei voi yrittää toipua ohjelmallisesti, koska se olisi vastoin väittämäkäsitteen määritelmää. Tätä kuvastaa sekin, että **`assert`**-mekanismin mukainen poikkeus `AssertionError` on sijoitettu `Error`-luokan perijäksi.

Koska tarkistamattomat poikkeukset liittyvät vakaviin virheisiin, niitä ei yleensä pidetä osana rutiinin tavanomaista käyttäytymistä. Tästä syystä niitä ei kirjata rutiinin loppuehtoon vaan ne esitellään tarvittaessa `@throws`-täkyn avulla.

Tarkistettavat poikkeukset

Muut kuin `Error`- ja `RuntimeException`-poikkeukset ovat tarkistettavia, joten niihin ohjelmoija joutuu ottamaan aina eksplisiittisesti kantaa (kääntäjä tarkistaa, että näin on tehty). Tarkistettavat poikkeusluokat kirjoitetaan `Exception`-luokan perijöiksi. Tavallisesti poikkeuksen tyyppi (luokkanimi) on riittävä siihen, että käsittelijä tunnistaa ongelman ja osaa tehdä tarvittavat toimenpiteet. Tarkistettavat poikkeukset liittyvät rutiinissa esiintuleviin (ja odotettavissa oleviin) erikoistilanteisiin.

Jos poikkeusta — olipa kyseessä tarkistettu tai tarkistamaton — ei haluta käsitellä, rutiini voi päättää nostaa sen korkeammalle tasolle (eli omalle kutsujalleen) ja lopettaa näin oman suorituksensa. Tämä tapahtuu automaattisesti silloin, kun rutiinissa ei ole kyseisen poikkeuksen käsittelijää. Sama saadaan aikaan ottamalla poikkeus kiinni ja nostamalla käsittelijässä uusi. Tällaista toimintaa tarvitaan, jos poikkeuksen tyyppi tai sen mukanaan välittämä tieto muuttuu selkeästi siirryttäessä abstraktiotasolta toiselle.

Esimerkki 2.14 Listauksessa 2.2 esitelty rutiini nostaa `NullPointerException`-poikkeuksen heti ensimmäisessä asetuslauseessa, jos `taulu`-viittaus on **`null`**. Koska poikkeukselle ei ole omaa käsittelijää, se nousee samannimisenä kutsujaan. Sen sijaan kutsujalle ei ole mitään järkeä välittää `IndexOutOfBoundsException`-poikkeusta, koska se liittyy suoraan toteutukseen eikä rutiinin määrittelyn ilmaisemaan abstraktioon. Tästä syystä nostettavalle poikkeukselle on valittu nimi, joka on minimirutiinin asiakkaan kannalta ilmeikkäämpi.

Poikkeuksen edelleenvälittämistä on syytä käyttää harkitusti. Syynä on se, että rutiinin välitön kutsuja tietää yleensä parhaiten, mistä asia kiikastaa, kun poikkeus syntyy. Mitä ylemmäs kutsuketjussa edetään ennen kuin poikkeus käsitellään, sitä yleisempi käsittelystä pakostakin tulee, koska kutsutasojen lisääntyessä mene-

Listaus 2.2: Rutiini `minimi`.

```
/**
 *  Palauttaa taulukon minimialkion.
 *  @.pre  true
 *  @.post FORALL(a : taulu; RESULT <= a)
 *  @throws NullPointerException
 *          Nostetaan jos taulu == null.
 *  @throws TyhjäTaulukkoPoikkeus
 *          Nostetaan jos taulu.length == 0.
 */
public static int minimi(int[] taulu)
                  throws NullPointerException, TyhjäTaulukkoPoikkeus
  {
  try
    {
    int pienin = taulu[0];
    for (int i = 1; i < taulu.length; i++)
      if (taulu[i] < pienin) pienin = taulu[i];
    return pienin;
    }
  catch (IndexOutOfBoundsException e)
    {
    throw new TyhjäTaulukkoPoikkeus("minimi: tyhjä taulu");
    }
  }
```

tetään informaatiota poikkeuksen alkuperäisestä syystä. Käsittelijä voi esimerkiksi yrittää vain lopettaa ohjelman suorituksen hallitusti (sulkea tiedostot, hoitaa kesken jääneet transaktiot yms.). Jos tarkastellaan kutsuketjuun osallistuvien rutiinien otsikoita, niiden käsittelemät poikkeusluokat pyrkivätkin abstrahoitumaan perimishierarkiassa mitä lähemmäs siirrytään juuriluokan `main`-rutiinia.

Kääntäjä varmistaa, että rutiinin signatuurissa ilmaistaan ainakin kaikki ne tarkistettavat poikkeukset, jotka rutiinin suorituksen aikana voivat nousta ja joita rutiini ei itse käsittele. On kuitenkin sallittua kirjoittaa rutiinin otsikkoon myös sellaisten poikkeusten nimiä, joita *rutiini ei itse eksplisiittisesti nosta*. Näin toimitaan esimerkiksi silloin, kun rajapintaan tai abstraktiin luokkaan kirjoitetaan rutiini, josta tiedetään vain signatuuri, mutta ei toteutusta. Rutiinin määrittelyssä pitää tietysti jo ylimmällä tasolla ilmaista, mihin tarkoitukseen kyseisiä poikkeuksia tarvitaan. Määrittelyn mukainen käyttäytyminen pitää sitten toteuttaa

perijöissä, jotka antavat rutiinille konkreetit toteutukset.

Vielä tärkeä Java-sääntö: vaikka kääntäjä tarkistaakin poikkeusten oikeellisuuden rutiinien signatuurista, *poikkeukset eivät ole osa rutiinin tyyppiä*. Käytännössä tämä tarkoittaa sitä, että rutiineja ei voi ylikuormittaa pelkillä poikkeusnimillä (ks. kohta 5.4).

Tarkastellaan lopuksi tilannetta, missä luokan `A` rutiini `f` toteutetaan uudelleen luokassa `B`, joka perii (suoraan tai epäsuorasti) luokan `A`. Jos rutiinin `A.f` otsikossa on mainittu tarkistettavia poikkeuksia, `B.f` ei saa esitellä uusia tarkistettavia poikkeuksia, joita perustoteutuksen poikkeukset eivät "kata". Toisin sanoen, `B.f` voi esitellä vain sellaisia poikkeustyyppejä, jotka ovat rutiinin `A.f` poikkeusten alityyppejä.[14] *Tällä taataan se, että perusluokkaa (*base class*) silmälläpitäen kirjoitettu kutsu toimii myös perijäluokan vastaavalle rutiinille* (polymorfismin ja dynaamisen sidonnan ansiosta asiakas ei ole aina tietoinen viittauksen kohteen tyypistä). Sääntöön nojautuen perusluokan rutiinin tarkistettavien poikkeusten listasta voidaan poistaa sellaiset, jotka perijäluokan koodi osaa hoitaa. Samasta syystä rutiinin `A.f` poikkeuslistassa voi olla vain esimerkiksi yksi "kattava" poikkeus (vaikkapa `Exception`), joka perijäluokassa korvataan usealla tarkemmalla poikkeuksella (esim. `AlkioEiLöydyException, DuplikaattiAlkioException`). Näin lueteltavien poikkeusten *määrä* voi joko kasvaa tai pienetä luokkahierarkiassa alaspäin mentäessä. Tästäkin huolimatta yllämainittu sääntö pitää huolen siitä, että rutiinin poikkeusten kirjo voi vain kaventua (erikoistua) mentäessä perijäluokkiin.

Tehtäviä

2-1 Kuvaile seuraavien metodien toimintaa.

(a)
```
/**
 *  @.pre  i <= j
 *  @.post RESULT >= i & RESULT <= j
 */
public static int aaa(int i, int j)
```

(b)
```
/**
 *  @.pre  k != null && k.length >= 1
 *  @.post FORALL(i : 0 <= i < k.length;
 *                 RESULT <= k[i] & k[i] == OLD(k[i]))
 */
public static int bbb(int[] k)
```

[14] Poikkeuksen tähän sääntöön muodostavat konstruktorit, jotka ovat toisistaan täysin riippumattomia.

(c)
```
/**
 *  @.pre  k != null
 *  @.post RESULT != null & FORALL(i : 0 <= i < RESULT.length;
 *         RESULT[i] < x & EXISTS(a : k; RESULT[i] == a))
 */
public static int[] ccc(int x, int[] k)
```

2-2 Kuvaile seuraavien rutiinien toimintaa.

(a)
```
/**
 *  @.pre  i <= j & i != 0 & j != 0
 *  @.post (RESULT % i == 0) & (RESULT % j == 0)
 */
public static int aaa(int i, int j)
```

(b)
```
/**
 *  @.pre  t != null && t.length >= 1
 *  @.post EXISTS(i: 0 <= i < RESULT.length;
 *           FORALL(j: 0 <= j <= i; RESULT[j] <= x) &
 *           FORALL(j: i + 1 <= j < RESULT.length; RESULT[j] > x))
 */
public static int[] bbb(int[] t, int x)
```

(c)
```
/**
 *  @.pre  s != null
 *  @.post RESULT != null &
 *         FORALL(i: 0 <= i < RESULT.length();
 *           RESULT.charAt(i) != m &
 *           EXISTS(j: 0 <= j < s.length();
 *             RESULT.charAt(i) == s.charAt(j))
 */
public static String ccc(String s, char m)
```

2-3 Tarkastele metodien `aaa`, `bbb` ja `ccc` määrittelyjä ja arvioi sen jälkeen annettujen väittämien paikkansapitävyyttä. Perustele vastauksesi lyhyesti.

```
/**
 * @.pre  t != null
 * @.post FORALL(i: 0 <= i <= RESULT.length - 2;
 *               RESULT[i] <= RESULT [i + 1])
 */
public static int[] aaa(int[] t)

/**
```

```
 * @.pre  t != null
 * @.post FORALL(i: 0 <= i <= RESULT.length - 2;
 *               RESULT[i] <= RESULT [i + 1]) &
 *               RESULT.length == t.length
 */
public static int[] bbb(int[] t)

/**
 * @.pre  t != null
 * @.post
 *    FORALL(i: 0 <= i <= RESULT.length - 2;
 *           RESULT[i] <= RESULT [i + 1]) &
 *    FORALL(i: 0 <= i <= RESULT.length;
 *           EXISTS(j: 0 <= j < t.length; RESULT[i] == t[j])) &
 *    FORALL(i: 0 <= i <= t.length;
 *           EXISTS(j: 0 <= j < RESULT.length; t[i] == RESULT[j])) &
 *    RESULT.length == t.length
 */
public static int[] ccc(int[] t)
```

(a) Metodit `aaa`, `bbb` ja `ccc` palauttavat kaikilla alkuehtojensa mukaisilla syötteillä ei-vähenevään suuruusjärjestykseen järjestetyn kokonaislukutaulukon.

(b) Metodi `aaa` saattaa palauttaa syötteellä $\{3, 2, 3\}$ taulukon $\{2, 3\}$.

(c) Metodi `bbb` rikkoo loppuehtoaan, jos se syötteellä $\{7, 3, 6\}$ palauttaa taulukon $\{3, 3, 7\}$.

(d) Syötteellä $\{1, 2, 1\}$ metodi `ccc` palauttaa varmasti taulukon $\{1, 1, 2\}$.

2-4 Millaisen kontrahdin olet tehnyt osallistuessasi ohjelmointikurssille? Millaisia velvoitteita ja oikeuksia sinulla on asiakkaana? Entä millaisia velvoitteita ja oikeuksia kurssin pitäjällä on toimittajana?

2-5 Tutki seuraavaa määrittelyä. Onko se järkevä?

```
/**
 *  Tarkistaa ovatko taulukon t alkiot järjestyksessä.
 *  @.pre  t != null
 *  @.post RESULT == (true jos alkiot ovat järjestyksessä; muuten false)
 */
public static boolean onJärjestyksessä(int[] t)
```

Toteuttaako seuraava rutiini tämän määrittelyn? Jos ei, niin miten korjaisit sitä?

```
public static boolean onJärjestyksessä(int[] t) {
   for (int i = 0; i < t.length; i++)
```

```
    if (!(t[i] < t[i + 1])) return false;
return true;
}
```

2-6 Kirjoita sopivat määritykset seuraaville metodeille:

(a) Metodi tarkistaa, onko annettu merkkijono palindromi. Palindromi on sana joka on sama lukusuunnasta riippumatta, esimerkiksi ”saippuakivikauppias”.

(b) Metodi palauttaa annetun merkkijonotaulukon (so. taulukko, jonka alkiot ovat merkkijonoja) pisimmän alkion indeksin.

2-7 Kirjoita sopivat määrittelyt (eli alku- ja loppuehdot) seuraaville rutiineille:

(a) Rutiini palauttaa tiedon, onko parametrina annettu kokonaisluku laillinen postinumero (ts. väliltä 00000–99999).

(b) Rutiini tarkistaa onko annetussa kokonaislukutaulukossa yhdelläkään alkiolla duplikaattia (ts. rutiini palauttaa **true** jos ja vain jos kukin luku esiintyy taulukossa vain kerran).

(c) Rutiini palauttaa annetun **double**-tyyppisiä alkioita sisältävän taulukon pienimmän alkion indeksin. Mikäli pienimmällä alkiolla on duplikaatteja, rutiini palauttaa niistä indeksiltään pienimmän alkion.

2-8 Määrittelyn eri osia esiteltäessä alaviitteessä 1 (s. 15) todetaan, että Javassa poikkeusten merkitys on ”epämääräinen”. Selvitä mistä on kysymys kokeilemalla Java-kääntäjällä seuraavia esimerkkejä (yksi **public**-luokka yhteen tiedostoon). (Vinkki: tarkastele tilanteita seuraavien seikkojen kannalta: poikkeusten periytymishierarkia sekä tarkistamaton/tarkistettava poikkeus.)

(a) Metodin ylikuormitus (*overload*):

```
public class EsimerkkiA {
   void f() throws java.io.IOException { }
   void f() throws CloneNotSupportedException { }
   }
```

(b) Metodin korvaus (*override*):

```
public class EsimerkkiB1 {
   void f() throws Exception { }
   }

public class EsimerkkiB2 extends EsimerkkiB1 {
   void f() throws java.io.IOException { }
   }
```

(c) Metodin korvaus:

```
public class EsimerkkiC1 {
   void f() throws java.io.IOException, NullPointerException { }
   }

public class EsimerkkiC2 extends EsimerkkiC1 {
   void f() throws Exception, ArithmeticException { }
   }
```

2-9 Tarkastele Sunin Java API -dokumentissa annettua määrittelyä `String`-luokan metodille `regionMatches(int, String, int, int)`. Onko siinä puutteita? Antaako se järkevän kontrahdin?

2-10 Tutki seuraavia Javan API-kirjastosta löytyviä `String`-luokan rutiinimäärittelyjä. Erottele niistä rutiinin alku- ja loppuehdot sekä kirjoita ne.

(a) `endsWith(String)`

(b) `substring(int, int)`

2-11 Millä tavoin hoitaisit seuraavat erikoistilanteet?

(a) Hakumetodi `int alkionIndeksi(int[] t, int v)` palauttaa luvun `v` ensimmäisen esiintymän indeksin taulukossa `t`. Metodilla haettavaa lukua ei ole taulukossa.

(b) `Henkilö`-luokasta luodun olion `pituus`-attribuutiin yritetään asettaa negatiivista lukua ko. luokassa määritellyssä rutiinissa.

(c) Tiedostonlukurutiinin argumenttina annetun nimistä tiedostoa ei ole olemassa.

(d) Rutiinille annettu syötetiedosto on olemassa, mutta se sisältää vääränmuotoista tietoa.

(e) Funktio `Object[] ota(int t)` palauttaa ja samalla poistaa pinon `t` päällimmäistä alkiota. Metodilla yritetään poistaa enemmän alkioita kuin mitä pino sisältää.

2-12 Oletetaan että meillä on osamäärän laskeva rutiini, jolla on seuraava signatuuri:

```
public static double osamäärä(double jaettava, double jakaja)
```

Tarkastele seuraavia määrittelyjä kyseiselle rutiinille. Ovatko ne oikein? Mikä niistä on käyttökelpoisin ja missä tapauksessa?

(a)
```
/**
 *  @.pre  jakaja != 0.0
 *  @.post Math.abs(jaettava - (jakaja * RESULT)) < 1.0e-10.0
 */
```

(b)
```
/**
 * @.pre  true
 * @.post jakaja != 0.0 ?
 *          Math.abs(jaettava - (jakaja * RESULT)) < 1.0e-10.0 :
 *          (nostaa poikkeuksen ArithmeticException)
 */
```

(c)
```
/**
 * @.pre  true
 * @.post jakaja != 0.0 ?
 *          Math.abs(jaettava - (jakaja * RESULT)) < 1.0e-10.0 :
 *          RESULT == Double.NaN
 */
```

2-13 Ariane 5 -kantoraketti tuhoutui 4. kesäkuuta 1996 vain 40 sekuntia lähdön jälkeen. Syynä on virhe, joka kaatoi ohjaustietokoneen. 500 miljoonan euron tappiot aiheuttaneen onnettomuuden syyksi paljastui seuraavan kaltainen ohjelmanpätkä:

```
short y;
float x;
...
y = convert(x);
```

Kirjoita muunnosmetodille `public short convert(float x)` alku- ja loppuehto. Alkuperäisessä ohjelmassa niitä ei ollut (tai niistä ei välitetty). Mikä mahtoi mennä pieleen? (Vinkki: tutustu eo. primitiivityyppejä vastaaviin kuoriluokkiin.)

2-14 Listauksessa 2.1 on esitelty määrittely `Pino`-luokalle. Valitse sille konkreetti esitystapa ja laadi toteutus.

2-15 Listauksessa 2.3 esitelty `Intervalli`-rajapintaluokka kuvaa suljettua kokonaislukuväliä $[i, j]$ kun $i \leq j$ (ts. ala- ja yläraja sisältyvät lukuväliin ja lukuväli sisältää vähintään yhden kokonaisluvun). Valitse sille konkreetti esitystapa ja laadi toteutus.

2-16 Moottoripyöräilyn onnettomuusriski korreloi vuorokaudenaikaan seuraavasti[15]:

- vähäinen riski: klo 0–10 ja klo 11–12
- kohtalainen riski: klo 10–11 ja klo 21–24
- suuri riski: klo 12–21

(a) Laadi määrittely rutiinille

```
public static String riski (int tunti)
```

[15]Lähde: ⟨http://www.bajahill.net/mponnettomuus1986-95.html⟩, taulukko 4.

Listaus 2.3: Rajapinta `Intervalli`.

```
public interface Intervalli
  {
  /**
   *  Asettaa intervallin alarajan.
   *  @.pre  raja <= annaYläraja()
   *  @.post annaAlaraja() == raja &
   *         annaKoko() == annaYläraja() - raja + 1
   */
  public void asetaAlaraja(int raja);

  /**
   *  Asettaa intervallin ylärajan.
   *  @.pre  raja >= annaAlaraja()
   *  @.post annaYläraja() == raja &
   *         annaKoko() == raja - annaAlaraja() + 1
   */
  public void asetaYläraja(int raja);

  /**
   *  Palauttaa intervallin alarajan.
   *  @.pre  true
   *  @.post RESULT == (lukuvälin alaraja)
   */
  public int annaAlaraja();

   /**
    *  Palauttaa intervallin ylärajan.
    *  @.pre  true
    *  @.post RESULT = (lukuvälin yläraja)
    */
  public int annaYläraja();

  /**
   *  Lisää annetun arvon alarajaan.
   *  @.pre  annaKoko() - muutos > 0
   *  @.post annaAlaraja() == OLD(annaAlaraja()) + muutos
   */
  public void lisääAlarajaan(int muutos);
```

Listaus 2.3 (jatkoa): Rajapinta `Intervalli`.

```
/**
 *  Lisää annetun arvon ylärajaan.
 *  @.pre annaKoko() + muutos > 0
 *  @.post annaYläraja() == OLD(annaYläraja()) + muutos
 */
public void lisääYlärajaan(int muutos);

/**
 *  Palauttaa intervalling koon.
 *  @.pre  true
 *  @.post RESULT = annaYläraja() - annaAlaraja() + 1
 */
public int annaKoko();

/**
 *  Tarkistaa sisältyykö annettu intervalli tähän intervalliin.
 *  @.pre  toinen != null
 *  @.post RESULT == (annaAlaraja() <= toinen.annaAlaraja()) &
 *                   (annaYläraja() >= toinen.annaYläraja())
 */
public boolean sisältää(Intervalli toinen);

/**
 *  Tarkistaa sisältyykö annettu lukuarvo tähän intervalliin.
 *  @.pre  true
 *  @.post RESULT == (annaAlaraja() <= arvo) &
 *                   (annaYläraja() >= arvo)
 */
public boolean sisältää(int arvo);
}
```

joka palauttaa merkkijonon `"vähäinen"`, `"kohtalainen"` tai `"suuri"` riippuen parametrina annetusta kokotunnista ja kyseisen ajankohdan onnettomuusriskistä.

(b) Listauksessa 2.3 esitelty `Intervalli`-rajapintaluokka kuvaa suljettua kokonaislukuväliä $[i, j]$. Oletetaan että olet asiakkaan roolissa. Tarkastele `Intervalli`-rajapintaa (mutta älä suotta pohdi sen toteutusta, sillä olethan luokan asiakas etkä sen toimittaja). Laadi sen pohjalta toteutus kohdassa (a) määrittelemällesi rutiinille `riski`.

2-17 Olet töissä pienessä mutta innovatiivisessa ohjelmistoyrityksessä Tasopix ja olet juuri palannut lomilta, kun pomosi askeltaa työpisteesi viereen seuraavan kiireellisen tehtäväksiannon kanssa: Luokkaan `Tasopiste` (ks. listaus 3.1, s. 67) on lisättävä funktio

```
public Tasopiste[] annaBresenhamit()
```

joka palauttaa kohdeolion määräämän ns. Bresenham-tasopisteiden joukon taulukkona. Pisteen **`this`** määräämä Bresenham-pisteiden joukko sisältää kaikki ne pisteet, jotka saadaan x- ja y-akselien peilauksien ja akselinvaihtojen yhdistelmillä. Hetken mietittyäsi hoksaat, että funktion implementointi itsessään ei ole ongelma vaan sen ytimekäs määrittely. Valmistaudu seuraavaan palkkaneuvotteluun suoriutumalla tehtäväksiannosta (so. funktion alku- ja loppuehtojen määrittelystä) mahdollisimman ymmärrettävästi; voit määritellä luokkaan `Tasopiste` myös työtäsi avittavia lisärutiineja.

Luku 3

Luokan muodostaminen

Olemme aiemmin keskittyneet tutkimaan pääasiassa luokan peruskomponenttien, yksittäisten piirteiden, toteutukseen liittyviä sääntöjä; sääntöjä, joilla piirteistä saadaan järkevästi käyttäytyviä ja helposti käytettäviä. Tässä luvussa selvitetään keskeisiä luokkakokonaisuuteen liittyviä asioita. Esimerkin avulla analysoidaan ensin käsitettä, jonka luokka toteuttaa. Analyysin pohjalta tehtävät päätökset vaikuttavat suoraan siihen, miten mielekkäänä potentiaaliset asiakkaat näkevät luokan julkisen liitännän. Asiakkaan huomio kiinnittyy ensi vaiheessa liitännän muodostamaan kokonaisuuteen — siis siihen, miten piirrevalikoima on jäsennelty, miten kattava se on ja miten piirteet toimivat yhdessä. Liitäntä ei saa olla liian laaja, koska se on omiaan sekoittamaan asiakasta ja hämärtämään luokan abstrahoimaa käsitettä. Ymmärrettävyyteen vaikuttavat myös yksityiskohdat: operaatioiden rajaukset ja niiden käytön selkeys. Nämä ominaisuudet ovat johdettavissa suoraan rutiinien määrittelyistä. Liitännän suunnittelussa voi myös käyttää apuna luokan abstraktia tietotyyppiä (*abstract data type*, ADT), josta kerrotaan enemmän liitteessä C.

Kun julkinen liitäntä on suunniteltu, mietitään vaihtoehtoja luokan sisäiselle esitysmuodolle. Perusteet valinnalle saadaan asettamalla liitännän operaatiot preferenssijärjestykseen suoritustehokkuuden mukaan ja valitsemalla annetut vaatimukset täyttävä esitysmuoto. Jos myöhemmin todetaan, että toinen asiakas haluaakin painottaa operaatioita eri tavoin, voidaan samasta käsitteestä tehdä uusi luokka, jonka sisäinen toteutustapa tyydyttää uuden asiakkaan tarpeet. Samalla käsitteellä voi olla siis useita eri luokkatoteutuksia.

Koska olioajattelun päämääränä on kirjoittaa luokkia, joita muut voivat käyttää hyväkseen yhä uudelleen, nousevat sellaiset ominaisuudet kuten rutiinien ja luokan toimivuus ja oikeellisuus erityisen korostettuun asemaan. Olioparadigmassa ohjelman suorituksen ajatellaan koostuvan systeemiin ajoaikana luoduista olioista,

jotka kommunikoivat keskenään lähettämällä toisilleen *viestejä* (*message*).[1] Oliot ovat itsenäisiä yksiköitä, joilla on oma sisäinen tilansa, jota ulkopuoliset pääsevät muuttamaan vain kontrolloidusti, hyvin määritellyn liitännän kautta (informaation piilottamisperiaate, engl. *information hiding principle*). Jos tämä periaate ei päde, koko rakennelma sortuu. Tästä syystä olion *sisäisen esitysmuodon eheyteen* (*integrity*) pitää kiinnittää erityistä huomiota varsinkin, jos käytetty ohjelmointikieli ei tue sitä kunnolla. Piirteiden suojausmääreet (**`private`**, **`protected`**, **`public`**) on syytä pitää hyvin tiukkoina, jotta ennakoimattomilta, olion ulkopuolelta tulevilta muutoksilta vältyttäisiin. Erityisen suurena apuna on myös *luokkainvariantin* (*class invariant*) käsite. Luokkainvariantti on väittämä, jonka avulla huolehditaan siitä, että luokkaan kirjoitetut operaatiot eivät itse riko olion eheyttä sen sisäpuolelta.

Luvun lopussa esitellään Javan luokkamekanismin erityispiirteitä, joita ovat *sisäluokat* (*inner classes*) ja **`enum`**-literaaliluokat. Kummatkin ovat syntaktisesti yksinkertaisia: sisäluokka tarkoittaa sitä, että luokkamääritys voidaan kirjoittaa toisen luokan sisälle, ja literaaliluokka mahdollistaa luokan instanssien rajoittamisen luetteloon. Ohjelmointifilosofian kannalta näillä kummallakin on kuitenkin laajempaa merkitystä.

3.1 Esimerkki: `LukuJoukko`

Seuraavassa tarkastellaan esimerkkinä matemaattisen joukkokäsitteen toteuttavaa luokkaa. Tällainen on toki Java-kirjastossa valmiinakin (rajapinta `Set` ja sen implementoivat jälkeläiset `HashSet` ja `TreeSet`), mutta oletetaan, että asiat eivät olisi näin onnellisesti, vaan joudumme itse rakentamaan ko. luokan. Oletetaan myös että alkuperäisessä tehtäväksiannossa asiakaskuntaa on luonnehdittu vain lyhyesti: ”Ne haluaa laittaa ainakin kokonaislukuja joukkoon.”[2] Jotta tehtävä olisi mahdollisimman konkreetti, päätetään että koska joukon alkiot ovat kokonaislukuja, luokalle valitaan nimeksi `LukuJoukko`.

Luokan kehittäminen etenee usean vaiheen kautta. Tarkastellaanpa nyt kutakin vaihetta erikseen.

3.1.1 Piirteiden kartoittaminen

Ensimmäinen toimenpide on miettiä, mitä joukkoluokalta yleisesti vaaditaan eli mikä on luokan peruspiirrevalikoima. Samalla piirteet kannattaa luokitella kolmeen eri kategoriaan: *luontioperaatiot* (*constructor* tai *creation procedure*), *havainnoin-*

[1] Java-maailmassa viesteistä käytetään nimitystä rutiinikutsu.

[2] Luokan rajapintaa ei voi koskaan määritellä käyttökelpoiseksi tietämättä käyttöyhteyttä eli sitä millä tavoin luokan asiakkaat haluavat hyödyntää oliota omassa toiminnassaan.

tioperaatiot (*query function* tai *accessor*) ja *muunnosoperaatiot* (*modifier* tai *mutator*). Luontioperaatiot eli *konstruktorit* ovat tietysti niitä, joilla joukko-olio saadaan luotua joko ”tyhjästä” tai käyttämällä hyväksi jo olemassa olevia rakenteita. Havainnoijat kertovat jotain olion ominaisuuksista, mutta eivät muuta sen tilaa. Ne ovat siis funktioita, jotka eivät tee sivuvaikutuksia. Muunnosoperaatioiden ainoa tehtävä on muuttaa abstraktin olion tilaa, joten ne eivät palauta mitään arvoja (ovat proseduureja). Tämän jaon perusteella asiakas pystyy helpommin valitsemaan haluamansa piirteen ja seuraamaan oliossa tapahtuvia muutoksia. Periaate helpottaa siis asiakaskoodin lukemista.

Julkisen liitännän muodostamisessa käytetään apuna sitä matemaattista taustaa, joka on opetettu jo peruskoulussa. Sen nojalla tiedämme, että keskeisiä joukko-operaatioita ovat ainakin seuraavat:

Luontioperaatiot On selvää, että tarvitaan konstruktori, joka luo käsitteenäkin tärkeän tyhjän joukon. Tämän lisäksi voidaan tarvita erikoiskonstruktoreita, joilla luotu joukko alustetaan argumenttina annetu(i)lla tiedo(i)lla. Erityisesti saattaisi olla tarpeen toteuttaa luontioperaatio, joka tekee ainokaisjoukon (*singleton*) argumenttina annetusta kokonaisluvusta. Tämän lisäksi tarvittaisiin ehkä myös luontioperaatio, joka alustaa joukon argumenttina annetun taulukon kokonaisluvuilla.

Havainnoijat Tämä piirreryhmä on ehkä helpoin hahmottaa: tarvitaan funktiot, jotka palauttavat tiedon siitä, onko joukko tyhjä, kuuluuko annettu alkio joukkoon, onko annettu joukko toisen osajoukko ja mikä on joukon koko. Lisäksi voisi olla hyödyllistä testata leikkaako argumenttina annettu joukko **`this`**-joukkoa.

Muunnosoperaatiot Tähän kategoriaan kuuluvat tyypilliset sivuvaikutukselliset kahden joukon väliset binäärioperaatiot, kuten unioni, leikkaus ja erotus sekä joukon ja yksittäisen alkion väliset operaatiot, kuten alkion lisääminen joukkoon tai poistaminen joukosta. Lisäksi tarvitaan ehkä komplementtioperaatio, joka tuottaa **`this`**-joukon komplementin.

Tämä luettelo antaa vain alustavan hahmotelman julkiseen liitäntään mukaanotettavista operaatioista.

Huomaa, että kaikki luetellut piirteet ovat todellakin *operaatioiden* avulla toteutettavia. Tämä johtuu siitä, että Javan sallii jäsenmuuttujien tahallisen tai tahattoman muuttamisen olion ulkopuolelta, jos ne on varustettu **`public`**-määreellä.[3] Koska olion sisältö halutaan suojata ulkopuolisilta mahdollisimman hyvin, kaikki

[3] Järkevämpää olisi ollut antaa kielen määrittelyssä julkisiin jäsenmuuttujiin olion ulkopuolelta *vain lukuoikeus*. Kirjoitusoikeutta ei tulisi antaa lainkaan olion ulkopuolelle, jotta informaation piilottamisperiaate toteutuisi parhaalla mahdollisella tavalla.

jäsenmuuttujat on syytä piilottaa **private**-määreellä ja antaa asiakkaalle operaatiot, joilla hän pääsee (epäsuorasti) käsiksi tarvittaviin tietoihin.

Toinen merkittävä syy operaatiototeutuksiin päätymiseen on, että Java-kieli ei salli mutatoituvien jäsenmuuttujien kirjoittamista rajapintaluokkiin (rajapinnan jäsenmuuttujat ovat oletuksena aina **static final**). Jos luokan toteuttamasta käsitteestä on siis tehty abstrakti rajapinta, konkreetin perijän on luonnollista käyttää hyväkseen rajapinnan määrittelyjä eikä esitellä uudelleen saman semantiikan omaavaa piirrettä, tällä kertaa jäsenmuuttujana.

Muunnosoperaatioiden toteutuksiin ja samalla myös signatuureihin vaikuttaa oleellisesti se lähestymistapa, joka on valittu standardiksi siinä ympäristössä, missä ohjelmointityötä tehdään. Tästä lähemmin seuraavassa.

3.1.2 Operaatioiden määrittely

Kun operaatiot on saatu kartoitettua, niiden määrittelyt on kirjoitettava, jotta voidaan varmistua operaatioiden järkevästä rajauksesta ja niiden moitteettomasta yhteistoiminnasta. Tämä saadaan aikaan kirjoittamalla esimerkiksi rajapintaluokka, jossa piirteet on esitelty ja ajamalla se Java-kääntäjän läpi. Samalla tavoin voidaan testata useista luokista koostuvan systeemin toimivuus.

Ennen kun voidaan lopullisesti lyödä lukkoon julkiseen liitäntään tulevat operaatiot ja niiden signatuurit, pitää päättää toteutetaanko binäärioperaatiot puhtaan funktionaalisesti vai sivuvaikutuksia käyttäen. Edellinen toteutustapa viittaa tilanteeseen, missä esimerkiksi unioni-operaatio toteutetaan määräämällä **this**-joukon ja argumenttijoukon unioni ja palauttamalla näin saatu joukko asiakkaalle. Tällöin operaation signatuuri olisi funktiotyyppinen

```
public LukuJoukko unioni(LukuJoukko toinen)
```

Funktionaalisessa lähestymistavassa kukin operaatio palauttaa aina uuden tulosjoukon ja operaatioon osallistuneet joukot, **this** mukaanlukien, pysyvät muuttumattomina. Kyse onkin siis eräänlaisesta konstruktorista. Tavallisesti tällaiset binäärioperaatiot toteutetaan kuitenkin sivuvaikutusten avulla eli muuttaen **this**-joukkoa (vrt. listauksen 2.1 rajapintaa `Pino`). Unionin tapauksessa se tarkoittaa, että argumenttina annetun joukon alkiot lisätään **this**-joukkoon, joten operaation signatuuri olisi proseduurityyppinen

```
public void unioni(LukuJoukko toinen)
```

Ero näiden lähestymistapojen välillä on, että funktionaalisessa tapauksessa asiakas tietää operaatioiden toimivan puhtaan matemaattisesti, jolloin operaation kohteena oleva olio ja argumenttioliot eivät koskaan muutu operaatioiden soveltamisen seurauksena. Lisäksi funktionaalinen toteutus on asiakkaan kannalta aavistuksen käyttökelpoisempi kuin jälkimmäinen, sivuvaikutusten avulla toimiva versio, koska asiakas voi ”muuttaa” omaa `munLuvut`-joukkoaan kirjoittamalla

```
munLuvut = munLuvut.unioni(sunLuvut);
```

Jos asiakas ei halua muutoksia kumpaankaan joukkoon (`munLuvut` ja `sunLuvut`), sivuvaikutusten avulla toimiva ratkaisu ei sovellu lainkaan. Jos taas muutoksia saa tehdä vain toiseen, kutsu on kirjoitettava ”oikein päin” niin, että operaatio kohdistuu mutatoituvaan olioon.

Kaikki ohjelmointitekniset syyt puoltavat mutatoitumattomuutta eli funktionaalista lähestymistapaa. Ongelmana on, että mutatoitumattomuus maksaa tyypillisesti jonkin verran ajoajassa. Kun esimerkiksi joukkoon halutaan lisätä uusi alkio, on huomattavasti nopeampaa tallettaa muistiin tämä yksi alkio kuin muodostaa aiemmasta joukosta uusi, jossa kyseinen alkio on mukana. Tämäntyyppinen ”tehottomuus” lienee syynä esimerkiksi siihen, että kaikki Java-kirjaston kokoelmaluokat toimivat sivuvaikutuksia käyttäen. Perussääntönä voidaan silti pitää seuraavaa: *pyri mutatoitumattomuuteen aina, kun se on mahdollista.*[4] Funktionaalisuuden ansiosta merkintä `OLD(x)`, joka viittaa tunnisteen `x` arvoon rutiinin suorituksen alkaessa voidaan unohtaa, koska sitä tarvitaan vain sivuvaikutusten yhteydessä.

Yksittäisen alkion lisäys- ja poisto-operaatioiden määrittelyä kirjoitettaessa pitää ottaa kantaa siihen, mitä tapahtuu, jos lisättävä alkio on jo valmiiksi joukossa (poistettavaa ei ole). Joukkokäsitteen kannalta — ”joukossa on vain yksi esiintymä siihen talletetuista tiedoista” — tuntuu luonnolliselta, että lisäysoperaation ei tarvitse tehdä mitään, jos alkio on jo joukossa (saman voidaan katsoa olevan voimassa poistolle, jos alkio ei ole joukossa). Asiakkaan lienee helppo hyväksyä tämä valinta esimerkiksi poikkeuksen noston sijaan (päätös on kirjattu poisto-operaation loppuehtoon).

Tässä yhteydessä on hyvä huomata, että olioon kohdistuva symmetrinen binäärioperaatio, kuten kahden joukon unioni, joudutaan oliokielissä esittämään epäsymmetrisesti: joukkojen `a` ja `b` unionikutsuun, olipa kyseessä funktionaalinen tai sivuvaikutuksia käyttävä lähestymistapa, viitataan merkinnällä `a.unioni(b)`, vaikka `a + b` olisi koodin lukijan kannalta paljon luonnollisempi (tässä ”`+`” tarkoittaisi unionia). Java ei enemmän tai vähemmän valitettavasti salli käytettävän tällaista *syntaktista sokeria* (*syntactic sugar*), joka mahdollistaa symbolien käyttämisen operaationiminä ja niiden käytön tuttujen binäärioperaatioiden tavoin argumenttien välissä (vrt. vaikka kahden kokonaisluvun yhteenlasku). Ainoaksi vaihtoehdoksi korostaa operaation symmetrisyyttä Java-kielessä jää siis melko epäoliomainen tapa: luokkarutiinit

```
public static LukuJoukko unioni(LukuJoukko a, LukuJoukko b)
```

Kirjattu operaatiojoukko lienee melko kattava. Koska toteutusta ei ole syytä tehdä tässä vaiheessa turhan monimutkaiseksi, jätetään komplementin käsittely toteutta-

[4] Puhtaan funktionaalisilla kielillä se on ainoa mahdollisuuskin, koska ne eivät salli sivuvaikutuksia lainkaan.

matta. Komplementti voidaan ymmärtää implisiittisesti niiden alkioiden joukoksi, jotka eivät ole joukossa, mutta jotka alkion tyyppi **`int`** sallisi.

3.1.3 Julkisen liitännän kiinnittäminen

Joukkokäsitteen toteuttamiseen tarvittavat operaatiot on nyt käsitelty melko kattavasti. Luokkaan tulee lisäksi vielä olioparadigmaan liittyviä perusoperaatioita (ks. luku 6), joihin pitää ottaa kantaa. `LukuJoukko` perii `Object`-luokalta mm. operaatiot `equals` (havainnoija) ja `toString` (havainnoija), joihin palataan tarkemmin luvussa 6. Kun vielä tehdään päätös toteuttaa joukko-operaatiot funktionaalisesti, jolloin ne siirtyvät muunnosoperaatioiden kategoriasta luontioperaatioiden kategoriaan, saadaan julkisesta liitännästä seuraavanlainen:

(a) Luontioperaatiot:

- konstruktori, joka luo tyhjän joukon
- konstruktori, joka luo ainokaisjoukon annetusta kokonaisluvusta
- konstruktori, joka luo annetuilla kokonaisluvuilla alustetun joukon
- `unioni`, joka muodostaa ja palauttaa kahden joukon unionin
- `leikkaus`, joka muodostaa ja palauttaa kahden joukon leikkauksen
- `erotus`, joka muodostaa ja palauttaa kahden joukon erotuksen
- `lisää`, joka palauttaa uuden joukon, jossa ovat **`this`**-joukon alkiot sekä lisättävä alkio
- `poista`, joka palauttaa uuden joukon, jossa ovat kaikki muut **`this`**-joukon alkiot paitsi poistettava

(b) Havainnoijat:

- `onTyhjä`, joka testaa onko joukko tyhjä
- `sisältää`, joka testaa onko alkio joukossa
- `sisältää`, joka kertoo onko joukko **`this`**-joukon osajoukko
- `koko`, joka palauttaa joukossa olevien alkioiden lukumäärän

(c) Muut:

- `equals`, joka palauttaa tiedon siitä, ovatko kaksi `LukuJoukko`-oliota ”samanlaiset”
- `toString`, joka tekee merkkijonomuotoisen esityksen joukosta

Operaatiot on tässä aluksi vain nimetty ja jaoteltu ryhmiin, itse signatuurit ja täsmälliset määrittelyt esitetään vasta toteutuksessa. Samalla kun operaatiojoukko kiinnitetään, tehdään päätös tarkentaa niiden määrittelyä (rutiinien otsikoita) niin, että asiakas on tekemisissä ainoastaan **int**- ja `LukuJoukko`-tyyppisten tietojen kanssa. Tämä tuntuu itsestään selvältä, mutta valittu konkreetti toteutustapa johdattelee helposti muunkinlaisiin ratkaisuihin. Asiakas on pidettävä aina etusijalla!

3.1.4 Konkreetin esitystavan valinta

Näiden pohdintojen jälkeen voidaan edetä itse toteutukseen. Ensiksi luokalle pitää valita konkreetti esitystapa. Kaikki tietorakenteet, joilla voidaan ylläpitää homogeenista alkiokokoelmaa, ovat periaatteessa käyttökelpoisia. Ylläpito taas edellyttää, että esitystavan pitää pystyä simuloimaan joukon dynaamisuutta. Tämä ei ole normaalisti mikään ongelma, mutta jotkin esitystavat ovat tässä suhteessa joustavampia kuin toiset, kuten kohta nähdään. Tutuista tietorakenteista annettuun tehtävään sopivat ainakin taulukko, lista (jota edustaa vaikkapa Java-rajapinnan `List` toteuttajat) ja binäärinen hakupuu. Näistä viimeinen poikkeaa oleellisesti muista vaihtoehdoista, sillä hakupuu nimittäin edellyttää, että joukon alkioita voidaan verrata keskenään! Koska tällaista ominaisuutta ei yleisessä tapauksessa voida vaatia joukon alkioilta, tämä vaihtoehto on syytä sulkea pois ja keskittyä tutkimaan muiden soveltuvuutta konkreetiksi esitystavaksi. Vaikka siis joukon alkioiden tiedetäänkin olevan tyyppiä **int**, kokonaislukujen ominaisuuksia käyttävää toteutusta yritetään välttää, jotta luokkaa voitaisiin myöhemmin yleistää erityyppisille alkioille (ks. kohta 7.2).

Luokka `ArrayList` toteuttaa `List`-rajapinnan sisäisesti taulukon avulla, jolloin alkioiden suorasaanti on nopeaa. Tämän lisäksi lista on täysin dynaaminen eli sen koko voi muuttua, kun taas taulukon pituus pitää kiinnittää heti luonnin yhteydessä. Jos olisimme alunperin päätyneet sivuvaikutuksilla toimivaan joukkoon, lista olisi ilmeinen valinta konkreetiksi esitystavaksi; taulukkototeutuksella olisi paljon vaikeampi simuloida joukkoa, sillä joukon koon muuttuessa olion olisi ylläpidettävä tietoa taulukkoalueesta, jossa joukon alkiot sijaitsevat. Mutta koska joukosta on tarkoitus tehdä mutatoitumaton, käytetyn rakenteen ei välttämättä tarvitse olla dynaaminen. Voisimme toki heittää tässä vaiheessa lanttia listan ja taulukon välillä, mutta lisäperuste listan puolesta voisi olla se, että se tarjoaa käyttökelpoisemman liittymän kuin taulukko. Valitsemme siis listan.

3.1.5 Tietojen hallinta valitussa esitystavassa

Vaikka konkreetti esitystapa on kiinnitetty luokan `ArrayList`-tyyppiseksi, on vielä ratkaistava yksi ongelma. Joukko-operaatioiden toteuttamistapaan vaikuttaa nimittäin taustalla olevan tietorakenteen ominaisuuksien lisäksi vielä se, miten

alkioita ylläpidetään rakenteessa. Tämä informaatio kerrotaan luokan *luokkainvariantissa*, josta lisää kohdassa 3.2.2. Mahdollisia talletusvalintoja ovat ainakin seuraavat:

(a) Kokonaislukuja ei pidetä missään järjestyksessä. Sama luku voi esiintyä listassa useaan kertaan (vaikka loogisesti niitä on joukossa aina vain yksi).

(b) Lukuja ei pidetä missään järjestyksessä, kukin luku esiintyy listassa tarkalleen kerran (ei duplikaatteja).

(c) Luvut pidetään listassa suuruusjärjestyksessä, ei duplikaatteja.

Koska tarkasteltavana ei ole erityistä sovellusta, mihin joukkoja tarvitaan, operaatioita ei voida asettaa mihinkään tärkeysjärjestykseen. Tältä kannalta katsoen vaihtoehdot (a) ja (b) ovat yhtä hyviä. Heitettyämme kolikkoa, valintamme osuu jälkimmäiseen.[5]

3.1.6 Toteutukseen liittyvät päätökset

Nyt olemme valmiita siirtymään itse `LukuJoukko`-luokan toteuttamiseen. Seuraavassa joitakin kommentteja sen yksityiskohdista:

- Toteutus muistuttaa sitä liitäntää, joka on esitelty Java-rajapinnassa `Set`. Tästä syystä `LukuJoukko`-luokan voisi toteuttaa ko. rajapinnan (ja kenties periytyä luokasta `AbstractSet`) ja toteuttaa kaikki siinä esitellyt operaatiot. Periytymiseen paneudutaan perusteellisesti vasta kurssin toisessa osassa, joten `LukuJoukko` olkoon itsenäinen ja Javan olemassaolevasta luokkahierarkiasta riippumaton.

- Konkreetin esitysmuodon yhteyteen on syytä kirjoitettaa *luokkainvariantti*, joka kertoo esitysmuotoa sitovat säännöt. Invariantin merkityksestä kerrotaan lisää kohdassa 3.2.2.

- Jotta operaatioiden `lisää` ja `poista` ei tarvitsisi käsitellä toisten saman luokan olioiden **`private`**-tietoja, luokassa kannattaa toteuttaa sen omaan käyttöön tarkoitettu **`private`**-konstruktori. Esimerkiksi `lisää`-operaatiossa kokoelmajäsenmuuttujan tiedot voitaisiin kopioida ensin uuteen listaan, johon sijoitetaan myös uusi alkio. Näin saatu lista annetaan konstruktorin argumentiksi, joka asettaa annetun listan uuden olion arvoksi. Vastaavaa sovelletaan käänteisesti `poista`-operaatioon.

[5] Nollatiedolla tasaisesti jakautunut satunnaisuus on paras valintastrategia.

- Kahden joukon välistä erotusoperaatiota `erotus` voidaan käyttää hyväksi joidenkin operaatioiden loppuehdoissa. Esimerkiksi `leikkaus`-operaatio voitaisiin toteuttaa myös yksinkertaisesti loppuehdosta napatulla osalla

  ```
  return this.erotus(this.erotus(toinen));
  ```

  Tämäntyyppinen toiminta, missä operaatiot käyttävät hyväkseen toisiaan mahdollisimman paljon on suositeltavaa, koska se vähentää niiden riippuvuutta luokan sisäisestä esitysmuodosta. Tällöin on kuitenkin syytä muistaa, että julkista rutiinia kutsuva rutiini osallistuu aina perijäsopimukseen (ks. kohta 5.5.1).

- Olion sisäisen esitysmuodon eheyden ylläpitoon (suojelemiseen ulkopuolisilta muutoksilta) vaikuttaa erittäin merkittävästi se, että `Integer`-oliot (joiksi ja joista operatioissa annetut **`int`**-luvut automaattisesti kuorrutetaan ja kuoritaan) ovat mutatoitumattomia.

- Luokassa kannattaa esitellä iteraattori (`iterator`-rutiini), jonka avulla asiakas saa joukon alkiot käsiinsä yksi kerrallaan. Iteraattoreiden merkityksestä lisää kohdassa 8.3.

3.1.7 Yleistäminen

Luokka `LukuJoukko` kannattaa toteuttaa niin, että riippuvuudet sisäiseen esitysmuotoon minimoituvat. Syynä on tietysti se, että esitysmuoto tai sen merkitys saattaa muuttua, jos olion sisäisen tilan tulkintaa halutaan muuttaa. Esimerkiksi riippuvuus konkreettiin esitysmuotoon `ArrayList` vähenee, kun sitä käyttävät rutiinit ovat pakotettuja käyttämään `List`-rajapinnan tarjoamia välineitä, koska toteutuskokoelmaan viittaava jäsenmuuttuja on esitelty `List`-tyyppiseksi. Rutiinien toiminnan on pohjauduttava siis pelkästään siihen, että joukon alkiot on laitettu peräkkäin johonkin linearisoinnin toteuttavaan tietorakenteeseen. Jos sisäistä esitystapaa joudutaan myöhemmin muuttamaan joksikin muuksi peräkkäisrakenteeksi, vain luokan konstruktoreihin ja arvosemanttisiin rutiineihin tarvitsee tehdä muutoksia. Abstrahointia voitaisiin jatkaa tietysti pitemmällekin, esimerkiksi korvata `List` ylirajapinnalla `Collection`, mutta ongelmaksi tulee se, että mentäessä periytymishierarkiassa ylöspäin välineet kokoelman käsittelyyn saattavat kaveta niin paljon, että luokan rutiinit eivät enää selviydy tehtävistään niiden avulla. Tämän lisäksi operaatiot käyttävät mahdollisuuksien mukaan hyväkseen muita saman luokan rutiineja, jolloin ne välttyvät viittaamasta suoraan konkreettiin esitystapaan. Tällöin on kuitenkin muistettava pitää huolta että aliluokan määrittelyt eivät missään vaiheessa pääse hajoittamaan yliluokkien määrittelemiä eheysehtoja.

Tehtyä joukkoluokkaa voidaan yleistää helposti vaihtamalla alkiotyyppi **`int`** geneeriseksi tyypiksi (tästä lisää kohdassa 7.2). Toteutushan suunniteltiin sillä mie-

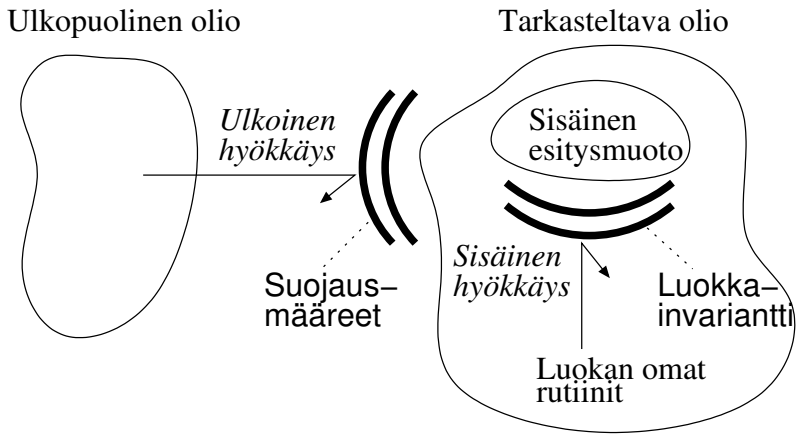
lellä, että **int**-tyyppisen tiedon erikoisominaisuuksia ei käytetä hyväksi, joten toteutus ei vaadi joukkoon talletettavilta alkioilta erikoiskykyjä. Itse asiassa geneerinen tyyppi yleistää joukkoa hyvinkin voimakkaasti, sillä nyt joukko voi olla heterogeeninen: sama joukko voi haluttaessa sisältää esimerkiksi reaalilukuja, pinoja, binääripuita tai joukkoja.

Yleistystä voidaan tehdä myös toisen ulottuvuuden, kokoelmatyypin, suhteen. Joukon yhteydessä luonnollinen yleistys olisi *monijoukko* (*multiset* tai *bag*), joka toimii muuten joukon tavoin, mutta ylläpitää tietoa siitä, kuinka monta kertaa kukin alkio esiintyy joukossa. Mikäli tämä yleistys tehdään periytymisellä, saatetaan päätyä ristiriitaiseen määrittelyyn. Viimekädessä luokkien yleistys/erikoistus -suhde määrittyy järjestelmään kohdistuvien voimien mukaan; on olemassa tapauksia joissa on järkevää toteuttaa em. käsitteiden suhde toisella tavalla.

3.2 Sisäisen esitysmuodon eheys

Olion *tila* (*state*) määräytyy niistä arvoista, jotka on tiettynä hetkenä talletettu olion jäsenmuuttujiin (sisäiseen esitysmuotoon). Jotta olio edustaisi laillista abstraktia käsitettä, olion tilan tulee täyttää sille asetetut vaatimukset (esimerkiksi `LukuJoukko`-oliossa ei saa esiintyä kokonaislukujen duplikaatteja). Mikäli tilatietoja muutetaan hallitsemattomasti, olion *sisäisen esitysmuodon eheys* (*integrity*) tuhoutuu ja siitä tulee näin muodoin käyttökelvoton jatkoprosessoinnin kannalta. Tila voi korruptoitua kahdesta syystä (ks. kuva 3.1): (a) muille olioille annetaan mahdollisuus päästä käsiksi olion sisäiseen tietoon (ulkoiset uhat) ja (b) luokan omat rutiinit jättävät tiedot laittomaan tilaan (sisäiset uhat). Edellinen pyritään estämään *suojausmääreillä* (*access modifiers*), jälkimmäinen *luokkainvariantin* (*class invariant* tai *representation invariant*) avulla (ks. kuva 3.2).

Ulkoiset uhat Olioajattelun keskeinen periaate on tiedon piilottaminen: olion sisäinen tila tulee *kapseloida* (*encapsulate*) niin, että kukaan ulkopuolinen ei pääse muuttamaan sen tietosisältöä muuta kuin julkiseen liitäntään kuuluvien operaatioiden kautta. Ohjelmointikielen tulisi siis jollakin mekanismilla tukea tällaista toimintaa yksinkertaisesti jo siitä syystä, että vain olio itse tietää, minkälaista tietoa sen jäsenmuuttujissa tulee olla ja minkälaisia sääntöjä niiden käytölle on. Viittaaminen toisen olion sisäisiin tietoihin on haitallista myös asiakkaan kannalta, sillä se muodostaa vahvoja kytköksiä asiakas- ja toimittajaluokkien välille. Jos nimittäin toimittajaluokan toteuttaja haluaa muuttaa jäsenmuuttujien nimiä, näkyvyyttä tai merkitystä, kaikkien asiakkaiden on muutettava koodiaan vastaavasti.



Kuva 3.1: Sisäisen esitysmuodon ulkoiset ja sisäiset uhat.

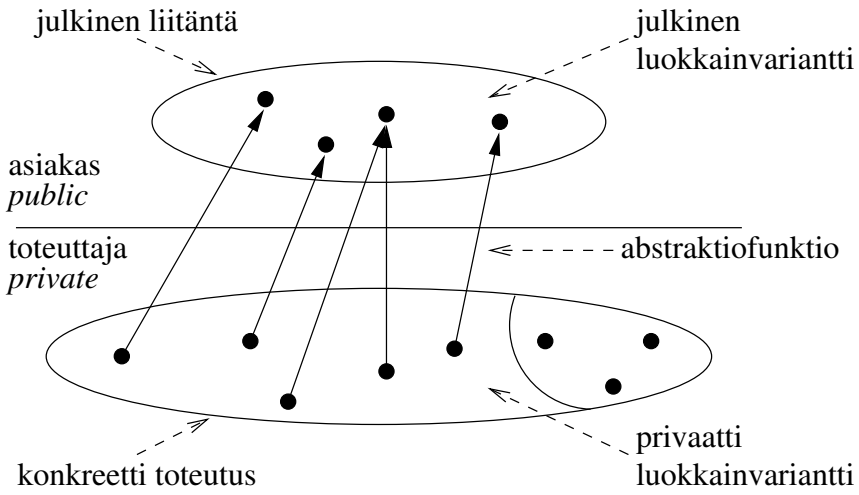
Sisäiset uhat Paitsi ulkopuolelta, oliolle ”vihamielisiä” tahoja voi löytyä myös sen sisältä: yksi tai useampia operaatiototeutuksia (joko luokan omia tai sen perillisessä määriteltyjä) voi toimia väärin niin, että sisäiselle esitysmuodolle asetetut vaatimukset rikkoutuvat. Esimerkiksi luokka `LukuJoukko` toteutettiin sitä periaatetta noudattaen, että listassa pidetään kustakin alkiosta vain yksi kopio. Jos operaatio `lisää` ei tiedä tätä, se voi lisätä alkion listaan tekemättä ensin tarkistusta, jolla selvitetään, onko lisättävä alkio listassa jo ennestään. Näin toimiessaan `lisää` vie pohjan kaikilta muilta operaatioilta. Esimerkiksi `koko` palauttaa virheellisen toiminnan vuoksi liian isoja arvoja, `poista` ei enää poista oikein jne. Säännöt, jotka ilmaisevat sisäiseen esitysmuotoon liittyvät vaatimukset, kirjataan luokkainvarianttiin, joka on normaali totuusarvoinen väittämä.

3.2.1 Suojausmääreet

Java tarjoaa kapselointimekanismiksi suojausmääreet, joilla hallitaan piirteiden näkyvyyttä asiakasluokkien lisäksi myös perijäluokkiin päin (ks. kohta 5.5). Suojausmekanismi on kaksitasoinen: ensin tarkistetaan, että asiakkaalla on pääsy haluamaansa luokkaan, ja sitten oikeudet tämän luokan piirteen käyttöön. Javan tarjoamat suojausmääreet ovat **`public`**, **`protected`** ja **`private`**. Suojausmääre voidaan jättää myös kokonaan kirjoittamatta. Tähän tilanteeseen viitataan tästä lähin sanalla *package*, joka kertoo näkyvyyden laajuuden tässä tapauksessa.[6]

Javan pakkaukset (*packages*) kokoavat loogisesti yhteen kuuluvia luokkia samaan ryhmään. Pakkaukset ovat kaikki kuitenkin samanarvoisia siinä mielessä, että kaikki voivat käyttää kaikkia pakkauksia (ts. niihin ei voi liittää suojausmää-

[6]Huomaa, että **`package`**-avainsanalla on Java-syntaksissa aivan oma semantiikkansa.



Kuva 3.2: Olion tila näkyy ulospäin (so. asiakkaalle) julkisen liitännän kautta. Siinä määriteltyjen rutiinien avulla voidaan ilmaista olion julkinen luokkainvariantti. Olion tila näkyy sisäänpäin (so. toimittajalle) konkreettina toteutuksena, jota privaatti luokkainvariantti rajaa. Abstraktiofunktio on kuvaus konkreetista toteutuksesta julkisessa liitännässä näkyvään käsitteeseen.

reitä). Luokkatasolla suojausmääreitä käytetään yleensä hyvin rajoitetusti; määreiden pääasiallisena käyttökohteena ovatkin siis yksittäiset piirteet. Itse kieli ei aseta mitään vaatimuksia piirteiden, erityisesti jäsenmuuttujien, käytölle: asiakas, jolle on annettu piirteen lukuoikeus, on siihen myös kirjoitusoikeus.[7]

Luokka (tai tarkemmin sanoen luokan otsikko) voidaan varustaa millä tahansa suojausmääreellä. Yleisemmin käytössä ovat *package* ja **`public`**, joista edellinen kertoo, että vain samassa pakkauksessa olevat "näkevät" kyseisen luokan ja jälkimmäinen, että luokka on kaikkien käytössä. Määreitä **`protected`** ja **`private`** käytetään vain sisäluokkien yhteydessä (ks. kohta 3.3.1).

Karkeasti ottaen piirteiden suojausmääre **`public`** on kohdistettu asiakkaisiin; määritelläänhän luokan julkinen liitäntä niiden piirteiden joukkona, jotka on leimattu **`public`**-määreellä. Määre **`protected`** on vastaavalla tavalla suunnattu perijöille ja **`private`** on tiukasti luokan omaa sisäistä toimintaa varten. Suojausmääreiden merkitys on kerrottu tarkemmin taulukossa 3.1: vasen sarake kertoo, mistä piirrettä yritetään käyttää ja rastit luvallisten viittausten määreet.

Taulukko 3.1 kertoo selvästi sen, että pakkaus on tärkeä käsite suojauksen kannalta: samassa pakkauksessa olevilla on huomattavasti oikeuksia; ainoastaan **`private`**-piirteet ovat sellaisia, joita pakkauksen muut luokat eivät näe.[8] Erityi-

[7]Käytäntö vaihtelee eri kielissä. Esimerkiksi Eiffel sallii vain julkisen jäsenmuuttujan lukemisen, mutta ei sen arvon muuttamista.

[8]Samassa pakkauksessa olevien luokkien ajatellaan siis olevan toisilleen ystävällismielisiä (*pac-*

viittauspaikka	**private**	*package*	**protected**	**public**
luokka itse	×	×	×	×
samassa pakkauksessa oleva				
perijä		×	×	×
ei-perijä		×	×	×
eri pakkauksessa oleva				
perijä			×	×
ei-perijä				×

Taulukko 3.1: Suojausmääreiden vaikutusalueet.

sesti siis tietyn luokan oliot voivat viitata suoraan muihin saman luokan olioiden **protected**-piirteisiin. Tämä voidaan ymmärtää siten, että *package*-suojausmääre on rinnakkainen muihin suojauksiin nähden, sillä luokka on aina jossain pakkauksessa.

Itse asiassa Java menee tässä vieläkin pidemmälle: saman luokan oliot voivat voivat käpistellä vapaasti toistensa *kaikkia piirteitä*, suojausmääreestä riippumatta (siis myös **private**-määreellä varustettuja). Saman luokan oliot ovat toisilleen siis tosi ystävällisiä! *Vältä tällaisen ominaisuuden käyttöä, vaikka kieli sen salliikin.* Hyvin suunnitellussa luokkakokonaisuudessa et tule koskaan tarvitsemaan sitä muualla kuin kenties vain arvosemanttisissa rutiineissa (ks. luku 6).

Julkisen liitännän muodostaminen

Kun suunnittelet luokan julkista liitäntää, mieti tarkkaan mitkä piirteet ovat julkisia. Käytännössä *olion konkreetti esitystapa on syytä aina piilottaa asiakkailta.* Myös luokan oliolle määrittelemät jäsenmuuttujat, vaikka ne tuntuisivat kuinka olennaisilta ja muuttumattomilta käsitteiltä ko. luokkaa ajatellen, on syytä piilottaa.

Esimerkki 3.1 Tarkastellaan xy-koordinaatiston pistettä kuvaava luokkaa

```
public class Tasopiste
  {
  public double x;
  public double y;
  public Tasopiste(double x, double y) { this.x = x; this.y = y; }
  }
```

Luokka tarjoaa asiakkailleen mahdollisuuden muuttaa koordinaattiarvoja olion ulkopuolelta. Mikäli asiakas näin tekee, se kytkeytyy vahvasti `Tasopiste`-luokkaan. Tämä tarkoittaa tietysti sitä, että jälkimmäiseen tehdyt muutokset — joko jäsenmuuttujien nimissä

kage friendly), kuten C++:n toistensa ”frendeiksi” määritellyt luokat.

tai merkityksessä — implikoivat muutoksia myös kaikkiin asiakkaisiin. Ei hyvä, ei todellakaan. Siksi kaikki jäsenmuuttujat tulee varustaa **private**-määreellä. Asiakkaille annetaan mahdollisuus lukea ja muuttaa jäsenmuuttujien arvoja rutiineilla, joiden nimet alkavat standardietuliitteillä `anna` ja `aseta` (ks. listaus 3.1).[9]

Esimerkissä esitellyn menettelyn tuloksena luokan julkisessa liitännässä on vain rutiineja eikä ainuttakaan jäsenmuuttujaa. Kaikkia jäsenmuuttujiin liittyviä `anna`- ja `aseta`-operaatioita ei tarvitse aina toteuttaa, ainoastaan tarpeen mukaan. Mikäli asiakkaalle ei haluta antaa oikeutta muuttaa jotakin jäsenmuuttujaa — esimerkiksi sen takia, että jäsenmuuttuja halutaan pitää mutatoitumattomana — jätetään vastaava `aseta`-proseduuri pois luokan julkisesta liitännästä (tämä ei estä sitä, etteikö rutiinia voisi toteuttaa ja käyttää sisäisesti luokan muiden operaatioiden apuna). Jos asiakkaalle ei tarjota ainuttakaan `aseta`-operaatiota, luokan mukaiset oliot ovat mutatoitumattomia: olion tila asetetaan luonnin yhteydessä, eikä sitä voi myöhemmin muuttaa. Esimerkiksi luokka `Integer` on tällainen.

Pakottamalla asiakas käyttämään `anna`/`aseta` -toimintoja jäsenmuuttujien arvoihin kohdistuviin muutosoperaatioihin saadaan upotettua lisäkontrollia. Esimerkiksi `aseta`-rutiineihin voidaan lisätä alkuehtoja, joilla tarkistetaan muutosten järkevyys. Tämän lisäksi rutiinin rungossa voidaan tarkistaa vaikkapa kutsujan oikeudet tehdä muutoksia. Jos muutokset ovat aina hyvin tarkkaan kiinnitettyjä, ei asiakkaalle anneta lainkaan mahdollisuutta vaikuttaa muutoksen laatuun muuta kuin niitä varten tehtyjen operaatioiden avulla.

Esimerkki 3.2 Kokonaislukulaskuri, joka kasvaa tai pienenee aina vain yhdellä, varustetaan operaatioilla `kasvata` ja `pienennä`. Jos asiakkaiden sallittaisiin muuttaa laskurijäsenmuuttujaa olion ulkopuolelta, voitaisiin siihen kohdistaa kaikkia operaatioita, jotka jäsenmuuttujan tyyppi (**int**) sallii.

Kun tiedon kapselointi ja sen piilottaminen on tehty asianmukaisesti, asiakas ja toimittaja saadaan riippumattomiksi toisistaan: jos sisäinen esitysmuoto myöhemmin muuttuu, riittää usein, että `anna`/`aseta` -rutiinit korjataan tilannetta vastaavaksi, eivätkä asiakkaat huomaa mitään. Yleisohjeena suojausmääreiden käytöstä voidaan sanoa seuraavaa:

- Luokan oliolle määrittelemien jäsenmuuttujien tulisi olla aina **private** siitäkin huolimatta, että luokalle tiedettäisiin tulevan perijöitä.

- Rutiinit, jotka on tarkoitettu vain luokan sisäiseen käyttöön, tulisi olla varustettu **protected**-suojausmääreellä, jotta perijät voisivat käyttää niitä hyväkseen.

[9] Olion attribuutti ja olion jäsenmuuttuja ovat aivan eri käsitteitä: attribuutti on asiakkaalle näkyvä olion tila ja jäsenmuuttuja toteuttajan tapa määritellä olion rakenne. ”Hyvällä tuurilla” attribuutti voidaan toteuttaa esim. yhdellä jäsenmuuttujalla.

Listaus 3.1: Luokka `Tasopiste`.

```
public class Tasopiste
  {
  private double x;
  private double y;

  public Tasopiste(double x, double y) { this.x = x; this.y = y; }

  //-- Havainnoijat
  /**
   *  @.pre  true
   *  @.post RESULT == (x-koordinaatti)
   */
  public double annaX() { return x; }

  /**
   *  @.pre  true
   *  @.post RESULT == (y-koordinaatti)
   */
  public double annaY() { return y; }

  //-- Muunnosoperaatiot
  /**
   *  @.pre  true
   *  @.post annaX() == uusiX
   */
  public void asetaX(double uusiX) { x = uusiX; }

  /**
   *  @.pre  true
   *  @.post annaY() == uusiY
   */
  public void asetaY(double uusiY) { y = uusiY; }
  }
```

- Konstruktoreiden tulisi yleensä olla **`public`**. Suojausmääre **`private`** liitetään konstruktoriin vain harvoin, koska sillä on kauaskantoisia seurauksia. Ensinnäkin jos luokan kaikki konstruktorit ovat **`private`**-tyyppisiä, asiakas ei voi luoda siitä olioita, koska kääntäjän generoimaa oletuskonstruktoria voidaan soveltaa vain, jos luokassa ei ole esitelty yhtään konstruktoria. Toiseksi tällä luokalla ei voi olla perijöitä, koska perijän luontioperaatio kutsuu *aina* yliluokan konstruktoria, mutta yliluokka ei anna sellaista (ja Javan generoima oletuskonstruktori on käytössä vain, jos luokkaan ei ole kirjoitettu yhtään luontioperaatiota). Määrettä **`protected`** käytetään konstruktorin yhteydessä yleensä kun halutaan kontrolloida sen parametreja omilla mekanismeilla.

Julkisen liitännän kautta välitettävä tieto

Olioajattelun peruskiviä on olioiden *jakaminen* (*sharing* tai *aliasing*): samaan olioon voi olla useita viittauksia eri paikoista. Kärjistetysti voidaan sanoa, että mitä syvällisemmin ymmärtää olion jakamiseen liittyviä hyötyjä, haittoja, ongelmia ja menetelmiä, sitä enemmän käsittää kuinka olioilla rakennetaan järjestelmiä ihan oikeasti. Jakaminen ei ole toimintaa, joka saadaan aikaan vain tarkoituksellisesti tekemällä ohjelmassa siihen tähtäävät toimenpiteet. Se on osa viittausmekanismia ja liittyy erityisesti kutsuttavalle rutiinille välitettäviin ja siitä palautettaviin tietoihin, käytetäänhän kummassakin tiedonvälitystavassa hyväksi asetuslauseen semantiikkaa.

On helppo saada aikaan tilanne, jossa olion sisäisen esitystavan osa on jaettu ulkopuolisten tahojen kanssa. *Jos jaettu tieto on mutatoitumaton, tässä ei ole mitään ongelmaa*, jollei olioon viittauksella ole jotain lisämerkitystä. Mutatoituvaa tietoa voi kuitenkin mennä muuttamaan kuka tahansa, jolla on pääsy jaettuun olioon. Koska muutosoikeudet olisi mukavaa antaa ainoastaan sille oliolle, jonka sisäisen esitystavan osana jaettu tieto on, tulisi kielessä olla mahdollisuus merkitä olion ”omistaja”, jolta muut voisivat kysyä lupaa mahdollisiin muutoksiin. Javaan, sen kummemmin kuin muihinkaan (yleisiin) oliokieliin ei tällaista jakamiseen liittyvää perusmekanismia ole integroitu.

Luokan konstruktoreiden ja havainnointioperaatioiden yhteydessä syntyy (melkeinpä huomaamatta) jaettuja olioita, joten muutama sana niiden toteutuksesta lienee paikallaan (toki sisäiseen esitysmuotoon voidaan ottaa osia muidenkin rutiinien kuin konstruktoreiden kautta välitettävistä tiedoista). Luontioperaation sekä `aseta`-operaatioiden argumenttien kautta välitetään usein mutatoituvaa viittaustyyppistä tietoa, joka tulee osaksi olion sisäistä esitystapaa. Jos operaatiossa tehdään pelkkä asetuslause, jolla argumenttiviittaus kopioidaan sisäisen jäsenmuuttujan arvoksi, tästä tiedosta tulee jaettu.

Esimerkki 3.3 Tarkastellaanpa seuraavaa luokkaa:

```
public class Ympyrä
  {
  private Tasopiste keskipiste;
  private double säde;

  public Ympyrä(Tasopiste p, double r) { keskipiste = p; säde = r; }

  public Tasopiste annaKeskipiste() { return keskipiste; }
  }
```

Tällaista oliota luotaessa asiakas antaa viittauksen `Tasopiste`-luokan olioon joka talletetaan osaksi `Ympyrä`-olion sisäistä esitysmuotoa. Tämän seurauksena sekä asiakas että sisäinen jäsenmuuttuja `keskipiste` jakavat saman `Tasopiste`-luokan olion. Asiakas voi siis liikuttaa ympyrää paikasta toiseen `Ympyrä`-olion sitä tietämättä. Jäsenmuuttujan `säde` kohdalla ei ole samanlaista ongelmaa, koska primitiivityypin arvo kopioidaan konstruktorin kutsun yhteydessä, joten sisäinen esitysmuoto käsittelee aina omaa lokaalia kopiotaan, johon muilla ei ole pääsyä. Koska konstruktorin kutsuja voi olion ulkopuolelta muuttaa (tahallisesti tai tahattomasti) konkreettia esitysmuotoa, mutatoituvat argumenttitiedot on yleensä syytä kopioida (esim. `clone`-operaatiolla, ks. kohta 6.4), jotta luokka saisi niistä omat versionsa yksityiskäyttöön.

Havainnoijafunktio `annaKeskipiste` antaa asiakkaille pääsyn olion sisäisen esitystapaan. Myös tällöin on mietittävä, kannattaisiko asiakkaalle viittauksen sijaan antaa kopio oliosta. Jos kyseessä on kokoelmaolio, kopiointi voidaan välttää kolmellakin eri tavalla: antamalla asiakkaalle vain *read-only* -näkymä olioon; luomalla muutokset estävä edusolio luokasta `Collections` löytyvillä jäädytyskuorirutiineilla (esimerkiksi `unmodifiableList`) tai palauttamalla tietoja yksi kerrallaan iteraattorin avulla (näistä lisää luvussa 8).

Tarinan opetus: vaikka kielen tarjoamia suojauksia kiristettäisiin kuinka paljon, olio-ohjelmoinnille luontainen jakaminen antaa muille koodinosille mahdollisuuden päästä käsiksi arkaluonteiseen tietoon helposti, mikäli ”vuotopaikkoja” ei tiedosteta ja tukita.

3.2.2 Luokkainvariantti

Oletetaan, että olion sisäiseen eheyteen kohdistuvat ulkoiset hyökkäykset on saatu hoidettua riittävän tehokkailla suojausmääreillä (ja mahdollisella kopioinnilla). Tämän lisäksi on pidettävä huoli siitä, että olio suojautuu myös itseltään eli sen oma toiminta noudattaa niitä sääntöjä, joiden oletetaan olevan voimassa sisäisen esitysmuodon ylläpidossa. Tärkeintä on kirjata konkreettiin esitystapaan liittyvät päätökset vaikkapa kommenteiksi luokkatekstiin ja mahdollisuuksien mukaan myös tarkistaa niiden voimassaolo ajoaikana.

Luokkainvariantti määritellään, kun konkreetti esitystapa on valittu ja sen käyttötapaan liittyvät säännöt on päätetty. Se kiinnitetään siis ennen piirteiden toteutusta, jotta ne tulisivat toimimaan kitkattomasti yhdessä. Luokkainvariantti

on konkreetin esitysmuodon globaaleja ominaisuuksia esittelevä väittämä ja siten yhteinen kaikille luokassa määritellyille julkisille operaatioille. Tämän takia invariantin tulisi olla voimassa aina ennen ja jälkeen jokaisen julkisen operaation suorituksen.[10] Mikäli oliot ovat tiukassa toteutuksellisessa kytköksessä toisiinsa, tätä sääntöä on yllättävän helppo rikkoa vahingossa.[11] Poikkeuksena ovat konstruktorit, joille luokkainvariantti on voimassa vasta suorituksen jälkeen. Tämä onkin luontioperaatioiden perustehtävä: jos jäsenmuuttujille annetut oletusarvot eivät täytä luokkainvarianttia, tarvitaan konstruktori, joka asettaa niille alkuarvot, joilla invariantti saadaan voimaan.

Luokkainvariantti vahvistaa sekä rutiinien alkuehtoja että loppuehtoja. Luokkainvariantti pidetään kuitenkin alku- ja loppuehdoista erillään, koska se on koko oliota eikä yksittäisiä rutiineja karakterisoiva käsite. Tämän lisäksi privaatissa luokkainvariantissa voidaan sitoa toisiinsa esitysmuodon ja julkisten operaation väliset yhteydet

Esimerkki 3.4 `LukuJoukko`-luokan toteutus nojautui siihen, että joukon alkioista ei ole duplikaatteja listassa. Tällä on tietenkin välitön vaikutus joukko-operaatioiden toteutukseen. Myös luokan ylläpitäjän on tunnettava nämä säännöt, koska muuten hän voi tehdä muutoksia, joiden tuloksena luokan oliot eivät enää käyttäydy halutulla tavalla. Luokkainvarianttiin voidaan koota tietoja luokan jäsenmuuttujien, funktioiden sekä konkreetin esitystavan välisistä relaatioista. `LukuJoukko`-luokan luokkainvariantti on esitelty listauksessa 3.2, jossa jäsenmuuttuja `lista` viittaa toteutustietorakenteeseen.

Kun päätetään paikkaa, johon tietty väittämä kirjoitetaan, voidaan peukalosääntönä sanoa, että operaation loppuehtoon kirjoitetaan informaatio, joka on merkityksellistä asiakkaalle, ja luokkainvarianttiin kerätään ylläpitäjälle tärkeät väittämät. Tämän jaon seurauksena asiakas ymmärtää loppuehdot luokan abstraktien ominaisuuksien perusteella ja luokan implementoija näkee luokkainvariantista konkreettiin esitystapaan liittyvät seikat pohtiessaan muutosten tai lisäysten vaikutuksia olion eheyteen. Luokkainvariantin käyttö johtaa myös siihen, että loppuehtoon joudutaan kovin harvoin sijoittamaan asiakkaille merkityksettömiä, konkreettia esitystapaa koskevia väittämiä (vrt. kohta 2.3.2).

Luokkainvarianttiin kirjoitetaan myös (sisäisten ja) julkisten *jäsenmuuttujien semantiikka*, niiden ”loppuehdot”. Invariantti pyritään pitämään mahdollisimman suppeana, joten siihen ei kannata kirjoittaa ehtoja, jotka voidaan johtaa muista väittämistä ellei niitä haluta erityisesti korostaa.

Huolimatta siitä, että luokkainvariantissa kerrotaan tarkasti sisäistä esitysmuotoa koskevat pysyväissäännöt, se ei kuitenkaan kerro lukijalle, miten tietoja pitää

[10] Huomaa pieni mutta selkeä ero julkisen ja ei-julkisen operaation välillä: **`private`**/**`protected`** -piirteitä voidaan käyttää apuna julkisten piirteiden toteutuksessa, mutta edellisten ei tarvitse välttämättä täyttää tätä ehtoa, riittää kun julkiset piirteet täyttävät sen.

[11] Sääntöä voidaan toki heikentää, mutta tämä ei kuulu enää kurssin aihepiiriin.

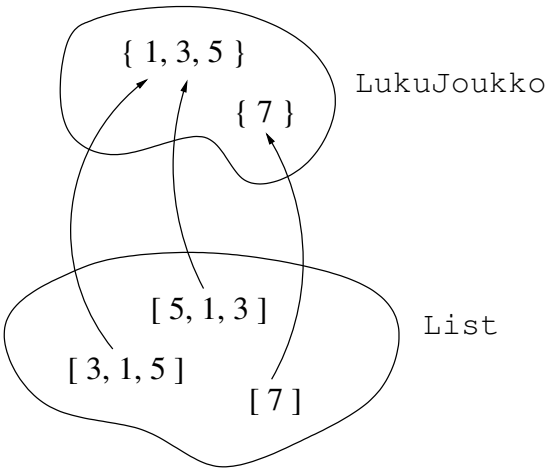
Listaus 3.2: Luokan `LukuJoukko` luokkainvariantti ja abstraktiofunktio.

```
/**
 *  LukuJoukko mallintaa kokonaislukujen matemaattisen joukon
 *  { x | this.sisältää(x) }.
 *
 *  @.classInvariant
 *      FORALL(a : this; !EXIST(b : this; a.equals(b) & a != b)) &
 *      FORALL(a : this; a.getClass().equals(Integer.class))
 *
 *  @.classInvariantPrivate
 *      (kokonaislukualkiot on talletettu jäsenmuuttujaan lista) &
 *      (koko() == lista.size())
 *
 *  @.abstractionFunction
 *      LukuJoukko on { x | lista.contains(x) }.
 */
public class LukuJoukko implements Iterable<Integer>
```

tulkita, jotta päästäisiin vastaavaan abstraktiin olioon. Tulkintatapa on kuitenkin tärkeää tietoa myös luokan ylläpitäjälle jo pelkästään siitä syystä, että konkreettien esitystapojen erilaiset tilat voivat edustaa samaa abstraktia oliota. Esimerkkejä `LukuJoukko`-esitystapojen ja niiden kuvaamien abstraktien olioiden (kokonaislukujoukkojen) välisistä vastaavuuksista nähdään kuvasta 3.3.

Tulkintaa konkreetista esitystavasta abstraktiksi olioksi kutsutaan *abstraktiofunktioksi* (*abstraction function*). Se kertoo, miten jäsenmuuttuja-arvojen muodostaman kokonaisuuden on ajateltu esittävän abstraktia oliota. Abstraktiofunktio olisi hyvä esitellä luokkainvariantin yhteydessä, jolloin se auttaa toteutuksen lukijaa ymmärtämään, miten alkuperäinen implementoija on ajatellut kuvauksen toimivan. Siinä voi käyttää abstraktille käsitteelle ominaista notaatiota. Esimerkiksi `LukuJoukko`-luokan abstraktiofunktio on esitelty listauksessa 3.2.

Abstraktiofunktio on todellakin matemaattinen funktio siinä mielessä, että jokainen luokkainvariantin toteuttava konkreetti esitystapa vastaa tarkalleen yhtä abstraktia oliota. Tämä on tietysti oleellinen vaatimus: jos yksi esitystapa voisi vastata useaa eri abstraktia oliota, esitystapa olisi monimielinen ja siksi vaillinainen. Abstraktiofunktio ei kuitenkaan ole välttämättä totaalinen, ts. *aivan kaikilla konkreeteilla esityksillä ei ole aina tulkintaa joksikin lailliseksi abstraktiksi olioksi.* Luokkainvariantti kertoo osittaisen abstraktiofunktion määrittelyalueen eli ne ehdot, jotka konkreetin esitystavan on täytettävä, jotta abstraktiofunktiota voidaan soveltaa. Lopuksi vielä tärkeä huomio: jos useat eri esitystavat voivat vastata sa-



Kuva 3.3: `LukuJoukko`-luokan abstraktiofunktio on kuvaus konkreettisesta toteutuksesta (kokonaislukulistasta) abstraktiin käsitteeseen (kokonaislukujoukkoon).

maa abstraktia oliota, olion sisäiseen esitystapaan voidaan tehdä muutoksia ilman, että asiakkaat huomaavat abstraktin olion käyttäytymisessä muutoksia. Tämä on edellytys mm. olion sisäisen rakenteen ajoaikaiselle optimoinnille.

3.2.3 Sivuvaikutuksista

Aiemmin esiteltiin periaate, jonka mukaan luokan rutiinit kannattaa jakaa toimintansa nojalla kahteen eri kategoriaan: havainnoijafunktiot palauttavat tietoa olion abstraktista tilasta muuttamatta sitä ja muunnosproseduurien ainoa tehtävä on muuttaa olion tilaa. Havainnoijien halutaan siis käyttäytyvän matemaattisten funktioiden tavoin: jos matemaatikko näkee lausekkeen $\sqrt{x}$, hän tietää saavansa luvun x neliöjuuren. Hän tietää myös, että muuttujan x arvo ei muutu laskennan tuloksena, se on neliöjuuren oton jälkeenkin sama x. Ohjelmoija ei ole näin onnellisessa asemassa, koska rutiini voi muuttaa argumenttejaan sivuvaikutuksena.[12] Jos ohjelmoija noudattaa em. sääntöä kurinalaisesti, hän kirjoittaa esimerkiksi luokkaan `Pino` eri operaatiot pinon päällimmäisen alkion palauttamiseksi (`päällimmäinen`) ja sen poistamiseksi (`poista`). Tämän seurauksena ohjelman lukeminen ja ymmärtäminen on paljon helpompaa, koska silloin tiedetään tarkalleen, milloin abstraktin olion tila voi muuttua. Näin staattisen ohjelmatekstin ja ohjelman dynaamisen suorituksen välinen ero pienenee.

[12]Mainittakoon selvyyden vuoksi, että esimerkki on hieman harhaanjohtava. Java käyttää arvoparametriperiaatetta rutiinikutsussa, joten primitiivityypin tiedot eivät voi muuttua kutsun seurauksena. Sen sijaan viitatavat oliot voivat hyvinkin muuttua. Edes määre **`final`** ei auta, koska se takaa vain, että viittausta itseään ei saa mennä muuttamaan, viittauksen kohdetta voi.

Esimerkki 3.5 Katsotaanpa mihin homma voi johtaa, jos sivuvaikutukset sallitaan havainnoijissa. Havainnoijafunktio

```
public int luku()
  {
  ++arvo;
  return arvo;
  }
```

palauttaa jäsenmuuttujan `arvo` sisällön ja muuttaa sitä sivuvaikutuksenaan. Jos `arvo` sisältää alkujaan nollan, lauseke `2 * luku()` saa arvokseen 2, kun taas `luku() + luku()` saa arvokseen 3. Ei kovin mukavaa, eihän?

Funktioiden sivuvaikutukset voitaisiin estää kokonaan (kääntäjä pystyy havaitsemaan ne), mutta tähän ei kuitenkaan haluta normaalisti mennä, koska jotkin sivuvaikutukset ovat harmittomia ja toiset jopa hyödyllisiä. Sivuvaikutuksia on kahdenlaisia: *abstrakteja* ja *konkreetteja*. Abstraktit sivuvaikutukset ovat sellaisia, jotka luokan asiakas havaitsee abstraktin olion käyttäytymisessä ja ne ovat juuri niitä, joista halutaan päästä kokonaan eroon. Sen sijaan konkreetit sivuvaikutukset eivät ole asiakkaan havaittavissa, mistä syystä ne eivät ole kriittisiä. Abstraktiofunktiosta puhuttaessa todettiin, että luokkainvariantti voi sallia useita erilaisia konkreettien esitystapojen tiloja, jotka kuitenkin esittävät samaa abstraktia oliota. Juuri tällaisissa tapauksissa *konkreetin esitystavan tilaa voidaan muuttaa ilman, että olion abstrakti tila muuttuu.*

Esimerkki 3.6 `LukuJoukko`-luokan `sisältää`-operaatio voisi sivuvaikutuksenaan siirtää listasolmun, jossa haettu alkio sijaitsee aina listan ensimmäiseksi siinä toivossa, että samaa alkiota kysellään `sisältää`-operaatiolla pian uudelleen (melko yleinen tilanne monessa käytännön sovelluksessa). Asiakas ei tietysti huomaa muuta eroa kuin mahdollisesti sen, että `sisältää`-operaation suoritus nopeutuu. Koska sivuvaikutukset kohdistuvat jäsenmuuttujiin, joihin ei päästä käsiksi luokan ulkopuolelta, asiakkaat eivät pysty havaitsemaan muutoksia.

3.3 Javan luokkamekanismin erityispiirteitä

Yhteen Javan luokkatiedostoon kirjoitetaan normaalisti vain yksi luokkamääritys, joskin niitä voi sijoittaa samaan tiedostoon useampiakin peräkkäin.[13] Erikoisempi tilanne saadaan aikaan kirjoittamalla luokkamääritys toisen sisään. Tällainen ratkaisu tehdään silloin, kun sisäluokka tarvitsee laajempaa viitekehystä, jonka pystyy tarjoamaan ainoastaan sitä vastaava ulkoluokka. Sen mukaan, onko sisäluokka

[13] Tällöin tiedostossa saa olla vain yksi **`public`**-suojausmääreellä varustettu luokka, jonka mukaan tiedosto nimetään.

staattinen vai ei, puhutaan joko *staattisista sisäluokista* (*nested class*) tai *esiintymäkohtaisista sisäluokista* (*inner class*). Perusidea on, että staattiset sisäluokat auttavat ryhmittelemään luokkatietoa määrittelyhierarkiaksi, kun taas esiintymäkohtaisten sisäluokkien avulla muodostetaan kiinteä ajoaikainen sidos kahden erityyppisen olion välille. Näiden lisäksi luokan sisälle on mahdollista määritellä *nimetön luokka* (*anonymous class*) tai *literaaliluokka* (*enumerated class*), joista luotujen olioiden määrä on rajoitettu.

3.3.1 Staattiset sisäluokat

Staattinen sisäluokka on normaali luokkamääritys, jonka otsikkoon on kirjoitettu määre **`static`**. Myös luokan sisällä esitelty rajapinta on aina automaattisesti staattinen. Staattinen sisäluokka käyttäytyy kuten mikä tahansa luokka. Se voi olla toisen luokan perijä tai rajapinnan toteuttaja ja sillä voi olla omia perijöitä. Siihen voidaan liittää määreet **`final`** ja **`abstract`** aivan kuten muihinkin luokkiin. Staattisen luokan sijoittaminen toisen sisään on merkki siitä, että ko. sisäluokka liittyy kiinteästi sitä ympäröivään *ulkoluokkaan* (*top-level*, *outer* tai *enclosing class*). Tämä ryhmittely näkyy syntaktisesti ohjelmatekstissä paitsi luokkien sisäkkäisyytenä myös asiakaskoodissa, koska sisäluokkatyyppiin viitataan pistenotaatiolla `Ulkoluokka.Sisäluokka`.

Aiemmin luokan näkyvyysalueeksi on voitu määritellä vain *package* tai **`public`**. Nyt tilanne on erilainen, koska sisäluokka samaistetaan luokan muihin piirteisiin. Näin ollen, jos sisäluokan suojausmääreeksi on asetettu **`private`**, se tunnetaan vain vastaavassa ulkoluokassa. Staattinen sisäluokka voi viitata ulkoluokan piirteisiin — myös **`private`**-suojattuihin — vapaasti. Ainoa vaatimus on, että kyseiset piirteet ovat **`static`**. Ulkoluokka voi puolestaan käyttää staattista sisäluokkaa kuin mitä tahansa luokkaa. On tyypillistä, että ulkoluokka luo ja käyttää staattisen sisäluokan olioita omaa toimintaansa varten. Se ei ole kuitenkaan välttämätöntä, sillä sisäluokan olioiden esiintymät ovat täysin riippumattomia ulkoluokan olioista (toisin kuin esiintymäkohtaisilla sisäluokilla).

Esimerkki 3.7 Luokan `Character`, joka on primitiivityypin **`char`** kuoriluokka, sisältä löytyy staattiset luokat `Character.Subset` ja `Character.UnicodeBlock`. Edellistä käytetään määrittelemään merkkien osajoukkoa, jollainen voi olla jälkimmäisen määrittelemät Unicode-merkistön sivut. Toki `Subset` ja `UnicodeBlock` olisi voitu esitellä omina (ulko)luokkinaan, mutta koska ne liittyvät nimenomaan merkkeihin, ne on päätetty liittää luokan `Character` staattisiksi sisäluokiksi. Staattisten sisäluokkien luontevin käyttötapa lieneekin juuri erilaisten vakioiden organisointivälineenä.

Esimerkki 3.8 Rajapinnan `Map` sisältä löytyy staattinen rajapinta `Map.Entry`, joka kuvaa `Map`-olioon talletettavaa tietoa. Tämä tieto koostuu kahdesta komponentista: *avain*, jonka avulla tietoa haetaan, ja avainta vastaava *arvo*. Kummallekin on havainnoijat ja

jälkimmäiselle lisäksi muunnosoperaatio. Kuten edellisessä esimerkissä, nytkin voidaan kysyä, olisiko ollut parempi tehdä `Entry`-rajapinnasta itsenäinen, koska vastaavaa kahden komponentin kokonaisuutta tarvitaan hyvin monessa eri tilanteessa.

Esimerkki 3.9 Eksoottisin (joskaan ei ehkä elegantein) staattinen sisäluokkaratkaisu löytyy luokasta `Point2D`, jolla on staattiset sisäluokat `Point2D.Float` ja `Point2D.Double`. Pientä aivovoimistelua aiheuttaa se, että kyseiset luokat ovat paitsi luokan `Point2D` sisäluokkia myös sen perillisiä!

3.3.2 Esiintymäkohtaiset sisäluokat

Esiintymäkohtaisten sisäluokkien avulla muodostetaan erittäin tiukka sidos sisäluokan olion ja sitä vastaavan ulkoluokan olion välille. Sisäluokan olio ei voi nimittäin olla olemassa ilman ulkoluokan oliota (sensijaan ulkoluokan olio voi olla olemassa ilman sisäluokan oliota). Tämä ei normaalitapauksessa johda ongelmiin, koska ympäröivällä ulkoluokalla on tyypillisesti joitakin jäsenmuuttujia, joista on viittaus sisäluokan olioon.[14]

Esimerkki 3.10 Listauksessa 3.3 luokka `Tapahtuma` on sisäluokka, joka avustaa ulkoluokkaa `Pankkitili` tilitapahtumien ylläpidossa.[15] Asiakas luo (tietämättään) `Tapahtuma`-olion kunkin noston ja panon yhteydessä.

Sisäluokan ja siihen liittyvän ulkoluokan välisten olioiden vahva sidos herättää kysymyksen **`this`**-viittauksen semantiikasta. Oltaessa sisäluokan oliossa, **`this`** viittaa normaaliin tapaan ko. olioon. Jos sisäluokasta, esimerkiksi `Tapahtuma`-luokasta, haluttaisiin viitata ulkoluokan olioon käytetään notaatiota **`Pankkitili.this`**. Tämä merkintä yleistyy rekursiivisesti, jos sisäluokalla on oma sisäluokkansa jne.

Edellisen esimerkin tilanne lienee melko tyypillinen: sisäluokka kapseloi sisäänsä jonkin ulkoluokkaan liittyvän erityistoiminnon. Ulkoluokalla on tavallisesti jäsenmuuttuja, joka on sisäluokan tyyppiä ja jolle delegoidaan sisäluokalle ominaiset toiminnot. Ulkoluokan asiakkaat eivät kuitenkaan havaitse ulkoluokan julkisesta liitännästä implementoinnin käyttävän sisäluokkia.

Jos asiakkaalla on pääsy sisäluokan liitäntään (sisäluokan suojaus *package* tai **`public`**), luokkaa voi käyttää kuten mitä tahansa toimittajaa. Tilanne muuttuu, jos sisäluokka kätketään asiakkailta (suojaus **`protected`** tai **`private`**). Tällöinkin ulkoluokka voi luoda sisäluokan olion ja antaa asiakkaalle viittauksen siihen. Koska

[14] Jos esiintymäkohtaiselle sisäluokalle kirjoitetaan perijä, joudutaan mahdottomaan tilanteeseen, koska ympäröivää ulkoluokkaa ei ole olemassa (kääntäjä estää tämän). Tätä, sen kummemmin kuin muitakaan erikoistilanteita, ei ruodita tässä sen enempää. Todetaan vain, että sisäluokkien villi ja vapaa käyttö saattaa johtaa hyvin erikoisiin tilanteisiin.

[15] Kääntäjä tekee sisäluokista omat `.class`-tiedostot, joiden nimessä on ulko- ja sisäluokkanimet `$`-merkillä erotettuna, siis esim. `Pankkitili$Tapahtuma.class`.

Listaus 3.3: Luokka `Pankkitili`.

```
import java.util.*;

public class Pankkitili
  {
  private long numero;
  private long saldo;
  private List<Tapahtuma> tilitapahtumat;

  public enum Tapahtumatyyppi { TALLETUS, NOSTO };

  public class Tapahtuma
    {
    private Tapahtumatyyppi tapahtumatyyppi;
    private long saldomuutos;

    public Tapahtuma(Tapahtumatyyppi t, long s)
      {
      tapahtumatyyppi = t;
      saldomuutos = s;
      }

    public String toString()
      {
      return numero + ": " + tapahtumatyyppi + "(" +
             saldomuutos + ")";
      }
    }

  public Pankkitili(long numero)
    {
    this.numero = numero;
    this.saldo = 0L;
    this.tilitapahtumat = new LinkedList<Tapahtuma>();
    }
```

Listaus 3.3 (jatkoa): Luokka `Pankkitili`.

```
public void pano(long rahamäärä)
  {
  saldo += rahamäärä;
  tilitapahtumat.add(new Tapahtuma(Tapahtumatyyppi.TALLETUS,
                                   rahamäärä));
  }

public void nosto(long rahamäärä)
  {
  saldo -= rahamäärä;
  tilitapahtumat.add(new Tapahtuma(Tapahtumatyyppi.NOSTO,
                                   rahamäärä));
  }
}
```

tämä on ulkoluokan asiakkaalle ainoa tapa saada sisäluokan olio käsiin, ulkoluokka pystyy kontrolloimaan sisäluokkien olioiden luontia tarkasti. Ongelmaksi tulee kuitenkin se, että asiakas ei tunne sisäluokan julkista liitäntää, joten saadulle oliolle voi soveltaa vain `Object`-luokan perusoperaatioita. Tämä ratkaistaan perimällä sisäluokka väljästi suojatulta luokalta, jolloin asiakas voi käyttää jälkimmäisen julkista liitäntää sisäluokan olion käsittelyyn. Tällä tavoin asiakas voidaan pakottaa käyttämään oliota polymorfisesti vaikkapa abstraktin luokan tai rajapintaluokan liitännän kautta.

Vaikka sisäluokka pääsee siis käsiksi ulkoluokan kaikkiin piirteisiin niiden suojausmääreistä riippumatta, tämä ei ole voimassa toiseen suuntaan: ulkoluokasta ei voida viitata sisäluokan **`private`**-määreisiin piirteisiin. Huomaa kuitenkin, että sisäluokka kuuluu aina samaan pakkaukseen kuin sen ulkoluokka, joten muut suojausmääreet eivät estä ulkoluokan pääsyä sisäluokkaan.

Iteraattorit (ks. kohta 8.3) ovat hyvä esimerkki esiintymäkohtaisista sisäluokista siinä mielessä, että sen eri osat eivät varsinaisesti kuulu kokoelmatyyppisen luokan julkiseen liitäntään: ne liittyvät kokoelman läpikäyntiin, eivätkä näin ollen edusta kokoelmaa, kun sitä tarkastellaan kokonaisuutena. Sisäluokan avulla iteraattorin toiminnot on saatu ryhmiteltyä erilliseksi käsitteeksi, jota ulkoluokkaa havainnoivat asiakkaat käyttävät apunaan. Iteraattorille on ominaista, että se ei muuta ulkoluokkaolionsa tilaa, vaikka se pääsee — niin halutessaan — käsiksi sen kaikkiin tilatietoihin. Täten sisäluokka on vastuussa, paitsi oman, myös ulkoluokan luokkainvariantin säilymisestä kaikissa toiminnoissa. Havainnoinnin yhteydessä jälkimmäinen pysyy triviaalisti aina voimassa. Toinen tapa taata sama asia on käyt-

tää vain ulkoluokan julkiseen liitäntään kuuluvia operaatioita. Koska sisäluokka voi vielä periä joltain toiselta luokalta, sen on mahdollisesti huolehdittava kolmesta luokkainvariantista yhtaikaisesti! On siis syytä miettiä tarkkaan, onko riittävästi perusteita sijoittaa luokka toisen sisälle vai pitäisikö se laittaa itsenäiseksi luokaksi kaikkien käyttöön. Onhan olioparadigman perusajatus uudelleenkäyttö!

3.3.3 Nimettömät luokat

Luokan voi sijoittaa vielä edellä esiteltyjä esimerkkejä eksoottisemmin jopa jonkin rutiinin tai vaikkapa rutiinissa käytetyn lauselohkon sisään. Ajatuksena on, että näin luokan näkyvyyttä saadaan rajattua haluttuun alueeseen. Tästä syystä luokalle ei saa kirjoittaa muita määreitä kuin **`final`**, joka normaaliin tapaan estää ko. luokalta perimisen. On vaikea nähdä näinkin rajattuun alueeseen kirjoitetun luokan soveltuvuutta käytännön tilanteisiin. Sen sijaan *nimetöntä luokkaa* käytetään tilanteissa, joissa luokan oliota tarvitaan ”kertakäyttöisesti”. Niitä käytetään yleensä tilanteessa, missä peritään luokka, jonka operaatioista halutaan korvata vain yksi tai kaksi piirrettä omiin tarpeisiin. Tällainen luokkamääritys on tyypillisesti lyhyt ja sen käyttö on hyvin lokaalia.

Esimerkki 3.11 Tarkastellaanpa seuraavaa koodinpätkää:

```
JFrame ikkuna = new JFrame("Akkuna");
ikkuna.addWindowListener(new WindowAdapter()
  {
  public void windowClosing(WindowEvent e)
    {
    System.exit(0);
    }
  } );
```

Luodulle `JFrame`-tyyppiselle ikkunalle halutaan siis lisätä `WindowAdapter`-luokasta periytyvästä nimettömästä luokasta luotu kuuntelijaolio, joka tarkkailee ikkunan sulkemista. Kyseinen nimetön luokka uudelleentoteuttaa `windowClosing`-metodin suorittamalla sovelluksen terminoinnin `System.exit`-metodikutsulla.

Kun nimettömällä luokalla ei kerran ole nimeä, sillä ei voi myöskään olla konstruktoria.[16] Täten kaikki alustustoimet (joilla luokan luokkainvariantti saadaan voimaan) on tehtävä asetuslauseilla. Koska luokalla ei ole otsikkoa, sillä ei voi myöskään olla otsikkoon kirjoitettavia määreitä. Todelliset Java-gurutkin [1] suosittelevat nimettömien luokkien käyttöä vain, jos ne ovat erittäin lyhyitä (enintään kuusi riviä!), koska ihmisen on vaikea hahmottaa tämäntyyppistä syntaksia.

[16] Java-kääntäjä toki nimeää nämäkin luokat käyttämällä juoksevaa numerointia, ts. ensimmäisen käännetyn anonyymin luokan nimi on `MunLuokka$1.class`, toisen `MunLuokka$2.class`, jne.

3.3.4 Literaaliluokka `enum`

Literaaliluokka määritellään **`enum`**-avainsanalla, jota seuraa luokkamääritys. Sille voidaan siis antaa suojausmääreitä ja se voi periytyä jostain toisesta luokasta. Ainoat rajoitteet ovat ne, että (a) literaaliluokkaa ei voi periä (ts. **`enum`**-avainsana on tarkoittaa määrittelyä **`final class`**) ja (b) literaaliluokkaa ei voi konstruoida muualla kuin sen omassa määrittelylohkossa. Yksinkertaisimmillaan literaaliluokkaa käytetään vakioarvojen luetteluun (ts. korvaamaan **`static final`** -vakioarvoja). Esimerkiksi listauksessa 3.3 `Pankkitili`-luokkaan on määritelty literaaliluokka `Tapahtumatyyppi`, joka luettelee mahdolliset vakioarvot.

Esimerkki 3.12 Lueteltu vakio voidaan muodostaa yksinkertaisesti:

```
enum Presidentti { STÅHLBERG, RELANDER, SVINHUFVUD, KALLIO,
                   RYTI, MANNERHEIM, PAASIKIVI, KEKKONEN,
                   KOIVISTO, AHTISAARI, HALONEN };
```

Toisin kuin esimerkiksi C++:ssa avainsanalla **`enum`** määritellään *luokka* eikä pelkkiä vakioarvoja. Literaaliluokalle voidaan siis antaa konstruktori sekä jäsenmuuttujia ja metodeja. On huomattava että luokan instansseja voidaan luoda kuitenkin vain **`enum`**-lohkon sisällä.

Esimerkki 3.13 Listauksessa 3.4 on esitelty pelikortin mallintava luokka `Kortti`. Luokassa on kaksi literaaliluokkaa, `Arvo` ja `Maa`. Kummallakin literaaliluokalla on privaatteja jäsenmuuttujia ja niiden havainnointimetodeja sekä konstruktorit, joilla asetetaan jäsenmuuttujien arvot. Olioiden konstruointi tapahtuu normaalista poiketen ilman **`new`**-avainsanaa luettelemalla instanssit ja niiden konstruktoriparametrit (huomaa puolipiste listan lopussa).

Luokalla on staattinen metodi `uusiPakka`, joka palauttaa sekoittamattoman korttipakan. Pakan luonti ja sekoittaminen voisi tapahtua vaikkapa seuraavasti:

```
List<Kortti> pakka = Kortti.uusiPakka();
Collections.shuffle(pakka);
```

Kaikki literaaliluokat ovat `Enum`-luokan perillisiä, joten niillä on mm. metodit `name`, joka palauttaa literaalin nimen, `ordinal`, joka palauttaa literaalin järjestysnumeron, ja luokkametodi `values`, joka palauttaa kaikki literaalityypin arvot taulukossa (esim. **`for`**-silmukassa iterointiin). Literaaliluokka toteuttaa `Comparable`-rajapintaan, joten literaaliolioiden luettelojärjestystä voidaan verrata `compareTo`-metodilla. Tämän lisäksi literaaliluokkaa voidaan käyttää suoraan **`switch`**-rakenteessa.

Tehtäviä

3-1 Kirjoita määrittely ja toteutus `LukuJoukko`-luokan metodille

Listaus 3.4: Luokka `Kortti`.

```
import java.util.*;

public final class Kortti
  {
  public enum Arvo
    {
    ÄSSÄ('A', 1), KAKSI(2), KOLME(3), NELJÄ(4), VIISI(5), KUUSI(6),
    SEITSEMÄN(7), KAHDEKSAN(8), YHDEKSÄN(9), KYMMENEN(10),
    JÄTKÄ('J', 11), KUNINGATAR('Q', 12), KUNINGAS('K', 13);

    private String symboli;
    private int numeroarvo;

    private Arvo(char s, int a)
      {
      symboli = Character.toString(s);
      numeroarvo = a;
      }

    private Arvo(int a)
      {
      symboli = Integer.toString(a);
      numeroarvo = a;
      }

    public String annaSymboli() { return symboli; }
    public int annaNumeroarvo() { return numeroarvo; }
    }

  public enum Maa
    {
    RISTI('\u2663'), RUUTU('\u2666'), HERTTA('\u2665'), PATA('\u2660');

    private String symboli;

    private Maa(char s) { symboli = Character.toString(s); }
    public String annaSymboli() { return symboli; }
    }
```

Listaus 3.4 (jatkoa): Luokka `Kortti`.

```
  private final Arvo arvo;
  private final Maa maa;

  private Kortti(Arvo arvo, Maa maa)
    {
    this.arvo = arvo;
    this.maa = maa;
    }

  public Arvo arvo() { return arvo; }
  public Maa maa() { return maa; }

  public String toString()
    {
    return maa.annaSymboli() + arvo.annaSymboli();
    }

  public static List<Kortti> uusiPakka()
    {
    List<Kortti> pakka = new ArrayList<Kortti>();
    for (Maa maa : Maa.values())
      for (Arvo arvo : Arvo.values())
        pakka.add(new Kortti(arvo, maa));
    return pakka;
    }
}
```

```
public LukuJoukko erotus(LukuJoukko toinen)
```

joka laskee **this**-joukon ja argumenttina annetun joukon erotuksen (ts. tulos sisältää kaikki sellaiset alkiot, jotka ovat **this**-joukossa mutta eivät `toinen`-joukossa).

3-2 Kirjoita määrittely ja toteutus `LukuJoukko`-luokan metodille

```
public void liitä(LukuJoukko toinen)
```

joka lisää argumenttina annetun joukon `toinen` kaikki alkiot nykyiseen (**this**) kokonaislukujoukkoon (ts. unioni sivuvaikutuksen avulla).

3-3 Muutetaanpa `LukuJoukko`-luokan konkreettia esitystapaa siten, että duplikaatit ovat sallittuja. Toteuta metodit `koko` ja `lisää` tätä esitystapaa käyttäen.

3-4 Miten `LukuJoukko`-luokkaa pitää muuttaa, jos se yleistetään tallentamaan **`int`**-tyyppisten primitiiviarvojen sijaan `Object`-olioita? Entä jos vertailun pitäisi perustua `equals`-vertailun sijaan olioiden identiteettien vertailulle?

3-5 Tutustu `java.math`-paketin luokkaan `BigInteger`. Mitä luokka tekee? Miten metodeja `add(BigInteger)` ja `negate` käytetään? Kuinka laskisit lausekkeiden $-x \cdot (y-1)$ ja $(a+b)^2/c$ arvon? `BigInteger`-olioiden sanotaan olevan mutatoitumattomia (*immutable*); mitä se tarkoittaa?

3-6 Tutustu `java.math`-paketin luokkaan `BigDecimal`. Mitä luokka tekee? Miten metodeja `add(BigDecimal)` ja `negate()` käytetään? Kuinka laskisit lausekkeiden $-((5+x)^5)$ ja $1/(x+1)$ arvot? `BigDecimal`-olioiden sanotaan olevan mutatoitumattomia (*immutable*); mitä se tarkoittaa?

3-7 Olion tilan oikeellisuudesta eli eheydestä on pidettävä huolta konstruktorista lähtien. Toisin sanoen järjestelmässä ei saa koskaan käyttää laittomassa tilassa olevia olioita. Syntyvän olion eheys on yleensä suoraviivaista saavuttaa, jos konstruktorilla ei ole alkuehtoa. Mitäpä sitten pitäisi tehdä, jos konstruktorilla on alkuehto? Tässä tehtävässä tutustumme kolmeen tapaan hoitaa asia:

- Konstruktori nostaa poikkeuksen, mikäli sen alkuehtolauseke ei ole voimassa.
- Luokkaan toteutetaan staattinen totuusarvon palauttava funktio (eli staattinen predikaatti), jonka avulla asiakas tarkistaa etukäteen onko konstruktorin alkuehto voimassa. Tietenkin tämä predikaatti on myös konstruktorin alkuehtona.
- Olioon lisätään totuusarvoinen tila, joka ilmaisee onko olio eheä vai ei. Olion eheyttä havainnoidaan siihen kohdistuvalla predikaatilla, joka myös liitetään olion muiden eheyttä vaativien rutiinien alkuehtoon. Tällöin konstruktorin alkuehto voidaan poistaa, sillä ehdon voimassaoloa vaaditaan vasta jälkijättöisesti.

Toteuta kolme luokkaa `Lauseke`, `Staattinen` ja `Viivytetty`, joiden avulla seuraava luokan `EiKaadu` koodi voidaan suorittaa ohjelman kaatumatta. Jokainen luokka sisältää konstruktorin, joka saa muodollisina parametreinaan kokonaisluvut `jaettava` ja `jakaja` (tässä järjestyksessä) sekä funktion:

```
public int jako() { return this.jaettava / this.jakaja; }
```

Luokan `Lauseke` konstruktorin alkuehto on `jakaja != 0`, luokan `Staattinen` konstruktorin alkuehto on `Staattinen.onEiNolla(jakaja)` ja luokan `Viivytetty` metodilla `jako()` on alkuehto **`this`**`.onEhyt()`.

```
public class EiKaadu {
   public static void main(String[] komentorivi) {
      int a = 12, b = 0;
```

```
      try { System.out.println((new Lauseke(a, b)).jako()); }
      catch ( IllegalArgumentException e )
         { System.out.println("Koppi: " + e); }

      if ( Staattinen.onEiNolla(b) )
         System.out.println((new Staattinen(a,b)).jako());
      else
         System.out.println("Taaskin jaetaan nollalla!");

      Viivytetty kolmas = new Viivytetty(a,b);
      if ( kolmas.onEhyt() ) System.out.println(kolmas.jako());
      else                   System.out.println("Vielä yksi moka.");
   }  }
```

3-8 Luokka `Tasopiste`, joka on esitelty listauksessa 3.1, mallintaa tason pisteen karteesisessa koordinaatistossa eli perinteisessä xy-koordinaatistossa, jossa pisteen sijainti määritellään yksikäsitteisesti xy-koordinaattiakselien arvoilla. Joskus tason piste halutaan kuitenkin ilmaista polaarisena eli etäisyytenä r origosta ja vastapäiväisenä kulmana α karteesisen koordinaatiston x-akselista. Laajennetaanpa luokkaa niin, että asiakas voi luoda pisteen sekä karteesisen että polaarisen koordinaatiston parametreilla.

(a) Toteuta metodit `karteesinen(double,double)` ja `polaarinen(double, double)` sekä suojattu konstruktori. Polaarinen piste (r, α) voidaan esittää karteesisena pisteenä (x, y) seuraavasti: $x = r\cos\alpha$ ja $y = r\sin\alpha$. (Vinkki: Tutustu luokkaan `java.lang.Math`.)

(b) Miksi luokan konstruktori on suojattu määreellä **`protected`**?

(c) Anna ohjelmakoodiesimerkki jossa luot tason pisteen karteesisesta kohdasta $(1, 1)$ sekä karteesisilla että polaarisilla parametreilla.

```
public class Tasopiste {
   /** Palauttaa pisteen karteesisesta kohdasta (x, y). */
   public static Tasopiste karteesinen(double x, double y) {
      // Toteuta tämä.
      }

   /**
    *  Palauttaa pisteen polaarisesta kohdasta (etäisyys, kulma).
    *  Kulma ilmaistaan radiaaneina
    *  (1 aste == Math.PI / 180.0 radiaania).
    */
   public static Tasopiste polaarinen(double etäisyys, double kulma) {
      // Toteuta tämä.
      }
```

```
/** Karteesinen toteutus (voidaan vaihtaa tarvittaessa). */
private double x,
               y;

/** Alustaa pisteen karteesisen koordinaatiston kohtaan (x, y). */
protected Tasopiste(double x, double y) {
   // Toteuta tämä.
   }
}
```

3-9 Oletetaan, että tehtävän 3-8 mukaisella luokalla `Tasopiste` halutaankin mallintaa pelkästään niitä pisteitä, jotka kuuluvat karteesisen koordinaatiston ensimmäiseen neljännekseen (eli pisteitä joiden karteesiset koordinaatit ovat ei-negatiivisia). Kirjoita tällaisen `Tasopiste`-luokan luokkainvariantti ja rutiinien alkuehdot. Toteuta alkuehtojen tarkistaminen myös ohjelmallisesti sopivaksi katsomallasi tavalla.

3-10 `Monijoukko`-luokka kuvaa alkioiden kokoelmaa, joka voi sisältää useita esiintymiä samasta alkiosta (esim. kokonaislukujen `Monijoukko`-kokoelmia ovat $\langle 1, 2, 2, 2, 3, 3\rangle$ ja $\langle 4, 4, 4, 1\rangle$) ja joka ei määrää alkioilleen järjestystä (ts. $\langle 1, 2, 2, 3\rangle = \langle 2, 1, 3, 2\rangle$).

`Monijoukko`-luokkaa vastaavan rajapinnan määrittely on annettu listauksessa 3.5. Valitse konkreetti esitystapa ja laadi toteutus metodeille `annaKoko`, `annaMäärä` ja `poista`.

3-11 Toteuta anagrammin tarkistamisen suorittava rutiini tehtävässä 3-10 esitellyn luokan `Monijoukko` avulla. Kaksi argumenttina annettua merkkijonoa ovat toistensa anagrammeja, jos ne sisältävät tarkalleen samat merkit, mutta mahdollisesti eri järjestyksessä (esim. ”taulu” ja ”luuta” ovat toistensa anagrammeja).

3-12 Toteuta tehtävässä 3-10 esiteltyyn `Monijoukko`-luokkaan sisäluokka `Pari`, joka kuvaa yhtä `Monijoukko`-olion alkiota. `Pari`-luokan `E`-tyyppisessä kentässä on monijoukossa oleva alkio ja **`int`**-kentässä kyseisen alkion esiintymien määrä. `Monijoukko`-luokka käyttää sisäisenä toteutuksena `List`-oliota, johon on tallennettu `Pari`-olioita. Tee `Monijoukko`-luokkaan myös sisäinen hakuoperaatio

```
private int annaIndeksi(E e)
```

joka hakee sen `List`-olion indeksin, jossa sijaitsee alkioltaan argumenttia `e` vastaava `Pari`-olio.

3-13 Suunnittele luokka, jonka avulla voidaan käsitellä suomi–englanti -sanakirjaa. Sanakirjaan talletetaan vastinsanapareja, esim. (kieli, language), (sanakirja, dictionary), jne., mutta ei taivutuksia, synonyymejä, selityksiä tms. Hakuja pitää voida tehdä kumpaankin suuntaan. Esittele erityisesti konkreetti esitystapa, luokkainvariantti ja luokan julkinen liitäntä (operaatioiden määrittelyt, toteutusta ei tarvita).

3-14 Paketissa `cia` oleva luokka `Agentti` näyttää seuraavalta:

Listaus 3.5: Rajapinta `Monijoukko`.

```
public interface Monijoukko<E>
    {
    /**
     *  Palauttaa monijoukon kaikkien alkoiden lukumäärän.
     *  @.pre  true
     *  @.post RESULT >= 0
     */
    public int annaKoko();

    /**
     *  Palauttaa monijoukon equal-mielessä erilaisten alkioiden
     *  lukumäärän.
     *  @.pre  true
     *  @.post RESULT >= 0 & RESULT <= annaKoko()
     */
    public int annaErilaistenMäärä();

    /**
     *  Hakee alkion e esiintymien määrän monijoukossa.
     *  Vertailuun käytetään equals-funktiota.
     *  @.pre  true
     *  @.post RESULT == (equals-mielessä yhtäsuurten
     *                    alkioiden lukumäärä)
     */
    public int annaMäärä(E e)

    /**
     *  Lisää alkion e monijoukkoon.
     *  @.pre  true
     *  @.post annaKoko() == OLD(annaKoko()) + 1 &
     *         annaMäärä(e) == OLD(annaMäärä(e)) + 1
     */
    public void lisää(E e);
```

Listaus 3.5 (jatkoa): Rajapinta `Monijoukko`.

```
/**
 *  Poistaa alkion e monijoukosta.
 *  Vertailuun käytetään equals-funktiota.
 *  @.pre  annaMäärä(e) >= 1
 *  @.post annaKoko() == OLD(annaKoko()) - 1 &
 *         annaMäärä(e) == OLD(annaMäärä(e)) - 1
 */
public void poista(E e)

/**
 *  Poistaa kaikki alkiot e monijoukosta.
 *  Vertailuun käytetään equals-funktiota.
 *  @.pre  annaMäärä(e) >= 1
 *  @.post annaKoko() == OLD(annaKoko()) - OLD(annaMäärä(e)) &
 *         annaMäärä(e) == 0
 */
public void poistaKaikki(E e)
}
```

```
package cia;

public class Agentti
  {
  private final int koodi = 27911180;

  boolean onkoSuurempi(int luku) { return (koodi > luku); }

  protected boolean onkoJaollinen(int luku)
    {
    return ((koodi % luku) == 0);
    }

  public boolean onkoKoodi(int luku) { return (koodi == luku); }
  }
```

Millä tavoin seuraavat luokat pystyvät selvittämään argumenttina annetun `Agentti`-olion `koodi`-kentän sisällön?

(a) Paketissa `cia` oleva luokka `Kaveri`:

```
package cia;
```

```
public class Kaveri
  {
  public int salaisuus(Agentti a) { /* ... */ }
  }
```

(b) Paketissa `cia` oleva luokka `HyväPoika`:

```
package cia;

public class HyväPoika extends Agentti
  {
  public int salaisuus(Agentti a) { /* ... */ }
  }
```

(c) Paketissa `kgb` oleva luokka `PahaPoika`:

```
package kgb;

public class PahaPoika extends Agentti
  {
  public int salaisuus(Agentti a) { /* ... */ }
  }
```

(d) Paketissa `kgb` oleva luokka `Vihollinen`:

```
package kgb;

public class Vihollinen
  {
  public int salaisuus(Agentti a) { /* ... */ }
  }
```

(Vinkki: Koska vain `Agentti`-oliot pääsevät käsiksi jäsenmuuttujaan `koodi`, sinun on siis mietittävä ajoaikainen *menetelmä*, jolla `koodi` saadaan paljastettua epäsuorasti. Menetelmien kuvailu riittää, niitä ei tarvitse toteuttaa.)

3-15 Luokan `Nostromo` toteutus näyttää seuraavalta:

```
import java.util.*;

public class Nostromo
  {
  private static Collection<CrewMember> roster =
                      new LinkedList<CrewMember>();

  public static class CrewMember
```

```
  {
  private String name;
  private String rank;
  private boolean alive;

  public CrewMember(String n, String r)
    {
    name = n; rank = r; alive = true;
    roster.add(this);
    }

  public String toString()
    {
    return rank + " " + name + " is " +
                (alive ? "alive." : "dead.");
    }

  public void implant() { new Alien(); }

  private class Alien
    {
    public Alien() { CrewMember.this.alive = false; }
    }
  }

 public static String report()
  {
  String result = "";
  for (CrewMember member : roster)
    result = result + member.toString() + "\n";
  return result;
  }
}
```

Selosta yksityiskohtaisesti, mitä seuraavan testiajon aikana tapahtuu. Mitä ohjelma tulostaa ja miksi?

```
Nostromo.CrewMember ripley =
        new Nostromo.CrewMember("Ellen L. Ripley", "Lieutenant");
Nostromo.CrewMember dallas =
        new Nostromo.CrewMember("A. J. Dallas", "Captain");
System.out.println(Nostromo.report());
dallas.implant();
System.out.println(Nostromo.report());
```

3-16 Oletetaanpa että olemme ohjelmoimassa järjestelmää, jossa tarvitaan rationaalilukuja. Rationaalilukuhan on sellainen reaaliluku, joka voidaan esittää kahden kokonaisluvun osamääränä: m/n. Lukua m sanotaan osoittajaksi ja lukua n nimittäjäksi. Valitaan tämä osamääräesitys luokan `Rationaaliluku` toteutusrakenteeksi, eli jokaisella `Rationaaliluku`-oliolla on **`int`**-tyyppiset jäsenmuuttujat `osoittaja` ja `nimittäjä`. Kirjoita tälle luokalle luokkainvariantti ja abstraktiofunktio. Itse luokkaa ei siis tarvitse toteuttaa.

(Vinkki: Ota kantaa ainakin seuraaviin asioihin: (a) Voiko nimittäjä olla nolla? Jos voi, mitä se tarkoittaa? (b) Kuinka negatiivinen luku esitetään? Esimerkiksi ovatko $-1/2$ ja $1/-2$ eheitä? (c) Esitetäänkö rationaaliluku supistettuna? Esimerkiksi ovatko $6/8$ ja $3/4$ eheitä? (d) Miten nolla esitetään? Esimerkiksi ovatko $0/1$ ja $0/2$ eheitä?)

3-17 Tutki seuraavaa sangen eksoottisen näköistä ratkaisua.

```
public interface Predikaatti
  {
  public boolean on(int luku);
  } // Predikaatti

public class Eksoottinen
  {
  public static void main(String[] argumentit)
    {
    for ( int i = 0; i < 100; ++i )
      {
      System.out.println(i + ":lle: " +
        new Predikaatti()
          {
          public boolean on(int luku) {/* Totetus poistettu. */}
          }.on(i));
      }
    }
  }
```

(a) Selitä lyhyesti, mitä `main`-metodissa tapahtuu.

(b) Halutaan, että `Predikaatti`-tyyppinen olio esiintyy vain `main`-metodissa. Miten toteutat tämän? Anna koodiesimerkki. (Vinkki: Esittele `main`-metodissa tyyppiä `Predikaatti` oleva lokaali muuttuja.)

(c) Halutaan, että `Predikaatti`-tyyppinen olio esiintyy vain `Eksoottinen`-luokassa. Miten toteutat tämän? Anna koodiesimerkki. (Vinkki: Esittele `Eksoottinen`-luokassa tyyppiä `Predikaatti` oleva jäsenmuuttuja.)

(d) Anna `Predikaatti`-määrittelyn `on`-metodille luokassa `Eksoottinen` sellainen toteutus, että se palauttaa **`true`**, jos parametrina annettu `luku` on alkuluku. Muutoin metodi palauttaa tietysti **`false`**.

Kaikkihan muistavat, että yksinkertainen (tosin myös yksi hitaimmista, mutta eipä välitetä siitä tässä) menetelmä luvun ℓ alkuluvullisuuden testaamiseksi on seuraavanlainen:

- Jos $\ell < 2$, niin se ei ole alkuku.
- Jos $\ell = 2$, niin se on alkuluku.
- Jos $\ell > 2$, niin sitä yritetään jakaa kaikilla luvuilla väliltä $2, \ldots, \ell - 1$. Jos ℓ on jaollinen jollakin em. välin luvuista, se ei ole alkuluku. Jos ℓ taas ei ole jaollinen millään em. välin luvuista, se on alkuluku.

3-18 Lisätään tehtävässä 2-15 esiteltyyn `Intervalli`-luokkaan seuraava iteraattoriolion palauttava metodi:

```
/**
 *  @.pre  true
 *  @.post RESULT == (koko lukuvälin läpikäyvä iteraattori)
 */
public IntIteraattori annaIteraattori()
```

kun `IntIteraattori`-rajapinnan määrittely on seuraava:

```
public interface IntIteraattori {
    /**
     *  @.pre  true
     *  @.post RESULT == (true jos kokoelmassa on vielä
     *                    läpikäymättömiä kokonaislukualkioita;
     *                    muutoin false)
     */
    public boolean onSeuraava();

    /**
     *  @.pre  onSeuraava()
     *  @.post RESULT == (läpikäymätön kokonaislukualkio)
     */
    public int seuraava();
}
```

Laadi toteutus `annaIteraattori`-metodille. (Vinkki: käytä sisäluokkaa.)

3-19 Luokkaa `Sali`, joka on esitetty listauksessa 3.6, käytetään tallentamaan opiskelijoiden istumapaikkavarauksia. Luokalle on valittu konkreetiksi esitystavaksi taulukko. Laadi luokan toteutus.

Listaus 3.6: Luokan `Sali` rutiinimäärittelyjä.

```
/**
 *  @.classInvariantPrivate
 *     Paikkavaraukset ovat tallennettuna taulukossa paikat.
 *     Jokaisessa taulukon paikassa on arvona ko. paikan varanneen
 *     opiskelijan opintorekisterinumero tai 0, jos paikkaa ei ole
 *     varattu. Taulukossa ei ole nollasta poikkeavia duplikaatteja.
 */
public class Sali {
    private int[] paikat;

    /**
     *  @.pre  koko >= 1
     *  @.post paikkoja() == koko &
     *         varattujaPaikkoja() == 0
     */
    public Sali(int koko)

    /**
     *  @.pre !onTäynnä() & !onVarannutPaikan(opnro)
     *  @.post varattujaPaikkoja() == OLD(varattujaPaikkoja()) + 1 &
     *         onVarannutPaikan(opnro) &
     *         etsiPaikka(opnro) == RESULT &
     *         varaaja(RESULT) == opnro
     */
    public int varaa(int opnro)

    /**
     *  @.pre  !onVarannutPaikan(opnro) & onLaillinenPaikka(paikka)
     *  @.post RESULT == (varauksen onnistuminen) &
     *         varattujaPaikkoja() == OLD(varattujaPaikkoja()) + 1 &
     *         onVarannutPaikan(opnro) &
     *         etsiPaikka(opnro) == paikka
     */
    public boolean varaaTietty(int opnro, int paikka)
```

Listaus 3.6 (jatkoa): Luokan `Sali` rutiinimäärittelyjä.

```
/**
 *  @.pre  onVarannutPaikan(opnro)
 *  @.post varattujaPaikkoja() == OLD(varattujaPaikkoja()) - 1 &
 *         !onVarannutPaikan(opnro)
 */
public void peruutaVaraus(int opnro)

/**
 *  @.pre  true
 *  @.post RESULT == (paikkojen lukumäärä)
 */
public int paikkoja()

/**
 *  @.pre  true
 *  @.post RESULT == (varattujen paikkojen lukumäärä)
 */
public int varattujaPaikkoja()

/**
 *  @.pre  onLaillinenPaikka(paikka)
 *  @.post RESULT == (true jos paikka on varattu;
 *                    muuten false)
 */
public boolean onVarattu(int paikka)

/**
 *  @.pre  true
 *  @.post RESULT == (paikkoja() == varattujaPaikkoja())
 */
public boolean onTäynnä()

/**
 *  @.pre  true
 *  @.post RESULT == (0 <= paikka & paikka < paikkoja())
 */
public boolean onLaillinenPaikka(int paikka)
```

Listaus 3.6 (jatkoa): Luokan `Sali` rutiinimäärittelyjä.

```
    /**
     *  @.pre  true
     *  @.post RESULT == (true jos opiskelijanumerolle oprno
     *                    on varattu paikka; muuten false)
     */
    public boolean onVarannutPaikan(int opnro)

    /**
     *  @.pre  onVarannutPaikan(opnro)
     *  @.post varaaja(RESULT) == opnro
     */
    public int etsiPaikka(int opnro)

    /**
     *  @.pre  onLaillinenPaikka(paikka)
     *  @.post etsiPaikka(RESULT) == paikka
     */
    public int varaaja(int paikka)
}
```

3-20 Jono (*queue*) on FIFO-tietorakenne (*first-in, first-out*, ts. aikaisemmin tullutta palvellaan ensin). Jono tukee seuraavia operaatioita: alkion lisäys jonon perään, jonon ensimmäisen alkion palautus sekä jonon ensimmäisen alkion poisto. Suunnittele luokan `Jono` toteutus: julkinen liitäntä (operaatioiden määrittelyt), konkreetti esitystapa ja luokkainvariantti.

3-21 Lista on eräs yksinkertaisimpia rekursiivisia tietorakenteita. Listaoliot määritellään seuraavasti:

i. Tyhjä lista on lista.

ii. Jos L on lista ja O on olio, niin (O, L) on lista.

iii. Muita listoja ei ole.

Toteuta listaolioiden luokka `DaList`. Siinä on oltava ainakin seuraavat metodit:

- **`static`** `DaList create()` luo tyhjän listan.
- `DaList cons(Object o)` palauttaa listan, jonka ensimmäisenä alkiona on olio `o` ja loppuosana lista **`this`**.
- **`boolean`** `isEmpty()` palauttaa **`true`**, jos ja vain jos lista on tyhjä lista.
- `Object head()` palauttaa ei-tyhjän listan ensimmäisen olion.
- `DaList tail()` palauttaa ei-tyhjän listan loppuosan.

Metodien välillä vallitsee yhteydet (oletetaan `Object olio` ja `DaList lista`):

- `(DaList.create()).isEmpty()`,
- `lista.cons(olio).head() == olio`, ja
- `lista.cons(olio).tail().equals(lista)`.

Havainnollista listojen käyttöä toteuttamalla luokan ulkopuolinen metodi, joka rakentaa listan, jossa on seuraavat oliot:

- merkkijono `"xxx"`,
- kokonaisluku `1`, ja
- lista, jossa on oliot `'x'` ja `"x"`.

3-22 Mallinnetaan sopulilaumaa, jossa olevien sopuleiden mielialat vaihtelevat yhteisen joukkotietoisuuden perusteella. Sovitaan ettei sopulia voi olla ilman mielialaa. Mieliala toteutetaan literaaliluokkana seuraavasti.

```
public enum Mieliala {
   TAPITETAAN, NUUHKITAAN, SYÖDÄÄN, JUOSTAAN,
   HYPÄTÄÄN, LEIKITÄÄN_SUKUPUUTTOA
   }
```

Sovitaan lisäksi että sopulilaumassa yhden sopulin mielialan muuttuminen muuttaa myös kaikkien muiden samaan joukkotietoisuuteen kuuluvien sopuleiden mielialat täksi samaksi mielialaksi välittömästi. Toteutetaan tämä oliorakennelma esiintymäkohtaisella sisäluokalla: luokka `Joukkotietoisuus` pitää sisällään tietoisuuden tunnistenimen sekä kaikkien siihen tietoisuuteen kuuluvien sopuleiden yhteisen mielialan. Nimen lisäksi sopulilla on kyky muuttaa mielialaansa, mikä mallinnetaan luokalla `Joukkotietoisuus.Sopuli`. Määrittele ja toteuta nämä luokat siten että ohjelmakoodi

```
Joukkotietoisuus sankarit = new Joukkotietoisuus("Oudot Oliot");

Joukkotietoisuus.Sopuli takku = sankarit.new Sopuli("Iso Takku");
Joukkotietoisuus.Sopuli lutu = sankarit.new Sopuli("Pörrö Lutunen");
Joukkotietoisuus.Sopuli tuhituhi = sankarit.new Sopuli("T. Tuhi");

System.out.println(takku + "\n" + lutu + "\n" + tuhituhi);
tuhituhi.asetaMieliala(Mieliala.NUUHKITAAN);
System.out.println("\n" + takku + "\n" + lutu + "\n" + tuhituhi);

Joukkotietoisuus hörhöt = new Joukkotietoisuus("Täh?");

Joukkotietoisuus.Sopuli[] lauma = new Joukkotietoisuus.Sopuli[5];

for ( int i = 0; i < lauma.length; ++i ) {
   lauma[i] = hörhöt.new Sopuli("Sopuli #" + (i + 1));
   }
System.out.println("\n" + java.util.Arrays.toString(lauma));
lauma[3].asetaMieliala(Mieliala.JUOSTAAN);
System.out.println("\n" + java.util.Arrays.toString(lauma));
```

tulostaa (jollakin tavalla muotoiltuna):

```
Olen Iso Takku tietoisuudesta Oudot Oliot ja me TAPITETAAN
Olen Pörrö Lutunen tietoisuudesta Oudot Oliot ja me TAPITETAAN
Olen T. Tuhi tietoisuudesta Oudot Oliot ja me TAPITETAAN

Olen Iso Takku tietoisuudesta Oudot Oliot ja me NUUHKITAAN
Olen Pörrö Lutunen tietoisuudesta Oudot Oliot ja me NUUHKITAAN
Olen T. Tuhi tietoisuudesta Oudot Oliot ja me NUUHKITAAN

[Olen Sopuli #1 tietoisuudesta Täh? ja me TAPITETAAN, Olen
Sopuli #2 tietoisuudesta Täh? ja me TAPITETAAN, Olen Sopuli
#3 tietoisuudesta Täh? ja me TAPITETAAN, Olen Sopuli #4
tietoisuudesta Täh? ja me TAPITETAAN, Olen Sopuli #5
tietoisuudesta Täh? ja me TAPITETAAN]
```

```
[Olen Sopuli #1 tietoisuudesta Täh? ja me JUOSTAAN, Olen
Sopuli #2 tietoisuudesta Täh? ja me JUOSTAAN, Olen Sopuli
#3 tietoisuudesta Täh? ja me JUOSTAAN, Olen Sopuli #4
tietoisuudesta Täh? ja me JUOSTAAN, Olen Sopuli #5
tietoisuudesta Täh? ja me JUOSTAAN]
```

Luku 4

Luokkakokonaisuuden muodostaminen

Ehkä vaikeinta koko olioajattelussa ja -ohjelmoinnissa on se, miten annettu tehtävä pirstotaan osiin niin, että osat ovat järkeviä, itsenäisiä ja uudelleenkäytettäviä kokonaisuuksia, jotka yhdessä ratkaisevat kyseisen ongelman. Valmista ohjelmistoa on sen sijaan yleensä helpompi arvioida. Tehdyt ratkaisut ovat luontevia, jos kaikilla osilla on helposti ymmärrettävät vastineensa ongelma- ja ratkaisumaailmassa ja osien vastuualueet on selkeästi määritelty.

Miten hyvään ratkaisuun päädytään onkin jo kokonaan toinen asia. Ei ole nimittäin olemassa mitään yksinkertaista toimintatapaa tai -ohjeistusta, jota noudattaen ohjelmistosysteemistä tulisi aina ”hyvä”. Ohjelmoijan kokemus sekä mallinnettavan maailman perinpohjainen tuntemus ovat ehkä suurimpia tekijöitä, jotka vaikuttavat tuloksena saatavan ohjelman laatuun. Pitkään alalla ollut osaa välttää mahdolliset sudenkuopat ja taitotiedollaan rakentaa ratkaisuja, jotka uudelleenkäyttävät aiemmin hyväksi havaittuja komponentteja ja tekniikoita kuten *suunnittelumalleja* (*design patterns*).

Kun puhutaan tehtävän jakamisesta osiin ja eri osien vastuualueista, ollaan tyypillisesti tekemisissä asiakasrelaation kanssa. Hieman kärjistäen voidaankin ehkä sanoa, että luokkien välinen periytymisrelaatio ei liity suoranaisesti ohjelmiston *analysointi- ja suunnitteluvaiheisiin* (*analysis* ja *design phase*), ei ainakaan ensi vaiheessa. Pikemminkin periytyminen on väline, jonka avulla mallinnettavaa maailmaa ymmärretään paremmin ja jonka avulla myös ohjelmakoodi voidaan organisoida järkevämmäksi (tämä tapahtuu yleistysvaiheessa, joka tehdään viimeisenä). Osasyynä tähän lienee sekin, että abstrakteja luokkia vastaavia liittymiä ei yleensä ole mallinnettavassa kohteessa johtuen siitä yksinkertaisesta syystä, että

reaalimaailman oliot ovat monesti kovin konkreetteja.[1]

Asiakasrelaatioita on kahdentyyppisiä: samalla abstraktiotasolla olevat asiakkaat/toimittajat sekä matalammalla abstraktiotasolla olevat toimittajat. Jälkimmäisiä tarvitaan, koska oliosuunnittelu pohjautuu ns. *kokoavaan* (*bottom-up*) rakentamisajatteluun, missä ensin tehdään perustason komponentit, joiden päälle rakennetaan seuraava asiakastaso jne. Tätä suunnittelua ohjaa jossakin määrin päämäärä, johon pyritään eli kokoavassa suunnittelussa on aina mukana myös *jäsentävä* (*top-down*) aspekti, missä annettua tehtävää tarkastellaan ylhäältä alaspäin: jos tehtävää ei osata ratkaista suoraan, se pyritään jakamaan pienempiin osiin, joiden toteutukset voivat vaatia edelleen osiinjakoa jne. Sekä kokoava että jäsentävä suunnittelu perustuvat ajatukseen, että ohjelmistosysteemi koostuu päällekkäisistä abstrahointitasoista. Kullakin tasolla jätetään huomioimatta joitakin alemman tason yksityiskohtia, jotta ongelma nähtäisiin yksinkertaisempana eikä koodin lukijaa rasitettaisi epärelevanteilla yksityiskohdilla. Tämä periaate onkin oleellinen hallittaessa suurten ohjelmistosysteemien kompleksisuutta. Jos sitten keskitytään tarkastelemaan *samalla abstraktiotasolla* olevia luokkia havaitaan, että juuri niiden löytäminen, rajaaminen ja niille vastuiden jakaminen (julkisen liitännän valinta) on ohjelmoijan kannalta ehkä kaikkein vaikeinta. Samalla tasolla olevat luokat kuuluvat tyypillisesti samaan pakettiin eli niitä sitoo loogisesti jokin laajahko käsite yhteen (esim. käyttöliittymät, tietokannat). Relevanttien luokkien määrääminen ja vastuiden jako riippuu aina siitä näkökulmasta, jonka ohjelmoija haluaa ottaa mallinnettavaan ulkoiseen maailmaan.

Tässä luvussa pyritään valottamaan hieman ohjelmiston analysointi- ja implementointivaiheesiin liittyviä ongelmia esimerkkien avulla. Tarkoituksena on miettiä, miten systeemille asetettujen vaatimusten mukaiset toiminnot saadaan aikaan ja mitä korkean tason ratkaisuja toteutuksessa käytetään. Esimerkkien kohdalla pyritään selvittämään eri ratkaisuvaihtoehtoja sekä lopulliseen rakenteeseen johtaneet syyt. Ennen sitä tarkastellaan kuitenkin yleisesti, miten luokkia voidaan löytää (tai hylätä) ohjelmiston suunnitteluvaiheessa.

4.1 Luokkien hahmottaminen

Suuren ohjelmiston rakentamisessa ensimmäisiä vaiheita on ongelmarajauksen sisältyvän maailman analysointi ja mahdollisimman hyvän mallin rakentaminen ko. analyysin pohjalta. Tässä vaiheessa selvitetään mm. ketkä osallistuvat systeemiin

[1] Toki ohjelmistoa voidaan rakentaa ottamalla olemassa olevia luokkia käyttöön periytymisen avulla ja siten lisätä uudelleenkäyttöä, mutta se on toteutusvalinta, ei mallintamisratkaisu. Toisaalta esimerkiksi pankkitoiminnan kuvaamisessa voidaan muodostaa ensin yleistä tilikäsitettä vastaava rajapinta tai abstrakti luokka ja tehdä sille jälkeläiset, mutta tämäntyyppiset ratkaisut osataan tehdä vasta, kun sovellusalue tunnetaan perinpohjin. Summa summarum: periytymisrelaatio on ohjelman rakentamisvaiheessa huomattavasti harvinaisempi kuin asiakasrelaatio.

ja miten tieto siinä kulkee. Mallin rakenne koostuu kuvattavaan kohteeseen liittyvistä luokista ja niiden välisistä yhteyksistä. Luokkien hakeminen on vaativa tehtävä, sillä relevantit luokat ovat sovelluskohtaisia eikä kaikissa tilanteissa päteviä yleisohjeita niiden löytämiseksi voida antaa.

Lähtökohta luokkien muodostamiselle on kuitenkin selvä: *jokainen luokka rakentuu aina sen sisältämän tiedon ympärille.* Jokaiseen mallinnettavaan käsitteeseen liittyy siis aina tietoa, joka määrää olion kulloisenkin tilan. Kun keskeinen tietosisältö on määritetty, *sitä käsittelevät rutiinit sijoitetaan samaan luokkaan.* Tämä periaate tuntuu yksinkertaiselta, mutta käytännössä on joskus hankala päättää, mihin luokkaan kukin operaatio tulisi sijoittaa. Joitakin ohjenuoria voidaan kuitenkin antaa. Esimerkiksi silloin, kun asiakasrelaatiot muodostavat vähänkin pitemmän ketjun toisiinsa liittyviä käsitteitä (ks. kuvan 4.2 luokkakaaviota), *piirre on syytä sijoittaa asiakaskaaviossa niin kauas oikealle ("primitiivisimpään" asiakkaaseen), missä sillä vielä on relevanssia.* Vastaavasti periytymishierarkiassa *piirre viedään niin korkealle abstraktiotasolle, missä se vielä sopii kokonaisuuteen.* Nämä periaatteet on syytä pitää mielessä, kun luokkia etsitään ja luokkakokonaisuutta hahmotellaan.

Usein mainittu ohje "hae systeemin määrittelevästä dokumentista kaikki substantiivit ja muodosta niitä vastaavat luokat" kertoo yksikäsitteisesti, miten luokat löydetään. Ohjeeseen on kuitenkin syytä suhtautua suurella varauksella, vaikka siinä saattaa osa totuutta piilläkin. Jo rutiinien määrittelyjen yhteydessä huomattiin, miten moniselitteistä luonnollinen kieli on ja miten helposti sen avulla lausuttu asia voidaan käsittää toisin kuin alunperin oli tarkoitus. Sitäpaitsi, luonnollisen kielen lauseet on usein helppo muuntaa samansisältöisiksi korvaamalla substantiivit verbeillä ja päinvastoin. Joskus systeemidokumentin kuvailema tilanne voi sisältää myös käsitteitä, joista ei ole lainkaan syytä muodostaa omaa luokkaa.

Esimerkki 4.1 Lauseen "Hissi sulkee ovensa ennen kuin se siirtyy toiseen kerrokseen" perusteella voitaisiin päätellä, että hissin toimintaa kuvaavaan systeemiin tarvitaan luokat `Hissi`, `Ovi` ja `Kerros`. Käytännössä lienee kuitenkin niin, että ovi ei ole hissin toiminnan kannalta kovinkaan keskeinen; yleensä riittää tieto siitä, onko ovi kiinni vai ei. Jos tällä tullaan toimeen, tieto voidaan tallettaa esimerkiksi `Hissi`-luokan **`boolean`**-tyyppiseen attribuuttiin.

Tämän lisäksi voidaan pohtia, onko `Ovi` erillinen tietotyyppi, jolla on omat selvästi tunnistettavat operaatiot ja oma (attribuuttien määrittelemä) tilansa. Joissakin tilanteissa vastaus saattaisi olla kyllä, esimerkiksi oven liikkumismekanismia ja sensoreita mallintavassa systeemissä, mutta hissin liikkumista kuvattaessa em. suljettu/avoin -tilainformaatio on todennäköisesti riittävä. Voidaan myös kysyä, onko luokka `Kerros` tarpeellinen? Mitä ominaisuuksia — muita kuin kerrosnumerot, jotka voidaan kuvata yksinkertaisesti **`int`**-tiedolla[2] — kerroksella on tässä yhteydessä? Jos ei mitään, vastaavaa luokkaa ei

[2]Ellei rakennuksesta puutu 13. kerros...

tarvita. Jos hissin liikkeitä halutaan kontrolloida niin, että joihinkin kerroksiin on pääsyoikeus vain etuoikeutetuilla käyttäjillä, kannattaa `Kerros`-luokka muodostaa, ja nimetä kullekin kerrosoliolle sille asetetut oikeudet.

Luokka `Siirtyminen`, joka sisältää tiedot lähtö- ja tulokerroksista ja tallettaa tietokantaan tiedon siirtymästään, voisi toimia keskeisenä komponenttina systeemissä vaikka tämännimistä substantiivia ei määrittelydokumentista löytyisikään.

Kyse on aina siis näkökulmasta, jonka ohjelmoija haluaa ottaa mallinnettavaan maailmaan: jos olion jokin ominaisuus tai operaatio ei ole relevantti kuvattavan systeemin kannalta, sitä ei ole syytä implementoida. Ohje substantiivien hakemisesta voi hyvinkin johtaa harhaan: se voi johdatella muodostamaan epärelevantteja luokkia ja toisaalta ohjelmiston rakentamisessa voidaan tarvita apuna luokkia, joita ei ulkoisesta maailmasta suoranaisesti löydy.

Luokkien valinta on prosessi, jossa etsitään hyviä kandidaatteja, arvioidaan ne ja lopulta hylätään tai hyväksytään osaksi rakennettavaa kokonaisuutta. Luokaksi ei yleensä kannata valita sellaista, joka koostuu vain yhdestä rutiinista tai jota kuvataan esimerkiksi sanomalla ”tämä luokka tulostaa tulokset”. Tällaiset luokat ovat nimittäin perua proseduraalisesta ajattelutavasta, missä systeemin katsotaan koostuvan toiminnoista, joilla haluttu lopputulos saavutetaan. Samasta on yleensä kyse myös silloin, kun luokan nimeksi on valittu verbi kuten `Jäsennä` tai `Tulosta`. Normaalisti nimi on substantiivi (`Hakupuu`, `Lista`,...) tai adjektiivi (`Comparable`, `Opettajuus`). Jälkimmäiset kuvaavat tavallisesti olion *toiminnallisia ominaisuuksia* (*behavioural property*), joita käytetään periytymisen kautta. Luokat, joissa ei ole lainkaan olion tilaan kohdistuvia operaatioita, voivat myös osoittautua virhevalinnoiksi. Sen sijaan esimerkiksi funktionaaliset operaatiototeutukset tuottavat aina uuden olion, eivätkä näin ollen muuta kutsun kohteena olevaa oliota. Tästä syystä tähänkin sääntöön on suhtauduttava yhtä suurella varauksella kuin aiempiin.

Ohjelmistoa tehtäessä ei yleensä lähdetä ”tyhjältä pöydältä” vaan aiempia, joko samaan tehtävään tai muuhun toimintaan liittyviä luokkakokonaisuuksia on jo olemassa. Luokkia kannattaakin hakea ensin valmiista ohjelmistoista ja kirjastoista. Jos näyttää siltä, että jokin luokka sopii omiin tarkoituksiin pienillä muunnoksilla, on syytä miettiä (a) onko luokka tehty vain sitä systeemiä silmälläpitäen, johon sitä on alunperin tarvittu, jolloin yleistämällä luokan piirteitä siitä saadaan käyttökelpoinen myös nykyisessä systeemissä vai (b) onko kannattavaa tehdä kirjastoluokalle perijä, jossa nykysysteemin vaatimat muutokset on toteutettu.

Luokkia haettaessa on usein käyttökelpoista miettiä, minkälaista käsitettä se edustaa. Jokaisen luokan voidaan katsoa kuuluvan johonkin seuraavista kategorioista (luokittelu ei ole poissulkeva siinä mielessä, että jonkin luokan voidaan katsoa kuuluvan kahteenkin eri kategoriaan):

Analyysiluokka Tällainen luokka on saatu suoraan annetun tehtävän määritte-

lystä. Täten se on mallinnettavan systeemin abstraktia tai konkreettia käsitettä kuvaava luokka.[3]

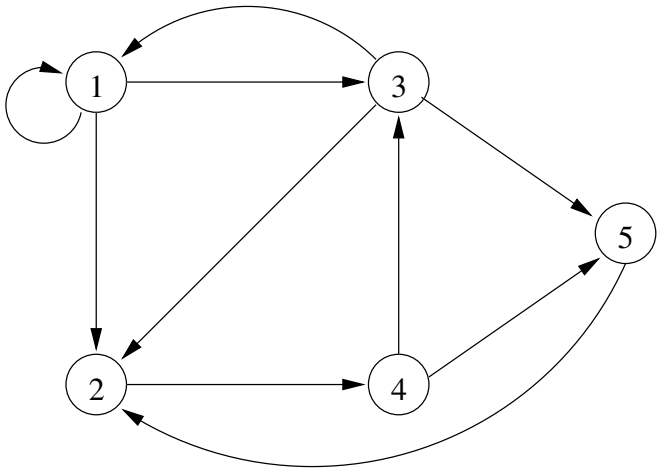
Suunnitteluluokka Ohjelmistosysteemin rakentamisessa voidaan tarvita luokkia, joilla ei ole suoraa vastinetta ongelmamaailmaan mutta jotka helpottavat toteutusta merkittävästi (esim. iteraattorit). Nämä luokat liittyvät usein käsitteisiin, joiden havaitseminen edellyttää abstraktia ajattelua.

Toteutusluokka Ohjelmistosysteemillä on tyypillisesti joitakin kyseiseen sovellusalueeseen kiinteästi liittyviä "alimman tason" luokkia, joita tarvitaan toistuvasti systeemin sisäistä toteutusta varten. Tällaisia ovat esimerkiksi erilaisia tietorakenteita (jonot, pinot, listat, puut, hajatustaulut jne.) edustavat luokat. Ohjelmoijan tärkeimpiin lähdeteoksiin kuuluukin aina jokin hyvä tietorakennekirja (esim. [3]). Vain hyvä tietorakennetuntemus auttaa tekemään hyviä luokkatoteutuksia.

Kun luokka on muodostettu, on syytä tarkastella sen tarjoamien piirteiden valikoimaa. Jos julkinen liitäntä kasvaa liian laajaksi, luokka voi kätkeä sisälleen enemmän kuin yhden hyvin määritellyn ja rajatun abstraktion. Tällöin on syytä miettiä, voitaisiinko osa toiminnallisuudesta sijoittaa toiseen (apuväline)luokkaan ja jättää alkuperäiseen luokkaan vain aivan keskeiset mallinnettavalle käsitteelle ominaiset operaatiot. Tämäntyyppinen tilanne tulee esille kohdassa 4.2, missä mietitään suunnatun graafin toteuttavaa luokkaa. Graafin perustoiminnot on helppo listata, mutta niiden lisäksi on helppo muodostaa lisävälineitä graafin läpikäyntiä varten. Rajanveto keskeisten ja "apuoperaatioiden" välillä voi olla vaikeaa. Kun julkinen liitäntä on mietitty — piirteitä ei ole liikaa, mutta valikoima on kuitenkin kattava — on vielä syytä tarkistaa rutiinien parametrilistat. Jos ne ovat pääsääntöisesti pitkät (esimerkiksi 4–6 parametria), pitää tarkistaa, onko niiden kautta välitetty tieto todellakin asiakkaalle kuuluvaa vai olisiko parempi laittaa niistä osa toimittajaluokan jäsenmuuttujiin ja tällä tavalla yksinkertaistaa liitäntää asiakkaan kannalta. Pitkät parametrilistat saattavat olla merkki siitä, että jokin keskeinen käsite on jäänyt huomaamatta.

Kun luokat on hahmoteltu, selvitetään tiedon kulku niiden välillä. Esimerkiksi *käyttötilanteiden* (*use cases*, *scenarios*) läpikäynti selkiyttää luokkien roolia tässä suhteessa. Yksittäinen käyttötilanne kertoo koko sen prosessin, joka tarvitaan systeemin (suhteellisen pienenkin) toiminnon läpiviemiseksi. Siihen osallistuu tyypillisesti jokin ohjelmiston käyttäjä. Esimerkiksi varastokirjanpidossa käyttötilanteena voidaan pitää tapahtumaa "asiakas tuli hakemaan tilauksensa", jolloin selvitetään

[3]Huomaa, että mallinnettavan systeemin abstraktia käsitettä voi vastata Java-ohjelmassa yksi tai useampi konkreetti luokka. Vastaavasti yksi luokka voi osallistua monen abstraktion konkretisointiin.



Kuva 4.1: Esimerkki suunnatusta kokonaislukugraafista.

koko tapahtumaketju tavaroiden luovutuksesta ja tilauslomakkeen käsittelystä tavaroiden maksuun. Kuten rutiinin toiminnallisessa kuvauksessa (ks. kohta 2.3.2), myös käyttötilanteen kuvaus sisältää yleensä temporaalisia (ajallisia) aspekteja. Tämän johdosta käyttötilannekuvaus on yleensä proseduraalinen sen sijaan, että keskityttäisiin olioparadigman mukaiseen tietotyyppiajatteluun.[4] Näistä syistä johtuen käyttötapauksien soveltamisen tulisi painottua analysointivälineen sijasta validointiin, jossa todetaan, että rakennettu systeemi toimii odotusten mukaisesti. Tietenkin halutun toiminnon kuvaavat käyttötilanteet antavat virikettä analysointivaiheessa siitä, minkälaiset ratkaisut voisivat toimia.

4.2 Esimerkki: suunnattu graafi

Suunnattu graafi (*directed graph*) $G = (V, E)$ koostuu solmujoukosta V ja kaarijoukosta E. Kukin kaari ilmaistaan muodossa (u, v), missä u on lähtö- ja v tulosolmu. Merkintä tarkoittaa sitä, että graafissa G on kaari solmusta u solmuun v. Solmu u on solmun v *edeltäjä* (*predecessor*) ja solmu v solmun u *seuraaja* (*successor*). Luokkakuvauksissa edeltäjästä käytetään myös nimitystä *lähtösolmu* ja seuraajasta nimitystä *tulosolmu*. Jotta solmuihin olisi helppo viitata, niihin liitetään normaalisti yksikäsitteiset leimat. Esimerkiksi kuvan 4.1 graafissa solmujen leimat ovat kokonaislukuja, solmujen joukko on $V = \{1, 2, 3, 4, 5\}$ ja kaarten joukko on $E = \{(1,1), (1,2), (1,3), (2,4), (3,1), (3,2), (3,5), (4,3), (4,5), (5,2)\}$.

Graafin *polku* (*path*) (v_i, v_j) määritellään sellaisten solmujen $v_1, v_2, ..., v_n$ jonona, missä $v_i = v_1$, $v_j = v_n$ ja kukin $(v_s, v_{s+1}),\ s = 1, \ldots, n-1$ on joukon E kaari.

[4] Aikajärjestysvaatimukset heijastuvat olio-ohjelmassa operaatioille asetettujen vaatimusten eli alku- ja loppuehtojen muodossa.

Esimerkiksi kuvan 4.1 graafista on löydettävissä kolmen pituinen polku $3, 1, 1, 2$. Usein kaariin liitetään jokin *paino* (*weight*), tyypillisesti reaaliluku, jolloin polun pituus voidaan määritellä, paitsi kaarten määränä, myös kuljettujen kaarten painojen summana. Jos kaarista poistetaan suunta, kyseessä on *suuntaamaton graafi* (*undirected graph*). Esimerkiksi tieverkostoa kuvaava graafi on yleensä suuntaamaton, koska matka kaupungista A kaupunkiin B on tavallisesti yhtä pitkä kuin matka kaupungista B kaupunkiin A. Sen sijaan, jos halutaan kuvata vaikkapa useampiosaisen työprosessin eri vaiheita, tuloksena on suunnattu graafi. Tällöin kukin kaari (u, v) kertoo, että solmuun u liittyvä työvaihe on tehtävä ennen solmuun v liittyvää työvaihetta. Kaaren painona on esimerkiksi työvaiheen vaatima aika.

4.2.1 Perusoperaatioita

Luokan `SuunnattuGraafi` perusoperaatioiden rutiinimäärittelyt on esitelty listauksessa 4.1. Solmut leimataan merkkijonoilla ja julkinen liitäntä on muodostettu siten, että asiakas pääsee käsittelemään solmuja ja kaaria vain leimojen kautta. Sisäisen toteutuksen tulee tietysti huolehtia siitä, että solmujen leimat pysyvät yksikäsitteisinä (kahdella eri solmulla ei ole samaa leimaa).

Esiteltyjen piirteiden lisäksi voisi olla hyödyllistä muodostaa iteraattorit, joilla saadaan graafin kaikki solmut ja kaaret käsiin. Hieman löyhemmin tähän luokkaan kuuluvat erilaiset graafikäsittelyn apuvälineet: (1) onko kahden annetun solmun välillä yhteys, (2) etsi lyhin polku kahden annetun solmun välillä, (3) etsi annetusta solmusta lähtien toista, tietyn leiman (tai muun ominaisuuden) omaavaa solmua käyttäen *leveyshakua* (*breadth-first search*), (4) toteuta edellinen *syvyyshaulla* (*depth-first search*) jne. Graafin käyttötarkoituksen mukaan piirrevalikoimaa voitaisiin laajentaa vielä näitäkin erikoisemmilla operaatioilla.

Perusoperaatioihin kuuluvat kaaren lisääminen, jossa annetaan lähtö- ja tulosolmujen leimat sekä kaaren paino. Jos kaari toteutetaan lähtösolmun olioviittauksena tulosolmuun, operaation toteutuksessa tulisi etsiä kyseisiä solmuja vastaavat oliot ja lisätä lähtösolmun seuraajaviittausten joukkoon viittaus tulosolmuun. Ongelmaksi tulee kuitenkin kaaren painon tallettaminen, koska sitä ei voi liittää olioviittaukseen. Helppo ratkaisu olisi sijoittaa lähtösolmuun lista, johon kaarten painot talletetaan. Tämä on huono ratkaisu siinä mielessä, että silloin itse kaari ja sen paino eivät ole enää loogisesti toisiinsa yhteydessä (joskin tämän tulisi näkyä selkeästi luokkainvarianttissa).

Jätetään tämä pohdiskelu kuitenkin kesken, koska meillä on paljon hankalampikin ongelma: miten päästä alunperin käsiksi lähtö- ja tulosolmuihin? Jokin graafin solmuista voitaisiin asettaa erikoisasemaan siinä mielessä, että siitä lähtien voidaan käydä graafia läpi, kunnes halutut solmut löytyvät. Mikään ei kuitenkaan takaa, että tästä erikoissolmusta aina päästäisiin lähtö- ja tulosolmuihin. Tämän takia meidän tulisikin muodostaa jonkinlainen ”hakemisto” kaikille solmuille. Yksinker-

Listaus 4.1: Luokan `SuunnattuGraafi` rutiinimäärittelyjä.

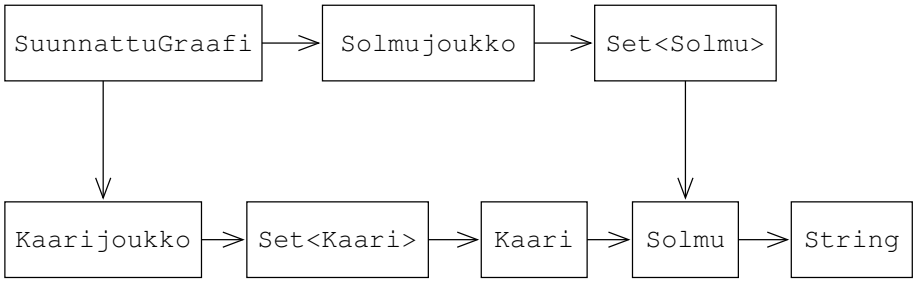
```
public class SuunnattuGraafi
  {
  /**
   *  @.pre  true
   *  @.post solmumäärä() == 0 & kaarimäärä() == 0
   */
  public SuunnattuGraafi()

  //-- Havainnointioperaatiot
  /**
   *  @.pre  true
   *  @.post RESULT == (solmujen määrä)
   */
   public int solmumäärä()

  /**
   *  @.pre  true
   *  @.post RESULT == (kaarten määrä)
   */
  public int kaarimäärä()



  /**
   *  @.pre  leima != null
   *  @.post RESULT == (leimalla varustettu solmu on graafissa)
   */
  public boolean sisältääSolmun(String leima)

  /**
   *  @.pre  lähtöleima != null & tuloleima != null
   *  @.post RESULT == (leimoilla varustettu kaari on graafissa)
   */
  public boolean sisältääKaaren(String lähtöleima, String tuloleima)
```

Listaus 4.1 (jatkoa): Luokan `SuunnattuGraafi` rutiinimäärittelyjä.

```
/**
 *  @.pre  (lähtöleima != null & tuloleima != null) &&
 *         sisältääKaaren(lähtöleima, tuloleima)
 *  @.post RESULT == (annetun kaaren paino)
 */
public double annaPaino(String lähtöleima, String tuloleima)

/**
 *  @.pre  true
 *  @.post RESULT == (solmujoukon läpikäyntiolio)
 */
public Iterator<Solmu> solmuIteraattori()

/**
 *  @.pre  true
 *  @.post RESULT == (kaarijoukon läpikäyntiolio)
 */
public Iterator<Kaari> kaariIteraattori()

/**
 *  @.pre  leima != null && sisältääSolmun(leima)
 *  @.post RESULT == (solmun edeltäjät)
 */
public Solmujoukko edeltäjät(String leima)

/**
 *  @.pre  leima != null && sisältääSolmun(leima)
 *  @.post RESULT == (solmun seuraajat)
 */
public Solmujoukko seuraajat(String leima)

//-- Muunnosoperaatiot
/**
 *  @.pre  leima != null && !sisältääSolmun(leima)
 *  @.post RESULT.sisältääSolmun(leima) &
 *         RESULT.poistaSolmu(leima).equals(this)
 */
public SuunnattuGraafi lisääSolmu(String leima)
```

Listaus 4.1 (jatkoa): Luokan `SuunnattuGraafi` rutiinimäärittelyjä.

```
/**
 *  @.pre  leima != null && sisältääSolmun(leima) &
 *         edeltäjät(leima).koko() == 0 & seuraajat(leima).koko() == 0
 *  @.post !RESULT.sisältääSolmun(leima) &
 *         RESULT.lisääSolmun(leima).equals(this)
 */
public SuunnattuGraafi poistaSolmu(String leima)

/**
 *  @.pre  (lähtöleima != null & tuloleima != null) &&
 *         sisältääSolmun(lähtöleima) & sisältääSolmun(tuloleima) &
 *         !sisältääKaaren(lähtöleima, tuloleima)
 *  @.post RESULT.sisältääKaaren(lähtöleima, tuloleima) &
 *         RESULT.poista(lähtöleima, tuloleima).equals(this) &
 *         RESULT.annaPaino(lähtöleima, tuloleima) == paino
 */
public SuunnattuGraafi lisääKaari(String lähtöleima, String tuloleima,
                                  double paino)

/**
 *  @.pre  (lähtöleima != null & tuloleima != null) &&
 *         sisältääKaaren(lähtöleima, tuloleima)
 *  @.post !RESULT.sisältääKaaren(lähtöleima, tuloleima) &
 *         RESULT.lisää(lähtöleima, tuloleima).equals(this)
 */
public SuunnattuGraafi poistaKaari(String lähtöleima, String tuloleima)

/**
 *  @.pre  (lähtöleima != null & tuloleima != null) &&
 *         sisältääKaaren(lähtöleima, tuloleima)
 *  @.post RESULT.equals(this) &
 *         RESULT.annaPaino(lähtöleima, tuloleima) == uusiPaino
 */
public SuunnattuGraafi asetaPaino(String lähtöleima, String tuloleima,
                                  double uusiPaino)

}
```



Kuva 4.2: Graafisysteemin luokkarakenne.

taisimmillaan hakemisto on joukko, josta on viittaus kaikkiin solmuihin. Tämä toimisi samalla kaikkien graafisolmujen yhteenkokoavana luokkana `Solmujoukko`, joka toteuttaa suoraan graafin $G = (V, E)$ joukon V. Samalla tavoin kaikki kaaret (so. joukko E) voitaisiin koota omaan luokkaansa `Kaarijoukko`. Mutta jos näin tehdään, voidaan jo kysyä, mihin tarvitaan kussakin solmussa fyysisesti sijaitsevia viittauksia toisiin. Ne tukevat joitakin havainnointioperaatioita kuten seuraajasolmujen hakua, mutta niiden merkitys vähenee oleellisesti hakemistojen käytön ansiosta. Tämä on yleinen *trade-off* -tilanne: mitä enemmän systeemiin lisätään apurakenteita eri operaatioiden tueksi, sitä helpommin rutiinit saadaan toteutettua. Apurakenteiden ylläpito kuitenkin maksaa ajoajassa ja näkyy joidenkin operaatioiden lievänä hidastumisena. Systeemiin tulee myös (lisää) redundanssia, joten tietojen konsistenttina pitämiseen on käytettävä enemmän työtä.

Jotta koko systeemin hahmottaminen olisi helpompaa, havainnollistetaan ratkaisua kuvan 4.2 asiakassuhteet ilmaisevalla luokkakaaviolla. Lähtökohtana on siis kaksi perusluokkaa, joilla toteutetaan suunnatun graafin yksittäinen solmu ja kaari. Käydäänpä seuraavaksi lyhyesti läpi nämä apuluokat ennen kuin palataan takaisin ne yhdistävään graafiluokkaan.

4.2.2 `Solmu`

Luokassa `Solmu` on tehty seuraavat valinnat:

- Koska solmun leima on mutatoitumaton merkkijono, se voidaan huoletta antaa asiakkaiden käsiin sellaisenaan; kukaan ei pysty leimaa muuttamaan kun se on kerran asetettu.

- Kahden solmun (pinta)samuus (`equals`) määräytyy niiden leimoista. Solmun haku solmujoukosta toteutetaan `equals`-operaatiolla ja siksi on tärkeää, että `Object`-luokalta peritty `equals` toteutetaan uudelleen.

Tietysti voisi kysyä, tarvitaanko luokkaa ollenkaan, sillä eikö se voisi olla suoraan `String`. On toki mahdollista että solmuihin lisätään myöhemmin jotain uusia piir-

teitä, joten mallinnetaan solmu omaksi käsitteekseen.

4.2.3 Kaari

Luokka `Kaari` käyttää hyväkseen luokkaa `Solmu`. Lähtö- ja tulosolmua ei voi muuttaa sen jälkeen kun ne on kerran (luonnin yhteydessä) asetettu (eli kaaren ei tarvitse olla kloonautuva), minkä vuoksi ne on syytä määritellä vakioiksi (**`final`**). Myös tässä luokassa `Object`-luokalta perityt rutiinit kannattaa toteutettaa uudelleen. Erityisen huomion arvoinen yksityiskohta on `equals`-rutiinin määrittely

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!getClass().equals(toinen.getClass())) return false;
  return ((Kaari)toinen).lähtösolmu.equals(lähtösolmu) &
         ((Kaari)toinen).tulosolmu.equals(tulosolmu);
  }
```

jossa siis vähät välitetään kaaren painosta.

4.2.4 Solmujoukko ja Kaarijoukko

Solmujoukkoa vastaava luokka `Solmujoukko` on nyt helppo toteuttaa. Huomion arvoiset yksityiskohdat ovat:

- Luokan havainnointioperaatiot delegoidaan suoraan sisäisen esitysmuodon `Set`-tyyppiselle kokoelmaoliolle mistä syystä `Solmujoukko`-operaatioihin tarvitaan vain yksinkertainen ”kuori”. Tämä kuori määrittelee tarkoituksenmukaisen semantiikan joukkokäsitteen päälle.

- Luokan julkinen liitäntä on tehty solmujen leimoille, asiakas ei ole lainkaan tekemisissä `Solmu`-tyyppisten tietojen kanssa. Tämän takia solmuleimojen eheysehdot on otettava mukaan alkuehtoihin tahi on tehtävä sopimus että ne tulevat implisiittisesti huomioon otetuiksi.

- Kaikki operaatiot on toteutettu `LukuJoukko`-luokan (ks. kohta 3.1) tapaan funktionaalisesti.

Graafin kaikki kaaret kootaan vastaavalla tavalla yhteen luokalla `Kaarijoukko`, johon pätee edellä kuvatut ominaisuudet.

4.2.5 `SuunnattuGraafi`

Nyt kaikki perusosat on saatu kasaan ja voidaan edetä listauksessa 4.1 määritellyn `SuunnattuGraafi`-luokan toteuttamiseen. Huomaa, että graafissa voi olla kaarettomia solmuja, mutta kaikkien kaarten päiden pitää aina olla olemassaolevia solmuja. Muita toteutukseen liittyviä huomioita on:

- Kun solmu poistetaan, kutsujan tulisi taata alkuehto eli ettei solmuun/solmusta ole enää kaaria, koska muuten kokonaisuus ei pysy järkevänä. Tämän testaamiseksi toimittajan on tarjottava validointirutiini, joka voidaan toteuttaa esimerkiksi ylläpitämällä laskuria, joka kertoo, montako lähtevää/tulevaa kaarta ko. solmuun liittyy. Valitulla toteutustavalla sama asia saadaan hoidettua vähän työläämmin, käymällä kaikkien kaarten joukko läpi ja tarkistamalla. Kolmas keino olisi käyttää jotain vähän fiksumpaa tietorakennetta kuten hajautustaulua. Tämä solmun poistoon liittyvä vaatimus tunnetaan vasta `SuunnattuGraafi`-tasolla, joten `Solmujoukko`- ja `Kaarijoukko`-luokkiin sitä ei voida sijoittaa — erityisesti siitäkin syystä, että joissakin jälkimmäisiä luokkia käyttävissä sovelluksia tämäntyyppisiä vaatimuksia ei haluta noudattaa.

- Graafiluokan julkinen liitäntä on valittu siten, että solmuihin viitataan aina leimoilla, koska (a) se on asiakkaan kannalta luonnollisin tapa ja (b) näin solmujen toteutus voidaan suojata asiakkaalta. Mikäli solmun sisältö on selkeästi rakenteisempi ja asiakas haluaa käsitellä solmuja suoraan, kannattaa `Solmu`-tyyppiset tiedot tuoda asiakkaan ulottuville.

- Kannattaa huomata, mihin luokkiin eri operaatiot on sijoitettu. Jokaisen paikka on valittu erikoistuneimmasta luokasta, johon käsite liittyy.

`SuunnattuGraafi`-luokan yhteydessä tulee esille kysymys luokan julkisen liitännän laajuudesta. Esimerkiksi rutiinin `poistaSolmu` alkuehtona on, että solmun edeltäjien ja seuraajien joukot ovat tyhjät, jolloin nämä operaatiot on luonnollista kirjoittaa samaan luokkaan. Entäpä yhteyden tutkiminen kahden mielivaltaisen solmun välillä? Vastaavan polun paino? Lyhin polku ko. välillä? Ja niin edelleen. Missä kohtaa pitää pysähtyä ja lopettaa julkisen liitännän paisuttaminen? Periaatteena on tietysti se, että pitäisi pitäytyä ”puhtaissa” perusoperaatioissa, mutta rajan vetäminen voi joskus olla hankalaa niin kuin tämäkin esimerkki osoittaa. Saattaisi olla järkevää sijoittaa rutiini annetun yhteyden tutkimiseksi `SuunnattuGraafi`-luokkaan, mutta kaikki polkujen erikoisominaisuuksia tutkivat operaatiot jo toiseen avustavaan luokkaan. Java-filosofian mukaan ne tulisivat olemaan **`static`**-tyyppisiä luokkametodeja, joille annetaan tutkittava graafi argumenttina. On olemassa myös muita keinoja erottaa käsiteltävät oliot käsittelyrutiineista.

Paitsi että luokan julkista liitäntää yritetään pitää yksinkertaisena miettimällä mitkä ovat sen perusoperaatiot ja niiden johdannaisoperaatiot, yksinkertaisuutta voi tavoitella myös miettimällä minkä abstraktion luokka tarjoaa. Graafiesimerkin abstraktiota voidaan yksinkertaistaa poistamalla solmun käsite graafin abstraktiosta ja jättämällä asiakkaan huoleksi mitä solmulla tarkoitetaan. Traafin keskeiseksi abstraktioksi nousee siis kaari ja sen ominaisuudet (esim. kaaren leima). Tällä lähestymistavalla voidaan rakentaa listauksen 4.2 mukainen määrittely `Graafi`-luokalle. Toteutuksesta kannattaa huomata, että kaaresta on tehty graafin esiintymäkohtainen sisäluokka (ks. kohta 3.3.2 ja että solmu esiintyy graafin tyyppiparametrina (ks. luku 7).

Tämä graafiesimerkki antanee kalpean aavistuksen siitä, miten laajoja kokonaisuuksia rakennetaan askeleittain. Tässä tilanteessa erilaisten systeemiä kuvaavien graafisten esitysten rooli alkaa myös korostumaan: enää ei riitä se, että on ohjelmalistaukset eri luokista, koska luokkien julkista liitäntää ja luokkien välisiä relaatioita ei ole helppo hahmottaa kokonaisuutena ohjelmalistauksista. On siirryttävä korkeammalle abstraktiotasolle ja käsiteltävä luokkia (sekä pakkauksia) peruskomponentteina, joiden väliset suhteet, sekä staattiset että dynamiset, kuvataan korkean tason esitystapaa käyttäen.

Tehtäviä

4-1 Toteuta `SuunnattuGraafi`-luokkaan operaatio

```
public boolean onPolku(String[] leimat)
```

joka palauttaa tiedon, löytyykö annettua merkkijonotaulukkoa vastaava polku graafista. Taulukon merkkijonot ovat graafin solmujen nimiä. Taulukon ensimmäinen merkkijono on polun aloitussolmun nimi ja taulukon viimeinen merkkijono polun päätepisteen nimi. Taulukon muut solmut ovat polun kulkujärjestyksessä.

4-2 `SuunnattuGraafi`-luokka on nimensä mukaisesti tarkoitettu suunnatuille graafeille. Miten toteutusta tulisi muuttaa, jos haluttaisiin käsitellä suuntaamattomia graafeja (ts. kutakin kaarta voidaan liikkua kumpaankin suuntaan)?

4-3 Monia pelilautoja voidaan ajatella graafeina. Tyypillisiä esimerkkejä ovat pelit, joissa pelaajan nappula on jossakin pelilaudan ruudussa (solmussa), ja noppaa heittämällä tai rahaa maksamalla siirrytään (kaaria pitkin) joidenkin sääntöjen mukaisesti toiseen ruutuun. Tällaisia pelejä ovat mm. Afrikan tähti (kaupungit, maa-, vesi- ja ilmareitit) ja Monopoli (seuraava ruutu, vankilaan joutuminen, sattuma-korttien hypyt).

Tässä tehtävässä tarkastellaan pelejä, joissa aloitetaan lähtösolmusta `"L"` ja tavoitteena on päästä neljäsivuista noppaa heittämällä maalisolmuun `"M"`. Pelikentän solmuja yhdistävät kaaret, joiden painot ovat joko `-1`, `0` tai `1`. Jokaisesta solmusta lähtee

Listaus 4.2: Luokan `Graafi` rutiinimäärittelyjä.

```
/** Graafi muodostuu solmuista ja niitä yhdistävistä kaarista.
 *  Solmun tyyppi on V ja kaaren leiman tyyppi on L.
 */
public class Graafi<V, L>
  {
  //-- Sisäluokat
  /** Kahden solmun välinen kaari, joka voidaan varustaa leimalla.
   */
  public class Kaari
    {
    //-- Konstruktorit
    /** Alustaa leimalla varustetun kaaren.
     *  @.pre  u != null & v != null
     *  @.post annaLähtösolmu() == u & annaTulosolmu() == v &
     *         annaLeima() == leima
     */
    protected Kaari(V u, V v, L leima)

    //-- Havainnointioperaatiot
    /** Palauttaa kaaren lähtösolmun.
     *  @.pre  true
     *  @.post RESULT == (lähtösolmu)
     */
    public V annaLähtösolmu()

    /** Palauttaa kaaren tulosolmun.
     *  @.pre  true
     *  @.post RESULT == (tulosolmu)
     */
    public V annaTulosolmu()

    /** Palauttaa kaaren leiman.
     *  @.pre  true
     *  @.post RESULT == (kaaren leima; voi olla null)
     */
    public L annaLeima()
    } // class Kaari
```

Listaus 4.2 (jatkoa): Luokan `Graafi` rutiinimäärittelyjä.

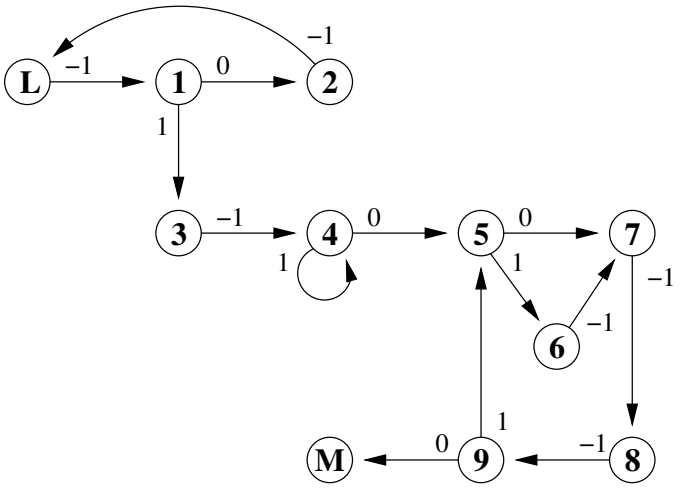
```
//-- Konstruktorit
/** Alustaa tyhjän (suunnatun) graafin.
 *  @.pre  true
 *  @.post annaSolmut().size() == 0 & annaKaaret().size() == 0
 */
public Graafi()

//-- Havainnointioperaatiot
/** Palauttaa graafin solmujoukon.
 *  @.pre  true
 *  @.post RESULT == (graafin solmut)
 */
public Set<V> annaSolmut()

/** Palauttaa graafin kaarijoukon.
 *  @.pre  true
 *  @.post RESULT == (graafin kaaret)
 */
public Set<Kaari> annaKaaret()

/** Palauttaa onko kaari graafissa.
 *  @.pre  true
 *  @.post RESULT == annaKaaret().contains(
 *                       new Graafi.Kaari(u, v, eiVäliä))
 */
public boolean sisältääKaaren(V u, V v)

/** Palauttaa annetun kaaren.
 *  @.pre  sisältääKaaren(u, v)
 *  @.post RESULT.annaLähtösolmu().equals(u) &
 *         RESULT.annaTulosolmu().equals(v)
 */
public Kaari annaKaari(V u, V v)
```

Listaus 4.2 (jatkoa): Luokan `Graafi` rutiinimäärittelyjä.

```
/** Palauttaa solmusta lähtevät kaaret.
 *  @.pre  annaSolmut().contains(v)
 *  @.post FORALL(e : RESULT; e.annaLähtösolmu().equals(v)) &
 *           !EXISTS(e : annaKaaret();
 *           e.annaLähtösolmu().equals(v) & !RESULT.contains(e))
 */
public Set<Kaari> annaLähtevätKaaret(V v)

/** Palauttaa solmuun saapuvat kaaret.
 *  @.pre  annaSolmut().contains(v)
 *  @.post FORALL(e : RESULT; e.annaTulosolmu().equals(v)) &
 *           !EXISTS(e : annaKaaret();
 *           e.annaTulosolmu().equals(v) & !RESULT.contains(e))
 */
public Set<Kaari> annaSaapuvatKaaret(V v)

//-- Muunnosoperaatiot
/** Lisää solmun.
 *  @.pre  !annaSolmut().contains(v)
 *  @.post annaSolmut().contains(v)
 */
public void lisääSolmu(V v)

/** Poistaa solmun.
 *  @.pre  annaSolmut().contains(v) &
 *         (annaLähtevätKaaret(v).size() == 0 &
 *          annaSaapuvatKaaret(v).size() == 0)
 *  @.post !annaSolmut().contains(v)
 */
public void poistaSolmu(V v)
```

Listaus 4.2 (jatkoa): Luokan `Graafi` rutiinimäärittelyjä.

```
/** Lisää leimalla varustetun kaaren.
 *  @.pre  (annaSolmut().contains(u) & annaSolmut().contains(v)) &&
 *                !sisältääKaaren(u, v)
 *  @.post OLD(this).equals(this.poistaKaari(u, v)) &
 *                this.annaKaari(u, v).annaLeima().equals(leima)
 */
public void lisääKaari(V u, V v, L leima)

/** Poistaa kaaren.
 *  @.pre  sisältääKaaren(u, v)
 *  @.post OLD(this).equals(this.lisääKaari(u, v,
 *                          OLD(this).annaKaari(u, v).annaLeima()))
 */
public void poistaKaari(V u, V v)

//-- Arvosemanttiset operaatiot
/** Tarkistaa graafien samuuden.
 *  @.pre  true
 *  @.post RESULT ==
 *         (sisältääkö graafi toinen samat solmut ja kaaret
 *          kuin tämä graafi ja ovatko molemmat graafit joko
 *          suunnattuja tai suuntaamattomia)
 */
public boolean equals(Object toinen)
} // class Graafi
```



Kuva 4.3: Pelilauta graafina.

yksi tai kaksi kaarta (paitsi maalisolmusta ei yhtään). Pelissä siirrytään kaaria pitkin siten että jokaista nopan silmälukua kohti ohitetaan yksi kaari. Jos solmusta lähtee kaksi kaarta, valitaan suunta seuraavasti:

- valitaan kaari, jonka paino `== 0`, jos nopan silmäluku on parillinen
- valitaan kaari, jonka paino `== 1`, jos nopan silmäluku on pariton

Kun solmusta lähtee vain yksi kaari, niin sen paino on aina `-1`; tällöin luonnollisesti edetään tätä yhtä kaarta pitkin, oli silmäluku mikä tahansa.

Pelaaja ei siis saa valita missään solmussa mihin suuntaan haluaa lähteä, vaan aina edetään em. sääntöjen mukaan. Maaliin ei tarvitse päästä tasaluvulla (silmälukuja saa jäädä ”yli”), mutta muutoin täytyy siirtyä silmäluvun osoittama matka.

Kuvan 4.3 esimerkkigraafi voitaisiin luoda `SuunnattuGraafi`-määrittelyllä seuraavasti:

```
SuunnattuGraafi peligraafi = new SuunnattuGraafi();

// Lisätään solmut ja painot.
peligraafi = peligraafi.lisääSolmu("L")
                       .lisääSolmu("1")
                       .lisääSolmu("2");
// jne...
peligraafi = peligraafi.lisääKaari("L", "1", -1.0)
                       .lisääKaari("1", "2", 0.0)
                       .lisääKaari("1", "3", 1.0)
                       .lisääKaari("2", "L", -1.0);
// jne...
```

Tehtävänä on toteuttaa pelin siirtologiikan ydinmetodi `siirry`, joka palauttaa sen solmun leiman, johon päädytään lähtemällä `nykysolmu`sta nopanheitolla `silmäluku`:

```
/**
 *  @.pre  peligraafi.sisältääSolmun(nykysolmu) & 1 <= silmäluku <= 4
 *  @.post RESULT == (solmun leima, johon päädytään nykysolmusta
 *                    silmäluvulla kun ja noudatetaan
 *                    siirtymissääntöjä)
 */
public String siirry(String nykysolmu, int silmäluku)
```

Esimerkkigraafissamme metodi toimisi siis seuraavien käyttöesimerkkien mukaisesti:

- `siirry("L", 4)` palauttaa `"1"`, kulkureitti L → 1 → 2 → L → 1
- `siirry("L", 3)` palauttaa `"4"`, kulkureitti L → 1 → 3 → 4
- `siirry("8", 3)` palauttaa `"6"`, kulkureitti 8 → 9 → 5 → 6

4-4 Listauksen 4.3 `Miespalvelija` ja listauksen 4.4 `Isäntä` muodostama luokkapari toteuttaa ns. *alisteisen 1:1 -assosiaation* kahden olion välille. Alisteisuus tarkoittaa tässä sitä että assosiaation omistajana (synnyttäjänä ja purkajana) on aina jokin `Isäntä`-olio. Tämä taataan luokan `Miespalvelija` rutiinin `asetaIsäntä(Isäntä)` alkuehdolla, joka estää assosiaation muodostamisen muista kuin `Isäntä`-olioista.

(a) Piirrä ajoaikainen kuva olioiden välisistä riippuvuuksista, kun seuraava ohjelmakoodi on suoritettu.

```
Miespalvelija jeeves = new Miespalvelija();
Isäntä wooster = new Isäntä();
wooster.asetaMiespalvelija(jeeves);
```

(b) Täydennä luokka `Isäntä` `annaToveri`-funktion ja `asetaToveri(Isäntä)`-proseduurin määrittelyillä, jotta `Isäntä`-oliot kykenisivät muodostamaan *tasa-arvoisia 1:1 -assosiaatioita* keskenään. Tasa-arvoisuus tarkoittaa tässä sitä että kumpi tahansa `Isäntä`-olio saa synnyttää tai purkaa liitoksen.

(c) Miksi `Isäntä`-luokan proseduurin `asetaMiespalvelija` alkuehto ei voi olla yksinkertaisesti vain **`true`**?

4-5 Kun tehtävän 4-4 luokat on määritelty, siirrytään seuraavaksi koodaamaan. Kuinka toteuttaisit seuraavat kyseisen tehtävän rutiinit:

- `asetaIsäntä(Isäntä)`
- `asetaMiespalvelija(Miespalvelija)`
- `asetaToveri(Isäntä)`

Listaus 4.3: Luokka `Miespalvelija`.

```
public class Miespalvelija
  {
  /**
   *  Oletusmiespalvelija.
   *  @.pre  true
   *  @.post annaIsäntä() == null
   */
  public Miespalvelija() { /* Toteutus poistettu. */ }

  /**
   *  Palauttaa this-miespalvelijan mahdollisen isännän.
   *  @.pre  true
   *  @.post RESULT == (isäntä; jos ei ole, niin null)
   */
  public Isäntä annaIsäntä() { /* Toteutus poistettu. */ }

  /**
   *  Asettaa isännäksi m:n.
   *  @.pre  m != null && m.annaMiespalvelija() == this
   *  @.post this.annaIsäntä() == m
  public void asetaIsäntä(Isäntä m) { /* Toteutus poistettu. */ }
  }
```

Listaus 4.4: Luokka `Isäntä`.

```
public class Isäntä
  {
  /**
   *  Oletusisäntä.
   *  @.pre  true
   *  @.post annaMiespalvelija() == null
   */
  public Isäntä() { /* Toteutus poistettu. */ }

  /**
   *  Palauttaa isännän mahdollisen miespalvelijan.
   *  @.pre  true
   *  @.post RESULT == (miespalvelija; jos ei ole, niin null)
   */
  public Miespalvelija annaMiespalvelija()
    { /* Toteutus poistettu. */ }

  /**
   *  Asettaa isäntä-miespalvelija -suhteen mahdollisen s:n kanssa.
   *  @.pre  s == null || (s.annaIsäntä() == null &
   *         this.annaMiespalvelija() == null)
   *  @.post this.annaMiespalvelija == s & (s == null &
   *         (OLD(this.annaMiespalvelija()) == null ||
   *         OLD(this.annaMiespalvelija()).annaIsäntä() == null)) |
   *         (s != null & s.annaIsäntä() == this)
   */
  public void asetaMiespalvelija(Miespalvelija s)
    { /* Toteutus poistettu. */ }
  }
```

4-6 Täsmällinen määrittely helpottaa myös testauksen suunnittelua. Tarkastele alla olevia määrittelyjä ja mieti, millaisilla syötteillä niiden määrittelemiä metodeja testaisit.

```
/**
 *  @.pre  t != null
 *  @.post RESULT == EXISTS(i: 0 <= i < t.length; t[i] == x)
 */
public static boolean sisältää(int x, int[] t)

/**
 *  @.pre  t1 != null & t2 != null
 *  @.post RESULT == FORALL(x: t1; sisältää(x, t2))
 */
public static boolean sisältää(int[] t1, int[] t2)
```

4-7 Harjoitellaan hiukan luokka- ja oliokaavioiden piirtämistä.

(a) Binääripuu koostuu otsakkeesta, joka edustaa puuta asiakkaan näkökulmasta, ja solmuista, jotka muodostavat puun varsinaisen rakenteen. Otsake tuntee puun juurisolmun. Solmuilla voi maksimissaan olla kaksi lapsisolmua, lisäksi solmu tietää vanhempansa. Piirrä kaavio luokkien välisistä asiakassuhteista. Hahmottele myös esimerkki binääripuun mahdollisesta oliorakenteesta.

(b) Näytönsäästäjän virkaa toimittavassa virtuaaliakvaariossa asuu joukko erilaisia kalaparvia ja kasveja. Kaloja ja kasveja voidaan lisätä ja poistaa akvaariosta. Lisäksi jokainen parvi on valinnut itselleen suosikkikasvipöheikön, jonka alle se säntää piiloon uhkaavissa tilanteissa. Esitä akvaarion luokkarakenteen asiakassuhteet (graafisesta puolesta ei tarvitse välittää). Perustele valitsemaasi rakennetta käyttöesimerkein pohtimalla vaikkapa kalan lisäämistä ja kalaparven säikähtämistä.

4-8 Markkinointitutkimus on paljastanut, että iso joukko kuluttajista haluaisi toiminnoiltaan mahdollisimman yksinkertaisen matkapuhelimen. Puhelutoimintojen (soittaminen ja vastaaminen) lisäksi matkapuhelimessa tulisi olla vain puhelinluettelo. Sinulle on avautunut elämäsi tilaisuus suunnitella tämän upouuden, nimellä Simplextron markkinoitavan, puhelimen ohjelmisto. Tarkemmin sanoen sinulta halutaan hahmotelma tarvittavista luokista ja luokkarakenteesta (ts. luokkien välisistä asiakassuhteista). *Vinkki: Vaikka nyt on näytön paikka, älä kuitenkaan harhaudu ajattelemaan asiaa liian käyttöliittymälähtöisesti. Keskittymällä oleelliseen pidät ratkaisun muutaman luokan kokoisena.*

Osa II

Olio-orientoituneisuus

Luku 5

Periytyminen mekanismina

Periytymisen keskeisin merkitys ohjelmiston rakentamisen kannalta lienee se, että periytyminen antaa mahdollisuuden tarkastella oliota monesta eri abstraktin tason käyttäytymisnäkökulmasta (tietotyyppinäkökulmasta). Paitsi periytymismekanismia, tämän päämäärän saavuttamiseksi tarvitaan ajoaikana apuna myös polymorfismia ja dynaamista sidontaa. Toisaalta periytyminen auttaa organisoimaan luokkia hierarkioiksi ja siten helpottamaan niiden hallintaa ja toteutusta (yhteiset osat kirjoitetaan vain yhteen paikkaan). Hyvän luokkakokonaisuuden muodostaminen onkin implementoijalle haaste, koska se pitää aina suunnitella sekä asiakkaita että mahdollisia perijöitä varten.

Eri oliokielet poikkeavat toisistaan sen suhteen tukevatko ne yksittäis- vai moniperiytymistä. Java-suunnittelijat ovat valinneet strategian, joka on jotakin näiden kahden väliltä: pohjana on yksittäisperiytyminen, mutta luokka voi periä mielivaltaisen määrän rajapintoja. Näin luokasta luotu olio voi toimia monissa eri rooleissa. Syntaktisesti käytössä on siis käyttäytymisen moniperiytyminen, joten siihen liittyvät ongelmat, erityisesti samannimisten piirteiden yhdistämissäännöt täytyy tuntea tarkasti, jotta välttyisi ongelmilta. Periytymisessä on erityisen tärkeää säilyttää myös perittyjen piirteiden semantiikka (määrittelyt), vaikka piirteille annettaisiinkin perijässä uusi toteutus esimerkiksi tehokkuussyistä.

5.1 Alityypitys, periytyminen ja polymorfismi

Java samaistaa käsitteet tietotyyppi ja luokka: jokainen luokka määrittelee tyypin ja toisaalta jokainen uusi tyyppi toteutetaan kirjoittamalla sitä vastaava luokka. Nämä on käsitteinä kuitenkin syytä pitää erillään. *Tyyppi määrittelee käyttäytymissäännöt eikä ota mitään kantaa siihen, miten käyttäytyminen toteutetaan.*

Tässä mielessä rajapintaluokka edustaa tyyppiä melko puhtaasti edellyttäen, että rajapinnan piirteille on annettu täydelliset määrittelyt. Jos tyyppi `Tb` on tyypin `Ta` *alityyppi*, tyypin `Tb` tiedon tiedetään käyttäytyvän tyypin `Ta` mukaisesti, joskin alityypillä on joitain omiakin, vain sille ominaisia piirteitä. Tämä sääntö tunnetaan nimellä *korvaavuusperiaate* (*substitution principle*) [7]:

> *Korvaavuusperiaatteella toteutetaan tyyppiperheisiin liittyvä, määrittelyyn perustuva abstrahointi vaatimalla, että alityypit käyttäytyvät siten kuin niiden ylityyppien määrittelyt ilmaisevat.*

Tämän perusteella `Tb`-tyypin tieto on *tyyppiyhteensopiva* (*conformant*) `Ta`-tyypin kanssa. On helppo nähdä, että tämä suhde ei ole symmetrinen, ts. jos `Tb`-tieto on yhteensopiva `Ta`-tiedon kanssa, niin `Ta`-tieto ei ole tyyppiyhteensopiva `Tb`-tiedon kanssa muuta kuin siinä erikoistapauksessa, että tyypit ovat tarkalleen samat. Tyyppirelaatio on kuitenkin transitiivinen: jos `Tb` on tyypin `Ta` alityyppi ja `Tc` tyypin `Tb` alityyppi, `Tc` on myös tyypin `Ta` alityyppi. Tyyppiyhteensopivuuden määritelmän perusteella on siis täysin laillista kohdistaa `Ta`-tyypin operaatioita `Tb`-tyypin (ja `Tc`-tyypin) tietoon.

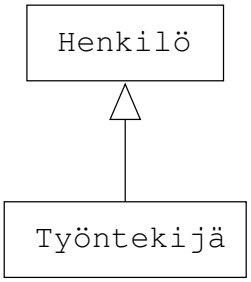
Javassa *luokka toteuttaa tyypin* ja tyyppien väliset relaatiot muodostetaan periytymisen avulla. Tämän nojalla myös tyyppiyhteensopivuus määräytyy luokkien välisen periytymishierarkian mukaisesti. Asetuslause `x = y` on oikein muodostettu, jos `x` on esitelty luokan `T` tunnisteeksi ja tunniste (lauseke) `y` luokan `T` jonkin jälkeläisen tunnisteeksi (mukaanluettuna luokka `T` itse).[1] Tunnisteen kyvystä viitata ohjelman suoritusaikana moneen erityyppiseen olioon käytetään nimitystä *polymorfismi* (monimuotoisuus).

Esimerkki 5.1 Oletetaan että `Työntekijä` on luokan `Henkilö` perijä (ks. kuva 5.1), tyypistä `Työntekijä` tulee tyypin `Henkilö` alityyppi. Täten ohjelmalohkon

```
Henkilö yksilö;
Työntekijä duunari;
duunari = new Työntekijä("B. Virtanen");
yksilö = duunari;
```

jälkimmäinen asetuslause on syntaktisesti oikein. Tämä on intuitiivisesti helppo ymmärtää, koska toimenpiteet, jotka tämän jälkeen voidaan kohdistaa tunnisteen `yksilö` viittaamaan olioon ovat ominaisia henkilöille ja työntekijä on myös henkilö — joskin hänestä tiedetään vähän enemmän (työpaikka, palkka,...) kuin pelkät henkilötiedot (nimi, osoite,...).

[1] Joissakin kielissä asetuslause on samanlainen kuin muutkin operaatiot ja siis =-merkin semantiikka on ohjelmoijan määriteltävissä. Java ei kuitenkaan — onneksemme — ole näin vapaamielinen.



Kuva 5.1: Luokkakaavio kahden luokan välisestä periytymisrelaatiosta.

Periytymisen lähtökohta on siis tyyppihierarkian toteuttaminen kielen tasolla. Tämän ajatuksen noudattaminen edellyttää, että toteutettavat luokat edustavat todellakin tietotyyppejä: luokan keskeisenä osana on (kapseloitu) tieto, johon on mahdollista kohdistaa luokassa esiteltyjä operaatioita. Mikäli tämä asia on kohdallaan ja periytymistä käytetään oikeaoppisesti tyyppiteoriaan[2] nojautuen, voidaan polymorfismin ja dynaamisen sidonnan tarjoamia etuja käyttää täysipainoisesti hyväksi. Erityisesti korvaavuusperiaatteen nojalla voidaan kirjoittaa koodia, joka käyttää yliluokkansa julkista liitäntää tuntematta todellisen olion esitystavasta muuta kuin, että se on käytetyn luokan jälkeläinen. Näin asiakas saattaa käsitellä perijäluokan oliota edes havaitsematta sitä. Toisella tavalla sanottuna: korvaavuusperiaatteen nojalla voidaan keskittyä todellisten tyyppien sijasta niiden yhteisiin ominaisuuksiin. Tämä on aivan keskeinen olio-ohjelmoinnin perusideologia. Sen seurauksena saadaan säännöt, joilla perittyä piirrettä voidaan muokata (ks. kohta 5.4). Myös ohjelmiston hallinta — ja siten ylläpito — helpottuu, sillä polymorfismi ja dynaaminen sidonta muodostavat voimakaksikon, jonka ansiosta ohjelmiston toteutusta voidaan yksinkertaistaa ja yleistää tehokkaasti.

5.2 Dynaaminen sidonta

Nyt on aika ottaa kantaa mitä tapahtuu, kun luokka `A` ja sen perijä `B` antavat eri toteutuksen rutiinille `f`, jota `A`-luokan asiakas kutsuu käyttäen muuttujaa `x`, johon on polymorfisesti liitetty perijäluokan olio. Jos valintapäätös tehdään käännösaikana, ainoa mahdollisuus on kutsua `A`-luokan `f`-rutiinia, koska kääntäjä ei voi millään tietää ennalta muuttujan `x` dynaamista tyyppiä ohjelman suorituksen kullakin hetkellä. Tällaisesta käännösaikaisesta sidonnasta käytetään nimitystä *staattinen sidonta* (*static binding*) ja se on tavallista vanhemmissa imperatiivisissa ohjelmointikielissä kuten C ja Pascal. Sidonnan yhteydessä tehdään staattinen tyypintarkistus eli kutsulle `y = x.f()` varmistetaan, että funktion `f` palauttama arvo on

[2]Eli (a) perijä on perittävän alityyppi ja (b) perijät noudattavat tiukasti perimiensä operaatioiden määrittelyjä.

tyyppiyhteensopiva muuttujan `y` kanssa. Tämän tarkistuksen ansiosta ajoaikaisen tukijärjestelmän ei tarvitse puuttua ohjelman suoritukseen mitenkään.

Oliokielet käyttävät lähes säännönmukaisesti *dynaamista sidontaa* (*dynamic binding*) eli ne tekevät päätöksen kutsuttavasta rutiinirungosta vasta ajoaikana muuttujan dynaamisen tyypin mukaan. Tällöin rutiini `f` tulisi valita aina muuttujaan `x` sillä hetkellä liitetyn olion tyypin perusteella. Menettely hidastaa tietysti ohjelman suoritusta, mutta antaa joustavuutta ja on edellytys todelliselle olio-ohjelmoinnille. Tietysti jos kääntäjä on fiksu, se voi käyttää staattista sidontaa havaitessaan, että mikään jälkeläinen ei ole korvannut piirrettä `f`, vaan tietty versio toimii kaikille jälkeläisluokille (esim. **`final`**-määreellä varustetut piirteet).

Java-kieli käyttää pääsääntöisesti dynaamista sidontaa eikä ohjelmoijalle ole annettu välineitä vaikuttaa sidonnan ajankohtaan.[3] Peruste tälle valinnalle on yksinkertainen: ei ole muuta järkevää vaihtoehtoa kuin dynaaminen sidonta (yhdistettynä staattiseen tyypintarkistukseen). Mikäli valinnan mahdollisuus annettaisiin, oletettaisiin ohjelmoijan tietävän kaikessa viisaudessaan jo etukäteen, miten luokan perijät tulevat piirrettä tarvitsemaan (ns. *clairvoyance principle*). Erikoista kyllä (hyvällä syyllä voisi sanoa jopa ”omituista kyllä”) dynaaminen sidonta toimii Javassa eri tavoin riippuen siitä, onko kysymyksessä jäsenmuuttuja vai rutiini.[4]

5.2.1 Staattinen ja dynaaminen tyyppi

Muuttujan *staattinen tyyppi* (*static type*) ilmaistaan muuttujan esittelyssä ja se pysyy nimensä mukaisesti muuttumattomana koko ohjelman suorituksen ajan. Muuttujan *dynaaminen tyyppi* (*dynamic type*) määräytyy sen mukaan, minkälaiseen olioon se kullakin hetkellä ohjelman suoritusaikana viittaa.

Esimerkki 5.2 Luontioperaation ja alustuksen

```
Henkilö yksilö = new Työntekijä("Lennart Nilkén");
Työntekijä duunari = new Työntekijä("B. Virtanen");
```

jälkeen muuttujan `yksilö` staattinen tyyppi on `Henkilö` ja dynaaminen tyyppi `Työntekijä`, kun taas muuttujan `duunari` sekä staattinen että dynaaminen tyyppi on `Työntekijä`.

Muuttujan *dynaaminen tyyppijoukko* tarkoittaa muuttujan staattisen tyypin alityyppien muodostamaa joukkoa (mukaanluettuna staattinen tyyppi itse). Dynaaminen tyyppijoukko on siis kaikkien niiden tyyppien muodostama joukko, jotka ovat tyyppiyhteensopivia muuttujan staattisen tyypin kanssa.

Esimerkki 5.3 Tyyppisäännöt kieltävät asetuslauseen

[3] Esimerkiksi C++-kielessä ohjelmoija voi tehdä valinnan staattisen ja dynaamisen sidonnan välillä, vaikka oletuksena onkin staattinen sidonta.

[4] Tässäkin yhteydessä korostuu se, että Javassa on haluttu tehdä selkeä ero jäsenmuuttujan ja parametrittoman funktion välille.

```
duunari = yksilö;
```

koska `Henkilö` ei ole yhteensopiva tyypin `Työntekijä` kanssa, koska siltä puuttuu työntekijän erityisominaisuudet. Samoin asetuslauseyhdistelmä

```
yksilö = duunari;
ansio = yksilö.annaPalkka();
```

on väärin, koska `Henkilö`-luokka ei tunne työntekijälle ominaisia piirteitä. Näin on siis siitäkin huolimatta, että kummassakin tapauksessa tiedetään varmasti, että `yksilö` viittaa työntekijäolioon.

Polymorfismi sallii olion tarkastelun, paitsi sen todellisen tyypin, myös kaikkien sen ylityyppien kautta.[5] Täten ohjelmoija tarvitsee työkalut tyyppimuunnoksien tekoon ajoaikana. Esimerkissä 5.1 oleva polymorfinen asetus `yksilö = duunari` tekee tyyppimuunnoksen automaattisesti. Kysymyksessä on luodun työntekijäolion *yleistulkinta* (*upcasting*). Nimitys juontaa juurensa siitä, että tyyppimuunnos tapahtuu luokkahierarkiassa ”ylöspäin” eli yleisemmän käsitteen suuntaan. Asetuksen jälkeen `Työntekijä`-oliota käsitellään muuttujan `yksilö` kautta `Henkilö`-roolissa. Jos olion tulkintaa on yleistetty syystä tai toisesta, on joskus tarpeellista päästä käsittelemään sitä myös todellisen tyyppinsä mukaan. Esimerkiksi edellisen asetuslauseen jälkeen voitaisiin jossakin vaiheessa tehdä *erikoistulkinta* (*downcasting*) takaisin todelliseen tyyppiin `Työntekijä` käyttämällä tyyppipakotusta:

```
duunari = (Työntekijä)yksilö;
```

minkä jälkeen oliota voidaan käsitellä taas työntekijänä. Mikäli asetuslause on oikein muodostettu, yleistyksiä ja erikoistuksia voidaan tehdä mihin kohtaan tahansa sillä luokkahierarkiapolulla, joka johtaa asetuslauseen vasemmalla puolella olevan muuttujan staattisesta tyypistä muuttujan dynaamiseen tyyppiin. Jos tulkinnassa mennään tämän alueen ulkopuolelle, seurauksena on `ClassCastException`-poikkeuksen nouseminen. Tyyppipakotusta käyttävät yleistulkinnat ovat erittäin harvinaisia; ainoa merkittävä tilanne, jossa niitä tarvitaan tulee eteen silloin, kun ylikuormitetun rutiinin kutsussa halutaan yksi tai useampia argumentteja pakottaa tietyntyyppisiksi, jotta oikea rutiiniversio tulisi valituksi.

5.2.2 Jäsenmuuttujien sidonta

Vaikka jäsenmuuttuja voidaan samaistaa parametrittoman funktion kanssa,[6] Java käsittelee niiden sidonnan rutiineista poikkeavalla tavalla: *jäsenmuuttujat sidotaan aina staattisesti.* Sen lisäksi että tämä poikkeaa yleissäännöstä, hämmennystä lisää myös se, että perijäluokka voi esitellä samannimisen jäsenmuuttujan kuin minkä

[5] Tätä voidaan tehdä vieläpä samanaikaisesti eli samaan olioon voi ajoaikana olla useita erityyppisiä viittauksia.

[6] Esimerkiksi Eiffel-kieli tekee näin.

se on jo perinyt yliluokaltaan. Uuden samannimisen jäsenmuuttujan esittely tekee perityn jäsenmuuttujan näkyvyysalueeseen (*scope*) ”aukon”, sillä uusi esittely peittää yleisemmissä luokissa annetut samannimiset jäsenmuuttujat.

Esimerkki 5.4 Tarkastellaanpa tilannetta, jossa luokka `A` määrittelee

```
public class A
  {
  public int luku = 3;
  }
```

ja siitä periytyvä luokka `B` määrittelee

```
public class B extends A
  {
  public double luku = 5.0;
  }
```

Jos nyt tehdään asetukset

```
A a = new B();
info = a.luku;
```

niin pitääkö muuttujan `info` olla tyyppiä **`double`** vai **`int`**? Koska sidonta on staattista, muuttujan `info` pitää olla tyyppiä **`int`** ja se saa arvokseen 3. Luokassa `B` määriteltyyn muuttujaan `luku` ei voida viitata *polymorfisesti* luokan `B` ulkopuolelta. Muuten tietysti voidaan; siis esimerkiksi

```
B b = new B();
info2 = b.luku;
```

viittaa luokassa `B` esiteltyyn **`double`**-arvoon. Samannimisen jäsenmuuttujan esittely perijässä siis peittää perityn jäsenmuuttujan näkyvyyden, joten luokan `B` sisällä sijaitsevat rutiinit viittaavat aina **`double`**-tyyppiseen muuttujaan. Mikäli halutaan viitata perittyyn **`int`**-tyyppiseen jäsenmuuttujaan, se on mahdollista käyttämällä merkintää **`super.`**`luku`.

Huomaa, että `luku` oli esitelty **`public`**-tyyppiseksi kummassakin luokassa. Se, että ko. suojausta on käytetty luokan `A` piirteen yhteydessä on tietysti perusedellytys asiakaskoodissa olevalle polymorfiselle viittaukselle: piirteen pitää löytyä luokan julkisesta liitännästä.

5.2.3 Rutiinien sidonta

Rajataan tarkastelu heti aluksi niihin rutiineihin, jotka eivät ole **`static`**-määreellä varustettuja luokkametodeja, sillä niillä ei ole kutsukohdeoliota ja siksi ne sidotaan — nimensä mukaisesti — aina staattisesti (vrt. jäsenmuuttujan sidonta, kohta 5.2.2). Myös **`final`**-määreillä varustetut rutiinit voidaan sitoa staattisesti, koska niille ei voi tehdä korvausta perijäluokassa. Hieman erikoisempaa on se, että

private-rutiinit sidotaan myös staattisesti. Koska *package*-piirre on toisessa pakkauksessa olevan luokan kannalta **private** — olipa jälkimmäinen sitten piirteen sisältävän luokan perijä tai ei — myös nämä sidonnat tehdään staattisesti. Kaikesta päätellen siis Java on melko rajoittunut dynaamisen sidonnan käytössä. Hieman yleistäen voidaan sanoa, että se toimii vain **protected**- ja **public**-rutiineille.

- Kääntäjä tarkistaa ensin, että kutsuttava metodi kuuluu luokan julkiseen liitäntään. Tällä varmistetaan se, että kutsu on staattisesti oikein. Ei ole nimittäin mitään mieltä kutsua rutiinia polymorfisesti, jos rutiini ei kuulu muuttujan staattisen tyypin ilmaisemaan luokkaan.

- Jos edellinen tarkistus (tehdään käännösaikana) on läpäisty, haetaan ajoaikana operaatiolle kaikkein erikoistunein versio eli se, joka parhaiten soveltuu muuttujan dynaamisen tyypin oliolle.

Tarkemmin sanottuna säännön jälkimmäinen osa on siis seuraava: *sidontaprosessi lähtee liikkeelle muuttujan dynaamisen tyypin ilmaisemasta luokasta* `B` *ja etenee luokkahierarkiassa ylöspäin, kunnes päätyy muuttujan staattisen tyypin mukaiseen luokkaan* `A`*. Suoritettavaksi operaatioksi valitaan ensimmäinen tällä polulla kohdattu toteutus.* Tämän menettelyn tarkoituksena on varmistaa, että valittu toteutus on *luokassa* `A` *annetun määrittelyn mukainen piirre, jota käytetään dynaamiselle tyypille ominaisella tavalla.* Yksityiskohtana mainittakoon, että jos polulta ei löydy piirteelle ainuttakaan (uutta) toteutusta, valitaan suoritettavaksi luokan `A` tuntema versio (jonka se on perinyt).

Perusedellytys dynaamiselle sidonnalle on, että staattisen tyypin mukaisessa luokassa ja dynaamisen tyypin mukaisessa luokassa on saman signatuurin omaava julkiseen liitäntään kuuluva piirre. Jos rutiini on **public** kummassakin ja suojausmääre voi muuttua vain sallivampaan suuntaan, tarkoittaa se sitä, että määre pysyy samana koko periytymispolulla. Tältä osin asia on siis varsin suoraviivainen.

Esimerkki 5.5 Kannattaa varoa tilanteita, joissa julkinen piirre kutsuu toista, johon on liitetty rajoittava suojaus:

```
class A
  {
  public void f() { g(); }
  private void g() { System.out.println("Yksityinen"); }
  }

class B extends A
  {
  public void g() { System.out.println("Julkinen"); }
  }
```

Luokille olettaisi dynaamisen sidonnan käyttävän olion todelliselle tyypille soveltuvaa piirrettä eli että rutiinin `f` polymorfinen kutsu tuottaisi tuloksen `"Julkinen"`. Todellisuudessa tulostuu `"Yksityinen"`, koska **`private`**-rutiini `g` on sidottu staattisesti. Tämä on mielekästä sikäli, että tilanne on kovin riskialtis: jos ohjelmoija perii joltain luokalta ja tekee sitten perijäluokkaan sattumalta samannimisen piirteen, joka on jossain yliluokassa (mahdollisesti hyvinkin korkealla hierarkiatasolla) esitelty **`private`**-tyyppiseksi ja jos dynaaminen sidonta toimisi tässä normaalisti, rutiinin `f` toiminta saattaisi muuttua hyvinkin radikaalisti. Muuttamalla rutiinin suojaus luokassa `A` **`protected`**-tyyppiseksi dynaaminen sidonta saadaan toimimaan taas normaalisti.

Muistin virkistämiseksi kerrataan vielä kaikki *staattisesti* sidottavat piirteet:

- kaikki staattiset piirteet
- kaikki jäsenmuuttujat
- **`final`**-rutiinit
- **`private`**-rutiinit
- *package*-rutiinit

Kaikki muu sidotaan siis dynaamisesti.

5.3 Määrittelyn erottaminen toteutuksesta

Ohjelmistosysteemin rakentaminen aloitetaan hahmottelemalla tarvittavat luokat ja niiden väliset yhteydet. *Rajapintaluokat* (*interfaces*) ja *abstraktit luokat* (*abstract classes*) ovat tärkeitä tässä toiminnassa, koska niiden avulla voidaan kuvata koko ohjelmiston rakenne (luokkien väliset relaatiot) ja semantiikka (määrittelyt). Luokkaa (tai jopa yksittäistä rutiinia) voidaan abstrahoida myös geneerisyyden avulla, mitä tarkastellaan lähemmin luvussa 7.

Luokkien välisistä asiakas- ja periytymisrelaatioista muodostettua korkean tason kuvausta voidaan käyttää myös dokumentointitarkoituksiin. Kun ohjelmiston kokonaisrakenne on hahmoteltu, suunnitelman oikeellisuus tulee pyrkiä varmistamaan eri tavoin, mm. käyttötapausten avulla. Näin toimimalla voidaan viivyttää myöhäisempään ajankohtaan niitä päätöksiä, jotka koskevat konkreetteja esitystapoja ja käytettäviä algoritmeja. Tällöin koko sovellus on tullut jo tutummaksi, mikä puolestaan lisää sitä todennäköisyyttä, että valitut kooditason ratkaisut ovat oikeaan osuneita.

Rajapinnat ja abstraktit luokat auttavat jäsentämään ohjelmistolla mallinnettavaa maailmaa. Kun ohjelmoija on sisäistänyt abstraktien käsitteiden merkityksen, hänellä on korkean tason työkalut, joilla käsitellä todellisia tietoja. Tämä näkyy tunnisteiden esittelystä: staattiseksi tyypiksi voidaan valita rajapinta tai

abstrakti luokka, joka määrittelee olion käyttäytymisen ottamatta kantaa sen todelliseen tyyppiin. Näin ohjelmoijan työ tulee helpommaksi (tuottavuus kasvaa), mutta samalla myös ohjelmiston sisäinen laatu kasvaa, koska se yksinkertaistuu ja ylläpidettävyys helpottuu (tehtävät muutokset ovat paikallistettavissa ja ne myös pysyvät hyvin lokaaleina eivätkä aiheuta muutostarpeita ohjelmiston muissa osissa).

5.3.1 Rajapintaluokka

Periytymisellä pyritään menettelyyn, jossa eri tyyppien yhteiset ominaisuudet kerätään yhteen (ja vain yhteen) paikkaan. Tämä rohkaisee kehittämään ohjelmistoa niin, että koodiosia uudelleenkäytetään ja muutetaan kontrolloidusti. Tämän seurauksena koodin ja käsitteiden uudelleenkäyttö on mahdollista laajassa mittakaavassa. Kun samantyyppisten luokkien yhteistä käyttäytymistä abstrahoidaan, kyseinen luokka sijoittuu periytymishierarkiassa helposti niin korkealle tasolle, ettei edes luokan konkreettiin esitysmuotoon haluta ottaa kantaa. Tästä johtuen kullekin operaatiolle voidaan kirjoittaa vain sen määrittely (signatuuri, alku- ja loppuehdot). Listauksen 2.1 (s. 27) pinoa kuvaava rajapinta `Pino` on hyvä esimerkki tästä.

Rajapintaluokan päätasolle ei voi siis kirjoittaa ainoatakaan rutiinitoteutusta. Rajapinnan syntaksi on tuttu: varatun sanan **`class`** tilalla on nyt **`interface`**. On myös erittäin tyypillistä, että rajapinta on näkyvyysmääreeltään **`public`** (*package*-määreisiä rajapintoja tarvitaan harvemmin). Rajapinnan kaikkien piirteiden tulee olla joko rutiinien määrittelyitä tai vakioita. Tästä syystä `Pino`-rajapinta ei voi sisältää esimerkiksi pinon kokoa ylläpitävää jäsenmuuttujaa. Rajapinnan piirteet ovat oletuksena aina **`public`** ja **`abstract`** — jopa siinäkin tapauksessa, että nämä avainsanat eivät esiinny rutiinin signatuurissa. Suojausmääre **`public`** takaa sen, että rutiinit säilyvät aina perijöiden julkisessa liitännässä. Tämä on luonnollinen vaatimus; koska rajapinta ei toteuta mitään rutiinia eikä voi ottaa kantaa konkreettiin esitysmuotoon, **`protected`**- tai **`private`**-rutiineja ei ole mielekästä esittää. Rajapinnassa esitellyt muuttujat ovat **`public`**-ominaisuuden lisäksi myös **`static`** ja **`final`** eli ne ovat vakioita. Nämä määreet voi jättää pois jos haluaa, mutta on suositeltavaa kirjoittaa ne aina näkyviin, jotta myöhemmin (kun nämä oletukset on jo unohtanut) ei tulisi tulkintaongelmia.

Rajapinnassa esiteltyjen *rutiinimäärittelyjen on tarkoitus säädellä tarkasti sitä, miten kyseiset piirteet käyttäytyvät jälkeläisissä.* Mikäli tästä säännöstä pidetään kiinni, *rajapinta toimii roolina, jossa konkreetin perijäluokan oliota voi käsitellä.* Java-kulttuuriin kuuluu kiinteästi myös menettely, missä alkuperäisestä alityypitysajatuksesta poiketaan muodostamalla hyvin pelkistettyjä rajapintoja, joissa on

vain yksi (esim. `Comparable`) tai ei yhtään (esim. `Cloneable`[7] ja `Serializable`[8]) piirrettä! Tällaiset rajapinnat eivät edusta mitään tietotyyppiä. Tarkoituksena onkin käyttää periytymismekanismia siihen, että (a) rajapinnan toteuttavien luokkien *taataan* osaavan rajapinnassa määritellyn operaation (esim. `compareTo` tai `clone`) tai (b) luokan kirjoittajalta halutaan *lisävarmistus* (luokan otsikossa) siitä, että ko. operaatioita halutaan todellakin käytettävän (esim. `Cloneable` tai `Serializable`) kyseisen luokan olioille.

Rajapinta voi normaalisti periä yhdeltä tai useammalta muulta rajapintaluokalta. Tällöin **`extends`**-avainsanan perään kirjoitetaan (pilkulla erotettuina) perityt luokat.

Esimerkki 5.6 Oletetaan että meillä on seuraavat rajapintamäärittelyt:

```
public interface OsaaKirjoittaa
  {
  public void kirjoita(Asiakirja a);
  }

public interface OsaaLukea
  {
  public void lue(Asiakirja a);
  }
```

Nyt nämä kaksi rajapintaa voidaan periä uudeksi rajapinnaksi

```
public interface Poliisi extends OsaaLukea, OsaaKirjoittaa
  {
  public void sakota(Henkilö h, double summa);
  public void pidätä(Henkilö h);
  }
```

Useita rajapintoja perittäessä saattaa syntyä tilanne, jossa perijä saa samannimisen rutiinin tai jäsenmuuttujan kahdelta tai useammalta eri rajapinnalta. Tällaisten nimikonfliktien ratkaisemisesta keskustellaan kohdassa 5.5.2.

5.3.2 Abstrakti luokka

Siirryttäessä rajapintaluokan perijöihin yleensä ainakin osa sen piirteistä voidaan toteuttaa, koska mallinnettavan käsitteen ominaisuuksista tiedetään enemmän.

[7] Oikeastaan rajapinnan nimi pitäisi olla `Clonable` mutta tätä Javan ensimmäiseen versioon pesiytynyttä kielioppivirhettä on ollut mahdoton kitkeä jälkikäteen. Tarinan opetus: Mieti mitä julkistat, sillä jo julkistettuja piirteitä on enää mahdoton poistaa tai perua.

[8] `Serializable`-rajapinnan toteuttavat oliot osaavat muuttaa sisältämänsä tiedot sarjamuotoon ja takaisin. Tätä ominaisuutta tarvitaan esimerkiksi silloin, kun olio talletetaan levylle tai siirretään linjaa pitkin paikasta toiseen.

Luokka, jossa on vähintään yksi piirre toteutettu ja vähintään yksi toteuttamatta, sanotaan Javassa *abstraktiksi luokaksi* (tämä ei ole ehdoton sääntö kuten kohta tullaan huomaamaan). Luokkaa, jonka kaikki piirteet on toteutettu, sanotaan puolestaan *konkreetiksi*. Vastaavasti toteuttamatonta piirrettä sanotaan abstraktiksi piirteeksi ja toteutettua konkreetiksi. Abstrakti luokka kuvaa tyypillisesti käsitettä, joka on saatu yleisemmästä abstraktista luokasta tai rajapinnasta kiinnittämällä jokin erikoisominaisuus. Tässä mielessä abstrakti luokka on usein "välivaihe" siirryttäessä rajapinnasta konkreettiin luokkaan.

Abstrakti luokka tunnistetaan siitä, että sen otsikkoon on kirjoitettu lisämääre **`abstract`**. Siinä missä rajapinta on aina **`public`**, abstraktin luokan näkyvyys voidaan rajata haluttaessa myös pakkauksen sisälle (eli *package*-näkyvyys). Abstrakti luokka voi periä toiselta abstraktilta tai konkreetilta luokalta (mainittu otsikossa **`extends`**-avainsanan jälkeen) ja samanaikaisesti usealta rajapinnalta (mainittu **`implements`**-avainsanan jälkeen). Tämä voidaan nähdä moniperiytymisen rajoittuneena muotona, joskin realistisempi moniperiytyminen saadaan aikaan sisäluokkia käyttäen (ks. kohta 9.3.2).

Koska rajapinnan kaikki piirteet ovat aina implisiittisesti abstrakteja ja konkreetin luokan kaikki piirteet ovat konkreetteja, abstraktissa luokassa voi olla — ja tyypillisesti onkin — sekaisin molempia. Tämän takia abstraktissa luokassa on kaikki abstraktit piirteet varustettava määreellä **`abstract`** ja konkreeteille annettava toteutus. Koska abstraktille piirteelle annetaan konkreetti toteututus vasta jossain perivässä luokassa, sitä ei saa varustaa määreillä **`private`**, **`static`** tai **`final`**.

Esimerkki 5.7 Listauksessa 5.1 on esitelty abstrakti luokka `RajoitettuPino`, joka toteuttaa osittain listauksessa 2.1 määritellyn rajapinnan `Pino`. Rajoitetulle pinolle on asetettu maksimikoko, jota suuremmaksi pino ei voi kasvaa. Tämän tiedon nojalla voidaan toteuttaa joitakin pinon kokoon liittyviä operaatioita. Koska konkreettia esitystapaa ei kuitenkaan haluta kiinnittää vielä tällä abstraktiotasolla, osa piirteistä jää abstrakteiksi. Niinpä `Pino`-rajapinnasta periytyvät rutiinit `päällimäinen`, `alimmaiset`, `lisää` ja `poista` ovat yhä abstrakteja.

Rajapinnassa `Pino` esiteltyjen ja määriteltyjen rutiinien alku- ja loppuehtoja ei ole toistettu tässä luokassa. Kyseessä ei ole sattuma tai välinpitämättömyys, vaan pikemminkin päinvastoin. Periaatteena on tottakai se, että rutiinin toteutus on perijässä sen mukainen, mitä alkuperäinen määrittely sanoo. Jo kohdassa 2.2.2 kuitenkin todettiin, että määrittelyn mukaisen sopimuksen voi korvata alisopimuksella ilman, että siitä olisi haittaa asiakkaille. Sama sääntö pätee periytymisessäkin. Tämän takia *perijäluokkiin kirjoitetaan vain ne lisäehdot*, jotka täydentävät aiemmin esiteltyjä alku- ja loppuehtoja. Määrittelyn kaikki osat saa selville keräämällä ne koko perintäpolulta yhteen.

Abstraktin luokan rutiini voidaan konkretisoida abstraktien piirteiden avulla. Niinpä abstraktiin luokkaan `RajoitettuPino` voitaisiin lisätä rutiini, joka vaihtaa pinon päällimmäisen alkion toiseksi:

```
public void vaihdaPäällimmäinen(T alkio)
```

Listaus 5.1: Abstrakti luokka `RajoitettuPino`.

```
public abstract class RajoitettuPino<T> implements Pino<T>
  {
  private int koko;
  private final int KAPASITEETTI;

  /**
   *  @.pre  maksimikoko >= 0
   *  @.post annaKoko() == 0 &
   *         annaKapasiteetti() == maksimikoko
   */
  protected RajoitettuPino(int maksimikoko)
    {
    KAPASITEETTI = maksimikoko;
    koko = 0;
    }

  //-- Havainnointioperaatiot

  public int annaKoko() { return koko; }

  public int annaKapasiteetti() { return KAPASITEETTI; }

  public boolean onTyhjä() { return (annaKoko() == 0); }

  public boolean onTäysi() { return (annaKoko() == KAPASITEETTI); }

  //-- Muutosoperaatiot

  public void tyhjennä() { koko = 0; }

  /**
   *  @.pre  -annaKoko() <= muutos <=
   *         (annaKapasiteetti() - annaKoko())
   *  @.post RESULT.annaKoko() == this.annaKoko() + muutos
   */
  protected void muutaKoko(int muutos) { koko += muutos; }
  }
```

```
{
poista();
lisää(alkio);
}
```

Rutiini tulee perijässä lopullisesti konkreetiksi, kun `poista` ja `lisää` saavat toteutuksensa. Valittu konkreetti esitystapa saattaa sallia tehokkaammankin toteutuksen, mutta rutiinin toteutuksen uusiminen jää perijäluokan laatijan harkinnan varaan.

Konkreetin esitystavan puuttuminen ja joidenkin rutiinien toteutuksen puuttuminen aiheuttavat sen, että abstraktista luokasta ei ole sallittua luoda olioita. Abstrakti luokka voi kuitenkin esitellä konstruktorin, jota sen perijät voivat käyttää apunaan (ks. kohta 6.1). Sama pätee myös `Object`-luokalta periytyviin perusoperaatioihin kuten `equals` ja `clone`.

5.4 Rutiinin korvaus ja ylikuormitus

Perivä luokka voi muuttaa yliluokassa annettua rutiinimäärittelyä ja sen toteutusta. *Korvauksessa* (*overriding*) perijä antaa rutiinille uuden toteutuksen. *Ylikuormituksessa* (*overloading*) annetaan samalle rutiininimelle rinnakkainen toteutus. Onkin siis paikallaan antaa yksityiskohtaiset säännöt, joiden mukaan korvaus ja ylikuormitus erotetaan toisistaan.

5.4.1 Kovarianssi, kontravarianssi ja novarianssi

Oliokielien rutiineihin liittyvä *kovarianssisääntö* (*covariance rule*) sanoo, että uudelleenmääriteltäessä ja -toteutettaessa yliluokan rutiinin argumenttityypin, tulostyypin ja/tai poikkeustyypin saa muuttaa alityypikseen. Javassa tämä sääntö koskee vain rutiinin palautustyyppiä.[9] Tämän lisäksi tarkistettavien poikkeusten lista on kovariantti yliluokan vastaavan poikkeuslistan kanssa.

Päinvastainen sääntö, missä perijäluokan rutiinin argumenttityypin, tulostyypin ja/tai poikkeustyypin voi vaihtaa ylityypikseen, on nimeltään *kontravarianssisääntö* (*contravariance rule*). Javassa kontravarianssia esiintyy ainoastaan tyyppiparametrin **super**-rajauksessa (ks. kohta 7.1.1).

Kolmas vaihtoehto, jossa tyypin vaihtamista ei sallita suuntaan tai toiseen, on nimeltään *novarianssi* (*novariance*). Javassa novarianssia sovelletaan rutiinin argumenttityyppeihin, jolloin poikkeavat argumenttityypit tulkitaan ylikuormitukseksi.

[9] Palautustyypin kovarianssi on tullut mukaan Javan versiossa 5.0. Kielen aiemmissa versiossa palautustyyppiin sovellettiin novarianssisääntöä.

Kovarianssi- ja kontravarianssisäännöillä on sekä etunsa että haittansa. Ei ole selkeää yksimielisyyttä siitä, kumpi sääntö on käyttökelpoisempi ohjelmoinnin kannalta. Korvaavuusperiaatteen perusteella ainoat tyyppiturvalliset käyttötavat ovat paluutyypin kovarianssi ja argumenttityypin kontravarianssi. Ilman lisärajoituksia kaikki muut tulkinnat tuovat mukanaan mahdollisuuden ristiriitaisiin käyttötilanteisiin.

5.4.2 Korvaus

Jos rutiinin nimi, argumenttien järjestys ja argumenttien tyypit sekä palautustyyppi ovat samat kuin yliluokan rutiinilla, kyseessä on korvaus. Kyseessä on korvaus myös silloin, jos rutiinin nimi, argumenttien järjestys ja argumenttien tyypit ovat samat sekä palautustyyppi on yliluokan rutiinin palautustyypin alityyppi (ns. palautustyypin kovarianssi).

Esimerkki 5.8 Jos luokassa `A` on määritelty rutiini

```
public Object f() { /* ... */ }
```

ja perijässä `B` esitellään rutiini

```
public String f() { /* ... */ }
```

jälkimmäinen rutiini korvaa aikaisemman, sillä sen paluutyyppi `String` on yliluokan rutiinin palautustyypin `Object` alityyppi.

Jos signatuurit ovat samat eikä palautustyyppi ole kovarianttinen, kääntäjä antaa virheilmoituksen.

Esimerkki 5.9 Jos luokassa `A` tunnetaan rutiini

```
public int f() { /* ... */ }
```

ja perijässä `B` esitellään uusi rutiini

```
public double f() { /* ... */ }
```

kääntäjälle tulisi ongelmia tulkitessaan perijäluokassa olevaa asetuslausetta

```
double tulos = f();
```

Sama ongelma tulee tietysti vastaan myös silloin, jos (a) perijäluokan `B` asiakas kutsuu rutiinia `f()` samanlaisessa yhteydessä tai (b) rutiinia `f` kutsutaan proseduurimuotoisesti (tallettamatta palautusarvoa).[10]

[10] On kieltämättä aika outoa, että Java-kääntäjä tarkistaa ja vaatii välttämättä jokaiseen funktioon **`return`**-lauseen, mutta sallii varsin kepeästi sen, että funktiosta palautettua arvoa ei oteta vastaan!

Vaikka suojausmääre ei varsinaisesti kuulu rutiinin signatuuriin, se on huomioitava siirryttäessä perijään. Perivän luokan korvaavan piirteen suojausmääre voidaan muuttaa vain väljemmäksi eli perijän piirteelle voidaan antaa laajempi näkyvyys kuin mitä yliluokan vastaavalla rutiinilla on.

Esimerkki 5.10 Jos luokassa `A` on määritelty rutiini

```
protected void f() { /* ... */ }
```

se voidaan korvata perijässä `B` rutiinilla

```
public void f() { /* ... */ }
```

sillä tällöin suojausmääre väljenee. Sen sijaan `B` ei voi tiukentaa rutiinia **`private`**-suojatuksi.

Korvauksessa yliluokalta peritty rutiini siis korvataan perijän omalla rutiinilla. Entäpä jos perijässä halutaan kutsua nimenomaan perittyä yliluokan rutiinia eikä sen korvausta? Ongelman ratkaisuksi Java tarjoaa varatun sanan **`super`**. Yleisesti merkintä **`super.f()`** merkitsee yliluokan rutiinin `f` kutsua. Merkintää ei voi kuitenkaan ulottaa välitöntä yliluokkaa pidemmälle (ts. muoto **`super.super.f()`** ei ole sallittu).

Jos rutiinien signatuurit ovat samat, mutta poikkeusten lista on erilainen, tilanne on virheellinen, mikäli perijäluokan rutiinin poikkeukset eivät ole tyyppiyhteensopivia yliluokan rutiinin poikkeusten kanssa.

Esimerkki 5.11 Luokassa `A` esitelty

```
public void f() throws ClassCastException { /* ... */ }
```

nostaa tarkistamattoman poikkeuksen ja luokassa `B` esitelty

```
public void f() throws CloneNotSupportedException { /* ... */ }
```

tarkistettavan poikkeuksen. Koska jälkimmäinen ei ole tyyppiyhteensopiva edellisen kanssa, kääntäjä antaa virheilmoituksen. Oikeanmuotoisia perijätotetuksia ovat sen sijaan:

```
public void f() throws ClassCastException  // Ei muutoksia
public void f() // Perijätotetus osaa hoitaa kaikki poikkeustilanteet
public void f() throws MyVeryOwnClassCastException
                // Nostettava poikkeus on itse tehty, luokan
                // ClassCastException perijä
```

Esitetty sääntö on helppo mieltää, kun asiaa ajattelee luokan `A` asiakkaan kannalta. Asiakas on valmistautunut hoitamaan ne poikkeukset, jotka luokan `A` rutiini `f` on signatuurissaan ilmoittanut. Jos polymorfismin ja dynaamisen sidonnan ansiosta käykin niin, että todellisuudessa suoritetaan luokan `B` vastaava rutiini, samainen asiakas ei koe tulleensa petetyksi, koska asiakkaan koodiin kirjoitetut poikkeustenkäsittelijät pystyvät edelleen hoitamaan ongelmatilanteet.

Poikkeuksissa noudatettava sääntö on yksinkertainen: siirryttäessä perijään myös poikkeuslistoissa saa siirtyä perijöihin. Periaate tunnetaan oliokielissä nimellä kovarianssisääntö, joskin se yleensä liitetään korvaukseen ja rutiinin argumentteihin (ks. kohta 5.4.1). Vaikka yllä annetut esimerkit koskivatkin vain yhtä signatuurissa esiteltyä poikkeusnimeä, sama sääntö yleistyy myös poikkeuslistalle. Sääntöön on yksi erikoistapaus: korvaava rutiini voi nostaa *tarkistamattoman poikkeuksen*, jota ei ole ilmoitettu yliluokan rutiinin signatuurissa; kuten muidenkin poikkeusten kohdalla, tämä ei kuitenkaan aiheuta ylikuormitusta.

5.4.3 Ylikuormitus

Jos signatuurit poikkeavat argumenttien osalta, kyseessä on ylikuormitus ja yliluokan rutiini tunnetaan sellaisenaan myös perijässä.

Esimerkki 5.12 Ylikuormitustilanne saattaa syntyä vahingossa silloin, kun aikomuksena on ollut tehdä korvaus. Korvattaessa `Object`-luokan `equals`-operaatiota, argumentiksi on helppo kirjoittaa "väärä" tyyppi. Luokassa `A` olisi luonnollista esitellä

```
public boolean equals(A toinen)
```

Koska `Object`-luokka tuntee vastaavan operaation signatuurilla

```
public boolean equals(Object toinen)
```

olisi tuloksena ylikuormitus, ei korvaus.

Esimerkki 5.13 Ylikuormitetut operaatiot voivat erota kutsujan kannalta toisistaan vain hyvin vähän toisistaan. Ei ole aivan selkeää kumman rutiineista

```
public void f(double arvo) { /* ... */ }
public void f(float arvo) { /* ... */ }
```

kutsu

```
f(100.0);
```

valitsee. Vastaus: argumentti tulkitaan **`double`**-tyyppiseksi, jonka perusteella valinta on yksikäsitteinen.

Esimerkki 5.14 Monimielinen tilanne, missä todellisen argumentin tyyppi on yhteensopiva usean eri muodollisen argumentin kanssa, saadaan tyypillisesti aikaan periytymisen yhteydessä. Oletetaan, että luokka `A` esittelee rutiinin

```
public void f(A tieto) { /* ... */ }
```

ja perijä `B` vastaavasti

```
public void f(B tieto) { /* ... */ }
```

Jos `olioB` on tyyppiä `B`, on kutsu `f(olioB)` laillinen kummallekin. Valinta tapahtuu seuraavan säännön mukaan: *jos kutsun argumentit ovat yhteensopivat kahden tai useamman rutiinin muodollisten argumenttien kanssa, valitaan suoritettavaksi rutiiniksi se, jonka muodollisiin argumentteihin todelliset argumentit tarkimmin sopivat.* "Tarkimmin" tarkoittaa tässä erikoistuneimpia tyyppejä, jotka ovat yhteensopivia. Tässä tapauksessa valinta kohdistuisi luokassa `B` esiteltyyn operaatioon. Jos tämä on asiakkaan kannalta väärä valinta, luokan `A` operaatio saadaan käyttöön yleistulkinnan avulla kirjoittamalla kutsu muotoon `f((A)olioB)`.

5.5 Luokkahierarkia

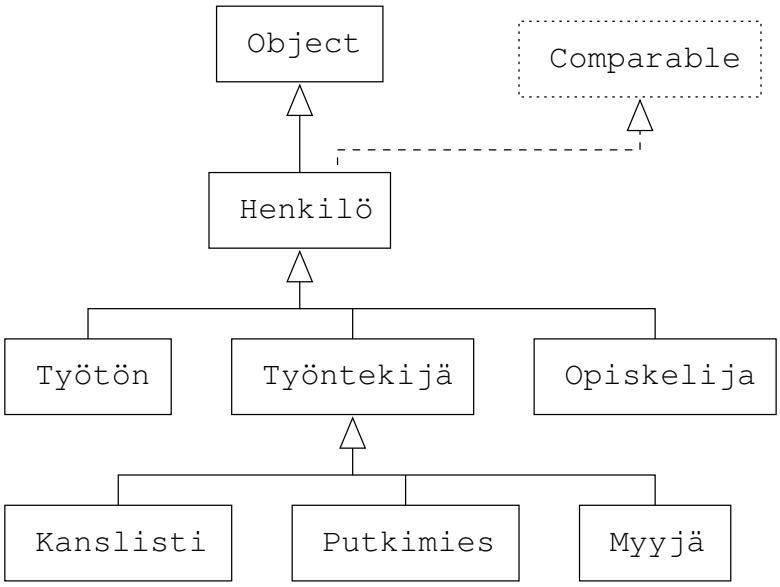
Javan luokkahierarkian tärkein luokka on `Object`, joka on kaikkein yleisin konkreetti luokka. Kaikki konkreetit ja abstraktit luokat perivät `Object`-luokan joko suoraan tai "välikäden" kautta (`Object`-piirteiden yhdenmukaista toteuttamista tarkastellaan tarkemmin luvussa 6). Koska konkreetti (abstrakti) luokka voi periä ainoastaan yhdeltä konkreetilta tai abstraktilta (vastaavasti abstraktilta) luokalta, polku annetusta luokasta `Object`-luokkaan on yksikäsitteinen. Sen sijaan rajapinnat eivät koskaan peri `Object`-luokalta, vaan muodostavat periytymishierarkian rinnalle oman totetushierarkiansa.

Esimerkki 5.15 Kuvassa 5.2 esitellyssä luokkahierarkiassa luokka `Henkilö` periytyy luokasta `Object`, joten jokaisella `Henkilö`-oliolla on `Object`-luokalta periytyvät piirteet kuten `clone`, `equals`, `hashCode` ja `toString`. Koska `Työntekijä` perii `Henkilö`-luokan, myös työntekijäoliot omaavat `Object`-kyvyt periytymisen transitiivisuuden nojalla. Tämän lisäksi luokka `Henkilö` (ja kaikki sen perilliset) lupaavat toteuttaa `Comparable`-rajapinnan.

Javan kieliopin mukaan luokka perii yliluokaltaan kaikki **`public`**- ja **`protected`**-piirteet (sekä *package*-piirteet, jos perijä on samassa pakkauksessa), paitsi konstruktorin, joka ei koskaan periydy. Perittyjen piirteiden suojausmääreitä ei voi muuttaa perimispolulla tiukemmaksi, joten perittävän luokan julkiset piirteet tulevat olemaan julkisia myös perijässä. Näin julkisessa liitännässä olevien piirteiden määrä kasvaa mentäessä luokkahierarkiassa alaspäin. Erikoistuminen tarkoittaa kuitenkin samalla myös sitä, että vastaavien olioiden joukko kaventuu samalla (so. putkimiehet muodostavat vain osan työntekijöistä ja työntekijät vain osan henkilöistä — eikä ole olemassa kanslisti-putkimiehiä).

5.5.1 Perijäsopimus

Informaation piilottamista (*information hiding*) pidetään yhtenä tärkeimmistä kulmakivistä koko olioajattelussa. On hyödyllistä, että asiakkaan ei tarvitse kantaa minkään näköistä huolta olioiden konkreetista esitystavasta tai siihen liittyvistä



Kuva 5.2: Esimerkki ammatinharjoittajien luokkahierarkiasta.

ylläpito-ongelmista. Näin asiakas ja toimittaja saadaan toisistaan riippumattomiksi, mikä edesauttaa huomattavasti ohjelmistosysteemin kehittämistä ja ylläpitoa. Periytymisessä asioita ajatellaan yleensä kuitenkin toisin: peruslähtökohta on, että perijällä pitää olla vapaa pääsy yliluokan konkreettiin esitystapaan, jotta uudet piirteet saataisiin toteutettua aiempia valintoja hyväksikäyttäen. Näin luokan toteuttajan on tasapainoiltava kahden eri intressipiirin välillä: (a) asiakkaille on muodostettava järkevä julkinen liitäntä ja piilotettava kaikki muu, ja (b) perijöille on annettava kaikki toteutuksen kannalta oleellinen materiaali. Tilannetta karakterisoidaankin usein sanonnalla *asiakas käyttää ulospäin annettua liitäntää, jälkeläinen luokan implementointia.*

Yliluokan toteutuksen käyttö perijässä sitoo nämä luokat tiukasti toisiinsa. Jos perittyihin jäsenmuuttujiin tehdään yliluokassa muutoksia (lisäyksiä, poistoja, tyyppimuutoksia tai vaikkapa vain niiden semantiikkaa muutetaan), ne aiheuttavat välittömästi muutostarvetta kaikissa luokan perijöissä. Näin voimakas riippuvuus antaa aiheen pohdiskella, olisiko mitään tapaa suojata perijää yliluokassa tapahtuvilta muutoksilta muodostamalla niiden väliin jonkinlainen suojamuuri, joka estää perityssä luokassa tapahtuvien muutosten etenemisen perijään. Tavoitteena on yksinkertaisesti miettiä *perijälle suunnattua liitäntää.*

Määrittelyn kannalta

Alkuperäistä rutiinimäärittelyä (so. alku- ja loppuehtoja) voidaan muuttaa, kunhan se tehdään kontrolloidusti: jos rutiini toteuttaa alkuperäisen määrittelyn ali-

sopimuksen, asiakas on tyytyväinen. Sama pätee myös periytymisessä. Alisopimuskäytäntö ilmenee periytymispolulla siten, että alkuperäinen määrittely esitetään silloin, kun piirre ensi kerran esitellään ja määrittelyä *täydennetään* tarvittaessa sitä mukaa, kun polulla siirrytään alaspäin. Jos siis haluaa muodostaa rutiinin täydellisen määrittelyn, kyseiset osat on kerättävä periytymispolulla olevista luokista. Alisopimusperiaatteen nojalla *korvaava (tai toteuttava) rutiini voi heikentää alkuehtoa ja loppuehto voi vahventua* (ks. kohta 2.2.2). Näiden sääntöjen tulisi olla voimassa, jotta asiakas olisi tyytyväinen kutsuessaan yliluokan rutiinia, joka dynaamisen sidonnan ansiosta hoidetaan perijäluokan korvaavalla toteutuksella. Luokan toteuttaja voi tietysti varustaa rutiinin **final**-määreellä, joka estää korvaavan toteutuksen antamisen perijöissä. Tällä taataan kyseisen rutiinin toimivan aina samalla tavoin.

Mutta ei tässä vielä kaikki. Usein perijäluokka on tyytyväinen niihin päätöksiin, jotka koskevat yliluokan konkreettia esitystapaa, koska se on sovelias perijällekin, joka voi perustaa omien (uusien) piirteiden toteutukset samaan esitystapaan. Kun näin toimitaan, pitää huomioida, että yliluokan operaatioiden oikeellisuus perustuu kyseisen luokan luokkainvariantin voimassaoloon. Näin ollen, jos perijäluokka käyttää yliluokan konkreettia esitystapaa hyväkseen, myös sen pitää kunnioittaa yliluokan luokkainvarianttia. Vain se takaa, että yliluokalta perityt piirteet toimivat perijässä oikein, siis määrittelyjensä mukaisesti. Perijäluokan koko luokkainvariantti muodostetaan siis lisäämällä perijälle ominaiset osat yliluokasta saatuun invarianttiin.

Esimerkki 5.16 Oletetaan, että luokkakirjastossa on luokka `ParillinenLaskuri`, joka mallintaa parillista kokonaislukulaskuria:

```
/**
 *  @.classInvariantProtected laskuri on aina parillinen
 */
public class ParillinenLaskuri
  {
  protected int laskuri;

  public ParillinenLaskuri() { laskuri = 0;}
  public int annaLaskuriArvo() { return laskuri; }
  public void kasvataKahdella() { laskuri += 2; }
  }
```

Luokka on oikein muodostettu, mutta se poikkeaa aiemmin käyttämästämme periaatteesta siinä, että se tarjoaa perijöilleen suoran pääsyn konkreettiin esitystapaan `laskuri` (yllättävän yleinen menettely alan oppikirjoissa ja siten ilmeisesti myös käytännössä).

Tuntuisi luonnolliselta hyväksikäyttää tehtyä toteutusta yleisessä laskuriluokassa:

```
public class YleinenLaskuri extends ParillinenLaskuri
```

```
{
public YleinenLaskuri() { laskuri = 0; }
public void kasvataYhdellä() { ++laskuri; }
}
```

Polymorfismi sallii saman olion käsittelyn usean erilaisen julkisen liitännän kautta, koska monet erilaisen staattisen tyypin omaavat tunnisteet voivat viitata samaan olioon. Tämä nähdään lausejonosta

```
ParillinenLaskuri parilliset;
YleinenLaskuri laskuri = new YleinenLaskuri();

parilliset = laskuri;
parilliset.kasvataKahdella();
laskuri.kasvataKahdella();
laskuri.kasvataYhdellä();
```

jonka päättyessä viittauksen `parilliset` kautta saatava näkökulma laskuriin ei ole enää kunnossa, koska sen arvo on pariton.

Analysoitaessa tilannetta havaitaan, että ongelma voidaan ratkaista kahdella tavalla:

- `YleinenLaskuri` ei pysty noudattamaan yliluokan luokkainvarianttia, joten koko rakennelma on väärin tehty ja pitää purkaa. Tämä olisi havaittu jo `YleinenLaskuri`-luokkaa muodostettaessa, jos `laskuri` olisi **`private`**-suojattu, koska silloin perijä voi muuttaa laskuria vain rutiinia `kasvataKahdella` käyttäen.
- Muuttamalla hieman näkökulmaa voidaan päättää, että `ParillinenLaskuri` ei välttämättä vaadikaan laskurin olevan aina parillinen, vaan lupaa ainoastaan kasvattaa sitä aina kahdella. Tällöin alkuperäinen rakennelma on kunnossa, mutta luokan `ParillinenLaskuri` luokkainvariantti pitää heikentää muotoon `laskuri >= 0`.

Yhteenvetona tästä kohdasta kerrataan vielä tärkeät perusajatukset:

- Yliluokalta saadut piirteet toimivat oikein vain, jos perijässä annetut toteutukset/korvaukset noudattavat yliluokassa annettuja määrittelyjä.
- Yliluokalta saadut piirteet toimivat oikein vain, jos perijä kunnioittaa yliluokan luokkainvarianttia.

Nämä säännöt yleistyvät luonnollisella tavalla yliluokasta kaikkiin aiempiin yliluokkiin periytymisen transitiivisuuden nojalla.

Toteutuksen kannalta

Javan lähtökohtana on, että siinä missä **`public`** määrittelee asiakkaalle suunnatun liitännän, **`protected`** (ja **`public`**) määrittelee perijän liitännän. Aiemmin esitetyn mukaan tämä ei ole riittävää, vaan käyttöön pitää ottaa järeämmät välineet. Ainoa

keino estää konkreetin esitystavan muutosten eteneminen perijöihin on julistaa kaikki jäsenmuuttujat **private**-suojatuiksi. Tällä on kuitenkin omat haittansa, sillä jos perijällä ei ole mitään pääsyä yliluokan esitystapaan:

- Perijä voi joutua toteuttamaan uudelleen saman, mikä on tehty jo aiemmin yliluokassa.
- Yliluokan luokkainvariantti ei ole välttämättä enää kaikilta osin relevantti perijälle.

Tämä ei ole tietenkään hyväksyttävää. Perijä pystyy käsittelemään yliluokan jäsenmuuttujia kuitenkin julkisen liitännän tarjoamilla havainnointi- ja muunnosoperaatioilla. Mikäli kullekin jäsenmuuttujalle on käytössä nämä operaatiot, perijä tulee normaalisti varsin hyvin toimeen julkisen liitännän avulla. Ainoa ongelmatilanne syntyy silloin, kun luokan konkreetin esitystavan ja olion ominaisuuksien välillä ei ole yksi-yhteen -vastaavuutta. Olion abstrakti ominaisuus voi nimittäin olla johdettu kahdesta tai useammasta konkreetin esitystavan osasta (”instanssimuuttujasta”), jolloin asiakkaan saama näkymä on kunnossa (sitä silmälläpitäenhän koko luokka on rakennettu), mutta perijä ei ole tyytyväinen, koska se haluaa yleensä päästä käsiksi itse esitystapaan, eikä siitä johdettuihin tietoihin. Perijällä pitää siis olla kutakin konkreetin esitystavan osaa (instanssimuuttujaa) kohti käytössä havainnointi- ja muunnosoperaatiot. Perijä saa näistä osan tavallisesti yliluokan julkisesta liitännästä ja puuttuvat toteutetaan **protected**-suojattuina.

Esimerkki 5.17 Tarkastellaan luokkaa `A`:

```
public class A
  {
  private int luku;

  public int annaLuku() { return luku; }

  protected void asetaLuku(int uusiLuku) { luku = uusiLuku }
  }
```

Jäsenmuuttujalle `luku` tarjotaan julkisen liitännän kautta havainnointioperaatio. Sen sijaan muunnosoperaatio on suunnattu luokan perijälle, mikä käy ilmi suojausmääreestä. Tämä mahdollistaa jäsenmuuttujan *kontrolloidun käsittelyn* vaikkapa alkuehtojen avulla.

Mietitäänpä esitellyn tekniikan seurauksia perijän kannalta:

- Jos perijällä on käytössä havainnointi- ja muunnosoperaatiot konkreetin esitystavan kullekin osalle, perijä pääsee käsiksi yliluokan esitystapaan. Näin samaa tietoa sisältäviä jäsenmuuttujia ei tarvitse esitellä perijäluokassa uudelleen.

- Yliluokan luokkainvariantti voidaan kirjoittaa käyttämällä havainnointi- ja muunnosoperaatioita ilman eksplisiittisiä viittauksia todelliseen esitysmuotoon. Tästä taas seuraa se, että luokkainvariantti on kaikilta osin merkityksellinen perijälle, joka voi täten huolehtia siitä, että sen operaatiot täyttävät yliluokan invariantin sanelemat velvoitteet.

- Jos yliluokan esitystapaan joudutaan tekemään muutoksia, riittää, että vastaava(t) havainnointi- ja muunnosrutiinit muutetaan vastaavasti, koska sen jälkeen perijöiden ja asiakkaiden koodit toimivat edelleen moitteetta.

- Jos perijä korvaa jonkin havainnointi- tai muunnosoperaation käyttäen mahdollisesti oman luokkansa konkreettia esitystapaa, myös kaikki perityt rutiinit toimivat edelleen oikein edellyttäen, että ne on toteutettu havainnointi- ja muunnosoperaatioiden avulla kuten yllä on esitetty.

Listan kaksi ensimmäistä kohtaa osoittavat, että perijä ei ole "hävinnyt" mitään verrattuna siihen vaihtoehtoon, että konkreetin esitystavan tiedot olisi suojattu **`protected`**-määreellä ja perijäliitännän operaatiot jätetty kirjoittamatta. Kaksi jälkimmäistä kohtaa kertovat ne hyödyt, jotka menettelyllä saavutetaan. Erityisen tärkeää on se, että muutos luokan konkreetissa esitystavassa ei aiheuta muutoksia perijöihin. Jos yliluokka on käsitellyt konkreettia esitystapaansa omien havainnointi- ja muunnosoperaatioiden kautta, myös perijässä tehdyt korvaukset havainnointi- ja muunnosoperaatioihin säilyttävät niitä käyttävien perittyjen piirteiden määrittelyt.

5.5.2 Käyttäytymisroolien yhdistäminen

Olemme tähän mennessä tarkastelleet periytymisen perustilannetta, missä konkreetti luokka perii toisen konkreetin luokan. Yleistetäänpä tilannetta siten, että perijä voi samalla toteuttaa myös mielivaltaisen määrän rajapintoja. Tämä voi aiheuttaa ongelmatilanteita erityisesti sen takia, että rajapinnoissa voi olla esiteltynä samannimisiä abstrakteja tai perijässä ja perittävässä samannimisiä konkreetteja piirteitä. Nämä konfliktitilanteet ratkeavat suhteellisen helposti: abstraktit piirteet voidaan *yhdistää* ja konkreettien tapauksessa tehdä joko korvaus tai ylikuormitus. Näin kaikki abstraktit piirteet saadaan hoidettua keskenään ja konkreetit keskenään. Mikäli tämän jälkeen on olemassa nimikonflikti vielä yksittäisen abstraktin ja konkreetin rutiinin välillä, jälkimmäinen antaa toteutuksen edelliselle.

Mietitäänpä rajapintojen yhdistämistä perijässä tarkemmin (huomaa, että perijä voi olla mikä tahansa luokka, myös rajapinta). Jos rajapinnoilla on samannimisiä piirteitä, asia tulkitaan perijässä seuraavasti:

- Saman signatuurin omaavat rutiinit yhdistyvät yhdeksi. Yhdistäminen on helppoa sikäli, että perityillä piirteillä ei ole toteutuksia. Tämä javamainen

näkemys on kuitenkin varsin pintapuolinen. Yhdistäminen ei ole käytännössä näin yksinkertaista, koska rutiineilla on aina myös määrittelyt siinä rajapintaluokassa, jossa ne on esitelty. *Yhdistäminen on laillista vain, jos määrittelytkin ovat tarkalleen samat, muussa tapauksessa ei.* Esimerkiksi luokka `Korttipakka` voisi toteuttaa rutiinin `nosto`, samoin rajapinta `Pankkitili` voisi esitellä samannimisen piirteen. Vaikka näiden `nosto`-rutiinien signatuurit olisivat samat, ei niitä tietenkään voi yhdistää perijässä (paitsi jos halutaan, että joka kerta kun pelaaja nostaa pakasta kortin, pankkitililtä häipyy 100 €), koska niiden määrittelyt poikkeavat selvästi toisistaan. Lopputulos on, että ajateltu periytymishierarkia on muodostettava toisin, käyttämällä toista käsitettä vaikkapa asiakasrelaation kautta.

- Jos rutiinien signatuurit ovat muuten samat, mutta niiden tulostyyppi on erilainen, rajapintoja ei voi yhdistää perijässä.

- Jos rutiineilla on eri määrä tai erityyppisiä argumentteja, kyseessä on ylikuormitus.

- Jos rutiinien otsikot eroavat vain poikkeusten osalta, ne voidaan yhdistää kuten korvaussääntöjen yhteydessä mainittiin (ks. kohta 5.4) ottamalla perijärutiiniin poikkeusnimien leikkaus.

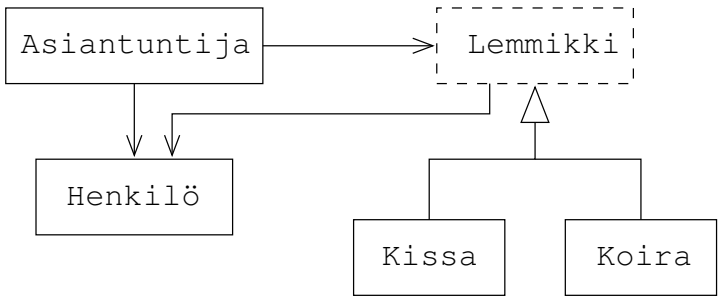
Kun rajapinnoista saadut piirteet on yhdistetty näiden sääntöjen mukaan, perijäluokka voi antaa ko. rutiinille toteutuksen itse tai se voidaan saada periytymisen kautta abstraktilta tai konkreetilta luokalta. On myös mahdollista, että perijä tekee korvauksen ei-rajapinnalta saamalleen rutiinille, joka samalla toimii rajapinnassa esitellyn rutiinin toteutuksena.

5.6 Esimerkkejä

Rajapintaluokat, polymorfismi ja dynaaminen sidonta ovat voimakkaita apuvälineitä ohjelman kirjoittajalle. Tarkastellaanpa niitä seuraavien esimerkkien avulla.

5.6.1 `Asiantuntija`

Seuraavassa on pieni esimerkki siitä, kuinka polymorfismi ja dynaaminen sidonta helpottavat ohjelmiston ymmärrettävyyttä ja ylläpitoa. Kyseessä on tietojärjestelmä, joka simuloi lemmikkejä tutkivaa eläinasiantuntijaa hoivanäkökulmasta. Yksittäistä lemmikkiä kuvaa abstraktilla tasolla luokka `Lemmikki`, jonka toteutus on esitelty listauksessa 5.2. Abstrakti rutiini `äännähdys` kertoo, miten lemmikki ääntelee. Listauksessa 5.3 esitelty kissaluokka `Kissa` ja listauksessa 5.4 esitelty koiraluokka `Koira` ovat `Lemmikki`-luokan konkreetteja perijöitä. Kummallekin lajille on



Kuva 5.3: Lemmikkien hoitojärjestelmän luokkakaavio.

annettu omat tunnusmerkilliset ominaisuutensa: kissan yhdeksän henkeä ja koiraveron maksutieto. Tämän jälkeen voidaan tarkastella eläinasiantuntijan hommia, jota mallintaa listauksessa 5.5 esitelty luokka `Asiantuntija`. Nyt kaikki tarvittavat peruselementit on kasassa (ks. kuva 5.3).

Luokkakokonaisuutta käyttävässä sovelluksessa voidaan esitellä muuttujat

```
Asiantuntija anja;
Collection<Lemmikki> höpönassut;
```

Muuttuja `höpönassut` alustettakoon sekalaiseksi kokoelmaksi kissoja ja koiria. Tämän jälkeen se voidaan antaa asiantuntijalle analysoitavaksi:

```
anja.asetaHoidettavat(höpönassut);
Collection<Lemmikki> harvatJaValitut =
                                anja.valitseLajittain(Kissa.class);
System.out.println("Kokoelman kissoja ovat: " + harvatJaValitut);
```

Kokoelman `harvatJaValitut` kaikki lemmikit ovat kissoja. Tulostus on mielenkiintoinen, sillä siinä kullekin `harvatJaValitut`-kokoelman alkiolle kutsutaan korvattua `toString`-rutiinia (joka antaa yleisten lemmikkitietojen lisäksi myös kissalle ominaiset tiedot). Kannattaa huomata, että tulostamisessa ei ole mitään kissoihin tai ei edes lemmikkeihin viittaavaa, vaan toiminta perustuu puhtaasti polymorfismiin ja dynaamiseen sidontaan. Itse asiassa tulostusrutiini on niin yleinen, että sillä voidaan tulostaa minkä tahansa `Collection`-tyyppisen kokoelman kaikki tiedot! Tässä näkyy mainittujen mekanismien ilmaisuvoima puhtaimmillaan: tulostustoiminto on saatu abstrahoitua niin korkealle tasolle (kokoelmaluokkaan), että se on käytettävissä missä tahansa ohjelmistosysteemissä sellaisenaan.

Asiantuntijaohjelmistoa laajennettaessa tulee varmasti tarvetta lisätä uusia lemmikkilajeja: marsuja, papukaijoja, pingviinejä jne. Mitä tämä tarkoittaa luokan `Asiantuntija` kannalta? Ei mitään, se toimii täsmälleen samalla tavalla kuin aiemminkin.

Listaus 5.2: Abstrakti luokka `Lemmikki`.

```
public abstract class Lemmikki
  {
  private String nimi;
  private int syntymävuosi;
  private Henkilö omistaja;

  /**
   *  @.pre  nimi != null & omistaja != null
   *  @.post annaNimi() == nimi &
   *         annaSyntymävuosi() == syntymävuosi &
   *         annaOmistaja() == omistaja
   */
  public Lemmikki(String nimi, Henkilö omistaja, int syntymävuosi)
    {
    this.nimi = nimi;
    this.omistaja = omistaja.clone();
    this.syntymävuosi = syntymävuosi;
    }

  public String annaNimi() { return nimi; }

  public int annaSyntymävuosi() { return syntymävuosi; }

  public Henkilö annaOmistaja() { return omistaja; }

  public abstract String äännähdys();

  public String toString()
    {
    return "(" + nimi + ", " + syntymävuosi + ", "  +
           omistaja.annaNimi() + ")";
    }
  }
```

Listaus 5.3: Luokka `Kissa`.

```
public class Kissa extends Lemmikki
  {
  private static final int ELÄMIEN_LUKUMÄÄRÄ = 9;
  private int jäljelläOlevatElämät;

  public Kissa(String nimi, Henkilö omistaja, int syntymävuosi)
    {
    super(nimi, omistaja, syntymävuosi);
    jäljelläOlevatElämät = ELÄMIEN_LUKUMÄÄRÄ;
    }

  public String äännähdys()
    {
    return !onKuollutKuinKivi() ? "Miau" : "";
    }

  public int elämät() { return jäljelläOlevatElämät; }

  /**
   *  @.pre  !onKuollutKuinKivi()
   *  @.post elämät() == OLD(elämät()) - 1
   */
  public void kuole() { jäljelläOlevatElämät--; }

  /**
   *  @.pre  true
   *  @.post RESULT == (elämät() == 0)
   */
  public boolean onKuollutKuinKivi()
    {
    return (jäljelläOlevatElämät == 0);
    }

  public String toString()
    {
    return "(" + super.toString() + ", elämiä " + elämät() + ")";
    }
  }
```

Listaus 5.4: Luokka `Koira`.

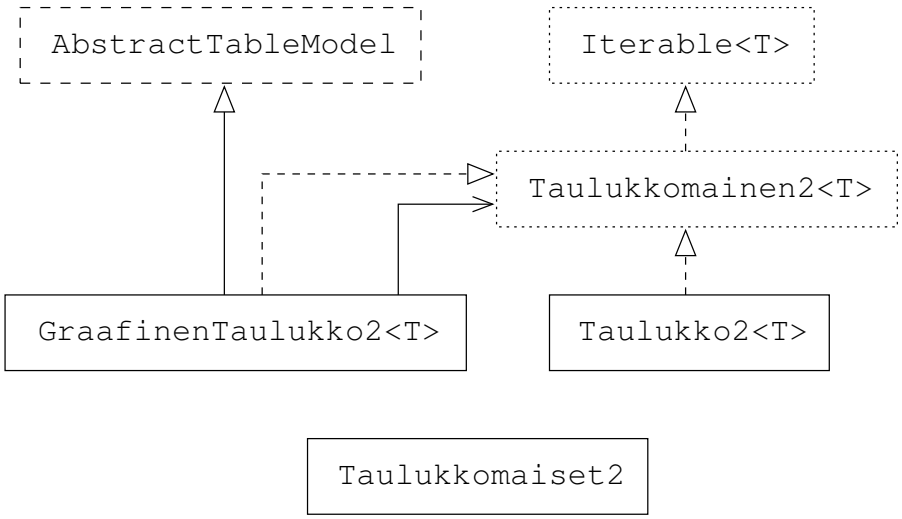
```
public class Koira extends Lemmikki
  {
  private boolean veroMaksettu;

  public Koira(String nimi, Henkilö omistaja, int syntymävuosi)
    {
    super(nimi, omistaja, syntymävuosi);
    veroMaksettu = false;
    }

  public String äännähdys() { return "Hau"; }

  /**
   *  @.pre  true
   *  @.post onVeroMaksettu()
   */
  public void maksaVero() { veroMaksettu = true; }

  /**
   *  @.pre  true
   *  @.post RESULT == (vero on maksettu)
   */
  public boolean onVeroMaksettu() { return veroMaksettu; }

  public String toString()
    {
    return "(" + super.toString() + ", vero " +
           (onVeroMaksettu() ? "maksettu" : "maksamatta") + ")";
    }
  }
```

Listaus 5.5: Luokka `Asiantuntija`.

```
public class Asiantuntija
  {
  private Henkilö henkilötiedot;
  private Collection<Lemmikki> hoidettavat;

  /**
   *  @.pre  tiedot != null
   *  @.post annaHenkilötiedot() == tiedot
   */
  public Asiantuntija(Henkilö tiedot) { henkilötiedot = tiedot; }

  /**
   *  @.pre  true
   *  @.post RESULT == (asiantuntija henkilönä)
   */
  public Henkilö annaHenkilötiedot() { return henkilötiedot; }

  /**
   *  @.pre  true
   *  @.post RESULT == (hoidettavat lemmikit näkymänä)
   */
  public Collection<Lemmikki> annaHoidettavat()
    {
    return Collections.unmodifiableCollection(hoidettavat);
    }

  /**
   *  @.pre  eläimet != null
   *  @.post annaHoidettavat().equals(eläimet)
   */
  public void asetaHoidettavat(Collection<Lemmikki> eläimet)
    {
    hoidettavat = new LinkedList<Lemmikki>(eläimet);
    }
```

Listaus 5.5 (jatkoa): Luokka `Asiantuntija`.

```
/**
 *  @.pre  laji != null
 *  @.post  annaHoidettavat().contains(RESULT) &
 *          FORALL(e : RESULT; e.getClass().equals(laji)) &
 *          !EXISTS(e : annaHoidettavat();
 *               e.getClass().equals(laji) & !RESULT.contains(e))
 */
public Collection<Lemmikki>
  valitseLajittain(Class<? extends Lemmikki> laji)
  {
  Collection<Lemmikki> tulos = new LinkedList<Lemmikki>();
  for (Lemmikki lutunen : hoidettavat)
    {
    if (lutunen.getClass().equals(laji))
      tulos.add(lutunen);
    }
  return tulos;
  }

/**
 *  @.pre  true
 *  @.post RESULT == (hoidettavien kootut äännähdykset)
 */
public String annaYhteishuuto()
  {
  String tulos = "";
  boolean eka = true;
  for (Lemmikki höpönassu : hoidettavat)
    {
    tulos += (eka ? "" : " ") + höpönassu.äännähdys();
    eka = false;
    }
  return tulos;
  }
}
```



Kuva 5.4: Taulukoinnin luokkakaavio.

5.6.2 `Taulukko2`

Kuvan 5.4 luokkakokonaisuudessa on eroteltu toisistaan taulukko ja sen esitysasu. Rajapinta `Taulukkomainen2` määrittelee kaksiulotteisen taulukon käsittelyyn liittyvät piirteet. Tälle rajapinnalle on annettu konkreetti toteutus luokassa `Taulukko2`. Luokka `Taulukkomaiset2` kokoaa yhteen luokkametodeina taulukkojen käsittelyalgoritmeja (esim. diagonaalin erottaminen).

Pakkauksessa `javax.swing.table` on määritelty `TableModel`-rajapinta, jolle on annettu osittainen toteutus abstraktissa luokassa `AbstractTableModel`. Tämän luokan perii `GraafinenTaulukko2`-luokka, joka kiinnittää sen yhteen `Taulukkomainen2`-rajapinnan toteuttavien luokkien kanssa asiakasrelaatiolla. Kyseessä on siis *bridge*-suunnittelumallin ilmentymä.

5.6.3 `Observer` ja `Observable`

Pakkauksesta `java.util` löytyvät luokka `Observable` ja rajapinta `Observer`, jotka toteuttavat *observer*-suunnittelumallin. Käytetäänpä niitä ja luodaan aluksi luokka `Kansalainen`, joka perii `Observable`-luokan:

```
public class Kansalainen extends Observable
  {
  private String ajatus;
  private String nimi;

  public Kansalainen(String nimi) { this.nimi = nimi; }
```

```
  public void ajattele(String ajatus)
    {
    this.ajatus = ajatus;
    setChanged();
    notifyObservers(ajatus);
    }

  public String toString() { return nimi; }
  }
```

Luokka `Firma` puolestaan toteuttaa `Observer`-rajapinnan ja siinä määritellyn metodin `update` seuraavasti:

```
public class Firma implements Observer
  {
  private String nimi;

  public Firma(String nimi) { this.nimi = nimi; }

  public void update(Observable o, Object a)
    {
    System.out.println(nimi + " tiedottaa: "
                       + o + " ajatteli: " + a);
    }
  }
```

Havainnoitava (so. `Observable`) `Kansalainen`-olio notifioi kaikkia tarkkailijoitaan aina kun metodia ajattele kutsutaan ja välittää samalla niille merkkijonon `ajatus`. Kun havaitseva (so. `Observer`) `Firma`-olio saa tiedon päivityksestä, `update`-metodi tulostaa ilmoituksen lähettäneestä olion ja sen viestin. Metodi `addObserver` liittää havainnoijan havainnoitavaan ja `deleteObserver` poistaa liitoksen. Yhdellä havainnoitavalla voi olla useita havainnoijia ja yksi havainnoija voi tarkkailla useita havainnoitavia. Niinpä seuraavat käskyt

```
Kansalainen minä = new Kansalainen("Matti Meikäläinen");
Kansalainen sinä = new Kansalainen("Anna Malli");
Firma nsa = new Firma("NSA");
Firma cia = new Firma("CIA");
Firma fbi = new Firma("FBI");

minä.addObserver(nsa);
sinä.addObserver(nsa);
sinä.addObserver(fbi);
minä.ajattele("Mistä sais nörttisuklaata?");
```

```
sinä.ajattele("Mul on tylsää...");
minä.addObserver(cia);
minä.ajattele("Onx tää kurssi kohta ohi?");
sinä.deleteObserver(fbi);
sinä.ajattele("Hei, vakoileex joku mua?");
```

tulostavat merkkijonot

```
NSA tiedottaa: Matti Meikäläinen ajatteli: Mistä sais nörttisuklaata?
FBI tiedottaa: Anna Malli ajatteli: Mul on tylsää...
NSA tiedottaa: Anna Malli ajatteli: Mul on tylsää...
CIA tiedottaa: Matti Meikäläinen ajatteli: Onx tää kurssi kohta ohi?
NSA tiedottaa: Matti Meikäläinen ajatteli: Onx tää kurssi kohta ohi?
NSA tiedottaa: Anna Malli ajatteli: Hei, vakoileex joku mua?
```

5.7 Periytymisen käyttö eri tilanteisiin

Vaikka periytyminen on alunperin suunniteltu alityypitystä varten, ja esimerkiksi dynaaminen sidonta toimii järkevästi vain silloin, kun korvaavuusperiaate toteutuu, käytetään periytymistä myös hämmästyttävän moneen muuhun tarkoitukseen. Seuraavassa on lueteltu yleisimmät käyttötavat.

- **Erikoistaminen** (*inheritance for specialization*): Tämä on ongelmattomin tapa käyttää periytymistä. Erikoistamisessa perijä edustaa jotain erikoistyyppistä yliluokan käsitettä ja korvaavuusperiaate on voimassa. Hieman kärjistäen (mutta vain hieman) voidaan sanoa, että luokkahierarkia on järkevä vain, jos muita periytymistapoja ei ole sovellettu. Erikoistamisessa on tyypillistä joidenkin rutiinien korvaaminen perijän tarpeisiin sopivaksi.

- **Tarkentaminen** (*inheritance for specification*): Tämä on erikoistapaus edellisestä. Nyt perijä ja perittävä omaavat tarkalleen saman julkisen liitännän. Ideana on, että perittävä on joko rajapintaluokka tai abstrakti luokka ja perijä toteuttaa abstraktit piirteet.

- **Yhdistäminen** (*inheritance for combination*): Yhdistäminen tarkoittaa tilannetta, missä luokka perii konkreetilta tai abstraktilta luokalta ja usealta rajapinnalta samanaikaisesti. Rajapinnat määrittelevät ne normaalia käyttäytymistä täydentävät roolit, joissa ko. luokan olio voi kulloinkin esiintyä.

- **Toteutusperiytyminen** (*inheritance for construction*, *mixing inheritance*): Joskus tulee eteen tilanteita, joissa on helppo nähdä, että olemassaoleva luokka tarjoaa melko pienillä muutoksilla kaikki ne toiminnot, jotka uuteen luokkaan halutaan. Periaatteessa tällöin tulisi käyttää asiakasrelaatiota, mutta

on houkuttelevaa periä kyseinen ”työkaluluokka”, koska silloin rutiinien toteutus tulee lyhyeksi. Ongelmana tässä lähestymistavassa on se, että perijän julkiseen liitäntään tulevat väistämättä mukaan *kaikki* työkaluluokan julkiset piirteet (alkuperäisine nimineen), myös sellaiset, joilla *ei ole mitään tekemistä perijään liittyvän abstraktin käsitteen kanssa.* Näin perijän asiakas saa käyttöönsä välineitä, joilla voi rikkoa luokkainvariantin.[11]

- **Ad hoc -periytyminen**: Tilanne, joka selviää helposti periytymistä käyttämällä, vaikka tulisikin ratkaista muilla tavoin.[12] Toteutusperimistä voidaan pitää yhtenä erikoistapauksena ad hoc -perimisestä.

Periytymisen käyttö erilaisiin tarkoituksiin antaa aiheen olettaa, että ko. mekanismin luonnetta ei ole vielä oivallettu kaikilta osin: sillä on järkevä peruslähtökohta, mutta muiden käyttötapojen ansiosta perijäluokkien julkisista liitännöistä tulee helposti sekavia tai dynaaminen sidonta johtaa odottamattomiin tilanteisiin. Asia vaatinee kypsyttelyä; ehkäpä jo seuraava menestyskieli ratkaisee osan esiintulleista ongelmista? Me joka tapauksessa palaamme vielä uudelleen asiakas/periytymis -vastakkainasetteluun luvussa 9.

Tehtäviä

5-1 Oletetaan seuraavat luokka- ja rajapintamäärittelyt:

```
public interface A
  {
  public int laske(double x, int y);
  public void asetaArvo(int x);
  }

public interface B
  {
  public int laske(int x, double y);
  public void asetaMerkki(char c);
  public String sisältö();
  }

public class X
  {
  public int annaArvo() { /* ... */ }
```

[11] Tarkastelepa Javan kokoelmaluokkia `Stack` ja `Vector` ja mieti kuinka helposti luokkainvariantti rikkoutuu niitä käytettäessä.

[12] Kaikkein selvimmin tämäntyyppinen perinnän väärinkäyttö näkyy esimerkiksi Java-luokissa `Cloneable` ja `Serializable`.

```
  public void asetaArvo(int x) { /* ... */ }
  public char annaMerkki() { /* ... */ }
  public String sisältö() { /* ... */ }
  }

public class Y extends X implements A, B
  {
  public int annaLuku() { /* ... */ }
  public void asetaMerkki(char c) { /* ... */ }
  public int laske(int x, double y) { /* ... */ }
  public int laske(double x, int y) { /* ... */ }
  public String sisältö() { /* ... */ }
  public Object olioi() { /* ... */ }
  }
```

(a) Selvitä periytymissuhteita kuvallisesti.

(b) Mitä metodeja seuraavilla `A`-, `B`-, `X`- ja `Y`-tyyppisillä viittauksilla pystytään kutsumaan?

- `A a = new Y();`
- `B b = new Y();`
- `X x = new Y();`
- `Y y = new Y();`

(c) Metodi `laske` on ylikuormitettu. Kumpaa metodia seuraavat kutsut käyttävät?

- `int i = y.laske(1, 1.0);`
- `int i = y.laske(1.0, 1);`
- `int i = y.laske(1, 1);`

5-2 Miksi abstraktia piirrettä ei voi varustaa määreellä:

(a) **`private`**

(b) **`static`**

(c) **`final`**

5-3 Laadi konkreetti toteutus luokalle, joka periytyy listauksessa 5.1 esitellystä abstraktista luokasta `RajoitettuPino`.

5-4 Toteuta koordinaattiparia (x, y) kuvaava luokka `Piste`, joka periytyy listauksessa 5.6 esitellystä luokasta `Lukupari`. Luokassa on oltava mm. muunnos- ja havainnointimetodit (`annaX()`, `annaY()`, `asetaX()`, `asetaY()`) sekä ylikuormitettu metodi `etäisyys`, joka palauttaa pisteen etäisyyden toiseen pisteeseen (`etäisyys(Piste)`) tai origoon (`etäisyys()`).

Listaus 5.6: Luokka `Lukupari`.

```
public abstract class Lukupari
  {
  private double arvo1;
  private double arvo2;

  protected Lukupari(double a1, double a2)
    { arvo1 = a1; arvo2 = a2; }

  protected double annaArvo1() { return arvo1; }
  protected double annaArvo2() { return arvo2; }
  protected void asetaArvo1(double a) { arvo1 = a; }
  protected void asetaArvo2(double a) { arvo2 = a; }

  protected void lisää(Lukupari toinen)
    {
    arvo1 += toinen.annaArvo1();
    arvo2 += toinen.annaArvo2();
    }

  protected double ero(Lukupari toinen)
    {
    double muutos1 = arvo1 - toinen.annaArvo1();
    double muutos2 = arvo2 - toinen.annaArvo2();
    return Math.sqrt(muutos1 * muutos1 + muutos2 * muutos2);
    }

  protected void negatoi()
    { arvo1 = -arvo1; arvo2 = -arvo2; }

  public String toString()
    { return "Lukupari: " + arvo1 + ", " + arvo2; }
  }
```

5-5 Toteuta kompleksilukua kuvaava luokka `Kompleksiluku`, joka periytyy listauksessa 5.6 esitellystä luokasta `Lukupari`. Luokassa on oltava mm. ylikuormitettu metodi `lisää` kompleksi- tai reaaliluvun lisäämiselle kompleksilukuun sekä `itseisarvo`-metodi, joka palauttaa kompleksiluvun itseisarvon, ja `negatoi`, joka muuttaa kompleksiluvun vastaluvuksi. (Kompleksilukujen $a+bi$ ja $c+di$ summa on $(a+c)+(b+d)i$ ja kompleksiluvun $a+bi$ itseisarvo on $\sqrt{a^2+b^2}$.)

5-6 Kirjoita listauksessa 5.5 annettuun `Asiantuntija`-luokkaan metodi `etsiVanhin`, joka etsii ja palauttaa parametrina annetun eläinlajin vanhimman edustajan. Mikäli annetun lajin edustajia ei ole, metodi palauttaa **`null`**.

5-7 Olkoon luokat `Henkilö` ja `Opiskelija` seuraavanlaiset:

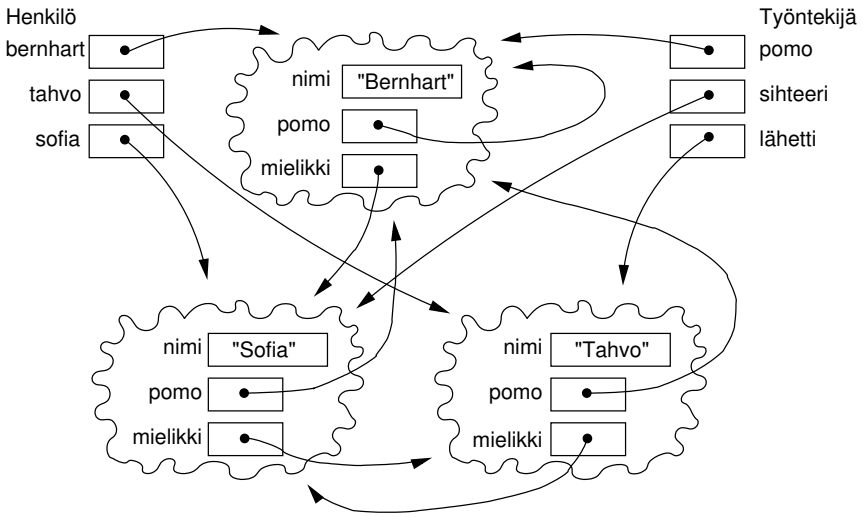
```
public class Henkilö
  {
  private String nimi;

  public Henkilö(String nimi) { this.nimi = nimi; }
  public String annaNimi() { return nimi; }
  public String toString() { return "Minä henkilö, " + nimi; }
  }

public class Opiskelija extends Henkilö
  {
  private int opNumero;

  public Opiskelija(String nimi, int opNumero)
    { super(nimi); this.opNumero = opNumero; }

  public int annaOpNumero() { return opNumero; }
  public String toString() { return "Minä opiskelija, " + opNumero; }
  }
```

Ajetaan eo. luokille seuraava testiohjelma:

```
Henkilö henkilö1 = new Henkilö("Sanna");
Henkilö henkilö2 = new Henkilö("Ville");
Opiskelija opiskelija1 = new Opiskelija("Matti", 99555);
henkilö1 = opiskelija1;
Opiskelija opiskelija2 = opiskelija1;
Opiskelija opiskelija3;
```

(a) Mitä vikaa on seuraavissa lauseissa?

```
System.out.println(henkilö1.annaOpNumero());
opiskelija2 = (Opiskelija)henkilö2;
```



Kuva 5.5: Oliokuva kolmiodraamasta, jossa auktoritäärinen Bernhart ja tunteikkaat Sofia ja Tahvo törmäävät. Muuttujat `bernhart`, `tahvo`, ja `sofia` ovat tyyppiä `Henkilö` sekä muuttujat `pomo`, `sihteeri` ja `lähetti` tyyppiä `Työntekijä`.

(b) Tee lause, jonka jälkeen `opiskelija3` viittaa muuttujan `henkilö1` viittaamaan olioon.

(c) Mitä seuraavat lauseet tulostavat kohdan (b) asetuslauseen jälkeen?

```
System.out.println(opiskelija3.getClass().getName());
System.out.println(henkilö1.getClass());
System.out.println(henkilö1.annaNimi().getClass());
```

5-8 Eräs ystäväsi on innostunut elokuvista ja niissä esiintyvistä kolmiodraamoista. Erityisesti häntä kiinnostaa elokuvassa sellainen moderni kolmiodraama, joka sijoittuu työpaikalle auktoritäärisen ympäristön ja yksilöiden tunteiden ristiaallokkoon. Tehtävänäsi on toteuttaa alla annettu konkreetti luokka `Työläinen`, jonka avulla ohjelman ajoaikana voidaan luoda kuvan 5.5 kaltaisia rakenteita. Luokkaan `Työläinen` ei saa määritellä lisää rutiineja, joten toteuta vain ko. luokassa jo annetut ja sinne periytyneet rutiinit. Anna lisäksi ohjelmakoodi, jolla kuvan 5.5 mukainen tilanne saadaan aikaan.

```
public interface Nimellinen
  {
  /**
   *  @.pre  true
   *  @.post RESULT.length() >= 1
   */
  public String annaNimi();
  } // Nimellinen
```

```
public interface Henkilö extends Nimellinen
  {
  /**
   *  @.pre  true
   *  @.post RESULT == (mielikki; null, jollei mielikkiä)
   */
  public Henkilö annaMielikki();

  /**
   *  @.pre  true
   *  @.post annaMielikki() == mielikki
   */
  public void asetaMielikki(Henkilö mielikki);
  } // Henkilö

public interface Työntekijä extends Nimellinen
  {
  /**
   *  @.pre  true
   *  @.post RESULT != null
   */
  public Työntekijä annaPomo();
  } // Työntekijä


public class Työläinen implements Henkilö, Työntekijä
  {
  /**
   *  @.pre  nimi != null && nimi.length() >= 1
   *  @.post RESULT.annaNimi() == nimi & RESULT.annaPomo() == RESULT
   */
  public static Työläinen luoPomo(String nimi) { /* Toteuta. */ }

  /**
   *  @.pre  (nimi != null && nimi.length() >= 1) & pomo != null
   *  @.post RESULT.annaNimi() == nimi & RESULT.annaPomo() == pomo
   */
  public static Työläinen luoAlainen(String nimi, Työntekijä pomo)
    { /* Toteuta. */ }

  protected Työläinen(String nimi) { /* Toteuta. */ }
  protected void asetaPomo(Työntekijä pomo) { /* Toteuta. */ }
  } // Työläinen
```

5-9 Tarkastellaanpa vähintään yhden osallistujan kilpailuja, joissa voittaminen määräytyy osavoittojen (kuten erävoittojen) perusteella. Huomaa että yleisessä tapauksessa voittavia osallistujia voi olla useita. Määritellään voittajan valinta rajapintana `Voittajavalinta`.

```
/**
 *  Voittajan valinta osavoittojen perusteella. Osallistujia on
 *  vähintään yksi ja jokaisella osallistujalla on yksikäsitteinen
 *  osallistujanumero.
 */
public interface Voittajavalinta
  {
  /**
   *  @.pre  true
   *  @.post RESULT == (osallistujien määrä)
   */
  public abstract int osallistujamäärä();

  /**
   *  @.pre  0 <= i <= osallistujamäärä() - 1
   *  @.post RESULT == (osallistujan i osavoittojen määrä)
   */
  public abstract int annaOsavoitto(int i);

  /**
   *  @.pre  0 <= i <= osallistujamäärä() - 1
   *  @.post annaOsavoitto(i) == OLD(annaOsavoitto(i)) + 1
   */
  public abstract void asetaOsavoitto(int i);

  /**
   *  @.pre  0 <= i <= osallistujamäärä() - 1
   *  @.post RESULT == (osallistuja i on voittaja)
   */
  public abstract boolean onkoVoittaja(int i);

  /**
   *  @.pre  true
   *  @.post RESULT == (voittaneiden osallistujien määrä;
   *                    0 jollei voittajia ole vielä selvillä)
   */
  public abstract int voittajamäärä();

  /**
```

```
 *  @.pre  true
 *  @.post RESULT.length == voittajamäärä() &
 *         FORALL(i : 0 <= i < osallistujamäärä();
 *            !onkoVoittaja(i) |
 *            EXISTS(j : 0 <= j < RESULT.length; RESULT[j] == i))
 */
public abstract int[] annaVoittajat();
} // Voittajavalinta
```

Toteuta `Voittajavalinta`-rajapinnalle konkreettiset luokat `Rajanylitys` ja `Etumatka`.

(a) Määrittelyn `Rajanylitys` mukaisia voittajia ovat sellaiset osallistujat, joiden osavoittojen määrä ylittää annetun rajan. Osallistujanumerot ovat peräkkäisiä **int**-lukuja luvusta `0` alkaen. Esimerkki: osallistujalla `0` on osavoittoja 5 kpl, `1`:llä osavoittoja 2 ja `2`:lla 4 sekä ylitysraja on 3; voittajia ovat osallistujat `0` ja `2` sillä heillä on vähintäin 3 osavoittoa.

```
public class Rajanylitys implements Voittajavalinta
  {
  /**
   *  @.pre  1 <= osallistujia & 0 <= ylitysraja
   *  @.post osallistujamäärä() == osallistujia &
   *         FORALL(i : 0 <= i < osallistujamäärä();
   *                annaOsavoitto(i) == 0) &
   *         annaYlitysraja() == ylitysraja
   */
  public Rajanylitys(int osallistujia, int ylitysraja)
    { /* Toteuta tämä. */ }

  /**
   *  @.pre  true
   *  @.post RESULT == (ylitysraja)
  public int annaYlitysraja() { /* Toteuta tämä. */}

  /**
   *  @.post RESULT == (annaOsavoitto(i) >= annaYlitysraja())
   */
  public boolean onkoVoittaja(int i) { */ Toteuta tämä. */ }

  public int osallistujamäärä() { /* Toteuta tämä. */ }
  public int annaOsavoitto(int i) { /* Toteuta tämä. */ }
  public void asetaOsavoitto(int i) { /* Toteuta tämä. */ }
  public int voittajamäärä() { /* Toteuta tämä. */ }
  public int[] annaVoittajat() { /* Toteuta tämä. */ }
  } // Rajanylitys
```

(b) Määrittelyn `Etumatka` mukaisia voittajia ovat sellaiset osallistujat, joiden osavoittojen määrä on annettua rajaa suurempi kuin seuraavaksi eniten osavoittoja saaneilla. Osallistujanumerot ovat peräkkäisiä **`int`**-lukuja luvusta `0` alkaen. Esimerkki: osallistujalla `0` on osavoittoja 2 kpl, `1`:llä osavoittoja 5 ja `2`:lla 5 sekä etumatkaraja on 3; voittajia ovat osallistujat `1` ja `2` sillä heillä on sama määrä osavoittoja ja seuraavaksi jäljempänä tulevalla on 3 osavoittoa vähemmän.

```
public class Etumatka implements Voittajavalinta
  {
  /**
   *  @.pre  2 <= osallistujia & 0 <= etumatkaraja
   *  @.post osallistujamäärä() == osallistujia &
   *         FORALL(i : 0 <= i < osallistujamäärä();
   *                annaOsavoitto(i) == 0) &
   *         annaEtumatkaraja() == etumatkaraja
   */
  public Etumatka(int osallistujia, int etumatkaraja)
    { /* Toteuta tämä. */ }

  /**
   *  @.pre  true
   *  @.post RESULT == (etumatkaraja)
  public int annaEtumatkaraja() { /* Toteuta tämä. */}

  /**
   *  @.post RESULT ==
   *           !EXISTS(j : 0 <= j < osallistujamäärä() & j != i;
   *                  (annaOsavoitto(i) - annaOsavoitto(j)) <
   *                   annaEtumatkaraja())
   */
  public boolean onkoVoittaja(int i) { */ Toteuta tämä. */ }

  public int osallistujamäärä() { /* Toteuta tämä. */ }
  public int annaOsavoitto(int i) { /* Toteuta tämä. */ }
  public void asetaOsavoitto(int i) { /* Toteuta tämä. */ }
  public int voittajamäärä() { /* Toteuta tämä. */ }
  public int[] annaVoittajat() { /* Toteuta tämä. */ }

  protected static final int
     JOHTAVIA = 0, ETUMATKA = 1, OSAVOITTOJA = 2;

  /**
   *  Palauttaa johtavien osallistujien lukumäärän, heidän
   *  etumatkansa ja heidän osavoittojensa määrän (kaikilla
```

```
 *  johtavilla osallistujilla etumatka ja osavoittomäärä ovat
 *  aina samoja). Johtavan osallistujan etumatka tarkoittaa hänen
 *  osavoittomääränsä ja häntä jäljempänä olevan lähimmän
 *  osallistujan osavoittomäärän erotusta. Jos kaikilla
 *  osallistujilla on sama määrä osavoittoja, etumatka on 0.
 *
 *  @.pre  true
 *  @.post RESULT[JOHTAVIA] == (johdossa olevien määrä) &
 *         RESULT[ETUMATKA] == (johtavien etumatka) &
 *         RESULT[OSAVOITTOJA] == (johtavien osavoittojen määrä)
 */
protected int[] johtavienTiedot() { /* Toteuta tämä. */ }
} // Etumatka
```

5-10 Itse kunkin on joskus asetettava asioita (tässä tehtävässä *kohteita*) tärkeysjärjestykseen. Mallintakaamme tällainen pyrintö seuraavien julkisten määrittelyjen avulla.

(a) Rajapinta `Tärkeistetty` määrittelee funktiot `annaTärkeys` ja `annaKohde`, joiden avulla oliolta voidaan kysyä kuinka tärkeä kohde on ja mistä kohteesta on kyse.

(b) Konkreetti luokka `Tärkeä` toteuttaa rajapinnan `Tärkeistetty` mutatoitumattomana (eli muuttumattomana) oliona. Tärkeys ja kohde annetaan konstruktorin parametreina.

(c) Rajapinta `Tärkeimmät` määrittelee funktioiden `annaEninMäärä`, `annaNykyinenMäärä` ja `annaTärkeimmästä` sekä proseduurin `lisää` avulla kuinka `Tärkeistetty`-yhteensopivat oliot asettuvat keskinäiseen tärkeysjärjestykseen.

(d) Konkreetti luokka `Rajoitettu` toteuttaa rajapinnan `Tärkeimmät` *taulukko-oliona*, jonka koko annetaan konstruktorin parametrina. Tällöin siis kyetään muistamaan enintäin vain taulukon koon verran tärkeistettyjä kohteita. Sopikaamme että samantärkeisistä kohteista tuoreempi on aina tärkeämpi kuin aikaisempi.

Kirjoita rajapintamäärittelyt sekä toteuta konkreettiset luokat seuraavan käyttöesimerkin mukaiseksi. Muista määritellä myös mahdolliset alku- ja loppuehdot (erityisesti proseduurille `lisää`). (Vinkki: Luokan `Rajoitettu` taulukko-olio kannattanee pitää ei-nousevassa tärkeysjärjestyksessä.)

```
/* Selvitä kohteet, niiden tärkeydet ja luo tyhjä muistilista. */
List<Object> kohteet = annaKohteet();            // Selvitä kohteet.
int[] tärkeydet = annaTärkeydet(kohteet);
                                  // Määrittele kohteiden tärkeydet.
Tärkeimmät muistilista = new Rajoitettu(listanKoko);
                                // Luo tyhjä rajallinen muistilista.

/* Lisää tärkeistetyt kohteet muistilistaan. */
int sijainti = 0;         // Kohteen tärkeydet-taulukon vastinkohta.
```

```
Iterator<Object> läpikävijä = kohteet.iterator();
while (läpikävijä.hasNext())
  {
  Tärkeistetty tärkeäKohde = new Tärkeä(tärkeydet[sijainti],
                                        läpikävijä.next());
  muistilista.lisää(tärkeäKohde);
  ++sijainti;
  }

/* Selvitä muistilistan tärkein ja merkityksettömin kohde.
   Tärkein on etäisyydellä 0 tärkeimmästä ja merkityksettömin on
   etäisyydellä (annaNykyinenMäärä() - 1) tärkeimmästä.  */
Tärkeistetty tärkein = muistilista.annaTärkeimmästä(0);
Tärkeistetty merkityksettömin = muistilista.annaTärkeimmästä(
                              muistilista.annaNykyinenMäärä() - 1);

/* Tulosta System.out:lla muistilistan sisältö tärkeimmästä
   vähiten tärkeään. */
for (int i = 1; i <= muistilista.annaNykyinenMäärä(); ++i )
  {
  Tärkeistetty tärkeäKohde = muistilista.annaTärkeimmästä(i - 1);
  int tärkeys = tärkeäKohde.annaTärkeys();
  Object kohde = tärkeäKohde.annaKohde();
  System.out.println(i + ". listassa: tärkeys = " + tärkeys +
                                     ", kohde = " + kohde + ".");
  }

/* Ilmoita mahdollinen vajausmäärä System.out-oliolla. */
int vajaus = muistilista.annaEninMäärä() -
             muistilista.annaNykyinenMäärä();
if (1 <= vajaus)
  System.out.println("Listaan olisi mahtunut vielä " +
                     vajaus + " kohdetta.");
```

5-11 On erinäisiä asioita, mitä voidaan kuluttaa: voidaan kuluttaa energiaa, vaatteita, pinnaa ja paljon muutakin. Tässä tehtävässä mallinnamme kulutustapahtumaa. Esimerkiksi pinnan kulutus loppuu, kun se katkeaa. Keskittykäämme kuitenkin ennen muuta energian kulutukseen: energiaa voidaan kuluttaa, kunnes sitä ei enää ole jäljellä.

Energian kuluttaja on jonkin työn tekijä, kuten sähkömoottori tai vaikka sitten kauramoottori. Energian kuluttajalla on siis varastoituna energiaa, jota kulutetaan. Muita energian kuluttajaan liitettäviä yleisiä ominaisuuksia ovat teho, jolla energiaa

käytetään, sekä hyötysuhde.

Mieti mitä rajapintoja, abstrakteja luokkia ja konkreetteja luokkia tarvitset. Piirrä luokkakaavio mallintaen edellä mainittuja asioita, ja toteuta kulutukseen liittyvät keskeiset metodit kyseisissä luokissa. *Vinkki: tehtävässä mallinnetaan kulutustapahtumaa, ei kuluttajia.*

5-12 Oletetaan että olemme tekemässä järjestelmää, jossa halutaan välttää viimeiseen asti sellaisten lukujen syöttämistä, joita ei kenties tarvitakaan tulosten laskennassa. Näin käy esimerkiksi silloin kun käyttäjän valinnat johtavat laskennan johonkin ”erikoistilanteeseen”: jos laskentalauseena on

```
tulos = (kerroin != 0) ? (kerroin * syöteluku) : -100;
```

ja `kerroin == 0`, muuttujan `syöteluku` arvoa ei tarvita.

Tämä järjestelmävaatimus voidaan mallintaa ”laiskan numeron” avulla: tehdään oma numero-olio, joka huolehtii syötteen pyytämisestä tarvittaessa. Javassa olion numerokäyttäytyminen voidaan määritellä perimällä luokka `java.lang.Number`. Laiskan numero-oliomme on syytä olla mutatoitumaton, joten sillä on oltava vähintäin seuraava abstraktin luokan `AbstraktiLuku` toiminnallisuus.

```
public abstract class AbstraktiLuku extends Number {
   /** Mahdollisesti annettu luku. Annettu jos onTiedossa == true. */
   private double luku;

   /** Onko luku jo annettu? */
   private boolean onTiedossa;

   /** @.pre  true
    *  @.post ! onTiedossa() */
   protected AbstraktiLuku() { onTiedossa = false; }

   /** @.pre  true
    *  @.post RESULT == (onko luku jo annettu) */
   protected boolean onTiedossa() { return onTiedossa; }

   /** @.pre  ! onTiedossa()
    *  @.post onTiedossa() & (annaLuku() == luku) */
   protected void asetaLuku(double luku) {
      onTiedossa = true;
      this.luku = luku;
      }

   /** @.pre  onTiedossa()
    *  @.post RESULT == (annettu luku) */
   protected double annaLuku() { return luku; }
```

```
//--- java.lang.Number -piirteet

public byte   byteValue()   { return (byte) annaLuku(); }
public double doubleValue() { return        annaLuku(); }
public float  floatValue()  { return (float)annaLuku(); }
public int    intValue()    { return (int)  annaLuku(); }
public long   longValue()   { return (long) annaLuku(); }
public short  shortValue()  { return (short)annaLuku(); }
} //AbstraktiLuku
```

Toteuta tämän pohjalta luokka `KysyttäväLuku`, joka tuottaa ohjelmakoodilla

```
Number[] lukuja = {
   new Integer(13),
   new Double(-36.5),
   new KysyttäväLuku(new java.util.Scanner(System.in), System.out,
                     "Tarvitaan liukuluku: "),
   new KysyttäväLuku(new java.util.Scanner(System.in), System.out,
                     "Tarvitaan toinenkin liukuluku: ")
   };

double summa = 0.0;
for ( Number n : lukuja ) {
   summa += n.doubleValue();
   System.out.println("summattava " + n.doubleValue() + "\n" +
                      "== summa " + summa);
   }
```

seuraavan ajotulostuksen:

```
summattava 13.0
== summa 13.0
summattava -36.5
== summa -23.5
Tarvitaan liukuluku: 100.0
summattava 100.0
== summa 76.5
Tarvitaan toinenkin liukuluku: 2000.0
summattava 2000.0
== summa 2076.5
```

5-13 Laajenna listauksessa 5.5 (s. 150) annettua `Asiantuntija`-luokkaa rutiinilla

```
public Lemmikki annaNuorin(Class<? extends Lemmikki> laji)
```

Rutiini palauttaa parametrina annetun eläinlajin nuorimman edustajan. Laadi rutiinille sopiva määrittely ja toteuta se.

5-14 Kuvassa 5.4 (s. 152) on esitetty yksi tapa erottaa olio ja sen graafinen ilmiasu toisistaan. Vastaa tämän pohjalta seuraaviin kohtiin.

(a) Piirrä oliokaavio tilanteesta, jossa muuttuja `gt` viittaa `GraafinenTaulukko2<T>`-olioon.

(b) Keksi jokin järjestelmän laajennettavuuteen liittyvä syy sille miksi konkreetilla luokalla `GraafinenTaulukko2<T>` on sekä asiakas- että periytymissuhde rajapintaan `Taulukkomainen2<T>`.

(c) Oletetaan että kuvan esittämä ratkaisu on jo olemassa, mutta se osoittautuukin jostain syystä ongelmalliseksi: enää ei halutakaan tilannetta jossa oma taulukko ja sen ilmiasu synnyttää kaksi erillistä oliota. Kuinka muuttaisit taulukko/ilmiasu-rakennelman yksiolioiseksi, jos olemassaolevaan ohjelmakoodiin halutaan kajota mahdollisimman vähän?

Tutustu `java.util`-paketin luokkaan `Vector` ja sen perilliseen `Stack`. Mieti jokin perustelu, joka puoltaa tätä periytymistä. Entä missä tilanteessa tämä periytyminen johtaa ongelmaan?

5-15 (a) Kuvaile periytymisen yleisimmät käyttötavat. Mitkä käyttötavoista ovat helpommin ”hyväksyttäviä” ja mitkä eivät? Miksi?

(b) Tarkastele alla olevien luokkien ja rajapintojen välisiä suhteita. Piirrä hierarkiaa kuvaava luokkakaavio ja kerro, millä kohdassa (15a) mainituilla tavoilla periytymistä on hyödynnetty.

```
public interface OrderedNode {
   /** Palauttaa solmun järjestysnumeron sen sisarusten joukossa. */
   public int getOrder();
   }

public abstract class XMLNode {
   /** Solmun nimi. */
   private String name;

   /** Solmun vanhempisolmu. */
   private XMLElementNode parent;

   /** Alustaa solmun. */
   public XMLNode(String name, XMLElementNode parent) {
      this.name = name;
      this.parent = parent;
      }
```

```
   /** Palauttaa solmun nimen. */
   public String getName() { return name; }

   /** Palauttaa solmun vanhempisolmun. */
   public XMLElementNode getParent() { return parent; }

   /** Palauttaa solmun elementtipolun. */
   public String getPath() {
      String suffix = "/" + getName();
      return ((getParent() == null) ? "" : getParent().getPath())
                                          + suffix;
      }
   }

public class XMLElementNode extends XMLNode implements OrderedNode {
   /** Solmun järjestysnumero. */
   private int order;

   /** Alustaa solmun. */
   public XMLElementNode(String name, XMLElementNode parent,
                         int order) {
      super(name, parent);
      this.order = order;
      }

   /** Palauttaa solmun järjestysnumeron sen sisarusten
    *  joukossa. */
   public int getOrder() { return order; }
   }

public abstract class XMLValuedNode extends XMLNode {
   /** Solmun arvo. */
   private String value;

   /** Alustaa solmun. @.pre parent != null */
   public XMLValuedNode(String name, XMLElementNode parent,
                        String value) {
      super(name, parent);
      this.value = value;
      }

   /** Palauttaa solmun arvon. */
   public String getValue() { return value; }
   }
```

```
public class XMLAttributeNode extends XMLValuedNode {
   /** Alustaa solmun. @.pre parent != null */
   public XMLAttributeNode(String name, XMLElementNode parent,
                           String value) {
      super(name, parent, value);
      }

   /** Palauttaa solmun elementtipolun. */
   public String getPath() { return getParent().getPath()
                                    + "/@" + getName(); }
   }

public class XMLTextNode extends XMLValuedNode implements OrderedNode {
   /** Solmun järjestysnumero. */
   private int order;

   /** Alustaa solmun. @.pre parent != null */
   public XMLTextNode(XMLElementNode parent, String value, int order) {
      super(parent.getName(), parent, value);
      this.order = order;
      }

   /** Palauttaa solmun elementtipolun. */
   public String getPath() { return getParent().getPath(); }

   /** Palauttaa solmun järjestysnumeron sen sisarusten joukossa. */
   public int getOrder() { return order; }
   }
```

5-16 Katsotaanpa kuinka käy, kun periytymisen käyttäminen ns. karkaa käsistä (*eli näin sitten tehdään* vain *hupailumielessä*). Ohessa on esitelty erilaisia asuntoja kuvaavien luokkien konstruktorit sekä niiden käyttämät metodit ja jäsenmuuttujat. Mitä tapahtuu ja missä järjestyksessä, kun

(a) luodaan `ErikoisenTavallinenYksiö`-olio?

(b) luodaan `Opiskelijaboksi`-olio?

(c) järjestelmää laajennetaan `Piparkakkutalo`-luokalla, jonka halutaan periytyvän luokasta `AinutlaatuinenAsunto`?

```
public abstract class Asunto {
   private String talotyyppi;
   private String kuvaus;
   private Integer huonemäärä;
```

```
   /** @.pre  true
    *  @.post (alustaa yksiön standarditaloon) */
   public Asunto() {
      asetaOminaisuudet("standarditalo", 1);
      asetaKuvaus();
      }

   /** @.pre  tt != null & hm >= 1
    *  @.post (alustaa hm-huoneisen tt-tyyppisen asunnon) */
   public Asunto(String tt, Integer hm) {
      asetaOminaisuudet(tt, hm);
      asetaKuvaus();
      }

   /** @.pre  tt != null & hm >= 1
    *  @.post (asettaa huonemääräksi hm:n ja talotyypiksi tt:n) */
   protected void asetaOminaisuudet(String tt, Integer hm) {
      talotyyppi = tt;
      huonemäärä = hm;
      }

   /** @.pre  (proseduuria asetaOminaisuudet(String,Integer)
    *          on kutsuttu)
    *  @.postPrivate (asettaa kuvaustekstin jäsenmuuttujaan kuvaus) */
   protected void asetaKuvaus() {
      if ( huonemäärä > 2 )
         kuvaus = talotyyppi + ", " + huonemäärä + " huonetta";
      else {
         String[] huoneisto = { "nollio", "yksiö", "kaksio" };
         kuvaus = huoneisto[huonemäärä] + " " + talotyyppi + "ssa";
       }
      }
   //.........
   }

public class Kerrostaloasunto extends Asunto {
   private Integer kerros;

   /** @.pre  hm >= 1 & krs >= 1
    *  @.post (alustaa hm-huoneisen kerrostaloasunnon kerrokseen krs) */
   public Kerrostaloasunto(Integer hm, Integer krs) {
      super("kerrostalo", hm);
      kerros = krs;
      }
```

```
   //.........
   }

public class Opiskelijaboksi extends Kerrostaloasunto {
   private String sijainti;
   private Integer talonumero;

   /** @.pre  krs >= 1 & s != null & nro >= 1
    *  @.post (alustaa paikassa s sijaitsevan yksiön,
    *         jonka talonumero on nro ja kerros krs) */
   public Opiskelijaboksi(Integer krs, String s, Integer nro) {
      super(1, krs);
      asetaOminaisuudet(s, nro);
      }

   /** @.pre  s != null & nro >= 1
    *  @.post (asettaa sijainniksi s:n ja talonumeroksi rno:n) */
   protected void asetaOminaisuudet(String s, Integer nro) {
      sijainti = s;
      talonumero = nro;
      }
   //.........
   }

public class TavallinenYksiö extends Asunto {
   //.........
   }

public class ErikoisenTavallinenYksiö extends TavallinenYksiö {
   private Integer tavallisuusaste;

   /** @.pre  ta >= 1 & ta <= 5
    *  @.post (alustaa yksiön, jonka tavallisuusaste on ta) */
   public ErikoisenTavallinenYksiö(Integer ta) {
      tavallisuusaste = ta;
      }
   //.........
   }

public class AinutlaatuinenAsunto extends Asunto {
   public static AinutlaatuinenAsunto asunto =
                                 new AinutlaatuinenAsunto(13);

   /** @.pre  hm >= 1
    *  @.post (alustaa hm-huoneisen asunnon) */
```

```
   private AinutlaatuinenAsunto(Integer hm) {
      super("Ainutlaatuinen ihmeasuntoque", hm);
      }
//.........
}
```

Luku 6

Olion perustoiminnoista

Jokainen Java-olio osaa perustoiminnot, jotka ovat peräisin `Object`-luokasta. Näiden toimintojen semantiikan ymmärtäminen on olennaista muodostettaessa uutta luokkaa ja arvioitaessa perittävien piirteiden riittävyyttä uuden luokan tarpeisiin. Koska joidenkin peruspiirteiden toteutus on Javassa hieman kimuranttia, tarkastellaan tässä luvussa myös korvaavan rutiinin toteutukseen liittyviä yksityiskohtia, jotta ohjelmoija osaisi välttää ainakin pahimmat karikot. Samalla valotetaan piirteiden välisiä riippuvuuksia.

Perustoiminnot osaavasta oliosta käytetään usein nimitystä *kanoninen olio* (*canonical object*), joka kuvastaa sitä, että jokaisen olion tulisi olla järkevästi ja odotusten mukaisesti käyttäytyvä näiden operaatioiden osalta. Paitsi `Object`-luokalta perityt piirteet, kunkin olion on hyvä osata myös muita perustaitoja: vertailuoperaatiot, oliotietojen linearisointi (esim. tiedonsiirtoa varten) sekä luokkainvariantin tarkistava funktio. Mutta ennen niitä, tarkastellaan ehkä tärkeintä olioon liittyvää operaatiota, olion luontia.

6.1 Alustus

Jo aiemmin on havaittu, miten erilainen luokan luontioperaatio on suhteessa muihin rutiineihin:

- konstruktorin nimen tulee aina olla tarkalleen sama kuin luokan nimi (tälle Java-kielen säännölle ei ole mitään sen kummempaa järkiperustelua),
- konstruktorilla ei ole palautusarvoa (ei signatuurissaan edes **`void`**-määrettä),
- konstruktoria voi kutsua vain **`new`**-operaattorin yhteydessä, eikä sitä voi kohdistaa normaalia pistenotaatiota käyttäen jo luotuun olioon,

- konstruktori ei periydy (eikä sitä siis voi korvata aliluokissa), ja
- konstruktorilla on oletuksena sama suojausmääre kuin luokalla.

Luontioperaatioita voidaan ylikuormittaa normaalisti. Jos luokassa ei ole esitelty ainoatakaan konstruktoria, Java-systeemi suorittaa luonnin yhteydessä automaattisesti *oletuskonstruktorin* (*default constructor*), jolla ei ole argumentteja ja jonka rungossa ei ole yhtään suoritettava käskyä. Näin ei tehdä, jos yksikin konstruktori on esitelty. Konstruktorin tarkoituksena on alustaa olion sisäinen esitysmuoto niin, että luokkainvariantti tulee voimaan. Mikäli jäsenmuuttujien oletusalustukset riittävät tähän, konstruktori on tarpeeton — vaikkakin sen toteuttaminen on aina suositeltavaa.

Normaali tapa alustaa luokan jäsenmuuttujat luonnin yhteydessä on kirjoittaa alustuskäskyt konstruktorin runkoon. Alustusarvo voidaan kuitenkin kirjoittaa myös suoraan jäsenmuuttujan esittelyn yhteyteen:

```
public class Koira extends Lemmikki
  {
  private boolean veroMaksettu = false;
  }
```

Asetuslauseen oikealla puolella saa olla lauseke, jossa voi käyttää kaikkia jo alustettuja tietoja (alustus tehdään siinä järjestyksessä, johon esittelyt on kirjoitettu). Myös viittaustyyppiset jäsenmuuttujat voidaan alustaa luomalla kyseistä tyyppiä oleva olio. Itse asiassa tunnisteen esittelyn yhteydessä voi kutsua mitä tahansa funktiota, joka palauttaa ko. tyyppiä olevan arvon. Tällä funktiolla voi olla argumenttina jokin sellainen tieto, joka on jo alustettu.

Staattiset jäsenmuuttujat alustetaan vain kerran, silloin kun luokkamäärittely (eli luokan class-tiedosto) ladataan ajoympäristöön. Tällaisten tunnisteiden tapauksessa on mahdollista käyttää myös *staattista lauselohkoa* (*static block*):

```
public class Pulju
  {
  static double alv;
  static Työntekijä myyntijohtaja;
  static
    {
    alv = 22.0;
    myyntijohtaja = new Työntekijä("Lennart Nilkén");
    }
  }
```

Lauselohkokin suoritetaan vain kun luokkamäärittely ladataan. Samanlaista lohkoa voidaan käyttää myös ei-staattisille tunnisteille:

```
public class Neliö
  {
  private Tasopiste p1, p2;
    {
    p1 = new Tasopiste(0.0, 0.0);
    p2 = new Tasopiste(1.0, 1.0);
    }
  }
```

Kuten näkyy, tämä on syntaktisesti jo hyvin lähellä nimetöntä luokkaa.

Kun oliolle on varattu muistitila, jäsenmuuttujat on alustettu oletusarvoilla ja esittelyjen yhteyteen kirjoitetut toimenpiteet tehty, aloitetaan konstruktorin suoritus. Periytyminen tuo tähän prosessiin oman lisämausteensa, joka liittyy perityissä luokissa olevien jäsenmuuttujien alustukseen. Jotta näillä jäsenmuuttujilla olisi järkevät alkuarvot, konstruktorikutsu aiheuttaa rekursiivisen luontiketjun: ennen konstruktorin suoritusta kutsutaan yliluokan (sen ainoan, joka ei ole rajapinta) konstruktoria, mikäli tällainen yliluokka on olemassa. Jos ei ole, rekursiossa on edetty luokkaan `Object` (rekursion perustapaus) ja sen luontioperaatio suoritetaan. Tämän jälkeen palataan rekursioketjussa takaisin saattaen kunkin luokan konstruktorin suoritus loppuun. Jos jokin luokka ei tässä ketjussa eksplisiittisesti kutsu yliluokan konstruktoria, ajoaikainen systeemi kutsuu oletuskonstruktoria automaattisesti. Jos tällaisessa tilanteessa löydetään luokka, johon ei ole kirjoitettu oletuskonstruktoria (mutta muita konstruktoreita kyllä), kääntäjä antaa virheilmoituksen.

Nyt voidaan täsmentää luonnin yhteydessä tapahtuvien toimintojen suoritusjärjestys:

1. Operaattori **new** varaa keskusmuistista tilaa olion kaikille jäsenmuuttujille, myös yliluokissa esitellyille, olipa niiden suojausmääre mikä tahansa.

2. Kullekin jäsenmuuttujalle annetaan oletusarvo (**int**-tiedolle nolla, viittaukselle **null**, totuusarvolle **false** jne.).

3. Yliluokan konstruktoria kutsutaan rekursiivisesti.

4. Nykyisen luokan jäsenmuuttujat alustetaan järjestyksessä (edeten ylhäältä alaspäin luokkatekstissä) esittelyn yhteyteen merkityllä tavalla.

5. Nykyisen luokan konstruktorirunko suoritetaan.

Tällä menettelyllä taataan luokkainvarianttien voimassaolo kullakin luokkatasolla ennen seuraavan (alemman) tason konstruktorin suoritusta. Huomaa myös että tämän vuoksi konstruktorissa ei saa koskaan kutsua dynaamisesti sidottua rutiinia!

6.2 Pinta- ja syväoperaatioista

Oliokielet tarjoavat normaalisti kolme eri tapaa olioiden samanlaisuuden testaamiseen. Nämä tavat liittyvät tasoihin, jolla asiaa tarkastellaan:

Identtisyys Jos tunnisteet `a` ja `b` viittaavat samaan olioon, ehto `a == b` on tosi. Tällöin puhutaan viittausten identtisyydestä. Tämä samuuden aste on kaikkein tiukin siinä mielessä, että tunnisteille, joille yhtäsuuruusehto on tosi, myös muut vertailut palauttavat arvon tosi.

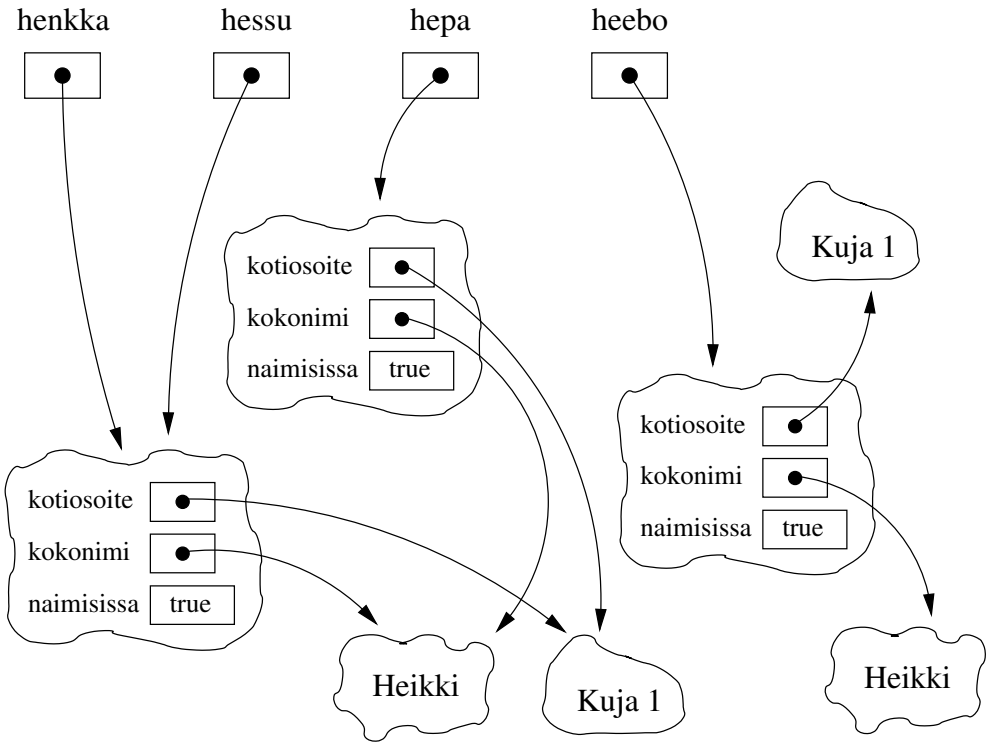
Pintasamuus Tunnisteiden `a` ja `b` viittaamat oliot ovat *pintasamoja* (*shallow equal*), jos olioiden jäsenmuuttujat sisältävät identtiset arvot. Viittaustyyppisille jäsenmuuttujille se tarkoittaa, että ne viittaavat samaan olioon. Jos oliot ovat pintasamoja, niiden on pakko olla myös syväsamoja.

Syväsamuus Tunnisteiden `a` ja `b` viittaamat oliot ovat *syväsamoja* (*deep equal*), jos rakenteet, joihin ne viittaavat, ovat paitsi pinnalta myös muualta samat. Tämä asia voidaan ilmaista myös toisin: kulkemalla kumpikin oliorakenne läpi primitiivityyppien mukaisissa vastintiedoissa on oltava samat arvot. Kolmas tapa: oliot `a` ja `b` ovat syväsamat, jos ne ovat pintasamat ja kukin niiden viittaustyyppisistä vastinjäsenmuuttujista on rekursiivisesti pintasamat.

Esimerkki 6.1 Kuva 6.1 havainnollistaa näitä eroja: `henkka` ja `hessu` ovat identtisesti samat, `hessu` ja `hepa` pintasamat (mutta eivät identtisesti) ja `hepa` ja `heebo` syväsamat (mutta eivät pintasamoja tai identtisiä).

Kutakin samuuden astetta kohti oliokielissä on yleensä operaatio, joka suorittaa vastaavan kopioinnin. Esimerkiksi, jos oliosta tehdään pintakopio, se on pintasama alkuperäisen olion kanssa. Yleisesti käytössä oleva operaatio ”identtisen kopion” tekemiseen on asetuslause: kuvan 6.1 tilanne on voinut syntyä asetuksen `henkka = hessu;` johdosta. Kysymys on tietysti pelkästään *olion jakamisesta* (*object sharing* tai *aliasing*), mutta tämä ei tarkoita sitä että tilanne olisi mitenkään yksinkertainen: Identtisyys on ainoa samuuden muoto, jonka tulkinta on olion asiakkaalle täysin ”havaittavissa”. Mikäli samuus määritellään jäsenmuuttujien avulla, asiakas ei voi tarkastella niitä. Asiakas pystyy kyllä havaitsemaan olion attribuutit, mutta jos ne ovat johdettuja, niillä ei välttämättä ole yksikäsitteistä jäsenmuuttujavastinetta. Tähän ongelmaan on olemassa ainakin osittaisia ratkaisuja, mutta niihin ei syvennytä tässä sen tarkemmin.

Javan `equals` ja `clone` liittyvät läheisesti toisiinsa, sillä Java-dokumentoinnin mukaan ehdon `x.clone().equals(x)` tulisi olla voimassa joskaan se ei ole aivan välttämätöntä (ks. `clone`-rutiinin määrittely). Myös `equals` määritellään näennäisen vahvasti, mutta selvää kantaa siihen, ovatko ne pinta- vai syväoperaatiota ei oteta



Kuva 6.1: Samanlaisuuden asteet.

missään. Tärkein lähde asian tulkitsemiseksi on Java-kielen määrittelyksikin mainittu kirja [1], joka sanoo: ”kopioinnin jälkeen uuteen kopioon tehtävät muutokset eivät saa vaikuttaa alkuperäisen olion tilaan”. Tämän perusteella Javan `clone` ja `equals` ovat syväoperaatioita.[1] Käytännössä vastaan tulee kuitenkin helposti tilanne, jossa kopiointia ei voida viedä aivan loppuun asti (tästä lisää kohdassa 6.4), joten sanottakoon, että kyseessä on syvähköt operaatiot. Niin kuin tämä ei olisi tarpeeksi, `equals`- ja `clone`-operaatioiden korvauksiin liittyy vielä runsaasti yksityiskohtia, joista ohjelmoijan tulee olla tietoinen saadakseen oikein käyttäytyvät toteutukset. Monet Javan omistakin luokista käyttävät `equals`-operaatiota, joten jos ohjelmoija ei osaa kirjoittaa määrittelyn mukaista toteutusta, sillä on melko laajalle ulottuvia seurauksia myös peruskirjastoluokkien käytössä.

[1] Sama lähde mainitsee myöhemmin (s. 94): ”The default implementation of clone (as given by `Object`) provides what is known as a shallow clone or copy — it simply performs a field by field copy. A deep clone would clone each object referred to by a field and each entry in an array. This would apply recursively and so deep cloning an object would clone all of the objects reachable from that object. In general `clone` is overridden to perform a deeper clone, whenever a shallow clone is not appropriate.”

6.3 `equals`

Seuraavaksi on tarkoitus selvittää `equals`-operaation toteutukseen liittyviä ongelmia. Ennen sitä muistutetaan kuitenkin tilanteista, jolloin `equals`-operaatiota *ei tarvitse* toteuttaa uudelleen, vaan `Object`-luokan oletustoteutus, joka käyttää identtisyystestausta(!), on riittävä:

- Luokasta luodut oliot ovat mutatoitumattomia. Tällöin identtisyys- ja pintasamuustestit palauttavat aina saman totuusarvon.
- Pintasamuustesti ei ole järkevä luokan olioille. Esimerkiksi `Random`-luokka, joka tuottaa satunnaislukuja, ei tarvitse `equals`-testiä (joka tarkistaisi, että generoidut satunnaislukujonot ovat samat).
- Yliluokalta peritty `equals` toimii perijässä sellaisenaan (ks. esim. `AbstractSet` ja `Set`).
- Luokka on **`private`**- tai *package*-suojattu, jolloin sen käyttö on rajattu ja toteuttaja tietää, että sen `equals`-operaatiota ei kutsuta ko. ympäristössä.

Jos uusi toteutus halutaan, asiaa on syytä lähteä kerimään `Object`-luokan dokumentoinnista, joka sanelee `equals`-operaatiolle seuraavat ehdot:

(i) Rutiinin on oltava *refleksiivinen*: ehdon `x.equals(x)` on oltava tosi kaikille olioille `x`.

(ii) Rutiinin on oltava *symmetrinen*: ehdon `x.equals(y)` pitää palauttaa arvo tosi silloin ja vain silloin kun `y.equals(x)` palauttaa arvon tosi.

(iii) Rutiinin on oltava *transitiivinen*: jos `x.equals(y)` on tosi ja `y.equals(z)` on tosi, myös ehdon `x.equals(z)` on oltava tosi.

(iv) Rutiinin on oltava *konsistentti*: ehtojen `x.equals(y)` ja `y.equals(x)` on palautettava johdonmukaisesti tosi tai epätosi edellyttäen, että pintasamuuteen liittyviä tietoja ei muuteta.

(v) Testin `x.equals(null)` tulee palauttaa arvo epätosi.

(vi) Rutiinin on toimittava, vaikka vertailuun osallistuisi erityyppisiä olioita, kuten luokasta ja sen perijästä luotuja olioita.

Vaatimukset tuntuvat intuitiivisesti hyvin selviltä; näinhän vertailuoperaation tulee käyttäytyäkin. `Object`-luokalta perittävä toteutus testaa olioiden identtisyyttä (tarkemmin sanoen olioiden muistiosoitteita) jättäen olioiden merkityksen toisarvoiseksi. Seuraavassa on esitetty kyseenalainen ”standarditoteutus” korvaavalle `equals`-operaatiolle luokkaan `Henkilö`:

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!(toinen instanceof Henkilö)) return false;
  Henkilö kaveri = (Henkilö)toinen;
  return nimi.equals(kaveri.nimi) &
         osoite.equals(kaveri.osoite) &
         syntymävuosi == kaveri.syntymävuosi;
  }
```

Ensimmäinen **if**-testi toteuttaa ehdon (v). Tähän riittäisi pelkkä **instanceof**-testikin, mutta selvyyden vuoksi **null**-testi on tehty erikseen. Toisen **if**-lauseen identtisyystesti on mukana lähinnä tehokkuussyistä. Toteutus käyttää hyväkseen Java-sääntöä, jonka mukaan saman luokan olioilla on vapaa pääsy toistensa jäsenmuuttujiin, olivatpa niiden suojausmääreet mitkä tahansa. Tämä on standarditapa, jota ei **instanceof**-komentoa käytettäessä saa missään nimessä korvata vastaavilla ja mahdollisesti dynaamisesti sitoutuvilla `anna`-operaatiokutsuilla. Miksiköhän?[2]

Koska syväsamuuden testaavassa rutiinissa delegoidaan tehtävä työ pääosin jäsenmuuttujien harteille soveltamalla niiden `equals`-operaatioita, pitää niillekin olla oma (järkevästi toimiva) toteutuksensa. Tämäkin korostaa sitä, että `equals` on osa kanonista oliota: se on toiminto, jonka jokaisen olion tulisi osata tehdä oikein.

Vaikka ehto (vi) on yksi tärkeimmistä ehdoista ja hyvin yleinen eri oliokielille, se ei ilmene suoraan Java-dokumentoinnista. Se osoittautuukin vaikeaksi toteuttaa ja voidaan sanoa, että yllä olevan kaavan mukainen `equals`-operaation toteutus toimii vain, jos vertailuun osallistuvat oliot ovat tarkalleen samasta luokasta. Tämä nähdään seuraavasta esimerkistä (modifioitu lähteestä [2]).

Esimerkki 6.2 Oletetaan, että listauksessa 3.1 esitettyä koordinaatistopistettä kuvaavaan luokkaan olisi kirjoitettu

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!(toinen instanceof TasoPiste)) return false;
  Tasopiste p = (Tasopiste) toinen;
  return this.x == p.x & this.y == p.y;
  }
```

Aivan järkevä toteutus, vai mitä? No, tehdäänpä luokalle perijä

[2]Vinkki: Mieti miten perilliset käyttävät **super**-tunnistetta.

```
public class Väripiste extends Tasopiste
  {
  private Color väri;

  public Väripiste(double x, double y, Color v)
    {
    super(x, y);
    väri = v;
    }
  }
```

joka mallintaa siis värillistä pistettä (luokka `Color` löytyy pakkauksesta `java.awt`). Tämän luokan `equals`-toteutuksessa tulee tietysti tarkistaa, että pisteiden paikka ja väri ovat samat:

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!(toinen instanceof VäriPiste)) return false;
  Väripiste vp = (Väripiste) toinen;
  return super.equals(toinen) & vp.väri == väri;
  }
```

Vaikka tämäkin toteutus on mallikelpoinen, symmetrisyysehto (ii) ei enää täyty, sillä normaalipisteen vertailu värilliseen ei ota huomioon väriä ja päinvastainen vertailu saa aina arvon **false**, koska rutiinin argumentin tyyppi on väärä. Jos esimerkiksi esitellään

```
Tasopiste tp = new Tasopiste(1.0, 2.0);
Väripiste vp = new Väripiste(1.0, 2.0, Color.RED);
```

niin `tp.equals(vp)` on tosi ja `vp.equals(tp)` epätosi. Toki väripisteen pintasamuuden testauksessa voidaan tarkistaa ”sekatestitilanne” ja jättää väri tällöin tarkistamatta:

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!(toinen instanceof Väripiste)) return toinen.equals(this);
  Väripiste vp = (Väripiste) toinen;
  return super.equals(toinen) & vp.väri == väri;
  }
```

Tämä korjaa symmetrisyysongelman, mutta ei täytä enää transitiivisuusehtoa (iii). Jos nimittäin esitellään

```
Väripiste p1 = new Väripiste(1.0, 2.0, Color.RED);
Tasopiste p2 = new Tasopiste(1.0, 2.0);
Väripiste p3 = new Väripiste(1.0, 2.0, Color.BLUE);
```

ehdot `p1.equals(p2)` ja `p2.equals(p3)` ovat tosia, koska ne eivät huomioi väriä. Yhtäsuuruustesti `p1.equals(p3)` huomioi värin ja palauttaa arvon epätosi, joten ehto (iii) ei ole voimassa.

Lopputulos tästä analyysistä: *ei ole mahdollista luoda systeemiä, missä perijäluokka tuo mukanaan uuden jäsenmuuttujan ja säilyttää silti* `equals`*-operaation määrittelyn.* Asia voidaan kyllä kiertää, mutta perinnästä saatavien hyötyjen kustannuksella tekemällä luokasta `Väripiste` luokkien `Tasopiste` ja `Color` asiakas:

```
public class Väripiste
  {
  private Tasopiste piste;
  private Color väri;

  public Väripiste(double x, double y, Color v)
    {
    piste = new TasoPiste(x, y);
    väri = v;
    }

  public Tasopiste annaPiste() { return piste.clone(); }

  public boolean equals(Object toinen)
    {
    if (toinen == null) return false;
    if (toinen == this) return true;
    if (!(toinen instanceof Väripiste)) return false;
    Väripiste vp = (Väripiste) toinen;
    return vp.piste.equals(piste) & vp.väri.equals(väri);
    }
  }
```

On tietysti aika yllättävää, että niinkin keskeinen operaatio kuin `equals` tuottaa näin paljon ongelmia toteuttajalle. Onneksi käytännössä tulee aika harvoin vastaan tilanteita, joissa vertaillaan sekaisin kahta eri tyyppistä oliota ja näiltä ongelmilta vältytään. Jos kuitenkin ajoaikana sattuu käymään niin, ohjelmoijan saattaa olla erittäin vaikeata ymmärtää systeemin outoa käyttäytymistä, mikäli näitä asioita ei tunne. Sitäpaitsi esimerkiksi kaikissa kokoelmaluokissa on `contains`-operaatio, joka hakee argumenttina annettua alkiota käyttäen juuri `equals`-operaatiota. Mikäli kokoelman alkiotyyppi on `Object`, sekavertailuja voi sattua. Kyllä `equals` näyttelee hyvin merkittävää roolia koko olioparadigmassa...

Mitenkäpähän sitten `equals`-predikaatti pitäisi toteuttaa? Jollei ole erityisen hyvää syytä, ehto (vi) kannattaa ottaa huomioon palauttamalla **`false`** erityyppisille olioille. Toisin sanoen `Henkilö`-luokan `equals` olisi tällöin seuraava:

```
public boolean equals(Object toinen)
  {
  // Tämä equals toimii kantamäärittelynä tämän periytymis-
  // haaran sellaiselle equals-samuudelle, joka ei vertaile
  // identtisyyttä. Toisin sanoen tässä rutiinissa super.clone()
  // viittaa Object-luokan toteutukseen.
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (getClass() != toinen.getClass()) return false;
  Henkilö kaveri = (Henkilö)toinen;
  return nimi.equals(kaveri.nimi) &
         osoite.equals(kaveri.osoite) &
         syntymävuosi == kaveri.syntymävuosi;
  }
```

Mutta ei tässä vielä kaikki: `Henkilö`-luokan `equals` toimii kyllä, mutta vain koska se on `Object`-luokan välitön perillinen. Entä jos `Työntekijä`-luokka perii `Henkilö`-luokan; miten sen `equals` toteutetaan?

```
public boolean equals(Object toinen)
  {
  // Tämä ei saa kutsua Object-luokan equals-toteutusta.
  if (!super.equals(toinen)) return false;

  Työntekijä toveri = (Työntekijä)toinen;
  return palkka == toveri.palkka;
  }
```

Ja vielä yksi muistutus: `equals`-samuus pitää ymmärtää *semanttisena samanlaisuutena*, joten standarditapa delegoida asia viittaustyyppisille jäsenmuuttujille ei toimi kaikissa tapauksissa. Esimerkiksi `LukuJoukko`-luokan (ks. kohta 3.1) asiakas näkee asian niin, että kokonaislukujoukot ovat samat, jos niissä on samat alkiot. Koska `LukuJoukko` tallettaa joukon alkiot listaan, eikä pidä niitä missään tietyssä järjestyksessä, testiä ei voi delegoida `ArrayList`-luokan `equals`-operaatiolle (joka vaatii, että listat ovat yhtä pitkät ja että kussakin listapositiossa on viittaus samaan olioon), vaan tarkastella asiaa abstraktin joukkokäsitteen kannalta:

```
public boolean equals(Object toinen)
  {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (getClass() != (toinen.getClass()) return false;
  LukuJoukko vertailtava = (LukuJoukko)toinen;
  return this.sisältää(vertailtava) & vertailtava.sisältää(this);
```

```
}
```

Toinen vaihtoehto olisi jäsenmuuttujana olevien rakenteiden normalisointi yksikäsitteiseen muotoon ennen ko. rakenneolion `equals`-kutsua.

Jos ja kun pintasamuuden vertailuun annetaan korvaava toteutus, on muistettava, että myös (a) `hashCode`-operaatio pitää korvata yhdenmukaiseksi ja (b) rajapinnan `Comparable` määrittelemän `compareTo`-metodin pitää olla yhtä mieltä yhtäsuuruustestiä tehtäessä. Näitä asioita selvitetään jatkossa.

6.4 clone

Jos `equals` on erikoinen, niin vielä mielenkiintoisempi (lue: oudompi) on `clone`. Jotta asiakas voisi kopioida esimerkiksi luokan `JangoFett` olioita, pitää (a) luokan `JangoFett` toteuttaa `Cloneable`-rajapinta ja (b) antaa `clone`-operaatiolle julkinen toteutus. Jostain syystä Java-suunnittelijat eivät ole halunneet `Cloneable`-rajapintaan `clone`-operaation määrittelyä, vaan rajapinta on tyhjä. Itse asiassa rajapinnan toteuttavan luokan otsikossa oleva **`implements`** `Cloneable` on vain leima, jonka olemassaolon `Object`-luokan `clone`-operaatio tarkistaa. Oletuksena `clone`-operaation suojausmääre on **`protected`** jostain kumman syystä.[3] Siispä jos perijäluokka `JangoFett` haluaa kloonauksen mukaan julkiseen liitäntäänsä, pitää kirjoittaa

```
public class JangoFett implements Cloneable
  {
  public JangoFett clone()
    {
    try
      {
      return (JangoFett)super.clone();
      }
    catch (CloneNotSupportedException e)
      {
      throw new InternalError(e.toString());
      }
    }
  }
```

Toteutus käyttää hyväkseen paluutyypin kovarianssia tiukentamalla `Object`-paluutyypin `JangoFett`-tyyppiseksi tyyppipakotuksella (*type cast*). Toteutus hoitaa myös

[3] Eckel [4] arvelee sen johtuvan siitä, että turvallisuusaspektit saattaisivat kärsiä pahasti, jos turvallisuuteen liittyviä olioita pääsisi joku kopioimaan. Tämä ei kuitenkaan selitä, miksi `clone`-operaatio ei voisi olla **`public`**-määritelty `Cloneable`-rajapinnassa.

operaation mahdollisesti nostaman poikkeuksen `CloneNotSupportedException`, joka syntyy, jos perijän otsikossa ei esiinny **`implements`** `Cloneable`. Aiemmin esitetyn nojalla tiedämme, että tämä ratkaisu on ”lopullinen” siinä mielessä, että luokan `JangoFett` perijöiden `clone` ei voi tämän jälkeen enää nostaa ko. poikkeusta. Se on kiusallista siinä tapauksessa, että luokan `JangoFett` perijä ei haluaisi antaa asiakkailleen kopiointimahdollisuutta. Toisin sanoen se mikä on julistettu olioliittymään kuuluvaksi, on ja pysyy olion ominaisuutena. Jos `clone`-metodin toteutus ei käsittele poikkeusta `CloneNotSupportedException`, kaikki asiakkaat joutuvat aina kirjoittamaan `clone`-kutsun **`try`**–**`catch`** -lohkoon tai **`throws`**-osan oman rutiininsa otsikkoon. Käytännön syistä lienee suositeltavampaa hoitaa poikkeus `clone`-rutiinissa, jolloin asiakkaiden ei tarvitse ottaa siihen kantaa.

Kopiointiprosessi toimii seuraavasti. Kun luokan `JangoFett` rutiini `clone` kutsuu yliluokan vastaavaa, sen tulisi tehdä sama omalle yliluokalleen jne. (vrt. olion luonti). Kun tässä ketjussa saavutaan lopulta `Object`-luokan perustoteutukseen, tarkistetaan ensin, että luokka `JangoFett` toteuttaa `Cloneable`-rajapinnan. Mikäli näin ei ole, nostetaan poikkeus `CloneNotSupportedException`. Muussa tapauksessa tehdään bitittäinen kopio *siitä oliosta, johon alkuperäinen* `clone` *kohdistui*. Miten `Object`-luokan `clone` tietää sen, on puhdasta magiaa (no, ehkä `clone`-kutsut välittävät sisäistä tyyppi-informaatiota toisilleen).

Peruskopiointi tarkoittaa siis pintakopiota: primitiivityypin tiedot kopioidaan sellaisenaan ja viittaustyyppisistä kopioidaan viittaukset. Tämä on riittävä toimenpide, jos luokan jäsenmuuttujat ovat kaikki primitiivityyppisiä tai viittauksia mutatoitumattomiin olioihin (esim. listauksen 3.1 `Tasopiste`). Muussa tapauksessa `clone`-rutiinissa on kopioitava ne viittausten päissä olevat oliot jotka osallistuvat `equals`-predikaattiin.

Esimerkki 6.3 `Asiantuntija`-luokan kopiointi tapahtuu seuraavasti:

```
public Asiantuntija clone()
  {
  try
    {
    Asiantuntija kopio = (Asiantuntija)super.clone();
    kopio.henkilötiedot = henkilötiedot.clone();
    kopio.hoidettavat = hoidettavat.clone();
    return kopio;
    }
  catch (CloneNotSupportedException e)
    {
    throw new InternalError(e.toString());
    }
  }
```

Huomaa, että jos pintakopio ei ole riittävä niin myös viittaukselliset jäsenmuuttujat on kloonattava, jotta kopio on todellakin eriytyy alkuperäisestä, ja että jäsenmuuttujat eivät saa olla **final**-tyyppisiä, koska se estäisi niiden arvon muuttamisen.

Javan kloonausoperaatio on edellä olevan esimerkin mukaisesti toteutettuna syväkopio, mutta käytäessä koko oliorakennetta läpi ja aina kun törmätään `Object`-tyyppiseen tietoon, kloonaus on pakko jättää tekemättä (tästä kohdan 6.2 termi ”syvähkö kopio”). Tästä syystä esimerkiksi kaikki Javan kokoelmaluokat (mukaanluettuna taulukkotyyppi (ks. luku 8) tekevät kopioinnin vain globaalin rakenteensa osalta, mutta jättävät varsinaiset alimman tason komponentit kloonaamatta, joten ne ovat operaation jälkeen yhteiset alkuperäiselle ja kopioidulle kokoelmalle. Kopioinnin suorittava asiakas on siis itse velvollinen huolehtimaan tarvittavista korjaustoimenpiteistä `clone`-kutsun jälkeen (asiakas tuntee olioiden todellisen tyypin).

Vaikka luokka ei haluaisikaan olioitaan kopioitavan, sen on silti syytä toteuttaa `clone`, jos pintakopio ei ole riittävä ko. luokan jäsenmuuttujille, koska jokin sen perijöistä saattaa haluta antaa kopiointioperaation asiakkailleen. Kuten luontioperaation yhteydessä, `clone`-kutsut etenevät aina `Object`-luokkaan asti, jonka seurauksena alkuperäisestä oliosta tehdään pintakopio. Tämän jälkeen kontrolli palaa kutsuketjua alaspäin ja jokainen luokka tekee tarvittavat lisätoimet viittaustyyppisten tietojen osalta. Vain silloin luokka, josta kutsu alunperin lähti, voi olla varma siitä, että tuloksena on todellinen kopio (poisluettuna em. `Object`-tyyppiset jäsenmuuttujat).

Luokan pitää aina palauttaa kutsujalleen olio, jonka se on saanut käsiinsä tekemällä kutsun **`super.clone()`**. Vain se takaa, että *palautettu kopio on perijän tyyppiä*. On helppo tehdä virhe, jossa yliluokan `clone` tekee kopion itsestään kutsumalla luokkansa konstruktoria ja palauttamalla sen perijälleen. Palautettu olio on kuitenkin yliluokan tyyppiä eikä voi siten edustaa perijäluokkaa (erikoistus perijäluokan tyypiksi ei toimi). Johtopäätös: kaikkien `clone`-toteutusten on aina kutsuttava yliluokan vastaavaa operaatiota ensimmäisenä toimenpiteenä. Toisin kuin luontioperaation tapauksessa, systeemi ei millään tavoin pakota tähän (tai generoi itse esimerkiksi oletuskopiointirutiinia), vaan se on kokonaan ohjelmoijan omalla vastuulla.

Toinen tapa on kopioida olio konstruktorien avulla. Tämä kuitenkin edellyttää, että luokkaan on määritelty *kopiokonstruktori* (*copy constructor*), joka tekee tämän mahdolliseksi. Tutkitaanpa kopiokonstrukoreita hieman lähemmin ja osoitetaan ettei sellaisista ole juurikaan hyötyä Javassa.

Esimerkki 6.4 Tarkastellaan luokkaa `Kolmio`, joka koostuu kolmesta `Tasopiste`-oliosta.

```
/**
 *  @.classinvariantPrivate
 *       Kärjet eivät saa olla samalla suoralla, eli kolmiolla
```

```
 *       on tilavuus: ei ole redusoitunut pisteeksi eikä janaksi.
 *       Ts. vektorit kärki1->kärki2 ja kärki1->kärki3 ovat
 *       lineaarisesti riippumattomat.
 */
public class Kolmio
  {
  private Tasopiste kärki1, kärki2, kärki3;

  /** Alustaa kolmion kärjiksi viittaukset k1, k2 ja k3. */
  public Kolmio(Tasopiste k1, Tasopiste k2, Tasopiste k3)
    { kärki1 = k1; kärki2 = k2; kärki3 = k3; }

  /** Palauttaa this:n kanssa pintasaman Kolmio-olion. */
  public Kolmio pintakopio()
    { return new Kolmio(kärki1, kärki2, kärki3); }

  /** Palauttaa this:n kanssa syväsaman Kolmio-olion. */
  public Kolmio syväkopio()
    {
    return new Kolmio(new Tasopiste(kärki1.annaX(), kärki1.annaY()),
                      new Tasopiste(kärki2.annaX(), kärki2.annaY()),
                      new Tasopiste(kärki3.annaX(), kärki3.annaY()));
    }
  } // Kolmio
```

Tämä ratkaisu toimii vain silloin kun `Tasopiste`-luokalla ei ole perivää luokkaa! Rikotaanpa lähdökohdaksi otettu ajatus kopiointikonstrukoreiden kaikkivoipaisuudesta nenäliinan kokoisiksi riekaleiksi. Oletetaan että luokka `TasompiPiste` perii luokan `Tasopiste`. Ajetaanpa seuraava ohjelmanpätkä:

```
TasompiPiste apu1 = new TasompiPiste(/* ... */),
             apu2 = new TasompiPiste(/* ... */),
             apu3 = new TasompiPiste(/* ... */);
Kolmio k = new Kolmio(apu1, apu2, apu3);
Kolmio kk = k.syväkopio();   // BANG! Ei ole kopio!
```

Alityyppinen olio ei siis syväkopioidu oikein konstruktoreilla!

Syöksykierre oikenee jos vaaditaan että `Tasopiste`-oliot alityyppisine olioineen osaavat syväkopioitua:

```
public class Kolmio
  {
  /* ... */
  public Kolmio syväkopio()
    {
    return new Kolmio(kärki1.syväkopio(),
```

```
                    kärki2.syväkopio(),
                    kärki3.syväkopio());
    }
  } // Kolmio
```

Koska syväkopioitumisominaisuutta vaaditaan monilta erityyppisiltä olioilta, seuraava kehitysaskel olisi tietenkin määritellä rajapinta `Syväkopioituva`:

```
public interface Syväkopioitava<T>
  {
  public abstract T syväkopio();
  }
```

Eli luokat `Kolmio`, `Tasopiste` ja `TasompiPiste` toteuttaisivat kyseisen luontifunktion. Kuulostaako tutulta? Nythän meillä on siis ns. virtuaalinen konstruktori (ts. dynaamisesti sidottu konstruktorikutsu), jolloin palataan takaisin jo olemassaolevaan `clone`-ajatteluun! Erona on että nyt olemme eriyttäneet pinta- ja syväkopioinnin omiksi funktioikseen ja kutsuja saa valita mieleisensä.

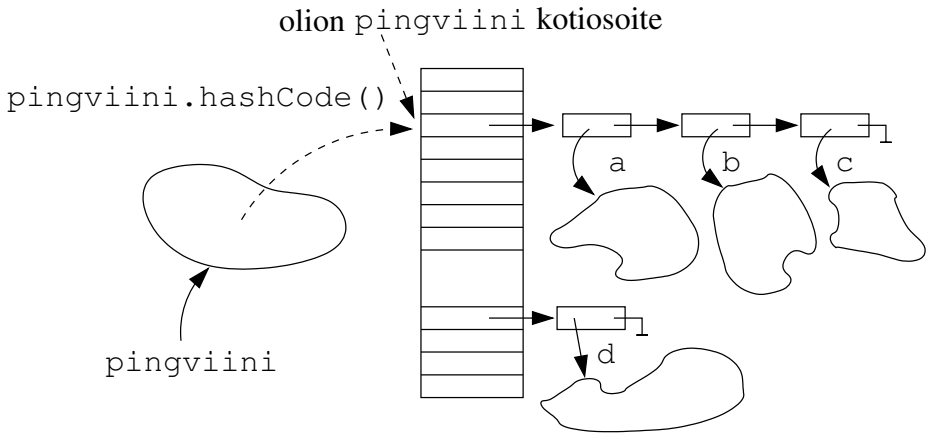
Minkä periaatteellisen syyn vuoksi tämä on näin kamalan hankalaa? Vastaus: Koska oliota käsitellään arvosemanttisesti, jolloin on samalla otettava kantaa dynaamisen sidonnan tarjoamaan kanssaolioiden tyypin variabiliteettiin.

Kolmas tapa hoitaa kopiointi — ja tehdä se kukaties kunnolla — on toteuttaa `java.io.Serializable`-rajapinta, jonka alkuperäinen tarkoitus on sarjaistaa annetun olion tiedot, jotta ne voitaisiin lähettää esimerkiksi linjaa pitkin toiselle koneelle. Idea on yksinkertainen: kopioitava olio sarjaistetaan ja talletetaan esimerkiksi levylle, josta ne luetaan toisen olion uudeksi arvoksi. Java huolehtii itse koko oliorakenteen läpikäynnistä, joten ohjelmoijan ei tarvitse puuttua siihen lainkaan. Yksinkertaista mutta totta!

6.5 hashCode

Jo aiemmin todettiin, että funktioille `hashCode` ja `equals` pitää antaa samanaikaisesti korvaavat toteutukset. Asian ymmärtämiseksi tarkastellaan ensin mihin mekanismiin funktiota `hashCode` tarvitaan. Idea voidaan esittää perustietorakenteen, *hajautustaulun* (*hash table*) avulla. Hajautustaulu on yksinkertainen taulukko, johon on talletettu tietoa ja jossa lasketaan `hashCode`-funktion avulla tiedon *kotiosoite* (*home address*) eli taulukkoindeksi, johon tieto talletetaan (tiedot voidaan organisoida hajautustauluun usealla eri tavalla ja hajautusfunktioita on useita eri periaatteella toimivia, ks. [3]).

Javan hajautustaulupohjaiset kokoelmat tallettavat tietoja linkitettyihin listoihin (ks. kuva 6.2). Oliosta `pingviini` lasketaan ensin kotiosoite, josta löytyvä lista käydään läpi. Listahaku suoritetaan vertailemalla haettavaa kuhunkin lista-alkioon talletettuun olioon. Vertailu käyttää siis `equals`-operaatiota: `a.equals(pingviini)`,



Kuva 6.2: Hajautustaulun toiminta.

`b.equals(pingviini)`,... Jotta `equals`-mielessä samanlainen olio olisi mahdollista löytää taulusta, pitää tällaisten olioiden `hashCode`-arvojen olla samat. Saamme siis ehdon: jos `a.equals(pingviini)` on tosi, pitää tällöin myös olla voimassa `a.hashCode() == pingviini.hashCode()`. Toiseen suuntaan ehdon ei tarvitse olla yleisesti voimassa: samasta kotiosoitteesta voi löytyä monenlaisia olioita. Tällöin on kyseessä ns. *törmäys* (*collision*).

`Object`-luokan toteuttama `hashCode`-funktio perustuu olioiden muistiosoitteisiin. Tämän perusteella kaksi *eri* oliota, olivatpa ne `equals`-mielessä samoja tai ei, päätyvät käytännöllisesti katsoen aina eri kotiosoitteisiin edellyttäen, että hajautusfunktio on muodostettu järkevästi.[4] Hajautusfunktion lisäksi hajautustaulun tehokkuuteen vaikuttaa myös taulun *täyttöaste* (*load factor*). Täyttöasteella tarkoitetaan ei-tyhjien taulukkoalkoiden määrän suhdetta taulukon kokoon ja se kuvaa, kuinka täynnä taulukko kyseisellä hetkellä on. Liian tyhjä taulukko tuhlaa muistia ja liian täysi taulukko merkitsee hakujen, lisäysten ja poistojen hidastumista. Hajautustaulupohjaisten kokoelmien `HashSet` ja `HashMap` luonnin yhteydessä voidaankin antaa parametrit täyttöaste ja alkukapasiteetti, joilla taulun käyttäytymistä voidaan kontrolloida. Jotta tiedot hajaantuisivat mahdollisimman hyvin koko taulukon osalle, kooksi kannattaa yleensä valita jokin alkuluku. Kun taulun täyttöaste (yleensä n. 75%) ylitetään, taulun kapasiteetti kaksinkertaistetaan ja kaikki vanhassa taulussa olleet tiedot hajautetaan uuteen tauluun.

Nyt voidaan selvittää tarkemmin, miten `hashCode` kannattaa toteuttaa. Ensin hyvät uutiset: riittää, että funktio palauttaa jonkin kokonaisluvun, sillä ajoaikai-

[4]Tiukasti ottaen tämä ei tietenkään pidä paikkaansa, koska kussakin linkitetyssä listassa on yleensä useita olioita. Hyvä hajautusfunktio ja oikein valittu taulukkokoko takaavat kuitenkin sen, että listat ovat käytännössä lyhyitä, enintään 2–3 alkion mittaisia. Tähän perustuu hajautustaulujen haun tehokkuus.

nen systeemi huolehtii siitä, että luku skaalataan oikealle taulukkoindeksivälille. Ja sitten huonot uutiset eli kaikki vaatimukset, joita `hashCode`-toteutuksen tulee noudattaa:

(i) Olion kotiosoitteen laskemisen pitää perustua tiedolle, joka pysyy muuttumattomana olion koko eliniän. Tarkemmin sanoen funktion `hashCode` pitää palauttaa aina sama kokonaislukuarvo edellyttäen, että mikään `equals`-testiin osallistuvan jäsenmuuttujan arvo ei ole muuttunut.

(ii) Jos oliot `a` ja `b` ovat samat `equals`-mielessä, niiden `hashCode`-arvojen pitää myös olla samat.

(iii) Jos oliot `a` ja `b` ovat `equals`-mielessä erilaiset, niiden `hashCode`-arvot voivat olla joko samat tai sitten ei. On kuitenkin selvää, että mitä tasaisemmaksi hajautusarvojen jakauma saadaan, sitä tehokkaampia hajautustaulupohjaiset kokoelmat ovat.

Vaatimukset ovat melko kovat. Yksinkertainen, aina vakioarvon palauttava toteutus houkuttelisi helppoudellaan, koska se täyttää edellä esitetyt vaatimukset. Se johtaa kuitenkin pahimpaan mahdolliseen eli siihen, että kaikki oliot sijoittuvat samaan kotiosoitteeseen, jolloin idea hajautustaulun käytöstä tehokkaana hakurakenteena vesittyy täysin. Tästä hiukan lievempi muoto on että kotiosoite lasketaan olion luokannimeä vastaavasta merkkijonosta. Tämäkään ei auta, jos hajautustaulu sisältää vain yhdentyyppisiä olioita.

Hyvän, joskaan ei optimaalista (tasaiseen jakaumaan johtavaa) hajautusfunktion saa noudattamalla seuraavaa proseduuria [2]:

1. Talleta kokonaislukumuuttujaan `tulos` jokin vakioarvo (esim. 17) tai yliluokasta periytyvä hajatustaulun arvo (ts. `super.hashCode()`).

2. Suorita seuraavat toimenpiteet jokaisella `equals`-vertailuun osallistuvalle jäsenmuuttujalle:

   (a) Laske jäsenmuuttujasta `a` kokonaislukuarvo `osaArvo` seuraavasti:

      i. Jos tyyppi on **`boolean`**, laske `(a ? 0 : 1)`.
      ii. Jos tyyppi on **`byte`**, **`char`**, **`short`** tai **`int`**, tyyppipakota `(int)a`.
      iii. Jos tyyppi on **`long`**, laske `(int)(a ^ (a >>> 32))`.
      iv. Jos tyyppi on **`float`**, laske `Float.floatToIntBits(a)`.
      v. Jos tyyppi on **`double`**, laske `Double.doubleToLongBits(a)` ja viimeistele saatu **`long`**-tieto kohdan 2(a)iii mukaisesti.

vi. Jos jäsenmuuutuja on viittaustyyppiä ja luokan `equals`-rutiini kutsuu rekursiivisesti tämän jäsenmuuttujan `equals`-operaatiota yhtäsuuruutta tutkiessaan, kutsu samalla tavoin `hashCode`-rutiinia rekursiivisesti ko. oliolle. Jos jäsenmuuttujan arvo on **`null`**, palauta nolla (tai joku muu vakio).

vii. Jos jäsenmuuttuja on taulukko, käsittele sitä ikäänkuin jokainen sen alkio olisi erillinen jäsenmuuttuja eli laske kullekin oma kotiosoitteensa ja yhdistä tiedot kohdan 2b mukaisesti.[5]

(b) Yhdistä kohdasta 2a saatu arvo `osaArvo` tunnisteen `tulos` aiempaan arvoon laskemalla `tulos = 37 * tulos + osaArvo;`

3. Palauta `tulos`.

4. Kun olet saanut homman tehtyä tarkista, että `equals`-mielessä samat oliot palauttavat saman hajautusarvon. Jos ei, korjaa tilanne.

Jos joidenkin jäsenmuuttujien arvot on johdettu toisista, edellisiä ei tarvitse ottaa mukaan hajautusarvon laskentaan. Erityisesti niitä jäsenmuuttujia, joita ei käytetä `equals`-testissä, ei missään tapauksessa saa ottaa mukaan laskentaan. Kohdan 2b kertolaskun nojalla lopullinen tulos riippuu siitä järjestyksestä, jolla jäsenmuuttujien tulokset yhdistetään. Laskennassa käytetyt vakiot (17 ja 37) on valittu satunnaisesti, joskin ne ovat alkulukuja.

Esimerkki 6.5 Luokan `Tasopiste` (ks. listaus 3.1) `hashCode`-metodi on ohjeen mukaisesti siis

```
public int hashCode()
  {
  int tulos = 17;
  long apuX = Double.doubleToLongBits(x);
  tulos = 37 * tulos + (int)(apuX ^(apuX >>> 32));
  long apuY = Double.doubleToLongBits(y);
  tulos = 37 * tulos + (int)(apuY ^(apuY >>> 32));
  return tulos;
  }
```

Huomaa että jos olio on asetettu yhteenkin hajautusta käyttävään rakenteeseen, olion `equals`/`hashCode` -arvoihin vaikuttavat jäsenmuuttujat eivät saa muuttua ilman että muuttunut olio sijoitetaan uudelleen ko. hajautusrakenteeseen! Mitä tästä seuraa? Ainostaan mutatoitumattomia olioita voidaan huoletta sijoittaa tällaisiin rakenteisiin.

[5]Tähän löytyy apua `java.util`-pakkauksen `Arrays.hashCode`-luokkametodeista.

6.6 toString

`Object`-luokasta periytyvä `toString` on melko pelkistetty: merkkijono koostuu luokkanimestä, `@`-merkistä ja hajautusosoitteesta. Havainnollisempaa on tietysti esittää abstraktissa muodossa olion sen hetkinen tila.

Esimerkki 6.6 Tutun `LukuJoukko`-luokan oliota kuvaa melko hyvin esimerkiksi muodossa `{ 2, 5, 7 }` annettu tulostus, joka saadaan aikaan toteutuksella

```
public String toString()
  {
  String tulos = "{";
  Iterator<Integer> it = this.iterator();
  while (it.hasNext())
    {
    tulos += (it.next()).toString();
    if (it.hasNext()) tulos += ", ";
    }
  return tulos + "}";
  }
```

On myös syytä miettiä, kannattaako tulostuksen muoto kiinnittää julkisesti. Jos näin tehdään, asiakkaat voivat käsitellä merkkkijonoa haluamallaan tavalla. Lisäksi jos luokkaan tehdään konstruktori, joka saa ko. muotoa olevan merkkijonon argumenttina, ohjelmoijan on helppo siirtyä merkkijonosta vastaavaan olioon ja oliosta sen merkkijonoesitykseen (kts. esim. pakkauksen `java.math` luokat `BigInteger` ja `BigDecimal`). Haittapuolena tulostusmuodon kiinnittämiselle on tietysti se, että se sitoo kädet koko loppuelämäksi, koska asiakkaat tukeutuvat siihen voimakkaasti. Niin tai näin, *ilmaise aikomuksesi selvästi `toString`-rutiinin määrittelyssä.* Huolehdi myös siitä, että kaikki sen tieto, jonka `toString` palauttaa on saatavissa myös luokan omilla havainnointimetodeilla. Mikäli näin ei ole, asiakkaan on pakko käyttää `toString`-operaatiota ja hakea merkkijonosta `String`-luokan operaatioilla haluamansa. Näin asiakas tulee kytketyksi merkkijonoesityksen muotoon, vaikka sitä ei olisi ”standardoitu”.

6.7 Comparable ja Comparator

Olion perusoperaatioiksi voidaan laskea myös vertailuihin osallistuvat `compareTo` (rajapinnasta `Comparable`) sekä `compare` (rajapinnasta `java.util.Comparator`), joilla on läheinen yhteys `equals`-operaatioon. Rajapintaa `Comparable` käytetään määrittelemään olioiden välinen luonnollinen järjestys, kun taas rajapintaa `Comparator`

käytetään kaikkiin muihin järjestysrelaatioihin. Useille luokille on yleensä määriteltävissä sille ominainen, intuitiivisesti järkeenkäypä ja helposti toteutettava järjestysrelaatio. Kun se on tehty, Javan luokkakirjastot tarjoavat mahdollisuuden lajitella, hakea minimi- ja maksimiarvoja sekä pitää tällaisten alkioiden joukkoa lajiteltuna.

Luokan `Comparable` operaatio `compareTo` on läheistä sukua `equals`-operaatiolle sillä määrittelevätkin molemmat sen, milloin kaksi oliota on yhtäsuuret (kaikki mitä tässä sanotaan rajapinnalle `Comparable` ja operaatiolle `compareTo`, pätee myös rajapinnalle `Comparator` ja sen operaatiolle `compare`). Erona näillä vertailuilla on, että `equals` tuottaa erisuuret/yhtäsuuret -tuloksen, kun taas `compareTo` antaa myös järjestyksen (onko olio pienempi, yhtä suuri vai suurempi kuin toinen). Erona on myös se, että

```
public int compareTo(T olio)
```

on valmis vertailemaan *vain tarkalleen saman luokan esiintymiä* ja nostaa poikkeuksen `ClassCastException`, mikäli argumentti `olio` ei täytä tätä vaatimusta. Palautettu kokonaislukuarvo on negatiivinen jos vertailun kohdeolio on pienempi, nolla jos yhtäsuuri, ja positiivinen jos suurempi kuin argumentti `olio`. Jotta välttyisi yllätyksiltä, luonnollisen järjestyksen tulisi olla konsistentti `equals`-operaation kanssa eli ehdon

```
(olio1.compareTo(olio2) == 0) == (olio1.equals(olio2))
```

tulisi olla voimassa kaikille saman luokan esiintymille `olio1` ja `olio2`.[6] Syynä tähän on se, että jotkin Javan luokista käyttävät näitä operaatioita sekaisin. Erityisesti `TreeSet` ja `TreeMap` ovat tällaisia luokkia. Niitä voivat käyttää vain luonnollisen järjestyksen määrittelevät ja toteuttavat oliot, koska kokoelman sisäistä järjestystä ylläpidetään ko. järjestyksen avulla. `Set`- ja `Map`-rajapinnat määrittelevät oman toimintansa kuitenkin `equals`-operaation avulla. Näin ollen, jos `olio1.equals(olio2)`, mutta `olio1.compareTo(olio2) != 0`, ja `TreeSet`-joukkoon lisätään ensin `olio1` ja myöhemmin `olio2`, tuloksena on joukko, josta puuttuu `olio2`, koska joukossa oli jo samanlainen alkio `Set`-käsitteen näkökulmasta.

Jotta `equals` ja `compareTo` olisivat konsistentteja, kummankin operaation pitää käyttää yhtäsuuruuden määräämisessä tarkalleen samoja jäsenmuuttujia samalla tavoin. Tämä tarkoittaa esimerkiksi viittaustyyppisten jäsenmuuttujien kohdalla rekursiivisia `compareTo`-kutsuja.

Esimerkki 6.7 Luokassa `Tasopiste` on vain primitiivityypin tietoa, ja sen olioiden järjestys voitaisiin määritellä seuraavasti:

```
public int compareTo(Tasopiste toinen)
```

[6] Java-dokumentaatio lipsuu tässä kohdassa, eikä vaadi tätä välttämättä vaan ”suosittelee sitä voimakkaasti”.

```
{
if (toinen == null)
  throw new NullPointerException();
if (x < toinen.x) return -1;
if (x > toinen.x) return +1;
if (y < toinen.y) return -1;
if (y > toinen.y) return +1;
return 0;
}
```

Toteutus on konsistentti luokan `equals`-määrityksen kanssa. Itse asiassa `equals`-metodi voisi kutsua tätä metodia omassa toteutuksessaan.

Tehtäviä

6-1 Lisää alla annettuun luokkaan `Sanajoukko` metodi `equals`, joka testaa, sisältääkö parametrina annettu sanajoukko täsmälleen samat sanat kuin metodikutsun kohdesanajoukko. Tarkista, että metodisi täyttää kaikki `equals`-metodille asetetut ehdot.

```
import java.util.Set;

/** Sanojen joukko. */
public class Sanajoukko {
   /** Sanajoukon sanat. */
   private String[] sanat;

   /**
    * Alustaa sanajoukon sanoiksi sanat.
    * @.pre  sanat != null && ! sanat.contains(null)
    * @.post java.util.Arrays.equals(annaSanat(),
    *                          sanat.<String>toArray(new String[0]))
    */
   public Sanajoukko(Set<String> sanat) {
      this.sanat = sanat.<String>toArray(new String[sanat.size()]);
      }

   /**
    * Palauttaa joukon sanat taulukossa.
    * @.pre  true
    * @.post Olkoon sj taulukkoa RESULT vastaava Sanajoukko-olio
    *        eli sj == new Sanajoukko(
    *                     new java.util.HashSet<String>(
    *                        java.util.Arrays.asList(RESULT)));
    *        this.equals(sj)
```

```
 */
public String[] annaSanat() { return sanat.clone(); }
}
```

6-2 `Object`-luokasta periytyvä `equals`-metodi tutkii identiteettisamuutta. Laadi tehtävässä 5-4 esitellylle `Piste`-luokalle uusi `equals`-metodin toteutus, joka on yhdenmukainen `Comparable`-rajapinnan `compareTo`-metodin kanssa. Anna myös korvaava toteutus `hashCode`-metodille.

6-3 Mitä huomioitavaa tai ongelmakohtia on syväkopioinnissa, jos oliorakenne ei olekaan hierarkkinen kuten esimerkin 6.4 (s. 187) `Kolmio`-määrittelyssä? `Kolmio`-luokassahan on selvä olioiden hierarkia: `Kolmio` viittaa `Tasopiste`-olioihin, ja `Tasopiste` koostuu primitiivisistä **`double`**-arvoista. Mitä esimerkiksi pitäisi ratkaista, kun kopioidaan oliokokonaisuus jossa olioviittaukset muodostavat silmukoita? Silloinhan olioviittauksia seuraamalla saatetaan palata takaisin samaan olioon mistä alunperin lähdettiin. Esimerkiksi tällainen tilanne syntyi aiempien tehtävän 4-4 `Isäntä`–`Miespalvelija` 1:1 -relaatioissa.

6-4 Miksi ainoastaan mutatoitumattomat oliot voidaan sijoittaa turvallisesti hajautustaulua käyttäviin rakenteisiin?

6-5 Luokka `Paparazzi` perii luokan `Toimittaja` seuraavan ohjelmakoodin mukaisesti.

(a) Toteuta luokkiin `Toimittaja` ja `Paparazzi` funktio `toString()`, joka palauttaa merkkijonoesityksen kohdeolion kaikkien jäsenmuuttujien arvoista.

(b) Toteuta luokkiin `Toimittaja` ja `Paparazzi` funktio `toString()` kuten kohdassa (a), mutta tulosmerkkijono alkaa olion tyyppiä vastaavalla merkkijonolla (esimerkiksi siis lausekkeen **`this`**`.getClass().getName()` tuloksella).

```
public class Toimittaja {
   private String nimi;
   private final int syntymävuosi;
   private String lehti;
   private String juttu;

   /** Alustaa toimittajan nimeksi n, syntymävuodeksi sv ja lehdeksi lh.
    *  @.pre  n != null & 1 <= sv & lh != null
    *  @.post annaNimi() == n & annaLehti() == lh & (syntymävuosi on sv)
    */
   public Toimittaja (String n, int sv, String lh) {
      nimi = n;
      syntymävuosi = sv;
      lehti = lh;
      juttu = "";
      }
```

```
   /** Lisää toimittajan jutun loppuun tekstin lt.
    *  @.pre  lt != null
    *  @.post annaJuttu().equals(OLD(annaJuttu()) + lt)
    */
   public void kirjoitaJuttua (String lt) {
      juttu = juttu.concat(lt);
      }

   /** Palauttaa merkkijonoesityksen.
    *  @.pre  true
    *  @.post RESULT == (tulos on tehtäväksiannon mukainen)
    */
   public String toString() { /* Toteuta tämä. */ }

   /* Sisältää myös julkiset kyselymetodit nimelle, lehdelle
    * ja jutulle. */
   }

public class Paparazzi extends Toimittaja {
   private String julkku;

   /** Alustaa paparazzin nimeksi n, syntymävuodeksi sv ja lehdeksi lh.
    *  @.pre  n != null & 1 <= sv & lh != null
    *  @.post annaNimi() == n & annaLehti() == lh & (syntymävuosi on sv)
    */
   public Paparazzi (String n, int sv, String lh) {
      super(n, sv, lh);
      julkku = "<ei ole>";
      }

   /** Asettaa kohdejulkkikseksi j:n.
    *  @.pre  j != null
    *  @.post (kohdejulkkis on j)
    */
   public void asetaJulkku(String j) {
      julkku = j;
      }

   /** Palauttaa merkkijonoesityksen.
    *  @.pre  true
    *  @.post RESULT == (tulos on tehtäväksiannon mukainen)
    */
   public String toString() { /* Toteuta tämä. */ }
```

```
}
```

6-6 Tee `Comparator`-rajapinnan toteuttava luokka, joka vertailee `Työntekijä`-luokan olioita ensisijaisesti työntekijöiden palkan ja toissijaisesti `Henkilö`-luokan toteuttaman `Comparable`-rajapinnan mukaisesti.

6-7 Oletetaan, että työskentelet pienen mutta innovatiivisen yhtiön Kaik Kyborix tuotekehitysosastolla, joka kehittää nörteille elektromekaanisia lisäkkeitä. Osallistut projektiin Tuikkusilmä ja saat seuraavan tehtäväksiannon.

tuotekuvaus Projekti Tuikkusilmä kehittää keinosilmää, joka korostaa ja/tai täydentää tämän silmän kautta tehtäviä näköhavaintoja erilaisilla aputiedoilla. Yksi keskeisin toiminto on optimaallisimman pizzaviipaleen automaattinen korostus: silmä korostaa herkullisinta näkökentässä olevaa pizzaviipaletta vilkkuvalla nuolella.

tehtävän rajaus Pizzaviipaleen herkullisuus on suhteellista, mikä mallinnetaan seuraavasti: Pizzaviipaleen luontaisiksi (l. objektiivisiksi) ominaisuuksiksi on valittu täytteen alla olevan pohjan määrä, reunan määrä, täytteen määrä sekä esteettisyyden määrä. Käyttäjän (l. silmäilijän) henkilökohtainen mieltymys otetaan huomioon näiden ominaisuuksien tärkeyttä kuvaavilla painokertoimilla. Viipale on sitä herkullisempi mitä suurempi on sen ominaisuuksien painotettu summa.

tehtäväksianto Keinosilmän laitteisto tuottaa `Pizzaviipale`-olioita, jossa olevaan sisäluokkaan `Painotus` sinun pitää toteuttaa viipaleen valinnan ydinfunktio `compareTo(Painotus)`. Funktio palauttaa negatiivisen luvun jos kohdeolio on vähemmän herkullinen kuin parametriolio, positiivisen luvun jos kohdeolio on herkullisempi ja nollan jos viipaleet ovat yhtä herkullisia.

Vinkki: Osiossa 6.7 käsitellään olion luonnollisen järjestyksen toteuttamista rajapinnan `Comparable<T>` funktiolla `compareTo(T)`. Ota myös huomioon, että parametri voi olla **`null`** tai eri tyyppiä kuin kohdeolio.

```
public final class Pizzaviipale {
   private double pohja;
   private double reuna;
   private double täyte;
   private double esteettisyys;

   /** Alustaa viipaleen pohjan, reunan, täytteen ja esteettisyyden
    *  suuruudet. Suuruuden skaalaus ja yksikkömitat ovat asiakkaan
    *  vastuulla.
    *
    *  @.pre  0.0 <= p, r, t, e
    *  @.post (pohjan suuruus on p, reunan r, täytteen t,
    *          esteettisyyden e)
```

```
 */
public Pizzaviipale(double p, double r, double t, double e) {
   pohja = p;
   reuna = r;
   täyte = t;
   esteettisyys = e;
   }

public final class Painotus implements Comparable<Painotus> {
   private double[] painot;

   /** Alustaa viipaleen pohjan, reunan, täytteen ja esteettisyyden
    *  painotukset (eli niitä vastaavien suureiden painokertoimet).
    *  Painojen keskinäiset skaalaukset ovat asiakkaan vastuulla.
    *
    *  @.pre  true
    *  @.post (pohjan painotus on pp, reunan pr, täytteen pt,
    *          esteettisyyden pe)
    */
   public Painotus(double pp, double pr, double pt, double pe) {
      painot = new double[] { pp, pr, pt, pe };
      }

   /** @.pre  true
    *  @.post RESULT == (viipaleen kertoimilla painotettujen
    *                    suuruuksien summa)
    */
   public double annaPainotettuna() {
      return pohja * painot[0] + reuna * painot[1] +
             täyte * painot[2] + esteettisyys * painot[3];
      }

   /** Katso java.lang.Comparable<T>. */
   public int compareTo(Painotus toinen) { /* Toteuta tämä. */ }
   } //Pizzaviipale.Painotus
} //Pizzaviipale
```

Luku 7

Geneerisyys

Geneerisyys laajentaa luokkakäsitettä sallimalla luokan käyttämien tyyppien parametrisoinnin. Näin saadut *geneeriset luokat* (*generic classes*) käyttävät todellisen tyypin sijaan tyyppiparametria, joka konkretisoituu vasta kun luokkaa käytetään. Tässä luvussa tutustutaan Javan tarjoamaan geneerisyyteen, joka tuli mukaan Java-kielen versiossa 5.0 (lisätietoa löytyy mm. lähteestä [6]). Lopuksi tarkastellaan yleisesti periytymisen ja geneerisyyden suhdetta.

7.1 Geneerisyys Javassa

Javassa geneerisyys on toteutettu *tyyppiparametrien* (*type parameter*) avulla. Toteutus on luokkapohjainen ja tuttu monista muista oliokielistä. Esimerkiksi Eiffel- ja C++-kielissä parametrisointi on syntaktisesti samankaltainen. Jälkimmäisessä tapauksessa parametrisoitua luokkaa sanotaan *tyyppimalliksi* (*template*). CLU-kielessä tyyppiparametrille asetetut vaatimukset ilmaistaan kertomalla rutiinien signatuurit, joiden vaaditaan todellisella tyyppiparametrilla olevan käytössä.

Javan tyyppiparametrit esitellään luokan otsikossa, jolloin niiden näkyvyysalueena on koko luokka. Luokkaa, jolla on yksi tai useampi tyyppiparametri, sanotaan *geneeriseksi* (*generic*) tai *parametrisoiduksi tyypiksi* (*parameterized type*). Parametrisoitu tyyppi koostuu luokka- tai rajapintatyypistä ja kulmasuluissa seuraavasta parametriosasta. Esimerkiksi

```
ArrayList<String> merkkijonolista = new ArrayList<String>();
```

käyttää parametrisoitua tyyppiä, jonka luokkatyyppinä on `ArrayList`-kokoelmaluokka ja tyyppiparametrina merkkijonoluokka `String`. Parametrina voi tietysti olla toinen parametrisoitu tyyppi, kuten

```
ArrayList<ArrayList<String>> merkkijonolistalista =
                             new ArrayList<ArrayList<String>>();
```

Mikäli parametriosa sisältää useita tyyppejä, ne erotetaan toisistaan pilkulla. Esimerkiksi

```
Map<Integer,String> kuvaus = new HashMap<Integer,String>();
```

määrittelee `Map`-tyypin, jonka avaimet ovat tyypiltään `Integer` ja arvot `String`.

Esimerkki 7.1 Listauksen 7.1 geneerisellä luokalla `Pari<S,T>` on kaksi tyyppiparametria `S` ja `T`. Kun geneerinen luokka `Pari<S,T>` luodaan, tyyppiparametrit korvataan annetuilla tyyppiargumenteilla. Esimerkiksi `Pari<Integer,String>` tyyppiparametri `S` korvataan tyyppiargumentilla `Integer` ja `T` tyyppiargumentilla `String`.

Listaus 7.1: Geneerinen luokka `Pari<S,T>`.

```
public class Pari<S,T>
  {
  private S eka;
  private T toka;

  public Pari(S e1, T e2)
    {
    eka = e1;
    toka = e2;
    }

  public S annaEka() { return eka; }
  public T annaToka() { return toka; }
  public void asetaEka(S e) { eka = e; }
  public void asetaToka(T e) { toka = e; }
  }
```

Geneerisyyttä voidaan käyttää minkä tahansa luokan tai sisäluokan kanssa. Tähän sääntöön on kolme poikkeusta:

(i) Nimetön luokka (ks. kohta 3.3.3) ei voi sisältää parametrisoituja tyyppejä, koska sen (yksi ja ainoa) instanssi luodaan suoraan lennossa.

(ii) Literaaliluokka **`enum`** (ks. kohta 3.3.4) ei voi sisältää parametrisoituja tyyppejä, koska sen instanssit luodaan ainoastaan sen oman määrittelylohkon sisällä.

(iii) Mikään poikkeus eli `Throwable`-luokan perillinen (ks. kohta 2.4.3) ei saa sisältää parametrisoituja tyyppejä. Koska poikkeustenkäsittelymekanismi on ajoaikainen, se ei pysty käsittelemään geneerisiä poikkeusluokkia.

7.1.1 Tyyppiparametrin rajaus

Parametrisoidun tyypin määrittelyssä tyyppiparametrille voidaan antaa rajaus (*bound*). Rajaus muodostuu luokka- tai rajapintatyypistä sekä lisärajapintatyypeistä. Rajauksessa voi siis olla korkeintaan yksi luokka ja sen täytyy esiintyä ensimmäisenä rajausmäärittelyssä. Mikäli rajaukseen liittyy useampi rajapinta, ne yhdistetään `&`-konnektiivilla.

Rajauksessa on käytössä seuraavat määrittelyt:

- **`extends`**: Rajaa tyyppiparametrin ylhäältä päin, ts. minkä luokan ja/tai rajapintojen kanssa tyyppiparametrin on oltava yhteensopiva.
- **`super`**: Rajaa tyyppiparametrin alhaalta päin, ts. minkä luokan ja/tai rajapintojen yliluokka tyyppiparametrin on oltava.

Jos tyyppiparametria ei ole rajattu, sen rajana pidetään `Object`-luokkaa.

Esimerkki 7.2 Tarkastellaanpa seuraavaa geneeristä luokkaa:

```
public class Numero<N extends Number & Comparable>
  {
  private N lukuarvo;

  public Numero(N lukuarvo)
    {
    this.lukuarvo = lukuarvo;
    }
  }
```

Tämä määrittelee luokan, jolla on tyyppiparametri `N`. Tyyppiparametri on rajattu niin että sen täytyy periytyä luokasta `Number` ja toteuttaa rajapinta `Comparable`. Koskapa `Integer` on tällainen luokka, se voidaan antaa luokalle parametrina

```
Numero<Integer> n = new Numero<Integer>(42);
```

7.1.2 Vapaa tyyppi

Tyyppiparametria ei tarvitse nimetä, vaan sen voidaan antaa olla *vapaa tyyppi* (*wildcard*) `?`. Rajoittamaton vapaa tyyppi sopii kaikkiin parametrisoidun tyypin esiintymiin. Esimerkiksi `List<?>` määrittelee että `List`-luokalla on tyyppiparametri, mutta sitä ei kiinnitetä mihinkään tiettyyn tyyppiin. Vapaita tyyppejä voidaan rajata kuten nimettyjä tyyppiparametreja. Niinpä esimerkiksi

```
List<? extends Number>
```

rajaa vapaan tyypin tarkoittamaan mitä tahansa `Number`-luokan perillistä ja

```
Comparator<? super String>
```

rajaa vapaan tyypin tarkoittamaan mitä tahansa `String`-luokan yliluokkaa.

Vapaata tyyppiä voidaan käyttää

- metodin parametrina ja paluutyyppinä,
- jäsenmuuttujan tai muuttujan tyyppinä,
- taulukon alkiotyyppinä,
- toisten parametrisoitujen tyyppien tyyppiargumenttina ja
- tyyppipakotuksessa.

Vapaata tyyppiä *ei* voi käyttää

- olioiden luonnissa,
- taulukoiden luonnissa (poislukien rajoittamaton vapaa tyyppi),
- poikkeusten käsittelyssä,
- **`instanceof`**-ilmauksissa (poislukien rajoittamaton vapaa tyyppi),
- yliluokkana tai
- literaaliluokkana.

7.1.3 Geneeriset metodit

Geneerisyys ei rajoitu pelkästään luokkatason käsitteeksi, vaan se voidaan ulottaa myös yksittäisiin rutiineihin.

Esimerkki 7.3 Listauksessa 7.2 esitelty geneerinen luokkametodi `vaihda` vaihtaa listan paikoissa `i` ja `j` olevat alkiot keskenään. Sitä kutsutaan geneerisessä kuplalajittelurutiinissa `kupla`. Jos `t` on `Integer`-alkioista koostuva lista, rutiinin kutsu on:

```
Lajittelu.<Integer>kupla(t);
```

Listaus 7.2: Luokka `Lajittelu` ja geneeriset luokkametodit `vaihda` ja `kupla`.

```
public class Lajittelu
  {
  /**
   *  @.pre  lista != null && (lista.size() >= 1 &
   *         0 <= i < lista.size() & 0 <= j < lista.size())
   *  @.post lista.get(i) == OLD(lista.get(j)) &
   *         lista.get(j) == OLD(lista.get(i))
   */
  public static <A> void vaihda(List<A> lista, int i, int j)
    {
    A apu = lista.get(i);
    lista.set(i, lista.get(j));
    lista.set(j, apu);
    }

  /**
   *  @.pre  alkiot != null && alkiot.size() >= 1
   *  @.post FORALL(a : OLD(alkiot);
   *                Collections.frequency(alkiot, a) ==
   *                Collections.frequency(OLD(alkiot), a)) &
   *         FORALL(i : 0 <= i < alkiot.size() - 1;
   *                alkiot.get(i).compareTo(alkiot.get(i + 1)) <= 0)
   */
  public static <N extends Comparable<N>> void kupla(List<N> alkiot)
    {
    for (int i = alkiot.size() - 1; i >= 0; i--)
      for (int j = 0; j < i; j++)
        if (alkiot.get(j).compareTo(alkiot.get(i)) > 0)
           Lajittelu.<N>vaihda(alkiot, i, j);
    }
  }
```

Geneeristen metodien yhteydessä törmätään pieneen syntaktiseen oikkuun. Vaikka voisi kuvitella, että eo. esimerkissä rutiinia `kupla` voisi kutsua geneeristä metodia `vaihda` suoraan `<N>vaihda(alkiot, i, j)`, tämä aiheuttaa jo käännösaikaisen virheen. Syntaksisäännöissä vaaditaan, että geneeriset luokkametodit täytyy täsmentää luokkanimellä — jopa siinäkin tapauksessa, että niitä kutsutaan saman luokan sisällä. Sama synktaksirajoitus koskee myös geneerisiä oliometodeja, sillä niiden kutsua täytyy edeltää olion instanssi. Tämä täytyy tehdä jopa siinäkin tapauksessa että metodia kutsutaan luokan sisältä (ts. **`this`**-viittaus).

Esimerkki 7.4 Tarkastellaanpa seuraavaa oliometodia:

```
public <S extends Number> int kokonaislukuna(S x)
  {
  return x.intValue();
  }
```

Mikäli kyseistä metodia halutaan kutsua saman luokan sisällä, kutsuun on liitettävä olioviittaus:

```
int tulos = this.<Double>kokonaislukuna(3.14);
```

7.1.4 Tyyppitypistys ja raakatyypit

Kuinka parametrisoidut tyypit sitten käännetään ajettavaksi koodiksi? Yleisesti ottaen vaihtoehtoja on kaksi:

1. *Koodin erikoistamisessa* (*code specialization*) kääntäjä generoi oman ajettavan koodin jokaiselle parametrisoidun tyypin instanssille.[1]

2. *Koodin jakamisessa* (*code sharing*) kääntäjä generoi vain yhden yleisen koodin. Kaikki parametrisoidun tyypin instanssit käyttävät tätä koodia ja suorittavat tarvittaessa tyyppitarkistuksia ja -pakotuksia.

Javan geneerisyystoteutuksessa on valittu jälkimmäinen lähestymistapa. Kääntäjä luo geneerisistä luokista yhden ajettavan (tavu)koodin ja soveltaa sitä kaikkiin luokan parametrisoituihin tyyppeihin.

Koska geneerisyys on tuotu Javaan jälkeenpäin, se on täytynyt sovittaa yhteen olemassaolevan kielen kanssa niin, että aikaisempi ei-geneerinen koodi on taaksepäin yhteensopivaa uuden, geneerisyyttä tukevan koodin kanssa. Koodin jakamisen Javassa mahdollistavaa tekniikkaa kutsutaan *tyyppitypistykseksi* (*type erasure*).

[1] Esimerkiksi C++ generoi ajettavan koodin jokaiselle tyyppimallin (*template*) instanssille.

Ajatuksena on käännösvaiheessa typistää pois kaikki parametrisoituihin tyyppeihin liittyvä informaatio ja lisätä tyyppitarkistuksia ja -pakotuksia sekä siltametodeja (*brigde methods*) tarpeellisiin paikkoihin. Yksinkertaistaen voitaisiin sanoa, että tyyppitypistys muuttaa geneerisen Java-koodin ei-geneeriseksi.

Tyyppitypistyksen jälkeen parametrisoitu tyyppi on muuttunut *raakatyypiksi* (*raw type*). Esimerkiksi `Set<Integer>` ja `Set<String>` käsitellään käännöksen jälkeen raakatyyppinä `Set`. Toisin sanoen Javan geneerisyys on pohjimmiltaan syntaktista sokeria. Koska geneerinen tyyppi ei ole olemassa sellaisenaan vaan se typistyy raakatyypikseen, kaikki geneerisyyteen liittyvät rakennelmat ovat lyhennysmerkintöjä, eikä niihin siksi voida soveltaa kaikkia normaaleja luokkaoperaatioita: parametrisoidulla tyypillä ei ole esimerkiksi **`class`**-literaalia, joten kutsu `Set<Integer>.class` tuottaa käännösvirheen.

Nyt herää kysymys, mikä on sitten rajoittamattoman vapaan tyypin ja raakatyypin ero? Ero on olematon. Kummatkin toimivat parametrisoitujen tyyppien eräänlaisena ylityyppinä. Kummatkin ovat konkretisoituvia (*reifiable*) tyyppejä, sillä niihin voidaan käyttää **`instanceof`**-ilmausta ja niitä voidaan käyttää taulukon alkiotyyppeinä, mikä ei ole mahdollista konreeteille ja rajoitetuille vapaille tyypeille. Suurin ero rajoittamattoman vapaan tyypin ja raakatyypin välillä on suositus, että raakatyyppejä pyritään vastedes välttämään ja niiden tilalla käytetään rajoittamatonta vapaata tyyppiä. Semanttisesti ne tarkoittavat samaa asiaa, mutta kääntäjä soveltaa tiukempia tarkastelusääntöjä rajoittamattomaan vapaaseen tyyppiin kuin raakatyyppiin.

Toinen mielenkiintoinen seuraus tyyppitypistyksestä on se, että parametrisoitujen tyyppien instanssit *eivät* ole kovarianttisia. Tässä suhteessa geneerisyys eroaa taulukoista, sillä `Object[]` on `String[]` yliluokka (so. kovarianssi). Tämä ei päde generiselle luokille, joten luokka `Set<Object>` ei ole luokan `Set<String>` yliluokka.

Taulukot (ks. osio 8.1) sisältävät lisäyllätyksiä, sillä parametrisoidusta tyypistä ei saa luoda taulukkoa. Jos `Pari<S,T>` on parametrisoitu tyyppi, niin konstruktio

```
Pari<Integer,Integer>[] taulukko = new Pari<Integer,Integer>[100];
```

tuottaa käännösaikaisen virheilmoituksen. Jaaha. Mikäs tässä nyt sitten mättää?

Kuten edellä todettiin, tyyppitypistys muuttaa `Pari<Integer,Integer>[]` -tyypin käännösaikana raakatyypiksi `Pari[]`. Ajoaikainen alkiotyypin tarkistus on mahdotonta, sillä esimerkiksi `Pari<String,String>[]` tyyppitypistyy samaksi raakatyypiksi `Pari[]`. Tästä seuraisi ristiriita, sillä niiden ei pitäisi olla tyyppiyhteensopivia. Koska dynaaminen sidonta uhkaa hajoata, Javan suunnittelijat ovat sopineet, että parametrisoidusta tyypistä ei saa luoda taulukkoa! Tätä kieltoa voidaan kiertää kolmella eri tavalla:

(a) Luodaan suosiolla suoraan raakatyyppinen taulukko:

```
Pari[] taulukko = new Pari[100];
```

Ongelmana on kuitenkin se että alkioiden ajoaikaisesta tyyppiyhteensopivuudesta ei ole mitään takeita.

(b) Luodaan taulukko vapaan tyypin avulla:

```
Pari<?,?>[] taulukko = new Pari<?,?>[100];
```

Ainoa parannus edellisen kohdan konstruktioon on se, että nyt koodi dokumentoi parametrisoitujen tyyppien olemassaolon (vaikkei se sidokaan niitä). Ajoaikaisesta tyyppisopivuudesta ei tässäkään tapauksessa ole takeita.

(c) Käytetään valmista geneeristä kokoelmaluokkaa:

```
List<Pari<Integer,Integer>> taulukko =
                        new ArrayList<Pari<Integer,Integer>>(100);
```

Nyt voimme olla varmoja ajoaikaisesta tyyppiyhteensopivuudesta.[2]

Summa summarum: taulukot ja geneeriset tyypit eivät viihdy kovin hyvin toistensa seurassa.

Toinen ja huomattavasti vakavampi ongelma on nimitörmäysten hallinta. Oletetaanpa että luokka käyttää parametrisoitua tyyppiä `S`. Jos systeemissä on myös luokka nimeltä `S`, edessä näyttää olevan nimitörmäys: kumpaan esimerkiksi muuttujamäärittely `S x` viittaa? Ongelman ratkaisu on ilmeinen: parametrinen tyyppi sitoo vahvemmin kuin samanniminen luokka, joten muuttuja saa sen tyypikseen. Mikäli halutaan kuitenkin viitata luokkaan, se täytyy spesifioida (kuten normaalien luokkien välisten nimitörmäysten tapauksessa) pakkausnimellä (esim. `fi.utu.cs.funktiot.S x`. Kaikki hyvin ja rauha maassa? Entäpä jos luokka `S` on (nimettömässä) oletuspakkauksessa? Tjaa, keinot alkavat olla vähissä, sillä koska oletuspakkaus on kerran nimetön(!), sillä ei ole nimeä, jolla sen luokat voitaisiin spesifioida. Täytyy vain elää tämän rajoituksen kanssa — ja kenties rajoittaa parametrisoitujen tyyppien nimet isoksi yksikirjaimiksi ja nimetä luokat vähintään kaksikirjaimisiksi.

7.2 Esimerkki: `Joukko<T>`

Aiemmin kohdassa 3.1 tarkasteltiin kokonaislukujoukkoa. Tuntuisi luonnolliselta yleistää joukkoa käsittelemään mitä tahansa tyyppiparametrina annettua alkiotyyppiä. Määritelläänpä siis geneerinen luokka `Joukko<T>`:

```
public class Joukko<T> implements Iterable<T>
```

[2]Oikeastaan olemme lakaisseet ongelman maton alle, sillä jonkun on tehtävä rumat temput. Tässä tapauksessa sen on tehnyt luokkakirjaston laatija. Herkempien ei kannatakaan kurkata `ArrayList`-luokan lähdekoodia.

Koska jo LukuJoukko-luokan suunnitteluvaiheessa kiinnitettiin huomiota ratkaisun yleistyvyyteen, geneerinen Joukko-luokka syntyy melko suoraviivaisesti korvaamalla Integer-tyypit tyyppiparametrilla T. Niinpä esimerkiksi metodi lisää näyttäisi seuraavalta:

```
/**
 * Lisää joukkoon alkion x.
 * @.pre  true
 * @.post RESULT.sisältää(x) &
 *        this.poista(x).equals(RESULT.poista(x)) &
 *        this.equals(OLD(this))
 */
public Joukko<T> lisää(T x)
  {
  if (this.sisältää(x)) return this;
  ArrayList<T> tulosLista = new ArrayList<T>(this.lista);
  tulosLista.add(x);
  return new Joukko<T>(tulosLista);
  }
```

Tyyppiparametria T käytetään luokassa Joukko kaikissa niissä paikoissa, joissa halutaan viitata joukon alkioihin. Kuten esimerkkirutiinista lisää näkyy, muutokset aiempaan ovat hyvin pieniä: ainoastaan argumenttityyppi on jouduttu muuttamaan. Todellinen tyyppiparametri annetaan, kun joukko-olio esitellään:

```
Joukko<Tasopiste> pistejoukko = new Joukko<Tasopiste>();
Joukko<Double> luvut = new Joukko<Double>();
```

Tämän jälkeen joukkoon pistejoukko voi tallettaa vain Tasopiste-tyyppistä tietoa ja joukkoon luvut vain Double-tyyppisiä lukuja. Kääntäjä tarkistaa, että näin tapahtuu, joten geneerisyys tuo mukanaan tyyppiturvallisuutta. Koska luokassa Joukko olevien operaatioiden toteutukset eivät edellytä T-tyypin olioilta mitään erikoisominaisuuksia, voidaan todelliseksi tyyppiparametriksi valita mikä tyyppi tahansa. Geneerisyyden voima tuleekin parhaiten esiin siinä, että aiemmin esitellyn luokan LukuJoukko määrittelyksi ja toteutukseksi riittäisi otsikkomäärittely

```
public class LukuJoukko extends Joukko<Integer>
```

ja muutama yliluokkaa kutsuva konstruktori. Yksinkertaista mutta totta![3]

[3]On tietysti eri asia kuinka järkevää tämä on...

7.3 Geneerisyys vai periytyminen?

Geneerisyys helpottaa ohjelmoijan työtä valtavasti, koska hänen tarvitsee kirjoittaa vain yksi yleinen luokka, jota käytetään aina uudelleen antamalla erilaisia todellisia tyyppiparametreja. Muita etuja ovat:

(a) Geneeristä luokkaa kirjoittaessaan ohjelmoija voi keskittyä kohteena olevan tietorakenteen käyttäytymisen toteuttamiseen.

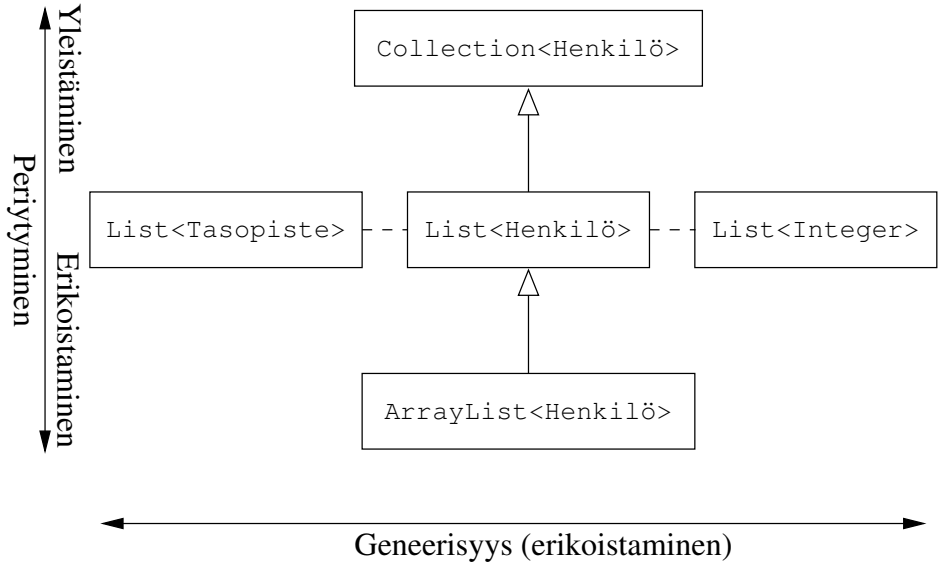
(b) Geneerisyys auttaa ohjelmoijaa yleistämään luokkaa ja siten ymmärtämään paremmin käsitettä, jota on implementoimassa.

(c) Yleistämisen johdosta ohjelmoija joutuu miettimään luokan piirrevalikoimaa syvällisemmin, irti siitä kontekstista, johon hän luokkaa kyseisellä hetkellä tarvitsee. Tämän johdosta luokasta tulee yleiskäyttöisempi.

(d) Ohjelmoija joutuu kiinnittämään huomiota siihen, mitä ominaisuuksia tyyppiparametrilla on oltava ja saa näin käsityksen siitä minkälainen rooli muodostettavalla geneerisellä luokalla on luokkahierarkiassa.

Tällaiset ohjelmointikielen yleistävät, siis uudelleenkäyttöä edistävät mekanismit korostavat jälleen kerran sitä, kuinka tärkeää on olla huolellinen luokkaa toteutettaessa. Mitä yleisemmäksi luokka on tehty, sitä useampi luokkaa tulee käyttämään. Periaatteena voisi sanoa, että yleistäminen kannattaa yleensä lopettaa viimeistään silloin kun uusien asiakkaiden kalastelu aiheuttaa enemmän työtä kuin nykyisten asiakkaiden palvelu.

Yleistäminen toimii kahdella tasolla: periytymisellä yleistetään mallinnettavia käsitteitä, geneerisyydellä käsiteltäviä tietoja (ks. kuva 7.1). Yleistäminen geneerisyyden avulla on melko suoraviivaista, joskaan sitä ei välttämättä nähdä tarpeelliseksi silloin, kun ohjelmistoa kirjoitetaan. Tosin tyyppiparametreihin tottuneelta ohjelmoijalta voidaan odottaa geneeristä ratkaisua moniin eri tilanteisiin, ja yleensä näissä tapauksissa kyseessä onkin vain tavasta, johon ohjelmoija on tottunut, mallintaminen sinänsä ei ole vaikeaa. Sen sijaan yleistäminen periytymisen avulla on yleensä pitkällisen ajatteluprosessin tulos. Onkin normaalia, että ohjelmiston yleistettävät osat nähdään vasta, kun se on jo valmistunut. Tällöin havaitaan, että yleistä käsitettä voidaan käyttää saman projektin eri paikoissa tai vastaavanlainen luokka on tehty jo aiemmin jonkun muun projektin yhteydessä. Kun tästä vielä ajatus kypsyy, pitkällisen ohjelmointiuran omaava henkilö voi havaita systeemistä tutun näköisiä luokkakokonaisuuksia (toistuvia toteutuksia samasta abstraktista ideasta). Tällöin on kyseessä suunnittelumalli, pienehkö joukko luokkia, joilla ratkaistaan tietyntyyppinen ongelma joustavasti asiakas- ja periytymisrelaatioita yhdistäen [5].



Kuva 7.1: Yleistämisen tasot: periytyminen ja geneerisyys.

Tehtäviä

7-1 Muunna listauksessa 3.1 (s. 67) annettu luokka `Tasopiste` geneeriseksi.

7-2 Laadi geneerinen luokkametodi `maksimi`, joka palauttaa saamastaan kahdesta parametristä suuremman. Mieti tyyppiparametrin rajaus.

7-3 Muunna listauksessa 5.6 (s. 157) annettua luokkaa `Lukupari` siten että periytyy listauksessa 7.1 esitellystä geneerisestä luokasta `Pari`.

7-4 Mieti miten voisit yleistää kohdassa 4.2 esitellyn luokkakokonaisuuden suunnatulle graafille. Mihin luokkiin sijoittaisit parametrisoituja tyyppejä ja minkälaisia rajauksia antaisit niille?

7-5 Lisää alla annettuun luokkaan kolme ylikuormitettua geneeristä toteutusta luokkametodille `duplikaatiton`, joka tarkistaa onko parametrina annetun kokoelmaluokan `Collection`, taulukon tai `CharSequence`-tyyppisen merkkijonosekvenssin alkioiden joukossa duplikaatteja. Luokan `main`-metodissa annetun testiajon tuloksena pitäisi siis syntyä:

```
[1, 2, 3] -> true
[1, 2, 1] -> false
abc -> true
aba -> false
[abc, aba] -> true
[abc, abc] -> false
```

```
public class Alkiollinen {
  /** @.pre  kokoelma != null
   *  @.post RESULT == (kokoelmassa ei ole equals-mielessä
   *                    samanlaisia alkiota)
   */
  //--> Anna signatuuri ja toteutus.

  /** @.pre  taulukko != null
   *  @.post RESULT == (taulukossa ei ole equals-mielessä
   *                    samanlaisia alkiota)
   */
  //--> Anna signatuuri ja toteutus.

  /** @.pre  merkkisekvenssi != null
   *  @.post RESULT == (merkkisekvenssissä ei ole samoja merkkejä)
   */
  //--> Anna signatuuri ja toteutus.

  public static void main(String... komentorivi) {
    Integer[] i1 = { 1, 2, 3 }, i2 = { 1, 2, 1 };
    String s1 = "abc", s2 = "aba";
    Collection<String> c1 = new ArrayList<String>(),
                       c2 = new ArrayList<String>();
    c1.add(s1); c1.add(s2);
    c2.add(s1); c2.add(s1);

    System.out.println(Arrays.toString(i1) + " -> " +
                       Alkiollinen.<Integer>duplikaatiton(i1));
    System.out.println(Arrays.toString(i2) + " -> " +
                       Alkiollinen.<Integer>duplikaatiton(i2));
    System.out.println(s1 + " -> " +
                       Alkiollinen.<String>duplikaatiton(s1));
    System.out.println(s2 + " -> " +
                       Alkiollinen.<String>duplikaatiton(s2));
    System.out.println(c1 + " -> " +
                       Alkiollinen.<String>duplikaatiton(c1));
    System.out.println(c2 + " -> " +
                       Alkiollinen.<String>duplikaatiton(c2));
    }
  }
```

7-6 `Collections`-luokassa on määritelty kokoelman alkioiden järjestämiseen ylikuormitettu geneerinen luokkametodi `sort`, jonka yhden toteutuksen signatuuri näyttää seuraavalta:

```
public static <T extends Comparable<? super T>>
                                          void sort(List<T> list)
```

Mitä metodi vaatii tyyppiparametrilta `T`?

7-7 Muunna tehtävässä 3-21 määritelty luokka `DaList` geneeriseksi.

Luku 8

Kokoelmat ja niiden käyttö

Kokoelmaluokkien avulla olioita voidaan ”niputtaa yhteen”. Olemme aiemmin käyttäneet esimerkiksi taulukoita ja listarakenteita (esim. `ArrayList`) ahkeraan, joten ne ovat tulleet jo tutuiksi. Tässä luvussa tarkastellaan, paitsi yksittäisten kokoelmaluokkien tarjoamia palveluja, erityisesti myös sitä, miten kokoelmaluokat on organisoitu perimishierarkiaan. Toimiihan kielen standardikirjaston rakenne yleensä referenssinä, jonka tulisi heijastaa mahdollisimman hyvin niitä periaatteita, joita kielessä pidetään merkittävinä. Samalla saadaan kokonaisvaltainen kuva kaikista niistä perustietorakenteista, jotka on kirjastossa valmiina.

Iteraattorit liittyvät kiinteästi oliokokoelmiin. Niiden avulla kokoelman alkiot voidaan käydä läpi niin, että asiakkaan ei tarvitse tietää mitään läpikäyntiin liittyvistä yksityiskohdista.

8.1 Taulukot

Javan hakasulkunotaatiota käyttävä *taulukko* (*array*) käyttäytyy monella tavoin eri lailla kuin muut kokoelmaluokat. Tämän lisäksi taulukkoa vastaavaa luokkaa ei ole olemassa, koska taulukkomekanismi on rakennettu Java-kielen ”sisään”. Täten taulukko ei voi olla osa kokoelmaluokkien perimishierarkiaa, vaan se on oma itsenäinen käsitteensä.

Taulukon käytön suosio perustuu sen tehokkuuteen, joka puolestaan on seurausta kolmesta perusasiasta: (a) taulukon luonnissa varattu muistitila muodostaa yhtenäisen alueen keskusmuistissa, (b) taulukon alkioihin on suorasaanti eli paikassa `p` oleva alkio saadaan käsiin nopeasti, kun taulukon alkupaikka ja indeksin `p` arvo tunnetaan, ja (c) taulukolle varattu muisti pysyy ohjelman suorituksen ajan kiinteänmittaisena. Kohdassa (b) mainitun muistiosoitteen tehokas las-

keminen edellyttää, että taulukon indeksointi alkaa aina nollasta (ja on tietysti kokonaislukutyyppiä). Java ei siis salli esimerkiksi negatiivisia kokonaislukuindeksejä tai muita indeksityyppejä (kuten vaikkapa Pascal). Useissa sovelluksissa olisi luonnollista aloittaa indeksointi ykkösestä. Tämä on helppo toteuttaa varaamalla tilaa yhden position verran enemmän kuin on tarve ja jättämällä paikka nolla käyttämättä. Hankalammissa tapauksissa indeksointialueen kuvaus standardi-indeksointiin jää ohjelmoijan harteille. Samoin käy esimerkiksi silloin, kun indeksinä halutaan käyttää vaikkapa **char**-tyypin tietoa. Ehdon (c) nojalla taulukon maksimikoko tulisi tietää mahdollisimman tarkkaan luontivaiheessa. Jos taulukkoa joudutaan laajentamaan suoritusaikana, sen sisältämät tiedot joudutaan kopioimaan paikasta toiseen. Tämä dynaamisuuden puute onkin taulukon käytön suurin haittapuoli.

8.1.1 Taulukko-operaatiot

Taulukko luodaan normaalisti **new**-operaatiolla:

```
Henkilö[] asukkaat = new Henkilö[100];
```

Luonnin jälkeen taulukon alkiot on alustettu oletusarvoilla (vrt. luokan jäsenmuuttujien alustus), joten taulukossa `asukkaat` on sata **null**-arvoa. Taulukolla on vain yksi attribuutti, `length`, joka kertoo taulukon kapasiteetin (pituuden). Tietoa siitä, montako oliota on talletettu taulukkoon, täytyy siis ylläpitää itse.

Taulukkoa vastaavaa luokkaa ei olemassa ja taulukon ajatellaan olevan vain osittain `Object`-luokan jälkeläinen. Tärkein näin saaduista perusoperaatioista lienee `clone`, jolla saa kopioitua taulukon alkiot toiseen.[1]

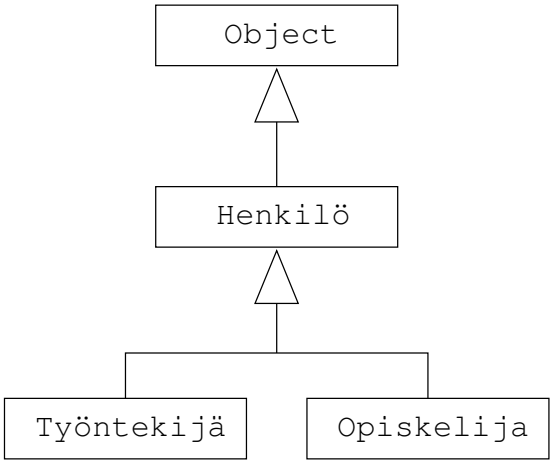
```
int[] luvut = { 1, 2, 3, 4, 5 };
int[] uudet = luvut.clone();
```

Huomaa, että jos sama tehtäisiin taulukolle `asukkaat`, uuteen taulukkoon kopioituisi vain viittaukset samoihin `Henkilö`-olioihin, mitä alkuperäisessä taulukossa on (kopio tehdään vain taulukkorakenteeseen, ei rekursiivisesti sen sisältämiin olioihin). Muista `Object`-luokalta saatavista rutiineista ei ole paljon käyttöä taulukoiden yhteydessä. Esimerkiksi `equals` on hämäävä, koska edellisten alustusten jälkeen suoritettavasta lauseparista

```
System.out.println(luvut.equals(uudet));
System.out.println(Arrays.equals(luvut,uudet));
```

ensimmäinen palauttaa arvon **false** (taulukoilla on eri identiteetti) ja toinen, pakkauksesta `java.util` löytyvää apuluokkaa `Arrays` käyttävä rutiini palauttaa arvon

[1] Toinen mahdollisuus on käyttää kopiointirutiinia `System.arraycopy`, jolla voidaan kopioida osa taulukosta toiseen. Rutiini toimii myös silloin, kun lähde- ja kohdetaulukot ovat samat. Koska kysymyksessä on natiivikoodilla toteutettu rutiini, se lienee tehokkaampi kuin `clone`.



Kuva 8.1: Henkilön ja työntekijän periytyminen.

true (komponenteittain tehty vertailu). Taulukon alkoiden tulostamisessa voi käyttää luokkametodia `Arrays.toString`.

Vielä sananen taulukoiden tyyppiyhteensopivuudesta. Aiemman esimerkkimme mukaan `Henkilö` perii `Object`-luokalta ja `Työntekijä` sekä `Opiskelija` perivät `Henkilö`-luokalta kuvan 8.1 mukaisesti. Taulukot ovat tyyppiyhteensopivat vastaavalla tavalla. Täten lausejono

```
Työntekijä[] työtaulu = new Työntekijä[100];
Henkilö[] asukkaat = työtaulu;
asukkaat[20] = new Työntekijä( /* ... */);
```

on oikein muodostettu.

Ja vaikka taulukoilla onkin jonkin verran oliomaisia piirteitä, niin kohtuus kaikessa. Niinpä taulukoille ei voi kirjoittaa esimerkiksi perijöitä.

8.1.2 Apuluokka `Arrays`

Luokka `Arrays` on muodostettu nimenomaan taulukoiden käsittelyä silmälläpitäen.[2] Perusvalikoima on seuraava:

```
public static <T> int binarySearch(T[] a, T k,
                                   Comparator<? super T> c)
```

Hakee binäärihaulla taulukosta `a` alkiota `k` käyttäen vertailijaa `c` (mikäli `c == null` käytetään luonnollista järjestystä) ja palauttaa alkion löytymispaikan indeksin. Jos alkiota ei löydy, palautetaan kokonaisluku `(-p - 1)`, missä `p` ilmaisee sen paikan (indeksin) jonossa, johon avainarvo `k` järjestetyssä jonossa kuuluisi.

[2]Ks. myös luokkaa `java.lang.reflect.Array`, jonka avulla taulukoita voidaan käsitellä ajoaikana.

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

Lajittelee taulukon `a` tiedot stabiililla lomituslajittelulla vertailijan `c` ilmaisemaan järjestykseen (mikäli `c == null`, käytetään luonnollista järjestystä).

```
public static void fill(Object[] a, Object v)
```

Tallettaa taulukon `v` jokaiseen positioon viittauksen olioon `v`.

```
public static boolean equals(Object[] a1, Object[] a2)
```

Palauttaa arvon **true**, jos taulukot ovat yhtä pitkät ja niiden vastinalkiot ovat `equals`-mielessä samat.

```
public static <T> List<T> asList(T... a)
```

Palauttaa listan, jossa on viittaukset taulukon `a` vastinalkioihin. Listan pituus on `a.length`. Toiseen suuntaan muunnos tehdään rutiinilla `Collections.toArray`.

Esimerkki 8.1 Kohdassa 6.7 annettiin luonnollisen järjestyksen määräävä `compareTo`-rutiini luokan `Tasopiste` olioille, joten pisteitä sisältävä taulukko `pisteet` voidaan lajitella yksinkertaisesti kirjoittamalla `Arrays.sort(pisteet)`. Jos tason pisteet halutaan lajitella johonkin muuhun järjestykseen, esimerkiksi *laskevaan* luonnolliseen järjestykseen, on turvauduttava `Comparator`-luokkaan.[3] Muodostettavalla rakennelmalla ei ole kuitenkaan enää mitään muuta yhteyttä `Tasopiste`-luokkaan kuin asiakasrelaatio: `Comparator` ja sen perijät muodostavat täysin oman "periytymissaarekkeensa". Lajittelua varten kirjoitetaan ensin

```
public class KäänteinenPistejärjestys
                                  implements Comparator<Tasopiste>
  {
  public int compare(Tasopiste p1, Tasopiste p2)
    {
    if (p1.annaX() < p2.annaX()) return +1;
    if (p1.annaX() > p2.annaX()) return -1;
    if (p1.annaY() < p2.annaY()) return +1;
    if (p1.annaY() > p2.annaY()) return -1;
    return 0;
    }
  }
```

Tämän jälkeen kutsutaan `Arrays`-luokan lajittelurutiinia

```
Arrays.<Tasopiste>sort(pisteet, new KäänteinenPistejärjestys());
```

[3]Koska käänteinen luonnollinen järjestys on hyvin yleinen lajittelukriteeri, sitä varten `Collections`-luokasta löytyy jo valmiina `reverseOrder`-rutiini.

Vaikka Java ei sallikaan funktioargumenttien välittämistä rutiineille, esimerkki osoittaa, miten sama saadaan aikaan kapseloimalla funktio yksinkertaisen luokan sisään ja välittämällä tämän luokan olio sitä tarvitsevalle (tästä lisää kohdassa 9.3.3). Kertakäyttöisesti sama voitaisiin tietysti tehdä myös nimettömällä luokalla (ks. kohta 3.3.3).

8.2 Kokoelmaluokat

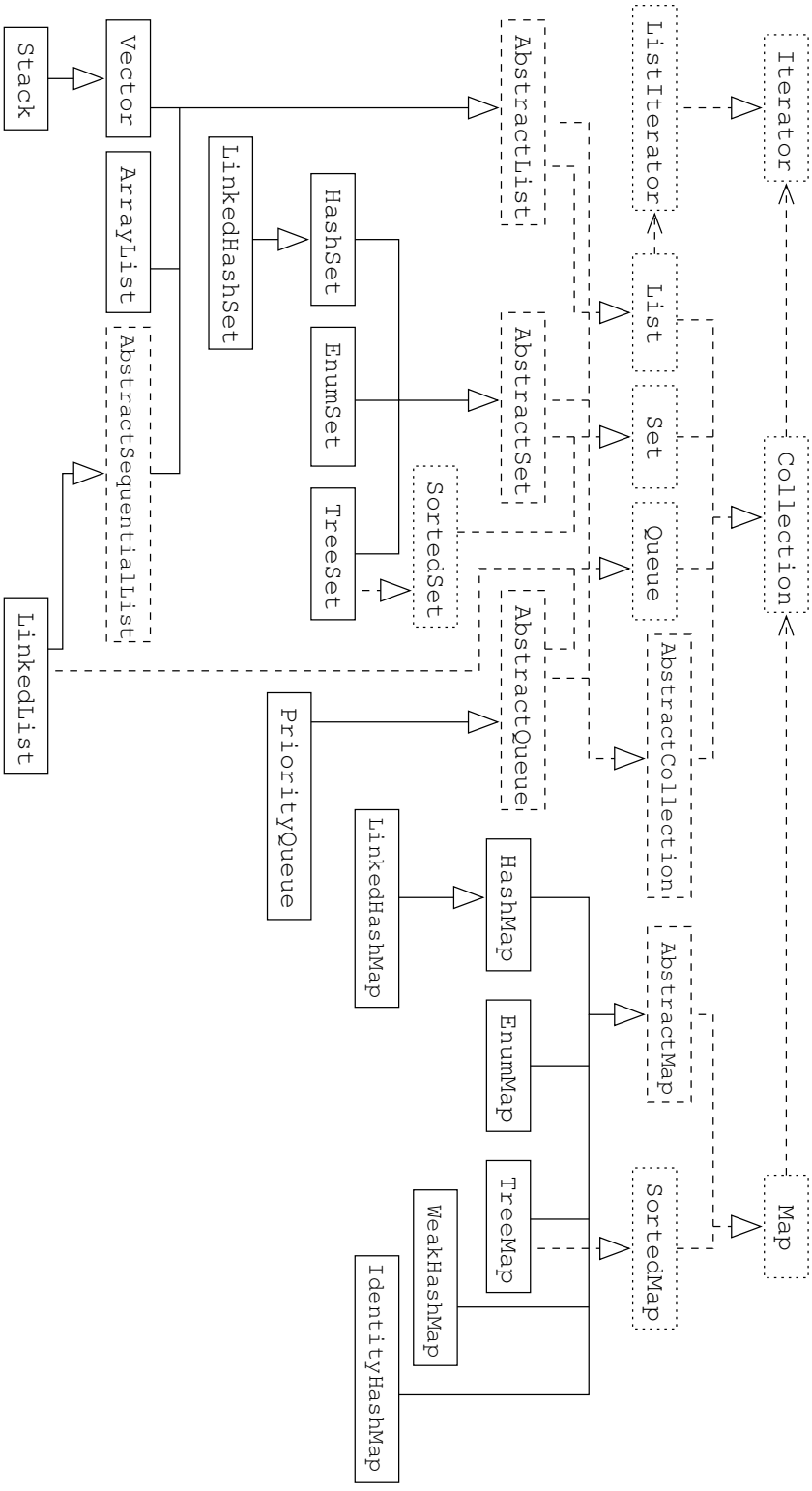
Pakkauksen `java.util` sisältämät kokoelmaluokat sisältävät apuvälineitä jokapäiväiseen ohjelmointiin: matemaattinen joukko, linkitetty lista, hajautustaulu, pino, jono,... Nämä ovat käsitteellisesti melko primitiivisiä, mutta kekseliään käyttäjän käsissä hyvinkin ilmaisuvoimaisia. Kokoelmaluokkien välille rakennettu hierarkia on esitetty kuvassa 8.2. Pisteviivalla merkityt luokat ovat rajapintoja, katkoviivalla merkityt abstrakteja luokkia ja yhtenäisellä viivalla merkityt konkreetteja luokkia. Umpikärkiset nuolet edustavat olion luontia; nuolen lähtö ilmaisee luokan, joka voi generoida olion, joka on nuolen tulokohdan tyyppiä. Kirjasto koostuu selkeästi kolmesta komponentista: (a) `Map` ja sen perijät, (b) `Collection` ja sen perijät sekä (c) iteraattoriluokat (joita käsitellään kohdassa 8.3).

Luokkahierarkia voidaan rakentaa kieleen useita eri periaatteita noudattaen. Javan hierarkia on hyvin yksinkertainen. Ylimmällä tasolla ovat `Collection` ja `Map`:

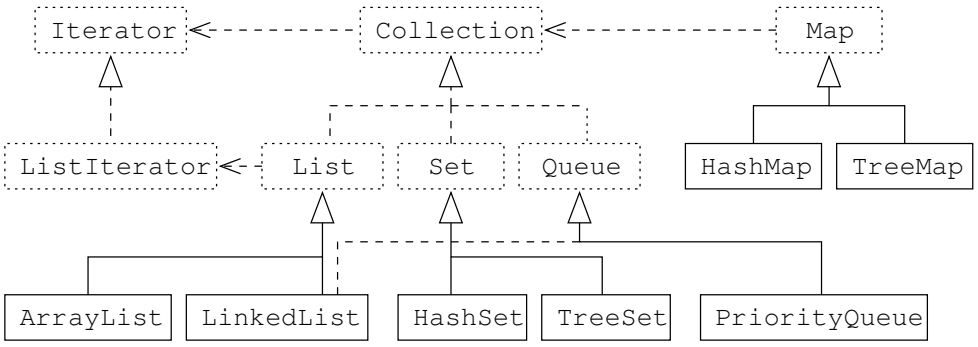
- `Collection` edustaa tavallista kokoelmaa, jossa kokoelmaan tallettavalla peruselementillä ei ole (ainakaan ulospäin) mitään rakennetta, vaan se on vain möhkäle tietoa.

- `Map` edustaa kokoelmaa, jonka alkiolla on *avainarvo* (*key*) ja siihen liittyvä *data-arvo*. Avaimet ovat yksikäsitteisiä, eri avaimiin liittyvät data-arvot voivat olla sen sijaan samanlaisia. Taustalla on matemaattisen funktion käsite ts. kuvaus (*mapping*) määrittelyjoukolta (avainarvot) tulosjoukolle (data-arvot). Esimerkiksi taulukkoa voidaan pitää tällaisena tietorakenteena: kokonaislukuindeksi on avain, jolla päästään käsiksi dataan eli taulukkoon talletettuun tietoon. Koska kukin `Map`-luokan peruselementti koostuu olioparista, `Map`-tyyppi yleistyy tarvittaessa useampiulotteiseksi taulukoiden tapaan (esimerkiksi kolmiulotteinen `Map` saadaan muodostamalla avainarvo sekä datakenttä, joka koostuu avainarvosta ja datakentästä).

`Collection`-rajapintaa tarkennetaan alemmalla abstraktiotasolla sen mukaan, voiko oliolla olla kokoelmassa duplikaatteja (`List` ja `Queue`) vai ei (`Set`). Samassa yhteydessä kiinnitetään olioiden järjestysominaisuus: `List`-kokoelman oliot on järjestetty aina lineaarisesti (joten niihin voidaan viitata luonnollisilla luvuilla), `Queue`-kokoelman alkioista tiedetään kulloinkin jonon päässä oleva alkio ja `Set`-kokoelman



Kuva 8.2: Kokoelmaluokkien hierarkia.



Kuva 8.3: Kokoelmahierarkian tärkeimmät rajapinnat ja luokat.

alkiot ovat järjestämättömiä. Seuraava korkean tason erottelukriteeri on, ylläpidetäänkö kokoelman tietoja lajiteltuna (`SortedSet` ja `SortedMap`) vai ei.

Kun nämä ominaisuudet on kiinnitetty, voidaan luokkahierarkiaan kirjoittaa konkreetit toteutukset, jotka voivat pohjautua mm. hajautustauluun (`HashMap` ja `HashSet`), punamustapuuhun (`TreeMap` ja `TreeSet`), taulukkoon (`ArrayList`) tai linkitettyyn listaan (`LinkedList`). Tärkeä syy näinkin matalan abstraktiohierarkian riittävyydelle on, että perusluokkiin on sijoitettu ns. *ei-pakollisia* (*optional*) piirteitä, joihin ei perijöissä välttämättä odotetakaan relevanttia toteutusta.

Parametrisoidun tyypin avulla määrätään kokoelman alkioiden tyyppi. On tietysti mahdollista käyttää `Object`-tyyppiä, mutta se johtaa helposti siihen, että kokoelmaan saatetaan tallettaa vahingossa useita erityyppisiä tietoja. Tällainen virhe tulee esille vasta ajoaikana.

Kokoelmaluokilla on `add`-operaatio, jolla lisätään yksittäinen alkio. Operaation todellinen toteutus on ominainen kyseiselle luokalle: esimerkiksi `Set`-joukkoon lisäys tehdään vain, jos ko. alkiota ei ole siinä ennestään, kun taas listaan ja jonoon lisäys tehdään aina (joko niiden loppuun tai listassa nimettyyn positioon).

Luokat, joiden nimet alkavat sanalla `Abstract`, ovat nimensä mukaisesti abstrakteja luokkia, joissa on toteutettu yksi tai useampia perittyjen rajapintaluokkien piirteitä. Jos siis teet oman kokoelmaluokan, peri abstrakteilta luokilta (jos vain mahdollista) äläkä rajapinnoilta. Toisaalta hierarkiaan valmiiksi tehtyjen kokoelmien käyttö tapahtuu tyypillisesti kahdella tasolla: kokoelma luodaan valitsemalla haluttu implementoiva luokka ja sitä käytetään rajapinnan tarjoaman julkisen liitännän kautta. Tätä käyttötapaa ajatellen abstraktit luokat voidaan jättää pois kuvasta 8.2, jolloin kuvan 8.3 mukaisesti hierarkia näyttää selkeämmältä (kuvasta on myös jätetty pois jäämistöluokat, engl. *legacy classes*, `Vector` ja `Stack` sekä **`enum`**-pohjaiset toteutukset).

8.2.1 Apuluokka `Collections`

Kuten perustyypeille, myös kokoelmien ”päälle” on tehty kuoritoteutuksia, jotka avustavat kokoelmien käyttöä. Nämä piirteet löytyvät luokasta `Collections`, joka tarjoaa samalla tavalla apuvälineitä kokoelmille kuin `Arrays` taulukoille. Kuoria on kolmenlaisia:

- *Synkronointikuori* (*synchronization wrapper*) kontrolloi alla olevaan kokoelmaan kohdistuvia operaatioita ja huolehtii siitä, että ne eivät tee kokoelmaan muutoksia samanaikaisesti esimerkiksi eri säikeistä.[4] Koska myös alkuperäiseen (ei-synkronoituun) kokoelmaan saattaa olla viittauksia, kannattaa niistä yrittää päästä eroon sekaannusten välttämiseksi. Esimerkiksi turvallinen synkronoitu `Tasopiste`-olioiden lista saadaan luotua kirjoittamalla

```
List<Tasopiste> lista = Collections.<Tasopiste>
                    synchronizedList(new ArrayList<Tasopiste>());
```

- *Jäädytyskuori* (*unmodifiable wrapper*) karsii käyttäjältä kaikki sellaiset operaatiot, joilla kokoelmaa voidaan muuttaa (vaikka muutosoperaatiot onkin toteutettu ko. luokassa). Tässä pätee sama kuin yllä: ohjelmassa voi olla samaan kokoelmaan myös sellaisia viittauksia, joiden kautta muutokset on sallittu. Ohjelmoijan on huolehdittava siitä, että kokonaisuus pysyy hallinnassa. Jäädytyskuorta voi käyttää esimerkiksi silloin, kun olion konkreetti esitysmuoto halutaan antaa suoraan asiakkaan käyttöön. Koska kokoelma on asiakkaan kannalta jäädytetty, hän ei pääse muuttamaan sitä ja siten tuhoamaan olion luokkainvarianttia.

- *Tyyppisamuuskuori* (*checked wrapper*) voidaan käyttää kokoelman rakennealkioiden samantyyppisyyden takaamiseen.

Näiden kuoritoteutusten lisäksi `Collections`-luokka tarjoaa rutiineja mm. kokoelmien kopiointiin, sekoittamiseen, lajitteluun, täyttöön, binäärihakuun ja minimin tai maksimin hakuun. Kannattaa tutustua ja hyödyntää tarjolla olevaa valikoimaa!

8.2.2 Rajapinta `Collection`

Rajapinta `Collection` on abstraktein kokoelmamäärittely ja siksi sitä käytetään polymorfisesti perijäluokkien esiintymiin esimerkiksi silloin, kun todellinen implementointi halutaan piilottaa asiakkaalta. Kaikilla `Collection`-luokan perijöillä on kaksi konstruktoria: toinen luo tyhjän joukon ja toinen alustaa luodun kokoelman

[4] Pakkauksesta `java.util.concurrent` löytyy samaan tarkoitukseen valmiita konkurrentteja perustyyppi- ja kokoelmaluokkia.

argumenttina annetulla `Collection`-tyyppisen olion sisällöllä. Jälkimmäisen konstruktorin avulla konversiot eri implementointien välillä sujuvat vaivatta. Esimerkiksi `HashSet<Integer>`-tyyppinen olio `lukujoukko` voidaan muuttaa listaksi tekemällä automaattinen yleistulkinta `Collection`-luokkaan ja antamalla se argumentiksi `ArrayList`-luokan konstruktorille:

```
List<Integer> munLuvut = new ArrayList<Integer>(lukujoukko);
```

`Collection`-rajapinta on esitelty listauksessa 8.1. Rutiini `add` lisää kokoelmaan alkion esiintymän, `remove` poistaa.[5] Kumpikin palauttaa totuusarvon **`true`**, jos kokoelman tila muuttuu rutiinin suorituksen seurauksena. Muutosta ei tapahdu lisäyksessä, jos olio oli kokoelmassa jo ennestään ja kokoelmatyyppi ei salli duplikaatteja, eikä myöskään poistossa, jos poistettavaa ei löytynyt kokoelmasta.

Listaus 8.1: Rajapinta `Collection`.

```
public interface Collection<E> extends Iterable<E>
  {
  //-- Perusoperaatiot
  public int size();
  public boolean isEmpty();
  public boolean contains(Object o);
  public boolean add(E o);                          // Ei-pakollinen
  public boolean remove(Object o);                  // Ei-pakollinen
  public Iterator<E> iterator();

  //-- Kokoelmiin liittyvät operaatiot
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c); // Ei-pakollinen
  public boolean removeAll(Collection<?> c);        // Ei-pakollinen
  public boolean retainAll(Collection<?> c);        // Ei-pakollinen
  public void clear();                              // Ei-pakollinen

  //-- Taulukko-operaatiot
  public Object[] toArray();
  public <T> T[] toArray(T[] a);
  }
```

Poistojen yleistykset `addAll` ja `removeAll` toimivat analogisesti. Tilanteessa, jossa yhdestä `Integer`-alkiosta `i` voi olla kokoelmassa `munLuvut` useita esiintymiä,

[5] `Collection` ei ota mitään kantaa siihen, miten `add` ja `remove` toimivat, erityisesti ne eivät kiinnitä näissä operaatioissa käytettävää yhtäsuuruutta.

ne saadaan kaikki poistettua kirjoittamalla

```
munLuvut.removeAll(Collections.<Integer>singleton(i));
```

sillä luokan `Collections` rutiini `singleton` palauttaa `Set`-tyyppisen kokoelman, jossa on vain argumenttina annettu alkio.

`Collection`-luokan `retainAll` poistaa kaikki sellaiset alkiot, jotka eivät löydy argumenttikokoelmasta. Rutiini `contains` testaa onko kokoelmassa alkiota, joka on `equals`-mielessä samanlainen argumentin kanssa. Kokoelman läpikäyntiä varten luokka tarjoaa asiakkaalle piirrettä `iterator`, joka palauttaa yleistä iteraattoriliitäntää `Iterator` käyttävän olion. Taulukko-operaatio `toArray` tallettaa kokoelman alkiot vastaanottavaan `Object[]`-tyyppiseen taulukkoon, jonka pituus on sama kuin kokoelman koko.[6] Näin päästään aidosta kokoelmaoliosta taulukkoon. Muunnos toiseen suuntaan tehtiin `Arrays`-luokan `asList`-operaatiolla.[7]

Jotkut `Collection`-luokan operaatioista on merkitty *ei-pakollisiksi* (*optional*). Tämä tarkoittaa sitä, että perivän luokan ei tarvitse välttämättä toteuttaa tällaista rutiinia. Jaaha. Mites tähän on tultu, eikö (a) se merkitse sitä, että perivä luokka on abstrakti ja eikö (b) rajapinnan tarkoitus ole juuri toimia yhteisenä yliluokkana, jonka mukaan kaikki perijät toimivat? Ennen kuin pohditaan vastauksia, todetaan syy tähän käytäntöön: rajapinnasta on haluttu tehdä mahdollisimman yleinen, jotta hierarkiaan ei tarvitsisi tehdä useita rajapintoja, joiden semantiikat eroavat vain hieman toisistaan. Ei-pakolliset piirteet ovat tästä syystä liian korkealla abstraktiotasolla ja voivat siten periytyä myös sellaisiin hierarkiahaaroihin, joissa niillä ei ole merkitystä. Jos käytössä olisi moniperiytyminen, hierarkiasta voitaisiin tehdä paljon hienosyisempi ja perijä voisi valita siitä itselleen sopivat ”viipaleet”. Näin saataisiin muodostettua rajapinnat, jotka mallintavat staattisia kokoelmia (muutoksia ei sallita), kiinteänkokoisia kokoelmia jne.

Mitä sitten tulee ylläoleviin kysymyksiin, kohta (a) ratkaistaan siten, että perivästä luokasta tehdään konkreetti antamalla valinnaiselle piirteelle toteutus, joka kertoo sen olevan soveltumaton (*unsupported*). Esimerkiksi jos `Collection`-luokan piirteelle `clear` ei ole sen määrittelyn mukaista toteutusta, kirjoitetaan

```
void clear() { throw new UnsupportedOperationException(); }
```

Tämä tietysti rikkoo sitä sääntöä, jonka mukaan perityn piirteen pitää säilyttää semantiikkansa. Kysymykseen (b) ei ole järkevää vastausta. Asiaa lieventää hieman se, että kaikki `Collection`-luokan toteutukset antavat kaikille perityille piirteille määrittelyn mukaisen implementoinnin. Jotta valinnaisuusmenettely toimisi yleisesti, täytyy seuraavien ehtojen olla voimassa:

[6] Tämän funktion pakollisella olemassaololla on melkoinen seuraus: `Collection`-tietorakenne ei voi koskaan olla ääretön eikä käytännössä edes kovinkaan iso. Tällaiset alkioiden generointiin perustuvat rakenteet on siis toteutettavissa vain iteraattoreilla.

[7] Näin saatuun listaan ei voi kohdistaa muutosoperaatioita `add` ja `remove`.

- Poikkeus `UnsupportedOperationException` on ajoaikana erittäin harvinainen. Valinnaisuusominaisuutta tulisi soveltaa vain silloin, kun hierarkiaan tehdään oma kokoelma, jossa osalla `Collection`-rajapinnan valinnaispiirteistä ei ole järkevää vastinetta.

- Poikkeusta `UnsupportedOperationException` käytetään vain testausmielessä ennen ohjelmiston luovuttamista laajaan käyttöön. Tämä käyttötapa auttaa löytämään loogisia virheitä ohjelmasta, sillä eihän "järjetöntä" rutiinia ole syytä mennä kutsumaan. Toisaalta tällainen ajattelutapa tuo haasteita testien suunnitteluun: miten taataan, että kaikki mahdolliset kutsutapaukset on käyty läpi?

8.2.3 Rajapinta `Set`

Rajapintaluokka `Set` ei esittele uusia piirteitä `Collection`-luokalta perimiensä lisäksi. `Set` toimii matemaattisen joukon tavoin eikä ylläpidä alkioista duplikaatteja. Jotta joukkoja voitaisiin vertailla järkevästi, luokka antaa uuden toteutuksen piirteelle `equals`: kaksi joukkoa on `equals`-mielessä samat, jos ne sisältävät vastinalkiot, jotka ovat `equals`-mielessä samat. Hajautustaulua käyttävä `HashSet`, punamustapuuta käyttävä `TreeSet` ja **`enum`**-literaaliluokalle tarkoitettu `EnumSet` ovat `Set`-luokan konkreetteja perijöitä.

Esimerkki 8.2 Katsotaanpa listauksen 8.2 ohjelmaa, joka ilmoittaa niistä sanoista, jotka esiintyvät annetulla komentorivillä useaan kertaan ja tulostaa lopuksi erilaisten sanojen lukumäärän ja ko. sanat. Toteutus käyttää joukko-operaatioita rajapintaluokan kautta, joten konkreettia esitystapaa vaihdettaessa ainoastaan joukon luontioperaatiota joudutaan muuttamaan. Jos ohjelmassa tarvitaan jotakin erityiseen konkreettiin esitystapaan liityvää operaatiota tämä lähestymistapa ei toimi, mutta laskettaessa abstraktiotasoa, jonka kautta tietorakennetta käytetään, kiinnitetään ohjelma käyttämään vain ko. toteutusta. Ohjelma saadaan tulostamaan kaikki erilaiset sanat aakkosjärjestyksessä siirtymällä `HashSet`-joukon käytöstä `TreeSet`-joukkoon. Huomaa kuitenkin, että siirtyminen esitystavasta toiseen ei ole mitenkään itsestään selvää kaikissa tilanteissa. Luokka `TreeSet` käyttää ominaisuutta, jota joukkoalkioilta ei yleensä edellytetä: alkioita pitää pystyä vertailemaan keskenään. Javassa se tarkoittaa sitä, että alkiot osaavat käyttäytyä `Comparable`-rajapinnan mukaisesti (mikä on voimassa `String`-olioille).

8.2.4 Rajapinta `List`

Rajapintaluokan `List` esittämälle listarakenteelle on ominaista, että alkiot on järjestetty lineaarisesti, joten ne voidaan numeroida järjestyksessä nollasta eteenpäin. Tämän takia lista-alkioihin voidaan viitata indeksin avulla kuten taulukoihinkin.

Listaus 8.2: Luokka `Ainutlaatuistaja`.

```
import java.util.*;

public class Ainutlaatuistaja
  {
  public static void main(String... komentorivi)
    {
    Set<String> ainutlaatuisetSanat = new HashSet<String>();
    for (String sana : komentorivi)
      if (!ainutlaatuisetSanat.contains(sana))
        ainutlaatuisetSanat.add(sana);
      else
        System.out.println("Komentorivillä on duplikaatti: " + sana);
    System.out.println("Komentoriviltä löytyi " +
      ainutlaatuisetSanat.size() + " erilaista sanaa: " +
      ainutlaatuisetSanat);
    }
  }
```

Lisänä `Collection`-rajapintaan listalla on siten (a) operaatioita, joille voidaan antaa argumenttina yksittäinen indeksi tai indeksiväli, (b) hakuoperaatioita, joiden tulos ilmoitetaan listapositiona sekä (c) listajärjestystä hyödyntäviä läpikäyntioperaatioita. Rajapinnasta `Set` lista eroaa siinä, että listaan lisätään alkio aina olipa sillä esiintymä siellä jo ennestään tai ei.

Rutiini `addAll` saa tarkennetun semantiikan: se katenoi argumenttilistan nykyisen listan perään. Tarkemmin sanottuna `addAll` luo **`this`**-listan perään uusia solmuja, joihin kopioidaan viittaukset argumenttilistan solmuista. Tästä operaatiosta on myös ylikuormitettu versio, jolle annetaan se **`this`**-listan indeksi, josta eteenpäin argumenttilistan kaikki alkiot viedään (listan lopussa olevat solmut siirtyvät vastaavasti eteenpäin). Piirre `remove` puolestaan poistaa ensimmäisen (pienimmän indeksin omaavan) `equals`-mielessä samanlaisen alkion (itse asiassa koko solmun) listasta. Luokasta `Object` peritty `equals` on toteutettu uudelleen niin, että se tarkistaa onko tutkittavissa listoissa `equals`-mielessä samat alkiot samassa järjestyksessä.

Listasta voidaan ottaa pala tarkasteltavaksi rutiinilla `subList`, jolle annetaan argumentiksi puoliavoin indeksiväli (alaraja suljettu, yläraja avoin). Näin saatu osalista on kuitenkin vain *näkymä* alkuperäiseen listaan, sillä `subList` kopioi vain viittaukset ko. alilistaan. Alilistalla on kuitenkin oma (nollasta alkava) indeksinumerointinsa. Esimerkiksi jos listasta `lista` halutaan poistaa indekseissä 10–15

olevat alkiot, se voidaan tehdä käskyllä:

```
list.subList(10,16).clear();
```

Alilistan käytössä pitää kuitenkin olla varovainen, sillä muutokset (poistot, lisäykset) peruslistaan saattavat aiheuttaa alilistassa odottamattomia ongelmia. Tästä syystä alilistaa suositellaankin käytettäväksi vain hetkellisesti jonkin tietyn operaation apuna.

`ArrayList` on tehokas listarakenne, jota kannattaa käyttää useimmissa niistä tilanteista, joissa normaali taulukko ei käy. Periaatteessa `ArrayList` edustaa dynaamista taulukkoa: lisäykset ja poistot sujuvat vaivattomasti. Taulukkoon verrattuna (a) tilan loppumisesta ei tarvitse huolehtia lisäyksen yhteydessä ja (b) tietorakenteeseen talletettujen alkioiden todellinen määrä on aina tiedossa. `ArrayList`-luokan sisäisestä toteutuksesta johtuen listan loppuun kohdistuvat poistot ovat huomattavasti nopeampia kuin alkuun kohdistuvat. Luokka antaa mahdollisuuden määrätä listan alustavan kapasiteetin (so. sisäisen pituuden) luonnin yhteydessä. Kapasiteettia voidaan myöhemmin muuttaa, jos se on tarpeen.

Toinen listarakenteen konkreetti toteutus on `LinkedList`, joka käyttää kaksisuuntaista linkitettyä listaa. Tämä dynaaminen toteutus sopii paremmin niihin tapauksiin, jolloin listan alkioihin kohdistetaan paljon lisäys- ja poisto-operaatioita.

8.2.5 Rajapinta `Queue`

Rajapintaluokka `Queue` määrittelee FIFO-tyyppisen (*first-in, first-out*) kokoelman ja laajentaa `Collection`-rajapintaa viidellä rutiinilla. Rutiini `offer` lisää alkion jonoon ja palauttaa totuusarvon, onnistuiko lisäys vai ei. Jonossa ensimmäisenä oleva alkio saadaan `poll`-rutiinilla, joka samalla poistaa ko. alkio jonosta. Kyseinen rutiini palauttaa **`null`** mikäli jono on tyhjä, toisin kuin `remove`, joka nostaa poikkeuksen `NoSuchElementException` mikäli poisto ja palautus epäonnistuvat. Rutiini `peek` palauttaa (muttei poista) jonon ensimmäisen alkion (tai arvon **`null`** mikäli jono on tyhjä). Rutiini `element` toimii vastaavasti, mutta nostaa `NoSuchElementException`-poikkeuksen mikäli jono on tyhjä.

Rajapinnan konkreetti toteutus `PriorityQueue` toteuttaa prioriteettijonon keon avulla. Jotta alkiot voitaisiin priorisoida, niiden täytyy toteuttaa `Comparable`-rajapinta tai prioriteettijonoa konstruoidessa täytyy antaa järjestyksen määräävä `Comparator`-toteutus.

Toinen `Queue`-rajapinnan toteuttava luokka on `LinkedList`, joka varsinaisesti periytyy `List`-hierarkiasta mutta toteuttaa sen lisäksi jonomaisuuksia.

8.2.6 Rajapinta `Map`

Rajapintaluokka `Map` sisältää objekteja, joilla on avain- ja datakenttä. Kukin alkio määrittelee *kuvauksen* (*mapping*) avainarvolta data-arvoon. Tämän takia kokoelman avaimet ovat yksikäsitteisiä ts. kukin avainarvo voi esiintyä siinä vain kertaalleen. `Map`-luokan tarjoamien piirteiden kirjo on listauksen 8.3 mukainen.

Listaus 8.3: Rajapinta `Map`.

```
public interface Map<K,V>
  {
  //-- Perusoperaatiot.
  public V put(K key, V value);
  public V get(Object key);
  public V remove(Object key);
  public boolean containsKey(Object key);
  public boolean containsValue(Object value);
  public int size();
  public boolean isEmpty();

  //-- Kokoelmaoperaatiot.
  public void putAll(Map<? extends K,? extends V> t);
  public void clear();

  //-- Näkymät kokoelmiin.
  public Set<K> keySet();
  public Collection<V> values();
  public Set<Map.Entry<K,V>> entrySet();

  //-- Sisärajapinta Entry, joka mallintaa kokoelmaan
  //   talletettuja alkioita.
  public interface Entry<K,V>
    {
    public K getKey();
    public V getValue();
    public V setValue(V value);
    }
  }
```

Kokoelmaan voidaan tallettaa `put`-operaatiolla oikeantyyppisiä oliopareja. Jos `get`-operaatio ei löydä argumenttina annettua avainta kokoelmasta, se palauttaa

	Set	List	Queue	Map
Hajautustaulu	HashSet			HashMap
Mukautuva taulukko		ArrayList		
Punamustapuu	TreeSet			TreeMap
Linkitetty lista		LinkedList	LinkedList	
Keko			PriorityQueue	

Taulukko 8.1: Kokoelmarajapinnat ja niiden toteutukset tietorakenteina.

arvon **null**, muuten avainta vastaavan data-arvon. Operaatio `remove` poistaa argumenttina annettua avainarvoa vastaavan olioparin.

`Map`-rajapinnan implementoivalla luokalla tulisi olla aina konstruktori, joka saa argumentikseen `Map`-tyyppisen olion, jonka sisällöllä uusi olio alustetaan. Näin konversio erilaisten `Map`-kokoelmien välillä käy vaivattomasti. `Map`-kokoelmasta voidaan ottaa myös *siivuja*: `keySet` palauttaa kaikki käytetyt avaimet (joukkona, koska avaimet ovat yksikäsitteisiä), `values` puolestaan kaikki data-arvot `Collection`-tyyppisenä niin, että duplikaatit säilyvät. Piirre `entrySet` palauttaa kaikki (avain, arvo) -parit `Entry`-tyyppisinä objekteina.

`Map` toimii kuten lista siinä mielessä, että muutokset näillä rutiineilla saatuihin kokoelmiin implikoivat muutoksia myös itse `Map`-olioon. Itse asiassa `Map`-oliosta generoituja kokoelmia ei olisi syytä mennä koskaan muuttamaan, ne ovat vain tarkastelua varten. Esimerkiksi kahden `Map<Integer,String>`-olion `a` ja `b` yhteiset avainarvot saadaan selville kirjoittamalla

```
Set<Integer> yhteisetAvaimet = new HashSet<Integer>(a.keySet());
yhteisetAvaimet.retainAll(b.keySet());
```

Esimerkki 8.3 Listauksen 8.4 ohjelma laskee annettujen sanojen frekvenssit.[8]

8.2.7 Toteutuksista

Kootaanpa taulukkoon 8.1 nyt tieto rajapintaluokista ja niiden implementoinneista. Jos ainoa valintakriteeri on haun tehokkuus (nopeus), kannattaa rajapintoja vastaaviksi konkreeteiksi luokiksi valita `HashSet`, `ArrayList` ja `HashMap`. Jotta hajautustaulupohjaisista toteutuksista saataisiin irti mahdollisimman suuri hyöty, pitää ohjelmoijan tuntea hieman hajautusfunktioiden teoriaa (ks. kohta 6.5) ja antaa järkevä taulukon koko sekä täyttöaste luonnin yhteydessä. Yleissääntönä voidaan sanoa, että taulukon koon tulisi olla noin kaksinkertainen talletettavien alkioiden lukumäärään nähden ja sen olisi mielellään oltava alkuluku.

[8] Laiskempi voisi tietysti kurkata `Collections`-luokan rutiinia `frequency` — jos vain jaksaa...

Listaus 8.4: Luokka `Frekvenssi`.

```
import java.util.*;

public class Frekvenssi
  {
  public static void main(String... komentorivi)
    {
    Map<String,Integer> esiintymät = new HashMap<String,Integer>();
    for (String sana : komentorivi)
      {
      int määrä = (esiintymät.containsKey(sana) ?
                                          esiintymät.get(sana) : 0);
      esiintymät.put(sana, määrä + 1);
      }
    System.out.println("Erilaisia sanoja löytyi " +
                                       esiintymät.size() + " kpl.");
    System.out.println("Niiden frekvenssit ovat " + esiintymät);
    }
  }
```

Peruskokoelmaluokkien lisäksi Javassa on niukalti tukea erikoiskokoelmien käsittelyyn. Pinoluokka `Stack` (LIFO, *last-in, first-out*) löytyy jäämistöluokkana.[9] Jono `Queue` (FIFO, *first-in, first-out*) on sentään olemassa rajapintana, mutta hakupuita (moderneista tietorakenteista puhumattakaan) Javan kokoelmaluokat eivät valitettavasti tarjoa.

8.3 Iteraattorit

Iteraattori eli *läpikäyntiolio* (*iterator*) tarkoittaa menettelyä, jolla asiakas saa käsiinsä oliokokoelman alkiot yksitellen tarvitsematta puuttua yksityiskohtiin, joita palvelijaluokka tarvitsee siirtyessään kokoelmassa alkiosta seuraavaan. Tekniikalla pyritään siis piilottamaan läpikäyntiin liittyvät yksityiskohdat asiakkaalta. Tämän nojalla on selvää, että

- iteraattorit liittyvät luokkiin, jotka edustavat tavalla tai toisella alkiokokoelmaa, ja
- iteraattoria käytetään yleensä vain silmukan yhteydessä.

[9] Ja sellaiseksi sen saa kyllä jättääkin — voi hyvä isä, vektorista periytyvä pino...

Yleisesti tehtävänjako on se, että iteraattori on vastuussa alkioiden jonkun peräkkäisjärjestyksen generoinnista ja silmukan sisältävä asiakas alkioille tehtävistä toimenpiteistä. Iteraattorin toteutuksessa tulee huomioida se, että kokoelman läpikäynti voi olla joko osittaista (kokoelmaa käydään läpi, kunnes annettu ehto tulee todeksi tai epätodeksi) tai täydellistä (kaikki alkiot halutaan käsitellä). Kummassakin tapauksessa olisi toivottavaa, että pelkän havainnoinnin lisäksi alkioita pystyttäisiin myös päivittämään.

Listauksessa 8.5 esitelty rajapinta `Iterator` on Javan yleisin iteraattoriluokka. Rutiini `hasNext` kertoo, onko seuraavaa alkiota (johon edetä) olemassa ja `next` palauttaa seuraavan alkion. Operaatio `remove` (jota ei ole pakko toteuttaa) poistaa alkion, joka on viimeksi saatu käsiin piirteellä `next`. Tästä syystä `remove`-operaation kutsua pitää edeltää aina `next`-kutsu.[10]

Listaus 8.5: Rajapinta `Iterator`.

```
public interface Iterator<E>
  {
  public boolean hasNext();
  public E next();
  public void remove();      // Ei-pakollinen
  }
```

Rajapinnassa `Iterable` määritellään metodi `iterator`, joka palauttaa `Iterator`-rajapinnan toteuttavan iteraattoriolion. Mikäli kokoelmaa halutaan käydä läpi **for**-iteraattorisilmukassa, sen täytyy toteuttaa `Iterable`-rajapinta.

Esimerkki 8.4 Tyypillinen läpikäyntioperaatio on kokoelman alkioiden tulostaminen:

```
public static void tulostaAlkiot(Collection<?> kokoelma)
  {
  Iterator<?> it = kokoelma.iterator();
  while (it.hasNext())
    System.out.println(it.next());
  }
```

Dynaaminen sidonta huolehtii siitä, että tulostukseen käytettävä `toString`-operaatio valitaan olion dynaamisen tyypin mukaisesta luokasta. Tulostussilmukka voitaisiin tietysti kirjoittaa myös käyttämällä **for**-rakennetta:

```
for (Object alkio : kokoelma)
  System.out.println(alkio);
```

[10] Miksi Javan iteraattori on epäsymmetrinen siinä mielessä, että se sallii vain poistot, mutta ei lisäyksiä, on hämärän peitossa. Ehkä kauniimpaa olisi ollut, että poisto-operaatiokin olisi jätetty pois.

Esimerkki 8.5 Seuraava rutiini palauttaa kokoelman, jossa on argumenttina annetusta kokoelmasta vain ne alkiot jotka ovat aidosti pienemmät kuin annettu yläraja. Vertailua varten alkioiden täytyy toteuttaa `Comparable`-rajapinta.

```
public static <T extends Comparable<? super T>> Collection<T>
                      pienemmät(Collection<T> kokoelma, T yläraja)
  {
  Collection<T> tulos = new LinkedList<T>();
  for (T alkio : kokoelma)
    if (alkio.compareTo(yläraja) < 0) tulos.add(alkio);
  return tulos;
  }
```

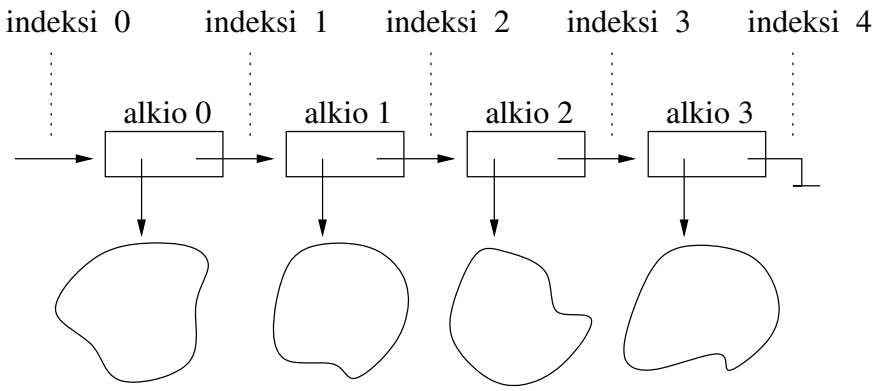
Esimerkki 8.6 Iteraattori-ideaa voidaan käyttää mihin tahansa tilanteeseen, jossa asiakas odottaa saavansa jonon olioita — siis vaikkapa lukujonojen generointiin. Jos jono on ääretön, piirre `hasNext` palauttaa aina arvon tosi. Satunnaislukujen generointiin voitaisiin käyttää seuraavaa iteraattoriluokkaa:

```
public class Satunnaisluku implements Iterator<Double>
  {
  private Random generaattori = new Random();

  public boolean hasNext() { return true; }
  public Double next() { return generaattori.nextDouble(); }
  public void remove() { throw new UnsupportedOperationException(); }
  }
```

Ainoa `Iterator`-rajapinnasta erikoistettu iteraattori — sekin rajapinta — on listoille tehty `ListIterator`, joka on esitelty listauksessa 8.6. Listaiteraattori tietää listojen lineaarisuusominaisuuden ja sallii liikkumisen kahteen suuntaan. Listaiteraattorilla on kaksi konstruktoria, joista toinen aloittaa läpikäynnin listan alusta ja toinen annetusta indeksistä. Läpikäynti toteutetaan ns. *kursorin* (*cursor*) avulla, joka osoittaa siihen kohtaan listaa, johon on edetty. Kuva 8.4 kertoo paikkanumeroinnin ja sen, että kursori on loogisesti aina kahden lista-alkion välissä. Piirre `nextIndex` (vastaavasti `previousIndex`) palauttaa seuraavan (edellisen) alkion indeksin. Sen lisäksi, että `next` ja `previous` palauttavat seuraavan ja edeltävän alkion, ne myös siirtävät kursoria sivuvaikutuksenaan. Tämä tarkoittaa mm. sitä, että `next`-kutsu ja heti sen jälkeen tehty `previous`-kutsu palauttavat saman lista-alkion. Listan alussa (lopussa) `previous` (vastaavasti `next`) palauttaa arvon **`null`**. Piirre `set` asettaa argumenttina annetun olion (viittauksen) siihen lista-alkioon, joka on viimeksi palautettu `next`- tai `previous`-operaatiolla. Piirre `add` (`remove`) lisää (poistaa) alkion listaan (listasta).

Listaus 8.6: Rajapinta `ListIterator`.

```
public interface ListIterator<E> extends Iterator<E>
  {
  public boolean hasNext();
  public E next();
  public boolean hasPrevious();
  public E previous();
  public int nextIndex();
  public int previousIndex();
  public void remove();           // Ei-pakollinen
  public void set(E o);           // Ei-pakollinen
  public void add(E o);           // Ei-pakollinen
  }
```



Kuva 8.4: Listaiteraattorin toiminta.

Tehtäviä

8-1 Mitä kokoelmarajapintaa (`Collection`, `List`, `Set`, `SortedSet`, `Map` tai `SortedMap`) käyttäisit seuraavissa tapauksissa? Entä minkä konkreetin kokoelmaluokan valitsisit toteutukseksi? Anna käyttöesimerkki kokoelman luonnista. Perustele valintasi.

(a) Tieto äänestäneistä `Äänioikeutettu`-olioista.

(b) Tieto vuoden aikana lapsille annetuista etunimistä.

(c) Urheilukilpailun lopputulos joka koostuu `Osallistuja`-olioista.

(d) Sanasto joka koostuu merkkijonoina annetuista termeistä ja niiden selityksistä.

8-2 Tutustutaanpa kuvan 8.2 kokoelmaluokkien rakennelmaan, jonka keskeisiä ominaisuuksia luku 8.2 esittelee. Tehtäväsi on miettiä miten luokat eroavat toisistaan ja mihin käyttötarkoitukseen luokkia voisi käyttää.

(a) Mitkä ovat seuraavien luokkien oleelliset erot?

(i) `Collection` ja `Map`

(ii) `List`, `Set` ja `Queue`

(iii) `ArrayList` ja `LinkedList`

(iv) `HashSet` ja `TreeSet`

(b) Mikä edellisistä luokista sopisi parhaiten tietorakenteeksi seuraaviin tapauksiin?

(i) Sovelluksen tehtävä on tallentaa sääasemalta kerättäviä lämpötilatietoja. Sovellus lukee anturilta lämpötilan ja tallentaa sen ohjelmamuistiin. Tämän sovelluksen ei tarvitse käsitellä lämpötiloja, vaan se antaa silloin tällöin kaiken tallentamansa datan eteenpäin järjestelmän seuraavalle ohjelmakomponentille.

(ii) Katsastusaseman tarkastushalliin halutaan sovellus, jonka avulla voidaan hallita katsastettavia ajoneuvoja. Ohjelman käyttäjää ei kiinnosta niinkään itse asiakas vaan hänen menopelinsä. Ajoneuvo tunnistetaan yksikäsitteisesti sen rekisterinumerosta. Minkä kokoelmaluokan valitsisit tilanteeseen?

8-3 `LukuJoukko`-luokan konstruktorin `LukuJoukko(Collection<Integer> k)` alkuehto vaatii, että kokoelmassa `k` ei saa olla duplikaatteja. Tämä alkuehto voidaan poistaa, jos konstruktori toteutetaan siten, että `k`-rakenteen tilalle luodaan sellainen uusi konstruktorin sisäinen rakenne, jossa ei esiinny duplikaatteja.

Toteuta tätä varten funktio

```
Collection<Integer> duplikaatittomana(Collection<Integer> k)
```

joka muodostaa ja palauttaa duplikaattoman version `k`-rakenteesta.

8-4 Rajapinta `Sanamuodostin` määrittelee muodostimen, joka tuottaa sanoja tiettyjen sääntöjen avulla. Rajapinnan ainoa vaatimus on metodi `muodosta(int i)`, joka palauttaa muodostimen tuottaman `i`:nnen sanan.

```
/** Rajapinta sanamuodostimille. */
public interface Sanamuodostin {
   /**
    *  Palauttaa i:nnen sanan.
    *  @.pre  i >= 0
    *  @.post FORALL(j : 0 <= j < i; !muodosta(i).equals(muodosta(j)))
    */
   public String muodosta(int i);
   }
```

Yksinkertainen esimerkki sanojen tuottamisesta on eritasoisten käskyhuudahdusten generointi. Jos käskysanan runkoa merkitään r:llä, parittoman tason sanapäätteen osaa a:lla ja parillisen tason päätteen osaa b:llä, voidaan tason i käsky $k(i)$ tuottaa merkkijonokatenaatiolla '+' seuraavasti: $k(i) = r + p(i)$, missä pääte $p(i)$ määritellään seuraavasti:

$$p(i) = \begin{cases} \texttt{""}, & \text{jos } i = 0 \\ a + p(i-1), & \text{jos } i > 0 \text{ ja } i \text{ on pariton, ja} \\ b + p(i-1), & \text{jos } i > 0 \text{ ja } i \text{ on parillinen.} \end{cases}$$

Esimerkiksi jos $r =$`"vedä"`, $a =$`"tä"` ja $b =$`"ty"`, voidaan sääntöjen avulla tuottaa käskyt `"vedä"`, `"vedätä"`, `"vedätytä"`, `"vedätätytä"`, `"vedätytätytä"`, `"vedätätytätytä"` jne.

(a) Kirjoita `Sanamuodostin`-rajapinnan toteuttava luokka `Käskymuodostin`, jolla voidaan tuottaa eritasoisia käskyjä em. sääntöjen mukaisesti. Luokan konstruktori `Käskymuodostin(String r, String a, String b)` saa syötteenään tuottamiseen tarvittavat merkkijonot ja metodi `muodosta(int i)` palauttaa `i`:nnen tason käskyn $k(i)$.

(b) Kirjoita `Set`-rajapinnan toteuttava luokka `Sanajoukko`, joka sisältää `määrä` ensimmäistä sanamuodostimella `muodostin` tuotettua sanaa. Luokassa olevan konstruktorin signatuuri on siis `Sanajoukko(Sanamuodostin muodostin, int määrä)`. Toteuta `Set`-rajapinnan pakolliset metodit `size`, `isEmpty`, `contains`, `containsAll`, `iterator` ja `toArray`.

8-5 `Map`-kokoelman sisältö voidaan ymmärtää matemaattisten kuvausparien $x \mapsto y$ joukkona, missä "avain" x kuvataan "arvoksi" y. Tällöin sanotaan, että arvo y on avaimen x *kuva* ja vastaavasti, että avain x on arvon y *kantakuva*. Toteuta luokkaan `HashMap` tai `TreeMap` (tai jommastakummasta johtamaasi omaan luokkaan) metodi, joka palauttaa kaikki annetun arvon kantakuvat (kokoelmana). Mieti lisäksi, miten toteutustasi voisi tehostaa, jos tiedetään, että jokaisella arvolla on täsmälleen yksi kantakuva.

8-6 Pakkauksen `java.util` luokissa `Arrays` ja `Collections` on muutamia mukavia metodeja. Tarkastelepa `Collections`-luokan metodin `sort(List, Comparator)` kuvausta API-dokumentista. Toteuta sen jälkeen alla olevassa `DilSorter`-luokassa kommentteina oleviin kohtiin (a) ja (b) alkioiden lajittelu käyttäen tätä `sort`-metodia.

```
import java.util.*;

public class DilSorter {
    public static final String[] STAFF = {
        "Dilbert", "Alice", "Wally",
        "LOUD HOWARD", "Asok the intern",
        "pointy-haired boss" };

    public static void main(String[] args) {
        List<String> staffList = Arrays.<String>asList(STAFF);
        System.out.println(staffList);

        // (a) Lajittele listan alkioina olevat merkkijonot
        //     leksikografiseen järjestykseen (so. aakkosjärjestykseen)
        //     välittämättä isoista ja pienistä kirjaimista.

        System.out.println(staffList);

        // (b) Lajittele listan alkioina olevat merkkijonot
        //     ei-laskevaan pituusjärjestykseen.

        System.out.println(staffList);
    }
}
```

8-7 Kirjoita toteutus abstraktille iteraattoriluokalle `Kokonaislukuiteraattori`, joka toteuttaa rajapinnan `Iterator<BigInteger>`. Kyseinen iteraattori generoi kokonaislukujonoja käyttäen äärettömän tarkkuuden kokonaislukuina `java.math`-pakkauksen `BigInteger`-luokkaa. Jonon kaksi ensimmäistä alkiota on kiinnitetty ($x_1 = n$, $x_2 = m$) ja muut saadaan soveltamalla abstraktia metodia

```
protected abstract BigInteger laske(BigInteger edellinenArvo,
                                    BigInteger nykyinenArvo);
```

jonon kahteen edelliseen termiin: $x_i =$ `laske`(x_{i-2}, x_{i-1}).

Peri edellä määritellystä iteraattoriluokasta konkreetti, Fibonaccin lukujonon generoiva luokka `FibonacciIteraatori`, jossa $n = 0$, $m = 1$ ja `laske` on yhteenlasku. Kirjoita myös sen toteutus.

8-8 Toteuta luokasta `ArrayList<E>` johdetulle luokalle `Aineisto<E>` iteraattorin palauttava metodi

```
public Iterator<E> näyte(int k)
```

sekä tarvittava iteraattoriluokka. Iteraattorin on tarkoitus antaa `k`:n alkion *palauttava otos* annetusta kokoelmaoliosta, eli se palauttaa `k` satunnaista alkiota kokoelmasta. Mieti myös miten voisi toteuttaa *palauttamattoman otoksen*, jossa samaa kokoelman alkiota ei saa palauttaa kuin enintään kerran.

8-9 `Iterator<E>`-rajapinta mallintaa peräkkäissaantiabstraktion eikä sen tarvitse liittyä minkään olemassaolevan tietorakenteen läpikäyntiin. Tämän havainnon kannustamana tehtäväsi on täydentää seuraava `Rengasiteraattori`-määrittely

(a) alku- ja loppuehdoilla sekä

(b) rutiinien toteutuksilla.

(c) Miksi `Rengasiteraattori` toteuttaa `Iterator`-rajapinnan lisäksi myös `Iterable`-rajapinnan? Keksi perusteluksi jokin yksinkertainen käyttöesimerkki.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Rengasiteraattori<E> implements Iterable<E>, Iterator<E> {

   //-- Rajapinnan Iterable<E> toteutus.

   /** Palauttaa rengasiteraattorin. */
   public Rengasiteraattori<E> iterator() { /* Toteuta. */ }

   //-- Rajapinnan Iterator<E> toteutus.

   private E[] rengas;
   private int nykykohta;

   /** Alustaa iteraattorin toistamaan taulukon r alkioita
    *  alusta alkaen päättymättömästi siten että viimeisen
    *  r:n alkion jälkeen palauttaminen jatkuu aina ensimmäisestä
    *  alkiosta.
    */
   public Rengasiteraattori(E... r) { /* Toteuta. */ }

   /** Katso java.util.Iterator<E>. */
   public boolean hasNext() { /* Toteuta. */ }

   /** Katso java.util.Iterator<E>. */
   public E next() { /* Toteuta. */ }
```

```
 /** Katso java.util.Iterator<E>. */
 public void remove() { /* Toteuta. */ }
 } //Rengasiteraattori
```

8-10 Oletetaan että järjestelmässä on määritelty rajapinta `Kasa` ja sen toteuttava konkreetti luokka `KonkreettiKasa` seuraavasti:

```
public interface Kasa<T>
  {
  /**
   *  @.pre  true
   *  @.post RESULT == (kasassa olevien olioiden määrä)
   */
  public abstract int annaKoko();

  /**
   *  @.pre  olio != null
   *  @.post RESULT == (olio on equals-mielessä kasassa)
   */
  public abstract boolean onMukana(T olio);

  /**
   *  @.pre  olio != null && !onMukana(olio)
   *  @.post RESULT.onMukana(olio)
   */
  public abstract void lisää(T olio);

  /**
   *  @.pre  olio != null && onMukana(olio)
   *  @.post !RESULT.onMukana(olio)
   */
  public abstract void poista(T olio);
  } // Kasa

public class KonkreettiKasa<T> implements Kasa<T>
  {
  // Jokin soveltuva toteutus.
  } // KonkreettiKasa
```

Oletetaan lisäksi että suurin osa järjestelmän tämänhetkisestä ohjelmakoodista viittaa luokasta `KonkreettiKasa` valettuihin olioihin `Kasa`-liittymän kautta. Järjestelmän seuraavassa versiossa rajapinnalle `Kasa` tulee kuitenkin sellaisia uusia asiakkaita, joiden on myös kyettävä käymään kaikki tällaisen rakenteen sisältämät oliot läpi yksitellen.

Tehtävänäsi on ratkaista näin syntynyt ongelma määrittelemällä ja toteuttamalla läpikäyntiolio eli iteraattori. Muodosta tarvittavalle iteraattorille rajapintamäärittely. Täydennä määrittelyjä `Kasa` ja `KonkreettiKasa` tarpeen mukaan. Koska luokan `KonkreettiKasa` toteutusta ei ole annettu, sinun on kuvailtava yleisellä tasolla kuinka konkreettisen iteraattoriolion rutiinit toteutetaan. Piirrä kuva olioista selityksesi tueksi. Anna lyhyt esimerkki kuinka kaikki rakenteen sisältämät oliot läpikäyvä asiakas käyttää uudistunutta `Kasa`-liittymää.

8-11 Oletetaan että käytössämme on seuraava intervallin toteuttava geneerinen luokka:

```
public class Intervalli<T extends Comparable<? super T>> {
  //-- Jäsenmuuttujat
  /** Intervallin alaraja. */
  private T alaraja;
  /** Intervallin yläraja. */
  private T yläraja;
  /** Onko alaraja suljettu (eli kuuluuko se intervalliin). */
  private boolean alarajaSuljettu;
  /** Onko yläraja suljettu (eli kuuluuko se intervalliin). */
  private boolean ylärajaSuljettu;

  //-- Konstruktorit
  /** Alustaa intervallin.
   *  @.pre  (alaraja != null & yläraja != null) &&
   *         alaraja.compareTo(yläraja) <= 0
   *  @.post annaAlaraja().equals(alaraja) &
   *         annaYläraja().equals(yläraja) &
   *         onAlarajaSuljettu() == alarajaSuljettu &
   *         onYlärajaSuljettu() == ylärajaSuljettu
   */
  public Intervalli(T alaraja, T yläraja,
                    boolean alarajaSuljettu, boolean ylärajaSuljettu) {
    this.alaraja = alaraja;
    this.yläraja = yläraja;
    this.alarajaSuljettu = alarajaSuljettu;
    this.ylärajaSuljettu = ylärajaSuljettu;
    }

  //-- Havainnointioperaatiot
  /** Palauttaa intervallin alarajan.
   *  @.pre  true
   *  @.post RESULT == (alaraja)
   */
  public T annaAlaraja() { return alaraja; }
```

```
/** Palauttaa intervallin ylärajan.
 *  @.pre  true
 *  @.post RESULT == (yläraja)
 */
public T annaYläraja() { return yläraja; }

/** Palauttaa onko intervallin alaraja suljettu.
 *  @.pre  true
 *  @.post RESULT == (alaraja on suljettu)
 */
public boolean onAlarajaSuljettu() { return alarajaSuljettu; }

/** Palauttaa onko intervallin yläraja suljettu.
 *  @.pre  true
 *  @.post RESULT == (yläraja on suljettu)
 */
public boolean onYlärajaSuljettu() { return ylärajaSuljettu; }

/** Tarkistaa onko intervalli tyhjä.
 *  @.pre  true
 *  @.post RESULT == annaAlaraja().compareTo(annaYläraja()) == 0 &
 *                   !(onAlarajaSuljettu & onYlärajaSuljettu())
 */
public boolean onTyhjä() {
  return annaAlaraja().compareTo(annaYläraja()) == 0 &
         !(onAlarajaSuljettu() & onYlärajaSuljettu());
  }

/** Vertaa alarajaa annettuun arvoon.
 *  @.pre  a != null
 *  @.post RESULT ==
 *     ((onAlarajaSuljettu() | annaAlaraja().compareTo(a) != 0) ?
 *                         annaAlaraja().compareTo(a) : 1)
 */
public int vertaaAlarajaan(T a) {
  return ((onAlarajaSuljettu() | annaAlaraja().compareTo(a) != 0) ?
             annaAlaraja().compareTo(a) : 1);
  }

/** Vertaa ylärajaa annettuun arvoon.
 *  @.pre  a != null
 *  @.post RESULT ==
 *     ((onYlärajaSuljettu() | annaYläraja().compareTo(a) != 0) ?
 *                         annaYläraja().compareTo(a) : -1)
```

```
 */
public int vertaaYlärajaan(T a) {
  return ((onYlärajaSuljettu() | annaYläraja().compareTo(a) != 0) ?
            annaYläraja().compareTo(a) : -1);
  }

/** Tarkistaa sisältyykö annettu arvo intervalliin.
 *  @.pre  a != null
 *  @.post RESULT == (vertaaAlarajaan(a) <= 0) &
 *                   (vertaaYlärajaan(a) >= 0)
 */
public boolean sisältää(T a) {
  return (vertaaAlarajaan(a) <= 0) & (vertaaYlärajaan(a) >= 0);
  }

//-- Arvosemanttiset operaatiot
/** Tarkistaa intervallien samuuden.
 *  @.pre  true
 *  @.post RESULT == (intervallit rajoittavat saman arvoalueen)
 */
@SuppressWarnings("unchecked")
public boolean equals(Object toinen) {
  if (toinen == null) return false;
  if (toinen == this) return true;
  if (!getClass().equals(toinen.getClass())) return false;
  Intervalli<T> vertailtava = (Intervalli<T>)toinen;
  return (this.alaraja.compareTo(vertailtava.alaraja) == 0 &
          this.yläraja.compareTo(vertailtava.yläraja) == 0 &
          this.alarajaSuljettu == vertailtava.alarajaSuljettu &
          this.ylärajaSuljettu == vertailtava.ylärajaSuljettu);
  }

/** Palauttaa intervallin hajautustauluosoitteen.
 *  @.pre  true
 *  @.post RESULT == (hajautustauluosoite)
 */
public int hashCode() {
  int tulos = 3677 * this.getClass().hashCode();
  tulos = (1223 * tulos) + alaraja.hashCode();
  tulos = (1223 * tulos) + yläraja.hashCode();
  tulos = (1223 * tulos) + (alarajaSuljettu ? 7561 : 5153);
  tulos = (1223 * tulos) + (ylärajaSuljettu ? 6029 : 2647);
  return tulos;
  }
```

```
/** Palauttaa intervallin merkkijonoesityksen
 *  @.pre  true
 *  @.post RESULT == (merkkijonomuotoinen esitys)
 */
public String toString() {
  return (alarajaSuljettu ? "[" : "(") + alaraja + ", " + yläraja +
         (ylärajaSuljettu ? "]" : ")");
  }
} // class Intervalli
```

(a) Monet intervallit ovat diskreettejä, jolloin siihen kuuluvia alkioita on rajallinen määrä. Tällöin intervallin alkiot voidaan käydä läpi iteraattorilla. Toteuta abstraktiin geneeriseen luokkaan `DiskreettiIntervalli` ne piirteet jotka ovat luonnollisia tälle asiayhteydelle:

```
public abstract class
              DiskreettiIntervalli<T extends Comparable<? super T>>
                       extends Intervalli<T> implements Iterable<T>
  {
  /** Palauttaa alkion a seuraajan intervallissa.
   *  @.pre  sisältää(a)
   *  @.post (RESULT == a:n seuraaja) ||
   *         (nostaa poikkeuksen Exception jos !sisältää(RESULT))
   */
  public abstract T seuraaja(T a) throws Exception;

  //--> Laadi luokan loppuosan toteutus!
  }
```

(b) Yksi diskreetin intervallin konkreeteista toteutuksista muodostuu kokonaisluvuista (vrt. tehtävä 2-15). Laadi toteutus kokonaislukujen intervallia mallintavalle luokalle `KokonaislukuIntervalli`.

8-12 Myöhään heränny sijoitusyhtiö Moctod on huomannut että web-hakukoneissa on tulevaisuus ja mahdollisuus suuriin voittoihin. Koska yhtiön puuhahenkilöt ovat kuulleet sinun osaavan ohjelmoida, he kääntyvät puoleesi keskeisen teknisen toteutuksen aikaansaamiseksi. Hakukoneen moottoriksi tarvittaisiin rakenne, jossa web-sivulta poimitut avainsanat (`String`-olioita) kuvataan joukkoon www-osoitteita (`java.net.URL`-olioita).

Tarkemmin asiaa ajateltuasi huomaat kyseessä olevan indeksoinnin (hakusanat viittavat joukkoon web-osoitteita). Tämän toiminnallisuuden toteuttavaa luokkaa voitaisiin vähäisin muutoksin käyttää vaikkapa kirjan hakemiston toteuttamiseen (so. `String`-oliot kuvautuvat `Integer`-oliojoukkoon). Koska olet realisti ja katsot tulevaisuuteen (joka on alati epävarma), päätät varmistaa selustasi tekemällä geneerisen

`Hakemisto<A,V>`-luokan, josta voidaan tarvittaessa erikoistaa `Hakemisto<String,URL>` — mikäli Moctodin sijoittajat vielä neuvottelujen jälkeen haluavat hakukonetta.

Mieti mitä piirteitä kuuluu `Hakemisto`-luokan julkiseen liittymään ja laadi niille alku- ja loppuehdot. Valitse sen jälkeen luokalle konkreetti totetus ja laadi määrittelemillesi rutiineille toteutukset.

Vinkki: Luokkaa voitaisiin käyttää seuraavalla tavalla:

```
Hakemisto<String,String> h = new Hakemisto<String,String>();
h.lisää("kissa", "naukuva");       h.lisää("kissa", "nelitassuinen");
h.lisää("kissa", "kehräävä");      h.lisää("koira", "haukkuva");
h.lisää("koira", "nelitassuinen"); h.lisää("rukki", "kitisevä");
h.lisää("rukki", "kehräävä");      h.lisää("pomo", "kaksijalkainen");
h.lisää("pomo", "kitisevä");       h.lisää("pomo", "haukkuva");
System.out.println(h);
System.out.println(h.annaKaikkiAvaimet());
System.out.println(h.annaKaikkiViittaukset());
System.out.println("sisältää avaimen 'rukki': " +
                   h.sisältääAvaimen("rukki"));
h.poista("rukki");
System.out.println(h);
System.out.println("sisältää avaimen 'rukki': " +
                   h.sisältääAvaimen("rukki"));
System.out.println("sisältää viittauksen 'haukkuva': " +
                   h.sisältääViittauksen("haukkuva"));
System.out.println("sisältää viittauksen 'ammuva': " +
                   h.sisältääViittauksen("ammuva"));
```

Tällöin suorituksen tuloksena saataisiin seuraava tulostus (jollakin tavalla muotoiltuna):

```
{rukki=[kitisevä, kehräävä], kissa=[naukuva, kehräävä,
nelitassuinen], pomo=[kitisevä, haukkuva, kaksijalkainen],
koira=[haukkuva, nelitassuinen]}
[rukki, kissa, pomo, koira]
[kitisevä, haukkuva, naukuva, nelitassuinen, kehräävä,
kaksijalkainen]
sisältää avaimen 'rukki': true
{kissa=[naukuva, kehräävä, nelitassuinen], pomo=[kitisevä,
haukkuva, kaksijalkainen], koira=[haukkuva, nelitassuinen]}
sisältää avaimen 'rukki': false
sisältää viittauksen 'haukkuva': true
sisältää viittauksen 'ammuva': false
```

Luku 9

Asiakas- ja periytymisrelaatioista

Kun tarkastellaan ohjelmistojärjestelmää luokkatasolla, huomataan että luokkien välillä vallitsee kahdenlaisia (tai Javassa kolmenlaisia) suhteita: asiakas-toimittaja -suhde ja periytyminen (sekä Javassa esiintymiskohtainen sisäluokka). Nämä mekanismit riittävät hyvin luokkien liittämiseen toisiinsa, ja ongelmana onkin pikemminkin se, kuinka niitä käytetään luovasti hyväksi. Vaikka tämän asian tarkempi pohdiskelu (esim. suunnittelumallit) jää tämän kurssin ulkopuolelle, tarkastellaanpa lyhyesti näiden luokkien välisten suhteiden periaatteellisia ominaisuuksia sekä muutamia Javaan liittyviä esimerkkitapauksia.

9.1 Suunnittelijan näkökulma

Kun luokat on valittu, systeemikokonaisuus hahmotellaan muodostamalla relaatioita luokkien välille. Valinta periytymisen ja asiakasrelaation välillä saattaa olla vaikea ja ratkaisu riippuukin usein mallintajan näkökulmasta asiaan. Yleisesti ottaen voidaan kuitenkin havaita, että luokat käyttävät toisiaan pääsääntöisesti asiakasrelaation kautta ja periytymistä nähdään harvoin, lähinnä pitkällisen analyysin jälkeen, kuten erilaisten perustyökaluluokkien yhteydessä.

Jos on vaikea ratkaista pitääkö tietyssä tilanteessa käyttää asiakas- vai periytymisrelaatiota, on hyvä muistaa, että periytyminen sidotaan vahvasti tyyppikäsitteeseen. Siksi periytymisrelaatiota kuvataan englanninkielisessä kirjallisuudessa termillä *is-a*, jolla halutaan viestittää, että kaikkia yliluokan operaatioita voidaan kohdistaa myös perijäluokan olioihin (korvaavuusperiaate): suutari *on eräänlainen* työntekijä. Vastaavasti asiakassuhteesta käytetään termiä *has-a*: suutari *omistaa* naskalin. Jotta ei tulisi kömmähdyksiä, on syytä painottaa, että *is-a* ei tarkoita sitä, että esimerkiksi luokka `Turku` olisi luokan `Kaupunki` perijä. Nämä käsitteet

eivät nimittäin juuri missään käyttöyhteydessä ole samantasoisia: toinen on olio, toinen luokka. Älä siis tee sitä virhettä, että vertailisit toisiinsa oliota ja luokkaa vaan vertaile aina luokkia keskenään. Asian varmistamiseksi voit termin *is-a* sijasta kokeilla myös sanontoja *is-a-kind-of* tai *can-act-as-a*.

Valintapäätöksen tueksi voidaan antaa myös seuraava ohjenuora: *omistaminen (has-a) on harvoin olemista (is-a), mutta useassa tapauksessa oleminen on myös omistamista.* On esimerkiksi selvää, ettei luokan `AutonOmistaja` tule periä `Henkilö`- ja `Auto`-luokilta, vaan käyttää `Auto`-luokkaa asiakkaana. Aina asia ei ole kuitenkaan yhtä yksioikoinen, kuten edellisen säännön loppuosa antaa ymmärtääkin. Ajatellaanpa lausetta "Jokainen sähköinsinööri on insinööri". OO-periaatteet sisäistäneen ohjelmoijan ensimmäinen reaktio on: tämähän on selvä tapaus: kyseessä on normaali *is-a* -relaatio, joten `Sähköinsinööri`-luokka on `Insinööri`-luokan perijä, sen erikoistunut versio — ja tähän suuntaan aiemmat esimerkkimmekin ovat vahvasti ohjanneet. Alkuperäinen toteamus voidaan kuitenkin lausua myös muodossa "Jokaisessa sähköinsinöörissä asuu pieni insinööri", ja ajatella asiaa niin, että jokaisella ihmisellä on ominaisuuksia, joiden summana hänen persoonallisuutensa määräytyy. Näkökulman vaihdon ansiosta *is-a* saatiin muutettua *has-a* -relaatioksi. Vaikka esimerkki onkin melko abstrakti (ohjelmointityötä ajatellen), voidaan sama muunnos tehdä lähes jokaisen *is-a* -relaation kohdalla.

Luokkien väliset periytymisrelaatiot voidaan rakentaa monella eri tavoin aina katsantokannasta riippuen. Voidaankin sanoa, että ei ole olemassa mitään yhtä ja ainoaa oikeaa hierarkiaa, vaan asiat täytyy punnita aina tilannekohtaisesti niiden vaatimusten pohjalta jotka luokkarakennelmassa tarkasteluhetkellä vallitsevat. OO-kieliä tukevat laajat kirjastot antavat yleensä hyvän kuvan siitä filosofiasta, jolla kielessä asiat on ajateltu. Näin kirjastot toimivat hyvänä esimerkkinä ohjelmoijalle, joka haluaa rakentaa omia luokkakokonaisuuksia. Kokonaan toinen juttu sitten on se, että periytymistä käytetään moneen muuhunkin tarkoitukseen kuin puhtaaseen alityypitykseen. Tällöin tuloksena saattaa olla hyvinkin spagettimaista verkostoa luokkien välille ja jos rakentamisessa ei ole käytetty mitään selkeää punaista lankaa, luokkien ymmärtäminen ja uudelleenkäyttö hankaloituu oleellisesti.

Käytetty OO-kieli ohjaa voimakkaasti koko periytymishierarkian muotoutumista. Java tukee vain yksittäisperiytymistä, joten tuloksena on helposti pitkiä periytymisketjuja ja alemmilla hierarkiatasoilla on näin ollen "liian lihavia" luokkia, ts. ne sisältävät paljon sellaisiakin piirteitä, joita siellä ei tarvita (mutta jotka ylemmät luokat ovat antaneet perijöiden käyttöön, koska eivät ole uskaltaneet sulkea niitä poiskaan). Ketjumainen hierarkiaosa saattaa olla merkki myös huonosta suunnittelusta: jos luokalla ei ole kuin yksi perijä, on syytä kyseenalaistaa sen olemassaolo. Samoin jos luokan julkinen liitäntä sisältää vain yhden piirteen, kyseessä saattaa olla työkalurutiini, eikä luokka siinä mielessä kuin se OO-ajattelussa mielletään. Nämä ovat mallintamisongelmia eivätkä sinänsä liity taustalla olevaan

OO-kieleen. On kuitenkin havaittavissa, että moniperiytyminen rohkaisee ajatteluun, jolla nämä karikot vältetään helpommin kuin yksittäisperintäisissä kielissä. Moniperiytymisellä on myös se etu, että periytymishierarkiasta voidaan valita aina sopivat ”siivut” luokan käyttöön, joten siitä luodut oliot käyttävät vähemmän muistitilaa sekä rutiinikutsut kohdistuvat vähempään määrään olioita, mikä puolestaan näkyy ajoaikana suorituksen nopeutumisena. Toisaalta moniperiytymisen myötä joudutaan helposti sellaisiin mallinnusongelmiin, joihin yksittäisperiytymisessä ei koskaan päädytä.

9.2 Toteuttajan näkökulma

Ohjelmiston rakentajan kannalta periytymisen mukanaantuoma mahdollisuus tarkentaa olemassaolevia luokkia askeleittain on merkittävä. Periytyminen luo luokkien välille verkoston, joka ilmaisee implementointiin liittyvät loogiset yhteydet. Periytymisen avulla yhteenkuuluvien tietoabstraktioiden samankaltaisuudet voidaan sijoittaa ”saman katon alle”. Uudelleenkäyttö maksimoidaan viemällä jokaisen piirteen määrittely yleisimpään yliluokkaan, jossa sillä on oma tehtävänsä (”kuuluu luokkaan”). Tällöin niiden toisto jää pois mahdollisimman monesta periytyjästä.

Asiakasrelaation etuna on riippumattomuus palvelijaluokasta ja tämä etu on todella merkittävä, koska se rajaa ohjelmistoon tehtävät muutokset pienelle alueelle. Tämä on peruslähtökohta koko OO-ajattelulle ja sen merkitystä ei ole syytä väheksyä. Periytymisrelaatioon päädytään toisaalta helposti siksi, että sen avulla luokan implementointi tulee yksinkertaisemmaksi. Tämä ei tietysti kelpaa syyksi oikeaoppiselle ohjelmoijalle, sillä asiakasrelaation aiheuttama pieni lisätyö luokan toteutusvaiheessa maksaa itsensä takaisin myöhemmin moninkertaisesti verrattuna siihen, että olisi päädytty periytymiseen ja kyseistä luokkaa jouduttaisiin myöhemmin muuttamaan. Kun luokka on selvästi toisen alityyppi, on tietysti syytä käyttää periytymistä; se säilyttää tärkeän kosketuksen yliluokan toteutukseen ja tuo mukanaan polymorfismin ja dynaamisen sidonnan tarjoamat edut. Jos esimerkiksi `Sähköinsinööri`, `Laivainsinööri`, `LVIinsinööri` yms. -luokat periytyvät `Insinööri`-luokasta, voidaan ohjelmassa esitellä `Insinööri`-tyyppinen tunniste ja liittää siihen tarpeen tullen erilaisia insinöörejä, kaikenlaiset insinöörit voidaan laittaa `Insinööri[]`-tyyppiseen taulukkoon jne. Jos `Sähköinsinööri`-luokka (ja muutkin erikoistuneet insinööriluokat) on toteutettu asiakasrelaation avulla (esittelemällä `Insinööri`-tyyppinen jäsenmuuttuja), insinööreillä ei ole muuta kokoavaa sateenvarjoluokkaa kuin esimerkiksi `Henkilö` (tai `Object`), joten insinööreihin kohdistuvat toimenpiteet eivät näy ohjelmassa selvästi vaan sekoittuvat muuhun henkilötietoja käsittelevään koodiin.

Yhteenvetona relaatioiden eduista ja haitoista voidaan esittää taulukko 9.1. Yleisesti voidaan sanoa, että tilanteessa, jossa kumpikin vaihtoehto näyttää käyt-

Tarkasteltava ominaisuus	Asiakasrelaatio	Periytymisrelaatio
Informaation piilottaminen	Toteutuu	Ei yleensä toteudu
Luokkien välinen riippumattomuus	Saavutetaan	Ei saavuteta
Polymorfismi ja dynaaminen sidonta	Ei toimi	Toimii
Luokan toteuttaminen	Turhaa painolastia	Yksinkertainen

Taulukko 9.1: Luokkien välisten relaatioiden edut ja haitat.

tökelpoiselta, kannattaa ensin harkita asiakasrelaation käyttöä. Seuraavat säännöt antavat konkreetimman pohjan valinnalle:

- Jos luokan `B` jokaisella esiintymällä on tyyppiä `A` oleva komponentti, jota mahdollisesti joudutaan muuttamaan ohjelman suoritusaikana erityyppiseksi, tee luokasta `B` luokan `A` asiakas.
- Jos tyypin `A` mukaisten tunnisteiden on tarve viitata `B`-tyyppisiin olioihin tai kokoelmarakenteeseen täytyy laittaa sekaisin `A`- ja `B`-tyyppisiä olioita, tee luokasta `B` luokan `A` perijä.

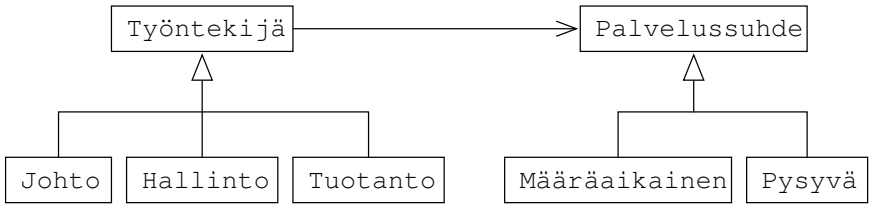
9.3 Esimerkkejä

Käydäänpä seuraavaksi esimerkkien avulla läpi perustapoja, joilla periytymistä ja asiakasrelaatiota voidaan soveltaa usein eteentulevien ongelmien ratkaisuun. Ensimmäinen esimerkki on suunnitteluun liittyvä ja kaksi seuraavaa sijoittuvat suunnittelun ja toteutuksen välimaastoon.

9.3.1 Poissulkeva luokittelu

Tarkastellaan yksinkertaista suunnittelumallia, joka yhdistää asiakas- ja periytymisrelaatiot mielenkiintoisella tavalla. Oletetaan, että yhtiö palkkaa palvelukseensa työntekijöitä. Palvelussuhteita on kahta eri tyyppiä: määräaikaiset ja pysyvät. Työntekijät voidaan jaotella myös työkuvansa perusteella tuotanto-, hallinto- ja johtotehtäviin. Mallintamisen ensimmäinen vaihe on miettiä, miten nämä kaksi kriteeriä (palvelussuhde ja työnkuva) liittyvät toisiinsa ja pitääkö ne asettaa jollain tavoin toisilleen alisteisiksi.

Suoraviivaisin vaihtoehto on tehdä luokista `Pysyvä`, `Määräaikainen`, `Tuotanto`, `Hallinto` ja `Johto` luokan `Työntekijä` välittömiä periytyjiä. Tämä ei kuitenkaan ole luonteva ratkaisu, koska järkevästi rakennetun periytymishierarkian yhdellä tasolla ei ole paljon toisistaan poikkeavia käsitteitä. Toinen mahdollisuus on valita `Työntekijä`-luokan perijöiksi palvelussuhdetta ja työnkuvaa edustavat luokat `Palvelussuhde` ja `Työnkuva`. Luokan `Palvelussuhde` alle laitetaan luokat `Pysyvä` ja



Kuva 9.1: Esimerkki poissulkevan luokittelun toteuttamisesta olioviittauksella.

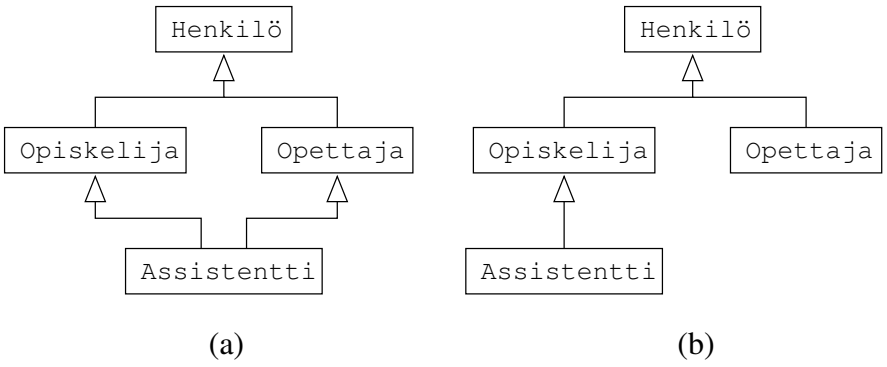
`Määräaikainen`, luokan `Työnkuva` alle `Tuotanto` ja muut työn tyyppiä kuvaavat luokat. Kun sitten muodostetaan määräaikaisessa työsuhteessa palvelevaa johtohenkilöä vastaava luokka, sen pitäisi periä kummastakin haarasta vastaavat luokat. Ideana on, että hierarkian eri haarat edustavat jotakin ominaisuutta ja perijät keräävät sitten tarpeellisen määrän ominaisuuksia eri haaroista. Hierarkian eri haarojen halutaan tavallisesti kuitenkin kuvaavan toisensa poissulkevia ominaisuuksia. Esitelty ratkaisu ei tue tätä periaatetta.

Kuva 9.1 esittää vaihtoehdon, jossa `Työntekijä` on `Palvelussuhde`-luokan asiakas. Tämän jäsennyksen ansiosta periytymishierarkia on selkeä. Esittelemällä abstrakti luokka `Palvelussuhde`, henkilö voi muuttaa työsuhdettaan dynaamisesti ilman, että ohjelmistoon joudutaan tekemään muutoksia. Suunnittelumallien (*design patterns*) terminologiassa tällaisella rakenteella on useita eri nimiä: *bridge*, *state* ja *strategy* riippuen siitä miten rakenteen on tarkoitus toimia ajoaikana.

9.3.2 Moniperiytymisen simulointi

Javan syntaksisääntöjen mukaan luokka voi periä ainoastaan yhden konkreetin tai abstraktin luokan ja sen lisäksi mielivaltaisen määrän rajapintoja. Tämä aiheuttaa ongelmia erityisesti sellaisissa tapauksissa, joissa olisi mukavaa periä useammalta ei-rajapinnalta. Tyypillinen esimerkki tällaisesta on kuvan 9.2 kohdan (a) mukainen ”timanttirakenne”: `Assistentti` kuvaa henkilöä, jolla on sekä opiskelijan että opettajan ominaisuuksia.

Koska moniperiytymistä ei ole käytössä, pitää hierarkiaa rakennettaessa tehdä päätös siitä, laitetaanko luokka `Assistentti` luokan `Opiskelija` vai luokan `Opettaja` alle. Tilanne on symmetrinen kummankin kannalta ja päätökseen vaikuttaa ensisijaisesti se, miten luokkien mukaisia objekteja käsitellään implementoitavassa ohjelmassa. Jos `Opiskelija`- ja `Assistentti`-luokkien objekteja halutaan käyttää ohjelmassa polymorfisesti, valinta tehdään kuvan 9.2 kohdan (b) mukaan. Päätös yliluokasta tehdään vain kerran ja sen on syytä pysyä voimassa myös jatkossa, koska näiden luokkien perijät on sidottu tiukasti tehtyyn päätökseen. Myöhemmin tehtävien ohjelmien kannalta päätös saattaa kuitenkin olla ”väärä” ja tehdä niiden koodista hankalamman näköistä kuin asia mallintamisen kannalta tosiasiassa on. Olipa `Assistentti` sitten kummassa haarassa tahansa on selvää, että siihen



Kuva 9.2: Esimerkki moni- ja yksittäisperiytymisestä.

joudutaan kirjoittamaan tarkalleen samaa koodia, jota löytyy jo olemassaolevasta luokasta (esimerkissä `Opettaja`-luokasta).

Analysoidaanpa tilannetta hieman tarkemmin. Perusajatus on se, että opiskelijat, opettajat ja assistentit ovat henkilöitä, joilla on erityisosaamista. Opiskelijalla se on opiskelijuuteen liittyvät ominaisuudet, opettajalla opettajuuteen liittyvät ominaisuudet ja assistentilla molemmat. Muodostamalla rajapinnat, jotka mallintavat näitä ominaisuuksia, tilanne näyttää kuvan 9.3 mukaiselta. Rajapinta `Opiskelijuus` kuvaa opiskelijan roolia ja itse `Opiskelija`-luokka on yksi tämän roolin toteuttajista. Nimen `Opiskelijuus` sijasta voitaisiin käyttää vaihtoehtoisesti myös `Opiskelijamaisuus` tai `Opiskeleva`, mikäli se auttaa ymmärtämään rajapinnan merkitystä paremmin.

Tehdyn rakennelman etuna on, että assistentteja voidaan käsitellä molempien rajapintojen kautta ja kaikkia voidaan käsitellä `Henkilö`-olioina. Haittapuolena on, että `Assistentti`-luokkaan joudutaan kopioimaan samat koodiosat, jotka löytyvät `Opiskelija` ja `Opettaja`-luokasta. Tämä on jyrkästi OO-periaatteita vastaan, joten ratkaisu ei tyydytä meitä. Tavoitteena on muodostaa systeemi, joka täyttää minimivaatimukset: (a) kaikkia olioita pitää pystyä käsittelemään henkilöinä ja (b) koodia ei tarvitse kopioida paikasta toiseen.

Kuvassa 9.4 esitelty ja molemmat ehdot täyttävä ratkaisu on yllättävän spagettimainen. Rajapinnat `Opiskelijuus` ja `Opettajuus` ovat aiemmin esitellyn mukaiset. Abstraktit luokat `AbstraktiOpiskelija` sekä `AbstraksiOpettaja` antavat rajapinnoille vaatimuksen (b) edellyttämät perustoteutukset (joita perivä luokka voi muuttaa, jos niin haluaa). Vaikka toteuttavat luokat on merkitty ja nimetty abstrakteiksi, ne ovat itse asiassa konkreetteja ja sisältävät kaikki tarvittavat piirteet. Abstraktileimaa tarvitaan sen takia, että luokat sisältävät ainoastaan tiettyyn rooliin liittyvät ominaisuudet, mutta eivät henkilötietoja. Näin ollen luokista ei ole mielekästä luoda itsenäisiä olioita, minkä **`abstract`**-määrite estääkin. Luokka `Opettaja` on merkitty `Henkilö`-luokan perijäksi polymorfisuusvaatimuksen (a)

takia. Jotta luokalla `Opettaja` olisi myös opettajaominaisuudet, siihen sijoitetaan `AbstraktiOpettaja`-luokalta perivä sisäluokka, joka antaa halutessaan korvaavia toteutuksia perimänsä luokan piirteille esim. `Henkilö`-tietoja käyttäen. `Opiskelija`-luokka muodostetaan samoja periaatteita noudattaen.

`Assistentti`-luokka, jossa sekä opiskelija- että opettajaominaisuuksien tulee yhdistyä, kertoo osaavansa nämä taidot, koska (i) se ilmoittaa toteuttavansa molemmat rajapinnat ja (ii) sillä on `AbstraktiOpettaja`- ja `AbstraktiOpiskelija`-tyyppiset sisäluokat, joille nämä toiminnot delegoidaan. Nyt `Assistentti`-tyypin objektia voidaan käsitellä polymorfisesti `Henkilö`-, `Opettajuus`- ja `Opiskelijuus`-liitännän kautta.

Tarkastelemamme luokat muodostavat vain hyvin pienen osan periytymishierarkiasta ja jos samaa periaatetta sovelletaan yleisesti, systeemin hallinta tulee erittäin vaikeaksi. Yhteenveto: moniperinnän simulointi on hankalaa, mutta jos sitä ei tehdä, luokkahierarkian rakentamisen yhteydessä tehdyt valinnat saattavat osoittautua myöhemmin ongelmallisiksi. Kun nokka irtoaa, pyrstö tarttuu.

9.3.3 Funktioargumenttien simulointi

Rutiinit eivät ole Javassa ”ensimmäisen luokan kansalaisia” siinä mielessä, että niitä voitaisiin käsitellä kuten muitakin olioita, mikä tulee esiin siinä, että rutiineja ei voi välittää argumentteina kutsutulle piirteelle. Kuitenkin kielen joustavuus kasvaa merkittävästi rutiiniargumenttien myötä. Funktioargumenttien kieltäminen ei silti ole niin tiukka rajoitus kuin äkkiseltään voisi luulla. Aiemmin on jo nähty esimerkkejä, joissa `Comparator`-tyyppinen argumentti on välitetty vaikkapa lajittelurutiinille kapseloimalla vertailufunktio omaan luokkaansa ja välittämällä tämän luokan (tai kokoavan yliluokan) tunniste piirteelle, joka sitä tarvitsee. Dynaaminen sidonta huolehtii tämän jälkeen siitä, että kutsu suorittaa oikean rutiinin. Toinen mahdollisuus on esitellä haluttu funktio abstraktiksi siinä luokassa, jossa sitä tarvitaan ja tehdä näin saadulle abstraktille luokalle perijöitä, joissa ko. rutiini on toteutettu.

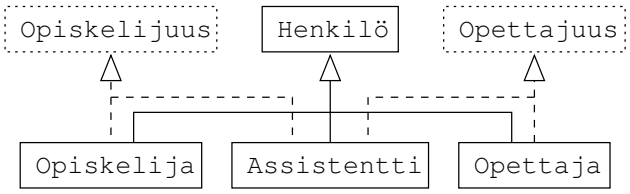
Simulointi periytymistä käyttäen

Tarkastellaan yksinkertaistettua esimerkkiä, jossa taulukon parametrisoitua tyyppiä `E` olevat alkiot halutaan yhdistää operaatiolla, jonka signatuuri on

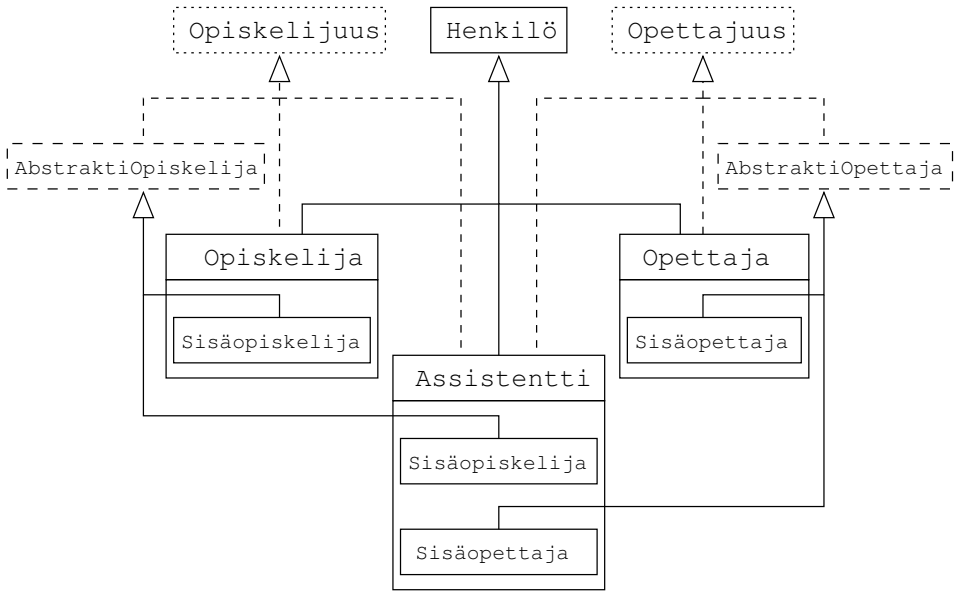
```
public E binäärioperaatio(E a1, E a2)
```

Funktiolla on kaksi samantyyppistä argumenttia, joita käyttäen se palauttaa edelleen samaa tyyppiä olevan tuloksen. Tällaisia funktioita ovat esimerkiksi yhteenlasku, kertolasku, maksimi jne. Yhdistely suoritetaan abstraktissa luokassa:

```
public abstract class Yhdistäjä<E>
```



Kuva 9.3: Esimerkki moniperiytymisestä rajapintojen avulla.



Kuva 9.4: Esimerkki moniperiytymisestä rajapintojen ja sisäluokkien avulla.

```
{
/**
 *  @.pre  alkiot != null && alkiot.length >= 2
 */
public E yhdistä(E[] alkiot)
  {
  E tulos = alkiot[0];
  for (int i = 1; i < alkiot.length; i++)
    tulos = binäärioperaatio(tulos, alkiot[i]);
  return tulos;
  }

public abstract E binäärioperaatio(E a1, E a2);
}
```

Alkuehto varmistaa että rutiini palauttaa kaikissa tilanteissa järkevän tuloksen. Perusfunktio `binäärioperaatio` on abstrakti. Tarkoituksena on kiinnittää sen toteutus perijäluokassa, jolloin yliluokan ohjelmarungosta saadaan aina eri tavalla käyttäytyvä riippuen siitä, miten sen kiinnittämättä oleva osa sidotaan suoritusaikana.

Jos taulukosta halutaan etsiä pienin alkio, kirjoitetaan konkreetti perijäluokka:

```
public class Minimoija<E extends Comparable<? super E>>
                                                extends Yhdistäjä<E>
{
public E binäärioperaatio(E a1, E a2)
  {
  return (a1.compareTo(a2) < 0 ? a1 : a2);
  }
}
```

Koska esimerkissämme on kyse yksiulotteiselle taulukolle tehtävistä operaatioista, luokka `Yhdistäjä` olisi luonnollista laittaa taulukkoluokan perijäksi. Java ei kuitenkaan salli sitä, joten ainoa mahdollisuus on tehdä siitä oma luokkansa, joka sisältää apuvälineitä taulukoiden käsittelyyn.[1]

Jätetään Java-spesifiset ongelmat taustalle ja mietitään tehdyn ratkaisun toimuvuutta yleisemmin. On helppo havaita, että yksi perijä osaa tehdä vain yhden tempun. Tämä on kiusallista ja se juontaa juurensa samasta ongelmasta, jota käsiteltiin jo poissulkevan luokittelun yhteydessä (ks. kohta 9.3.1): kiinnittämällä avoi-

[1] Hakasulkunotaatiota käyttävä taulukko on erikoisluokka Javassa, koska sillä ei ole edes omaa nimeä. Joissakin tilanteissa (esim. kyseltäessä class-tietoja) tällaisen pseudoluokan kuvitellaan kuitenkin olevan olemassa. Esimerkiksi **int**-alkioita sisältävää taulukkoa vastaava luokkanimi on `[I`.

met piirteet periytymismekanismin avulla niitä ei voi ajoaikana enää muuttaa. Jos siis `Työntekijä`-luokan perijäksi laitetaan palvelussuhdetta kuvaava `Määräaikainen` ja sen perijäksi edelleen työnkuvaan liittyvä `Tuotanto`, tämän luokan mukainen henkilö on aina määräaikainen tuotantoon osallistuva työntekijä. Jos jompaankumpaan ominaisuuteen tulee muutos, kyseistä henkilöä vastaava olio joudutaan rakentamaan kokonaan uudestaan. Tämän takia muuttuvat ominaisuudet kannattaa valita luokan jäsenmuuttujiksi, jolloin niitä voidaan muuttaa ajoaikana "lennossa". Samaan ajatukseen perustuu seuraavakin ratkaisu, joskin sen ulkoinen muoto poikkeaa aiemmasta siinä, että nyt on kyse rutiinin argumenteista.

Simulointi asiakasrelaatiota käyttäen

Edellä kohtaamamme hankaluudet juontavat juurensa siis itse asiassa siitä, että `binäärioperaatio` on sijoitettu samaan luokkaan muiden operaatioiden kanssa. Erotetaan ne toisistaan ja muodostetaan ensin yksinkertainen rajapinta:

```
public interface Binäärifunktio<E>
  {
  public E binäärioperaatio(E a1, E a2);
  }
```

Tekemällä tälle rajapinnalle jälkeläisiksi `Minimoija` ja muut binäärifunktiot, saadaan muodostettua hierarkiaosa, joka on riippumaton `yhdistä`-piirteestä. Tämän seurauksena `Binäärifunktio` ja sen perijät ovat muidenkin luokkien käytettävissä. Piirre `yhdistä` voidaan sijoittaa luokkaan, jossa on muitakin taulukoiden käsittelyyn tarvittavia rutiineja:

```
/**
 *  @.pre  (alkiot != null && alkiot.length >= 2) & f != null
 */
public static <E> E yhdistä(E[] alkiot, Binäärifunktio<E> f)
  {
  E tulos = alkiot[0];
  for (int i = 1; i < alkiot.length; i++)
    tulos = f.binäärioperaatio(tulos, alkiot[i]);
  return tulos;
  }
```

Tehtäviä

9-1 Tarkastele seuraavia `Ostos`- ja `Ostoslista`-luokkia.

```
public class Ostos {
```

```
   /** Ostoksen nimi. */
   private String nimi;
   /** Ostoksen hinta. */
   private double hinta;

   /** Alustaa ostokselle nimen ja hinnan.
    *  @.pre  nimi != null & hinta > 0
    *  @.post annaNimi() == nimi & annaHinta() == hinta
    */
   public Ostos(String nimi, double hinta) {
      this.nimi = nimi;
      this.hinta = hinta;
      }

   /** Palauttaa nimen.
    *  @.pre  true
    *  @.post RESULT == (ostoksen nimi)
    */
   public String annaNimi() { return nimi; }

   /** Palauttaa hinnan.
    *  @.pre  true
    *  @.post RESULT == (ostoksen hinta)
    */
   public double annaHinta() { return hinta; }
   } //Ostos

public class Ostoslista extends java.util.ArrayList<Ostos> {
   /** Palauttaa ostosten yhteenlasketun hinnan.
    *  @.pre  true
    *  @.post RESULT == (listan ostosten yhteenlaskettu hinta)
    */
   public double annaHinta() {
      double tulos = 0.0;
      for ( Ostos ostos : this ) {
         tulos += ostos.annaHinta();
         }
      return tulos;
      }
   } //Ostoslista
```

(a) Mitä keinoja `Ostoslista`-luokka tarjoaa asiakkailleen alkioiden poistamiseen? *Vinkki:* huomaa myös `ArrayList`-luokan metodit `iterator()` ja `listIterator()`.

(b) Miten `Ostoslista`-luokkaa pitäisi muuttaa, jos halutaan, ettei siitä voida poistaa alkioita?

(c) Kirjoita `Ostoslista`-luokka uudelleen siten että se käyttää `ArrayList<Ostos>`-luokkaa asiakkaana. Toteuta luokalle seuraavat ominaisuudet: alkioiden lisäys, listan iterointi, listan ostosten yhteenlasketun hinnan palauttaminen sekä listan sisältämien alkioiden määrän palauttaminen. Tee iteraattorista sellainen, että sen avulla ei voida poistaa listan alkioita.

9-2 Kuvassa 9.4 on esitetty ratkaisu moniperiytymisen simuloimiseksi rajanpintojen ja sisäluokkien avulla. Laadi toteutus kuvan luokille. Ratkaisu ei täysin poista saman koodin kopiointia luokasta toiseen. Minne tämä turha koodin kopiointi siirtyy?

9-3 Laadi abstraktille luokalle `Yhdistäjä` perillinen `Summaaja`, joka operoi `Double`-tyyppisillä alkioilla.

Oikeastaanhan luokan olisi hyvä käyttää parametrisoitua tyyppiä `E extends Number`. Mitä toteutuksellisia ongelmia sen käytöstä seuraa?

9-4 Laadi esitetylle `Binäärifunktio<E>`-rajapinnalle perillinen

```
public class Minimi<E extends Comparable<? super E>>
                                        implements Binäärifunktio<E>
```

Anna esimerkki sen käytöstä, kun kokonaislukutaulukosta `new int[] { 5, 3, 8, 2, 5, 4 }` haetaan minimialkio.

Luku 10

Epilogi

Objektiorientoitunut ohjelmointikieli tuo mukanaan runsaasti uusia käsitteitä, välineitä ja menettelytapoja, joita ei ole perinteisissä imperatiivisissa kielissä, kuten esimerkiksi C tai Pascal. Vaikka OO-kielen ydin eli itse käskyvalikoima pysyy edelleen kompaktina, lisäävät periytyminen, polymorfismi ja dynaaminen sidonta kielen kompleksisuutta ja ajoaikaisen käyttäytymisen intuitiivista mallintamista. Yhdistelemällä näitä ohjelmassa järkevästi päädytään kuitenkin kauniisiin ratkaisuihin, jotka varmasti saavat itse kunkin vakuuttuneeksi siitä, että kyseiset mekanismit ovat olennaisia hyvän ohjelmoijan työkaluja. Tämä on kuitenkin kaksiteräinen miekka, sillä huonosti käytettynä samat mekanismit tuottavat hyvin erikoisesti käyttäytyviä ohjelmia. Jos kieli ei tue ja ohjaa riittävästi oikeaan suuntaan esimerkiksi rutiinin määrittelyjen ja luokkainvarianttien muodossa, ohjelmoijalta vaaditaan suurta kurinalaisuutta noudattaa periaatteita, jotka ovat aivan keskeisiä esimerkiksi dynaamisen sidonnan onnistumiselle.

Myös ohjelmointiympäristö laajenee, se ei sisällä ainoastaan editoria ja kääntäjää, vaan paljon työkaluja myös ohjelmiston korkean tason kuvaukseen ja rakentamiseen. Mukaan ovat tulleet myös graafisten käyttöliittymien tekoon tarvittavat osat, tietokantaliittymiin ja hajautettuihin järjestelmiin tarvittavia välineitä jne. Nämä on jätetty tämän kurssin ulkopuolella ajan puutteen takia; nyt on haluttu keskittyä vain olennaisiin olioajattelun kulmakiviin.

OO-ohjelmointifilosofia tuottaa uudelleenkäytettäviä ”peruspalikoita”, jotka on testattu, todettu tehokkaiksi ja asetettu yleiseen käyttöön kirjastoon. OO-kielten mukana tulevat valmiit kirjastot ovatkin tyypillisesti hyvin laajoja. Jotta niistä löytäisi etsimänsä, kirjaston pitäisi olla hyvin strukturoitu ja sen päälle tulisi rakentaa apuväline (hakuohjelma), jonka avulla ohjelmoija saa käsiinsä relevantit luokat esimerkiksi muutaman hakusanan perusteella. Myös sovellusohjelmien tekijältä vaaditaan huolellisuutta ja perusteellista harkintaa silloin, kun hän päättää

asettaa oman ohjelmistokomponenttinsa juuri tiettyyn paikkaan kirjastoluokkien periytymishierarkiassa. Hyviä kirjastoja ei tehdä yhdessä yössä, vaan se vaatii pitkäaikaisen kokemuksen ja kypsyttelyn.

Aika näyttää mihin suuntaan OO-kielten käsitteet kehittyvät ja voidaanko niitä yksinkertaistaa, mutta jo nyt on selvästi nähtävissä, että luokkapohjainen ohjelmasuunnittelu on vienyt koko alaa aimo askeleen parempaan suuntaan. Rutiinien määrittelyjen avulla sekä ohjelman koodaaminen että suunnittelu voidaan perustaa samaan formalismiin. Koska suunnittelun ja toteutuksen välinen kuilu saadaan näin häivytettyä, vaiheista tulee *saumattomasti* (*seamless*) yhteenliittyviä. Kuilun umpeenkurominen johtaa ohjelmiston toteutus- ja suunnitteluvaiheiden välisten yhteensovittamisongelmien radikaaliin vähenemiseen.

Jos ohjelmiston suunnittelu OO-kielellä tuntuu hankalalta, on syytä huomata, että kompleksisuus ei useinkaan johdu itse kielestä; reaalimaailman abstrakti mallintaminen nyt vain on vaikeaa. Yhtä kaikki, asioiden jäsentely ja luokittelu tällä abstraktiotasolla on vähintään yhtä kiinnostavaa kuin yksittäisen rutiinin kirjoittaminen!

Tehtäviä

10-1 Etsi jokin aikaisemmin tekemäsi harjoitustyö (tai jokin muu laatimasi ohjelmakokonaisuus) ja tarkastele sen rakennetta ja toteutusta.

10-2 Mitä muuttaisit (jos mitään) aikaisemmin tekemästäsi harjoitustyöstä tästä kirjasta oppimasi perusteella?

Liitteet

Liite A

Javadoc-aputiedostot

Tässä liittessä on esitelty Javadoc-työkalun käyttöä helpottavat tiedostot (ks. kohta 1.1.2, s. 5). Tiedosto `common.jd` sisältää yleisiä täkymäärityksiä ja tiedostoa `project.jd` voi muokata ja uudelleennimetä käsillä olevan projektiin sopivaksi lisäämällä sinne ne luokat josta haluaa luoda Javadoc-dokumentin.

A.1 `common.jd`

```
-tag .jd.common.internal.comment:X:Argumenttitiedosto_javadocille
-tag .jd.common.internal.version:X:2007-02-19

-author
-keywords
-linksource
-private
-source 1.6
-version

-tag .toDo:a:To do:
-tag deprecated
-tag .classInvariant:t:Class invariant:
-tag .classInvariantProtected:t:Protected class invariant:
-tag .classInvariantPrivate:t:Private class invariant:
-tag .abstractionFunction:t:Abstraction function:
-tag .pre:cm:Precondition:
-tag .post:cm:Postcondition:
```

```
-tag .postProtected:cm:Protected postcondition:
-tag .postPrivate:cm:Private postcondition:
-tag .time:cmf:Time complexity:
-tag .space:cmf:Space complexity:
-tag throws
-tag param
-tag return
-tag see
-tag author
-tag .correspondence:a:Correspondence:
-tag version
-tag since
-tag serial
-tag .download:a:Download:
```

A.2 `project.jd`

```
-tag .jd.project.internal.comment:X:Argumenttitiedosto_javadocille
-tag .jd.project.internal.version:X:2007-02-19

-d docs

-windowtitle 'Projektin otsikko'
-doctitle 'Dokumentin otsikko'
-overview mahdollinen-projektin-yleiskuvaus.html
-bottom '&copy; 200Y Etunimi Sukunimi. All Rights Reserved.
<p align="right">Conforms to Java<sup><small>TM</small></sup>
2 Platform Standard Edition 6.0'

EsimerkkiTiedosto1.java
EsimerkkiTiedosto2.java
```

Liite B

Silmukan oikeellisuus

Silmukka erilaisine ilmenemismuotoineen on ohjelmoinnin perusrakenne, jota ilman tietokoneita tuskin edes käytettäisiin. Jotta voisit varmistua omien silmukoidesi oikeasta toiminnasta, on syytä tutustua käsitteeseen *silmukkainvariantti* (*loop invariant*). Se on väittämä, jonka tulee olla voimassa (while-)silmukan alustuksen jälkeen ja jokaisen kierroksen lopussa. Invariantissa kerrotaan eksplisiittisesti se idea, joka toteuttajalla on ollut mielessään kirjoittaessaan silmukkarungon. Kysymys ei siis ole minkään uuden keksimisestä, ainoastaan oman ajatuksen esittämisestä kirjallisesti. Tästäkin huolimatta silmukkainvariantin kirjoittamista pidetään suhteellisen vaikeana. Syynä tähän lienee useimmiten se, että ohjelmoija ei ole tottunut kirjoittamaan väittämiä. Invariantti on syytä kirjoittaa aina mahdollisimman vahvaksi siinä mielessä, että se edustaa tarkalleen vain sitä silmukkaa, johon se on kirjoitettu. Esimerkiksi invariantti `0 <= i < t.length` on voimassa kaikissa taulukon `t` läpikäyvissä silmukoissa. Se ei kuitenkaan kerro mitään siitä, mitä silmukassa todella tapahtuu.

Keskitymme tässä while-silmukoiden invariantteihin, mutta yhtälailla niitä voidaan määritellä hiukan harvemmin käytettäville until- ja for-silmukoillekin. Tämän valinnan syynä on lähinnä se että while-silmukkaan liittyvä invariantti on selkeimmin kuvailtavissa ja että silmukkarakenteet voidaan tarvittaessa muuntaa helposti toisikseen.

Kokonaislukujako Yksinkertainen esimerkki silmukkainvariantista on seuraava osamäärän laskeva rutiini, jossa invariantti on kirjoitettu kommentiksi silmukan eteen.

```
/**
 * Palauttaa jakolaskun kokonaisosan.
```

```
 *  @.pre  jaettava >= 0 & jakaja > 0
 *  @.post jakaja * RESULT <= jaettava < jakaja * (RESULT + 1)
 */
public static int kokonaisjako(int jaettava, int jakaja)
  {
  int jäännös = jaettava;
  int tulos = 0;
  /*
   * Invariantti: jaettava == (jakaja * tulos + jäännös) &
   *              0 <= jäännös
   */
  while (jäännös >= jakaja)
    {
    jäännös -= jakaja;
    ++tulos;
    /* Invariantti on aina voimassa tässäkin kohdassa. */
    }
  return tulos;
  }
```

Invariantin avulla voidaan todeta, että silmukan jokainen suorituskerta toimii oikein käyttämällä samaa päättelyä kuin matemaattisessa induktiossa. Silmukan alustus tarvitaan, jotta invariantti tulisi voimaan ennen silmukan ensimmäistä suorituskertaa. Toisaalta silmukan jälkeinen tilanne saadaan yhdistämällä silmukkainvariantti ja negaatio (while-)silmukan lopetusehdosta, koska silloin molempien pitää olla voimassa. Koska esimerkin rutiinissa ei ole silmukan jälkeen enää muita suorittavia lauseita kuin **return**, on rutiinin lopussa voimassa

```
jaettava == (jakaja * tulos + jäännös) & 0 <= jäännös &
jäännös < jakaja
```

mikä on kokonaislukujakolaskun tuloksen matemaattinen määritelmä.

Siitäkin huolimatta, että silmukan runko toteuttaa invariantin joka kierroksella, toteutus voi olla väärin tehty. Silmukan suorituksen pitää nimittäin myös päättyä joskus. Tämä taataan *variantin* (*variant*) avulla. Variantti on kokonaislukuarvoinen lauseke, jonka arvo pienenee jokaisella kierroksella. Kun lisäksi vaaditaan, että variantilla on jokin kiinteä alaraja, on osoitettu, että suorituskertoja on vain äärellinen määrä. Esimerkkitapauksessamme variantiksi voidaan valita yksinkertaisesti yhden muuttujan lauseke `jäännös` ja rajaksi nolla. Alarajaa ei siis tarvitse välttämättä saavuttaa, riittää, kun sen olemassaolo tiedetään.

Lisäyslajittelu Yksiulotteisen taulukon `t[]` indeksivälillä `a...y` (oletetaan että indeksivälit ovat suljettuja, ts. päätepisteet kuuluvat mukaan tarkasteltavaan alu-

Kuva B.1: Lisäyslajittelun ulkosilmukan invariantti.

eeseen) sijaitsevat alkiot voidaan lajitella ei-laskevaan suuruusjärjestykseen *lisäyslajittelumenetelmällä* (*insertion sort*), joka toimii seuraavasti: Kun algoritmi aloittaa käsittelemään kohdassa `i` olevaa alkiota, se on jo järjestänyt kyseisen alkion vasemmalla puolella olevat, paikoissa `a...i - 1` sijaitsevat alkiot suuruusjärjestykseen. Tehtävänä on nyt etsiä alkiolle `t[i]` paikka `k`, ($\texttt{a} \leq \texttt{k} \leq \texttt{i}$) jo järjestetyssä jonossa ja kun se on löydetty, siirretään paikkojen `k...i - 1` alkiot yhden position verran oikealle. Lopuksi paikassa `i` alunperin sijainnut alkio asetetaan kohtaan `t[k]`.

Tässä menetelmäkuvauksessa oletetaan että tyhjä indeksiväli esitetään siten että välin alaraja on yhtä suurempi kuin yläraja; ts. tyhjä indeksiväli on muotoa `x + 1...x`. Tällaisen tulkinnan ansiosta eo. lajittelun kuvaus toimii myös tapauksille missä syöte on tyhjä tai sisältää vain yhden alkion.

Lajittelun toteutukseen tarvitaan silmukkaa, jonka sisällä haku ja alkioiden siirto suoritetaan. Tavallisesti nämä kaikki toiminnot sijoitetaan samaan rutiiniin, mutta seuraavassa haku ja siirto on toteutettu havainnollisuuden vuoksi erillisinä. Kuva B.1 kertoo, missä tilassa taulukko on algoritmin ulomman (while-)silmukan jokaisen kierroksen lopussa (ulomman silmukan invariantti). Antamalla järjestystä kuvaavalle ehdolle

```
FORALL(k : alin <= k < ylin; t[k] <= t[k + 1])
```

nimi `järjestetty(t, alin, ylin)` ja kahden taulukon permutaatioehdolle[1] nimi `permutaatio(t0, t1, alin, ylin)` (ts. kummastakin taulukosta löytyy samalla indeksivälillä `alin...ylin` olevat alkiot permutoituna), voidaan kuvan B.1 mukainen tilanne ilmaista väittämäparina

```
järjestetty(t, a, i - 1) &
permutaatio(t, OLD(t), a, i - 1)
```

Tätä lajitteluproseduurin silmukkainvarianttia voitaisiin vielä täydentää osalla, joka ilmaisee, että indeksiväleillä `0...a - 1` ja `i...t.length - 1` sijaitseviin tietoihin ei ole tehty muutoksia. Kuten loppuehtojen kohdalla, myös silmukkainvarianttien tapauksessa oletuksena on, että vain muuttuvista tiedoista raportoidaan, paitsi silloin, kun muuttumattomuutta halutaan erityisesti korostaa. Nyt lisäyslajittelu voidaan kirjoittaa proseduraaliseen muotoon

[1]Permutointi tarkoittaa sitä, että alkioiden paikat ovat saattaneet muuttua, mutta ainuttakaan alkiota ei ole hukattu (ja näin muodoin myöskään uusia alkioita ei ole voitu tuoda joukkoon). Esimerkiksi multijoukot $\{1, 2, 1, 1, 3\}$ ja $\{3, 2, 1, 1, 1\}$ ovat toistensa permutaatioita.

```
/**
 *  Lajittelee taulukon t[a...y] alkiot ei-laskevaan
 *  suuruusjärjestykseen.
 *  @.pre  t != null && (0 <= a <= (y + 1) & y < t.length)
 *  @.post järjestetty(t, a, y) &
 *         permutaatio(t, OLD(t), a, y)
 */
public static void lisäyslajittele(int[] t, int a, int y)
  {
  int i = a + Math.min(1, y - a + 1);
  /*
   * Invariantti: järjestetty(t, a, i - 1) &
   *              permutaatio(t, OLD(t), a, i - 1)
   * Variantti:   y - i
   */
  while ( i <= y )
    {
    int haettava = t[i];
    int paikka = annaPaikka(t, a, i - 1, haettava);
    siirräEteenpäin(t, paikka, i - 1);
    t[paikka] = haettava;
    ++i;
    /* Invariantti on aina voimassa tässäkin kohdassa. */
    }
  /*
   * Tässä kohdassa on voimassa invariantti ja
   * lauseke (i + 1) == y eli lajittelun loppuehto.
   */
  }
```

Lajitteluproseduurin otsakkeen selitetekstin lyhennysmerkintä `t[a...y]` tarkoittaa taulukon `t` osaa, joka muodostuu alkioista `t[a]`, ..., `t[y]`. Huomaa myös että silmukan alustuksessa käytetään `min`-funktiota yhdistämään tapaukset ”lajittele tyhjä taulukko” ja ”lajittele yhden alkion taulukko” koska ne ovat jo valmiiksi järjestyksessä. Silmukan runkoa suoritetaan vain kun `t` sisältää vähintäin kaksi alkiota.

Funktiossa `annaPaikka` haetaan alkiolle `t[i]` paikkaa taulukon `t` lajitellusta osasta `a...i - 1` kulkien oikealta vasemmalle. Jos etsinnässä on edetty kohtaan `p`, voidaan hakusilmukan invariantti esittää muodossa

```
FORALL(k : p <= k <= i - 1; t[i] < t[k])
```

Huomaa, että väittämä on aina voimassa silloin, kun annettu indeksialue on tyh-

jä. Toisin sanoen hakusilmukka voidaan alustaa asettamalla `p = i`. Hakufunktion toteutus on

```
/**
 *  Palauttaa taulukon t[a...y] suurimman kohdan p siten että
 *  taulukon t[p..y] alkiot ovat aidosti suurempia kuin alkio
 *  haettava.
 *  @.pre  t != null && (0 <= a <= (y + 1) & y < t.length)
 *  @.post (a <= RESULT <= y + 1) &
 *         (RESULT == a || t[RESULT - 1] <= haettava) &
 *         FORALL(k : RESULT <= k <= y; haettava < t[k])
 */
public static int annaPaikka(int[] t, int a, int y, int haettava)
  {
  int p = y + 1;
  /*
   * Invariantti: (a <= p <= y + 1) &
   *              FORALL(k : p <= k <= y; haettava < t[k])
   * Variantti:   p - a
   */
  while ( (a <= p - 1) && (haettava < t[p - 1]) )
    {
    --p;
    /* Invariantti on aina voimassa tässäkin kohdassa. */
    }
  /*
   * Tässä kohdassa on voimassa invariantti ja lauseke
   * (a == p || t[p - 1] <= haettava) eli hakemisen loppuehto.
   */
  return p;
  }
```

Huomaa että jos haettava alkio on suurin kuin mikään jo lajitellun osan alkioista, `annaPaikka` palauttaa `y + 1` eli lajittelusilmukan kohdan `i`. Näin pitääkin menetellä sillä tällöin `t[i]` suurimpana alkiona on jo valmiiksi omalla paikallaan lajitellun osan viimeisenä. Jos haettava alkio on pienin lajitelluista, sen kohdaksi annetaan `a`.

Proseduuri `siirräEteenpäin` siirtää annetun taulukko-osan alkiot position verran oikealle. Alkioiden siirto kannattaa tehdä oikealta vasemmalle, joten edettyämme silmukkarungon lopussa kohtaan `p` tiedetään, että

```
FORALL(k : p + 1 <= k <= y + 1; t[k] == OLD(t)[k - 1])
```

Alkioiden siirtoproseduurin tarkempi määrittely ja toteutus ovat

```
/**
 *  Siirtää taulukon t[a...y] alkiot yhden position eteenpäin.
 *  @.pre  t != null && (0 <= a <= (y + 1) & y < t.length - 1)
 *  @.post FORALL(k : a + 1 <= k <= y + 1; t[k] == OLD(t)[k - 1])
 */
public static void siirräEteenpäin(int[] t, int a, int y)
  {
  int p = y + 1;
  /*
   * Invariantti: FORALL(k : p + 1 <= k <= y + 1;
   *                         t[k] == OLD(t)[k - 1])
   * Variantti:   p - a
   */
  while ( a <= p - 1 )
    {
    t[p] = t[p - 1];
    --p;
    /* Invariantti on aina voimassa tässäkin kohdassa. */
    }
  /*
   * Tässä kohdassa on voimassa invariantti ja lauseke
   * a == p eli siirtämisen loppuehto.
   */
  }
```

Alkuehto ilmaisee, että siirrettävä alue ei saa ulottua taulukon viimeiseen alkioon asti, koska sen kohtalo jäisi siirrossa epäselväksi (mihin paikassa `t.length - 1` sijaitseva kokonaisluku siirretään?). Kuten muissakin aliohjelmassa, siirron indeksialue voidaan määritellä tyhjäksi. Tätä ominaisuutta tarvitaan silloin, kun rutiinissa `lisäyslajittele` kohdataan alkio `t[i]` joka on jo omalla paikallaan, ts. se on suurin jo lajitelluista alkioista eikä mitään siirrettävää lohkoa ole olemassa. Tällaisessa tapauksessa funktio `annaPaikka` palauttaa `i`, joten proseduuria `siirräEteenpäin` kutsutaan indeksivälille `i`...`i - 1`.

Olemme oikeastaan *todistaneet* silmukkainvarianttien avulla vaiheittain, että kummankin aliohjelman `annaPaikka` ja `siirräEteenpäin` toteutus täyttää sille annetun loppuehdon (edellyttäen että vastaava alkuehto on voimassa). Koska `lisäyslajittele` ei koskaan riko näiden aliohjelmien alkuehtoja, olemme myös todistaneet silmukkainvariantin avulla että `lisäyslajittele` täyttää oman loppuehtonsa annetuilla alkuehdoilla. Toisin sanoen voimme olla vakuuttuneita ilman minkäänlaista testaamista että `lisäyslajittele` toimii määrittelynsä mukaisesti (mikäli tehdyssä todistuksessa ei olla tehty virheitä...). Tämä havainto osoittaa että varsinkin pienten aliohjelmien kohdalla oikeaksitodistaminen on joskus var-

teenotettava vaihtoehto testaamiselle. Yhtenä lähestymistavan valintakysymyksenä voidaan käyttää sitä kuinka monia toisistaan riippumattomia erikoistapauksia ongelmalla on; jokaiselle tällaiselle tapausyhdelmällehän pitäisi joko käsin määritellä tai ohjelmallisesti generoida omat testinsä. Esimerkiksi lajittelulla on ainakin seuraavat kolme erikoistapauskategoriaa: (i) taulukon alkioiden määrä (ei lainkaan, yksi, kaksi, useita), (ii) alkioiden frekvenssi (ei samoja, kaksi samaa, joitain samoja, kaikki samoja) sekä (iii) alkioiden järjestys (valmiiksi lajiteltu, käänteisesti lajiteltu, satunnainen). Nyt valintakysymys on siis että kumpi vie kauemmin: näiden kategorioiden kombinaationa syntyvän noin kahdenkymmenen erilaisen testitapauksen toteuttaminen ja ajaminen vai ohjelman (riittävän) formaali oikeaksitodistaminen. Tähänhän puolestaan vaikuttaa suoraan se että kuinka tuttu kyseinen lähestymistapa on.

Lisäyslajittelun toteuttaminen kahden yksinkertaisen apurutiinin (`annaPaikka` ja `siirräEteenpäin`) avulla saattaa tuntua turhan monimutkaiselta. Olio-ohjelmoinnilla kuitenkin pyritään muuhunkin kuin aliohjelmien toteutusten yksinkertaisuuteen: kyse on tarpeellisten käyttäytymisrajapintojen määrittelystä järjestelmään (sekä luokka- että aliohjelmatasolla). Jos alkioiden järjestykseen liittyvä tarkastelu kuuluu olellisesti järjestelmän toimintaan, saattaa olla ymmärrettävyyden, ehtotarkistusten implementoimisen ja/tai uudelleenkäytettävyyden kannalta eduksi että aliohjelman `lisäyslajittele` toteuttamisen yhteydessä toteutetaan myös siihen läheisesti liittyviä aliohjelmia. Näitä ovat mm. taulukon aliosaan kohdistuvat predikaatit (mm. `järjestetty`, `permutaatio`, `kaikkiAidostiSuurempia`), funktiot (mm. `annaPaikka`) sekä proseduurit (mm. `siirräEteenpäin`). Jos alkioiden järjestystarkastelut eivät ole oleellisia järjestelmän toteuttamisessa, edellä esitetyt aliohjelmat voidaan aina laventaa pois vaivattomasti. Esimerkkimme lisäyslajittelun implementointi voidaan saattaa seuraavaan alirutiinittomaan muotoon.

```
/**
 *  Lajittelee taulukon t[a...y] alkiot ei-laskevaan
 *  suuruusjärjestykseen.
 *  @.pre  t != null && (0 <= a <= (y + 1) & y < t.length)
 *  @.post järjestetty(t, a, y) &
 *         permutaatio(t, OLD(t), a, y)
 */
public static void lisäyslajittele(int[] t, int a, int y)
  {
  int i = a + Math.min(1, y - a + 1);
  /*
   * Invariantti (1): järjestetty(t, a, i - 1) &
   *                  permutaatio(t, OLD(t), a, i - 1)
   * Variantti: y - i
   */
```

```
while ( i <= y )
  {
  int haettava = t[i];
  int paikka = i;
  /*
   * Invariantti (2):
   *   (a <= paikka <= i) &
   *   FORALL(k : paikka <= k <= i - 1; haettava < t[k]) &
   *   FORALL(k : paikka + 1 <= k <= i; t[k] == OLD(t)[k - 1])
   * Variantti: paikka - a
   */
  while ( (a <= paikka - 1) && (haettava < t[paikka - 1]) )
    {
    t[paikka] = t[paikka - 1];
    --paikka;
    /* Invariantti (2) on aina voimassa tässäkin kohdassa. */
    }
  /*
   * Tässä kohdassa on voimassa:
   *   (a <= paikka <= i) &
   *   (paikka == a || t[paikka - 1] <= haettava) &
   *   FORALL(k : paikka <= k <= i - 1; haettava < t[k]) &
   *   FORALL(k : paikka + 1 <= k <= i; t[k] == OLD(t)[k - 1])
   */
  t[paikka] = haettava;
  ++i;
  /* Invariantti (1) on aina voimassa tässäkin kohdassa. */
  }
/*
 * Tässä kohdassa on voimassa:
 *   järjestetty(t, a, y) & permutaatio(t, OLD(t), a, y)
 */
}
```

Aliohjelmaa `lisäyslajittele` voidaan myös muokata siten että sen uloin while-silmukka korvataan for-silmukalla sillä se käy järjestyksessä läpi kaikki taulukon `t` alkiot kohdissa `a + 1`...`y`.

Liite C

Abstrakti tietotyyppi

Ohjelmointikielen käsitteet *tyyppi* (*type*) ja siitä johdettu *tyyppiyhteensopivuus* (*type conformance*) ovat keskeisiä imperatiivisen ohjelmoinnin perustoiminnon, asetuslauseen (ja sitä käyttävän parametrivälityksen) kannalta. Java-ohjelmoijahan tietää, että asetus

```
double arvo; arvo = 3;
```

on oikein muodostettu. Vaikka asetuslauseen vasemmalla puolella olevan tunnisteen tyyppi on **double** ja oikean puolen literaalin tyyppi on **int**, jälkimmäinen on tyyppiyhteensopiva edellisen kanssa. Toisaalta tiedetään myös, että asetuslause

```
Henkilö pentti; pentti = new Työntekijä();
```

on oikein muodostettu, jos luokka `Työntekijä` on luokan `Henkilö` perillinen. Työntekijä on siis yhteensopiva henkilön kanssa. Miten tämä on mahdollista, nythän ovat kyseessä *luokat* (ajoaikana oliot), eivätkä *tyypit*? Vastaus: *Java samaistaa käsitteet tyyppi ja luokka.* Jokainen luokka määrittelee siis kieleen tyypin. Koska käsite "luokka" on jo tuttu, tarkastellaan lähemmin mitä käsite "tyyppi" tarkoittaa.

Tietotyyppi määritellään (epäsuorasti) tarkoittavan niiden tietojen joukkoa, joilla on samanlainen attribuuttien rakenne ja käyttäytyminen. Esimerkiksi Javakielen **int**-tyyppi tarkoittaa kaikkien mahdollisten kokonaislukujen joukkon $\mathbb{Z}$ alijoukkoa $\{-2^{31}, \ldots, 2^{31}-1\}$. Tämän tyypin alkioiden käyttäytyminen määritellään normaalien aritmeettisten laskutoimitusten avulla, joilla on vielä tuttu ilmiasukin: `+`, `-`, `*` ja `/`. Huomaa, että tyypin määrittely ei ota kantaa tiedon sisäiseen esitysmuotoon; eturmerkillisten kokonaislukujen tapauksessa se voisi olla yhtä hyvin esimerkiksi kahden komplementtiesitys kuin excess-n -esityskin. Tässä mielessä tyyppi on siis abstraktimpi käsite kuin luokka. Siinä missä yhtä luokkaa voi (ajoaikana) vastata useita siitä luotuja olioita, samaa tietotyyppiä vastaten voidaan muodostaa useita luokkia, jotka eroavat toisistaan sisäisen esitysmuodon, ja

siten julkiseen liitäntään kuuluvien operaatioiden suoritusnopeuden tai tilankäytön kannalta. Näille luokille on kuitenkin ominaista se, että niiden julkinen liitäntä on kaikilla tarkalleen sama, ja se määrittelee näistä luokista luotujen olioiden ominaisuudet (vrt. tietotyypin määrittely yllä). Tämän vuoksi myös (luokkien välinen) periytymishierarkia ja (tyyppien välinen) tyyppiyhteensopivuus kytkeytyvät kiinteästi toisiinsa.

Tarkasti ottaen voidaan siis sanoa, että tyyppi määrittelee sen mukaisten tietojen käyttäytymisen ja kukin luokka on tietyn tyypin eräs toteutus. Kun tätä abstraktisuuseroa halutaan painottaa, puhutaan *abstraktista tietotyypistä* (*abstract data type*, ADT). ADT-kuvauksessa ei haluta ottaa kantaa luokan konkreettiin esitystapaan tai ohjelmointikielen syntaksiin, ainoastaan tiedolle soveltuviin operaatioihin, erityisesti niiden välisiin relaatioihin. Tyyppiä vastaavassa luokassa nämä relaatiot (operaatioiden väliset riippuvuudet) näkyvät tavallisesti piirteiden loppuehdoissa ja luokkainvariantissa, joskin kirjoittamista vaikeuttaa yleensä se, että tietoa muuntavat rutiinit toteutetaan normaalisti sivuvaikutusten avulla eikä matemaattisina funktioina.

Abstraktin tietotyypin määrittely esimerkiksi pinotyypille voidaan esittää kuvan C.1 mukaisesti. ADT-määrittelyssä voidaan käyttää apuna toisia ADT:itä sekä niiden funktioita. Pinon esityksessä esimerkiksi oletetaan että meillä on jo abstrakti tietotyyppi nimeltä Boolean joka mallintaa totuusarvojen TRUE ja FALSE keskinäiset suhteet. Pinon ADT-määrittelyssä on käytetty operaatioista samoja nimiä kuin aiemmassa rajapintaluokassa (ks. listaus 2.1). Itse ADT-määrittely koostuu aina neljästä osasta [8]:

1. TYYPPI-osa kertoo määriteltävän tietotyypin nimen. Parametri X kertoo, että kyseessä on *geneerinen* tietotyyppi: tyyppiparametrin X tilalle voidaan laittaa mikä tahansa todellinen tyyppi. Esimerkiksi tämän ADT:n toteuttava luokka `Pino<Integer>` edustaa kokonaislukupinoa, `Pino<Tree>` puupinoa jne. (geneerisyydestä lisää luvussa 7). Tyyppimäärittely on riippumaton alkiotyypistä X.

2. FUNKTIOT-osaan kirjoitetaan ne operaatiot, joita voidaan kohdistaa tyypin tietoihin. Jokaisella rivillä on muotoa $A_1 \times A_2 \times \ldots \times A_m \rightarrow B_1 \times B_2 \times \ldots \times B_n$ oleva matemaattinen funktiomäärittely, missä ainakin yksi A_i tai B_j on määriteltävää ADT-tyyppiä; esimerkissämme siis muotoa Pino[X]. Jatkossa rajoitutaan, yleisyyden siitä mitenkään kärsimättä funktioihin, joilla on vain yksi tulosjoukko ($n = 1$). Nuoli, jonka yli on vedetty viiva, kuvaa osittaista funktiota. Merkinnät ovat puhtaasti matemaattisia eivätkä ota kantaa ohjelmalliseen toteutukseen. Esimerkiksi sivuvaikutusten avulla toteutetun `lisää`-operaation signatuuri saadaan funktiomerkinnästä poistamalla Pino[X] sekä määrittely- että arvojoukosta, koska sekä lähde- että kohdepino ovat tunnetut ja samat (eli **`this`**).

Funktiokuvauksen muodosta nähdään, että (a) määriteltävä tyyppi esiintyy luontifunktioissa vain nuolen oikealla puolella, (b) havainnoijissa vain vasemmalla puolella ja (c) muunnosoperaatioissa kummallakin puolella.

3. ALKUEHDOT-osa liittyy osittaisiin funktioihin. Se kertoo alkuehdot, joiden on oltava voimassa, jotta kyseistä funktiota saa käyttää.

4. AKSIOOMAT on ADT:n määrittelyn tärkein osa, sillä se kertoo tyypin mukaisten tietojen käyttäytymisen (semanttiset ominaisuudet). Osaan kerätyt lausekkeet ovat tosiarvoisia väittämiä. Implisiittisenä oletuksena on, että niissä esiintyvät funktiokutsut noudattavat ALKUEHDOT-osan ehtoja, ts. kutsut ovat oikein muodostettuja. On selvää, että FUNKTIOT-osan antaman informaation mukaisesti toimivia tyyppejä voi muodostaa monia, mutta vasta AKSIOOMAT-osa karakterisoi tyypin olioiden käyttäymistä. Pinotyypille tärkein ominaisuus on poista(lisää(x, s))=s, joka kertoo, että pino toimii LIFO-periaatteen (eli *last-in, first-out*) mukaisesti. AKSIOOMAT-osaan tulisi kerätä kaikki operaatioiden väliset riippuvuudet, mutta ei yhtään enempää. Esimerkiksi päällimmäinen(lisää(x, lisää(y, s)))=x on voimassa pinolle, mutta sitä ei ole syytä kirjata tähän osaan, koska se ei anna mitään uutta informaatiota.

   Yleisperiaatteena on että AKSIOOMAT-osan kuvauksessa voidaan käyttää (a) muuttujia, (b) `if-then-else`-rakennetta, (c) boolenlausekkeita, (d) rekursiota sekä (e) tarvittaessa tyyppimäärittelyn sisäisiä apufunktioita.

Pitääkö opetella taas uusi abstrakti formalismi, vaikka ohjelmointi on mitä konkreettisinta toimintaa, kysyy tarkkaavainen lukija. Kyseessä onkin abstraktin ajattelun apuväline, joka on hyödyllinen tuki silloin, kun reaalimaailman jäsentäminen oliokielen käsitteiden avulla käy ongelmalliseksi.[1] ADT:n suurin merkitys on siinä, että tietotyypin toteuttaja tulee miettineeksi huolellisesti eri operaatioiden välisiä yhteyksiä ja tekee toteutuksen vasta tämän analyysin pohjalta. AKSIOOMAT-osan väittämien toteutuminen tulisikin aina tarkistaa implementoivassa luokassa. Näin toimien lisätään todennäköisyyttä, että toteutetut operaatiot toimivat kitkatta yhdessä.

Toki ADT-kuvauksella on hyvin läheinen yhteys ohjelmointiin, nimenomaan abstrakteihin rajapintoihin. Esimerkiksi aiemmin esitelty `Pino`-rajapinta on erittäin informatiivinen ja hyvin lähellä vastaavaa ADT:tä, koska se esittelee rutiinien semantiikan. *Itse asiassa se on lähes identtinen ADT-kuvauksen kanssa*:

(a) TYYPPI-osa vastaa luokan nimeä,

[1] Jos et vielä ole huomannut, oliokielellä ohjelmointi on suurelta osin käsitteiden mallintamista ja siten hyvin kaukana ”matalan tason koodaamisesta”. Tämä käsitys on vahvasti esillä periytymiseen liittyvissä kysymyksissä.

TYYPPI

Pino[X]

FUNKTIOT

Pino:		$\rightarrow$ Pino[X]
lisää:	Pino[X] $\times$ X	$\rightarrow$ Pino[X]
poista:	Pino[X]	$\not\rightarrow$ Pino[X]
päällimmäinen:	Pino[X]	$\not\rightarrow$ X
onTyhjä:	Pino[X]	$\rightarrow$ Boolean

ALKUEHDOT

poista(s: Pino[X]) **vaatii** onTyhjä(s) = FALSE
päällimmäinen(s: Pino[X]) **vaatii** onTyhjä(s) = FALSE

AKSIOOMAT

Jokaista alkiota x: X ja pinoa s: Pino[X] kohti on voimassa:
onTyhjä(Pino()) = TRUE
onTyhjä(lisää(s, x)) = FALSE
poista(lisää(s, x)) = s
päällimmäinen(lisää(s, x)) = x

Kuva C.1: Pinon abstraktin tietotyyppin määrittely.

(b) FUNKTIOT-osa kertoo piirteiden signatuurit,

(c) ALKUEHDOT ilmaisee rutiinien alkuehdot, ja

(d) AKSIOOMAT-osan väittämät löytyvät rutiinien loppuehdoista (ja luokkainvariantista).

Joskus AKSIOOMAT-osan predikaattien lausumiseen tarvitaan avuksi rekursiota, kuten näkyy geneerisen *jonorakenteen (queue)* ADT-määrittelystä kuvassa C.2. Jonon ydinajatuksenahan on toimia FIFO-periaatteella (eli *first-in, first-out*). Huomaa, miten poista- ja ensimmäinen-operaatioiden kuvaamiseen on käytetty aksioomaparia, jonka ensimmäinen osa kuvaa operaation käyttäytymistä tyhjälle ja jälkimmäinen ei-tyhjälle jonolle. Se, että q on ei-tyhjä jälkimmäisissä aksioomissa seuraa taas siitä, että funktiokutsujen oletetaan täyttävän ALKUEHDOT-vaatimukset. Jos tällaista riippuvuutta on vaikea nähdä, kyseiset aksioomaparit voidaan esittää myös valintarakenteen avulla:

poista(lisää(q, x)) = **if** onTyhjä(q) **then** Jono() **else** lisää(poista(q),x)
ensimmäinen(lisää(q, x)) = **if** onTyhjä(q) **then** x **else** ensimmäinen(q)

Vertaamalla pino- ja jonotyyppien FUNKTIOT-osia nähdään, että funktiot ovat hyvin samankaltaisia (ja nimetkin täsmäävät pääosin). Tämä korostaa sitä, että ADT-kuvauksen tärkein osa, AKSIOOMAT, on syytä lukea tarkkaan, koska vasta se lopullisesti paljastaa tyypin ominaisuudet (esimerkeissä erottaa LIFO- ja FIFO-käyttäytymisen).

Tässä kohden on myös aiheellista kysyä että johtuuko pinon ja jonon ADT-määrittelyjen yksinkertaisuus kenties siitä että niiden operaatiot ovat ennalta kohdistettuja tiettyyn kohtaan tietorakennetta (eli ne ovat ns. *dispenser*-tietorakenteita). Tarkastellaanpa tilannetta, jossa alkiota (tyyppiä X) käsitellään erillisen hakuavaimen (tyyppiä K) kautta. Voimme määritellä tällaiselle käyttäytymiselle ADT:n Säiliö[K, X] esimerkiksi kuvan C.3 mukaisesti. Huomaa että vaikka säiliörakenteen aksioomia on vain yksi enemmän kuin jonorakenteella, säiliön toiminta on paljon monipuolisempaa. Esimerkiksi lisää- ja poista-funktio on määritelty siten että ADT pystyy poiston jälkeen ”palauttamaan” aikaisemmin tehdyn avain-alkio -sidoksen mikäli sellainen on olemassa. Jotta hakuavainta voitaisiin käyttää, tyypin K on annettava määrittely alkioittensa =-vertailulle.

Säiliö- ja jono-ADT:n vertailusta voimme päätellä että ADT-määrittelyn yksinkertaisuus ei johdu pelkästään yksinkertaisista tyyppiin kohdistuvista funktioista. Kyse näyttäisi enemmänkin olevan funktioiden keskinäisten suhteiden luontaisesta yhteensopivuudesta ja funktioiden lukumäärästä. Tämähän puolestaan tarkoittaa, eksaktin tyypin määrittelyn lisäksi, että ADT-ajattelua voidaan käyttää ohjaamaan myös luokan piirteiden määrittelyä kohesiiviseksi.

TYYPPI

Jono[X]

FUNKTIOT

Jono:		$\rightarrow$ Jono[X]
lisää:	Jono[X] $\times$ X	$\rightarrow$ Jono[X]
poista:	Jono[X]	$\not\rightarrow$ Jono[X]
ensimmäinen:	Jono[X]	$\not\rightarrow$ X
onTyhjä:	Jono[X]	$\rightarrow$ Boolean

ALKUEHDOT

poista(q: Jono[X]) **vaatii** onTyhjä(q) = FALSE
ensimmäinen(q: Jono[X]) **vaatii** onTyhjä(q) = FALSE

AKSIOOMAT

Jokaista alkiota x: X ja jonoa q: Jono[X] kohti on voimassa:

onTyhjä(Jono()) = TRUE
onTyhjä(lisää(q, x)) = FALSE
poista(lisää(Jono(),x)) = Jono()
poista(lisää(q, x)) = lisää(poista(q),x)
ensimmäinen(lisää(Jono(), x)) = x
ensimmäinen(lisää(q, x)) = ensimmäinen(q)

Kuva C.2: Jonon abstraktin tietotyyppin määrittely.

TYYPPI

Säiliö[K, X]

FUNKTIOT

Säiliö:		$\rightarrow$ Säiliö[K, X]
lisää:	Säiliö[K, X] $\times$ K $\times$ X	$\rightarrow$ Säiliö[K, X]
poista:	Säiliö[K, X] $\times$ K	$\rightarrow$ Säiliö[K, X]
hae:	Säiliö[K, X] $\times$ K	$\not\rightarrow$ X
sisältää:	Säiliö[K, X] $\times$ K	$\rightarrow$ Boolean

ALKUEHDOT

hae(r: Säiliö[K, X], k: K) **vaatii** sisältää(r, k) = TRUE

AKSIOOMAT

Jokaista alkiota x: X, hakuavainta k, k': K ja säiliötä r: Säiliö[K, X] kohti on voimassa:

sisältää(Säiliö(), k) = FALSE
sisältää(lisää(r, k, x), k') = **if** $k = k'$ **then** TRUE **else** sisältää(r, k')
poista(Säiliö(), k') = Säiliö()
poista(lisää(r, k, x), k') = **if** $k = k'$ **then** r **else** lisää(poista(r, k'), k, x)
hae(lisää(r, k, x), k') = **if** $k = k'$ **then** x **else** hae(r, k')

Kuva C.3: Abstraktin tietotyyppin määrittely avaimen mukaan tietoa käsittelevälle säiliötietorakenteelle.

Tietotyypin käyttäytyminen voidaan ilmaista, paitsi aksioomien, myös ”primitiivisemmän” tietotyypin rakenteen avulla, jolloin on kyseessä ns. *konstruktiivinen kuvaus* (*constructive specification*). Tämä tapa on lähellä luokka-ajattelua, missä sisäinen esitystapa vastaa primitiivistä tietotyyppiä. Tyyppien muodostama kokonaisuus voidaan tällöin nähdä päällekkäisten tietotyyppien sipulirakenteena. Esimerkiksi tyyppi ”jono” voidaan toteuttaa käsitteen ”lista” ja sille ominaisten operaatioiden avulla. Esimerkiksi jonon päivitysfunktiot voitaisiin määritellä konstruktiivisesti seuraavalla tavalla:

$$\text{lisää}([x_0, x_1, \ldots, x_m],\, x) = [x_0, x_1, \ldots, x_m] \;||\; [x]$$
$$\text{poista}([x_0] \;||\; [x_1, \ldots, x_m]) = [x_1, \ldots, x_m]$$

missä merkintä $[x_0, \ldots, x_m]$ edustaa listaa jossa alkiot $x_0, \ldots, x_m$ ovat peräkkäin (joten $[\,]$ tarkoittaa tyhjää listaa) ja $||$-funktio liittää kaksi listaa yhteen.

Tyypin ”lista” kuvaus voisi puolestaan nojautua vaikkapa yksiulotteiseen taulukkoon jne. Näin siis abstraktit tietotyypit (ja luokat, joilla ne toteutetaan) muodostavat muiltakin tieteenaloilta tutun mallihierarkian, jossa alemman tason mallit antavat yksityiskohtaisemman selvityksen niistä ilmiöistä, jotka tapahtuvat ylemmän tason malleissa. On syytä huomata, että tässä hierarkiassa voidaan sekoittaa tietotyyppejä tavalla, joka ei välttämättä ole aina intuitiivista. Esimerkiksi normaali lineaarinen lista, missä alkiot pidetään nousevassa suuruusjärjestyksessä, voidaan hyvinkin toteuttaa binäärisenä hakupuuna. Tällä mallihierarkialla ei tietenkään ole mitään tekemistä periytymishierarkian kanssa, sillä konstruktiivisessa kuvauksessa ”alemman tason” tyyppiä käytetään asiakasrelaationomaisesti. Vaikka konstruktiivinen ADT-kuvaus siis saattaa ensialkuun tuntua houkuttelevan selkeältä, suosittelemme pelkästään algebrallista, aksioomiin perustuvaa, määrittelyä. Tällöin huomio tulee kiinnittyneeksi aina abstraktioon, ei sen ”toteutukseen”.

Kirjallisuutta

[1] **Arnold, K. & Gosling. J.:** *The Java Programming Language,* 2nd Edition, Addison-Wesley, 1999

[2] **Bloch, J.:** *Effective Java Programming Language Guide,* Addison-Wesley, 2001

[3] **Cormen, T. & Leiserson, C. & Rivest, R.:** *Introduction to Algorithms,* McGraw-Hill, 1991

[4] **Eckel, B.:** *Thinking in Java,* 2nd Edition, Prentice-Hall, 2000

[5] **Gamma, E. & Helm, R. & Johnson, R. & Vlissides, J.:** *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995

[6] **Langer, A.:** *Java Generics FAQ,* ⟨http://www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html⟩

[7] **Liskov, B. & Guttag, J.:** *Program Development in Java,* Addison-Wesley, 2001

[8] **Meyer, B.:** *Object-Oriented Software Construction,* 2nd Edition, Prentice-Hall, 1997

[9] **Sommerville, I.:** *Software Engineering,* 3rd Edition, Addison-Wesley, 1989

Hakemisto