Haage-Helia

Perintä ja rajapinnat

CC BY NC SA

Haaga-Helia

Mitä perintä on?

- Perinnän avulla olemassa olevia luokkia voidaan käyttää pohjana uusille luokille siten, että perivä luokka perii kaikki perittävän luokan oliomuuttujat ja -metodit.
- Kun osa luokkasi tarvitsemasta toiminnallisuudesta on jo olemassa toisessa luokassa, perimällä tämän luokan vältyt kirjoittamasta samaa koodia toistamiseen.
- Perinnän avulla voidaan myös toteuttaa erilaisia vaihtoehtoisia toteutuksia:
  - ArrayList ja LinkedList perivät molemmat AbstractList-luokan, ja ovat kaksi erilaista toteutusta Javan listoista

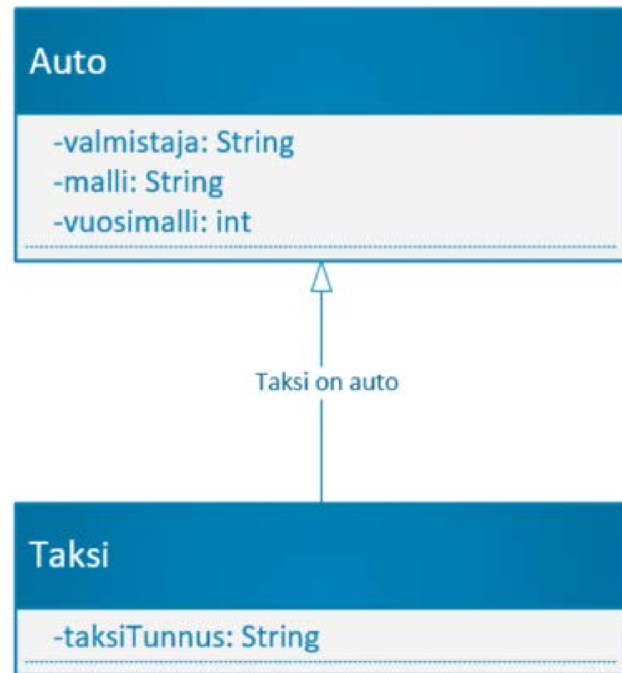
Terminologiaa:

- Perittävä luokka
  - Yläluokka / superclass
- Perivä luokka
  - Aliluokka / subclass

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Haaga-Helia

Esimerkki perinnästä



- Ohjelmassa käsitellään autoja, joita on erilaisia: tavallisia autoja ja takseja.
- Kaikki taksit ovat autoja, mutta kaikki autot eivät ole takseja.
- Kaikkien autojen yhteiset ominaisuudet voidaan toteuttaa Auto-luokkaan.
- Taksi-luokka voi periä yhteisen toiminnallisuuden ja toteuttaa vain omat erityispiirteensä.

- Taksi on tässä tapauksessa Auton aliluokka ja Auto on Taksin yläluokka

Haaga-Helia

Luokkien toteutus

Auto on kuin mikä tahansa luokka

extends Auto, eli Taksi perii luokan Auto

```
public class Auto {

    private String valmistaja;
    private String rekisterinumero;

    public Auto(String valm, String rekkari) {
        this.valmistaja = valm;
        this.rekisterinumero = rekkari;
    }

    public String getValmistaja() {
        return valmistaja;
    }

    public String getRekisterinumero() {
        return rekisterinumero;
    }
}
```

```
public class Taksi extends Auto {

    private String taksinumero;

    public Taksi(String valm, String rekkari, String numero) {
        super(valm, rekkari);
        this.taksinumero = numero;
    }

    public String getTaksinumero() {
        return taksinumero;
    }
}
```

Auto-luokan metodit periytyvät **Taksi**-luokalle. Ne löytyvät Taksi-olioilta vaikka niitä ei Taksi-luokan lähdekoodissa näykään.

Haaga-Helia

Olioiden luonti ja käyttäminen

```
Auto auto = new Auto("Opel", "ABC-123");
Taksi taksi = new Taksi("Skoda", "DEF-456", "H123");

auto.getRekisterinumero();

taksi.getRekisterinumero();

taksi.getTaksinumero();
```

Molemmilla luokilla on omat konstruktorinsa. Taksille annetaan lisäksi taksinumero.

Sekä Autoilla että Takseilla on Auto-luokan metodit.

Taksilla on lisäksi Taksi-luokan metodit.

Haaga-Helia

Ylä- ja aliluokan konstruktorit

- Aliluokan konstruktorista voidaan kutsua yläluokan konstruktoria **`super`**-avainsanalla
- Yläluokan konstruktori alustaa yläluokan ominaisuudet ja aliluokan konstruktori alustaa aliluokan ominaisuudet
- **`super`**-konstruktorikutsun on oltava ensimmäisenä, mikäli sellaista käytetään

```
public Taksi(String valm, String rekkari,
             String taksinumero) {
    super(valm, rekkari);
    this.taksinumero = taksinumero;
}
```

Yläluokan tarvitsemat arvot annetaan eteenpäin yläluokan konstruktorille `super`-kutsulla (superclass)

```
public Taksi(String valm, String rekkari,
             String taksinumero) {
    this.valmistaja = valm;
    this.rekisterinumero = rekkari;
    this.taksinumero = taksinumero;
}
```

Muuttujia ei voida asettaa yläluokan muuttujiin suoraan konstruktorissa, koska niiden näkyvyydeksi määritettiin `private`.

Haaga-Helia

Rajapinnat

interfaces

Haaga-Helia

Motivointi

- Osaamme tässä vaiheessa käyttää listoja ja oliota sekä suorittaa ehto- ja toistorakenteita:
  - Näin voimme etsiä esimerkiksi listalta pienimmän tai suurimman muodon tai maan
- Nykyisiä taitojamme soveltaen osaamme toteuttaa olioiden vertailun ja etsimisen, mutta joudumme toteuttamaan sen erikseen maille ja muodoille.
- Itse algoritmi suurimman alkion etsimiseen on täysin sama riippumatta siitä, ovatko vertailtavat oliot maita, muotoja tai muita vertailtavia tyyppejä.
- Miten voisimme siis toteuttaa vertailun ja suurimman alkion etsimisen siten, että sama koodi toimisi mille tahansa luokille?
  - **Hyödynnetään niin sanottuja rajapintoja!**

Rajapinnat

- **Luokat voivat periä vain yhden yläluokan, mutta ne voivat toteuttaa useita rajapintoja**
- Rajapintoihin määritellään sellaiset metodit, jotka rajapinnan toteuttavien luokkien on toteutettava
- Rajapinnat eivät voi sisältää konstruktoreita tai oliomuuttujia
- Rajapinnat määrittelevät usein yksittäisen metodin, joka toteuttaa tietyn käyttötarkoituksen:
  - Javan standardikirjastossa on lukuisia *able-nimisiä rajapintoja, kuten: AutoCloseable, Cloneable, Closeable, Comparable, Iterable, Runnable, Serializable…
- Tähän asti olemme käyttäneet mm. seuraavia rajapintoja:
  - List, Map…
  - Esimerkiksi meille tutut luokat ArrayList ja HashMap toteuttavat edellä mainitut rajapinnat.

Haaga-Helia

Message-rajapinta

Rajapinnan määrittelyn otsikkoon tulee luokasta poiketen avainsana "interface"

```
public interface Message {

    public String getRecipient();
    public String getContent();
}
```

- Rajapintaan määritellään ne metodit, jotka rajapinnan toteuttavien luokkien on sisällettävä
  - Nimet, näkyvyydet, paluuarvot, parametriarvot

- *Rajapintaan voidaan määritellä myös toteutuksia metodeille, mutta sitä ei käsitellä tällä kurssilla*

Haaga-Helia

Rajapinnan toteuttaminen

- Rajapinnan toteuttavan luokan otsikkoon lisätään "`implements`"-avainsana ja sen jälkeen rajapinta, jonka luokka toteuttaa
- Luokkaan on toteutettava rajapinnassa määritellyt abstraktit metodit samoilla parametri- ja paluuarvoilla
- Luokan oliomuuttujat, konstruktorit ja muut metodit voidaan toteuttaa täysin vapaasti

Luokka `EMail` toteuttaa rajapinnan `Message`

```
public class EMail implements Message {

    private String recipient, content;

    public EMail(String recipient, String content) {
        this.recipient = recipient;
        this.content = content;
    }

    @Override
    public String getRecipient() {
        return this.recipient;
    }

    @Override
    public String getContent() {
        return this.content;
    }
}
```

Haaga-Helia

Rajapinta muuttujan tyyppinä

```
Message email = new EMail("user@example.com", "This is an email!");
Message sms = new SMS("+35850555555", "This is an SMS!");

email.getRecipient();
sms.getRecipient();
```

- Rajapintaa voidaan käyttää mm. muuttujien tyyppinä, jolloin:
  - muuttujaan voidaan sijoittaa kaikkia rajapinnan täyttäviä olioita
  - muuttujan kautta voidaan kutsua rajapinnassa määritettyjä metodeja
- Vertaa:

```
List<String> lista = new ArrayList<>();
```

Luokka `ArrayList` toteuttaa `List` –rajapinnan. Tähän muuttujaan voitaisiin yhtä hyvin sijoittaa `LinkedList`-olio.

Haaga-Helia

Javan valmiit rajapinnat

Esimerkki: Comparable-rajapinta

Javan valmiit rajapinnat

- Javassa on lukuisia valmiita rajapintoja, joita hyödynnetään standardikirjaston luokissa.
- Toteuttamalla tietyn rajapinnan omassa luokassasi, pystyt hyödyntämään sitä käyttävää logiikkaa standardikirjastosta.
- Esimerkiksi `Collections.sort` –metodi osaa järjestää mitä tahansa listoja, joiden alkiot toteuttavat `Comparable`-rajapinnan.
- Sinun ei siis tarvitse toteuttaa oman luokkasi olioiden järjestelyä itse, vaan riittää, että toteutat rajapinnan ja sort-metodi hoitaa loput:

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

https://docs.oracle.com/javase/9/docs/api/java/util/Collections.html

Comparable-rajapinnan toteuttaminen 1 / 2

- Jos haluamme hyödyntää Javan valmista järjestämislogiikkaa myös omassa Muoto-luokassamme, voimme toteuttaa siinä Comparable-rajapinnan:

```java
public abstract class Muoto implements Comparable<Muoto> {
    // ...
}
```

Kuten kokoelmat, Comparable-rajapinta on "geneerinen", eli sille määritellään minkä tyyppisiin arvoihin tätä luokkaa voidaan vertailla. Tässä tapauksessa haluamme verrata muotoja toisiin muotoihin.

Comparable-rajapinnan toteuttaminen 2 / 2

- `Comparable`-rajapinnassa on yksi abstrakti metodi: `compareTo`
- `compareTo` saa parametrinaan sen tyyppisen olion, joka `implements`-avainsanan jälkeen määriteltiin vertailtavaksi tyypiksi
- Metodin tulee palauttaa esim. jokin seuraavista kokonaisluvuista:

-1	jos tämä on pienempi kuin vertailtava olio
0	jos oliot ovat yhtä suuria
1	jos tämä on suurempi kuin vertailtava olio

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

https://docs.oracle.com/javase/9/docs/api/java/lang/Comparable.html

Haaga-Helia

compareTo-metodi Muoto-luokassa, esimerkiksi:

```
@Override
public int compareTo(Muoto toinen) {
    if (this.pintaAla() < toinen.pintaAla()) {
        return -1;
    } else if (this.pintaAla() == toinen.pintaAla()) {
        return 0;
    } else {
        return 1;
    }
}
```

Kaikkia muotoja voidaan nyt vertailla keskenään niiden pinta-alan mukaan. Collections.sort pystyy nyt järjestämään muodot kooltaan kasvavaan järjestykseen.

Kokonaislukujen vertailu voitaisiin suorittaa myös esimerkiksi [Integer.compare]-metodilla, mutta tällöin rajapinnan idea ei tulisi yhtä hyvin esiin.