



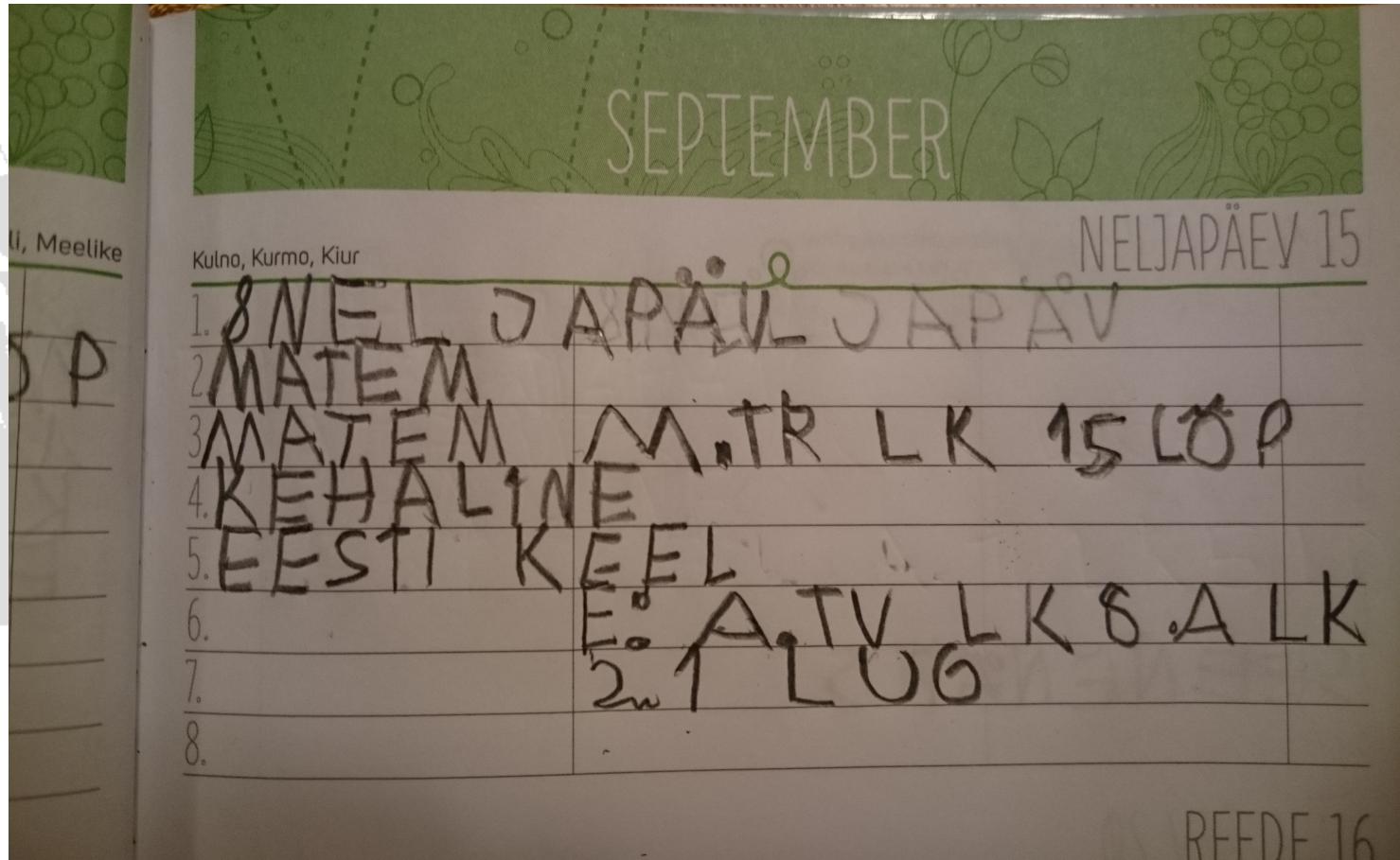
1918

**TALLINNA TEHNIAÜLIKOOL**  
TALLINN UNIVERSITY OF TECHNOLOGY

# IDK0051 Objektorienteeritud programmeerimine Javas

Martin Rebane ([martin.rebane@ttu.ee](mailto:martin.rebane@ttu.ee))

# Clean code



# Mida oleme õppinud?

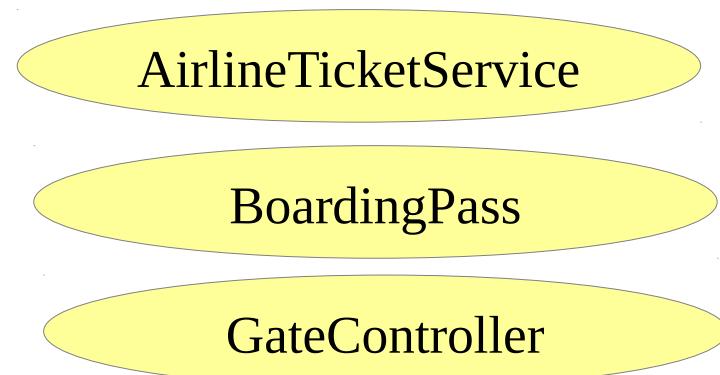
- Mille poolest erineb objektorienteeritud programmeerimine protseduurilisest?
- Infosüsteemide loomisel saab liideseid (*interface*) kasutada tehnilise kokkuleppena, mis võimaldab süsteemi arendamist sõltumatute osadena

# Mida oleme õppinud?

- Tüpiseeritud keeles on igal muutujal tüüp, mis võimaldab koodi õigsust kontrollida juba kompileerides (ennetades vigu töö ajal)
- Tüüpidel on hierarhia, mille piires on teatud määral võimalik objekte teisendada ühest tüübist teise

# Mida veel õppisime?

- Lisaks tehnilistele kontseptsioonidele, saame klasse liigitada ka ärilise otstarbe järgi
  - teenus (service)
  - domeenimudel (model)
  - kontroller (juhib tööd)



# Äriloogika ja domeen

- Äriloogika all peame silmas reegleid, mis puudutavad süsteemi tegelikku kasutust (ja ei tähenda tingimata, et tarkvara peaks kasutama äriettevõttes)
- Näide: lennufirma pardakaarti **saab/ei saa** luua enne kui pilet eest on raha laekunud
  - Tehnilisi piiranguid ei ole, äriloogika

# Ärikoogika ja domeen

- Näide: õppeinfosüsteemis ei saa lõputöö teemat kinnitada kui tudengil on õppemaksu võlg
  - Tehnilist piirangut ei ole, ärikoogiline otsus
- Näide: sulgpalliväljakut ei saa broneerida kaks inimest samaks ajaks
  - Ärikoogiline piirang, infotehnoloogiliselt saaks küll

# Ärioloogika ja domeen

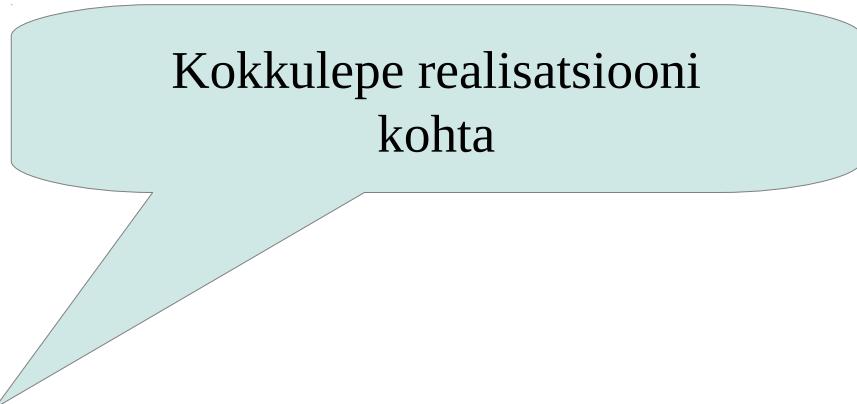
- Domeen
  - Laiemas mõttes valdkond ja sellega seonduv, nt lennundus
  - Kitsamas mõttes mingi alamosa, nt BoardingPass

# Äriloogika ja domeen

- Äriloogika ehk tegelikud kasutusjuhud ja vajadused suunavad objektorienteeritud disaini samavõrd kui tehnilised aspektid
- Äriloogika ja domeenid peegelduvad paketinimedes, liidestes, klassides, meetodites

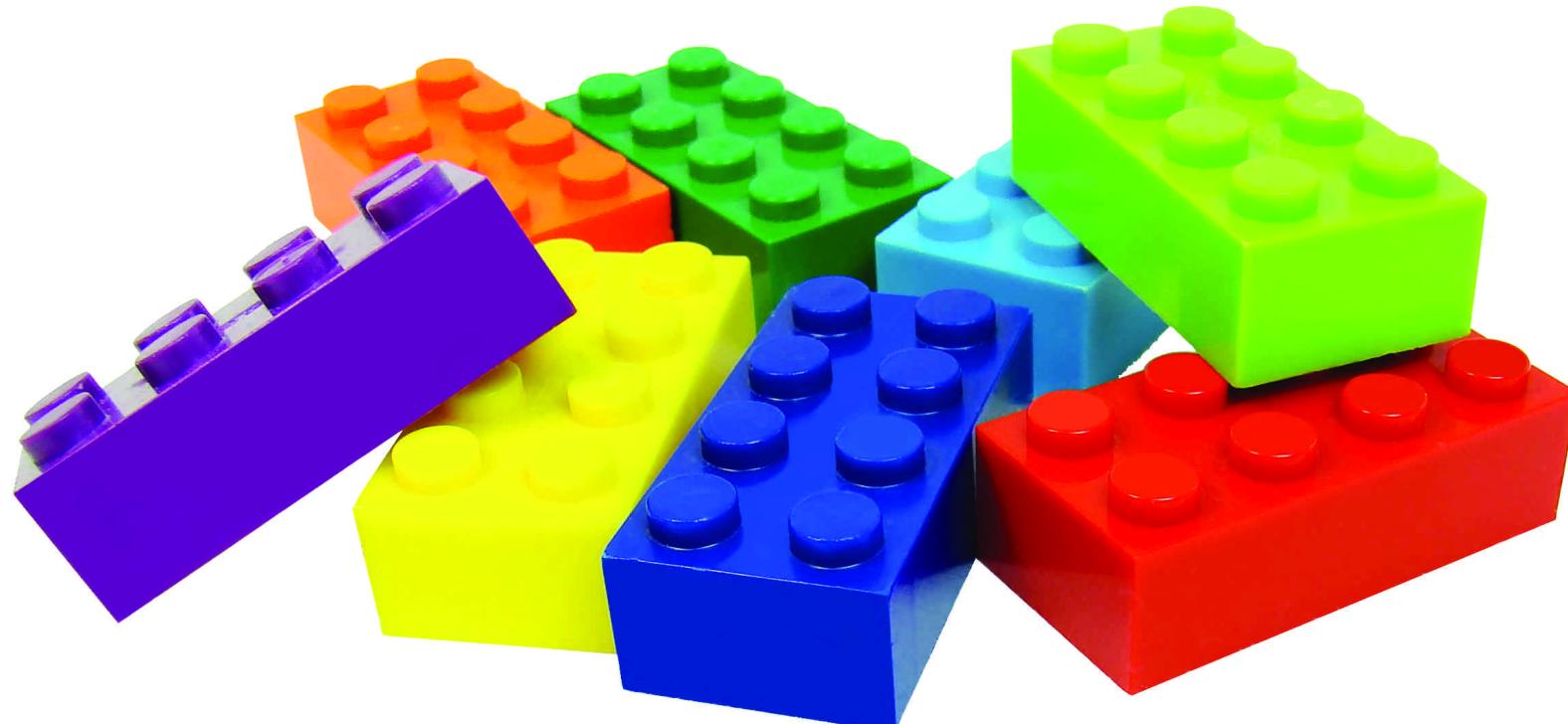
# Eelmisel nädalal

- Klasside laiendamine (pärimine)
- Paketid
- Kompositsioon
- Klass kui tüüp
- Ülemtüüp ja alamtüüp
- Tüübiteisendus (*casting*)
- Liidesed (*interface*), liides kui tüüp
- Geneeriline tüüp (*generic type*)



Kokkulepe realisatsiooni  
kohta

# Eelmisel nädalal



Täna



[https://www.klotkipood.ee/files/prod\\_main/rqmkm152c446276b5a2.jpg](https://www.klotkipood.ee/files/prod_main/rqmkm152c446276b5a2.jpg)

# Ettekanne: Java 9 moodulid

# Pakett: tüüpide kogu

Klassid

Enumeraatorid

Liidesed

Annotatsioonid

Liigendab koodi

Kontrollib ligipääsu  
(*package-private* nähtavus)

Välđib nimekonflikte  
*java.util.List* vs *java.awt.List*

Enamasti avalikud meetodid e API

Liides

Klass *implements* Liides

Konstruktor

Alamklass extends Klass

Konstruktor

Esitab nõudmised  
realisatsioonile

Realiseerib liideses  
nõutud meetodid (lisab uusi)

Pärib ülemtüüpide meetodid  
(+ lisab uusi, kirjutab üle)

Objekt

Konstruktor loob objekti, tagastab viite objektile

# Mis tüüpi objektiga on tegu?

- Liides
- Object
- Klass
- Alamklass

Iga klassi ülemklass

See objekt pärib Object,  
Liides, Klass ja  
Alamklass meetodid  
ja väljad

Objekt

# Meetodi ülekirjutamine ja ülelaadimine

# Meetodi ülekirjutamine (*override*)

- Alamklass defineerib sama nime ja samade argumentidega meetodi
- Tagastustüüp peab olema sama või originaaltüubi alamtüüp:  
Ülemklass: public Number getTwo() {  
Alamklass: public Integer getTwo() {

# Meetodi ülelaadimine (*overload*)

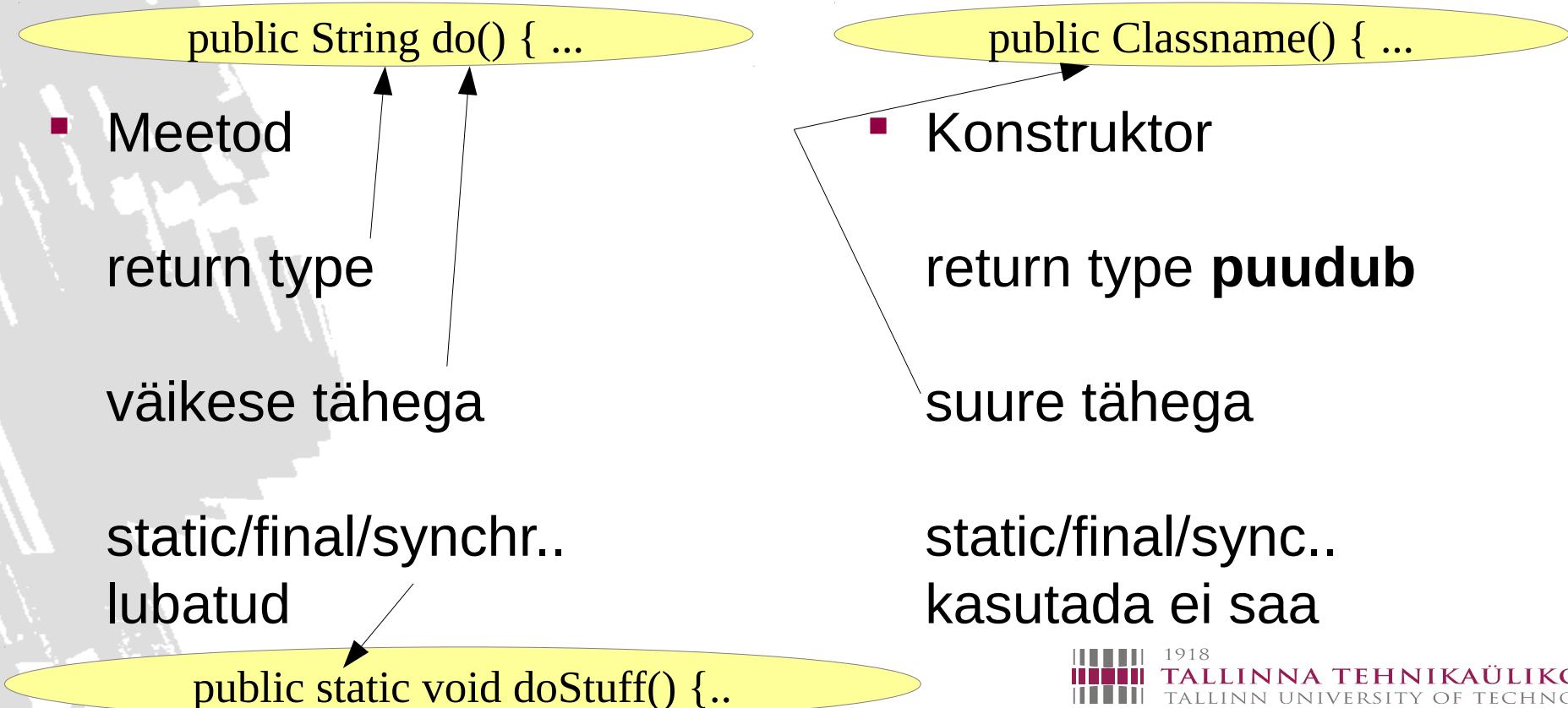
- Mitu sama nime, kuid erinevate argumentidega meetodit

# Konstruktorid

# Konstruktorid

- Meetod, mis käivitatakse automaatselt uue objekti loomisel
- Tagastab viite loodud objektile
- Kasutatakse objekti algväärtustamiseks

# Süntaks: konstruktor vs tavaline meetod



# Automaatne argumendita konstruktor

- Kui teie klassis puudub konstruktor, luuakse vaikimisi argumendita konstruktor

# Konstruktorite ahel

- Uut objekti luues kutsutakse alati välja ülemklassi argumendita konstruktor **enne** kui täidetakse kästud selle klassi konstruktoris

```
public class Class2 extends Class1{  
    public Class2() {  
        System.out.println("Class 2");  
    }  
}
```

- Objekti loomisel Class2-st käivitatakse Class1 konstruktor ja siis Class2 konstruktor

# Argumendiga konstruktor

- Kui kirjutate ainult argumendiga konstruktori, siis kompilaator argumendita versiooni automaatselt ei lisa

# Argumendiga konstruktor

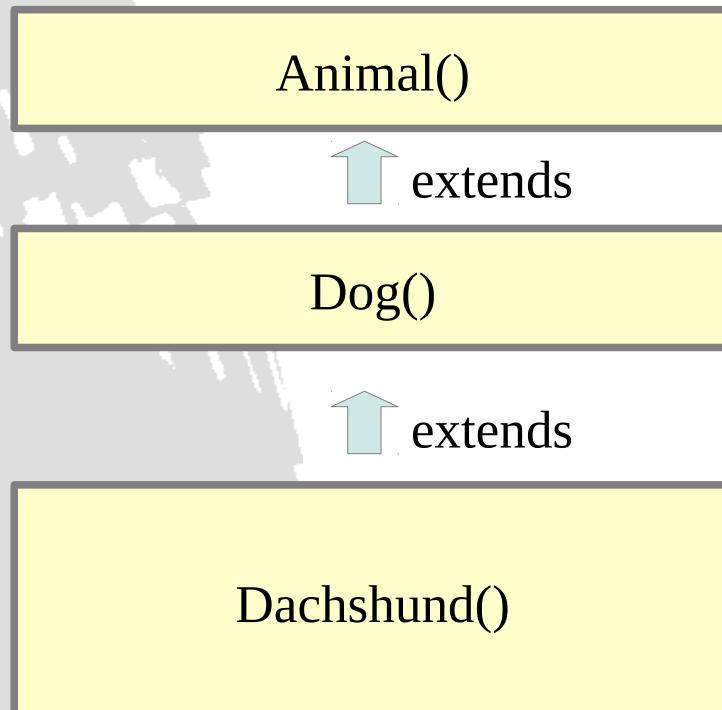
- Mis juhtub kui alamklassis on argumendiga konstruktor, aga ülemklassis konstruktor **puudub**?
- Kompilaator lisab ülemklassile argumendita konstruktori, alamklassist objekti luues käivitatakse see enne alamklassi konstruktorit

## Argumendiga konstruktor (2)

- Mis juhtub kui alamklassis on argumendiga konstruktor ja ülemklassis samuti ainult **argumendiga** konstruktor?
- Programm ei kompileeru, sest automaatselt pole võimalik käivitada ülemklassi konstruktorit

Peate ülemklassi konstruktori  
koodis ise välja kutsuma

# Konstruktorite käivitusahel



**new Dachshund();**

Käivitamise järjekord

1. Animal() konstruktor

2. Dog() konstruktor

3. Dachshund() konstruktor

# Käivitusahela tagajärg:

- Alamklassi konstruktori käivitades on ülemklassi väljad algväärtustatud
- St alamklassi konstruktoris saame kasutada ülemklassi välju ja meetodeid

# Konstruktorite omavaheline väljakutse

- Teil on mitu konstruktorit, näiteks:

```
public YellowSubmarine() {}
```

```
public YellowSubmarine(int weight) {}
```

- Hea tava on käivitada argumendiga konstruktori seest argumendita konstruktor, mitte samu asju dubleerida

# Konstruktorite omavaheline väljakutse

- Näide:

```
public class YellowSubmarine {  
    private int i;  
    private int a;  
    public YellowSubmarine() {  
        a = 4;  
    }  
    public YellowSubmarine(int age) {  
        this();  
        this.i = age;  
    }  
}
```

Argumendiga konstruktor kutsub ülejäänud väljade väärustamiseks argumendita konstruktorit

See väljakutse peab olema enne ülejäänud koodi (1. real)

# Ülemklassi võimalused

- super() - ülemklassi konstruktori väljakutse
- super – võtmesõnaga saame kasutada ülemklassi (ülekirjutatud) võimalusi, sh ülemklass konstruktorit, välju ja meetodeid

super.overRiddenMethod()

# Viidad

# Viidad

```
public void magicMaker() {  
    Car car = new Car();  
    car.drive();  
    this.doMagic();  
}
```

Viit Car-tüüpi objektile

Objekt viitab iseendale



1918  
**TALLINNA TEHNIKAÜLIKOOL**  
TALLINN UNIVERSITY OF TECHNOLOGY

# Viit objektile endale - this

- Enamasti ei ole tarvis kasutada, piisab meetodi või välja nimest:

```
doSomething();
```

- Kasutatakse nimekonfliktide vältimiseks ja konstruktorites - sh *this()*

# Polümorphism

# Polümorfismi üldine idee

- *poly* – palju, *morphē* – vorme (kreeka k)
- Üldiselt kolme tüüpi polümorfismi:
  - Ülelaadimisest tulenev (ühe nimega meetod, erinevad väljakutsed):  
*run(double distance), run(int time)*
  - Geneeriline polümorfism (geneeriliste tüüpide abil)
  - Pärimisest tulenev polümorfism (*subtyping*)

*List<Car>, List<Toy>*

Objektorienteeritud programmeerimises peame polümorfismi all silmas eelkõige pärimisest tulenevat polümorfset käitumist

# Polümorfism - taustateadmised

- Nägime, et üks objekt võib olla mitut erinevat tüüpi
  - Liidese-tüüpi
    - sh mitme erineva liidese tüüpi
  - Klassi-tüüpi
    - sh ülemklassi tüüpi
    - sh alamklassi tüüpi

# Mis tüüpi objektiga on tegu?

- Liides
- Object
- Klass
- Alamklass

Iga klassi ülemklass

Alamklassi tüüpi  
objekt pärib Object,  
Liides, Klass ja  
Alamklass meetodid  
ja väljad

Objekt

# Polümorfism OOPis

- Erinevat tüüpi objektide kasutamine ühise tüübi abil,  
näiteks:
  - Car-objekt (aga samas ka Vehicle)
  - Bus-objekt (aga samas ka Vehicle)
- Realiseerivad meetodit `noOfPassengers()` erinevalt
  - Kood mis käsitleb kumbagi `Vehicle`-tüüpi objektina, ei pea teadma tegelikku tüüpi, millest objekt loodi ja saab `noOfPassengers()` meetodit üheselt välja kutsuda

# Polümorphse koodi kasutuskohad

- Põhimõtteliselt igal pool
- Infosüsteemid
- Algoritmid
- Võrdlused
- Kõikjal, kus on vaja üldistust

# Alamtüüpe ei esitata väljana ja ifidega!

```
public class Car {  
    private String carType;  
  
    public void drive() {  
        if(carType.equals("Truck")) {  
            // do something  
        } else {  
            // do something else  
        }  
    }  
}
```



Mis viga on?

# OOPis alamtüübide klassidega

```
public class Car {  
    public void drive() {  
        // do something  
    }  
}  
  
public class Truck extends Car {  
    @Override  
    public void drive() {  
        // do something else  
    }  
}
```

