Ord og begreper

# 1.1

Polymorfisme:

Dette er at et objekt kan «utgi» seg for å være et annet objekt, så lenge begge objektene er i samme arv. Eksempel:

public abstract class Animal {

    private String name;

    private String sientificName;

}

public class Bird extends Animal {

    private boolean canFly;

}

public class Invertebrate extends Animal {

    private int legs;

}

Hvis man her vil ha en funksjon som sammenligner forskjellige dyr, kan man skrive en funksjon som tar in en Bird og en Invertebrate, men det er veldig tungvint å gjøre. Derfor kan man bruke polymorfisme, da skriver man en funksjon som tar inn Animal. Siden Bird og Invertebrate arver fra Animal, kan disse da sammenlignes (så Bird og Invertebrate bil da sett på som Animal).

Ulempen med dette er at vi kan ikke bruke instans variablene til orginal objektet, dette gjør at Bird ikke vil ha canFly variabelen når vi sammenligner med et annet Animal.

# 1.2

Primitive datatyper:

Dette er «enkle» datatyper. Denne formen for data er ikke lik et objekt, siden de ikke har noen ekstra funksjoner. Java har 8 primitive data former: boolean, byte, char, double, float, int, long og short. Leg merke til at ingen av de skrives med stor forbokstav.

*double* testD = 234.5;  
testD.byteValue();  
  
Double testDD = 234.5;  
testDD.byteValue();

Her er tallet 234.5 lagret i den primitive data formen double og den «objekt orienterte» formen. Forskjellen på disse to et at siden testDD er et objekt har man funksjoner som kan gjøre ting. Som en innebygd sammenligning, formatering til andre datatyper. testD har ingen andre funksjoner. Dette vises med at IntelliJ ikke gjør byteValue() i testD.byteValue() blåt, det er ikke en funksjon testD har.

# 1.3

Forskjellen på en klasse og et objekt:

En klasse er en måte å definere hvordan et objekt skal være. Man kan se på det som en blueprint på et objekt.

Et objekt er en instans av klassen man har lagd. Hvis klassen er blueprinten til en bil, er objektet selve bilen. Man kan ha flere biler som er litt forskjellig, men alle bruker samme blueprint.

Eksempel:

public class Car {

    private String color;

    public Car(String color) {

*this*.color = color;

    }

}

Car bil1 = *new* Car("blue");

Car bil2 = *new* Car("Red");

Her har vi definert klassen (blueprint’en) til hvordan man skal lage en bil. De to siste linjene lager vi to biler hvor vær bil har forskjellig farge, bil1 er blå og bil2 er rød. bil1 er et objekt og bil2 er et objekt.

Man kan se litt på en klasse som å definere en ny datatype man kan bruke i programmet.

# 1.4

Forklar programmeringsparadimene:

**Imperativ programmering (Hvordan man gjør noe):**

Når man skriver kode på en imperativ måte, skriver man kode som detaljert forteller hva man vil gjøre. Eksempel:

Array list = [1,2,3,4]

*for* (int i = 0; i < list.length; i++) {

    list[i] = list[i] + 1

}

*//List = [2,3,4,5]*

Her er det klart definert hva jeg skal gjøre. Jeg skal ha en liste med tallene [1,2,3,4]. Så skal jeg ta den listen å gå gjennom vært element og for vært element skal jeg legge 1 til det elementet. Så jeg ender opp med en liste som inneholder [2,3,4,5].

**Funksjonell programmering (Imperativ basert på funksjoner):**

Funksjonell programmering er en måte å lage mer generelle kode biter. Dette gjør at man kan bruke samme logikken på flere verdier. Eksempel:

Array list = [1,2,3,4]

Func add(Array list) {

*for* (int i = 0; i < list.length; i++) {

        list[i] = list[i] + 1

    }

*return* list

}

add(list) *//[2,3,4,5]*

Her gjør jeg mye av det samme som i imperativ, bare at jeg har en funksjon som kan ta in hvilken som helst liste med tall og legge 1 til på vært element. Så jeg kan sende inn en liste som inneholder [1,2,3,4,5,6,7] eller [7,6,5,4,3,2,1] eller [1,1,1,1,1]. Alle disse listene vil fungere, men hadde jeg kodet på en imperativ måte måtte jeg ha skrevet alt på nytt for vær liste.

Funksjonell programmering er også at man lagrer konstante funksjoner. Eksempel på dette er map funksjonen til JavaScript. Denne funksjonen er linket opp til alle lister og gjør gjennom alle elementer og utførere en frivillig funksjon på vær av elementene i listen. Eksempel:

Array list = [1,2,3,4]

Func add(Array list) {

*return* list.map( Func (i) { *return* i + 1 } )

}

add(list) *//[2,3,4,5]*

Her er map() en funksjon som tar inn en anonym funksjon og utfører den funksjonen på alle elementene av list. Eksempel på hvordan map() kan se ut:

map(Func func(i), Array list) {

*for* (int i = 0, i < list.length; i++) {

        func(i)

    }

}

Fordelen med funksjonell programmering er at man skriver mye mindre kode. Siden alle operasjoner på verdier er skrevet en gang er det mye lettere å holde styr på hva som blir gjort.

**Deklarativ programmering (Hva man vil gjøre):**

Dette er en ganske annerledes måte å programmere på en imperativ og funksjonell programmering. Hvor vi i imperativ og funksjonell programmering forteller programmet hvordan vi vil utføre noe, i deklarativ programmering forteller vi programmer hva vi vil ha utført. Noen eksempler på dette er:

- Database språk som SQL:

SELECT \* FROM cars WHERE Color="red"

Her forteller vi programmet at vi vil velge alle rader fra listen cars, hvor kolonnen Color er red. Vi har aldri forklart hvordan vi vil at programmet skal utføre denne oppgaven, eneste er at vi fortalte den hva vi ville ha (alle biler som er røde).

Hvis vi skulle gjort dette på en funksjonell måte kan vi skrive det slik:

Array cars

Array redCars = cars.filter( car => car.color == "red" )

- Markup språk som HTML:

<div *id*="container">

    <h1>Header</h1>

    <p>Text</p>

</div>

Her forteller vi maskinen at vi skal ha en <div> med attributt id som har verdien «container», inni den skal vi ha en <h1> som skal inneholde teksten Header, etter det vil vi ha en <p> som inneholder teksten Text.

Hvordan programmet går frem å vister dette på skjermen har vi ikke definert, det er jobben til nettleseren.

**Kort oppsummering** (bake eksempel)

Imperativ er akkurat hvordan man vil gjøre noe. Alle stegene for å bake en kake, men en satt mengde ingredienser.

Funksjonell er en generalisert imperativ kode. En oppskrifts nettside med hvordan man lager en kake, hvor man kan spesifisere en variabel mengde ingredienser.

Deklarativ er hva man vil ha. Man forteller noen hvilken type kake man vil ha, men bryr seg ikke om hvordan kaken blir lagd.

# 1.5

Ord og begreper

* Class
* Object (konsept, ikke klassen)
* Instance variabel
  + Variabel som man lager øverst i klassen så hvilke variabler man skal sette til objektet.
* Overloading
* Overriding
* Extends
* Private, public, (protected) (klasse, variabel, metode)
* This og super
* Refaktorere
* Static (variabel, metode)
* Final (variabel, metode, klasse)
* Abstract (klasse, metode)
* Interface
* Anonymus inner class
* MVC (konseptet, og hver enkelt del)
* Exception
* Threads
* Collections Framework
  + List
  + Map
  + Queue
  + Stack