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
  + Dette kan bli sett på som en blueprint til et objekt. hvis man skal lage en bil må bilen ha en motor, hjul, en farge, dører, osv. Dette definerer man i klassen slik at når man skal lage et objekt kan man enkelt sette verdiene definert uten å måtte skrive alt på nytt.
* Object (konsept, ikke klassen)
  + Konseptet bak objekter er at hvis man skal lage noe mer enn en gang kan man lage et objekt. Når man lager et objekt kan man se på det som at man lager en ny data type. Så istedenfor å lage en int med et navn og verdi, kan man lage et bil objekt, med navn og verdier. Dette gjør at hvis trenger den bilen igjen trenger man bare å spørre etter navnet, lit som når man spør om verdien til en int.
* Instance variabel
  + Dette er variabler man lager inni en klasse. Når man skal lage klassen til bilen må man lagre all dataen (farge, antal hjul, hvilken mottor, osv), disse dataene definerer man som instance variabler.

Grunnen til at det heter instance variabler er at vær gang man lager et nytt objekt (eller en ny instance) vil det objektet få sine egene variabler.

* Overloading
  + Hvis man skal lage et objekt men vet ikke om man kommer til å definere en verdi, da kan man overloade metoden (koden man skriver for å generer objektet). Dette betyr at man har en metode der man definerer et sett med verdier, og en annen metode, med samme navn, hvor man definerer andre verdier (minst en verdi må være annerledes, og med verdi menes data type).
* Overriding
  + Hvis objektet arver fra et annen objekt, men lager en ny implementasjon. Eksempel er at motorsykkel arver fra kjøretøy, men hvordan man kjører en motorsykkel er annerledes enn en bil. Da må man override funksjonen som forteller hvordan man kjører motorsykkelen, mens bil kan ha en funksjon som arves fra kjøretøy.
* Extends
  + Hvis man skal lage flere type kjøretøy, person bil, motorsykkel, ATV, lastebil, kan man lage en generell klasse som har alle felles funksjoner inni, som farge, antall hjul, motor. Så når man lager lastebil klassen kan man si at den extends kjøretøy, da vil lastebil få alle instansvariabler kjøretøy har og alle fungsjoner, men man kan også lage nye instans variabler og fungsjoner. Eksempel er hvor mange ton last bilen kan ha.
* Private, public, (protected) (klasse, variabel, metode)

public class Car extends vehicle{

    private String color;

    private protected int wheels;

    public Car (String color, int wheels, Engine engine) {

*this*.color = color;

*super*(wheels, engine);

    }

    public String getColor() {

*return* color;

    }

    private void testIfWheelsIsNotNull() {

        ...

    }

}

Car bil = *new* Car("red", 4);

* + I dette eksempelet ser vi at color og wheels er private. Disse verdien kan ikke nås utenfor objektet. Wheels er også protected. Dette betyr at bare fungsjoner i selve klassen og underklasser kan endre på verdien, den er litt strengere enn bare private.
  + Vi ser også at Car og getColor er public, dette betyr at vi kan nå disse funksjonene utenfor selve objektet. Man kan skrive bil.getColor(), da vil man få variabelen color tilbake. Car er jo en metode så den må være public for at man skal kunne initialisere den fra andre steder, som i main.
* This og super
  + This refererer til seg selv. Så i eksemplet over vil this.color referere til «private String color» instans variabelen, mens color refereter til parameteret color.
  + Super refererer til klassen som blir arvet fra. Så hvis vehicle har en metode som initialiserer vehicle, kan man bruke den metoden for å lage dette objetet. Da skriver man in super og sender parameterene som metoden trenger i riktig rekefølge, litt som i eksemplet over hvor vi sender wheels og engine til super for å slippe å skrive implementasjonen i dette objekte siden vi har det over.
* Refaktorere
  + Refaktorering er nesten som å endre navn på en variabel, eneste forskjelden er at når man refakorerer noe vil programmet gå gjennom hele koden å oppdatere alle steder. Dette kan også gjøres hvis man endrer en variabel fra en data type til en annen, da går programmet gjennom alle filer å ser hvor den variabelen var brukt å endrer til riktig type.
* Static (variabel, metode)
  + Hvis en variabel er static i et objekt vil den være delt mellom alle instanser av objektet, så hvis wheels hadde vært static ville alle biler brukt samme verdien for wheels. Fordelen med dette er at hvis man trenger noe som er likt for alle objekter kan man sette en variabel static. Eksempel er hvis man trenger en stigene id for alle objekter, da setter man en static int id = 0 først, og når man har initialisert første objektet har man en fungsjon som setter den statiste verdien til en lokal verdi og høyner id med et tall.
  + Når man har en metode som er static kan man bruke den metoden (som ligger i klassen) uten å initialisere objektet først. Eksempel er Math objektet i Java. Hvis man skal ta Math.abs(-8) trenger man ikke skrive Math noe = new Math(); så noe.abs(-8), siden abs() funksjonen er static.
* Final (variabel, metode, klasse)
  + En final variabel er litt som en konstant. Hvis man har definert en verdi til en variabel vil man aldri kunne endre verdien til den variabelen. Hvis man ikke har definert en verdi vil den låse seg til første verdi.
  + En final metode gjør man at klasser som arver fra denne ikke kan skrive om metoden. Så hvis toString() i Java hadde vært final måtte alle bruke samme implementasjonen av toString()
  + En final klasse er en klasse som man ikke kan arve fra (eller lage underklasser). Så Math klassen er en klasse som kan være final, siden man kanskje ikke trenger å lage en annen klasse som arver fra Math klassen.
* Abstract (klasse, metode)
  + Abstarct av en klasse gjør at man ikke kan initialisere klassen, for eksempelet over (den med bil) hvis vehicle hadde vært abstact kan man ikke lage et objekt av vehical. Så man kan ikke skrive Vehical vehical = new Vehical. Men man kan arve fra disse klassene, så man kan lage bil klassen.
  + Når en metode er abstact sier man til programmet at denne klassen ikke trenger en implementasjon, men alle underklasser trenger en. Så hvis man vil at alle objekter som arver fra vehical skal ha en testIfWheelsIsNotNull(){} fungsjon, kan man sette den abstract i vehical.
* Interface
  + Interface er litt likt som abstact, stårste forskjellden er at hvis man lager et interface kan man lage det som en egen fil. Enda en av forskjeldene er at et interface ikke kan ha en kode kropp i det hele tatt (så ingen {}).
  + Det er også andre ting man kan gjøre med interface, som å lage et fungsjonelt interface, men det forstå jeg ikke helt så jeg skal ikke forklare hva det er.
* Anonymus inner class
  + En anonym indre klasser er hvis man har en klasse som trenger en annen klasse (som i bil eksemplet over, der den trenger en Engine), trenger man ikke å initialisere den med et navn, man kan droppe alt før = tegnet. Til eksemplet:

Engine engine1 = *new* Engine(...);

Car car = *new* Car("blue", 4, engine1);

Car car = *new* Car("orange", 4, *new* Engine());

Disse to gjør det samme, bare at i den nederste trenger vi ikke å lage et nytt objekt med navn, vi trenger bare å lage et nytt objekt. Siden for å kunne lese motoren til bilen trenger vi bare å skrive car.getEngine(), så engine trenger ikke et «navn».

* MVC (konseptet, og hver enkelt del)
  + MVC er et konsept på hvordan front og back end skal kommunisere med hverandre. I dette konseptet bruker man Model klasser (backend), en View (frontend) og en Controller (den som kommuniserer med front og back end).
    - Fordelen med å gjøre front- og back-end som dette er at man kan endre på enten front- eller back-end uten at den andre vil ha noe effekt.
  + Model er den delen som vil gjøre mesteparten av prosesseringen, i vårt tilfelle er dette Java. Her lages alle klasser og alle kalkulasjoner regnes her. Denne sender all informajsonen til kontrolleren, etter at kontrolleren har spurt om noe data.
  + View er den delen brukere av en tjeneste vil se. Dette pleier å være HTML, CSS og javascript. Denne delen kommuniserer bare med kontrolleren når noe skjer (en knapp trykkes, ny side lastes inn, osv.)
  + Controller er en mellom man som er den som overfører data og info mellom Moden og View (front- og back-end). Denne trenger ikke mye logikk siden den for det meste ikke trenger å regne mye, den må bare sende riktig data til riktig sted.
* Exception
  + Unntak.
  + Exception er når programmet møter på et uforventet unntak i koden. Eksempel hvis vi skal lese en fil, kan det hende at filen ikke er der og hvis vi ikke har noe kode for å håndtere det vil programmet krase. Hvis vi da har skrevet et program for å lese en fil så kan vi kjøre programmet i en try/catch. Det betyr at koden vil prøve å kjøre alt som står i try, men hvis try kaster en exception vil det bli hentet i catch og kode vil kjøre som håndterer problemet, som at filen blir generert.
  + Det finnes mange exception og man kan lage sine egne.
* Threads
  + Når program kjører blir all kode kjørt serielt. Det betyr at koden kjører linje for linje, og programmet venter med å kjøre neste linje til forrige linje er ferdig. Hvis man har en fil skriver vil da hele fil skrivingen kjøre før programmet fortsetter, så hvis man har en veldig stor fil kan det ta veldig lang tid. Der for har programmer og CPU’er tråder.

Disse trådene kan kjøre kode parallelt (men noen restriksjoner). Så vi kan da sette fil skriving til en annen tråd. Da vil programmet kjøre fil skrivingen parallelt med resten av koden.

Restriksjonene med dette er at hvis man har 2 tråder som bruker samme variabel må man passe på at disse leser variabelen til riktig tid, hvis begge trådene skal legge til noe i variabelen vil verdien være det siste som ble lagt til. Så tråd 1 kan lese variabelen, så leser tråd 2 variabelen, men tråd 2 blir ferdig fortere så den lagrer verdien, så blir tråd 1 ferdig og overskriver hva tråd 1 gjorde.

* Collections Framework
  + List
    - Denne lagrer en form for verdi i en liste. Alle elementer i listen vil få en index på hvor de er i listen, første element er i posisjon 0… Hvis man vil ha en verdi kan man lett enten hente ut verdien hvis man vet indexen eller loope gjennom listen å finne riktig verdi sånn.
  + Map
    - En liste der man må definere en nøkkel og en verdi.
    - Denne måten å lagre data på er litt andreledes en List, men også litt likt. Her må man ha en Key og en Value. Key og Value kan være hva som helst, men istedenfor at alle verdier har en satt index på hvor de er, kan man sette en nøkkel som peker til en verdi (Key, Value).

Alle nøkler må være unike, men verdiene må ikke. Så hvis man vil hente ut noe data trenger man bare å oppgi nøkkelen så får man verdien.

* + Queue
    - Når man lagrer data i en Queue er det litt som å lagre data i en kø. Alle elementer vet hvem som er foran og hvem som er bak, men de vet ikke hvor langt bak eller foran de er. For å hetne ut data her må man gå bakover i køen for å finne verdien sin, så du må da spørre, hvem er bak deg, så må du spørre den om hvem som er bak der igjen, osv.

Dette fungerer veldig bra hvis man vil holde styr på en kø med ting som skal gjøres, siden man kan enkelt fjerne et første element å gå til neste element.

* + Stack
    - Dette er litt som queue, bare at man dataen blir lagret mer som i en stabel enn i en kø. Er har man bare tilgang til det siste man la in, og hvis man vil ha det under må man fjerne det elementet over. Dette fungerer bra hvis man vil lagre historikk på hva som ble gjort sist. Som frem og tilbake i nettleseren.