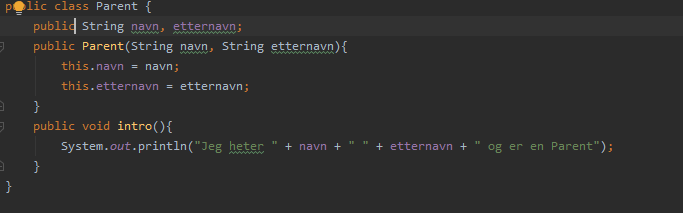
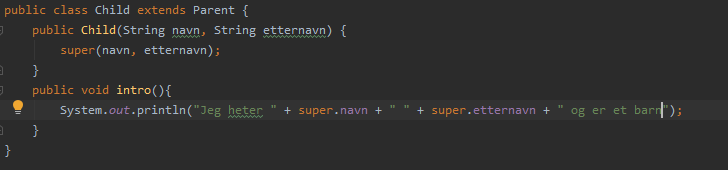
**Oppgave 1 - Ord og begreper**

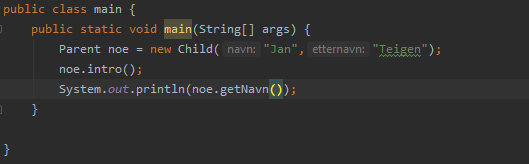
**Oppgave 1.1**

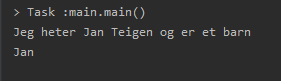
**Forklar begrepet polymorfisme og eksemplifiser med enkel kode.**

Polymorfisme vil si at når en klasse arver fra en annen klasse(parent) så man kan lage et objekt som benytter seg av metodene og variablene som er laget i parent klassen. Dette hjelper for gjenbruk av metoder, så du slipper å skrive mer kode.







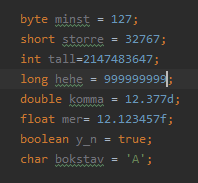


Her har jeg laget en parent klasse og en Child klasse. Child klassen arver fra parent klassen og benytter seg av variablene og metodene som er laget i parent. getNavn er da laget i Parent klassen.

**Oppgave 1.2**

**Forklar hva primitive datatyper er, nevn gjerne noen eksempler.**

Primitive datatyper er datatyper du spesifiserer størrelse og type.



Byte,short,int og long er tall uten desimaler, mens float og double er.

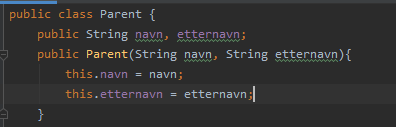
Boolean spesifiserer du om noe er sant eller usant.

Char er for å lagre en spesifikk bokstav

**Oppgave 1.3**

**Forklar forskjellen på en klasse og et objekt. Vis gjerne et lite eksempel på hvordan man definerer en klasse, og oppretter et objekt.**

En klasse er slags ‘’Blueprint’’ du lager som objektet skal benytte seg av. Altså blir klassen som en slags bruksanvisning du lager, mens objektet blir det som bruksanvisningen skal lage.



Dette er da hvordan man definerer en klasse. Du har da navnet Parent, variabler navn og etternavn, og en konstruktør(må ikke ha det) for å opprette et objekt.



Objektet blir opprettet i main metoden og ser slik ut. Man skriver navnet på klassen man vil bruke først, så navnet på objektet, og deretter new navnPåKlassen(). Inne i klammene skriver man noe dersom man lager en konstruktør med parametere. Ved default så er de tomme.

**Oppgave 1.4**

**Forklar programmeringsparadigmene:**

**Imperativ programmering**

Imperativ programmering vil si at du programmerer med mål om hvordan koden skal oppnå det den skal. Her er koden fyllt med statements om hva som faktisk skjer når man bruker koden. Dette er en oversiktlig måte å programmere på da du kan steg for steg se hva som skjer i funksjonen du lager.

**Funksjonell programmering**

Funksjonell programmering vil si at funksjonene/metodene du lager skal ikke påvirke globale variabler. Det vil si at du skal holde variablene separert fra funksjonene når du definerer funksjonene. Det vil da være mye bedre å ha variabelen som et argument slik at om du f.eks skal returnere den variabelen, så vil den returnerte verdien bare befinne seg i funksjonen som bruker den. Om du ikke har et argument om den variabelen, så vil return påvirke dens globale verdi.

En annen ting er at du ikke må bruke den samme variabelen og oppdatere/endre den hele tiden. Det er mye bedre å deklarere nye variabler så det blir mer oversiktlig, selv om endringene er minimale.

Funksjoner kan i tillegg bli deklarert av variabler, brukt som argument og returnert i en annen funksjon.

**Deklarativ programmering**

Deklarativ programmering er at koden din går rett på sak uten spesifikke statements, altså at man kan for eksempel bruke innebygde funksjoner/metoder for å oppnå noe. Du sier da at du vil noe, men spesifiserer ikke hvordan det skal skje. Du stoler på funksjonen du benytter deg av. Du bruker mye færre linjer, og er mer oversiktlig (Om du vet hva metodene gjør og ikke glemmer det).

**Oppgave 1.5**

**Lag deg en oversikt over hva følgende ord/begreper betyr, med egne ord. Det bør forklares såpass tydelig at det kommer klart frem hva det er for en som ikke kan det fra før, og du får vist din forståelse av det.**

**Class**

En Class er det som definerer et objekt. Det er en oversikt/blueprint over hva som skal med for å lage objektet. En Class vil da inneholde metoder, variabler etc som er relevant. Dette blir som regel laget først. Du kan velge å lage en konstruktør, som er en metode med samme navnet som classen. Blir da public classNavnet(){}. Inne i () så skriver du variablene du har definert, som du ønsker å ha med. I {} så definerer du hva variablene skal peke på. Dette gjør det lettere å opprette et objekt, fordi da forcer du objektet til å definere variablene under opprettelse.

**Object (konseptet, ikke klassen)**

Object er da som blir laget og tar i bruk det som har blitt laget i Classen. Den benytter seg av hva som skal med og definerer variablene under opprettelse. Oppretter det ved å velge hvem Class du vil bruke. Du skriver da Class navnetPåObjektet = new Class();. Om en konstruktør er definert i klassen, så skriver du inn verdiene i disse (). Har den ingen konstruktør må du kalle på set metodene som er laget i klassen for å legge inn data til objektet. Objekt må opprettes i main.

**Instance variable**

Instans variabler er variabler man definerer når man lager en Class. Variablene kan være public eller private. Når man deretter oppretter et objekt av den klassen. Så instansierer du variablene for dette objektet. Eksempel:

Class person{public String name;}.

Person fyr = new Person();

fyr.name(‘’Tom‘’);

**Overloading**

Overloading vil si at du lager metoder som har samme navn, men med forskjellige argumenter. Dette forenkler programmet slik at du ikke trenger å huske på så mange navn.

**Overriding**

Ovveride vil si at du skal endre en allerede eksisterende funksjon(toString, get eller set metoder etc). Dette kan være om du ønsker at den skal skrive ut mer informasjon, eller vil bare ha en liten vri.

**Extends**

Extends bruker du om du ønsker at et Class skal arve fra en annen Class. Dette er for gjenbruk av metoder og variabler, så du slipper å skrive samme kode flere ganger.

**private,public,(protected) (klasse,variabel,metode)**

Private: Private metoder og instansevariabler kan bare blir brukt i den instansen av klassen du lager. Eksempelvis om du lager et objekt: objekt h = new objekt();. Når du bruker metodene for å legge inn variabler, så vil de variablene bare telle for dette objektet h. Er en metode private, så kan bare klassen som opprettet metoden bruke det. Om en klasse er private, så kan bare den klassen aksessere metodene og variablene.

Public: Når en instansevariabel/metode er public, så kan hvem som helst gå inn og endre/bruke den. Samme med klasse.

Protected: Protected metoder/variabler/klasser kan man bare komme innpå om man er i klassen som lagde den, eller om andre klasser arver fra den.

Eks: Class parent{kode}, Class barn extends Parents{kode}, Class fremmed{kode}

Koden som er protected i Parent kan bli brukt i barn, men om det blir laget et objekt av Parent i fremmed, så kan ikke metoden blir brukt.

**this og super**

Når man skal implementere metoder i klassen, så kan det være kjekt å bruke this. This vil si at når du har opprettet et objekt, så forteller du metoden at det er nettopp denne variabelen du vil bruke (som er knyttet til objektet). Det vil si at om du oppretter f.eks

Person enperson = new Person(«Tomt»,»Glass»);

Og konstruktøren i klassen sier 🡪

public Person (String ord, String ord2){this.ord = ord; this.ord2 = ord2;}

Så vil Tomt peke på ord og Glass peke på ord2. Dette medfører at om andre metoder bruker this.ord eller this.ord2, så vil objektet hente Tomt eller Glass. Det er en mer oversiktlig måte å skrive kode på.

Super brukes om en klasse arver fra en annen. I kontruktøren til Person over for eksempel, så kan det hende at Person arver fra en annen klasse (som også har opprettet en konstruktør). Istedenfor å skrive this igjen da, så kan man bruke super(variabelnavn);. Dette kan man også benytte for metoder som for eksempel toString. Ønsker du å bruke variablene i den klassen som den arver fra, så kan du gå inn i toString metoden som Person har opprettet og skrive super.toString(). Da vil denne toString metoden bli brukt før det som blir skrevet i Perosn sin. Dette hjelper mot kodeduplisering.

**Refaktorere**

Refraktorering vil si at du vil ha så lite dupliserbar kode som mulig.

Eksempel på dette kan være at du extender fra en klasse som er abstrakt.

Den abstrakte klassen vil da inneholde variabler og metoder som også skal være et annet sted, men istede for å skrive det flere ganger så arver underklassen fra den.

Eks: class abstract Person{

private String etternavn;

private String name;

public Person(String name, String etternavn){

this.name = name;

this.etternavn = etternavn;

}

get/set metoder etc

}

class tyv extends Person{

public tyv(String name, String etternavn){

super(name,etternavn);

}

}

Istede for at vi da skriver inn variablene name og etternavn i tyv, så arver vi de istede. Dette gjør at når vi bruker konstruktøren til tyv, så får vi at vi skal skrive inn name og etternavn.

Alt i alt så sier vi at å refaktorere er å øke kvaliteten på koden så godt som mulig.

**Static (variabel, metode)**

Bruk av variabler eller metoder i en klasse uten at vi lager et objekt.

Variabler: private static int tall = 0; Alle objektene du lager etter vil ha den samme verdien som variabelen er. De får ikke noen egen verdi, selvom de har en egen instanse.

Metode: En statisk metode, er en metode som vil oppdatere seg eller være den samme uavhengig av objektene som er laget for klassen.

Eks:

class calc{

static int count = 0;

calc(){ //konstruktør

count++; //Legger til 1 i count

}

}

calc s = new calc(); // Gir 1

calc f = new calc(); // Gir 2

Uten static ville count begynt på 0 for begge og lagt til 1 i initialiseringen slik at begge hadde vært 1.

**Final (variabel, metode, klasse)**

Variabel: Kan ikke endres etter den er initialisert.

metode: Kan ikke @override fra den

Klasse: Kan ikke arve fra den

**Abstract (klasse, metode)**

Klasse: En klasse som er abstrakt går ikke an å lage et objekt av.

Eks: class abstract Person{}

En klasse kan bare arve fra en Abstract klasse.

Metode: En metode som er abstrakt har ingen return verdi,

og klassen som arver fra en abstrakt klasse må bruke denne metoden.

De må override metoden.

En vanlig klasse kan bare extende fra en abstrakt klasse.

Klassen som lager en abstrakt metode må være abstrakt.

Eks: public abstract int getAlder(LocalDate fodselsdato);

**Interface**

Har ingen implementasjon selv, så klasser kan implementere et interface.

Et interface kan kun inneholde abstrakte metoder.

Eks:

public interface TegnBar {

void tegn();

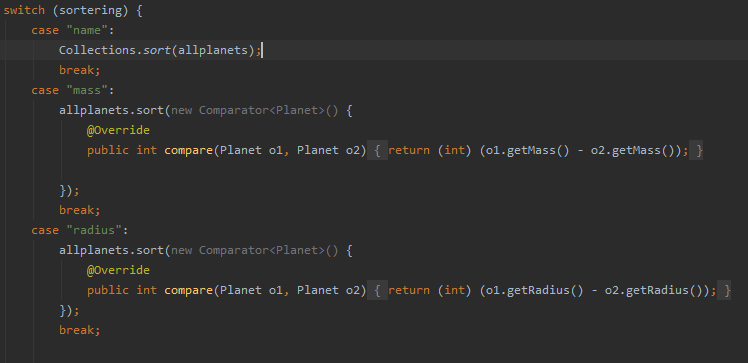
}

En klasse kan implementere mange interface.

Variablene vil alltid være public og final.

**Anonymous Inner Class**

Klasser man definerer i et uttrykk. Disse blir laget om man bruker klassen bare en gang.



Her så overrider jeg compareTo, som ligger i klassen CelestialBody. Denne metoden sammenlikner navn, men jeg har lyst til å sammenlikne masse og radius. Dermed overrider jeg den to ganger. Altså sier at den ene casen skal sammenlikne masse, ,mens den andre skal sammenlikne radius. Selve metoden i seg selv sammenlikner navn(Collection.sort(allplanets)).

**MVC (konseptet, og hver enkelt del)**

En oppdeling av filene vi har for å lage web rammeverket.

Model: Representere data som behandles i programmet

Der alle javafilene ligger 🡪 Alle klassene som produserer data.

Her ligger all dataen som er produsert, og hvordan dataen skal oppdateres etc.

View – grensesnittet, altså det brukeren ser. Presentasjon av data (GUI)

Der Vue.js filene ligger. Representeringen av dataen fra model, altså bare det vi ser. Om modellen skulle endre seg, så må viewet også endre seg.

Controller – kodelogikken i programmet. Kommuniserer mellom Model og View.

Controlleren får inn bruker forespørsler og håndterer de. Den henter så data fra model, og så henter den presentasjonen fra viewet som håndtere den type data. Deretter sender den presentasjonen tilbake til brukeren.

**Exception**

Feil som kan skje når du kjører programmet.

Vi kan benytte oss av try, catch og finally kodeblokker for å sjekke feil som kan oppstå. Kan ha flere catch blokker for å håndtere flere feil.

Try: Kode som vi kjører som det kan skje en feil med

Catch: Håndtering av feilen

Finally: Kode som skal kjøre uansett

Typiske feil er for eksempel å referere til objekter som ikke eksisterer, åpne filer som ikke eksisterer, hente fra en index i et array som ikke finnes.

**Threads**

En prosess som kjører en kodeblokk.

Kan ha mange tråder som kjører parallelt, som medfører mer CPU bruk.

Gjør at ting tar mindre tid.

Enten extends Thread eller implementere Runnable(foretrekkes).

Koden skjer i metoden run(), som må startes med start().

**Collections Framework**

**List**

* List :

ArrayList og Linkedlist

Bruker get(index) for å hente data

Kan definere størrelse på slutten: Arralist<Integer> tall = new Arraylist<>(100);

Spesifisere objekt inne i <>.

LinkedList<Integer> tallLenke = new LinkedList<>();

Bruker add og remove som ArrayList

Elementene vet hvilke elementer som står før og etter den.

getFirst() og getLast()

Om vi legger til 0 på plass 0 i en liste med {1,2,3}, så vil 0 få null som sin venstre og 1 som sin neste referanse. Alle andre elementer oppdateres automatisk sine referanser.

**Map**

* HashMap:

HashMap<integer, String> etmap = new HashMap<>();

Integer = key og String = value.

Key er unike, mens value er ikke.

Key må ikke være en integer, kan være hva som helst.

Samme med value.

Legger inn element med etmap.put(key, value);

Siste keyen som er lagt inn teller, duplisering går ikke

Det vil si 🡪 etmap.put(1, «Hei»); etmap.put(1, «Hello»);

Key verdien 1 vil da ha verdien «Hello».

**Queue**

* Queue:

Queue er delt opp i 3 deler:

Head, tail og Body

Head: Første element

Tail: Siste element

Body: alle andre elementer

Baseres på prinsippet FIFO: First inn, First out.

Legger vi til tallet 5 i denne {1,2,3,4}

{1,2,3,4,5} 🡪 Så kan vi bare hente ut verdi 1 pga 5 havner sist.

Er et interface

Har ulike implementasjoner

* ArrayBlockingQueue

Kan bruke add() og remove()

Offer(object o) når vi legger til

Poll() for å hente første element, fjerner også elementet

Peek() for å se første element

**Stack**

* Stack:

Er en egen klasse hvor lagrinsstrukturen er som en bunke med elementer. Denne bunken følger Last in-First out prinsippet. Det vil si at når man legger til et element så vil den havne sist(kan bli plukket ut først). Kan se det på som en kortstokk, når man legger et kort på toppen, så kan man ta den opp først også.

Legger til 🡪 plassert på toppen, bruker Offer(object o) her også

Tar ut 🡪 Tar det som ligger på toppen, bruker pop()

Ser på 🡪 Ser på det siste elementet, bruker peek()

**Referanser:**

**Tidligere obliger for oppgave 1.5 med noen forandringer noen steder.**

<https://tylermcginnis.com/imperative-vs-declarative-programming> 🡪 imperativ vs declarative

<https://www.youtube.com/watch?v=dAPL7MQGjyM> 🡪 Funksjonell programmering

<https://www.youtube.com/watch?v=5w0NQex4uII> 🡪 private, public(protected)

<https://www.youtube.com/watch?v=DUg2SWWK18I> 🡪 Mvc

Forelesningsnotater