Programmering 2, Oblig 6

Teori

Oppgave 1.1

***Polymorphism*:** Det vil si at alle objektene kan «utgi» seg for å være et objekt av alle klasser den arver fra. Det vil si at man får muligheten til å sammenligne/behandle en gruppe med objekter på likt grunnlag.

Kode eksempel:
I dette tilfellet kan man se for seg at Fugler arver fra Animal, som vil da bli foreldreklassen.

Animal måke = new Fugler(“make”);

Her vil da objektet «måke» være av klassen Animal, men vil utgi seg som et objekt fra Fugler-klassen. Ulempen her er at objektet «måke» vil kun få tilgang til metodene i Animal-klassen og vil da ikke ha tilgang til noe fra Fugler-klassen. Men fordelen er nok at man har muligheten til «sortere» objektene som kommer inn via foreldreklassen. Dette er noe som er relevant i kodeoppgaven, hvor man bruker animal i konstruktøren til observasjon. Man definerer hovedsakelig dyrene i sine egne klasser, men vil bli lagret som Animal via konstruktøren.

Oppgave 1.2

***Primitive datatyper*:** Er hovedsakelig de mest grunnleggende datatypen. Det vil altså si at dataen vil inneholde den direkte verdien av typen den får. Det finnes to typer primitive datatyper, numerisk og boolean. Boolean går ut på sann eller usann, altså true or false. Dette kan i tillegg skrives som 1 og 0. så har man de numeriske typene som består av char, integar og floating point.

Oppgave 1.3

Forskjellen på en klasse og et objekt: En klasse er en blueprint på hvordan koden skal være. I en klasse vil man kunne lage konstruktører med forskjellige formål, men i dette tilfellet kan man tenke at man vil opprette noe. Elementet som blir opprettet vil være et objekt, altså det vil være en instans av klassen. Dette objektet vil da være tilknyttet klassen og har da muligheten til å kalle på og bruke forskjellige metoder som er i klassen.

Kode eksempel:

```
Public class Animal {
String name;
int age;

        Public Animal(String name, int age) {
                this.name = name;
                this.age = age;
 }}
```

I main filen skriver man:

```
Animal lion = new Animal("lion", 30);
```

Her vil *lion* være et objekt av klassen Animal.

Oppgave 1.4

Imperativ programmering: Programmering som baserer seg på å beskrive utsagn som kan endre tilstanden i kode. Her kan altså en boolen variabel avgjøre om hvilken tilstand en maskin er i, noe som er mye brukt ved maskiner som adruino uno. Her vil da hovedfokuset være på å beskrive hvordan koden kjører og utfører funksjoner. Dette brukes som kontrast med deklarativ programmering som i stedet vil kun beskrive utfallet men ikke fremgangen.

Funksjonell programmering: Programmering som hovedsakelig blir utnyttet for matematiske funksjoner, altså til utregning. For å være mest optimalt, vil man unngå tilstandsendringer og data som kan forandres. Her vil man kunne kalle på en funksjon med et argument og få en tilsvarende verdi tilbake. Altså en x-verdi vil gi f(x) ut. Disse funksjonene kan være bygget opp på flere måter som eksponent, første grad og mye mer. Dette kun bygges videre til mye mer og større funksjoner. Java er et eksempel på dette.

Deklarativ programmering: Programmering som er logikk basert og blir mer og mer tatt i bruk i dag. Dette er noe som er grunnleggende for dagen maskinlæring og vil bli mer og mer essensielt. Det kan tolkes som at det vil kunne beskrive resultatet som skal oppnås uten å si noe om fremgangsmåten.

Oppgave 1.5

***Class*:** Klasse er en blueprint på hvordan programmet skal være. Dette vil da kunne bestå av flere felt, metoder og forskjellige typer konstruktører. Ved definering av klasser som *public class Main {}*, vil class bety at innholdet i «{}» vil tilhøre Main-klassen.

***Objekt*:** er en realisering / instanse av en klasse. Objektet kan inneholde metoder som bruker argumentene som er gitt til utføring av oppgaver.

***Instansvariabel*:** Er variablene som blir deklarert i klassene, for hovedsakelig kun lokal bruk. Dette gir da en bedre flyt og forståelse for koden. Disse må da bli definert med public/privat/protected og med hvilken datatype som blir tatt i bruk.

***Overloading*:** Er en funksjon som gir muligheten til å ha flere funksjoner med samme navn i en klasse. Det vil si at man kan lage flere metoder som krever f.eks. forskjellige antall av input argumenter som kreves for å gjennomføre metoden.

***Overriding*:** Vil hovedsakelig si å overskrive noe. Dette gir da muligheten til å overskrive «innebygde» metoder som toString(). Dette gir utvikleren muligheten til å lage metoder med likt navn for de forskjellige klassene med forskjellige funksjoner.

***Extends*:** Kan tolkes som å bygge videre på noe. I vårt tilfelle vil man «extende» en klasse med en annen. Det vil si at man «overfører» metoder osv til klassen som tar i bruk «extend» funksjonen. Dette gir da muligheten til å arve alle public metodene og variablene, noe som kan forenkle koden.

***Privat, public, (protected) (klasse, variabel, metode)*:** Privat, public er hovedsakelig tatt i bruk for å definere tilgjengeligheten til klasse/variabel/metode. Privat gir kun tilgang til variabelen eller metoden i den spesifikke klassen som brukes til å definere den. Public vil gi muligheten til å endre og bruke variabelen/metoden/klassen i et arvende eller videreførende klassen. Protected vil gi noe som ikke kan endres, det vil si at verdien vil være fastsatt.

***This og super*:** This blir brukt til å få tilgang til metoder og variabler i den nåværende klassen og super blir brukt til å få tilgang til metoder og variabler fra foreldreklassen.

Refaktorer: Endre koden for å optimalisere det uten å endre på funksjonen. Det vil si å forminske og bruke exetend, interface, abstract osv. Dette fører da til at programmet kjører raskere og mer effektivt, noe som bruker mindre cpu kraft osv.

Static: En static deklarasjon er for variabler og metoder som skal brukes uten at man må instansiere et objekt før man kan bruke disse. Dette er da ikke assosiert med instansen av klassen. Det vil si at variabelen eller metoden er på en måte «separert» fra klassen og den funksjon.

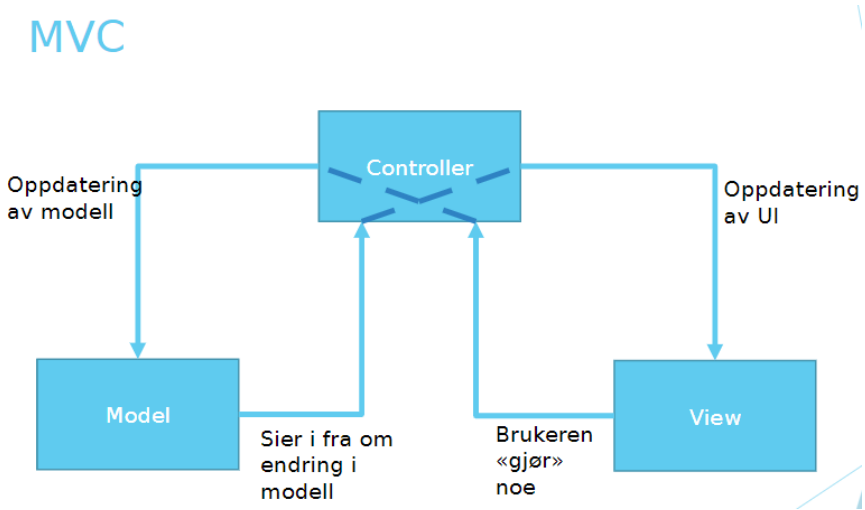
Final: Har flere betydninger ut ifra hva det blir brukt til. Ved bruk til klasser vil man ikke kunne arve fra den. Når man setter final til en metode vil man ikke kunne *override* metoden. I tillegg kan man bruke dette på variabler som fører til at man ikke kan endre variabelen etter at den har blitt initialisert.

Abstract: Abstrakt metode vil si at en metode vil kun være definert ved hjelp av hodet, altså at den ikke har noe kropp. Ved en abstrakt klasse vi man ikke kunne instansiere et objekt. Disse er mye i bruk for å kunne forenkle koden ved hjelp av extend.

Interface: Definerer en kontrakt, hvor klassene kan implementere interfacet som har blitt dannet. I tillegg vil et interface kun inneholde abstrakte metoder. Disse interfacene vil ha en tom kropp. En klasse kan implemente flere forskjellige interfaces, noe som skiller abstrakte klasser.

***Anonym klasse*:** er en nestet klasse som blir dannet for å gjøre koden mer konsis. Hovedfunksjonen er at man kan deklarere og instansiere klassen samtidig. Dette blir for det meste tatt i bruk i forbindelse med å lage implementasjoner av Interface i klassen som tar disse i bruk.

***MVC*:** står for model view controller, noe som er oppbygningen og sammenkoblingen av front og bakenden. Her vil de forskjellige funksjonalitetene bli oppdelt etter sin bruk, noe som lager god struktur og oppbygning.



- ***Model*:** representerer data som behandles i programmet. Klassene som blir dannet vil representere data som blir tatt i bruk. Inneholder hovedsakelig bare informasjon om modellene.
- ***View*:** grensesnittet, altså det brukeren ser. Presenterer dataen (GUI). Har muligheten til å forstå om noe blir utført på nettstedet, men kan ikke utføre noe uten hjelpen fra controlleren.
- ***Controller*:** kodelogikken i programmet. Kommuniserer mellom Model og View, dvs. at den henter ut data fra modellen og sender det til viewet som har bruk for det. Controlleren har et overblikk over alle klassene som er tilgjengelig i koden.

Exception: er hovedsakelig et problem som forekommer under kjøring av koden. Når en exception forekommer vil ikke koden kunne kjøre som normalt noe som fører til feilmeldinger og error. Disse kan man komme frem til ved bruk av try catch.

Tråder: Kan sees på som en «Liten» prosess som inneholder kodesekvenser. Det kan anses på som å utføre et lite arbeid. Når man singleThreader vil man kun utføre «et» arbeid, men det mer normale multithread vil utføre flere ting samtidig. Dette er noe vi bruker hele tiden, som på telefonen og datamaskiner. For å bruke dette i Java må klassen implemente Runnable og ha en override metode for run. I tillegg må man ha en Start()metode for å kunne kjøre Run.

List: Dette er noe som vi er godt kjent med fra ArrayList, noe som er en del av collection frameworket i Java. List kan tolkes som «foreldreklassen» til ArrayList og LinkedList, og har derfor like egenskaper og mye mer.

HashMap: Er en klasse i Java som implementerer Map interfacet og kan holde på 2 elementer som er koblet sammen, ved hjelp av key-value. Forskjellig fra Lister må disse «nøklene» være unike, men verdien kan være like og ulike. Man får da muligheten til å hente ut verdien ved hjelp av de spesifikke nøklene i stedet for å ta i bruk indeksen som i Lister.

Queue: er en lagringsstruktur som gir utvikleren muligheten til å kontrollere i hvilken rekkefølge man vil benytte innholdet i køen. Denne køen deler man opp i *hode, kropp og halen*. Normale «køer» baser seg på First in First Out, men Queue gir oss muligheten til å manipulere dette etter behovene. Dette er relativt viktig grunnet hvor mye buffering har å si innenfor kodeverden. Queue er et interface i Java og har flere metoder, som add, remove og poll for å utføre forskjellige ting.

Stack: Er en egen klasse som baser seg på Last in-First out, noe som vil si at siste inn blir først ut. Man kan tenke seg at man stabler opp en bunke med bøker. For at man skal komme seg gjennom bøkene må man starte med den siste man la på og avslutte med den første som ligger i bunnen.

Referanser

forum, 2020. *Stackoverflow.* [Internett]
Available at: https://stackoverflow.com/
[Funnet 24 04 2020].

Manokaran, J., 2020. *Tidligere obliger tatt i bruk på oppgave 1.5,* s.l.: s.n.

SNL, 2020. *Store Norske Leksikon.* [Internett]
Available at: https://snl.no/programmeringsparadigme
[Funnet 21 04 2020].

wikipedia, 2020. *Wikipedia.* [Internett]
Available at: https://en.wikipedia.org/wiki/Declarative_programming
[Funnet 20 04 2020].

Wikipedia, 2020. *Wikipedia.* [Internett]
Available at: https://no.wikipedia.org/wiki/Funksjonell_programmering
[Funnet 20 04 2020].

Majoriteten av oppgave 1.5 er tatt fra mine egne tidligere programmerings oppgaver.