IT - eksamensspørgsmål (5-12) + 17

**5: Paradigmer: Programmeringsstile**

Fortæl om deklarativ kontra imperativ programmering - har du brugt det til at lave brugergrænseflader? Hvad er fordelen ved det ene kontra det andet?

Hvordan er strukturen i JavaFX bygget op?

**Deklarativ programmering - WHAT**

Her fokuseres der på hvad programmet skal udføre.

Der bruges her en simplere måde at programmere, samt en masse shortcuts, færre variabler.

**Imperativ programmering - HOW**

Dette er nok den mest anvendte form for programmering. Dette er kort at man fortæller computeren hvad den skal gøre.

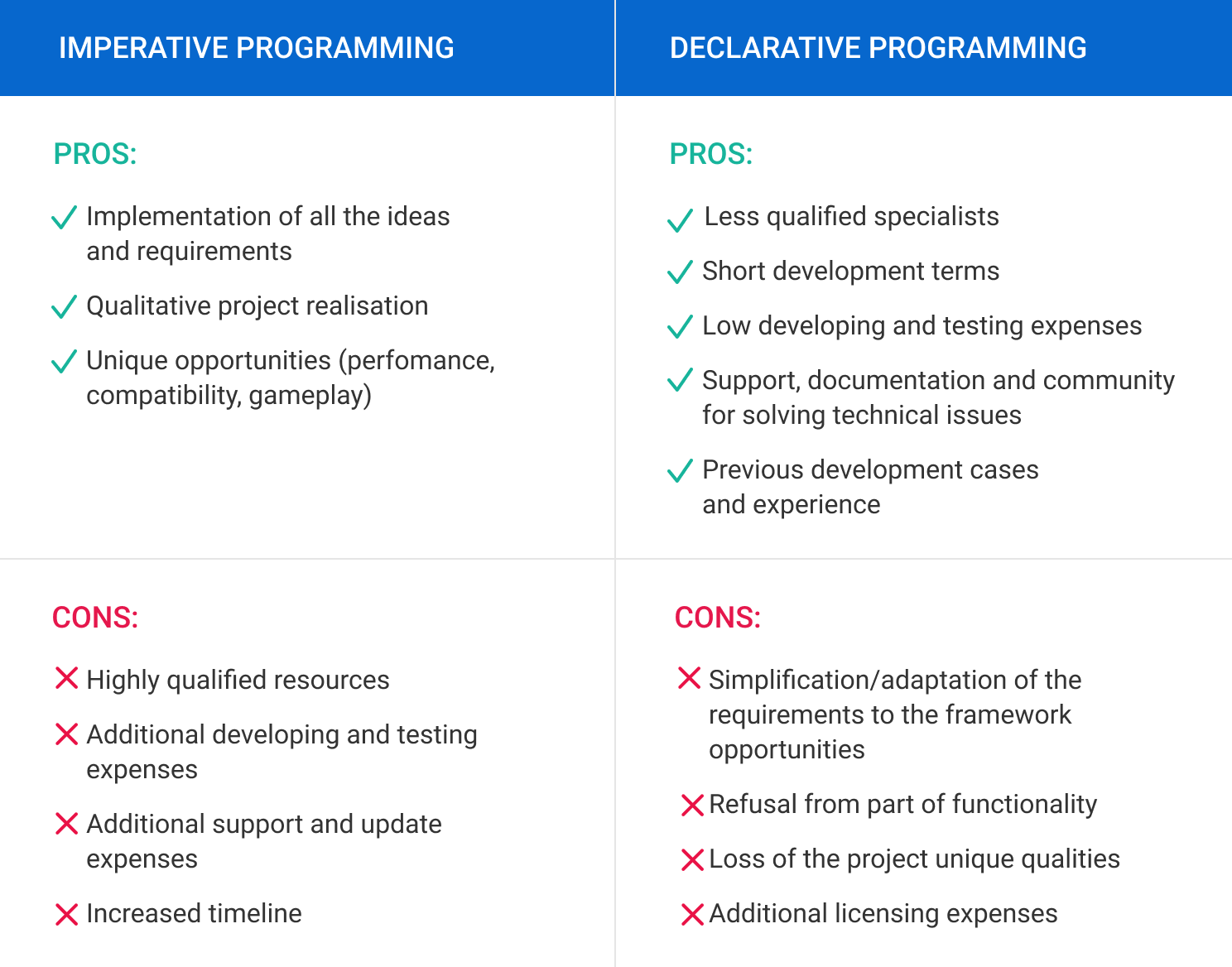
Her fokuseret der på hvordan programmet skal opnå resultatet (step by step).

**Brugt til grænseflade?**

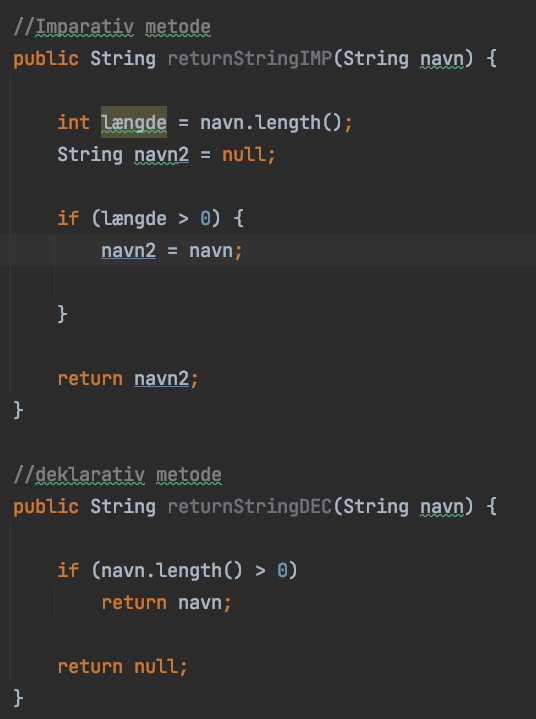
Vi har brugt imperativ programmering hele vejen igennem, da dette er den nemmere metode der bliver brugt af nybegyndere. Deklarativ metode er lidt mere avanceret, og bruges af lidt mere udviklede programmører.

**Fordelene vs hinanden**





**Eksempel:**



I den imperative metode, laver vi en metode hvor der indgår en String med navnet navn. Derefter laver vi en int som vi kalder længde, som er længden på vores variabel navn. Vi laver herefter også en String som vi kalder navn2, som er lig med null.

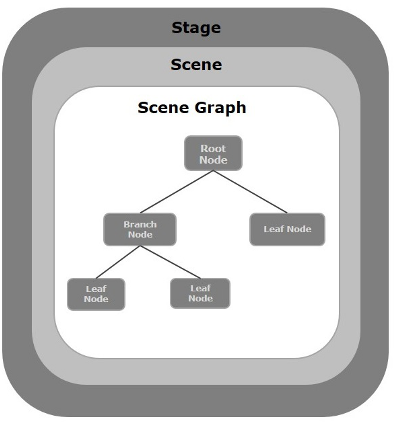
Vi laver herefter en if statement, hvor vi siger at hvis vores variabel længde er større end 0, så skal navn2 være lig med navn, som jo er null.

Til sidst retuneres navn2, altså null.

I deklarativ metode, laver vi ikke nogen variabler. De er unødvendige.

Vi laver direkte et if statement, hvor vi siger at hvis længden af navn er større end 0, så skal navnet returneres.

Og til sidst siger vi return null, istedet for at give null et navn som før.



**Strukturen i JavaFX**

JavaFX er opbygget af 2 dele:

* Stage
* Scene
  + Nodes

Stage

Dette er et vindue, som indeholder alle javaFX objekter.

En stage har 2 mål, nemlig bredde og højde.

For at kunne vise Stage er det vigtigt at bruge en show(), ellers vil den ikke kunne vises.

*Scene*

Scene angiver det fysiske indhold i en applikation, som position(bredde og højde).

Du kan oprette en scene ved at starte scene klassen og her vælge position, sammen med nodes.

En scenegraf er en trelignende datastruktur(hierarkisk), der repræsentere indholdet af en scene. En knude er visuelt/grafisk objekt i en scenegraf og kan inkluderer:

* Geometriske objekter
* UI-kontroller (knapper, tekstområde osv.)
* Layout panes
* Medielementer

**6: Paradigmer: Kodestruktur**

Hvad er Separation of Concerns, Don’t Repeat Yourself? Har du brugt det i dine projekter på studiet? Hvordan holder JavaFX sine elementer adskilt fra hvad der skal foregå, når man fx. trykker på en knap? har du fx. brugt det i nedarvning?

**Hvad er Separation of Concerns, Don’t Repeat Yourself? Har du brugt det i dine projekter på studiet?**

SOC = Adskilning af problemer, er en del af at skrive organiseret kodning, hvor koden forklarer hver del af applikationen step for step på en forudsigelig måde og som er let forstående.

SOC er et designprincip som opdeler ens applikation i flere forskellige funktioner, så der så lidt som mulig overlapning i funktionalitet. Dvs. hver del behandler et særskilt problem, ved den adskilning får man flere muligheder for genbrug og uafhængig udvikling, uden at skulle kende detaljer i andre sektioner eller foretager tilsvarende ændringer i andre sektioner.

Så SOC giver dig muligheden for at opgradere et komplekst system stykkevis uden midlertidig tab af funktionalitet.



DRY

SOC og DRY går hånd i hånd. DRY-princippet sigter mod at reducere gentagelser i softwaren ved at danne abstraktioner, her er nøglepunkter for anvendelse af abstraktion til at skrive kode uden gentagelser:

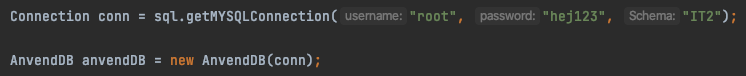
* Oprette funktioner for almindelige software mønstre( højere ordens funktioner )
  + Kort - ændring af hver element i et array baseret på en given regel
  + Filter - at få et subset af det array der passer det stillede krav
  + Reducere - kombinere alt i arrayet basseret på en given regel

Ideen bag princippet er at gentagelser er dårligt for kodning, da at have samme kode forskellige steder gør vedligeholdelses evnen svære, når ændringer i koden er nødt til at ske flere steder i stedet for et sted.

Vi har primært gjort brug af SOC princippet i form af at have flere main klasser, for adskille mellem main for app klassen(javafx), main for DBtester(mySQL) og en main for resten af kodning.

Derudover er DRY princippet blevet brugt løbende i form af ændring af kodning, for at gøre det mere simpelt og ikke gentage kode flere gange.

Vi har bl.a. Et eksempel fra klassen “DBtester”, hvor vi her navngiver en kommando et navn, således at det ikke er hele den lange kommando der hele tiden skal gentages, men dog kun det korte simple navn.



Fra klassen “DBtester”.

**Hvordan holder JavaFX sine elementer adskilt fra hvad der skal foregå, når man fx. trykker på en knap? har du fx. brugt det i nedarvning?**

**7: Paradigmer: Event Driven Programming:**

Hvad forstås ved en Event Listener i Java? Hvordan bruger du det i JavaFX? Kan det kun være events fra brugerne? Kan systemkomponenter bidrage til at lave events?

**Hvad forstås ved en Event Listener i Java?**

Event-driven programming er et programmerings paradigme, hvor flowet i et program er afhængigt af events, altså bl.a. Klik eller tastetryk.

Programmet er her designet til at opdage brugerinteraktion (events) i forskellige former imens de sker, og derefter handle på dem.

Events overvåges af en kode der kaldes for event listener, og de opdager hvis en event er sket. Her vil den sende en reaktion til “event handler”, der kommer med en handling.

**Hvordan bruger du det i JavaFX?**

Vi bruger Event-listener i vores javaFx i form af onAction i de forskellige controllere. Hvor onAction er programmeret til at reagere på input/signal fra “event handler”(som f.eks. Appsetroot i vores programmering)

**Kan det kun være events fra brugerne?**

**Kan systemkomponenter bidrage til at lave events?**

**8: Afprøvning: Usability testing 1**

Beskriv opgavebaseret usability testing og hvad man forventer at få ud af det - Kom med eksempler hvor I har brugt eller kunne bruge det i jeres projekt. Kom gerne ind på resultater, hvilket format man forventer at få ud af det - Sammenhold gerne med Black Box / Funktionel testing. Hvilke stadier kan man lave denne afprøvning på?

**9: Afprøvning: Usability testing 2:**

Beskriv de forskellige typer afprøvning, som I har brugt her på semestret - fx. eksplorativ, performance testing, og stress testing - funktionel, black box testing

**Eksplorativ testing**

Testeren udforsker aktivt testdesignet mens testene bliver kørt, dvs. Der ikke på forhånd er et specifikt problem man sigter efter, men løbende får afprøvet. Det er ligeså vel en afprøvningsform som en læringsproces. Det er muligt at lave eksplorativ testing ud fra stikord om hvad man leder efter, dog handler det mere om at teste “flydende” ved at lære ud fra selv at prøve det af, udforske og lære fra det. Det en afprøvningsform hvor det er testeren der i kontrol over hvad der sigtes efter og hvad der skal ændres, på den måde går man også i dybden efter fejl, dog kræver det også en erfaren tester.

**Performance testing**

Det er en ikke funktion testteknik, der bruges til at teste systemet/programmets hastighed, responstid, stabilitet, pålidelighed og skalerbarhed… under en givet arbejdsbyrde. Hovedformålet er identificere og fjerne fejl/problemer i systemet.

Man ser hvor hurtig systemet reagere, den maksimale bruger belastning som systemet kan håndtere og hvor stabil systemet er under forskellige belastninger. Det er en “præsentationsafprøvning” da man ser på hvordan systemet fungerer i forhold til specifikke vinkler. Der kan være forskellige årsager til en performance testing, så som:

* Kan systemet håndtere 500 brugere på en gang
* Til at sammenligne med andre systemer
* At måle stabilitet under peak
* ...

**Stress testing**

Det udføres for at finde systemets grænsekapacitet og for at bestemme, hvordan systemet fungerer, hvis den aktuelle belastning går langt over det systemet kan tåle/grænse.

**Funktionel testing**

Det er en type software test, der validere software systemet i forhold til de funktionelle krav/specifikationer. Med denne type test er formålet at teste hver funktion i systemet ved at levere passende input og kontrollere output med funktionelle krav.

Det involverer primært black box testing og tager ikke hensyn til kildekoden, i stedet tages brugergrænseflade, API, databaser, sikkerhed, brugeren og andre specifikationer i betragtning.

Hovedformålet er at kontrollere software systemets funktionalitet, dens hovedfunktioner som brugervenlighed, tilgængelighed og fejl betingelser. Funktionel test er centralt element i kvalitetssikring for at forbedre den interne og eksterne tillid i en virksomhed. Man opnår på den måde maksimale kvalitet og sikrer at projektet når sine mål. Det er en test som umiddelbart tages i slutningen af udvikling, når produktet er færdigt. Dog udføres den også gennem hele udviklingsprocessen for at sikre målene og kravene bliver opfyldt.

**Black box testing**

Det er en metode til software test der undersøger funktionaliteten af en applikation uden at kigge ind i dens interne strukturer eller funktioner. Denne test kan foretages på nærmest alle niveauer, derudover behøves ingen specifik viden inden for den interne kode og struktur. Man afprøver programmet/systemet som en helhed. Man basere sig på programmets/systemets funktion, opførsel og tester om der kan udføres det som er påstået.

Black box testing er udelukkende baseret på software krav og specifikationer, derudover er det kendt som behavioral testing, da man ser på programmets opførsel.

**10: Krav og testing:**

Hvilke typer af krav kan man stille til software? Hvordan afdækkes krav? Hvad er kendetegnende ved gode krav? Hvordan validerer man at krav til software er opfyldt? Hvilke typer af tests findes der og hvordan adskiller de sig fra hinanden? Hvilke typer af fejl ser vi i software?

**Hvilke typer af krav kan man stille til software?**

Funktionelle, nonfunktionelle, prioterede krav.

Funktionelle krav

De beskriver systemets opførsel, og er derfor beskriver de specifikke funktionaliteter.

Non-funktionelle krav

Det er kvalitetsmæssige krav til systemet. Fx, svartid, tilgængelighed, sikkerhed osv.

Prioriterede krav kan variere alt efter kundens behov, man kan evt. Gå ud fra MoSCOW prioritering til kravene ( “must have”, “should have”, “could have”, “would have” ).

**Hvordan afdækkes krav?**

Kravene afdækkes løbende og via kunde kontakt. Det kan være omkring hvad kunden skal bruge projektet til, hvilke hardware komponenter, der kræves osv.

**Hvad er kendetegnende ved gode krav?**

Et godt krav er specifikt, detaljeret og har et konkret defineret kriterie for hvad der kræves.

Et dårligt krav kan fx være “Brugervenligt - nem at bruge”.

**Hvordan validerer man at krav til software er opfyldt?**

Ved at finde ligheder og forskelle mellem de funktioner der er og dem som er lavet i ens design afsnit. Et eks. Kan sensoren give en form for data, som kan blive vist på brugergrænsefladen. Giver det mening at tallene står som de gør ?

**Hvilke typer af tests findes der og hvordan adskiller de sig fra hinanden?**

Der findes bl.a. Disse typer :

* Black box / funktionel testing
* Brugerorienteret / explorativ testing
* White box / kodeafprøvning

Funktionel testing er en test af hele programmet som en hel. Her kigger man på Unit testing, som er at se om et komponent gør hvad den skal. Og man bruger Integrations testing, hvor man kigger op hvordan de forskellige komponenter spiller sammen.

Explorativ testing. Testeren udforsker aktivt testdesignet mens testene bliver kørt, dvs. Der ikke på forhånd er et specifikt problem man sigter efter, men løbende får afprøvet. Det er ligeså vel en afprøvningsform som en læringsproces. Det er muligt at lave eksplorativ testing ud fra stikord om hvad man leder efter, dog handler det mere om at teste “flydende” ved at lære ud fra selv at prøve det af, udforske og lære fra det. Det en afprøvningsform hvor det er testeren der i kontrol over hvad der sigtes efter og hvad der skal ændres, på den måde går man også i dybden efter fejl, dog kræver det også en erfaren tester.

White box/Kodeafprøvning, her går man ind og kigger i selve kildekoden, og tjekker om hver linje giver mening og er korrekt.

**Hvilke typer af fejl ser vi i software?**

1. Funktionsfejl

Dette er en fejl der opstår, hvis noget man forventer at systemet skal kunne, pludselig ikke fungerer.

Fx hvis en knap til at åbne et nyt projekt ikke kan trykkes på.

1. Kommunikationsfejl

Det er fejl som opstår i kommunikationen fra software til slutbruger. Alt hvad slutbrugern skal vide for at kunne gøre brug af software skal gørse tilgængelig på skærmn, dvs. Hjælpeinstruktioner, menu, en knap med navnet gem( som gemmer og ikke sletter ). Så at man i bund og grund er klar i sin kommunikation.

1. Manglende kommando fejl

Når en forventet kommando mangler. F.eks. Mangler en tilbage knap, hvor der kun er en frem knap.

1. Syntaktisk fejl

Grammatiske fejl.

1. Fejlhåndteringsfejl

Fejl der opstår mens brugeren interagere med softwaren, skal håndteres på en klar og meningsfuld måde, som at der ikke bare står fejl, men en forklaring på hvilken fejl. Hvis ikke det fremkommer er det en fejlhåndteringsfejl.

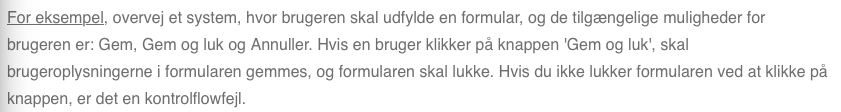
1. Beregningsfejl

Dette er fejl der opstår på grund af følgende:

* Dårlig logik
* Forkerte formler
* Datatypefejl
* Kodningsfejl

1. Control flow errors

* Styringensflowet for en software, beskriver hvad det vil gøre næste gang og i hvilken tilstand. Er der fejl i dette, kaldes det en Control flow error.

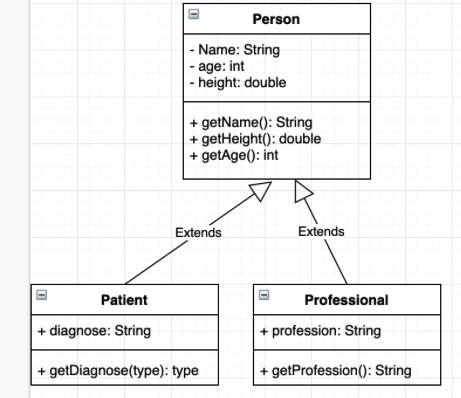


**11: Nedarvning og polymorfi I:**

Forklar om begreberne nedarvning, subklasser og superklasser - fortæl om *protected* keywordet i nedarvning - Og fortæl om differentiering af klasser - Kom evt. ind på hvor du har brugt det i projekter - keywordet *extends*

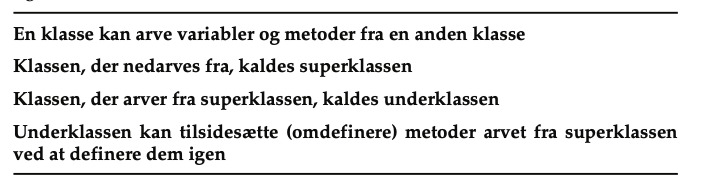
**Nedarvning, subklasser og superklasser:**

En tom pil i et design-klassediagram angiver at noget nedarver.

****

Det bruges typisk til at vise at en klasse A er af typen klasse B.

Nedarvninger bruges, når man ønsker en klasse der ligner den eksisterende klasse, men dog er den ikke præcis det samme.

****

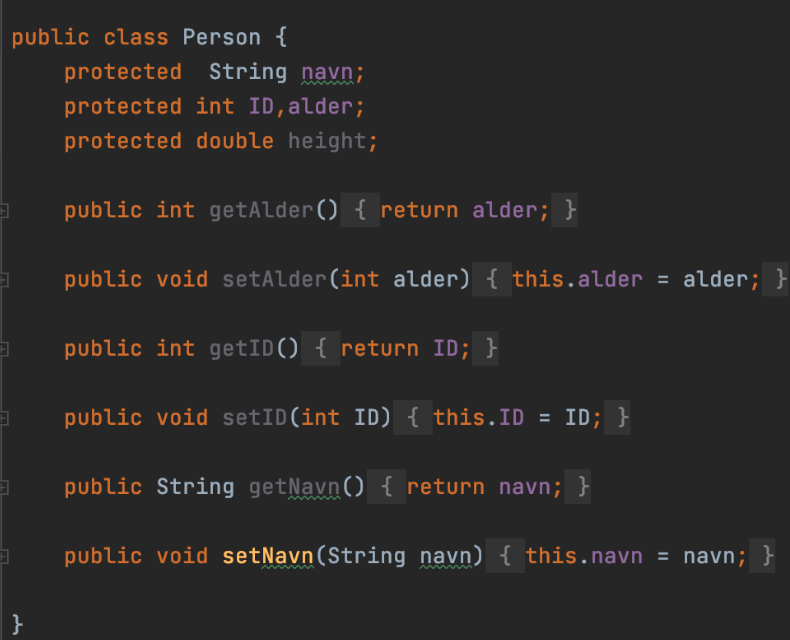
En subklasse, kaldes også en underklasse.

***Protected* keywordet i nedarvning + extends:**

Extend betyder at den arver.

Hvis vi tager et terning eksempel. Så er der mange der snyder med deres terninger og ændre den således at der ingen 1’er side er. Hermed vil man have en nedarvning fra “Terning” til en ny klasse, “FalskTerning”, hvor man starter erklæringen med “public class FalskTerning extends Terning”.

Nøgleordet protected bruges, hvis man vil beskytte en metode (normalt bruges her “private”) men på samme tid stadig ønsker at underklasserne kan kalde metoden.



**Differentiering af klasser:**

**12: Nedarvning og polymorfi II:**

Hvad er en polymorf metode? Hvad markerer @Override i Java? Hvad er formålet med polymorfe metoder? Hvad er visibility? Hvad er indkapsling og Accessors? Hvordan implementerer man det? Hvad er formålet med det?

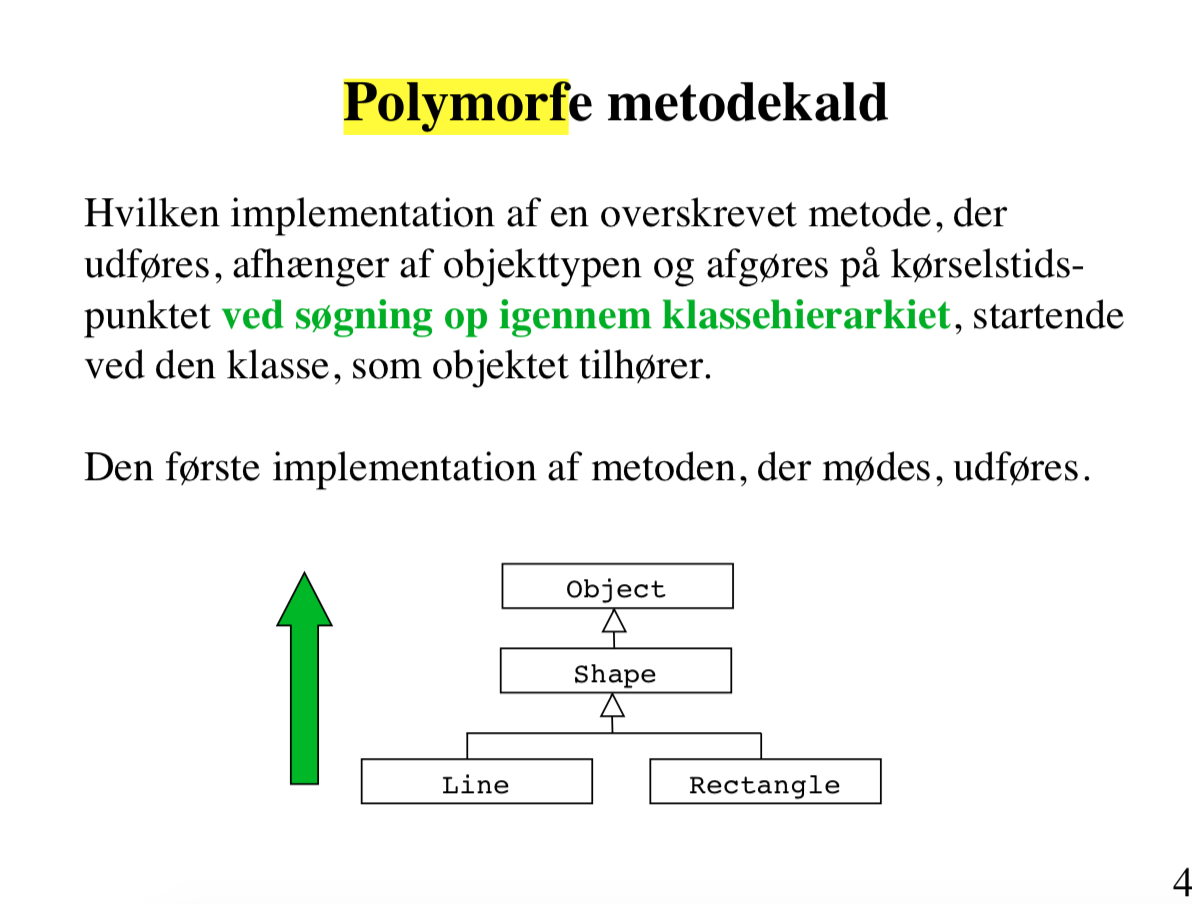
**Hvad er en polymorf metode?**

Polymorfi = mange former

Når der dannes en extend af en superklasse, vil underklassen/subklassen kende til metoderne i superklassen, dog vil den prioriterer metoderne i dens egen klasse først.

Forklares kort og kontant i denne video:

<https://www.youtube.com/watch?v=Ft88V_rDO4I>



**Hvad markerer @Override i Java?**

@Override, gør det muligt for en underklasse at tilbyde en specifik implementering af en metode som allerede er givet fra en af dens super/overordnede- klasser.

Med override omdefinere man en metode, der allerede eksistere i superklassen.

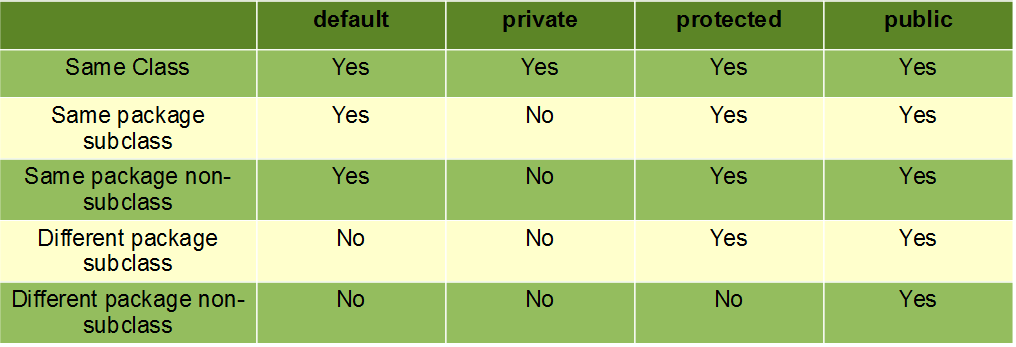
**Hvad er formålet med polymorfe metoder?**

At man har mange former af den samme metoder, i stedet for nye metoder ?

**Hvad er visibility/access modifiers?**

Access modifiers bruges til at indstille hvilken form for adgang der i bland klasser, variabler, metoder og konstruktører.

Skemaet viser hvem der har adgang til de metoderne i de forskellige klasser.

****

En default klasse er en klasse som man ikke angiver, om det er private, protected eller public. Her er den kun synlig for klasser med samme pakker.

For en private klasse, er det kun den samme klasse der har adgang til metoderne i klassen, og dermed ingen andre.

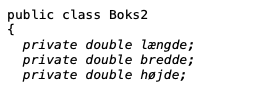
For en protected klasse, kan alle ens subklasser få adgang til metoderne, dog kan en anden klasse der ikke har nogen relation til den ikke få adgang.

For en public klasse, kan alle klasser uanset relation få adgang til metoderne i klassen.

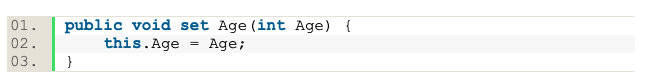
**Hvad er indkapsling og Accessors? Hvordan implementerer man det? Hvad er formålet med det?**

Indkapsling af data betyder at man ikke lader andre klasser bruge objekterne efter eget ønske. Dette kan fx være, at en klasse er public, mens nogle variabler inde i klassen er private for at holde dem skjulte. Man gør visse dele af objekterne utilgængelige uden for klassens metoder. Herved sætter man nogle regler op for, hvordan man kan benytte objekterne.

Dette er vigtigt, da hvis det var muligt for alle klasser at ændre i variablerne, vil der opstå situationer som kommer ud af kontrol ved større komplekse systemer.



En accessor er en getter, hvor en mutator er en setter. En accessor bruges til at returnere værdien af en privat variabel.

****

**17: Databaser spørgsmål 3:**

Hvad er et ER-diagram/ en ER-model? Hvordan interagerer et java-program med en SQL database? Hvilke metoder er der til at interagere med databasen? Hvad er et ResultSet? [Hints](https://www.sqlitetutorial.net/sqlite-java/)