Programmering 2 eksamensnoter

Andreas Troelsen
andtro87@cs.au.dk

Mathias Bak Bertelsen
bufas@cs.au.dk

December 24, 2012

Contents

0.1 Læs dette først!

Dokumentet er ment som en hjælp til at læse op til eksamen i Programmering 2. Det er på ingen måde komplet, og vi giver ingen garanti for korrektheden af indholdet. Hvis du opdager fejl, eller har forslag til forbedringer, er du mere end velkommen til at skrive til en af os. Emails findes på forsiden af dokumentet.

Dokumentet er opbygget med et kapitel for hvert eksamensemne. Først er der noter til emnet og til sidst to eksempler på dispositioner. Nogle af punkterne i dispositionerne er markeret med fed skrift - det er disse punkter, vi anbefaler, at du skriver på tavlen, efter du har trukket dit emne.

0.2 Generelt om mundtlig eksamen

Det er vigtigt at forberede sig godt på emnerne og komme i gang i god tid. Det tager som regel længere tid at forberede sig, end man regner med. En god teknik er at arbejde sammen i grupper af to eller tre personer og fremlægge for hinanden. På denne måde får man feedback med det samme og kan således nå at forbedre sin fremlæggelse inden eksamen. Tag også tid på dine fremlæggelser. Det er vigtigt at tiden passer nogenlunde, så du får sagt alt det, du gerne vil, og at du ikke er færdig efter fem minutter. "Øvelse, øvelse, øvelse... og atter øvelse" - Holger A. Nielsen.

Det kan derudover være en god ide at overvære hinandens eksamener. Aftal med din læsemakker, at I går med ind til hinandens eksamener. På den måde kan I få ekstra feedback på jeres præsentation, og dermed forbedre jer til næste eksamen. Husk på, at skulderklap ikke hjælper ret meget, så man skal kunne tåle at høre, hvad man har gjort galt. Udover at få god feedback finder du også ud af, hvilke spørgsmål censor og eksaminator stiller, og muligvis hvilke emner, der bør undgås.

0.3 Fremlæggelse af patterns

Når du skal fremlægge et pattern er det vigtigt at have styr på tre ting:

- UML-diagrammet for det pågældende pattern skal være helt på plads. Pilene skal tegnes korrekt og der skal være styr på relationerne mellem klasser og interfaces. Det kan være en god ide at tegne dette som noget af det første, da du resten af tiden vil kunne referere til det og kigge på det, hvis du glemmer noget.

- Hvilket problem dette pattern løser og i hvilke situationer det er relevant at benytte. Dette er skrevet på punktform for hvert pattern i Horstmann-bogen, så det burde ikke være så svært at finde ud af.

- Et konkret eksempel på brug. Sørg for at have et godt eksempel som du føler dig tryg ved (brug fx afleveringerne, hvis det specifikke pattern er blevet anvendt i en aflevering). Hvis du ikke selv bringer det op kan du risikere at blive spurgt. Det kan være svært at finde på et godt eksempel midt under eksaminationen.

1 Recursive Methods & Data Structures

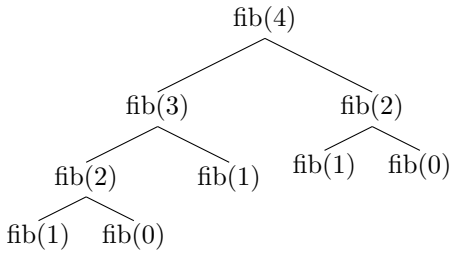
Rekursive metoder er metoder, som kalder sig selv. Rekursion kan i nogle tilfælde gøre kodestykker meget kortere og være meget lettere at implementere end iteration (fx Koch-kurver). Derimod kan rekursion nemt være meget langsommere, og nogle gange fylde meget mere i hukommelsen, hvis man ikke er varsom.

1.1 Fibonacci

- Fibonacci-tallene er et godt eksempel på en noget som kan defineres som en rekursiv funktion. Sekvensens definition siger, at hvert nyt tal i sekvensen er summen af de to foregående tal, og de to første tal er begge 1. Kvadrater med sidelængde af disse tal kan sættes op omkring hinanden i spiraler.
  - Sekvensen ser således ud: 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.
  - $1 + 1 = 2$, $2 + 1 = 3$, $3 + 2 = 5$, $5 + 3 = 8$, etc.
- Rekursivt kan metoden defineres således:

$$F_n = \begin{cases} 1 & n \in \{1, 0\} \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

- Man kan skrive en tilsvarende metode i Java således:
  -
```
public int fib(int n) {
    if (n == 0 || n == 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```
  - Testen for om n er 0 eller 1 kaldes en stopbetingelse. En rekursiv metode bør altid have en stopbetingelse, så man er sikker på, at den ikke fortsætter med at kalde sig selv uendeligt.
- Rekursive kald giver anledning til visualisering vha. træstrukturer. For et kald til fib(4) laves to yderligere kald, fib(3) og fib(2), og hver af disse kald laver yderligere to kald, etc. Følgende træstruktur viser kaldet til fib(4):

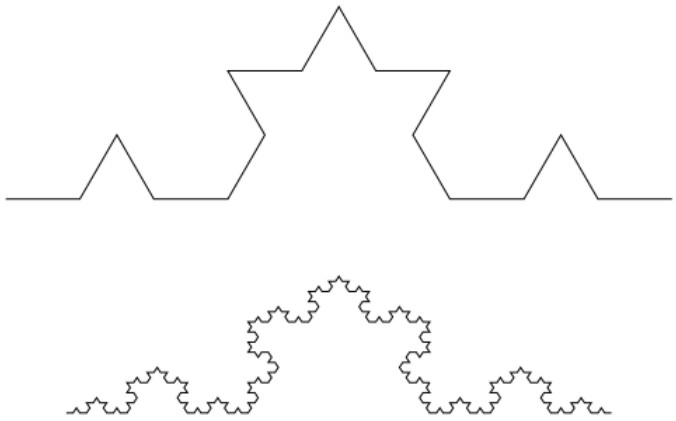


- Bemærk, at fib(2) og fib(0) kaldes to gange, mens fib(1) kaldes tre gange. Dette er et perfekt eksempel på en langsom/ineffektiv implementation, hvor de samme værdier beregnes flere gange.

1.2 Fraktaler

- Fraktaler er rekursiv grafik. Fraktaler er kurver som aldrig er glat nok til at kunne approximeres med linjestykker. Hvis man "zoomer ind" på en fraktal vil den altid "udfolde sig" mere. Velkendte fraktaler inkluderer Mandelbrot-mængden, Sierpinski-trekanter og Koch-linjer.

– En Koch-linje er et linjestykke med et "trekantet hak" i midten, som gør at hvert linjestykke brydes ned i fire linjestykker, som hver brydes ned i fire linjestykker, osv.



– Koch-linjen kan defineres rekursivt vha. et "tegneobjekt", fx et Crayon objekt c, hvis `turn(x)` metode drejer objektet x grader, og `move(y)` tegner en streg af længde y.

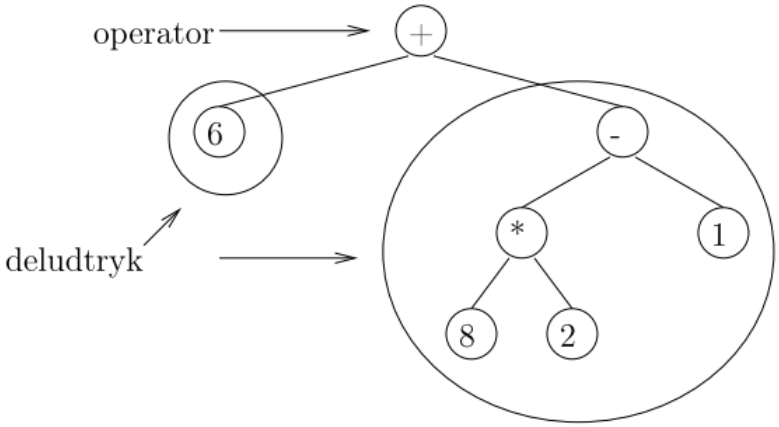
```
private void kochLine(int order, double len) {
    // Stopkriterie
    if (order == 0) c.move(len);

    // Rekursionsskridt
    kochLine(order-1,len/3); c.turn(-60);
    kochLine(order-1,len/3); c.turn(120);
    kochLine(order-1,len/3); c.turn(-60);
    kochLine(order-1,len/3);
}
```

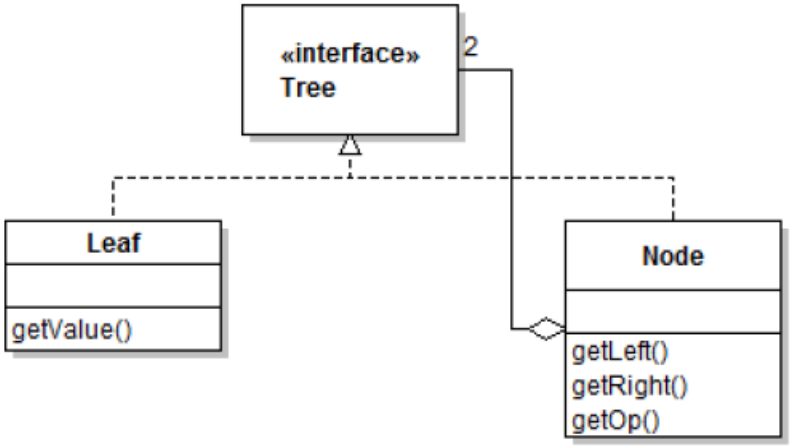
– Hvis stopbetingelsen ikke er opfyldt vil hvert kald til kochLine lave 4 rekursive kald. Det bliver derfor meget hurtigt kompliceret manuelt at gennemløbe sådan en metode, og træstrukturen vil blive meget bred, meget hurtigt.

1.3 Evaluering af udtryk

- Rekursion kan bruges til evaluering af matematiske udtryk. Man kan have et udtryk som $3+4*5$, som ikke beregnes "lineært". Her kan man opdele de matematiske udtryk i forskellige elementer.
  – Operatorer
  – Deludtryk, som kan være enten et tal eller endnu et deludtryk



- På den måde vil en metode, fx `getValue(String exp)` kalde sig selv rekursivt, hver gang den har brudt et udtryk ned i to deludtryk. Når deludtrykkene til sidst udmunder i tal bliver der langsomt returneret delværdier, indtil hele udtrykket er beregnet.



1.4 Dispositionsforslag 1

- **Hvad er rekursion?**
  - Fordele: simplere kode, let at implementere
  - Ulemper: adskillige 'pitfalls'
- **Rekursive metoder**
  - Fibonacci-tal
  - Rekursionstræer
- **Rekursive datastrukturer**
  - Tree, Node, Leaf
  - UML-diagram
- **Visitor Pattern**
  - UML-diagram
  - accept(), visitNode() og visitLeaf()
  - FileSystemNode-aflevering
  - Sammenlign med Composite Pattern

1.5 Dispositionsforslag 2

- **Rekursion vs iteration**
  - Stopkriterie og rekursionstilfælde (skriv matematisk notation og kode for fibonacci eller fakultet)
  - Mange problemer er rekursive af natur og derfor simplere at løse på denne måde. Kodens læsbarhed kan også øges markant.
  - Eks. nyttigt til fraktaler (husk dIntProg)
  - Kan være (meget!) ineffektivt i forhold til iteration (kan føre til stackoverflow)
  - Tegn kald-træ for fibonacci
  - Tradeoff mellem simplicitet og effektivitet
- **Visitor pattern**
  - Referer til afleveringen
  - Tegn UML og forklar hvordan et accept()-kald fungerer
  - **Composite pattern**

2 Class Design & Invariants

En klasse er en samling af fields og metoder. Ofte bruges klasser som skabeloner, som objekter kan instantieres fra. Nogle klasser kan således betragtes som "blueprints", som man kan skabe objekter ud fra. Et velgennemtænkt og godt struktureret klassedesign er meget vigtigt for brugbarhed af klasser.

2.1 Klassestruktur

- Klasser indeholder typisk nogle fields, en eller flere constructors og en række metoder. Klassens fields udgør dens tilstand, og dens metoder udgør dens interface (klasse-interface).
- Constructors sørger for at objekter instantieres fra klasser med nogle bestemte værdier for deres fields, dvs. en starttilstand.
- Metoder er "funktioner" som kan kaldes på klassen selv (static), eller på individuelle objekter instantieret fra en klasse
- Utility-klasse: En klasse med en private constructor (kan ikke instantieres), og udelukkende static metoder (fx Math, Collections).

2.2 Indkapsling

- Visibility modifiers: public/private/protected
  - Constructors er generelt public.
  - Metoder kan være begge dele. Ofte bruges private metoder som hjælpemetoder til public metoder. (Én gang public, altid public - gælder også for protected)
  - Fields bør altid være private, og objekters/klassers tilstande bør altid tilgås/manipuleres gennem public metoder.
- static
  - Fields og metoder tagged som static er tilgængelige på klassen selv, og alle objekter instantieret fra klassen. Static fields deles mellem alle objekter af klassen.
  - Static metoder (kaldes på klassen) kan kun tilgå static fields.
  - Non-static metoder (kaldes på objekter) kan tilgå både static og non-static fields.
- Et objekts tilstand (dvs. fields) aflæses gennem "accessorer" (fx `get()` metoder), og modificeres gennem "mutatorer" (fx `set()` metoder).
  - Objekters tilstande må aldrig modificeres af accessorer (da de så vil være mutatorer), men de må gerne aflæses af mutatorer, så længe der samtidig er en tilsvarende accessor-metode, som aflæser tilstandene uden at ændre på dem.
  - Scanner er et godt eksempel på dårlig stil: Ingen accessor (`getCurrent()`), så hvis man skal bruge det "nuværende" element skal man efter kaldet til `next()` gemme værdien/referencen i en lokal variabel, da `next()` flytter cursoren ét element frem inden den returnerer et element.
- Et objekt er mutérbart, hvis dets fields kan manipuleres med enten via direkte adgang (public) eller gennem mutator-metoder. Omvendt kaldes det immutérbart, hvis det kun består af accessor-metoder (fx String), og således ikke kan ændres på, efter det er blevet skabt.
  - Kun immutérbare objekter bør deles mellem andre objekter.

  - Mutérbare objekters indre tilstand kan ændres, hvis deres fields af reference-typer returneres direkte af `get()` metoder (fx ArrayList, HashMap), da sådanne return-values returnerer objekt-referencen, fremfor blot at returnere en simpel værdi (som med primitive typer).
  - Hvis mutérbare objekter skal returneres i `get()` metoder bør objekterne klones først vha. `clone()`.
  - På samme måde bør constructors klone eventuelle mutérbare objekter, som de får som parametre.

2.3 Cloning

- Deep/shallow clone.

2.4 Quality of class interfaces

- Cohesion: En klasse er "cohesive" eller sammenhængende, hvis dens metoder alle har at gøre med det samme overordnede emne. Fx en Mailbox-klasse som kun har metoder, som har at gøre med breve/beskeder.
- Completeness: En klasse er "complete" eller fuldstændig, hvis den har metoder til ethvert tænkeligt formål indenfor dens scope. Fx Math-klassen, som har en stor mængde metoder til matematiske beregninger (dog ingen metoder til beregning af egentlige brøker, dvs. ikke helt complete).
- Convenience: En klasse er "convenient" eller bekvem, hvis simple operationer og opgaver er nemme at løse. Et klasseinterface skal være nemt at bruge, så brugere ikke skal kalde 10 forskellige metoder for at få et simpelt resultat.
- Clarity: En klasse er "clear" eller letforståelig, når der ikke kan opstå meget forvirring omkring dens interface. ListIterator's `next()` og `add()` metoder er meget analog til hvordan et tekstbehandlingsprogram virker, men dens `remove()` metode er meget klodset. Dårlig clarity.
- Consistency: En klasse er "consistent" hvis dens metoder ikke afviger i fx parameter-syntax. Fx tager Java's GregorianCalendar klasse et månednummer mellem 0 og 11, men et dagnummer mellem 1 og 31, i dens constructor, hvilket er meget fjollet.

2.5 Programming by contract

- En måde at undgå tunge, performance-sænkende gyldighedschecks. Ved at opstille "kontrakter" for hvordan metoder må/skal benyttes kan man fralægge sig ethvert ansvar for effekterne af at kalde metoderne med ugyldige parametre (dvs. hvis preconditions ikke er opfyldt). Samtidig kan man garantere at metoden gør hvad den skal (postconditions), hvis preconditions er opfyldte.
- Preconditions
  - Skal være opfyldt for at metoden kan garantere en korrekt/gyldig kørsel.
  - For at gøre preconditions "fair" skal kalderen af metoden kunne checke om preconditionen er opfyldt eller ej, før den laver metodekaldet.
  - Man bør ikke kaste exceptions som resultat af uopfyldte preconditions; det er netop pointen med preconditions: "hvis ikke du opfylder disse krav, så garanterer jeg intet og du må sejle i din egen sø."
- Postconditions

  - Skal være opfyldt når metoden er fuldført.

- Invarianter
  - En måde at checke pre- og postconditions er ved at lave assertion-metoder, som checker et objekts fields' gyldighed.
  - Assertion-metoderne kaldes efter constructor og efter metoder, og bruges til runtime debugging.

2.6 Dispositionsforslag 1

- **Klasser og objekter**
  - Hvordan er en klasse opbygget?
  - Tilstand (fields), klasseinterface (metoder)
- **Indkapsling**
  - Visibility modifiers (once public, always public)
  - Accessors vs. mutators
  - Mutable vs. immutable
- **Programming by contract**
  - Invarianter generelt
  - Preconditions, postconditions
  - JavaDocs, exceptions/returværdier
- **Cloning**
  - Deep vs. shallow
  - Java API'en, ArrayList
  - Pointer-diagram

2.7 Dispositionsforslag 2

- **Indkapsling**
  - Regler
  - Mutable
  - Immutable (klasser uden mutators eks. String)
    - Husk at klone før returnering
  - **Visibility modifiers**
    - En gang public, altid public[1]
    - Ingen public feltvariable! Dette vil resultere i klassen aldrig kan ændre sin interne repræsentation. Benyt altid getters og setters.
- **Kloning**
  - Deep / Shallow (tegn og forklar på tavlen)
- **Programming by contract**
  - Invarianter - Sand efter constructoren og sand efter enhver metode
  - Preconditions hos subklasser max lige så stærke som hos superklasse
  - Postcondions hos subklasser minimum lige så stærke som hos superklasse
    - Pre- og post skal nemlig overholde Liskov's substitution
  - Throws definerer hvilke exceptions metoden kan smide (lav kodeeksempel)
  - Assertions (kan bruges til at tjekke pre- og postconditions)

[1] Når en metode har været public kan den aldrig laves protected eller private. Hvis andre afhænger af denne metode kan ændringen ødelægge deres programmer.

3 Polymorphism & Interfaces

Polymorfi betyder "mangeformet", med hvilket der menes evnen for en variabel af type B at udgive sig for at være af type A, hvis A er en supertype til B. Fx kan klasserne Dog og Cat begge udgive sig for at være Animal objekter.

3.1 Interfaces

- Et interface er en samling metodesignaturer. Dvs. metoder, som ingen krop har. Metoderne er ikke implementerede.
- En klasse kan implementere et interface, hvilket betyder, at alle interfacets metoder skal implementeres i klassen. Dette betyder, at hvis en klasse implementerer et interface, kan compileren være sikker på, at interfacets metoder eksisterer i klassen.
  - Java-keyword: `implements`
  - `public class Dog implements Animal { ... }`

3.2 Polymorfi

- Hvis en klasse implementerer et interface er klassen en subtype til interfacet. Oprettes en variabel a af type Animal kan den indeholde et objekt af en klasse som implementerer Animal interfacet, fx Dog. Nu har variablen a en formel type (Animal) og en aktuel type (Dog).
- Liskov's substitutionsprincip siger, at et object af en subclass kan benyttes nårsomhelst et objekt af dens superclass forventes. Eksempelvis kan et Manager objekt altid lægges i en Employee variabel, men ikke den anden vej rundt. Da Manager objekter altid er garanteret at have samme metoder og variable (og lidt til) som Employee objekter, er dette netop muligt.
  - På samme måde kan Dog objekter altid lægges i Animal variable.

3.3 Nedarvning

- "is-a" forhold: <subclass> "is-a" <superclass>
  - Java-keyword: `extends`
  - `public class Manager extends Employee { ... }`
- Subclasses arver alle metoder og variable fra deres superclass
- Subclasses kan override deres superclass' metoder, dog ikke hvis de er erklæret `final`
  - Polymorfi bestemmer hvilken metode der kaldes vha. objektets aktuelle type. En Employee variabel kan fx indeholde et Manager objekt (Liskov's substitutionsprincip), og polymorfien sørger for at kalde Manager-objektets overridden metode - hvis en sådan eksisterer.
  - Override-metoder skal have samme metodesignatur (samme return type, samme parametre), som superclassen's metode.
  - Selvom en metode er overridden kan subclasses godt kalde deres superclass' version af metoden med keywordet `super`.
  - Preconditions til overridden metoder må ikke være mere strikse end superclassens preconditions. Hvis ingen preconditions findes i superclassen må subclassens overridden metode heller ikke have preconditions.
  - Postconditions skal omvendt være mindst ligeså strikse.

  - En overridden metode må ikke kaste yderligere check exceptions.

- Subclasses kan kun tilgå private variable gennem `get()`-metoder, dog kan de tilgå protected variable (en slags "intern public")

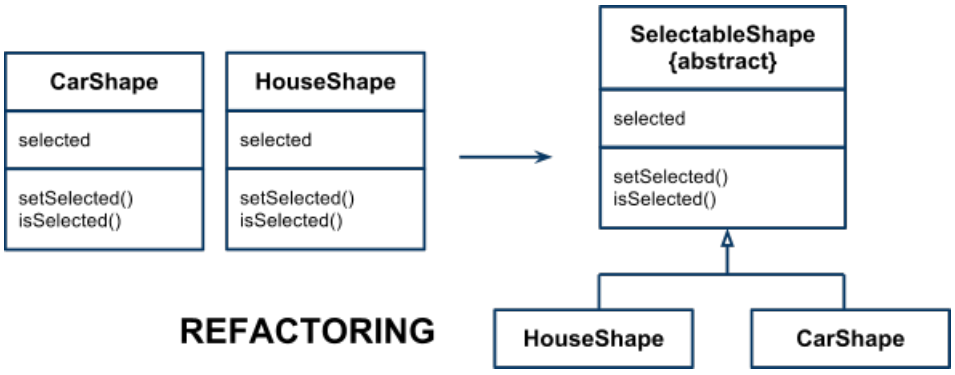
- Nedarvning af classes betyder, at metoder og funktionalitet er tilgængelige med det samme, og principielt behøver man ikke skrive noget yderligere kode for at bruge sin nye subclass. Implementering (`implements`) af interfaces betyder derimod, at man selv skal supplere koden for hver af de metoder, som interfacet definerer.

3.4 Abstract classes

- Abstract classes er ikke instansiérbare - dvs. man kan ikke lave objekter af abstract classes. En class tagges som abstract med keywordet "abstract".
  Fx: `public abstract class SelectableShape implements SceneShape`

  - Eksempel: SelectableShape. Abstract class der implementerer SceneShape interfacet, og som definerer nogle af interfacets metoder, men ikke alle. De metoder som ikke er definerede skal derfor defineres i subclasses (fx CarShape, HouseShape).
  - Abstract classes implementerer typisk noget af funktionaliteten i de interfaces de evt. implementerer, men ikke alt. Resten af metoderne er abstract.
  - Fordele: Man kan lægge al ensartet funktionalitet for flere classes i én superclass, som implementerer et interface, som er nyttigt for disse classes.
  - Ulemper: Classes kan kun extende én superclass, men implementere adskillige interfaces.

- Refactoring - at omskrive kode så det er "bedre" eller lettere at forstå.
  Fx: "Extract superclass" - Lav en superclass ud fra de fælles features to eller flere classes har.

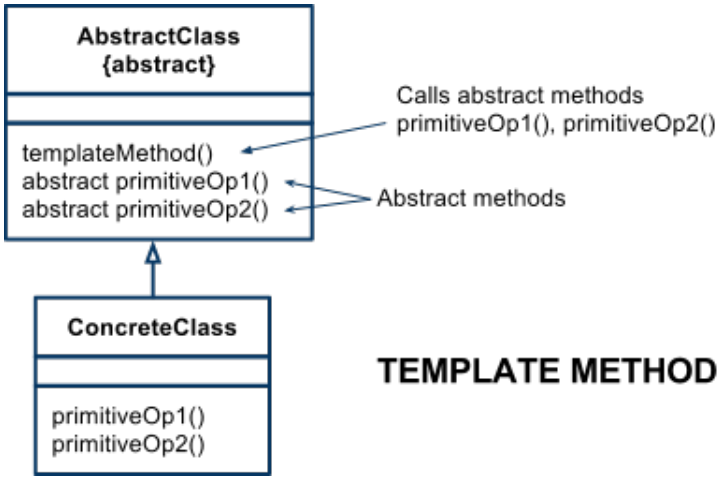


  - Refactoring adskiller sig fra design patterns idet refactoring handler om at omskrive allerede eksisterende kode, mens design patterns handler om at planlægge sit arbejde så man kan undgå at skulle benytte sig af refactoring.

- Template method - en metode som flyttes fra subclasses til en superclass, og som kalder primitive operations, som kan være forskellige, i subclasses.

  - Design pattern
  - En `templateMethod()` er en metode i en abstract class, som kalder abstract methods (dvs metoder som bliver defineret af subclasses som extender denne abstract class). Altså er en `templateMethod()` en metode, som kalder andre metoder, som ikke endnu er blevet defineret.

– Eksempel: `drawSelection()` i SelectableShape - `drawSelection()` er defineret i SelectableShape, men kalder metoder `draw()` og `translate()`, som er abstract methods i SelectableShape, men som bliver defineret i CarShape og HouseShape.



3.5 Dispositionsforslag 1

- **Interfaces**
  - Hvad er et interface?
  - Ingen instantiering, ingen implementation
  - Eksempel: `public class Hest implements Animal ...`
- **Polymorfi og nedarvning**
  - Hvad er polymorfi?
  - Formel/aktuel type
  - Lisskovs substitutionsprincip
  - Eksempel: `public class Manager extends Employee ...`
- **Abstrakte klasser**
  - En slags class-interface 'hybrid'
  - Ingen instantiering, delvis implementation
  - Eksempel: `public abstract class Carnivore implements Animal ...`
- **Template Method Pattern**
  - UML-diagram
  - Eksempel: Animal-Carnivore-Cat fra review-opgaverne
  - Eksempel: Applets og metoderne `init()`, `start()`, etc.

3.6 Dispositionsforslag 2

- **Interfaces**
  - Programmør behøver ikke kende egentlige klasser (jvf. Frameworks)
  - Kan implementere mange interfaces modsat nedarvning
  - Visibility modifiers (public, protected, private)
    - private - Bruges som regel til hjælpemetoder
    - public - Bruges til metoder der skal kunne kaldes af andre
    - protected - Kan kun kaldes af subklasser og klassen selv
  - Kan ikke instantieres (kan kun bruges som statisk type)
- **Nedarvning**
  - "arve" metoder fra superklassen
  - super keyword
  - Kan kun nedarve fra én klasse
  - Regler ved nedarvning
    - "is-a relationen" (<subtype>is a <supertype>)
    - Liskov substitution[2] (en subklasse kan altid indsættes i stedet for en superklasse)
    - Eksempel der overholder Liskov substitution
      - Både is-a og Liskov glæder (Crab extends Animal (fra dIntProg) eller JonglerendeKlovn extends Klovn)
      - Kun is-a gælder (Square extends Rectangle[3])
    - Preconditions (maks lige så stærke)
    - Postconditions (mindst lige så stærke)
  - Centralisering af kode ved at flytte det op i supertype (referer til aflevering om BankAccount)
- **Polymorfi**
  - Flere metoder med samme navn
    - typesystemet sørger for korrekt kald
    - dynamisk/statisk type
    - Hvis man skal modellere cirkusartister har man "Linedanser" og "KanonKonge". Programmøren ved ikke hvilken rækkefølge artisterne skal på scenen så et interface "Artist" benyttes. Interfacet har metoderne `udfør_nummer()`og `buk()`.Begge klasserne implementerer interfacet og programmøren kan nu håndtere forskellige artister som var de ens. Typesystemet i Java sørger for at metoden bliver kaldt på den rigtige klasse. Altså `artist.udfør_nummer()`er forskellig alt efter om artist er af typen Linedanser eller KanonKonge. Altså metoden der bliver udført afhænger af den dynamiske type.
  - Det giver bedre muligheder for udvidelse
    - Hvis der skal bruges en ny type artist kan denne blot tilføjes uden videre. Typesystemet sørger for at den korrekte metodeimplementation bliver kaldt.
- **TEMPLATE METHOD pattern**

[2] Husk at Barbara Liskov er en kvinde
[3] Da square er mere restriktiv end rectangle

- Eks. Gå over vejen (menneske og hund)
  - primitive metoder (kig() og gå())
  - Da hunden har 4 ben går den anderledes end mennesket
  - Den kigger sig nok heller ikke så godt for

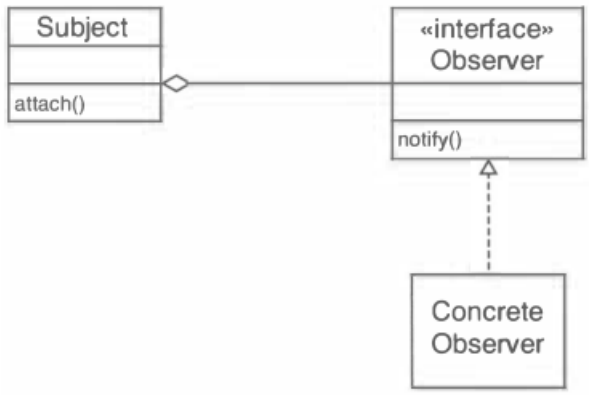
4 Design Patterns

Design patterns er guidelines for hvordan man kan designe programmer og klasser. De kan betragtes som generelle løsninger til generelle problemer. Konceptet begrænser sig ikke til datalogi, og blev faktisk opfundet af en arkitekt. Design patterns kan sammenlignes med algoritmer som også er software mønstre, men algoritmer løser beregningsproblemer, mens design patterns løser designproblemer.

4.1 Observer Pattern

- Hvis et objekt (Subject) starter events (som fx "mine data har ændret sig"), som andre objekter (Observers) er interesserede i at vide noget om, kan man bruge OBSERVER mønstret. Stort set alle GUI elementer (fx knapper, checkboxes, slidere) er "Subjects" i implementationer som følger OBSERVER mønstret.
- Subject har en samling observer objekter, som bliver tilknyttet med en attach() metode. Når en event sker, meddeler subjectet det til alle observerne.
- Eksempel: JButtons (Subject) har ActionListeners (Observers). Ofte bruges anonyme klasser i stedet for at lave ConreteObserver klasser:

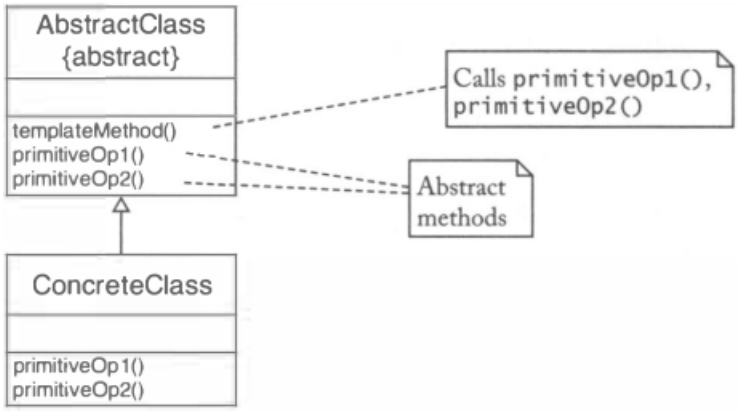
```
myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do something
    }
});
```



4.2 Template Method Pattern

- Hvis en algoritme eller metode er brugbar i flere klasser, og kan brydes ned i "primitive operationer" (som godt kan være defineret forskelligt i de forskellige klasser) kan man bruge TEMPLATE METHOD mønstret.
- Metoden eller algoritmen, `templateMethod()`, flyttes ind i en abstract superclass, sammen med abstract metoder for hver af de "primitive operationer", som `templateMethod()` kalder. Disse primitive operationer defineres IKKE i superclassen, men i subclasses (som extender den abstrakte superclass).
- Altså er en `templateMethod()` en metode som kalder andre metoder som ikke endnu er blevet defineret.

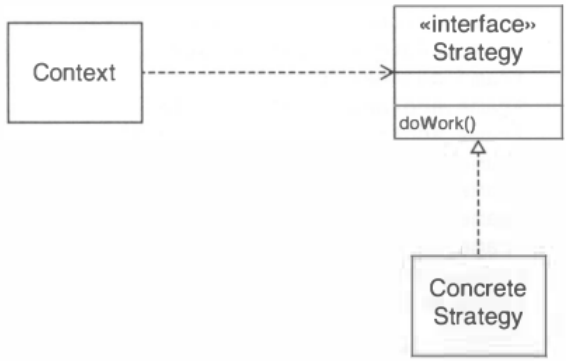
- Eksempel: `drawSelection()` i SelectableShape - `drawSelection()` er defineret i SelectableShape, men kalder metoder `draw()` og `translate()`, som er abstract methods i SelectableShape, men som bliver defineret i CarShape og HouseShape.



4.3 Strategy Pattern

- Hvis en klasse kan udnytte forskellige variationer af en algoritme, kan man bruge STRATEGY mønstret. LayoutManagers i Java er designet omkring STRATEGY mønstret. Containers (Context, fx JPanel, JFrame) kan udnytte forskellige layout managers (ConcreteStrategy, fx BorderLayout, GridLayout) til at arrangere deres indhold.

- De forskellige variationer af algoritmen defineres i klasser som implementerer et Strategy-interface. Context-klassen kalder så den rigtige metode på det konkrete Strategy-objekt. Dvs. enhver klasse som implementerer Strategy-interfacet kan benyttes af Context-klassen, men Context-klassen er også afhængig af (dependency) Strategy-interfacet.

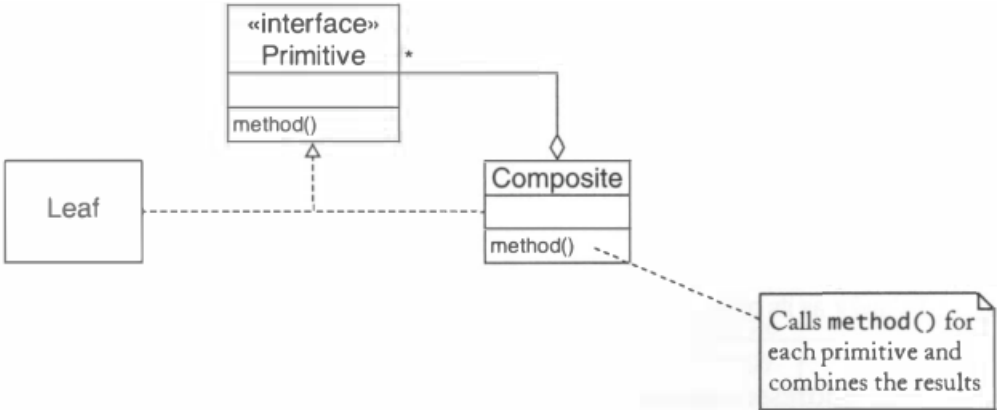


4.4 Composite Pattern

- Hvis man har behov for at kombinere en række primitive objekter i en "bundle", som bliver opfattet af klienter som værende et enkelt primitivt objekt selv, kan man benytte COMPOSITE mønstret.

- Man lader en klasse (Composite) implementere det samme interface som det, de primitive objekter implementerer. Composite-klassen betragtes således af klienter på samme måde som de primitive objekter. Desuden aggregerer Composite-klassen også interfacet, hvilket betyder at den har en samling (fx ArrayList) af primitive objekter.

- Metoderne fra interfacet defineres i Composite-klassen ved at kalde metoderne på hvert af de primitive objekter, som er indeholdt i klassen. De skal selvfølgelig kaldes på en bestemt måde - fx skal en Bundle-klasse som indeholder Item-objekter udregne alle Item-objekternes samlede pris, når `getPrice()` kaldes på Bundle.



4.5 Dispositionsforslag 1

- **Design Patterns**
  - Hvad er design patterns?
  - Universel kommunikation
  - Komplet pattern: navn, kontekst, løsning og UML-diagram
- **Observer Pattern**
  - UML-diagram
  - Subject "has-an" Observer (implementerer interfacet)
  - Event på Subject → kald på Observer
  - Eksempel: JButton og ActionListeners
- **Visitor Pattern**
  - UML-diagram
  - accept(), visitNode() og visitLeaf()
  - FileSystemNode-aflevering
  - Sammenlign med Composite Pattern
- **Template Method Pattern**
  - UML-diagram
  - Eksempel: Animal-Carnivore-Cat fra review-opgaverne
  - Eksempel: Applets og metoderne `init()`, `start()`, etc.

4.6 Dispositionsforslag 2

Det vigtige i dette emne er at du vælger så få at du har tid til at gå i dybden med dem alle, men ikke det er også vigtigt at du kan fylde tiden ud. Hellere have forberedt et pattern for meget end et for lidt.
For hvert pattern skal der være styr på UML diagrammet og hvilket problem det løser. Derudover ville det også være fint at kunne give et eksempel på brug.

- **Composite**
  - mønstret realiserer at en samling af objekter kan håndteres som et enkelt objekt. (Eks. en dagligvare og en indkøbspose med varer)
- **Adapter**
  - et objekt skal bruges af en klasse uden at der ændres på objektet. Objektet implementerer ikke det rigtige interface og skal adaptes til det rigtige interface.
  - En adapter-klasse oprettes som implementerer det rigtige interface og bruger objektets metoder til at realisere de nye metoder
- **Template Method**
  - Relater til frameworks og nævn prototype pattern
  - Hvad: en klasse der kalder primitive metoder på subklasser af et interface og kombinerer disse til en kompleks metode.
  - Hvornår: en algoritme bruges i flere klasser. Algoritmen kan brydes ned til mindre dele. Delalgoritmerne defineres i klasserne. En superklasse kalder de primitive algoritmer i en rækkefølge så det danner den komplekse algoritme. Algoritmen er nu samlet et sted.
- **Proxy**
  - Hvad: en "stand-in" klasse for det oprindelige subjekt. Har samme metoder, dog med nogle modifikationer der gør den fordelagtig.
  - Hvordan: en proxyklasse implementerer samme interface som det oprindelige subjekt. Proxyen kalder metoderne på subjektet.

5 Inheritance & Abstract Classes

Nedarvning handler om at lave specialiserede udgaver af allerede eksisterende og mere generelle klasser. Eksempelvis Employee-Manager forholdet diskuteret i Horstmann kap. 6; alle Managers er Employees, men ikke alle Employees er Managers. Managers er specielle typer af Employees, som har ekstra "features".

5.1 Superclass (generel) og subclass (specialiseret)

- "is-a" forhold: <subclass> "is-a" <superclass>
  - Java-keyword: `extends`
  - `public class Manager extends Employee { ... }`
  - En class som er erklæret `final` kan ikke extendes
  - Alle classes extender Object
- Subclasses arver alle metoder og variable fra deres superclass
- Subclasses kan override deres superclass' metoder, dog ikke hvis de er erklæret `final`
  - Polymorfi bestemmer hvilken metode der kaldes vha. objektets aktuelle type. En Employee variabel kan fx indeholde et Manager objekt (Liskov's substitutionsprincip[4]), og polymorfien sørger for at kalde Manager-objektets overridden metode - hvis en sådan eksisterer.
  - Override-metoder skal have samme metodesignatur (samme return type, samme parametre), som superclassen's metode.
  - Selvom en metode er overridden kan subclasses godt kalde deres superclass' version af metoden med keywordet `super`.
  - Preconditions til overridden metoder må ikke være mere strikse end superclassens preconditions. Hvis ingen preconditions findes i superclassen må subclassens overridden metode heller ikke have preconditions.
  - Postconditions skal omvendt være mindst ligeså strikse.
  - En overridden metode må ikke kaste yderligere check exceptions.
- Subclasses kan kun tilgå private variable gennem `get()` metoder, dog kan de tilgå protected variable (en slags "intern public")
- Nedarvning af classes betyder, at metoder og funktionalitet er tilgængelige med det samme, og principielt behøver man ikke skrive noget yderligere kode for at bruge sin nye subclass. Implementering (`implements`) af interfaces betyder derimod, at man selv skal supplere koden for hver af de metoder, som interfacet definerer.

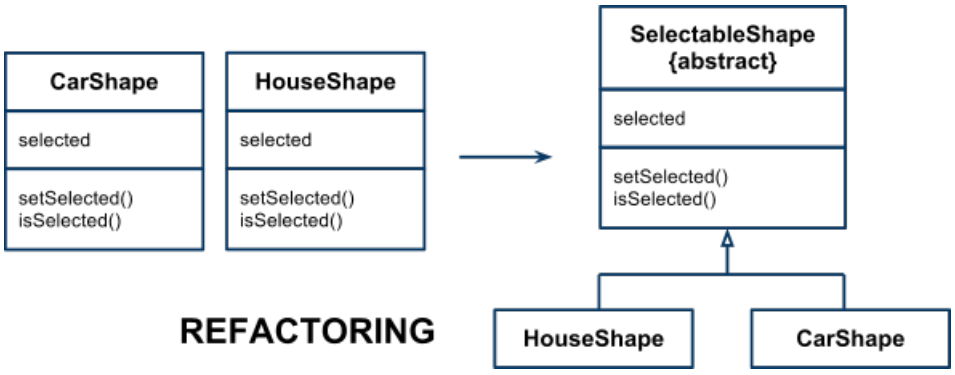
5.2 Abstract classes

- Abstract classes er ikke instansiérbare - dvs. man kan ikke lave objekter af abstract classes. En class tagges som abstract med keywordet `abstract`.
  Fx: `public abstract class SelectableShape implements SceneShape`

[4] Liskov's substitutionsprincip siger, at et object af en subclass kan benyttes nårsomhelst et objekt af dens superclass forventes. Eksempelvis kan et Manager objekt altid tildeles en Employee variabel, men ikke den anden vej rundt. Da Manager objekter altid er garanteret at have samme metoder og variable (og lidt til) som Employee objekter, er dette netop muligt.

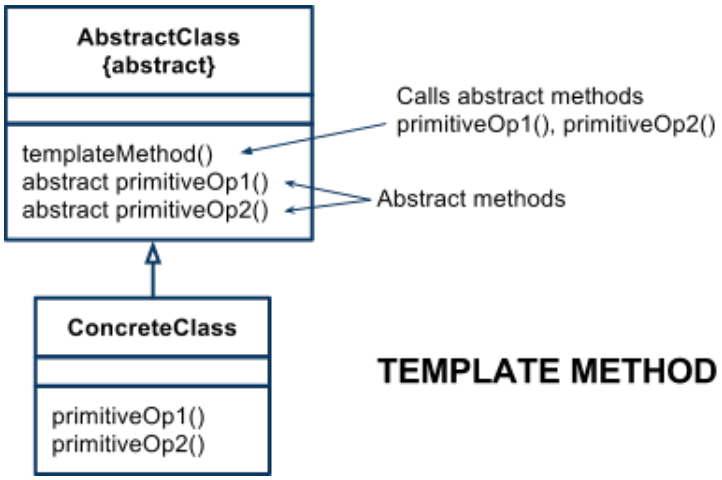
- Eksempel: SelectableShape. Abstract class der implementerer SceneShape interfacet, og som definerer nogle af interfacets metoder, men ikke alle. De metoder som ikke er definerede skal derfor defineres i subclasses (fx CarShape, HouseShape).
- Abstract classes implementerer typisk noget af funktionaliteten i de interfaces de evt. implementerer, men ikke alt. Resten af metoderne er abstract.
- Fordele: Man kan lægge al ensartet funktionalitet for flere classes i én superclass, som implementerer et interface, som er nyttigt for disse classes.
- Ulemper: Classes kan kun extende én superclass, men implementere adskillige interfaces.

- Refactoring - at omskrive kode så det er "bedre" eller lettere at forstå.
  Fx: "Extract superclass" - Lav en superclass ud fra de fælles features to eller flere classes har.



- Refactoring adskiller sig fra design patterns idet refactoring handler om at omskrive allerede eksisterende kode, mens design patterns handler om at planlægge sit arbejde så man kan undgå at skulle benytte sig af refactoring.

- Template method - en metode som flyttes fra subclasses til en superclass, og som kalder primitive operations, som kan være forskellige, i subclasses.
  - Design pattern
  - En `templateMethod()` er en metode i en abstract class, som kalder abstract methods (dvs metoder som bliver defineret af subclasses som extender denne abstract class). Altså er en `templateMethod()` en metode, som kalder andre metoder, som ikke endnu er blevet defineret.
  - Eksempel: `drawSelection()` i SelectableShape - `drawSelection()` er defineret i SelectableShape, men kalder metoder `draw()` og `translate()`, som er abstract methods i SelectableShape, men som bliver defineret i CarShape og HouseShape.



- Protected interfaces
  - protected er en mellemting mellem public og private; kan ses som "internt public" for alle subclasses til en class, men "eksternt private" for alle andre classes.
  - Metoder kan kun bruge protected metoder/fields fra objekter af samme class som de ligger i. Fx kan HouseShape ikke kalde `add()` på en CarShape, selvom de begge extender CompoundShape, som har en protected `add()` metode.
  - Protected fields bør undgås af samme grund som public fields bør undgås.

5.3 Hvornår er nedarvning ikke en god ide?

- Nedarvning bruges til "is-a" forhold.
  - "is-a" forhold betyder at en class er en specialiseret version af en anden class.
    Fx, "CarShape" is a "SelectableShape"
- Aggregering bruges til "has-a" forhold.
  - "has-a" forhold betyder at en class har et field hvis type er en anden class.
    Fx, "Circle" has a "Point" - Circle classen har et Point objekt som sit centrum. Den er ikke et Point objekt i sig selv.
- Nedarvning bør ikke bruges, hvis resultatet bryder med Liskov's substitutionsprincip. Fx, `Stack<T> extends Vector<T>` er en dum idé, da en stack og en vektor ikke har noget med hinanden at gøre. Med en stack kan man kun push og pop elementer på og af, og i LIFO[5] orden. Med en vektor kan man fjerne elementer tilfældige steder i arrayet.

[5] LIFO = Last In First Out

5.4 Dispositionsforslag 1

- **Nedarvning og polymorfi**
  - Hvad er polymorfi?
  - Formel/aktuel type
  - Lisskovs substitutionsprincip
  - Eksempel: `public class Manager extends Employee ...`
- **Abstrakte klasser**
  - En slags class-interface 'hybrid'
  - Ingen instantiering, delvis implementation
  - Eksempel: `public abstract class Carnivore implements Animal ...`
- **Refactoring**
  - Genbruge kode
  - Fælles kode i abstrakte superklasser
  - Mindre arbejde i underklasser
  - Eksempel: Account-aflevering
- **Template Method Pattern**
  - UML-diagram
  - Eksempel: Animal-Carnivore-Cat fra review-opgaverne
  - Eksempel: Applets og metoderne `init()`, `start()`, etc.

5.5 Dispositionsforslag 2

- **Nedarvning**
  - "arve" metoder fra superklassen
  - visibility modifiers
  - (super keyword)
  - Kan kun nedarve fra én klasse
  - Regler ved nedarvning
    - "is-a relationen" (<subtype>is a <supertype>)
    - Liskov substitution[6] (en subklasse kan altid indsættes i stedet for en superklasse)
    - Eksempel der overholder Liskov substitution
      - Både is-a og Liskov glæder (Crab extends Animal (fra dIntProg) eller JonglerendeKlovn extends Klovn)
      - Kun is-a gælder (Square extends Rectangle[7])
    - Preconditions (maks lige så stærke)
    - Postconditions (mindst lige så stærke)
  - Centralisering af kode ved at flytte det op i supertype (referer til aflevering om BankAccount)
- **Polymorfi**
  - Flere metoder med samme navn
    - typesystemet sørger for korrekt kald
    - dynamisk/statisk type
  - Eksempel på polymorfi
    - Artist a = new Klovn();
- **Abstrakte klasser**
  - Flere klasser har ens metoder
  - Centralisering af kode
  - tvinge subklasser til at implementere metoder
- **TEMPLATE METHOD pattern**
  - Eks. Gå over vejen (menneske og hund)
    - primitive metoder (kig() og gå())
    - Da hunden har 4 ben går den anderledes end mennesket
    - Den kigger sig nok heller ikke så godt for

[6] Husk at Barbara Liskov er en kvinde
[7] Da square er mere restriktiv end rectangle

6 Exceptions & Files

Exceptions er en smart måde at håndtere programfejl på. Brug af exceptions betyder, at man kan undgå situationer hvor return-values for metodekald skal valideres på alle mulige måder, fx `if (value == -1)` eller `if (value != null)`. Klienter kan samtidig tvinges til at tage handling, hvis der sker fejl.

6.1 Defensive programming

- Gå ud fra, at programmet bliver afviklet i et fjendtligt miljø med egentlige fjender, men også inkompetente brugere.
- Check gyldighed af inputs, return-values, etc.

6.2 Hvis man ikke har exceptions?

- Return-values som fungerer som "fejl"-values (fx -1, null)
- Kræver mange if-else-statements, som gør, at det bliver mere uoverskueligt og kompliceret at separere fejlhåndteringskode fra andet kode.
- Ingen måde at tvinge "klienter" til at checke om returværdier er gyldige eller ej, så der er en uheldig risiko for, at programmet dermed fortsætter afvikling med ugyldige værdier, som ødelægger andre resultater - eller at programmet crasher.

6.3 Throwable hierakiet

- Unchecked exceptions (Throwable ← Exception ← RuntimeException)
  - Compileren kræver ikke, at unchecked exceptions håndteres, men man kan stadig håndtere dem (try-catch-blok) hvis man vil, på samme måde som med checked exceptions.
  - Bruges til situationer hvor et program kan fejle som resultat af inkompetente programmører på den ene eller anden måde, fx hvis en metode bliver bedt om at hente data fra plads "-1" i et array. Dette er en logisk fejl, som sikkert kunne undgås vha. nogle checks.
  - Fx NullPointerException, IllegalArgumentException
- Checked exceptions (Throwable ← Exception)
  - Compileren kræver håndtering af checked exceptions (try-catch-blok)
  - Bruges til situationer hvor et program kan fejle som resultat af noget, som programmøren ikke har kontrol over - fx hvis man forsøger at skrive en fil til en disk som er fuld.
  - Fx IOException, FileNotFoundException

6.4 Exception handling

- For at kaste exceptions bruges `throw`.
  - Fx `throw new IOException()`
  - Metoder, som kaster checked exceptions skal i metode-signaturen indeholde `throws` (bemærk s'et) efterfulgt af hvilke typer exceptions som kastes.
  - Fx `public void addFile(String filename) throws IOException`
  - Metoder som kun kaster unchecked exceptions behøver ingen `throws` i signaturen, men der er ingen regler imod det.

  - Metoder kan "kaste exceptions videre" (propagate an exception). For checked exceptions skal metoderne have `throws` i deres signatur, også selvom det ikke er metoden selv som kaster en exception (det kunne være et kald til en anden metode, som netop kaster exceptions).
  - For unchecked exceptions kan `throws` udelades.

- Exceptions håndteres ved at "beskytte" kode, som kan kaste exceptions, med en try-blok, efterfulgt af en catch-blok, som "fanger" den type exceptions der kan blive kastet fra try-blokken.

  - ```
    try {
        writeToFile(file);
    } catch (IOException e) {
        System.out.println("Unable to save to " + file);
    }
    ```

  - Catch efterfølges af en parentes indeholdende den type af exception som der ønskes håndteret i den efterfølgende blok. Man kan have flere catch-blokke efter en try-blok, så man kan specificere hvilken type fejl programmet har lavet.
  - `catch (Exception e)` vil fange ALLE exceptions, da alle exceptions er subtyper til Exception (polymorfi).
  - Det kræves, at en catch-blok, som følger en anden catch-blok, ikke fanger exceptions, som er supertyper til dem der fanges i ovenstående catch-blokke. Fx må `catch (Exception e)` ikke komme før `catch (IOException)`.

- try-catch-blokke kan følges af en valgfri `finally` blok.

  - finally-blokken udføres uanset om der kastes en exception eller ej. Den er ofte udeladt, men kan bruges til "oprydningsarbejde".
  - Den er ikke redundant da den også køres selvom der fx er et return-statement i try-blokken. Desuden køres den også selvom en kastet exception ikke bliver fanget af catch-blokken.

6.5 Error recovery and avoidance

- Man kan indkapsle try-catch-blokken i et loop, hvor man laver en slags "retry" ved at modificere elementer i catch-blokken.
- Fx hvis man forsøger at læse en fil som ikke eksisterer, kan man i catch-blokken ændre på filnavnet eller prøve at kigge i nærliggende mapper, inkrementere en `attempts` integer `MAX_ATTEMPTS` gange, og så lade try-blokken køre igen.
- Man kan forsøge at undgå exceptions og errors ved at lave relevante checks på return-values, før man blindt benytter dem.
- Typisk check `if (value != null)`

6.6 File handling

- Exceptions er meget relevante for filhåndtering, da der er mange tilfælde hvor programmøren ingen kontrol har over det miljø filerne befinder sig i.

  - Typiske filhåndteringsproblemer:
    * Åbne ikke-eksisterende filer.
    * Åbne filer som er i brug.
    ```

    - Skrive filer til en fuld harddisk.

- `java.io` bibliotek til filhåndtering
  - To typer filer
    - Tekstfiler: filer bestående af læselige bogstaver. Håndteres med readers/writers: FileReader, BufferedReader, FileWriter
    - Binary filer: filer bestående af byte-sekvenser. Håndteres med streams: FileInputStream, FileOutputStream
  - Filen åbnes, data læses/skrives, filen lukkes.
    - Til skrivning af tekstfiler benyttes FileWriter.

      ```
      FileWriter writer = new FileWriter(file);
      writer.write(text);
      writer.close();
      ```

    - Til læsning af tekstfiler wrappes FileReader objekter ofte i BufferedReaders, da FileReader ikke kan læse hele linjer ad gangen.

      ```
      FileReader fr = new FileReader(file);
      BufferedReader reader = new BufferedReader(fr);
      reader.readLine();
      reader.close();
      ```

  - Omringes af try-catch-blok, hvori bl.a. IOExceptions fanges.
  - Scanner
    - Kan læse input fra terminalen
    - Kan også læse input fra tekstfiler og finde betydningsfulde dele af inputtet vha. dens "parsing" metoder, fx `nextInt()`.

- Serialization
  - Hvis en klasse implementerer interfacet Serializable kan et objekt af klassen gemmes som en binær fil i én write operation (ObjectOutputStream)
  - Objektet kan ligeledes indlæses igen (ObjectInputStream)
  - Hvis dele af klassens tilstand ikke er Serializable (fx hvis man har en feltvariabel som er et HashMap) kan de markeres så de skippes af processen vha. Javas `transient` keyword. Ellers kastes en exception.

6.7 Dispositionsforslag 1

- **Exceptions**
  - Hvad er exceptions?
  - Hvis man ikke havde exceptions?
- **Checked vs. unchecked**
  - UML-diagram, Throwable-hierarkiet
  - Regler for checked exceptions (`throws`, `try-catch`)
- **Exception handling**
  - Defensive programming
  - Exceptions kastes med `throw new Exception()`
  - Avoidance, unchecked exceptions kan ofte undgås vha. checks
  - Recovery, for-løkke med 'attempts'-tæller for checked exceptions
- **Filhåndtering**
  - Hvorfor er exceptions relevante for filer?
  - Tekstfiler: `FileReader`, `BufferedReader`, `FileWriter`
  - Andre filer: `InputStream`, `OutputStream`

6.8 Dispositionsforslag 2

- hvad er exceptions og hvad bruges de til
- hvad ville man gøre uden exceptions[8] (sproget C har ingen exceptions)
- **2 typer exceptions** (checked og unchecked)
  - Tegn exceptions klassehierarki (Throwable, Exception, RuntimeException, Error)
  - Egne exceptions (Hvilken klasse skal nedarves fra)
  - Try / catch / finally (husk finally køres ALTID, og bruges ofte til cleanup)
  - Throws keyword
- **Fil håndtering**
  - Meget kan gå galt
  - Eksempler (Filen eksisterer ikke, ingen adgang til filen)
  - Relater evt. til aflevering om filhåndtering
- **Defensive programming**
  - Minimerer antallet af fejl
  - Tager højde for alt (især brugerinput)
  - Brugeren er ond
  - Programmør fejl
  - Sørg for at programmet ikke crasher

[8] I sproget C returnerer de fleste metoder en værdi der fortæller om kaldet gik godt eller skidt. For systemkald repræsenterer en negativ returværdi oftest at noget er gået galt.

7 The Java Type System and Object Model

7.1 Typer og værdier

- Alle typer i Java er én af følgende:
  - Primitive typer: int short long byte char float double boolean
  - Klassetyper, fx String, Rectangle
  - Interfacetyper, fx Comparable, Shape
  - Arraytyper, fx String[], int[], char[]. En arraytypes elementer kaldes arrayets "component type".
  - Null-typen, null
- Alle values i Java er én af følgende:
  - En værdi til en primitiv type, fx 13, 5.0, 'k'
  - En reference til et objekt af en klasse, fx new String("Hest")
  - En reference til et array, fx new int[] 2, 3, 5, 7, 11, 13
  - null
  - Man kan ikke have værdier til interface-typer, da de ikke kan instantieres.

7.2 Subtype-forhold

- Subtype-forholdet i Java følger Liskov's substitutionsprincip om, at subtyper kan benyttes hvor supertyper forventes. Vi betragter to typer, S og T. S er en subtype til typen T, hvis én af følgende regler passer:
  - S og T er samme type
  - S og T er begge klassetyper, og S er en direkte eller indirekte subclass til T
  - S og T er begge interface-typer, og S er direkte eller indirekte subinterface til T
  - S er en klassetype, T er en interface-type, og S eller en superclass til S implementerer T eller et subinterface til T
  - S og T er arraytyper, og component-typen til S er en subtype til component-typen til T
  - S er ikke en primitiv type, og T er Object
  - S er en arraytype, og T er Cloneable eller Serializable
  - S er null og T er ikke en primitiv type
- Ved subtype-forhold mellem to arraytyper kan man "trække array brackets fra" begge arrays indtil én eller begge typer ingen brackets har.
  - `Object[]` og `int[]`, træk brackets fra på begge arrays, `Object` og `int`, og da `int` er en primitiv type er den ikke en subtype til `Object`, og dermed er `int[]` ikke en subtype til `Object[]` og omvendt. `int[]` er dog en subtype til `Object`, så `int[][]` er en subtype til `Object[]`.
- Compiler checker kun formelle typer, hvilket vil sige, at selv om man fodrer en subtype til en supertype (Manager i Employee variabel, Liskov substitution), kan man kun kalde supertypens metoder, fordi compileren altid kun kigger på formelle typer.
  - Typecasting betyder at man omdanner et objekts statiske/formelle type til en anden type, fx

```
    Employee e = new Manager();
e.getBonus()               // COMPILE ERROR!
((Manager)e).getBonus()     // OK!
```

- Primitive typer kan også typecastes, men der er nogle bestemte termer i den forbindelse:
  - Widening, når man fx caster en int til en double. (`2 -> 2.0`) Sker automatisk! Fx double `x = 2 -> x = 2.0`
  - Narrowing, når man fx caster en double til en int. (`2.5 -> 2`) Kræver typecast! Fx `int y = (int) 5.2 -> y = 5`

7.3 Wrappers (aka. Snoop Dogg)

- De 8 primitive typer har tilhørende wrappers (klassetyper): Integer, Short, Long, Byte, Character, Float, Double, Boolean. De har altså samme navne som deres primitive modstykker, pånær Integer (int) og Character (char).
- Wrapper-classes kan wrappe primitive typer i objekter, hvor det er nødvendigt. Fx hvis man skal have en ArrayList med ints. ArrayList kan ikke have primitive typer.
- Auto-boxing er den automatiske konvertering mellem primitive typer og wrapper objekter. Dette gør at man ganske enkelt kan tilføje int-værdier til ArrayLists af Integers, uden at skulle konvertere manuelt først.

7.4 Enum

- Enumerated types er typer med et endeligt sæt værdier. Eksempler på enum typer er Suit med fire værdier: SPADE, CLUB, HEART, DIAMOND, eller Size, som fx kunne have værdierne: SMALL, MEDIUM, LARGE.
- Keywordet enum definerer en class med en private constructor (nye objekter kan ikke laves fra den) med nogle fields tagged som `public static final`. Det er forsvarligt at lade dem være public, idét klassen ingen metoder har, og dens fields er alle final.

7.5 Type inquiry

- Man kan teste om et objekt eller en expression er en reference til et objekt af en bestemt type, eller én af dens subtyper med instanceof operatoren.
  - `if (e instanceof Shape)`
- Man kan finde et objekts klasse med getClass() metoden. Den returnerer et Class objekt. Dette objekt indeholder en masse information om det objekt, metoden blev kaldt på. Fx navnet på den klasse objektet tilhører, eller klassens eventuelle superclass.
  - Med getName() kaldt på et Class objekt kan man få det præcise klassenavn for et objekt.
  - Class objekter kan beskrive enhver type, også primitive typer.
- Vil man teste om et objekt tilhører en bestemt class, kan man kalde `getClass()` på objektet og sammenligne med den bestemte class således:
  - `if (e.getClass() == Rectangle.class)`
- Type inquiry bør ikke bruges som en erstatning for polymorfi.

7.6 The Object class - mother of all classes

- Alle Java-classes er subclasses til Object. Hvis en class ikke extender nogle andre classes er den en direkte subclass til Object. Derfor er alle metoder i Object tilgængelige for alle objekter. De fire vigtigste (som også altid bør overrides i custom classes) er:
  - toString() som returnerer en String-repræsentation af objektet.
    - Bruges automatisk af forskellige standard-metoder. Fx hvis man prøver at printe et objekt vha. `System.out.println()`.
  - equals() som sammenligner et objekt med et andet.
    - Man kan nemt sammenligne to ints med hinanden som (`x == y`), men bruger man == operatoren på objekter sammenlignes objekternes referencer.
    - Det ønskes ofte at to objekter betragtes som værende "ens", hvis dele eller hele tilstande af objekter er ens. Fx kan to Beer-objekter betragtes som ens hvis deres name og percent fields er ens.
      Hvis to lister indeholder de samme elementer i forskellige rækkefølger kan de også i nogle situationer betragtes som ens.
    - Den perfekte equals metode starter ifølge Horstmann således:

```
        if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
```

    - Metoden skal være reflexive (`x.equals(x)`) og symmetrisk (`x.equals(y) <-> y.equals(x)`)
  - hashCode() som returnerer en hash code for objektet.
    - Hash codes skal være konsistente med equals metoden, så to objekter som "equals" hinanden genererer samme hash codes.
  - clone() som returnerer en kopi/klon af objektet.
    - Cloneable interface skal implementeres
    - Der skælnes mellem deep og shallow copies af objekter vha. clone metoden.
    - En shallow copy er en kopi, som ganske enkelt bare kopierer alle værdierne i originalens fields (`orig.x = ? -> clone.x = orig.x`). Hvis disse værdier er objektreferencer (hvilket de er, hvis de ikke er primitive værdier), vil kopien nu dele objektreferencer med originalen, hvilket betyder at en ændren i kopiens værdier også ændrer på originalens værdier!
    - En deep copy er en kopi som er magen til originalen, men har sine egne field-værdier (`orig.x = ? -> clone.x = new x`). Ændringer i kopien ændrer således ikke på originalen.
    - En "sufficiently deep copy" er en kopi som kun opretter nye objektreferencer til objekter som er mutérbare. String er immutérbare, så typiske name fields er ikke kritiske.

7.7 Serialization

- Man kan gemme objekter som binære filer, hvis klassen objekterne tilhører implementerer `Serializable` interfacet.

7.8 Generic Types

- Generics er typer som har en eller flere type-variable, som betegnes med bogstavet E, fx `ArrayList<E>` som altså instantieres med en ikke-primitiv type. Det kunne være `ArrayList<Integer>` eller `ArrayList<String>`. Metoderne i generic types arbejder med dette E, som altså ved instantiering er en egentlig type.

7.9 Dispositionsforslag 1

- **Typer**
  - Hvad er typer? Hvilke findes der?
  - Primitive typer, reference-typer
  - `null`-pointers
- **Subtype-forhold**
  - Keywords: `extends` og `implements`
  - Casting af reference-typer
  - Casting af primitive typer, widening/narrowing
- **Generics og Wrappers**
  - Generics omgår IntArrayList, StringArrayList, etc.
  - Tillader at benytte en generisk klasse med vilkårlige typer
  - Wrapper-klasser nødvendige for primitive typer
- **The Object Class**
  - Metoder: `toString()`, `equals()`, `hashCode()`
  - Forhold mellem `equals()` og `hashCode()`
  - Evt. `clone()`
- **Cloning**
  - Deep vs. shallow
  - Java API'en, ArrayList
  - Pointer-diagram

7.10 Dispositionsforslag 2

- **Typer**
  - Primitive typer
    - wrappers og auto boxing
    - widening / narrowing
    - type casts
  - Reference typer
    - klasse / interface / array
    - statisk / dynamisk type (skriv kode og forklar)
  - null (special case da null ikke er en type)
- **Object klassen**
  - Alle referencetyper nedarver fra Object
  - Implementerer generelle metoder
  - **Equals** (overvej at skrive kode[9])
    - $x = x$
    - $x = y \implies y = x$
    - $x = yy = z \implies x = z$
  - **Clone** (deep og shallow) - overvej at komme ind på immutable klasser
- **Generics**
  - Wildcards / extends / super
  - Eksempel på brug `Collections.sort()`
  - Kodeeksempel på generisk klasse

[9] Det tager lang tid at skrive koden og det er vigtigt at det er helt korrekt. Overvej om det er det værd, eller om du hellere vil have mere tid til at tale om de andre emner.

8 Frameworks & Collections

Frameworks er en samling af klasser som alle arbejder sammen og er relevante for et bestemt emne. Fx er Swing et framework, og dets klasser benyttes alle til at lave GUI'er med. Et andet framework er Applet, som bruges til at lave Java applets med, så de kan køres i web browsers.

8.1 Frameworks og libraries

- Frameworks adskiller sig fra normale program libraries på forskellige måder
  - Inversion of control - I modsætning til almindelige libraries fungerer det meste af execution flowet for et framework i selve framework klasserne. Dvs. frameworkets metoder kan overrides, men det er frameworket som sørger for at kalde dem. Dette fænomen kaldes også Hollywood-princippet: "Don't call us, we call you."
  - Extensibility - Brugere kan benytte frameworkets default behaviour eller override og specialisere dets metoder.
  - Niks pille - Frameworkets kode må generelt ikke ændres. Brugere er nødt til at extende og override for at specialisere funktionalitet.
- Frameworks er ikke design patterns, og der er ingen egentlige design regler for hvordan frameworks skal se ud - de består udelukkende af et sæt klasser, som udgør en slags generel service for en bestemt type program, og de benytter typisk mange forskellige design patterns.

8.2 Applets

- Applets er programmer som afvikles i web browsers. java.applet er et framework, som bruges til lige netop det formål.
- I de fleste selvstændige programmer er der en main metode, som er det første der bliver kørt når programmet starter. I Applet-frameworket er denne metode allerede defineret, og programmer som extender Applet, skal bare override nogle eller alle af følgende metoder:
  - init - Kaldes én gang, lige når appletten loades.
  - start - Kaldes når appletten startes (efter init)
  - stop - Kaldes når appletten eller browseren lukkes
  - destroy - Kaldes når browseren lukkes for at frigive resourcer
  - paint - Kaldes når appletten skal repaintes (ofte).
- Framework karakteristika for applets:
  - Applet-programmøren extender Applet frameworket (inheritance)
  - Applet-klassen sørger for alle de trivielle ting, som at parse parametre og checke hvornår appletten er synlig og sådan. Applet-programmøren overrider bare metoder som ønskes specialiseret.
  - Inversion of control gør, at programmøren ikke bekymres med hvornår og hvordan metoderne skal afvikles. Det sørger frameworket for.

8.3 Collections

- Collections er både en samling af almene datastrukturer og et framework for nye collection klasser. Frameworket indeholder nogle forskellige interfaces, fx:
  - **Collection** - Det mest generelle collection interface
  - **Set** - En usorteret collection some ikke tillader duplicates
  - **SortedSet** - En sorteret collection som ikke tillader duplicates. Elementerne sorteres vha. Comparable eller Comparator
  - **List** - En ordnet collection der kan indeholde duplikater
- Frameworket tilbyder desuden konkrete klasser som implementerer disse interfaces. Nogle af de vigtigste er:
  - **HashSet** - Et set som bruger hashing/hash codes til at finde dets elementer
  - **ArrayList** - En liste som internt er baseret på arrays. ArrayLists er gode, hvis man ønsker hurtig "random access" i listen, da dens "lookups" fungerer som i arrays. Til gengæld er ArrayList langsom til at tilføje og fjerne elementer, da der ved disse operationer laves en kopi af det nuværende array, plus/minus ét element.
  - **LinkedList** - En liste som ikke er baseret på arrays. LinkedLists er gode hvis man ønsker hurtige add/remove metoder, men deres "lookups" er meget langsomme sammenlignet med ArrayLists, da LinkedLists altid gennemløbes fra start til det index man søger.
- AbstractCollection er en abstrakt klasse, som implementerer Collection interfacet, og definerer alle metoderne, undtagen `iterator()` og `size()`. Metoder som `toArray()` bruger netop `iterator()` og `size()`, hvilket vidner om Template Method designmønstret.
  - AbstractCollection efterlader kun følgende to metoder til subclasses:
    - `int size()`
    - `Iterator<E> iterator()`
  - De fleste konkrete collection classes som extender AbstractCollection overrider dog `add()` metoden, da AbstractCollection's version er en dummy udgave som bare kaster en UnsupportedOperationException, hvilket giver mening for collections som skal være immutable.

8.4 Dispositionsforslag 1

- **Frameworks**
  - Frameworks vs. libraries
  - Inversion of Control
  - java.lang.Math er et library, ingen IoC
  - Java Applets hører til et framework
- **Applets**
  - Simpelt framework
  - Metoderne `init()`, `start()`, etc. → IoC
- **Collections**
  - Alle niveauer af et framework
  - Interfaces, fx List
  - Abstrakte klasser, fx AbstractCollection
  - Færdige klasser, fx ArrayList
  - Sammenligning af forskellige collections, ArrayList vs TreeSet
- **Template Method Pattern**
  - UML-diagram
  - MultiSet-aflevering

8.5 Dispositionsforslag 2

- **Framework vs program library**
  - Hollywood princippet "Don't call us, we call you"
  - hot spots / frozen spots
  - Eks. RPG spil eller applets[10]
    - Extend applet med metoderne (init(), start(), stop() og paint()) frameworket kalder selv disse metoder. Appletklassen er altså et hot spot.
- **Prototyper (PROTOTYPE pattern)**
  - Frameworket skal kunne kalde brugerens klasser
  - Laver nye objekter ved at klone en prototype
- **Collections framework**
  - sort / shuffle / min / max
  - Collection interface (hot spot) (generisk)
    - set (ingen dupletter) vs. list (elementers rækkefølge er bibeholdt)
    - Benyt muligvis muligheden for at tale om generiske klasser
  - AbstractCollection<E>
    - Opret nye collections ud fra denne klasse der implementerer de fleste metoder
    - **TEMPLATE METHOD pattern** (vær klar på at give specifikt ekesmpel)

[10] Det vigtige her er at du har styr på dit eksempel og føler dig sikker på at tale om det.

9 Multithreading

Tråde er er programstykker som kan eksekveres parallelt med hinanden, dvs. "på samme tid". På single-core systemer skifter operativsystemet konstant mellem trådene, hvilket giver illusionen af, at de kører parallelt. På multi-core systemer kan operativsystemet derimod virkelig afvikle tråde parallelt - én på hver core.

9.1 Fordele

- Gør det muligt at separere program-kode fra kontrol-kode, fx kan lange rekursive beregninger udføres i én tråd, mens main-tråden fortsat reagerer på inputs fra brugeren.
- Kan således forhindre at programmer "fryser" mens lange beregninger finder sted.

9.2 Ulemper

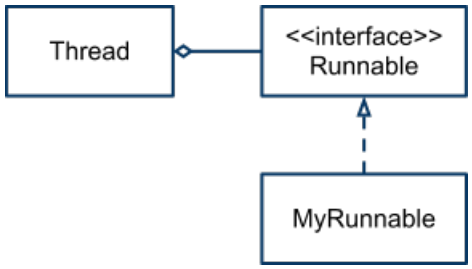
- Der kan opstå problemer, fx race conditions ved synkronisering hvis flere tråde deler data.
- Deadlocks hvis alle aktive tråde venter på at nogle andre tråde "gør noget"

9.3 Runnable interfacet

- En class som implementerer Runnable interfacet skal override metoden run(), hvorefter et Thread-objekt kan laves med et objekt af denne "Runnable class". Herefter kaldes start() på Thread-objektet, og så afvikles koden i Runnable objektets run() metode.

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

- UML diagram

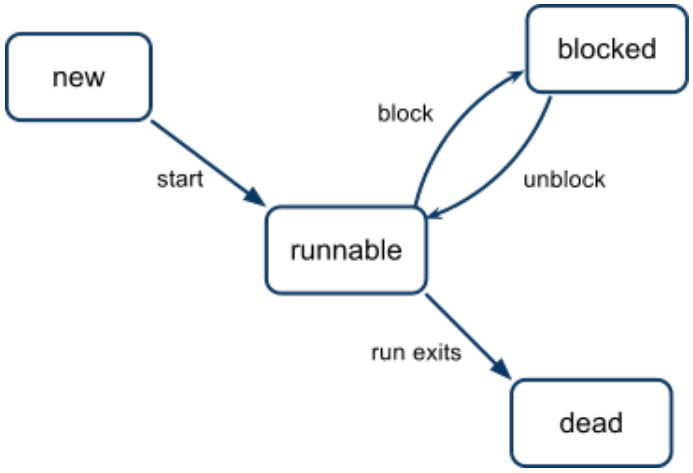


9.4 Selfishness

- En tråd skal lade andre tråde "komme til", ellers er den selfish. På nogle platforme kan en selfish tråd forhindre andre tråde i at blive afviklet, hvilket ødelægger formålet med tråde.
- Tråde kan sættes til at "sove" med sleep(). Når en tråd sættes til at sove betyder det, at andre tråde kan komme til.
- En tråds afviklingstid kaldes en "time slice".
  - main() metoden i ethvert Java program er sin egen tråd.
  - Hvis man har startet flere tråde i sit program afsluttes det først når alle trådene er færdige, eller er blevet interrupted.

9.5 Tilstand

- Diagram



  - Når der laves et nyt Thread objekt er den new. Når `start()`metoden kaldes bliver tråden runnable. Herfra er tråden klar til at blive kørt. Tråden kan blive blocked hvis der kaldes `wait()`eller `sleep()`.Dette sker som regel når tråden venter på en lås. Når tråden vækkes af enten `notify()`eller `signal()`bliver den igen runnable. Når `run()`metoden returnerer dør tråden.

- En tråd kan være blocked hvis den sover, venter på input/output, venter på en lock eller en condition.
- Når en tråd er blocked forbliver den blocked indtil den event den venter på sker. Fx, hvis en tråd sover bliver den først runnable igen når dens sovetid er gået.

9.6 interrupt()

- Tråde kan afbrydes/stoppes med `interrupt()`
- sleep() og wait() kaster en InterruptedException, hvis interupt() kaldes på en "sovende tråd". Denne exception skal fanges, og man bør derfor omringe al koden i run() med en try-blok.
- Hvis tråden ikke er blocked vil interrupt() bare sætte trådens interrupted field til true. Dette field kan checkes med metoden isInterrupted().
- I tilfælde af I/O eller andet "beskidt" arbejde bør tråde altid have noget oprydningskode efter try-blokken.

9.7 Race conditions

- Når flere tråde deler data, som kan korrupteres hvis trådene ikke afvikles i en bestemt rækkefølge.
- Hvis to tråde fx forsøger at lægge nyt indhold i en queue, og samtidig inkrementerer queuens tail, så kan race conditions gøre, at en tråd ikke når at inkrementere tailen før en ny tråd tager over. I så fald overskrives den første tråds data. Se evt. Horstmann s. 376
- Locks kan bruges til at blokkere alle andre tråde, som forsøger at tilgå den samme data som den nuværende tråd, indtil denne er færdig med sin kode. Når den første tråd frigiver locken, bliver de andre tråde unblocked.

  - En tråd kan "acquire" en lock ved at kalde lock() på locken. Hvis en anden tråd forsøger at acquire en locked lock bliver den blocked, indtil den første tråd kalder unlock() på locken.
  - Locks sikrer, at en tråd færdiggør sit arbejde med noget data før eventuelle andre tråde kan korruptere det.

- Deadlocks - når ingen tråde kan fortsætte fordi de venter på at en anden tråd gør noget først. Fx hvis en tråd venter på at der bliver plads i en fuld queue, men queuen ikke bliver tømt, da ingen tråde kan få locken, så de kan fjerne elementer fra queuen.
  - En lock kan have et tilknyttet Condition objekt. Hvis man kalder await() på et Condition objekt kan man lade andre tråde få den lock som den nuværende tråd har.
  - En tråd, som har kaldt await() på et tilhørende Condition objekt er blokkeret indtil en anden tråd kalder signalAll() på Condition objektet som tråden venter på.
  - Typisk laves checks på Condition objekter i while-løkker,
    fx `while (not OK to proceed) Condition.await()`

- Der er object locks bygget ind i alle Java objekter. Disse kan bruges på samme måde som ReentrantLocks, men i stedet for at opsætte locks skal synkroniserede metoder tagges med "synchronized" keywordet, fx "`public synchronized void add(E newValue)`".
  - Når en synchronized metode kaldes låses object locken.
  - Når en synchronized metode er færdig frigives object locken.
  - Object locks har én (anonym) Condition.
  - `wait() <-> await()` og `notifyAll() <-> signalAll()`, dvs. man kan frigive object locken på næsten samme måde som med ReentrantLock, og ligeledes signalere til alle ventende tråde.

- java.util.concurrent biblioteket har en BlockingQueue klasse, som er designet med race conditions i baghovedet. Queuens put() og take() metoder afventer på hhv. plads i køen og elementer i køen, før de fortsætter.
  - Concurrent = samtidig

9.8 Dispositionsforslag 1

- **Tråde**
  - Hvad er tråde? Hvad bruges de til?
  - Tråde i Java; `implements Runnable` eller `extends Thread`
  - UML diagram (Horstmann s. 364)
- **Tilstande**
  - new, runnable, blocked, dead
  - interrupt()
- **Race conditions**
  - Deadlocks (og livelocks)
  - Queue eksempel; korruptering af data, tegn på tavlen
  - ReentrantLocks vs. object locks
  - `wait()` og `notify()`
- **java.util.concurrent og BlockingQueue**
  - Masser af færdige, brugbare klasser
  - Metoder som `take()` blocker automatisk

9.9 Dispositionsforslag 2

- **Tråde**
  - Hvad er en Tråd
    - Producer / Consumer[11]
  - Hvorfor bruge threads
    - Referer til binomial aflevering, hvor beregningen kunne køres i en separat tråd, så GUIen ikke ville fryse
    - Kan benytte alle processorens kerner
  - **Runnable** - Hvordan laves en tråd
    - Implementer Runnable interfacet og benyt start() metoden på Thread
    - 4 states (new, runnable, blocked, dead) - lav tegning
  - Interrupts - try/catch i run() metoden
- **Synkronisering**
  - **Race conditions**
  - **Locking**
    - Eksplicitte locks fra java.util.concurrency (`await()` / `signalAll()`)
    - Synchronized keyword (Nemmere at bruge, men mindre fleksible)
    - **Deadlocks og livelocks**[12]
  - Køer
    - Udveksling af data mellem tråde
    - Benyttes ofte i forbindesle med producer/consumer
    - Undgå raceconditions ved at benytte en synkroniseret kø fx LinkedBlockingQueue

[11] En producer tråd laver data, fx læser ord i en fil. Oftest tilføjes denne data til en kø. Consumer tråd bruger denne data, fx lægger antal ord sammen for flere filer. Normalvis venter consumertråden på at der kommer data i køen.

[12] Livelocks er ikke ligeså vigtive som deadlocks, så undlad at snakke om disse hvis du ikke har styr på dem.