

Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- synlighetsmodifikatorer
- tilgangsmetoder
- valideringsmetoder og unntak
- innkapsling og implementasjon



- Java

- valideringsmetoder og unntak



- VS Code

- generering av tilgangsmetoder



Læringsmål for forelesningen

- OO
 - sikring av gyldig tilstand
 - synlighetsmodifikatorer
 - tilgangsmetoder
 - valideringsmetoder og unntak
 - innkapsling og implementasjon
- Java
 - valideringsmetoder og unntak
- VS Code
 - generering av tilgangsmetoder



Gyldig tilstand



- For (nesten) all type tilstand vil det finnes regler for gyldighet, f.eks.
 - *verdiområde* for tall
 - *syntaks* for tekst
- Regler for både enkeltverdier og kombinasjoner av verdier
 - kjønn og fødselsdato har egne regler, men er innbyrdes uavhengig
 - personnummer avhenger av både kjønn og fødselsdato (<http://no.wikipedia.org/wiki/Fødselsnummer>)
- Vi må sikre at all tilstand er gyldig og innbyrdes konsistent

Eksempel: Møtetidsrom

- Vi ønsker en klasse som holder styr på start- og slutt-tidspunkt for møter (eller avtaler generelt)
- Hva slags variabler (felt) trengs for å holde styr på dette?
- Hva slags regler for gyldighet gjelder?
- Er alle reglene like viktige?

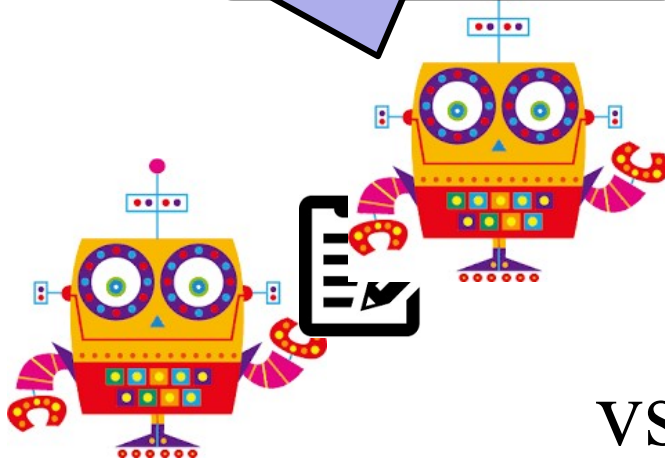
Møterom

- Starttid
- Sluttid
- Tema
- Hvem som booker
- Skal man kunne se hva andre bruker møterommet til, eller bare når?
- Innenfor tidsrom? Viktig?
- Ikke dobbeltbooking? Viktig?

Sikre gyldighet

- Todelt teknikk
 - tilby sikre *endringmetoder* med egen *valideringskode*
 - *hindre direkte tilgang* til tilstanden ved å angi at tilstandsvariablene skal ha begrenset tilgjengelighet vha. *synlighetsmodifikatoren* **private**
- Kun med ordentlig *innkapsling* kan man sikre gyldig tilstand
 - alle endringmetoder må *validere* argumentene, inkludert konstruktøren

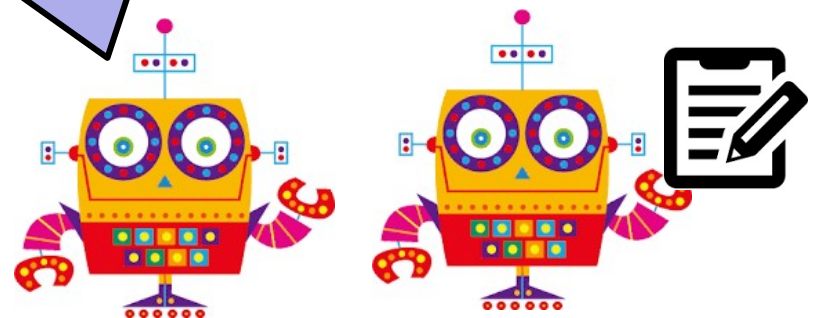
Jeg bare endrer
denne verdien,
jeg ...



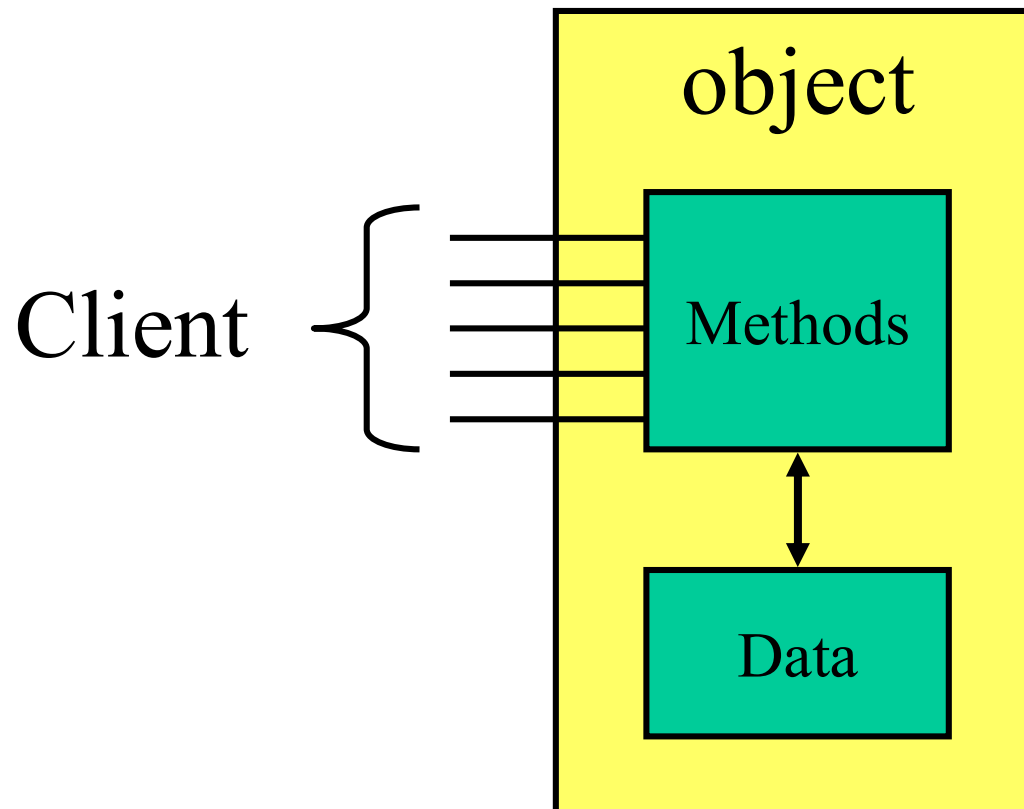
vs.

Sikre gyldig
tilstand

Gjør dette
for meg, er
du snill!



Illustrasjon av *innkapsling* fra en tidligere bok





Innkapsling

- Viktig prinsipp i objektorientering
 - Engelsk: *encapsulation*
 - Norsk: *innkapsling*
- Et objekt skal beskytte og håndtere sin egen informasjon
- Endringer i et objekts tilstand skal bare gjøres via trygge metoder
- Vi bør designe objektene slik at ikke andre objekter kan gå inn og endre tilstanden på en ukontrollert måte



Innkapsling

- Poenget er å definere
 - hva som skal være **private** egenskaper til et objekt, og
 - hva som skal være **offentlig** kjent av andre
- Skille mellom den private innsiden og den offentlige utsiden til et objekt
- Definere utsiden utelukkende vha. metoder (inkl. konstruktører) og regler for samhandling

Koding

- Vi skal lage en Bøk. Den må ha
 - Tilstand
 - Tittel
 - Antall sider
 - Hvor langt en har kommet i lesingen

Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- **synlighetsmodifikatorer**
- tilgangsmetoder
- valideringsmetoder og unntak
- innkapsling og implementasjon



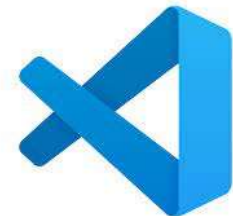
- Java

- valideringsmetoder og unntak



- VS Code

- generering av tilgangsmetoder



Synlighetsmodifikatorer

- Såkalt *synlighetsmodifikatorer* (visibility modifiers) brukes for å spesifisere hva som er privat og hva som er offentlig
 - **private** – skjult for andre klasser
 - **public** – åpent for alle klasser
 - **<ingenting>** - åpent for klasser i samme pakke
- Spesifikke tilgangsmetoder er offentlige og kan brukes for å lese eller endre objektets tilstand (feltene)



private

- “private visibility”
- Egenskaper (felt og metoder) som er deklarerert som **private** kan KUN brukes direkte av kode i **samme klasse**



public

- “public visibility”
- Egenskaper (felt og metoder) som er deklarerert som **public**, kan brukes direkte av kode i **alle klasser**
 - også i andre pakker



<ingenting>

- “package visibility”
- Felter og metoder uten spesifikk tilgang kan KUN brukes direkte av kode i **samme pakke**

La oss prøve

- pakke1.Klasse1
 - private felt1;
 - felt2;
 - public felt3;
- pakke1.Klasse2
- pakke2.Klasse3

- Hvilke av **pakke1.Klasse1** sine felt får **pakke1.Klasse2** og **pakke2.klasse3** tilgang til?



Felter hører til innsiden, (utvalgte) metoder hører til utsiden

- Grunnregel for styring av tilgang
 - felter *skal* være private
(unntaket er konstanter, deklarerert som final static)
 - get- og set-metoder *kan* være offentlige
- Generelt prinsipp
 - så lite som mulig skal være synlig utenfor en klasse
 - dess mindre som er synlig, dess mer kan endres uten at annen kode blir påvirket



Effekten av *private* og *public*

	public	private
Felter	Bryter med prinsippet om innkapsling	Håndhever Innkapsling
Metoder	Gir funksjonalitet til andre objekter	Interne støttefunksjoner i klassen

Utvide boken

- Sjekk synlighetsmodifikatorer

Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- synlighetsmodifikatorer
- **tilgangsmetoder**
- valideringsmetoder og unntak
- innkapsling og implementasjon



- Java

- valideringsmetoder og unntak



- VS Code

- generering av tilgangsmetoder





Felt og tilgangsmetoder

- Set-metoden *beskytter* mot gal bruk av felt
 - kan inneholde kode for konvertering og validering av verdien, f.eks. sjekke om dato er frem i tid, beløp er positivt, navn er gyldig, osv.
- Get- og set-metoder kan gi en *illusjon* av felt som egentlig ikke finnes
 - en get-metode kan *beregne* verdien sin, fra eksisterende felter
 - en set-metode kan tilsvarende endre (deler av) andre felter
- Sett utenifra er det viktigere å vite hvilke tilgangsmetoder et objekt har, enn hvilke felt som faktisk finnes

Fra felt til tilgangsmetoder

- Alle felt markeres som private
 - `<type> <feltnavn>` blir til
 - `private <type> <feltnavn>`
- Relevante, offentlige tilgangsmetoder legges til
 - ```
public <type> get<Feltnavn>() {
 return <feltnavn>;
}
// evt. følgende, når <type> er boolean
public boolean is<Feltnavn>() {
 return <feltnavn>;
}
- public void set<Feltnavn>(<type> <feltnavn>) {
 this.<feltnavn> = <feltnavn>;
}
```
- Merk konvensjonen for bruk av stor bokstav etter "get" og "set"-prefiksene
- Reglene er såpass enkle at VS Code (med utvidelser) har dem innebygget, inkludert en funksjon for å generere dem

# Utvidelse av boken

- Konstruktører
  - Lage en bok (flere måter?)
- Metoder
  - Skal metodene være tilgjengelige utenfor boken selv?
  - Vi må ha metoder for å sette tittel og antall sider
  - Vi må ha metoder for å lese ut tilstand
  - Vi må også kunne oppdatere tilstand med leste sider
  - Kan vi lage en finere utskrift av informasjonen for en Bok?



# Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- synlighetsmodifikatorer
- **tilgangsmetoder**
- valideringsmetoder og unntak
- innkapsling og implementasjon



- Java

- valideringsmetoder og unntak



- VS Code

- generering av tilgangsmetoder



# Fra felt til tilgangsmetoder

- Liste-felt gir mer kompliserte tilgangsmetoder, siden en ikke kan tillate tilgang til tabell-verdien direkte (hvorfor ikke?)
  - `List<type> <flertall>` blir til
  - `private List<type> <flertall>`
  
- Relevante, offentlige tilgangsmetoder legges til, f.eks.:
  - `public int get<Entall>Count();`
  - `public int indexOf<Entall>(<type> <entall>);`
  - `public <type> get<Entall>(int i);`
  - `public void set<Entall>(int i, <type> <entall>);`
  - `public void add<Entall>(<type> <entall>);`
  - `public void add/insert<Entall>(int i, <type> <entall>);`
  - `public void remove<Entall>(int i);`
  - `public void remove<Entall>(<type> <entall>);`
  
- Her finnes det ikke like klare regler som for enkle felt

# Kan vi lage en Bokhylle?

- Den skal kunne lagre mange bøker

# Eksempel:

## Person innkapsler

### List<Person> children

```
public class Person {

 // private felt
 private List<Person> children;

 // lesemetoder
 public int getChildCount();
 public int indexOfChild(Person child);
 public Person getChild(int i);

 // endringsmetoder
 public void setChild(int i, Person child);
 public void addChild(Person child);
 public void addChild(Person child, int i);
 public void removeChild(int i);
 public void removeChild(Person child);
}
```

# Advarsel

Ikke lag tilgangsmetoder ukritisk:

- Account – deposit og withdraw, ikke setBalance
- Stack – push og pop, ikke add og remove

# Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- synlighetsmodifikatorer
- tilgangsmetoder
- **valideringsmetoder og unntak**
- innkapsling og implementasjon



- Java

- **valideringsmetoder og unntak**



- VS Code

- generering av tilgangsmetoder



# Valideringsmetoder

- En bør definere egne metoder for å sjekke gyldighet, fordi
  - koden blir ryddigere, når kompleks logikk fordeles på flere metoder
  - utvalgte metoder kan gjøres til en del av innkapslingen, så andre klasser kan sjekke gyldighet på forhånd
- Utløser *unntak* ved feil...

# throw

- throw brukes for å *utløse et unntak*, dvs. si fra at et unntak har oppstått:

**throw new**

**<unntaksklasse> (...);**

- Java-maskineriet gjør dette, når du f.eks. prøver å gjøre ulovlige ting med null-verdier, deler på 0, refererer forbi enden av en tabell/array, osv.
- spesielle ting kan skje ved kjøring, f.eks. vil noen typer evige løkker gi StackOverflowException
- mange standard Java-metoder gjør dette, f.eks. mange metoder for filbehandling



# throw

- Du bør selv bruke throw når du oppdager at noe er galt, som ikke kan håndteres på en god måte der problemet oppdages
  - `IllegalArgumentException` brukes for ugyldige argumenter
  - `IllegalStateException` brukes når metoden kalles (på et tidspunkt) hvor det pga. tilstanden ikke er lov
- Vi skal etterhvert se hvordan en kan definere egne typer unntaksklasser (Exception-subklasser)

# Dato-eksempel

- Klasse med felt for dag, måned og år
- Regler for
  - enkeltverdier, f.eks.  $1 \leq \text{dag} \leq 31$
  - innbyrdes konsistens, f.eks. finnes ikke 31/2
- Anta én endringsmetoder pr. verdi
  - `setDay(int day)` – oppdaterer dagen
  - `setMonth(int month)` – oppdaterer måneden
  - `setYear(int year)` – oppdaterer året
- Hvordan bør validering håndteres?

# Dato-eksempel

- To typer valideringsmetoder
  - validering av enkeltverdi, f.eks. `isValidDay`
  - validering av innbyrdes konsistens, `isValidDate`
- To typer unntak
  - enkeltverdi-feil: **`IllegalArgumentException`**
  - feil ift. eksisterende tilstand: **`IllegalStateException`**
- Skal valideringsmetoden utløse unntaket?
  - + gjør endringsmetoden enklere
  - gjør metoden mindre praktisk å bruke for andre klasser

<https://www.ntnu.no/wiki/display/tdt4100/Gyldig+tilstand>

## Utvide boken...

- Vi kan ikke lese flere sider enn det er igjen å lese i boken!
- Hvis man forsøker det, så skal det utløses et unntak.

# Læringsmål for forelesningen

- OO

- sikring av gyldig tilstand med innkapsling
- synlighetsmodifikatorer
- tilgangsmetoder
- valideringsmetoder og unntak
- **innkapsling og implementasjon**



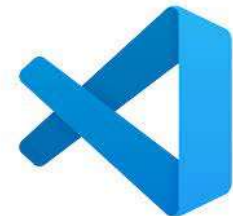
- Java

- valideringsmetoder og unntak



- VS Code

- generering av tilgangsmetoder



# Innkapsling og implementasjon

- Innkapsling gir større frihet til å endre *intern realisering*, fordi implementasjonsdetaljer ikke “lekker”
- Eksempel: Person-klasse med metodene
  - get/setGivenName
  - get/setFamilyName
  - get/setFullName
- To ulike realiseringer, samme innkapsling:
  - attributter for givenName og familyName
  - ett fullName-attributt

```
package encapsulation;

public class Person1 {

 private String givenName;
 private String familyName;

 public Person1(String givenName, String familyName) {
 this.givenName = givenName;
 this.familyName = familyName;
 }

 public String getGivenName() {
 return this.givenName;
 }
 public void setGivenName(String givenName) {
 this.givenName = givenName;
 }

 public String getFamilyName() {
 return this.familyName;
 }
 public void setFamilyName(String familyName) {
 this.familyName = familyName;
 }

 public String getFullName() {
 return this.givenName + " " + this.familyName;
 }
 public void setFullName(String fullName) {
 int pos = fullName.indexOf(' ');
 this.givenName = fullName.substring(0, pos);
 this.familyName = fullName.substring(pos + 1);
 }
}
```

**package** encapsulation;

**public class** Person2 {

**private** String fullName;

**public** Person2(String fullName) {  
        **this.fullName** = fullName;  
    }

**public** String getFullName() {  
        **return this.fullName**;  
    }

**public void** setFullName(String fullName) {  
        **this.fullName** = fullName;  
    }

**public** String getGivenName() {  
        **return this.fullName.substring(0, this.fullName.indexOf(' '))**;  
    }

**public void** setGivenName(String givenName) {  
        **this.fullName** = givenName + " " + getFamilyName();  
    }

**public** String getFamilyName() {  
        **return this.fullName.substring(this.fullName.indexOf(' ') + 1)**;  
    }

**public void** setFamilyName(String familyName) {  
        **this.fullName** = getGivenName() + " " + familyName;  
    }  
}



```
package encapsulation;
```

```
import junit.framework.TestCase;
```

```
public class PersonTest extends TestCase {
```

```
 private Person1 person1;
```

```
 private Person2 person2;
```

```
 @Override
```

```
 protected void setUp() throws Exception {
```

```
 person1 = new Person1("Ole", "Vik");
```

```
 person2 = new Person2("Ole Vik");
```

```
 }
```

```
 public void testPerson() {
```

```
 assertEquals("Ole", person1.getGivenName());
```

```
 assertEquals("Vik", person1.getFamilyName());
```

```
 assertEquals("Ole Vik", person1.getFullName());
```

```
 assertEquals("Ole", person2.getGivenName());
```

```
 assertEquals("Vik", person2.getFamilyName());
```

```
 assertEquals("Ole Vik", person2.getFullName());
```

```
 }
```

```
 public void testSetGivenFamilyNames() {
```

```
 person1.setGivenName("Jo");
```

```
 person1.setFamilyName("Eik");
```

```
 assertEquals("Jo", person1.getGivenName());
```

```
 assertEquals("Eik", person1.getFamilyName());
```

```
 assertEquals("Jo Eik", person1.getFullName());
```

```
 person2.setGivenName("Jo");
```

```
 person2.setFamilyName("Eik");
```

```
 assertEquals("Jo", person2.getGivenName());
```

```
 assertEquals("Eik", person2.getFamilyName());
```

```
 assertEquals("Jo Eik", person2.getFullName());
```

```
 }
```

```
 public void testSetFullName() {
```

```
 person1.setFullName("Jo Eik");
```

```
 assertEquals("Jo", person1.getGivenName());
```

```
 assertEquals("Eik", person1.getFamilyName());
```

```
 assertEquals("Jo Eik", person1.getFullName());
```

```
 person2.setFullName("Jo Eik");
```

```
 assertEquals("Jo", person2.getGivenName());
```

```
 assertEquals("Eik", person2.getFamilyName());
```

```
 assertEquals("Jo Eik", person2.getFullName());
```

```
 }
```