



NTNU

Kunnskap for en bedre verden

Vi fortsetter igjen 11:15

**Forelesningen tas
opp automatisk mens
det røde lyset ved
kateteret er på. 🎥**



NTNU

Kunnskap for en bedre verden

Forelesningen begynner 10:15

**Forelesningen tas
opp automatisk mens
det røde lyset ved
kateteret er på. 🎥**

TDT4100 Objektorientert programmering | 02.02.2024





NTNU

Kunnskap for en bedre verden

Øvingsforelesning 2

TDT4100 Objektorientert programmering

02.02.2024

Mathea Berg Vindsetmo

Vitenskapelig assistent, TDT4100

matheabv@stud.ntnu.no



Agenda

- Administrativt
- Litt mer om typer, konstruktører og **this**
- Presentasjon av øving 2
- Innkapsling og validering
- Praktisk oppgaveløsning

Godkjenning av øvinger

- Leveringsfrist for øving 1 er i dag klokken 23:59 (02.02)
 - Det er kun *.**java**-filer (ikke tester) og eventuelle tekstsvaer på oppgaver som skal leveres inn
 - Diagrammer tas med til læringsassistent på sal, og trenger *ikke* leveres inn på blackboard
- Husk å demonstrere øvingen for læringsassistent, fristen på dette er siste veiledningstime for læringsassistenten din **uken etter** (neste uke)
- Du får **ikke** poeng på øvingen før den er demonstrert for studass

Syntax og terminologi

En kjapp repetisjon av det vi har lært
hittil (og noe nytt)



Book.java

Vi skal gå gjennom steg for steg av koden fra forrige ukes øvingsforelesning for å bli kjent med java-syntaks og terminologi

```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```

Klassenavn / objekttype



```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```

Felter / attributter / tilstand



```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```

Konstruktør



```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```



NTNU

Metoder / oppførsel



```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```



NTNU

Main-metode



```
package of2.kode;

public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }

    public void setTitle(String title) {
        this.title=title;
    }

    public String getTitle() {
        return this.title;
    }

    @Override
    public String toString() {
        return "Boken heter" + this.title;
    }

    public static void main(String[] args) {
        Book book = new Book("Big Java");
    }
}
```

Klassenavn / objekttype

Felter / attributter / tilstand

Konstruktør

Metoder / oppførsel

Main-metode

Dette er en delen av programmet som kjøres når du kjører en .java-fil.
Sammenlignbart med å kjøre kode i Python

```
package of2.kode;  
  
public class Book {  
    private String title;  
  
    public Book(String title) {  
        this.title = title;  
    }  
  
    public void setTitle(String title) {  
        this.title=title;  
    }  
  
    public String getTitle() {  
        return this.title;  
    }  
  
    @Override  
    public String toString() {  
        return "Boken heter" + this.title;  
    }  
  
    public static void main(String[] args) {  
        Book book = new Book("Big Java");  
    }  
}
```

Parameter vs. Argument

Parameter

```
public void setName(String name) {  
    this.name = name;  
}
```

Argument

```
person.setName("Magnus");
```

* Argument er et begrep som brukes om de spesifikke instansene som sendes inn som parametere i en funksjon, mens parameter er navnet på variabelen som refererer denne dataen

toString, konstruktører og this

Noen viktige byggestener i Java / OOP



*Brand: Ford
Model: S-Max
Owner: Magnus
...*

toString()-metoden

- Gir en **tekstlig representasjon** av objektet.
- Det eksisterer en standard toString()-metode for alle objekter, men denne er ikke så informativ.
- Ved å bruke **@Override** over metoden, overskriver vi den og kan bestemme denne representasjonen selv.
- Et objekt kan inneholde mye informasjon, men vi velger ofte at toString-metoden returnerer informasjon om data-*feltene* / *tilstanden*.
- **Object.toString()** kalles automatisk når vi printer objektet.

toString()

Eksempel

Vi autogenerer en passende
toString-metode til **Car**-klassen



Standardkonstruktør

- Vi kan definere en konstruktør for å ta inn parametre slik at vi kan tilpasse **start-tilstanden** til objektene, men vi kan også la Javakompilatoren lage en konstruktør **automatisk**:

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person() {  
        this.name = "";  
        this.age = 0;  
    }  
}
```



```
public class Person {  
  
    private String name = "";  
    private int age = 0;  
  
}
```

Standardkonstruktør

Eksempel

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
    public String getName() {  
        return name;  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        System.out.println(p.getName());  
        System.out.println(p.getAge());  
    }  
}
```



```
<terminated> Person [Java Application] /Users/magnus/.p2/pool/p  
null  
0
```

Standardkonstruktør

Eksempel

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        System.out.println(p.getName());  
        System.out.println(p.getAge());  
    }  
}
```

Standardkonstruktøren vil kun legges inn dersom du **ikke** har definert noen andre konstruktører

Exception in thread
"main" java.lang.Error:
Unresolved compilation
problem:

The constructor Person()
is undefined at
foreksempel/of3.kode.Pers
on.main([Person.java:14](#))

Standardkonstruktør

Eksempel

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person() {  
    }  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
    }  
}
```

Løsningen blir i dette tilfellet å definere en ekstra tom konstruktør!



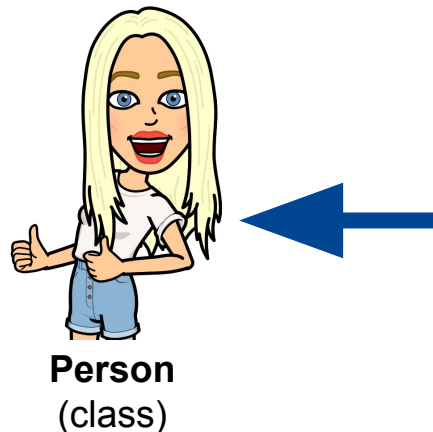
Flere konstruktører per klasse

I dette tilfellet lar klassen **Person** oss opprette objekter av denne typen selv om vi ikke vet alderen på forhånd.

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person(String name) {  
        this.name = name;  
    }  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person("Ola nordmann");  
        Person p2 = new Person("Kari nordmann", 23);  
    }  
}
```

Hva er: this

- **this** refererer alltid til selve **objektet** som koden kjøres i.
- Gir oss tilgang til alle metoder og felter for **denne spesifikke instansen** av klassen.



Eksempel this

Du er i Trondheim og ønsker å finne byens beste restaurant.



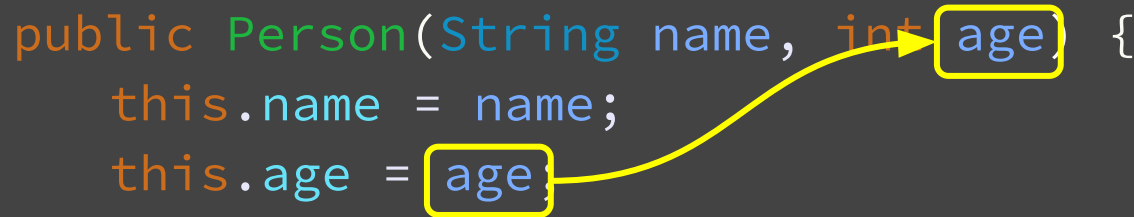
```
public class City {  
  
    private String name = "Trondheim";  
    private String bestRestaurant = "Speilsalen";  
  
    ...  
  
    public String getBestRestaurant(){  
        return this.bestRestaurant;  
    }  
}
```

Visualisering: this

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

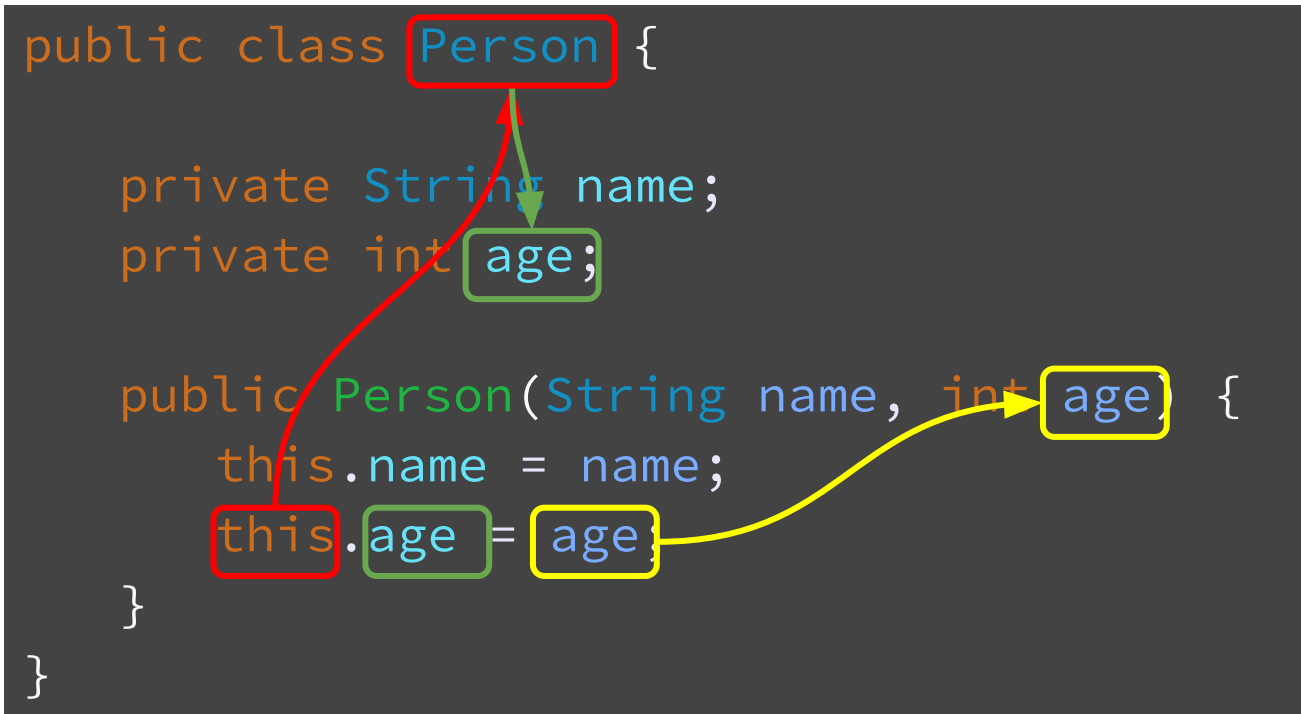
Visualisering: this

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```



Visualisering: this

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```



Bruk av this

Eksempel

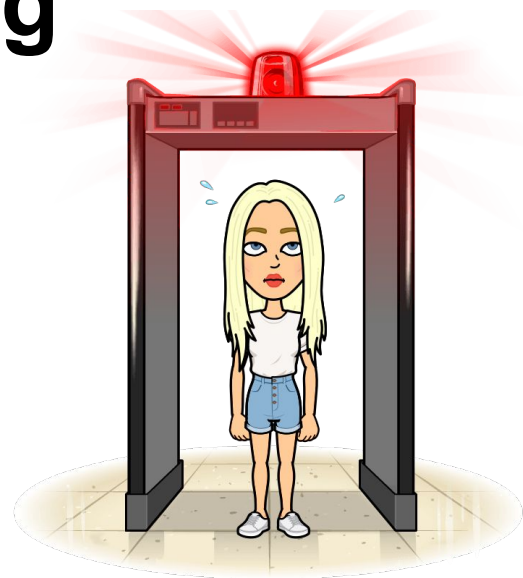
Vi oppretter en konstruktør for bruktbil i **Car.java**

Øving 2

- Tema: **Innkapsling og validering**
 - Lære å **innkapsle** klasser og metoder etter god programmeringsskikk
 - Lære å **validere** argumenter for å sikre gyldig tilstand
- Krav for å få godkjent øvingen:
 - Svare på teorispørsmål
 - Gjøre minst to oppgaver
- Leveringsfrist på Blackboard fredag 10. februar kl. 23:59
- Denne gangen skal dere gjøre øvingen under **ovinger/src/main/java/oving2** og du finner testene i **ovinger/src/test/java/oving2**

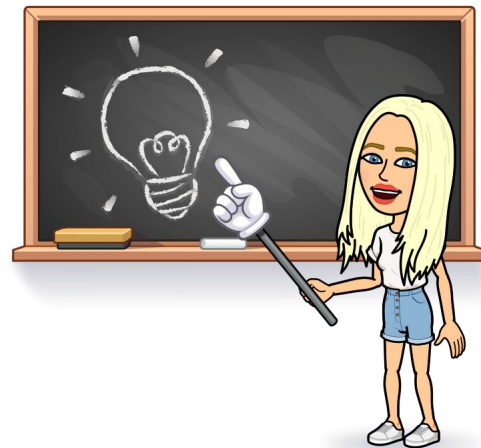
Innkapsling og validering

Essensiell teori for øving 2



Hva er motivasjonen bak dette?

- Vi skal lage oss noen **regler** for hvordan man kan bruke koden / klassene vi skriver.
- Reglene må vi definere **selv**. Hva det vil si at et objekt er i **gyldig tilstand** er helt opp til den som designer klassen.
- God innkapsling og validering kan sørge for **færre feilmeldinger** og gjøre det enklere for **andre** å sette seg inn i koden din.



Eksempel

La oss ta objektet Magnus som et eksempel:

Magnus er et objekt av typen **Person** og har (bla.) et **public** felt som heter **age**. Vi setter alderen hans til å være **25** og alt er bra

```
public class Person {  
    String name;  
    int age;  
}
```

```
Person person = new Person("Magnus", 25);
```

Eksempel

Senere kommer noen og endrer **Magnus** sitt **age**-felt til å være **-99** år.

Dette gir ikke mening, og med mindre andre har tatt høyde for at alder kan være -99 så vil vi nok møte på feil flere steder nå

```
person.age = -99;
```

Eksempel

Ved å **innkapsle** og **validere** input kan vi unngå ugyldige verdier i feltene, som igjen fører til **ugyldig tilstand**


```
public class Person {  
    private String name;  
    private int age;  
  
    public void setAge(int age) {  
        if (age ≥ 0) {  
            this.age = age;  
        }  
    }  
}
```

```
person.setAge(-99);  
person.setAge(26);
```



Eksempel

Ved å **innkapsle** og **validere** input kan vi unngå ugyldige verdier i feltene, som igjen fører til **ugyldig tilstand**




```
public class Person {  
    private String name;  
    private int age;  
  
    public void setAge(int age) {  
        if (age ≥ 0) {  
            this.age = age;  
        }  
    }  
}
```


```
person.setAge(-99);  
person.setAge(26);
```

Eksempel

Ved å **innkapsle** og **validere** input kan vi unngå ugyldige verdier i feltene, som igjen fører til **ugyldig tilstand**



```
public class Person {  
    private String name;  
    private int age;
```




```
    public void setAge(int age) {  
        if (age ≥ 0) {  
            this.age = age;  
        }  
    }  
}
```


```
person.setAge(-99);  
person.setAge(26);
```

Eksempel

Ved å **innkapsle** og **validere** input kan vi unngå ugyldige verdier i feltene, som igjen fører til **ugyldig tilstand**



```
public class Person {  
    private String name;  
    private int age;
```



```
    public void setAge(int age) {  
        if (age ≥ 0) {  
            this.age = age;  
        }  
    }  
}
```



```
person.setAge(-99);  
person.setAge(26);
```

Innkapsling

- Definisjon fra læreboken:
 - *Innkapsling er en programmeringsteknikk som har som formål å **skjule implementasjonsdetaljer** og heller gi metoder for **datatilgang***
- Definisjon fra wikien:
 - *Innkapsling er en programmeringsteknikk som har som formål å **hindre direkte tilgang** til tilstanden til et objekt fra objekter av **andre klasser***

Innkapsling

- Todelt motivasjon:
 - Det er viktig å sikre at tilstanden til alle objektene er gyldig. Dette gjøres best ved at alle endringer av tilstanden skjer ved å kalle objektets **metoder** (hvor vi kan bestemme regler), heller enn å endre på **attributtene** direkte.
 - Det er viktig at koden for en klasse ikke er avhengig av detaljer i en annen, f.eks. eksakt **hvilke attributter og datatyper som brukes for å representere data**, fordi dette gjør endringer lettere å håndtere.

Synlighetsmodifikatorer

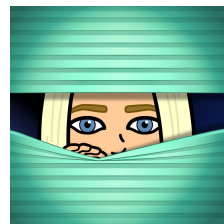
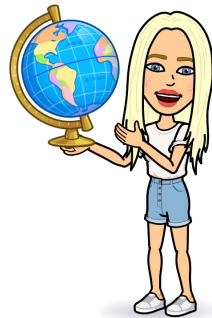
- Synlighetsmodifikatorer brukes for å spesifisere hva som er privat og hva som er offentlig, styrer tilgjengeligheten til klasser, variabler, metoder og konstruktører

```
private int numPages;  
private String title;  
  
public Book(int numPages, String title) {  
    this.numPages = numPages;  
    this.title = title;  
}  
  
public void setNumPages(int numPages) {  
    this.numPages = numPages;  
}
```

Synlighetsmodifikatorer

Hvilken synlighet gir de?

- **Public:** Klasse, pakke, subklasse, verden
- **Protected:** Klasse, pakke, subklasse
 - Bruker som regel bare dette til **arv** i dette emnet
 - (vi kommer tilbake til det...)
- **Ingen modifikator:** Klasse, pakke
- **Private:** Klasse



Når bør man bruke hva?

- Felter (intern tilstand)
 - Skal som regel være satt til **private**
 - Unntak: *Konstanter* (deklarert med **final static**)

```
public class Book {  
  
    private int numPages;  
    private String title;  
}
```

Når bør man bruke hva?

- Metoder, slik som **gettere**, **settere**, etc. (oppførsel)
 - Som regel **public**

```
public void setTitle(String title) {  
    this.title = title;  
}  
  
public String getTitle() {  
    return title;  
}
```

Når bør man bruke hva?

- Hvis vi derimot lager en **hjelpemetode** eller **valideringsmetode** som bare skal brukes **internt** i klassen så velger vi gjerne **private**

```
private void checkNumberNotNegative(double number) {  
    if (number < 0) {  
        throw new IllegalArgumentException("Number cannot be negative!");  
    }  
}
```

- Generelt: Minst mulig synlighet utenfor en klasse (stram innkapsling), følgelig kan mer innad i klassen endres uten å påvirke annen kode

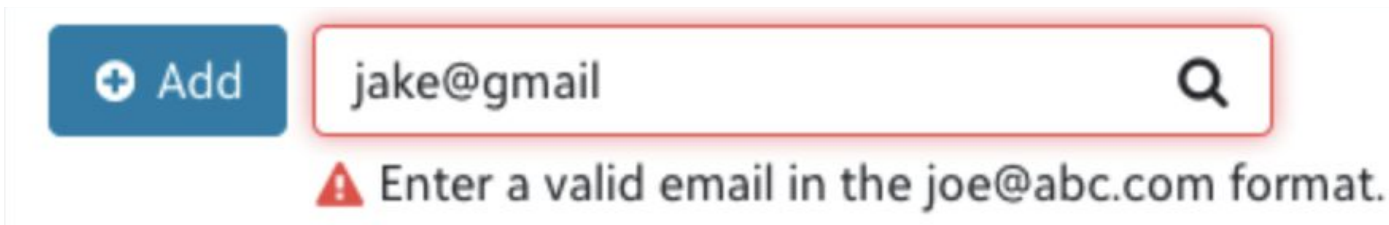
Getters og setters

- Et viktig ledd i innkapsling
- **Setters**
 - Lar oss definere regler for hvordan feltene kan endres (**validering**)
- **Getters**
 - Hjelper oss med å skjule **implementasjonsdetaljer** og er nødvendig ettersom feltene er **skjult** for omverden

```
public String getModel() {  
    return model;  
}  
  
public void setModel(String model) {  
    this.model = model;  
}
```

Validering

- Mange er sikkert kjent med at nettsider sjekker at du skriver inn en **gyldig** e-postadresse. Dette er et eksempel på **validering**:



A screenshot of a web form for adding an email address. On the left is a blue button with a white plus icon and the text "Add". To its right is a text input field containing "jake@gmail". The input field has a red border and a magnifying glass icon on the right. Below the input field is a red warning triangle icon followed by the text "Enter a valid email in the joe@abc.com format."

- Dersom man hadde skrevet inn en epostadresse som ikke er på riktig format kunne vi fått diverse feilmeldinger i programmet, og vil ville sagt at epost-objektet ikke var i **gyldig tilstand**



Uten valideringsmetode

```
public Book(int numPages, String title) {  
    if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
    this.numPages = numPages;  
    this.title = title;  
}  
  
public void setNumPages(int numPages) {  
    if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
    this.numPages = numPages;  
}
```

Uten valideringsmetode

```
public Book(int numPages, String title) {  
    → if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
    this.numPages = numPages;  
    this.title = title;  
}  
  
public void setNumPages(int numPages) {  
    → if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
    this.numPages = numPages;  
}
```


Med valideringsmetode

```
public Book(int numPages, String title) {  
    checkNumPagesIsValid(numPages);  
    this.numPages = numPages;  
    this.title = title;  
}  
  
public void setNumPages(int numPages) {  
    checkNumPagesIsValid(numPages);  
    this.numPages = numPages;  
}  
  
public void checkNumPagesIsValid(int numPages) {  
    if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
}
```

*For å gjøre koden bedre og mer tydelig bør valideringsmetode oftest returnere en logisk verdi, **true** dersom man mater inn gyldige verdier og **false** for ugyldige

Med valideringsmetode

```
public Book(int numPages, String title) {  
    checkNumPagesIsValid(numPages);  
    this.numPages = numPages;  
    this.title = title;  
}  
  
public void setNumPages(int numPages) {  
    checkNumPagesIsValid(numPages);  
    this.numPages = numPages;  
}  
  
public void checkNumPagesIsValid(int numPages) {  
    if (numPages < 0 || numPages % 2 != 0) {  
        throw new IllegalArgumentException("Number of pages not valid!");  
    }  
}
```



Kjapt om *unntakshåndtering*

- Dersom en valideringsmetode gir feil vil man ofte utløse **unntak** (gi en feilmelding), denne syntaksen er litt annerledes i Java sammenlignet med Python:

```
throw new IllegalArgumentException("Number of pages not valid!");
```

Unntakstype

Egendefinert feilmelding

Praktisk oppgaveløsning

For de neste ukene



Selvbetjent kassaapparat

Det nasjonale butikkonsernet *OOP mini* ønsker å lage nye selvbetjente kassaapparater for å gjøre handlingen enklere og mer effektiv.



Dagens oppgave (som vi kanskje også fortsetter med neste uke) er å utvikle en **prototype** for dette kassasystemet som kan gjøre enkle operasjoner som å scanne inn varer, regne ut pris / MVA (totalt og per vare), gi rabatt på enkelte varer / dager, samt printe ut en kvittering.



SelfServiceCheckout - del 1

Vi skal opprette en klasse som representerer den selvbetjente kassen,

Lag en klasse som heter **SelfServiceCheckout** som har følgende felter:

day	En tekststreng som representerer ukedag
days	En såkalt <i>konstant</i> som skal være satt som static og final . Dette gjør at feltet ikke kan endres i etterkant, og vil være tilgjengelig uten å instansiere et objekt. Feltet skal inneholde en liste med strenger av dager på formatet mon, tue, wed, thu, fri, sat, sun
phoneNumber	Et telefonnummer som kunder skal kunne skrive inn for å eksempelvis få rabatter eller tilsendt kvittering på mobil.

SelfServiceCheckout - del 2

Siden første versjon av av kassaapparatet er ganske enkel så må de ansatte hos OOP mini starte maskinen manuelt hver morgen og oppgi hvilken ukedag det er, da OOP har rabatter og kampanjer avhengig av hvilken ukedag det er.

- a) Lag metoden **validateDay** som sjekker at **day** er en ukedag på et av formatene: **mon, tue, wed, thu, fri, sat, sun**.
- b) Opprett en konstruktør for klassen som tar inn verdier for feltet **day**. Husk å kalle på valideringsfunksjonen fra oppgave a).

SelfServiceCheckout - del 3

Opprett en **setter** for feltet **phoneNumber**. Metoden skal ta inn et telefonnummer som skal være på et gyldig format (per norsk standard). Metoden skal akseptere et variabelt antall mellomrom i telefonnummeret, og det må starte med *enten* **+47** eller **0047**. I tillegg kan vi sjekke om det er et gyldig **mobilnummer** ved å sjekke om selve nummeret starter med **4** eller **9**.

Funksjonen skal kaste et **IllegalArgumentException** dersom argumentet er ugyldig.

SelfServiceCheckout - del 4

Vi skal nå lage funksjonen **scanItem** som tar inn navnet på en vare (**String**), prisen på denne (**double**), samt antallet varer av denne typen (**int**). Funksjonen skal legge til prisen på varen (minus eventuell rabatt) i totalsummen på kassaapparatet, og deretter printe ut følgende:

<antall>x <varenavn>: <pris minus rabatt> kr

Dersom brukeren har tastet inn et telefonnummer så skal de også få en umiddelbar **10% rabatt** på varen hvis det er *Tilbuds-Torsdag* hos OOP mini.

SelfServiceCheckout - Ekstra

Vi kan fortsatt **validere** koden vi har skrevet enda litt mer

Her kan man gjøre mye forskjellig, men vi skal konkret prøve nå å ta for oss noen tilfeller hvor brukeren gjør noe vi hittil **ikke har forventet...**

Tjeneste- og dataorienterte objekter

```
public class Book {  
  
    private String title;  
    private int numPages;  
  
    public Book(int numPages, String title) {  
        this.title = title;  
        this.numPages = numPages;  
    }  
  
    public String getTitle() {  
        return this.title;  
    }  
  
    public int getNumPages() {  
        return this.numPages;  
    }  
  
    public void setTitle(String title) {  
        this.title = title;  
    }  
}
```

```
public class KaffeMater {  
  
    public void serverKaffe(Person person) {  
        while (person.nokKaffe() == false) {  
            System.out.println("KaffeMater mater "+person.getNavn());  
            person.drikkKaffe();  
        }  
    }  
  
    public KaffeMater() {  
        System.out.println("Vi lager en KM!");  
    }  
  
    Run | Debug  
    public static void main(String[] args) {  
        KaffeMater kaffeMater = new KaffeMater();  
  
        Person person = new Person("Bernt");  
        System.out.println("Nok kaffe: "+person.nokKaffe());  
        kaffeMater.serverKaffe(person);  
        System.out.println("Nok kaffe: "+person.nokKaffe());  
    }  
}
```

Hva kan vi si om disse klassene?

Lykke til med ukas øving!

Spørsmål og tilbakemeldinger kan sendes til
matheabv@stud.ntnu.no