



Last ned Java-koden mens du venter:

I Eclipse: File -> Import... -> Git -> Projects from Git -> Clone URI ->

URI:

<https://github.com/Magssch/Eksamensforelesning-Java-2019.git>

# Eksamensforelesning

TDT4100 Objektorientert  
programmering



# Eksamensforelesning

TDT4100 Objektorientert  
programmering

15.05.2019

Magnus Eide Schjølberg



# Les først!

- Dette er **IKKE** en offisiell eksamensforelesning.
  - Jeg vet ingenting om årets eksamen
  - Dette er ikke en fasit.
- Greit å vite:
  - Jeg prøver å forenkle begreper og konsepter slik at det forhåpentligvis blir enklere å forstå
  - Noen ting kan derfor potensielt være litt upresist i forhold til hva man bør svare på eksamen hvis det kommer teorispørsmål.



## Praktisk info

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

- Gi tilbakemelding med “give advice”

Hent slides fra Drive: <https://s.ntnu.no/java-slides>



# Praktisk info

- Vi holder på i ca. 6 timer
- **Si ifra** hvis det går for raskt framover! (“Confused?” på Sembly)
- Vi kjører akademisk kvarter -> pause fra hel til kvart over
- **Still spørsmål!** Jeg kan ikke garantere et smart svar, men jeg skal gjøre mitt beste :-) (“Ask” på Sembly”)
  - Husk: hvis det er noe du lurer på er det mest sannsynlig flere som lurer på det samme.
  - Dere kan også svare hverandre på Sembly.





# Tema: helse



Last ned koden

Clone repository i Eclipse:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>

Eller kopier koden fra github og lim inn i Eclipse manuelt



# Agenda

- 12:00 - 14:00
  - Basics i Java og OOP
    - Innkapsling og validering
    - this, synlighetsmodifikatorer, static og final
    - konstruktører
  - Diagrammer
  - Collections/Lists/Arrays mm.
- 14:15 - 16:00
  - Arv, grensesnitt, abstrakte klasser
  - Iterator/Iterable
  - Comparator/Comparable
  - Funksjonelle grensesnitt
  - Lambda-uttrykk og Streams
- 16:15 - 18:00
  - Delegering
  - Observable/Observator
  - I/O (lese og skrive til fil)
  - Exceptions
  - Testing/JUnit
  - JavaFX





# Agenda

- Hvis vi får tid:
  - Kjapp gjennomgang av Javadocs
  - Ser litt mer på mulige oppgaver opp mot temaet “helse”

# Husk å bruke Stackoverflow!

- Stackoverflow er guds gave til alle som driver med programering
- Hemmeligheten er at ingen egentlig har peiling på hvordan man gjør det meste.
- Alle bare googler det man lurer på.
- Å bli god i programering handler i stor grad om å bli god på å google og finne svaret/koden du trenger



A screenshot of a Stack Overflow question page. The question is titled "Difference between append vs. extend list methods in Python". It asks, "What's the difference between the list methods `append()` and `extend()`?" The question has 2808 views, 547 answers, and was asked by Claudio on Oct 31 '08 at 5:55. The accepted answer, by codeforester, shows the following code:

```
x = [1, 2, 3]
x.append([4, 5])
print (x)
```

The answer gives the output: [1, 2, 3, [4, 5]]. A note below explains that `append` appends objects at the end, while `extend` extends the list by appending elements from the iterable.



The basics

Java™



# Innkapsling

- Hindre direkte tilgang til felt for andre klasser
- Dette gjøres med synlighetsmodifikatorer
  - **public, private, protected**
  - I tillegg, har vi **final, static**.
- Gir kun tilgang til det andre klasser trenger
  - get/set-metoder
- Forskjellige måter å innkapsle
  - Tenk fornavn/etternavn vs fullt navn



# Validering

- Sikre at metoder brukes slik de er ment å skulle brukes
- Sikre at objektet alltid har en gyldig tilstand
- Lurt: Ha en egen metode som f.eks (**checkNumberIsValid**)
  - Dette gjør at andre klasser kan spørre om det går greit før et evt unntak utløses.
- Utløse unntak når argumentet er ugyldig
  - **Throw new IllegalArgumentException**

# this

Hva er greia

- Referer til selve objektet som kjører koden.
- Kan også brukes som inputargument i andre funksjoner. Det representerer et objekt.
  - Som med alt annet i Java.
- Kan være greit å være konsekvent på å bruke det hvis man refererer til intern tilstand (variabler utenfor metoder)
- Hvis det er en variabel man definerer (lokalt) i en metode eller som viser til et inputargument, skal man IKKE bruke this.

```
public class Account {  
    private double saldo;  
}
```

Eksempel på  
bruk av **this**

```
public class Person {
```

```
    private int personID;
```

```
-
```

```
-
```

```
-
```

```
    public void setPersonID(int personID) {
```

```
        int temporary = 5;
```

```
        this.personID = personID;
```

```
        temporary = 3;
```

```
}
```



# Modifikatorer

Seks typer du må kjenne til

**public**  
**protected**  
**private**

**static**  
**final**  
**abstract**



# De tre essensielle synlighetsmodifikatorene

<b>public</b>	->	Synlig for alle
<b>protected</b>	->	Synlig i klassen og <u>underklasser</u>
<b>private</b>	->	Synlig <u>kun</u> i klassen



# De tre essensielle synlighetsmodifikatorene

## public

- Spør deg selv: er denne metoden nyttig for en annen klasse?
- Hvis ja: da er du good.
- *Variabler* bør generelt ikke være public i klasser.
  - Ref. innkapsling



# De tre essensielle synlighetsmodifikatorene

## **private**

- Hvis metoden kun har noen funksjon / skal brukes innad i klassen
- Bruker som regel **private** på *variabler* i klasser.





# De tre essensielle synlighetsmodifikatorene

## **protected**

- Bruk denne hvis du har en klasse og du skal lage underklasser som skal arve fra denne klassen og har bruk for å ha tilgang til denne metoden.
- Eventuelt hvis en underklasse har behov for *direkte tilgang* til en variabel. Bør likevel forsøke å bruke private/getter/setter så mye som mulig



# De litt mer forvirrende

## final

- Ved å bruke denne så gjør man det ulovlig å senere endre verdien av en variabel
- Kan også brukes på metoder og klasser, men dette er ikke relevant i faget
- Merk: Selv om en liste er markert som **Final** kan den fortsatt endres



# Eksempel fra K2017 Del 1 a)

“Både **Course** og **Meal** skal initialiseres med navn og beskrivelse, som siden ikke skal kunne endres. I den utgitte koden er det brukt to varianter for å håndtere dette. **Course** har public-felt og ingen get- eller set-metoder, mens **Meal** har private-felt og public get-metoder. Angi fordeler og ulemper med hver **kodingsteknikk**. Hvilken anbefaler du? Begrunn svaret!”

```
public class Course {  
  
    public final String name, description;  
  
    public Course(String name, String description) {  
        super();  
        ... initialization ...  
    }  
}  
  
/**  
 * Represents a set of (pre-defined) Courses that are ordered as a whole  
 */  
public class Meal {  
  
    private final String name, description;  
  
    public Meal(String name, String description, Course[] courses) {  
        super();  
        ... initialization ...  
        this.courses = Arrays.asList(courses);  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
}
```



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>



# De litt mer forvirrende

## **abstract**

- Brukes kun når vi skal lage en abstrakt klasse



# De litt mer forvirrende

## static

- Vi bruker denne dersom vi ikke trenger å bruke intern tilstand i en metode
- Kan også brukes når vi har variabler som er felles for **alle** objekter i en klasse
- Variabelen/metoden tilhører hele **klassen**, og ikke en spesifikk instans.

```
public class Person {  
  
    private static int nextPersonID = 1;  
    private int personID;  
  
    public Person() {  
        this.personID = nextPersonID;  
        nextPersonID++;  
    }  
}
```

```
public static void main(String[] args) {  
  
    // Vanlige variabler/metoder:  
    Person Magnus = new Person();  
    System.out.println(Magnus.personID);  
  
    // Når variabelen/metoden er statisk:  
    System.out.println(Person.nextPersonID);  
}
```

- 
- Dersom nextPersonID er satt til static, kan vi hente ut verdien på denne måten:

```
System.out.println(Person.nextPersonID);
```

- Hvis nextPersonID IKKE hadde vært static, måtte vi først instansiert et objekt av klassen Person:

```
Person Magnus = new Person();
```

```
System.out.println(Magnus.nextPersonID);
```



## Statiske metoder

- Som vanlige metoder, men kan ikke bruke nøkkelord som “this”

```
public static void incrementAge() {  
    this.age += 1;  
}
```

# Konstruktør: hva er det egentlig?

```
private double balance;
private double interestRate;

public Account(double balance, double interestRate) {
    checkNotNegative(balance, "Balance");
    this.balance = balance;
    setInterestRate(interestRate);
}

public Account(){}
```



# Greit å huske om konstruktører

- Lar oss instansiere klasser som objekt.
- En klasse kan ha flere konstruktører, gitt at de har forskjellig **antall** eller forskjellig **type** argumenter
- I konstruktøren kan man kalle på andre metoder i klassen, i tillegg til metoder og konstruktør fra superklasse

# Greit å huske om konstruktører

- Du kan instansiere objekter uten å ha definert en konstruktør, hvis alle objekter som lages skal være like

```
public class Account {  
  
    private double saldo;  
  
    public Account() {  
        this.saldo = 100;  
    }  
}
```



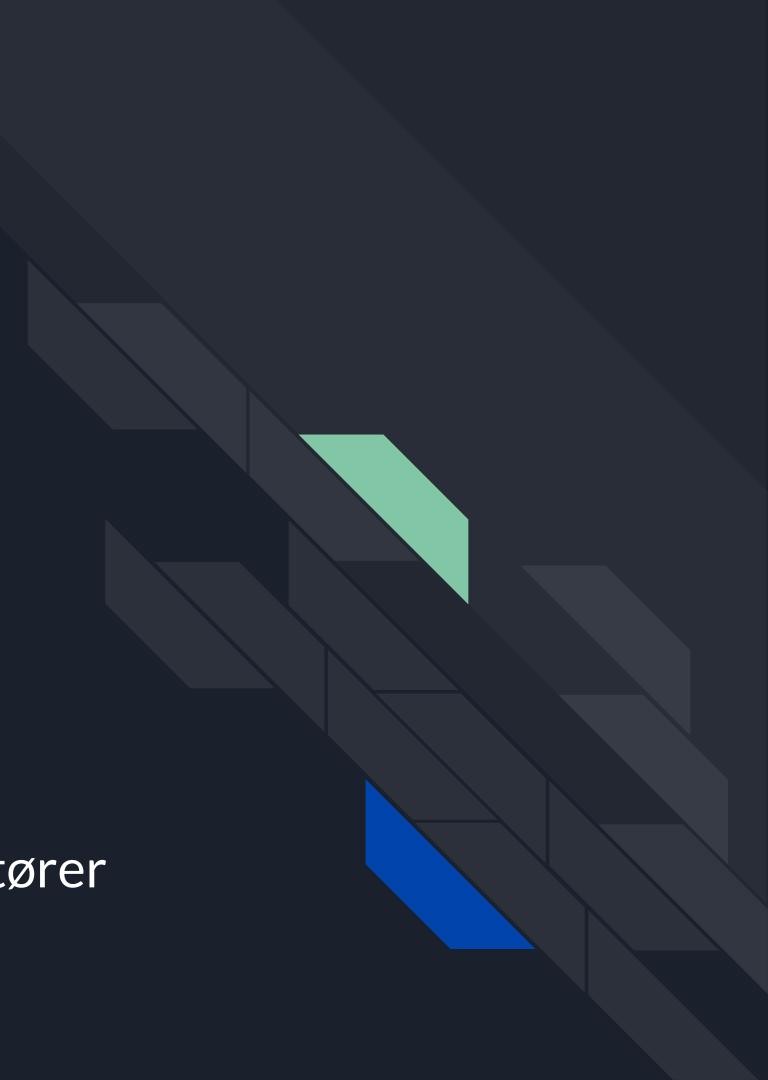
```
public class Account {  
  
    private double saldo = 100;  
}
```

```
public class Person2 {  
  
    private String name = "Ola Nordmann";  
    private int age = 0;  
    private double height = 0;  
  
    public Person2(String name) {          // Hvis vi kun vil definere navn  
        this.name = name;  
    }  
  
    public Person2(int age) {              // Hvis vi kun vil definere alder  
        this.age = age;  
    }  
  
    public Person2(double height) {       // Hvis vi kun vil definere høyde  
        this.height = height;  
    }  
}
```



```
public Person2(String name, int age, double height) {  
    this.name = name;  
    this.age = age;  
    this.height = height;  
}
```

# Vi skriver klassen **Medication**



- Lager gettere og setttere for innkapsling
- Sørger også for validering av price-feltet
- Vi prøver også med å lage flere konstruktører



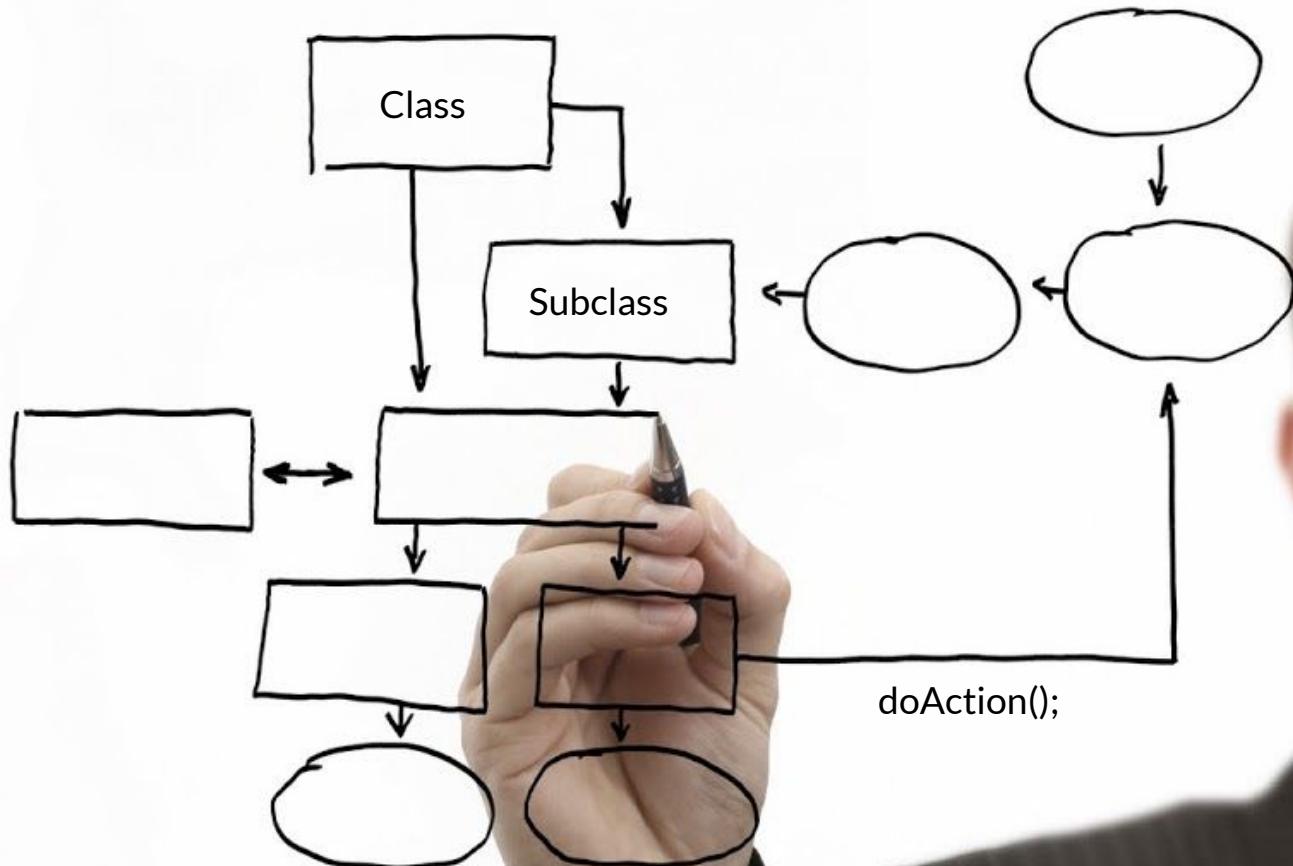
Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

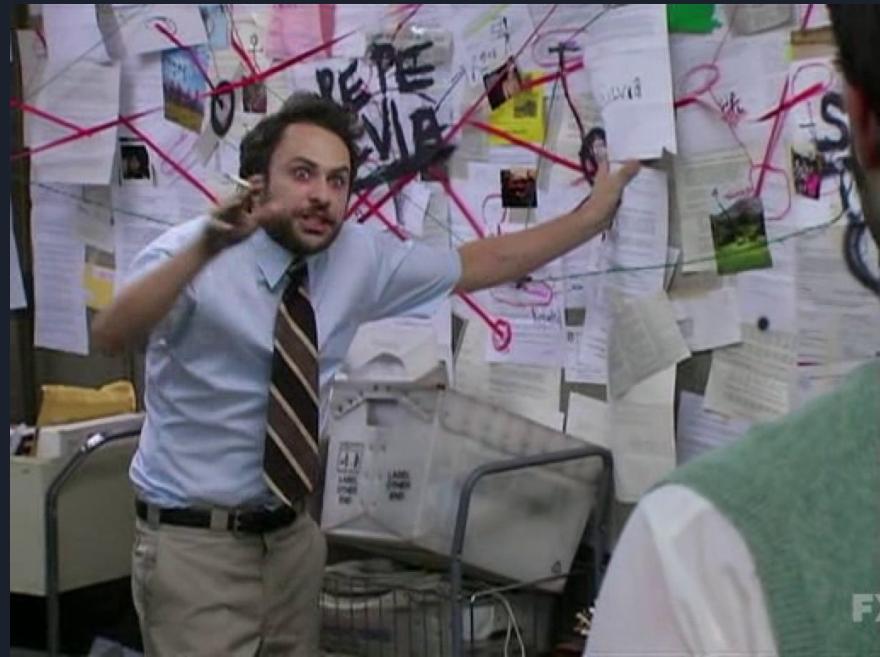
<https://github.com/Magssch/Eksamensforelesning-Java-2019>



Diagrammer

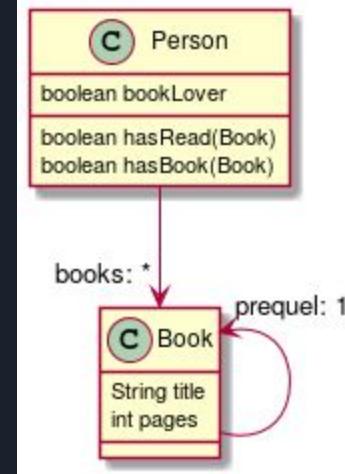
# Diagramtypene i TDT4100

- Klassediagram
- Objektdiagram
- Objektilstandsdiagram

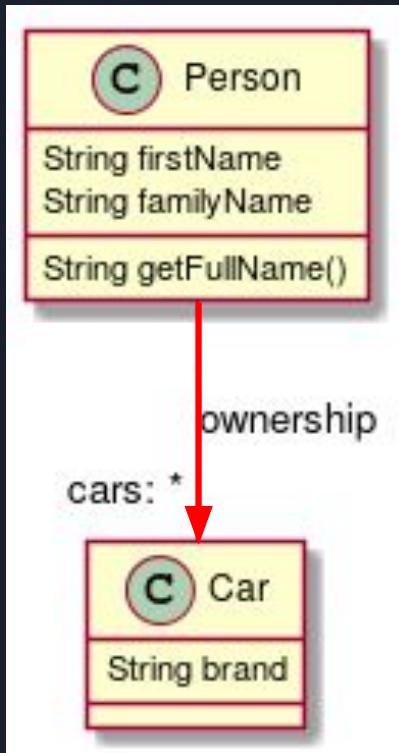


# Klassediagram

- Representerer objekter av **forskjellige typer/klasser** og hvordan de er koblet sammen.
- Viser hvilke klasser som arver av andre klasser
- Viser hvilke klasser som implementerer hvilke grensesnitt

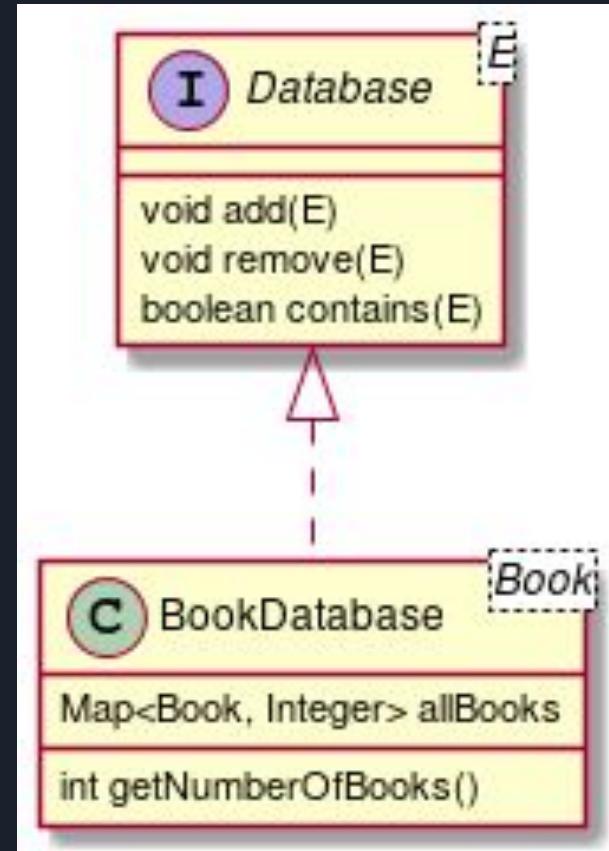
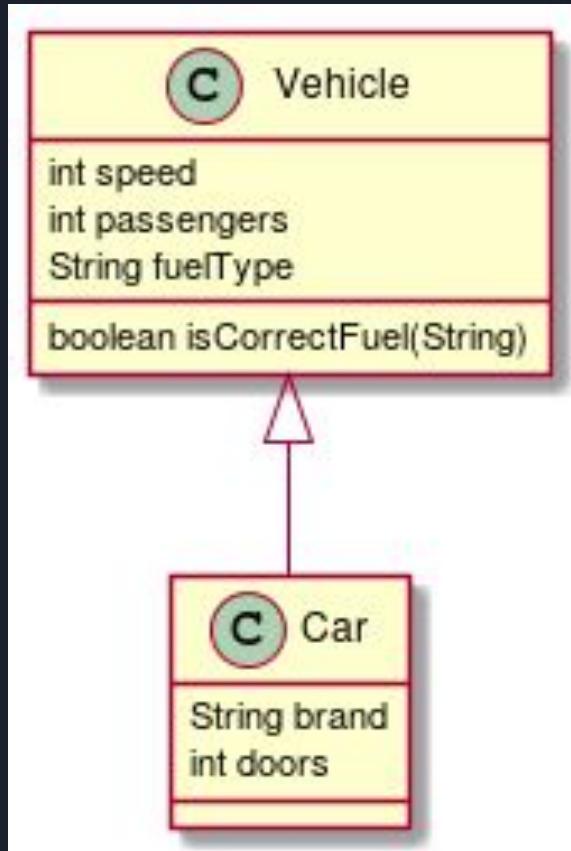


# Klassediagramm

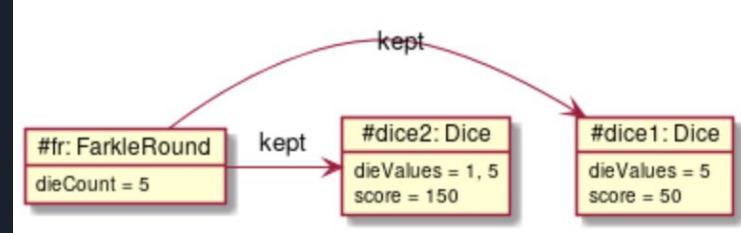


```
class Person {  
    String firstName;  
    String familyName;  
    String getFullName();  
    Collection<Car> cars;  
}  
  
class Car {  
    String brand;  
}
```

# Klassediagram: Arv og grensesnitt



# Objektdiagram



- Representere objekter og hvordan de er koblet sammen.
- Viser tilstand kun på ett **spesifikt** tidspunkt
- For å tegne disse er du nødt til å se på denne delen av koden:

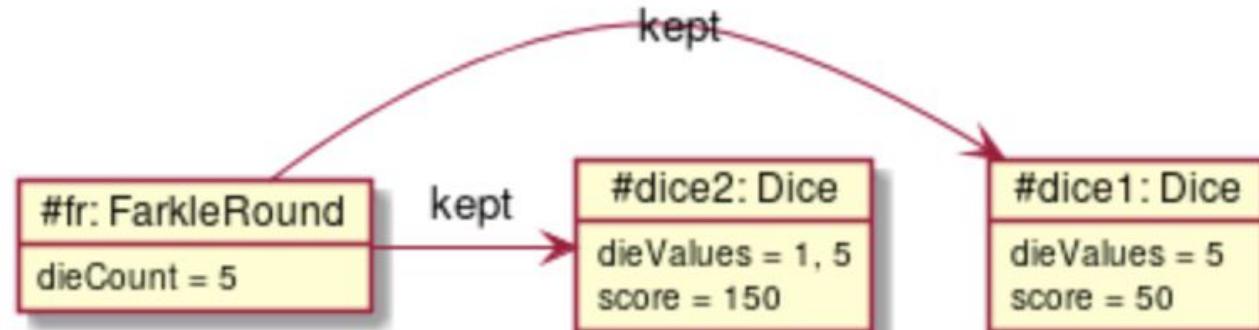
```
/*
public class FarkleRound {
    private int dieCount;
    private Collection<Dice> kept = new ArrayList<>();
    private Dice currentDice;
```



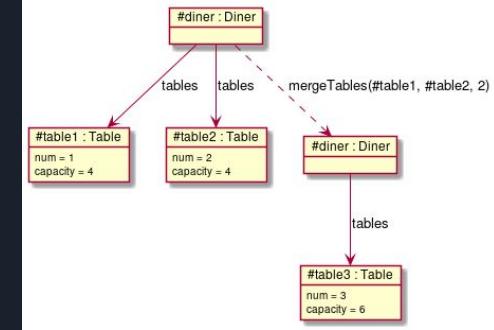
## Eksempel fra eksamen V2018 Del 4 e)

En kaster 2, 3, 4, 5 og 5. En legger til side en 5-er for 50 poeng, og kaster de fire gjenværende terningene. En får 1, 3, 5 og 6. En legger til side 1-eren og 5-eren for 150 nye poeng, altså 200 til sammen så langt

```
public class FarkleRound {  
    private int dieCount;  
    private Collection<Dice> kept = new ArrayList<>();  
    private Dice currentDice;  
  
    /**  
     * Initializes a new round, where dieCount number of dice is immediately rolled.  
     * Note that the round may be immediately finished, if the initial roll give no points.  
     * @param dieCount the number of dice rolled  
     * @param scoring the object responsible for computing the score  
     */  
    public FarkleRound(int dieCount) {  
        this.dieCount = dieCount;  
        roll(dieCount);  
    }  
    private void roll(int dieCount) {  
        this.currentDice = new Dice(Dice.randomDieValues(dieCount), -1);  
        if (computeDiceScores(currentDice).isEmpty()) {  
            this.kept.clear();  
            this.currentDice = null;  
        }  
    }  
}
```



# Objekttilstandsdiagram



- Representere objekter og hvordan de er koblet sammen
- Viser hvilke verdier (intern tilstand) disse objektene har etter at man kaller på metoder (altså over tid)
- Vi er nå nødt til å se på alle variabler (intern tilstand), i tillegg til metoder



Eksempel fra eksamen 2017

Tegn et objekttilstandsdiagram som  
illustrerer virkemåten til mergeTables.



Table-klassen:

# Eksempel fra eksamen 2017

## Tegn et objekttilstandsdiagram

```
public void mergeTables(Table table1, Table table2, int lostCapacity) {  
    checkNotOccupied(table1);  
    checkNotOccupied(table2);  
    Table table = new Table(table1.getCapacity() + table2.getCapacity() - lostCapacity);  
    removeTable(table1);  
    removeTable(table2);  
    addTable(table);  
}
```

```
private static int tableCounter = 1;  
  
private final int num;  
private final int capacity;  
  
public Table(int capacity) {  
    this.num = tableCounter++;  
    this.capacity = capacity;  
}
```

Table-klassen:

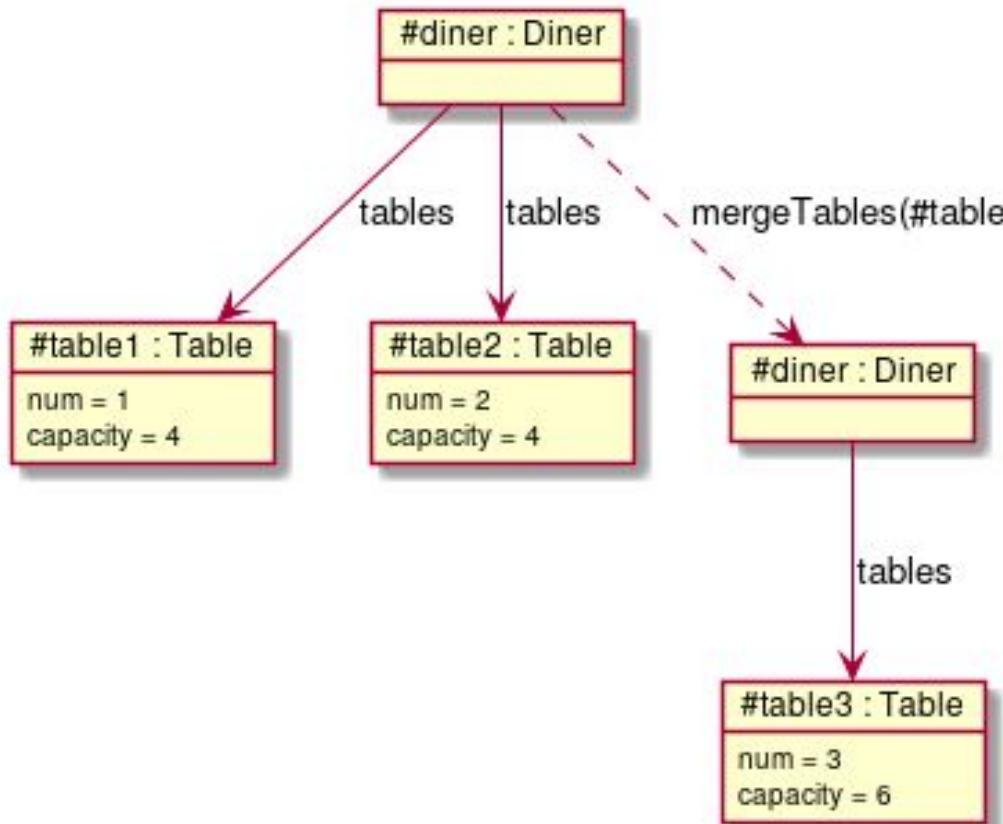
```
private static int tableCounter = 1;  
  
int num;  
int capacity;  
  
int capacity) {  
n = tableCounter++;  
capacity = capacity;
```

- lostCapacity)

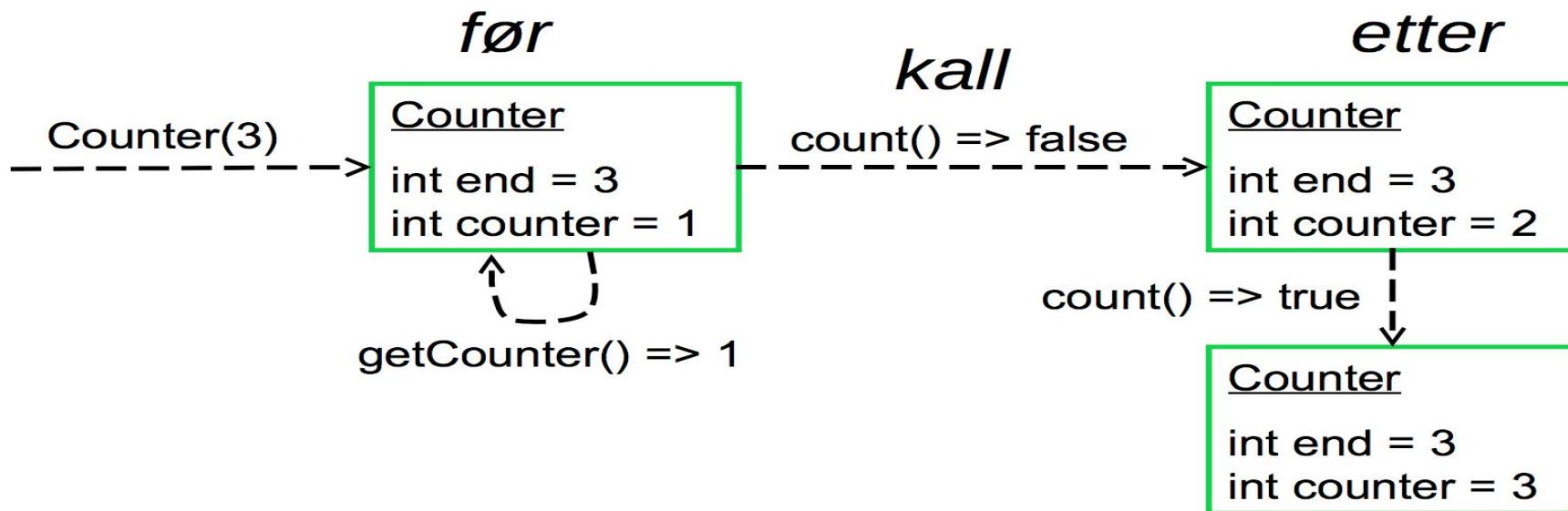
# Eksempel fra eksamen 2017

T

```
public void mer  
checkNot  
checkNot  
Table ta  
removeTa  
removeTa  
addTable  
}
```



# Nok et eksempel





Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>

# Collections og arrays

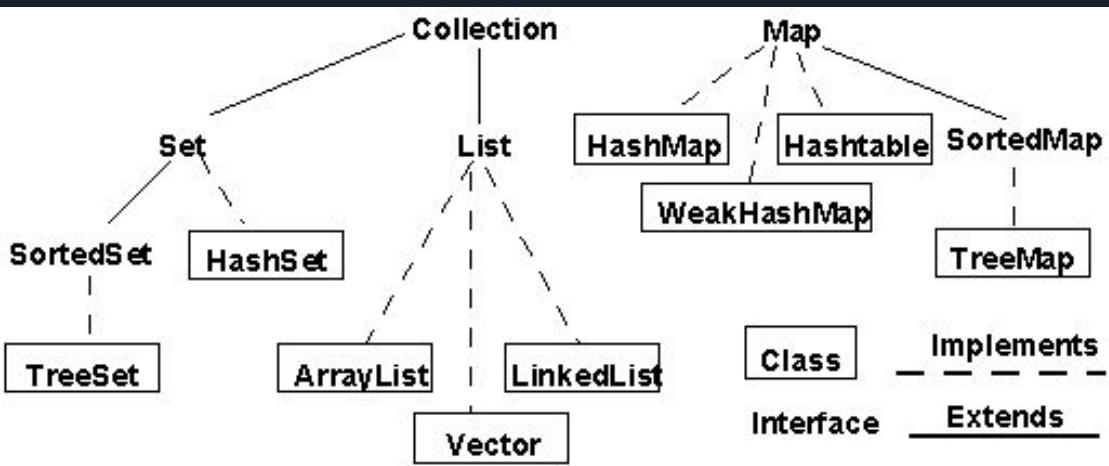
# Arrays

- Som lister i Python, men immutable
  - Dvs. kan ikke endre størrelse i etterkant
- Veldig primitiv i sammenligning med ArrayList
- I noen tilfeller er det enklere å bruke Array, i noen tilfeller ArrayList
- Kan konvertere fra array til ArrayList på denne måten:

```
String[] array = {"Honda", "Ford", "BMW"};
List<String> list = Arrays.asList(array);
```

# Collections og Lister

Vi bruker



- **Collection** hvis vi ikke ønsker å aksessere på index
  - Har metodene
    - **add(element)**
    - **remove(element)**
    - **contains(element)**
- **List** hvis vi ønsker å aksessere på index
  - Legger til metodene
    - **get(index)**
    - **remove(index)**
    - **indexOf(element)**



question ★

## Collections

```
Collection<Integer> list1 = new ArrayList<Integer>();  
List<Integer> list2 = new ArrayList<Integer>();
```

Hva er forskjellen på list1 og list2 i praksis?

- Dersom du trenger **indeks** -> Bruk List
- Hvis ikke -> Bruk Collection

# ArrayList

```
private List<Employee> employees = new ArrayList<>();
```

- Kjenn til disse metodene:
  - **add(element)**
  - **remove(element)**
  - **contains(element)**
  - **get(index)**
  - **remove(index)**
  - **indexOf(element)**
  - **size()**
  - **isEmpty()**

# Hashmap

- Samme prinsipp som dictionary i Python
- Kan gjøre noe oppgaver enklere
  - Prioriter å bli rutinert på ArrayList

```
// Create a HashMap object called capitalCities
HashMap<String, String> capitalCities = new HashMap<String, String>();

// Add keys and values (Country, City)
capitalCities.put("England", "London");
capitalCities.put("Germany", "Berlin");
capitalCities.put("Norway", "Oslo");
capitalCities.put("USA", "Washington DC");
```

# Viktig om Collections:

- En kan ikke ha primitiver i listen
  - Dvs: ikke int, double etc
- Bruker heller wrapper-klassen: Integer, Double
- Referanser - Lister i Java er bare pekere til objekter!



```
List<int> jegFårTrekkPåEksamens = new ArrayList<int>();
```



```
List<Integer> jegFungerer = new ArrayList<Integer>();
```

# Varargs

Kan ta inn 0 eller flere  
antall argumenter

```
public class Person3 {  
  
    private String name = "Ola Nordmann";  
    private int age = 0;  
    private double height = 0;  
    private List<String> hobbies = new ArrayList<String>();  
  
    // De tre punktumene (ellipsis på engelsk) etter String før hobbies markerer dette som en "vararg"  
    // På denne måten kan man ta inn flere Strings av gangen  
    // Disse strengene blir da på formatet Array  
    public Person3(String name, int age, double height, String... hobbies) {  
  
        this.name = name;  
        this.age = age;  
        this.height = height;  
  
        // Koden nedenfor bruker den statiske metoden asList i klassen Arrays for å gjøre om  
        // fra Array til en ArrayList  
        this.hobbies = Arrays.asList(hobbies);  
    }  
}
```

# Vi lager klassen **Doctor**



- Bruker varargs for å kunne legge til assistenter i konstruktør.



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>

Arv

# Testament



# Arv

Hva:

- En metode for å la **subklasser** få egenskaper fra en **superklasse**
- Gjenbrukbar kode
- Speiler virkelige verden

Hvorfor:

- Klasser deler egenskaper
- Vi ønsker å gjenbruke kode
- Forenkler mange operasjoner
- Lister med forskjellige typer elementer som kan ses på som en del av en overordnet “type”

# super

- samme som **this** bare at det refererer til superklassen (klassen som arves fra) i stedet.
- **super()**; gir oss konstruktøren til superklassen
  - Denne kan ha inputparametre på akkurat samme måte som en konstruktør!



```
public Meal(String name, String de  
super(name, description);  
this.courses = Arrays.asList(
```



# Grensesnitt / Interface

- Som klasser - men kan kun deklarere variabler og metoder uten å definere noe innhold.
  - Merk: variabler deklarert i grensesnitt blir *automatisk static final*
- Klasser som **implementerer** et grensesnitt må definere alle variablene og metodene i grensesnittet

# Grensesnitt

```
public interface Named {  
  
    public String getGivenName();  
    public void setGivenName(String name);  
  
    public String getFamilyName();  
    public void setFamilyName(String name);  
  
    public String getFullName();  
    public void setFullName(String name);  
}
```



# Grensesnitt implementert av en klasse

```
public interface Named {  
  
    public String getGivenName();  
    public void setGivenName(String name);  
  
    public String getFamilyName();  
    public void setFamilyName(String name);  
  
    public String getFullName();  
    public void setFullName(String name);  
}
```

```
public class Person1 implements Named {  
  
    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);  
    }  
}
```



# Abstrakte klasser

- Grensesnitt - med litt kode (Hvis man vil)
  - Blanding av superklasse og grensesnitt
- Abstrakte klasser kan ikke instansieres, men klasser som arver fra de kan.
- Abstrakte klasser brukes gjerne når vi har objekter som det ikke gir mening at står for seg selv



# Abstrakt klasse

```
public static void main(String[] args) {  
    Animal hey = new Animal("buddy", 1);  
}
```

```
public abstract class Animal {  
  
    private String name;  
    private int age;  
  
    public Animal(String name, int age) {  
        if(age < 0) {  
            throw new IllegalArgumentException("Invalid age");  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public abstract void eat(Food foodItem);  
}
```



Dog arver fra en abstrakt klasse og kan  
instansieres

```
public class Dog extends Animal {  
  
    public Dog(String name, int age) {  
        super(name, age);  
    }  
}
```

# Eksempel fra K2017

## Del 2 a) og b)

Course- og Meal-klassene har en del felles egenskaper, som kan samles i en felles superklasse kalt **MenuItem**. Skriv koden til **MenuItem**.

```
public class Course {  
  
    // fordeler/ulemper med en slik implementasjon av en  
    // klasse med data som ikke skal kunne endres  
  
    public final String name, description;  
  
    public Course(String name, String description) {  
        // auto-generert av Eclipse, hva betyr super() her?  
        super();  
        this.name = name;  
        this.description = description;  
    }  
}
```

```
public class Meal implements Iterable<Course> {  
  
    private final String name, description;  
  
    private Collection<Course> courses;  
  
    **  
    * Initializes the Meal with the provided name, description and courses  
    * @param name  
    * @param description  
    * @param courses  
    */  
    // vis med et eksempel hvordan denne konstruktøren kalles  
    public Meal(String name, String description, Course[] courses)  
    super();  
    this.name = name;  
    this.description = description;  
    // hva ville skjedd om en at lot this, her:  
    // - feilmelding ved kompilering (rød strek i Eclipse)  
    // - varsel om mulig feil (gul strek i Eclipse)  
    // - kræsj ved kjøring  
    // - logisk feil ved kjøring  
    // - det virker som det skal  
    this.courses = Arrays.asList(courses);  
}  
  
public String getName() {  
    return name;  
}  
  
public String getDescription() {  
    return description;  
}  
}
```

# Den abstrakte klassen MenuItem kan se slik ut

```
public abstract class MenuItem {  
    private final String name, description;  
  
    public MenuItem(String name, String description) {  
        this.name = name;  
        this.description = description;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
}
```



Meal og Course  
kan da se slik  
ut når de arver  
fra MenuItem

```
/*
public class Meal extends MenuItem implements Iterable<Course> {

    private Collection<Course> courses;

    /**
     * Initializes the Meal with the provided name, description and
     * @param name
     * @param description
     * @param courses
     */
    // vis med et eksempel hvordan denne konstruktøren kalles
    public Meal(String name, String description, Course[] courses) {
        super(name, description);
        this.courses = Arrays.asList(courses);
    }

    //
```

```
public class Course extends MenuItem {

    public Course(String name, String description) {
        super(name, description);
    }
}
```

Vi lager grensesnittet  
**Employee** og gjør  
**Person** abstrakt

- **Doctor** og **Nurse** skal arve fra den abstrakte klassen **Person**
- De skal også implementere grensesnittet **Employee**



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>

A hooded figure sits at a desk in a dimly lit, high-tech control room, facing away from the camera. They are looking at a large computer monitor displaying the text "Avanserte grensesnitt". The room is filled with multiple screens showing complex data and code, and the atmosphere is filled with blue and green glowing lights.

Avanserte  
grensesnitt





# Iterator og Iterable

- Iterable = Markerer klassen som at den kan itereres over **på samme måte som en liste**
- Iterator = Et objekt som lar oss iterere over en liste/collection.
  - Du bruker implisitt en iterator når du bruker en foreach-løkke
- Ofte har vi klasser som inneholder lister og som skal implementere Iterable-grensesnittet.



# Iterable vs. iterator

- Hvis man skal implementere iterable må man implementere følgende metode:
  - **iterator()**
- Hvis man skal implementere iterator så må man implementere følgende metoder:
  - **hasNext()**
  - **next()**



# Eksempel fra K2018 Del 1 d)

## Oppgave d) - Dicelterator (6 poeng)

Lag en implementasjon av `Iterator<Integer>` kalt **Dicelterator**, slik at den kan brukes slik det er vist i **Dice** sin `iterator()`-metode. Du kan anta at `valueCounters`-feltet i **Dice** er synlig i **Dicelterator**-klassen.

› LF

# LF

Her må man huske å tenke på Dice som en liste

`next()` vil inkrementere dieNum slik at vi gradvis går gjennom listen som ligger i Dice-klassen

MERK at vi her skal implementere en Iterator og ikke Iterable

```
public class DiceIterator implements Iterator<Integer> {  
    private final Dice dice;  
    private int dieNum = 0;  
  
    public DiceIterator(final Dice dice) {  
        this.dice = dice;  
    }  
  
    @Override  
    public boolean hasNext() {  
        return dieNum < dice.getDieCount();  
    }  
  
    @Override  
    public Integer next() {  
        final int value = dice.getDieValue(dieNum);  
        dieNum++;  
        return value;  
    }  
}
```

Vi lager klassen **Patient**  
og gjør denne iterable

- Skal kunne iterere over en liste med  
Symptoms
- Skal også implementere metoden  
**receiveMedication()**



# Comparator og Comparable

- Hva er forskjellen?
  - Comparable: Grensesnitt som klasser kan implementere for å sammenligne objekter av denne typen
  - Comparator -> Et grensesnitt som kan implementeres som en klasse. Denne klassen kan sammenligne to objekter av en annen type / klasse.

# Comparator

```
public class Student {  
    private final int id;  
    private final String name;  
    private final int age;  
  
    public Student(int id, String name, int age) {  
        this.id = id;  
        this.name = name;  
        this.age = age;  
    }  
  
    public int getId() { return id; }  
    public String getName() { return name; }  
    public int getAge() { return age; }  
}
```

with:

```
public class AgeComparator implements Comparator<Student> {  
    public int compare(Student s1, Student s2) {  
        if (s1.getAge() == s2.getAge()) {  
            return 0;  
        } else {  
            return s1.getAge() < s2.getAge() ? -1 : 1;  
        }  
    }  
}
```

# Comparable

```
// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }
}
```

# Comparable Vs Comparator

Comparator	Comparable
Int compare(Object o1, Object o2)	Int CompareTo(object o)
Returnerer: Negativt tall om o1<o2 0 hvis o1=o2 Positivt tall hvis o1>o2	Samme som Comparator, med this-objektet som o1.
Ny klasse separat fra annen kode	Modifisere eksisterende klasse
Sammenligne på mange måter	Sammenligne på en måte
	Finnes innebygd i mange klasser (String f.eks)



# Når bruker vi det?

- Når vi skal sortere en liste
- Hvis vi bruker ***comparable*** kan vi bare bruke:
  - **Arrays.sort(object)**
- Hvis vi bruker ***comparator*** må vi også definere hvilken ***comparator*** vi skal bruke:
  - **Arrays.sort(object, comparator)**

# Hva tror du denne koden vil gjøre?

Fruit-klassen:

```
public class Fruit{  
  
    private String fruitName;  
    private String fruitDesc;  
    private int quantity;
```

main-metoden:

```
Fruit[] fruits = new Fruit[4];  
  
Fruit pineappale = new Fruit("Pineapple", "Pineapple description", 70);  
Fruit apple = new Fruit("Apple", "Apple description", 100);  
Fruit orange = new Fruit("Orange", "Orange description", 80);  
Fruit banana = new Fruit("Banana", "Banana description", 90);  
  
fruits[0]=pineappale;  
fruits[1]=apple;  
fruits[2]=orange;  
fruits[3]=banana;  
  
Arrays.sort(fruits);
```

# Hva tror du denne koden vil gjøre?

Fruit-klassen:

```
public class Fruit{  
  
    private String fruitName;  
    private String fruitDesc;
```

main-metoden:

```
Fruit[] fruits = new Fruit[4];  
  
Fruit pineappale = new Fruit("Pineapple", "Pineapple description", 70);  
Fruit apple = new Fruit("Apple", "Apple description", 100);  
Fruit orange = new Fruit("Orange", "Orange description", 80);
```

Exception in thread "main" java.lang.ClassCastException:  
com.mkyong.common.Fruit cannot be cast to java.lang.Comparable  
at java.util.Arrays.mergeSort(Unknown Source)  
at java.util.Arrays.sort(Unknown Source)

Vi gjør klassen  
**Medication**  
comparable

- Må implementere  
**compareTo()**-funksjonen



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>



# Funksjonelle grensesnitt

- Funksjonelt betyr egentlig bare her at man behandler **funksjoner** som et objekt
  - Man kan dermed bruke en funksjon som **inputparameter** i andre funksjoner/metoder igjen.
- Funksjonelle grensesnitt er litt tricky, men det er ofte samme type oppgaver som går igjen



## Eksempel fra H2018 Del 5 a)

```
public interface DiceScorer {  
    Dice getScore(Dice dice);  
}
```

- 1) har kun én abstrakt metode og
- 2) er ment å være primærfunksjonen til klassen som implementerer den.



# Slik kan en klasse som implementerer DiceScorer se ut

ThreeOrMoreOfAKind har tre metoder, men kun én av de er public.

Vi kunne implementert samme klasse med kun én metode

Det er heller ingen intern tilstand

Ergo er DiceScorer funksjonelt

```
public class ThreeOrMoreOfAKind implements DiceScorer {

    // hjelpekode som beregner poengsum for count antall av value
    private int getScore(final int value, int count) {
        int score = value * 100;
        while (count > 3) {
            score = score * 2;
            count--;
        }
        return score;
        // Samme som: return value * 100 * (int) Math.pow(count - 3, 2);
    }

    // går gjennom hver mulige verdi 1-6 og finner den med nok like som gir flest poeng
    protected int getBestValue(final Dice dice) {
        int value = 0, max = 0;
        for (int i = 1; i <= 6; i++) {
            int count = dice.getValueCount(i);
            int score = getScore(i, count);
            if (count >= 3 && score > max) {
                value = i;
                max = score;
            }
        }
        return value;
    }

    @Override
    // finner den beste og lager et nytt Dice-objekt med riktig antall terningverdier og poengsum
    public Dice getScore(final Dice dice) {
        int best = getBestValue(dice);
        if (best == 0) {
            return null;
        }
        int count = dice.getValueCount(best);
        Collection<Integer> dieValues = new ArrayList<Integer>(count);
        for (int i = 0; i < count; i++) {
            dieValues.add(best);
        }
        return new Dice(dieValues, getScore(best, count));
    }
}
```



Men hvorfor er Comparable bare “teknisk  
sett” funksjonelt?

```
public interface Comparable<T> {  
  
    public int compareTo(T o);  
  
}
```



# Slik kan en klasse som implementerer Comparable se ut

Gir ikke mening å implementere Comparable uten noen intern tilstand eller andre metoder

```
public class Medication implements Comparable<Medication> {

    private String name;
    private double price = Double.NaN;

    public Medication(String name) {
        this.name = name;
    }

    public Medication(String name, double price) {
        this.name = name;
        if(price < 0) throw new IllegalArgumentException("Invalid price");
        this.price = price;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    // Comparable-grensesnittet:
    @Override
    public int compareTo(Medication medication) {
        return Double.compare(this.getPrice(), medication.getPrice());
    }

    @Override
    public String toString() {
        return this.getName();
    }
}
```



# Lambda-uttrykk

- Strengt tatt bare en annen måte å skrive funksjoner på.
- Slipper å definere typer, i motsetning til alle andre steder
- Vi bruker disse for å **instansiere** funksjonelle grensesnitt



```
// Lambdauttrykk:  
  
(input -> doSomethingWith(input))
```

```
// Vanlig metode:  
  
public Object funksjonsNavn(Object input) {  
    // Code  
    return doSomethingWith(input);  
}
```



```
// Lambdauttrykk:
```

```
(input -> doSomethingWith(input))
```

```
// Vanlig metode:
```

```
public Object funksjonsNavn(Object input) {  
    // Code  
    return doSomethingWith(input);  
}
```



# Streams

- Dette er **ikke** en liste
  - Likevel, en enkel måte å forstå hva det gjør er ved å bare se på det som en spesiell type liste
- Streams lar oss forkorte veldig store deler av koden.
- Brukes i forbindelse med funksjonelle grensesnitt



# Eksempel, med animasjon!

```
people.stream()  
    .filter(person -> person instanceof Human)  
    .collect(Collectors.toList());
```

```
people.stream()
```

```
.filter(person -> person instanceof Human)  
.collect(Collectors.toList());
```

```
filter(p -> (p instanceof Human))
```

```
people.stream()
```

```
collect(Collectors.toList())
```

Vi lager metoden  
**getNurseList()** i  
**Doctor**

- Metoden skal filtrere ut alle  
**Employee**-objekt som ikke er av typen  
Nurse og returnere en liste med objekter  
av kun denne typen

## Method reference i streams

Hei, gjør kont 2017 og i del 1 under table klassen skal vi skrive metoden getPrice(). Jeg har lyst til å gjøre den med streams og har gjort dette:

```
public double getPrice() throws IllegalStateException {
    try{
        double price = 0.0;
        price += courses.stream().mapToDouble(c -> menu.getPrice(c)).sum();
        price += meals.stream().mapToDouble(m -> menu.getPrice(m)).sum();
        return price;
    } catch(IllegalArgumentException e) {
        throw new IllegalStateException(e);
    }
}
```

IntelliJ "Klager" er på at "c -> menu.getPrice(c)" kan skrives med method reference, så prøvde dette:

```
public double getPrice() throws IllegalStateException {
    try{
        double price = 0.0;
        price += courses.stream().mapToDouble(menu::getPrice).sum();
        price += meals.stream().mapToDouble(menu::getPrice).sum();
        return price;
    } catch(IllegalArgumentException e) {
        throw new IllegalStateException(e);
    }
}
```

Han klager ikke lenger, men er usikker på om det er riktig måte å gjøre det på? Fasiten har gjort det slik:



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>

# Standardteknikker





# Delegering

- Vi gir bare oppgaven videre til et annet objekt.

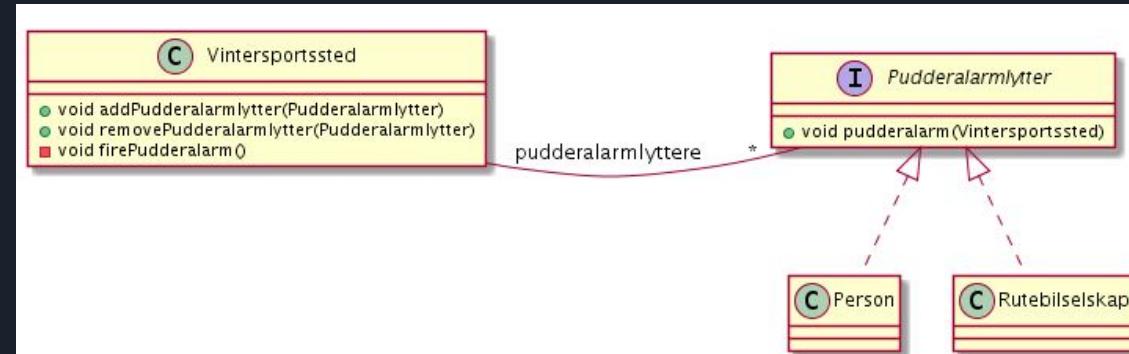
```
@Override  
public void printDocument(String document) {  
    nextDelegate++ ;  
    getTaskDelegate().printDocument(document);  
}
```

# Vi implementerer metoden **medicate()**

- Både **Nurse** og **Doctor** skal implementere denne metoden, men **Doctor** skal deletere oppgaven videre til en tilfeldig **Nurse**

# Observable - Observator

- Dette er egentlig bare en teknikk, ikke noe som er innebygd i Java
- Teknikk for å kunne lytte på tilstander hos andre.
- Navnene blir fort kronglete, men prøv å skill mellom grensesnitt og faktisk implementasjon





# Observatør-Observervert: hva vi trenger

**Observervert:**

- Metode for å legge til lyttere
- Metode for å fjerne lyttere
- Metode for å si ifra til lyttere

**Observatør: (lytter)**

- Metode for å ta imot informasjonen om endret tilstand

Vi lager **WaitingList** og gjør denne observerbar

- **WaitingList** skal kunne observeres av **Doctor**-objekter

The background of the image is divided into four horizontal sections of equal width. From top to bottom, the colors are red, blue, yellow, and green. Each section contains a stylized, leafy silhouette in a color matching the background. Overlaid on this background is the word "Kahoot!" in a large, white, sans-serif font. The exclamation mark is particularly prominent, being much larger than the other letters and having a thick white outline.

Kahoot!

A problem has been detected and Windows has been shut down to prevent damage to your computer.

#### DRIVER\_IRQL\_NOT\_LESS\_THAN\_OR\_EQUAL\_TO

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

#### Technical information:

\*\*\* STOP: 0x000000D1 (0x00000000, 0x00000000)

# I/O - lese og skrive til fil

- Vi ønsker å kunne lagre tilstanden til et program uten at det må kjøres hele tiden/uten at en må ha all informasjonen i minnet.
- Finnes utallige måter å gjøre I/O på.
- Prøv å finne hovedsakelig én måte som funker for deg
- Lag oppå lag oppå lag.





# BufferedWriter og BufferedReader

```
BufferedWriter writer = new BufferedWriter(new FileWriter(path));
writer.write("Hva enn du ønsker å skrive til fil her...");
writer.close();
```

```
BufferedReader reader = new BufferedReader(new FileReader(path));
String currentLine = reader.readLine();
// Her kan du lese flere linjer med reader.readLine()
reader.close();
```

## Eksempel fra eksamen V2016 Del c)

```
public void save(OutputStream out) throws IOException {  
    PrintWriter pw = new PrintWriter(out);  
    pw.println("# all persons");  
    for (Person person : members) {  
        pw.print(person.getGender());  
        pw.print(" ");  
    }  
}
```



## TLDR; I/O

- Bruk Javadocs til dette!
- Det er så mange måter å gjøre I/O på, så skulle du få en helt ny case på eksamen så er det bare å bruke javadocs og se hvilke metoder og konstruktører de forskjellige klassene har.

# Input-Output

## Output -PrintWriter:

- Trenger ingen underliggende klasser, kan skrive direkte til et filnavn
- Printwriter pw = new PrintWriter(String filename)
  - Greit å vite
  - pw = new PrintWriter(File file)
  - pw = new  
PrintWriter(OutputStream out)
- Kjekke metoder en får med
  - pw.println() - automatisk “\n”
  - pw.print()

## Inputstream - Scanner

- Trenger en underliggende klasse
- Scanner scanner = new Scanner(new File(String filename))
  - scanner = new  
Scanner(InputStream in)
- Nyttige metoder
  - hasNext() /hasNextLine()
  - nextLine();
  - NB: Kan være greit å holde seg unna nextInt() - metodene i hvertfall om en kombinerer tekst og tall.



# Unntak / Exceptions

- Finnes flere kategorier med exceptions.
- Checked exceptions (f.eks. fra I/O) må håndteres (med try-catch)
  - **IOException**
  - **FileNotFoundException** (er bare en underklasse av **IOException**)
- Unchecked - må ikke håndteres - (vanskeligere å recovere fra)
  - **IllegalArgumentException**
  - **IllegalStateException**
  - **IndexOutOfBoundsException**

# Unntakshåndtering

```
public void tryCatch() {  
    try {  
        int horribleMistake = 5/0;  
    }  
    catch (ArithmetricException e) {  
        System.out.println("Wuuups");  
    }  
}
```

Throws

Sender feilen “et hakk opp i systemet” - neste metode som bruker denne må spesifisere hvordan det skal løses

```
public void throwsMethod() throws ArithmetricException {  
    int horribleMistake = 5/0;  
}
```

# Vi lager metodene **save** og **load** i **PatientLog**

- Begge metodene skal bruke **throws**.
- Vi bruker **BufferedReader** og **-Writer** for enkelhets skyld
- Bruker **Streams**, men dette kan også gjøres med vanlige løkker



Minner om...

Still spørsmål på Sembly: <https://s.ntnu.no/java-sporsmal>

Hent slides fra Drive: <https://s.ntnu.no/java-slides>

Koden ligger på Github:

<https://github.com/Magssch/Eksamensforelesning-Java-2019>



# Testing / JUnit

- Må kjenne til Try/Catch i tillegg til to JUnit-metoder:
  - **fail("Feilmelding")**
  - **assertEquals(arg1, arg2)**
    - Andre Assert-metoder er basically bare spesialversjoner av denne



## Eksamensoppgave V2018 Del 5 oppgave c)

Skriv en eller flere test-metoder for **Dice** sin **valueOf**-metode. Du kan anta det finnes en metode kalt **assertDieValues** som sjekker terningverdiene til en **Dice**:

**assertDieValues(dice, 1, 2, 3)** utløser assert-unntak hvis dice ikke har terningverdiene 1, 2, 3 og bare disse.

LF

- Bruker try-catch for å teste at metoden kaster IllegalArgumentException
- Må tenkte omvendt fra vanlig tenkemåte

```
@Test
public void testValueOf() {
    Dice dice1 = Dice.valueOf("[1, 1, 2]");
    assertDieValues(dice1, 1, 1, 2);
    Assert.assertEquals(-1, dice1.getScore());

    Dice dice2 = Dice.valueOf("[1, 1, 2] = 200");
    assertDieValues(dice2, 1, 1, 2);
    Assert.assertEquals(200, dice2.getScore());

    try {
        Dice.valueOf("1, 5, 2 = 200");
        Assert.fail();
    } catch (final Exception e) {
        Assert.assertTrue(e instanceof IllegalArgumentException);
    }

    try {
        Dice.valueOf("1, x, 2 = 200");
        Assert.fail();
    } catch (final Exception e) {
        Assert.assertTrue(e instanceof IllegalArgumentException);
    }

    try {
        Dice.valueOf("1, 5, 2 = x");
        Assert.fail();
    } catch (final Exception e) {
        Assert.assertTrue(e instanceof IllegalArgumentException);
    }
}
```



# Annet eksempel fra eksamen V2016

```
import junit.framework.TestCase;

public class PersonTest extends TestCase {

    public void testAddChild() {
        Gender female = Gender.valueOf("female"), male = Gender.valueOf("male");
        Person mother = new Person("Chris"); mother.setGender(female);
        Person father1 = new Person("Pat"); father1.setGender(male);
        Person father2 = new Person("Alex"); father2.setGender(male);
        Person child = new Person("Jean");

        mother.addChild(child);
        assertEquals(1, mother.getChildCount());
        assertTrue(mother.hasChild(child));
        assertEquals(mother, child.getMother());
        mother.addChild(child);
        assertEquals(1, mother.getChildCount());

        father1.addChild(child);
        assertTrue(father1.hasChild(child));
        assertEquals(father1, child.getFather());

        father2.addChild(child);
        assertFalse(father1.hasChild(child));
        assertTrue(father2.hasChild(child));
        assertEquals(father2, child.getFather());

        father2.setGender(female);
        father2.addChild(child);
        // ???
        child.addChild(father2);
        // ???
    }
}
```



# JavaFX

- Hva du må kunne på eksamen:
  - *Sammenhengen mellom FXML-elements (tags) og JavaFX-klasser*
  - *Sammenhengen mellom FXML og kontrollerklassen (navn, felt og metoder)*

```
<HBox xmlns:fx="http://javafx.com/fxml/1" fx:controller="counter.CounterController">
    <Text fx:id="counterOutput" text="Current counter value: 0"/>
    <Button text="Count" onAction="#handleCountAction"/>
</HBox>
```

```
public class CounterController {

    Counter counter;

    @FXML
    Text counterOutput;

    @FXML
    void initialize() {
        counter = new Counter(5);
    }

    @FXML
    void handleCountAction() {
        counter.count();
        counterOutput.setText("Current counter value: " + counter.getCounter());
    }
}
```



## Obs!

- Husk **@FXML**-tags
- For å hente ut fra / sette verdier i **TextField** må man bruke:
  - `.getText()`
  - `.setText("Teksten din her")`
- Gjelder også for **Labels**



# Effektiv bruk av Javadoc

<https://javadoc.it.ntnu.no/docs/api/index.html>

# Protips videre

- Rimelig kontroll på pensum? - Lær deg **Streams**.
- Fokuser på å lære deg én metode som fungerer på flere tilfeller
  - Eks: Filhåndtering og lister.
- Sett deg inn i hva de forskjellige objektene representerer. Når vi har en iterator - hva er dette? Hva er egentlig en ArrayList? osv.
- Øvingsoppgaver:
  - Øving 8 - The Office
  - Øving 6 - BinaryComputingIterator (om iteratorer og funksjonelle grensesnitt)





# Tilbakemelding ønskes!

<https://s.ntnu.no/java-sporsmal>

- Trykk på give advice



Dette klarer du!

