NTNU

Kunnskap for en bedre verden

Øvingsforelesning 7

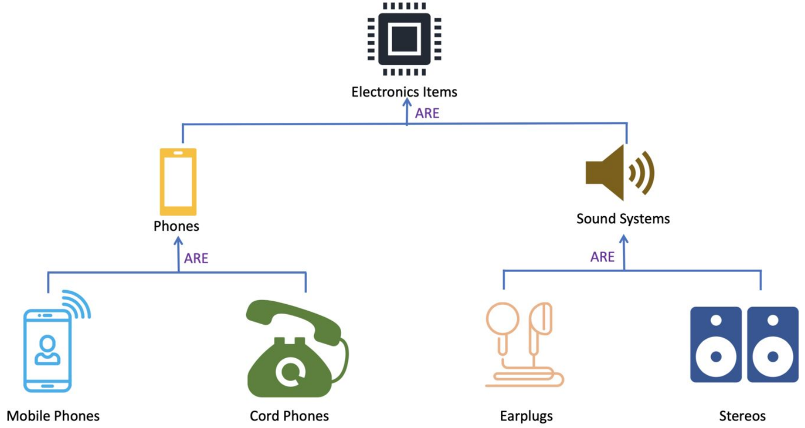
TDT4100 Objektorientert programmering

15.03.2022

Jostein Hjortland Tysse
Vitenskapelig assistent, TDT4100
jostein.h.tysse@ntnu.no

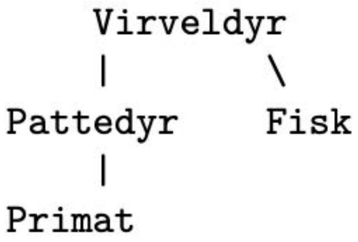
Arv

Teori


Electronics Items
ARE
Phones
Sound Systems
ARE
ARE
Mobile Phones
Cord Phones
Earplugs
Stereos

Arv

- Arv er en **mekanisme** for å bygge videre på eksisterende klasser.
- En **subklasse arver** superklassen (klassen den arver fra) sine egenskaper (felter, metoder, etc.) og kan legge til flere. På denne måten fungerer subklassen som en **spesialisering** av superklassen.
- Når arv brukes riktig, kan den gjøre koden **ryddigere** og enklere å **gjenbruke**.



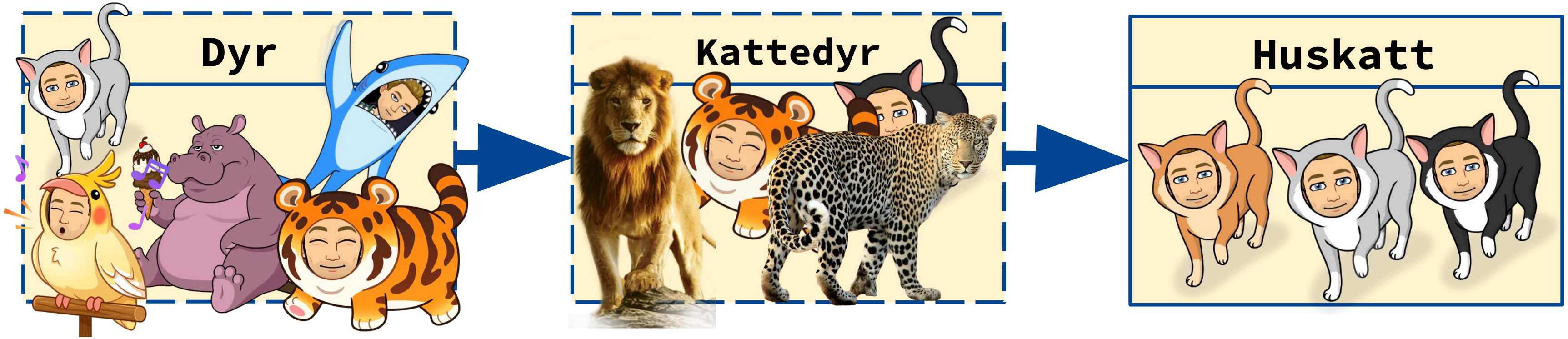
```
     Virveldyr
     |        \
Pattedyr     Fisk
     |
Primat
```

Arv

Visualisering

Mest generell

Mest spesialisert



Dyr

Kattedyr

Huskatt

Arv vs. Interface

- **Interface** definerer kun **“fasaden”** ut mot verden, hvilke **metoder** som må være med, ikke **hvordan** de implementeres, ei heller hvordan intern data (felter) lagres / håndteres.

- **Arv** lar oss i motsetning definere at objekter av forskjellige typer kan ha mye av den samme **implementasjonen** men også bygge videre på hverandre, samtidig som de kan brukes om hverandre (som med interface)

Subklasse og Superklasse

Subklasse: klassen som arver felter, metoder (inkludert konstruktører) og interface fra en annen klasse.

Superklasse: Klassen som det arves fra. For at subklassen skal ha tilgang til felter og metoder fra superklassen må de være deklarert med **`public`** eller **`protected`**. Hvis de er satt til **`private`** vil de være **usynlig** for subklassen.

- Unngå package-private (uten synlighetsmodifikator)

```
package com.example;

public class MyClass {
    protected int myProtectedField;

    protected void myProtectedMethod() {
        // do something
    }

    private bool myPrivateField; // not accessible for other classes

    int myNumber; // no modifier, avoid this

}


public class AnotherClass extends MyClass {
    public void accessProtectedMember() {
        myProtectedField = 10; // OK, accessed from subclass
        myProtectedMethod();  // OK, accessed from subclass
    }
}
```

Synlighetsmodifikatorer

- **Ingen modifikator** gjør det synlig for alle klasser i samme pakke, men da er det vanskeligere å håndheve innkapsling.
- **`private`** er synlig kun inne i selve klassen det er deklarert.
- **`public`** er synlig “over alt”.
- **`protected`** brukes i forbindelse med arv som vi skal se videre på.

Synlighetsmodifikatorer

- **Ingen modifikator** gjør det synlig for alle klasser i samme pakke, men det da er det vanskeligere å håndheve innkapsling.
- **`private`** er synlig kun inne i selve klassen det er deklarert.
- **`public`** er synlig "over alt".
- **`protected`** brukes i forbindelse med arv, som vi skal se videre på nå.

Synlighetsmodifikatorer

`protected`

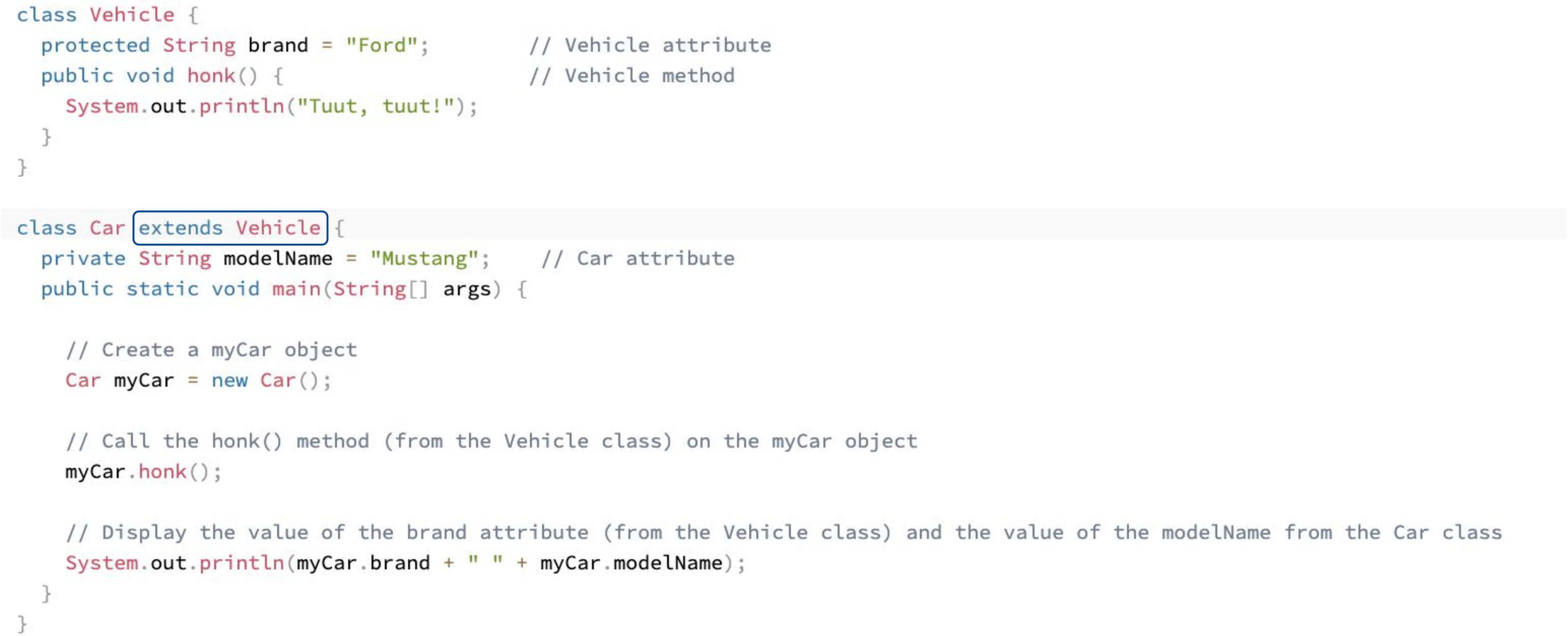
- Med denne synlighetsmodifikatoren vil kun **subklasser** og andre **klasser i pakken** ha **tilgang** til feltet eller metoden
    - Det siste her kan dere se litt bort ifra ifb. med dette emnet.

- Vanligvis bruker vi denne synlighetsmodifikatoren kun ifb. **arv** i dette emnet

- Lar deg f.eks. deklarere felter i superklassen og få tilgang til og modifisere disse fra subklassen, uten å måtte eksponere feltet for omverdenen med **`public`**

Arv

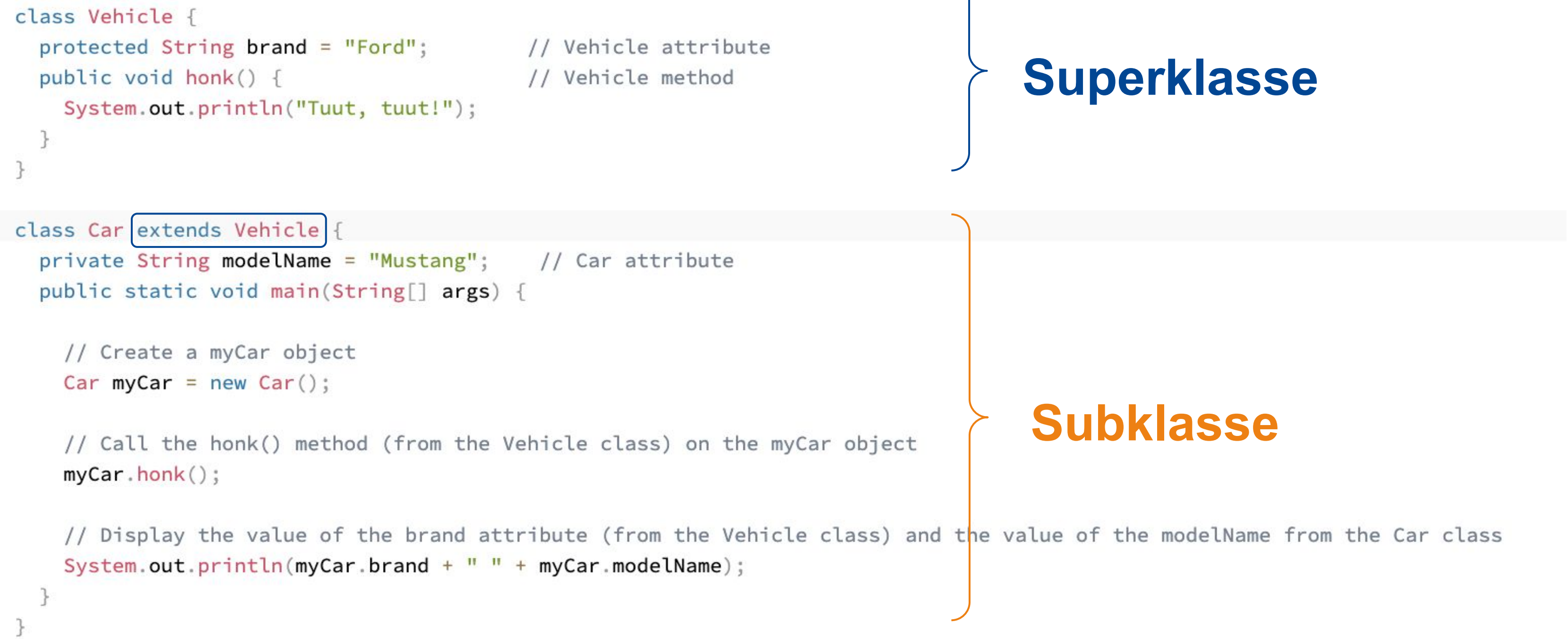
`extends` er nøkkelordet som brukes for å arve fra en klasse.



```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Arv

`extends` er nøkkelordet som brukes for å arve fra en klasse.



```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Superklasse

Subklasse

Hvorfor bruke arv?

- **Intuitiv organisering**
  Klassehierarkier vil ofte representere et naturlig/intuitiv organisering av fenomener i verden - som for eksempel dyrearter.

- **Forenkling av kode**
  Arv forenkler strukturering av komplekse problemer og programmer (mye av det samme vi bruker interface til).

- **Gjenbruk av kode**
  Lar oss lage subklasser som kan redefinere en eller flere metoder, men fortsatt bygge videre på den grunnleggende logikken i en superklasse

`@Override`

- **Overriding** eller **overstyring** er en egenskap som lar en subklasse spesifisere sin **egen implementasjon** av en metode som **allerede** er spesifisert i superklassen eller et interface og “gjemme” den som er definert tidligere.

- **Felter** kan også **overstyres i** subklassen ved å definere et felt med samme navn og type.

- Ved hjelp av nøkkelordet **`super`** kan man få tilgang til metoder man har overstyrt eller felter man har “gjemt”.

`super`

- **`super`**-nøkkelordet er en referansevariabel som lar oss referere til **superklassen** på samme måte som at **`this`** refererer til det nåværende objektet.
    - Eksempelvis ved bruk av **`super.metodeNavn()`** eller **`super.feltNavn`** aksesserer man metodene eller feltene som tilhører superklassen.

- **`super(<parametere>);`** referer til **konstruktøren** i superklassen. Denne kalles i konstruktøren til subklassen og lar oss bygge videre på superklassen sin konstruktør. Man trenger ikke å bruke **`@Override`** på konstruktøren i subklassen fordi den har et annet navn enn den i superklassen.

```
public class Dog extends Animal {

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

Kjekt å vite om arv

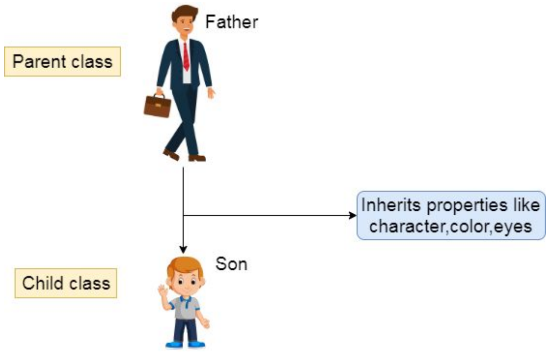
- **Alle** klasser i Java (også de vi lager selv) arver implisitt fra klassen **`Object`**

- Eksempel på dette i praksis er når vi lager en **`toString()`**-metode som overstyrer den ferdigdefinerte **`toString()`**-metoden i klassen **`Object`** og må skrive ***`@Override`*** over denne

- Det er derfor det går an å definere f.eks. **`ArrayList<Object>`** som gir oss en liste som kan inneholde **alle** typer objekter.
  - Dette vil vi dog vanligvis unngå

Praktisk oppgaveløsning

Vi skal nå gjøre noen oppgaver der vi bruker arv!

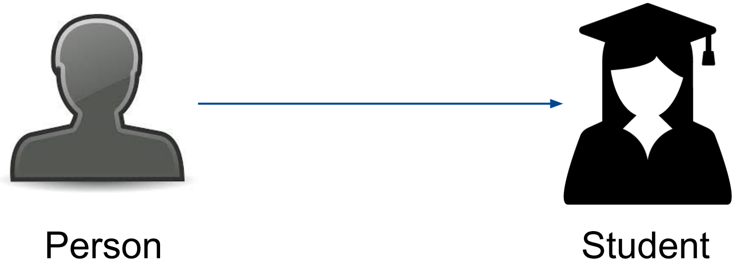


Oppgave 1: Person -> Student

Tema: arv

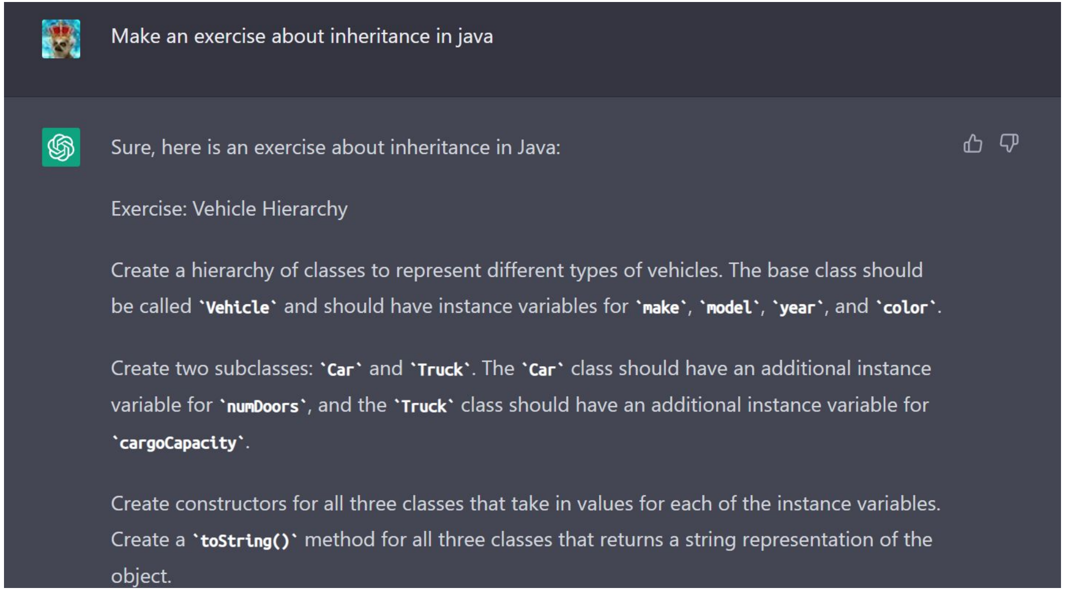
Lag to klasser, **Person** og **Student.**

- **Person**-klassen skal ha to attributter, **navn** og **alder.**
- **Student**-klassen skal arve fra **Person,** og skal i tillegg ha en **studentId**-attributt.
- Lag passende konstruktører, get- og set-metoder.



Oppgave 2: Vehicle

Vi spør Chat GPT om å lage en oppgave:



Make an exercise about inheritance in java

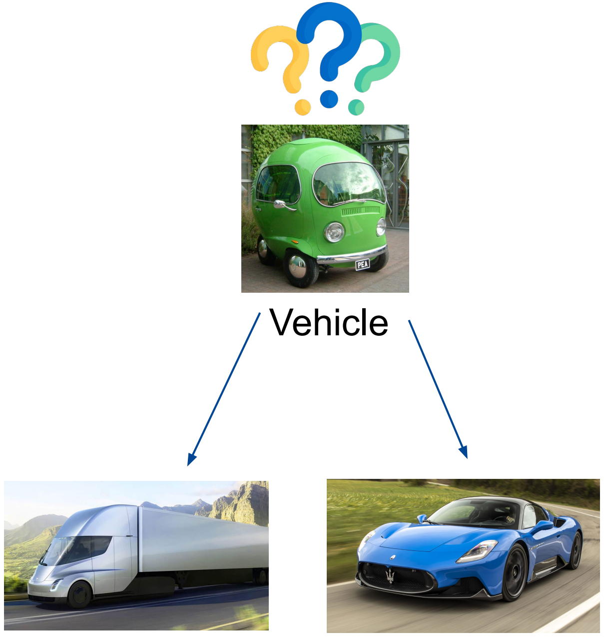
Sure, here is an exercise about inheritance in Java:

Exercise: Vehicle Hierarchy

Create a hierarchy of classes to represent different types of vehicles. The base class should be called `Vehicle` and should have instance variables for `make`, `model`, `year`, and `color`.

Create two subclasses: `Car` and `Truck`. The `Car` class should have an additional instance variable for `numDoors`, and the `Truck` class should have an additional instance variable for `cargoCapacity`.

Create constructors for all three classes that take in values for each of the instance variables. Create a `toString()` method for all three classes that returns a string representation of the object.



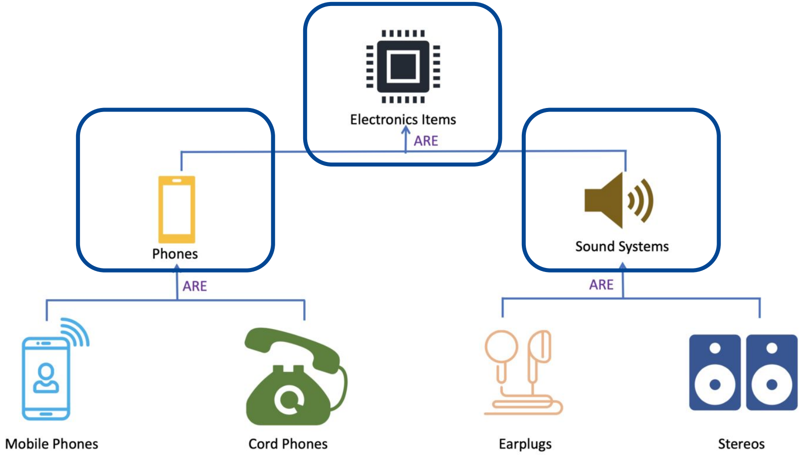
Vehicle

Truck

Car

Abstrakte klasser

Teori



Abstrakte klasser

En **abstrakt klasse** er en klasse som er deklarert **abstrakt** - det vil si at den **ikke** kan instansieres som objekter. En abstrakt klasse kan dermed kun benyttes ved at at en subklasse **arver fra den abstrakte klassen**.

- Abstrakte klasser **kan** (men må ikke) inneholde **ordinære metoder** og **felter** som har en implementasjon (som andre klasser) og **abstrakte metoder**.
- En **abstrakt metode** er en metode som er deklarert **uten en implementasjon**, som vi tidligere har sett når vi har laget interface.
- En klasse som har **abstrakte metoder** må deklareres abstrakt.
- En **subklasse** som arver en abstrakt klasse lager ofte en implementasjon av de abstrakte metodene. Hvis den derimot ikke gjør det, må subklassen også deklareres som abstrakt.

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

Abstrakte klasser vs. Interface

- Abstrakte klasser er på mange måter en **mellomting** mellom klasser og interface.

- Man kan ikke instansiere abstrakte klasser, og man kan lage metoder **både med og uten** implementasjon

- Abstrakte klasser kan ha felter som ikke er static og final, og definere public, protected, og private metoder (for interface er alle felter automatisk public, static og final, og alle metoder public).

- Man kan kun arve fra én abstrakt klasse, i motsetning til interface som man kan implementere **så mange man vil** av.

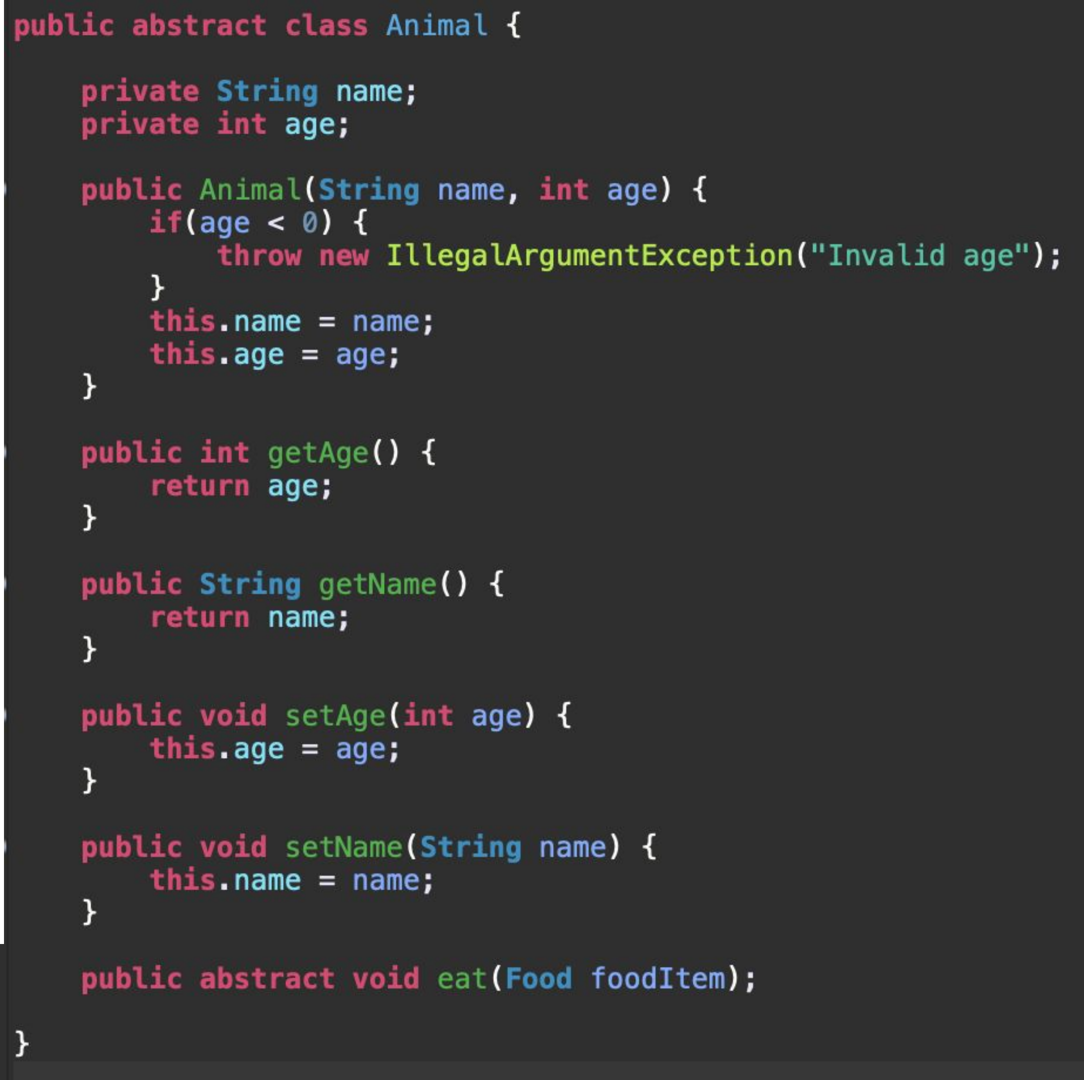
Abstrakte klasser

Når bruker man det?

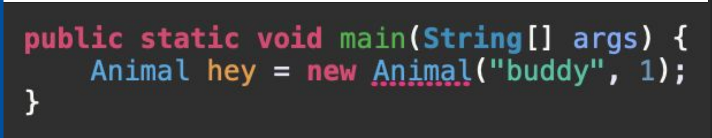
- Nyttig i et klassehierarki hvor det er **meningsløst** å ha instanser av akkurat denne klassen.
    - Eksempelvis vil en **`Animal`**-klasse ikke gi mening å kunne instansiere i seg selv, men det er mye som kan generaliseres fra forskjellige typer dyr
    - Det finnes ingenting som eksisterer som kun **`Animal`** den virkelige verden, det finnes riktignok “objekter” som hunder (**`Dog`**), katter (**`Cat`**) som **`extends Animal`** og kan klassifiseres som dette men som samtidig er en mer spesifikk art
- Når klassen er **ufullstendig**, ved at en eller metoder ikke er implementert og i stedet settes til å være abstrakte

Abstrakte klasser

Eksempel



```java
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);

}
```

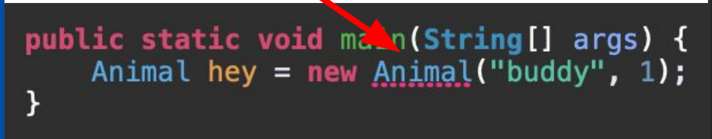


```java
public static void main(String[] args) {
    Animal hey = new Animal("buddy", 1);
}
```

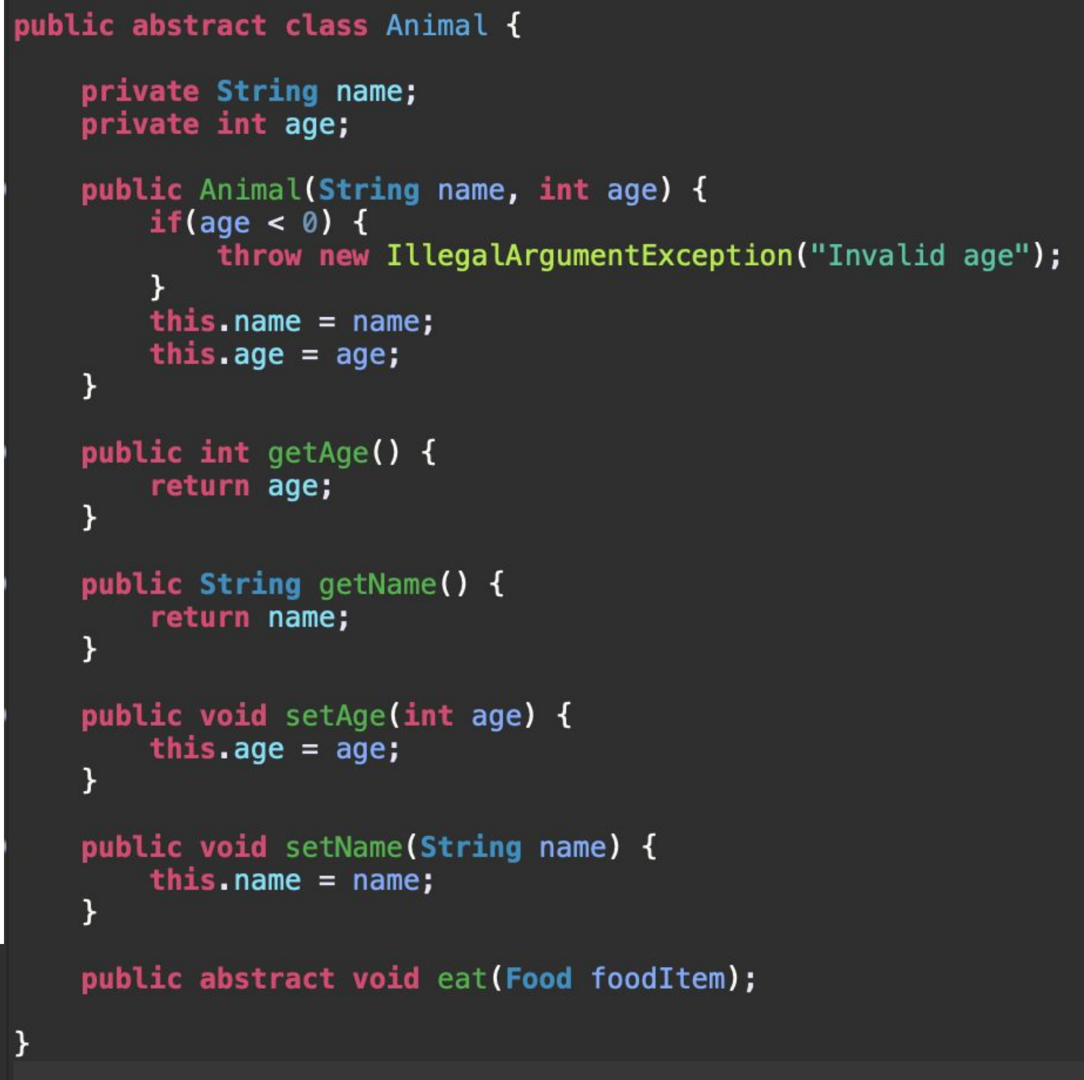
Abstrakte klasser

Eksempel

Vi får feilmelding;
Kan ikke
instansiere en
abstrakt klasse



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



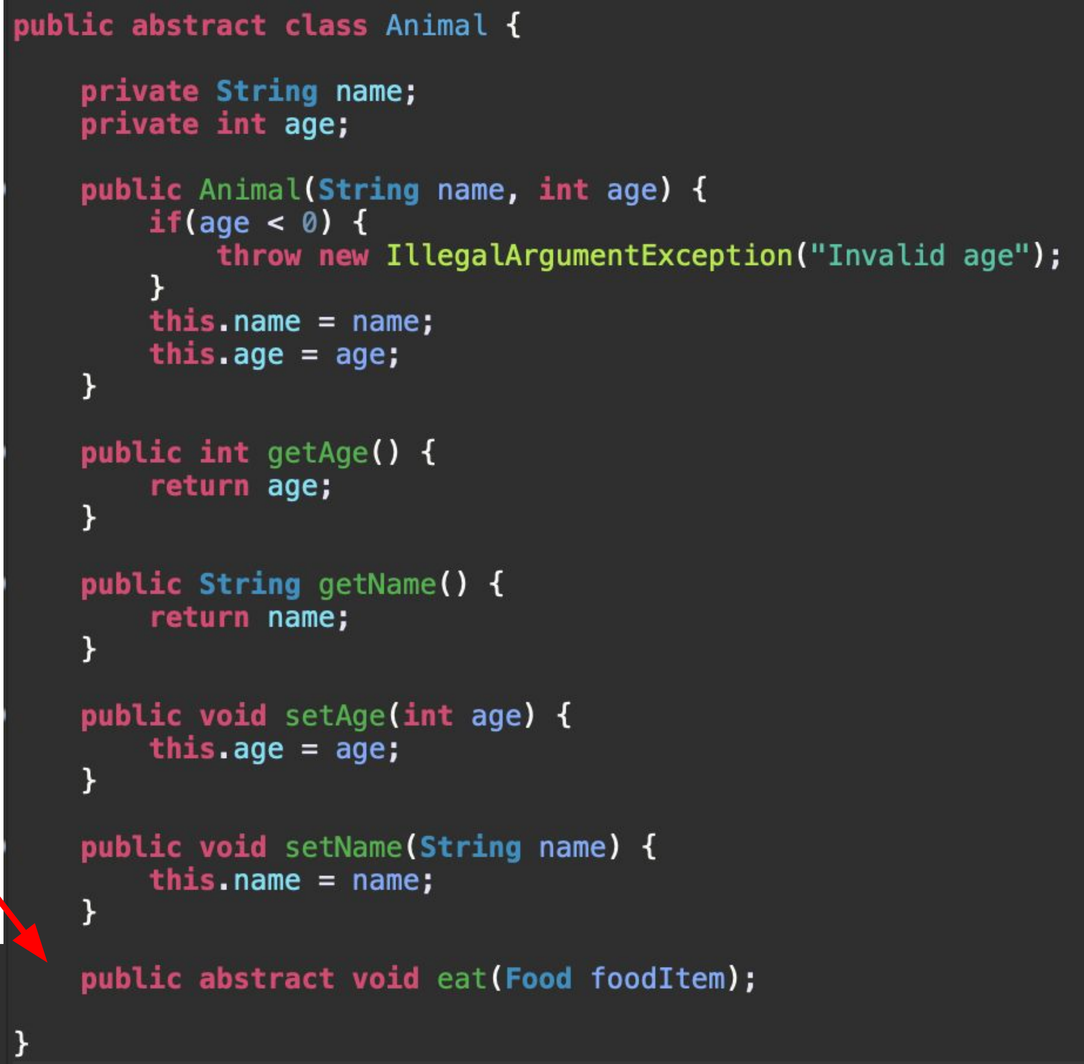
```java
public abstract class Animal {

    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);

}
```

Abstrakte klasser

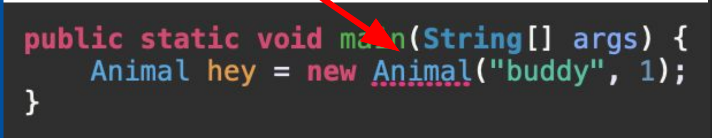
Eksempel

Siden vi tillater abstrakte klasser å ha abstrakte metoder så kan ikke disse instansieres (på samme måte som interface, som kun kan ha abstrakte metoder)

Vi får feilmelding;
Kan ikke instansiere en abstrakt klasse



```java
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);

}
```



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

`implements` vs. `extends`

Regler for bruk i Java

- **Arv** spesifiseres vha. `extends`-nøkkelordet Det er kun mulig å arve fra **én** klasse, dette gjelder også arv fra abstrakte klasser
- **Implementasjon** av interface defineres med `implements`. Det er mulig å implementere **flere** interface i én klasse. Dette gjelder **også** når en abstrakt klasse implementerer et interface.
- Et interface kan også **arve**, dvs. utvide, et annet interface vha. **extends**.
  - Et interface kan **også** utvide **flere** interface
- En klasse kan **ikke** arve fra **flere** klasser
    - (Dette er i Java, men det er mulig i enkelte andre språk!)

Hva er hva?

Arv, abstrakt klasse og interface

```
public interface Employee {

    void medicate(Patient patient, Medication medication);
    Hospital getEmployer();

    void setAvailable(boolean isAvailable);
    boolean isAvailable();

}
```

```
public abstract class Person {

    private String name;
    private int age;

    public Person(String name, int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age");
        }
        this.setName(name);
        this.setAge(age);
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age");
        }
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    public abstract void doWork();
```

```
public class Nurse extends Person implements Employee {

    private Hospital employer;
    private boolean isAvailable = true;

    public Nurse(String name, int age, Hospital employer) {
        super(name, age);
        this.employer = employer;
    }

    @Override
    public void medicate(Patient patient, Medication medication) {
        if(patient.getCurrentMedication() == null) {
            patient.receiveMedication(medication);
            this.employer.getPatientDB().getPatientLog(patient).add
            System.out.println(this.getName()+ " medicated "+ patie
        }
    }

    @Override
    public void doWork() {
        this.employer.handlePatient();
    }

    @Override
    public Hospital getEmployer() {
        return employer;
    }

    @Override
    public void setAvailable(boolean isAvailable) {
        this.isAvailable = isAvailable;
    }
```

Hva er hva?

Arv, abstrakt klasse og interface

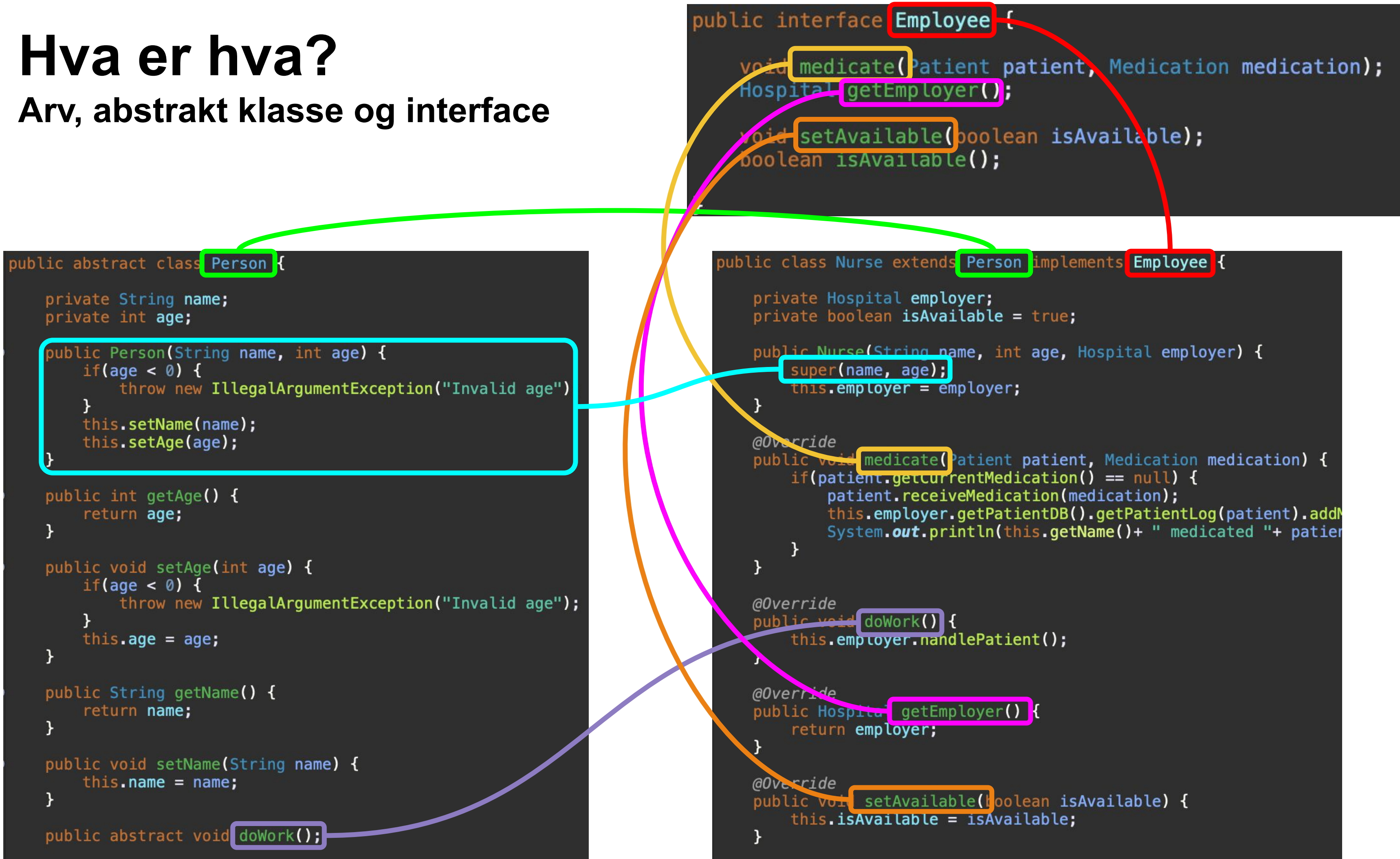
```java
public abstract class Person {

    private String name;
    private int age;

    public Person(String name, int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age");
        }
        this.setName(name);
        this.setAge(age);
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age");
        }
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    public abstract void doWork();
```

```java
public interface Employee {

    void medicate(Patient patient, Medication medication);
    Hospital getEmployer();

    void setAvailable(boolean isAvailable);
    boolean isAvailable();

}
```

```java
public class Nurse extends Person implements Employee {

    private Hospital employer;
    private boolean isAvailable = true;

    public Nurse(String name, int age, Hospital employer) {
        super(name, age);
        this.employer = employer;
    }

    @Override
    public void medicate(Patient patient, Medication medication) {
        if(patient.getCurrentMedication() == null) {
            patient.receiveMedication(medication);
            this.employer.getPatientDB().getPatientLog(patient).addM
            System.out.println(this.getName()+ " medicated "+ patier
        }
    }

    @Override
    public void doWork() {
        this.employer.handlePatient();
    }

    @Override
    public Hospital getEmployer() {
        return employer;
    }

    @Override
    public void setAvailable(boolean isAvailable) {
        this.isAvailable = isAvailable;
    }
```

Hva er hva?

Arv, abstrakt klasse og interface



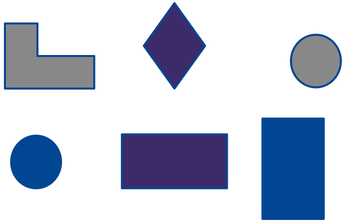
```java
public interface Employee {

    void medicate(Patient patient, Medication medication);
    Hospital getEmployer();

    void setAvailable(boolean isAvailable);
    boolean isAvailable();
}
```

```java
public abstract class Person {

    private String name;
    private int age;

    public Person(String name, int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age")
        }
        this.setName(name);
        this.setAge(age);
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age < 0) {
            throw new IllegalArgumentException("Invalid age");
        }
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    public abstract void doWork();
```

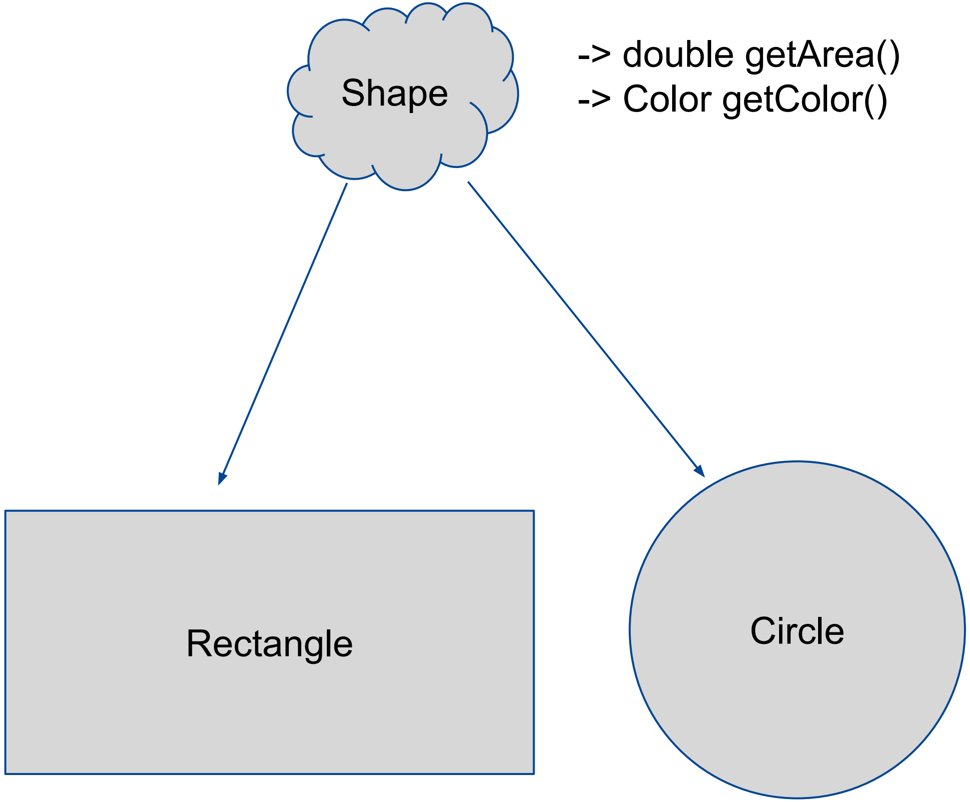
```java
public class Nurse extends Person implements Employee {

    private Hospital employer;
    private boolean isAvailable = true;

    public Nurse(String name, int age, Hospital employer) {
        super(name, age);
        this.employer = employer;
    }

    @Override
    public void medicate(Patient patient, Medication medication) {
        if(patient.getCurrentMedication() == null) {
            patient.receiveMedication(medication);
            this.employer.getPatientDB().getPatientLog(patient).addM
            System.out.println(this.getName()+ " medicated "+ patien
        }
    }

    @Override
    public void doWork() {
        this.employer.handlePatient();
    }

    @Override
    public Hospital getEmployer() {
        return employer;
    }

    @Override
    public void setAvailable(boolean isAvailable) {
        this.isAvailable = isAvailable;
    }
```

Oppgave 3 - Geometri

Tema: abstrakte klasser og arv



- Lag en abstrakt klasse, **Shape**
- **Shape** har to metoder, getArea() og getColor()
  - Implementer getColor, og la getArea() være abstrakt
- Lag klassene **Rectangle** og **Circle** som **arver** fra **Shape.** Lag passende konstruktører, og implementer getArea()



Shape

-> double getArea()
-> Color getColor()

Rectangle

Circle

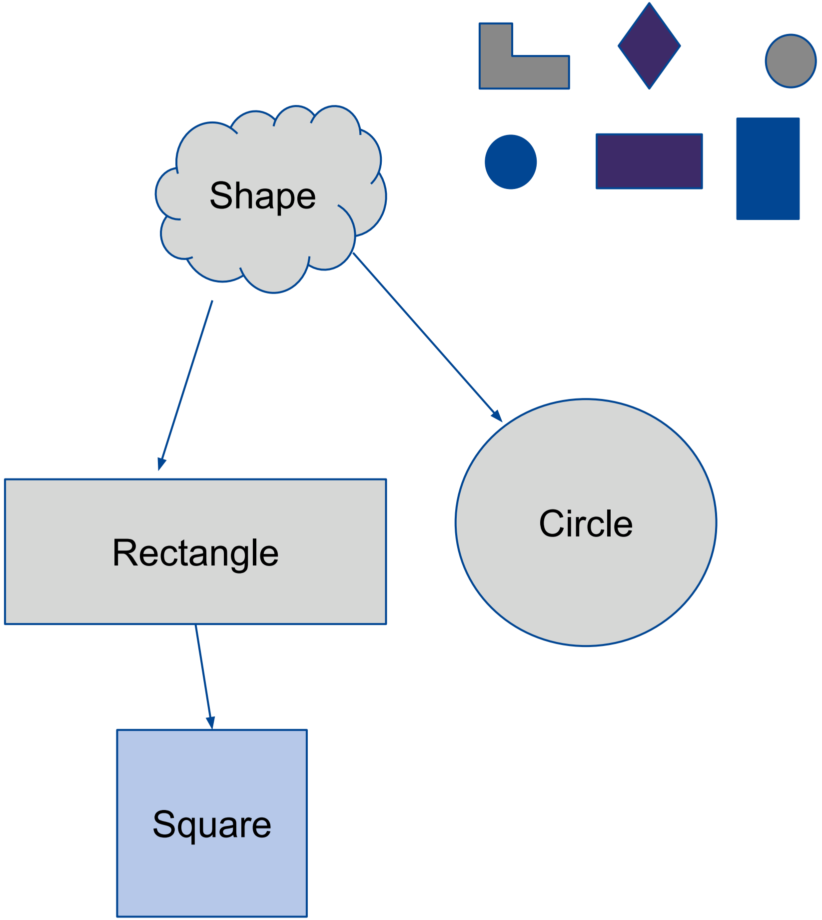
Oppgave 4 - Kvadrat

Tema: abstrakte klasser og arv

Subklasser kan også arves fra!

Et kvadrat er jo egentlig et rektangel med like lange sider.

- Lag klassen **Square** som arver fra **Rectangle.** Hvordan kan man bruke **super**() på en smart måte her?



Oppgave 5: ShapeContainer

Vi ønsker å ha en plass å samle flere figurer.

Lag klassen **ShapeContainer** som skal holde styr på **Shape**-objekter.

- **ShapeContainer** skal ha følgende metoder:
  - AddShape(Shape s)
    - Legger til en Shape i containeren
  - GetTotalArea()
    - Returnerer det totale arealet til alle figurene i containeren

Oppgave 6 (hvis tid): Kapasitet

Vi ønsker å kunne begrense kapasiteten i **ShapeContainer.**

- Legg til parameteren **Capacity** i konstruktøren, og utløs et unntak dersom det forsøkes å legge til en Shape det ikke er plass til.

- Lag en egen unntaksklasse som arver fra RuntimeException.

Oppgave 7 (hvis tid): Sortering

Vi ønsker å kunne sortere figurene i **ShapeContainer.**

- La **Shape** implementere **Comparable**, og gjør slik at de største figurene blir sortert først.

- Legg til metoden **getSortedShapes** i **ShapeContainer** som returnerer en sortert liste av figurene den inneholder (returner en kopi, og ikke endre på rekkefølgen i den originale listen).

Oppgave 8 (hvis tid)**:**

- Lag en liten applikasjon som lar brukeren se eksisterene ShapeContainers og lage nye Shapes som puttes i containerne