

# TDT4100 - Interface

Læringsmål denne uken:

- Objektorientering
  - Grensesnitt/interface
- Java-programmering
  - interface-konstruksjonen
  - implements-nøkkelordet

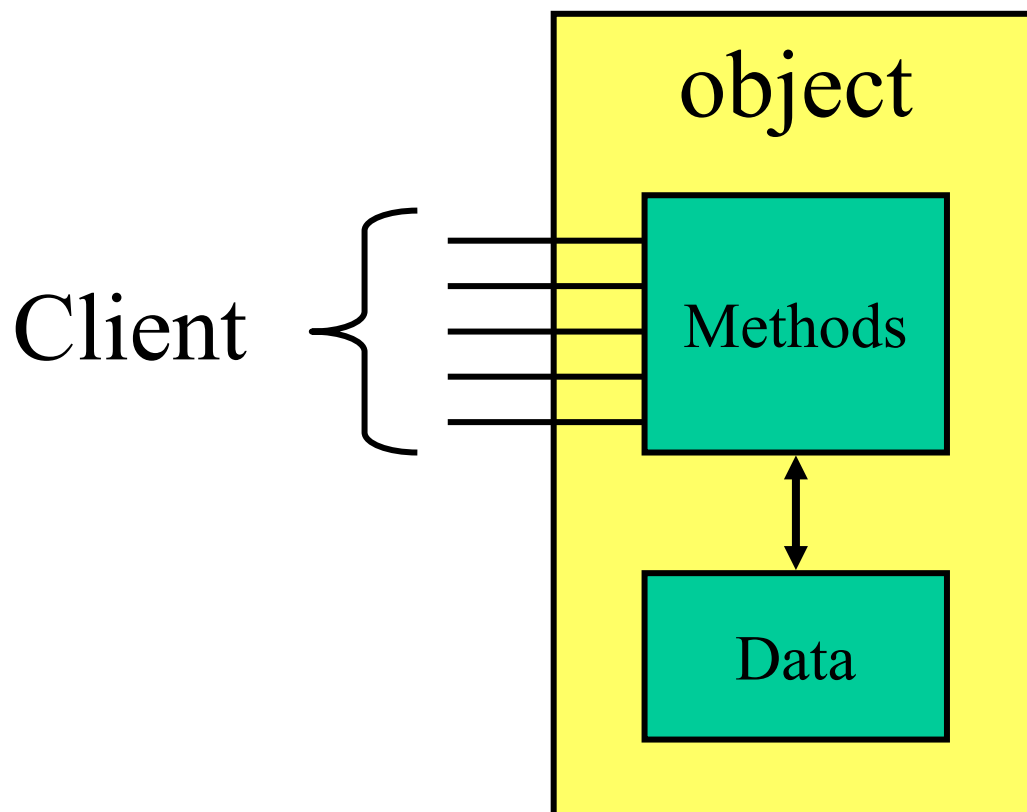
# Begreper denne uken

- Noen interfaces som vi skal se på
  - Predicate<T>
  - Comparator<T>
  - Comparable<T>

# Dere starter nå på en reise...

- som noen ganger kan se ut som en litt unødvendig (vill?) omvei
- Men som dere lærer dels fordi de er viktige:
  - dels fordi de leder til enklere måter å gjøre ting
  - gir oss generelle mekanismer som vi kan gjenkjenne og gjenbruke
  - Og som til slutt vil gi oss noen kule måter å gjøre ting på
- *Se pakken **personcomparator***

# Illustrasjon av innkapsling (fra en tidligere bok)



# Grensesnittet til en klasse

- Alle offentlige metoder i en klasse
- To perspektiver på grensesnitt
  1. fra klassen: dette er metodene jeg tilbyr
  2. fra klienten: dette er metodene jeg trenger
- Det beste er å møtes på midten:
  - klienten: dette er det jeg trenger
  - fra klassen: jeg kan (blant annet) det

# De tre ‘grensesnittene’

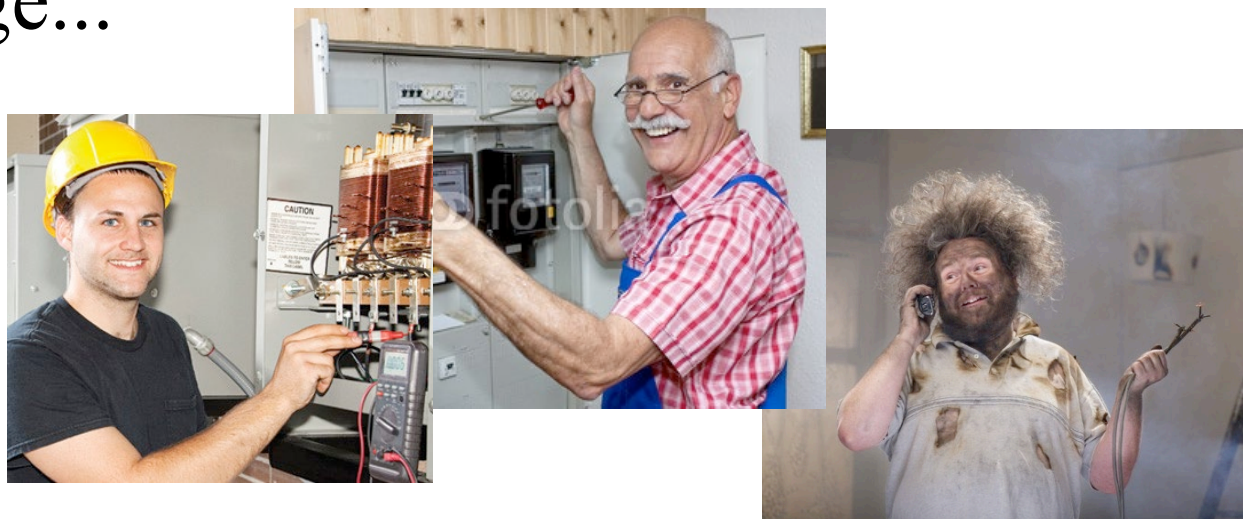
- (Grafisk) Brukergrensesnitt: JavaFX
- Forrige lysark: Det som er synlig av en klasse
- Denne forelesningsuken: Java sitt

## **Interface**

*som er en konrektisering av (en del av)  
en klasses grensesnitt*

# Analogi: Elektriker

- Du trenger en elektriker, hvilken skal du velge...

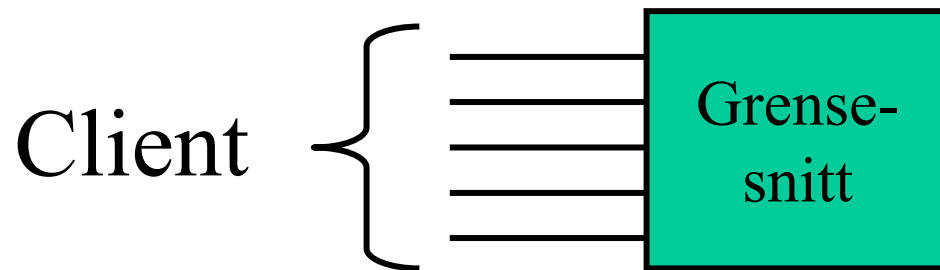


- Den som er sertifisert og kan faget sitt!



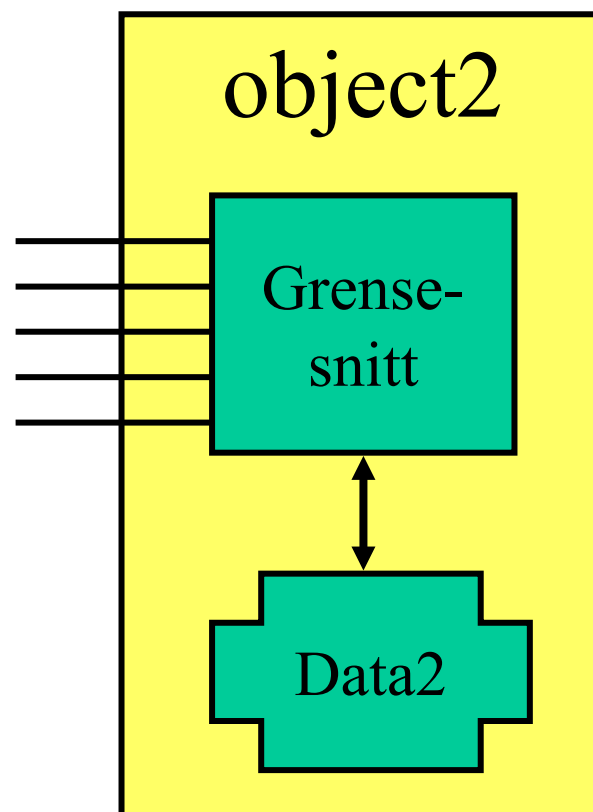
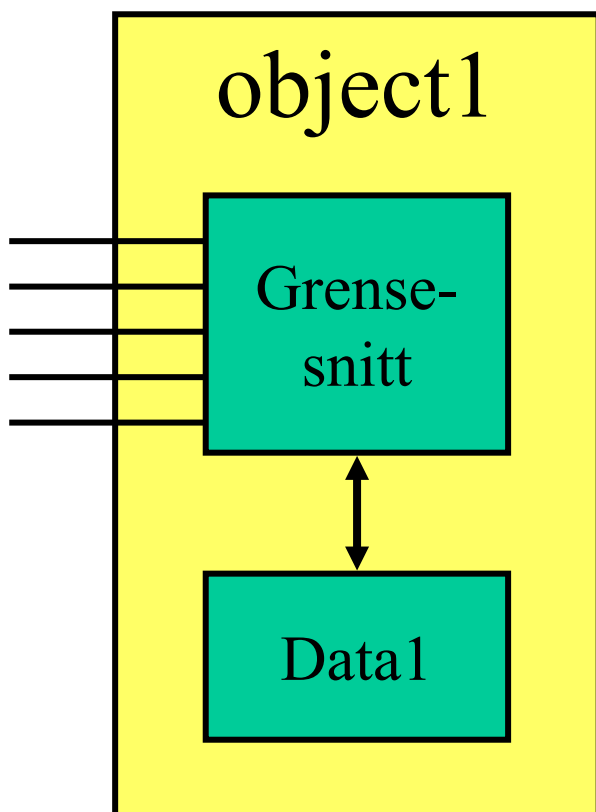


*interface*: klasse med  
metodedeklarasjoner,  
uten tanke på innhold forøvrig



- *Sett fra koden som bruker en klasse (klienten), så er metodegrensesnittet alt en trenger for å sikre gyldig kode.*
- *Det er nyttig å kunne si “trenger et objekt som kan dette...” uten å si hvilken klasse den er en instans av.*
- *(Et Interface kan implementere metoder, også statiske...)*

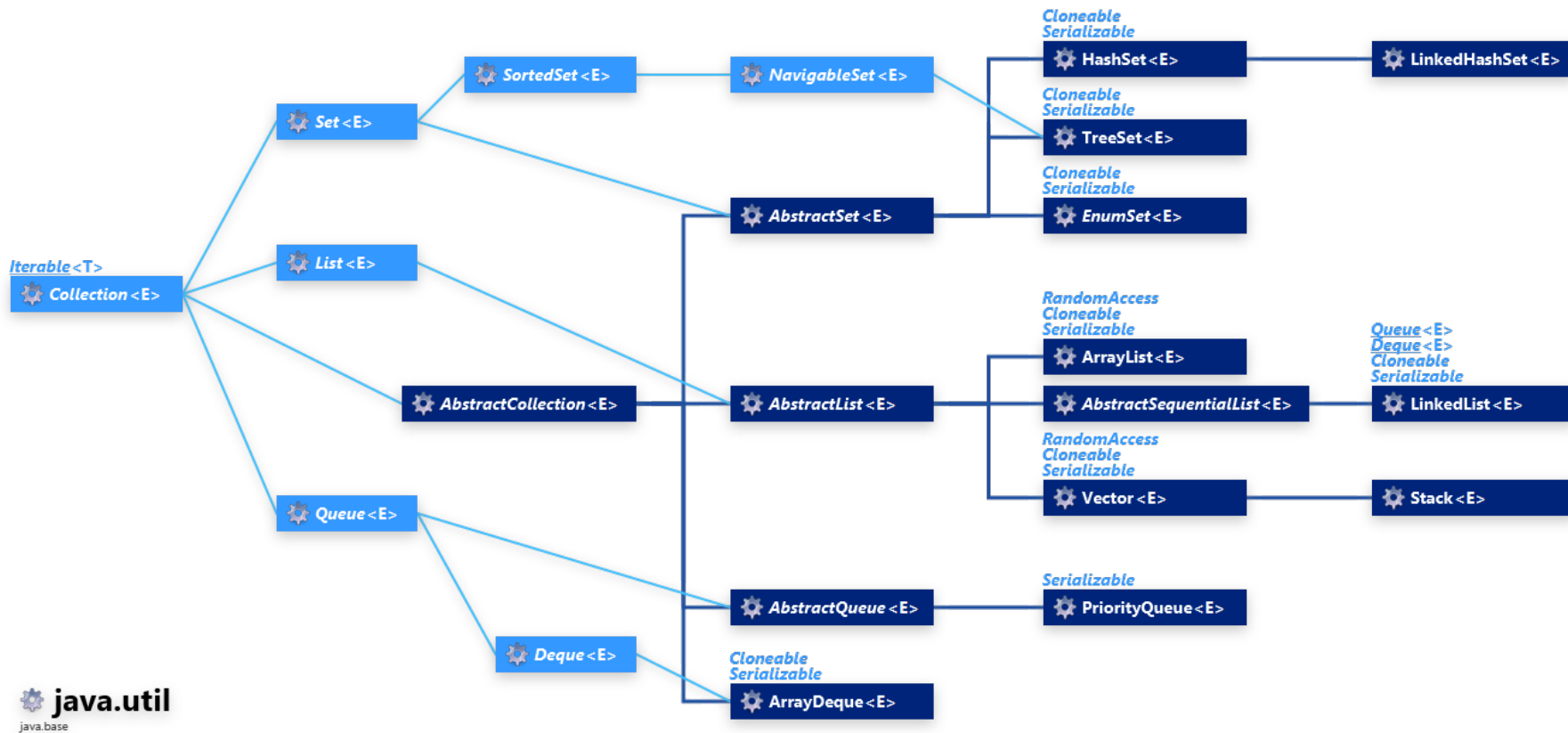
# Ulik innmat, samme grensesnitt



VI har allerede sett flere eksempler på Interface.

Lyseblått er **Interfaces**, mørkeblå er klasser.

Interfaces og klasser kan begge utvides og redefinert ved hjelp av arvemekanismen (inheritance).



# Arv (eng: inheritance)



Arv gir muligheten til definere nye klasser og grensesnitt fra andre klasser eller grensesnitt.

*Subklassene* arver alle felt og metoder fra *superklassen*, men hvor vi kan

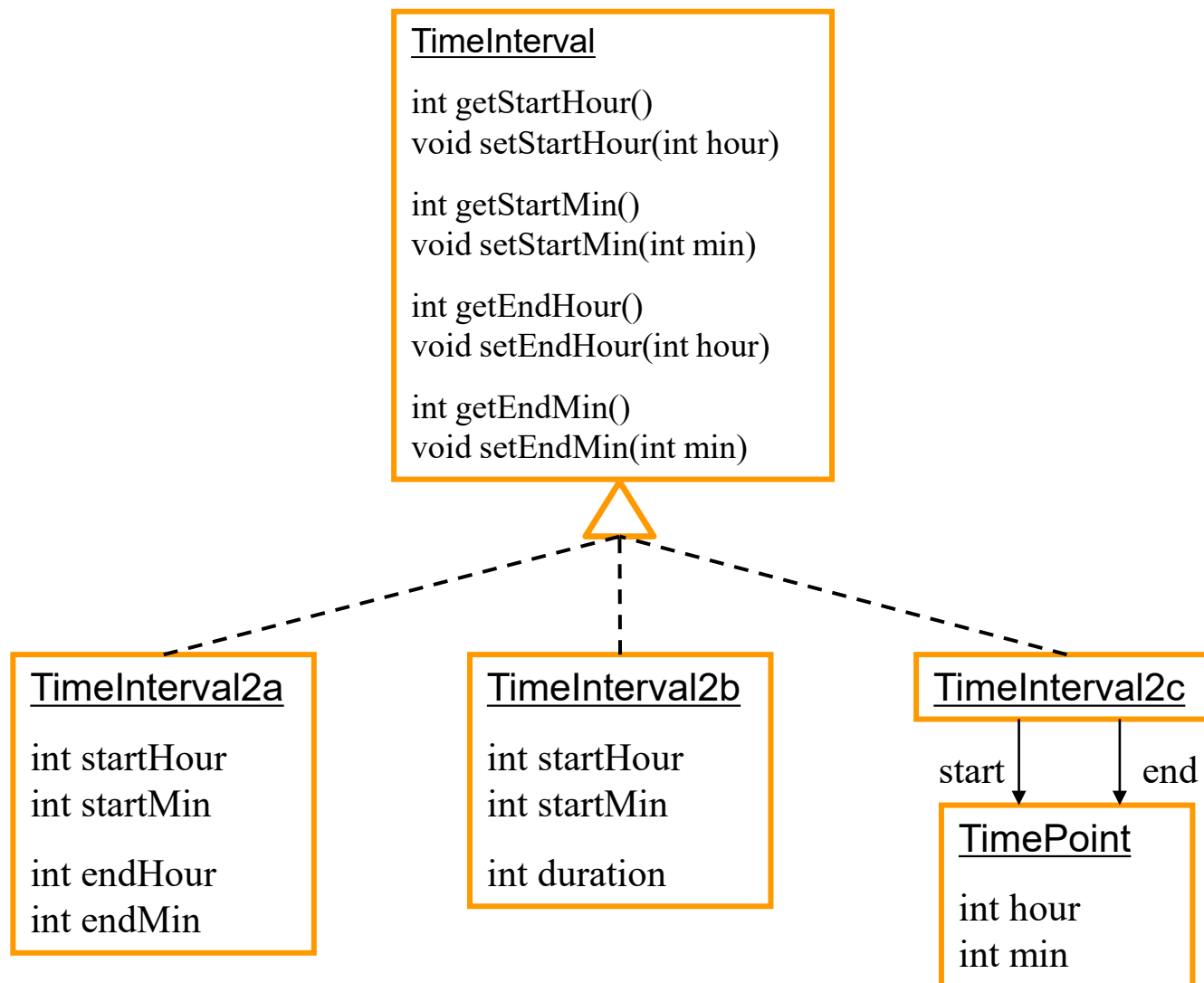
- **redefinere** metoder (vi har allerede sett @Override når vi f.eks. redefinerer toString())
- utvide med nye felt og metoder.
- Variable deklartert av en type, kan også referees til objekter av subklasse-typer.

Se: kortomarv, Account og CreditAccount.

# TimeInterval-eksemplet

- Tre varianter
  - TimeInterval1: startHour, startMin, endHour, endMin
  - TimeInterval2: startHour, startMin, duration
  - TimeInterval3: start, end (av typen TimePoint)
- Alle tre kan det samme:
  - getStartHour() og setStartHour(int hour)
  - getStartMin() og setStartMin(int min)
  - getEndHour() og setEndHour(int hour)
  - getEndMin() og setEndMin(int min)

# TimeInterval-grensesnitt



*Se pakke timeintervalinterface på gitlab*

# Person-klasse, med for-, etter- og fullt-navn

## To implementasjoner, samme metoder

```
public class Person {

    private String fulltNavn;

    public String getFornavn() {
        return fulltNavn.substring(0, fulltNavn.indexOf(' ') + 1);
    }

    public void setFornavn(String fornavn) {
        fulltNavn = fornavn + " " + getEtternavn();
    }

    public String getEtternavn() {
        return fulltNavn.substring(fulltNavn.indexOf(' ') + 1);
    }

    public void setEtternavn(String etternavn) {
        fulltNavn = getFornavn() + " " + etternavn;
    }

    public String getFulltNavn() {
        return fulltNavn;
    }

    public void setFulltNavn(String fulltNavn) {
        this.fulltNavn = fulltNavn;
    }

}
```

```
public class Person {

    private String fornavn, etternavn;

    public String getFornavn() {
        return fornavn;
    }

    public void setFornavn(String fornavn) {
        this.fornavn = fornavn;
    }

    public String getEtternavn() {
        return etternavn;
    }

    public void setEtternavn(String etternavn) {
        this.etternavn = etternavn;
    }

    public String getFulltnavn() {
        return fornavn + " " + etternavn;
    }

    public void setFulltNavn(String fulltNavn) {
        int pos = fulltNavn.indexOf(' ');
        setFornavn(fulltNavn.substring(0, pos));
        setEtternavn(fulltNavn.substring(pos + 1));
    }

}
```

# Person-grensesnitt med to implementasjoner

```

Person1.java
package uke7forelesning1;

public class Person1 implements Person {

    private String fulltNavn;

    public String getFornavn() {
        return fulltNavn.substring(0, fulltNavn.indexOf(' '));
    }

    public void setFornavn(String fornavn) {
        fulltNavn = fornavn + " ";
    }
}

Person2.java
package uke7forelesning1;

public class Person2 implements Person {

    private String fornavn, etternavn;

    public String getFornavn() {
        return fornavn;
    }

    public void setFornavn(String fornavn) {
        fornavn = fornavn;
    }
}

Person.java
package uke7forelesning1;

public interface Person {

    public String getFornavn();
    public void setFornavn(String fornavn);

    public String getEtternavn();
    public void setEtternavn(String etternavn);

    public String getFulltNavn();
    public void setFulltNavn(String fulltNavn);
}
    
```

*Se pakke personinterface på git*

# Eksempel: søke etter objekt

- Mange metoder som søker og finner objekter bygger på to uavhengige “ferdigheter”:
  - metoden for å løpe gjennom elementene systematisk
  - kriteriet for hvilket/hvilke object/objekter som skal returneres
- Fordel å kunne skrive koden for den generelle metoden, uten å måtte bestemme kriteriet
- Hvis en kan “plugge inn” kriteriet, så kan samme søkemetode brukes for mange ulike søk
  - søke etter filer med bestemt fil-endelse eller -type
  - søke etter filer endret etter en bestemt data
  - søke etter tomme mapper
- **Tar File/Folder på onsdag, hvis tid.**

# Eksempel: søke etter objekt (onsdag)

- FindCriterion-grensesnitt
  - to metoder som avgjør om mappe eller fil skal tas med
  - boolean keepFolder(Folder) avgjør om mappa skal med
  - boolean keepFile(File) avgjør om filen skal med
- findAll(FindCriterion)
  - helt generell navigering/traversering i mappestruktur
  - kaller keepFolder- og keepFile-metoder i FindCriterion-objekt for å avgjøre om hhv. mapper og file skal med
- findAllFiles(Predicate<File>)
  - samme som findAll, men kun for filer
  - bruker standardgrensesnittet Predicate for å avgjøre om en fil skal med eller ikke

# interface: Predicate<T>

- Generelt grensesnitt for å teste (godkjenne) om et objekt tilfredsstillter et kriterium
- Har én metode **boolean test(T objekt)** som sier ja (**true**) eller nei (**false**) for hvert objekt det blir spurt om
- **Predicate<T>**-objektet er en instans av en klasse som implementerer **Predicate** for en bestemt type **T**.

# Predicate-eksempel (onsdag)

```
public class FileNamePredicate implements Predicate<File> {  
  
    private final String name;  
  
    public FileNamePredicate(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean test(File file) {  
        return this.name.equals(file.getName());  
    }  
}
```

*La oss lage en enkel en først – i  
pakken personpredicate*

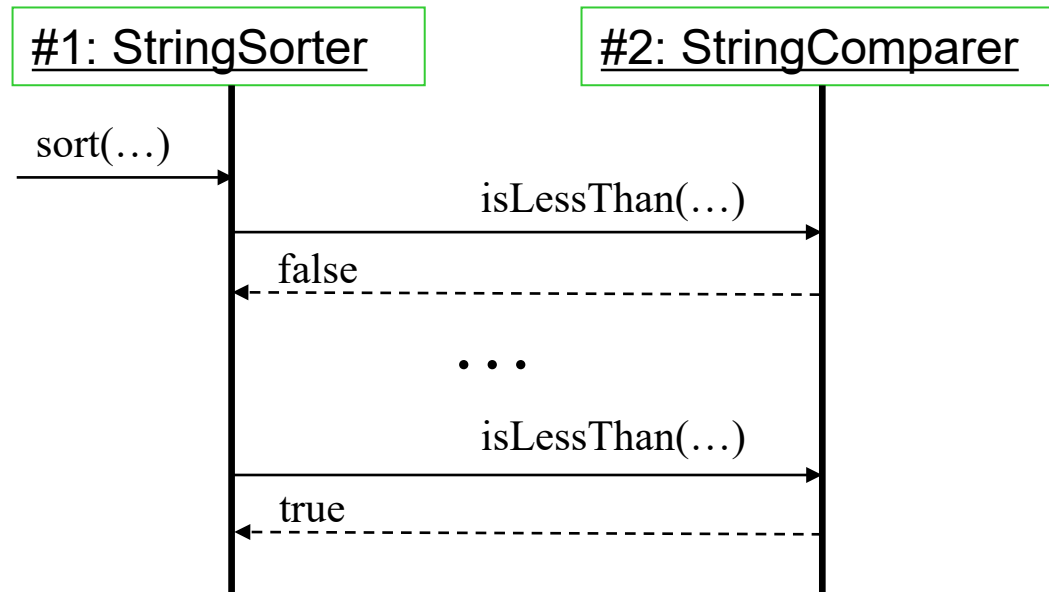
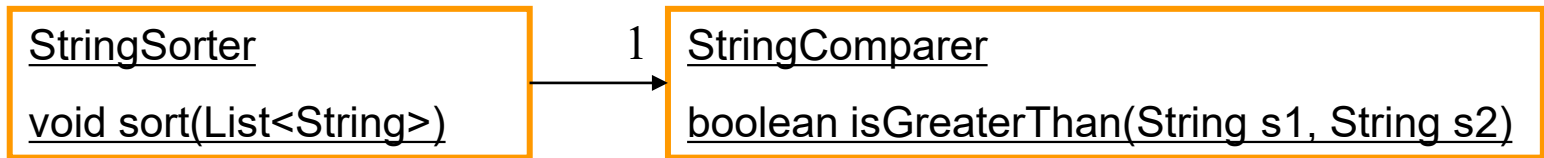
# Eksempel: sortering

- Mange sorteringsmetoder bygger på to uavhengige “ferdigheter”:
  - metoden for å løpe gjennom elementene systematisk
  - regel for parvis sammenligning av elementer, altså  $\geq$ -reglene
- Fordel å kunne skrive koden for den generelle metoden, uten å måtte bestemme  $\geq$ -regelen
- Hvis en kan “plugge inn”  $\geq$ -regelen, så kan samme sorteringsmetode brukes for mange ulike sorteringer
  - sortere String-objekter alfabetisk eller på lengde
  - sortere personer på etternavn og så fornavn
  - sortere stigende eller synkende

<https://www.toptal.com/developers/sorting-algorithms>

	 <u>Insertion</u>	 <u>Selection</u>	 <u>Bubble</u>	 <u>Shell</u>	 <u>Merge</u>	 <u>Heap</u>	 <u>Quick</u>	 <u>Quick3</u>
 <u>Random</u>								
 <u>Nearly Sorted</u>								
 <u>Reversed</u>								
 <u>Few Unique</u>								

# StringSorter og StringComparer



# Sortering og grensesnitt

- Sortering er en generell funksjon
  - krever minimal kjennskap til objektene som skal sorteres
  - mange sorteringsalgoritmer krever kun at objektene kan sammenlignes, dvs. at en kan avgjøre hvilket av to objekter som betraktes som "minst".
- Java har mange klasser som støtter sortering av sine objekter, men for å kunne sortere objektene må objektene implementere en sammenligningsmetode
- Java definerer to grensesnitt som brukes for å gjøre nye typer objekter sorterbare for eksempel vha. sort-metoden i `java.util.Arrays`-klassen:
  - `java.lang.Comparable`
    - implementeres av objekter som skal kunne sorteres
  - `java.util.Comparator`
    - implementeres av støtteklasse, for å gjøre andre typer objekter sorterbare
  - `Collections.sort()` virker ikke på `Collection`, man må opp på `List`-nivå

# Comparator<T>

- Comparator-grensesnittet kan brukes for å definere en ordning av *andre* objekter, basert på parvis sammenligning med **compare**-metoden
- En kan tenke seg flere Comparator-implementasjoner for Person-klassen:
  - **HeightComparator** implements **Comparator<Person>**: sammenligner høyden
  - **AgeComparator** implements **Comparator<Person>**: sammenligner alderen
  - **NameComparator** implements **Comparator<Person>**: sammenligner navn, først etternavn og så fornavn

# Comparator<T>

- Én metode:
  - `int compare(T o1, T o2);`
- Sammenligner to andre objekter og sier om det første er mindre, like eller større enn det andre
- Returverdi sier om
  - `o1` er mindre enn `o2`  $\Rightarrow$  returverdi  $< 0$
  - `o1` er lik `o2`  $\Rightarrow$  returverdi  $== 0$
  - `o1` er større enn `o2`  $\Rightarrow$  returverdi  $> 0$

# Comparator-eksempel

*Klasse som implementerer  
Comparator*

```
import java.util.Comparator;

public class PersonAgeDesc implements Comparator<Person> {

    public int compare(Person o1, Person o2) {
        return ((Person)o2).getAge() - ((Person)o1).getAge();
    }

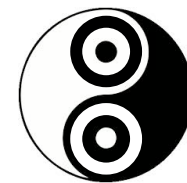
}
```

*Sortering*

```
Person[] personer = {
    new Person(2), new Person(1), new Person(3) };
java.util.Arrays.sort(personer, new PersonAgeDesc());
```

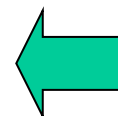
*Se package personcomparator*

# Forskjellige typer objekter



data

- Dataorienterte objekter
  - primært laget for å lagre sammenhørende dataverdier eller knytte objekter sammen
- "Vanlige" objekter
  - til sammenhørende data hører operasjoner (også kalt metoder) for å manipulere dataene
- Funksjonsorienterte objekter
  - brukes utelukkende for evnene til å gjøre noe, f.eks. behandle data som ligger i andre objekter



funksjoner

# Comparable<T>

- T er som regel klassen selv, f.eks.  
`Person implements Comparable<Person>`
- Én metode:
  - `int compareTo(T annetObject);`
- Returverdi sier om **this**-objektet, altså instans av klassen som implementerer Comparable:
  - er mindre enn annetObject => returverdi < 0
  - lik annetObject => returverdi == 0
  - er større enn annetObject => returverdi > 0

*Se package personcomparator*

# Når benytter man grensesnitt?

- Når det er mange varianter av samme logiske funksjon/tjeneste
  - mange type konti, men kun ett sett metoder
  - mange såkalte tegnstrømmer (strenger, fil, nettverk, tastatur, ...), som håndteres med et fåtall metoder
- Når en generell metode/algoritme kun trenger et begrenset sett med metoder
  - mange sorteringsalgoritmer trenger kun å sammenligne to og to elementer, for å finne ut i hvilken rekkefølge de skal være

# Definisjon og bruk av grensesnitt



1. Det defineres et såkalt *interface* (grensesnitt), som angir hvilke metoder som er relevante
2. I koden som bruker metodene, må det deklarereres at objekt(er) med dette grensesnittet behøves
3. Klassene som har de relevante metodene, må eksplisitt si fra at interfacet (og de relevante metodene) er implementert

# interface

- Er en samling av “abstrakte” metoder
  - Metoder som ikke har en implementasjon
  - Metoden deklarerer, men har ingen metodekropp
  - Lagres som ei fil, på samme måte som klasser
  - Klasser kan *implementere* grensesnitt (ett eller flere)

```

Person.java
package uke7forelesning1;

public interface Person {

    public String getFornavn();
    public void setFornavn(String fornavn);

    public String getEtternavn();
    public void setEtternavn(String etternavn);

    public String getFulltNavn();
    public void setFulltNavn(String fulltNavn);
}
    
```

```

Person1.java
package uke7forelesning1;

public class Person1 implements Person {

    private String fulltNavn;

    public String getFornavn() {
        return fulltNavn.substring(0, fulltNavn.indexOf(' '));
    }

    public void setFornavn(String fornavn) {
        fulltNavn = fornavn + " ";
    }
}
    
```

Interfacet definerer nødvendige metoder  
og reglene som disse må tilfredsstille

Vi trenger ikke vite akkurat når og  
hvordan de blir kalt,  
bare reglene blir fulgt!

# Oppgave

- La Person-klassen implementere Comparable<Person>-grensesnittet
- Implementer metoden som kreves, at personer sorteres i alfabetisk rekkefølge
- Utvid metoden til å ta hensyn til fornavn, dersom etternavnet er likt
- Hva må endres for å snu rekkefølgen?

# Comparable-eksempel

```
public class Person implements Comparable {  
  
    private int age;  
  
    public Person(int age) {  
        this.age = age;  
    }  
  
    public int compareTo(Object o) {  
        return this.age - ((Person)o).age;  
    }  
}
```

*Klasse som implementerer  
Comparable*

```
Person[] personer = {  
    new Person(2), new Person(1), new Person(3)};  
java.util.Arrays.sort(personer);
```

*Sortering*

*Se package personcomparator, litt utvidet kode*

# Oppgave forts.

- Lag en Comparator-klasse som sorterer to Personer p1 og p2 etter alder.
- Hint: alt klassen trenger er en metode `compare(<objekt1>,<objekt2>)` som returnerer en int som sier noe om sammenlikning mellom objektene
- Hint2: Person har `getAlder()`
- Kall denne fra main (gjort i gitkode)

# Hvordan finner man på grensesnitt?

- Varianter av samme logiske funksjon/tjeneste
  - tjenesten kan identifiseres på forhånd, med spørsmål av typen "hva er det egentlig en konto må kunne?"
  - en ser fellestrekk ved ulike klasser og ser at med litt omskriving så kan de brukes helt likt
  - eksempel: en konto, med metodene getBalance, deposit og withdraw
- En generell metode/algoritme som en ønsker å tilpasse (på samme sted hver gang)
  - den delen en ønsker å tilpasse skilles ut som en eller flere metoder
  - disse flyttes så over i et grensesnitt, en instans av grensesnittet tas inn som parameter og instansens metoder kalles på de relevante stedene
  - eks.: sortering og sammenligning av verdier

# Hva er/kan en konto?

- En konto har en saldo og muligheten til å sette inn eller ta ut et bestemt beløp
  - `int getBalance()` // hva er saldoen
  - `void deposit(int amount)` // sett inn
  - `int withdraw(int amount)` // ta ut og hvor mye fikk jeg
- Det finnes flere typer konti, med ulike regler for hvordan deposit og withdraw håndteres
  - standardkonto, uten mulighet for overtrekk
  - konto med kreditt, dvs. mulighet til å gå et visst beløp i minus
  - gullkonto, med ubegrenset kreditt
- En minibank (ATM) trenger bare kjenne til metodene nevnt over, ikke alle variantene

*Sjekk pakken `accountinterface`*

# Definere Konto-grensesnittet

- Kravene til hvilke metoder alle konto-klasser må ha, kan defineres med interface-konstruksjonen:


```
public interface Account {  
    public int getBalance();  
    public int deposit(int amount);  
    public int withdraw(int amount);  
}
```



# Implementere Konto-grensesnittet

- De spesifikke Account-typene må implementere Account-grensesnittet:

```
public class SavingsAccount implements Account {
    int balance;

    public int getBalance() {
        return this.balance;
    }

    public int deposit(int amount) {
        
        return balance;
    }

    public int withdraw(int amount) {
        if (
            
        ) else
            
        }
    }
}
```

*Se package accountinterface*

# Kontointerface som type

- Identifikatorer og uttrykk kan være av en interface-type (som Collection og List)

```
Account konto = new SavingsAccount();  
konto.deposit(500);  
int uttak = konto.withdraw(3000);
```

*Se package accountinterface*

# Datastyrt løkke

Kan man gjøre noe lignende her?

```
List<Character> charList;
...
for (int i = 0; i < charList.size(); i++) {
    Character c = charList.get(i);
    print(c);
}
```

```
String s;
...
for (int i = 0; i < s.length(); i++) {
    Character c = s.charAt(i);
    print(c);
}
```

```
char[] charArray;
...
for (int i = 0; i < charArray.length; i++) {
    Character c = charArray[i];
    print(c);
}
```

*Se pakke loekkeinterface*

# Deklarerte og faktiske typer

- Den *deklarerte* typen til et uttrykk, f.eks. variabel, innebærer en *garanti* for hva vi kan gjøre med objektet (bruke felt/kalle metoder)
  - `String s` // vi kan utføre `s.substring(4)`
  - `Comparable<String> cs` // vi kan utføre `cs.compareTo(s)`
- Den *faktiske* typen til et objekt, er klassen som en tok **new** på
- Disse bør stemme overens, men er ikke alltid like, pga. grensesnitt og arv...
- Med **instanceof X** kan en sjekke om et objekt kan brukes som en X, f.eks. med casting (X)

# instanceof

- Hva om vi vil vite hvilken type et objekt egentlig er?
  - For eksempel for å kalle metoder som ikke er tilgjengelige fra grensesnitt-typen
- Operatoren `instanceof` sjekker om et objekt er av en bestemt klasse
- `<objekt> instanceof <klasse>`  
gir enten true eller false som resultat

# casting

- *Casting* tvinger Java til å *akseptere* at en objekt(referanse) er av en annen type enn deklarasjonene *garanterer*
- Operator:
  - (typenavn) typenavn i parantes
- Bør sjekke om et objekt kan castes til X:

```
if (objekt instanceof X) {  
    X x = (X) objekt;  
}
```

# Eksempel

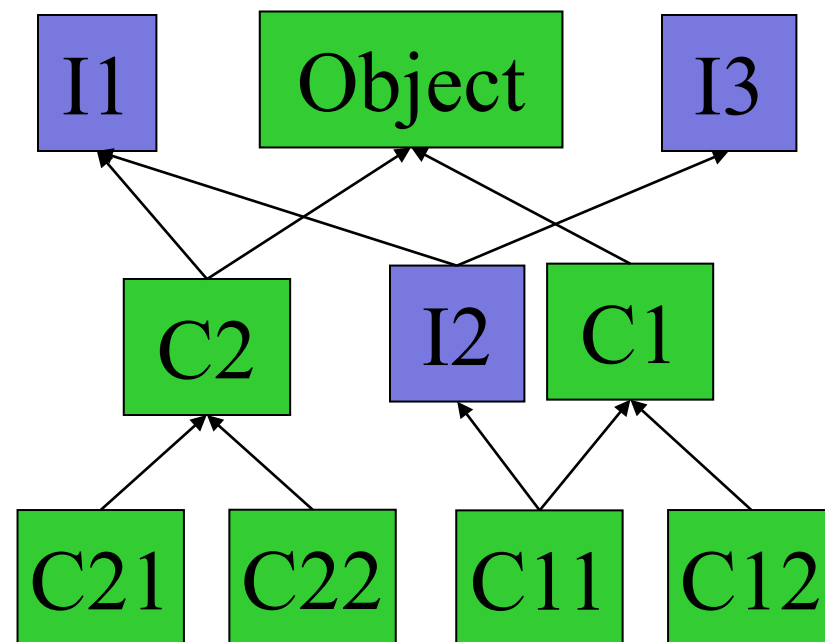
- Hvis CreditAccount har metoden
  - `int getCredit()`
- Men referansen er (kun garantert å være) av typen Account?

```
Account konto = new CreditAccount(5000);

if (konto instanceof CreditAccount) {
    CreditAccount cKonto = (CreditAccount) konto;
    System.out.println("Kreditt = " + cKonto.getCredit());
}
```

# Lynkurs i arv

- Klasser struktureres i et arvingshierarki, f.eks. C1, C11, C12, C2, C21, C22
- En kan også implementere/arve ha en eller flere *interface*, som I1, I2, I3
- Et objekt laget som en instans av en klasse C, er **instanceof** C og alle C sine *superklasser*



→ peker på superklassen

```

C21 c21 = new C21();

c21 instanceof C2 == true
c21 instanceof I1 == true
  
```

# Læringsmål for forelesningen

- Objektorientering
  - Grensesnitt
- Java-programmering
  - interface-konstruksjonen
  - implements-nøkkelordet

