

# TDT4100 - unntak

# Læringsmål for unntakshåndtering

- Objektorientering
  - Håndtering av unntak (eng: exceptions)
- Java-programmering
  - Håndtering av unntak
    - Exception-objekter og –klasser
    - try, catch og finally
    - throw og throws
- VS Code
  - Se på klassehierarki med F4



# Unntakshåndtering

- Motivasjon - hvorfor unntakshåndtering
- Basismekanismen
  - Hvordan fange andres unntak med **try/catch**
  - **Exception**-objekter
  - Hva skjer ”egentlig” når unntak oppstår?
- Litt mer avansert
  - Unntak og arv
    - **Exception**-objekter er strukturert i et klassifiseringshierarki
    - checked vs. unchecked
  - Hvordan si fra om unntak med **throws**
  - Eksempel på checked unntaksklasse: `IOException`
  - Bruk av **finally**

# Unntak (eng: exceptions)

- I et program er det mye som kan gå galt
  - koden kan inneholde feil som viser seg ved kjøring, f.eks. at attributter får gale verdier eller det er inkonsistens mellom to ender av en relasjon
  - uønskede, men forventede ting skjer, som det vil kludre til koden å håndtere overalt i koden, f.eks. at en filoverføring brytes og må gjenopptas senere
    - hvordan returnere info om feilsituasjoner, til forskjell fra normal verdier?
  - omgivelsene kan endre seg underveis, slik at uventede situasjoner oppstår, f.eks. en fil som slettes mens programmet leser fra eller skriver til den
- Hva gjør en programmerer med slikt?

# Unntak (eng: exceptions)

- Det finnes to prinsippielt ulike strategier for å håndtere slikt:
- **Proaktivt:** En kan (fortvilt) prøve å unngå feil, f.eks. ved
  - bedre metoder for å avdekke behov og krav
  - grundig gjennomgang av kode før den ”settes i produksjon”
  - bedre og grundigere testing, slik at flere feil lukes bort
- **Reaktivt:** En kan oppdage og reagere på dem på en ryddig måte
  - validering av input fra omgivelsene (filer, nettet, brukere, ...) og parameterverdier til metoder, slik at feil ikke får forplante seg videre.

# Unntak (eng: exceptions)

1. En kan (fortvilt) prøve å unngå feil
  2. En kan oppdage og reagere på dem på en ryddig måte
- Det vil alltid gjenstå feil en ikke har oppdaget og situasjoner en ikke har forutsett, som gjør at en må regne med unntak.
  - Det har vist seg å være ryddigere å håndtere unntakene vha. en egen unntaksmekanisme
  - Java har en mekanisme for å si fra om og håndtere slike unntak, som gjør at kode for normalsituasjonen ikke blir for tilkludret.

# Unntaksmekanismen

- Java sin unntaksmekanisme er todelt
  - **throw** brukes for å si fra om at et unntak har oppstått (eng: throw an exception)
  - **throw** brukes internt av Java når vi bruker språket og standardklassene feil, f.eks. følger en null-referanse
  - velger å si at **throw utløser** unntak, ikke ~~kaster~~!
  - **try { ... } catch ( ... ) { ... }** brukes for å angi at en er beredt til å håndtere en bestemt type unntak (eng: catch an exception)

# Unntaksmekanismen

- Java sin unntaksmekanisme er todelt:  
**throw** utløser unntak  
**try/catch** håndterer dem
- **throw** <unntaks-objekt> brukes for å si fra om at et unntak har oppstått
  - f.eks. **throw** new IllegalArgumentException(«Feil, alarm!»);
  - **throw** brukes internt av Java når vi bruker språket og standardklassene feil, f.eks. følger en null-referanse
  - Vi sier *utløse* unntak, ikke ~~kaste~~!



# Unntaksmekanismen

- **try { ... }**  
**catch (<unntaksklasse>...) { ... }**

brukes for å angi at en er beredt til å håndtere en bestemt type unntak

– f.eks.

**String input = ... // leser input**

**try {**

**int num = Integer.valueOf(input); // gjør om input til et tall**

**catch (NumberFormatException nfe) {**

**System.out.println(«Du må skrive inn et gyldig tall»);**

**}**

# Unntaksobjekter

- Instanser av (subklasser av) **Exception**, brukes for å lagre data om unntaket
  - **Exception**-objektet bør inneholder relevant informasjon om hva som gikk galt.
  - **catch**-setningen angir hvilke typer **Exception**-instanser den kan "ta imot", dvs. instanser av hvilke subklasser den håndterer. Data fra **Exception**-instansen som er "utløst" kan brukes i **catch**-koden for å finne ut hva som evt. kan gjøres med situasjonen.
- Under vil **nfe** referere til en instans av *unntaksklassen* `NumberFormatException`:

```
String input = ... // leser input
try {
    int num = Integer.valueOf(input); // gjør om input til et tall
} catch (NumberFormatException nfe) {
    System.out.println(«Du må skrive inn et gyldig tall»);
}
```

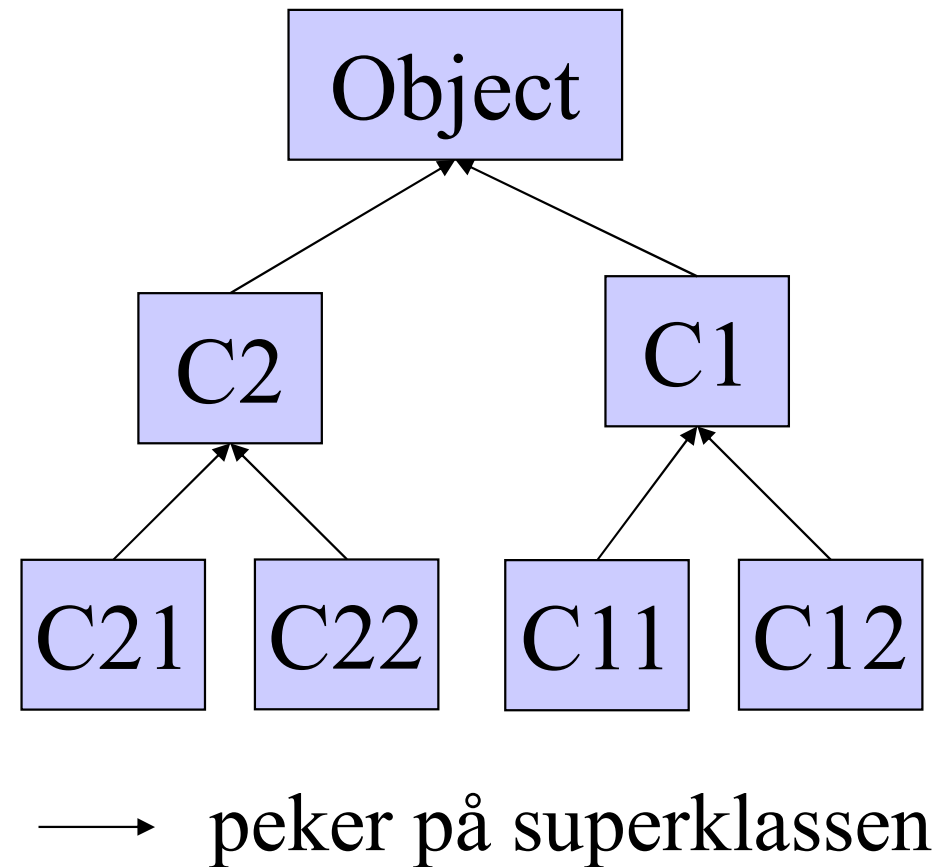
# try/catch og kall-stacken

- Når **throw** utløser unntak, så avbrytes metodekallene på stacken én etter én
- **catch** setter opp sperrer for avbrytingen så programmet kan fortsette som «normalt»

```
        throw new IllegalArgumentException(«...»)  
met6()  _____  
met5()  _____ catch (IOException e) { ... }  
met4()  _____  
met3()  _____ catch (IllegalArgumentException e)  
met2()  _____ { ... }  
met1()  _____
```

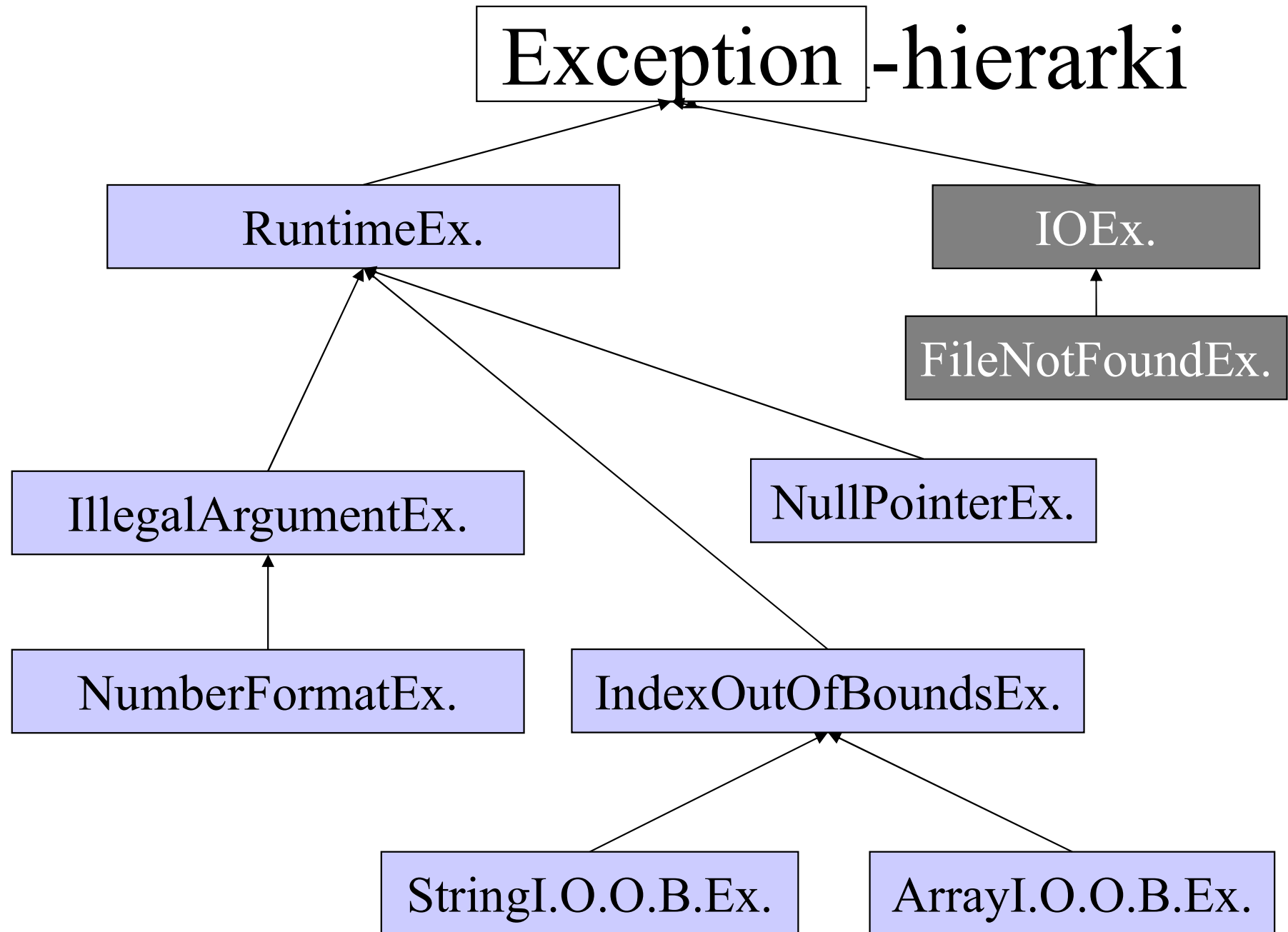
# Lynkurs i arv

- Klasser struktureres i et hierarki, f.eks. C1, C11, C12, C2, C21, C22
- Et objekt laget som en instans av en klasse C, er **instanceof** C og alle C sine superklasser



```
C21 c21 = new C21();
c21 instanceof C2 == true
```

# Exception -hierarki



# throw

- Java-maskineriet utløser spesifikk unntak i spesifikke feilsituasjoner f.eks. når du prøver å gjøre ulovlige ting med null-verdier, deler på 0, refererer forbi enden av en tabell/array, osv.
- Du bør selv bruke throw når du oppdager at noe er galt, som ikke kan håndteres på en god måte der problemet oppdages, f.eks. bruke `throw new IllegalArgumentException("...")`, dersom du oppdager at et parameter har en ugyldig verdi
- Du kan velge blant eksisterende Exception-klasser eller definere nye (krever bruk av arv)

```
try { ... }
```

```
catch { ... }
```

## oppsummert

- **throw** sier fra at et unntak har oppstått
- **try/catch** sier fra at vi i løpet av utførelsen av en kodesnutt kan håndtere en eller flere typer unntak, dersom de måtte oppstå
- Dersom et unntak oppstår og vi har en tilsvarende catch-del, vil denne bli utført, og så vil koden etter **try/catch**-blokken fortsette som normalt

# Boksmodellen

- metode1 kaller metode2 som kaller metode3
- metode1 har en **try/catch**-blokk
- Metode3 og metode2 bruker **throw** for å angi et unntak
- aktivering av metode3 og metode2 brytes og metode1 fortsetter i **catch**-delen som passer til unntakstypen

metode3(int) : minKlasse

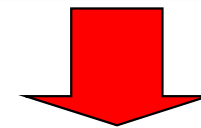
...

metode2(int) : minKlasse

...

metode1(int) : minKlasse

catch Exception



metode1(int) : minKlasse

...



# Valg av catch-blokk

- Unntaksobjektene er instanser av subklasser av **Exception**
- **catch**-blokken som fanger opp unntaket velges ved å finne den *nærmeste* som håndterer unntaks(super)typen
- **try/catch** kan ha *flere* **catch**-blokker, som prøves i tur og orden. Den første som passer brukes.
- *Anta at metode3 kaller **Double.valueOf(String)** og det gir et unntak av typen **NumberFormatException**. Hvilken metode vil da fange opp unntaket?*

```
metode3(String)
```

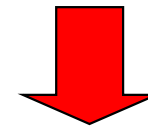
```
...
```

```
metode2(String)
```

```
catch IllegalArgumentException
```

```
metode1(String)
```

```
catch NumberFormatException
```



```
metode?(int)
```

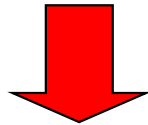
```
...
```

# Valg av catch-blokk

metode3(String) : minKlasse  
...

metode2(String) : minKlasse  
catch IllegalArgumentException

metode1(String) : minKlasse  
catch NumberFormatException



metode2(int) : minKlasse  
catch IllegalArgumentException

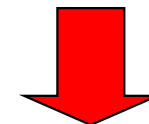
metode4(String) : minKlasse  
...

metode3(String) : minKlasse  
catch NumberFormatException

metode2(String) : minKlasse  
catch IllegalArgumentException

metode1(String) : minKlasse  
catch Exception

Hva hvis metode4  
utfører throw new  
IllegalArgumentException?



# Egendefinerte unntakstyper

- Egne unntakstyper kan defineres ved å lage en *subklasse* av **Exception** eller av en **Exception**-subklasse
- Eksempel, unntak for ulovlige radiuser

```
public class UlovligRadiusException
    extends IllegalArgumentException {

    public UlovligRadiusException(String s) {
        super(s);
    }
}
```

- En unntaksklasse er en helt vanlig klasse, og må følgelig definere konstruktører og bruke `super(...)` for å kalle superklassens konstruktør
- Egne unntaksklasser kan inkludere variabler med detaljer om hva som var feil, slik at **catch**-koden blir bedre informert

# RuntimeException vs. Exception

- **RuntimeException** (såkalt *unchecked*) er en subklasse av **Exception** for unntak som er uventede (i en eller annen forstand)

*Feil i koden eller feil bruk av koden*

- Andre typer, (såkalt *checked*) dvs. subklasser av **Exception** som ikke samtidig er subklasser av **RuntimeException** brukes for feil som er forventet (i en eller annen forstand)

*Forventede komplikasjoner, skapt av omgivelsene*

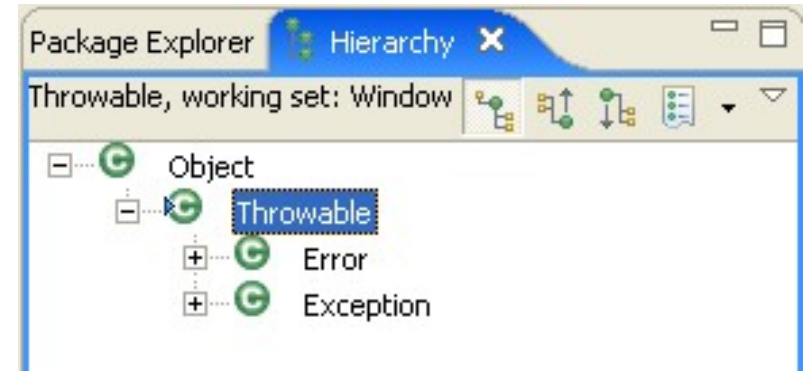
# Fra Liang (tidligere bok): Checked Exceptions vs. Unchecked Exceptions

**RuntimeException** and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

# throws

- To spesifikke krav stilles dersom en metode kan komme til å bli avbrutt av et *checked* unntak (som ikke er en **RuntimeEx.**):
  - den må eksplisitt deklare unntakstypen(e) vha. **throws** **<unntakstype>** etter parameterlista
  - metoder som kaller en metode med en slik deklarasjon, må enten fange opp unntaket vha. **try/catch** for denne unntakstypen eller selv deklare unntakstypen vha. en **throws**-deklarasjon
  - **IOException** f.eks., er checked og ikke ignoreres
- Ved riktig valg av superklasse for egendefinerte unntaksklasser, kan en altså tvinge kallende metoder til å enten håndtere unntaket eller si fra at unntaket kan oppstå

# La oss ta en titt på unntakshierarkiet



- Viktige **RuntimeException**-klasser
  - **NullPointerException** – gjøre noe med null
  - **ClassCastException** – prøver å cast'e verdi til ulovlig klasse
  - **IndexOutOfBoundsException** – negativ eller for stor index til tabell- eller String-operasjoner
  - **NoSuchElementException** – for mange next() på en Iterator
  - **UnsupportedOperationException** – angis at valgfri metode i grensesnitt ikke er implementert, f.eks. **remove()** i **Iterator**
  - **IllegalArgumentException** – feil ved validering av argument
    - **NumberFormatException** – feil i tallformat
  - Runtime er *unchecked*, resten av unntakene *unchecked*
- Viktige **Exception**-klasser
  - **IOException**

# java.io.IOException

- **IOException** er en **Exception** som er *checked* og derfor må fanges opp eller deklarereres at kan oppstå, dvs. enten
  - `try { ... } catch (IOException ioe) { ... },` eller
  - `throws IOException`
- IO-feil er såpass vanlige og alvorlige, og må derfor håndteres.
- Eksempel på lesing fra fil følger...



# Filinnlesingseksempel

- Vanlig kode for lesing av fil:

- lag Reader-objekt(er)
- les i løkke
- pass på lukke Reader-objektet til slutt

```
File fil = new File(filnavn);
FileReader fileReader = new FileReader(fil);
BufferedReader reader = new BufferedReader(fileReader);
String linje = null;
int count = 0;
while (reader.ready()) {
    linje = reader.readLine();
    count++;
    System.out.println("Linje nr. " + count + ": " + linje);
}
reader.close();
```

- Hvis noe går galt nå, vil ikke fila bli lukket ordentlig!

# Må sikre oss at reader.close() blir kalt

```
try {
    FileReader fileReader = new FileReader(fil);
    reader = new BufferedReader(fileReader);
    String linje = null;
    int count = 0;
    while (reader.ready()) {
        linje = reader.readLine();
        count++;
        System.out.println("Linje nr. " + count + ": " + linje);
    }
} catch (IOException ioe) {
}

try {
    reader.close();
} catch (IOException ioe) {
}
```

Hva skjer hvis vi  
underveis får en  
annen type Exception?

# finally

- **finally**-blokken i en **try/catch** sikrer at en bestemt kodesnutt alltid blir utført, uansett hva som skjer i **try**-delen

```
try {  
    // her skjer det noe rart  
} finally {  
    // rydd opp etter deg  
    // blir utført uansett om det oppstår  
    // unntak eller ikke  
}
```

# finally sikrer at reader.close() alltid blir kalt

```
try {
    FileReader fileReader = new FileReader(fil);
    reader = new BufferedReader(fileReader);
    String linje = null;
    int count = 0;
    while (reader.ready()) {
        linje = reader.readLine();
        count++;
        System.out.println("Linje nr. " + count + ": " + linje);
    }
} finally {
    try {
        reader.close();
    } catch (IOException ioe) {
    }
}
```

Hva skjer hvis vi får en feil  
*før* reader settes?

# Closeable

- En egen variant av **try/catch** gjør lukking av ressurser som implementerer **Closeable** enklere
- **// FileReader implementerer Closeable**  
**try (FileReader reader = new FileReader(...)) {**  
    **// her kan unntak bli utløst**  
**}**
- Trenger ingen **finally** her, fordi **reader** er **Closeable** og blir lukket automatisk

# Ikke bruk try/catch i utide

- ```
try {  
    Iterator it = liste.iterator();  
    while (true) {  
        Object o = it.next();  
        // gjør noe med o her  
    }  
} catch (Exception e) {  
}
```
- Her brukes **try/catch** som vanlig kontrollstruktur, hvor **it.hasNext()** skulle vært brukt. Fy, fy!

# Læringsmål for forelesningen

- Objektorientering
  - Håndtering av unntak (eng: exceptions)
- Java-programmering
  - Håndtering av unntak
    - Exception-objekter og –klasser
    - checked og unchecked
    - try, catch og finally
    - throw og throws
- VS Code
  - Se på klassehierarki med F4



# Har du nådd læringsmålene?

- Java-programmering
  - Lag en NorskeTall-klasse som parser norske tall skrevet med ord, f.eks. "en", "tjueen", "førtiseks" osv. for tall opp til 100.
  - Lag en egen type unntak som subklasser `NumberFormatException`, som `NorskeTall`-klassen bruker til å si fra om feilformattede tall. Det skal angis om feilen var for hele tallet (dvs. uforståelig), tierdelen (f.eks. "xxxen" eller enerdelen (f.eks. "tjueti").
  - Hva skjer når du endrer koden slik at du subklasser `Exception` istedenfor `NumberFormatException`, og hva må du gjøre?