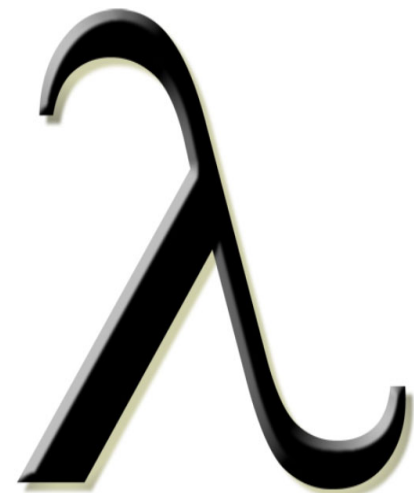


# ● Forelæsning Uge 5 – Torsdag

---

- **Funktionel programmering i Java (Kapitel 5)**
  - De dele af Java, som I har set indtil nu, er **imperative**
  - **Funktionelle** programmeringssprog fungerer på en helt anden måde, som vi skal se nærmere på om lidt
- **Forskellige slags objektsamlinger (Kapitel 6)**
  - **List** (liste) – kendt fra ArrayList
  - **Set** (mængde)
  - **Map** (afbildning / funktion)
  - Polymorfe variabler
- **Dokumentation af jeres egne klasser**



# ● Imperative og funktionelle sprog

---

- **De dele af Java, som I har set indtil nu, er imperative**
  - En udførsel af et program forstås som en række **operationer**, der ændrer **systems tilstand**, f.eks. via **assignments** til feltvariabler
  - Objekt-orienterede sprog (og de fleste andre programmeringssprog) er (primært) imperative
  - Eksempler på imperative sprog: Java, C#, C og C++
- **Funktionelle programmeringssprog fungerer anderledes**
  - En udførsel af et program forstås som en **evaluering** af et **matematisk udtryk** (uden brug af assignments)
  - Programmer skrevet ved hjælp af funktionel programmering er ofte kortere, mere letlæselige og nemmere at bevise korrekte
  - Eksempler på funktionelle sprog: Standard ML, OCaml, F#, Lisp, Haskell og Erlang
- **Moderne sprog er ofte både imperative og funktionelle**
  - Java indeholder **lambda**'er (som I skal lære om i denne forelæsning)
  - OCaml indeholder **mutable** data (som kan ændres med assignments)

# Lambda calculus

- **Funktionelle programmeringssprog bygger på lambda calculus**
  - Formalisme til beskrivelse af beregninger (introduceret i 1930)

- **Java**

```
public int addOne(int n) {  
    return n+1;  
}
```

Metode, der lægger 1 til parameteren  
**Funktion**  $\approx$  metode (synonymer)

- **Standard ML (funktionelt sprog)**

Tæt på den notation vi kender fra matematik

```
fun addOne(n) = n+1;
```

```
addOne(n) = n+1 for alle n
```

```
int  $\rightarrow$  int
```

Man behøver ikke at angive typerne  
Dem deducerer oversætteren selv

```
fn(n) => n+1;
```

Anonym funktion (uden navn)  
Kan bruges som parameter til  
en anden funktion

- **Lambda calculus**

```
 $\lambda n. n+1$ 
```

Datalogistuderende vil lære meget  
mere om  $\lambda$ -calculus i senere kurser

# Funktionelle aspekter i Java

---

- **Java er (primært) et imperativt programmeringssprog**
  - Men de nyere versioner af Java (fra og med version 8 i 2014) indeholder også aspekter fra funktionelle programmeringssprog
  - Det gør sproget mere kompliceret (fordi der er flere ting at lære)
  - Til gengæld kan man (som I snart skal se) udtrykke visse ting simple, mere elegant og mere læseligt
- **De funktionelle dele af Java vinder hurtigt indpas og er dermed et "must" for alle kompetente Java programmører**
  - De er bl.a. yderst velegnede til gennemsøgning og sortering af collections (objektsamlinger)
  - Om lidt vil vi se, at vi ved hjælp af funktionel programmering
    - kan omskrive vores fem algoritmeskabeloner, så de bliver mere kompakte og letlæselige
    - sortere uden selv at skulle skrive en compareTo eller compare metode

# ● Observationer af dyr (eksempel)

```
public class Sighting {  
    private final String animal;    // Which animal  
    private final int spotter;      // Who saw it  
    private final int count;        // How many  
    private final int area;         // Where  
    private final int period;       // When  
  
    public Sighting(String animal, int spotter,  
                     int count, int area, int period) {  
        this.animal = animal;  
        this.spotter = spotter;  
        ...  
    }  
  
    public String toString() {  
        return animal +  
            ", count = " + count +  
            ", area = " + area +  
            ", spotter = " + spotter +  
            ", period = " + period;  
    }  
    ...  
}
```

"Elephant, count = 24, area = 2, spotter = 3, period = 2"

- BlueJ bogen kalder metoden for getDetails
- Som vi skal se om et øjeblik, er det bedre at kalde den toString

# AnimalMonitor klassen

addAll metoden i ArrayList klassen tager en Collection (af Sighting objekter) som parameter og tilføjer dem bagerst i arraylisten



```
import java.util.ArrayList;  
public class AnimalMonitor {
```

```
    private ArrayList<Sighting> sightings;
```

```
    public AnimalMonitor() {  
        this.sightings = new ArrayList<>();  
    }
```

```
    // Add sightings from file
```

```
    public void addSightings(String filename) {  
        SightingReader reader = new SightingReader();  
        sightings.addAll(reader.getSightings(filename));  
    }
```

Returnerer en ArrayList<Sighting>

```
    public void printList() {  
        for(Sighting s : sightings) {  
            System.out.println(s);  
        }  
    }
```

println metoden kalder automatisk toString på s

```
    ...
```

Elephant, count = 24, area = 2, spotter = 3, period = 2

```
}
```

# ● Lambda'er i Java

- En lambda er en "kodestump"
  - Kan bruges som parameter værdi bruges i et metodekald
  - Den kaldte metode kan så udføre lambda'en ("kodestumpen")
  - Skellet mellem kode og data forsvinder

## Imperativ kode

```
public void printList() {  
    for(Sighting s : sightings) {  
        System.out.println(s);  
    }  
}
```

### For-each løkke

- Bruger kroppen på alle elementer

## Funktionel kode

```
public void printList() {  
    sightings.forEach(s -> System.out.println(s));  
}
```

Lambda



forEach er en metode i ArrayList klassen (og andre collections)

- Tager en lambda som parameter
- Bruger lambda'en på alle elementerne i arraylisten (en efter en)

# Java syntaks for Lambda'er

---

- Den generelle syntax er som følger

```
(P p, Q q, ...) ->
{
    code;
}
```

- **Simplifikationer**

- Vi kan (i de fleste tilfælde) udelade typerne på parametrene, idet oversætteren selv kan deducere dem
- Hvis der kun er én parameter (uden typeangivelse) kan vi udelade ( )
- Hvis kroppen kun har én sætning kan vi udelade { } og semikolonnet

```
p -> code
```

Én parameter

```
(p, q, ...) -> code
```

Flere parametre

- **Eksemplet fra før**

```
sightings.forEach( s -> System.out.println(s) );
```

↑  
Lambda



# ● Streams i Java (interfacet Stream<T>)

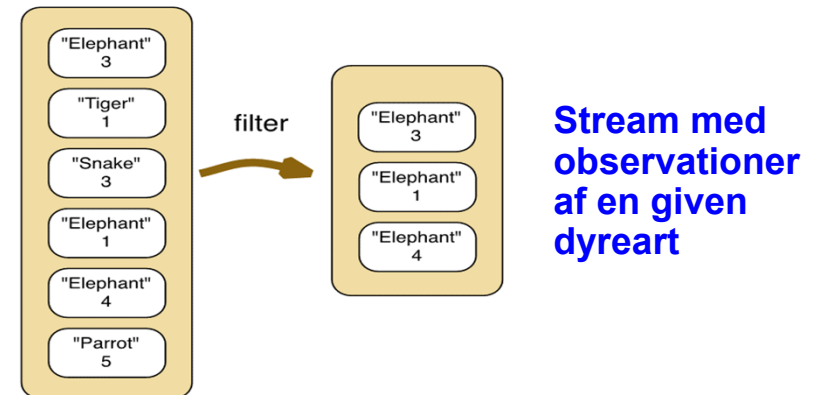
---

- **En stream er sekvens af data, f.eks.**
  - Elementerne i en Collection (f.eks. en arrayliste)
  - Tekstlinjer fra en tekstfil
  - Tegn (char værdier) fra en tekststreng (String)
  - Data der "strømmer" ind via et netværk
- **Karakteristika for streams**
  - Elementer tilgås ikke via et index (men i den rækkefølge, de kommer)
  - Streams er **immutable** (rækkefølgen og elementer kan ikke ændres), men man kan lave en ny stream ud fra den gamle
  - Streams kan være potentielt "uendelige" (have ubegrænset længde)
- **Elementer i en stream kan behandles parallelt på en multi-core maskine**
  - Potentiel stor **effektivitetsgevinst** uden ekstra programmeringsindsats
- **En arrayliste er ikke en stream**
  - Men ArrayList klassen har en metode, som skaber en stream ud fra arraylistens elementer (analogt for andre collections)

# Streams har tre vigtige metoder (funktioner)

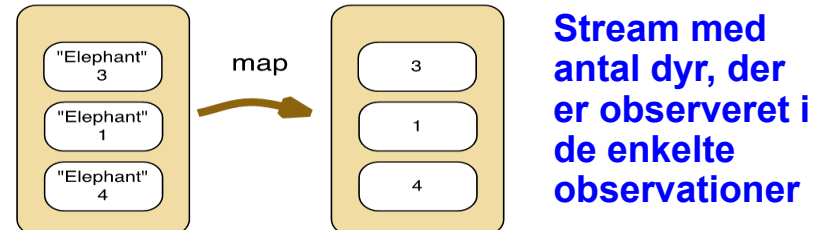
- filter funktionen**

- Gennemløber en stream og skaber en ny stream indeholdende de elementer fra den gamle, som opfylder en given betingelse



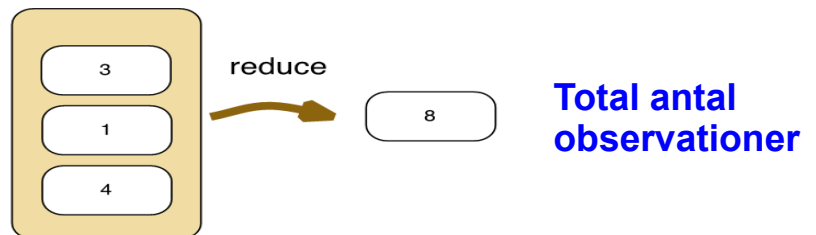
- map funktionen**

- Gennemløber en stream og skaber en ny stream ved at bruge en lambda på hvert element i den gamle stream



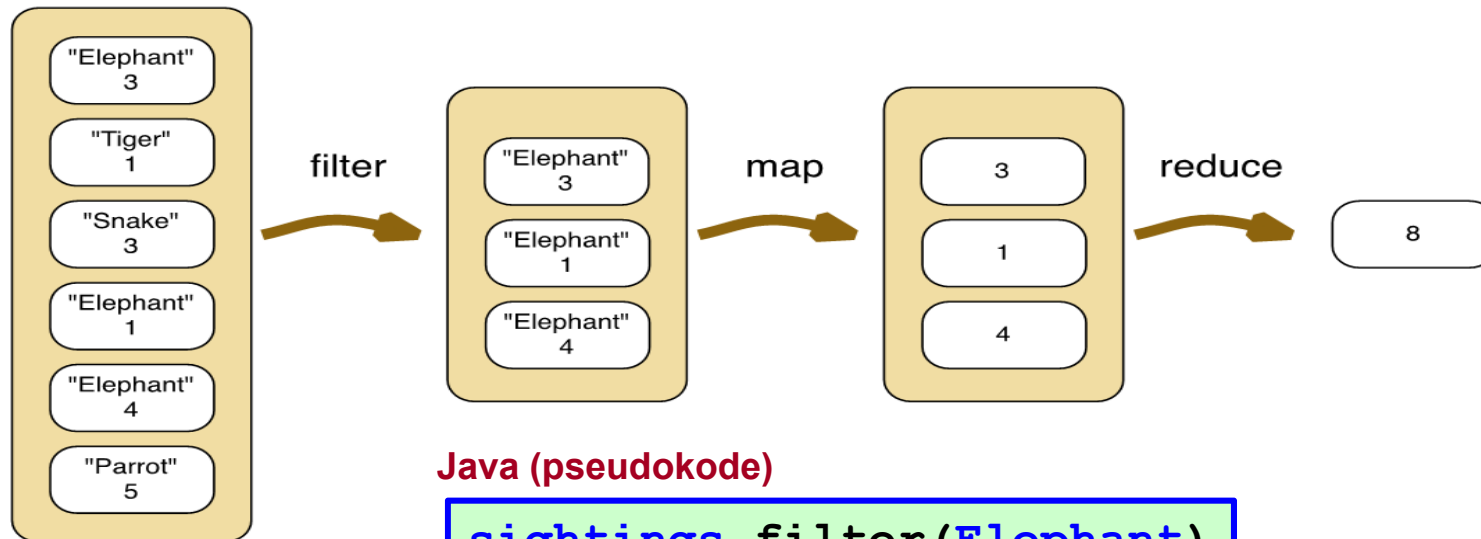
- reduce funktion**

- Gennemløber en stream og returnerer en enkelt værdi (f.eks. ved at lægge alle værdierne i stream'en sammen)



# Pipelines (sammensætning af funktioner)

- **Stream funktioner kan sættes sammen til en pipeline**
  - Nedenstående pipeline beregner hvor mange elefanter der er observeret



Java (pseudokode)

```
sightings.filter(Elephant)
          .map(count)
          .reduce(sum) ;
```

- **For at få eksekverbar Java kode mangler vi to ting**
  - Arraylisten sightings skal "omdannes" til en stream
  - Parametrene til filter, map og reduce funktionerne skal formaliseres

# Opbygning af pipelines

---

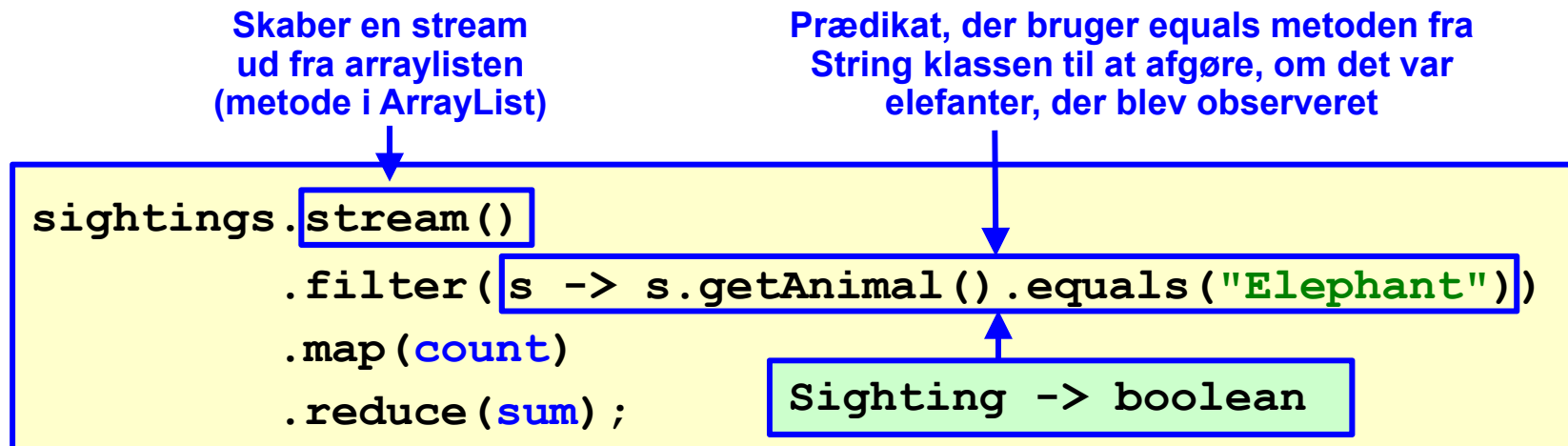
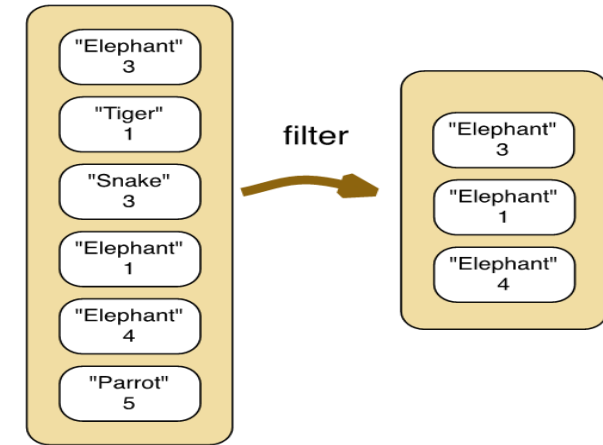
- **Pipelines er opbygget af**
  - en source (kilde)
  - et antal intermediate (mellemliggende) operationer
  - en terminal (afsluttende) operation, som producerer en værdi (eller har resultattypen void)
- **Hver intermediate operation producerer en ny stream**
- **Eksemplet fra før**
  - sightings er kilden
  - filter og map er intermediate
  - reduce er terminal
- **Man kan nemt lave andre beregninger**
  - Hvad gør denne pipeline?

```
sightings.filter(Elephant)  
          .map(count)  
          .reduce(sum) ;
```

```
sightings.filter(spotterID)  
          .filter(dayID)  
          .map(count)  
          .reduce(sum) ;
```

# Filter funktionen

- Gennemløber en stream og skaber en ny indeholdende de elementer fra den gamle, som opfylder en given betingelse
  - Intermediate operation
  - Udvælgelsen sker via et **prædikat** (predicate), dvs. en lambda med returtype boolean
  - Input stream ændres ikke (streams er immutable)
  - Den nye stream har samme type objekter som den gamle, men der er ofte færre

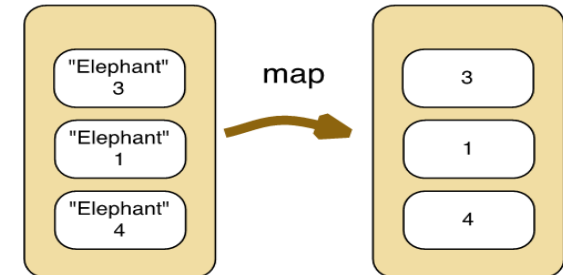


- Vi behøver ikke at specificere typen for variablen s
- Oversætteren ved at sourcen leverer Sighting objekter

# Map funktionen

- Gennemløber en stream og skaber en ny ved at bruge en **lambda** på hvert element i den gamle stream

- Intermediate operation
- Mapningen sker ved hjælp af en **lambda**
- Input stream ændres ikke (streams er immutable)
- Den nye stream har samme antal objekter som den gamle, men de er ofte af en anden type



```
sightings.stream()  
    .filter( s -> s.getAnimal().equals("Elephant"))  
    .map( s -> s.getCount() )  
    .reduce( sum );
```

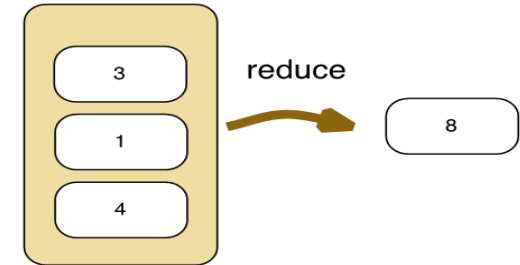
Sighting -> int

- Vi behøver ikke at specificere typen for variabelen s
- Oversætteren ved at sourcen, og dermed filter metoden, leverer Sighting objekter
- Den nye stream er af typen Stream<Integer>

# Reduce funktionen

- **Gennemløber en stream og returnerer én værdi**

- Terminal operation
- Metoden har to parametre
  - Første parameter er en startværdi
  - Anden parameter er en lambda med to parametre, hvor den første er det hidtidige mellemresultat, mens den anden er det element, der pt behandles
- Input stream ændres ikke (streams er immutable)



```
sightings.stream()  
    .filter(s -> s.getAnimal().equals("Elephant"))  
    .map(s -> s.getCount())  
    .reduce(0, (result, elem) -> result + elem);
```

Startværdi

Lambda: `int * int -> int`

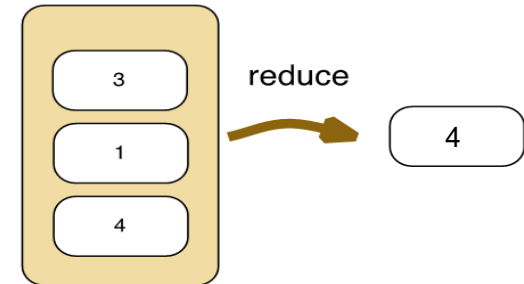
- **result** initialiseres til startværdien og holder det foreløbige resultat
- For hvert element **elem** i stream'en bruges lambda'en til at beregne den nye værdi af **result**

$(0, 3) \rightarrow (3, 1) \rightarrow (4, 4) \rightarrow 8$

I dette tilfælde findes **summen** af heltallene i stream'en

# Alternativ reduce funktionen

- Det **maksimale** antal elefanter, set i en enkelt sighting, kan beregnes ved at ændre den lambda, der gives som parameter til reduce



```
sightings.stream()  
  .filter(s -> s.getAnimal().equals("Elephant"))  
  .map(s -> s.getCount())  
  .reduce(0, (result, elem) -> Math.max(result, elem));
```

Startværdi

Lambda: `int * int -> int`

- **result** initialiseres til startværdien og holder det foreløbige resultat
- For hvert element **elem** i stream'en bruges lambda'en til at beregne den nye værdi af **result**

$(0, 3) \rightarrow (3, 1) \rightarrow (3, 4) \rightarrow 4$

I dette tilfælde findes det **maksimale** af heltallene i stream'en



# Færdig metode (med streams og lambda'er)

```
/**
 * Return the number of sightings of the specified animal.
 * @param animal Type of animal.
 * @return Count of sightings of the given animal.
 */
public int getCount(String animal) {
    return sightings.stream() ● Stream<Sighting>
        ● ArrayList<Sighting> .filter( s -> s.getAnimal().equals(animal)) ● Stream<Sighting>
        .map( s -> s.getCount() ) ● Stream<Integer>
        .reduce( 0 , (result, elem) -> result + elem ); ● int
}
```

Vores pipeline (med parameteren **animal** indsat i stedet for konstanten **"Elephant"**)

- Hvis man vil have parallel eksekvering af elementerne i stream'en og dermed åbne op for multi-core processing, skal man erstatte **stream()** med **parallelStream()**
- Giver kun en tidsgevinst, hvis man har **mange elementer**

# Andre Stream metoder

---

- **Stream klassen har ca. 40 forskellige metoder, hvoraf vi i det følgende vil bruge nedenstående**
  - **count** returnerer antallet af elementer i en Stream
  - **findFirst** returnerer første element i en stream af typen `Stream<T>` som et objekt af typen **`Optional<T>`**
- **`Optional<T>` er et alternativ til at bruge **null** til at angive, at man ikke har et objekt**
  - Buges i de funktionelle dele af Java
  - Metoden **isPresent** fortæller, om der er et T objekt eller ej
  - Hvis der er et T objekt, kan dette hentes via metoden **get**
  - Optional objekter er **immutable**
- **Optional klassen har metoder til at arbejde videre med Optional værdier**
  - **orElse(T other)** returnerer det objekt, der er gemt i Optional objektet (hvis der er et sådan) og ellers værdien af parameteren **other**
  - **orElse(null)** giver os det objekt, der er gemt i Optional objektet (hvis der er et sådan) og ellers **null**

# IntStream

- **Stream klassen har en metode, der kan producere en IntStream**
  - **mapToInt** producerer en IntStream ud fra en Stream (ved hjælp af en brugerspecificeret lambda, der mapper hvert enkelt element i et heltal)
- **Det er vigtigt at skelne mellem IntStream og Stream<Integer>**
  - Begge er en sekvens af heltal, men IntStream har nogle metoder, som en "almindelig" Stream ikke har
  - **sum** returnerer summen af elementerne
  - **min** og **max** returnerer mindste og største element (som en OptionalInt)
  - **average** returnerer gennemsnittet (som en OptionalDouble)
- **Ved at bruge en IntStream, kan vi ofte slippe for at skrive vores egen reduce metode**

Hvorfor findes de ikke i Stream<Integer>?  
Hvorfor returnerer de sidste en Optional?

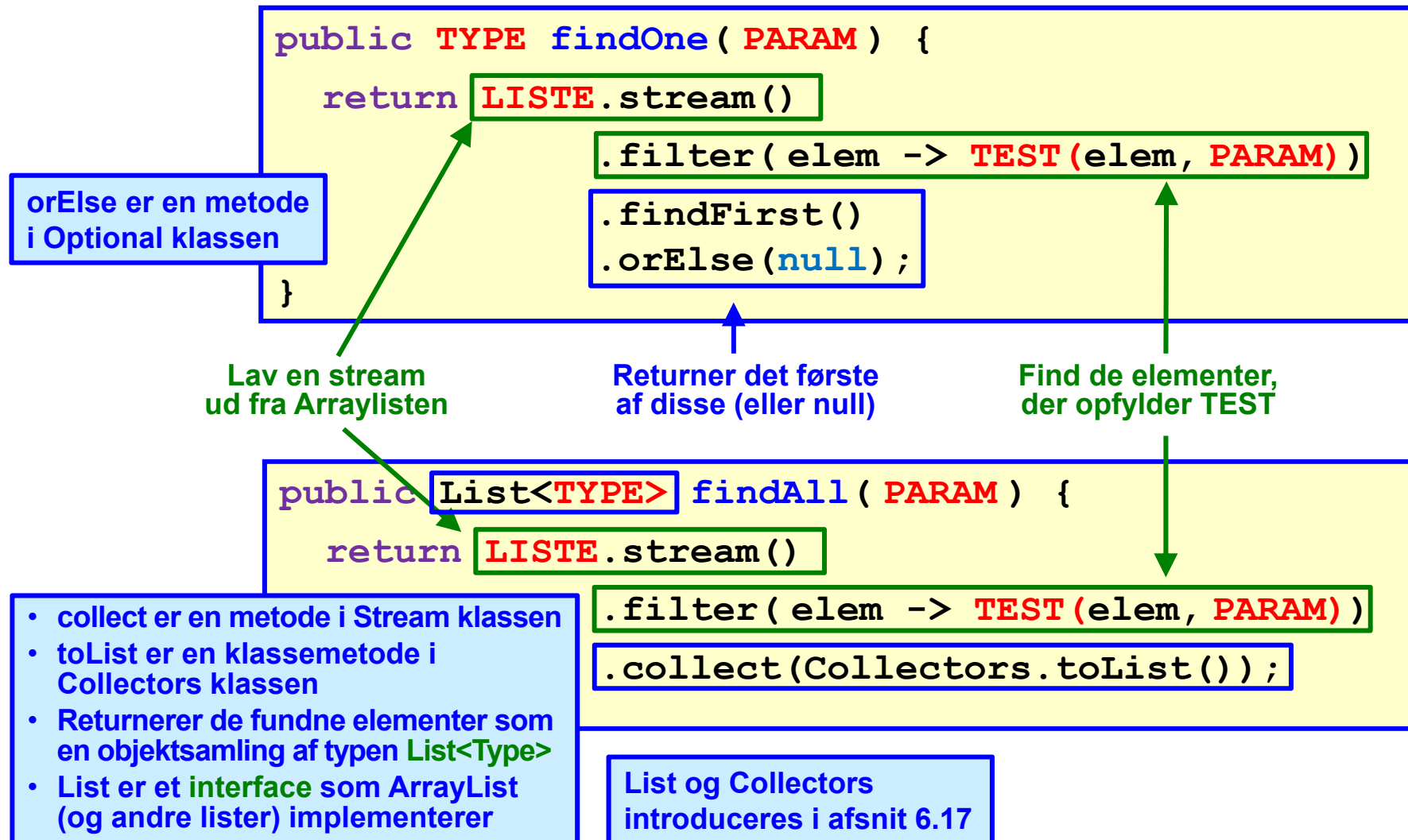
```
.mapToInt ( s -> s.getCount() ) ● IntStream  
.sum () ;
```

```
.mapToInt ( s -> s.getCount() ) ● IntStream  
.max () ;
```

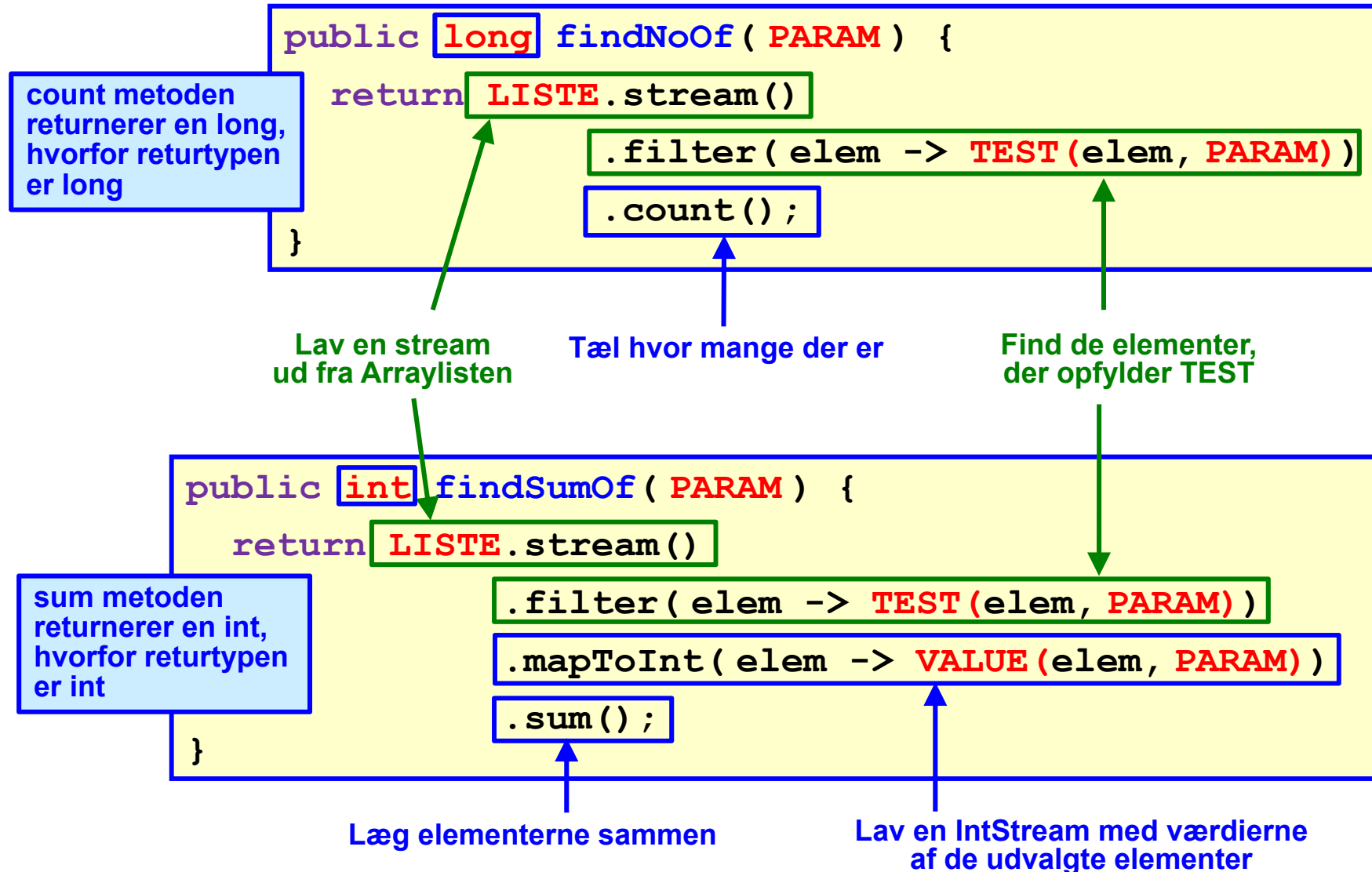
Analogt, kan man mappe en Stream til en DoubleStream eller en LongStream

# ● Algoritmeskabelonerne, findOne + findAll

- Vores fem algoritmeskabeloner kan implementeres via streams og lambda'er



# findNoOf og findSumOf



# findBest

Find de elementer  
der opfylder TEST

```
public TYPE findBest( PARAM ) {  
    return LISTE.stream()  
        .filter( elem -> TEST( elem, PARAM ) )  
        .max( new BEST() )  
        .orElse( null );  
}
```

Lav en stream  
ud fra Arraylisten

Returner bedste element (eller null)

- Ordningen bestemmes ved hjælp af et objekt fra en klasse **BEST**, der implementerer Comparator interfacet
- Men er det ikke "bøvlet" at skulle lave en sådan klasse?

- **Comparator interfacet har en klassemetode, der kan gøre det for os**

```
public Dog oldestDog() {  
    return dogs.stream()  
        .max( Comparator.comparing( d -> d.getAge() ) )  
        .orElse( null );  
}
```

Klassemetode

- **Comparing** tager en **lambda** som parameter og returnerer et **Comparator objekt** (dvs. et objekt fra en klasse, der implementerer Comparator interfacet)
- Parameteren "udpeger" den feltvariabel, hvis værdier skal sammenlignes (ved hjælp af den naturlige ordning)
- Hvis man vil finde den yngste hund, ændres max til min

# Sammenligning af algoritmeskabelonerne

- De funktionelle er mere **kompakte og mere ens** end de imperative

De to første linjer i kroppen er helt ens

Det er kun de sidste 1-2 linjer og returtypen, der er forskellige

```
public ??? findXXX( PARAM ) {  
    return LISTE.stream()  
        .filter( elem -> TEST( elem, PARAM ) )  
    . ???  
}
```

- Mange synes, at det er svært at huske, hvor de forskellige metoder ligger
- Men det skal man heldigvis heller ikke
- Det fremgår af den sammenhæng, de anvendes i

**findOne**

```
.findFirst()  
.orElse( null );
```

TYPE

**findAll**

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

List<TYPE>

**findNoOf**

```
.count();
```

long

**findSumOf**

```
.mapToInt( elem -> VALUE( elem, PARAM ) )  
.sum();
```

int

**findBest**

```
.max( Comparator.comparing( elem -> elem.getField() ) )  
.orElse( null );
```

TYPE

**Pause**

For at bruge de funktionelle skabeloner skal man importere Collections, Comparator, og Optional via `import java.util.*` og Collectors via `import java.util.stream.Collectors`

Ved køreprøven skal de to sidste opgaver løses ved hjælp af **funktionel** programmering, dvs. Streams og lambda'er (samt de funktionelle algoritmeskabeloner)

# ● Sortering

- Indtil nu har vi sorteret ved at skrive en compareTo metode
  - For Person klassen ser dette ud, som vist nedenfor
  - Vi sorterer efter alder og hvis to personer er lige gamle alfabetisk efter navn

```
public int compareTo(Person p) {  
    if(this.age != p.age) {  
        return this.age - p.age;  
    }  
    // Alderen er identisk  
    return this.name.compareTo(p.name);  
}
```

← Fastlæggelse af ordning  
via compareTo metode

```
public void printPersons() {  
    Collections.sort(persons);  
    for(Person p : persons {  
        System.out.println(p);  
    }  
}
```

← Sortering (via den naturlige ordning  
fastlagt af vores compareTo metode)

← Udskrift af den  
sorterede arrayliste



# Funktionel sortering version 1

- Som vi har set, har **Comparator** interfacet en klassemetode, der gør det let at definere en ordning uden selv at skrive en **compareTo** metode
  - For **Persons** kan dette anvendes, som vist nedenfor
  - Som før sorteres efter alder, og hvis to personer er lige gamle alfabetisk efter navn

```
public void printPersons() {  
    Collections.sort(persons, Comparator.comparing(p -> p.getName()));  
    Collections.sort(persons, Comparator.comparing(p -> p.getAge()));  
    persons.forEach(p -> System.out.println(p));  
}
```

Udskrift af den sorterede arrayliste

Vi sorterer to gange

- Vil den sidste sortering ikke blot ødelægge den første?
- Nej, sorteringerne er stabile (stable). Det betyder, at de kun bytter om på elementer, når det er nødvendigt
- Derfor vil den første sortering stadig være gældende for de elementer, der har samme ordning i den anden sortering

- Sortering via to **Comparator** klasser (der anvender den naturlige ordning for henholdsvis **String** og **int**)
- Klassemetoden **comparing** returnerer et **Comparator** objekt
- Bemærk, at vi starter med det mindst betydende kriterie og slutter med det mest betydende
- Hvis man vil have de ældste først sætter man et **minus** på lambda'ens højre side

# Funktionel sortering version 2

- **Man kan nøjes med én sortering**
  - Før brugte vi to forskellige Comparator objekter, hvor det ene sorterede efter navn og det anden efter alder
  - Nu bruger vi **ét** Comparator objekt, der primært sorterer efter alder og sekundært efter navn

Klassemetoden **comparing** returnerer et **Comparator objekt**

Nu er det nødvendigt at hjælpe oversætteren ved at angive p's type

```
public void printPersons() {  
    Collections.sort(persons, Comparator.comparing((Person p) -> p.getAge())  
                                         .thenComparing(p -> p.getName()));  
    persons.forEach(p -> System.out.println(p));  
}
```

- Bemærk, at **comparing** er en klassemetode, mens **thenComparing** er en almindelig metode
- Det fremgår af den sammenhæng, som de anvendes i

- **Metode i Comparator interfaceet, der kaldes på det Comparator objekt, som comparing returnerer**
- **thenComparing returnerer også et Comparator objekt, og det er dette der bestemmer den ordening, der sorteres efter**

Bemærk også, at vi nu starter med det mest betydende kriterie og slutter med det mindst betydende (hvilket gør koden mere logisk og lettere at forstå)

# Funktionel sortering version 3 og 4

- I stedet for at sortere arraylisten kan vi sortere en stream

```
public void printPersons() {  
    persons.stream()  
        .sorted(Comparator.comparing((Person p) -> p.getAge())  
                .thenComparing(p -> p.getName()))  
        .forEach(p -> System.out.println(p));  
}
```

Metode i Stream klassen (fungerer analogt til sort metoden i ArrayList klassen)

- Bemærk at arraylisten og den Stream, der produceres ud fra den ikke ændres.
- Metoden sorted returnerer en ny Stream, der er sorteret (som angivet af Comparator objektet)

- Derudover kan vi erstatte de tre lambda'er med metode referencer

```
public void printPersons() {  
    persons.stream()  
        .sorted(Comparator.comparing(Person::getAge)  
                .thenComparing(Person::getName))  
        .forEach(System.out::println);  
}
```

Forkortelse for lambda'en  
(Person p) -> p.getAge()

- Metode referencer er beskrevet på side 219-220 i BlueJ bogen
- De kan også bruges i findBest algoritmeskabelonen

Forkortelse for lambda'en  
p -> System.out.println(p)

Forkortelse for lambda'en  
(Person p) -> p.getName()

reversed() metoden i Comparator klassen returnerer et Comparator objekt med omvendt sortering

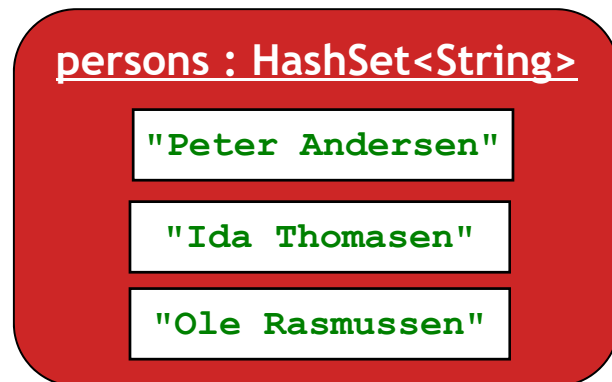
# ● Set (mængde)

---

- **Matematisk mængde**

- Et element kan højst forekomme **én gang** i mængden
- Indsætter man elementet en gang til, har det ingen effekt
- Der er mange forskellige implementationer af mængder – på samme måde, som der er forskellige implementationer af lister
- Her vil vi se på **HashSet<E>** klassen

- **En mængde af personnavne kan modelleres via HashSet<String>**



# Implementation af mængde af personer

```
import java.util.HashSet;
...
// Oprettelse af mængde
HashSet<String> persons = new HashSet<>();

// Indsættelse af personnavne
persons.add("Peter Andersen");
persons.add("Ida Thomasen");
persons.add("Ole Rasmussen");
System.out.println(persons.size());
...
// Indsæt et navn, der allerede er i mængden
persons.add("Ida Thomasen");
System.out.println(persons.size());
```

- add metoden indsætter elementer
- returnerer true, hvis mængden ændres

3

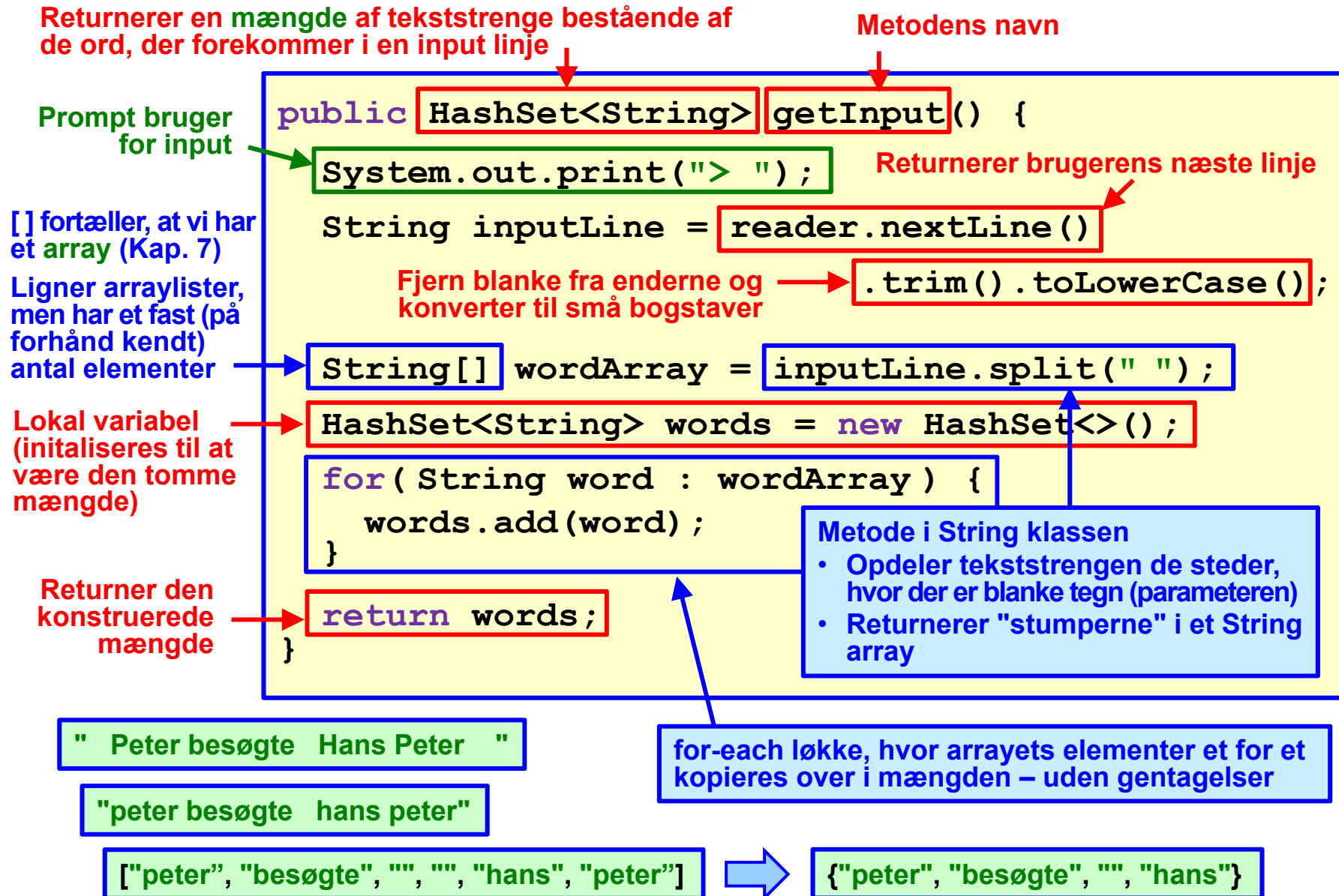
- size metoden fortæller, hvor mange elementer, der er i mængden

3

- add metoden returnerer false, hvis mængden ikke ændres
- size ændres ikke

- Det er **equals** metoden (for element typen E), der bruges til at afgøre, om elementet allerede forekommer i mængden
  - Object klassen (som alle klasser er underklasser af) har en equals metode
  - Dette sikrer at alle klasser har en equals metode

# Eksempel: Indlæsning af kommandoer



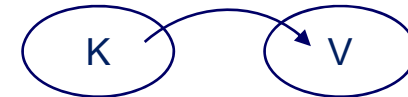
# ● Map (afbildning / funktion)

---

- **Matematisk funktion (afbildning) fra en mængde til en anden**

- Der er mange forskellige implementationer af maps – på samme måde, som der er forskellige implementationer af lister og mængder
- Her vil vi se på **HashMap<K, V>** klassen

- **Parametriseret klasse med 2 type parametre**



- Første parameter **K** angiver **keys (nøgler)** – den type der afbildes fra
- Anden parameter **V** angiver **values (værdier)** – den type der afbildes til

- **Et Map objekt indeholder par på formen (k,v), hvor k er af typen K og v af typen V**

- Hvis man kender nøglen **k**, kan man slå værdien **v** op (ved hjælp af Map objektet)
- En værdi **v** kan være knyttet til flere nøgler (afbildningen behøver ikke være injektiv)
- Omvendt har en nøgle højst én tilknyttet værdi (ellers ville det være en relation og ikke en afbildning)

# Telefonliste

---

- **En telefonliste er et typisk eksempel på brug af Map**
  - K er personer, mens V er deres telefonnumre
  - Begge kan repræsenteres som tekststreng (String)

<u>contacts : HashMap&lt;String, String&gt;</u>	
"Peter Andersen"	"2674 5681"
"Ida Thomasen"	"4525 2512"
"Ole Rasmussen"	"hemmeligt"

- **Alternativt kan man bruge HashMap<String, Integer>**
  - Nu er værdierne heltal (og hemmeligt nummer angives som 0)

<u>contacts : HashMap&lt;String, Integer&gt;</u>	
"Peter Andersen"	26745681
"Ida Thomasen"	45252512
"Ole Rasmussen"	0



# Implementation af telefonliste

```
import java.util.HashMap;
...
// Oprettelse af kontaktliste
HashMap<String, String> contacts = new HashMap<>();
// Operettelse af kontakter
contacts.put("Peter Andersen", "2674 5681");
contacts.put("Ida Thomasen", "7412 3716");
contacts.put("Ole Rasmussen", "hemmeligt");
contacts.put("Ida Thomasen", "4525 2512");
// Opslag i kontaktlisten
String number = contacts.get("Ida Thomasen");
System.out.println(number);
```

- put metoden indsætter et nyt par
- Hvis nøglen allerede er i brug glemmes det gamle par

"4525 2512"

- get metoden laver opslag
- Returnerer den værdi, der er knyttet til den anvendte nøgle (null hvis nøglen ikke er i brug)

- **Andre metoder i HashMap**

- **size** metoden fortæller, hvor mange par, der er i afbildningen
- **keySet** metoden returnerer en **mængde** indeholdende alle de nøgler (keys), der er i brug
- I alt er der ca. 20 metoder (kan ses i Java API'en)

# ● Collections (objektsamlinger)

---

- **Forskellige måder at gruppere objekter**
  - ArrayList, LinkedList, ... (lister / sekvenser)
  - HashSet, LinkedHashSet, TreeSet, ... (mængder)
  - HashMap, LinkedHashMap, TreeMap, ... (afbildninger / funktioner)
  - **Sidste del** af navnet angiver, om det er en liste, mængde (set) eller afbildning (map)
  - **Første del** af navnet angiver implementationsmetoden
  - Der er ca. 30 forskellige slags collections (objektsamlinger)
- **Alle collections er parametriserede typer**
  - Parametrene skal være objekt typer
  - For de primitive typer bruges de tilsvarende wrapper typer
- **Alle collections bruger de samme metodenavne**
  - F.eks. **size**, **clear**, **isEmpty**, **get** og **remove**
  - Bemærk dog, at man i lister og mængder indsætter via **add** metoden, mens man i maps indsætter via **put** metoden

# Polymorfe variabler

---

- Når vi skriver et program, behøver vi ikke fra start at fastlægge, hvilken type objektsamling vi vil anvende

- I stedet for at erklære en variabel til at referere til en arrayliste

```
ArrayList<Person> persons;
```

kan man med **stor fordel** nøjes med at angive, at den refererer til en liste

```
List<Person> persons;
```

Variablen persons er **polymorf**, fordi den kan pege på værdier af **forskellig type**

- Man kan så senere let **udskifte** en liste implementation med en anden

- Det eneste sted man skal ændre i koden er der, hvor listen oprettes. Her angiver man, hvilken liste implementation, man vil bruge

```
persons = new ArrayList<>();
```

```
persons = new LinkedList<>();
```

- Tilsvarende kan vi bruge polymorfe variabler for mængder og afbildninger

```
Set<Person> persons;
```

```
Map<Person, Person> farthers;
```

List, Set og Map er **interfaces** (som vi skal kigge nærmere på i Kap. 12)

# ● Dokumentation

- Når I fremover konstruerer en klasse **skal** den dokumenteres lige så godt som klasserne i Javas API
  - Ellers får I genaflevering
  - Gælder dog **ikke** køreprøvesættene

- **For klassen skal I angive**

- En kommentar der beskriver klassens overordnede formål og virkemåde (se eksempler i Javas API)
- Versionsinfo (som bør indeholde datoen)
- Forfatterens navn(e)

```
/**  
 * Comment  
 */
```

```
@version 2019-12-24
```

```
@author Kurt Jensen
```

- **For hver konstruktør/metode skal I angive**

- En kommentar der beskriver virkemåden
- Beskrivelse af de enkelte parametre
- Beskrivelse af den returnerede værdi

```
@param animal Type of animal.
```

```
@return List of all sightings.
```

Skift fra **Source code** til **Documentation** view (i BlueJ editoren) for at kontrollere, at jeres dokumentation ser fornuftig ud

```
/**  
 * Comment  
 */
```

- Første sætning bruges i "Summary"-delen
- Hele kommentaren bruges i "Details"-delen

# ● Opsummering

---

- **Funktionel programmering i Java (Kapitel 5)**

- Forskellen på imperative og funktionelle programmeringssprog
- Lambda'er (kdestumper, der kan bruges som parametre i et metodekald)
- Streams (sekvenser / strømme af data)
- De fem algoritmeskabeloner implementeret ved hjælp af streams og lambda'er
- Sortering ved hjælp af lambda'er

- **Forskellige objektsamlinger (Kapitel 6)**

- Liste (kendt fra ArrayList)
- Sæt (mængde)
- Maps (afbildning / funktion)
- Polymorfe variabler

- **Dokumentation af jeres egne klasser**

## Køreprøven

- Opgave 1-10 skal løses ved hjælp af **imperativ programmering**. Man må altså **ikke** bruge streams og lambda'er
- Opgave 11-12 skal løses ved hjælp af **funktionel programmering**, dvs. streams, lambda'er (og de funktionelle algoritmeskabeloner)
- Testserveren bruges **ikke** under køreprøven

Husk at I **SKAL** tjekke køreprøveopgaverne ved hjælp af **testserveren**, før I afleverer dem

- Så ved I, at de virker
- Instruktorerne kigger på koden i testserveren

# Resten af kapitel 6 i BlueJ bogen

---

- **Kapitel 6 er forholdsvis langt, men det indeholder mange ting, som I allerede er stødt på her i kurset og derfor vil have let ved at læse**
  - Læsning og skrivning af Java dokumentation
  - Brug af klassen Random til at generere tilfældige tal
  - Import af klasser og pakker fra Javas klassebibliotek
  - Automatisk konvertering af værdier mellem primitive typer og de tilhørende wrapper klasser
  - Brug af nøgleordene public og private
  - Klassevariabler og klassemetoder (static)
  - Konstanter (final)
- **Læs kapitlet grundigt – uden at springe afsnit over**
  - Det er nyttig repetition og tilføjer **mange nye detaljer**

# Status

---

- **Forelæsninger**
  - Dagens forelæsning er den sidste før køreprøve og efterårsferie
  - I har nu haft 10 forelæsninger og mangler kun 7
- **Når I når frem til køreprøven, har I afleveret**
  - 6 programmeringsopgaver (Raflebæger, Skildpadde og Billedredigering)
  - 7 køreprøvesæt
  - 5 quizzer
- **I mangler så kun 7 programmeringsopgaver**
  - De er noget større end dem, som I hidtil har haft, men de enkelte dele er ikke meget sværere
  - Al erfaring viser, at hvis I kan klare opgaverne frem til køreprøven, kan I også klare de sidste syv
  - Der er stort set ingen, der falder fra i kursets sidste halvdel

# Forberedelse til køreprøven

---

- **Husk at se videoerne om køreprøvesættene**
  - Hvis du ikke allerede har set videoerne om Phone og Pirate er det **på høje tid**, at du ser dem nu
  - Er du usikker på brugen af algoritmeskabeloner bør du også se Car og Turtle
  - Se også Penguin (der løses ved hjælp af funktionel programmering)
- **Det er ikke nok at se videoerne**
  - Efter hvert sæt, bør I **selv** prøve at løse opgaverne
  - Hvis det kniber, ses videoerne igen
  - Bliv ved, indtil I kan løse sættet hurtigt og sikkert (tag tid)
- **Løs tidligere opgavesæt**
  - Kan findes på Brightspace siden "Køreprøvesæt fra tidligere år" under "Afleveringsopgaver"
  - Det er helt normalt, at det på nuværende tidspunkt tager 1 time at løse et opgavesæt
  - Til køreprøven kan de fleste studerende klare det på 30 minutter
  - Test din besvarelse ved hjælp af testserveren (gælder også de sæt, der er på videoerne, og de sæt, som I skal aflevere i uge 5 og 6)
- **Deltag i prøve-køreprøven ved første øvelsesgang i uge 7**



# Træning i mundtlig præsentation

---

- **I kurset sidste halvdel er der intensiv træning i, hvordan man går til mundtlig eksamen**
  - I skal hver især lave 2 præsentationer af et eksamensspørgsmål
  - Det er den eneste gang under jeres studier, at I får systematisk oplæring i og feedback omkring, hvordan man laver en god mundtlig præsentation
  - Kan få stort betydning for jeres fremtidige eksaminer og jeres fremtidige job
- **Held og lykke med køreprøven**
  - Hvis I forbereder jer godt, har I intet at frygte
  - De sidste år har 75% afleveret fuld besvarelse
  - Mere end 90% har fået mindst 4 tjekpunkter godkendt
  - Rekorden for fuld besvarelse er imponerende 9 minutter og 55 sekunder (indehaves af Mads Odgaard, der i år er instruktør for Hold 1)

**Det var alt for nu.....**

**... spørgsmål**

---

