Probe- / Klausur (siehe LSF)

Probeklausur

Di 15.07.2025, 14- 16 Uhr, WH-C 350

Klausur

Di 29.07.2025, 10- 12 Uhr, WH-C 350

Di 30.09.2025, 14:30-16:30, WH-C 350

Mit der Eingabe des Passworts in der Moodle-Klausur bestätigen Sie Ihre Prüfungsfähigkeit - ab diesem Moment gilt der Versuch als angetreten.

Ausweis mit Foto offen bereitlegen

Sie dürfen ein beidseitig von Hand beschriebenes Din A4 Blatt verwenden, sowie ein leeres Blatt und Stift um sich zwischendurch Notizen zu machen.

Kommunikation mit anderen über jedwede Mittel während der Prüfung ausser mit den Betreuern/Prüfern ist verboten und kann als Betrugsversuch gewertet werden.

Das gleiche gilt für das Verwenden weitere Hilfsmittel als der oben genannten.

Betrugsversuche führen zu einem Ausschluss aus der Prüfung und Bewertung der Prüfung mit 5,0

der Raum darf während der Prüfung nicht verlassen bzw. nach verlassen nicht wieder betreten werden.

Allowed:

- 1 DIN A4 sheet, handwritten on both sides
- single sheets of paper for notes, pen
- Water in a sealable bottle
- Clock
- Everything else has to be in the bag, turn off your phone and put it in your bag/backpack
- ID with photo should be placed openly

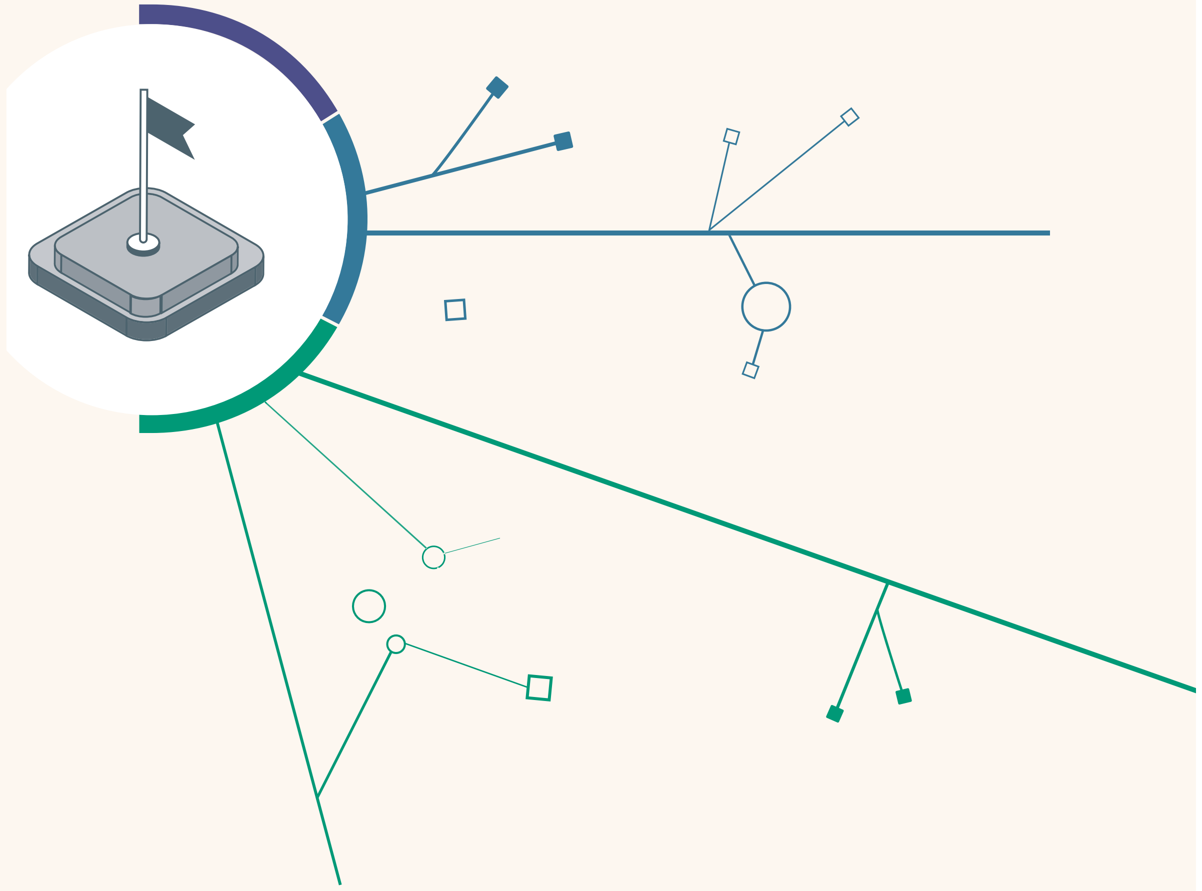
Rules:

- Communication with others using any means during the exam, except with supervisors/examiners, is prohibited and may be considered an attempt at fraud.
- The same applies to the use of additional tools other than those mentioned above.
- Attempts at fraud will result in exclusion from the exam and the exam being rated with 5.0.
- The room may not be left and re-entered during the exam

By entering the password in the Moodle exam, you confirm your ability to take the exam (Prüfungsfähigkeit) - from this moment, the attempt(Versuch) is

Agenda

- Probeklausur / Klausur
- Procedural vs declarative programming
- **Lambda: declarative implemention of tasks associated with iteration over collections**
- Streams: filter, map, reduce



provide an alternative means of implementing tasks associated with iteration over collections.

https://de.wikipedia.org/wiki/Boilerplate-Code

Klassifikation

Programmiersprachen

Abstrahierungsgrad

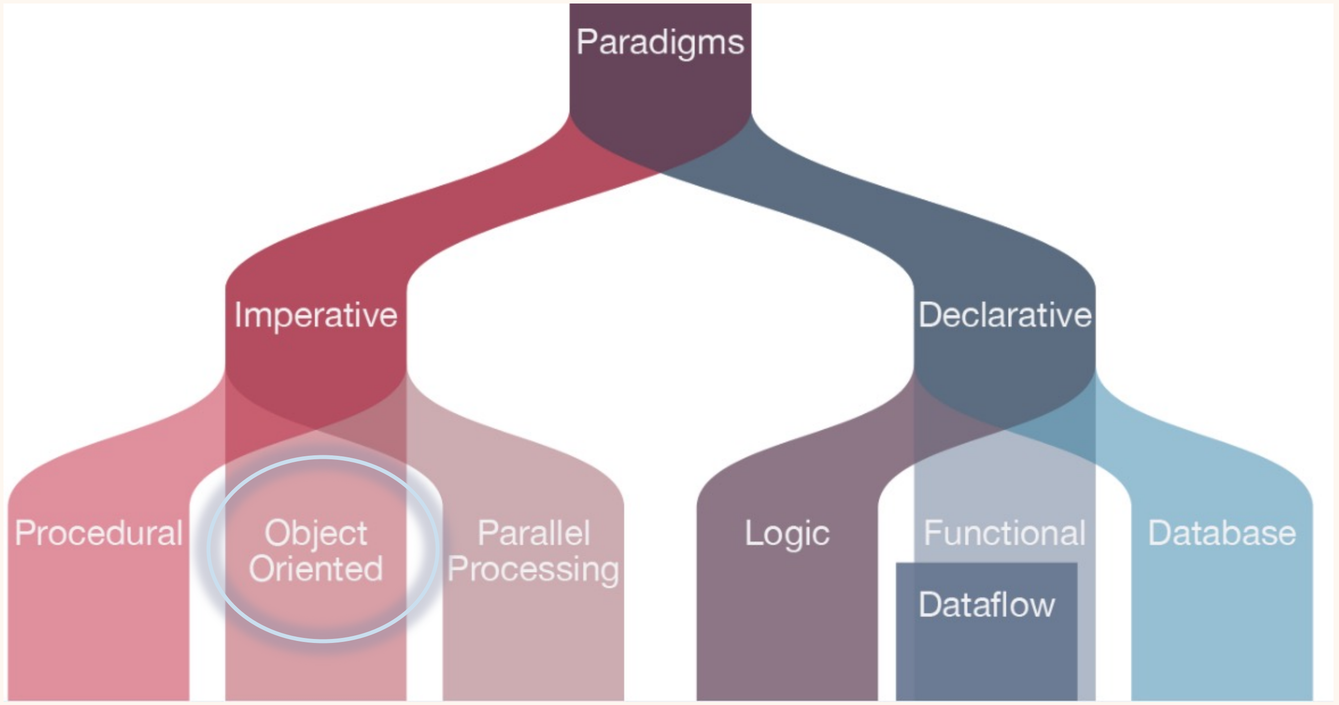
- maschinennah
- anwendungsorientiert

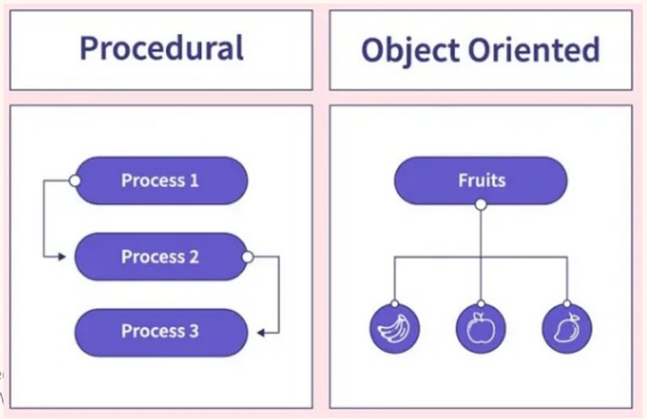
Komplexität

- problemspezifisch
- „general-purpose“

Paradigmen

- imperativ
- deskriptiv





https://medium.com/@kalyanasundaramthivaharan/object-oriente
programming-oop-and-procedural-oriented-programming-pop-tw
paradigms-one-7a11cc143f0c

What Does “Declarative” Mean?

In programming, declarative means:

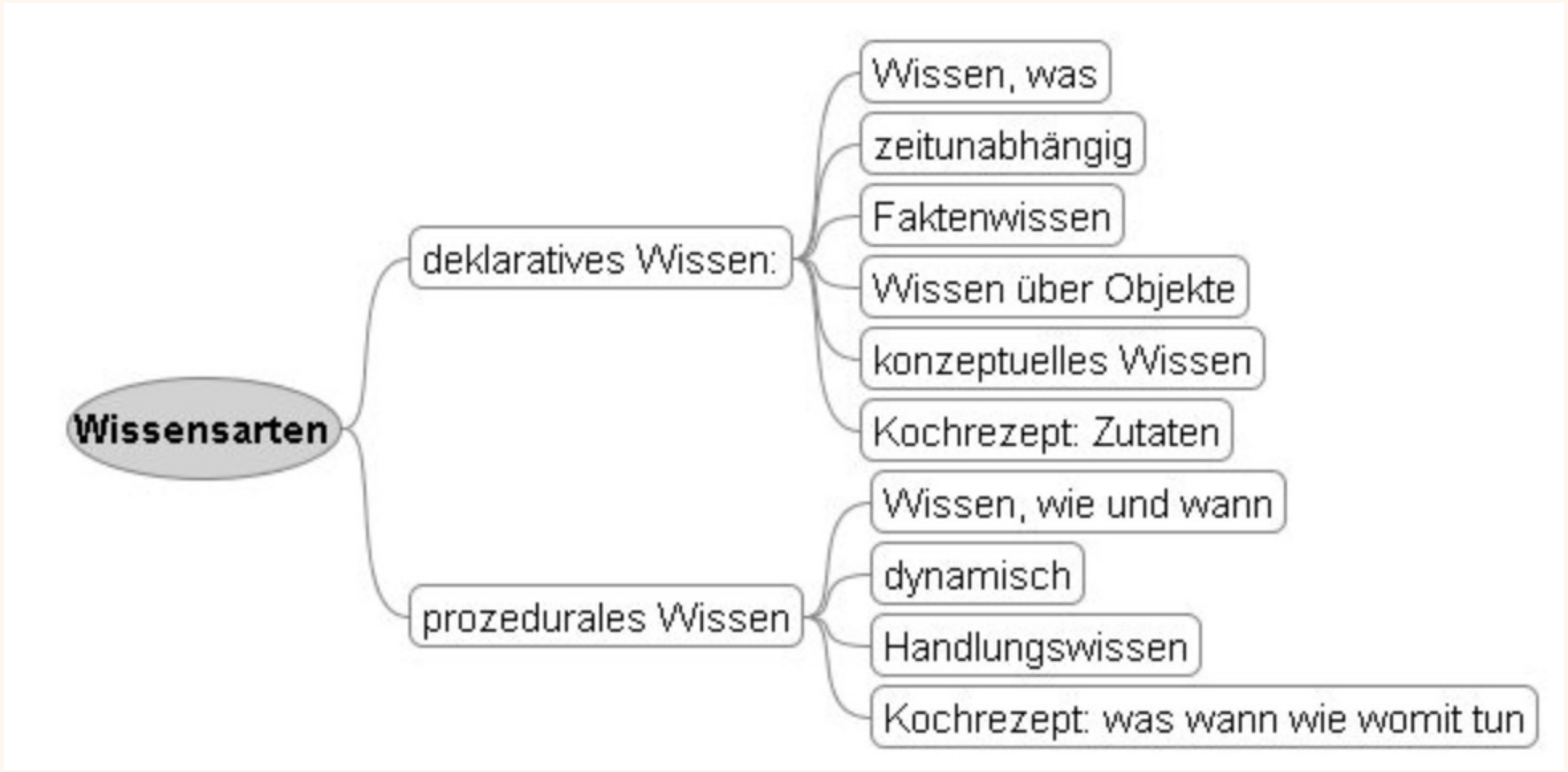
Tell computer what you want, not how to do it step by step.

It’s about describing the desired result, not the process.

Procedural vs Declarative

Aspect	Procedural (Imperative)	Declarative
Focus	How to do it	What to do
Mental Model	Describe each step	Describe the goal
Code Style	Detailed, instructional	Concise, expressive
Control Flow	Explicit (for, while, if)	Abstracted (e.g. .filter(), .map())
Abstraction Level	Low	High
Typical Tools	Loops, conditionals, variable assignments	Streams, HTML, Lambdas
Example	for (int x : list) if (x > 10) result.add(x);	list.stream().filter(x -> x > 10)
Boilerplate Code	Often high	Low
Optimized by	Programmer	System (e.g. Stream API)
Best For	Precise control, performance tuning	Readability, maintainability

Deklarativ vs prozedural



Example scenario

Animal monitoring in a national park (*animal-monitoring* project).

Spotters send back reports of animals they have seen (**`Sighting`** objects).

Base collates sighting reports to check on population levels.

Review version 1 of the project, which is implemented in a familiar (imperative) style:

- The **`AnimalMonitoring`** class has methods to:
  - List all sighting records;
  - List sightings of a particular animal;
  - Identify animals that could be endangered;
  - Calculate sighting totals;
  - Etc.

https://www.youtube.com/watch?v=M3wG-vowhQo

Example scenario

Tierbeobachtung in einem Nationalpark (Tierbeobachtungsprojekt).

Sichter (Beobachter) senden Berichte über die von ihnen gesichteten Tiere (**`Sighting object`**).

Die Basis stellt die Sichtungsberichte zusammen, um die Populationszahlen zu überprüfen.

Version 1 des Projekts, das in einem vertrauten (imperativen) Stil implementiert ist:

- Die Klasse **`AnimalMonitoring`** hat Methoden, um:
- Auflisten aller Sichtungsberichte;
- Auflisten der Sichtungen eines bestimmten Tieres;
- Identifizierung von Tieren, die gefährdet sein könnten;
- Berechnung der Gesamtzahl der Sichtungen;
- usw.

Project Class Structure

- Sighting class stores sighting data,
- SightingReader reads sighting records, and
- AnimalMonitor collates and analyzes data.

Data Source: CSV file with comma-separated fields.

Project Goal: Analyze sighting data from multiple spotters in various areas.

Classes and methods

Class Sighting:
- Fields
- getDetails()

Class AnimalMonitor:
- ArrayList sightings
- addSightings()
- Print all sightings
- Methoden zur auswahl von sightings nach bestimmten Vorgaben

Project Class Structure

- Sighting class stores sighting data,
- SightingReader reads sighting records, and
- AnimalMonitor collates and analyzes data.

Data Source: CSV file with comma-separated fields.

Project Goal: Analyze sighting data from multiple spotters in various areas.

Classes and methods

Class Sighting:

- Fields
- getDetails()

Class AnimalMonitor:

- ArrayList sightings
- addSightings()
- Print all sightings
- Methoden zur auswahl von sightings nach bestimmten Vorgaben

Alternative implementation with lambdas and streams

Motivation / why?:

- Reduce boilerplate (see control flow (for, if) of methods in AnimaMonitor)
- Focus on outcome

Use case / when?

- Processing a collection

Method and lambda equivalent

parameter
body
lambda syntax

ArrayList <Sightings> sightings;

Method

```
public void printSighting(Sighting record)
{
    System.out.println(record.getDetails());
}
```

Lambda

```
(Sighting record) ->
{
    System.out.println(record.getDetails());
}
```

Processing a collection – the usual approach (for each)

```
loop (for each element in the collection):
    get one element;
    do something with the element;
end loop
```

```
for(Sighting record : sightings) {
    printSighting(record);
}
```

for each

Lambdas

Bear a strong similarity to simple methods.

They have:

- A return type. (inferred from lambda body)
- Parameters.
- A body.

They don’t have a name (anonymous methods).

They have no associated object.

They can be passed as parameters:

- As *code* to be executed by the receiving method.

Processing a whole collection- lambda

Passed parameter is not data but a piece of code (lambda)

```
collection.doThisForEachElement(some code);
```

Object: ArrayList <Sightings> sightings

Method

```
sightings.forEach((Sighting record) ->
     {
          System.out.println(record.getDetails());
     }
);
```

Reduced lambda syntax:infer type

```
sightings.forEach((~~Sighting~~ record) ->
    {
        System.out.println(record.getDetails());
    }
);
```

Reduced lambda syntax: single parameter

```
sightings.forEach(record ->
    {
        System.out.println(record.getDetails());
    }
);
```

Reduced lambda syntax:single statement

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

Lambda syntax

Kein Parameter: `() -> System.out.println("Zero parameter lambda");`

Mit einem Parameter: parameter -> expression

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

Mit mehreren Parametern: `(parameter1, parameter2) -> expression`

Mit komplexerem Code: `(parameter1, parameter2) -> { code }`

Lambda syntax

Ein Lambda-Ausdruck mit einem Parameter hat die Form:

```
parameter -> expression
```

Ein Lambda-Ausdruck mit mehreren Parametern hat die Form:

```
(parameter1, parameter2) -> expression
```

Review

Functional Style

- Lambdas are used to write code in a more functional programming style.

Lambda Syntax

- They have no explicit name.
- The return type is inferred by the compiler or interpreter.
- They do not have an associated object, unlike methods in object-oriented programming (anonymous function).

Usage: Lambdas are commonly **passed as arguments to methods** where they are executed.

Lambdas

Bear a strong similarity to simple methods.

They have:

- A return type.
- Parameters.
- A body.

They don't have a name (anonymous methods).

They have no associated object.

They can be passed as parameters:

- As *code* to be executed by the receiving method.

Example scenario

Tierbeobachtung in einem Nationalpark (Tierbeobachtungsprojekt).

Sichter (Beobachter) senden Berichte über die von ihnen gesichteten Tiere (**`Sighting object`**).

Die Basis stellt die Sichtungsberichte zusammen, um die Populationszahlen zu überprüfen.

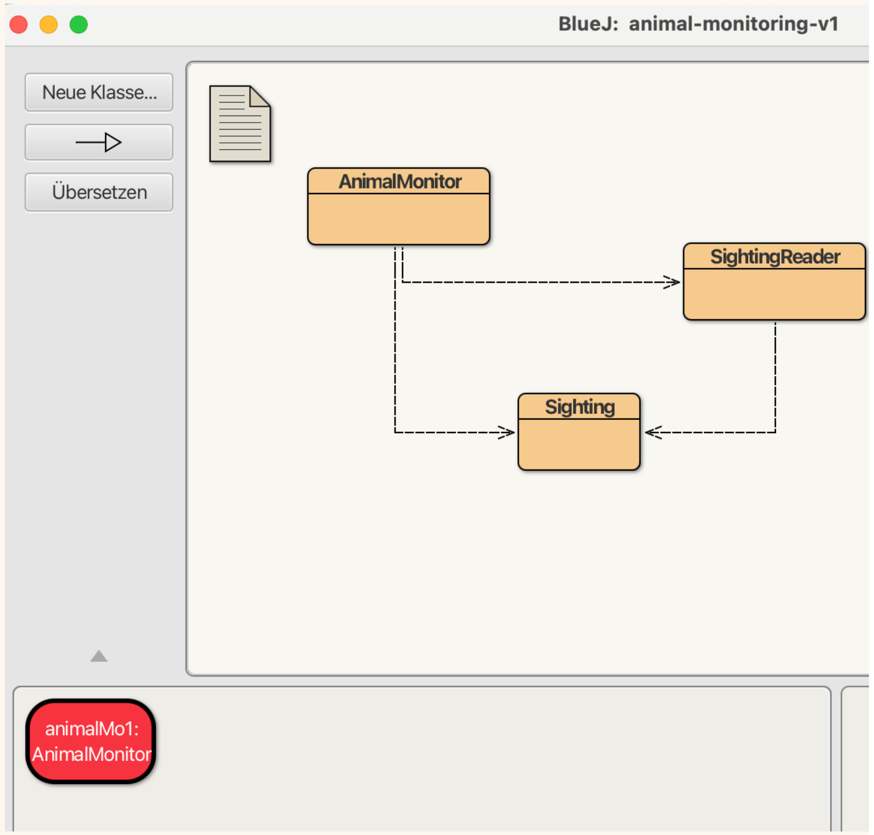
Version 1 des Projekts, das in einem vertrauten (imperativen) Stil implementiert ist:

- Die Klasse **`AnimalMonitoring`** hat Methoden, um:
- Auflisten aller Sichtungsberichte;
- Auflisten der Sichtungen eines bestimmten Tieres;
- Identifizierung von Tieren, die gefährdet sein könnten;
- Berechnung der Gesamtzahl der Sichtungen;
- usw.

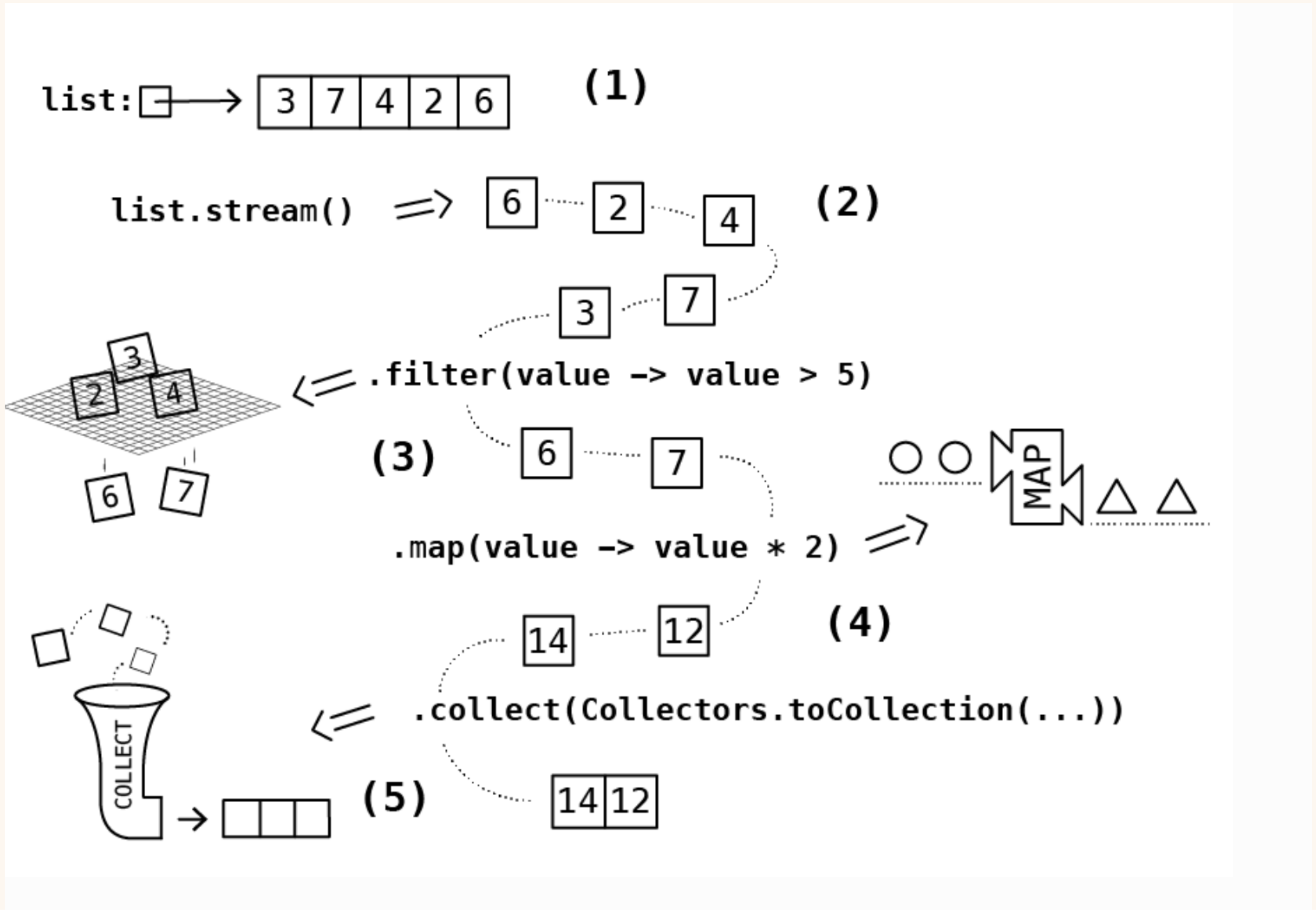
Exploring the project

```
// Sighting fields
private final String animal;
private final int spotter;
private final int count;
private final int area;
private final int period;
```

```
# Fields: spotter,animal,count,area,period
0,Mountain Gorilla,3,1,0
0,Buffalo,10,1,0,
0,Elephant,0,1,0
1,Mountain Gorilla,1,2,0
2,Mountain Gorilla,3,3,0
```



Streams: convoyer band / Förderband



https://www.youtube.com/watch?v=U85I2y-cHRE

Streams

- Stream operations provide an alternative means of implementing tasks associated with iteration over collections.
- Some existing library classes have been retro-fitted to support streams and lambda.
- Streams often involve multi-stage processing of data in the form of a pipeline of operations.
  - (Streams beinhalten oft eine mehrstufige Verarbeitung von Daten in Form einer Pipeline von Operationen.)

Lambda syntax

Kein Parameter: `() -> System.out.println("Zero parameter lambda");`

Mit einem Parameter: parameter -> expression

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

Mit mehreren Parametern: `(parameter1, parameter2) -> expression`

Mit komplexerem Code: `(parameter1, parameter2) -> { code }`

Practice

P1. If you have a collection called myList, what Java code would you have to write to apply some code to each of the members in the list?

```
myList.forEach((item) ->
     {
          //code
     }
);
```

```
for(Type item : myList) {
     //code
}
```

for each

Streams

- Streams are often created from the contents of a collection.
- An **`ArrayList`** is not a stream, but its **`stream`** method creates a stream of its contents.
- Elements in a stream are not accessed via an index, but usually sequentially.
- The contents and ordering of the stream cannot be changed – changes require the creation of a new stream.
- A stream could potentially be infinite!
- Elements in a stream can be processed in parallel.

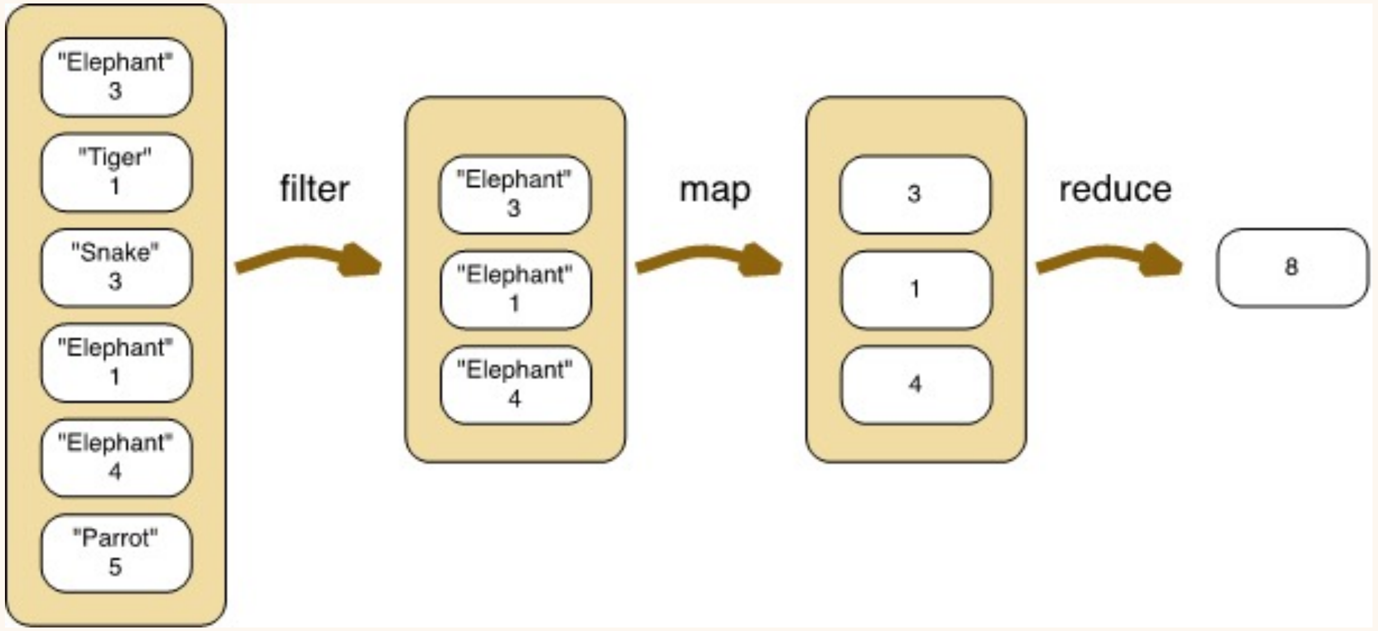
Operations: Filters, maps and reductions

Streams are immutable* (unveränderlich), so operations often result in a new stream.

There are three common types of operation:

- **Filter**: select items from the input stream to pass on to the output stream.
- **Map**: replace items from the input stream with different items in the output stream.
- **Reduce**: collapse the multiple elements of the input stream into a single element.

A Pipeline of Operations



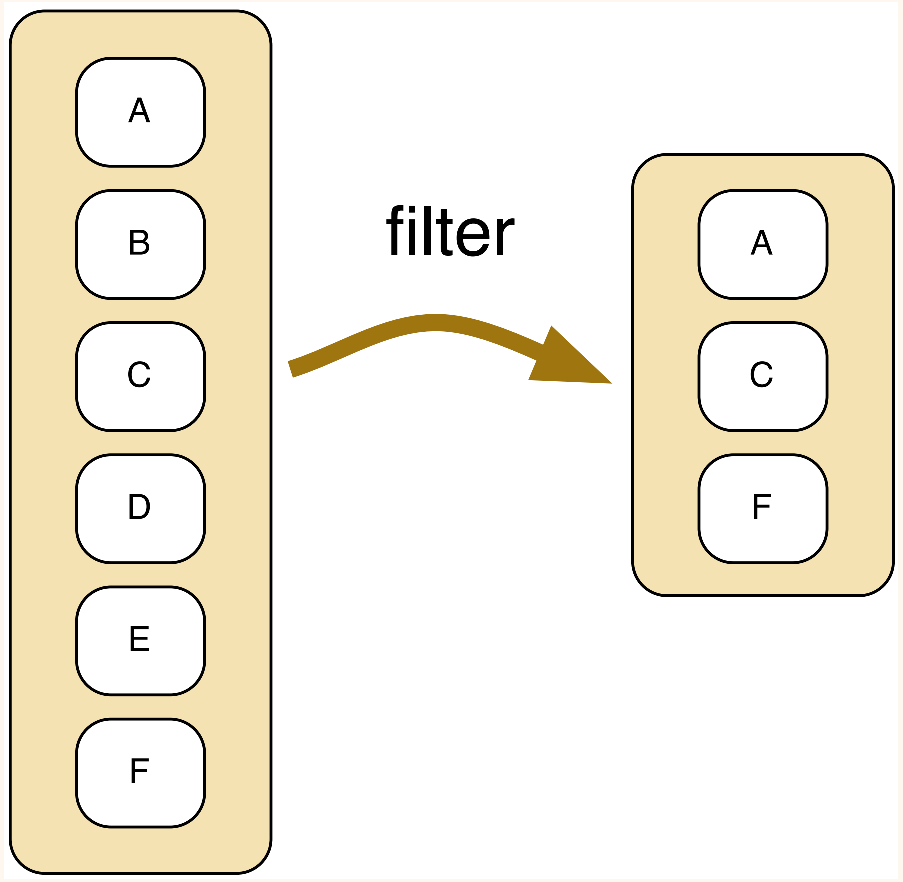
```
filter(name is elephant).map(count).reduce(add up)
```

https://www.youtube.com/watch?v=U85I2y-cHRE

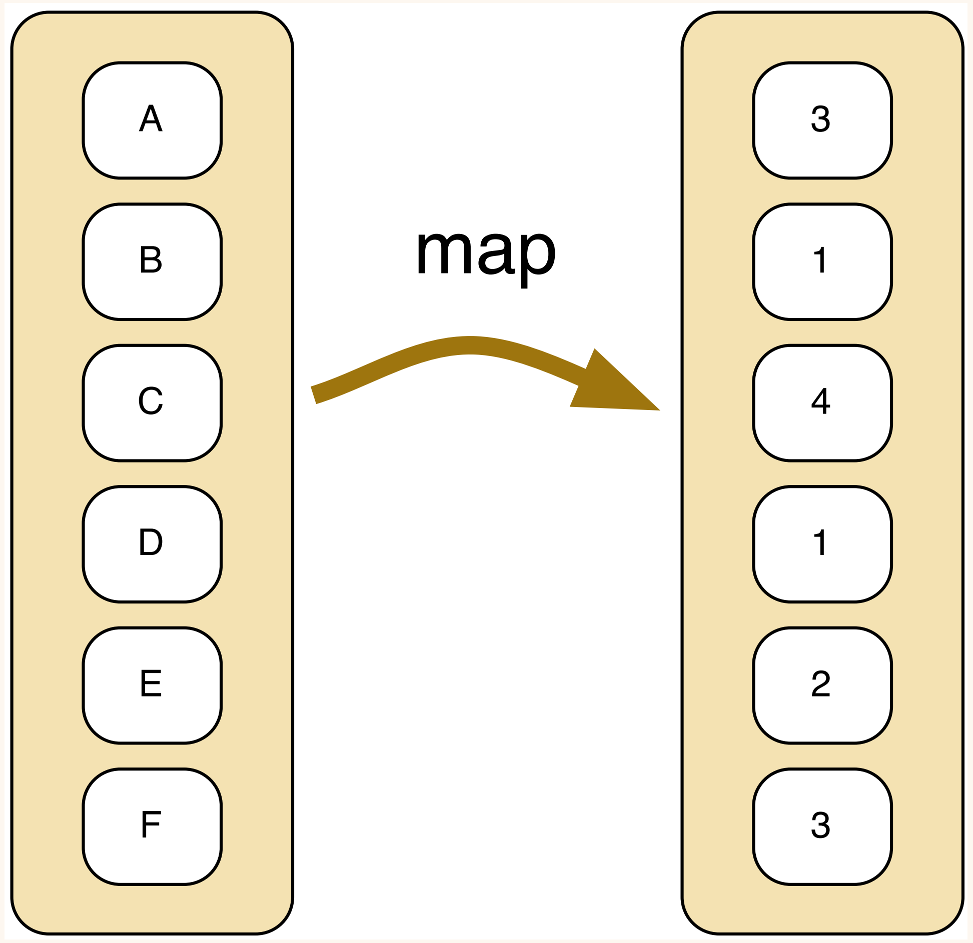
Pipelines

- Pipelines start with a source.
- Operations are either:
    - Intermediate: produce a new stream as output.
    - Terminal: are the final operation in the pipeline
- They might have a void return type.

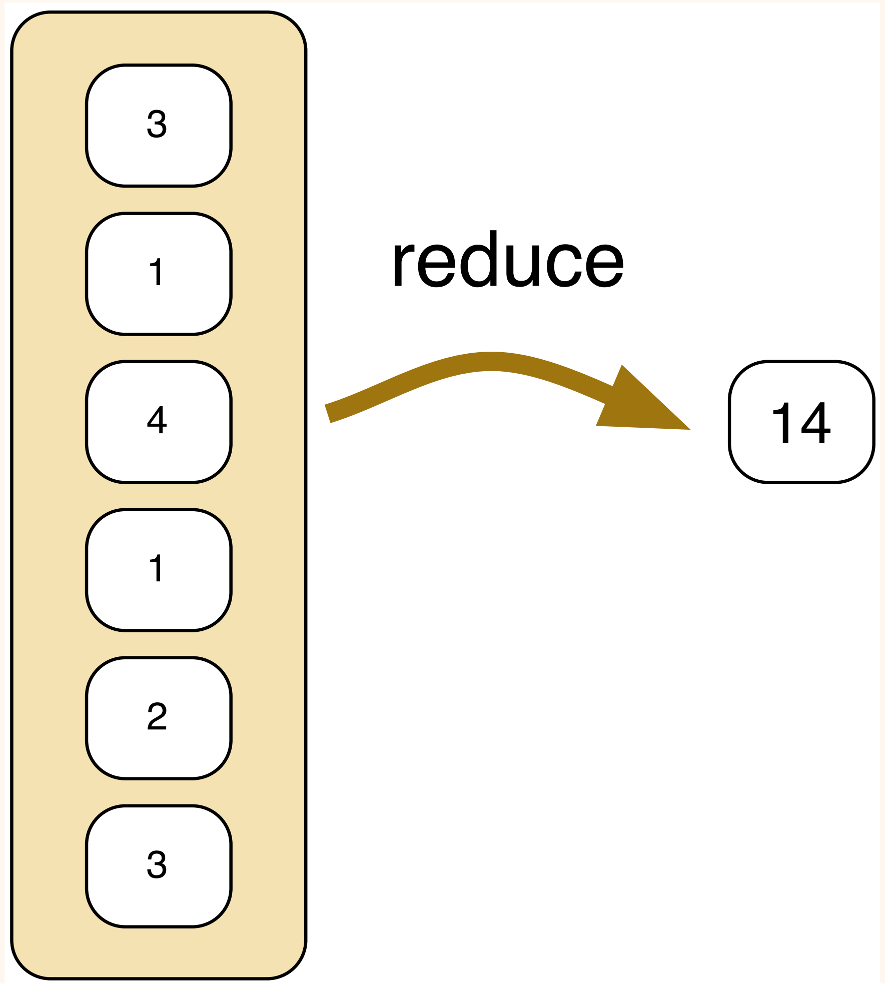
Filter



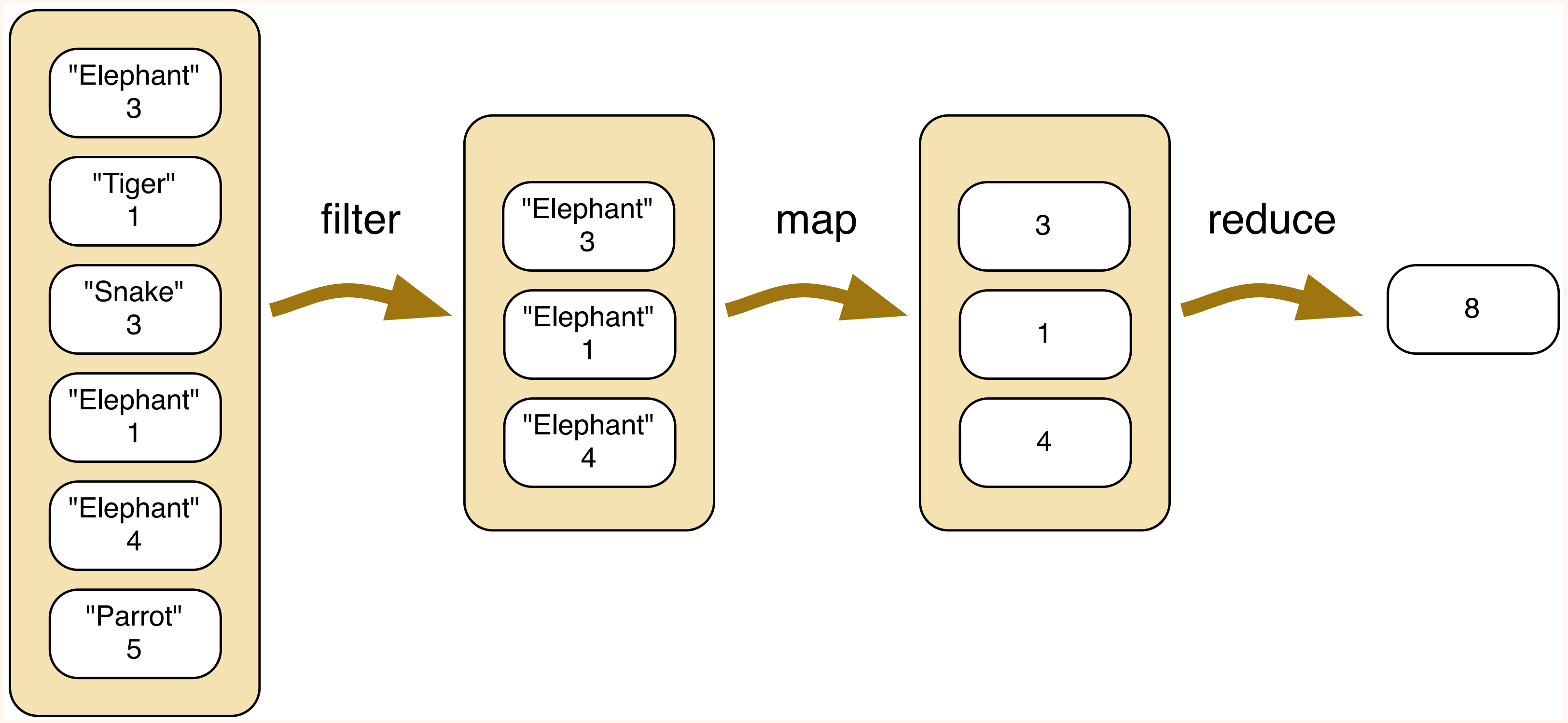
Map



Reduce



A pipeline of operations



```
// sightings.filter(name == elephant).map(count).reduce(add up)
```

Practice

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

P3. Given the animal-monitoring-v1 example, write pseudo-code for determining how many elephants a particular spotter saw on a particular day.

P4. Given the animal-monitoring-v1 example, write pseudo-code to create a stream containing only those sightings that have a count greater than 0.

Stream methods

forEach(): executes an action (side effect) for each element in the collection or stream.

- Often used for printing

stream(): Converts a collection (like an ArrayList, Set, or Map) into a Stream, which enables functional-style operations using lambdas

- Think of stream() as the *starting point* — it puts your data on a virtual conveyor belt so you can process it step by step.

filter(): Filters the elements in a stream based on a condition (predicate)

- Keeps only the items that match the condition.

map():transforms each element in the stream using a given function.

- Modifies elements, like a machine on the conveyor belt that stamps or changes each item as it passes by.

reduce(): Combines all the elements in a stream into a single result, using an accumulator function.

- Can be used to sum numbers, concatenate strings, etc.

Practice

Filter and Transform a List with Streams.

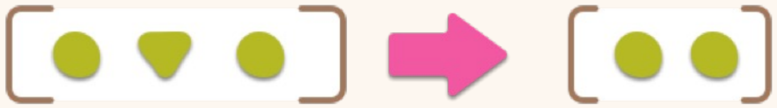
You are given a list of names: anna, bella, carla, achim

Your goal is to:

1. Filter the list to include only names that start with the letter “a”.
2. Convert these names to uppercase.
3. Print each resulting name to the console.

Use Java Streams and Lambda expressions to complete this task.

Filters



Filters require a **`Boolean`** lambda as a parameter.

A **`Boolean`** lambda is called a *predicate*.

If the predicate returns true for an element of the input stream then that element is passed on to the output stream; otherwise it is not. *(Filters determine which elements to retain.)*

Some predicates:

- `s -> s.getAnimal().equals("Elephant”)`
- `s -> s.getCount() > 0`
- `(s) -> true // Pass on all elements.`
- `(s) -> false // Pass on none.`

Example: print details of only the Elephant sightings.

```
sightings.stream()
         .filter(s -> "Elephant".equals(s.getAnimal()))
         .forEach(s -> System.out.println(s.getDetails()));
```

The **`map`** method

The type of the objects in the output stream is often (but not necessarily) different from the type in the input stream.

E.g., extracting just the details **`String`** from a **`Sighting`**:

```
sightings.stream()
            .map(sighting -> sighting.getDetails())
            .forEach(details ->
                        System.out.println(details));
```

The **`reduce`** method

More complex than both **`filter`** and **`map`**.

Its task is to 'collapse' a multi-element stream to a single 'value'.

It takes two parameters: a value and a lambda:

```
reduce(start, (acc, element) -> acc + element)
```

The first parameter is a starting value for the final result.

The lambda parameter itself takes two parameters:

- an accumulating value for the final result, and
- an element of the stream.

The lambda determines how to merge an element with the accumulating value.

- The lambda's result will be used as the **`acc`** parameter of the lambda for the next element of the stream.
- The **`start`** value is used as the first **`acc`** parameter that is paired with the first element of the stream.

The **`reduce`** method – a comparative example

```
sightings.stream()
.filter(sighting -> animal.equals(sighting.getAnimal())
.map(sighting -> sighting.getCount())
.reduce(0, (total, count) -> total + count);
```

Initial value

```
int total = 0;
for(Sighting sighting : sightings) {
    if(animal.equals(sighting.getAnimal())) {
        int count = sighting.getCount();
        total = total + count;
    }
}
```

Accumulation

Removal from a collection using a predicate lambda

```
/**
 * Remove from the sightings list all of
 * those records with a count of zero.
 */
public void removeZeroCounts()
{
    sightings.removeIf(
         sighting -> sighting.getCount() == 0);
}
```

Summary

Streams and lambdas are an important and powerful new feature of Java.

They are likely to increase in importance over the coming years.

Expect collection processing to move in that direction.

Lambdas are widely used in other areas, too; e.g. GUI building for event handlers.

Summary

A collection can be converted to a stream for processing in a pipeline.

Typical pipeline operations are filter, map and reduce.

Parallel processing of streams is possible.