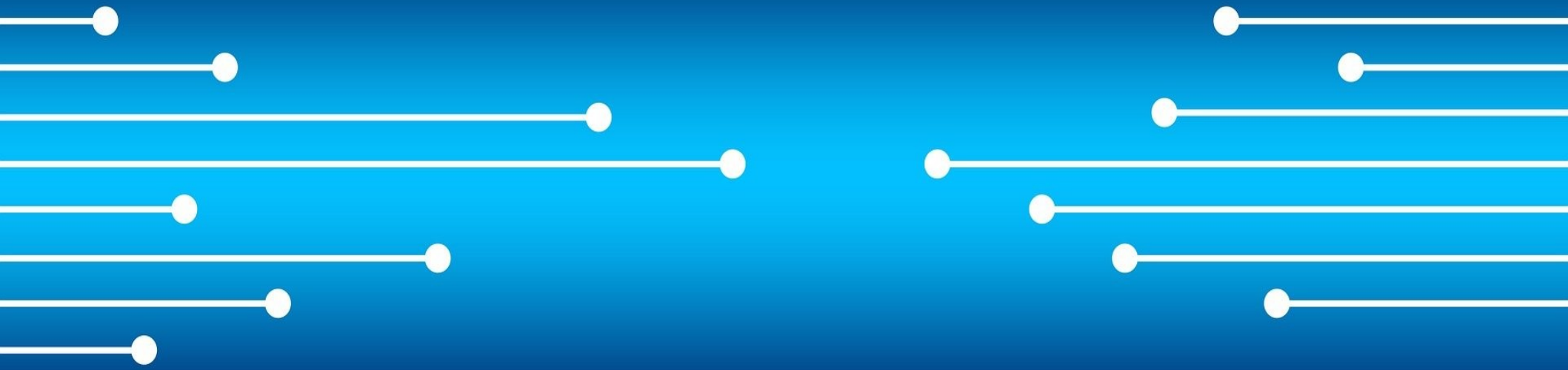
Datenverarbeitung

Teil des Moduls 5CS-DPDL-20



Prof. Dr. Deweß

Thema 6

Lambdas und Elemente-Streams (bei „Collections“)

- nötiges Vorwissen
- Was versteht man unter „Lambda-Ausdrücken“?
- Nutzung von Lambdas im Zusammenhang mit Elemente-Streams bei „Collections“

Vorwissen – Programmierparadigmen

Einige Programmierprinzipien

Imperative Programmierung

- **Idee:** Programm, welches genauen Ablauf vorgibt (lat. „imperare" zu deutsch „anordnen", „befehlen"), wie ein Ergebnis erreicht werden soll.
- prozedurale und **objektorientierte Programmiersprachen** verfolgen meist hauptsächlich die imperative Programmierung

Deklarative Programmierung

- **Idee:** Programm, welches das genaue Ziel vorgibt, welches erreicht werden soll, aber nicht den genauen Ablauf vorgibt (z.B. Abfrage in SQL).
- Programmablauf basiert auf Funktionsaufrufen, wobei Funktionen nicht Daten verändern, sondern Ergebnisse erzeugen, so dass Funktionen anstatt von Daten übergeben werden können

„Lambda"-Ausdrücke, funktionale Schnittstellen und die (seit Java 8) neuen Elemente-Streams ermöglichen eine (einfachere) deklarative Programmierung innerhalb von Java

Vorwissen – Varianten von „For"-Schleifen in Java

Was wir bisher können ...

einfache „For"-Schleife: Syntax

```
for(Initialisierung; Ausdruck; Erhöhung) {
  Anweisungen;
}
```

Beispiel

```
int sum = 0;
for(int i = 1; i < 10; i++) {
  sum = sum + i;
}
System.out.println(sum);
```

Summe der Zahlen von 1 bis 9

Vorwissen – Varianten von „For"-Schleifen in Java

Was es seit Java 5 gibt …

erweiterte „For"-Schleife: Syntax

```
for(Typ element : Elemente) {
  Anweisungen;
}
```

Beispiel

```
int[] myIntegers = {1, 2, 3};
for(int anInteger : myIntegers) {
  System.out.println(anInteger);
}
```

Ausgabe:
1
2
3

```
List<String> myNumberWords = new ArrayList<>();
myNumberWords.add("one");
myNumberWords.add("two");
myNumberWords.add("three");
for(String aWord : myWords ) {
  System.out.println(aWord);
}
```

Ausgabe:
one
two
three

Vorwissen – Varianten von „For"-Schleifen in Java

Was es seit Java 8 gibt …

„forEach()"-Methode in der „iterable"-Schnittstelle

```
void forEach(Consumer<? super T> action)
```

führt die gegebene „action" für jedes Elemente des „iterablen" was-auch-immer aus, bis alle Elemente verarbeitet wurden oder die „action" eine Ausnahme geworfen hat

Damit rückt die deklarative Sicht in den Vordergrund – der funktionale Aspekt, was für jedes Element gemacht wird, ist wichtig.
Und wenn uns auch die Reihenfolge nicht mehr interessiert, können wir mit der Methode vielleicht einfach auch Aufgaben parallel und damit zeitsparender erfüllen als bei Verwendung einer klassischen „For"-Schleife.

Vorwissen – Varianten von „For"-Schleifen in Java

Was es seit Java 8 gibt ...

„forEach()"-Methode in der „iterable"-Schnittstelle

```
void forEach(Consumer<? super T> action)
```

führt die gegebene „action" für jedes Elemente des „iterablen" was-auch-immer aus, bis alle Elemente verarbeitet wurden oder die „action" eine Ausnahme geworfen hat

Was für die Verwendung interessant ist:

1. Die „Consumer"-Schnittstelle ist eine „funktionale Schnittstelle".
2. Daher wird die „forEach"-Methode vorwiegend mit Lambda-Ausdrücken verwendet.

wenn wir jetzt noch wissen würden, was das uns sagen soll ...
am Ende des Themengebietes wissen wir es :-)

Vorwissen – „Consumer<T>“ und „forEach()“

Dokumentation von „Consumer<T>“:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/Consumer.html

```
@FunctionalInterface
public interface Consumer<T>
```

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.

This is a functional interface whose functional method is `accept(Object)`.

Dokumentation von „forEach()“:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Iterable.html#forEach(java.util.function.Consumer)

forEach

```
default void forEach(Consumer<? super T> action)
```

...

The default implementation behaves as if:

```
for (T t : this)
    action.accept(t);
```

Vorwissen – „Consumer<T>“ und „forEach()“

```
package de.baleipzig.lambdasstreams;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class TestLambdas {

  public static void main(String[] args) {
    List<String> myNumberWords = new ArrayList<>();
    myNumberWords.add("one");
    myNumberWords.add("two");

    Consumer<String> aNumberWordConsumer = new Consumer<String>() {
      public void accept(String aWord) {
        System.out.println("Number: " + aWord);
      }
    };

    myNumberWords.forEach(aNumberWordConsumer);
  }

}
```

anonyme Klasse, die „Consumer<String>“ implementiert, also die Methode „accept()“ implementiert

entsprechend der Dokumentation wird also nun für jedes Element von myNumberWords die „accept()“-Methode unseres aNumberWordConsumer ausgeführt

Vorwissen – „Consumer<T>“ und „forEach()“

```
package de.baleipzig.lambdasstreams;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class TestLambdas {

  public static void main(String[] args) {
    List<String> myNumberWords = new ArrayList<>();
    myNumberWords.add("one");
    myNumberWords.add("two");

    myNumberWords.forEach(new Consumer<String>() {
      public void accept(String aWord) {
        System.out.println("Number: " + aWord);
      }
    });
  }
}
```

anonyme Klasse, die „Consumer<String>“ implementiert, also die Methode „accept()“ implementiert

entsprechend der Dokumentation wird also nun für jedes Element von myNumberWords die „accept()“-Methode unseres anonymen „Consumer“ ausgeführt

Vorwissen – „Consumer<T>“ und „forEach()“

Aufgabe 1 – Wiederholung eingebettete Klassen (anonyme Klassen)
Erstellen Sie ein neues Paket für die Aufgaben zu Lambdas und Streams und darin eine neue Testklasse.
Deklarieren Sie dort in der „main“-Methode analog zu „myNumberWords“ eine Variable „myFruitWords“. Fügen Sie der damit repräsentierten Liste zwei passende Wörter hinzu und implementieren Sie eine vergleichbare Konsolenausgabe, die „forEach()“ mit passender, anonymer Klasse nutzt, bei der den einzelnen Wörtern statt „Number: “ ein „Fruit: “ vorangestellt ist.

```
    myNumberWords.forEach(new Consumer<String>() {
      public void accept(String aWord) {
        System.out.println("Number: " + aWord);
      }
    });
  }
}
```

entsprechend der Dokumentation wird also nun für jedes Element von myNumberWords die „accept()“-Methode unseres anonymen Consumer ausgeführt

Vorwissen – „Consumer<T>“ und „forEach()“

```
package de.baleipzig.lambdasstreams;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class TestLambdas {

  public static void main(String[] args) {
    //...
    List<String> myFruitWords = new ArrayList<>();
    myFruitWords.add("apple");
    myFruitWords.add("banana");

    myFruitWords.forEach(new Consumer<String>() {
      public void accept(String aWord) {
        System.out.println("Fruit: " + aWord);
      }
    } ) ;
  }
}
```

Viel Aufwand, dafür dass uns das anonyme Objekt der Klasse nicht weiter interessiert und nur die Funktion interessant ist, bei der sich als einzige etwas geändert hat; hier wäre eine bessere Möglichkeit der deklarativen Programmierung angebracht

Funktionale Schnittstellen

- jede Schnittstelle mit nur einer einzigen abstrakten Methode (SAM – Single Abstract Method) ist eine funktionale Schnittstelle

Anmerkungen

- es ist nur die Anzahl der abstrakten Methoden relevant, default-Methoden zählen also nicht mit
- zur besseren Unterscheidbarkeit von anderen Schnittstellen (und als informative Absichtserklärung, die zu absichernden Fehlermeldungen bei Verletzung führen können), sollten funktionale Schnittstellen durch die Annotation `@FunctionalInterface` gekennzeichnet werden

Viele funktionale Schnittstellen findet man im Paket java.util.function:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/package-summary.html
(es gibt aber noch mehr [java.lang.Runnable; java.util.Comparator; java.awt.event.ActionListener; ...], auch ältere Schnittstellen, die schon immer „Funktionale Schnittstellen" waren, auch wenn es die Bezeichnung dafür noch nicht gab – dort fehlt dann historisch bedingt eventuell nur die Annotation)

Funktionale Schnittstellen - Vorteil

```
myFruitWords.forEach(new Consumer<String>() {
  public void accept(String aWord) {
    System.out.println("Fruit: " + aWord);
  }
});
```

- „forEach()" erwartet als Parameter einen „Consumer"
- „Consumer" ist eine funktionale Schnittstelle (hat nur die abstrakte Methode „accept()")

➔ wir sagen dem Compiler viel, was er eigentlich schon weiß, er weiß:
- er bekommt einen „Consumer<String>" (weil „myFruitWords" aus „Strings" besteht)
- die einzige mögliche Methode, die ausgeführt werden kann, ist „accept()"

- das einzige, was der Compiler nicht weiß: wie „accept()" implementiert ist

Lambda-Ausdrücke zur Nutzung „Funktionaler Schnittstellen“

```
myFruitWords.forEach(new Consumer<String>() {
  public void accept(String aWord) {
    System.out.println("Fruit: " + aWord);
  }
});
```

mit Lambda-Ausdruck:

```
myFruitWords.forEach(aWord -> System.out.println("Fruit: " + aWord));
```

wir beschränken uns aufs Wesentliche – die **Implementierung der Methode:** wird „accept“ mit Parameter „aWord“ aufgerufen, soll die angegebene Konsolenausgabe erfolgen

(dass es sich um die Methode „accept“ von „Consumer“ handelt, kann der Rechner selbst herausfinden)

Lambda-Ausdrücke – anonyme Methoden

```
myFruitWords.forEach(aWord -> System.out.println("Fruit: " + aWord));
```

- Begriff entstammt dem „λ-Kalkül“:
  - der „λ-Kalkül“ wurde von Alonzo Church entwickelt und ist Grundlage funktionaler Sprachen
  - ist konzeptuell gleich mächtig wie die Turingmaschine
  - allgemein wichtiger Aspekt: Funktionen als Objekte
- ein Lambda-Ausdruck:
  - ist wie eine anonyme Instanz einer funktionalen (nicht benannten) Schnittstelle, wobei sich auf die nötigen Parameter und Anweisungen (syntaktisch getrennt durch einen Pfeil-Operator) der einzigen Methode dieser Schnittstelle beschränkt wurde (daher auch Bezeichnung „anonyme Methode“)
  - wird zur Laufzeit durch ein Lambda-Objekt repräsentiert
  - Lambda-Objekt und Typ werden dynamisch zur Laufzeit erzeugt
  - kompakte Möglichkeit, Funktionen als Objekte zu behandeln und sie an andere Methoden zu übergeben
- in Java als neues Sprachelement seit Java 8

Lambda-Ausdrücke – anonyme Methoden: Anmerkungen

```
myFruitWords.forEach(aWord -> System.out.println("Fruit: " + aWord));
```

1. Letztendlich macht man zwar meist alles in einer Zeile, praktisch muss man das aber auch mit Lambda-Ausdrücken nicht, „forEach" erwartet einen „Consumer" und der Lambda-Ausdruck kann auch dem „Consumer" zugordnet werden, den er repräsentiert.

```
Consumer<String> aFruitWordConsumer =
  aWord -> System.out.println("Number: " + aWord);
myNumberWords.forEach(aFruitWordConsumer);
```

2. Diese Trennung würde bei wiederholter Nutzung verschiedener Ausdrücke Sinn ergeben, wofür hier beispielhaft ein kurzes, ansonsten weniger sinnvolles Beispiel folgt.

```
Consumer[] twoFruitwordConsumer = new Consumer[2];
twoFruitwordConsumer[0] = aWord -> System.out.println("Fruit: " + aWord);
twoFruitwordConsumer[1] = aWord -> System.out.println("Food: " + aWord);
for (int i = 0; i < 2; i++) {
  myFruitWords.forEach(twoFruitwordConsumer[i]);
}
```

Lambda-Ausdrücke – Syntaktische Beispiele

```
(null, ein oder mehrere Parameter ggf. mit Typangabe) -> Anweisung;
```

oder

```
(null, ein oder mehrere Parameter ggf. mit Typangabe) -> {
  Anweisung1;
  Anweisung2;
  ...
};
```

mit Typangabe z.B. (int a, int b) -> a + b;
(ohne Typangabe kontextabhängig)

- `(a, b) -> a + b;` ← Rückgabewert automatisch bestimmt
-
```
(a, b) -> {
  System.out.println(„First: “ + a);
  System.out.println(„Second: “ + b);
};
```
← kein Rückgabewert
-
```
(a, b) -> {
  int c = a + b;
  System.out.println("Sum: " + c);
  int d = c * a * b;
  System.out.println("Then multiplied with sum: " + d);
  return d;
};
```
Rückgabewert explizit festgelegt
- `() -> 42;` ← Rückgabewert 42 („the answer to life, the universe and everything“ ;-))

Lambda-Ausdrücke – Nutzung am Beispiel

Funktionale Schnittstelle „IntBinaryOperator" - Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/IntBinaryOperator.html

java.util.function

Interface IntBinaryOperator

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
int	applyAsInt(int left, int right) Applies this operator to the given operands.

Aufgabe 2a

Erstellen Sie beispielhaft (z.B. in Ihrer Klasse aus Aufgabe 1) einen Lambda-Ausdruck, welcher mit dieser Schnittstelle genutzt werden könnte.

Lambda-Ausdrücke – Lösung 2a

Funktionale Schnittstelle „IntBinaryOperator" - Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/IntBinaryOperator.html

```
IntBinaryOperator mySumSquareOperator = (a, b) -> (a + b) * (a + b);
```

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
int	**applyAsInt**(int left, int right) Applies this operator to the given operands.

Lambda-Ausdrücke – Lösung 2a

Funktionale Schnittstelle „IntBinaryOperator" - Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/IntBinaryOperator.html

```
IntBinaryOperator mySumSquareOperator = (a, b) -> (a + b) * (a + b);
```

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
int	`applyAsInt(int left, int right)` Applies this operator to the given operands.

Aufgabe 2b

Nutzen Sie (z.B. in Ihrer Klasse aus Aufgabe 1) noch die in Aufgabe 2a mittels Lambda-Ausdruck anonym implementierte Methode der Schnittstelle innerhalb einer Konsolenausgabe, indem Sie Ihren eben implementierten Operator mit dieser Methode und passenden Argumenten aufrufen.

Lambda-Ausdrücke – Lösung 2b

Funktionale Schnittstelle „IntBinaryOperator" - Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/IntBinaryOperator.html

```
package de.baleipzig.lambdasstreams;
//import ...
import java.util.function.IntBinaryOperator;
public class TestLambdas {
  public static void main(String[] args) {
    //...
    IntBinaryOperator mySumSquareOperator = (a, b) -> (a + b) * (a + b);
    System.out.println(mySumSquareOperator.applyAsInt(1,2));
  }
}
```

Lambda-Ausdrücke – Anmerkung zu Lösung 2b

Funktionale Schnittstelle „IntBinaryOperator" - Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/IntBinaryOperator.html

```
IntBinaryOperator mySumSquareOperator = (a, b) -> (a + b) * (a + b);
System.out.println(mySumSquareOperator.applyAsInt(1,2));
```

hier können Sie nicht wie bei dem „forEach"-Beispiel den Lambda-Ausdruck direkt als Argument bei „println" benutzen, da „println" den Typ „IntBinaryOperator" nicht als Parametertyp hat und damit der Compiler nicht automatisch den Typ des Lambda-Ausdrucks bestimmen kann; wenn Sie in der Klasse z.B. noch eine überladene statische Methode „**lambdatypgiven**" wie folgt hätten, ginge:

```
public static void lambdaTypeGiven(IntBinaryOperator test) {
  System.out.println(test.applyAsInt(1,2));
}
public static void lambdaTypeGiven(String test) {
  System.out.println(test);
}
public static void main(String[] args) {
  //..
  lambdaTypeGiven((a, b) -> (a + b) * (a + b));
}
```

Lambda-Ausdrücke – Aufgabe 3a

Aufgabe 3a
Erstellen Sie in Ihrem Paket folgende Schnittstelle.

```
package de.baleipzig.lambdasstreams;

public interface TestInterface {
  int getSomething();
}
```

Handelt es sich dabei um eine „Funktionale Schnittstelle“?

Lambda-Ausdrücke – Lösung 3a

Lösung 3a

Die Schnittstelle enthält nur eine abstrakte Methode und ist damit auch ohne diesbezügliche Annotation eine „Funktionale Schnittstelle".
Besser wäre natürlich trotzdem die Benutzung einer Annotation:

```
package de.baleipzig.lambdasstreams;

@FunctionalInterface
public interface TestInterface {
  int getSomething();
}
```

Lambda-Ausdrücke – Aufgabe 3b

Aufgabe 3b

Da es sich bei unserer Schnittstelle um eine „Funktionale Schnittstelle" handelt, können wir für diese in unserer Testklasse einen Lambda-Ausdruck benutzen.

(Bitte das Beispiel nicht implementieren, wir ändern nachher noch unsere Schnittstelle.)

Warum würde der auskommentierte Lambda-Ausdruck aktuell nicht funktionieren?

```
package de.baleipzig.lambdasstreams;
// ...

public class TestLambdas {

  public static void main(String[] args) {
    // ...
    TestInterface myLambdaTest; // interface reference
    myLambdaTest = () -> 42;
    System.out.println(myLambdaTest.getSomething());
    myLambdaTest = () -> (int) (Math.random() * 100);
    System.out.println(myLambdaTest.getSomething());
    System.out.println(myLambdaTest.getSomething());

    //myLambdaTest = () -> "42";
    System.out.println(myLambdaTest.getSomething());
  }

}
```

Lambda-Ausdrücke – Lösung 3b

Lösung 3b

„42" ist ein String-Literal, die zu implementierende Methode der Schnittstelle erwartet aber ein „int".

```
package de.baleipzig.lambdasstreams;

public interface TestInterface {
  int getSomething();
}
```

Zur Problembehebung müsste man entweder die „42" noch in ein „int" umwandeln oder in der Schnitttstelle statt einem „int" ein „Object" als Rückgabewert erwarten.

```
myLambdaTest = () -> Integer.valueOf("42");
```

oder

```
Object getSomething();
```

Lambda-Ausdrücke – Aufgabe 4

Aufgabe 4

a) Ändern Sie die Testschnittstelle „TestInterface" derart, dass die dortige Methode einen Parameter vom Typ „int" entgegennehmen muss.

```
package de.baleipzig.lambdasstreams;

public interface TestInterface {
  int getSomething();
}
```

b) Deklarieren Sie in Ihrer Testklasse „TestLambdas" eine Referenz auf die Testschnittstelle.

c) Weisen Sie der Schnittstellenreferenz einen Lambda-Ausdruck zu, der bei einem Argument *n* einen Rückgabewert von 2**n* ergibt. Testen Sie Ihren Lambda-Ausdruck (mit Anzeige in einer Konsolenausgabe).

Zusatzaufgabe:

Schreiben Sie einen Lambda-Ausdruck, der bei einem Argument ***n*** einen Rückgabewert von ***n!*** ergibt. Nutzen Sie dafür einen Anweisungsblock, eine dort lokale Variable result, eine For-Schleife zur Fakultätsberechnung und entsprechend eine return-Anweisung für die Ergebnisrückgabe.

Lambda-Ausdrücke – Lösung 4

Lösung 4

a)

```
package de.baleipzig.lambdasstreams;

public interface TestInterface {
  int getSomething(int n);
}
```

b)

```
TestInterface myLambdaTest; // interface reference
```

c)

```
myLambdaTest = (n) -> 2*n;
System.out.println(myLambdaTest.getSomething(3));
```

Zusatzaufgabe:

```
myLambdaTest = (n) -> {
  int result=1;
  for(int i=1; i <= n; i++)
    result = i * result;
  return result;
};
System.out.println(myLambdaTest.getSomething(3));
```

Lambda-Ausdrücke – Bemerkungen

- eine Referenz auf eine „Funktionale Schnittstelle" kann mit jedem kompatiblen (Typ beachten!) Lambda-Ausdruck benutzt werden
- wird ein Rückgabewert erwartet, dann ergibt sich dieser bei einer einfachen Anweisung (z.B. a + b) durch die Anweisung ohne „return" von selbst, während bei Anweisungsblöcken der Rückgabewert explizit mit „return" übergeben werden muss
- man kann innerhalb eines Lambda-Ausdrucks die Typen der für die zu implementierende Methode benötigten Parameter (z.B. Typ „Object a") durch spezielle Typangabe bei den Parametern (z.B. „int a") weiter einschränken, aber ansonsten nicht generell festlegen
- der Typ des Rückgabewertes eines Lambda-Ausdrucks wird durch die benutzte „Funktionale Schnittstelle" vorgegeben
- ein Lambda-Ausdruck selbst kann nicht generisch sein, lediglich die damit verbundene „Funktionale Schnittstelle" kann generisch sein (*„generisch" müssen Sie hier noch nicht verstehen, schauen wir uns später an*)
- ein Lambda-Ausdruck kann direkt als Argument für eine Methode benutzt werden, sofern der Parametertyp der Methode eine „Funktionale Schnittstelle" kompatibel mit dem Lambda-Ausdruck ist (siehe „forEach"-Beispiel)
- *prinzipiell können weitere Variablen in gewissem Rahmen in Lambda-Ausdrücken benutzt werden*

Methoden-Referenzen – Verwandte von „Lambdas“

nicht klausurrelevant

- manche Lambda-Ausdrücke rufen nur eine schon vorhandene Methode (z.B. System.out.println()) auf
- für diese Fälle würde es eigentlich reichen, sich nur auf die vorhandene Methode und deren Namen zu beziehen

Methoden-Referenzen (Operator „::“) sind spezielle Lambda-Ausdrücke für schon benannte Methoden, die insbesondere in Streams oft anstatt normaler Lambda-Ausdrücke verwendet werden;
wir werden bei den Streams klausurkonform bei normalen Lambda-Ausdrücken bleiben, Sie behalten aber bitte in Hinterkopf, dass es mit den Methoden-Referenzen manchmal noch kompaktere Ausdrücke gibt

Methoden-Referenzen – Verwandte von „Lambdas“

Arten von Methoden-Referenzen (nicht klausurrelevant)

- `(parameter) -> Klassenname.statischeMethodeName(parameter)`
  wird zu
  `Klassenname::statischeMethodeName`

- `(parameter) -> objekt.instanzMethodeName(parameter)`
  wird zu
  `Objekt::instanzMethodeName`

- `(parameter vom Typ Klassenname, rest) -> parameter.instanceMethodName(rest)`
  wird zu
  `Klassenname::instanceMethodName`

- `(Parameter) -> new Klasse`
  wird zu
  `Klassenname::new`

Methoden-Referenzen – Verwandte von „Lambdas“

Methoden-Referenzen – Aufgabe 5

```
package de.baleipzig.lambdasstreams;
// ...
public class TestLambdas {

  public static void main(String[] args) {
    // ...
    myFruitWords.forEach(aWord -> System.out.println(aWord));

    myFruitWords.sort((aFruitWord, anotherFruitWord) ->
      aFruitWord.compareToIgnoreCase(anotherFruitWord));

  }
}
```

Benutzen Sie statt der gegebenen Lambda-Ausdrücke jeweils eine Methoden-Referenz. Verändern Sie für Testzwecke Ihr „myFruitWords“ so, dass die Sortiermethode zu einer sichtbaren Änderung von „myFruitWords“ führt. Lassen Sie sich abschließend „myFruitWords“ anzeigen.

Methoden-Referenzen – Verwandte von „Lambdas“

Methoden-Referenzen – Lösung 5

```
package de.baleipzig.lambdasstreams;
// ...
public class TestLambdas {

  public static void main(String[] args) {
    // …
    List<String> myFruitWords = new ArrayList<>();
    myFruitWords.add("banana");
    myFruitWords.add("apple");

    myFruitWords.forEach(aWord -> System.out.println(aWord));
    myFruitWords.forEach(System.out::println);

    myFruitWords.sort((aFruitWord, anotherFruitWord) ->
      aFruitWord.compareToIgnoreCase(anotherFruitWord));
    myFruitWords.sort(String::compareToIgnoreCase);

    myFruitWords.forEach(System.out::println);
  }

}
```

Lambdas und

Elemente-Streams (bei „Collections“)

- nötiges Vorwissen
- Was versteht man unter „Lambda-Ausdrücken“?
- **Nutzung von Lambdas im Zusammenhang mit Elemente-Streams bei „Collections“**

Stream API – mit Blick auf Arrays und „Collections“

Stream

- Abstraktion zur Ausführung von (Massen-)Operationen auf Elementen einer Sequenz

  „Collection“, Array (wir betrachten jetzt nur diese beiden) oder sonstige Sequenz

- prinzipiell kein Datenspeicher (eher ein Fließband zur Elementbearbeitung)
- hält entweder Verweis auf einen Datenspeicher oder nutzt einen Generator zur Elementerzeugung (und könnte dann theoretisch von unbeschränkter Länge sein)
- beinhalten daneben Listen an auf den Elementen anzuwendenden Operationen
- in Java als neues Bibliothekselement seit Java 8
- wurde mit Lambda-Ausdrücken im Hinterkopf entworfen

Stream API – mit Blick auf Arrays und „Collections“

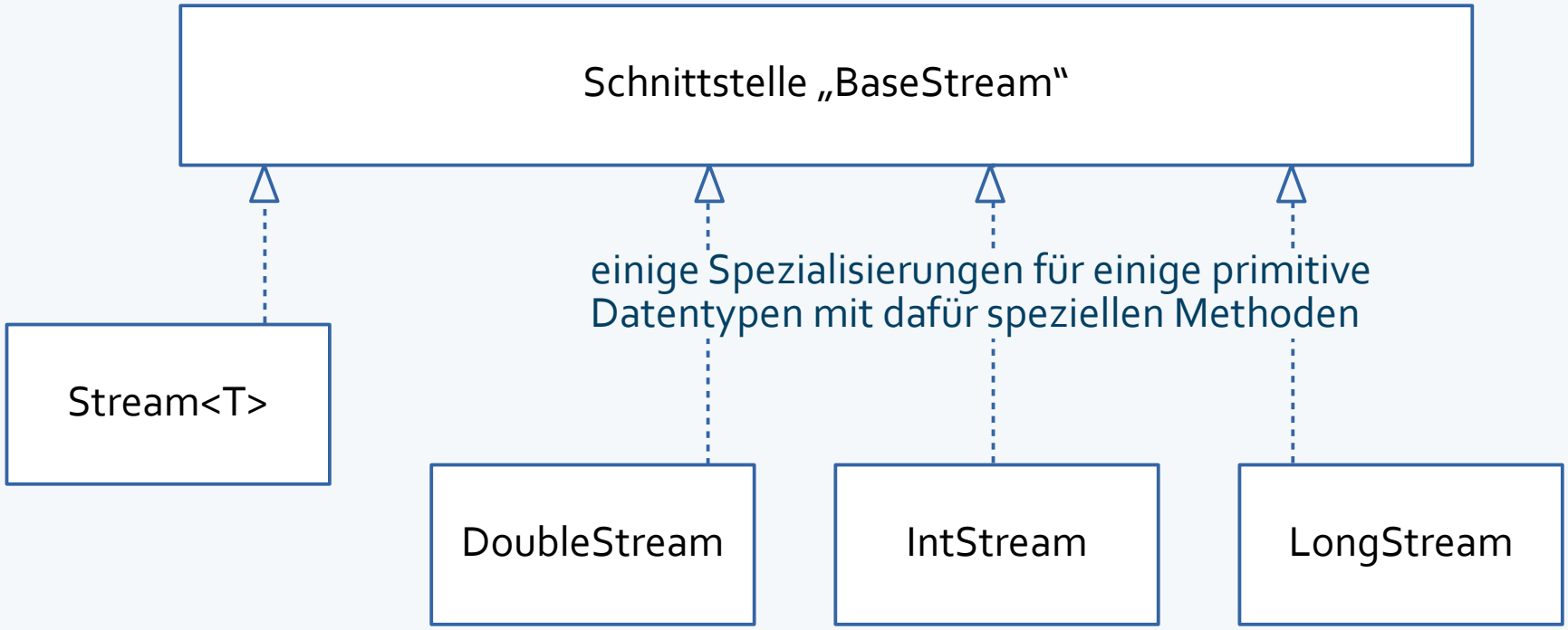
Vorteile von Streams im Vergleich zu herkömmlichen Methoden

- Alternative zur Iterator-basierten, schleifenbasierten Datenverarbeitung
- problemlose Realisierung unendlicher Streams (als Datenstruktur bekommen Sie das aufgrund der Ressourcenbeschränkungen nicht hin)
- Möglichkeit der deklarativen Programmierung

- können teilweise parallelisiert werden
- erlauben schnelle Massenoperationen auf Elementen ohne explizite Zwischenspeicherung
- bringen viele nützliche Methoden mit (Bilbiothekserweiterung)

Stream API – mit Blick auf Arrays und „Collections“

Streams in der Stream API



Stream API – mit Blick auf Arrays und „Collections“

Erzeugen eines Streams

- aus „Collections“: „stream()“- und „parallelStream()“-Methoden

```
List<String> myBooks = new ArrayList<>();
myBooks.add("Der kleine Bär und sein kleines Boot");
myBooks.add("Die Schildkröte hat Geburtstag");
Stream<String> sequentialListStream = myBooks.stream();
Stream<String> parallelListStream   = myBooks.parallelStream();
```

- aus „Arrays“: Array.stream(Object[])

```
String[] myMovies = {"Airport","Die Monster AG","Forrest Gump"};
Stream<String> sequentialArrayStream = Arrays.stream(myMovies);
Stream<String> parallelArrayStream   = Arrays.stream(myMovies).parallel();
```

- mit statischen Methoden der Stream-Klassen: z.B. Stream.of(Object[]), IntStream.range(int, int) ...

```
Stream<String> sequentialStringStream = Stream.of("und", "noch", "einer");
IntStream sequentialPrimitiveIntStream = IntStream.range(0,100);
```

- Stream-Methoden weiterer Klassen

Stream API – mit Blick auf Arrays und „Collections“

Erzeugen eines Streams – Aufgabe 6

Erstellen Sie eine neue Testklasse und darin analog zu den Beispielen mit „Stream<String>“ sowohl aus einer „Collection“ als auch aus einem Array entsprechende „Stream<Integer>“.

Stream API – mit Blick auf Arrays und „Collections“

Erzeugen eines Streams – Lösung 6

Erstellen Sie eine neue Testklasse und darin analog zu den Beispielen mit „Stream<String>“ sowohl aus einer „Collection“ als auch aus einem Array entsprechende „Stream<Integer>“.

```
List<Integer> myListIntegers = new ArrayList<>();
myListIntegers.add(1);
myListIntegers.add(2);
myListIntegers.add(3);
Stream<Integer> sequentialListIntegersStream = myListIntegers.stream();
Stream<Integer> parallelListIntegersStream =
  myListIntegers.parallelStream();

Integer[] myArrayIntegers = {1, 2, 3};
Stream<Integer> sequentialArrayIntegersStream =
  Arrays.stream(myArrayIntegers);
Stream<Integer> parallelArrayIntegersStream   =
  Arrays.stream(myArrayIntegers).parallel();
```

Stream API – mit Blick auf Arrays und „Collections“

Erzeugen eines Streams – „Spezial-Streams“

Bei „ints“ müssen Sie natürlich keinen „Stream<Integer>“ erstellen:

```
Integer[] myArrayIntegers = {1, 2, 3};
Stream<Integer> sequentialArrayIntegersStream =
  Arrays.stream(myArrayIntegers);
Stream<Integer> parallelArrayIntegersStream   =
  Arrays.stream(myArrayIntegers).parallel();
```

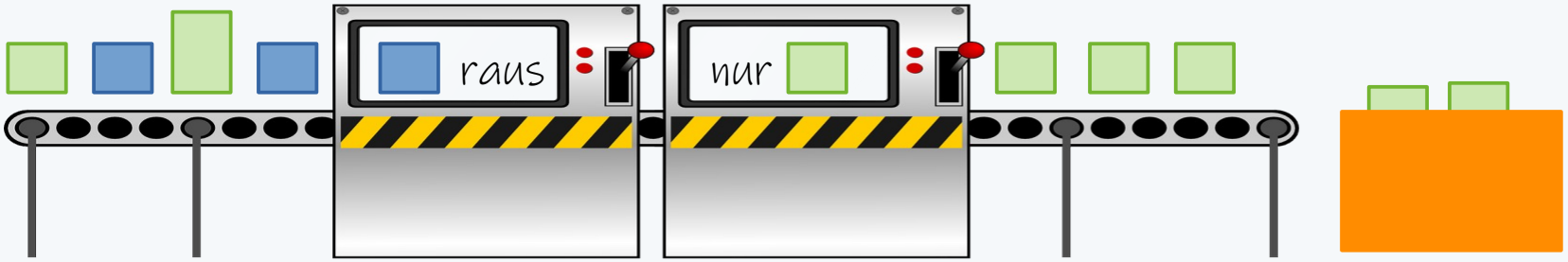
Bei „ints“ können Sie die Spezialisierung „IntStream“ nutzen:

```
int[] myArrayInts = {1, 2, 3};
IntStream sequentialArrayIntsStream =
  Arrays.stream(myArrayInts);
IntStream parallelArrayIntsStream =
  Arrays.stream(myArrayInts).parallel();
```

Stream API – mit Blick auf Arrays und „Collections“

auf Elemente eines Streams anzuwendende Operationen

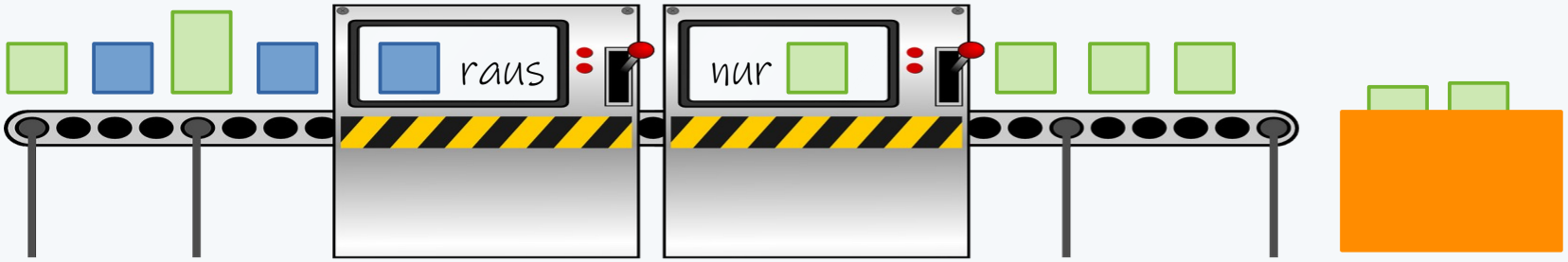
- intermediäre Operationen
  - Operationen auf einem Stream, die wieder einen Stream liefern
  - ermöglicht schrittweise Verarbeitung im Rahmen einer Pipeline
    (es wird nicht auf alle Elemente die erste Operation angewendet und anschließend auf alle Elemente die zweite Operation, sondern die erste und zweite Operation werden auf das erste Element angewendet, dann auf das zweite usw.)
  - werden nur ausgeführt, wenn eine terminale Operation vorhanden ist
    (ich würde auch kein Fließband laufen lassen, wenn nicht klar ist, was am Ende des Fließbandes passiert – wer will schon Scherben; Stichwort: „ Lazy Invocation“)
- terminale Operationen
  - Operation, die gleichzeitig den Stream schließt



Stream API – mit Blick auf Arrays und „Collections“

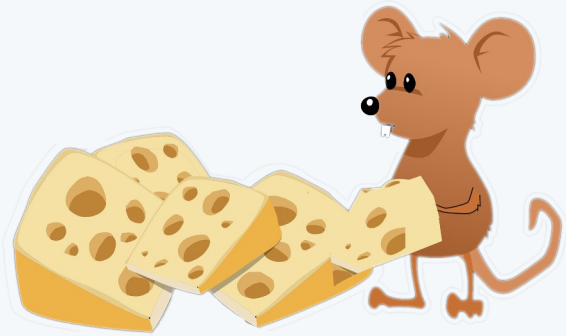
Terminale Operationen, die den Stream schließen

- alle Operationen auf einem Stream, die keinen Stream zurückliefern, z.B.
  - forEach
  - findFrist, findAny,allMatch, nonMatch, anyMatch
  - min, max, count
  - toArray, reduce, collect, iterator
  - isPresent, get, orElse, orElseThrow, ifPresent, isEmpty, … (aus java.util.Optional<T>) (nicht klausurrelevant, da wir uns „Optional“ nicht explizit anschauen)
- können nur einmal aufgerufen werden
- schließen damit den Stream (und machen ihn damit für weitere Operationen unbrauchbar; ein geschlossener Stream ist und bleibt zu!)
  (wie in der Realität: Sind die Waren vom Band verpackt, sind sie nicht mehr auf dem Band.)



Stream API – mit Blick auf Arrays und „Collections“

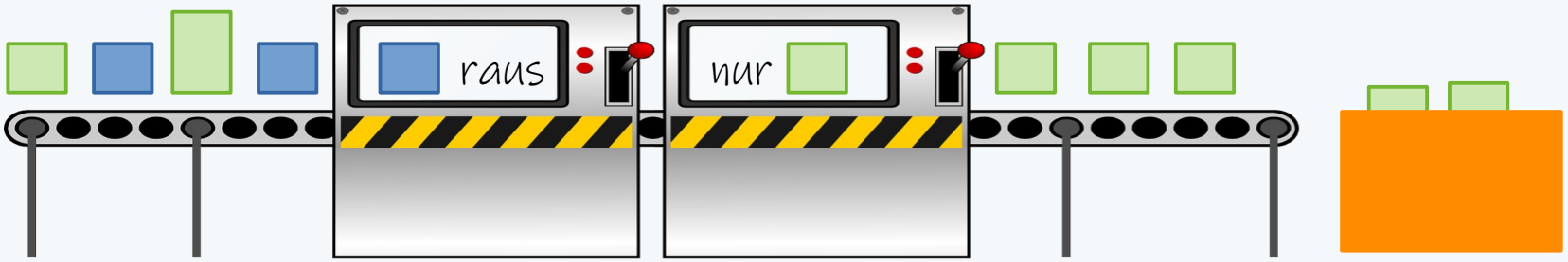
Terminale Operationen – am Beispiel



Aufgabe 7 - zur Vorbereitung
Erstellen Sie in Ihrem Paket zu Lambdas und Streams eine Klasse „Mouse“.
Jede Maus hat als („private“) Eigenschaften einen Namen („name“ vom Typ „String“) und besitzt Käsestücke („ownedCheese“ vom Typ „int“).

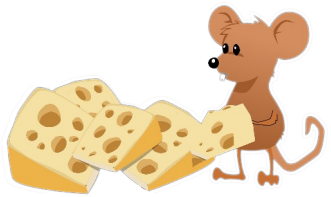
Implementieren Sie mit Hilfe Ihrer Entwicklungsumgebung passend dazu:

- alle Getter und Setter,
- einen Standard-Konstruktor (der keine Parameter entgegennimmt),
- einen Konstruktor, der Name und Anzahl Käsestücke als Parameter hat und
- überschreiben Sie die „toString“-Methode für eine sinnvolle Mausanzeige.



Stream API – mit Blick auf Arrays und „Collections“

Lösung 7 - zur Vorbereitung



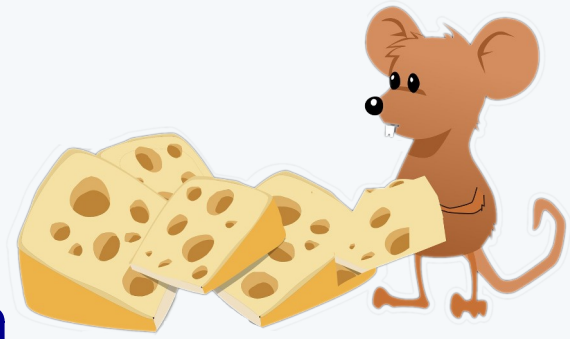
```java
package de.baleipzig.lambdasstreams;
public class Mouse {
  private String name;
  private int ownedCheese;
  //Getter und Setter
  //...
  public Mouse(String name, int ownedCheese) {
    this.name = name;
    this.ownedCheese = ownedCheese;
  }
  public Mouse() {
    this.name = "Church Mouse";
    this.ownedCheese = 0;
  }
  @Override
  public String toString() {
    return "Mouse [name=" + name + ", ownedCheese=" + ownedCheese + "]";
  }
}
```

Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

Weitere Vorbereitung

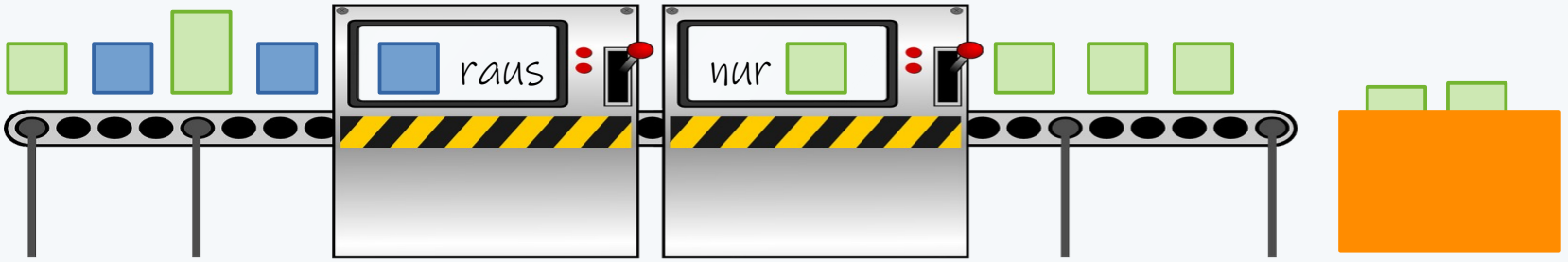
Wir brauchen zunächst mehrere Mäuse.
Erstellen Sie in Ihrer Testklasse im Paket zu Lambdas und Streams folgende Liste samt Inhalt:



```
List<Mouse> ourMice = new ArrayList<>();
ourMice.add(new Mouse("Stuart", 10));
for (int i = 0; i < 100; i++) {
  ourMice.add(new Mouse("Maus " +
    UUID.randomUUID().toString().substring(0,7),
    (int) (Math.random() * 50)));
}
```

Maus nach Ihrer Wahl

100 weitere Mäuse; wenn Sie es nicht schöner z.B. mit einer externen „Faker“-Bibliothek machen möchten, können Sie die Schleife hier übernehmen, ohne diese verstehen zu müssen



Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

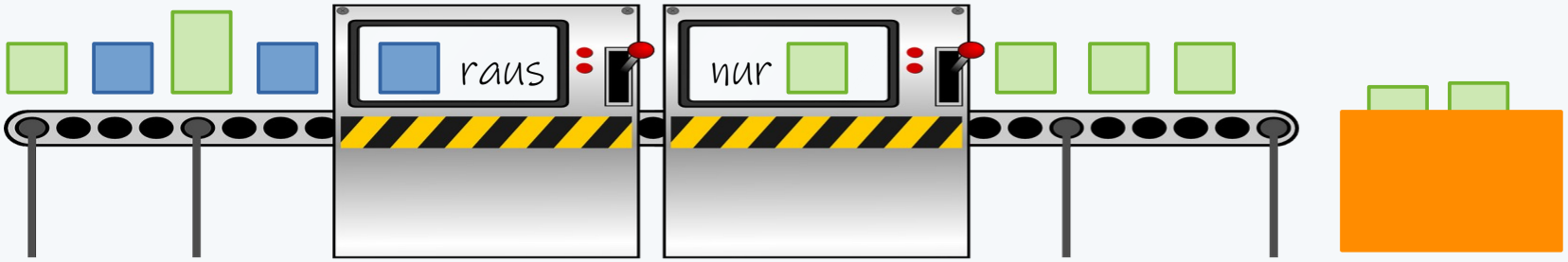
- forEach: Aktion für jedes am Ende ankommende Element

aus unserer Liste einen Stream an Mäusen erzeugen

```
ourMice.stream()
  .forEach(mouse -> System.out.println("Name: " + mouse.getName()));
```

forEach mit Lambdaausdruck, bei dem für jede Maus ihr Name ausgegeben wird

Aufgabe 8 - „forEach“
Erweitern Sie die terminale Operation, so dass für eine Maus zusätzlich zum Mausnamen jeweils noch deren Käsebesitz angezeigt wird.

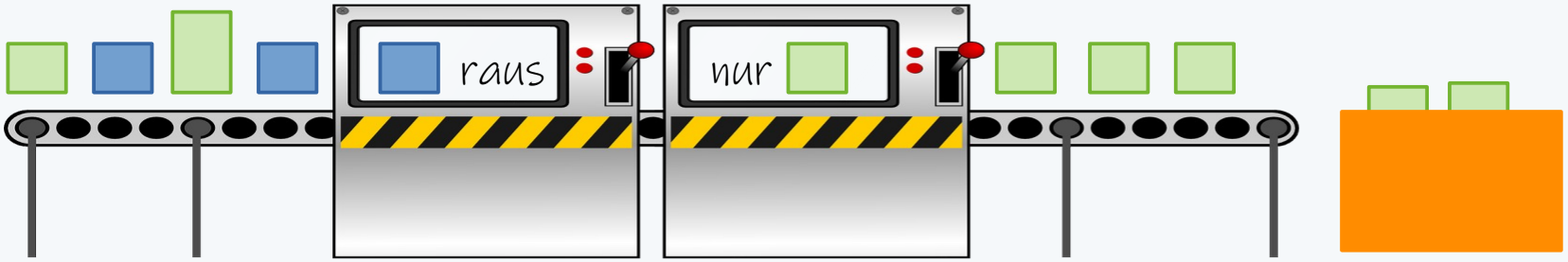


Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

Lösung 8 - „forEach“
Erweitern Sie die terminale Operation, so dass zusätzlich zum Mausnamen jeweils noch deren Käsebesitz angezeigt wird.

```
ourMice.stream()
  .forEach(mouse -> System.out.println("Name: " + mouse.getName() +
  " Cheese: " + mouse.getOwnedCheese()));
```



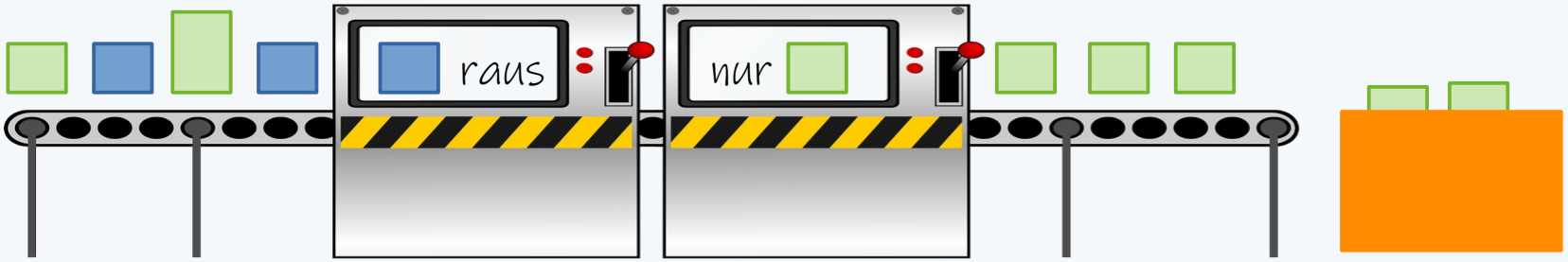
Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

- **`findFirst`**, `findAny`, `allMatch`, `nonMatch`, `anyMatch`:
  Aktionen zum Finden von Elementen bzw. zum Testen auf Existenz bestimmter Elemente

```
System.out.println("first mouse " + ourMice.stream().findFirst());
```

Aufgabe 9 - „findAny“
Wir brauchen gar nicht die erste Maus im Stream (zumal man diese sowieso nur erhält, wenn, wie bei unseren Mäusen [durch Verwendung einer Liste als Stream-Basis], eine Anordnungsreihenfolge existiert), sondern nur irgendeine.
Ändern Sie die terminale Operation entsprechend, so dass eine beliebige Maus angezeigt wird.

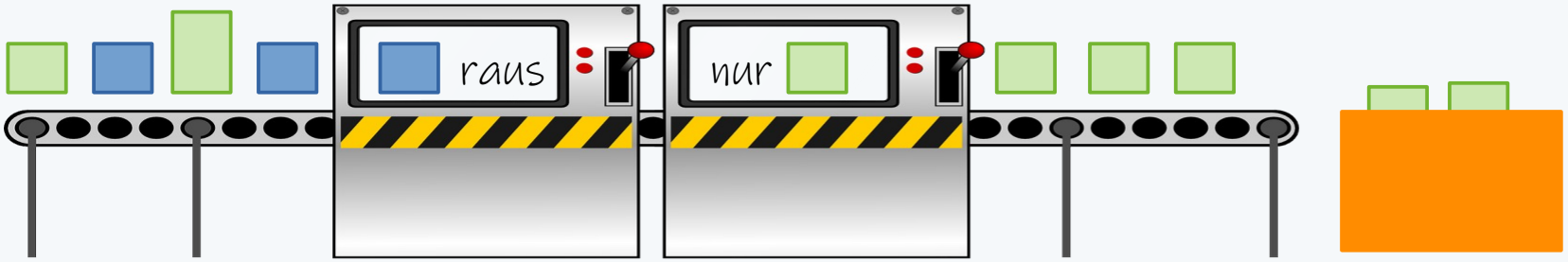


Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

Lösung 9 - „findAny“

```
System.out.println("a mouse " + ourMice.stream().findAny());
```



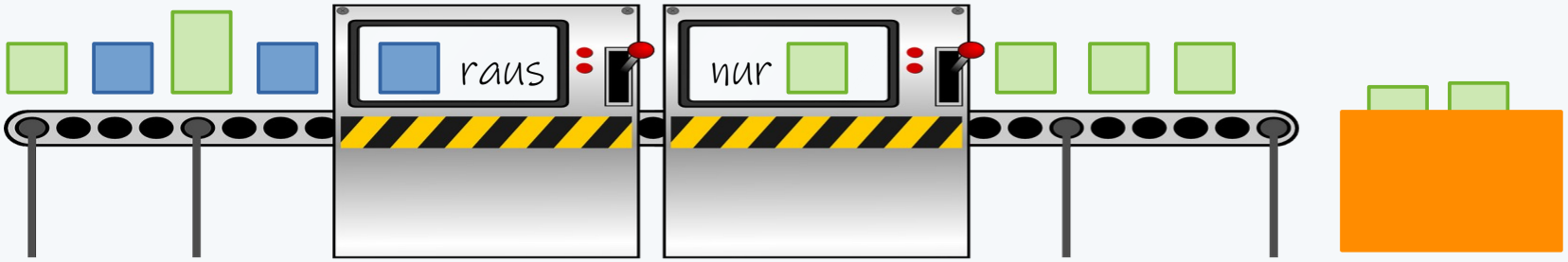
Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

- `findFirst, findAny, allMatch, nonMatch, anyMatch:`
  Aktionen zum Finden von Elementen bzw. zum Testen auf Existenz bestimmter Elemente

```
System.out.println("is there a rich mouse? " +
  ourMice.stream().anyMatch(mouse -> mouse.getOwnedCheese() > 10));
```

Aufgabe 10
Uns interessiert nicht, ob es irgendeine reiche Maus gibt, sondern ob alle Mäuse über dem Armutslevel leben.
Ändern Sie die terminale Operation entsprechend, so dass getestet wird, ob alle Mäuse mindestens 5 Käsestücke besitzen.

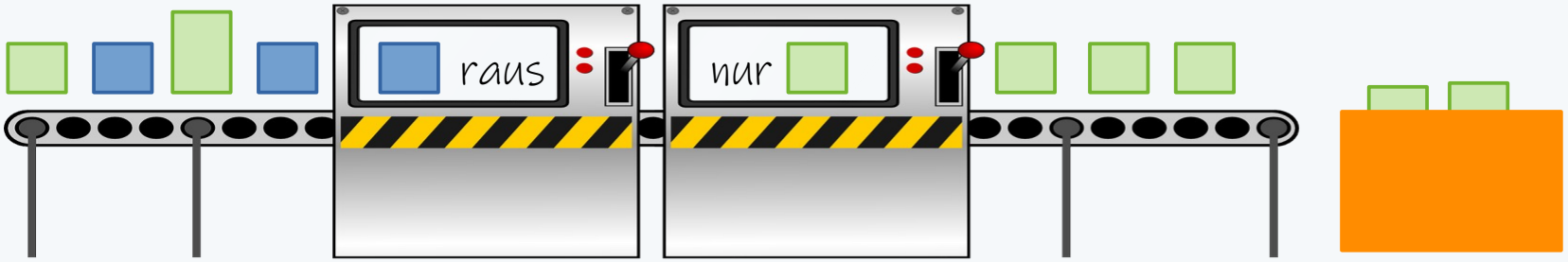


Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

Lösung 10 - „allMatch“

```
System.out.println("all above the poverty line? " +
  ourMice.stream().allMatch(mouse -> mouse.getOwnedCheese() > 5));
```



Stream API – mit Blick auf Arrays und „Collections“

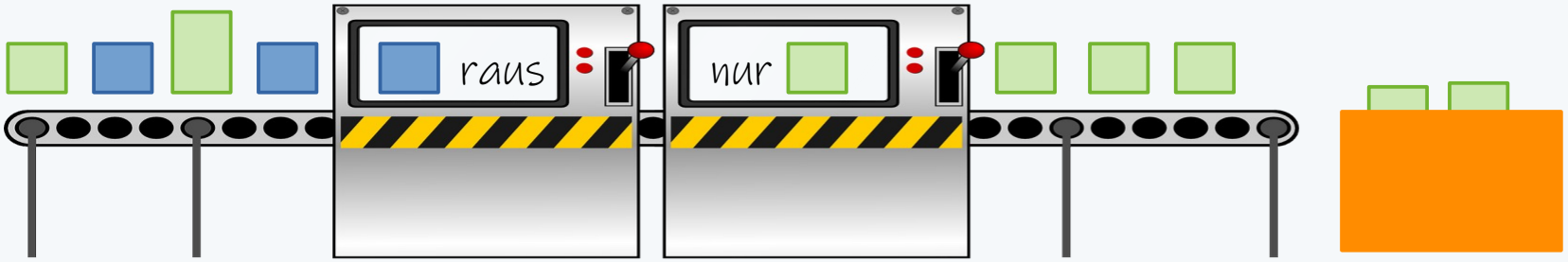
Terminale Operationen – am Beispiel

- `min, max, count`:
  Aktionen zum vergleichenden Finden und Zählen von Elementen

```
System.out.println("population size: " + ourMice.stream().count());
System.out.println("min: " + ourMice.stream()
  .min((mouse1, mouse2)
    -> Integer.compare(mouse1.getOwnedCheese(),
                       mouse2.getOwnedCheese()))
  .orElse(null));
```

im Beispiel mit Backup aus dem nichtklausurrelevanten Bereich

Aufgabe 11
Neben einer ärmsten Maus wollen wir auch eine reichste Maus finden. Implementieren Sie eine Ausgabe mit einem Stream mit einer entsprechenden terminalen Operation.



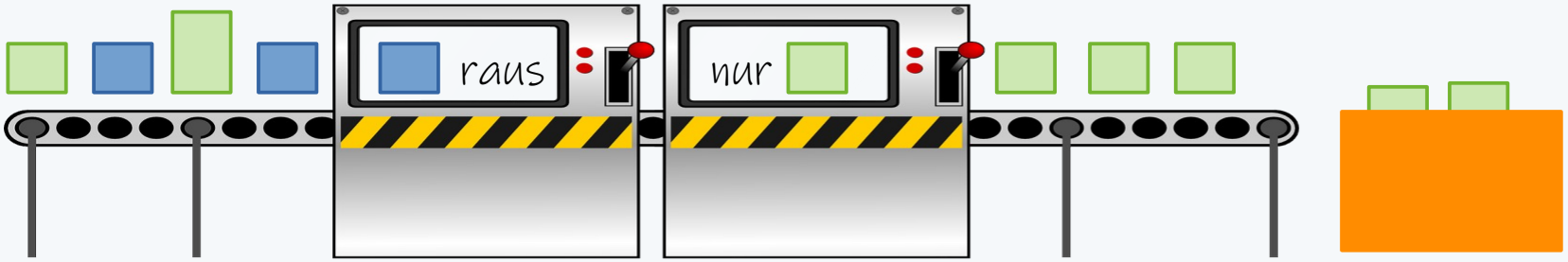
Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

Lösung 11 - „max“

```
System.out.println("max: " + ourMice.stream()
  .max((mouse1, mouse2)
    -> Integer.compare(mouse1.getOwnedCheese(),
                       mouse2.getOwnedCheese()))
  .orElse(null));
```

hier mit einfachem Backup aus dem nichtklausurrelevanten Bereich;
in einer Klausur wäre der letzte Teil nicht nötig



Stream API – mit Blick auf Arrays und „Collections“

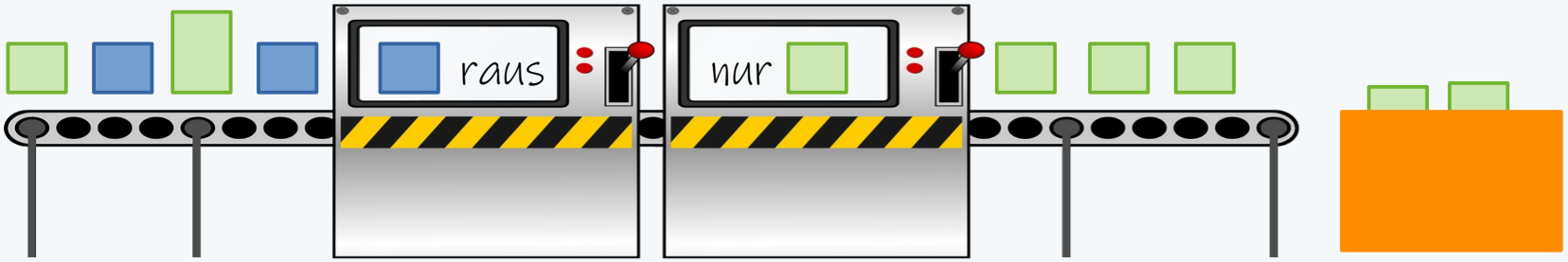
Terminale Operationen – am Beispiel

- `toArray, reduce, collect, iterator:`
  Aktionen zur Vorbereitung einer noch weiteren Verwendung der Elemente

```
Mouse[] someMiceArray =
  ourMice.stream().toArray(size -> new Mouse[size]);

List<Mouse> someMiceList =
  ourMice.stream().collect(Collectors.toList());

Iterator<Mouse> mouseIterator = ourMice.stream().iterator();
while (mouseIterator.hasNext()) {
  Mouse nextMouse = mouseIterator.next();
  System.out.println(nextMouse);
}
```



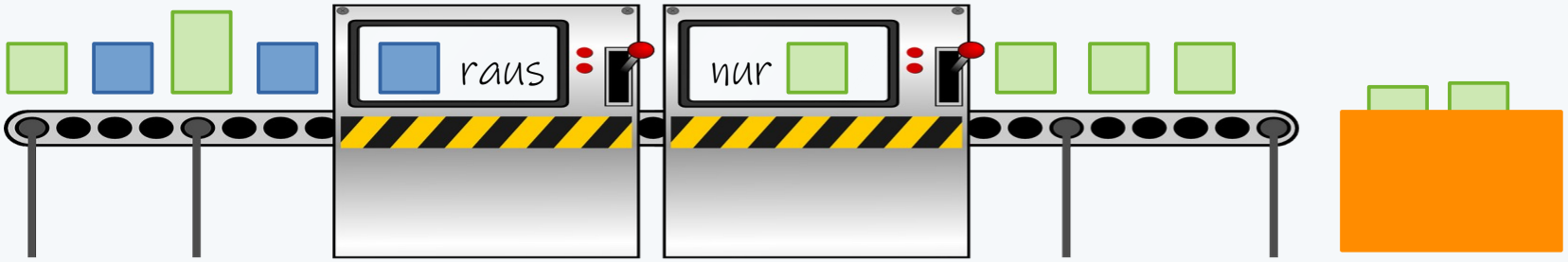
Stream API – mit Blick auf Arrays und „Collections“

Terminale Operationen – am Beispiel

```
Stream<Mouse> ourMiceStream = ourMice.stream();
ourMiceStream.forEach(mouse
  -> System.out.println("Name: " + mouse.getName()));
ourMiceStream.forEach(mouse ->
  System.out.println("Cheese: " + mouse.getOwnedCheese()));
```

Wenn „ourMiceStream“ komplizierter wäre, wäre es ggf. sehr ärgerlich, wenn wir zwei verschiedene terminale Operationen darauf ausführen wollen würden und dafür extra den ganzen Stream neu bauen müssten.

Ausweg: Supplier verwenden (nicht klausurrelevant)



Stream API – mit Blick auf Arrays und „Collections“

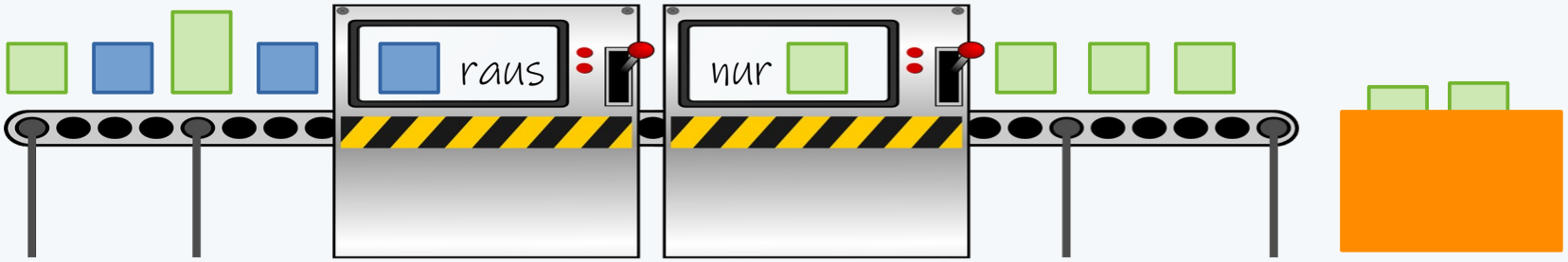
Supplier für einen Stream – am Beispiel (nicht klausurrelevant)

```
Stream<Mouse> ourMiceStream = ourMice.stream();
ourMiceStream.forEach(mouse
  -> System.out.println("Name: " + mouse.getName()));
ourMiceStream.forEach(mouse ->
  System.out.println("Cheese: " + mouse.getOwnedCheese()));
```

```
Supplier<Stream<Mouse>> streamSupplier = () -> ourMice.stream();
streamSupplier.get().forEach(mouse
  -> System.out.println("Name: " + mouse.getName()));
```

Aufgabe 12

Nutzen Sie den obigen Stream über den „Supplier“ nochmals, um auch die Käseangabe noch extra auszugeben.



Stream API – mit Blick auf Arrays und „Collections“

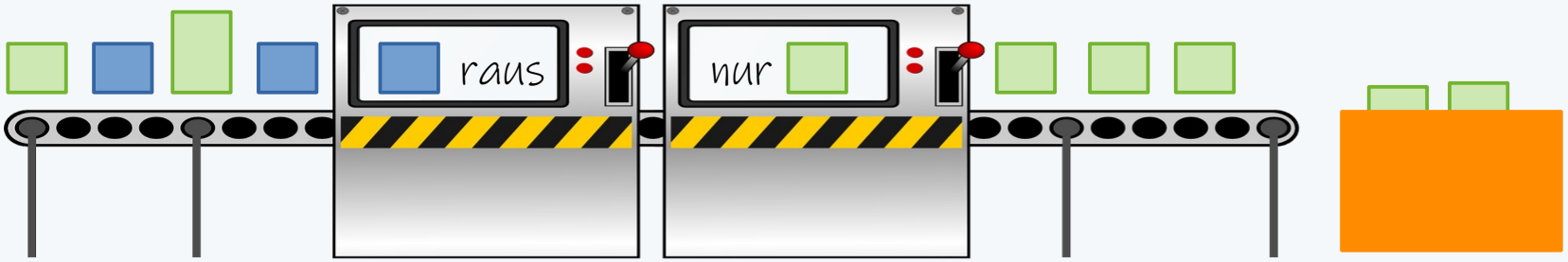
Supplier für einen Stream – am Beispiel (nicht klausurrelevant)

Lösung 12 - „Supplier“

```
Supplier<Stream<Mouse>> streamSupplier = () -> ourMice.stream();

streamSupplier.get().forEach(mouse
  -> System.out.println("Name: " + mouse.getName()));

streamSupplier.get().forEach(mouse
  -> System.out.println("Cheese: " + mouse.getOwnedCheese()));
```



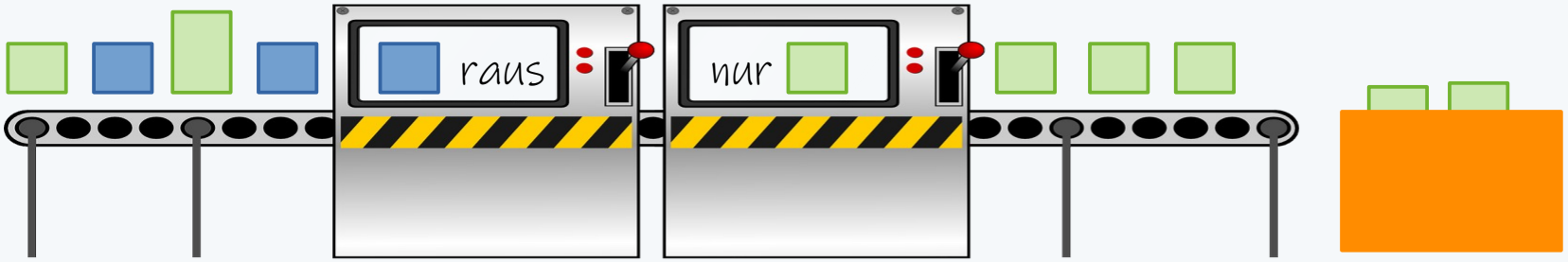
Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

- Operationen auf einem Stream, die wieder einen Stream liefern und eine schrittweise Verarbeitung im Rahmen einer Pipeline ermöglichen, z.B.:
  - distinct
  - limit, skip
  - filter, sorted, map
  - sum, average, range … für die „Zahlen-Spezialstreams“
  - neuere: takeWhile, dropWhile, …

  (wir wollen schließlich nicht nur unsere Elemente aufs Fließband packen und am Ende irgendwie wieder einsammeln, sondern auf dem Fließband auch etwas erledigen …)
- werden nur ausgeführt, wenn eine terminale Operation vorhanden ist

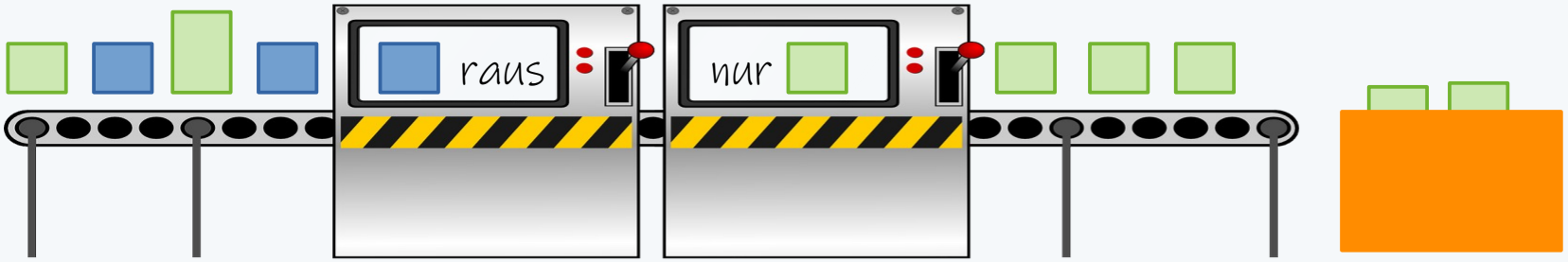
  (da kennen wir inzwischen welche ;-))



Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – weitere Unterscheidung (nicht klausurrelevant)

- **„stateful operations“** (Operationen mit Zustand):
  zusätzlich zum Element und der auszuführenden Operation werden weitere Informationen benötigt; dies erschwert oder verhindert eine parallele Ausführung und unterbricht praktisch die Pipeline (Performance!);
  - wenn man z.B. eine gewisse Anzahl Elemente aus dem Stream entnehmen möchte, muss man dies mitzählen oder
  - wenn man z.B. die Elemente im Stream sortieren möchte, dann braucht man dazu jeweils die anderen Elemente zum Vergleich
- **„stateless operations“** (Operationen ohne Zustand):
  es wird jeweils nur ein Element und die auszuführende Operation benötigt; diese Operationen lassen sich gut parallel ausführen;
  - wenn man z.B. einfach nur alle Elemente herausfiltern möchte, die eine gewisse Eigenschaft in einer gewissen Ausprägung aufweisen



Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

- `distinct`: Duplikate werden herausgefiltert, wobei für den Vergleich die „equals“-Methode benutzt wird; macht also meist nur Sinn, wenn „equals“ implementiert ist

intermediäre Operation auf unserem Mäuse-Stream

```
ourMice.stream().distinct()
  .forEach(mouse -> System.out.println("Distinct: " + mouse.getName()));
```

Aufgabe 13
Fügen Sie „ourMice“ zwei bzgl. der Eigenschaftswerte identische Mäuse hinzu und erweitern Sie Ihre Klasse „Mouse“ derart, dass dann bei obiger Ausgabe die doppelte Maus nicht doppelt angezeigt wird.

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – am Beispiel

Lösung 13 – neue Mäuse in der Testklasse

```
ourMice.add(new Mouse("Feivel", 2));
ourMice.add(new Mouse("Feivel", 2));
```

Lösung 13 – Veränderungen bei „Mouse“

```
@Override
public int hashCode() {
  return Objects.hash(name, ownedCheese);
}
@Override
public boolean equals(Object obj) {
  if (this == obj) return true;
  if (obj == null) return false;
  if (getClass() != obj.getClass()) return false;
  Mouse other = (Mouse) obj;
  return Objects.equals(name, other.name) &&
    ownedCheese ==   other.ownedCheese;
}
```

z.B. von Entwicklungsumgebung generierte „equals“ und dazu passende „hashCode“ einfügen

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

- `limit, skip`: Teil der Elemente bis oder ab einem Element entnehmen; insb. limit ist bei unendlichen Streams sehr nützlich

intermediäre Operation auf unserem Mäuse-Stream;
nur die ersten 5 Mäuse werden weiter benutzt

```
ourMice.stream().limit(5)
  .forEach(mouse -> System.out.println("Limit(5): " + mouse.getName()));
```

Aufgabe 14
Schauen Sie das Gegenstück „skip“ an und versuchen Sie, damit die ersten 90 Mäuse aus dem Stream zu nehmen.

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – am Beispiel

Lösung 14 – „skip“

```
ourMice.stream().skip(90)
  .forEach(mouse -> System.out.println("Skip(90): " + mouse.getName()));
```

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

- `filter, sorted, map`: Elemente herausfiltern, sortieren, abbilden

```
System.out.println(ourMice.stream()
  .filter(mouse->mouse.getOwnedCheese()>10).count());

ourMice.stream().sorted((mouse1,mouse2)
    ->mouse1.getOwnedCheese()-mouse2.getOwnedCheese())
  .forEach(mouse
    -> System.out.println("Sorted: Name: " + mouse.getName() +
         " Cheese: " + mouse.getOwnedCheese()));

ourMice.stream().map(mouse -> mouse.getOwnedCheese())
  .forEach(partOfMouse
    -> System.out.println("Cheese information: " + partOfMouse));
```

Aufgabe 15
Versuchen Sie, alle Mäuse sortiert nach Käsebesitz beginnend mit der Maus mit dem meisten Käse anzuzeigen.

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – am Beispiel

Lösung 15 – „sorted“

```
ourMice.stream().sorted((mouse1,mouse2)
    ->mouse2.getOwnedCheese()-mouse1.getOwnedCheese())
  .forEach(mouse -> System.out.println("Sorted 2: Name: " +
     mouse.getName() + " Cheese: " + mouse.getOwnedCheese()));
```

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

```
ourMice.stream().filter(mouse -> mouse.getOwnedCheese()>10)
  .forEach(mouse -> System.out.println("Rich Mouse Name: " +
    mouse.getName()));
```

Aufgabe 16 – *jetzt wird es schwieriger*
Die Klasse „String“ (java.lang.String) hat die Methode „startsWith“, mit der man einen „String“ auf einen bestimmten Anfang testen kann. Versuchen Sie, den Filter in obigem Beispielcode so neu zu formulieren, dass nur Mäuse angezeigt werden, deren Name mit „Maus 3“ beginnt.

Link zur Dokumentation:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/String.html#startsWith(java.lang.String)

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – am Beispiel

Lösung 16 – „filter“

```
ourMice.stream().filter(mouse -> mouse.getName().startsWith("Maus 3"))
  .forEach(something -> System.out.println("Mouse starts with 3: Name: "
    + something.getName() + " Cheese: " + something.getOwnedCheese()));
```

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen

… und weitere intermediäre Operationen („Pipeline“) – am Beispiel

```
ourMice.stream()
  .filter(mouse -> mouse.getOwnedCheese()>20)
  .sorted((mouse1,mouse2)
    -> mouse1.getOwnedCheese()-mouse2.getOwnedCheese())
  .forEach(mouse -> System.out.println("Very Rich Mouse Name: " +
     mouse.getName() + " Cheese: " +  mouse.getOwnedCheese()));
```

intermediäre Operationen liefern einen Stream und können daher direkt nacheinander ausgeführt werden

Aufgabe 17 – auch schwierig, aber machbar, wenn Sie der Reihe nach arbeiten
Nehmen Sie die ersten 20 Mäuse aus Ihrem Stream,
bilden Sie diese mittels „map“ nur auf Ihren Käsevorrat ab,
nutzen Sie danach „distinct“, um Dopplungen zu entfernen und
speichern Sie diese Werte letztendlich in einer „List<Integer>“.
Geben Sie den Inhalt der Liste auf der Konsole aus.

Zusatzaufgabe:
Sortieren Sie nach der Dopplungsentfernung noch die Werte.

Stream API – mit Blick auf Arrays und „Collections“

Intermediäre Operationen – am Beispiel

Lösung 17 inkl. Zusatzaufgabe

```
List<Integer> distinctNumberOfCheese =
  ourMice.stream()
  .limit(20)
  .map(mouse -> mouse.getOwnedCheese())
  .distinct()
  .sorted()
  .collect(Collectors.toList());

System.out.println(distinctNumberOfCheese);
```

Stream API – mit Blick auf Arrays und „Collections“

Bemerkungen (nicht klausurrelevant)

Für eine Datenquelle q und eine Operation o sollte man die Benutzung von `q.parallelStream().operation(o)` in Betracht ziehen, sofern:

- Operation o zustandslos ist, also für jedes Element unabhängig von anderen Informationen durchgeführt werden kann
- Datenquelle q effizient teilbar ist
  - viele „Collections“ sind effizient teilbar („splittable“)
  - viele andere Ressourcen (die meisten I/O-Ressourcen) sind nicht effizient teilbar
- die Ausführung mit einem sequentiellen Stream „langsam“ ist

Lambdas, Methoden-Referenzen und Stream API

Bemerkungen (nicht klausurrelevant)

- Lambdas:
  sollten immer sonst aufwendigen Klasseninstanzierungen vorgezogen werden
- Methoden-Referenzen:
  sollten eigentlich meist normalen Lambda-Ausdrücken vorgezogen werden
- Streams:
  sollten benutzt werden, wenn es damit gut funktioniert
- Parallele Streams:
  sollten benutzt werden, wenn es damit besser funktioniert

Java ist trotzdem im Kern **keine „Funktionale Programmiersprache"**.
Die funktionalen Bestandteile sind nur zu verwenden, falls benötigt, und wenn man nur solche benötigt, sollte man vielleicht eine andere Programmiersprache benutzen.

Vielen Dank für Ihre Aufmerksamkeit.