DHBW Stuttgart

# Einsatz funktionaler Programmiersprachen

## Vorlesung TM40503.1

Andreas Stadelmeier

# Java Wildcards

# Java Wildcards

```
List<String> listOfStrings = new ArrayList<>();
List<Object> listOfObjects = new ArrayList<>();

listOfStrings = listOfObjects;
```

- Ist der Code typkorrekt?

# Wiederholung: Java Wildcards

```
List<String> listOfStrings = new ArrayList<>();
List<Object> listOfObjects = new ArrayList<>();

listOfStrings = listOfObjects;

$$listOfObjects.add(100);
```

- Ist der Code typkorrekt?
- Nein!

# Java Wildcards

- `List<Object>` ist kein Subtyp von `List<String>`
- `listOfString = listOfObjects;` erzeugt Typfehler!
- Korrekt wäre:

```
List<? super String> listOfStrings;
List<Object> listOfObjects = new ArrayList<>();

listOfStrings = listOfObjects;
```

# Java Wildcards

```
List<? super String> listOfStrings;
List<Object> listOfObjects = new ArrayList<>();

listOfStrings = listOfObjects;
```

- Strings anfügen ist noch erlaubt:
  ```
  listOfStrings.add("String als Parameter");
  ```
- Die Rückgabe von `String` kann aber nicht mehr garantiert werden. Folgendes erzeugt Typfehler:
  ```
  String text = listOfStrings.get(0);
  ```

# Wildcards in Funktionstypen

Auszug aus der Klasse `java.util.function.Function`:

```
interface Function<T, R>{
        R apply(T param);
}
```

# Wildcards in Funktionstypen

```
List<String> input = ...;
List<Number> output = ...;
Function<?, ?> mapper;
for(String in : input){
        Number out;
        out = mapper.apply(in);
        output.add(out);
}
```

# Wildcards in Funktionstypen

```
List<String> input = ...;
List<Number> output = ...;
Function<? super String, ? extends Number> mapper;
for(String in : input){
        Number out;
        out = mapper.apply(in);
        output.add(out);
}
```

# Wildcards in Funktionstypen

```
List<String> input = ...;
List<Number> output = ...;
Function<? super String, ? extends Number> mapper;
for(String in : input){
        Number out;
        out = mapper.apply(in);
        output.add(out);
}
```

- In Java gilt:
- Rückgabetypen sind Kovariant
- Argumenttypen sind Kontravariant

# Motivation für Wildcards

- Die folgende Methode:

```
void deleteLowest(List<Number> numbers);
```

- lässt sich nicht mit `List<Integer>` aufrufen:

```
List<Integer> input = new ArrayList<>();
deleteLowest(input); //Fehler!
```

- Um die Methode `sort` allgemeiner auszudrücken können Wildcards verwendet werden:

```
void deleteLowest(
            List<? extends Number> numbers);
```

# Subtyprelation

- Anmerkung: Im folgenden werden die Subtyp Beziehungen in Java ohne Raw Types und Primitive Typen betrachtet
- Beispiele:
  - `ArrayList` ist ein Subtyp von `List`
  - `ArrayList<String>` $\leq^*$ `List<String>`

- ABER:
  - `List<String>` ist **kein** Subtyp von `List<Object>`
  - `List<Object>` ist **kein** Subtyp von `List<String>`

# Beispiel für `List<String>` $\not\leq^*$ `List<Object>`

```
List<String> listOfStrings = new List<String>();
List<Object> listOfObjects = new List<Object>();

listOfObjects = listOfStrings;
//Füge Object an List<String> an! Fehler
listOfObjects.add(new Object());

listOfStrings = listOfObjects;
//Fügt ein Object an List<String> an! Fehler
listOfStrings.add(listofStrings.get(0));
```

# Java Wildcards

- ? extends SimpleType
- ? super SimpleType
- ?
  - Kurzform für ? extends Object

# Subtyping mit Wildcards

Beispiele:

- List<Integer> $\leq^*$ List<Integer>
- List<Integer> $\leq^*$ List<? extends Number>
- List<Number> $\leq^*$ List<? super Integer>
- List<? super Number> $\leq^*$ List<? super Integer>
- List<? extends Integer> $\leq^*$ List<? extends Number>
- List<? super Integer> $\not\leq^*$ List<Number>
- List<? extends Integer> $\not\leq^*$ List<Integer>

```
class Beispiel<A>{
        A met1(A argument){...}
        void met2(A argument){...}
        A met3(){...}
}

Beispiel<? extends Integer> var1;
Beispiel<? super Integer> var2;
```

- Auf `var1` lassen sich die Methoden `met1` und `met2` nicht anwenden
- Auf `var2` lassen sich die Methoden `met1` und `met3` nicht anwenden

# Vergleich zu Generics

- ```
  void deleteLowest(
              List<? extends Number> numbers);
  ```
- liese sich auch mit Generics ausdrücken:

```
<T extends Number> void
        deleteLowest(List<T> numbers);
```
```

# Vergleich zu Generics

- Allerdings lässt sich folgendes nicht mit Generics beschreiben:

```
<A> void test(){
        Vector<? extends Object> v;
        v = new Vector<String>();

        Vector<A> a;
        //Fehler!
        a = new Vector<String>();
}
```

# Vergleich zu Generics

- ebenso bei Feldern
- hier sind keine Generics zugelassen

```
class Beispiel{
        //Nicht erlaubt:
        <A> Vector<A> feld;

        //Erlaubt:
        Vector<?> wcFeld;
}
```

# Lambda Ausdrücke

- Ein Lambda Ausdruck hat folgende Form:

```
(Integer p1, Integer p2) -> {
                    return p1 + p2;
          }
```

- Die Argumenttypen können ausgelassen werden:

```
(p1, p2) -> {
                    return p1 + p2;
          }
```

- Der Lambda Body kann auch nur aus einer Expression bestehen:

```
(p1, p2) -> p1 + p2;
```

# Typisierung von Lambda Ausdrücken

- Mittels eines *Functional Interface*. Dieses muss folgende Eigenschaften erfüllen:
  - Ein Interface
  - mit nur einer Methodendefinition
  - (zusätzliche statische Methoden sind erlaubt)

- Beispiel:

```
interface FunctionalInterfaceBeispiel{
        void run();
}
```

# Aufruf von Lambda Ausdrücken

```
//aus java.util.function:
interface Function<T,R>{
        R apply(T p);
}

Function<Integer, String> lambda
        = (in) -> Integer.toString(in);

System.out.println( lambda.apply(42) );
```

# Typisierung von Lambda Ausdrücken

Beispiel:

```
interface Runnable{
        void run();
}

Runnable lambda = () -> System.out.print(...);

interface Fun1<P1,R>{
        R apply(P1 p1);
}

Fun1<Object, Object> id = (a) -> a;
```

# Method References

```
interface Runnable{
        void run();
}

class MethodReference{
void methode(){
        System.out.println("Test");
}

Runnable lambda = this::methode;
}
```

# Method References

Mit statischen Methoden:

```
class MethodReference{
static void methode(){
        System.out.println("Test");
}

Runnable lambda = MethodReference::methode;
}
```

# FunN

- Wir definieren eine Menge von Klassen:
  `Fun0, Fun1, Fun2 ...`
- mit folgendem Aufbau:

```
interface FunN<R, T1, ..., TN>{
        R apply(T1 t1, ..., TN tn);
}

interface FunNVoid<T1, ..., TN>{
        void apply(T1 t1, ..., TN tn);
}
```

# Subtyping mit Funktionstypen

- in der Typtheorie gilt:
  $\Theta \rightarrow \Theta' \quad \leq^* \quad \overline{\Theta} \rightarrow \overline{\Theta'}$
  , falls $\overline{\Theta} \leq^* \Theta$ und $\Theta' \leq^* \overline{\Theta'}$
  - Contravarianz in den Argumenttypen
  - Covarianz im Returntyp

Fun1<Object, Integer> $\leq^*$ Fun1<? super String, Integer>

Fun1<String, Integer> $\leq^*$ Fun1<String,? extends Number>

# Echte Funktionstypen in Java

- Ein Feature des JavaTX Projekts
  `http://www2.ba-horb.de/~pl/javatx.html`
- Das funktionale Interface `FunN` kann sich bei der Vererbung wie Funktionstypen in Haskell verhalten
- Typsicherheit ist trotzdem gegeben
- Der JavaTX Compiler geht davon aus, dass:
  Fun1<Object, Integer> $\leq^*$ Fun1<String, Number>

# Java vs. Haskell

## Java :

```java
Arrays.asList("hallo", "Welt").stream()
        .reduce("", (a, b) -> a + b)
        .collect(Collectors.toList());
```

## Haskell :

```haskell
foldr (\a b -> a ++ b) "" ["hallo", "Welt"]
```

# Java vs. Haskell

## Java :

```
Arrays.asList(1, 2).stream()
          .map(Integer::toString)
          .collect(Collectors.toList());
```

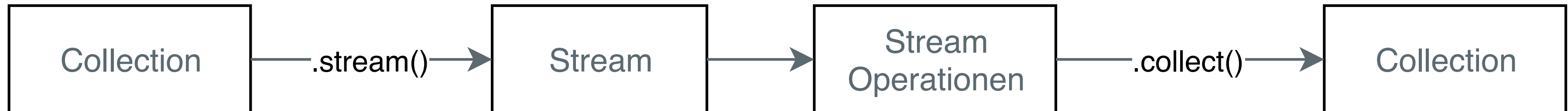
## Haskell :

```
map show [1, 2]
```

# Stream API

- Das Interface `Collection` enthält seit Java 8 die Methode `stream()`
- Stream enthält alle Objekte aus der ursprünglichen Collection
- Die Klasse Stream enthält verschiedene nützliche Funktionen
- Mit diesen können Operationen auf den im Stream enthaltenen Daten ausgeführt werden
- Anschließend kann der Stream auch wieder mittels `collect()` zu einer Collection oder Liste zusammengefasst werden

# Stream API Anwendung




- Stream-Operationen:
  - `map, filter, reduce, forEach`

- Können von der JVM Lazy ausgeführt werden
- oder parallel (`.parallelStream()`)

# Stream API - Beispiele

## Beispiel 1:

```
List<String> persons = ...
persons.stream()
        .filter(
           (name)-> name.equals(withName))
        .collect(Collectors.toList());
```

# Stream API - Reduce

Stream `reduce`:

```
interface Stream<T>{

T reduce(T identity,
        BinaryOperator<T> accumulator);
}
```

# Stream API - Beispiele

- Beispiel Reduce:

```
String concat(List<String> strings){
        return strings.stream()
                .reduce("", String::concat);
}
```

- Bei Aufruf mit folgender Liste:

`{"hallo", "Welt"}`

ist die Ausgabe: `halloWelt`

# Stream API - Collect

Stream `collect`:

```
interface Stream<T>{

<R> R collect(Supplier<R> supplier,
        BiConsumer<R,? super T> accumulator,
        BiConsumer<R,R> combiner);
}
```

# Stream API - Beispiele

- Beispiel collect:

```
List<String> streamToList(
                    Stream<String> input){
          return input.collect(
                    ()->new ArrayList<String>(),
                    ArrayList::add,
                    ArrayList::addAll);
}
```

# Stream API - Beispiele

- Beispiel Collector:

```
List<String> streamToList(
                    Stream<String> input){
        return input.collect(Collectors.toList());
}
```

# BiConsumer

```
interface BiConsumer<T,U>{
        void accept(T t, U u);
}

interface Supplier<T>{
        T get();
}
```

# Optional

- enthält entweder einen Wert oder `empty`
- nützlich um `null`-Values zu vermeiden
- Verwendung ähnlich einer Monade (Haskell)

```
//Leeres Optional-Objekt
Optional<String> stringOptional = Optional.empty();

//Optional mit Wert
stringOptional = Optional.of("Railroad-Programming");
```

# Optional - map()

- wendet Funktion auf Inhalt des `Optional` an
- ist `Optional` leer, dann wird Funktion nicht ausgeführt

```
Optional<String> stringOptional
        = Optional.of("Railroad-Programming");

stringOptional = stringOptional.map((s -> "Hallo " + s));
System.out.println(stringOptional.get());
```

Ausgabe: `Hallo Railroad-Programming`

# Optional - map()

- wendet Funktion auf Inhalt des `Optional` an

```
Optional<String> stringOptional = Optional.empty();

//Problemlos ausführbar:
stringOptional = stringOptional.map((s -> "Hallo " + s));

//Fehler!
System.out.println(stringOptional.get());
```

Exception!

# Optional

```
Optional<String> stringOptional = Optional.empty();

stringOptional = stringOptional.map((s -> "Hallo " + s));

System.out.println(stringOptional.orElse("empty"));
```

Ausgabe: `empty`

# Analogie zu Haskell

- `map`

**Java:**
```
optional.map( a -> a + "...");
```
**Haskell:**
```
fmap (\a -> a ++ "...") optional
```

# Either

- Prinzip: Klasse kann einen von zwei verschiedenen Werten annehmen

```
Either.left("Test").either(
        a -> System.out.println("Left: " + a),
        b -> System.out.println("Right: "+ b)
);
// Ausgabe: "Left: Test"

Either.right("Test").either(
        a -> System.out.println("Left: " + a),
        b -> System.out.println("Right: "+ b)
);
// Ausgabe: "Right: Test"
```

# Analogie zu Haskell

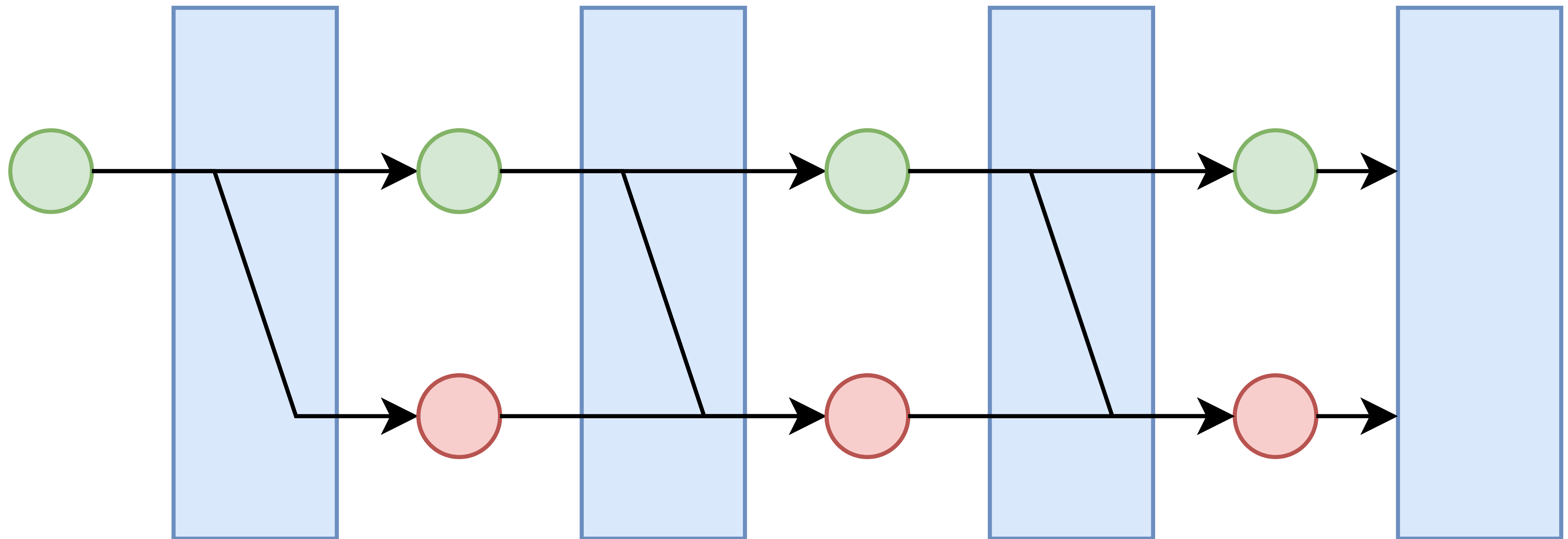
- Konstruktoren

**Java:**
```
Either.left( "...");
Either.right( "...");
```
**Haskell:**
```
Left "..."
Right "..."
```

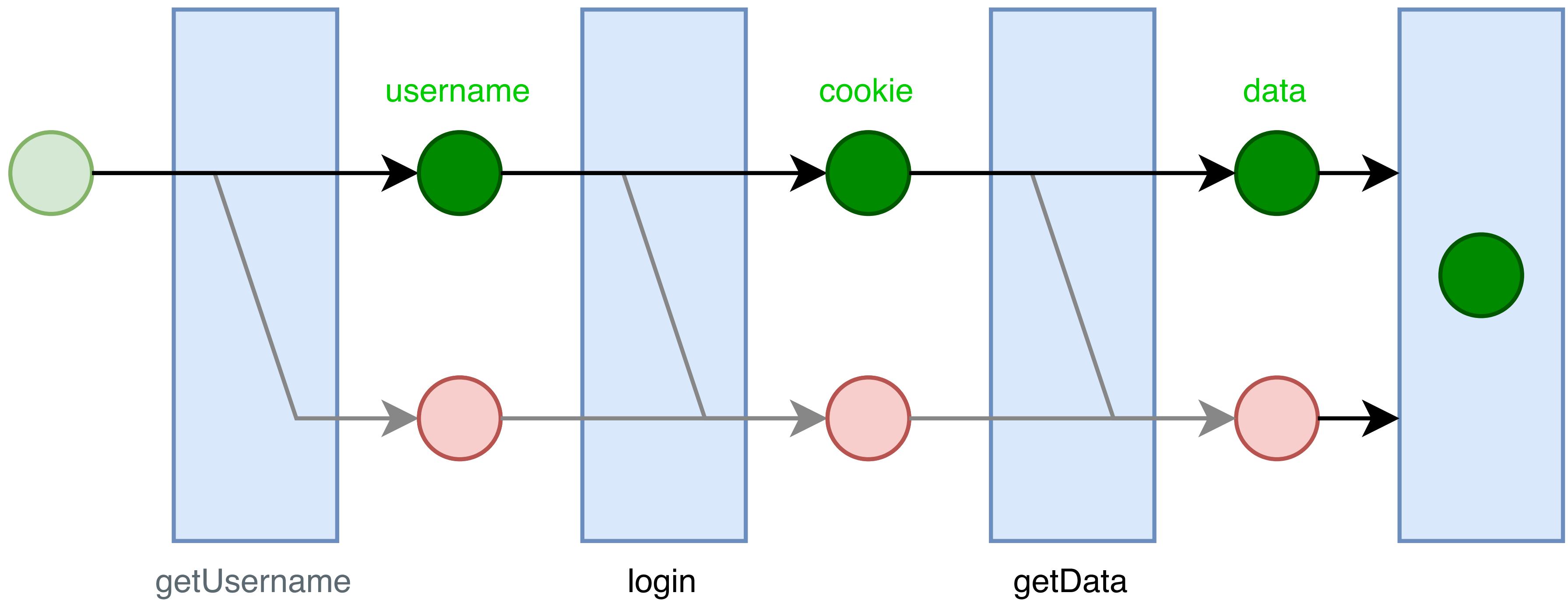
- `combine`

**Java:**
```
either.combine(a -> Either.left(a);
```
**Haskell:**
```
either >>= (\a -> Left a)
```

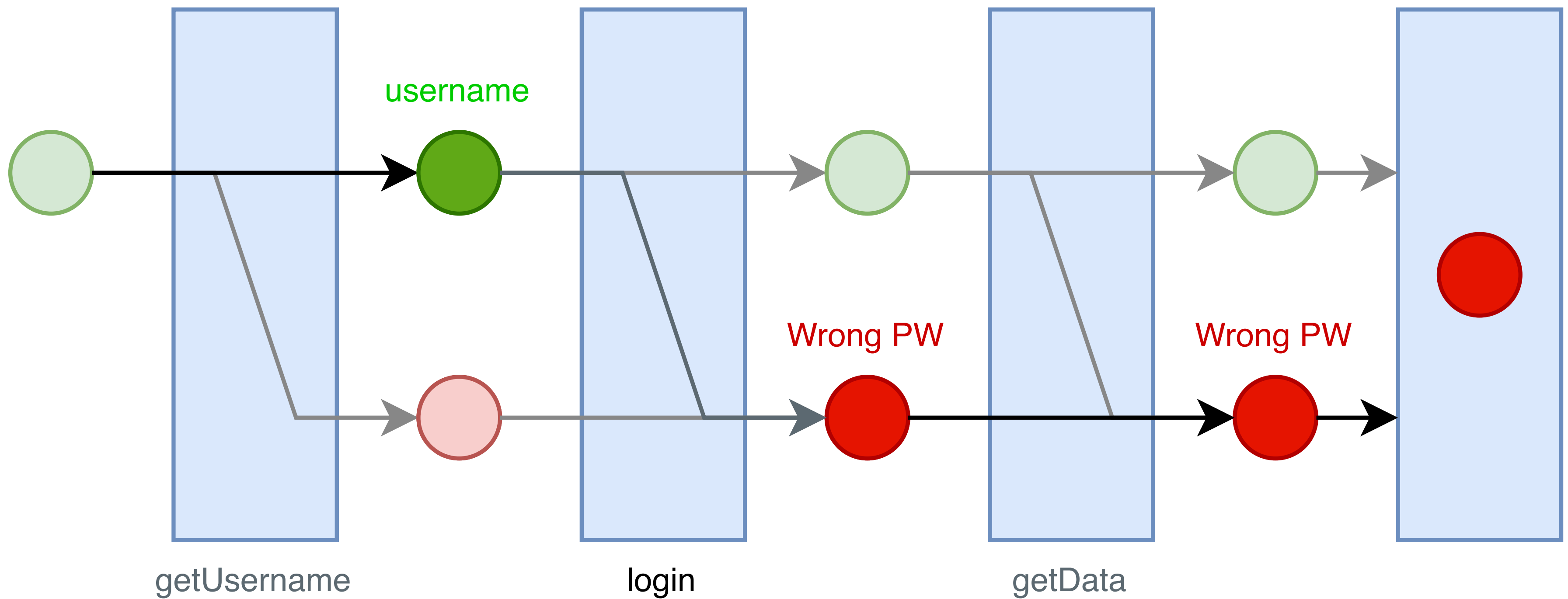
# Railroad Programming



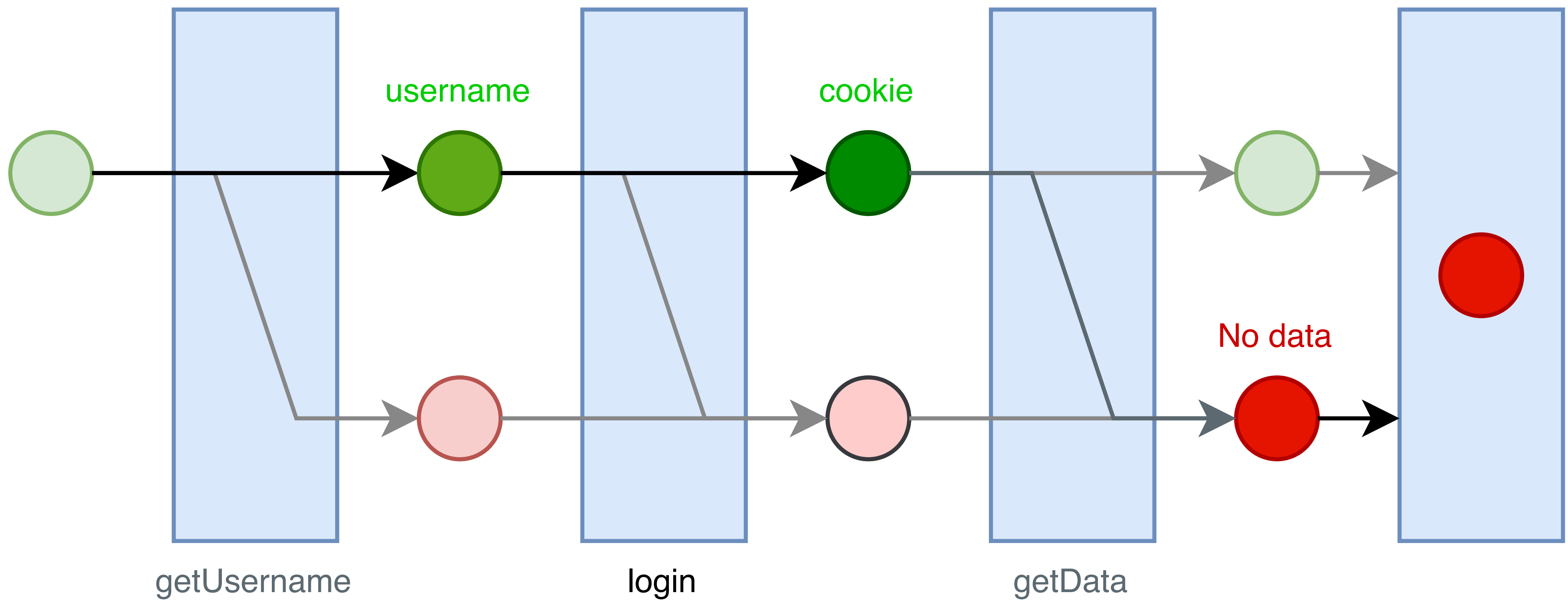
# Railroad Programming



# Railroad Programming



# Railroad Programming



# CompletableFuture - join()

```
CompletableFuture<String> completedFuture
        = CompletableFuture.completedFuture("Result String");

//join() wartet auf den Result der CompletableFuture
System.out.println(completedFuture.join());
```

# CompletableFuture - join()

```
CompletableFuture<String> completedFuture
        = CompletableFuture.completedFuture("Result String");

//join() wartet auf den Result der CompletableFuture
System.out.println(completedFuture.join());
```

```
CompletableFuture<String> failedFuture
        = CompletableFuture.failedFuture(new NullPointerException());

//join() kann Runtime-Exceptions erzugen:
System.out.println(failedFuture.join());
```

# CompletableFuture - thenApply()

- `CompletableFuture` miteinander verketten

```
CompletableFuture<String> completedFuture
        = CompletableFuture.completedFuture("Result String");

CompletableFuture<String> appliedFuture
        = completedFuture.thenApply(s -> s + " attached String");

System.out.println(appliedFuture.join());
```

# CompletableFuture - exceptionally()

```
CompletableFuture<String> completedFuture
        = CompletableFuture.completedFuture("Result String");

CompletableFuture<String> appliedFuture
        = completedFuture.thenApply(s -> {throw new RuntimeException()

CompletableFuture<String> errorChecked
        = appliedFuture.exceptionally(
                exception -> {
                System.out.println("Error");
                return null; //Alternative
        });

//Keine Exception dafür "null" ausgabe
System.out.println(errorChecked.join());
```

# HTTPClient

- HTTP-API, welche mit Java 11 eingeführt wurde
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html`