Lektion 23

Generische Methoden, Lower-Bounded Wildcards

Funktionale Programmierung
(Lambda-Ausdrücke, Functional Interfaces, Stream-API)

Generische Methoden

Manchmal wollen wir generische Methoden schreiben, ohne gleich eine generische Klasse zu schreiben.

Wir erinnern uns an unsere Mannschaft, die das Comparable Interface implementiert hat.

```
public class Mannschaft implements Comparable<Mannschaft>
{
  String name;
  int anzahlGespielteSpiele;
  int tore;
  int gegentore;
  int punkte;
  …
  @Override
  public int compareTo(Mannschaft m)
  {
    //wenn Punkte kleiner sind als die von der anderen Mannschaft, kommt this später in der Liste
    if (this.punkte < m.punkte) return 1;
    else if (this.punkte > m.punkte) return -1;
    else
    {
      if (this.tore - this.gegentore < m.tore - m.gegentore) return 1;
      else if (this.tore - this.gegentore > m.tore - m.gegentore) return -1;
    }
    return this.name.compareTo(o.name);
  }
}
```

Wir können eine Methode schreiben, die das Maximum in einer Collection von vergleichbaren Objekten findet.

```
public class Collections
{
  public static A max(Collection<A> xs)
  {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext())
    {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
```

Woher weiß der Compiler, dass mit A eine generische Typvariable (sogar eine vergleichbare generische Typvariable) gemeint ist und keine normale Klasse mit Namen A?

Wir stellen die generische Typinformation dem Rückgabetypen der Methode (anstatt dem Klassenkörper) voran, so dass vor der ersten Typangabe klar ist, welche Typen generisch sind.

```
public class Collections
{
  public static <A> A max(Collection<A> xs)
  {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext())
    {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
```

In der statischen Methode max ist die Variable A generisch.

Da A vergleichbar sein soll, müssen wir das in der Definition berücksichtigen.

```
public class Collections
{
  public static <A extends Comparable<A>> A max(Collection<A> xs)
  {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext())
    {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
```

A kann irgendein Objekt sein, dass das Comparable-Interface implementiert.

example from: http://angelikalanger.com/GenericsFAQ/FAQSections/ParameterizedMethods.html

Aus der Standardbibliothek:

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll) {
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();

    while (i.hasNext()) {
        T next = i.next();
        if (next.compareTo(candidate) > 0)
            candidate = next;
    }
    return candidate;
}
```

extends Object sorgt für Abwärtskompatibilität. Vor Java 5 hat Collections.max Object als Rückgabetyp gehabt.

Bei <T extends Comparable<? super T>> hätte die Methode Comparable als Rückgabetyp.

Funktionale Programmierung in Java I

Lambda-Ausdrücke

Functional Interfaces

Lambda-Ausdrücke

Wir haben den Fall kennengelernt, dass eine anonyme Klasse mit nur einer Methode verwendet wird.

```
public class AnonymerKlasseThread
{
  public static void main(String[] args)
  {
    int zahlZumTesten = 633910099;
    Runnable r = new Runnable()
    {
      @Override
      public void run()
      {
        boolean istPrimzahl;
        if (zahlZumTesten < 2) istPrimzahl = false;
        else istPrimzahl = true;
        for (int divisor = 2; divisor < zahlZumTesten; divisor++)
          if (zahlZumTesten % divisor == 0) istPrimzahl = false;

        System.out.println(zahlZumTesten + " ist " + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
      }
    };
    Thread t = new Thread(r);
    t.start();
  }
}
```

Wäre es nicht einfacher nur die Methode anzugeben?

```
public class AnonymerKlasseThread
{
  public static void main(String[] args)
  {
    int zahlZumTesten = 633910099;
    Runnable r =
    {
      @Override
      public void run()
      {
        boolean istPrimzahl;
        if (zahlZumTesten < 2) istPrimzahl = false;
        else istPrimzahl = true;
        for (int divisor = 2; divisor < zahlZumTesten; divisor++)
          if (zahlZumTesten % divisor == 0) istPrimzahl = false;

        System.out.println(zahlZumTesten + " ist " + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
      }
    };
    Thread t = new Thread(r);
    t.start();
  }
}
```

Das wäre schon besser.

Das Interface Runnable besteht sowieso aus nur einer Methode.

Wäre es nicht noch einfacher, nur die Methodenimplementierung anzugeben.

```
public class AnonymerKlasseThread
{
  public static void main(String[] args)
  {
    int zahlZumTesten = 633910099;

    Runnable r = () ->
    {
      boolean istPrimzahl;

      if (zahlZumTesten < 2) istPrimzahl = false;
      else istPrimzahl = true;
      for (int divisor = 2; divisor < zahlZumTesten; divisor++)
        if (zahlZumTesten % divisor == 0) istPrimzahl = false;

      System.out.println(zahlZumTesten + " ist " + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
    };

    Thread t = new Thread(r);
    t.start();
  }
}
```

Seit Java 8 ist es möglich eine anonyme Methode (Lambda Expression) anzugeben.

Der Compiler “errät” die aufzurufende Methode und kann den Typ des zu erstellenden Objekts (im Beispiel Runnable) herleiten.

Wir kommen später zu dem Beispiel zurück!

Im Programmieralltag arbeitet man häufig mit Collections.

- Bestellungen eines Kunden
- Handkarten bei einem Kartenspiel
- Freundesliste in einem sozialen Netzwerk
- Artikel in einem Online-Shop
- …

Um die Brücke von Arrays/Varargs zu Listen zu schlagen, gibt es zwei Methoden:

Arrays.asList erstellt eine veränderbare Liste.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
```

List.of erstellt eine unveränderbare Liste.

```
List<String> bBrothers = List.of("Burt", "Bronski", "Peter");
```

B-Brüder

Burt Bronski Peter

Ein sehr einfaches Beispiel zeigt, wie man normalerweise mit Collections arbeitet:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

for (int i = 0; i < bBrothers.size(); i++)
{
  System.out.println(bBrothers.get(i));
}
```

Anfällig für Fehler mit Indizes bzw. Operatoren (<, <=)

Iteration erfolgt extern.

Der Code verstößt gegen das **Tell, don‘t ask** Prinzip.

Der Code ist eher „low-level“.

Besser mit einer for-Each Schleife:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

for (String name : bBrothers)
{
   System.out.println(name);
}
```

Keine Indizes mehr.

Iteration erfolgt immer noch extern.

Der Code verstößt gegen das **Tell, don‘t ask** Prinzip.

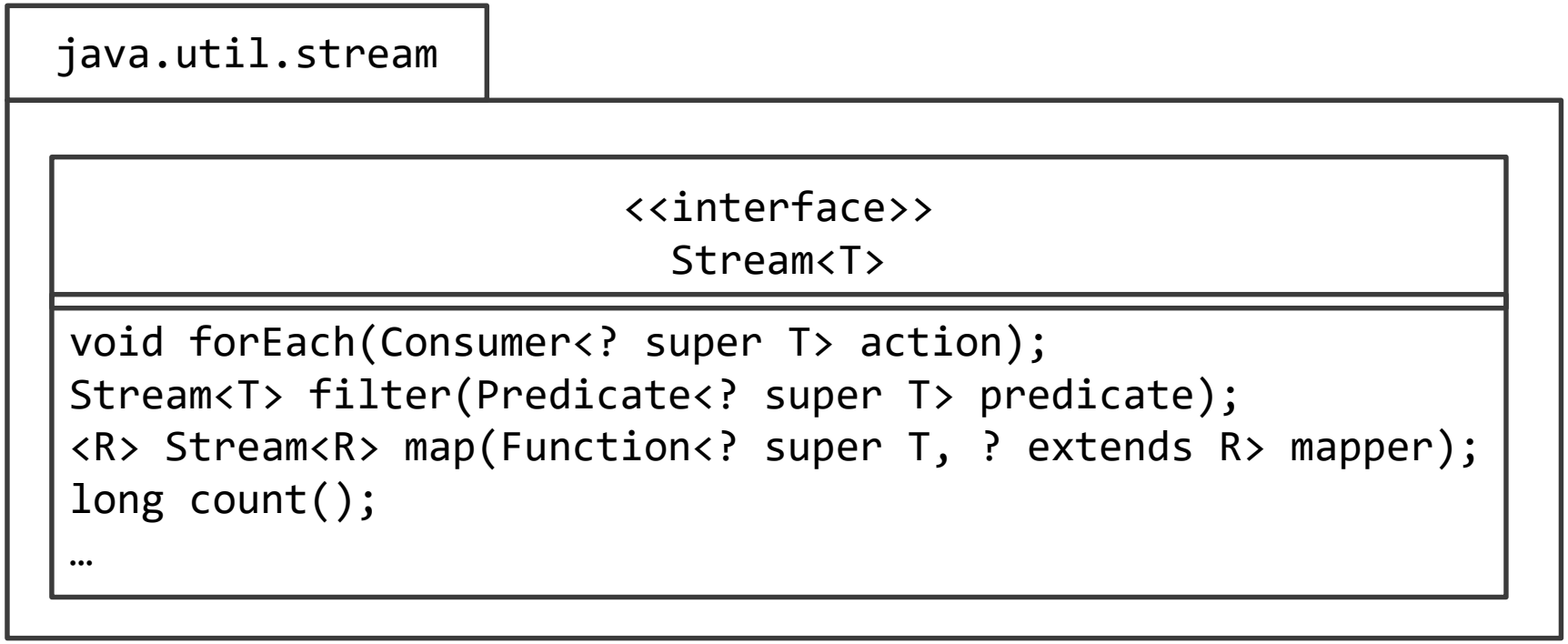
Noch besser mit Hilfe von Stream und einer anonymen Klasse?

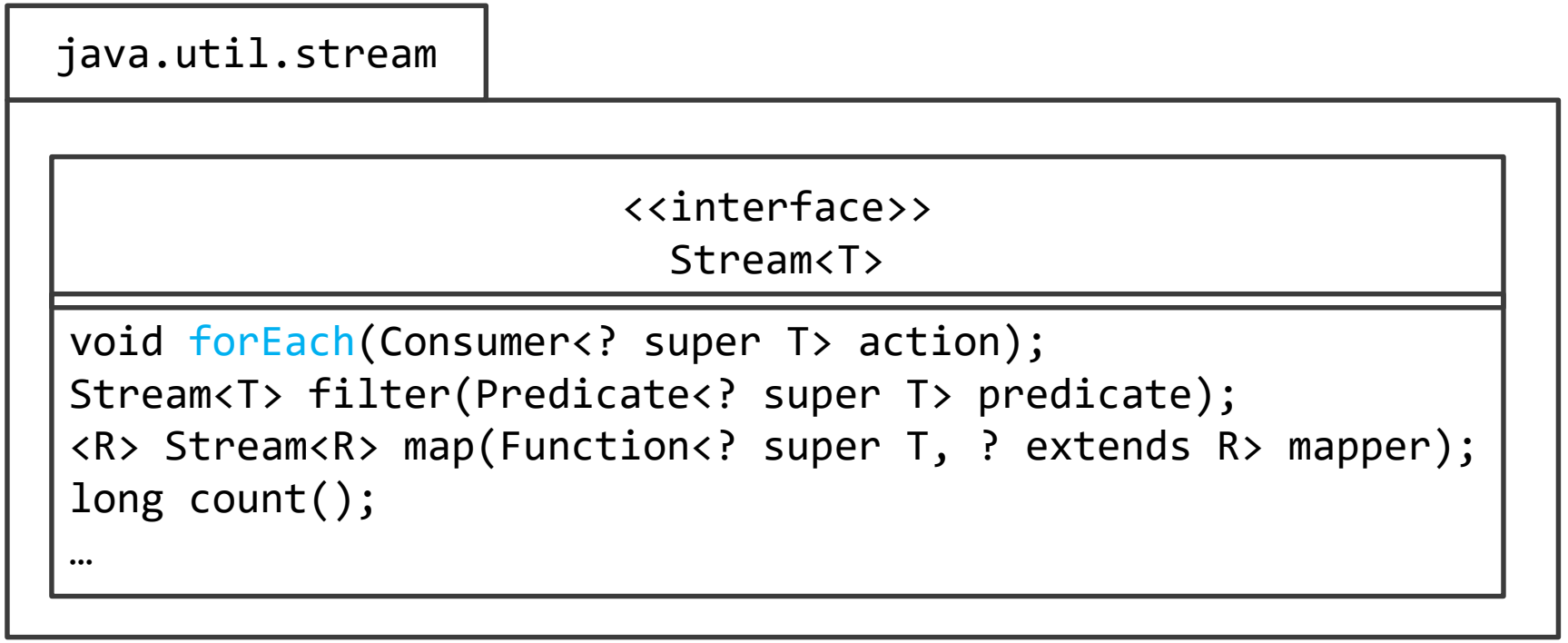
```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(...);
```

Durch den Aufruf von stream() wird ein Stream
aus den Elementen der Collection erstellt.

(Nicht zu verwechseln mit den Input- und Output-Streams)

Ein Stream stellt eine Sequenz von Elementen dar, auf der parallel oder sequentiell gearbeitet werden kann.





Die Methode forEach führt für jedes Element der Sequenz eine Aktion durch.

Die Methode forEach() erwartet ein Objekt vom Typ **Consumer.**

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(new Consumer<String>()
{
  public void accept(String name)
  {
    System.out.println(name);
  }
});
```

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

  void accept(T t);
  …
}
```

Wie ist dieser Ansatz?

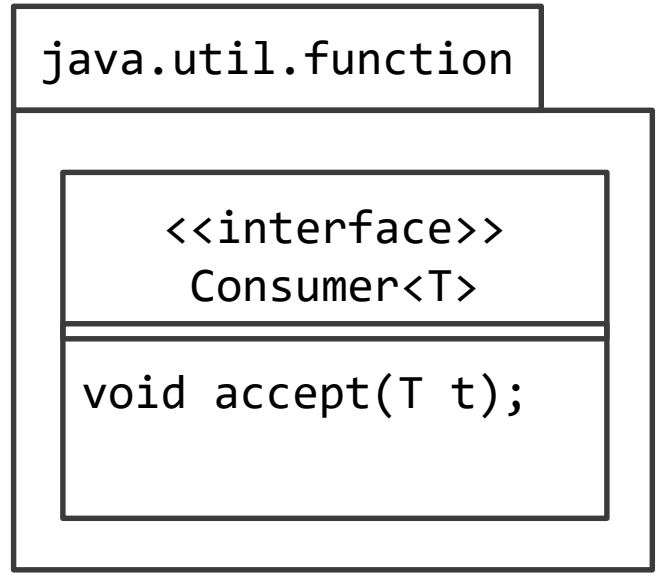
Iteration erfolgt intern.

Wir verstoßen nicht mehr gegen das **Tell, don‘t ask** Prinzip.

Aber: der Code ist lang geworden.

Consumer ist ein Functional Interface mit genau einer abstrakten Methode:

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

  void accept(T t);
  …
}
```



Jedes Interface mit genau einer abstrakten Methode ist ein Functional Interface

und **kann** mit

@FunctionalInterface
annotiert werden.

@FunctionalInterface ist dabei rein informativ für den Entwickler und wird vom Compiler ignoriert.

```
@FunctionalInterface
public interface Runnable {

  public abstract void run();
}
```

An jeder Stelle an der ein Objekt,
das ein Functional Interface implementiert, erwartet wird,
kann eine **Lambda Expression** verwendet werden.

Funktioniert eine Lambda Expression besser?

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(name -> System.out.println(name));
```

Iteration erfolgt intern.

Wir verstoßen nicht mehr gegen das **Tell, don‘t ask** Prinzip.

Der Code ist kurz und prägnant.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(name -> System.out.println(name));
```

ist semantisch gleichbedeutend mit

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(new Consumer<String>()
{
  public void accept(String name)
  {
    System.out.println(name);
  }
});
```

Bei der Verwendung des Lambda Ausdrucks “errät” der Compiler den Typ des zu erstellenden Objekts (Consumer) und die aufzurufende Methode (accept).

Wie ist eine Lambda Expression aufgebaut?

```
(parameter1, parameter2, …) -> {
                                 Anweisung1;
                                 Anweisung2;
                                     …
                                 AnweisungN;
                                 return result;
                               }
```

Parameterliste

Methodenrumpf mit Anweisungen

Rückgabewert

Eine Lambda Expression ist eine anonyme Methode, d.h. eine Methode mit Parameterliste und Anweisungen, aber ohne Namen.

Beispiel:

```
(name) -> { System.out.println(name); }
```

An die anonyme Methode wird der Parameter **name** übergeben.

Im Methodenrumpf wird **name** ausgegeben.

Lambda Expressions mit einem Parameter oder nur einer Anweisung lassen sich noch etwas kürzen:

```
(name) -> { System.out.println(name); }
```

```
name -> { System.out.println(name); }
```

```
name ->  System.out.println(name)
```

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(name -> System.out.println(name));
```

Für jedes Element des Streams wird eine anonyme Methode ausgeführt,

- die das Element als **name** an die Methode übergibt und
- System.out.println(**name**) als Anweisung ausführt.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream().forEach(name -> System.out.println(name));
```

Woher weiß der Compiler, welchen Datentyp **name** hat?

List<String> ist eine Liste von Strings

Stream<String> ist eine Sequenz von Strings

Jedes Element aus dem Stream ist auch vom Typ String.

Beispiel: Wir wollen einen 30% Discount für Artikelpreise umsetzen:

```
List<Double> articlePrices = Arrays.asList(10.0, 15.0, 20.0);

articlePrices.stream().forEach(price -> System.out.println(price * 0.7));
```

Woher weiß der Compiler, welchen Datentyp **price** hat?

List<Double> ist eine Liste von Double

Stream<Double> ist eine Sequenz von Double

Jedes Element aus dem Stream ist auch vom Typ Double.

Was würde ohne Generics passieren?

```
List badArticlePrices = Arrays.asList(10.0, 15.0, 20.0);

badArticlePrices.stream().forEach(price -> System.out.println(price * 0.7));
```

Woher weiß der Compiler, welchen Datentyp **price** hat?

List ist eine Liste von Objects

Stream ist eine Sequenz von Objects

Der Compiler wirft einen Fehler bei der Multiplikation von einem Objekt (price) mit 0,7

(The operator * is undefined for the argument type(s) Object, double).

Kommen wir zum Primzahl Lambda-Beispiel zurück:

```
() ->
    {
      boolean istPrimzahl;

      if (zahlZumTesten < 2) istPrimzahl = false;
      else istPrimzahl = true;
      for (int divisor = 2; divisor < zahlZumTesten; divisor++)
        if (zahlZumTesten % divisor == 0) istPrimzahl = false;

      System.out.println(zahlZumTesten + " ist " + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
    }
```

Obige Lambda Expression hat

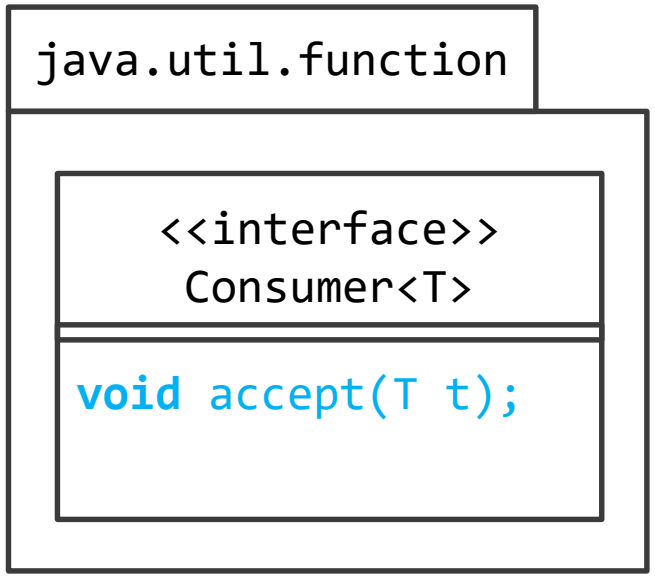
- eine leere Parameterliste
- einen Methodenrumpf mit der Primzahlenberechnung.

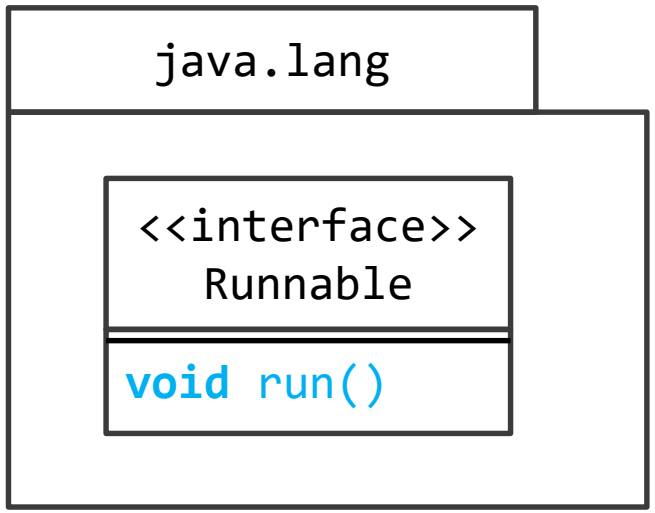
Kommen wir zum ersten Lambda-Beispiel zurück:

```
Runnable r = () ->
    {
      boolean istPrimzahl;

      if (zahlZumTesten < 2) istPrimzahl = false;
      else istPrimzahl = true;
      for (int divisor = 2; divisor < zahlZumTesten; divisor++)
        if (zahlZumTesten % divisor == 0) istPrimzahl = false;

      System.out.println(zahlZumTesten + " ist " + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
    };
```

Der Compiler errät, dass er ein Objekt vom Typ Runnable erstellen muss und die anonyme Methode der Methode void run() vom Interface Runnable entspricht.

Das Interface Runnable ist – wie Consumer – ein Functional Interface.





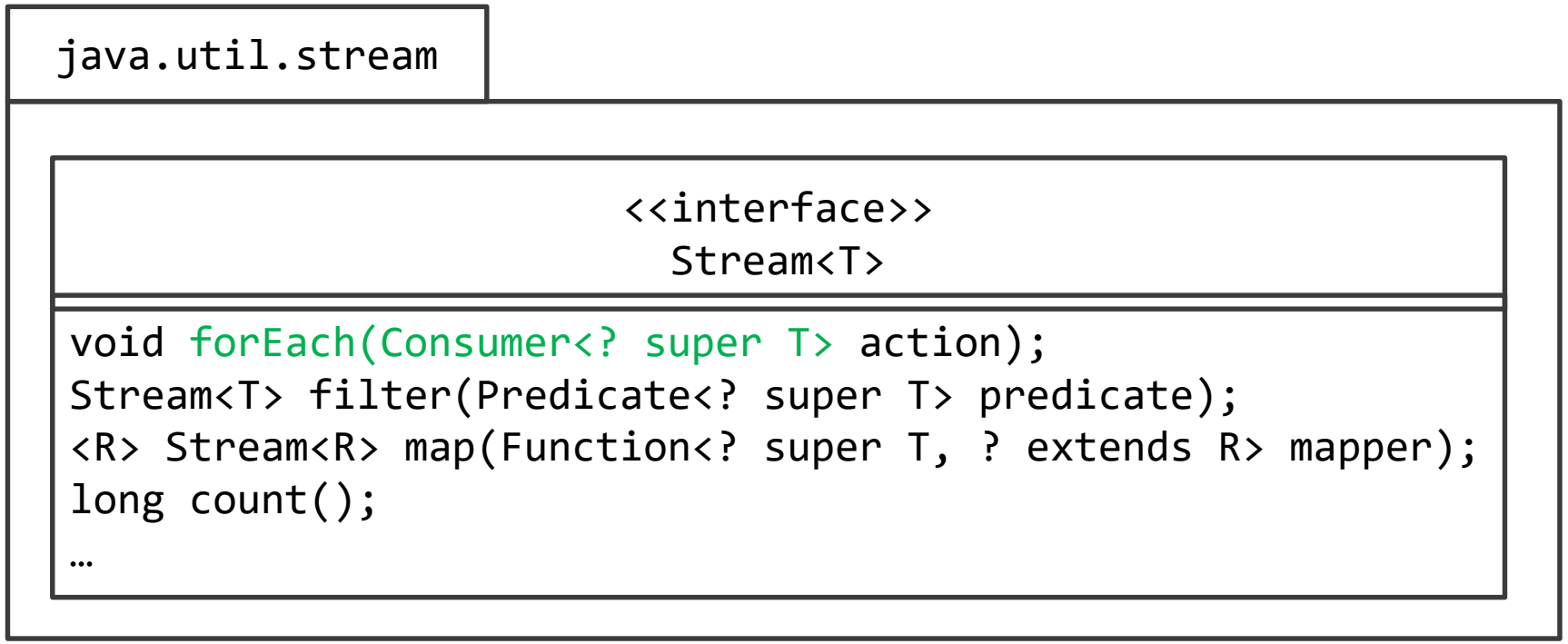
Beide Methoden haben Rückgabetyp **void**.

Unsere Lambda Expressions hatten bisher dementsprechend auch keinen Rückgabewert.

Generics

Lower-Bounded Wildcards

Was bedeutet Consumer<? super T> im Interface Stream<T>?

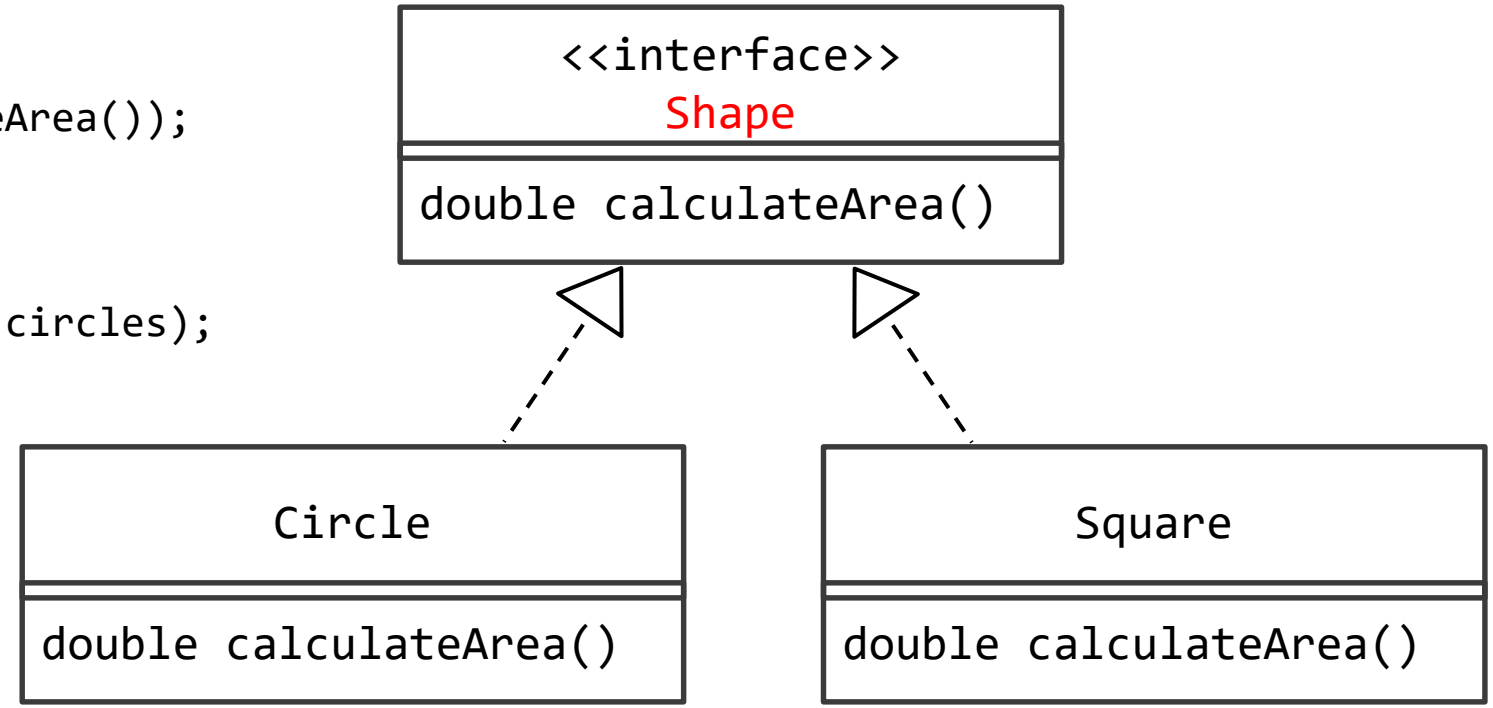


Die Methode forEach akzeptiert einen **Consumer für Typ T**
oder einen **Consumer für eine Oberklasse von T**

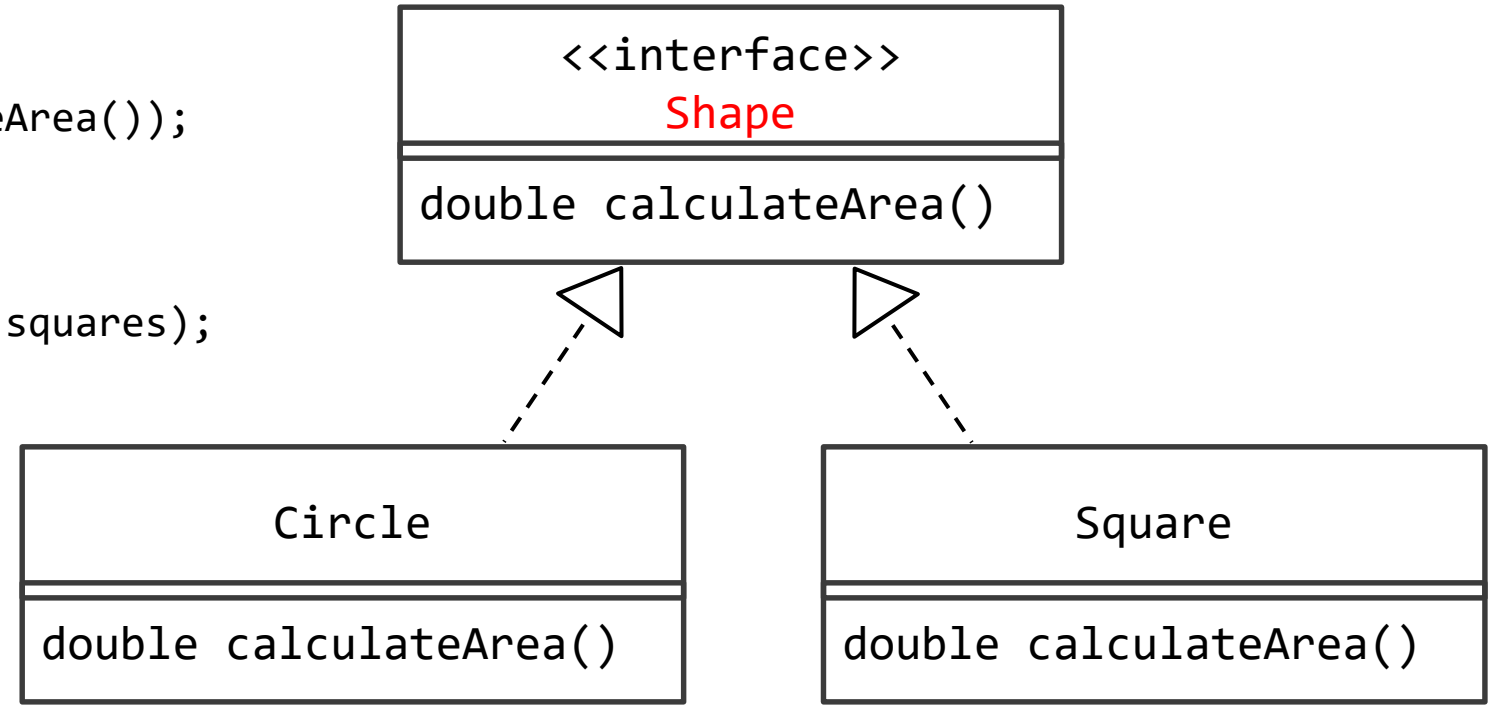
Warum ist das Akzeptieren von Typ T nicht in jedem Fall ausreichend?

```
public static void main(String[] args)
{
  Circle c = new Circle(5);
  Circle c2 = new Circle(4);
  Circle[] circles = new Circle[]{c,c2};

  Consumer<Shape> consumer = new Consumer<Shape>()
  {
    @Override
    public void accept(Shape shape)
    {
      System.out.println(shape.calculateArea());
    }
  };

  Stream<Circle> stream = Arrays.stream(circles);
  stream.forEach(consumer);
}
```

obigen Consumer könnte man auch für ein Array von Squares wiederverwenden.

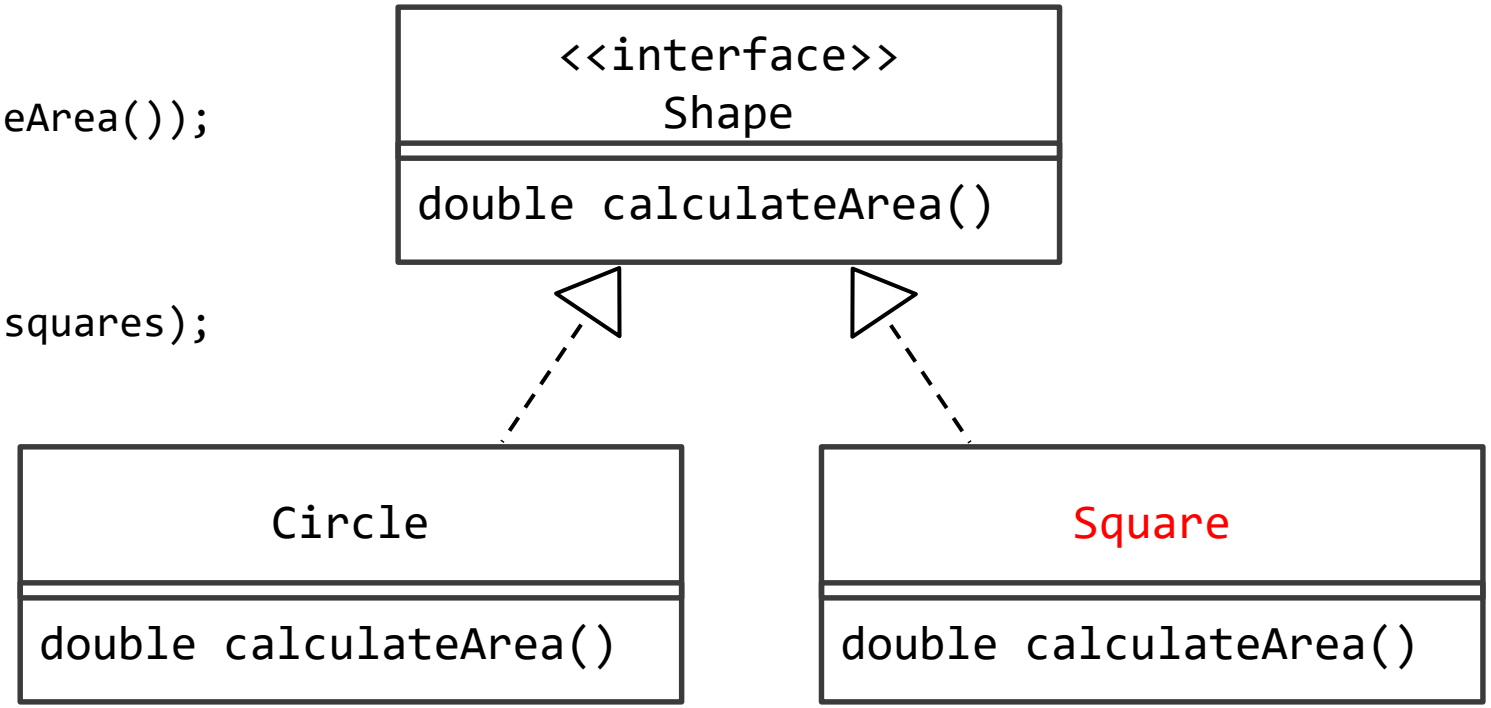


```
public static void main(String[] args)
{
  Square s = new Square(5);
  Square s2 = new Square(4);
  Square[] squares = new Square[]{s,s2};

  Consumer<Shape> consumer = new Consumer<Shape>()
  {
    @Override
    public void accept(Shape shape)
    {
      System.out.println(shape.calculateArea());
    }
  };

  Stream<Square> stream = Arrays.stream(squares);
  stream.forEach(consumer);
}
```



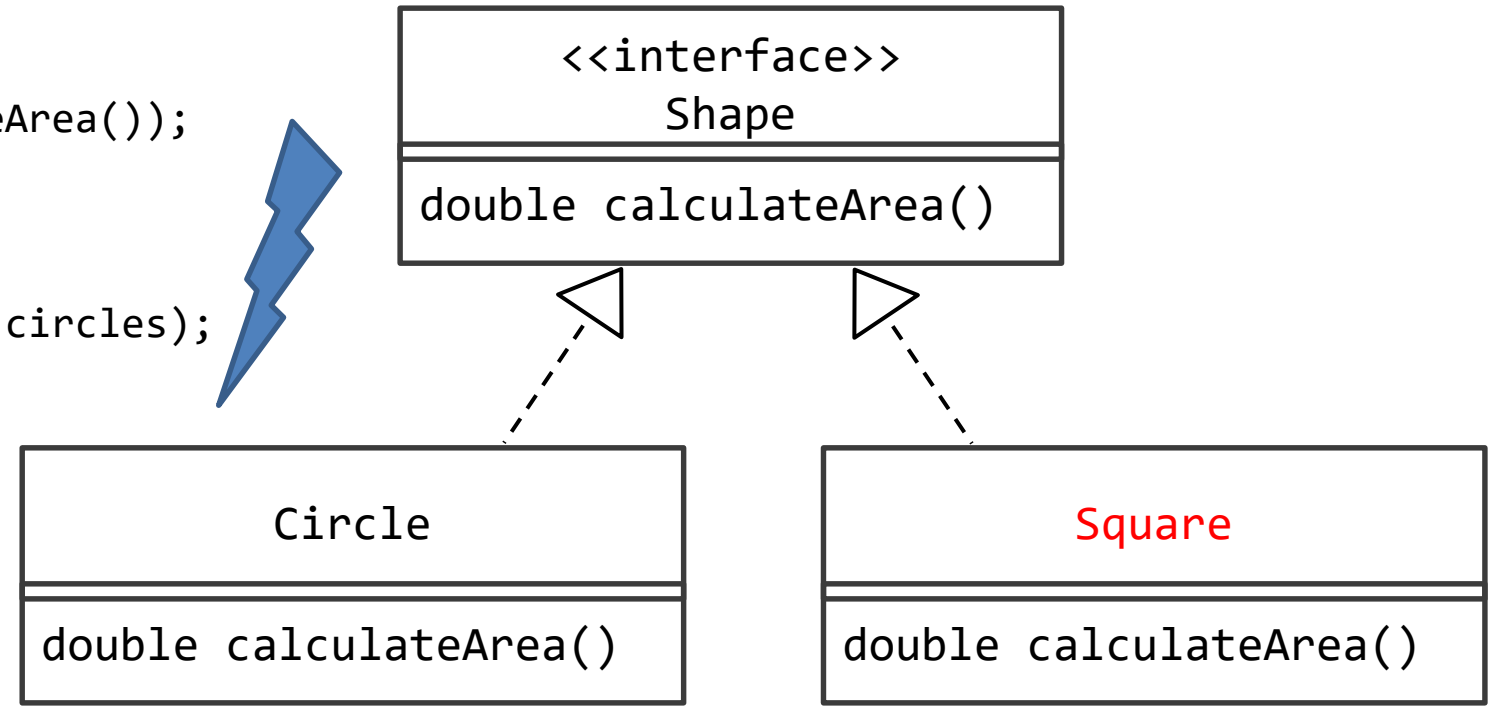
```
public static void main(String[] args)
{
  Square s = new Square(5);
  Square s2 = new Square(4);
  Square[] squares = new Square[]{s,s2};

  Consumer<Square> consumer = new Consumer<Square>()
  {
    @Override
    public void accept(Square square)
    {
      System.out.println(square.calculateArea());
    }
  };

  Stream<Square> stream = Arrays.stream(squares);
  stream.forEach(consumer);
}
```

Man kann nur Square als Typ für den Consumer verwenden, …

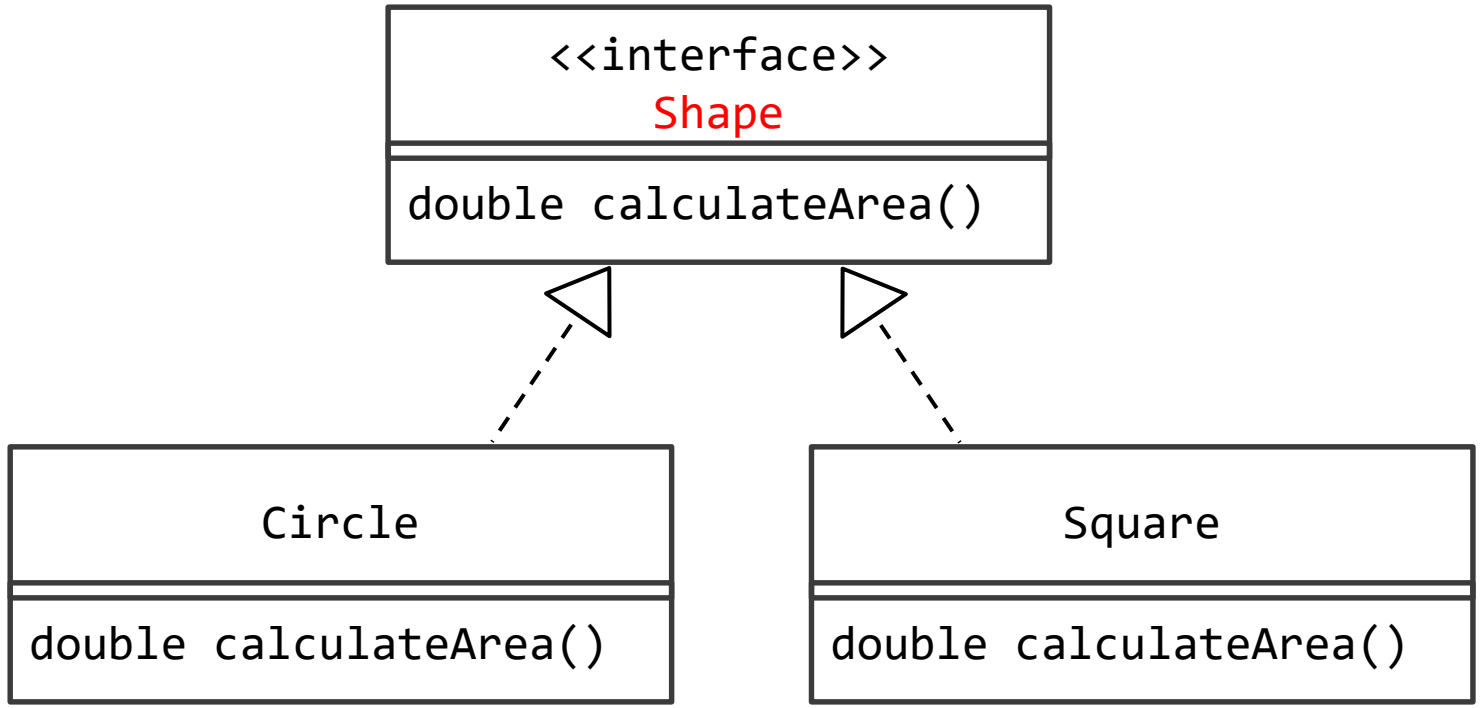


```
public static void main(String[] args)
{
  Circle c = new Circle(5);
  Circle c2 = new Circle(4);
  Circle[] circles = new Circle[]{c,c2};

  Consumer<Square> consumer = new Consumer<Square>()
  {
    @Override
    public void accept(Square shape)
    {
      System.out.println(shape.calculateArea());
    }
  };

  Stream<Circle> stream = Arrays.stream(circles);
  stream.forEach(consumer);
}
```

…dann wäre er aber nicht für Circles anwendbar.



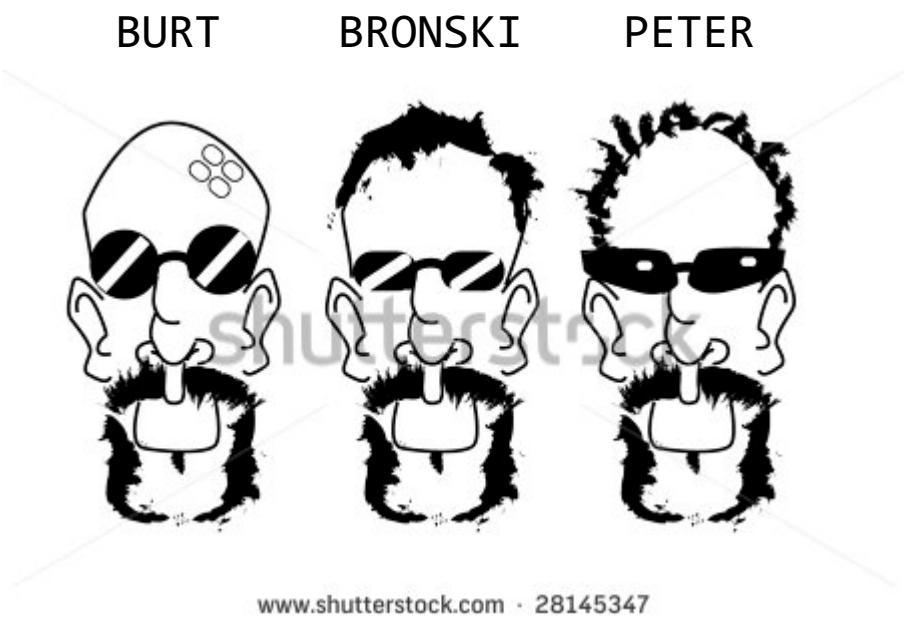
```
public static void main(String[] args)
{
  Circle c = new Circle(5);
  Circle c2 = new Circle(4);
  Circle[] circles = new Circle[]{c,c2};

  Consumer<Shape> consumer = shape -> System.out.println(shape.calculateArea());

  Stream<Circle> stream = Arrays.stream(circles);
  stream.forEach(consumer);
}
```



Den Consumer kann man auch mit Hilfe einer Lambda Expression formulieren.

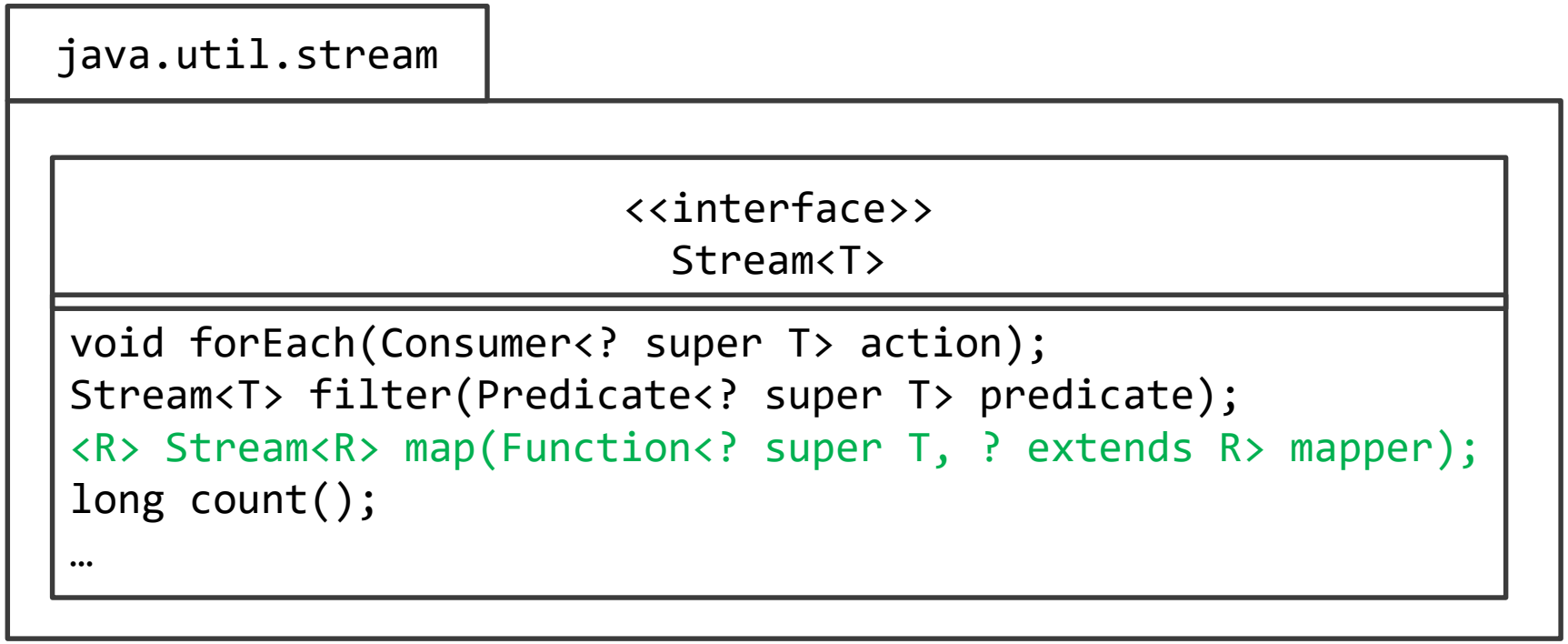
map-Methode

Damit die B-Brüder einschüchternder wirken,
schreiben sie ihre Namen immer in Großbuchstaben.



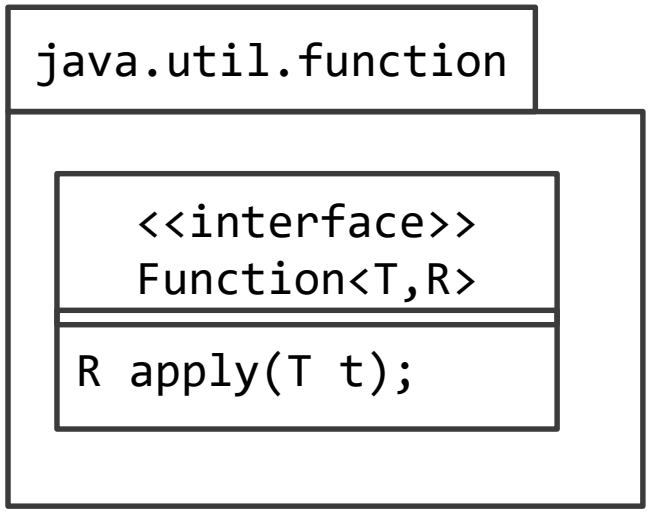
Dazu müssen wir alle Elemente im Stream „ändern“.

Um „Änderungen“ auf jedem Element des Streams durchzuführen, verwenden wir die map-Methode.



map() gibt einen **neuen** Stream von einem anderen (oder gleichen) Datentyp zurück.

Die Änderung, die für jedes Element durchgeführt werden soll,
wird durch ein Function-Objekt spezifiziert,
das an die map-Methode übergeben wird.



```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
.map(



)
.forEach(name -> System.out.println(name));
```

Wir führen auf dem Stream zuerst die map-Methode aus.

Wir geben die Function durch eine anonyme Klasse an.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
.map(new Function<String,String>(){
  @Override
  public String apply(String name)
  {
    return name.toUpperCase();
  }
})
.forEach(name -> System.out.println(name));
```

Der String name ist ein Element aus dem Stream.

map() erzeugt einen (neuen) Stream von Strings

name.toUpperCase() erstellt aus name einen neuen String
und fügt ihm dem (neuen) Stream hinzu.

Da **Function** ebenfalls ein Functional Interface ist, können wir auch hier eine Lambda Expression verwenden.

Eine Function hat einen Rückgabewert.
Wie können wir den Rückgabewert in der Lambda Expression angeben?

Explizit mit einer return-Anweisung in einem Block:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(name -> { return name.toUpperCase(); })
    .forEach(name -> System.out.println(name));
```

Da **Function** ebenfalls ein Functional Interface ist, können wir auch hier eine Lambda Expression verwenden.

Eine Function hat einen Rückgabewert.
Wie können wir den Rückgabewert in der Lambda Expression angeben?

Implizit, wenn nur eine Anweisung ausgeführt wird:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(name -> System.out.println(name));
```

Das Ergebnis von name.toUpperCase() wird zurückgegeben.

Die B-Brüder haben eine ausgeklügelte Regel, wie sie denjenigen ermitteln, der den nächsten Auftrag ausführt.

BURT BRONSKI PETER

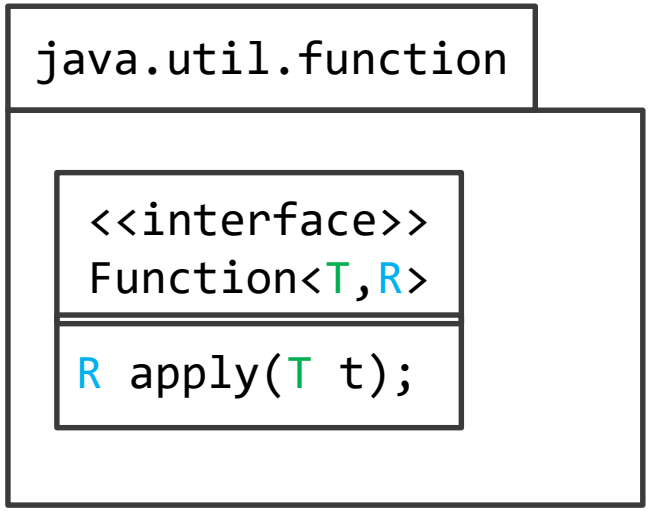
Die B-Brüder haben eine ausgeklügelte Regel, wie sie denjenigen ermitteln, der den nächsten Auftrag ausführt.

BURT BRONSKI PETER

Es ist der mit dem längsten Namen.

Da durch die apply-Methode von Function auch ein anderer Datentyp zurückgegeben werden kann,
können wir einen String (Namen) auf einen Integer (dessen Länge) mappen.

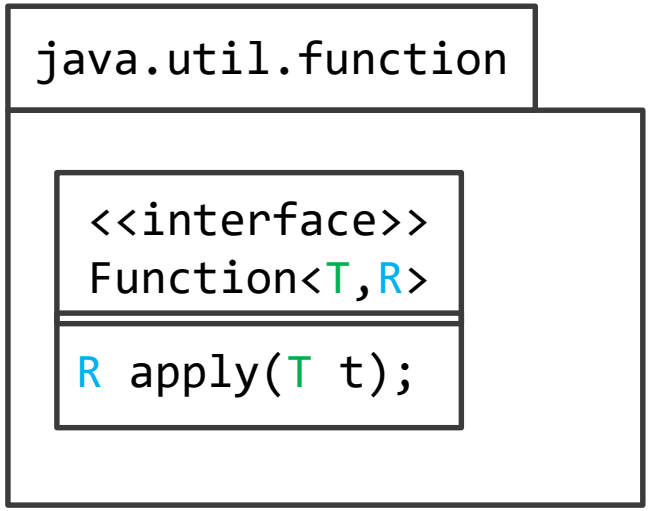
```
bBrothers.stream()
    .map(new Function<String, Integer>(){
      @Override
      public Integer apply(String name)
      {
        return name.length();
      }
    })
    .forEach(count -> System.out.println(count));
```



map() erzeugt einen Stream von Integer

Oder mit einer Lambda Expression:

```
bBrothers.stream()
    .map(name -> name.length())
    .forEach(count -> System.out.println(count));
```



Die map()-Methode arbeitet übrigens sehr ähnlich zu einer mathematischen Funktion.

Jedes Element der Definitionsmenge (d.h. des Streams) wird anhand einer Abbildungsvorschrift auf ein neues Element eines Wertebereichs (d.h. eines neuen Streams) abgebildet.

$$f: \mathbb{R} \to \mathbb{R}$$
$$f(x) := x + 1, \quad \forall x \in \mathbb{R}$$

$$f: \mathbb{R} \to \mathbb{R}$$
$$x \mapsto x + 1, \quad \forall x \in \mathbb{R}$$

```
.map(x -> x + 1)
```

Ein Element x wird auf x + 1 abgebildet.

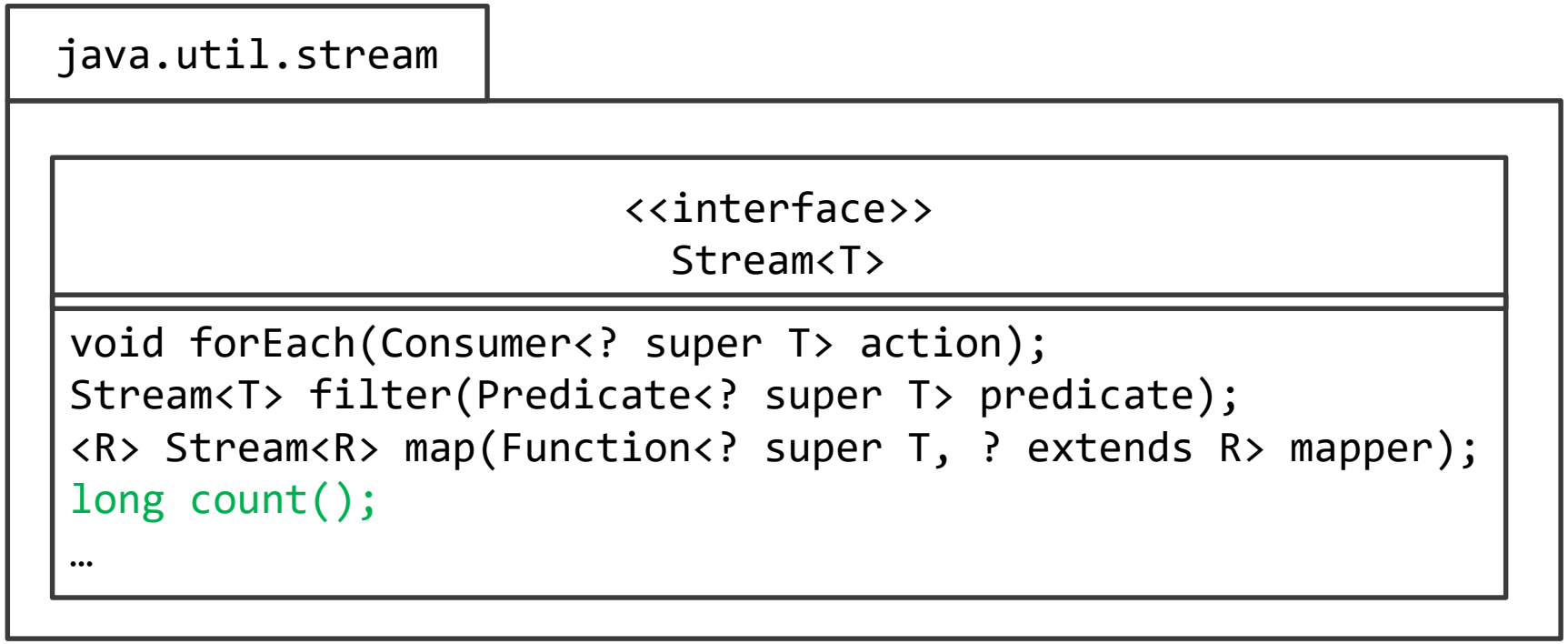
filter-Methode

Wir haben den Verdacht, dass sich ein „falscher“ Bruder bei den B-Brüdern eingeschmuggelt hat.

Burt Bronski Peter

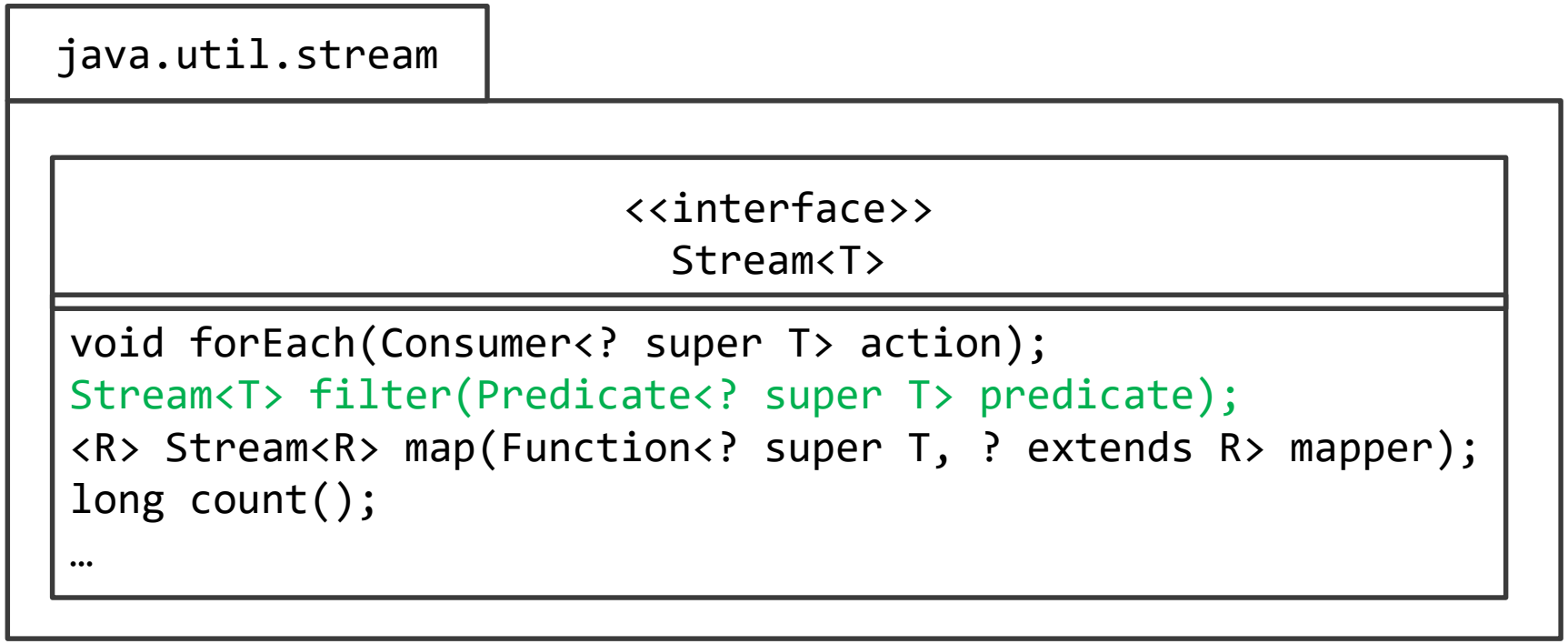
Wir wollen überprüfen, ob wirklich jeder B-Bruder mit einem B anfängt.

Die Methode count() liefert die Anzahl der Elemente im Stream zurück.



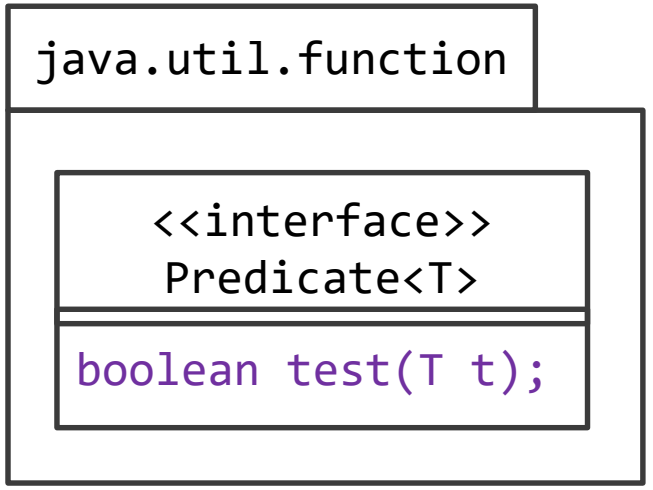

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

long totalNumberOfBBrothers = bBrothers.stream().count();
```

Die Methode filter gibt einen Stream zurück, der nur die Elemente enthält, die eine bestimmte **Bedingung** erfüllen.



Die Bedingung wird über ein Predicate festgelegt.

Genauer: Die test-Methode von Predicate legt diese **Bedingung** fest.



Wir wollen den Verdacht, dass sich ein „falscher“ Bruder bei den B-Brüdern eingeschmuggelt hat, überprüfen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

long numberOfRealBBrothers = bBrothers.stream()
 .filter(




  )
 .count();
```

Den Verdacht überprüfen wir mit Hilfe der filter-Methode.

Die Überprüfung findet durch ein Predicate statt.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

long numberOfRealBBrothers = bBrothers.stream()
 .filter(new Predicate<String>()
 {
   @Override
   public boolean test(String name)
   {
     return name.startsWith("B");
   }
 })
 .count();
```

Die filter-Methode gibt einen **neuen** Stream zurück, der nur die Elemente enthält, für die die test-Methode true zurückgegeben hat.

Dasgleiche mit einem Lambda-Ausdruck:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

long numberOfRealBBrothers = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .count();
```

Die filter-Methode ähnelt stark der „Für die gilt“-Einschränkung in mathematischen Mengen

$$M = \{n \in Q \mid n \geq 18\}$$

```
.filter(n -> n >= 18)
```

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

long totalNumberOfBBrothers = bBrothers.stream().count();

long numberOfRealBBrothers = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .count();

if (numberOfRealBBrothers < totalNumberOfBBrothers) System.out.println("Alarm!");
```

Ein falscher B-Bruder wurde erwischt.

Wiederverwendung von Lambda-Ausdrücken
DRY

Wir wollen die B-Brüder ausgeben, die mit P anfangen
und die, die mit B anfangen.

Wir erweitern vorheriges Programm:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
   .filter(name -> name.startsWith("B"))
   .forEach(name -> System.out.println(name));

bBrothers.stream()
   .filter(name -> name.startsWith("P"))
   .forEach(name -> System.out.println(name));
```

die markierten Stellen verstoßen gegen das
Don’t Repeat Yourself (DRY) Prinzip

Der Consumer der forEach-Methode ist in beiden Fällen dergleiche.

Wir können den Lambda-Ausdruck in einer Variablen speichern:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

Consumer<String> printOut = name -> System.out.println(name);
bBrothers.stream()
  .filter(name -> name.startsWith("P"))
  .forEach(printOut);

bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .forEach(printOut);
```

Wenn wir die Ausgabe ändern, können wir das jetzt an einer einzelnen Stelle tun.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

Consumer<String> printOut = name -> System.out.println(name);
bBrothers.stream()
  .filter(name -> name.startsWith("P"))
  .forEach(printOut);

bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .forEach(printOut);
```

die markierten Stellen
unterscheiden sich lediglich in
einem konstanten Wert (“P” vs “B”)

Das verstößt ebenfalls gegen das
DRY-Prinzip.

Wir ziehen die beiden Lambda-Ausdrücke nach vorne.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

Consumer<String> printOut = name -> System.out.println(name);
Predicate<String> pFilter = name -> name.startsWith("P");
Predicate<String> bFilter = name -> name.startsWith("B");

bBrothers.stream()
  .filter(pFilter)
  .forEach(printOut);

bBrothers.stream()
  .filter(bFilter)
  .forEach(printOut);
```

Das hat noch nicht viel geholfen. Der Code unterscheidet sich immer noch nur in einem konstanten Wert.

Wir benötigen eine Methode!

```
private static Predicate<String> letterFilter(String letter)
{
    return name -> name.startsWith(letter);
}

public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);
    Predicate<String> pFilter = letterFilter("P");
    Predicate<String> bFilter = letterFilter("B");

    bBrothers.stream()
      .filter(pFilter)
      .forEach(printOut);

    bBrothers.stream()
      .filter(bFilter)
      .forEach(printOut);
}
```

wir rufen die Methode letterFilter auf, die ein Predicate zurückgibt

```
private static Predicate<String> letterFilter(String letter)
{
    return name -> name.startsWith(letter);
}

public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);
    Predicate<String> pFilter = letterFilter("P");
    Predicate<String> bFilter = letterFilter("B");

    bBrothers.stream()
      .filter(pFilter)
      .forEach(printOut);

    bBrothers.stream()
      .filter(bFilter)
      .forEach(printOut);
}
```

letterFilter() gibt ein Predicate zurück, das die Variable letter verwendet

```
private static Predicate<String> letterFilter(String letter)
{
    return name -> name.startsWith(letter);
}

public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);
    Predicate<String> pFilter = letterFilter("P");
    Predicate<String> bFilter = letterFilter("B");

    bBrothers.stream()
      .filter(pFilter)
      .forEach(printOut);

    bBrothers.stream()
      .filter(bFilter)
      .forEach(printOut);
}
```

letter ist nach Ablauf der Methode nicht mehr gültig.

Wie kommt es, dass wir den Lambda Ausdruck dennoch später verwenden können?

Zum besseren Verständnis verwenden wir statt des Lambda-Ausdrucks eine anonyme Klasse.

```
private static Predicate<String> letterFilter(String letter)
{
    return new Predicate<String>() {
      @Override
      public boolean test(String name)
      {
        return name.startsWith(letter);
      }
    };
}
```

Gleiche Situation:

letter ist nach Ablauf der Methode nicht mehr gültig.

Wie kommt es, dass wir das Objekt der anonymen Klasse dann später verwenden können?

Der Compiler macht aus der anonymen Klasse sinngemäß ungefähr folgenden Code:

```
private static Predicate<String> letterFilter(String letter)
{
    return new Predicate<String>(){
      final String _letter = letter;

      @Override
      public boolean test(String name)
      {
        return name.startsWith(_letter);
      }
    };
}
```

Die anonyme Klasse erhält für jede verwendete lokale Variable je ein (final) Attribut zum Zwischenspeichern.

Diese Konstante wird dann im Objekt der anonymen Klasse bzw. im Lambda Ausdruck verwendet.

Der Compiler macht aus der anonymen Klasse sinngemäß ungefähr folgenden Code:

```
private static Predicate<String> letterFilter(String letter)
{
    return new Predicate<String>(){
      final String _letter = letter;

      @Override
      public boolean test(String name)
      {
        return name.startsWith(_letter);
      }
    };
}
```

Ein Lambda-Ausdruck kann daher nur auf lokale Variablen angewendet werden, die **final** oder **effectively final** sind.

Zurück zur Lösung mit dem Lambda-Ausdruck:

```
private static Predicate<String> letterFilter(String letter)
{
    return name -> name.startsWith(letter);
}

public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);

    bBrothers.stream()
     .filter(letterFilter("P"))
     .forEach(printOut);

    bBrothers.stream()
     .filter(letterFilter("B"))
     .forEach(printOut);
}
```

Zur Parametrisierung von Lambda-Ausdrücken haben wir eine eigene Methode geschrieben.

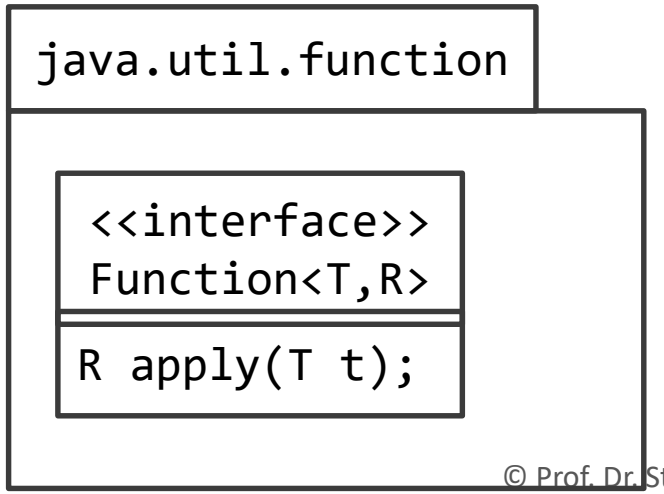
Es ist sogar möglich den Scope der statischen Methode so einzuschränken, dass sie nur für die Lambda-Ausdrücke sichtbar ist.

```
private static Predicate<String> letterFilter(String letter)
{
    return name -> name.startsWith(letter);
}
```

```
public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);

    bBrothers.stream()
     .filter(letterFilter("P"))
     .forEach(printOut);

    bBrothers.stream()
     .filter(letterFilter("B"))
     .forEach(printOut);
}
```

Wir haben eine Funktion definiert, die ein Predicate zurückgibt.

Das können wir auch über das Function-Interface.



Wir benötigen eine Function, die einen String (den Startbuchstaben) auf ein Predicate abbildet.

```
Function<String, Predicate<String>> letterFilter = …
```

Wir benötigen eine Function, die einen String (den Startbuchstaben) auf ein Predicate abbildet.

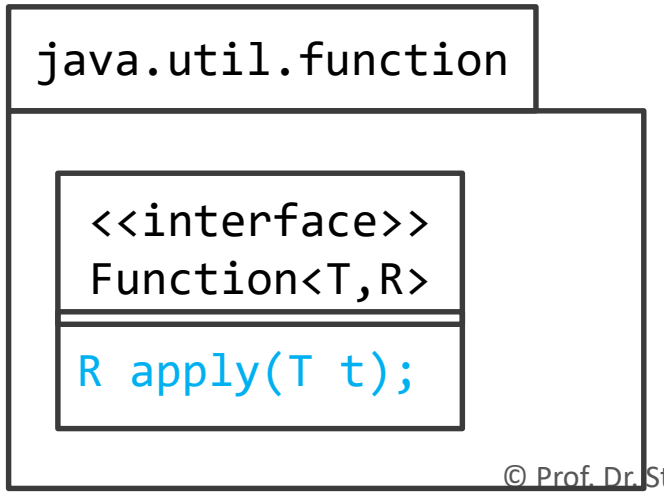
```
Function<String, Predicate<String>> letterFilter = letter -> …
```

Wir benötigen eine Function, die einen String (den Startbuchstaben) auf ein Predicate abbildet.

```
Function<String, Predicate<String>> letterFilter = letter -> (name -> name.startsWith(letter));
```

Wir benötigen eine Function, die einen String (den Startbuchstaben) auf ein Predicate abbildet.

```
public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);

    Function<String, Predicate<String>> letterFilter = letter -> (name -> name.startsWith(letter));

    bBrothers.stream()
     .filter(letterFilter.apply("P"))
     .forEach(printOut);

    bBrothers.stream()
     .filter(letterFilter.apply("B"))
     .forEach(printOut);
}
```



Wir benötigen eine Function, die einen String (den Startbuchstaben) auf ein Predicate abbildet.

```
public static void main(String[] args)
{
    List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");
    Consumer<String> printOut = name -> System.out.println(name);

    Function<String, Predicate<String>> letterFilter = letter -> (name -> name.startsWith(letter));

    bBrothers.stream()
     .filter(letterFilter.apply("P"))
     .forEach(printOut);

    bBrothers.stream()
     .filter(letterFilter.apply("B"))
     .forEach(printOut);
}
```

Änderungen an der Function können jetzt zentral an einer Stelle erfolgen, anstatt an mehrerem Stellen wie zuvor.

Collectors I

Was kennen wir bisher?

Transformationen haben wir mit **map** durchgeführt.

Einschränkungen der Elemente mit **filter**.

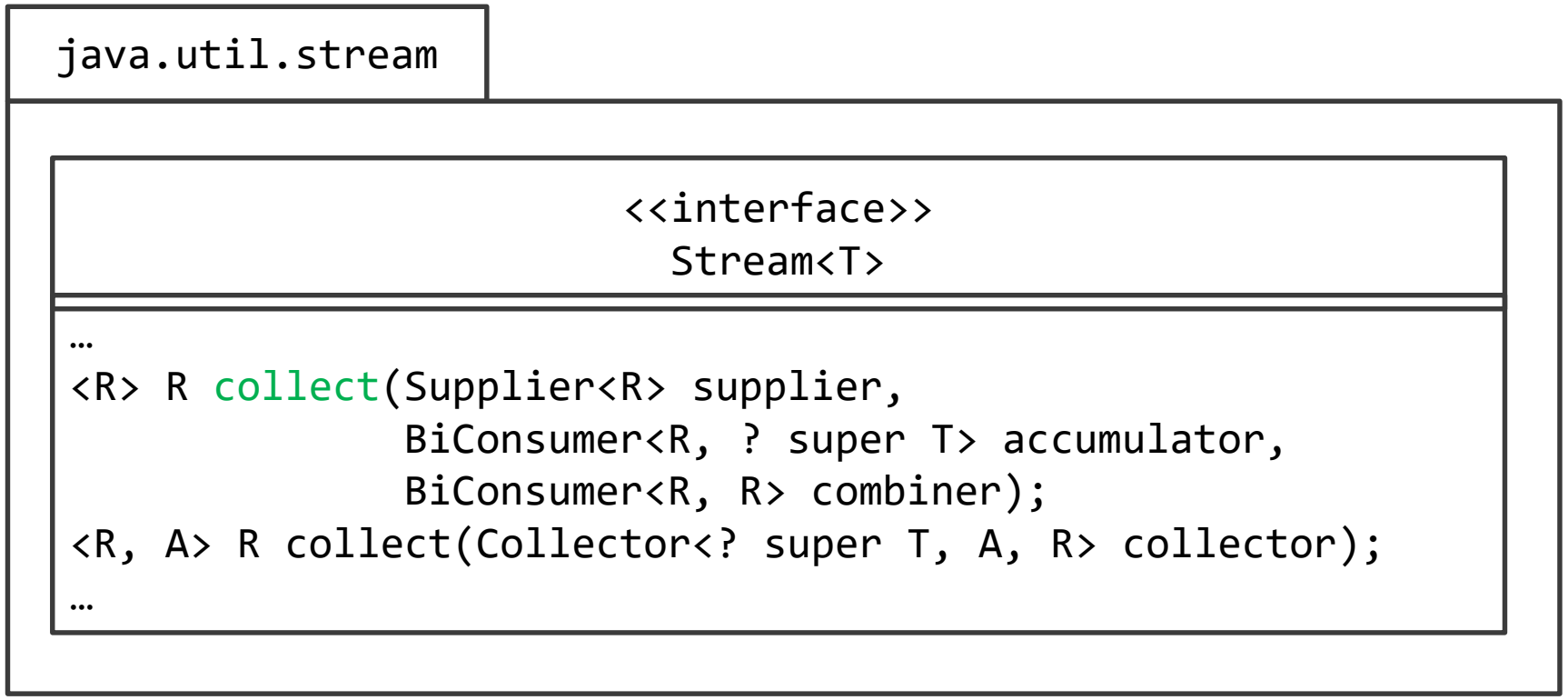
Häufig ist es notwendig,
die Ergebnisse in einer neuen Collection zu speichern.

Der Verdacht hat sich bestätigt.

Der B-Bruder Peter muss rausgeschmissen werden.

Burt Bronski Peter

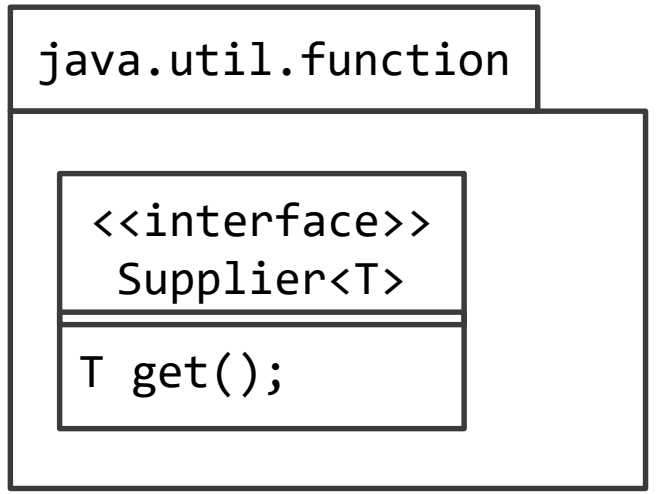
Nach Anwendung des Filters wollen wir aus den übrig gebliebenen B-Brüdern eine neue Collection erstellen.



Die collect-Methode benötigt drei Parameter:

- Einen Supplier, der einen Container (z.B. Liste) anlegt, in den sie speichern kann.
- Einen Accumulator, der beschreibt, wie Elemente im Container gesammelt werden.
- Einen Combiner, der beschreibt wie bei paralleler Verarbeitung Teilcontainer zusammengelegt werden.

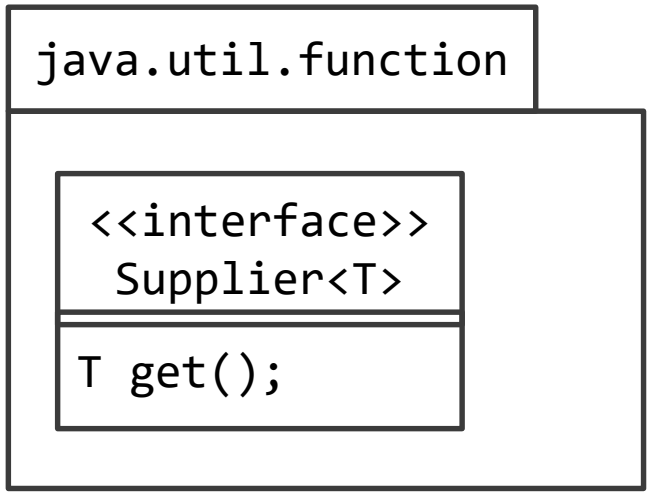
Supplier ist ein FunctionalInterface, das eine parameterlose Methode zur Verfügung stellt, die ein Objekt vom Typ T zur Verfügung stellt (ähnlich einer Factory).



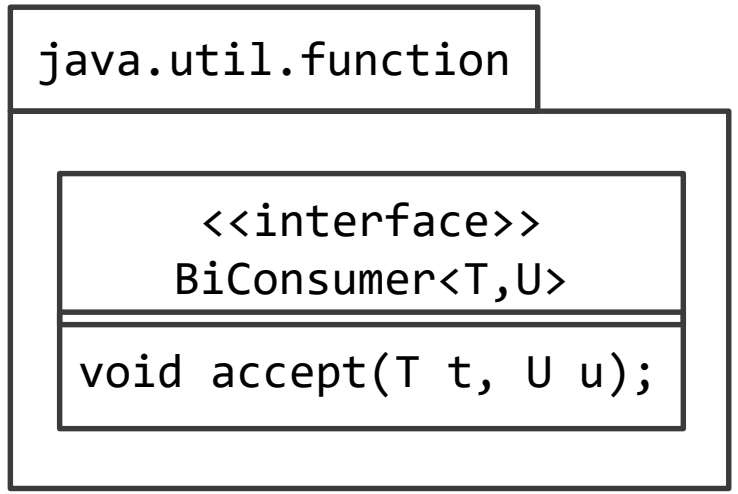
Nach Anwendung des Filters wollen wir aus den übrig gebliebenen B-Brüdern eine neue Collection erstellen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(
    new Supplier<ArrayList<String>>()
    {
       public ArrayList<String> get()
       {
         return new ArrayList<>();
       };
    },
    new BiConsumer<ArrayList<String>, String>()
    {
      public void accept(ArrayList<String> l, String name)
      {
        l.add(name);
      }
    },
    new BiConsumer<ArrayList<String>, ArrayList<String>>()
    {
      public void accept(ArrayList<String> l1, ArrayList<String> l2)
      {
        l1.addAll(l2);
      }
    });
```

Supplier



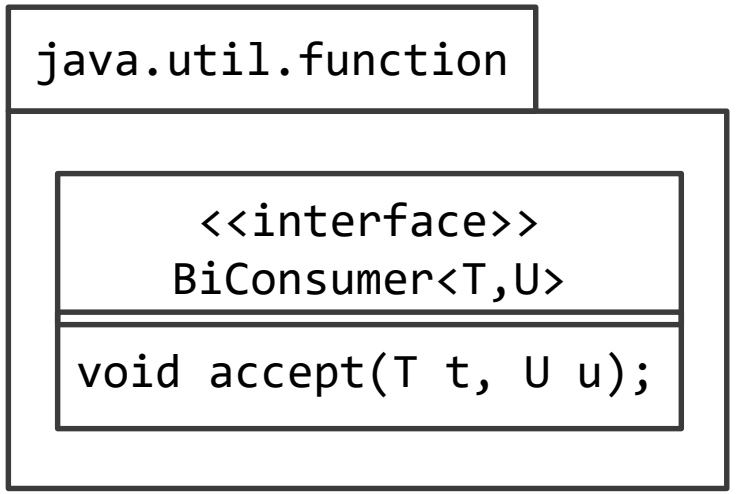
Ein BiConsumer ist ein Consumer mit zwei statt einem Parameter.



Nach Anwendung des Filters wollen wir aus den übrig gebliebenen B-Brüdern eine neue Collection erstellen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(
    new Supplier<ArrayList<String>>()
    {
       public ArrayList<String> get()
       {
         return new ArrayList<>();
       };
    },
    new BiConsumer<ArrayList<String>, String>()
    {
      public void accept(ArrayList<String> l, String name)
      {
        l.add(name);
      }
    },
    new BiConsumer<ArrayList<String>, ArrayList<String>>()
    {
      public void accept(ArrayList<String> l1, ArrayList<String> l2)
      {
        l1.addAll(l2);
      }
    });
```

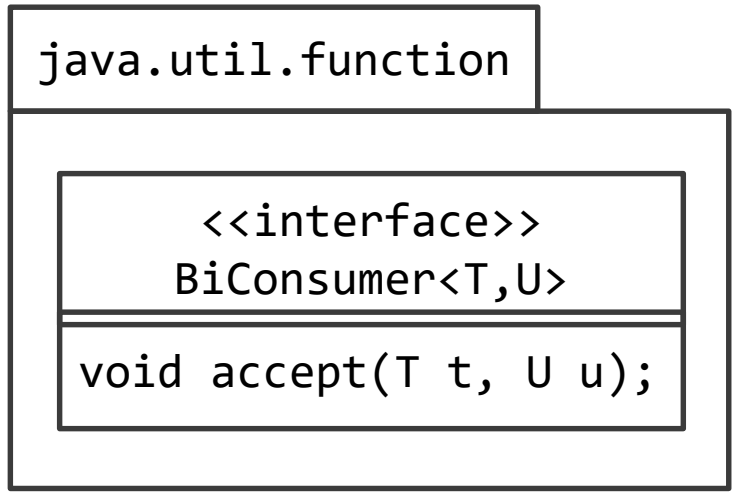
Accumulator



Nach Anwendung des Filters wollen wir aus den übrig gebliebenen B-Brüdern eine neue Collection erstellen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(
    new Supplier<ArrayList<String>>()
    {
       public ArrayList<String> get()
       {
         return new ArrayList<>();
       };
    },
    new BiConsumer<ArrayList<String>, String>()
    {
      public void accept(ArrayList<String> l, String name)
      {
        l.add(name);
      }
    },
    new BiConsumer<ArrayList<String>, ArrayList<String>>()
    {
      public void accept(ArrayList<String> l1, ArrayList<String> l2)
      {
        l1.addAll(l2);
      }
    });
```

Combiner



Nach Anwendung des Filters wollen wir aus den übrig gebliebenen B-Brüdern eine neue Collection erstellen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(
    new Supplier<ArrayList<String>>()
    {
       public ArrayList<String> get()
       {
         return new ArrayList<>();
       };
    },
    new BiConsumer<ArrayList<String>, String>()
    {
      public void accept(ArrayList<String> l, String name)
      {
        l.add(name);
      }
    },
    new BiConsumer<ArrayList<String>, ArrayList<String>>()
    {
      public void accept(ArrayList<String> l1, ArrayList<String> l2)
      {
        l1.addAll(l2);
      }
    });
```

java.util.stream

<<interface>>
Stream<T>

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

Kürzer über Lambda-Ausdrücke

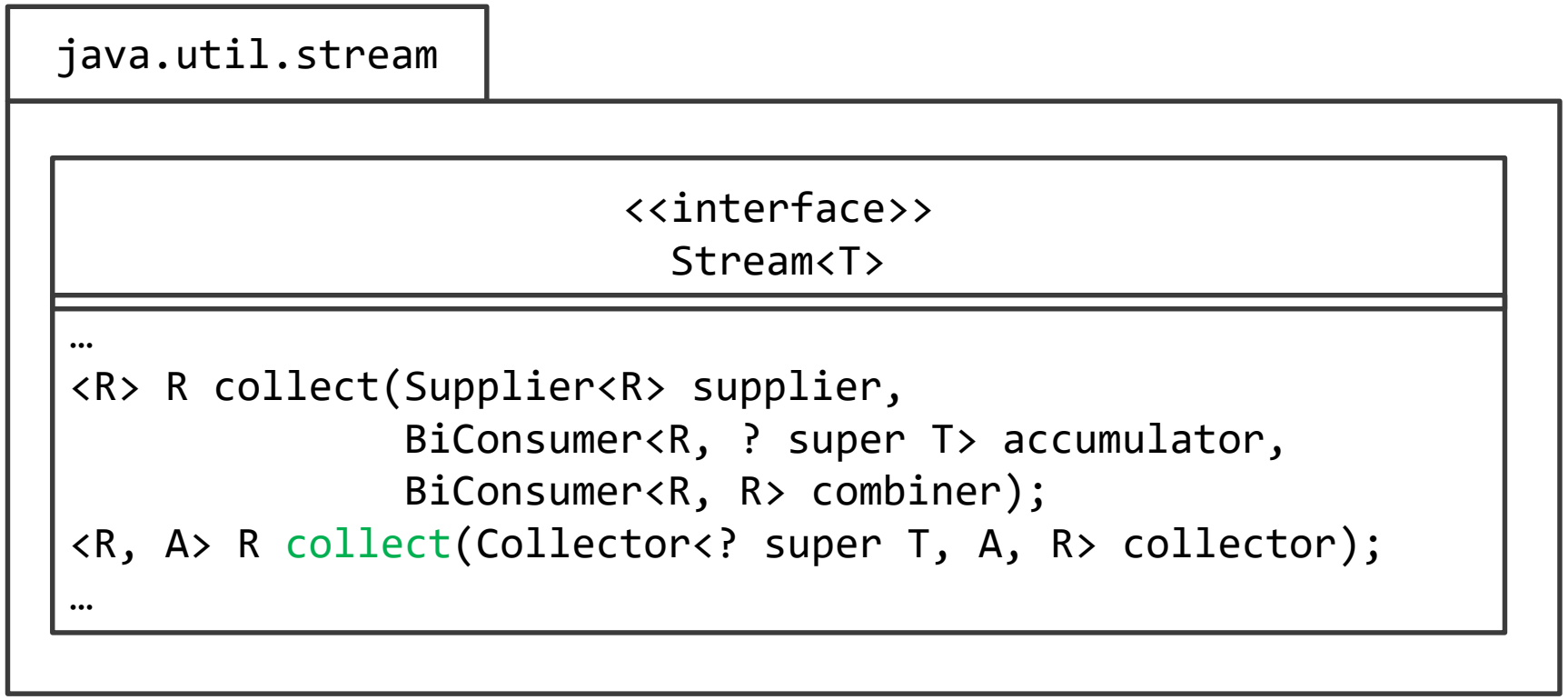
```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(() -> new ArrayList<>(), (l, name) -> l.add(name), (l1, l2) -> l1.addAll(l2));
System.out.println(list);
```

Supplier, der angibt, wie der Container (die Liste) zu Beginn erstellt wird.

Accumulator, der angibt, wie die Liste befüllt wird (Elemente werden der Liste hinzugefügt).

Combiner, der angibt, wie Teillisten zusammengefügt werden.

Alternativ können wir auch einen vorgefertigten Collector nutzen.



Alternativ können wir auch einen vorgefertigten Collector nutzen.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(Collectors.toList());
System.out.println(list);
```

Durch einen static import können wir noch das Collectors weglassen.

```
import static java.util.stream.Collectors.*;
…

List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(toList());
System.out.println(list);
```

Es gibt eine Reihe von vorgefertigten Collectors, die eine Transformation in eine Reihe von Klassen zulassen:

- toList()
- toSet()
- toMap(…)
- toCollection(…)
- …

Falls wir das Ergebnis nur zusammenführen wollen, können wir mehrere Strings zu einem einzelnen String zusammenfassen.

```
import static java.util.stream.Collectors.*;
…

List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter", "Bruno");

String namensliste = bBrothers.stream()
  .filter(name -> name.startsWith("B"))
  .collect(Collectors.joining(", "));
System.out.println(namensliste);
```

Ausgabe:

Burt, Bronski, Bruno

Literaturempfehlung