Streams

Streams

- Applying functions to lists of objects is so popular, that Java 8 has some special classes for this: Stream, BasicStream, IntStream,...
    - A stream is a sequence of objects

- Like our FList, there are many useful operations defined for streams, like map, filter,...
- Example:

```
Stream<Integer> stream=Stream.of(1,2,3,4,5);

Stream<Integer> mappedStream=stream.map( (i)->i+1 );
```

- Remember that streams are a concept from functional programming. In general, stream operations don’t/shouldn’t have side effects.

Creating streams

```java
class Account {
    private int value;
    public Account(int value) { this.value=value; }
    public int getValue() { return value; }
}


// Create streams from values
Stream<Integer> stream1=Stream.of(1,2,3,4,5);
Stream<Account> stream2=Stream.of(new Account(100), new Account(200));


// Create a stream from an array.
// For base types (like int and double), Java has optimized stream
// implementations. In general, it is more efficient to use a DoubleStream
// instead of Stream<Double>
double[] a=new double[]{ 1.0, 2.0, 3.0 };
DoubleStream stream3=Arrays.stream(a);


// Create a stream from a list
LinkedList<Integer> list=new LinkedList<>();
list.add(3); list.add(4);
Stream<Integer> stream4=list.stream();
```

Working with streams

```
Stream<Integer> stream1=Stream.of(1,2,3,4,5);

// apply function on each element
Stream<Integer> mappedStream=stream1.map( (i)->i+1 );

// filter elements
Stream<Integer> filteredStream=mappedStream.filter( (i)->i<4 );

// sort stream
Stream<Integer> sortedStream=filteredStream.sorted();

// transform object stream into IntStream
Stream<Account> stream2=Stream.of(new Account(100), new Account(200));
IntStream intStream=stream2.mapToInt( (a)->a.getValue() );
```

- Check the documentation of the Stream class and the special versions (IntStream, DoubleStream,...) to see what operations exist! There are many useful ones

Getting the data out of a stream

- When working with streams, usually the last operation is an operation that returns some result that is not a stream
- Examples:
  - Counting all elements greater than 5:

```
int n=stream.filter( (i)->i>5 ).count();
```

  - Transforming the stream into an array:

```
Stream<Account> stream=Stream.of(new Account(100), new Account(200));
int[] values=stream.mapToInt( (a)->a.getValue() ).toArray();
```

  - Doing something with each element:

```
Stream<String> stream = Stream.of("Hello", "World");
stream.forEach( (s)-> System.out.println(s) );
```

This is like a map-operation, but it does not have a result.

Reduce

- Often, we want to do an operation with all elements of a stream and calculate a single result value. For example, we want to calculate the sum of all elements in a stream with integers:
- For this, we can use the reduce method:

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);
int result = stream.reduce(0, (a,b)->a+b);
```

- Reduce needs two arguments:
  - A start value s for the reduction operation (Here: $s = 0$)
  - A function f that is applied on the elements $e_1, e_2, \dots, e_n$ as follows:

$$r_1 = f(s, e_1)$$
$$r_2 = f(r_1, e_2)$$
$$r_3 = f(r_2, e_3)$$
$$\dots$$
$$result = f(r_{n-1}, e_n)$$

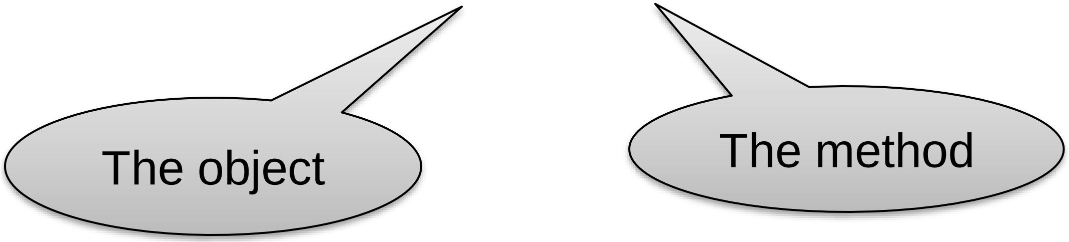
A shorter way to use methods in lambda expressions

- The line

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

can be written shorter as:

```
stream.forEach( System.out::println );
```



The object

The method

- This also works with methods with more than one parameter. The line

```
BiFunction<String,Integer,PrintStream> f = (s,i)->System.out.format(s,i);
```

can be written as:

```
BiFunction<String,Integer, PrintStream> f = System.out::format;
```

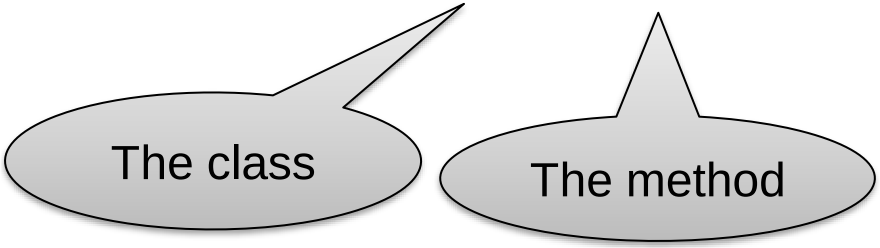
A shorter way to use methods in lambda expressions (2)

- The notation with “::” can be also used without an object. The line

```
Stream<Integer> stream= Stream.of("Hello", "World").map( (s)-> s.length() );
```

can be written as:

```
Stream<Integer> stream= Stream.of("Hello", "World").map( String::length );
```



The class

The method

- This even works with constructors! This line

```
Stream<Account> stream = Stream.of(1,2,3,4,5).map( (i)-> new Account(i) );
```

can be written as:

```
Stream<Account> stream = Stream.of(1,2,3,4,5).map( Account::new );
```

A shorter way to use methods in lambda expressions (3)

- The notation with “::” also works with static methods. Example:
- The Integer class defines a static method “sum” that calculates the sum of two integers

```
public static int sum(int a, int b) {
    return a + b;
}
```

- Instead of

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);
int n = stream.reduce(0, (a,b)->a+b);
```

we can write:

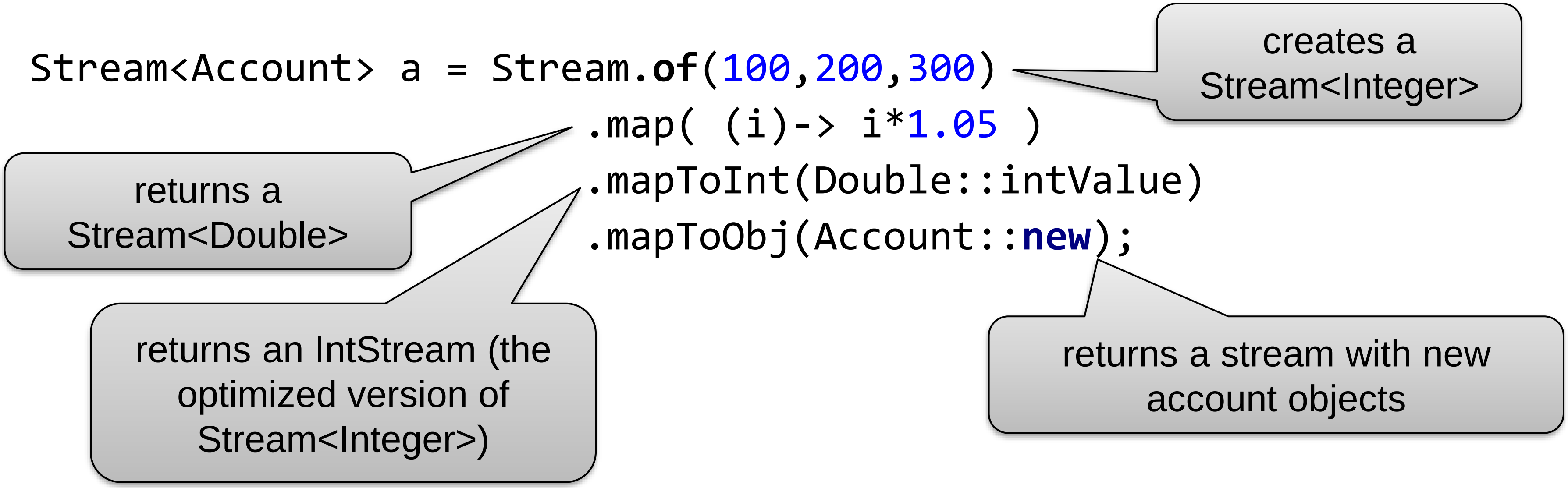
```
Stream<Integer> stream = Stream.of(1,2,3,4,5);
int n = stream.reduce(0, Integer::sum);
```

The class

The static method

A complete example

- Our task: Take a list of amounts, add 5%, and create accounts for them



```
Stream<Account> a = Stream.of(100,200,300)
                           .map( (i)-> i*1.05 )
                           .mapToInt(Double::intValue)
                           .mapToObj(Account::new);
```

creates a Stream<Integer>

returns a Stream<Double>

returns an IntStream (the optimized version of Stream<Integer>)

returns a stream with new account objects

- Note:
  - The map method of Stream returns a Stream. Use mapToInt to return an IntStream.
  - The map method of IntStream returns an IntStream. Use mapToObj to return a Stream.

Streams are lazy!

- What will this code print?

```
Stream<Integer> s1 = Stream.of(1,2,3,4,5);
Stream<Integer> s2 = s1.map( (i)-> { System.out.println(i); return i+1; } );
```

- You could think that the code does the following:
  1. A stream with elements 1,2,3,4,5 is created
  2. The function `(i)->{ System.out.println(i); return i+1; }` is applied on each element
  3. A new stream with 2,3,4,5,6 is returned
- However, this is wrong! The above code does not print anything. Streams are lazy. The operations are only executed if the result is needed, for example to calculate a result:

```
Stream<Integer> s1 = Stream.of(1,2,3,4,5);
Stream<Integer> s2 = s1.map( (i)-> { System.out.println(i); return i+1; } );
Object[] a = s2.toArray();  // <- here, the elements of the stream are needed
```

Streams are lazy! (2)

- What does this code print?

```
Stream<Integer> s1 = Stream.of(1,2,3,4,5);
Stream<Integer> s2 = s1.map( (i)-> { System.out.println(i); return i+1; } );
s2.forEach(System.out::println);
```

- It prints

```
1          // System.out.println for i=1
2          // System.out.println in forEach
2          // System.out.println for i=2
3          // System.out.println in forEach
…
```

because streams work like this:

1. forEach needs the first element of s2. To obtain the first element, map is executed with the first element of the stream s1
2. forEach needs the second element of s2. To obtain the second element, map is executed with the second element of the stream s1
3. …

Streams are lazy (3)

- You should be now able to say what this code prints:

```
Stream<Integer> s1 = Stream.of(1,2,3,4,5);
Stream<Integer> s2 = s1.map( (i)-> { System.out.println(i); return i+1; } );
int n=s2.findFirst().get();  // get first element of stream s2
```

- Answer: It only prints “1”. Only the first element of the result is needed. The variable n will have the value 2 at the end.

Streams are lazy (4)

- Because streams are lazy, they can be also used in situations where it is not known in advance how long the stream is.
- Example: print all lines of a text file in upper case:

returns a Stream<String> where each element is a line from the file

```
Stream<String> stream = Files.lines(Paths.get("somefile.txt"));
stream.map(String::toUpperCase).forEach(System.out::println);
```

- What this code does **not** do:
  - Read the entire file, then print it.
    That would use a lot of memory if the file is very big!
- What this code does do:
  - Read the first line, print it, read the second line, print it,...

Only once

- Note that you can only go once through the same stream. Once an element has been processed, it is not possible anymore to access the element again.
- This will not work:

```
Stream<Integer> stream = Stream.of(1,2,3,4,5);

int n = stream.reduce(0, (a,b)->a+b);

Stream<Integer> stream2 = stream.map( (i)-> i+1 );
```

Error: "stream has already been operated upon or closed"