Functional Programming

8 Lazy Evaluation and Streams

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

- By-name parameters
- lazy vals
- Case example: Streams in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Non-strict evaluation

Strict evaluation means call-by-value:

- expressions for parameters in methods calls are evaluated
- the value is passed in the method call

Non-strict evaluation means call-by-name

- expressions for parameters in method calls are not evaluated
- but are passed as unevaluated expressions

With non-strict evaluation expressions can be evaluated

- in a specific context
- lazily by-need, i.e., when the value is really needed

Non-strict evaluation can be achieved

- by-name parameters (Scala)
- by function parameters
- by lazy vals

By-name Parameters

Parameters with call-by-name semantics

```
code : => T
```

- actual argument **expressions** are passed **unevaluated**
- but **parameter** in method body is **replaced** by **expression**

```
def method[T](param : => T) = {
   .... context ...
       param
   .... context ...
}
```

```
method(expression)
```

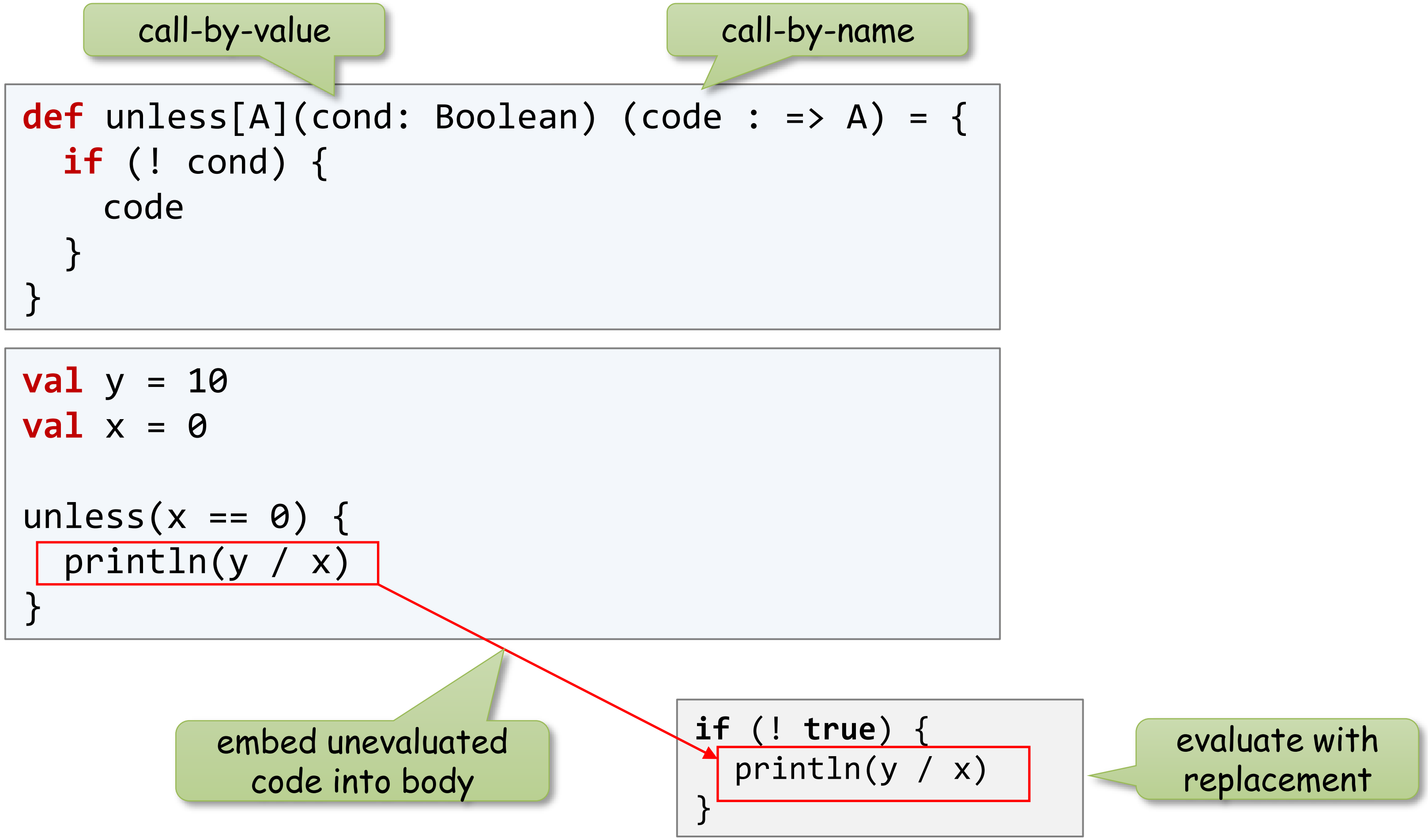
```
{
   .... context ...
       expression
   .... context ...
}
```

Note:

- by-name parameters are NO function objects (*not first-class objects*)!
- only parameter passing is call-by-name

Example Unless

Method unless for conditional evaluation of expression



call-by-value

call-by-name

```
def unless[A](cond: Boolean) (code : => A) = {
  if (! cond) {
    code
  }
}
```

```
val y = 10
val x = 0

unless(x == 0) {
  println(y / x)
}
```

embed unevaluated code into body

```
if (! true) {
  println(y / x)
}
```

evaluate with replacement

Example from Scala Library: break

- **break** for loops not supported in Scala
- provided by library object **Breaks** with methods **breakable** and **break**

Application

```
import scala.util.control.Breaks._
var sum = 0
breakable {
  for (i <- 0 to 1000) {
    sum += i
    if (sum >= 1000) break
  }
}
```

Implementation

```
object Breaks {

 private val breakException = new BreakControl

 def break : Nothing = {
   throw breakException
 }

 def breakable(body: => Unit) {
   try {
     body
   } catch {
     case ex: BreakControl => {...}
 }
```

throws execption for breaking

by-name parameter

catch exception in breakable context

Example from Scala Library: synchronized

- Application

```
  val lock = new Object();
  lock.synchronized {

    … code synchronized on lock …

  }
```

- Implementation as method of class AnyRef
  - □ with by-name code parameter

```
class AnyRef {

  def synchronized(syncCodeBlock: => Unit) {
    ...
  }
  ...
}
```

Non-strict Evaluation with Function Objects

Function objects evaluated by-need

Example: unless in Java

- with Runnable function object for action

```
public static void unless(boolean cond, Runnable action) {
    if (cond) {
        action.run();
    }
}
```

```
unless( x == 0,
        () -> System.out.println(y / x)
);
```

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

- By-name parameters
- lazy vals
- Case example: Streams in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Lazy Vals

lazy keyword for **val** variables means delaying initialization until variable is first accessed

initialized when first accessed

```
lazy val lazyX = initialization-expression
```

- Applications
  - □ delaying executing initialization code until value is really needed
  - □ avoiding executing initialization code when value is never needed

Lazy Vals

lazy keyword for **val** variables means delaying initialization until variable is first accessed

```
lazy val lazyX = initializeLazyX()
```

```
private def initializeX() = {
  println("Initializing lazyX")
  "-- value of lazyX --"
}
```

- Demo

```
println("Before declaration of lazyX")
lazy val lazyX = initializeLazyX()
println("After declaration of lazyX")

println("Before printing lazyX")
println(lazyX)
println("After printing lazyX")

println("Before printing lazyX as second time")
println(lazyX)
println("Before printing lazyX as second time")
```

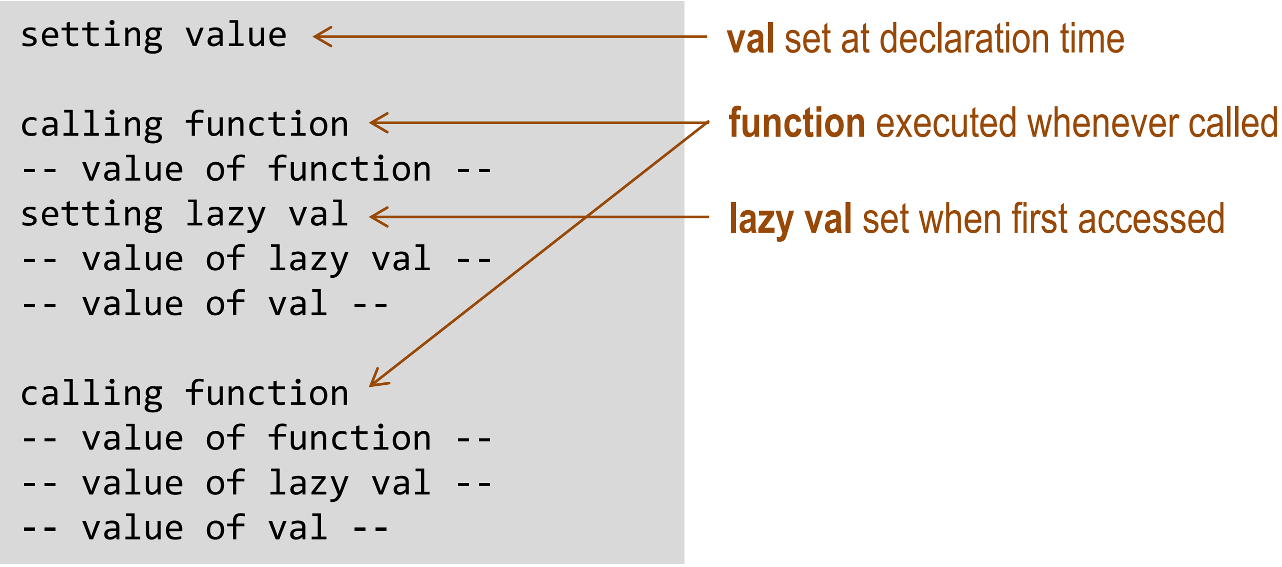
```
Before declaration of lazyX
After declaration of lazyX
Before printing lazyX
Initializing lazyX
-- value of lazyX --
After printing lazyX
Before printing lazyX as second time
-- value of lazyX --
Before printing lazyX as second time
```

not initialized when declared

initialized at first printout

Comparison val, lazy val and Function

```
val value = {
  println("setting value")
  "-- value of val --"
}

lazy val lazyVal = {
  println("setting lazy val")
  "-- value of lazy val --"
}

def function = {
  println("calling function")
  "-- value of function --"
}

println()

println(function)
println(lazyVal)
println(value)

println()

println(function)
println(lazyVal)
println(value)
println()
```



```
setting value

calling function
-- value of function --
setting lazy val
-- value of lazy val --
-- value of val --

calling function
-- value of function --
-- value of lazy val --
-- value of val --
```

val set at declaration time

function executed whenever called

lazy val set when first accessed

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

- By-name parameters
- lazy vals
- Case example: Streams in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Streams in Scala

compare Java Streams

- Streams are lazy sequences of elements
  - □ uses *thunks* to represent generation of elements

```scala
val largeColl = List(10, 20, 7, 33, -15, 6, 9, 2, ...)

val stream = largeColl.toStream

val tlStream = stream.tail
```

not evaluated = thunk!

```
stream   : Stream[Int] = Stream(10, ?)

tlStream : Stream[Int] = Stream(20, ?)
```

  - □ elements generated only by need (= on access)

```scala
val someElems = stream.drop(2).take(3).filter(_ > 0).toList
```

```
someElems  : List[Int] = List(7, 33, 6)
```

Stream in Scala

Implementation simplified version of library type **Stream**

Stream implementation equivalent to List but

- with function objects for head and tail
- which are evaluated lazily on access

```
sealed trait Stream[+A] :
  val isEmpty : Boolean
  def size : Int
  ...

case object Empty extends Stream[Nothing] :
  override val isEmpty = true
  override def size = 0

case class Cons[+A](val hdFn: () => A, val tlFn: () => Stream[A]) extends Stream[A] :
  override val isEmpty = false
  override def size = 1 + tlFn().size
```

hdFn and **tlFn** are function objects for creating head element and tail list

forces (evaluates) **tlFn** function object for accessing tail

more in Exercise 9

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

What are Streams?

“A sequence of elements supporting sequential and parallel aggregate operations”

- access to a sequence of elements from a data source
- fluid API for processing data elements with higher-order functions
- sequential and parallel processing
- aggregate operations with internal iteration
- lazy processing
- Monads with *map* – *filter* – *reduce* pattern

A First Stream Example

- Functional programming
  - □ aggregate operations
  - □ *map* – *filter* – *reduce* pattern

```
List<Article> articles = List.of(new Article("iPhone", 1500), new Article("Galaxy S8", 700), …);

List<String> cheapArticleNames =

    articles.stream()

      .filter(a -> a.price < 1000.0)

      .sorted(Comparator.comparingDouble((Article a) -> a.price))

      .map(a -> a.name)

      .collect (Collectors.toList());
```

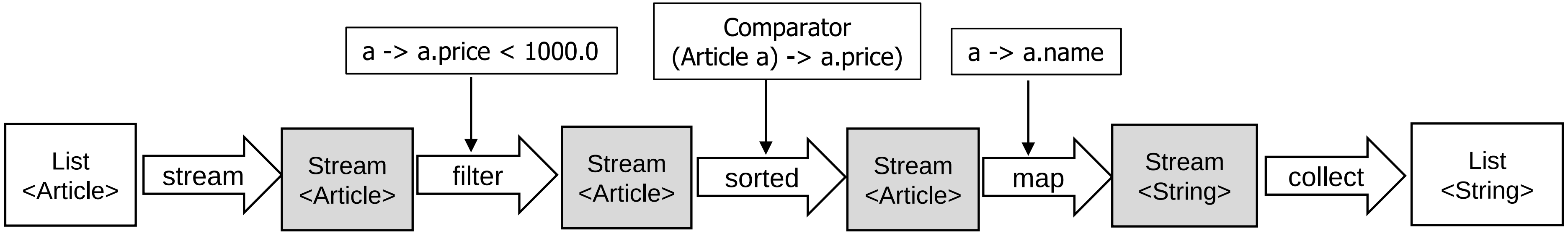
create stream from source

filter low calorie dishes

sort elements

map to names

collect in result list



Collections vs. Streams

Streams

- provider of data elements
- accessing elements only
- one single iteration only
- internal iteration
- lazy computation

Collections

- storage for data elements
- adding and removing elements
- allows iterating multiple times
- external iteration
- eager computation

External vs. Internal Iteration

External Iteration

- □ Iteration done in user code

Internal Iteration

– Iteration provided by stream

Example: Collect names of articles

```
List<String> names = new ArrayList<>();
for (Article a : articles) {
    names.add(a.name);
}
```

external iteration by for-loop

```
List<String> names =
   articles.stream()
           .map(a -> a.name)
           .collect(Collectors.toList());
```

internal iteration within stream

Example: Add filter operation to collect names of articles with price < 1000.0

```
List<String> highPriceNames = new ArrayList<>();
for (Article a : articles) {
    if (a.price < 1000.0) {
       highPriceNames.add(a.name);
    }
}
```

Must edit loop body for extension
➔ imperative style (how)

```
List<String> highPriceNames =
   articles.stream()
           .filter(a -> a.price < 1000.0)
           .map(a -> a.name)
           .collect(Collectors.toList());
```

Add additional processing step
➔ functional style (what)

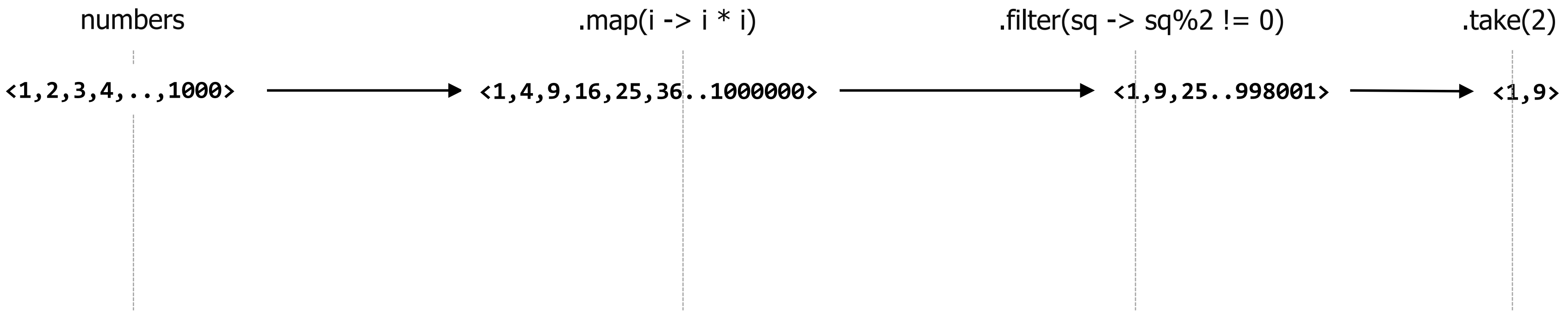
Eager Processing

- Eager computation with **List** (from vavr)
  - process whole list in **push** mode
  - create intermediate results

Example: Given a list of integers, get first two squares which are odd

```
List<Integer> twoOddSquares =
          numbers.map(i -> i * i)
                 .filter(sq -> sq % 2 != 0)
                 .take(2);
```

eager aggregate operations on lists



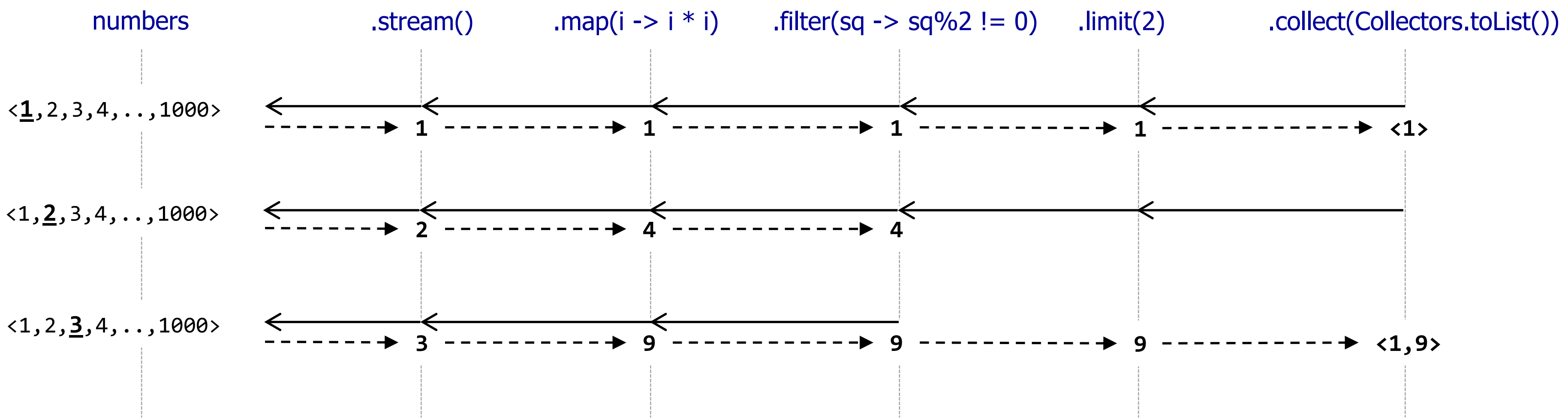
iterates all elements, creates intermediate result lists

LAZY PROCESSING

- process elements one by one in **pull** mode
- retrieve elements as needed

Example: Given a list of integers, get first two squares which are odd

```
List<Integer> twoOddSquares =
      numbers.stream()
            .map(i -> i * i)
            .filter(sq -> sq % 2 != 0)
            .limit(2)
            .collect(Collectors.toList());
```



NO superfluous computations, NO intermediate result lists needed

Types of Streams

- Generic Streams

```java
public interface Stream<T>
```

- Streams for built-in types **int**, **long**, **double**

```java
public interface IntStream
```

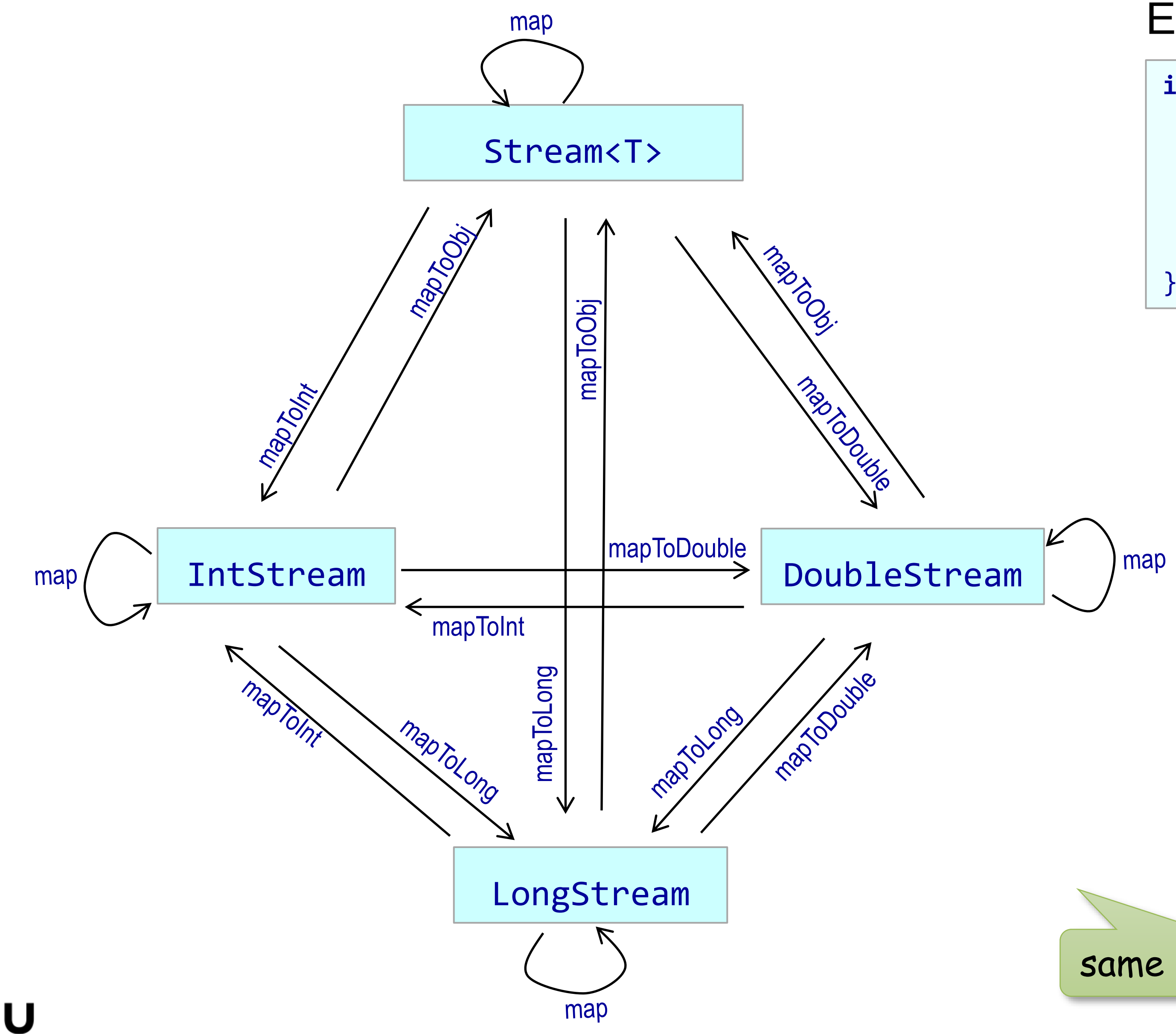
```java
public interface LongStream
```

```java
public interface DoubleStream
```

Introduced for efficiency reasons!

Note: no streams for char, float and boolean

Mapping between Different Steam Types



Example Interface **IntStream**

```
interface IntStream {
  <U> Stream<U> mapToObj(IntFunction<? extends U> m);
  LongStream mapToLong(IntToLongFunction m);
  DoubleStream mapToDouble(IntToDoubleFunction m);
  IntStream map(IntUnaryOperator m);
  …
}
```

```
@FunctionalInterface
public interface IntFunction<R> {
  R apply(int value);
}
```

```
@FunctionalInterface
public interface IntToDoubleFunction {
  double applyAsDouble(int value);
}
```

```
@FunctionalInterface
public interface IntToLongFunction {
  long applyAsLong(int value);
}
```

same for flatMap

Map – Filter – Reduce Pattern

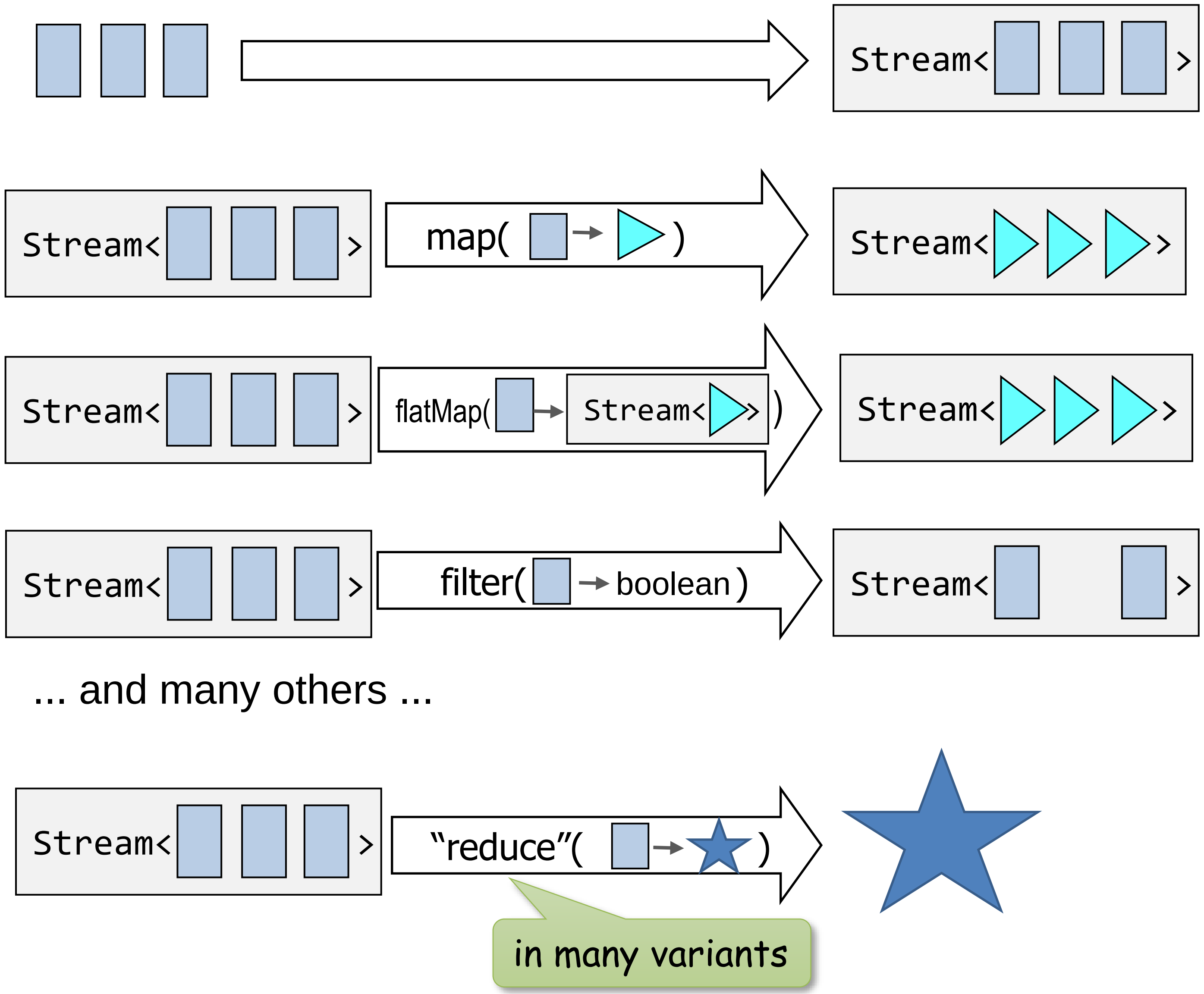
Building a Stream

Intermediate Operations

- Mapping
- Filtering
- etc.

Terminal Operations

- Materialize stream



Stream< >

Stream< > map( ) Stream< >

Stream< > flatMap(Stream< >) Stream< >

Stream< > filter(boolean) Stream< >

... and many others ...

Stream< > "reduce"( )

in many variants

SUMMARY OF STREAM OPERATIONS

Building streams (Source → Stream)

From collections and arrays	`collection.stream(), Stream.of(T... values), Arrays.stream(T[] array)`
Generator methods	`Stream.generate, Stream.iterate, IntStream.range, IntStream.rangeClosed, …`
Library methods	`Files.lines, Files.walk, BufferedReader.lines, CharSequence.chars, …`

Intermediate operations (Stream → Stream)

Mapping	`map, mapToInt, mapToDouble, mapToLong`
	`flatMap, flatMapToInt, flatMapToDouble, flatMapToLong`

Filtering	`filter`
Sorting	`sorted(), sorted(Comparator<? super T> comparator)`
Subsets	`limit, skip, distinct, takeWhile, dropWhile`

Terminal operations (Stream → Result)

Iteration	`forEach, forEachOrdered`
Reduction	`reduce`
Special reduction operations	`count, sum, min, max, average`
Collecting elements	`collect`
Finding elements	`findFirst, findAny`
Boolean quantifiers	`allMatch, anyMatch, noneMatch`

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Building Streams: Overview

From collections

```
public interface Collection<E> extends Iterable<E>
  default Stream<E> stream() {…}
```

```
List<String> words = new ArrayList<String>();
Stream<String> wordStream = words.stream();
```

From arrays

```
public final class Arrays
  public static <T> Stream<T> stream(T[] array)
  public static IntStream stream(int[] array)
```

```
String[] words = new String[] { "Functional", "Programming", … };
Stream<String> wordStream = Arrays.stream(words);
```

```
int[] numbers = new int[] { 1, 2, 3, 5, 7, 11, 13 };
IntStream numberStream = Arrays.stream(numbers);
```

From files and sockets

```
public final class Files
  static Stream<String> lines(Path path)
  static Stream<Path> list(Path dir)
```

```
Stream<String> lines = Files.lines(Paths.get("data.txt"));
…
lines.close();
```

```
public class BufferedReader extends Reader
  public Stream<String> lines()
```

```
BufferedReader reader =
  new BufferedReader(new InputStreamReader(socket.getInputStream());
Stream<String> lines = reader.lines();
```

Character stream from string

```
public final class String
  public IntStream chars()
```

```
String text = "This is a text";
IntStream charactersInText = text.chars();
```

Building Streams: Overview

Creating streams

```
public interface Stream<T>
  static<T> Stream<T> of(T... values)
  static <T> Stream<T> concat(Stream<? extends T> a,
                                   Stream<? extends T> b)
  static<T> Stream<T> empty()
```

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Stream<String> single = Stream.of("Me");
Stream<String> all = Stream.concat(single, names);
Stream<String> none = Stream.empty();
```

Generating streams

- iterate and generate

```
public interface Stream<T>
  static<T> Stream<T> iterate(final T seed,
                              final UnaryOperator<T> f)
  static<T> Stream<T> generate(Supplier<? extends T> s)
```

```
Stream<Integer> intsFrom0 = Stream.iterate(0, i -> i + 1);
```

```
public interface IntStream
  static IntStream generate(IntSupplier s)
```

```
final Random r = new Random();
IntStream randStream = IntStream.generate(() -> r.nextInt(100));
```

- Random number streams

```
public class Random
  public IntStream ints()
  public IntStream ints(int origin, int bound)
  public DoubleStream doubles(long streamSize)
  …
```

```
IntStream infiniteInts  = rand.ints();
IntStream infiniteInts0_9 = rand.ints(0, 10);
DoubleStream infiniteDoubles0_100 = rand.doubles(0.0, 100.0);
```

Stream from Collections and Arrays

- Stream from collections

```
List<String> words = new ArrayList<String>();
words.add("Java 8"); ...

Stream<String> wordStream = words.stream();
```

```
public interface Collection<E> extends Iterable<E> {
  ...
  default Stream<E> stream() {…}
}
```

stream() is a default method of **Collection**

- Stream from arrays

```
String[] words = new String[] { "Functional", "Programming", "in", "Java" };
Stream<String> wordStream = Arrays.stream(words);
```

Static method of class **Arrays**

```
int[] numbers = new int[] { 1, 2, 3, 5, 7, 11, 13 };
IntStream numberStream = Arrays.stream(numbers);
```

IntStream from **int**-array!

- Stream by enumerating values

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Stream<String> single = Stream.of("Me");
Stream<String> all = Stream.concat(single, names);
Stream<String> none = Stream.empty();
```

form a Monoid

Stream Operations for Files and Folders

- Class **Files** with static methods for creating streams from accessing files and traversing folders

```
public final class Files {
  static Stream<String> lines(Path path)
  static Stream<Path> list(Path dir)
  static Stream<Path> find(Path start, int maxDepth,
                      BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options)
  static Stream<Path> walk(Path start, FileVisitOption... options)
  static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options)
  …
}
```

- Stream of lines of a file
- Stream with files and folders
- Stream of files with certain properties
- Stream of files and folders in subdirectory

Example: Stream of lines in a text file

```
Stream<String> lines = Files.lines(Paths.get("data.txt"), Charset.defaultCharset());
…
lines.close();
```

must be closed as external resource

Stream from Socket

- Generate Stream of lines from BufferedReader of sockets

```
public class BufferedReader extends Reader
  public Stream<String> lines()
```

```
BufferedReader reader =
      new BufferedReader(
            new InputStreamReader(
                  socket.getInputStream());

Stream<String> lines = reader.lines();
```

Stream of Characters

- Stream of characters from Strings (or CharSequence)
  - □ creates IntStream

```
String text = "This is a text";

IntStream charactersInText = text.chars();
```

IntStream !

Generators

- Stream.generate
  - Uses a Supplier<T> to create a stream

```
static <T> Stream<T> generate(Supplier<T> s)
```

```
final Random r = new Random();
IntStream randStream = IntStream.generate(() -> r.nextInt(100));
```

infinite stream of random numbers

- **Stream.iterate**
  - Starting at a seed value
  - Computes the next value based on the previous value

```
static <T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

```
Stream<Point> randomWalk = Stream.iterate(
                                new Point(100, 100),
                                p -> new Point(p.x + r.nextInt(DIST), p.y + r.nextInt(DIST))
                        );
```

infinite stream of random points

Ranges

- Creating IntStreams of values within range and rangeClosed
  - □ ints within range of 0 to 100 (exclusive)

```
IntStream range0_99 = IntStream.range(0, 100);
```

  - □ ints within range of 0 to 100 (inclusive)

```
IntStream range0_100 = IntStream.rangeClosed(0, 100);
```

- **Same for LongStream**

Random number Streams

Random number Streams from java.until.Random

```
Random rand = new Random();
```

- random IntStream

```
IntStream infiniteInts          = rand.ints();
IntStream infiniteInts0_9       = rand.ints(0, 10);
IntStream twentyInts            = rand.ints(20L);
IntStream twentyInts0_9         = rand.ints(20L, 0, 10);
```

infinite streams

streams with 20 elements

- random DoubleStream

```
DoubleStream infiniteDoubles         = rand.doubles();
DoubleStream infiniteDoubles0_100    = rand.doubles(0.0, 100.0);
DoubleStream twentyDoubles           = rand.doubles(20L);
DoubleStream twentyDoubles0_100      = rand.doubles(20L, 0.0, 100.0);
```

- same for LongStream

SplittableRandom allows parallel generation of randoms

```
SplittableRandom splitableRandom = new SplittableRandom(7);
IntStream infiniteInts0_9 = splitableRandom.ints(0, 10);
```

cf. Section on parallel streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Intermediate Operations: Overview

```
public interface Stream<T>
```

Mapping

```
<R> Stream<R>  map(Function<? super T,? extends R> mapper)
```

```
IntStream      mapToInt(ToIntFunction<? super T> mapper)
LongStream     mapToLong(ToLongFunction<? super T> mapper)
DoubleStream   mapToDouble(ToDoubleFunction<? super T> mapper)
```

```
<R> Stream<R>  flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

Filtering

```
Stream<T>      filter(Predicate<? super T> predicate)
```

Substreams

```
Stream<T>      limit(long maxSize)
```

```
Stream<T>      skip(long n)
```

```
Stream<T>      takeWhile(Predicate<? super T> predicate)
```

```
Stream<T>      dropWhile(Predicate<? super T> predicate)
```

Intermediate Operations: Overview

```
public interface Stream<T>
```

Removing duplicates

```
Stream<T>      distinct()
```

Sorting

```
Stream<T>      sorted()
Stream<T>      sorted(Comparator<? super T> comparator)
```

Performing action:

- performs action and forwards elements unchanged

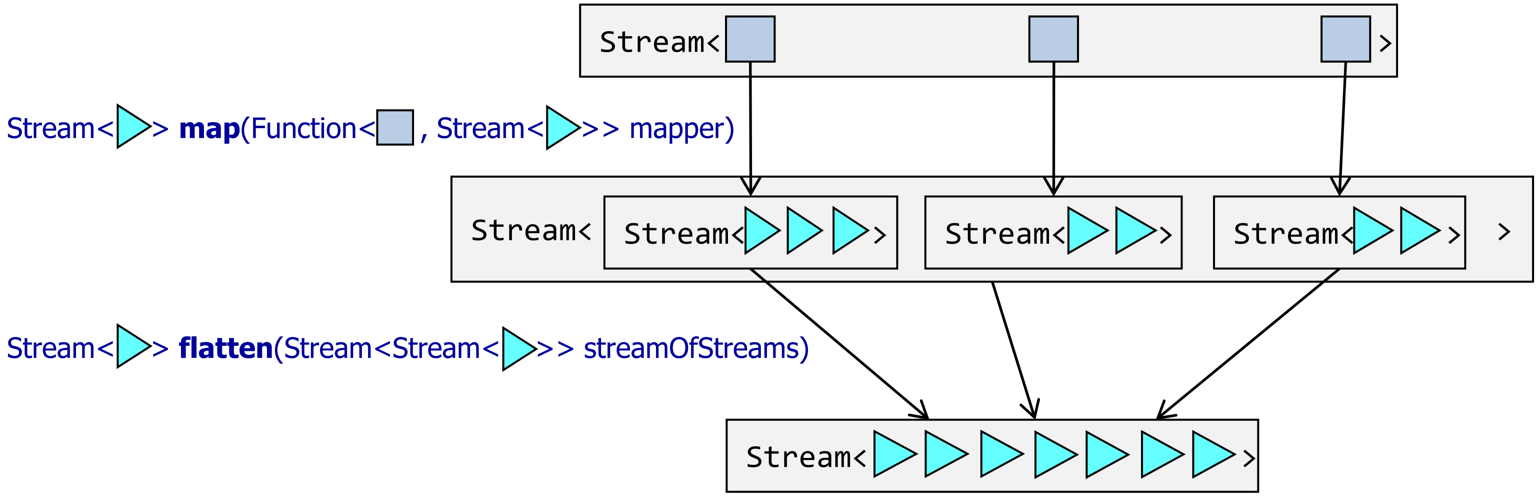
```
Stream<T>      peek(Consumer<? super T> action)
```

FlatMap

- **flatMap**: map each element to stream and then flatten all streams

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

function T -> Stream<R>



Stream<▷> **map**(Function<□, Stream<▷>> mapper)

Stream<▷> **flatten**(Stream<Stream<▷>> streamOfStreams)

Mapping

- flatMap: map each element to stream and then flatten all streams

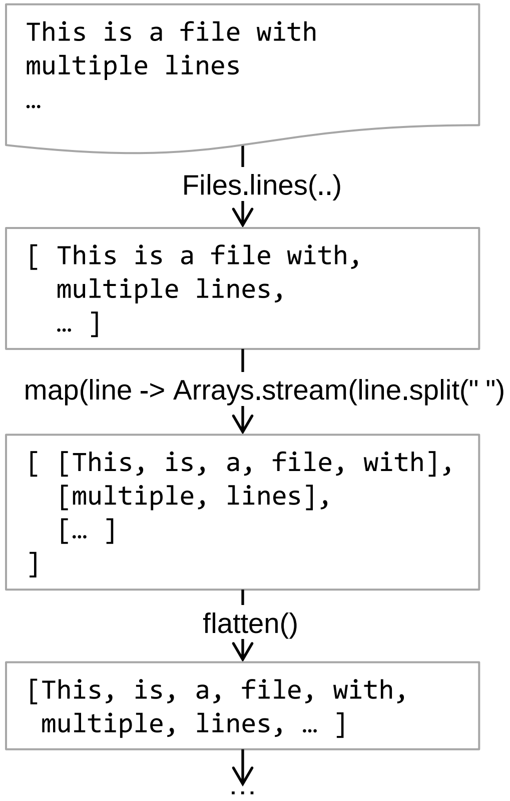
```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

function T -> Stream<R>

Example: List of words in a text

- □ start with stream of lines
- □ then each line mapped to stream of words
- □ then flatten the streams for each word to a single stream of words
- □ collect in list

```
Stream<String> lines = Files.lines(Paths.get("faust.txt"), Charset.defaultCharset());

List<String> words =
  lines
    .flatMap(line -> Arrays.stream(line.split(" ")))
    .collect(Collectors.toList());
```



```
This is a file with
multiple lines
…
```

Files.lines(..)

```
[ This is a file with,
  multiple lines,
  … ]
```

map(line -> Arrays.stream(line.split(" ")

```
[ [This, is, a, file, with],
  [multiple, lines],
  [… ]
]
```

flatten()

```
[This, is, a, file, with,
 multiple, lines, … ]
```

…

Mapping

- map

```
<R> Stream<R>  map(Function<? super T,? extends R> mapper)
```

Example:

```
Stream<String> names = Stream.of("Ann", "Pat", "Mary", "Joe");
Stream<String> initials = names.map(a -> a.substring(0, 1));
```

```
A,P,M,J
```

- mapToInt, mapToLong, mapToDouble

```
IntStream     mapToInt(ToIntFunction<? super T> mapper)
LongStream    mapToLong(ToLongFunction<? super T> mapper)
DoubleStream  mapToDouble(ToDoubleFunction<? super T> mapper)
```

Example:

```
IntStream length = names.mapToInt(a -> a.length());
```

```
3,3,4,3
```

FILTERING

- filter

```
Stream<T> filter(Predicate<? super T> predicate)
```

Example:

```
Stream<String> lines = Files.lines(Paths.get("faust.txt"), Charset.defaultCharset());

List<String> words =
  lines
    .filter(line -> line.length() > 0)                           // remove empty lines
    .flatMap(line -> Arrays.stream(line.split(" ")))
    .filter(word -> Character.isLetter(word.charAt(0)))          // remove non-words
    .collect(Collectors.toList());
```

LIMIT, SKIP, TAKEWHILE, DROPWHILE

- **limit**: cut of stream after maxSize elements

```
Stream<T> limit(long maxSize)
```

Example:

```
IntStream numbers =  IntStream.of(10,9,8,7,6,5,4,3,2,1);
numbers.limit(5);
```

```
10,9,8,7,6
```

- **skip**: skip first n elements

```
Stream<T> skip(long n)
```

Example:

```
numbers.skip(5);
```

```
5,4,3,2,1
```

- **takeWhile / dropWhile**: take / skip elements as long predicate fulfilled

```
Stream<T> takeWhile(Predicate<? super T> predicate)
```

```
Stream<T> dropWhile(Predicate<? super T> predicate)
```

Example:

```
numbers.takeWhile(n -> n >= 5);
```

```
10,9,8,7,6,5
```

SORTED AND DISTINCT

stateful methods !

- **sorted**: sort elements

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

Example:

```
Stream<String> namesSorted = names.sorted();
```

```
"Ann", "Joe", "Mary", "Pat"
```

```
Stream<String> namesSorted = names.sorted(Comparator.naturalOrder().reverse());
```

```
"Pat", "Mary", "Joe", "Ann",
```

- **distinct**: remove duplicates (using equals)

```
Stream<T> distinct()
```

Example:

```
IntStream numbers =  IntStream.of(10, 9, 8, 7, 6, 6, 7, 8, 9, 10);
numbers.distinct();
```

```
10,9,8,7,6
```

Peek

- **peek**: allows side effect and forwards stream unchanged

```
Stream<T> peek(Consumer<? super T> action)
```

Example:

```
List<String> wordStream =
  lines
    .flatMap(line -> Arrays.stream(line.split(" ")))
    .peek(word -> {
      count(word);
      System.out.println(word);
    })
    …
```

```
void count(String word) {
  ...
}
```

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Terminal Operations

Terminal operations compute results from streams

- they initiate and control retrieval and processing of elements
- are eager

Terminal operations follow various processing patterns

- **Iteration**
  - □ perform action on elements
- **Finding elements**
  - □ terminate as soon as first element found
- **Reduction**
  - □ process elements and combine elements to single results
- **Collecting**
  - □ collect elements in **mutable** result container

Special reduce which is more efficient in Java!

ITERATION

- **forEach**: iterate over all elements and apply a **Consumer<T>**

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

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

```
Ann
Joe
Mary
Pat
```

- **forEachOrdered**: same as **forEach** but order is enforced also in parallel execution

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

➔ useful only for parallel streams, see next section on parallel streams

FIND

findFirst

```
Optional<T> findFirst();
```

- □ returns first element in a stream in Optional
- □ with Optional.emtpy if no element in stream
- □ usually used in the combination with filter

Note streams are processed lazy
➔ elements only filtered until first element found

Example: Find first element in stream which is even

```
Stream<Integer> numbers =  Stream.of(7, 3, 8, 7, 6, 6, 7, 8, 9, 10);

Optional<Integer> evenNumber = numbers.filter(x -> x % 2 == 0).findFirst();
```

only three elements processed until 8 found

findAny

```
Optional<T> findAny();
```

- □ finds any element in stream

➔ useful for parallel streams, see next section on parallel streams

Special Find Operations

- Boolean quantifiers
  - □ **anyMatch**: find element for which predicate is fullfilled

```java
boolean existsZero = numbers.anyMatch(x -> x == 0);
```

  - □ **allMatch**: find element for which predicate is ***not*** fullfilled

if one found, return false, otherwise true

```java
boolean allPositive = numbers.allMatch(x -> x > 0);
```

  - □ **noneMatch**: allMatch with negated predicate

```java
boolean noNegative = numbers.noneMatch(x -> x < 0);
```

Reduction

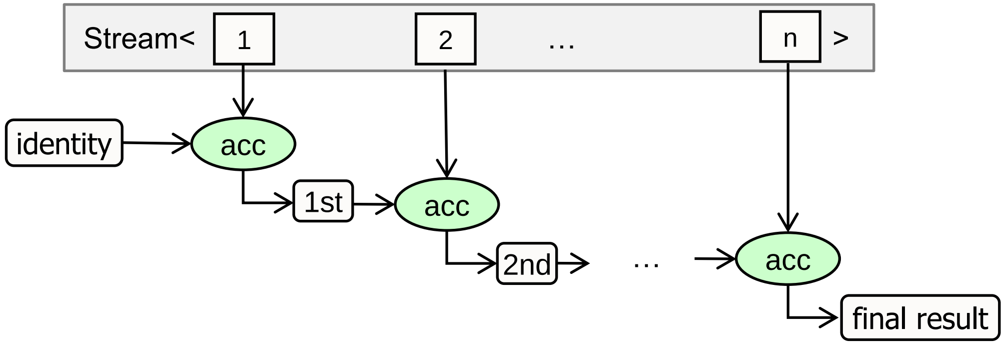
reduce

- Reduces elements of type T to result of type U

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

combiner for parallel execution only

☐ intial value of type **U**
☐ accumulator function **U x T → U**
☐ combiner function **U x U → U**



REDUCTION

reduce

- Reduces elements of type T to result of type U

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

Beispiel: Sum of **length** of strings of **Stream<String>**

```
Stream<String> nameStream = Stream.of("Franz", "Fritz", "Berta", "Xaver");

int totalLength = nameStream.reduce( 0,
                                     (sum, n) -> sum + n.length(),
                                     (sum1, sum2) -> sum1 + sum2 );
```

identity

accumulator

combiner

reduce: Comparison with Monoids and reduceMap

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

- **identity** is identity value of Monoid
- **accumulator** combines map and operation of Monoid
- **combiner** is operation of Monoid

Example: totalLength

Stream

```
nameStream.reduce(
   0,
   (sum, n) -> sum + n.length(),
   (sum1, sum2) -> sum1 + sum2  );
```

identity value of plus monoid

mapping

plus operation

Reducible

```
reducible.reduceMap(n -> n.length(), intPlusMonoid );
```

mapping

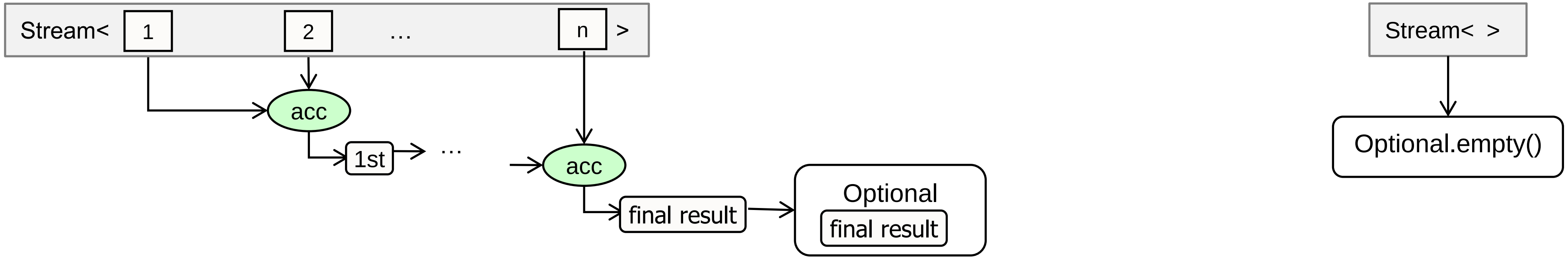
Monoid with identity and plus operation

Reduction

Variant of reduce with BinaryOperator

- first value in Stream as starting value
- **BinaryOperator** as acckumulator and combiner function
- **Optional** as result with **Optional.empty()** if stream is empty

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```



Example: Greatest number in stream of integers

```
Optional<Integer> maxOpt =
    numbers.reduce((currentMax, x) -> (x > currentMax) ? x : currentMax);
```

first element is first value for **currentMax**

Special Reduction Operations

Special forms of reduce!

- **count**: Count elements in stream

```
long count()
```

```
long nUniqueWords =  Files.lines(Paths.get("faust.txt"), Charset.defaultCharset())
                       .flatMap(line -> Arrays.stream(line.split(" ")))
                       .distinct()
                       .count();
```

- **max, min**

```
Optional<T> max/min(Comparator<? super T> comparator)
```

```
Optional<String> longestWord = words.max((w1, w2) -> w1.length() – w2.length());
```

Special Reductions for Number Streams

- **sum of values**

```
int lengthOfWords =  Files.lines(Paths.get("faust.txt"), Charset.defaultCharset())
                       .flatMap(line -> Arrays.stream(line.split(" ")))
                       .mapToInt(w -> w.length())
                       .sum();
```

- **average of values**

```
OptionalDouble optAverageLength =
     Files.lines(Paths.get("faust.txt"), Charset.defaultCharset())
                        .flatMap(line -> Arrays.stream(line.split(" ")))
                        .mapToInt(w -> w.length())
                        .average();
```

- **statistics over of values with**
  - **average, min, max, count and sum**

```
IntSummaryStatistics lengthStatistics =
       Files.lines(Paths.get("faust.txt"), Charset.defaultCharset())
                       .flatMap(line -> Arrays.stream(line.split(" ")))
                       .mapToInt(w -> w.length())
                       .summaryStatistics();
```

same for LongStream and DoubleStream

Collect

- Add elements of type **T** to some mutable result container of Type **R**

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

combiner used in parallel execution

- □ supplier function to provide an initial container of type **R**
- □ use accumulator function **R x T → Void** to add element to the container
- □ use combiner function **R x R → Void** which adds all the elements of second container to the first container

Example: Collect all elements of stream in an ArrayList

```
List<String> wordList =
    words.collect(
                () -> new ArrayList<String>(),
                (list, w) -> list.add(w),
                (list1, list2) -> list1.addAll(list2)
              );
```

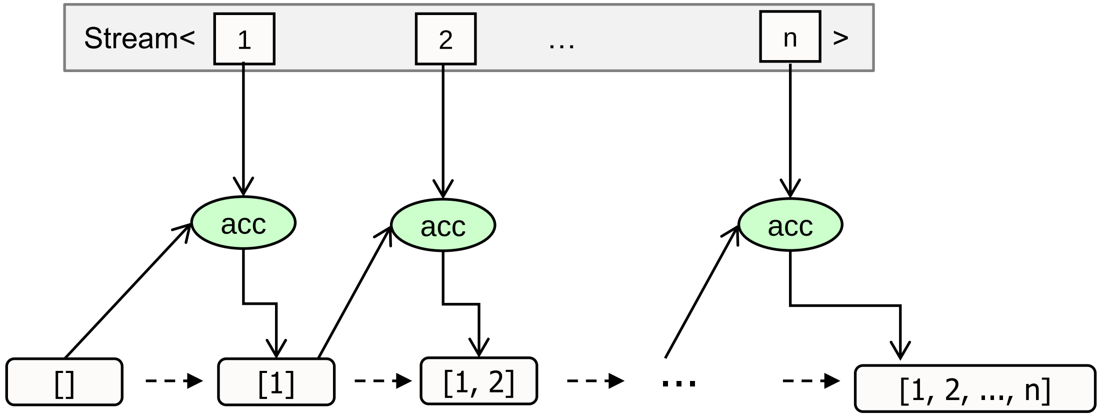
Supplier<ArrayList<String>>

BiConsumer<ArrayList<String>, String>

BiConsumer<ArrayList<String>, ArrayList<String>>

Collect

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



creating initial container by supplier

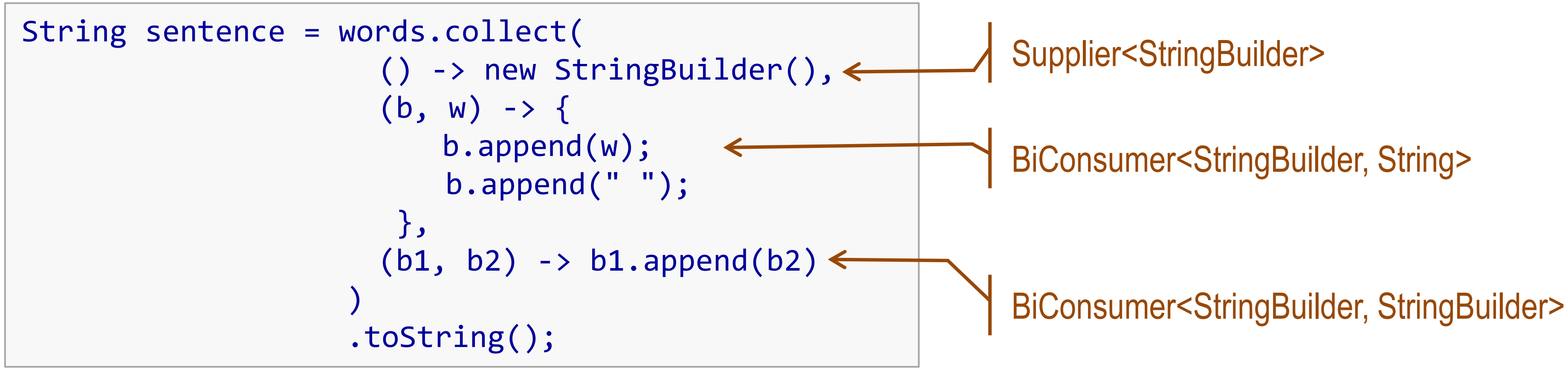
adding to container

returning filled container

Collect: Examples

- **collect** not restricted to collections

Example: Append all words in mutable StringBuilder



```
String sentence = words.collect(
                     () -> new StringBuilder(),
                     (b, w) -> {
                         b.append(w);
                         b.append(" ");
                      },
                     (b1, b2) -> b1.append(b2)
                   )
                   .toString();
```

Supplier<StringBuilder>

BiConsumer<StringBuilder, String>

BiConsumer<StringBuilder, StringBuilder>

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Collector

- **collect** with **Collector**

```
<R, A> R collect(Collector<? super T, A, R> collector)
```

- □ with collector is object combining elements **supplier**, **accumulator** and **combiner**
- □ plus additional **finisher** function

```
public interface Collector<T, A, R> {
  Supplier<A> supplier();
  BiConsumer<A, T> accumulator();
  BinaryOperator<A> combiner();
  Function<A, R> finisher();
}
```

T … type of elements
A … type of container
R … type of result of collect

Example: same collect as previous example

```
List<String> wordList
    = words.collect( Collector.of(
                      () -> new ArrayList<String>(),
                      (list, w) -> list.add(w),
                      (list1, list2) -> {
                          list1.addAll(list2);
                          return list1;
                        }
                    ));
```

BinaryOperator<ArrayList<String>>

Collector Example

```
public interface Collector<T, A, R> {
  Supplier<A> supplier();
  BiConsumer<A, T> accumulator();
  BinaryOperator<A> combiner();
  Function<A, R> finisher();
}
```

T … type of elements
A … type of container
R … type of result of collect

Example: Building average of length of words

- array **int[] a** with length 2 is container
  - sum of word lengths in position **a[0]**
  - number of words in position **a[1]**
- finisher computes average as **a[0] / a[1]**

```
double avrgLength =
    words.collect( Collector.of(
                     () -> new int[2],
                     (a, w) -> { a[0] += w.length(); a[1]++; },
                     (a1, a2) -> { a1[0] += a2[0]; a1[1] += a2[1]; return a1; },
                     a -> a[1] != 0 ? (double) a[0] / a[1] : 0.0
                   )
                 );
```

finisher

Collectors: Factory Methods

- Class Collectors provides factory methods for creating Collector objects

```
public final class Collectors {
  static <T> Collector<T, ?, List<T>> toList()
  static <T> Collector<T, ?, Set<T>> toSet()
  static <T, C extends Collection<T>> Collector<T, ?, C  toCollection(Supplier<C> collectionFactory)
  static <T, K, U> Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
                                                   Function<? super T, ? extends U> valueMapper)
  static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)
  static <T, K, A, D> Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> downstream)
  static <T, K, A, D> Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
                                                            Supplier<M> mapFactory,
                                                       Collector<? super T, A, D> downstream)
  static <T> Collector<T, ?, Double> averagingDouble(ToDoubleFunction<? super T> mapper)
  static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> pred)
   static Collector<CharSequence,?,String> joining(CharSequence delimiter)
   static <T, U, A, R>  Collector<T, ?, R>  mapping(Function<? super T, ? extends U> mapper,  Collector<? super U, A, R> downstream)


   .. and many more ..
}
```

extremely powerful!

Collectors: toList, toSet, toCollection

- **toList** and **toMap**

```
static <T> Collector<T, ?, List<T>> toList()
static <T> Collector<T, ?, Set<T>>  toSet()
```

Examples:

```
List<Person> personList = personStream.collect(Collectors.toList());
```

```
Set<String> setOfWords = wordStream.collect(Collectors.toSet());
```

- **toCollection**: for other collections

```
static <T, C extends Collection<T>> Collector<T, ?, C  toCollection(Supplier<C> collectionFactory)
```

Example: create SortedSet

```
SortedSet<Person> personSortedSet = personStream.collect(Collectors.toCollection(TreeSet::new));
```

COLLECTORS: TOMAP

- **toMap:** creates map from elements

```
static <T, K, U> Collector<T, ?, Map<K,U toMap(Function<? super T, ? extends K> keyMapper,
                                                Function<? super T, ? extends U> valueMapper)
```

  - □ uses functions for getting **keys** and **values** from elements
  - □ requires **unique keys** for all elements; otherwise throws **IllegalStateException**

Example: from Persons create map from names to Persons

```
Map<String, Person> personMap =
   personStream.collect(Collectors.toMap( Person::getName,
                                          Function.identity() );
```

Example: from words create map from word to length

```
Map<String, Integer> wordLengthMap =
   wordStream.distinct()
            .collect(Collectors.toMap( Function.identity(),
                                       w -> w.length() );
```

Note: Throws exception if keys are not unique!

Collectors: toMap with Merge Function

- **toMap: with additional merge function for combining values for equal key**

```
static <T, K, U> Collector<T, ?, Map<K,U toMap( Function<? super T, ? extends K> keyMapper,
                                                 Function<? super T, ? extends U> valueMapper,
                                                 BinaryOperator<U> mergeFunction )
```

  - □ **mergeFunction** used for **combining values** in case of equal keys

Example: map of words to number of occurrences in text

```
Map<String, Integer> wordCount =
    wordStream.collect (
        Collectors.toMap(  w -> w,
                           w -> 1,
                           (n, one) -> n + one) );
```

words are keys

value is 1 for one occurrence

merge function builds sum of occurrences

Collectors: groupingBy

■ groupingBy

```
static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)
```

- □ with **classifier function** providing keys for elements
- □ values are lists of elements falling into group
- □ result is map from keys to **list of elements**

Example: Group words based on initial character

```
Map<Character, List<String>> map =
               words.collect(Collectors.groupingBy(w -> Character.toLowerCase(w.charAt(0))));
```

Result is map from initial character to list of words

```
f -> [Functional, functions]
c -> [comprehensive]
a -> [A]
i -> [in, introduction]
j -> [Java]
l -> [Lambda]
…
```

COLLECTORS: PARTITIONINGBY

- **partitioningBy**

```
public static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> pred)
```

- □ partitions elements based on predicate
- □ result is map from Boolean to list of elements

Example: Partition words starting either with upper and lower case letters

```
Map<Boolean, List<String>> upperLowerWords =
             words.collect(Collectors.partitioningBy(w -> Character.isUpperCase(w.charAt(0))));
```

```
true -> [Functional, Programming,Java, Lambda, A]
false -> [in, with, lambdas,comprehensive, introduction, ...]
```

Mapping true to list of words starting with upper case letter and false to list of words starting with lower case letter

Collectors: joining

- **joining**
  - □ compute a result string, possibly with a delimiter and prefix and postfix string

```
public static Collector<CharSequence, ?, String> joining()
```

```
public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)
```

```
public static Collector<CharSequence, ?, String> joining(CharSequence delimiter,
                                                         CharSequence prefix,
                                                         CharSequence suffix)
```

Example: String of elements with **"["** and **"]"** as prefix and suffix an **","** as delimiter

```
Stream<String> words = Stream.of("Functional", "Programming", "in", ....);
```

```
String wordsSet = words.collect(Collectors.joining(", ", "[", "]"));
```

```
[Functional, Programming, in, Java, with, Lambda, functions, A, comprehensive, introduction]
```

Downstream Collectors

Downstream Collectors

- for collecting results in **groupingBy**, **partitioningBy** collections

```
Collector<T, ?, Map<K, D>> groupingBy( Function<? super T, ? extends K> classifier,
                                       Collector<? super T, A, D> downstream)
```

Example: **groupingBy** with sorted sets as entry values

```
Map<Character, SortedSet<String>> wordGroups =
     wordStream.collect(Collectors.groupingBy(
                          w -> w.charAt(0),
                          Collectors.toCollection(TreeSet::new) )
                    );
```

Use TreeSet as downstream collector

Methods for Downstream Collectors

- Class **Collectors** provides many methods for creating downstream collectors

Method	**Collector for**
<T> Collector<T, ?, Long> counting()	Counting elemens
<T> Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> m) … Analogeous methods for Long and Double …	Summing number values
<T> Collector<T, ?, Integer> averagingInt(ToIntFunction<? super T> m) … Analogeous methods for Long and Double …	Averaging
T> Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> m) … Analogeous methods for Long and Double …	Statistics
Collector<T, ?, Optional<T>> minBy(Comparator<? super T> comparator) Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)	Maximum and minimum
<T, U> Collector<T, ?, U> reducing(U identity, Function<? super T, ? extends U> mapper, BinaryOperator<U> op) <T> Collector<T, ?, T> reducing(T identity, BinaryOperator<T> op) <T> Collector<T, ?, Optional<T>> reducing(BinaryOperator<T> op)	Reducing
<T, R> Collector<T, ?, R> filtering(Predicate<? super T> predicate, Collector<? super T, A, R> downstream)	Filtering plus collecting with next downstream collector
<T, U, A, R> Collector<T, ?, R> mapping(Function<? super T, ? extends U> m, Collector<? super U, A, R> downstream)	mapping plus collecting with next downstream collector
<T, U, A, R> Collector<T, ?, R> flatMapping(Function<? super T, ? extends Stream<? extends U>> mapper, Collector<? super U, A, R> downstream)	Mapping, flattening and with next downstream collector
<T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)	Collecting then computing final result with finisher function

allows chaining collectors

Downstream Collectors: Examples

Example: groupingBy and counting

```
Map<Character, Long> wordGroupsCounts =
     wordStream.collect(Collectors.groupingBy(
                                w -> w.charAt(0),
                                Collectors.counting()
                        ));
```

Example: grouping persons by age and result are sorted set of names

```
Map<Integer, SortedSet<String>> ageToNames =
       personStream.collect(
           Collectors.groupingBy( p -> p.getAge(),
                 Collectors.mapping(
                     person -> person.getName(),
                     Collectors.toCollection(TreeSet::new)
                 )
            )
       );
```

chaining collectors

Example: grouping words by initial character, with mapping to length and averaging

```
Map<Character, Double> wordGroupsAvrgLength =
     wordStream.collect (Collectors.groupingBy(w -> w.charAt(0),
                                   Collectors.mapping(w -> w.length(),
                                           Collectors.averaging())
                              );
```

COLLECTORS: FACTORIES

- Factories allow using special result container

```
<T, K, D, A, M extends Map<K, D>> Collector<T, ?, M> groupingBy(
                                      Function<? super T, ? extends K> classifier,
                                      Supplier<M> mapFactory,
                                      Collector<? super T, A, D> downstream
                                  )
```

Example: grouping persons by age and result are sorted set of names in **SortedMap**

```
SortedMap<Integer, SortedSet<String>> ageToNamesSorted =
      personStream.collect(
          Collectors.groupingBy(
                Person::getAge,
                TreeMap::new,
                Collectors.mapping(
                    p -> p.getName(),
                    Collectors.toCollection(TreeSet::new)
                )
           )
         );
```

getting **SortedMap**

using TreeMap

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

in lab

Exercise

- See Excercises

8 Lazy Evaluation and Streams

Lazy evaluation

- Lazy lists

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Use Only Once

- Streams can only be iterated once
  - □ ➔ after iterating, the stream has to be recreated

```
List<String> words = …;
Stream wordStream = words.stream();
wordStream.forEach(System.out::println);

wordStream.forEach(System.out::println); // erronous
```

This does not work!
stream already used

```
List<String> words = …;
Stream wordStream = words.stream();
wordStream.forEach(System.out::println);

Stream wordStream = words.stream();
wordStream.forEach(System.out::println);
```

Must recreate stream

Limit Infinite Streams

- No terminal operation which processes whole stream on infinite streams

```
IntStream posInts = IntStream.iterate(0, i -> i + 1);

IntStream evenSqrs =  posInts .filter(x -> x % 2 == 0) .map(x -> x * x);

evenSqrs.forEach(System.out::println); // erronous
```

This is fine as streams are lazy!

This will run forever!

```
IntStream posInts = IntStream.iterate(0, i -> i + 1);

IntStream evenSqrs  posInts .filter(x -> x % 2 == 0) .map(x -> x ** x);

evenSqrs
  .limit(100)
  .forEach(System.out::println);
```

With limit its fine!

```
IntStream randInts = IntStream.rands(0, 100);

evenSqrs.filter(x -> isPrime(x)).findFirst();
```

find is fine on infinite stream
but still is infinite when no prime found

Stateless vs. Stateful Operations

- Intermediate operations can be stateless or stateful

Stateless operations

- □ each element can be processed without information about other elements
- □ Examples: **map**, **filter**

```
articles.stream()
  .filter(a -> a.price > 1000)
  .map(a -> a.name)
  …
```

no information about other elements needed for filtering or mapping a single article

Stateful operations

- □ a state about the processing of other/all elements is needed
- □ Examples: **sorted**, **distinct**

```
articles.stream()
  .filter(a -> a.price > 1000)
  .sorted(Comparator.comparingDouble((Article a) -> a.price))
  …
```

for determining the position of an element one must know all the other elements

```
words.stream()
  .distinct()
  …
```

for knowing if a word is new, one must know all the other words already processed

Stateless vs. Stateful Operations

- There is a significant difference in the execution of stateless and stateful operations

Stateless operations

- □ elements pass all stateless operations one-by-one

```
words = List.of("Functional", "Programming", "in", "Java" );
```

prints show execution

```
words.stream()
    .filter(w -> {
       System.out.println("- filter " + w);
       return w.length() > 1;
    })
    .map(w ->  {
       System.out.println("- map " + w);
       return w.toLowerCase();
    })
    …
```

each element processed by filter and map

```
- filter Functional
- map Functional

- filter Programming
- map Programming

- filter in
- map in

- filter Java
- map Java
```

Stateless vs. Stateful Operations

- Stateful operation sorted
  - □ all elements must be processed by sorted before elements can be forwarded

```
words.stream()
      .filter(w -> {
        System.out.println("- filter " + w);
        return w.length() > 1;
      })
      .sorted()
      .map(w ->  {
        System.out.println("- map " + w);
        return w.toLowerCase();
      })
      .forEach(w -> System.out.println(w));
      …
```

```
- filter Functional
- filter Programming
- filter in
- filter Java

- map Functional
- map in
- map Java
- map Programming
```

sorted requires all elements filtered

after sorting map can be performed

  - □ thus, stateful operations on infinite streams fail

```
Stream<Point> randomWalk = Stream.iterate( new Point(100, 100),  p -> new Point(p.x + r.nextInt(DIST), p.y + r.nextInt(DIST)) );

randomWalk
    .sorted(Comparator.comparing(p -> p.x))
    .limit(100)
    .forEach(System.out::println);
```

cannot sort an infinite stream

Combinations of Operations

Combinations of operations must be considered

- do not use **sorted** and **TreeSet** together → will sort twice

```
SortedSet<String> sortedWords =
    words.stream()
        .map(String::toLowerCase)
        .sorted()
        .collect(TreeSet<String>::new, TreeSet<String>::add, TreeSet<String>::addAll);
```

sort twice:

- once by sorted,
- another time by TreeSet

- do not use **sorted** and **HashSet** together
  → HashSet will not preserve sorting

```
Set<String> unsortedWords =
      words.stream()
          .map(String::toLowerCase)
          .sorted()
          .collect(HashSet<String>::new, HashSet<String>::add, HashSet<String>::addAll);
```

Result is not sorted because
sorting not preserved by HashSet

```
in
java 8
action
lambdas
```

Combinations of Operations

Combinations of operations must be considered

- do not use **distinct** and **Sets** together → Sets will eliminate duplicates anyway

```
Set<String> distinctWords =
      words.stream()
          .map(String::toLowerCase)
          .distinct()
          .collect(HashSet<String>::new, HashSet<String>::add, HashSet<String>::addAll);
```

eliminate duplicates twice:
- once by distinct,
- another one by HashSet

java 8
in
lambdas
action

8 Lazy Evaluation and Streams

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Spliterator

Splitable
iterator

- Spliterators are the internal realization of streams
- for sequential and parallel processing
  - □ **Iterator** for sequential processing
  - □ **Split** for splitting the stream for parallel processing

```
public interface Spliterator<T> {

    Spliterator<T> trySplit();                                        Split

    long estimateSize();

    default long getExactSizeIfKnown() {
        return (characteristics() & SIZED) == 0 ? -1L : estimateSize();
    }

    boolean tryAdvance(Consumer<? super T> action);

    default void forEachRemaining(Consumer<? super T> action) {
        do { } while (tryAdvance(action));
    }                                                              iterator

    int characteristics();

    default boolean hasCharacteristics(int characteristics) {
        return (characteristics() & characteristics) == characteristics;
    }
    ...
}
```

for parallel execution
➔ see Part 3

works similar
to an iterator

For sequential processing.
Elements are processed by
consumer function action.

For investigating
properties of stream

Class Spliterators

- Utility class with static methods for creating and handling Spliterators

```
public final class Spliterators {

  public static <T> Spliterator<T> emptySpliterator()

  public static <T> Spliterator<T> spliterator(Collection<? extends T> c,
                                               int characteristics)
  public static <T> Spliterator<T> spliterator(Iterator<? extends T> iterator,
                                               long size,
                                               int characteristics)
  public static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator,
                                                          int characteristics)

  ...

  public static<T> Iterator<T> iterator(Spliterator<? extends T> spliterator)

  ... and many more ...
}
```

- creating Spliterators from Collections and Iterators
- with different characteristics
- creating Iterators from Spliterators

Stream.Builder

- A mutable building for collecting elements and creating streams

```
public interface Stream<T>   {
  …
  public static<T> Builder<T> builder() { … }

  public interface Builder<T> extends Consumer<T> {
     void accept(T t);
     default Builder<T> add(T t)
     Stream<T> build();
  }
}
```

same for IntStream etc.

```
Stream.Builder<String> b = Stream.builder();

b.accept("A");
b.accept("B");
b.accept("C");

Stream<String> absStrm = b.build();
```

StreamSupport

- StreamSupport provides methods for creating streams from a Spliterator

```
public final class StreamSupport {

  public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)

  ...

}
```

```
Spliterator<String> spliterator = new AbstractSpliterator() { … }

Stream<String> strm = StreamSupport.stream(spliterator, false);
```

Non-strict evaluation in Scala

Java Streams

- Introduction
- Building streams
- Intermediate operations
- Terminal operations
- Collectors
- Examples
- Hints
- Low-Level API
- Summary

Summary

- Streams are a powerful mechanisms for processing sequences of elements
- from difference sources
  - collections
  - generators
  - files
  - …
- Streams implement map – filter – reduce pattern with chains of
  - a source operation
  - some intermediate operations
  - a terminal operation
- Streams are lazy
- Streams support parallel processing → see next section