Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- Functional Interfaces & Lambda Expressions
- Streams and Functional Programming

Literature

- J. Bloch, **Effective Java**, Third Edition, Addison Wesley, 2018
- Richard Warburton**, Java 8 Lambdas,** O'Reilly Media, 2014
- James Gosling, et al. **Java Language Specification**

  https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html#jls-LambdaBody

Lambda Expressions (Anonymous Functions)

- Lambda expressions enable to pass functionality/code to methods by instantiating functional interfaces instead of using local or anonymous classes.

Syntax:

```
LambdaParameters -> LambdaBody

LambdaParameters:
        ( [FormalParameterList] )
        ( InferredFormalParameterList )

LambdaBody:
        Expression
        Block
```

Example:

```
interface Predicate<T> { boolean test(T t); }
```

Functional Interface
(java.util.Function)

```
printPersons(pl, p -> p.getAlter() < 18);
```

Lambda Expressions (Anonymous Functions)

- Lambda expressions make it possible to treat functionality as method argument, or to treat code as data.

- Lambda expressions offer a more compact syntax than (anonymous) classes.

- Java lambda expressions are a limited form of closures as available in other languages (cf. Scala, Groovy, ...).

Example 1: Old Style (bad)

```
public class PrintPersons {
  public static void printPersonsOlderThan(List<Person> pl, int age) {
      for(Person p:pl)
         if (p.getAge() > age) System.out.println(p.getName());
  }
  public static void printPersonsYoungerThan(List<Person> pl, int alter) {
      for(Person p:pl)
         if (p.getAge() < age) System.out.println(p.getName());
   }

  public static void main(String[] args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println("Persons older than 18:");
      printPersonsOlderThan(pl, 18);

      System.out.println("Persons younger than 18:");
      printPersonsYoungerThan(pl, 18);
  }
}
```

How can we avoid writing
a new print-method
for each new test?

Example 1: Anonymous Class

```
interface CheckPerson { public boolean test(Person p); }

public class PrintPersons {
   public static void printPersons(List<Person> pl, CheckPerson tester) {
      for(Person p:pl)
          if (tester.test(p)) System.out.println(p.getName());
   }

   public static void main(String[] args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println("Persons older than 18:");
      printPersons(pl, new CheckPerson() {
         public boolean test(Person p) { return p.getAge() > 18;}});

      System.out.println("Persons younger than 18:");
      printPersons(pl, new CheckPerson() {
         public boolean test(Person p) { return p.getAge() < 18;}});
   }
}
```

Functional Interface

Anonymous Class

Example 1: Lambda Expressions

Functional Interface

```
interface CheckPerson { public boolean test(Person p); }

public class PrintPersons {
   public static void printPersons(List<Person> pl, CheckPerson tester) {
      for(Person p:pl)
          if (tester.test(p)) System.out.println(p.getName());
   }

   public static void main(String[] args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println(" Persons older than 18:");
      printPersons(pl, p -> p.getAge() > 18);

      System.out.println(" Persons younger than 18:");
      printPersons(pl, p -> p.getAge() < 18);
   }
}
```

Functional Interfaces

- Functional interfaces provide target types for lambda expressions and method references.
- Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameters and return types are matched or adapted.
- Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

```
Predicate<String> p = s -> s.isEmpty();   // Assignment context

stream.filter(e -> e.getSize() > 10)...   // Method invocation ctx.

stream.map((ToIntFunction) e -> e.getSize())... // Cast context
```

See: package java.util.function

Example 1: Lambda Expressions

```
interface Predicate<T> { boolean test(T t); }
```

Functional Interface
(java.util.Function)

```
public class PrintPersons {
   public static void printPersons(List<Person> pl,
                                   Predicate<Person> tester) {
      for(Person p:pl)
          if (tester.test(p)) System.out.println(p.getName());
   }

   public static void main(String[] args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println("Persons older than 18:");
      printPersons(pl, p -> p.getAge() > 18);

      System.out.println("Persons younger than 18:");
      printPersons(pl, p -> p.getAge() < 18);
   }
}
```

Example 1: Lambda Expressions + Streams

```
interface Predicate<T> { boolean test(T t); }
```

Functional Interface
(java.util.Function)

```
public class PrintPersons {
   ~~public static void printPersons(List<Person> pl,~~
                                              ~~Predicate<Person> tester) {~~
      ~~for(Person p:pl)~~
          ~~if (tester.test(p)) System.out.println(p.getName());~~
   ~~}~~

   public static void main(String[] args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println("Persons older than 18:");
      pl.stream().filter(p -> p.getAge() > 18)
                 .forEach(p -> System.out.println(p.getName()));
      ...
   }
}
```

Functional Interfaces - Examples

Some basic functional interfaces ...

```
interface Consumer<T> { void accept(T t); }
   // Represents an operation that accepts a single input argument
   // and returns no result.

interface Function<T,R> { R apply(T t); }
   // Represents a function that accepts one argument and produces
   // a result.

Predicate<T> { boolean test(T t); }
   // Represents a predicate (boolean-valued function) of one
   // argument.

Supplier<T> { T get(); }
   // Represents a supplier of results.
```

See: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Lambda Expressions - Examples

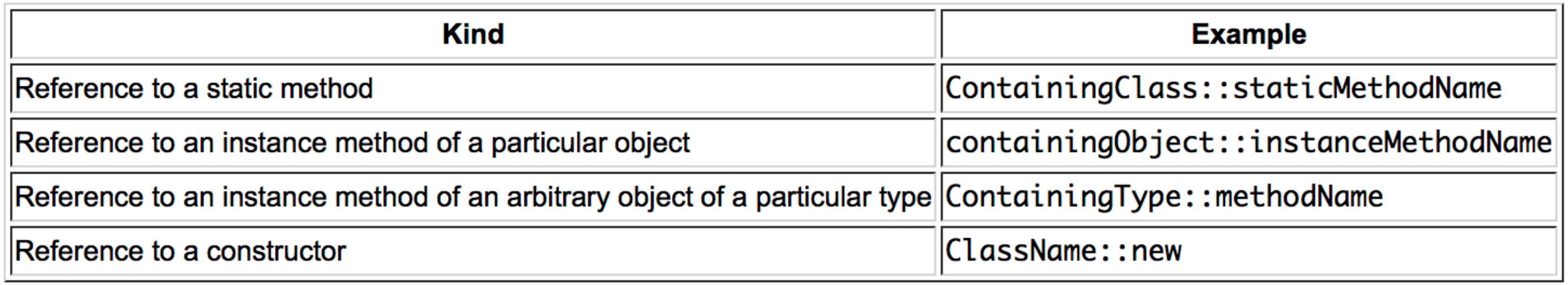
```
(int x) -> x+1              // Single declared-type parameter
(int x) -> { return x+1; }  // Single declared-type parameter
(x) -> x+1                  // Single inferred-type parameter
x -> x+1                    // Parentheses optional for
                                 // single inferred-type parameter

(String s) -> s.length()      // Single declared-type parameter
(Thread t) -> { t.start(); }  // Single declared-type parameter
s -> s.length()               // Single inferred-type parameter
t -> { t.start(); }           // Single inferred-type parameter

(int x, int y) -> x+y  // Multiple declared-type parameters
(x, y) -> x+y          // Multiple inferred-type parameters
(x, int y) -> x+y     // Illegal: can't mix inferred/declared types
(x, final y) -> x+y   // Illegal: no modifiers with inferred types
```

Method References

- If a lambda expression just calls an existing method it can be substituted by a method reference.



Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

```
Predicate<String> p1 = String::isEmpty; // method reference

Predicate<String> p2 = s -> s.isEmpty();
```

See: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Example 2: List Files - Anonymous Class

```
class File {
   ...
   String[] list(FilenameFilter filter));
}
```

```
// see package java.io
public interface FilenameFilter {
   boolean accept(File dir, String name);
}
```

```
import java.io.*;

public class ListFilesAnonymousClass {

   public static void main(String[] args) {
      File dir = new File("src");   // directory to list
                                    // dir.list() requires
      String[] filelist = dir.list( // a FilenameFilter
         new FilenameFilter() {
            public boolean accept(File dir, String name) {
               return name.endsWith(".java");
            }
         }
      );
      for (String s : filelist) System.out.println(s);
   }
}
```

Example 2: List Files - Local Class

```java
import java.io.*;

public class ListFilesLocalClass {

   public static void main(String[] args) {
      File dir = new File("src");      // The directory to list

      class myFilenameFilter implements FilenameFilter{
         public boolean accept(File dir, String name) {
            return name.endsWith(".java");
         }
      }

      myFilenameFilter myFF = new myFilenameFilter();

      String[] filelist = dir.list(myFF);
      for (String s : filelist) System.out.println(s);
   }
}
```

Example 2: List Files – Lambda Expression

```
public interface FilenameFilter {
   boolean accept(File dir, String name);
}
```

```
import java.io.*;

public class ListFilesLambda {

   public static void main(String[] args) {
      File dir = new File("src");       // The directory to list

      class myFilenameFilter implements FilenameFilter{
         public boolean accept(File dir, String name) {
            return name.endsWith(".java");
         }
      }

      myFilenameFilter myFF = new myFilenameFilter();

      String[] filelist = dir.list((d, n) -> n.endsWith(".java"));
      for (String s : filelist) System.out.println(s);
   }
}
```

Assignment 1 – old style code duplication

```
public int countApartments(){
   return pmDAO.getApartments().size();
}

public int countOwnedAppartments(){
   int count = 0;
   for (Appartment a : pmDAO.getApartments()) {
      if (a instanceof OwnedAppartments) count++;
   }
   return count;
}

public int countRentedAppartments(){
   int count = 0;
   for (Appartment a : pmDAO.getApartments()) {
      if (a instanceof RentedAppartments) count++;
   }
   return count;
}
```

DRY principle: Don't repeat yourself

Assignment 1 – Lambda Expressions

- No code duplication!
- Method **`count()`** also works if new sub-classes of **`Appartment`** are added at a later time!

```
import java.util.function.Predicate;
...

public int count(Predicate<Appartment> p){
   int cnt = 0;
   for (Appartment a : pmDAO.getApartments()) {
       if (p.test(a)) cnt++;
   }
   return cnt;
}
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
nrAppartments      = count(a -> a instanceof Appartments);
nrOwnedApartments  = count(a -> a instanceof OwnedAppartments);
nrRentedApartments = count(a -> a instanceof RentedAppartments);
```

Streams

A Stream

- provides an interface to a finite or infinite sequence of elements.
- consumes from a data-providing source such as collections, arrays, or I/O resources using (default) methods that return `Stream`.
- doesn't actually store elements; they are computed on demand.
- keeps the order of the data as it is in the source.

Stream interface is available in **`java.util.stream`** package since Java 8.

Streams

- Streams support aggregate operations (functional programming style)

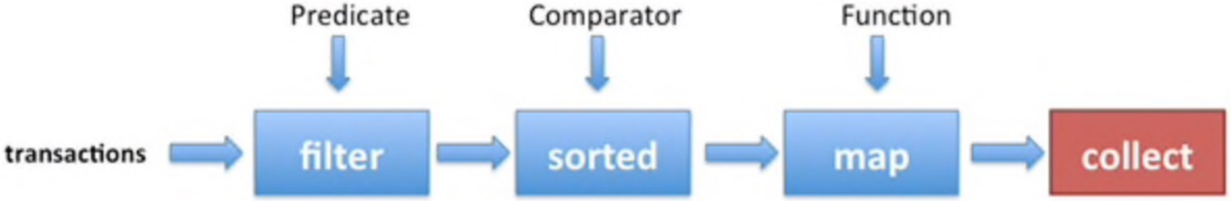
  `filter`, `map`, `reduce`, `find`, `match`, `sorted`, etc. .
- Streams support pipelining (chaining of operations) .
- Streams support internal iterations (e.g., `forEach, filter`, …).
- Operations can be executed sequentially or in parallel.

```
List<Person> pl = new ArrayList<>();
...
System.out.println("Persons older than 18:");
pl.stream()
  .filter(p -> p.getAge() > 18)
  .forEach(p -> System.out.println(p.getName()));
```

Example

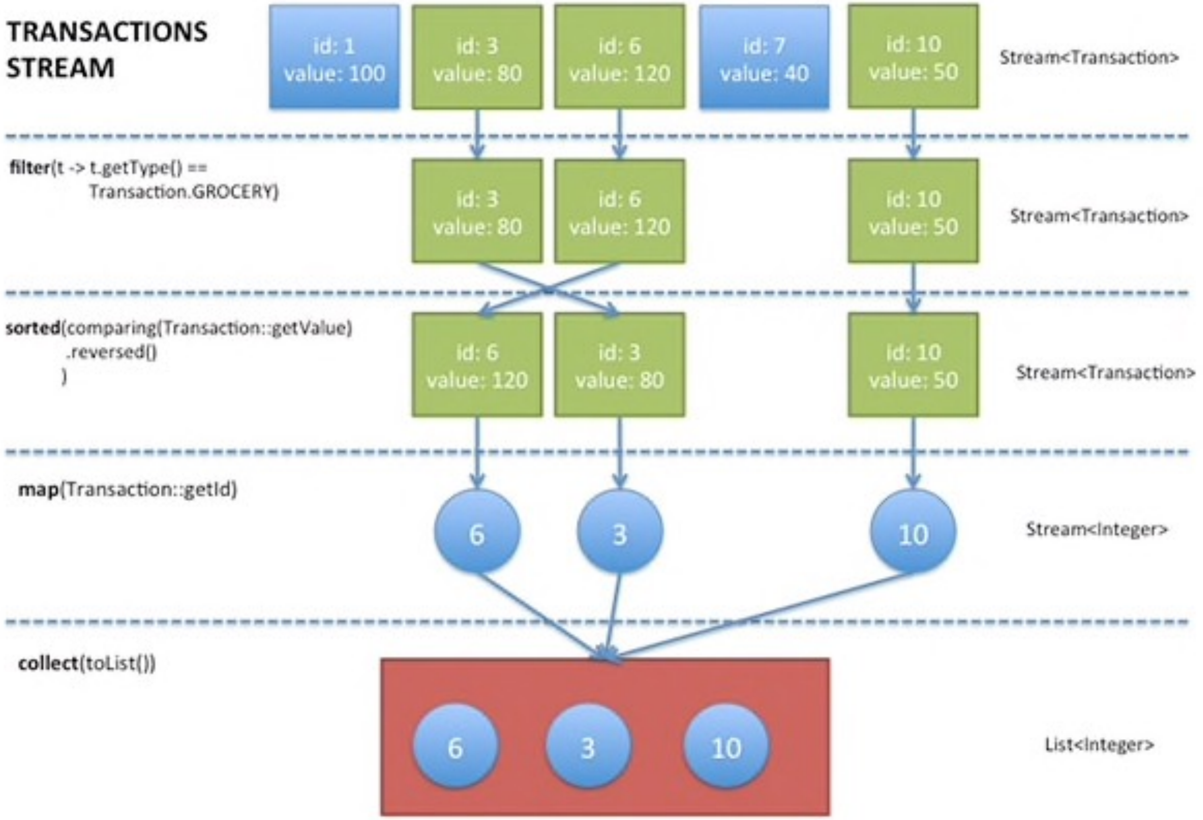
https://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
  if(t.getType() == Transaction.GROCERY)
    groceryTransactions.add(t);
}
Collections.sort(groceryTransactions, new Comparator(){
  public int compare(Transaction t1, Transaction t2){
    return t2.getValue().compareTo(t1.getValue());
  }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions)
  transactionIds.add(t.getId());
```



```
List<Integer> transactionIds = transactions.stream()
             .filter(t -> t.getType() == Transaction.GROCERY)
             .sorted(comparing(Transaction::getValue).reversed())
             .map(Transaction::getId)
             .collect(toList());
```

Example



```
List<Integer> transactionsIds = transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Streams vs. Collections

- No storage

  A stream is not a data structure that stores elements; instead, it conveys elements from a source through a pipeline of computational operations.

- **Functional** in nature

  An operation on a stream produces a result, but does not modify its source.

- Laziness

  Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization.

- Possibly unbounded

  While collections have a finite size, streams need not. Short-circuiting operations such as **`limit(n)`** or **`findFirst()`** can allow computations on infinite streams to complete in finite time.

Stream Operations and Pipelines

- Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines.
- A stream pipeline consists of
  - a source (such as a Collection, an array, a generator function, or an I/O channel);
  - followed by zero or more intermediate operations such as filter or map;
  - and a terminal operation such as **`forEach`**, **`reduce`** or **`collect`**.

```
List<Integer> transactionsIds = transactions.stream()
             .filter(t -> t.getType() == Transaction.GROCERY)
             .sorted(comparing(Transaction::getValue).reversed())
             .map(Transaction::getId)
             .collect(toList());
```

Streams – Intermediate Operations

- Intermediate operations such as **`filter()`** are always lazy. They do not actually perform any filtering, but instead produce a new stream that, when traversed, contains only the elements that match the given predicate.
- Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.
- Processing streams lazily enables many optimizations, e.g., to fuse filtering, mapping, and summing into a single pass on the data, with minimal intermediate state, or to avoiding examining all the data.

```
IntStream is = IntStream.rangeClosed(1, N)
              .map(i -> ThreadLocalRandom.current().nextInt(1, N))
              .sorted();
...
int iArray = is.toArray();  // sorting is done here !!!
```

Streams – Terminal Operations

- Terminal operations, such as **`forEach()`** or **`sum()`**, may traverse the stream to produce a result or a side-effect.

- After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used;

  `java.lang.IllegalStateException: stream has already been operated upon or closed`

- Most terminal operations are eager, completing traversal of the data source and processing before returning.

Streams – Stateless vs. Stateful Operations

Stateless operations

- such as `filter` and `map`, retain no state from previously seen elements when processing a new element - all elements can be processed independently.
- Pipelines containing only stateless intermediate operations can be processed in a single pass with minimal data buffering.

Stateful operations

- such as `distinct` and `sorted`, may incorporate state from previously seen elements when processing new elements.
- Stateful operations may need to process the entire input before producing a result (e.g., sorting).

Streams – Filtering

Operations to filter elements from a stream:

- **`filter(predicate):`** takes a predicate (**`java.util.function.Predicate`**) as an argument and returns a stream including all elements that match the given predicate
- **`distinct()`**: Returns a stream with unique elements (according to **`equals()`**) of this stream
- **`limit(n):`** Returns a stream that is no longer than the given size n
- **`skip(n):`** Returns a stream with the first n number of elements discarded

```
int nrOwnedAppartments = (int) pmDAO.getAppartments()
                          .stream()
                          .filter(v -> v instanceof OwnedAppartment)
                          .count();
```

Streams – Matching and Finding

- **`anyMatch(p), allMatch(p), noneMatch(p)`**

  take a predicate **`p`** as an argument and return a boolean as the result indicating whether some elements match a given property.

```
boolean expensive = transactions
                   .stream()
                   .allMatch(t -> t.getValue() > 100);
```

- **`findFirst()`** and **`findAny()`** retrieve arbitrary elements from a stream. They can be used in conjunction with other stream operations such as filter. Both return an **`Optional`** object.

```
Optional<Transaction> = transactions
            .stream()
            .filter(t -> t.getType() == Transaction.GROCERY)
            .findAny();
```

Streams – Optional Class

- The `Optional<T>` container class represents the existence or absence of a value. An `Optional` object may or may not contain a non-null value.
- `findAny()` returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
- `isPresent()` returns true if a value is present and `get()` returns the value.
- `orElse(T other)` returns a default value if value not present
- `ifPresent()` executes a block of code if the value is present

`Optional<T> findAny()`
Returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty.

```
transactions.stream()
            .filter(t -> t.getType() == Transaction.GROCERY)
            .findAny()
            .ifPresent(System.out::println);
```

Streams – Mapping

- Streams support the various map methods, which take a function (`java.util.function.Function`) as an argument to project the elements of a stream into another form.
- The function is applied to each element, mapping it into a new element.

```
List<Appartments> al = ...

Predicate<Appartment> p = a -> a instanceof OwnedAppartment;
...
double averageAgeOwnedAppartment = al.stream()
                                 .filter(p)
                                 .mapToDouble(Appartment::getAge)
                                 .average()
                                 .getAsDouble();
```

Streams – Reductions

- A reduction operation takes a sequence of input elements and combines them into a single result by repeated application of a combining operation, such as the sum or maximum of a set of numbers.

- The streams classes provide multiple reduction operations, including `reduce(),collect(),sum(),max(),count().`

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

sum = numbersList.stream().reduce(0, (a, b) -> a + b);

// or equivalently

sum = numbersList.stream().reduce(0, Integer::sum);
```

Streams – Collectors

Collectors are implementations of the interface **`Collector`** that implement various useful reduction operations, such as

- accumulating elements into collections,
- summarizing elements according to various criteria.

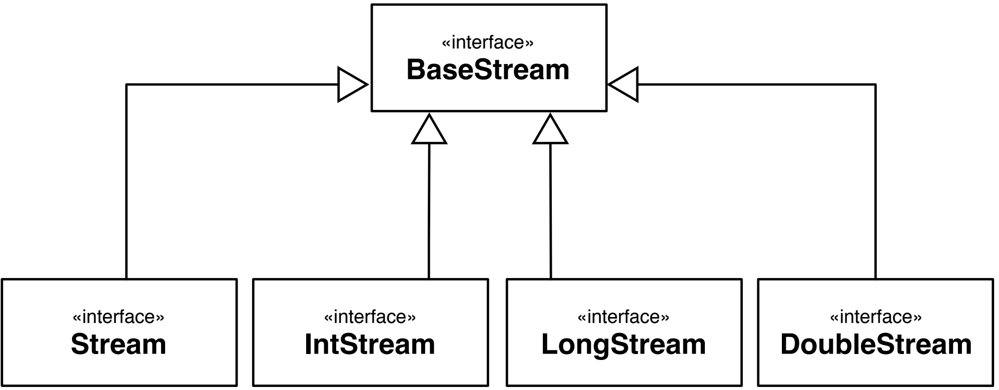
```
// Partition students into passing and failing

Map<Boolean, List<Student>> passingFailing = students
       .stream()
       .collect(Collectors.partitioningBy(s -> s.getGrade() >=
                                               PASS_THRESHOLD));
```

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/stream/Collectors.html

Streams – Numeric Streams

- There are three specialized interfaces for streams of elements of primitive type int, long or double.



- For performance reasons (avoiding boxing operations); `IntStream`, `DoubleStream`, and `LongStream` should be preferred over streams containing `Integer`, `Double`, or `Long` objects.

```
int numbersArray[] = {1,2,3,4,5,6,7,8,9,10};

IntStream is = Arrays.stream(numbersArray);
int sum = is.reduce(0, (a, b) -> a + b);    // or just use sum()
```

```
Stream<Integer> s = Stream.of(1,2,3,4,5,6,7,8,9,10);

sum = s.reduce(0, (a, b) -> a + b);
```

Inefficient!!

Streams – Infinite Streams

- `Stream.iterate()` and `Stream.generate()` allow creating a stream from a function.
- Because in streams elements are calculated on demand, these two operations can produce elements "forever."
- This is called an infinite stream: as opposed to fixed size stream as created from a collection.
- An infinite stream can be turned into a fixed-size stream using the `limit()` operation.

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10); //infinite

numbers.limit(4).forEach(System.out::println); // 0, 10, 20, 30
```

Example 1: Lambda Expressions & Streams

```
import java.util.*;

public class PersonMgmt {

   public static void main(String...args) {
      List<Person> pl = new ArrayList<>();
      pl.add(new Person("Sara", 30));
      pl.add(new Person("Hans", 28));
      pl.add(new Person("Lisa", 17));

      System.out.println("Persons older than 18:");
       pl.stream()
         .filter(p -> p.getAlter() > 18)
         .forEach(p -> System.out.println(p.getName()));
   }

}
```

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

```
@FunctionalInterface
public interface Consumer<T>{
   void   accept(T t)
}
```

Stream<T> filter(**Predicate**<? super T> predicate)
Returns a stream consisting of the elements of this stream that match the given predicate.

void forEach(**Consumer**<? super T> action)
Performs an action for each element of this stream.

Assignment 1 – Lambda Expressions

- No code duplication!
- Method **`count()`** also works if new sub-classes of **`Appartment`** are added at a later time!

```
import java.util.function.Predicate;
...

public int count(Predicate<Appartment> p){
   int cnt = 0;
   for (Appartment a : pmDAO.getApartments()) {
       if (p.test(a)) cnt++;
   }
   return cnt;
}
```

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

For loop can be replaced by a more concise stream expression.

```
nrAppartments        = count(a -> a instanceof Appartments);
nrOwnedApartments    = count(a -> a instanceof OwnedAppartments);
nrRentedApartments = count(a -> a instanceof RentedAppartments);
```

Assignment 1 – Lambda Expressions & Streams

- No code duplication!
- Method **`count()`** also works if new sub-classes of **`Article`** are added later!

```
import java.util.function.Predicate;
...

public int count(Predicate<Appartment> p) {
    return (int) pmDAO.getAppartments()
                      .stream().filter(p).count();
}


nrOwnedAppartments    = count(a -> a instanceof OwnedAppartment);
```

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

Use of stream expresssion instead of for loop.

```
nrOwnedAppartments = pmDAO.getAppartments()
            .stream()
            .filter(a -> a instanceof OwnedAppartment)
            .count();
```

Just use streams.

Streams – Parallelism

- Streams operations can execute either in serial (default) or in parallel.
- `Collection.stream()` returns a sequential stream
- `Collection.parallelStream()` returns a possibly parallel stream

```
double averageAgeOwnedAppartment = al.stream()
                            .filter(p)
                            .mapToDouble(Appartment::getAge)
                            .average()
                            .getAsDouble();
```

- Alternatively, a sequential stream may be transformed into a parallel stream by invoking its `BaseStream.parallel()` method.

```
double averageAgeOwnedAppartment = al.stream().parallel()
                            .filter(p)
                            .mapToDouble(Appartment::getAge)
                            .average()
                            .getAsDouble();
```

Example: 2D Stencil

```
final int N = 100_000_000; final int MAX_STEPS = 1000;
double a[] = new double[N+1]; double anew[] = new double[N+1];

System.out.println("Cores: " + Runtime.getRuntime().availableProcessors());
a[0] = 0; a[N] = 1; anew[0] = 0; anew[N] = 1;

long t1 = System.currentTimeMillis();
for(int step = 1;step <= MAX_STEPS; step++) {
   IntStream.range(1, N)
  .parallel()
  .forEach(i -> anew[i] = (a[i-1]+a[i+1])/2
                        - Math.sqrt(1+a[i-1]) + Math.sqrt(1+a[i+1]));
   IntStream.range(0, N+1)
  .parallel()
  .forEach(i -> a[i] = anew[i]);
}
long t2 = System.currentTimeMillis();
System.out.println("time: " + (t2-t1) +" (ms)");
```

Apple M1, 4+4 cores, 16GB RAM

- Sequential execution time: 166s
- Parallel execution time: 69s

Apple M1 Ultra, 16+4 cores, 64GB RAM

- Sequential execution time: 145s
- Parallel execution time: 13s