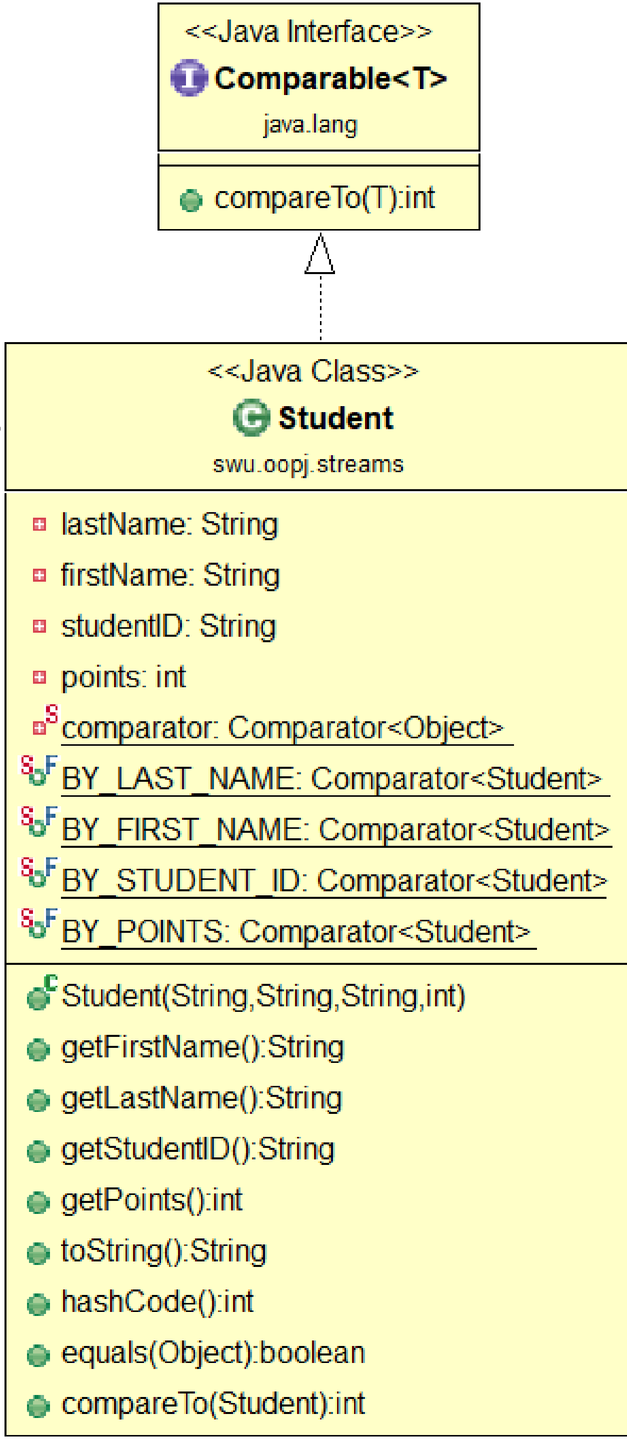
Object Oriented Programming in Java

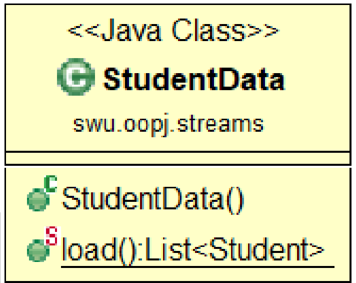
12: Stream API

Licence

Data classes used in examples

- Student has first name, last name, id, and number of points
  - read only attributes, values set in constructor
- Can be used in various collections as it
  - overrides *equals* and *hashcode*
  - implements Comparable
    - natural comparator (compares by id)
  - 4 additional comparators (one for each attribute)
    - Comparing strings in specific language using appropriate comparator (see next slide)
- Hard coded list of students in *StudentData*





Language specific comparators

- Natural comparator for *String* compares strings by their encoding value but each language has own way to compare

12_.../swu/oopj/streams/CompareTest.java

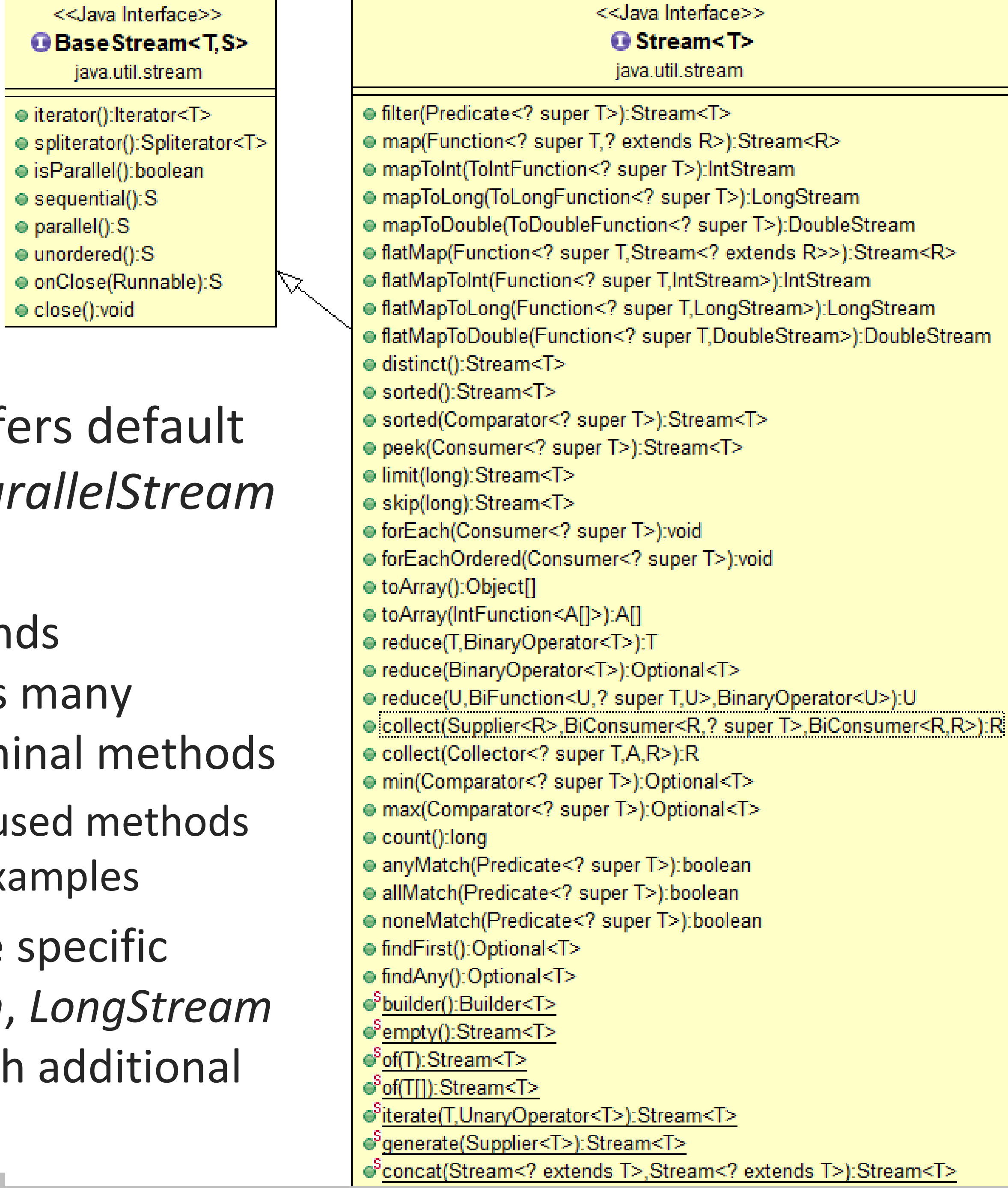
```
Comparator<Object> compHR = Collator.getInstance(
                     Locale.forLanguageTag("hr")); //Croatian
Comparator<Object> compChina = Collator.getInstance(Locale.CHINA);
String s1 = "王"; //Wáng should be before Zhōu
String s2 = "周"; //Zhōu
System.out.println(s1.compareTo(s2)); // 7971
System.out.println(compChina.compare(s1, s2)); // -1
System.out.println(compHR.compare(s1, s2)); // 1

s1 = "č"; //in Croatian č (like Ch in Chongqing) should be before ć
s2 = "ć"; //ć (like q in Chongqing)
System.out.println(s1.compareTo(s2)); // 6
System.out.println(compChina.compare(s1, s2)); // 1
System.out.println(compHR.compare(s1, s2)); // -1
```

Java streams and Java Stream API

- The Java Stream API provides a functional approach to processing collections of objects
  - Not to be confused with input/output streams (of bytes)
- A stream pipeline consists of
  - a **source** (an array, a collection, an I/O channel, …)
  - zero or more **intermediate operations** which transform a stream into another stream (e.g. filtering, sorting, mapping to another type)
  - a **terminal operation** which produces a result (e.g. count, average, new collection) or side-effect (e.g. printing elements)
- Streams are lazy
  - computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.
  - Creating new stream does not create a copy of a source

Creating a stream from a collection

- Interface Collections offers default methods *stream* and *parallelStream* that creates a stream
  - Interface *Stream* extends *BaseStream* and offers many intermediate and terminal methods
    - Some of frequently used methods are shown later in examples
  - There also exists more specific streams like *IntStream*, *LongStream* and *DoubleStream* with additional methods



<<Java Interface>>
BaseStream<T,S>
java.util.stream

- iterator():Iterator<T>
- spliterator():Spliterator<T>
- isParallel():boolean
- sequential():S
- parallel():S
- unordered():S
- onClose(Runnable):S
- close():void

<<Java Interface>>
Stream<T>
java.util.stream

- filter(Predicate<? super T>):Stream<T>
- map(Function<? super T,? extends R>):Stream<R>
- mapToInt(ToIntFunction<? super T>):IntStream
- mapToLong(ToLongFunction<? super T>):LongStream
- mapToDouble(ToDoubleFunction<? super T>):DoubleStream
- flatMap(Function<? super T,Stream<? extends R>>):Stream<R>
- flatMapToInt(Function<? super T,IntStream>):IntStream
- flatMapToLong(Function<? super T,LongStream>):LongStream
- flatMapToDouble(Function<? super T,DoubleStream>):DoubleStream
- distinct():Stream<T>
- sorted():Stream<T>
- sorted(Comparator<? super T>):Stream<T>
- peek(Consumer<? super T>):Stream<T>
- limit(long):Stream<T>
- skip(long):Stream<T>
- forEach(Consumer<? super T>):void
- forEachOrdered(Consumer<? super T>):void
- toArray():Object[]
- toArray(IntFunction<A[]>):A[]
- reduce(T,BinaryOperator<T>):T
- reduce(BinaryOperator<T>):Optional<T>
- reduce(U,BiFunction<U,? super T,U>,BinaryOperator<U>):U
- collect(Supplier<R>,BiConsumer<R,? super T>,BiConsumer<R,R>):R
- collect(Collector<? super T,A,R>):R
- min(Comparator<? super T>):Optional<T>
- max(Comparator<? super T>):Optional<T>
- count():long
- anyMatch(Predicate<? super T>):boolean
- allMatch(Predicate<? super T>):boolean
- noneMatch(Predicate<? super T>):boolean
- findFirst():Optional<T>
- findAny():Optional<T>
- S builder():Builder<T>
- S empty():Stream<T>
- S of(T):Stream<T>
- S of(T[]):Stream<T>
- S iterate(T,UnaryOperator<T>):Stream<T>
- S generate(Supplier<T>):Stream<T>
- S concat(Stream<? extends T>,Stream<? extends T>):Stream<T>

Printing collection content using Stream API

- Print all students from a list using streams and terminal method *forEach*
  - forEach consumes source elements and do action defined with argument of type *Consumer<? super T>*
  - Note: interface List also has *forEach*, but it is not the same method

```
List<Student> students = StudentData.load();
// using anonymous class
students.stream().forEach(new Consumer<Student>() {
       @Override
       public void accept(Student t) {
              System.out.println(t);
       }
});

// using lambda
students.stream().forEach(t -> System.out.println(t));
// students.stream().forEach(System.out::println);
```

12_.../swu/oopj/streams/Example1.java

Filtering a stream

- Filter is an intermediate method that creates new stream based on a predicate

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

- Note: it just create a new stream, it does not consume data, nor it copy source content
- Can be chained with another filter method or anoother intermediate method
- An example: print students that have 40 points or more
  - A variant with lambda example is shown on the slide, a variant with anonymous classes is available in examples source code

12_.../swu/oopj/streams/Example2.java

```
students.stream()
        .filter(s -> s.getPoints() >= 40)
        .forEach(t -> System.out.println(t));
```

Note on terminal method(s)

- On a stream only one terminal method can be applied
  - Applying another terminal method on a stream that has been consumed causes exception → new stream should be created

12_.../swu/oopj/streams/Example3.java

```
List<Student> students = StudentData.load();
Stream<Student> st = students.stream();
st.forEach(t -> System.out.println(t)); //OK

//st.forEach(t -> System.out.println(t));
//causes exception: java.lang.IllegalStateException
// because stream has already been operated upon or closed

students.stream().forEach(t -> System.out.println(t)); //OK
//.stream() creates new stream
```

Sorting streams

- Sorted streams can be sorted using
  - sorted() using natural
  - sorted(Comparator<? super T>) with custom comparator
- To compare by multiple criteria composite comparator should be used as an argument
- sorted is intermediate method and it does not change the source

12_.../swu/oopj/streams/Example4.java

```
List<Student> students = StudentData.load();
//print all students with 40 or more points, sorted by last name
students.stream()
        .filter(s -> s.getPoints() > 40)
        .sorted(Student.BY_LAST_NAME)
        .forEach(t -> System.out.println(t));

//notice that source has not been changed
students.stream().forEach(t -> System.out.println(t));
```

Stateless and stateful intermediate operations

- Intermediate operations are divided into stateless and stateful operations.
- Stateless operations (e.g. filter), retain no state from previously seen element when processing a new element. Each element can be processed independently of operations on other elements.
- Stateful operations may need to process the entire input before producing a result.
  - For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream.
  - As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data.

Stream mapping

- A stream can be transformed to a new stream by mapping each object to another using provided function
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
  - Each t of type T from Stream<T> on input is mapped to R by *calling mapper.apply(t)* thus forming a Stream<R>
  - An example: Stream of students (with more than 30 points) is transformed (mapped to) stream of students' surnames
    - Later those names are stored to a new list (shown later)

12_.../swu/oopj/streams/Example5.java

```
lastNames = students.stream()
                     .filter(s -> s.getPoints() > 30)
                     .map(s -> s.getLastName())
                       ...
```

Collecting streams elements (1)

- *collect* is terminal methods which in combination with custom implementation of *Collector* interface or using built-in collectors(from class *Collectors*) copy stream elements to map, list, set, …

12_.../swu/oopj/streams/Example5.java

```
List<String> lastNames = students
                      .stream()
                      .filter(s -> s.getPoints() > 30)
                      .map(s -> s.getLastName())
                      .sorted(comp)
                      .collect(Collectors.toList());


//print new collection
lastNames.stream()
        .forEach(t -> System.out.println(t));
```

Collecting streams elements (2)

- The result of collecting depends on collector
- When working with Stream<String> an useful collector that joins all elements (strings) into new string separated by desired delimiter can be used

12_.../swu/oopj/streams/Example5b.java

```
List<Student> students = StudentData.load();
Comparator<Object> comp = Collator.getInstance(Locale.CHINA);
String lastNames = students.stream()
                           .filter(s -> s.getPoints() > 30)
                           .map(s -> s.getLastName())
                           .sorted(comp)
                           .collect(Collectors.joining(", "));

System.out.println(lastNames);
```

Mapping to stream of “primitive” types (1)

- Due to nature of Java Generics Stream<T> can contain only classes and not primitive types (i.e. Stream<Integer> vs *Stream<int>*)
  - However, when mapping function should produce Integer, Double or Long, instead of *map* method, the following methods can be used *mapToInt, mapToLong, mapToDouble* returning *IntStream, LongStream, DoubleStream*

12_.../swu/oopj/streams/Example6.java

```
students.stream()
        .filter(s -> s.getPoints() > threshold)
        .mapToInt(s -> s.getPoints()) //returns IntStream
```

Mapping to stream of “primitive” types (2)

- *IntStream*, *DoubleStream*, and *LongStream* offers additional methods, e.g. calculate min value, max value, average value, …
- *average()* is a **reduction** method (terminal method that reduce only one value) - returns an *OptionalDouble*
- *OptionalDouble* can contain *double* value (method *isPresent()*) which can be get with *getAsDouble()* that returns double or throws *NoSuchElementException* if not present
  - A better approach is to use methods *ifPresent* or *ifPresentOrElse* from *OptionalDouble*

12_.../swu/oopj/streams/Example6*.java

```
double avgGrade = students.stream()
       .filter(s -> s.getPoints() > threshold)
       .mapToInt(s -> s.getPoints())  // IntStream
       .average() //OptionalDouble
       .getAsDouble(); // double or exception throws
```

Zip file as a source stream

- Stream are not necessarily related to collection
- Streams can be created from various sources
- zip file can be a source for a stream and each element in the stream is subclass of *ZipEntry*
  - Example: Print first three lines from each txt file in zip file

12_.../swu/oopj/streams/ZipExample.java

```
try(ZipFile zip = new ZipFile(filename)){
   zip.stream()
      .filter(entry ->
                 entry.getName().toLowerCase().endsWith(".txt"))
     .forEach(entry -> write3LinesWithScanner(zip, entry));
} ...

private static void write3LinesWithScanner(
                             ZipFile zip, ZipEntry entry){
 try (Scanner sc = new Scanner(zip.getInputStream(entry),"UTF-8")) {
 ...
```