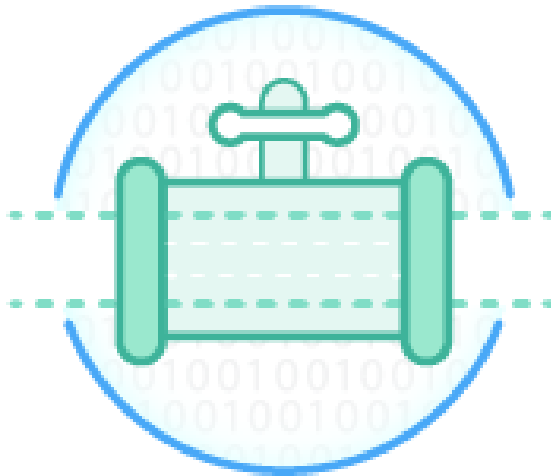




Lambdas and Streams



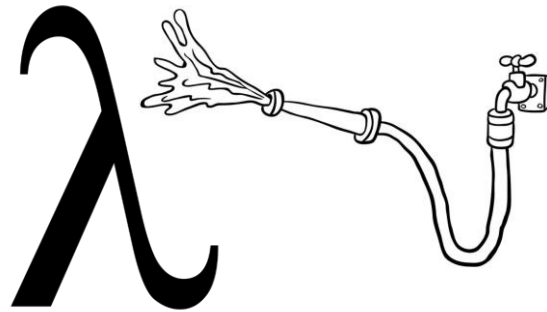
λ

Dr. Abdelkarim Erradi
CSE@QU

Content

1. Lambdas and Streams
2. Stream Operations
3. Read / Write JSON File

Lambdas and Streams



The most important principle in Programming?

KEEP IT SHORT & SIMPLE

KISS

Declarative programming using **Lambdas**
helps us to achieve KISS

Imperative vs. Declarative

Imperative Programming

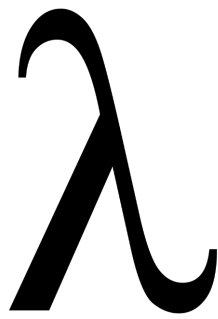
- You tell the computer **how** to perform a task.

Declarative Programming

- You tell the computer **what you want**, and you let the computer (i.e. the compiler or runtime) figure out for itself the best way to do it.
- Also known as **Functional Programming**



What is a Lambda?



- Lambda is very similar to *a method*. It has:
 - Parameters
 - A body
 - A return type
- They **don't have a name** (anonymous method)
- They have no associated object
- They **can be passed as parameters** to other methods:
 - As *code* to be executed by the receiving method
- Concise syntax (introduced in Java 8):

Parameters  Body

 Arrow token

Lambda Expressions

- Lambda expression can be passed as a parameter to methods such as *forEach*, *filter* and *map* methods :

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
numbers.forEach( e -> System.out.println(e) );
```

- Left side of **->** operator is a parameter variable.
- Right side is code to operate on the parameter and compute a result.
- When used a lambda is used with Stream or List the compiler can determine the parameter type.
- Multiple parameters are enclosed in parentheses. E.g., lambda passed to get longest word:

```
words.stream().max((v, w) -> v.length() - w.length())
```

Removal from a collection using a predicate lambda

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

//Remove elements < 4
numbers.removeIf(n -> n < 4);
```



forEach and **removeIf** can be used **directly** on a list without a stream
forEach is used to loop over the list elements

Syntax Lambda Expressions

Syntax *Parameter variables -> body*

Omit parentheses
for a single parameter.

`w -> w.length() > 10`

The body can be
a single expression.

Parameter variables

`(String w) -> w.length() > 10`

Optional parameter type

`(v, w) -> v.length() - w.length()`

These functions
have two parameters.

Use braces and
a return statement for
longer bodies.

`(v, w) ->`

`{`

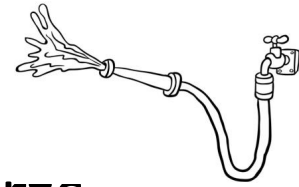
`int difference = v.length() - w.length();`

`return difference;`

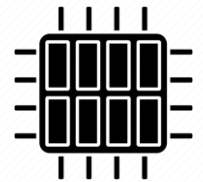
`}`



What is a Stream?



- A stream is a **sequence** of objects that supports convenient methods that can be **pipelined** to process a **list** of objects and produce the desired result
- They **don't store their own data**. They are just programmatic **wrappers** on existing data sources such as List, Array or File stream
- Support **Automatic parallelization**
 - Can split work **over multiple processors** using `stream().parallel()`
- **Lazy evaluation**: streams defer doing most operations until you actually request the results
- Streams were designed to **work well with lambdas**:



```
stream.filter( w -> w.length() > 10 )
```

Producing Streams

- Any list can be turned into a stream:

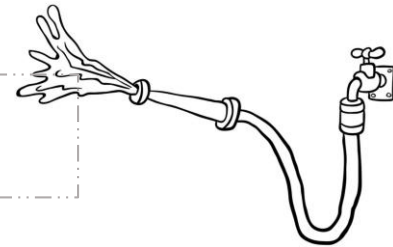
```
List<String> wordList = new ArrayList<>();  
Stream<String> words = wordList.stream();
```



© microgen/iStockphoto.

- Can create a Stream from a file

```
String filePath = "data/countries.txt";  
Stream<String> countries = Files.lines(Paths.get(filePath));
```



- You can make infinite streams:

```
Stream<Integer> integers = Stream.iterate(0, n -> n + 1);  
integers.forEach(System.out::println);
```

- You can turn any stream into a *parallel stream* using `stream().parallel()`

- Operations such as `filter` and `count` run in parallel, each processor working on chunks of the data

The Stream Concept

- Algorithm for counting words longer than 10 characters:

```
List<String> words = . . .;  
long count = 0;  
for (var w : wordList) {  
    if (w.length() > 10) { count++; }  
}
```

- With Stream library:

```
List<String> words = . . .;  
long count = words.stream().filter( w -> w.length() > 10 ).count();
```

- You tell **what** you want to achieve (filter long strings then count them) => **Declarative programming**
- You **don't** program the **how** (loop through each element in turn, if it is long, increment a counter)
- "What, not how" makes code more concise + Operations can be executed in parallel



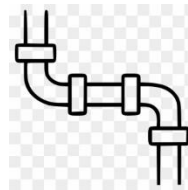
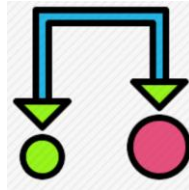
© pullia/iStockphoto.

Producing Streams

Example	Result
<code>Stream.of(1, 2, 3)</code>	A stream containing the given elements. You can also pass an array.
<code>Collection<String> coll = . . . ; coll.stream()</code>	A stream containing the elements of a collection.
<code>Files.lines(<i>path</i>)</code>	A stream of the lines in the file with the given path. Use a try-with-resources statement to ensure that the underlying file is closed.
<code>Stream<String> stream = . . . ; stream.parallel()</code>	Turns a stream into a parallel stream.
<code>Stream.generate(() -> 1)</code>	An infinite stream of ones
<code>Stream.iterate(0, n -> n + 1)</code>	An infinite stream of Integer values
<code>IntStream.range(0, 100)</code>	An IntStream of int values between 0 (inclusive) and 100 (exclusive)
<code>Random generator = new Random(); generator.ints(0, 100)</code>	An infinite stream of random int values drawn from a random generator
<code>"Hello".codePoints()</code>	An IntStream of code points of a string

Stream Operations

Filter, Map, Reduce, and others



Filter



Keep elements that satisfy a condition

// Imperative

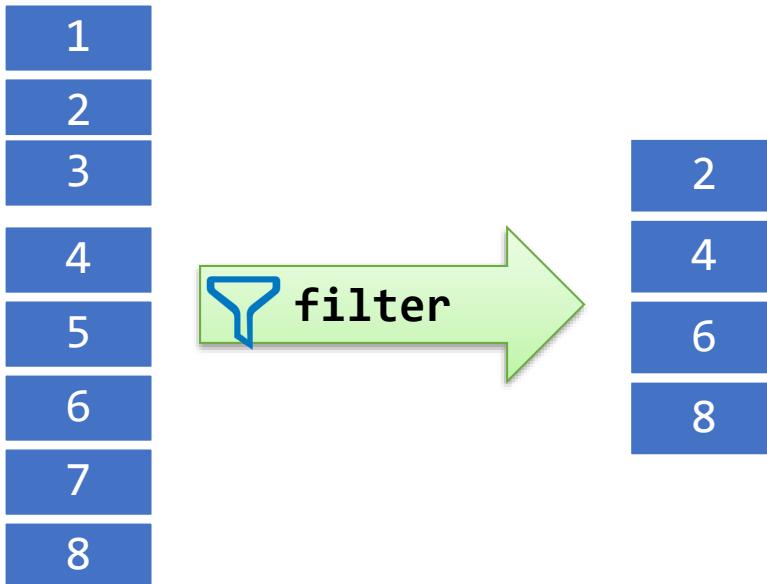
```
List<Integer> evens =  
    new ArrayList<>();
```

```
for (var num : numbers) {  
    if (num % 2 == 0) {  
        evens.add(num);  
    }  
}
```

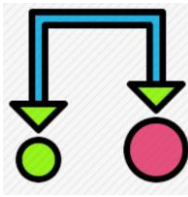
// Declarative

```
List<Integer> evens = new ArrayList<>();  
numbers.stream()  
    .filter (n -> n % 2 == 0)  
    .forEach (n -> evens.add(n));
```

forEach - Calls a Lambda
on Each Element of the Stream



Map

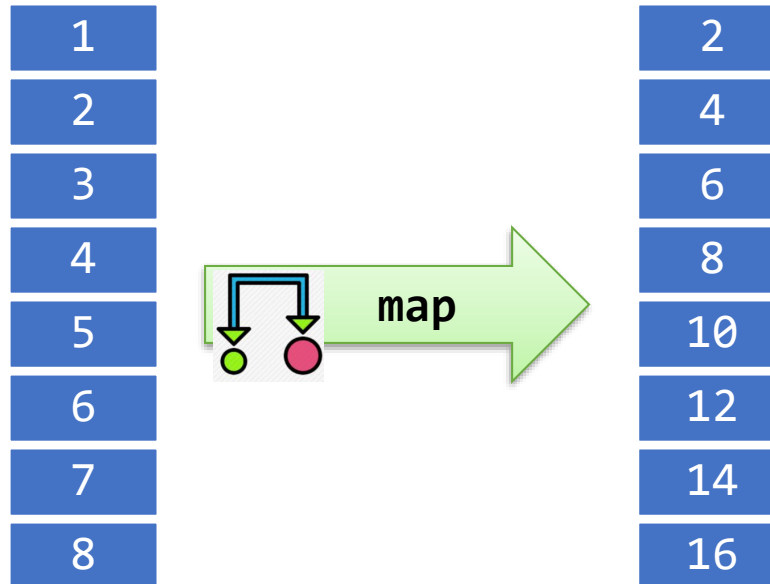


Transform elements by applying a Lambda to each element

```
// Imperative
List<Integer> doubled =
    new ArrayList<>();

for (var num : numbers) {
    doubled.add(num * 2);
}
```

```
// Declarative
Stream<Integer> doubled =
    numbers.stream()
        .map (n -> n * 2);
```



Reduce



Apply an accumulator function to each element of the list to reduce them to a single value.

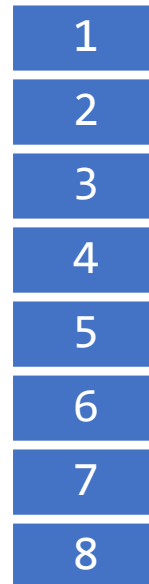
```
// Imperative
int total = 0;
for (var num : numbers) {
    total += num;
}
```

```
// Declarative
int total =
    numbers.stream()
        .reduce(0, (sum,n) -> sum + n);
```

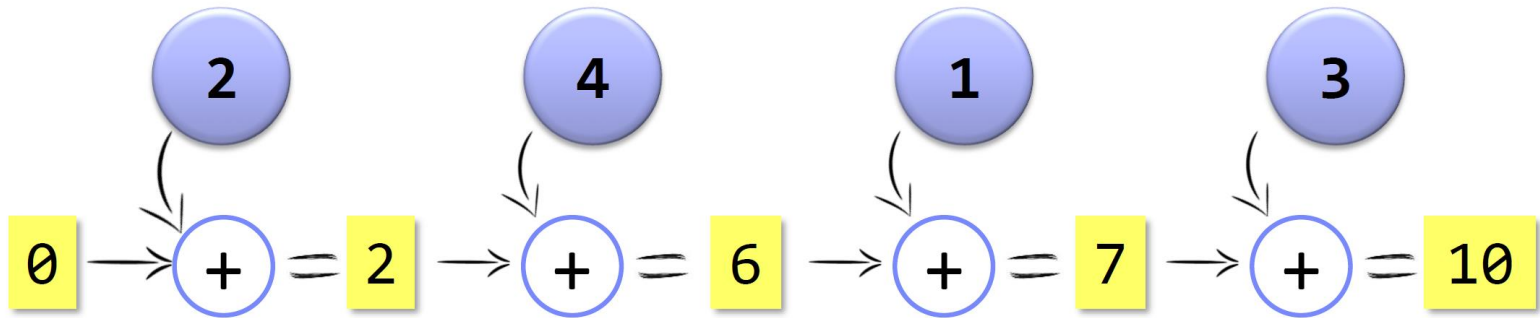
Initial value

Accumulation

Collapse the multiple elements of the input stream into a single element



Reduce



.reduce(θ , (sum,n) \rightarrow sum + n);

Reduce is **terminal** operation that yields a single value

Convenience Reducers

Sum, Average, Count, Min, Max

```
List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
int sum = nums.stream().mapToInt(Integer::intValue).sum();  
  
long count = nums.stream().mapToInt(Integer::intValue).count();  
  
double average =  
    nums.stream().mapToInt(Integer::intValue).average().orElse(0);  
  
int max = nums.stream().mapToInt(Integer::intValue).max().orElse(0);  
  
int min = nums.stream().mapToInt(Integer::intValue).min().orElse(0);
```

- They work with **int**, **long** and **double** streams only
- They are **terminal** operations that yield a single value

Flat Map

Do a map and flatten the results into 1 list

```
List<String> books =  
    students.stream()  
        .flatMap(s -> s.getBooks().stream())  
        .distinct()  
        .collect(Collectors.toList());
```

Each student has a list of books, the example above produces a list of **all** distinct books

flatMap - Each lambda application *produces a Stream*, then the Stream elements are **combined** into a single Stream.

findFirst



Return first element satisfying a condition

// Imperative

```
Integer firstEven;
```

```
for (var num : numbers) {  
    if (num % 2 == 0) {  
        firstEven = num;  
        break;  
    }  
}
```

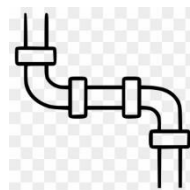
// Declarative

```
Optional<Integer> firstEven =  
    numbers.stream()  
        .filter (n -> n % 2 == 0)  
        .findFirst();
```

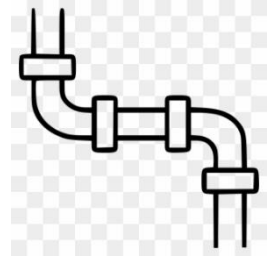
- Returns an `Optional<Integer>` for the **first entry** in the Stream. There might none, so the `Optional` could be empty.
- Filter **stops** after a single entry is found.



A pipeline of operations



Stream Operations Pipeline



- **A pipeline of operations:** a sequence of operations where the output of each operation becomes the input into the next
- Operations are either **Intermediate** or **Terminal**
- **Intermediate operations** produce a new stream as output (e.g., map, filter, ...)
 - These operations don't get processed until a terminal operation is called (**lazy evaluation!**)
- **Terminal operations** are the final operation in the pipeline (e.g., findFirst, reduce, collect, sum ...)
 - Once a terminal operation is invoked, the Stream is considered consumed and no more operations can be performed on it

Example of Lazy Evaluation

```
employees.stream()  
    .filter(e -> e != null)  
    .filter(e -> e.getSalary() > 500000)  
    .findFirst()  
    .orElse(null));
```

- **Apparent behavior**

- Check all elements for null, call getSalary on all non-null (& compare to \$500K) on all remaining, find first, return it or null

- **Actual behavior (lazy evaluation)**

- Check first element, if not null call getSalary, if salary > \$500K, return employee and exit. Otherwise repeat...
- Return null if you get to the end and never found a match

Collecting Results

- After a stream operation (e.g. `map`, `filter`) you can **harvest** the results into a `List` or `Set` using **collect**:



© Jamesmcq24/iStockphoto.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 1, 2, 3, 4, 5);  
List<Integer> evens =  
    numbers.stream().filter(e -> e % 2 == 0)  
        .collect(Collectors.toList());
```

- A stream can be collected into a single string:

```
String evens = numbers.stream()  
    .filter(e -> e % 2 == 0)  
    .map(String::valueOf)  
    .collect(Collectors.joining(", "));
```


Limiting Stream Size: **limit** and **skip**

- **limit(n)** returns a Stream of the first **n** elements
- **skip(n)** throws away the first **n** elements
- **Examples**

- Return first 10 elements

someLongStream.limit(10)

- Skip first 5 elements

someLongStream.skip(5)

Checking Matches: `anyMatch`, `allMatch`

- `anyMatch` and `allMatch` check if stream elements satisfy a boolean condition
 - **`anyMatch`** would immediately return true if it finds an element that satisfies the lambda condition
 - **`allMatch`** would immediately return false if it finds an element that fails the lambda condition
 - They stop processing once an answer can be determined
- **Examples**

```
boolean isSomeoneMegaRich = employees.stream()  
    .anyMatch(e -> e.getSalary() >= 100000);  
boolean isEveryoneMegaRich = employees.stream()  
    .allMatch(e -> e.getSalary() >= 100000);
```

Sorting

Example: Sort strings by length (longest to shortest) and then alphabetically

```
List<String> words = Arrays.asList("The quick brown fox jumps over the lazy dog".split(" "));  
  
// Sort words by word length then alphabetically  
String sortedWords = words.stream()  
    .sorted( Comparator.comparing(String::length)  
        .thenComparing(Comparator.naturalOrder())  
    )  
    .collect(Collectors.joining(" "));  
  
System.out.println( sortedWords );
```

Comparator that compares strings by their length.

Can add a secondary comparison with **thenComparing**.
Compare first by length **then** alphabetically.

Stream Operators - Summary

Example	Comments
<code>stream.filter(<i>condition</i>)</code>	A stream with the elements matching the condition.
<code>stream.map(<i>function</i>)</code>	A stream with the results of applying the function to each element.
<code>stream.mapToInt(<i>function</i>)</code> <code>stream.mapToDouble(<i>function</i>)</code> <code>stream.mapToLong(<i>function</i>)</code>	A primitive-type stream with the results of applying a function with a return value of a primitive type
<code>stream.limit(n)</code> <code>stream.skip(n)</code>	A stream consisting of the first n, or all but the first n elements.
<code>stream.distinct()</code> <code>stream.sorted()</code> <code>stream.sorted(<i>comparator</i>)</code>	A stream of the distinct or sorted elements from the original stream.

Grouping Results

- Grouping is used to split results into groups
 - E.g., Group by Continent and get the Countries count

```
Map<String, Long> countryCountByContinent =  
    countries.stream()  
        .collect(  
            Collectors.groupingBy(c -> c.getContinent(),  
                                   Collectors.counting())  
        )  
    );
```


The **Collectors.groupingBy** produces a map:


- the **key** is the Continent
- the **value** is the Countries **count** computed using **Collectors.counting()**

Other Collectors could be used such as `summingInt` and `averagingInt`

Method Expressions


- Common to have lambda expressions that just invoke a method
- Use **method expression**: *ClassName::methodName*



Equivalent to...



`(String w) -> w.toUpperCase()`


`String::toUpperCase`



Equivalent to...



`(String s, String t) -> s.compareTo(t)`


`String::compareTo`



Equivalent to...



`(double x, double y) -> Double.compare(x, y)`

`Double::compare`


Equivalent to...



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

`System.out::println`

Constructor Expressions

- Like method expression using special **new** method

≡
Equivalent to...

{
() -> new BankAccount()
BankAccount::**new**

≡
Equivalent to...

{
(n: int) -> new String[n]
String[]::**new**

```
//Convert a stream of words to an array of Strings  
String[] array = wordStream.toArray(String[]::new);
```

Read / Write JSON File



JSON Data Format

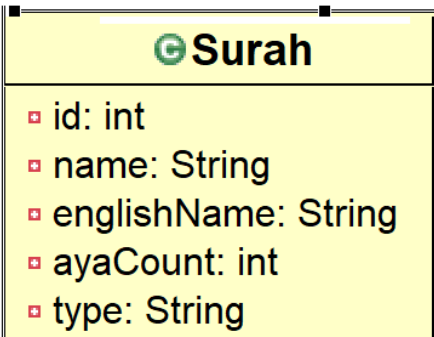
- **JSON** (JavaScript Object Notation) is a very popular **lightweight data format** to transform an object to a **text** form to ease storing and transporting data
- **Gson** library could be used to transform an object to json or transform a json string to an object

Transform an instance of Surah class to a JSON string:

```
Gson gson = new Gson();  
Surah surah = new Surah(1, "الفاتحة", "Al-Fatiha", 7, "Meccan");  
String surahJSON = gson.toJson(surah);
```



```
{  
  "id": 1,  
  "name": "الفاتحة",  
  "englishName": "Al-Fatiha",  
  "ayaCount": 7,  
  "type": "Meccan"  
}
```



Read / Write JSON file

- Read a JSON file and convert its content to objects

```
Gson gson = new Gson();  
String filePath = "data/surah.json";  
String fileContent = Files.readString(Paths.get(filePath));  
Surah[] surahs = gson.fromJson(fileContent, Surah[].class);
```

- Write objects to a JSON file

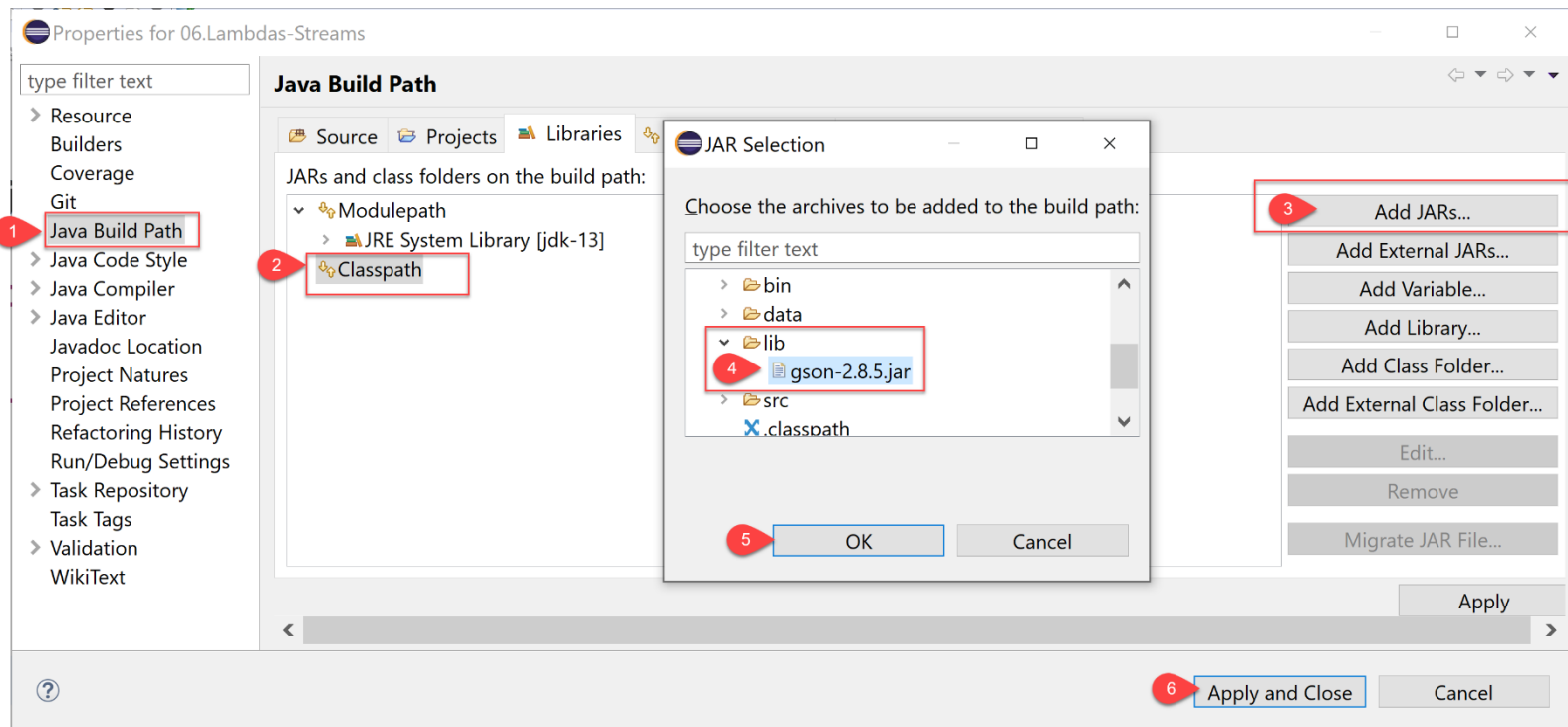
```
String surahsJSON = gson.toJson(surahs);  
Files.writeString(Paths.get(filePath), surahsJSON);
```



You may use <https://codebeautify.org/json-to-java-converter> to generate a Java class from a json string!

Steps to use Gson library

- Create a subfolder named **lib** under your project folder
- Download **Gson** library into **lib** subfolder
<https://repo1.maven.org/maven2/com/google/code/gson/gson/2.8.5/gson-2.8.5.jar>
- Write-click your project and select **Properties...**
- Select **Java Build Path**. Click **Classpath** then click **Add JARS...** select **gson-2.8.5.jar** from your project **lib** subfolder (see image below)



Summary

- To start thinking in the functional style ***avoid loops*** and instead use Streams and Lambdas
 - Widely used for list processing and GUI building to handle events
- A list can be converted to a stream for processing in a pipeline
 - Typical pipeline operations are filter, map and reduce
- Streams allow automatic parallel processing using `.stream().parallel()`
- JSON is a very popular lightweight data format to **transform an object to a text form** to ease storing and transporting data

Summary of Stream Operations

- **Make a Stream**
 - `someList.stream()`, `Stream.of(objectArray)`, `Stream.of(e1, e2...)`
- **forEach** [void output]
 - `employeeStream.forEach(e -> e.setPay(e.getPay() * 1.1))`
- **map** [outputs a Stream]
 - `numStream.map(Math::sqrt)`
- **filter** [outputs a Stream]
 - `employeeStream.filter(e -> e.getSalary() > 50000)`
- **findFirst** [outputs an Optional]
 - `stream.findFirst().get()`, `stream.findFirst().orElse(other)`
- **Collect output from a Stream**
 - `stream.collect(Collectors.toList())`
 - `stream.toArray(ClassName[]::new)`