Agenda

- Stream Programming
- File IO

Java 8 Streams

- Java 8 Stream is NOT IO streams.
- java.util.stream package.
- Streams follow functional programming model in Java 8.
- The functional programming is based on functional interface (SAM).
- Number of predefined functional interfaces added in Java 8. e.g. Consumer, Supplier, Function, Predicate, ...
- Lambda expression is short-hand way of implementing SAM -- arg types & return type are inferred.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types

1. Intermediate operations: Yields another stream.

- intermediatte operations are again classified as

    1. stateless operation
    - filter(), map(), flatMap(), limit(), skip()
    2. stateful operation
    - sorted(), distinct()

2. Terminal operations: Yields some result.

- reduce()
- forEach()for (Employee e : arr) System.out.println(e);
- collect(), toArray()
- count(), max(), min()
- Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

1. No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
2. Immutable: Any operation doesn't change the stream itself. The operations produce new stream of results.
3. Lazy evaluation: Stream is evaluated only if they have terminal operation. If terminal operation is not given, stream is not processed.
4. Not reusable: Streams processed once (terminal operation) cannot be processed again.

Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method

- Stream interface: static generate() method
- Stream interface: static iterate() method
- Stream interface: static empty() method
- nio Files class: `static Stream<String> lines(filePath)` method

Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();
// ...
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()

```
  Double arr[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
    Stream<Double> strm =  Arrays.stream(arr);
```

- Stream interface: static of() method

```
  Stream<Integer> strm = Stream.of(arr);
```

- Stream interface: static generate() method
  - generate() internally calls given Supplier in an infinite loop to produce infinite stream of elements.

```
  Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method
  - iterate() start the stream from given (arg1) "seed" and calls the given UnaryOperator in infinite loop to produce infinite stream of elements.

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method

- nio Files class: static Stream lines(filePath) method

Stream operations

- Source of elements

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh",
"Rohan", "Pradnya", "Rohan", "Pooja", "Lalita"};
```

- Create Stream and display all names

```
Stream.of(names)
    .forEach(s -> System.out.println(s));
```

- filter() -- Get all names ending with "a"
  - `Predicate<T>`: (T) -> boolean

```
Stream.of(names)
    .filter(s -> s.endsWith("a"))
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case
  - `Function<T,R>`: (T) -> R

```
Stream.of(names)
    .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order
  - String class natural ordering is ascending order.
  - sorted() is a stateful operation (i.e. needs all element to sort).

```
Stream.of(names)
    .sorted()
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order
  - `Comparator<T>`: (T,T) -> int

```
Stream.of(names)
    .sorted((x,y) -> y.compareTo(x))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
Stream.of(names)
    .skip(2)
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names
  - duplicates are removed according to equals().

```
Stream.of(names)
    .distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names
  - terminal operation: returns long.

```
long cnt = Stream.of(names)
    .count();
System.out.println(cnt);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = Stream.of(names)
        .collect(Collectors.toList());
// Collectors.toList() returns a Collector that can collect all stream
elements into a list
```

```
Set<String> set = Stream.of(names)
    .collect(Collectors.toSet());
// Collectors.toSet() returns a Collector that can collect all stream
elements into a set
```

- reduce() -- addition of 1 to 5 numbers

```
int result = Stream
    .iterate(1, i -> i+1)
```

```
        .limit(5)
        .reduce(0, (x,y) -> x + y);
```

- max() -- find the max string
    - terminal operation
    - See examples.

Collect Stream result

- Collecting stream result is terminal operation.
- Object[] toArrray()
- R collect(Collector)
    - Collectors.toList(), Collectors.toSet(), Collectors.toCollection(), Collectors.joining()
    - Collectors.toMap(key, value)

Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- IntStream class
    - IntStream.of() or IntStream.range() or IntStream.rangeClosed() or Random.ints()
    - sum(), min(), max(), average(), summaryStatistics(),
    - OptionalInt reduce().

Java IO framework

- Input/Output functionality in Java is provided under package java.io and java.nio package.
- IO framework is used for File IO, Network IO, Memory IO, and more.
- Two types of APIs are available file handling
    - FileSystem API -- Accessing/Manipulating Metadata
    - File IO API -- Accessing/Manipulating Contents/Data

Java IO

- Java File IO is done with Java IO streams.
- Java IO Streams are completly different from java.util.Stream. No relation between them
- Stream generally determines flow of data
- Java supports two types of IO streams.
    - Byte streams (binary files) -- byte by byte read/write
    - Character streams (text files) -- char by char read/write
- Stream is abstraction of data source/sink.
    - Data source -- InputStream(Byte Stream) or Reader(Char Stream)
    - Data sink -- OutputStream(Byte Stream) or Writer(Char Stream)
- All these streams are AutoCloseable (so can be used with try-with-resource construct)

Chaining IO Streams

- Each IO stream object performs a specific task.
    - FileOutputStream -- Write the given bytes into the file (on disk).

- BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
- DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
- ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutput interface.
- PrintStream -- Convert given input into formatted output.
- Note that input streams does the counterpart of OutputStream class hierarchy.

- Streams can be chained to fulfil application requirements.

Primitive types IO

- DataInputStream & DataOutputStream -- convert primitive types from/to bytes
  - primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
    - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
  - primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
    - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
  - writeUTF(String s)
  - writeInt(int i)
  - writeDouble(double d)
  - writeShort(short s)
  - ...
- interface DataInput
  - String readUTF()
  - int readInt()
  - double readDouble()
  - short readShort()
  - ...

Serialization

- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes

  - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
    - ObjectOutput interface provides method for conversion - writeObject().
  - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
    - ObjectInput interface provides methods for conversion - readObject().

- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.

- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).

- Converting (serialized) bytes back to the Java object is referred as Deserialization.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
  - writeObject(obj)
- interface ObjectInput extends DataInput
  - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.