Agenda

- Lambda Experssion Revision
- Stream Programming
- ~~Enum~~
- ~~Local and Nested Classes~~

Method Call Instructions

- Java bytecode traditionally had 4 method invocation instructions

Bytecode	Use Case
invokestatic	Static methods
invokevirtual	Instance methods (normal)
invokespecial	Constructors, private, super
invokeinterface	Interface methods

- All these require method binding at compile-time, or through fixed type hierarchies.
- invokedynamic is a bytecode instruction in the Java Virtual Machine (JVM)
- invokedynamic is different:
- It allows dynamic method invocation at runtime, unlike traditional method calls which are resolved at compile time.
- Binding is done dynamically using a bootstrap method
- When the JVM encounters invokedynamic, it uses a bootstrap method which is a special method that defines how to resolve the call. The invokedynamic instruction says:
- “I don’t know yet which method to call. JVM, please resolve this for me using the bootstrap logic.”

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().

Optional<> type

- Few stream operations yield Optional<> value.
- Optional value is a wrapper/box for object of T type or no value.
- It is safer way to deal with null values.
- It mostly helps to avoid exceptions
- To create the optional object
    - opt = Optional.of("A")
    - opt = Optional.empty() -> cretes an optional with no value
- Get value from the Optional<>:
    - optValue = opt.get(); // if you know value exists
    - optValue = opt.orElse(defValue); // if you dont know value is present or not
- Consuming Optional<> value:
    - opt.isPresent() --> boolean;
    - opt.ifPresent(consumer);

Assignment

1. String[] arr = { "Nilesh", "Shubham", "Pratik", "Omkar", "Prashant" }; count number of strings having length more than 6
2. Write a program to calculate sum of given integer array using streams. (try using reduce)