Core Java

Agenda

- Q & A
- Java 8 Streams
- JDBC

Q & A

- Set vs Map
- HashSet vs HashMap
- TreeSet -- ClassCastException
- new TreeMap(comparator)
- Overriding hashCode()

Quick Revision

- Queue interface and classes
  - Queue -- Utility data structures like Stack, Queue, Deque, PriorityQueue.
  - Queue -- offer(), poll(), peek().
    - `Queue<Integer> q = new LinkedList<>();`
  - Deque extends Queue interface
    - offerLast(), pollLast(), peekLast(), offerFirst(), pollFirst(), peekFirst().
    - Implementations of Deque: LinkedList, ArrayDeque.
    - Can used as Stack (LIFO) or Queue (FIFO).
  - PriorityQueue inherited from Queue interface
    - Internally it is binary heap data structure
    - offer(), poll(), peek().
    - Element with highest priority comes out/popped first.
    - priority decided by Comparable or Comparator (given in constructor).

        - When used with toString()/iterator/for each loop/forEach(), the output is not sorted -- The output is presenetation of binary heap.
- Set interface and classes
    - No duplication
    - HashSet -- Any order (based on hashcode)
        - Fastest
        - Duplication is decided on basis of hashCode() and equals()
        - `HashSet<K>` = `HashMap<K,null>`
    - LinkedHashSet -- Maintains order of insertion
        - Slower than HashSet
        - Duplication is decided on basis of hashCode() and equals()
        - `LinkedHashSet<K>` = `LinkedHashMap<K,null>`
    - TreeSet -- Sorted order by Comparable or given Comparator
        - Slower than LinkedHashSet
        - Duplication is decided on basis of compareTo() or compare()
        - `TreeSet<K>` = `TreeMap<K,null>`

```
// since comparator not given, elements arranged in Natural order.
// i.e. Book class must be Comparable
TreeSet<Book> set = new TreeSet<>();
set.add(new Book(...)); // if Book is not Comparable, add() will throw ClassCastException.
```

```
Comparator<Book> cmp = (x,y) -> Double.compare(x.getPrice(), y.getPrice());
TreeSet<Book> set = new TreeSet<>(cmp);
set.add(new Book(...));
```

- Map interface and classes
    - Stores key-value so that value can be searched in fastest possible time.
    - No duplicate keys allowed
    - HashMap -- Any order (based on hashcode)

        - Fastest
        - Duplication is decided on basis of hashCode() and equals() of key.
    - LinkedHashMap -- Maintains order of insertion
        - Slower than HashMap
        - Duplication is decided on basis of hashCode() and equals() of key
    - TreeMap -- Sorted order by key using Comparable or Comparator
        - Slower than LinkedHashMap
        - Duplication is decided on basis of compareTo() of key.
- If keys can be duplicated, then value should be a collection.
    - Example: Key is subject, and Value is mark"s".
        - Eng: 25, 23, 80
        - Math: 23, 22, 76

```
Map<String,List<Integer>> map = new HashMap<>();
map.put("Eng", new ArrayList<>());
List<Integer> list = map.get("Eng");
list.add(25);
list.add(23);
list.add(80);

map.put("Math", new ArrayList<>());
list = map.get("Math");
list.add(23);
list.add(22);
list.add(76);
```

    - Example: Key is userId, Value is liked quoteIds.

```
Map<Integer, Set<Integer>> map = new HashMap<>();
map.put(userId1, new HashSet<Integer>());
```

```
// userId1 likes quote1
Set<Integer> set = map.get(userId1);
set.add(quoteId1);
// userId1 likes quote2
set = map.get(userId1);
set.add(quoteId2);

// userId1 unlikes quote1
set = map.get(userId1);
set.remove(quoteId1);
```

hashCode() method

- Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).
- To use any hash-based data structure hashCode() and equals() method must be implemented.
- If two distinct objects yield same hashCode(), it is referred as collision. More collisions reduce performance.
- Most common technique is to multiply field values with prime numbers to get uniform distribution and lesser collsions.

```
class Distance {
    private int feet;
    private int inches;
    // constructor
    // getter/setters
    public boolean equals(Object oth) {
        if(oth == null)
            return false;
        if(this == oth)
            return true;
        if(!(oth instanceof Distance))
            return false;
        Distance other = (Distance)oth;
        if(this.feet != other.feet)
            return false;
        if(this.inches != other.inches)
```

```
            return false;
        return true;
    }
    public int hashCode() {
        int result = 1;
        result = 31 * result + this.feet;
        result = result + this.inches;
        return result;
    }
}
```

- hashCode() overriding rules
  - hash code should be calculated on the fields that determines equality of the object.
  - hashCode() should return same hash code each time unless object state is modified.
  - If two objects are equal (by equals()), then their hash code MUST be same.
  - If two objects are not equal (by equals()), then their hash code MAY be same (but reduce performance).

Java 8 Streams

Functional Programming

- Functions are First class Values (as equal as int or String).
  - Functions like other languages.
  - Functions as variables/objects.
  - Functions as argument or return value of function.
  - Functions defined in other function.
  - Functions as anonymous.
- Output of functions are mapped to their inputs.
  - Immutability – produce new result (instead of changing in place)
  - Function result depends solely on the input and also doesn't modify state of args or other objects.
  - Functions should be referentially transparent i.e. it should be easily replaceable by its result without changing program semantic.
  - Such functions are said to be side-effect free functions or pure functions.

- Why functional programming?
  - Developer's productivity: better reusability and modularity.
  - Simplified testing: functions without side effects.
  - Scalability: easily portable to multi-core and distributed computing (no sync. issues)
- Limitations of functional programming
  - Difficult to understand: new and complex programming paradigm for beginners.
  - Tricky: pure functions are not obvious in complex programs and tricky reusability.
  - Resource hungry: Needs more memory to store state and also more computing.
- Scope for functional programming
  - CPU and memory resources are cheaper (cloud computing).
  - Multi-core and distributed programming is common (huge data processing).
  - Popular languages are functional or support functional.
- Java functional programming is blended into object oriented programming.

Stream Introduction

- Java 8 Stream is NOT IO streams.
- java.util.stream package.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types
  - Intermediate operations: Yields another stream.
    - filter()
    - map(), flatMap()
    - limit(), skip()
    - sorted(), distinct()
  - Terminal operations: Yields some result.
    - reduce()
    - forEach()
    - collect(), toArray()
    - count(), max(), min()
  - Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

- No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
- Immutable: Any operation doesn't change the stream itself. The operations produce new stream of results.
- Lazy evaluation: Stream is evaluated only if they have terminal operation. If terminal operation is not given, stream is not processed.
- 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

Java 8 Streams

- 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.

Stream creation

- Collection interface: stream() or parallelStream()

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

- Arrays class: Arrays.stream()
- 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

```
Stream.iterate(1, i -> i+1);
```

- max() -- find the max string
  - terminal operation

```
// Optional.isPresent()
```

```
// Optional.orElse()
```

```
//Optional.ifPresent(consumer)
```

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.
- Get value in the Optional<>:
  - optValue = opt.get();
  - optValue = opt.orElse(defValue);
- Consuming Optional<> value:
  - opt.isPresent();
  - opt.ifPresent(consumer);

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().

Reference

- https://winterbe.com/posts/2014/03/16/java-8-tutorial/

Java Database Connectivity (JDBC)

- RDBMS understand SQL language only.
- JDBC driver converts Java requests in database understandable form and database response in Java understandable form.
- JDBC drivers are of 4 types
    - Type I - Jdbc Odbc Bridge driver
        - ODBC is standard of connecting to RDBMS (by Microsoft).
        - Needs to create a DSN (data source name) from the control panel.
        - From Java application JDBC Type I driver can communicate with that ODBC driver (DSN).
        - The driver class: sun.jdbc.odbc.JdbcOdbcDriver -- built-in in Java.
        - database url: jdbc:odbc:dsn
        - Advantages:
            - Can be easily connected to any database.
        - Disadvantages:
            - Slower execution (Multiple layers).
            - The ODBC driver needs to be installed on the client machine.
    - Type II - Partial Java/Native driver
        - Partially implemented in Java and partially in C/C++. Java code calls C/C++ methods via JNI.
        - Different driver for different RDBMS. Example: Oracle OCI driver.
        - Advantages:
            - Faster execution
        - Disadvantages:

        - Partially in Java (not truely portable)
        - Different driver for Different RDBMS
    - Type III - Middleware/Network driver
        - Driver communicate with a middleware that in turn talks to RDBMS.
        - Example: WebLogic RMI Driver
        - Advantages:
            - Client coding is easier (most task done by middleware)
        - Disadvantages:
            - Maintaining middleware is costlier
            - Middleware specific to database
    - Type IV
        - Database specific driver written completely in Java.
        - Fully portable.
        - Most commonly used.
        - Example: Oracle thin driver, MySQL Connector/J, ...

MySQL Programming Steps

- step 0: Add JDBC driver into project/classpath. In Eclipse, project -> right click -> properties -> java build path -> libraries -> Add external jars -> select mysql driver jar.
- step 1: Load and register JDBC driver class. These drivers are auto-registered when loaded first time in JVM. This step is optional in Java SE applications from JDBC 4 spec.

```
Class.forName("com.mysql.cj.jdbc.Driver");
    // for Oracle: Use driver class oracle.jdbc.driver.OracleDriver
```

- step 2: Create JDBC connection using helper class DriverManager.

```
// db url = jdbc:dbname://db-server:port/database
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/classwork", "root", "manager");
```

```
    // for Oracle: jdbc:oracle:thin:@localhost:1521:sid
```

- step 3: Create the statement.

```
Statement stmt = con.createStatement();
```

- step 4: Execute the SQL query using the statement and process the result.

```
String sql = "non-select query";
int count = stmt.executeUpdate(sql); // returns number of rows affected
```

  - OR

```
String sql = "select query";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()) // fetch next row from db (return false when all rows completed)
{
    x = rs.getInt("col1");      // get first column from the current row
    y = rs.getString("col2");   // get second column from the current row
    z = rs.getDouble("col3");   // get third column from the current row
    // process/print the result
}
rs.close();
```

- step 5: Close statement and connection.

```
stmt.close();
con.close();
```

MySQL Driver Download

- https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.30

SQL Injection

- Building queries by string concatenation is inefficient as well as insecure.
- Example:

```
dno = sc.nextLine();
sql = "SELECT * FROM emp WHERE deptno="+dno;
```

- If user input "10", then effective SQL will be "SELECT * FROM emp WHERE deptno=10". This will select all emps of deptno 10 from the RDBMS.
- If user input "10 OR 1", then effective SQL will be "SELECT * FROM emp WHERE deptno=10 OR 1". Here "1" represent true condition and it will select all rows from the RDBMS.
- In Java, it is recommeded NOT to use "Statement" and building SQL by string concatenation. Instead use PreparedStatement.

PreparedStatement

- PreparedStatement represents parameterized queries.

```
String sql = "SELECT * FROM students WHERE name=?";
PreparedStatement stmt = con.prepareStatement(sql);
System.out.print("Enter name to find: ");
String name = sc.next();
stmt.setString(1, name);
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    int roll = rs.getInt("roll");
    String name = rs.getString("name");
```

```
        double marks = rs.getDouble("marks");
        System.out.printf("%d, %s, %.2f\n", roll, name, marks);
    }
```

- The same PreparedStatement can be used for executing multiple queries. There is no syntax checking repeated. This improves the performance.

JDBC Tutorial (Refer after Lab time - If required)

- JDBC 1 - Getting Started : https://youtu.be/SgAVBLZ_rww
- Jdbc 2 - PreparedStatement and CallableStatement : https://youtu.be/GzSUyiep7Mw

Assignment

1. Calculate the factorial of the given number using stream operations.
2. Create a program to calculate sum of integers.
3. Create an IntStream to represent numbers from 1 to 10. Call various functions like sum(), summaryStatistics() and observe the output.
4. Implement following functionalities using JDBC for quotes application in a menu-driven program. Use PreparedStatement.
    - SignIn()
    - SignUp()
    - ChangeProfile()
    - ChangePassword()
    - displayAllQuotes()
    - displayQuotesOfUser() -- input userId from user
    - addQuote() -- input userId from user
    - editQuote()
    - deleteQuoteById()

Optional Assignment

1. In above assignment add following functions.
    - likeUnlikeQuote() -- input userId and quoteId from user
    - displayFavoriteQuotes() -- input userId from user