Agenda

- Stream API

    - collect
    - reduce

- File IO

- JDBC

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

  ```
  int result = Stream
      .iterate(1, i -> i+1)
      .limit(5)
      .reduce(0, (x,y) -> x + y);
  ```

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);

File

- File is a collection of data and information on a storage device.
- File = Data + Metadata
- collection of data/info on storage disk
- data = contents
- metadata = Information

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Used to access/manipulate metadata of the file/directory.
- Provides FileSystem APIs
  - String[] list() -- return contents of the directory
  - File[] listFiles() -- return contents of the directory
  - boolean exists() -- check if given path exists
  - boolean mkdir() -- create directory
  - boolean mkdirs() -- create directories (child + parents)
  - boolean createNewFile() -- create empty file
  - boolean delete() -- delete file/directory
  - boolean renameTo(File dest) -- rename file/directory
  - String getAbsolutePath() -- returns full path (drive:/folder/folder/...)
  - String getPath() -- return path
  - File getParentFile() -- returns parent directory of the file
  - String getParent() -- returns parent directory path of the file
  - String getName() -- return name of the file/directory
  - static File[] listRoots() -- returns all drives in the systems.
  - long getTotalSpace() -- returns total space of current drive
  - long getFreeSpace() -- returns free space of current drive
  - long getUsableSpace() -- returns usable space of current drive
  - boolean isDirectory() -- return true if it is a directory
  - boolean isFile() -- return true if it is a file
  - boolean isHidden() -- return true if the file is hidden
  - boolean canExecute()
  - boolean canRead()
  - boolean canWrite()
  - boolean setExecutable(boolean executable) -- make the file executable
  - boolean setReadable(boolean readable) -- make the file readable

- boolean setWritable(boolean writable) -- make the file writable
- long length() -- return size of the file in bytes
- long lastModified() -- last modified time
- boolean setLastModified(long time) -- change last modified time

transient fields

- writeObject() serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".
- The transient and static fields (except serialVersionUID) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then InvalidClassException will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each write() operation on FileOutputStream will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- BufferedOutputStream classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
  - Java object --> ObjectOutputStream --> BufferedOutputStream --> FileOutputStream --> file on disk.
- Data is sent to underlying stream when buffer is full or flush() called explicitly.
- BufferedInputStream provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

PrintStream class

- Produce formatted output (in bytes) and send to underlying stream.
- Formatted output is done using methods print(), println(), and printf().
- System.out and System.err are objects of PrintStream class.
- It is used only to write the formatted data in to the file.

Scanner class

- Added in Java 5 to get the formatted input.
- It is java.util package (not part of java io framework).

```
Scanner sc = new Scanner(inputStream);
// OR
Scanner sc = new Scanner(inputFile);
```

- Helpful to read text files line by line.

Character streams

- Character streams are used to interact with text file.
- Java char takes 2 bytes (unicode), however char stored in disk file may take 1 or more bytes depending on char encoding.
    - https://www.w3.org/International/questions/qa-what-is-encoding
- The character stream does conversion from java char to byte representation and vice-versa (as per char encoding).
- The abstract base classes for the character streams are the Reader and Writer class.
- Writer class -- write operation
    - void close() -- close the stream
    - void flush() -- writes data (in memory) to underlying stream/device.
    - void write(char[] b) -- writes char array to underlying stream/device.
    - void write(int b) -- writes a char to underlying stream/device.
- Writer Sub-classes
    - FileWriter, OutputStreamWriter, PrintWriter, BufferedWriter, etc.
- Reader class -- read operation
    - void close() -- close the stream
    - int read(char[] b) -- reads char array from underlying stream/device
    - int read() -- reads a char from the underlying device/stream. Returns -1
- Reader Sub-classes
    - FileReader, InputStreamReader, BufferedReader, etc.

Java NIO

- Java NIO (New IO) is an alternative IO API for Java.
- Java NIO offers a different IO programming model than the traditional IO APIs.
- Since Java 7.
- Java NIO enables you to do non-blocking (not fully) IO.
- Java NIO consist of the following core components:
    - Channels e.g. FileChannel, ...
    - Buffers e.g. ByteBuffer, ...
    - Selectors
- Java NIO also provides "helper" classes Paths & Files.
    - exists()
    - ...

Paths and Files

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

- Files class (Files) provides several static methods for manipulating files in the file system.

```
static InputStream newInputStream(Path, OpenOption...) throws IOException;
static OutputStream newOutputStream(Path, OpenOption...) throws IOException;
static DirectoryStream<Path> newDirectoryStream(Path) throws IOException;
static Path createFile(Path, attribute.FileAttribute<?>...) throws
IOException;
static Path createDirectory(Path, attribute.FileAttribute<?>...) throws
IOException;
static void delete(Path) throws IOException;
static boolean deleteIfExists(Path) throws IOException;
static Path copy(Path, Path, CopyOption...) throws IOException;
static Path move(Path, Path, CopyOption...) throws IOException;
static boolean isSameFile(Path, Path) throws IOException;
static boolean isHidden(Path) throws IOException;
static boolean isDirectory(Path, LinkOption...);
static boolean isRegularFile(Path, LinkOption...);
static long size(Path) throws IOException;
static boolean exists(Path, LinkOption...);
static boolean isReadable(Path);
static boolean isWritable(Path);
static boolean isExecutable(Path);
static List<String> readAllLines(Path) throws IOException;
static Stream<String> lines(Path) throws IOException;
```

Channels and Buffers

- All IO in NIO starts with a Channel.
- A Channel is similar to IO stream.
- From the Channel data can be read into a Buffer.
- Data can also be written from a Buffer into a Channel.

NIO Channels

- Java NIO Channels are similar to IO streams with a few differences:
  - You can both read and write to a Channels. Streams are typically one-way (read or write).
  - Channels can be read and written asynchronously (non-blocking).
  - Channels always read to, or write from, a Buffer.
- Channel Examples
  - FileChannel

  - DatagramChannel // UDP protocol
  - SocketChannel, ServerSocketChannel // TCP protocol

NIO Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.
- Using a Buffer to read and write data typically follows this 4-step process:
  - Write data into the Buffer
  - Call buffer.flip()
  - Read data out of the Buffer
  - Call buffer.clear() or buffer.compact()
- Buffer Examples
  - ByteBuffer
  - CharBuffer
  - DoubleBuffer
  - FloatBuffer
  - IntBuffer
  - LongBuffer
  - ShortBuffer

Channel and Buffer Example

```
RandomAccessFile aFile = new RandomAccessFile("somefile.txt", "rw");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(32);

int bytesRead = inChannel.read(buf); // write data into buffer (from channel)
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip(); // switch buffer from write mode to read mode

    while(buf.hasRemaining()){
        System.out.print((char) buf.get()); // read data from the buffer
    }

    buf.clear(); // clear the buffer
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

RandomAccessFile

- RandomAccessFile class from java.io package.
- Capable of reading and writing into a file (on a storage device).
- Internally maintains file read/write position/cursor.

- Homework: Read docs.

Java NIO vs Java IO

- IO: Stream-oriented
- NIO: Buffer-oriented
- IO: Blocking IO
- NIO: Non-blocking IO

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

1. 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.

2. 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

3. 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

4. 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.

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

MySQL Driver Download

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j/8.1.0

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 concepts

java.sql.Driver

- Implemented in JDBC drivers.
- MySQL: com.mysql.cj.jdbc.Driver
- Oracle: oracle.jdbc.OracleDriver
- Postgres: org.postgresql.Driver
- Driver needs to be registered with DriverManager before use.
- When driver class is loaded, it is auto-registered (Class.forName()).
- Driver object is responsible for establishing database "Connection" with its connect() method.
- This method is called from DriverManager.getConnection().

java.sql.Connection

- Connection object represents database socket connection.
- All communication with db is carried out via this connection.
- Connection functionalities:
  - Connection object creates a Statement.
  - Transaction management.

java.sql.Statement

- Represents SQL statement/query.
- To execute the query and collect the result.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(selectQuery);
int count = stmt.executeUpdate(nonSelectQuery);
```

- Since query built using string concatenation, it may cause SQL injection.

java.sql.PreparedStatement

- Inherited from java.sql.Statement.
- Represents parameterized SQL statement/query.
- The query parameters (?) should be set before executing the query.
- Same query can be executed multiple times, with different parameter values.
- This speed up execution, because query syntax checking is done only once.

```
PreparedStatement stmt = con.prepareStatement(query);
stmt.setInt(1, intValue);
stmt.setString(2, stringValue);
stmt.setDouble(3, doubleValue);
stmt.setDate(4, dateObject); // java.sql.Date
stmt.setTimestamp(5, timestampObject); // java.sql.Timestamp

ResultSet rs = stmt.executeQuery();
// OR
int count = stmt.executeUpdate();
```

java.sql.ResultSet

ResultSet represents result of SELECT query. The result may have one/more rows and one/more columns. Can access only the columns fetched from database in SELECT query (projection).

```
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
  int id = rs.getInt("id");
  String quote = rs.getString("quote");
  Timestamp createdAt = rs.getTimestamp("created_at"); // java.sql.Timestamp
// ...
}
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
  int id = rs.getInt(1);
  String quote = rs.getString(2);
  Timestamp createdAt = rs.getTimestamp(3); // java.sql.Timestamp
// ...
}
```