Core Java

Agenda

- Q & A
- Multi-threading
  - Thread life cycle
  - Synchronization
  - Inter-thread communication
- Garbage collection

Q & A

- DbUtil
  - static final fields -- DB_DRIVER, DB_URL, DB_USER, DB_PASSWORD.
  - static block -- Class.forName(DB_DRIVER);
  - public static -- Connection getConnection()
    - return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
- DAO -- Data Access Object
  - All JDBC related code for a table(s) is written under a single class.
  - DAO is independent of application type (console, gui, web, service, ...) -- reusable across any application type.
  - DAO methods -- CRUD operations + specific operations
    - Arguments -- Pojo objects or Primitive types
    - Return -- Pojo objects or Pojo collection or update count
  - DAO fields -- Connection object, PreparedStatement objects
    - Initialized in constructor.
      - con = DbUtil.getConnection();
      - stmt1 = con.prepareStatement("sql ? ?");

        - Closed in close() method -- Optionally overridden from AutoCloseable
            - stmt1.close();
            - con.close();

- Map
    - Key-Value entries.
    - Faster searching for given key.

```
// search quote for given quote id
Map<Integer,Quote> map = new HashMap<>();
map.put(q.getId(), q);

// search user for given user id
Map<Integer,User> map = new HashMap<>();
map.put(u.getId(), u);
```

```
String sql = "SELECT q.id qid, q.quote, q.author, q.created_at, u.id uid, u.first_name, u.last_name,
u.email, u.passwd, u.mobile FROM quotes q JOIN users u ON q.user_id=u.id";
stmt = con.prepareStatement(sql);
// search user for given quote
Map<Quote,User> map = new HashMap<>();
    // key = Quote -- It must have hashCode() and equals() overridden -- based on quote id

ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    int quoteId = rs.getInt("qid");
    String quote = rs.getString("quote");
    String author = rs.getString("author");
    long time = rs.getTimestamp("created_at").getTime();
    Quote q = new Quote(quoteId, quote, author, new Date(time));
```

```
        int userId = rs.getInt("uid");
        String firstName = rs.getString("first_name");
        String lastName = rs.getString("last_name");
        String email = rs.getString("email");
        String passwd = rs.getString("passwd");
        String mobile = rs.getString("mobile");
        User u = new User(userId, firstName, lastName, email, passwd, mobile);

        map.put(q, u);
    }
```

- Stream operations
    - Sum of all even numbers between 1 to 10.

```
Stream<Integer> strm = Stream.iterate(1, i->i+1).limit(10);
Integer total = strm
    .filter(i -> i % 2 == 0)
    .reduce(0, (a,i)->a+i);
System.out.println(total);
```

```
Stream<Integer> strm = Stream.iterate(1, i->i+1).limit(10);
int total = strm
    .mapToInt(i -> i)       // Stream<Integer> --> IntStream
    .filter(i -> i % 2 == 0)
    .sum();
System.out.println(total);
```

```
IntStream strm = IntStream.rangeClosed(1,10);
int total = strm
```

```
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println(total);
```

- Set to List

```
Set<Integer> set = new HashSet<>();
Collections.addAll(set, 11, 22, 43, 55, 66, 77, 88);
List<Integer> list = new ArrayList<>();
list.addAll(set);
```

```
Set<Integer> set = new HashSet<>();
Collections.addAll(set, 11, 22, 43, 55, 66, 77, 88);
List<Integer> list = set.stream().collect(Collectors.toList());
```

Multi-Threading

Thread life cycle

- Thread.State state = th.getState();
- NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
  - NEW: New thread object created (not yet started its execution).
  - RUNNABLE: Thread is running on CPU or ready for execution. Scheduler picks ready thread and dispatch it on CPU.
  - BLOCKED: Thread is waiting for lock to be released. Thread blocks due to synchronized block/method.
  - WAITING: Thread is waiting for the notification. Waiting thread release the acquired lock.
  - TIMED_WAITING: Thread is waiting for the notification or timeout duration. Waiting thread release the acquired lock.
  - TERMINATED: Thread terminates when run() method is completed, stopped explicitly using stop(), or an exception is raised while executing run().

Synchronization

- When multiple threads try to access same resource at the same time, it is called as Race condition.
- Example: Same bank account undergo deposit() and withdraw() operations simultaneously.
- It may yield in unexpected/undesired results.
- This problem can be solved by Synchronization.
- The synchronized keyword in Java provides thread-safe access.
- Java synchronization internally use the Monitor object associated with any object. It provides lock/unlock mechanism.
- "synchronized" can be used for block or method.
- It acquires lock on associated object at the start of block/method and release at the end. If lock is already acquired by other thread, the current thread is blocked (until lock is released by the locking thread).
- "synchronized" non-static method acquires lock on the current object i.e. "this". Example:

```
class Account {
    // ...
    public synchronized void deposit(double amount) {
        double newBalance = this.balance + amount;
        this.balance = newBalance;
    }
    public synchronized void withdraw(double amount) {
        double newBalance = this.balance - amount;
        this.balance = newBalance;
    }
}
```

- "synchronized" static method acquires lock on metadata object of the class i.e. MyClass.class. Example:

```
class MyClass {
    private static int field = 0;
    // called by incThread
    public synchronized static void incMethod() {
        field++;
    }
    // called by decThread
```

```
    public synchronized static void decMethod() {
        field--;
    }
}
```

- "synchronized" block acquires lock on the given object.

```
// assuming that no method in Account class is synchronized.

// thread1
synchronized(acc) {
    acc.deposit(1000.0);
}

// thread2
synchronized(acc) {
    acc.withdraw(1000.0);
}
```

- Alternatively lock can be acquired using RentrantLock since Java 5.0. Example code:

```
class Example {
    private final ReentrantLock rl = new ReentrantLock();
    public void method() {
        rl.lock();
        try {
            // ...
        }
        finally {
            rl.unlock();
        }
```

```
        }
    }
```

- Synchronized collections
    - Synchronized collections (e.g. Vector, Hashtable, ...) use synchronized keyword (block/method) to handle race conditions.

Inter-thread communication

- wait()
    - Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
    - The current thread must own this object's monitor i.e. wait() must be called within synchronized block/method.
    - The thread releases ownership of this monitor and waits until another thread notifies.
    - The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- notify()
    - Wakes up a single thread that is waiting on this object's monitor.
    - If multiple threads are waiting on this object, one of them is chosen to be awakened arbitrarily.
    - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
    - This method should only be called by a thread that is the owner of this object's monitor.
- notifyAll()
    - Wakes up all threads that are waiting on this object's monitor.
    - The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.
    - This method should only be called by a thread that is the owner of this object's monitor.

Garbage collection

- Garbage collection is automatic memory management by JVM.
- If a Java object is unreachable (i.e. not accessible through any reference), then it is automatically released by the garbage collector.
- An object become eligible for GC in one of the following cases:
    - Nullify the reference.

```

```

```
MyClass obj = new MyClass();
obj = null;
```

- Reassign the reference.

```
MyClass obj = new MyClass();
obj = new MyClass();
```

- Object created locally in method.

```
void method() {
    MyClass obj = new MyClass();
    // ...
}
```

- Island of isolation i.e. objects are referencing each other, but not referenced externally.

```
class FirstClass {
    private SecondClass second;
    public void setSecond(SecondClass second) {
        this.second = second;
        second.setFirst(this);
    }
}
class SecondClass {
    private FirstClass first;
    public void setFirst(FirstClass first) {
        this.first = first;
    }
```

```
}
class Main {
    public static void method() {
        FirstClass f = new FirstClass();
        f.setSecond(new SecondClass());
        f = null;
    }
    // ...
}
```

- GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects.
- Before object is destroyed, its finalize() method is invoked (if present).
- One should override this method if object holds any resource to be released explicitly e.g. file close, database connection, etc.

```
class MyClass {
    private Connection con;
    public MyClass() throws Exception {
        con = DriverManager.getConnection("url", "username", "password");
    }
    // ...
    @Override
    public void finalize() {
        try {
            if(con != null)
                con.close();
        }
        catch(Exception e) {
        }
    }
}
class Main {
    public static void method() throws Exception {
        MyClass my = new MyClass();
        my = null;
```

```
            System.gc(); // request GC
        }
        // ...
    }
```

- GC can be requested (not forced) by one of the following.
    - System.gc();
    - Runtime.getRuntime().gc();
- GC is of two types i.e. Minor and Major.
    - Minor GC: Unreferenced objects from young generation are reclaimed. Objects not reclaimed here are moved to old/permanent generation.
    - Major GC: Unreferenced objects from all generations are reclaimed. This is unefficient (slower process).
- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html