Core Java

Agenda

- Java 8 interfaces
- Functional Interface
- Anonymous Inner class
- Lambda expression
- JDBC
  - JDBC Driver
  - Statement
  - SQL Injection
  - PreparedStatement
  - DAO classes

Java 8 Interfaces

Default methods

- Java 8 allows default methods in interfaces. If method is not overridden, its default implementation in interface is considered.
- This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparator, …

```
interface Emp {
    double getSal();
    default double calcIncentives() {
        return 0.0;
    }
}
class Manager implements Emp {
    // ...
    // calcIncentives() is overridden
```

```
    double calcIncentives() {
        return getSal() * 0.2;
    }
}
class Clerk implements Emp {
    // ...
    // calcIncentives() is not overridden -- so method of interface is considered
}
```

```
new Manager().calcIncentives(); // return sal * 0.2
new Clerk().calcIncentives(); // return 0.0
```

- However default methods will lead to ambiguity errors as well, if same default method is available from multiple interfaces. Error: Duplicate method while declaring class.
- Superclass same method get higher priority. But super-interfaces same method will lead to error.
    - Super-class wins! Super-interfaces clash!!

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FirstClass implements Displayable, Printable { // compiler error: duplicate method
    // ...
}
class Main {
```

```
    public static void main(String[] args) {
        FirstClass obj = new FirstClass();
        obj.show();
    }
}
```

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class Superclass {
    public void show() {
        System.out.println("Superclass.show() called");
    }
}
class SecondClass extends Superclass implements Displayable, Printable {
    // ...
}
class Main {
    public static void main(String[] args) {
        SecondClass obj = new SecondClass();
        obj.show(); // Superclass.show() called
    }
}
```

- A class can invoke methods of super interfaces using InterfaceName.super.

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FourthClass implements Displayable, Printable {
    @Override
    public  void show() {
        System.out.println("FourthClass.show() called");
        Displayable.super.show();
        Printable.super.show();
    }
}
class Main {
    public static void main(String[] args) {
        FourthClass obj = new FourthClass();
        obj.show(); // calls FourthClass method
    }
}
```

Functional Interface

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, …
- @FunctionalInterface annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
@FunctionalInterface    // okay
interface Foo {
```

```
    void foo();     // SAM
}
```

```
@FunctionalInterface    // okay
interface FooBar1 {
    void foo();     // SAM
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // error
interface FooBar2 {
    void foo();     // AM
    void bar();     // AM
}
```

```
@FunctionalInterface    // error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // okay
interface FooBar4 {
    void foo();         // SAM
    public static void bar() {
        /*... */
    }
}
```

- Functional interfaces forms foundation for Java lambda expressions and method references.

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
  - `Predicate<T>`: test: T -> boolean
  - `Function<T,R>`: apply: T -> R
  - `BiFunction<T,U,R>`: apply: (T,U) -> R
  - `UnaryOperator<T>`: apply: T -> T
  - `BinaryOperator<T>`: apply: (T,T) -> T
  - `Consumer<T>`: accept: T -> void
  - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction, ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
};
Arrays.sort(arr, cmp);
```

```
// Anonymous object of Anonymous inner class.
Arrays.sort(arr, new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
});
```

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However, code is complex to read and un-efficient to execute.
- Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.

- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

```
// Anonymous inner class
Arrays.sort(arr, new Comparator<Emp>() {
    public int compare(Emp e1, Emp e2) {
        int diff = e1.getEmpno() - e2.getEmpno();
        return diff;
    }
});
```

```
// Lambda expression -- multi-liner
Arrays.sort(arr, (Emp e1, Emp e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- multi-liner -- Argument types inferred
Arrays.sort(arr, (e1, e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- single-liner -- with block { ... }
Arrays.sort(arr, (e1, e2) -> {
    return e1.getEmpno() - e2.getEmpno();
});
```

```
// Lambda expression -- single-liner
Arrays.sort(arr, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- Practically lambda expressions are used to pass as argument to various functions.
- Lambda expression enable developers to write concise code (single liners recommended).

Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y;
    System.out.println("Result: " + res)
}
```

- In functional programming, such functions/lambdas are referred as pure functions.

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2; // must be effectively final
BinaryOperator<Integer> op = (a,b) -> a + b + c;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y + c;
    System.out.println("Result: " + res);
}
```

- Here variable c is bound (captured) into lambda expression. So it can be accessed even out of scope (effectively). Internally it is associated with the method/expression.
- In some functional languages, this is known as Closures.

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/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 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

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
        // ...
    }
```

Quick Revision

Statements

- interface Statement: executing SQL queries
  - Drawback: Prepare queries by String concatenation. May cause SQL injection.
- interface PreparedStatement extends Statement: executing parameterized SQL queries
  - Prevent SQL injection
  - Efficient execution if same query is to be executed repeatedly.
- interface CallableStatement extends PreparedStatement: executing stored procedures in db -- will be discussed in next class.
  - Prevent SQL injection
  - More efficient execution if same query is to be executed repeatedly.

Executing statements

- Load and register class. In JDBC 4, this step is automated in Core Java applications (provided class is available in classpath).

```
static {
    try {
        Class.forName(DB_DRIVER);
    }
    catch(Exception ex) {
        ex.printStackTrace();
        System.exit(0);
    }
}
```

- Executing SELECT statements

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
    String sql = "SELECT * FROM students WHERE marks > ?";
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.setDouble(1, marks);
        try(ResultSet rs = stmt.executeQuery()) {
            while(rs.next()) {
                int roll = rs.getInt("roll");
                String name = rs.getString("name");
                double smarks = rs.getDouble("marks");
                Student s = new Student(roll, name, marks);
                System.out.println(s);
            }
        } // rs.close()
    } // stmt.close()
} // con.close()
catch(Exception ex) {
    ex.printStackTrace();
}
```

- Executing non-SELECT statements

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
    String sql = "DELETE FROM students WHERE marks > ?";
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.setDouble(1, marks);
        int count = stmt.executeUpdate();
        System.out.println("Rows Deleted: " + count);
    } // stmt.close()
} // con.close()
catch(Exception ex) {
```

```
        ex.printStackTrace();
    }
```

Assignment

1. Write a menu driven program to do following operations on candidates table using JDBC PreparedStatement.
    - Insert new candidate
    - Display all candidates
    - Increment votes of candidate with given id
    - Delete candidate with given id
    - Find candidate of given id
    - Find candidates of given party
    - Display total votes for each party
2. Write a menu driven program to do following operations on users table using JDBC PreparedStatement.
    - Insert new user (Voter)
    - Display all users
    - Delete voter with given id
    - Change status of voter with given id to true
    - Change the name and birth date of voter
3. Use following method to count number of strings with length > 6 in given array.

```
public static int countIf(String[] arr, Predicate<String> cond) {
    int count = 0;
    for(String str: arr) {
        if(cond.test(str))
            count++;
    }
    return count;
}
```

```
public static void main(String[] args) {
    String[] arr = { "Nilesh", "Shubham", "Pratik", "Omkar", "Prashant" };
    // call countIf() to count number of strings have length more than 6 -- using anonymous inner class
    int cnt = countIf(arr, new Predicate<String>() {
        public boolean test(String s) {
            return s.length() > 6;
        }
    });
    System.out.println("Result: " + cnt); // 2

    // call countIf() to count number of strings have length more than 6 -- using lambda expressions
}
```

4. Create a functional `interface Arithmetic` with single abstract method `double calc(double,double)`. Write a static method calculate() in main class as follows. In main(), write a menu driven program that inputs two numbers from the user and calls calculate() method with appropriate lambda expression (in arg3) to perform addition, subtraction, multiplication and division operations.

```
static void calculate(double num1, double num2, Arithmetic op) {
    double result = op.calc(num1, num2);
    System.out.println("Result: "+result);
}
```

5. Create a functional `interface Check<T>` with single abstract method `boolean compare(T x, T y)`. Create a static method in main class to test array elements `static <T> int countIf(T[] arr, T key, Check<T> c)`. This method should return count of elements in the array for which given check is satisfied. The check will be given as lambda expression. Example call to countIf() from main() will be as follows.

```
Integer [] arr = {44, 77, 99, 22, 55, 66};
Integer key = 50;
int cnt = countIf(arr, key, (x,y)-> x > y);
System.out.println("Count = " + cnt); // 4 (because 4 elements in array are greater than given key i.e. 50)
```

6. In above assignment, create one more array of Double (constant values) where few elements are repeated. Input a key from user and check how many times key is repeated in the array using appropriate lambda expression.