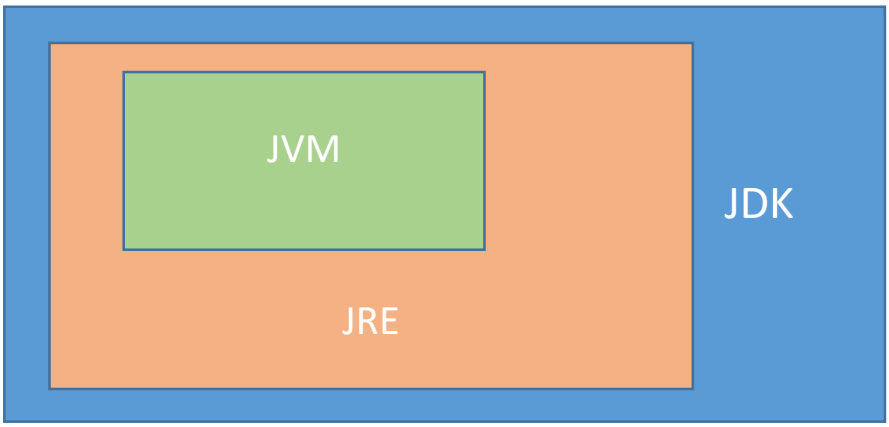
JAVA

Java Architecture:



JVM:

- Does not physical exist
- It provides run time environment where java bytecode can be executed
- It performs tasks like load, verify & execute the code

JRE:

- It contains set of software tools that are used to develop java applications
- It is implementation of JVM
- It will interpret all the keywords
- It physically exists
- It also contains set of libraries & other files that are used by JVM

JDK:

- Software development environment that is used to develop java applications
- It physically exists
- It contains the JRE + additional development tools
- It also has the compiler

DataTypes:

Primitive: Int, char, float, Boolean, double, byte, long, short

Non-primitive: Array, String, Classes, Interfaces

Operators:

Unary: +, -, ++, --

Arithmetic: +, -, *, /, %

Relational: <, >, <=, >=, ==, !=

Logical: &, |, ^

Assignment: =, +=, -=, *=, /=

Ternary: ? :

Conditional Statement.. (IF):

```
If (condition) {
//set of statements
}

If (condition) {
//set of statements1
} else {
//set of statements2
}

If (condition1) {
//set of statements1
} else if (condition2) {
//set of statements2
}} else if (condition3) {
//set of statements3
} else {
//set of statements2
}
```

Nested if:

```
If (condition) {
        If (condition) {
```

```
        //set of statements
        }
} else {
        If (condition) {
                //set of statements1
        } else {
                //set of statements2
        }
}
```

Conditional Statement.. (SWITCH):

```
Switch (expression) {
Case value1:
        //set of statements1
        Break;
Case value2:
        //set of statements2
.
.
.

Default:
        //set of statementsD
}
```

Loop Statements:

```
For(initialization; condition; incr/decr) {
        //set of statements
}

While (condition) {
```

//set of statements

}

Do {

//set of statements

} While (condition)

Access Modifier:

Public: it can be accessed anywhere in the project

Protected: can be accessed within the package only & even outside package by using sub-class

Private: can be accessed only in a class

Default: can be accessed only within the package only

Example 1:

```
package ex1_JavaBasics;

public class ArrayExample {

    public static void main(String[] args) {

        int arr[] = {12, 23, 44, 56, 78};

        int count = arr.length;

        System.out.println("Total numbers in array
: " + count);

//      for(int i=0; i<count; i++){
//          System.out.println(arr[i]);
//      }

        for(int k:arr){
            System.out.println(k);
        }
```

```
    }

}
```

Example 2:

```
package ex1_JavaBasics;

public class SwitchExample {

    public static void main(String[] args) {

        int number = 40;

        switch(number){
        case 10:
            System.out.println("Ten");
            break;
        case 20:
            System.out.println("Twenty");
            break;
        case 30:
            System.out.println("Thirty");
            break;
        default:
            System.out.println("Not in 10,20 or
30");
        }
    }

}
```

Creation of Object:

1. By new keyword
2. By newInstance() method

Examples:

Main within Class:

```
package ex1_JavaBasics;

public class Student1 {

    int id;
    String name;

    public static void main(String[] args) {

        Student1 s1 = new Student1();
        System.out.println(s1.id);
        System.out.println(s1.name);
    }

}
```

Main outside the Class:

```
package ex1_JavaBasics;

public class Student {

    int id;
    String name;
}
```

```
package ex1_JavaBasics;

public class TestStudent {
```

```
    public static void main(String[] args) {

        Student s1 = new Student();
        System.out.println(s1.id);
        System.out.println(s1.name);

    }

}
```

Object Initialization thru reference

```
package ex1_JavaBasics;

public class Student {

    int id;
    String name;
}
```

```
package ex1_JavaBasics;

public class TestStudent {

    public static void main(String[] args) {

        Student s1 = new Student();
        s1.id = 101;
        s1.name = "John";
        System.out.println(s1.id);
        System.out.println(s1.name);

    }

}
```

Object Initialization by Method:

```
package ex1_JavaBasics;

public class Student {

    int id;
    String name;

    void insertRecord(int a, String b){
        id = a;
        name = b;
    }

    void displayInfo(){
        System.out.println(id + " " + name);
    }
}
```

```
package ex1_JavaBasics;

public class TestStudent {

    public static void main(String[] args) {

        Student s1 = new Student();
        Student s2 = new Student();
        s1.insertRecord(111, "Jack");
        s2.insertRecord(222, "Robert");
        s1.displayInfo();
        s2.displayInfo();
    }

}
```

Example for Inheritance:

Class 1

```
package ex2_Inheritance;

public class Father {

    int asset = 10000;

    void education(){
        System.out.println("Father is a Doctor");
    }
}
```

Class 2

```
package ex2_Inheritance;

public class Son extends Father{

    int age = 35;

    void education(){
        System.out.println("Son is an Engineer");
    }

//  public static void main(String[] args) {
//
//      Son s = new Son();
//      System.out.println("Father asset is : " +
s.asset);
//      s.education();
//      System.out.println("Age of son : " +
s.age);
//
//      Father f = new Father();
//      f.education();
//  }
```

```
}
```

Class 3:

```
package ex2_Inheritance;

public class GrandChild extends Son{

    int standard = 5;

    void education(){
        System.out.println("Grandchild is a
student");
    }

    public static void main(String[] args) {

        GrandChild gc = new GrandChild();
        System.out.println("Grandfather asset :"  +
gc.asset);
        System.out.println("Age of father : " +
gc.age);
        gc.education();
        System.out.println("GrandChild standard : "
+ gc.standard);

    }

}
```

Example for Polymorphism - Overloading:

Class 1

```
package ex3_Polymorphism;

public class Adder {

    static int add(int a, int b){
```

```
        return a + b;
    }

    static int add(int a, int b, int c){
        return a + b + c;
    }

    static float add(float a, float b){
        return a + b;
    }
}
```

Class 2

```
package ex3_Polymorphism;

public class TestAdder {

    public static void main(String[] args) {

        System.out.println(Adder.add(10, 15));
        System.out.println(Adder.add(10, 15, 20));
        System.out.println(Adder.add(12.5f,
10.5f));
    }

}
```

Example for Polymorphism - Overriding:

Class 1

```
package ex3_Polymorphism;

public class Vehicle {

    void run(){
        System.out.println("Vehicle is running");
    }
```

```
}
```

Class 2

```
package ex3_Polymorphism;

public class Bike extends Vehicle{

    void run(){
        System.out.println("Bike is running safe");
    }

    public static void main(String[] args) {

        Bike b = new Bike();
        b.run();
    }

}
```

Example for Abstraction:

Class 1:

```
package ex4_Abstraction;

abstract class Bike {

    abstract void run();
}
```

Class 2:

```
package ex4_Abstraction;

public class Honda extends Bike{

    void run(){
        System.out.println("Honda Bike is Black
color");
```

```
    }

    public static void main(String[] args) {

        Bike b = new Honda();
        b.run();

    }

}
```

Example for Encapsulation:

Class 1:

```
package ex5_Encapsulation;

public class Student {

    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

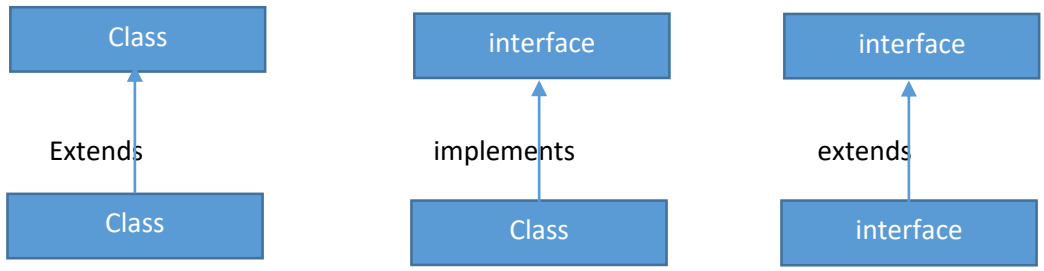
Class 2:

```
package ex5_Encapsulation;

public class TestStudent {

    public static void main(String[] args) {
```

```
        Student s = new Student();
        s.setName("Rick");
        System.out.println("Name of student : " +
s.getName());
    }

}
```

Interface:

- It is a blueprint of a class or a skeleton
- It is a mechanism to achieve abstraction
- It has static constants & abstract methods only
- Interface fields are public, static & final by default
- Interface methods are public & abstract
- It consists of rules not properties
- It is invoked using “implements” keyword
- When a method is invoked from interface in a class, it’s always an “override”
- A class that implements an interface must implement all the methods declared in the interface



Example for Interface:

Interface:

```
package ex6_Inteface;

public interface Vehicle {

    void run();

    float amt();

    static int square(int x){
```

```
        return x * x;
    }
}
```

Implementation Class 1:

```
package ex6_Inteface;

class Car implements Vehicle{

    public void run(){
        System.out.println("Car runs in 80km
speed");
    }

    public float amt(){
        return 15.3f;
    }
}
```

Implementation Class 2:

```
package ex6_Inteface;

class Truck implements Vehicle{

    public void run(){
        System.out.println("Truck runs at 50km
speed");
    }

    public float amt(){
        return 25.6f;
    }
}
```

Class 3:

```java
package ex6_Inteface;

public class Speed {

    public static void main(String[] args) {

        Vehicle v1 = new Car();
        v1.run();
        System.out.println(v1.amt());

        Vehicle v2 = new Truck();
        v2.run();

        System.out.println("Result of square :  " +
Vehicle.square(6));
    }

}
```

Collections:

- It is a framework that provides an architecture to store & manipulate set of data.
- It also stores & manipulates group of objects.
- It allows all the operations performed on data such as search, sort, insert, delete, etc
- It is a single unit of objects
- It is an Interface
- It provides an in-built iterator

Three types of Collection Interfaces:

1. List
2. Set
3. Map

List:

- ✓ It is a type of data structure that can store an ordered collection of data
- ✓ It uses index like array & also uses internal memory reference
- ✓ It auto scales itself while a data is added or removed

- ✓ It allows duplicate values
- ✓ Memory is allocated for 10 values initially

List is implemented by three classes:

(i) ArrayList
(ii) LinkedList
(iii) Vector

ArrayList:

- ➢ It uses dynamic array to store data
- ➢ It maintains the insertion order
- ➢ It is non-synchronized
- ➢ Data can be randomly accessed
- ➢ Insertion & retrieval is easier
- ➢ Deletion is easier but expensive
- ➢ It resizes by growing half of its initial size
- ➢ It offers better performance

Linked List:

- ➢ It uses doubly-linked list to store elements
- ➢ It maintains insertion order
- ➢ It is not synchronized
- ➢ Addition is easier but retrieval is not easier
- ➢ Deletion is easier & cheaper too
- ➢ Manipulation is fast because no shifting is needed

Vector:

- ➢ It is similar to ArrayList but synchronized
- ➢ It uses many legacy methods that are not part of collections framework
- ➢ It re-sizes by doubling its initial size
- ➢ Performance is poor

Example for ArrayList:

```
package ex7_Collections;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
```

```java
public class ArrayListExample {

    public static void main(String[] args) {

        List<String> nameList = new ArrayList<>();

        nameList.add("john");
        nameList.add("hi");
        nameList.add("jack");
        nameList.add("rick");
        nameList.add("Jacob");
//      nameList.add("john1");
//      nameList.add("hi1");
//      nameList.add("jack");
//      nameList.add("rick1");
//      nameList.add("Jacob");
//      nameList.add("rose");

        System.out.println("Size of list : " +
nameList.size());

//      nameList.remove(2);
//      System.out.println("Size of list after
removal: " + nameList.size());

//      Boolean b = nameList.contains("rick");
//      Boolean b = nameList.contains("ri");
//      System.out.println(b);

//      nameList.add(2, "rose");

//      System.out.println(nameList);

//      for(int i=0; i<nameList.size(); i++){
//          System.out.println(nameList.get(i));
//      }
```

```
        ListIterator<String> iterator =
nameList.listIterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }

    }

}
```

Exercises:

1. Practice the same above example with other methods of ArrayList
2. Do the above example in LinkedList

MAP:

- ✓ It is a type of data structure that can store unordered collection of data
- ✓ It uses key (like an index), but this key may be of any data type
- ✓ Key should be unique
- ✓ It can store any data type against the key
- ✓ Retrieval is by using the key
- ✓ It allows duplicate values
- ✓ Deletion is easier

Map is implemented by three classes:

(i) HashMap
(ii) LinkedHashMap
(iii) TreeMap

Map(key, value)

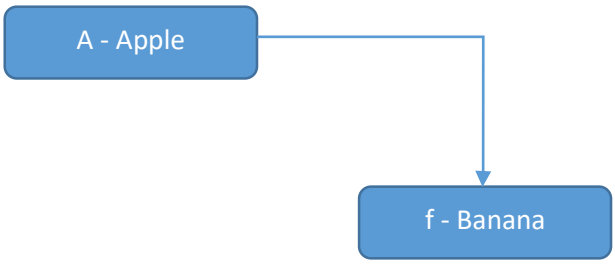
Example for Map:

```
package ex7_Collections;

import java.util.Map;
```

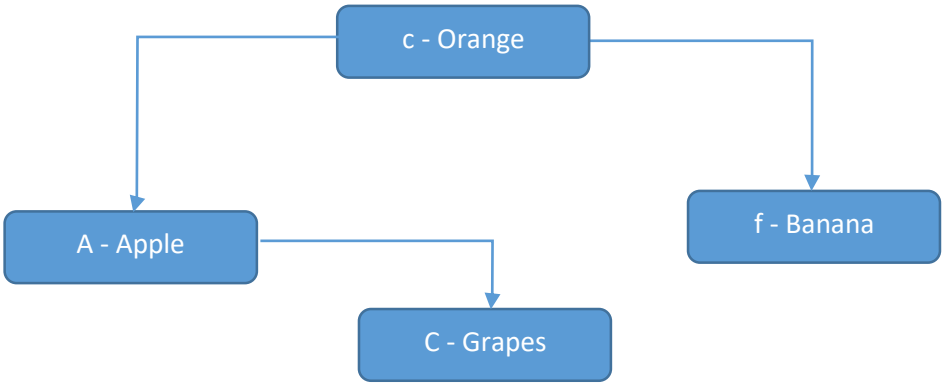
```java
import java.util.TreeMap;

public class MapExample {

    public static void main(String[] args) {

        Map<Character, String> fruitMap = new
TreeMap<>();

        fruitMap.put('A', "Apple");
        fruitMap.put('f', "Banana");
        fruitMap.put('c', "Orange");
        fruitMap.put('C', "Grapes");

        System.out.println("Size of Map : " +
fruitMap.size());

//      System.out.println("Value in Map : " +
fruitMap.get('c'));
//
//      System.out.println("Given key exists : " +
fruitMap.containsKey('a'));
//      System.out.println("Given value exists : "
+ fruitMap.containsValue("Orange"));
//      System.out.println("Is map empty : " +
fruitMap.isEmpty());
//
//      fruitMap.remove('c');

//      fruitMap.remove('f');

        for(String s : fruitMap.values()){
            System.out.println(s);
        }
    }

}
```

Order in which data stored in Map:

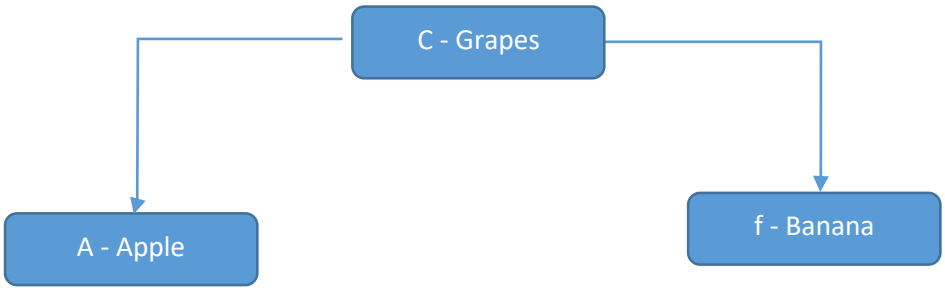
After adding first 2 values:



After adding first 4 values:



After adding removing root values:



Example to use Collections Class:

```
package ex7_Collections;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ArrayListCopy {
```

```
    public static void main(String[] args) {

        List<String> srcList = new ArrayList<>();
        srcList.add("john");
        srcList.add("jack");
        srcList.add("jill");
        srcList.add("jacob");

        List<String> destList = new ArrayList<>();
        destList.add("aaa");
        destList.add("bbb");
        destList.add("ccc");
        destList.add("ddd");

        System.out.println("Source List : " +
srcList);
        System.out.println("Dest List : " +
destList);

        Collections.copy(destList, srcList);

        System.out.println("After copying from
source to dest : ");
        System.out.println("Source List : " +
srcList);
        System.out.println("Dest List : " +
destList);

    }
}
```

Exercises:

1. Practice the above examples once
2. Do the same Map example for other methods
3. Do the same Map example for HashMap
4. Identify & practice other important Collections class methods as I mentioned.

Java I/O:

- I/O is used to process the input & produce the output
- Java.io package contains all the classes needed for i/o operations
- File handling is also performed with Java I/O API

Stream:

It is a sequence of data that are composed by byte

1. System.out
2. System.in
3. System.err

Two types:

1. Output Stream
2. Input Stream

Output Stream:

- To write data to a destination (may be file, array, peripheral device, etc)

OutputStream Class – an abstract class; superclass of all classes

FileOutputStream Class – use for writing data to a file

BufferedOutputStream Class – uses buffer to store data & write to file

Input Stream:

- To read data from a source (may be a file, array, peripheral device, etc)

InputStream class – an abstract class; super class of all classes

FileInputStream class – used to obtain bytes from a file

BufferedInputStream class – used to read information from stream; internally uses buffer to store data

Example for FileOutputStream:

```
package ex8_FileHandling;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
```

```java
import java.io.IOException;

public class FileOutputStreamExample {

    public static void main(String[] args) throws
IOException {

        FileOutputStream fout = new
FileOutputStream("E:\\Selenium\\Programs\\CSDQEA24S
D1234_Java\\Files\\testout.txt");

        fout.write(71);

        String s = "This is Java session";

        byte b[] = s.getBytes();

        fout.write(b);
        fout.close();
        System.out.println("File written
successfully");
    }

}
```

Example for BufferedFileOutputStream:

```java
package ex8_FileHandling;

import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class BufferedOutputStreamExample {
```

```
    public static void main(String[] args) throws
IOException {

        FileOutputStream fout = new
FileOutputStream("E:\\Selenium\\Programs\\CSDQEA24S
D1234_Java\\Files\\testout.txt");
        BufferedOutputStream bout = new
BufferedOutputStream(fout);

        String s = "This is File Handling";
        byte b[] = s.getBytes();
        bout.write(b);
        bout.flush();
        bout.close();
        fout.close();
        System.out.println("File written
successfully");
    }

}
```

Example for FileInputStream:

```
package ex8_FileHandling;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FileInputStreamExample {

    public static void main(String[] args) throws
IOException {

        FileInputStream fin = new
FileInputStream("E:\\Selenium\\Programs\\CSDQEA24SD
1234_Java\\Files\\testout.txt");
```

```
//        int i = fin.read();
//        System.out.println(i);

          int i=0;
          while((i=fin.read())!=-1){
              System.out.print((char)i);
          }
          fin.close();
      }

}
```

Example for BufferedFileInputStream:

```
package ex8_FileHandling;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BufferedInputStreamExample {

    public static void main(String[] args) throws
IOException {

          FileInputStream fin = new
FileInputStream("E:\\Selenium\\Programs\\CSDQEA24SD
1234_Java\\Files\\testout.txt");
          BufferedInputStream bin = new
BufferedInputStream(fin);

          int i;
          while((i=bin.read())!=-1){
              System.out.print((char)i);
          }
```

```
            bin.close();
            fin.close();
        }

}
```

JDBC:

- Java DataBase Connectivity
- It is a Java API to connect & execute the query with the database
- JDBC API uses JDBC drivers to connect with the database
- Save, update, delete & fetch data are some operations performed with database thru JDBC
- Import the java.sql package

Set of interfaces used by JDBC API:

- Driver Interface
- Connection Interface
- Statement Interface
- ResultSet Interface
- RowSet Interface

Set of classes used by JDBC API:

- DriverManager class
- Type class
- BLOB class
- CLOB class

Following activities are performed:

1. Connect with the database
2. Execute queries to fetch data
3. Update statements to the database
4. Retrieve the result received from database

Steps to connect java program with database using JDBC:

1. Register the Driver class – forName()
2. Create connection – getConnection()
3. Create statement – createStatement()
4. Execute statement – executeQuery()
5. Close the connection – close()

Connectivity procedure:

1. Import the driver class
2. Use the connection URL along with database name
3. Give username
4. Give Password

How to load jar file for driver?

i. Download jdbc driver for mysql
ii. Store it in a folder
iii. Map it to the program suing “Build Path”

Pre-requisite:

- ✓ Install MySQL or any other database
- ✓ Have DB & Table created with data

Example 1 (Statement Interface):

```
package ex9_JDBC;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCExample1 {

    public static void main(String[] args) throws
ClassNotFoundException, SQLException {

        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost
:3306/sampledb", "root", "root1");

        Statement st = con.createStatement();
//      ResultSet rs = st.executeQuery("select *
from student");

//      while(rs.next()){
```

```
//          System.out.println(rs.getInt(1) + " " +
rs.getString(2));
//      }
//      con.close();

//      st.executeUpdate("insert into student
values(113, 'James', 'BTech', 'SSCollege')");
//      System.out.println("One record inserted");

//      int result = st.executeUpdate("update
student set course = 'MBA' where studentid=113");
//      System.out.println(result + " records
updated");

        int result = st.executeUpdate("delete from
student where studentid=113");
        System.out.println(result + " records
deleted");

    }

}
```

Example 2 (PreparedStatement Interface):

```
package ex9_JDBC;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCExample2 {

    public static void main(String[] args) throws
ClassNotFoundException, SQLException {
```

```
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost
:3306/sampledb", "root", "root1");

//      PreparedStatement prst =
con.prepareStatement("insert into student
values(?,?,?,?)");
//      prst.setInt(1, 114);
//      prst.setString(2, "Bill");
//      prst.setString(3, "MBA");
//      prst.setString(4, "KKCollege");
//      int i = prst.executeUpdate();
//      System.out.println(i + " records
inserted");

        PreparedStatement prst =
con.prepareStatement("update student set college=?
where studentid=?");
        prst.setString(1, "RRCollege");
        prst.setInt(2, 114);
        int i = prst.executeUpdate();
        System.out.println(i + " records updated");
    }

}
```

Example 3 (ResultSetMetaData & DatabaseMetaData):

```
package ex9_JDBC;

import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
```

```
import java.sql.SQLException;

public class JDBCExample3 {

    public static void main(String[] args) throws
SQLException, ClassNotFoundException {

        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost
:3306/sampledb", "root", "root1");

//      PreparedStatement ps =
con.prepareStatement("select * from student");
//      ResultSet rs = ps.executeQuery();
//      ResultSetMetaData rsmd = rs.getMetaData();
//      System.out.println("Total columns : " +
rsmd.getColumnCount());
//      System.out.println("Column name of 1st
column : " + rsmd.getColumnName(1));
//      System.out.println("Column Type of 2nd
column : " + rsmd.getColumnTypeName(2));

        DatabaseMetaData dbmd = con.getMetaData();
        System.out.println("Driver Name : " +
dbmd.getDriverName());
        System.out.println("User Name : " +
dbmd.getUserName());
        System.out.println("Database Product Name :
" + dbmd.getDatabaseProductName());
    }

}
```

Constructor:

- A block of codes similar to method
- Special type of method that is used to initialize the object
- It is called when an instance of class is created
- A constructor initializes the object when it is created
- At time of calling constructor, memory for the object is allocated
- It is called “Constructor” because it constructs values at the time of object creation
- It is not necessary to have a constructor in a class

Rules:

1. Constructor name should be same as class name
2. It will not have an explicit return type
3. Need not be abstract, static or final

Example 1: Default Constructor:

```
package ex10_Constructor;

public class Bike {

    Bike(){
        System.out.println("Bike is created");
    }

    public static void main(String[] args) {

        Bike b = new Bike();

    }

}
```

Example 2: Default Constructor with default values:

```
package ex10_Constructor;

public class Student {
```

```
    int id;
    String name;

    void displayInfo(){
        System.out.println( id + " " + name);
    }
}
```

```
package ex10_Constructor;

public class TestStudent {

    public static void main(String[] args) {

        Student s = new Student();
        s.displayInfo();

    }

}
```

Example 3: Parametrized Constructor:

```
package ex10_Constructor;

public class Student {

    int id;
    String name;

    Student(int a, String b){
        id = a;
        name = b;
    }

    void displayInfo(){
```

```
        System.out.println( id + " " + name);
    }

    public static void main(String[] args) {

        Student s1 = new Student(111,"John");
        s1.displayInfo();
        Student s2 = new Student(222,"Jacob");
        s2.displayInfo();    }
}
```

Example 4: Constructor Overloading:

```
package ex10_Constructor;

public class Student {

    int id;
    String name;
    int age;

    Student (int a){
        id = a;
    }

    Student (String b){
        name=b;
    }

    Student(int a, String b){
        id = a;
        name = b;
    }

    Student(int a, String b, int c){
        id = a;
        name = b;
```

```
        age = c;
    }

    void displayInfo(){
        System.out.println( id + " " + name + " " +
age);
    }

    public static void main(String[] args) {

        Student s1 = new Student(111);
        s1.displayInfo();
        Student s2 = new Student(222,"Jacob");
        s2.displayInfo();
        Student s3 = new Student("John");
        s3.displayInfo();
        Student s4 = new Student(333,"Mike",20);
        s4.displayInfo();
    }
}
```

Example 5: Copy Constructor:

```
package ex10_Constructor;

public class Student {

    int id;
    String name;

    Student(int a, String b){
        id = a;
        name = b;
    }

    Student(Student s){
        id = s.id;
```

```
        name = s.name;
    }

    void displayInfo(){
        System.out.println( id + " " + name);
    }

    public static void main(String[] args) {

        Student s1 = new Student(111, "John");
        Student s2 = new Student(s1);
        s1.displayInfo();
        s2.displayInfo();

    }
}
```

Example 6: Empty Constructor or Copying values without Constructor

```
package ex10_Constructor;

public class Student {

    int id;
    String name;

    Student(int a, String b){
        id = a;
        name = b;
    }

    Student(){}

    void displayInfo(){
        System.out.println( id + " " + name);
    }
```

```
    public static void main(String[] args) {

        Student s1 = new Student(111, "John");
        Student s2 = new Student();
        s2.id = s1.id;
        s2.name = s1.name;
        s1.displayInfo();
        s2.displayInfo();

    }
}
```

Lambda Expressions:

- It provides implementation of “functional interface”
- “Functional Interface” – is an interface that has only one abstract method
- It provides a clear & concise way to represent “Functional Interface” by using an expression
- Better uses in Collection library
- It helps to iterate, filter & extract data.
- It save a lot of code
- Don’t need to define a separate method for providing implementation

Syntax:

(argument-list)->{body}

Argument-list : It can be empty or non-empty

->: Arrow-token (used to link the argument-list and body of expression)

Body – contains the expressions & statement as lambda expression

No Parameter:

() -> {

//Body of no parameter lambda expression

}

One Parameter Lambda:

(p1) -> {

//Body of single parameter lambda expression

}

Two Parameter Lambda:

(p1, p2) -> {

//Body of multiple parameter lambda expression

}

Example 1: Simple Lambda

```
package ex11_LambdaExp;

interface Student {

    public void show();
}

package ex11_LambdaExp;

public class LambdaExample {

    public static void main(String[] args) {

        int id = 101;

        Student s1=()->{
            System.out.println("Student id : " +
id);
        };
        s1.show();
    }

}
```

Example 2: Without Parameter

```
package ex11_LambdaExp;

interface Student {

    public String show();
}
```

```
package ex11_LambdaExp;

public class LambdaExample {

    public static void main(String[] args) {

        Student s1=()->{
            return "Student registered
successfully";
        };

        System.out.println(s1.show());
    }

}
```

Example 3: Single Parameter

```
package ex11_LambdaExp;

interface Student {

    public String show(String name);
}
```

```java
package ex11_LambdaExp;

public class LambdaExample {

    public static void main(String[] args) {

        //with function parantheses
        Student s1=(name)->{
            return "Student name is " + name;
        };
        System.out.println(s1.show("John"));

        //without function parantheses
        Student s2=name->{
            return "Student name is " + name;
        };
        System.out.println(s2.show("Jack")); }

}
```

Example 4: multiple parameter

```java
package ex11_LambdaExp;

interface Student {

    int marks(int m1, int m2);
}
```

```java
package ex11_LambdaExp;

public class LambdaExample {

    public static void main(String[] args) {
```

```
        //without data type / return keyword
        Student s1=(a,b)->(a+b);
        System.out.println("Total marks of Student1
: " + s1.marks(75, 80));

        //with data type & without return keyword
        Student s2=(int a, int b)->(a+b);
        System.out.println("Total marks of Student2
: " + s2.marks(75, 75));

        //with data type & return keyword
        Student s3=(int a, int b)->{
            return(a+b);
        };
        System.out.println("Total marks of Student3
: " + s2.marks(75, 70));
    }

}
```

Example 5: Without Functional Interface & For-Each loop

```
package ex11_LambdaExp;

import java.util.ArrayList;
import java.util.List;

public class LambdaForLoop {

    public static void main(String[] args) {

        List<String> list = new
ArrayList<String>();
        list.add("John");
        list.add("Jack");
        list.add("Mike");
```

```
        list.add("Jill");

        list.forEach(
                (n)->System.out.println(n)
            );
    }

}
```

Example 6: Streams & Lambda

```
package ex11_LambdaExp;

public class Student1 {

    int id;
    String name;
    String course;

    public Student1(int id, String name, String
course){
        super();
        this.id = id;
        this.name = name;
        this.course = course;
    }
}
```

```
package ex11_LambdaExp;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class LambdaStream {
```

```
    public static void main(String[] args) {

        List<Student1> studList = new
ArrayList<Student1>();
        studList.add(new Student1(111,"Jack",
"BE"));
        studList.add(new Student1(112,"Mike",
"BTech"));
        studList.add(new Student1(113,"John",
"ME"));

        Stream<Student1> strData =
studList.stream()
                .filter(p -> p.id <113);

        strData.forEach(student ->
        System.out.println(student.name + " & " +
student.course)
        );
    }

}
```