Cmd 🡪 location of pom.xml file

Cmd 🡪 mcn clean install.

* Invokes the parent constructor through the child constructor.
* super( ) should be the first line of code in the child constructor .
* super( ) is optional , even you don’t define . It is generated by default.
* If we create a parenthesized constructor, it is mandatory to create the non-parenthesized constructor.

//CHILD

**public** **class** Child **extends** Parent {

**public** Child() {

**super**(5); //Invoking the parent constructor. super() should be the first line of code

System.***out***.println("In Child");

}

}

//PARENT

**public** **class** Parent {

**int** variable;

**public** Parent() {

System.***out***.println("In Parent");

}

**public** Parent(**int** variable) {

**super**();

**this**.variable = variable;

System.***out***.println(variable);

}

}

* Major advantage of inheritance is: we can access the grand parent’s methods, parent methods, child’s methods through the child instance.
* Multi-Level Inheritance :

A 🡪 B (A extends B)

B 🡪 C (B extends C)

C 🡪 Object (C extends object)

* No multiple inheritance :

A 🡪 B and A🡪 C not possible.

* **INTERFACE:** It’s a collection of variables but each variable is **public static final variable**.
  + Interface is a collection of one or more abstract method / pure abstract method.
  + Cannot have constructor.
  + Interface cannot be instantiated
  + If two interfaces have same abstract class signature , only one needs to be overridden
* **Why interface?**
  + Interface is introduced in the java to achieve multiple inheritance as the java does not support it
  + When we want common behaviour.
  + For Abstraction – Hiding the implementation details of the class.
* **How to use?**
  + ClassName extends ClassName2 { }
  + ClassName implements Interface1 { }
  + ClassName implements Interface1 , Interface2 { }
  + ClassName extends ClassName2 implements Interface1 ,Interface2 { }
  + After creating an instance , create the class name which implements that interface with the suffix “Impl” //best coding practice
    - if Interface is “Calculator” , class name must be “CalculatorImpl”
* **Syntax of defining an Interface?**
  + Interface should start with capital letter

**package** com.model;

**public** **interface** Calculator {

}

* + Interface <InterfaceName> {

Public static final int VARIABLE = 456; // by default all the variables are public static final constants

Public static final String VARIABLE = “Hello”;

Public static final Boolean VARIABLE = true;

Public abstract Void Behaviour(); // by default the methods are public abstract

Public abstract Void Behaviour1();

Public abstract Void Behaviour2 (); // method declaration, no definition, no body

}

* **STATIC?**
  + Keyword : Reserved word
  + Only one copy is created
  + Memory is pre created.
  + Static variables and methods are pre created by the JVM even before the object variables gets instantiated.
  + Static can be at the variable level and
    - Ex: static int var;
  + static can be at the method level or
    - Ex: static int add(){return 10;)
  + it can be at the block level
    - Ex :static { }

**public** **class** StaticDemoMain {

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

StaticDemo staticDemo = **new** StaticDemo();//create object

staticDemo.intVariable = 5;

staticDemo.display();

StaticDemo staticDemo2 = **new** StaticDemo();

staticDemo2.intVariable = 13456;

System.***out***.println(StaticDemo.*STATICVAR*); // class name variable

StaticDemo.*display2*(); // class name variable

}

**static** {

System.***out***.println("static block");

}

}

**Output:** static block //executed even before main

non static function

66

static function

* Where to use static block?
  + When there is a common data for all instances ( new)
  + Where developer does not want to create an object.
  + static can be at the class level variable
    - EX : <ClassName>.staticVariable;

Instance variable – first instantiate then call the variable

Employee emp = new Employee ();

Emp.intVar = 20;

* **@override:** this is a meta information to the compiler. It is an annotation. It is an interface.
* **INTERFACE?**
  + Interface is a key word. It cannot be instantiated.
* **POLYMORPHISM?** 
  + Static – method overload
  + Runtime – method override (extends)
* **Method overload:** if a class has two or more functions have same name with different arguments/parameters and return type does not matter in overload , then its k/as Method overloading.
* **Method Override:** Override is possible only in inheritance/Interface i.e., (extends/ implements)
  + The parent class has the method signature and the subclass also has the same method signature of the parent.
  + The return type does matters in method override.
  + Parent can have lower or equal access modifier.
  + But child should have equal or greater access modifier.
* **Protected:** It is an access modifier.
  + Represented by # in the class diagram
  + Protected would act like private to other classes but public to inherited class.
* **MARKER INTERFACE:** An interface which does not have any abstract method. It’s an empty interface (no variables, no methods)
* **Extending Interface :**  An interface can extend another interface.
  + Public interface A {

//abstract method of A

}

Public interface C {

//abstract methods of C

}

Public interface B extends A, C {

//includes Abstract methods of A and C,

//also methods of B

}

* ***STRING*** : String is a Class. (predefined class)
  + ***Default package : java.lang.\* ;***
  + ***Specific : java.lang.String;***
  + ***Import java.lang.\* ;*** //prefer specific locationrather than default
  + java.lang.String; //preferable
* Two types :
  + String object
    - String str = new String (“Hello”);
    - Str = null;
  + String literal :
    - Ex : String str = “Hello”; //created in the string pool
* Strings are immutable (fixed == Constant)
* Cmd -> javap java.lang.String
* Update/Alter/Modify the string data
* **StringBuffer:**

**24 March 2022**

**STATIC DATA AND DYNAMIC DATA:**

* Static data = data is hard coded. Int a = 10;// not advised
* Dynamic data = data is taken from the user.

Consuming data from the Console (Command line argument):

Cmd > Javac FileName.java // javac – java compiler, FileName.java = source File/src

This command generates Byte Code i.e., Filename. Class //not human or machine readable.

Cmd> java FileName // JVM will read the .class file and convert it into low level language/machine level language

Cmd < java FileName arg1 arg2 //arguments passed on run time

Class FileName {

Public static void main (String [ ] args){ //read data from console as array of strings [] from args

SOP{args[0]); // arg1

SOP{args[1]); //arg2

}

In Eclipse : use run configuration and set the arguments to be passed on to the main method args

**WRAPPER CLASS :**

* Byte - byte
* Character -char
* Short -short
* Long -long
* Integer - int
* Float -float
* Boolean -boolean
* Double -double
* A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

**Need of Wrapper Classes**

* They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
* The classes in java.util package handles only objects and hence wrapper classes help in this case also.
* Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
* An object is needed to support synchronization in multithreading.

**SCANNER:**

* Scanner (ctrl+space)
* Scanner scan = new Scanner(System.in);

Alt+Shift +S +R 🡪 generates setters and getters in eclipse

**EXCEPTION:**

* Under “Object class “, there is a class called “Throwable “ (subclass)
* Under throwable – 2 subclasses – Error, and Exception
* Object 🡪 Throwable 🡪 Error, Exception
* Errors - cannot be handled.
* Anything outside the JVM we cannot handle it.
* Out of memory
* Exceptions – can be handled by the developers in programs.
* Checked Exception and unchecked Exception
* Checked Exception – we have to check during development **(java.util ) (java.io)**
* If we misses out then the JVM will handle it as an unchecked exception. **Java.lang ,**

**Java.lang.\* Exception .**will have all the possible exceptions which might get missed. We need to handle it.

**try {**

**22/0;**

**}**

**catch(java.lang.ArithmeticException a ){**

**//handle error , provide solution**

**SOP(a);**

**}**

**catch(java.lang.ArrayIndexOutOfBoundException a ){**

**//handle error , provide solution**

**SOP(a);**

**} catch(classCastException a ){**

**//handle error , provide solution**

**SOP(a);**

**} catch(NullPointerException a ){**

**//handle error , provide solution**

**SOP(a);**

**}**

**catch(Exception a ){ //Generic Exception**

**//handle error , provide solution**

**SOP(a);**

**}**

* Try must be mandatorily followed by catch.
* **try {**

**22/0;**

**}**

**catch(Exception a ){ //Generic Exception**

**//handle error , provide solution**

**SOP(a);**

**}**

* **FINALLY BLOCK :**
* **try {**

**22/0;**

**}**

**catch(Exception a ){ //Generic Exception**

**//handle error, provide solution**

**SOP(a);**

**}**

**Finally(**

**//even if there is an exception or not, this block will be executed.**

**//works to deal with the abrupt ending of the program due to exception.**

**//Why finally: ”*to Close the resources. i.e., can use dereference and close.”***

**}**

* **try{**

**//mandatory**

**}**

**Catch() or finally{} – one is mandatory**

**Try{}**

**Catch{}**

**Try{}**

**Finally{}**

**try**

{

Sysout(arg

s[999]); //possible of raising an exception

}

**catch**(java.lang.ArrayIndexOutOfBoundException var)

{

Sysout(“solution”);

}

**USER DEFINED EXCEPTION/CUSTOM EXCEPTION: THROW:**

Throw new Exception ;

We can throw Custom Exception;

Class MyClass extends Exception {

}

* create an new package for all the exceptions , com.exception
  + create a new class and inherit from java.util.Exception
  + Add a String variable and the constructor method
  + We can use the override method in the exception block
* Catch (PositiveNumberException | ArithmeticException e){

//we can use pipeline to add multiple exceptions in the catch block

}

**COLLECTION FRAMEWORK:**

**Collection – collection of objects**

Cmd > javap java.util.collection

Collection is an interface and it inturn extends interable. //12 methods

Java.util.interable(); //3 methods

Javap java.util.set

Set extends collection

Javap java.util.list : list extends collection

Javap java.util.map : does not extend anything

Javap java.util.HashSet

//hashing technique , it’s a class and it extends AbstractSet and it implements Set

So the other 15 methods are overridden by HashSet ., this is the concrete class

//interface var = new Impl();

Set set = **new** HashSet();

**import** java.util.HashSet; //implementation of the set

**import** java.util.Set; //interface

March 28th 2022

JAVA

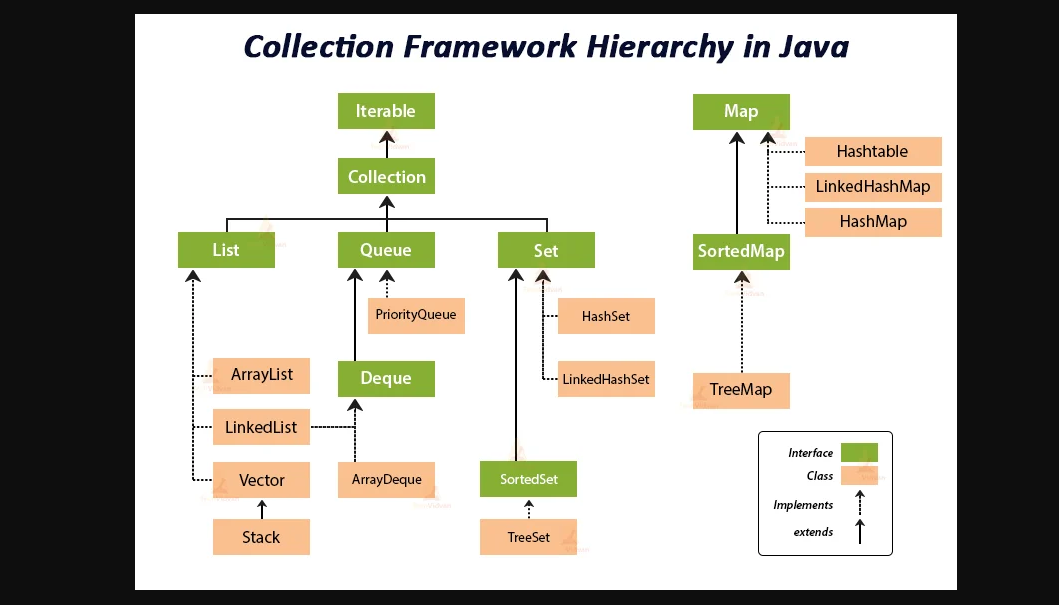
API 🡪 predefined 🡪 .class , .package , interface 🡪 jar files

All predefined .jar files 🡪 rt.jar file : location , c-> program files -> java -> java version-> lib

Predefined Packages 🡪 java.lang.\* ; (all .class ) – default package .

Java.util.\* ; 🡪 utility 🡪 collection 🡪 it is a collection of APIs ( classes, packages , interfaces , Algorithms ), because java does not support pointers . 🡪 Collection also has something called Data Structures , (how data is stored and retrieved internally)

Java Collection :



Interface can extend another interface, all the methods in the parent interface has to be inherited by the child interface

SET <interface>:

* Set is a collection of objects and no primitive data type.
* Set is **Dynamic** in size (add /remove ‘n’ number of objects dynamically).
* Set has **NO** **Duplicates** (not the data, but object from the same memory address).
* While iterating / looping, no order is maintained.
* Set is an **interface** which extends another interface called Collection. Collection extends iterator. Instead of overriding all these methods, we use **Concrete** Class/ **Complete** Class/ **Implementation** Class. They are HashSet, LinkedHashSet and TreeSet.
* **HashSet** – it follows unsorted Order
* **TreeSet** – it follows sorted order
* **LinkedHashSet** – Double Linked
* **Maven Local Repository** (local repo) will be stored in the c-> users->nithinkumarp-> .m2-> repository.

**In for loop : While using a iterator for loop , if the set has all the same type of Objects then we can specify the type of Object in the place of object .If the collection contains different type of objects its preferable to use the generic Object as a type in the statement.**

**for** (Iterator iterator = set.iterator(); iterator.hasNext();) {

Object object = (Object) iterator.next();

}

**Type Casting in java:**

Int a = ( int ) 10.45f;

Object a = (employee) Emp1;

**Instance of in JAVA:**

**If( object instanceof Employee ){ // object instanceof <type**

**Employee new\_name = (Employee ) object; // <type> new\_name = (<type>) object;**

**}**

***“ALWAYS FOLLOW “***

**<Interface> var = new <ClassImpl>();**

***GENERIC COLLECTION:***

Set<Object\_name> set = new HashSet<Object\_name> ( );

Set <Employee> set = new HashSet<set> ( ); // This set can only keep Employee Object

* **Final** : constant , avoids overloading, over ridding
* **Finally** : a block which can come along try block
* **Finalize** : Garbage the collection . Helps in protecting the files before the GC acts on a particular data.
* Every java file can have multiple Classes with default access modifiers. IF there is a class with public access modifier then it should be the file name. There should be only 1 class as public and it should be same as that of the File name

***COMPARABLE & COMPARATOR:***

* Java provides 2 **interfaces** to **sort objects** 🡪 Comparable and Comparator
* Comparable comes under java.lang. Package.
* Javap java.lang.Comparable
* Comparator comes under java.util. Package.
* Javap java.util.Comparator 🡪have to override the public abstract int compareTo( T ) method available in the interface. (T represents any object type)
  + Int return type – it returns 3 values , 1 , 0 , -1
  + If (this.age == arg.age) its equal it returns 0
  + If (this.age > arg.age) its > returns 1
  + If (this.age < arg.age) its < returns -1

Set: no insertion order-no duplicates

Sorted Set – order/sequence -- HashSet

Unsorted Set – no order/ no sequence -- TreeSet

Tree – an algorithm, a binary tree

List: insertion order is maintained – allows duplicate –

* ***Javap java.util.Collections*** : it’s a class has various methods which can be used.
* Comparable provides single sorting sequence , comparator can have multiple sequence
* ***Comparator :*** class Employee implements java.util.Comparator

Client server architecture

Database 🡪 MySql 🡪 open 🡪 Java🡪 Oracle

Persistence 🡪 Storage 🡪 3rd party

Using an App we will store data into the database/data source.

Even if the data is not used, data is available in the DB.

//we can use java to write and then .net to read :

Java 🡪 store 🡪 DB 🡪 Read 🡪 .Net

**LOGGING:**

SOP 🡪 replace 🡪 Logger

Various Logger

🡪 log4j is one of the vendor , his product is part of Apache. But vulnerable

**Logging in JAVA**

🡪log4j has certain vulnerabilities, so we can use, **java.util.log**

API 🡪 jar files ( \*.jar )

How to get log4j files ??

Build tool 🡪 Maven /Gradel

Step 1: Create a Maven project

Step 2 : Maven repository :

Add the log4j maven repository available on the net to the pom file.

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com</groupId>

<artifactId>log4jdemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<log4j.version>1.2.16</log4j.version>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/log4j/log4j -->

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>${log4j.version}</version>

</dependency>

</dependencies>

</project>

Step 2 : Create a property log4j.properties

It uses LHS = RHS concept

App uses LHS and while running it uses RHS data

Add this file to the 🡪 **src/main/resources**

The resources required by the app are added in this resources

Exception : while creating give the super class as Exception, create a variable of message and a constructor , to pass the message use the override method

SQL : Structured Query Language.

SQL commands are mainly categorized into four categories.

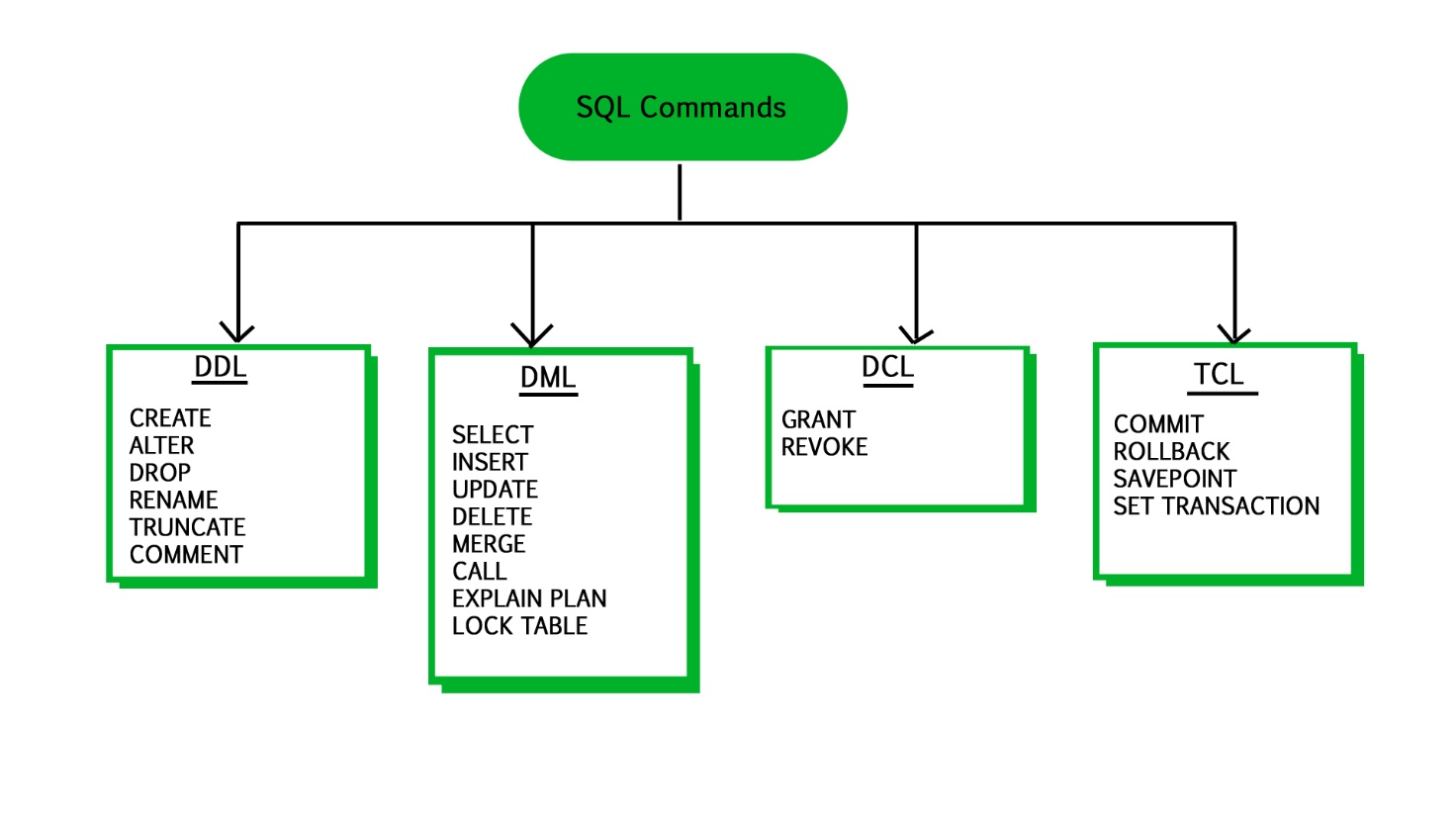
Data Definition language: CREATE, DROP, ALTER, TRUNCATE, COMMENT, RENAME.

Data Query language: SELECT

Data manipulation language: INSERT, UPDATE, DELETE, LOCK

Data control language: GRANT (PRIVILEGES TO THE DATABASE), REVOKE(WITHDRAWS THE USERS ACCESS)

Transaction control language: COMMIT,ROLLBACK,SAVEPOINT



Data od MySQL (open source / freeware) 🡪 oracle

Latest : 8.X

Ways to install :

CLI : command line interface

GUI : Graphic user interface

Default user name: root

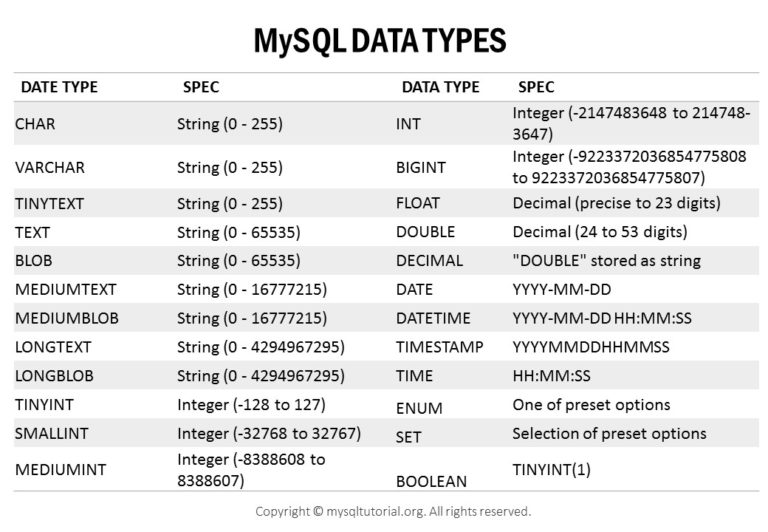
Default password: Reset123/root

**;** 🡺 MANDATORY TO TERMINATE THE STATEMENT IN MYSQL

ALL THE KEY WORDS /RESERVED WORDS ARE IN CAPITAL LETTER – CODING STANDARDS; BUT OTHERWISE NOT CASE SENSITIVE

BLOB – Binary Large Object

There are about 24 different data types in MYSQL



mysql > select version(); //version of mysql

mysql > select current\_date; //current date

mysql > select now(); //date and time

SCHEMA (databases)

mysql > show databases; //gives the databases(i.e., SCHEMA ) available

mysql> select database // current working database

mysql > CREATE DATABASE <demodb1>; //creates a database

mysql > use <demodb1>; //moves the working directory to the database we created

mysql >show tables ; // shows the available tables

mysql >CREATE TABLE IF NOT EXISTS <usertable> (

<id> int(5) NOT NULL auto\_increment ,

<username> varchar(25) NOT NULL DEFAULT ‘’ ,

<password> varchar(25) NOT NULL DEFAULT ‘’ ,

PRIMARY KEY ( id )

);

**CREATE TABLE IF NOT EXISTS library\_table(**

**accession\_Number INT ,**

**book\_Title varchar(25) NOT NULL ,**

**author\_Name varchar(25) NOT NULL,**

**book\_Price FLOAT NOT NULL,**

**issued\_Status BOOLEAN NOT NULL ,**

**PRIMARY KEY (accession\_Number)**

**);**

**INSERT INTO library\_table (accession\_Number, book\_Title, author\_Name, book\_Price, issued\_Status)**

**VALUES(100,’java’ , ‘nithin’ ,500.0,0);**

mysql >DELETE FROM <usertable> where id = 123; //delete a record

mysql >COMMIT //to save

mysql >ALTER TABLE <usertable> RENAME COLUMN <id> TO <userId> ; //rename the column in a table

mysql >ALTER TABLE <usertable> ADD <userlocation> varchar(40) AFTER <password>;

mysql > ALTER TABLE <usertable> DROP COLUMN <userlocation>; //delete column

mysql> ALTER TABLE table\_name RENAME TO new\_table\_name; //rename table

1. mysql > DELETE FROM table\_name WHERE condition; //delete a record

mysql > SELECT

officeCode,

city,

phone,

country

FROM

offices

WHERE

country

IN ('USA' , 'France'); // search for a column in the given set of values

mysql > SELECT

DISTINCT lastname

FROM

employees

ORDER BY

lastname;

Mysql > **SELECT**

customerName,

country,

salesrepemployeenumber

**FROM**

customers

**WHERE**

Salesrepemployeenumber

**IS NULL**

**ORDER BY**

customerName; //to check if the value is null or not(**IS NOT NULL**)

SELECT

select\_list

FROM t1

INNER JOIN t2 ON join\_condition1

INNER JOIN t3 ON join\_condition2

...;

***JDBC:***

Database 🡪 RDBMS 🡪 Multiple table (3rd part)

JAVA application uses API (\*.java) 🡪 Database 🡪 RDBMS 🡪 Multiple table (3rd part)

Here java uses JDBC API to connect to database.

JDBC 🡪 JAVA DATABASE CONNECT

4 types:

1. Type 1 Driver 🡪 java used to connect to Microsoft through ODBC (Open DB connectivity). ODBC is connected to the data source /database. (mySql)
2. Type 2
3. Type 3
4. Type 4 🡪 THIN Driver 🡪 java can directly connect to Database and it requires only (<database>.jar) files. [ex: if oracle, oracle.jar]
   1. Java 🡪 java.sql.\*

🡪 java.sql.\* is an Interface

🡪DriverManager(class)

🡪Mysql has to implement the rules of java.sql to use this JDBC

JDB DEMO:

Step 1. Create a Maven Project (pom.xml)

1. JDK 1.8
2. Mysqlconnector.jar

Step 2. Create packages [com.main , com.service , com.model , com.dao (also k/as Data Access Object /com.repository) , com.exception ].

* 1. Only com.dao has the java.sql statements
  2. Com.dao has to return the POJO files

Step 3. In the Com.dao( Appl🡪 database)

1. To establish the connection, we need the location of the database server .

(URL: location)

It requires the credentials to access the DB

DriverManger(sql) Connection <interface>

Serialization 🡪 to maintain the state of the object , POJO files need to implement Serializable

select \* from library\_table WHERE accession\_number = 101;

# JAVA 8 FEATURES

* JSE java Standard Edition – core java (JDK 17), till Java 8 is open source, most of the companies prefer.
* JEE Java Enterprise Edition – (Advances Java)
* Change in INTERFACE ,

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.
* **Interface** :
  + An interface has static constants final variables and abstract methods An interface is completely “abstract class” (collection of pure abstract method)
  + Interfaces form a contract between the class and the outside world
  + Multiple inheritance
  + To have common behaviour/functions/methods/Business Logic
  + Now in java 8 , interfaces can have concrete methods, static variables , static methods
* **Default Methods :**
  + Default methods are also known as defender methods or virtual extension methods.
  + Backward Compatible (why default).
  + Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.
  + Complete method () == concrete method () == method definition {//body}
  + In case if we have same default methods in different interfaces and it is implemented by a class then it is mandatory to override
  + Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
  + As name implies, default methods in java 8 are simply default. If you do not override them, they are the methods which will be invoked by caller classes. They are defined in interfaces.
  + Public interface Interface\_name{
    - * Public default int sum()\_{
      * }
      * };
* **STATIC METHOD in an Interface:** 
  + Keyword : Static
  + Only one
  + Pre created/ pre memory allocation
  + To access - ClassName.method( )
  + Also k/as Class level Method / variable
  + Interface can also have static methods
  + These static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.
* **Functional Interface : (SINGLE ABSTRACT METHOD – SAM)**
  + A functional interface is an interface that contains only one abstract method.
  + They can have only one functionality to exhibit.
  + From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
  + A functional interface can have any number of **default** methods.
  + A functional interface can have any number of **STATIC** methods.
  + Runnable, ActionListener, Comparable are some of the examples of functional interfaces.
  + Functional interfaces can be annotated. (annotation .i.e., represented by ‘@’ – to give additional information about any particular part of the code)
  + Use : @FunctionalInterface //optional but , it is better to mention the annotations

Because it helps in stopping from having more than 1 abstract method.

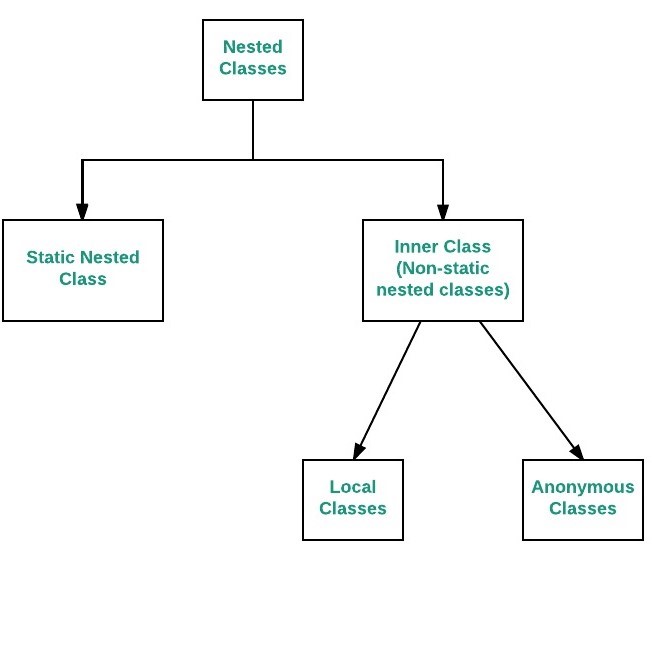
* What’s the difference between **abstract class** and java 8 **interface**?
  + Myclass extends OneClass (no multiple inheritance)
  + MyClass implements InterfaceOne , InterfaceTwo (multiple Interface)
  + Abstract Class:
    - Abstract class can have variables + concrete /complete () and define the class abstract. Because the class ins abstract we cannot instantiate( new())
    - In a class if we have one or more abstract method then the class is abstract
    - Abstract class can have constructor but cannot be used.
  + Interface :
    - Public static final VARIABLE = value;
    - Public abstract method(); // method declaration
    - In JAVA 8 ,
      * Default 🡺 complete/concrete/method definition () // with logic
      * Backward Compatibility.
      * Java8 INTERFACE is very equivalent to CLASS
      * Public static Final - Variable - variable
      * Abstract method(); - definition{ }
      * Default method(); { }
    - **INSTANTIATION** is not possible in JAVA **INTERFACE**.
    - **No constructor** can be defined.
* **package** com.service;
* **public** **interface** RulesInterface {
* **public** **static** **final** **int** ***var*** = 20; // public static final = const
* **public** **abstract** **void** display();
* **public** **default** String displayVar() {
* **return** "Welcome to Default";
* };

}

C 🡪 Structural;

C++, JAVA 🡪 OBJECT ORIENTED;

JAVA 8 🡪 Functional programming

* **Functional Programming:** 
  + Function within a function
  + Int add(int a , int b){ }
  + Int sub (int add (int a, int b ) { } )
  + To implement a functional programming java 9 introduced (lambda express , method reference )
  + LAMBDA Expression :
    - **( ) - > notation**
  + Change in interface in java 8, prior to JDK <1.8>.
  + There can be only abstract methods (pure abstract method) and variables(static final constants) in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritance in Java. Interfaces form a contract between the class and the outside world. Java Interface also represents the IS-A relationship.
  + In the Java programming language, an *interface* is a reference type, similar to a class that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces. To implement an interface we use the keyword implements
  + **// A simple interface**
  + **Public interface Player**
  + **{**
  + **Public static final int id = 10;**
  + **Public abstract int move();**
  + **}**
  + **Class InterfaceImpl implements <interface\_name>**
* **Nested Classes in Java**
  + In Java, it is possible to define a class within another class, such classes are known as nested classes.
  + They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.
  + The scope of a nested class is bounded by the scope of its enclosing class/
  + Nested classes are divided into two categories:
  + Static nested class: Nested classes that are declared static are called static nested classes.
  + Inner class: An inner class is a non-static nested class.
* 
* **Anonymous Inner Class in Java**
  + It is an inner class without a name and for which only a single object is created.
  + An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overriding methods of a class or interface, without having to actually subclass a class.
  + An anonymous class has access to the members of its enclosing class.
  + An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
  + Like a nested class, a declaration of a type (such as a variable) in anonymous class shadows any other declarations in the enclosing scope that have the same name.
  + An anonymous class has access to the members of its enclosing class.
  + An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
  + Like a nested class, a declaration of a type (such as a variable) in anonymous class shadows any other declarations in the enclosing scope that have the same name.
  + An Anonymous inner class will be terminated with a semi colon (;)