What is Java?

* It is a programming language.
* It is high level, robust, object oriented and secure programming language.
* It is platform independent. It was introduced in 1995.

<https://docs.oracle.com/javase/8/docs/api/index.html>

OpenJDK

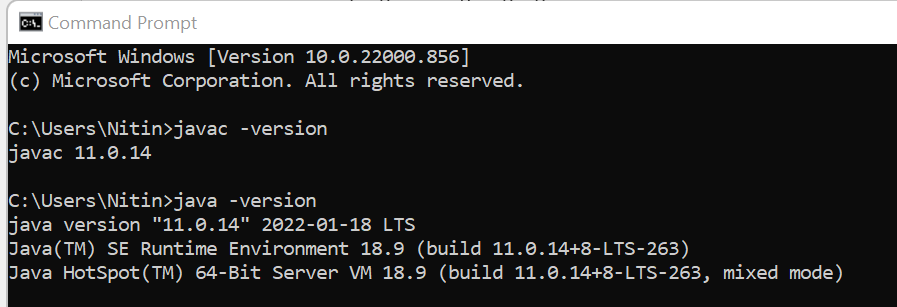
<https://www.oracle.com/java/technologies/downloads/>

Oracle JDK

<https://www.oracle.com/java/technologies/downloads/>

after installation - verify the installation

go to command prompt / terminal and ensure the version for javac & java are the same.



If it doesn’t work,

Set the environment variable that is PATH.

set PATH = %PATH%; C:\Program Files\Java\jdk-11.0.14\bin

alternatively we can go to Edit System Variable and add Java Home path to Path variable.

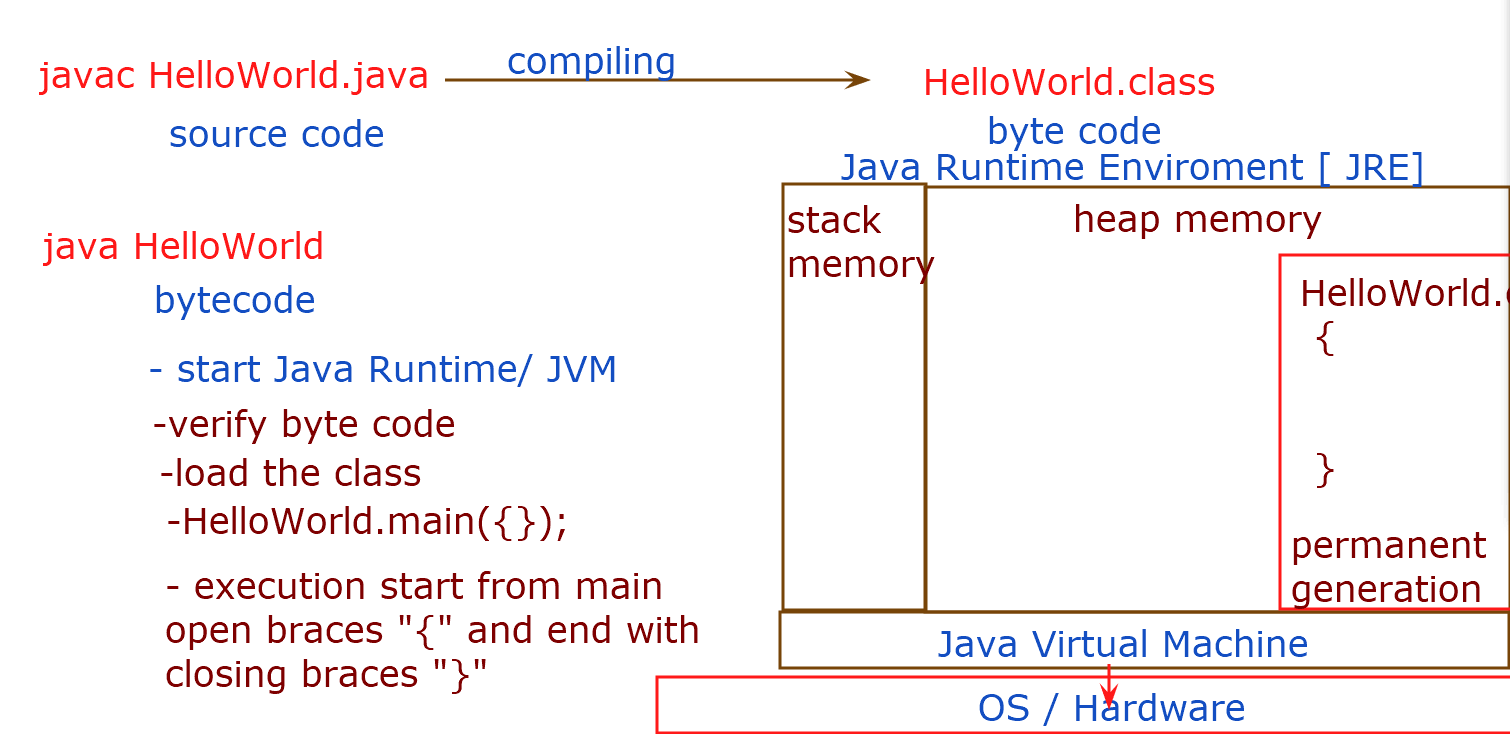
MacOS [ pls check the java home path ]

export PATH = $PATH: /usr/java/jdk-11.0.14/bin

javac === Java compiler / it is used to translate high-level Language to Byte Code.

java == Java Runtime / it is responsible for starting JVM and executing Java Main class that is the starting point of your java application.

**Write Once and Run Anywhere**



**Types of Java Edition**

1. Java Standard Edition [ Java SE ]
   * Desktop applications
   * Console Application [ No GUI ]
   * Java Libraries
2. Java Enterprises Edition [ Java EE ]
   * Web Application
   * Distributed Applications [ Enterprise Application]
3. Java Micro Edition [ Java ME ]
   * Mobile Application
   * Java TV
   * Java Card [ program information for smart card chipset]

**Object Oriented Programming**

1. Encapsulation & Data Hiding
2. Inheritance
3. Abstraction
4. Polymorphism
5. **Encapsulation & Data Hiding**

Wrapping up data & behavior into one capsule / block is known as encapsulation. A class is a way to encapsulate the state[ data ] & behavior.

This is to form blue print.

Data hiding - To hide the data from direct accessibility to prevent direct modification.

Graphical user interface, text

Description automatically generated

**Syntax for Creating Object**

<ClassName> ref-variable = new <ClassName>();

* Java is case Sensitive
* Java follows certain naming convention

1. ClassName - upper camel case
2. Field members [ state ] & methods [ behavior ] – lower camel case
3. Constant variable -- UPPER CASE

Text

Description automatically generated

**Types of Variables**

1. Primitive Type [ 8 types ]
   1. byte 8 bits -128 to 127
   2. short 16 bits -32768 to 32767
   3. **int**  32 bits -2147483648 to -2147483647
   4. long 64 bits
   5. float 32 bits
   6. **double**  64 bits
   7. char 16 bits Unicode char ‘A’
   8. boolean \*1 bit [ actual size may vary depending on OS] true/false

Any number with a precision literal value like 1.0 , is by default double type

All whole numbers 10 are by default-int

1. Reference Type or Non-Primitive

Class, Interface, Enum & Array types of variables are known as reference types.

Diagram, schematic

Description automatically generated

**Scope of Variables**

1. Instance Variable / Field members – Accessible in all methods in the given class. All the instance variables get initialized with default values at the time of instantiation.

byte, short, int , long ------------------------------ 0 [ default value ]

float, double ------------------------------------------ 0.0

boolean ----------------------------------------------- false

char --------------------------------------------------- ‘\u’

all reference type , the default value is null.

1. Local Variables - Accessible only in the block that they are declared in. you must initialize it before you use it.
2. Static Variables / Class Variable – Accessible by all instances and maintains only copy per JVM.

**Static Variables** – they get memory allocated at the time of class loading. They belong to class memory. It is initialized with a default value.

There is the only copy of a static variable is created per JVM. This can be shared by all instances of a class.

**Static Methods:**

You can use a static keyword to declare a method. This method can access only static variables. You will have only one copy of static context per JVM.

Relational Operators

>

<

<=

>=

==

!=

Arithmetic Operators

+

-

\*

/

%

++

--

Logical Operator

&&

||

!

Access Modifier “private” keyword can be used to hide.

**Constructor** is a method with the same name as classname and no return type. It is invoked automatically at the time of instantiation. It is used to initialize the fields.

A). No Argument Constructor [ default constructor ] is supplied by the compiler to the byte code of a class only if no constructors were added by the developer.

B) No default constructor will be added if you have your own constructor. You may need to add it manually in your code.

**Inheritance**

**public** **class** Manager **extends** Employee {

**private** String deptName;

**public** Manager(**int** id, String name, **double** salary, String deptName) {

**super**(id,name,salary);

**this**.deptName=deptName;

}

**public** String getDeptName() {

**return** deptName;

}

**public** **void** setDeptName(String deptName) {

**this**.deptName = deptName;

}

}

**Method Overriding**

Re-writing the super class method that is inherited, is known as method overriding.

* Method in subclass must have the same name, signature [ arguments types, numbers and sequence] and return type.
* Overridden method can not have less accessibility than super class method.

**Method Overloading**

Writing multiple methods with the same name but with different/same functionality in a class. Constructors can be overloaded as well.

Rule 1 – there must be difference in method’s signature.

TNS / TNO = Type, Number and Sequence/Order of Arguments.

System.out.println();

String s=”Nitin”;

System.out.println(s);

int num=10;

System.out.println(num);

Employee e1=new Employee(1,”Alex”,1290.00);

System.out.println(e1);

class Maths{

public static int sum(int x, int y){

return x+y;

}

public static int sum(int x, int y, int z){

return x+y;

}

public static double sum(double x, double y){

return x+y;

}

public static double sum(double x, int y){

return x+y;

}

public static double sum(int y, double x){

return x+y;

}

}}

**Conditional Statements**

if(condition/s){

//statements when the condition is true

}

if(condition/s){

//statements when the condition is true

}else{

//statements when the condition is false

}

if(condition1){

//statements when the condition1 is true

}else if(condition2){

//statements when the condition2 is true

}else if(condition\_N){

//statements when the conditionN is true

}else{

//statements when the none of the above are true

}

\*\*\*\*\*\* Nesting of If is possible as n when required.

switch(variable){

case value1:

//statement when variable==value1

If(var==value1){

//statement when variable==value1

}else if(var==value2){

//statement when variable==value2

}else if(var==value\_N){

}else{

break;

case value2:

//statement when variable==value2

break;

case value\_N:

//statement when variable==valueN

break;

default:

//statement when none were satisfied

}

\*\*\*\*values in the switch are either literals or constant variables.

If(var==value1 or var=value2){

//statement when either conditions are true

}

switch(variable){

case value1:

case value2:

//statement when variable==value1 or value2

break;

case value\_N:

//statement when variable==valueN

break;

default:

//statement when none were satisfied

}

**Loops :**

**repeating the execution of statements as long as conditions are true.**

while(condition/s){ // iterate the statements zero or more times.

//statements

}

do{ // iterate the statements one or more times.

//statements

}while(condition/s);

for(initialize var ; conditions ; increment/decrement){

statements

}

**int** i=1;

**while**(i<=10) {

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

i++;

}

**int** i=1;

**do** {

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

i++;

}**while**(i<=10);

**for**(**int** i=1; i<=10;i++) {

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

}

**Array**

Array is a set of a finite number of elements of same type. Each elements share the same name as array.

**Declaring an Array**

Type[] arrayName;

**Instantiating An Array**

arrayName=new type[no. of elements];

**Declaration & Instantiation**

type[] arrayName=new type[no. of elements];

**Declaration ,Instantiation & Inialization**

type[] arrayName={value1, value2, value3, ValueN};

\*\*\* number of values will determine the number of elements.

**\*\*\* elements gets initialized with the default values as per type.**

**Example**

int[] numbers=new int[5];

numbers[0]=10;

numbers[1]=20;

numbers[4]=50;

**OR**

int[] numbers={1,2,3,4,5};

Diagram

Description automatically generated

for(int i=0; i<numbers.length ; i++){

System.out.println(numbers[i]);

}

OR

Enhanced For loop / for Each loop

for(int x:numbers){ // introduced first time in Java 5

System.out.println(x);

}

**Multi-Dimensional Array**

type[][] arrayname=new type[no. of array][no. of elements];

type[][] arrayname=new type[row][col];

int [][] numbers =new int[4][4];

Table, calendar

Description automatically generated with medium confidence

Employee[] empList=**new** Employee[4];

empList[0]=**new** Employee(1,"Nitin",2000.00);

empList[1]=**new** Employee(2,"Alex",3000.00);

empList[2]=**new** Employee(3,"Dereje",4000.00);

empList[3]=**new** Employee(4,"Gerges",5000.00);

// for(Employee e:empList) {

// e.print();

// System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

//

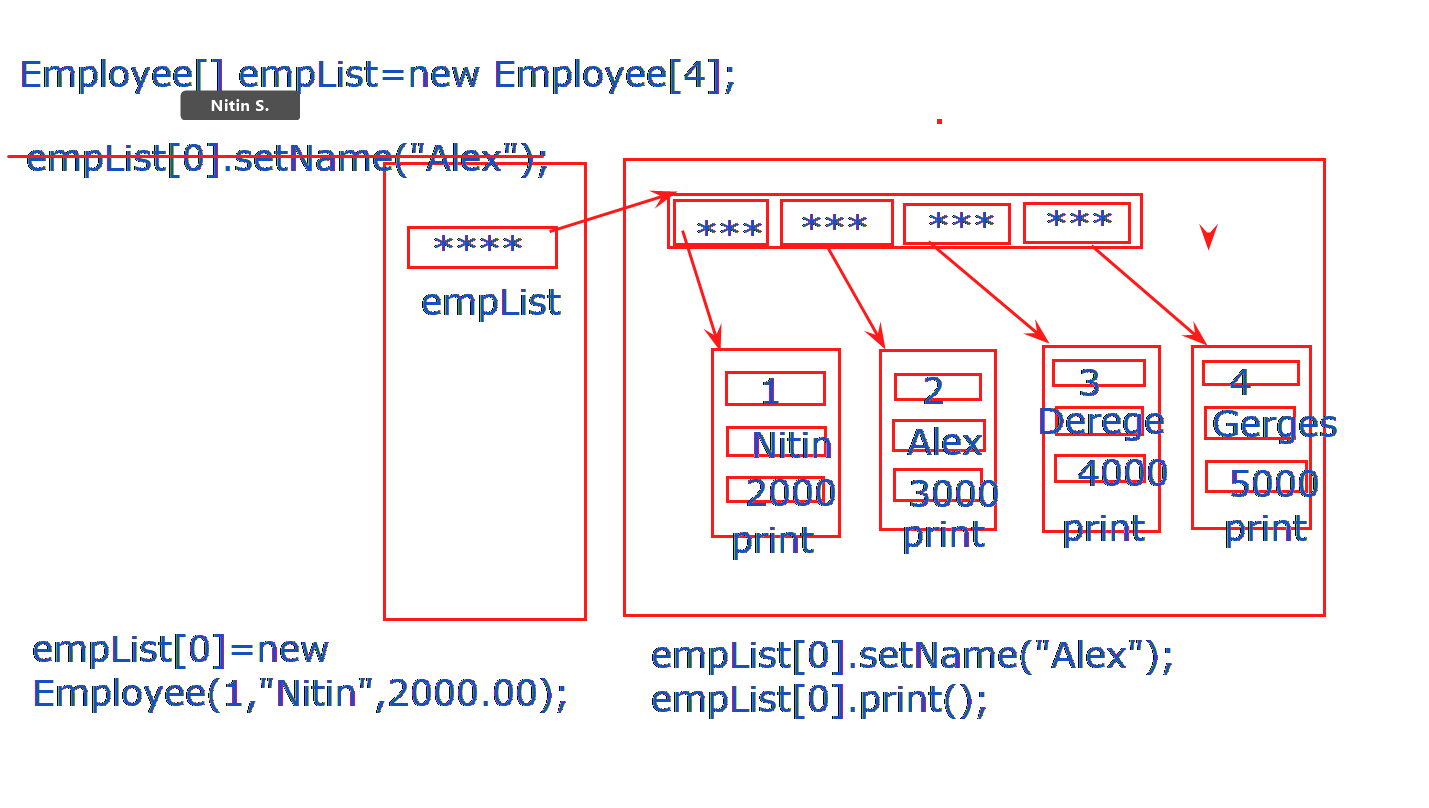
// }

**for**(**int** i=0; i<4; i++) {

empList[i].print();

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

}



**Abstraction & Polymorphism**

* **Abstract Class :** A class that need not be instantiated and can be extended only, is an abstract class.

**public** **abstract** **class** Employee {

**private** **final** **int** id;

**private** String name;

**private** **double** salary;

**public** Employee(**int** id, String name, **double** salary) {

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

//setters n getters

}

* **Abstract Method.** A method that must be overridden by all sub-classes and can be declared without definition[body] in an abstract class, is an abstract method.
  + **Subclass has only two choice :**
    - **To override/implement the abstract method**
    - **Or declare the class as the abstract**

---- see the examples – WalkingApp

**Polymorphism**

Poly = many

Morphism = faces

Employee e= ?;

e.print(); //print() must exist in Employee class

at the run time the code that will execute it can be either from Manager Class [ if overridden ] or from Employee class.

\*\*\*\* Virtual Method Invocation / Dynamic method call.

private – accessible with in the class.

default - accessible with-in the class and also by classes in the same package.

protected – accessible with-in the class, class in the same package and by subclasses from different packages.

public - accessible by all.

**Interfaces :**

* Contains only abstract methods. This is by default.
* All the methods in interfaces are by default public.
* A variable is by default public final static.

public interface A{

int x=10;

public void m1();

void m2();

}

Diagram

Description automatically generated

interface Returnable{

public void doReturn();

}

|  |  |
| --- | --- |
| class Shirt extends Clothing implements Returnable{  ---- properties----  public vod doReturn(){  S.o.p(“5 days return policy”);  }  } | class Trouser extends Clothing implements Returnable{  ---- properties----  public vod doReturn(){  S.o.p(“7 days return policy”);  }  } |
| class Bellies extends Footwear implements Returnable{  ---- properties----  public vod doReturn(){  S.o.p(“10 days return policy”);  }  } | class Flats extends Footwear implements Returnable{  ---- properties----  public vod doReturn(){  S.o.p(“15 days return policy”);  }  } |

Returnable r=?

new Shirt();

new Trouser();

new Flats();

new Bellies();

Clothing c= new Shirt();

if(c instanceof Shirt) {

Returnable r=(Returnable) c;

}

Returnable r=(Returnable) c; // ClassCastException will be throawn by Java Runtime if c refers to the object not implementing Returnable.

// refer example “The WalkingApp” in Github folder 09-12-2022

**String**

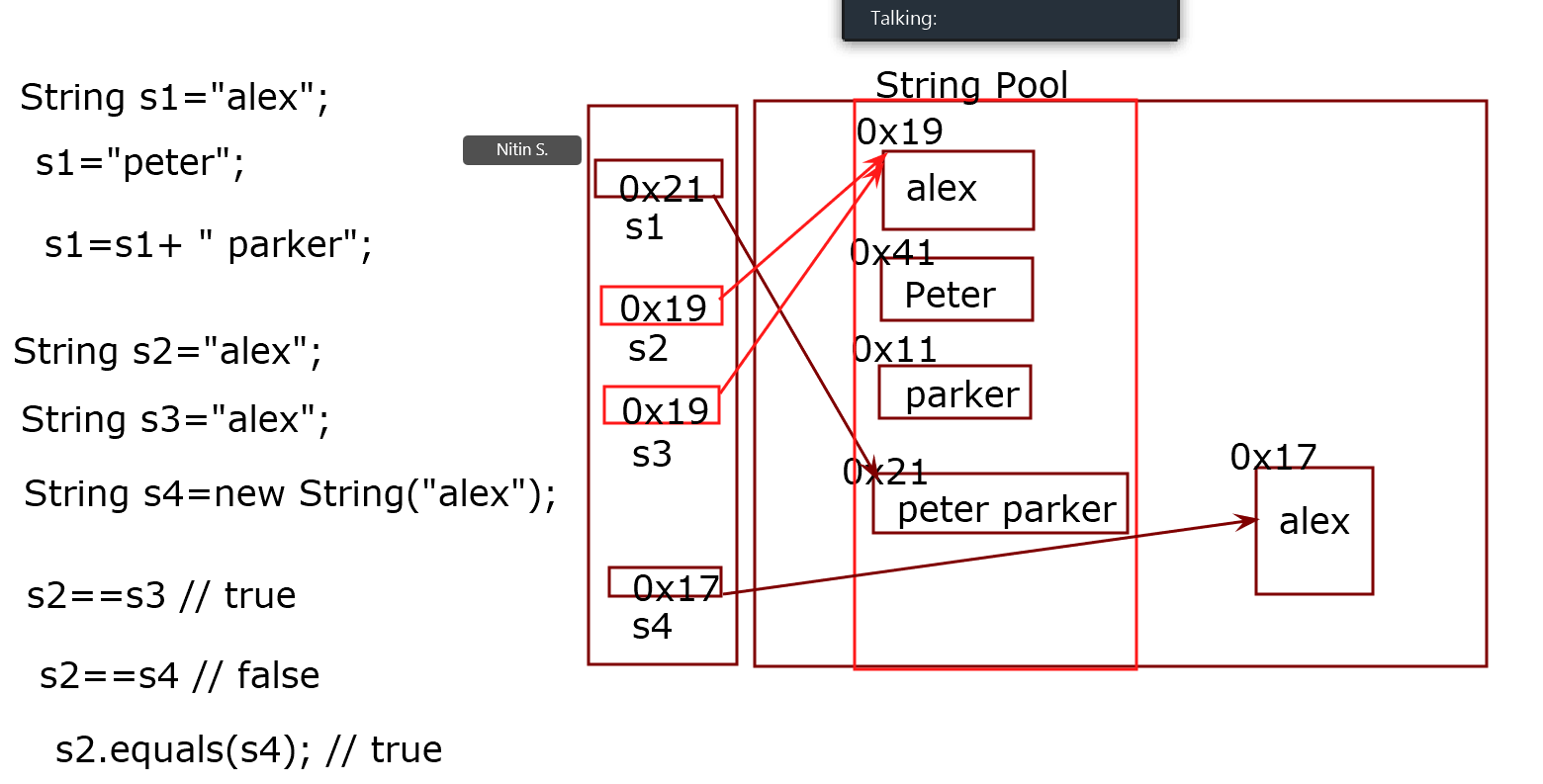
* String is a class and is found in java.lang package.
* Any String literal “abc” is itself an implementation of String class [ i.e. an object of String class ]
* String is immutable [ constant ]

“HelloWorld” is an object of String class.

String s1=”HelloWorld”;

However you may create String object explicitly as well.

String s1=new String(“HelloWorld”); /// not recommended



StringBuider

// add description

**Singleton**

--- class design such a way to produce only one instance per JVM.

1. Make the constructor private
2. Create a private static final reference variable to hold the object reference itself.
3. Create public static getInstance() to return object reference.

**public** **class** PaymentGateway {

**private** **static** **final** PaymentGateway ***pg***=**new** PaymentGateway();

**private** String cardNo;

**private** **double** amount;

**private** String cardType;

**private** PaymentGateway() {

}

//setters n getters

**public** **static** PaymentGateway getInstance() {

**return** ***pg***;

}

}

**Static Initializer** - executed only once at the time of class loading and can be used to initialized static members only.

class A{

static int number;

int x;

static{

number=0;

//x=10; // error

// this block gets executed only once

}

}

**Object class**

All the java classes by default inherit properties from the class Object. It is root of every class hierarchy.

**public** **class** B{

**int** number;

B(**int** number){

**this**.number=number;

}

@Override

**public** String toString() {

**return** ""+number;

}

}

B b=**new** B(10);

//System.out.println(b.toString());

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

**Collection Classes**

**java.util packages , you can find collections classes.**

* Collection classes can hold only reference types of elements.
* This is dynamic. It can grow or shrink in size at the runtime.

Employee[] empList=new Employee[3];

empList[0]=new Employee(…………………);

empList[1]=new Employee(…………………);

empList[2]=new Employee(…………………);

//empList[3]=new Employee(…………………); // error will throw ArrayIndexOutOfBoundsException

//ArrayList<Employee> empList=new ArrayList<Employee>(3);

ArrayList<Employee> empList=new ArrayList<E>(3);

empList.add(new Employee(…………………));

empList.add(new Employee(…………………));

empList.add(new Employee(…………………));

empList.add(new Employee(…………………));

empList.add(new Employee(…………………));

int x=empList.size(); // 5

**Generic Class**

|  |  |
| --- | --- |
| class CacheShirt{  ArrayList<Shirt> list;  public CacheShirt(){  list=new ArrayList<>();  }  public void add(Shirt s){  list.add(s);  }  public Shirt get(int idx){  Shirt s=list.get(idx);  return s;  }  } | class CacheFlats{  ArrayList<Flats> list;  public CacheFlats(){  list=new ArrayList<>();  }  public void add(Flats s){  list.add(s);  }  public Flats get(int idx){  Flats f=list.get(idx);  return f;  }  } |

<> introduced in Java 8 to represent copy of Type [ T ] in generic at the time of instantiation. This is known as **Diamond Reference** Operator.

CacheShirt cs=new CacheShirt();

cs.add(new Shirt(……………..));

Shirt s=cs.get(0);

CacheFlats fs=new CacheFlats();

fs.add(new Flats(……………..));

Flats f=cs.get(0);

|  |
| --- |
| class CacheAny<T>{  ArrayList<T> list;  public CacheAny(){  list=new ArrayList<>();  }  public void add(T s){  list.add(s);  }  public T get(int idx){  return list.get(idx);  }  } |

CacheAny<Shirt> cs=new CacheAny<>();

cs.add(new Shirt(……………..));

Shirt s=cs.get(0);

CacheAny<Flats> fs=new CacheAny<>();

fs.add(new Flats(……………..));

Flats f=fs.get(0);

CacheAny<Trouser> ft=new CacheAny<>();

ft.add(new Trouser(……………..));

Flats f=ft.get(0);

Iterable<E>

Collection<E>

HashMap<K,V>

Map<K,V>

SortedSet<E>

TreeSet<E>

HashSet<E>

ArrayList<E>

Set<E>

List<E>

**List:**

It is dynamic. Size of List can grow or shrink at the runtime depending on requirement.

It is an ordered list of collections.

**Set:**

It contains unique elements. It is going to use hashcode & equals method to verify if two objects are same or not.

Map: it Contains object that holds key n value pair. Key is unique [ Set type ]

Username=alex

Password=2233

Hostname=localhost

Port=5545

**\*\*\*\* if you donot specify generic type – collection will old “Object” type of elements**

List<String> nameList=**new** ArrayList<>(); // Object type

nameList.add("Alex");

nameList.add("Nitin");

// nameList.add(10);

**for**(String o:nameList) {

//System.out.println(o.toString());

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

}

**Sorting is possible for those element’s type that implements Comparable interface.**

**Collection classes , do not support primitive**

**Primitive Wrapper Classes**

int Integer

char Character

float Float

double Double

byte Byte

short Short

long Long

boolean Boolean

int x=10;

Integer iobj=new Integer(x);

OR

Integer iobj = x; // auto boxing

int x=iobj.intValue();

int x=iobj; // auto unboxing

**Exception Handling**

**Unchecked**: no need to handle exceptions rather fix the issue.

**Checked:** must handle the exception

Exceptions are runtime Errors. If it appears in your code, your program will be terminated abnormally.

* Compile Time Error
* Run Time Error
  + Logical Mistakes in your code. **[ RuntimeException ] // unchecked exception**

int[] n=new int[5];

n[5]=100;

* + Error because of Application Issue due to external component. **[ All Exception excluding RuntimeException] // checked**

The database connection failed because of a password change.

If the database server is down

If the files/folder that you are referring, to no longer exist at the given

location.

* + Error due System level issue **[ Error class ] // unchecked**

JVM is corrupted.

Not enough Heap Memory. / RAM is exhausted.

Diagram

Description automatically generated

**throw** statement is used to raise exceptions by coding.

throw new Exception(); // any Exception class object…

class MyException extends Exception{

}

If(someconditions) {

throw new MyException();

}

**Handling Exception**

1. By declaring it
2. try…..catch statement

try{

// code & all dependent code that may throw exception

}catch(ExceptionClassName ex){

// code to execute when exception is caught

}

try{

// code & all dependent code that may throw exception

}catch(ExceptionClassName1 ex){

// code to execute when exception is caught

} catch(ExceptionClassName2 ex){

// code to execute when exception is caught

} catch(ExceptionClassName3 ex){

// code to execute when exception is caught

}

\*\*\*Bottom to Top Hierarchy of classes are from the same hierarchical structure

**Multi-Catch Statement // Java 7 new Features**

try{

}catch(Exception1 | Exception2 | Exception3 exob){

}

\*\*\*\* All exceptions in list must be from a different hierarchy.

**Important Notes**

1. If you don’t want to handle the exception bypass it to the caller of the methods, use

throws statement to declare it with method declaration.

1. Exception is propagated if not caught. If programs get interrupted and terminated because the exception reaches JVM through propagation and interrupts the thread responsible for execution.

**Inner Class / Nested Class**

1. Non-Static Nested Class
   1. Member Inner Class
   2. Local Inner Class
   3. Anonymous Inner Class
2. Static-Nested Class

**Member Inner Class :** A class created within class and outside member method:

class Car{

Engine e=new Engine();

private class Engine{

boolean ignite;

void start(){

ignite=true;

}

}

public void start(int kyecode){

if(keycode==101){

e.start();

}

}

}

Car c=new Car();

c.start(101);

**Local Inner Class:**

A class is created within the member method.

class SavingAccount{

private double balance;

boolean overdraft;

public void transfer(double amount){

if(amount<=balance){

balance=balance-amount;

}else{

If(overdraft){

//EMI Calculator

}

}

class EMICalculator{

private double interest;

public double calculate(double amount){

//logic

return inAmount;

}

}

}//end of transfer method

}

**Anonymous Inner Class**

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

**public** **boolean** test(**int** n);

}

**public** **class** Main {

**public** **static** **int** sum(**int**[]numbers, Calculator c) {

**int** total=0;

**for**(**int** v:numbers) {

**if**(c.test(v)) {

total+=v;

}

}

**return** total;

}

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

**int**[] numbers= {1,2,3,4,5,6,7,8,9,10};

//sum of all even numbers

**int** result=*sum*(numbers,**new Calculator() {**

**@Override**

**public boolean test(int n) {**

**return(n%2==0);**

**}**

**}**);

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

//sum of all odd numbers

**Calculator c=new Calculator() {**

**@Override**

**public boolean test(int n) {**

**return(n%2!=0);**

**}**

**};**

result=*sum*(numbers,c);

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

}

}

Lambda : It provides an implementation for interfaces that have a single abstract method(SAM).

An interface that contains only one abstract method is known as a **Functional Interface.** It doesn’t generate byte code for the implementation logic like anonymous

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

**int**[] numbers= {1,2,3,4,5,6,7,8,9,10};

//sum of all even numbers

**int** result=*sum*(numbers,(**int** n)->{ **return**(n%2==0);});

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

//sum of all odd numbers

Calculator c=(**int** n)-> {

**return**(n%2!=0);

};

result=*sum*(numbers,c);

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

//sum of all even numbers

result=*sum*(numbers,(**int** n)-> {

**return**(**true**);

});

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

//sum of all numbers > 5

result=*sum*(numbers,(**int** n)-> {

**return**(n>5);

});

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

}

**A simplified version of code**

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

**int**[] numbers= {1,2,3,4,5,6,7,8,9,10};

//sum of all even numbers

**int** result=*sum*(numbers,n->n%2==0);

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

//sum of all odd numbers

Calculator c= n-> n%2!=0;

result=*sum*(numbers,c);

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

//sum of all even numbers

result=*sum*(numbers,n-> **true**);

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

//sum of all numbers > 5

result=*sum*(numbers,n-> n>5);

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

}

**Functional Interface**

**java.util.function**

this package contains only functional interfaces.

1. **Predicate<T>**

interface Predicate<T>{

public boolean test(T t);

}

1. **Consumer<T>**

Interface Consumer<T>{

public void accept(T t);  
}

Consumer<Integer> c= n -> System.out.println(n);

1. **Supplier<T>**

Interface Supplier<T>{

public T get();  
}

Supplier<PeerSingleton> s=()-> PeerSingleton.getInstance();

PeerSingleton p=s.get();

1. **Function<T,R> : transform the value from one type to other**

Interface Function<T,R>{

public R apply(T t);

}

Function<Employee,Double> f=(e)-> e.getSalary() ;

double sal=f.apply(e1);

Function<Clothing,Double> f=(c)-> c.getPrice();

clothList.forEach(c->System.***out***.println(f.apply(c)));

1. **IntPredicate**

interface IntPredicate{

public boolean test(int t);}

1. **LongPredicate**
2. **BiPredicate<T,U>**

**\*\*\* Refer Oracle Documentation for more details on Primitive & Bi – Variant of Functional Interfaces.**

Predicate<Integer> pr=(Integer n)->{

return(n%2==0);

};

OR

Predicate<Integer> pr=n-> n%2==0;

**Java 8 Enhancement In Interface**

**Default method**: concrete method or method with body

interface A{

public void m1();

public default void m2(){

//----

}

}

**Static Method:**

interface A{

public void m1();

public default void m2(){

//----

}

public static void m3(){

//---

}

}

Class Sample implements A, B{ //

}

Sample s=new Sample();

s.m2();

**Enum**

**package** com.example;

**public** **enum** PowerStatus {

***OFF***("System is turned off"),

***ON***("System is turned on"),

***SLEEP***("System is in sleep mode");

**private** String desc;

**private** PowerStatus(String s) {

**this**.desc=s;

}

**public** String getDesc() {

**return** desc;

}

}

**public** **class** Computer {

**private** PowerStatus status; // 0=OFF, 1=ON, 2=SLEEP

**public** PowerStatus getStatus() {

**return** status;

}

**public** **void** setStatus(PowerStatus status) {

**this**.status = status;

}

}

**import** **static** com.example.PowerStatus.***OFF***;

**import** **static** com.example.PowerStatus.***ON***;

**import** **static** com.example.PowerStatus.***SLEEP***;

**public** **class** Main {

**public** **static** **void** main(String... args) {

Computer c=**new** Computer();

c.setStatus(***OFF***);

// System.out.println(c.getStatus().ordinal());

System.***out***.println(c.getStatus());

System.***out***.println(c.getStatus().getDesc());

c.setStatus(***ON***);

System.***out***.println(c.getStatus().name()); // default

System.***out***.println(c.getStatus().getDesc());

c.setStatus(***SLEEP***);

System.***out***.println(c.getStatus());

System.***out***.println(c.getStatus().getDesc());

}

}

**Comparable & Comparator**

**Comparable : java.lang**

* Single Sort option in a class and that is for natural ordering. This means we can provide a built-in sort algorithm to a class. Therefore, all the objects used the same algorithm if sorted.

Integer and String classes have already implemented Comparable.

**Comparator: java.util**

It is external to class and can provide multiple sort option

**// Refer WalkingApp for example.**

**Introduction to Stream**

**Stream API**

A stream is a sequence of Elements.

1. Source
   1. Collection Objects.
2. Intermediate Operation
   1. filter
   2. map
3. Terminal Operation
   1. forEach
   2. count
   3. sum
   4. many other….!

**MYSQL Database**

MySQL Community Edition

<https://dev.mysql.com/downloads/>

Database Management System is a software that is used to define, create and maintain database. This provides controlled access to the data.

Relational Database Management System – Advance Version of DBMS. – Data is stored in the form of tables that are related to each other. Normalization is present. It presents redundancies

**Prof E.F Codd had defined 12 rules for RDBMS**.

1 Nitin 20120.00 101 201

2 Dereje 23020.00 101 201

201 Manizha 32010.00 101 0

101 Cloud Ops Jersey City

102 HR New York

show databases OR show schemas

create database batch2 OR create schema batch2

use batch2; // make your database a current database;

show tables; // to list all tables in your current database;

**describe tablename // show you the structure of table**

**Select**

**Select [\* | col1, col2, col\_N ] from table\_name [order by col [asc|desc] ] [ where condition ]**

select \* from city;

select \* from city limit 10;

select name, population from city limit 10;

select name, population, population \* 1.01 as NewPopulation from city limit 10;

select name, population, population \* 1.01 as NewPopulation from city order by population limit 10;

select name, population, population \* 1.01 as NewPopulation from city order by population desc limit 10;

**select \* from city where countrycode='NLD';**

**Create Table TableName(**

**ColumnName Type [ Constraint ],**

**ColumnName Type ,**

**ColumnName Type**

**)**

**CREATE TABLE EMPLOYEE(**

**EMP\_ID int primary key,**

**EMP\_NAME varchar(20) not null,**

**SALARY float default 0 ,**

**DEPT\_ID int**

**) ;**

**insert into employee(EMP\_ID, EMP\_NAME, SALARY, DEPT\_ID) values(1,'NITIN',2300.00,101);**

**insert into employee values(1,'NITIN',2300.00,101);**

**delete from employee; // delete all records**

**delete from employee where emp\_id=1;**

**String Data Types**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| **CHAR(size)** | **A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1** |
| **VARCHAR(size)** | **A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum column length in characters - can be from 0 to 65535** |
| **BINARY(size)** | **Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1** |
| **VARBINARY(size)** | **Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.** |
| **TINYBLOB** | **For BLOBs (Binary Large OBjects). Max length: 255 bytes** |
| **TINYTEXT** | **Holds a string with a maximum length of 255 characters** |
| **TEXT(size)** | **Holds a string with a maximum length of 65,535 bytes** |
| **BLOB(size)** | **For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data** |
| **MEDIUMTEXT** | **Holds a string with a maximum length of 16,777,215 characters** |
| **MEDIUMBLOB** | **For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data** |
| **LONGTEXT** | **Holds a string with a maximum length of 4,294,967,295 characters** |
| **LONGBLOB** | **For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data** |
| **ENUM(val1, val2, val3, ...)** | **A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them** |
| **SET(val1, val2, val3, ...)** | **A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list** |

**Numeric Data Types**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| **BIT(*size*)** | **A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.** |
| **TINYINT(*size*)** | **A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)** |
| **BOOL** | **Zero is considered as false, nonzero values are considered as true.** |
| **BOOLEAN** | **Equal to BOOL** |
| **SMALLINT(*size*)** | **A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)** |
| **MEDIUMINT(*size*)** | **A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)** |
| **INT(*size*)** | **A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)** |
| **INTEGER(*size*)** | **Equal to INT(size)** |
| **BIGINT(*size*)** | **A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)** |
| **FLOAT(*size*, *d*)** | **A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions** |
| **FLOAT(*p*)** | **A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()** |
| **DOUBLE(*size*, *d*)** | **A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter** |
| **DOUBLE PRECISION(*size*, *d*)** |  |
| **DECIMAL(*size*, *d*)** | **An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.** |
| **.DEC(*size*, *d*)** | **Equal to DECIMAL(size,d)** |

**Date and Time Data Types**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| **DATE** | **A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'** |
| **DATETIME(*fsp*)** | **A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time** |
| **TIMESTAMP(*fsp*)** | **A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP in the column definition** |
| **TIME(*fsp*)** | **A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'** |
| **YEAR** | **A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.** |

**// Normalization // Self – learning [ until 3NF ]**

**// constraint type // self – learning**

**// UNION [ALL] / CROSS JOIN**

**Cardinality/Relationship**

1. One To One
2. One To Many
3. Many To One
4. Many To Many

Graphical user interface, application

Description automatically generated with medium confidence

**JOIN QUERY**

1. INNER JOIN
2. LEFT OUTER JOIN
3. RIGHT OUTER JOIN

**INNER JOIN**

SELECT COL\_LIST FROM TABLE1 **INNER** JOIN TABLE2 ON TABLE1.COL=TABLE2.COL;

OR

SELECT COL\_LIST FROM TABLE1 JOIN TABLE2 ON TABLE1.COL=TABLE2.COL;

SELECT e.EMP\_ID, e.NAME, e.SALARY, e.DEPT\_ID , d.NAME AS "DEPT\_NAME", d.LOCATION FROM **EMPLOYEE e JOIN DEPARTMENT d ON e.DEPT\_ID=d.DEPT\_ID**;

CREATE VIEW EMP\_DEPT\_VIEW AS

SELECT e.EMP\_ID, e.NAME, e.SALARY, e.DEPT\_ID , d.NAME AS "DEPT\_NAME", d.LOCATION FROM **EMPLOYEE e INNER JOIN DEPARTMENT d ON e.DEPT\_ID=d.DEPT\_ID**;

SELECT \* FROM EMP\_DEPT\_VIEW;

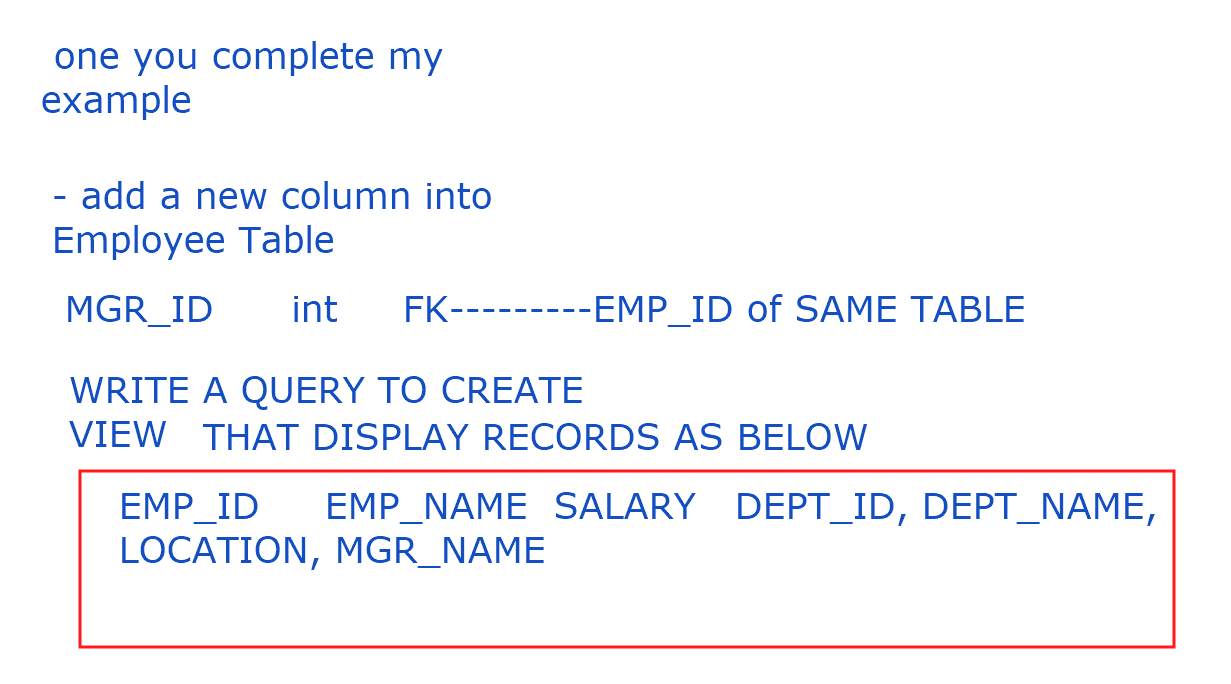
**LEFT OUTER JOIN :** SHOWS ALL RECORDS FROM LEFT TABLE and ONLY MATCHING FROM RIGHT

SELECT e.EMP\_ID, e.NAME, e.SALARY, e.DEPT\_ID , d.NAME AS "DEPT\_NAME", d.LOCATION FROM EMPLOYEE e **LEFT OUTER JOIN** DEPARTMENT d ON e.DEPT\_ID=d.DEPT\_ID;

**RIGHT OUTER JOIN :** SHOWS ALL RECORDS FROM RIGHT TABLE and ONLY MATCHING FROM LEFT

SELECT e.EMP\_ID, e.NAME, e.SALARY, e.DEPT\_ID , d.NAME AS "DEPT\_NAME", d.LOCATION FROM EMPLOYEE e **RIGHT OUTER JOIN** DEPARTMENT d ON e.DEPT\_ID=d.DEPT\_ID;

**PRACTICE**



**Connecting Java to Database**

Diagram

Description automatically generated

**JDBC API: Java Data Base Connectivity API.**

**java.sql** contains classes & interfaces that can help you to connect to the database and execute SQL commands to read from or write to the database server.

DriverManager Class : Bootstrap loader class to load Database driver and specify the database credential. It returns Connection Object.

**public** **class** Main {

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

String jdbcURL="jdbc:mysql://localhost:3306/javaoursoul2";

//Class.forName("com.mysql.cj.jdbc.Driver");

List<Employee> empList=**new** ArrayList<>();

**try** {

Connection con=DriverManager.*getConnection*(jdbcURL,"root","admin#123");

Statement st=con.createStatement();

ResultSet rs=st.executeQuery("select \* from employee");

**while**(rs.next()) {

**int** id=rs.getInt("EMP\_ID"); //int id=rs.getInt(1); //

String name=rs.getString("name");

**double** salary=rs.getDouble("salary");

**int** did=rs.getInt("DEPT\_ID");

empList.add(**new** Employee(id,name,salary,did));

}

} **catch** (SQLException e) {

System.***out***.println(e.toString());

//e.printStackTrace();

}

empList.forEach(e->System.***out***.println(e));

}

}

**Data Access Object Design Pattern**

PreparedStatement

CallableStatement

Closing Connection