Notes

Topics:

* Core Java
* Hibernate
* Spring Framework

Core Java Softwares required

* JDK 8
* IDE (Integrated Development Environment) - Eclipse 2020+

Java: Is a platform independent programming language & object oriented programming language.

Platform Independent: It can run on multiple platforms without recompiling

Object Oriented Language: You create real world entities which are objects, they will have properties & behaviours.

Building blocks of any Object Oriented Language

1. Class: A Blueprint or a template of an object
2. Object: Created from the class, it is an instance of the class.

Syntax to create the class:

class ClassName {   
 // properties   
 // behaviours  
}

Fundamentals of Java

* Datatypes
* Variables
* Operators
* Conditional Statements
* Arrays
* Classes & Objects

Datatypes: They help you to store different type of value in a variable

Types of datatypes

1. Primitive types - byte(1), short(2), int(4), long(8), float(4), double(8), char(2), boolean(1)
2. Non-Primitive types - Unkown

1 byte = 8 bits

byte x = 2;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MSB(0/1) | 0 | 0 | 0 | 0 | 1 | 0 |

byte y = 127;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MSB(0/1) | 1 | 1 | 1 | 1 | 1 | 1 |

128 = MSB 1000 0000

Range of 1 byte is -128 to +127

Operators:

+, -, \*, /, %, =, ++, --, <, >, <=, >=, !=

&&, ||

% Modulus operator: It returns the reminder

2 % 3 returns 2

3 % 3 returns 0

4 % 3 returns 1

Control statements: if, if-else, if-else if-else if-….. else, switch, for, while, do-while

int marks = 50;

if(marks >= 70) { // print FCD }

else if (marks < 70 && marks >= 60) { // print FC }

else if (marks < 60 && marks >= 50) {// print SC }

else if(marks < 50 && marks >= 35) { // print pass }

else { // print Failed }

switch: It is used to apply conditions based on options it is similar to if - else if … else

Syntax:

switch(options) {   
 case option: …… break;  
 case option: ….. break;  
 default: ….  
}

Loops: It is used to execute same statements again and again until some condition is satisfied

* for loop
* while loop
* do while loop

Arrays:

It is used to store multiple values in the same container/variable

ex: Storing multiple users in the same variable, Storing all the days in a variable, Storing all the departments names in a variable

type[] variableName = {value1, value2, …. }

type[] variableName = new type[size];

Suppose you have department numbers

int[] departmentNumbers = {10, 20, 30, 40}

String[] genders = {“Male”, “Female”}

genders[0] will return Male

genders[1] will return Female

genders[2] will throw Exception

departmentNumbers[4] will throw Exception

departmentNumbers[3] will return 40

deparmentNumbers.length will return 4

Enhanced for loop:

It is a simple way of writing for loops to iterate the arrays

Syntax:

for(type variable : collection) {  
  
}

Classes & Objects:

Classes are templates that describe object and object is an instance created from the class.

What all the things we can write inside class

1. methods
2. variables
3. constructors

Methods: These will have logics

Variables: These will store data

Constructors: These will initialize variables, by default class will have a default constructor if no constructors are specified explicitly.

Ex:

class A {   
 // variables  
 int x;  
 // constructors  
 A() {   
 // statements  
 }  
 //methods  
 void test() {   
 // statements  
 }   
}

this: It is a keyword used to access the object properties when their name is same as parameters name (local variables)

Initializer blocks: These are automatically executed without any explicit invocation, we have two types of initializer blocks

1. Static Initializer Block: Executed only once at the time class loading
2. Instance Initializer Block: Executed on each object creation regardless of the type of constructor use

Types of variables

There are 3 types of variables

1. Local Variable: Visible within the context like inside method, inside a block
2. Instance Variable: Every object will have their own copies of instance variable
3. Static Variable: All the objects share single copy of static variable.

OOPs principles

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Encapsulation: It hides the data i.e., object properties and the only way you can access it is using public methods like setters (writing properties) & getters (reading properties)

public class Employee {   
 private int id;  
 private String name;  
 \*\*\* Employee(int id, String name) { …. }  
 public void setName(String name) { this.name = name; } // updates name  
 public int getId() { return id; } // returns id  
 public String getName() { return name; } // returns name  
}   
Employee e1 = new Employee(….)

// outside Employee class this is not possible

e1.id = 112; e1.name = “Raj”; // error

// outside Employee class this is possible

e1.getId(); e1.setName(“Rajesh”); e1.getId();

Short cut to generate setters & getters

Right Click -> Source -> Generate Setters & Getters -> Choose setters & getters for each properties

Inheritance:

Acquiring properties & behaviours of an object from another object

Constructor calls in Inheritance:

Every subclass constructors automatically calls the default constructor of their super class implicitly, i.e., super() is added by compiler in every sub-class constructor, however you can use super(args) to explicitly call argument constructor.

ex: super(name, gender) calls super class constructor taking 2 arguments matching to the name & gender types

Activity:

Create a Student class that inherits Person, the Student must have properties rollno & grade, create a constructor that accepts rollno, name, gender and grade, initialize all the properties properly, from main method create a student object & print student information via getter.

Polymorphism:

A method with many forms, single method that can perform multiple operations

Ex: Power button is polymorphic

There are two ways to achieve polymorphism

1. method overloading
2. method overriding

// method overloading

class Calculator {   
 void add(int x, int y) { }

void add(double x, double y) { }   
}

// method overriding

class A {   
 void test() { }   
}

class B extends A {  
 void test() { } // test is overridden   
}

B b = new B();   
b.test(); // calls test method in B if overridden else calls test method in A

In Person, Employee & Student we can create display() method that prints 3 different informations based on the object you are using to call display();

Access modifiers in Java

There are four access modifiers

1. private: Visible only within the class
2. doesn’t have any keyword - default access / package scope: Visible only within the package
3. protected: Visible within the package & outside the package only to the subclass i.e., outside you can only inherit
4. public: Visible to all

final keyword:

It is used on variables, methods & classes

final variable: It creates constant, it can’t be modified

final int PI = 3.14; you can’t modify PI value now.

final methods: It can’t be overridden in the sub-class

class A {   
public final void test() { }   
}

class B extends A {   
 public void test() { } // error!  
}

final class: It can’t be inherited

final class A { }

class B extends A { } // error!

Abstraction: Hiding the complexity from the user & showing only the necessary details i.e., only method signature to the developers & hiding their implementations

Note: With abstraction the code will become flexible for changes

Abstraction is achieved in two ways

1. interface: Complete abstraction: All the methods abstract
2. abstract class: Partial abstraction: 0 or more methods can be abstract

abstract methods: Methods that are declared but not implemented

Object class:

It is the root class in Java, if a class doesn’t extend any class then it automatically extends Object, every class will have methods of Object class, some of the methods are

toString(): It returns the object data in string format, by default it returns address with class name i.e., com.example.Employee@798aef

equals(): It is used to compare objects, by default it compares objects address

hashCode(): It is used to store objects through hash code, by default it returns memory address in integer

Math class:

Math is a predefined class which has methods that perform mathematical operations all the members of Math class are static, so you can use classname to access Math class members

i.e., Math.PI, Math.E, Math.sqrt(25), Math.pow(2, 3), Math.random()

Static Imports: It is used to import static members so that you don’t have to use class names to access the members instead you use the members directly

import static java.lang.Math.PI;  
import static java.lang.Math.sqrt;

You can use PI and sqrt without using Math.PI & Math.sqrt

Var args: Varying arguments are used to pass 0 or more arguments to a method it is similar to array but in case of array arguments its mandatory to pass the argument, however in var args its optional

void test(int[] a) { }, we need to pass array argument mandatorily

void demo(int… a) { }, we can pass 0 or more arguments to the demo

i.e, demo(2, 3, 4); // ok

demo(); // ok

demo(4, 5, 6, 7, 8, 9); //ok

If you want to call test you will pass array and its mandatory

test(new int[]{1, 2, 4}); // ok

test(); // Error!

JAR: Java Archive, it wraps all the packages & classes into single artifact file which can be reused in any projects and also you can share this jar to the public so that they can use your classes by adding the jar to their project class path

Classpath: It is an environment that recognizes the packages & classes

How to create jar file in Eclipse

Project -> Export -> Jar -> create jar

How to add jar to the project classpath

Project -> Build path -> Select the jar -> Finish

Note: We use now a days build tools like Maven or Gradle to create jars and add jars, the above approach is not recommended.

String related classes

* String
* String Buffer

String: It is used to create Immutable Strings, i.e., Once a string object is created it can’t be modified

String Buffer: It is used to create Mutable Strings, i.e., you can modify the strings

String s1 = “HELLO”;  
String s2 = “hello”;

System.out.println(s1); // HELLO

s1.concat(“1234”); // HELLO1234

System.out.println(s1); HELLO  
System.out.println(s1.equals(s2)) // false  
System.out.println(s1.equalsIgnoreCase(s2)); // true

s1 = s1.concat(“1234”); // HELLO1234

System.out.println(s1); HELLO1234

Note: Once string is modified it doesn’t change the existing string content, instead it creates a new string object

String s1 = “hello1234”;  
String s2 = “hello1234”;  
String s3 = “hello”;  
String s4 = “1234”;  
String s5 = s3.concat(s4); >> hello1234

String Buffer:

It is used to create mutable string objects, it has methods like append(), delete(), insert() and so on

StringBuffer sb1 = new StringBuffer(“hello”);  
StringBuffer sb2 = new StringBuffer(“hello”);  
System.out.println(sb1); // hello  
System.out.println(sb2); // hello

sb1.equals(sb2); // false, because equals method is not overridden to compare content, it calls Object class equals that compares memory address

sb1.append(“12345”); modifies the hello to hello12345

Wrapper classes

In Java we have a wrappers to wrap primitives to objects, primitives are not objects i.e., they don’t have any methods like we have it in String, StringBuffer and so on, with primitives you can only perform arithmetic operations, but if you convert it into object then you can perform arithmetic operations + other operations like converting from String to int to String, double to String to double, number to hexadecimal, number to binary, Identifying maximum & minimum value primitives support and so on.

We have wrapper classes for each primitives

int >> Integer  
byte >> Byte  
short >> Short  
long >> Long

double >> Double  
float >> Float

char >> Character

boolean >> Boolean

Autobox: Converting from primitive type to object

Autounbox: Converting from object to primitive type

Ex of AutoBox

int a = 20; // primitive  
Integer b = a; // a will be automatically boxed to Integer object, now you can access methods of Integer class from ‘b’

Ex of AutoUnbox

Integer c = 20; // auto box

int d = c; // auto unbox

Note: Autobox & Autounbox are useful in collection framework

Byte.MAX\_VALUE >> 127

Byte.MIN\_VALUE >> -128

Short.MAX\_VALUE >> 32767

Short.MAX\_VALUE >> -32768

Integer.MAX\_VALUE >> 21474,83,647

Integer.toHexString(15) >> f

Integer.toBinaryString(7) >> 111

Try out the above statements and print their values

Exception Handling:

Exception: Runtime errors which terminates the program abnormally if not handled

Java has provided 5 keywords that can be used in Exception Handling mechanism.

* try
* catch
* finally
* throw
* throws

try block: Use this block to write the statements that might cause exceptions like reading/writing files, interacting with the database, connecting to external resources

catch block: It handles exceptions generated from try block

ex:

try { /\*some code\*/ }   
catch (ExceptionType e) { /\*some code\*/ }

Note: After try block you can have more than one catch block can be written

i.e., try … catch .. catch… catch….

finally block: It is definitely executed regardless of whether exception handled or not

Note: finally must be written either after try or after catch

Valid: try … finally

Valid: try … catch … catch … finally

Valid: try .. catch … finally

Invalid: finally …. Just a finally block without try / catch is invalid

throw: It is used to manually generate the Exception

Ex: *throw* new ExceptionName();

throws: It is used to propagate the exceptions to the caller so that caller would use the try-catch to handle the exception

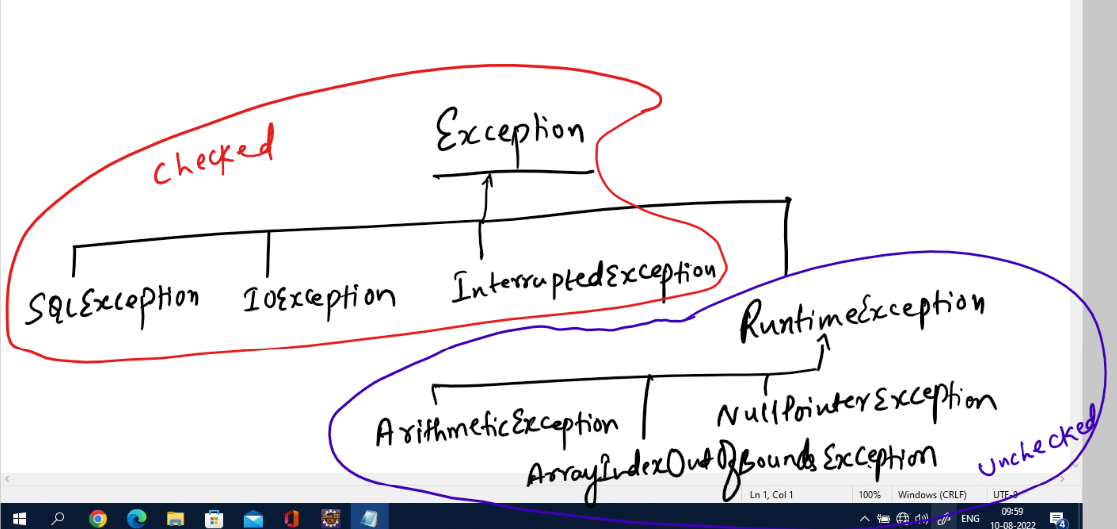
Ex:   
public void debit(double amount) throws ExceptionName { … }

callerOfDebit() {   
 try { ………  
 debit(amount);   
 }  
 catch(ExceptionName ex) {  
 // some codes   
 }   
}

There are two types of exceptions

1. Checked Exceptions: Should be handled at the compilation time
2. Unchecked Exceptions: Ignored at the compilation time

Checked Exceptions are something not under programmers hand to control, however Unchecked Exceptions can be avoided programmatically



Custom Exceptions

Creating our own Exceptions as per the business requirement, you can extend any one of the Exception classes

1. To create checked exception extend Exception class
2. To create unchecked exception extend RuntimeException class

public InsufficientBalanceException extends Exception {   
 // create constructors to initialize the error message   
}

debit(double amount) throws InsufficientBalanceException {

if(amount > balance) {   
 *throw* new InsufficientBalanceException(“Amount is: “+amount+”, balance is: “+balance);  
}

IO Streams

It is used to perform read and write operations, there are 2 types of streams

1. Byte Streams: ASCII data/Binary data
2. Character Streams: Character data

List of classes in Byte Streams

* InputStream, OutputStream (abstract class)
* FileInputStream, FileOutputStream (read/write files)
* ObjectInputStream, ObjectOutputStream (read/write objects)
* DataInputStream, DataOutputStream (read/write primitives)

List of classes in Character Streams

* Reader, Writer (abstract class)
* FileReader, FileWriter (read/write files i.e., text data)
* InputStreamReader, OutputStreamWriter (bridge between byte-stream & character stream)
* BufferedReader, BufferedWriter (read/write memory)
* PrintWriter (can write to any destination like console, browser, network and etc)

RandomAccessFile: A class that can perform both read/write operation

File: It takes care of creating/deleting files & folders

Note: All these classes are part of java.io package

Note: Many of the methods throw checked exception

FileReader & FileWriter can read/write one character at a time, it degrades the performance of the application as it needs to fetch one character return it & write it then it has to jump to the next character and follow the same operations till it reaches end of file

To increase the performance of read/write we can use BufferedReader and BufferedWriter, because it can read / write one line at a time by loading all the contents of the file in the buffer.

FileReader fileReader = new FileReader(filename)  
BufferedReader buffRead = new BufferedReader(fileReader);

FileWriter fileWriter = new FileWriter(filename);  
BufferedWriter buffWrite = new BufferedWriter(buffWrite)

String line = null;  
do {   
 line = buffRead.readLine(); // reads one line of content in the buffer  
 if(line!= null) buffWrite.write(line); // writes the line read to the file  
} while(line != null);

Reading & Writing Complex Objects

Serialization: Writing the complex data/objects to the file

Deserialization: Reading the complex data/objects from the file

The classes we need to use are:

ObjectOutputStream to write objects

ObjectInputStream to read objects

FileOutputStream should be used to write the objects to file via ObjectOutputStream, FileInputStream should be used to read the objects from the file via ObjectInputStream

Serializable interface: It is an interface used to mark the objects that wants to be serialized, Serializable interface is marker interface which doesn’t have any methods but marks the objects to a Serializable type

Ex:   
class Person { } // Person object can’t be serialized, because it doesn’t implement Serializable

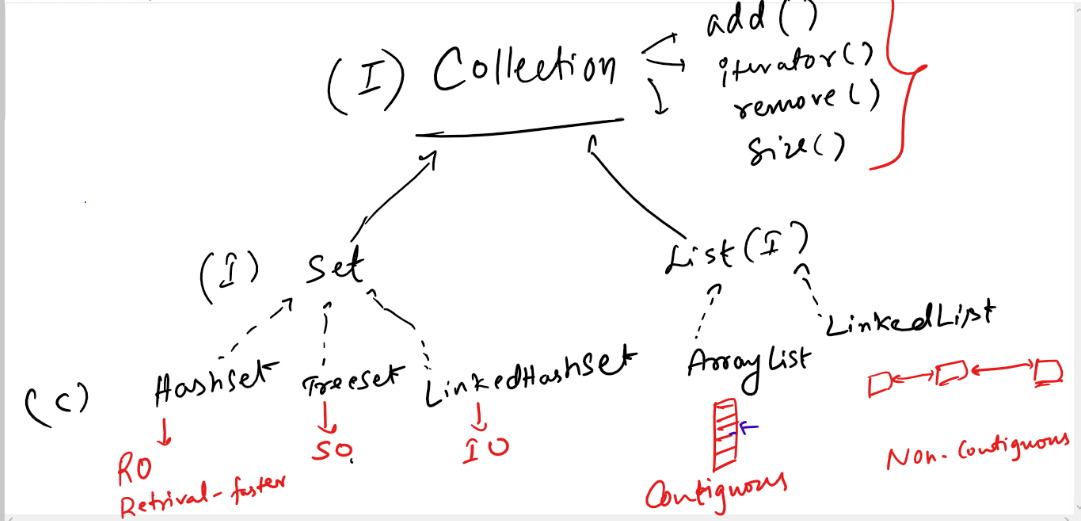
class Employee implements Serializable { } // Employee object can be serialized

Note: ObjectOutputStream checks the objects are of Serializable type before serializing if they are not of Serializable type it throws an Exception

ObjectOutputStream >> writeObject(obj); >> writes the object to the destination

ObjectInputStream >> readObject() >> returns the objects from the source

Collection Framework



Collection is a container to maintain the group of objects, it has methods like add(), iterator(), remove(), size(), clear() and so on they are implemented by different classes in different way

Collection is extended by Set & List both are interfaces, Set supports uniqueness, List supports duplicates

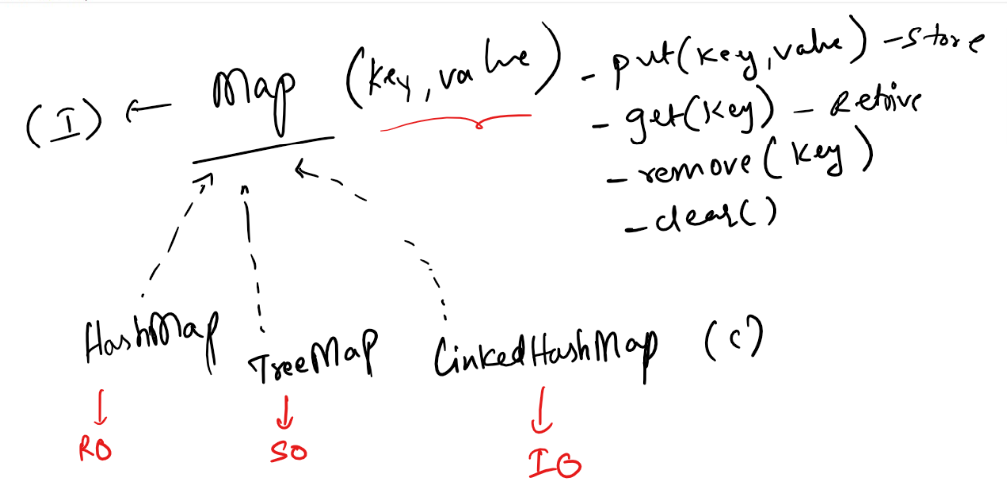
Set implementations are:

1. HashSet: Maintains the data in Random order, retrieval is faster
2. TreeSet: Maintains the data in Sorted order
3. LinkedHashSet: Maintains the insertion order

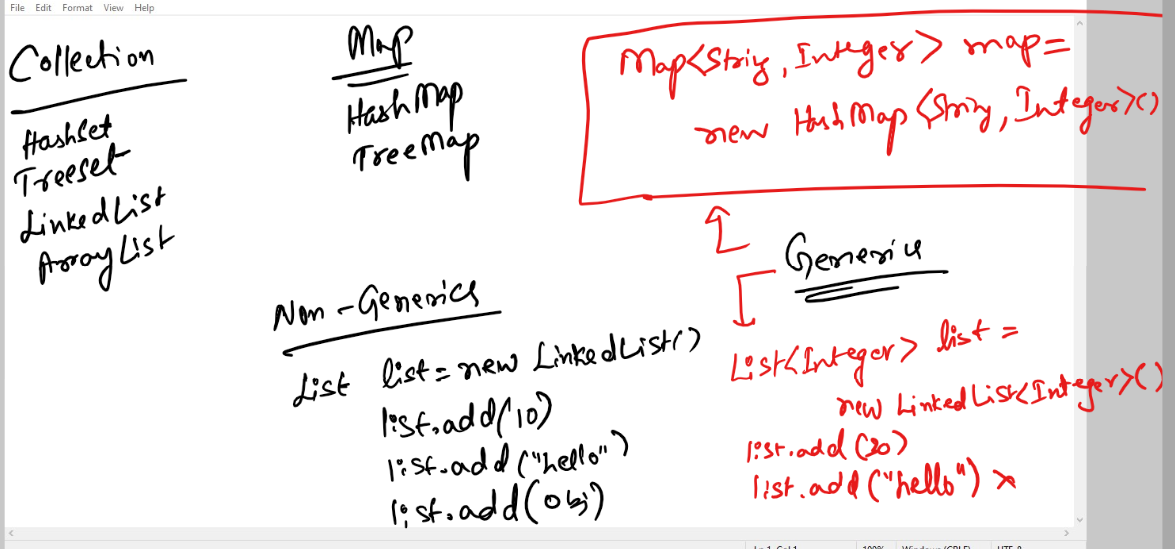
List implementations are:

1. ArrayList: Maintains the data in contiguous memory address, retrieval is faster, however the adding and removing the items are slower because items are moved to different address while adding & removing
2. LinkedList: Maintains the data in non-contiguous memory address, retrieval is slower, however adding and removing is faster compare to array list as it doesn’t shift the elements instead the links between the elements are modified

Map: It is also like collection, but maintains the data in Key & Value pairs, It is an Interface that has 3 implementations like HashMap, TreeMap & LinkedHashMap



Generics: It is for type safety, you can use for Map & Collection both



Sorting complex objects in Collection

Comparable types: It is used to compare one object with another and return a number to the sorting methods, the number returned can be -1 or 0 or +1 based on that number sorting methods would sort the objects with other objects

What is the compare type used?

Comparator is the interface we need to implement that returns the number like -1 or 0 or +1, it has a method called compare(Object, Object) that takes 2 arguments of objects we want to compare

Collections:

It is an utility class which has methods to work with Collection especially List, it has methods like:

sort(List<T>, ComparatorImplementation): Sorts as per ComparatorImplementation  
shuffle(List<T>): Shuffles the list

Activity on IO Streams and Collection Framework

Create a menu of below options in main method

1. Store Employee
2. Retrieve Employee by Id
3. Retrieve All Employees
4. Delete Employee by Id
5. Sort Employees
6. Exit

Show the menu in a loop until you enter “6”  
on Input 1: Ask id, name & salary and initialize employee object and pass the object to save method that must store the Employee object in a file

on Input 2: Ask id and call fetch() method that returns the Employee which matches to id else it throws EmployeeNotFoundException, print that exception in the console.

on Input3: call fetchAll() that returns the List<Employee> which will have all the employees stored in the file, iterate it in main method and print all the employees

on Input4, on Input5: Try it once you implement all the above options

on Input 6: Exit from the program

Note: All the employees must be maintained in a List<Employee> and store that in the file

Multithreading

An application can do more than one tasks simultaneously because of multiple threads, here each thread needs CPU time to perform its work.

In Java we have two API’s in Multithreading

1. Thread class: To create and manage thread objects
2. Runnable interface: Has run() method which is an entry point for the threads

main() method is to start the program and main() is also executed by one thread called as ‘main’ thread.

Some of the methods of Thread class

start(): To registers the thread to get the CPU time

setName(String name): To set name to your threads, by default thread name starts Thread-0, Thread-1, ….

getName(): Returns the name of the Thread

setPriority(int priority): Used to set the priority for threads in terms of getting CPU time early, but it doesn’t work always

There are some static methods

sleep(long s): To give a delay to the thread execution

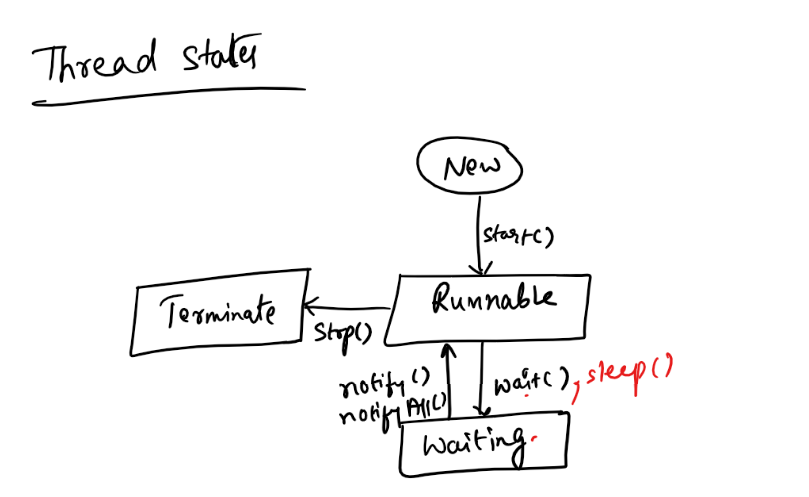
currentThread(): Returns the thread that has CPU time / the thread under execution

Synchronization:

Allowing only one thread to execute a block of code in a multi-threaded program while performing some critical operations, other threads can’t enter in a synchronized block even if they get CPU time, because they need to wait for the thread which entered inside synchronized block to exit.

We have a synchronized keyword to achieve synchronization

Thread States



New: A Newly created thread

Runnable: A thread will go to Runnable state on start() method, or when sleep time is over or when waiting thread is notified by any other thread

Waiting: A thread will go to wait state on wait() method,

Terminated: When the thread completes its task it goes to terminated state

Interthread communication

Thread communicates with other threads in an synchronized environment when they are manipulating a common shared resource like files, memory and so on.

Threads can call Object class methods to communicate like

* wait()
* notify()
* notifyAll()

Assertions:

Assertions are like some assumptions that certain conditions would always work, it is used to check the assumptions that is properly implemented.

In Java you have a keyword assert, if it is true then everything is fine, if assert is false then it throws AssertionError

By Default assertions are not enabled, you to enable using -ea option.

Syntax:   
assert boolean : “some string message”;

Ex:   
assert amount > balance : “Amount > Balance”;

Reflection API

It is used to analyse the classes to figure out the members and constructors like Fields, Methods & Constructors

Class c = Class.forName(“com.example.Employee”);

Method[] m = c.getDeclaredMethods(); // returns array of all the methods

Field[] f = c.getDeclaredFeilds(); // returns array of all the fields

Constructor[] c = c.getDeclaredConstructors(); // return array of all the constructors

Annotations: These are metadata that gives some extra features to the class, variables & methods, some of the inbuilt annotations

1. @Override
2. @SupressWarning
3. @Deprecated: It is marked when you want certain methods to not use in future, ex: Lot of methods in Date class is deprecated

Java 8 features

1. LocalDate, LocalTime classes for date & time informations
2. Static methods & Default Methods in the interface
3. Lambda Expressions
4. Stream APIs

Date & Time

LocalDate.now(): Returns current date in yyyy-MM-dd format

ex: 2022-08-12

LocalTime.now(): Returns current time in hh:mm:ss format

ex: 15:55:30

LocalDate.of(1998, 12, 25): To create specific date, it will stored as 1998-12-25

LocalDate.parse(“yyyy-MM-dd”): To convert string to LocalDate,

LocalDate today = LocalDate.now();  
today.getMonth() >> returns 8  
today.getDay() >> returns 12  
today.getYear() >> returns 2022

Java 8 supports static methods & default methods in interface

interface A {   
  
 public static void m1() { … } // now we can create static method in the interface

public default void m2() { … } // this is the default implementation m2() which you can //change by overriding   
}

Lambda Expression:

It is used to simplify writing the anonymous class, it enables passing function as an argument instead of object, Lambda expression can be written only to the functional interfaces.

Functional Interfaces: These are the interfaces which will have only one abstract method in it, Functional interface concept was introduced to enable functional based programming, where you pass functions as an argument instead of object.

Some of the inbuilt functional interfaces are:

1. Comparator<T>: it has an abstract method compare(T t1, T t2): int
2. Runnable: It has an abstract method run(): void
3. Consumer<T>: It has an abstract method accept(T t): void
4. Predicate<T>: It has an abstract method test(T t): boolean

Implementation for the above functional interfaces with lambda expression

Comparator<T>

Comparator<Employee> comparatorInstance = new Comparator<Employee>() {  
 public int compare(Employee e1, Employee e2) {   
 return Integer.compare(e1.getId(), e2.getId());  
 }  
}   
// lambda expression

Comparator<Employee> comp = (a, b) -> Integer.compare(a.getId(), b.getId());

Lambda Expression for Predicate<T> & Consumer<T>

Consumer<T> c = (t) => sout(t);

Stream APIs

Streams help you to manipulate the data in collections in a declarative way through lambda expressions, declarative way helps you write operations in a simpler & easier way

Ex: SQL queries are declarative hence they are easy and simple

Stream API helps you to perform complex operations involving multiple tasks in a single line like filters, sorting, iterating and etc can be done in one line.

Firstly you need to convert your collection to stream using stream() method that provides methods that gives you another stream like filter(), sort(), map(), and so on

list.stream().filter(predicate).filter(predicate).sort(comparator).forEach(consumer);

the above line does multiple filters, sort and iteration

Predicate<T>: Functional interface that has test(): boolean

Implementation to Predicate: t -> booleanExpression

Comparator<T>: Functional interface that has compare(T t1, T t2): int

Implementation to Comparator: (t1, t2) -> intExpression;

Consumer<T>: Functional interface that has accept(T t): void

Implementation to Consumer: (t) -> SomeStatements

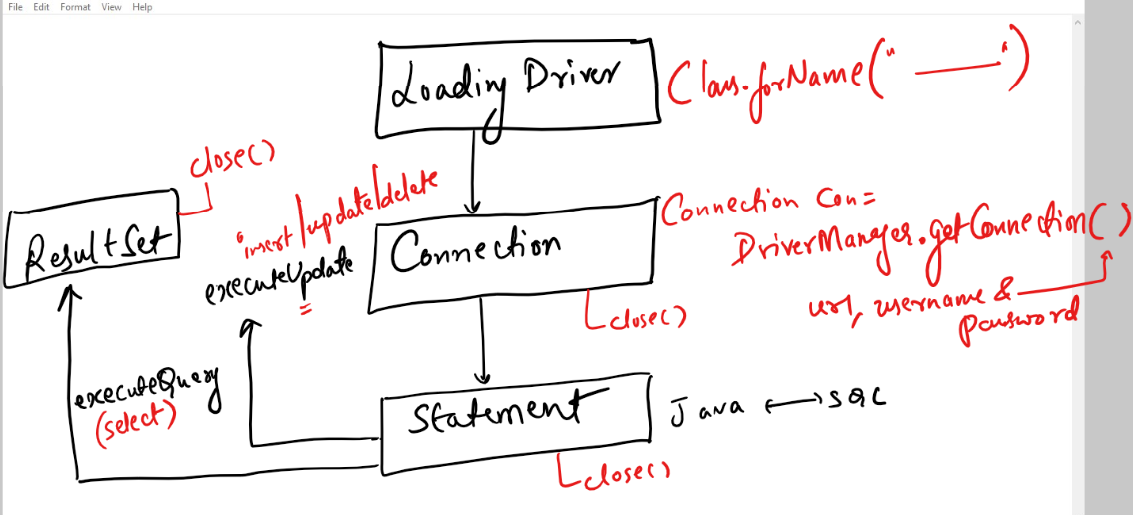
JDBC

Java Database Connectivity, it is an API provided by Java which is implemented every database vendors like MySQL, Oracle, Sybase, Derby, IBM DB2, and so on, these vendors provides the implementation via Jar files which we need to add in our project to interact with the particular Database.

JDBC uses JDBC drivers in order to interact with the database, there are mainly 4 types of drivers

1. JDBC-ODBC driver: It works only in Windows, now it is deprecated, because it was making problem with 64 bit windows
2. Native Protocol driver: It is platform independent, even this is also deprecated
3. Net API driver: It is pure java driver, but it is licensed
4. Thin driver: It is pure java driver, it is provided by database vendor through their jars.

Steps to interact with the database



Code to interact

// Loading the JDBC Driver

Class.forName(Driver.class.getName()) // best way to detect the driver from jar

[ OR ]

Class.forName(“com.mysql.cj.jdbc.Driver”); // another way to detect the driver from jar

// Establishing Connection

Connection connection = DriverManager.getConnection(url, username, password);

// Connection & DriverManager is from java.sql. package

url = jdbc:mysql://localhost:3306/sbig

username = root

password = dabase\_password

// Statement object: It converts Java to SQL

PreparedStatement pstmt = connection.prepareStatement(sqlQuery);

sqlQuery could be ‘insert’, ‘update’, ‘select’ (or) ‘delete’ queries

Ex: insert into employee values (?, ?, ?);

Note: ? is a place holder that maps to the column index, ? position doesn’t always match to column position

Setting values to the ‘?’ using setter methods of PreparedStatement

pstmt.setInt(?\_index, intValue);  
pstmt.setString(?\_index, stringValue);  
pstmt.setDouble(?\_index, doubleValue);

Note: When you set value, set for ? position not for column position, because 1st ? may map to any column position not always to the 1st column position of the table

Note: pstmt.setterMethods() sets value to the ‘?’ in the query, but the query is executed when you call execute methods like executeUpdate() {insert, update, delete} and executeQuery {select}

RowSet:

It is used to extract the results from the database by providing datasource information through setter methods present in ResultSetProvider

* It is much more simpler than using ResultSet
* You don’t’ have to use Connection, PreparedStatement, ResultSet
* RowSet establishes connection with the help of Connection Pool

Connection Pool: It is a container that will have reusable connection objects, you don’t have to close these connections

Hibernate

It is a ORM (Object Relational Mapping) Framework, which helps you to map Java objects to the Database tables.

Hibernate internally takes care of lot of low level operations to make development faster

* Converting Types (Java to SQL to Java)
* Handling Checked Exceptions
* Generating queries for a particular database
* Separating Datasource information’s from the code

hibernate.cfg.xml:

It is a file that will datasource configuration information like database\_url, database\_driver\_class, database\_username, database\_password, database\_dialect, entity\_class and so on

Note: It is a one time configuration file

To read this configuration you need to use Configuration object from hibernate library

Configuration cfg = new Configuration();  
cfg.configure(); // automatically loads hibernate.cfg.xml

[ OR ]

cfg.configure(“someOtherfile.XML”); // if you want other xml files to load

SessionFactory: It maintains connection pool that can be provided in the form Session object

Configuration object reads the configuration file to load the informations, but using it Connection should be established for that we have a SessionFactory object

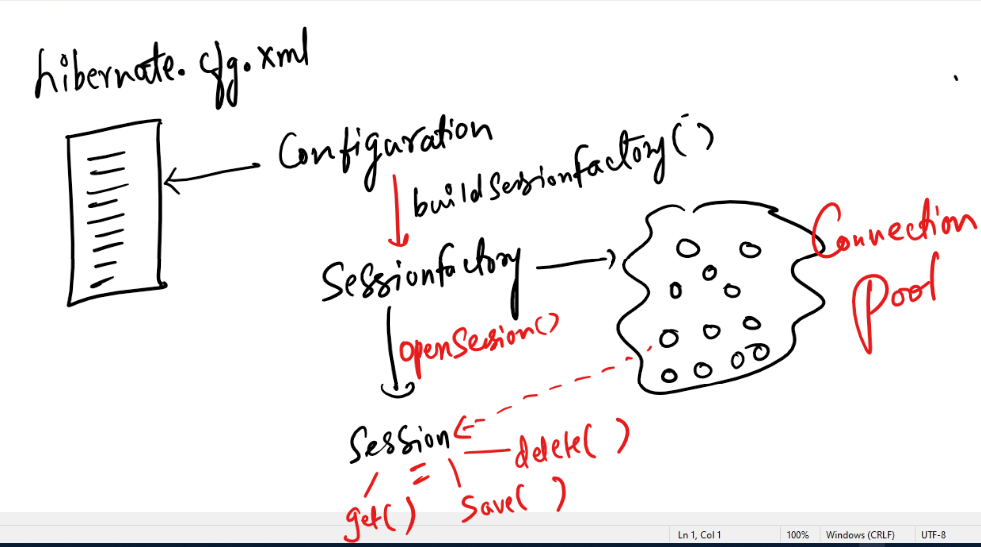
SessionFactory factory = cfg.buildSessionFactory();

SessionFactory will have factory of sessions that can perform CRUD operations, it maintains the connections in Connection pool that can are re-usable

Session:

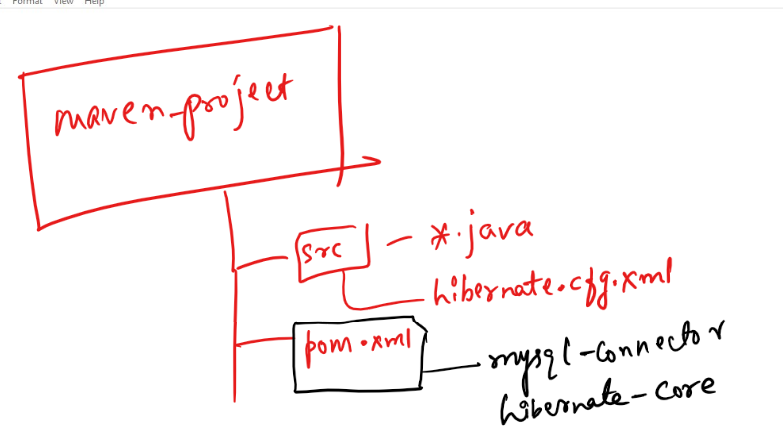
It is used to obtain the Connection object from the Session Factory

Session session = factory.openSession();



All these hibernate libraries are available in hibernate-core artifact of maven

Project Structure



List of methods in Session object

save(entity): To store the entity

get(EntityClass, primaryKey): To retrieve the entity

delete(entity): To delete the entity

update(entity): To update the entity

NamedQuery:

It is to create HQL at the entity level so that you can reuse it in multiple places

*@NamedQuery(name = “findAll”, query = “select e from Employee e”)  
@Entity  
public class Employee { … }*

You can use:

session.createNamedQuery(“findAll”, Employee.class)

It checks the findAll in the Employee entity and replaces the query mapped to it.

NamedQuries:

It is to group multiple @NamedQuery when you want Entity to have multiple queries you can use @NamedQueries( { @NamedQuery(), @NamedQuery()} )

*@NamedQueries({   
 @NamedQuery(name = “findAll”, query = “select e from Employee e”),   
 @NamedQuery(name = “findAllCommonName”,   
 query = “select e from Employee e where e.id = ?1”)  
})  
@Entity  
public class Employee { }*

Query<Employee> query =   
 session.createNamedQuery(“findAllCommonName”, Employee.class)

query.setParameter(1, value);

List<Employee> list = query.getResultList();

Association Mapping

1. @OneToOne
2. @OneToMany
3. @ManyToOne
4. @ManyToMany

Understanding @OneToMany and @ManyToOne with State, City & State\_City tables

State Table

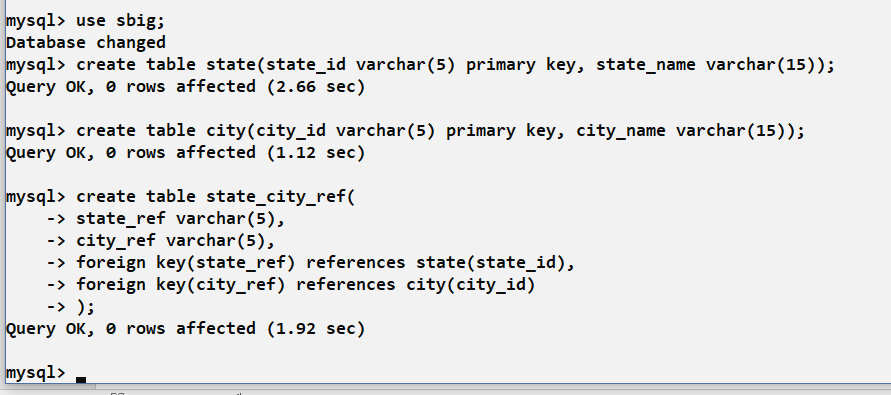
|  |  |
| --- | --- |
| State\_Id (Primary Key) | State\_Name |
| KA | Karnataka |
| MH | Maharastra |

City Table

|  |  |
| --- | --- |
| City\_Id (Primary Key) | City\_Name |
| PUN | Pune |
| MUM | Mumbai |
| BLR | Bangalore |
| MYS | Mysore |

State\_City\_Ref

|  |  |
| --- | --- |
| State\_Ref(Foreign Key) | City\_Ref(Foreign Key) |
| KA | BLR |
| KA | MYS |
| MH | PUN |
| MH | MUM |



@Entity   
public class State {   
 private String stateId  
 private String stateName;  
 *@OneToMany  
 @JoinTable(name = “state\_city\_ref”, joinColumns = “state\_ref”, inverseJoinColumns = “city\_ref)*  
 private List<City> cities;  
}

@Entity   
public class City { }