**Routine**

1. Git
2. Maven
3. Postman
4. Travis CI
5. Jenkins
6. Java
7. Collections
8. Threads

**Git**

Git - Distributed version control system.

GitHub – Collaboration platform

A place to host and share Git projects

GitHub provides a number of features to collaborate more effectively.

These features include:

1. Issues - used to track bugs and feature requests
2. Pull Requests - adding, modifying, or deleting files
3. Projects – visualize the work in Kanban style

Creating Issue, Branch

Pull – Getting it from git

Push – loading it into Git

Pull request – WIP (Work in Progress)

Code review - One of the best ways to ensure code quality is to make peer reviews a part of every Pull Request

Merge – taking the content and history from feature branch and adding it to the content and history of the main branch

Merge conflict – error while merging to branch after making the changes.

Git stash - Git stash takes the changes made to working tree ("dirty") and puts them in a holding pattern until it is ready to re-apply them, so it is fresh working directory ("clean").

List, Show, Drop, Clear, Pop, Apply, Branch, Create, Store

We can store large file in git using Git LFS

**Maven**

Maven - Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework

Maven provides developers ways to manage the following −

* Builds
* Documentation
* Reporting
* Dependencies
* SCMs
* Releases
* Distribution
* Mailing list

A comprehensive model for projects, which is reusable, maintainable, and easier to comprehend. Plugins or tools that interact with this declarative model.

POM - POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml

POM file may consists of

* project dependencies
* plugins
* goals
* build profiles
* project version
* developers
* mailing list

Example:

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

xmlns:xsi = "http://www.test.com/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.companyname.project-group</groupId>

<artifactId>project</artifactId>

<version>1.0</version>

</project>

Build profile

1. Per Project
2. Per user
3. Global

Repository

1. Local
2. Central
3. Remote

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to −

1. create jar file
2. create war file
3. compile code files
4. unit testing of code
5. create project documentation
6. create project reports

Common plugins - Clean, compiler, surefire, jar, war, Javadoc, antrun

**Postman**

Postman - software testing tool which is used for API testing.

To send the API request, we need an HTTP method. Some commonly used methods are POST, GET, DELETE, PUT, and PATCH.

GET: This HTTP method is used to access the data from an API.

POST: This method transmits new data.

DELETE: This is used to remove or delete the existing data.

PATCH: This method is used to update the existing data.

PUT: This method is used to update the existing data.

Steps to send API request

1. select request method
2. setting request URL
3. Authorize request
4. Add headers
5. Send

Response will contain status code, time, size and response body

Response body can be seen in three ways

1. Pretty
2. Raw
3. Preview

We can also use mock servers to test API’s

**Travis CI**

[Refer to this tutorial](https://www.youtube.com/watch?v=Uft5KBimzyk)

**Jenkins**

[Refer to this tutorial](https://www.youtube.com/watch?v=FX322RVNGj4)

**JAVA**

Inheritance, Polymorphism, Abstraction, Encapsulation

Object cloning, Wrapper class

Exception handling - handle the runtime errors so that the normal flow of the application can be maintained

Three types of exception according to Oracle

1. Checked Exception
2. Unchecked Exception
3. Error



Hierarchy of Java Exception classes

Keywords

1. The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
2. The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
3. The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
4. The "throw" keyword is used to throw an exception.
5. The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

We can have multiple catch block

**Collections**

The Collection framework - A unified architecture for storing and manipulating a group of objects.

It has

1. Interfaces and its implementations, i.e., classes
2. Algorithm

**Hierarchy of Collection**

**JAVA ArrayList**

Java ArrayList class uses a dynamic array for storing the elements

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

Example:

import java.util.\*;

public class ArrayListExample1{

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>();//Creating arraylist

list.add("Mango");//Adding object in arraylist

list.add("Apple");

list.add("Banana");

list.add("Grapes");

//Printing the arraylist object

System.out.println(list);

}

}

**JAVA LinkedList**

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

Example:

import java.util.\*;

public class LinkedList1{

public static void main(String args[]){

LinkedList<String> al=new LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

**JAVA HashSet**

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

Example:

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

import java.util.\*;

class HashSet1{

public static void main(String args[]){

//Creating HashSet and adding elements

HashSet<String> set=new HashSet();

set.add("One");

set.add("Two");

set.add("Three");

set.add("Four");

set.add("Five");

Iterator<String> i=set.iterator();

while(i.hasNext())

{

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

}

}

}

**JAVA Linked HashSet**

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

Example:

import java.util.\*;

class LinkedHashSet1{

public static void main(String args[]){

//Creating HashSet and adding elements

LinkedHashSet<String> set=new LinkedHashSet();

set.add("One");

set.add("Two");

set.add("Three");

set.add("Four");

set.add("Five");

Iterator<String> i=set.iterator();

while(i.hasNext())

{

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

}

}

}

**JAVA TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

Example:

import java.util.\*;

class TreeSet1{

public static void main(String args[]){

//Creating and adding elements

TreeSet<String> al=new TreeSet<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

//Traversing elements

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

**Priority Queue**

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

Example:

import java.util.\*;

class TestCollection12{

public static void main(String args[]){

PriorityQueue<String> queue=new PriorityQueue<String>();

queue.add("Amit");

queue.add("Vijay");

queue.add("Karan");

queue.add("Jai");

queue.add("Rahul");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

System.out.println("iterating the queue elements:");

Iterator itr=queue.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

while(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}

**JAVA HashMap**

Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

Example:

import java.util.\*;

public class HashMapExample1{

public static void main(String args[]){

HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap

map.put(1,"Mango"); //Put elements in Map

map.put(2,"Apple");

map.put(3,"Banana");

map.put(4,"Grapes");

System.out.println("Iterating Hashmap...");

for(Map.Entry m : map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

There are many methods defined in the Collection interface. These are as follows:

add() This method returns a Boolean value true if it inserts the specified element in this collection.

addAll() This method returns a Boolean value true if it adds all the elements of specified collection in the invoking collection.

clear() It removes all the elements automatically from this collection.

contains() It returns a Boolean value true if this queue contains the specified element.

containsAll() It returns a Boolean value true if this collection contains all the elements in the specified collection.

equals() This method returns a boolean value true if the specified object is equal with this collection.

hashCode() It returns a hash code value for this collection.

isEmpty() This method returns true if this collection contains no elements or is empty.

iterator() It returns an iterator over the elements in proper sequence.

remove() It removes the specified element from this queue, if it is present in the collection.

removeAll() It removes all the elements of this collection which are also present in the specified collection.

removeIf() It removes all the elements of this collection that satisfy the given predicate filter.

retainAll() This method retains only those elements in this collection that are present in the specified collection.

size() It returns the total number of the elements in this collection.

spliterator() It returns a spliterator over the elements in this collection.

toArray() It returns an array containing all the elements of this collection which are in proper sequence.

**Threads**

Multithread - Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.



Create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface

**File name: Multi.java(extending Thread class)**

class Multi extends Thread{ public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

}

}

**File name: Multi3.java (implementing Runnable interface)**

class Multi3 implements Runnable{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1); // Using the constructor Thread(Runnable r)

t1.start();

}

}

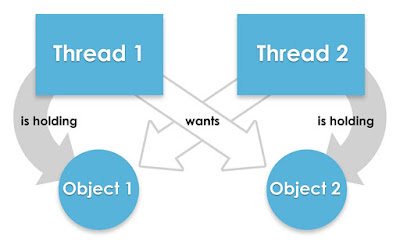
**Thread priority**

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling)

**Java Garbage collection**

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

**How to avoid deadlock in Java?**



**Example 1: Code that will result in dead lock**

/\*\*

\* Java program to create a deadlock by imposing circular wait.

\*

\* @author Radhesh

\*

\*/

public class DeadLockDemo {

/\*

\* This method request two locks, first String and then Integer

\*/

public void method1() {

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

}

}

}

/\*

\* This method also requests same two lock but in exactly

\* Opposite order i.e. first Integer and then String.

\* This creates potential deadlock, if one thread holds String lock

\* and other holds Integer lock and they wait for each other, forever.

\*/

public void method2() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

}

**Example 2: Code that resolves deadlock**

public class DeadLockFixed {

/\*\*

\* Both method are now requesting lock in same order,

\* first Integer and then String.

\* reverse e.g. first String and then Integer even this works,

\* both will solve the problem

\* as long as both method are requesting lock

\* in consistent order.

\*/

public void method1() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

public void method2() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

}