Java

Class can be only public or package-private (default) but not protected or private. No protected because there is nothing such as package-inheritance or sub packages and no private because u cannot instantiate it ever. Inner classes can be protected or private.

Variables are of 3 types :-

1. Reference Variables (local) – for referencing/accessing a data type.
2. Instance Variables – per instance of the object has its instance variable.
3. Class Variables. – class variables are one per class and not per instance

Public, private, protected static

Class, method, interface, variable

Interface methods don’t need the access modifiers as it is dependent on the class that implements it

Intellij tricks

1. sout
2. alt+insert (auto generate getters and setters)
3. psvm
4. refactor ()
5. inselection to make changes only in selected area
6. psfs (private static final Stirng

Variables :-

Key words gone through till now

1. Byte,
2. short,
3. int,
4. long,
5. char,
6. Boolean,
7. float,
8. double,
9. void
10. true,
11. false,
12. null,
13. if,
14. else,
15. While
16. Do
17. For
18. Public
19. Private
20. Protected
21. Static
22. Class
23. Abstract
24. Interface
25. Default
26. Extends
27. Implements
28. Throws
29. Throw
30. Try
31. Catch
32. Finally
33. Import
34. Package
35. New
36. Return
37. Final
38. Switch
39. Case
40. This
41. break

Primitive data types :- byte, short ,int, long, Boolean, char, float, double,

Understanding strings:-

Methods in java

Classes :-

If u want to create class, use “new” keyword

Constructior has the format <access modifier> class\_name(args);

Rest all functions have <access\_modifier> <return\_type> function\_name(args)

Eg:- u can set a default constructor to have the req fields by doing

this(parameters) --- this would call the constructor to set the req fields during class creation and this statement should be the first line in the constructor body

similarly super should be the first argument.

inheritance :-

super() to call the super constructor

super.function\_name() calls the super class function name;

references vs objects vs instances vs class

class – blueprint for an object(dog)

each dog created is an object also known as instance

address of that dogs location is reference

we can pass references as parameters to constructors and methods

u always have references to objects. U cannot access object directly. Everything is done using reference

this vs super

this() can be used only in a constructor

call to this() and super() should be first statement

super is used to access parent class members and methods

this is used to access the current class members and methods.

This and super cannot be used in static blocks and static areas;

Inheritance => is-a relationship

Composition => has-a relationship (one class has its reference inside an other class)

Eg:- class a1 {

Int a,b

}

Class b1 { b1 has a1 (=> composition is achieved i.e has-a relationship)

int c,d;

a1 a;

}

Encapsulation :-

Enclosing the data of that class in itself i.e. having private data variables and using getters to get their values

Polymorphism :-

An object able to take many forms based on the value it has been assigned is called polymorphism

Arrays :-

Only for primitive data types we can do this

Int[] a = {1,2,3,4,5,6,7,8,9,10};

Typecasting is (data\_type)

arraylist :

ArrayList<>

.get(position); .add(element);add(1,2)=>add 2 at position 1 .size(); set(position, new item); .remove(position); .contains(element)(gives Boolean if it has that element or not); indexOf;

addAll -> to add all the elements

arraylist<> array = new arraylist<>(arr) // arr is the array u want to copy

toArray to convert the arraylist to array of objects

LinkedList

LinkedList<T> ; .add(value);add(position,value);.remove(position); getFirst() == “” =>no elements; .empty();

Iterator<T> = <variable>.iterator; iterator.hasNext(); iterator.next();

ListIterator<T> = variabld.listIterator; .hasNext(); .previous()// go to previous//; .next(); previous(); hasPrevious();

Add in the last => use .add() :P

Object.compareTo => gives 0 if equal and greater than 0=> object is bigger and less => object is smaller

Interface :-

Interface variable can be assigned to the class that implements it.

Power of interface is that the variable of interface can be assigned to any thing.

Eg: 1st implementation u use ArrayList. Then to change to LinkedList, u just need to change the place where u assign the new ArrayList to LinkedList if the variable used is an interface variable. Else, u need to modify everywhere.

InnerClass :-

Identical to top level class just that it is packaged in the outer class rather than the package. i.e it cannot accesss the non static members of the outer class without creating the instance of that class.

If outer class and inner class have same variable, to refer to the outer class variable in the inner class, u have to use,

Outerclass.this.variable (as it is shallowed by the inner class variable.)

Instantiating the inner class

Outerclass.innerclass var = instance\_of\_outerclass.new innerclass();

If the inner class is private, only the outer class can create an object of it and not the instance of it.

Similarly for interface instead of classes but the difference is u cannot instantiate an interaface. :P

Anonymous classes :- It is an inner **class** without a name and for which only a single object is created.

Example :- creating a class of an interface/abstract class without putting them in a concrete class is an example of anonymous class.

Abstract Class :-

Difference between interface and abstract class is that it is based on the design of is-a relation ship. This relationship implies abstract class to be used. And for any method help, interface can be used.

Eg :- abstract class bird , interface canFly, as not only birds fly but there are exceptions.(bat, dragonfly)

Abstract class can have member variables but not interfaces(their variables need to be public static final

Interfaces cannot have constructors but abstract classes can have them

All methods of interfaces are public but abstract class can have any visibility.

Abstract classes can have methods implemented but not interfaces (until java8, in java9 using the default key word u can do that)

Generics :-

Class <T>

Class <T extends (some base class)> => only the one’s which extend the base class can have methods under them.

Things u will name in java :-

Packages - always in lower case

Classes --- CamelCase

Interfaces ----- CamelCase

Methods ------ mixedCase

Constants ----- all caps

Variables ------ mixedCase

Type Parameters

Packages :-

Java.lang is automatically imported as it provides the basic functionality.

Import java.awt.\* imports all the classes in that package but not the packages within that package.

Adding jar code

Jar is actually collection of executable java files(class files). If u create a jar file, then just import that jar file and u get all the executables and then u can use them as u want

Importing a package as a jar file. Amd then using it as ur need

Scope :-

Visibility of class, method, variable,

If inner class and outer class have a same variable, it is differentiated using outerclass.this.variable and for inner class, it is this.variable and the same goes for methods in case of inner, outer classes

Outerclass.this.method(), this.method()

Private variables in the outer class can be accessed by the inner class and also instance of inner class in a method of outer class can access the private variable of inner class directly.

Access Modifiers :-

Only classes, interfaces, enums can be at the top level and everything must be included within one of these

Package-private : -means default(i.e.no access modifier is given)

Classes, interfaces,enums can be public or default

Private or protected can exist only in the inner classes/interfaces/enums

If a class is public and its constructor is default, the object can be instantiated only in the class.

If a constructor of a class is private, u cannot create an instance of it.

Member level

Public, package-private, protected, private

Static :-

Static keyword, stores the value of the member variables. For variables, static means all the objects will have only one copy of the static variable. The value modified will be modified for everyone.

For a static method, we can call the static method with the classname.method and don’t need to instantiate the object.

Why static methods can’t have non static functions ?

Because non static methods cant be called unless an object is instantiated and static methods don’t require an instantiation of object. Hence all the static methods should have a static method if that method is required.

Final :-

Final is used for constants in a class. Once the object is instantiated with a final value, its value can’t be changed. I.e. the value can be assigned in the constructor.

Why is static final used :- since the value is constant, and we need only one instance of the variable for all the objects, static final is used.

Making the method final prevents it from being overridden

Static blocks are called before the constructor

i.e static {

do something

}

Collections :-

Java.util.collections

ArrayList<>

List<>

List.forEach(lambda expression)

Arrays.asList(<Objects>….);

Instead of using list we can use Collection (i.e. being more generic)

set, list(which has ArrayList, LinkedList, Vector, Stack), queue, deque -> implement collection;

Comparable<T> Interface (to compare 2 objects) :

Has the method compareTo

Collections.binarySearch(listofobject, searchobject, null);

Collections.reverse(list);

Collections.sort(list);

Collections can be used only on lists

Copying is easy bt shallow copy is done. Ie

List<T> copylist = new ArrayList<>(originalList);

Separate lists buts contains the reference to the same objects. (sorting one would give one in sorted order and the other as it was before)

Collections.shuffle(list)

Collections.min(list)

Collections.max(list)

Collections.swap(list, swap\_position1, swap\_position2);

Collections.copy(list1, list2)

To copy elements from list1 to list2, first new of all the list1 elements needs to be done. Only then can it be copied to list2 to avoid shallow copy.

Java.util.arrays. :-

These are the methods that can be used for arrays and not for collection types like arraylist and so onn.

**public static int binarySearch(Object[] a, Object key)**

**public static boolean equals(long[] a, long[] a2)**

**public static void fill(int[] a, int val)**

**public static void sort(Object[] a)**

for printing A01 A02 and so onn

“A” + String.format(“%02d”,Number);

Comparable and Comparator :-

Question :- multiple sorting requirements i.e sort on firstname then on second name and so on.

Comparator I feel is better then comparable

Map :-

Java.util.Map

Map <T1, T2>

Map.put(<key>, <value>);

Map.containsKey(<key>) => check if the key is already present

Map.keySet(); key set

Map.remove(key); boolean

Map.remove(key,value); Boolean

Map.replace(<key>, <value>) <previous value if exists> else null

Map.replace(<key>, <old\_value>, <new\_value>) // replace if key value is old value and then make it to new value

(Map.Entry<T1, T2> item : list.entrySet() ) => Only used for iteration through a map.

Item.getKey()

Item.getValue()

Collections.unmodifiableMaps(<map\_var>) // only for read-only cannot write

Map.get(<key>)

1. HashMap<>

If a key is put more than once, it will overwrite the previous value with the new value(put method)

If a key value pair is added for the first time, it returns null. Or else, it gives the previous value

To know if the value is added for the first time.

To print all the keys, values in the map,

For (<T> keys : var.keySet())

U get the list of keys

Hashmap doesnot maintain the order in which they are inserted

Immutable classes :-

Classes whose fields cannot be modified to increase security and reduce bugs i.e. increase encapsulation

Declaring the class as final doesnot allow the class to be subclassed

Sets and HashSet

Set

1. Set has no defined ordering
2. Set cannot contain duplicates
3. Has methods add, remove, clear, isempty,
4. Methods
   1. Int s1.size()
   2. Boolean s1.isEmpty()
   3. **boolean s1.**contains(Object o)
   4. **boolean s1.**add(E e)
   5. **boolean s1.**remove(Object o)
   6. **void** clear()
   7. s1.addAll(s2) // can be used to do a union of objects
   8. s1.retainAll(set2) // intersection
   9. s1.containsAll(s2) // is s1 subset of s2
   10. s1.removeAll(s2) // se difference s1 – s2,

If an overriding of equals method is done, u should override the hashcode method as well. Since all the collections relating to the hash code are implemented on this basis.

Hashcode is directly linked to the performance of a class.

Instanceof to check which class it belongs to

Marking the method final doesnot allow the method to get overridden

Better mark the equals and hashcode as final to prevent overriding of it in the subclasses

Set intersection, union, superset/ subset

Arrays.asList(array[]) gives a list of that array. // converts []array to ArrayList

Sorted Collections :-

LinkedHashMap same as map but gives the result in the order that they are inserted.

TreeMap :- It will give the result in sorted order of the keys by using the compareTo method to compare objects

Input Output java ( java.util.io) :-

Integer:-

Integer.parseInt(String, <radix>) radix means the base

Integer.MAX\_VALUE, Integer.MIN\_VALUE; Integer.toHexString, Integer.toOctaString, …

Scanner scanner = new Scanner(System.in)

After doing a scan of scanner.nextInt(), flush the output since it might contain space,\n which might not give expected results by doing scanner.nextLine();

Exceptions :- (checked and unchecked exceptions)

Checked exceptions should be handled or else compile time error

Unchecked exceptions might take place in runtime like division by 0

inputMismatchException

Arithmetic Exception

NullPointerException

InterruptedException

IOException… etc InputMismatchException

Stacktrace() and CallTrace()

Surround the code that mmight lead to exception in try and catch the exception() so that the program doesnot crash.

Exception:- event that occurs in the middle of a program that disrupts normal flow

All the exceptions are subclasses of class Exception

If an exception occurs, by default java prints the stacktrace()

Catch the exception, this means that some method in the code will throw this exception. U are catching it and then printing the appropriate message.

Throw new <exceptionclass>(“what u want to print”); // throw is used to throw the exception to the function that has called it.

Catching multiple exceptions :-

Try {} catch {e1} catch{e2} and so on… (or) :-

Catch(ArithmeticException | NoInputException e1) {}

Method throws Exception() // that this method throws an exception which has to be handeled

Try with resources statement :- try(o1;o2…) will automatically close the resources and u don’t need to write finally to close the resources

Introduction to IO :-

FileWriter file = new Filewriter(“<filename>”)

FileReader file = new FileReader(“<filename>”)

For reading filestream also we can use scanner. i.e either from system.in or from the file☺

Scanner scanner = new Scanner(file);

Scanner.useDelimiter(“,”) to set the delimiter and based on the input, at right position, u need to mention a scanner.skip(scanner.delimiter()) to skip the delimiter (using proper delimiters // if u use a delimiter, the previous one is overwritten, so u should write it properly. i.e. “,|\n|\r\n|\\s+”

Scanner.hasNextLine() to check if next line is there or not

Scanner.close() to close the scanner, close the file as well.

BufferedReader/BufferedWriter :- (meaning that they just buffer the input/output for fast access.)

Use is that it reads chunk of data on the go. Rather than disturbing the disk everytime.hence for fast access, bufferedReader is prefered

For getting input, we can do it using the scanner.useDelimiter(<regex>) or read line and then string.split(<regex>)

BufferedWriter :- to write data to disk in one go rather than accessing the disk everytime which would slow down the speed.

Buffer.write(String)

Byte Streams :-

DataOutputStream is a wrapper so that byte data can be delt easily. It has methods writeInt(), writeUTF(<string>) to write string.

DataOutputStream file = new DataOutputStream(new BufferedWriter( new FileOutputStream(<filename>));

Similarly DataInputStream() / BufferedReader, FileInputStream. Methods readInt(), readUTF()

EndOfFileException :- set the Boolean eof = false

While(!eof) {

Try{

Do something

} catch (EndOfFileException e)

Eof =true;

}

Serializable Interface implemented by object and the objects it holds as well. No methods are there in it though:-

Recommended to declare private long serialVersionUID because it’s the key to encoding and decoding data to file. If it is changed, u will not be able to read data from filr with wrong UID

ObjectOutputStream/ ObjectInputStream

readObject/writeObject

RandomAccessFile :-

File is 0 based i.e starts from 0; RandomAccessFile rand = new RandomAccessFile(“filename”, “rwd”);

Don’t know yet

Java NIO :-

// not interested currently //

Tempfile creation = new File.createTempFile(“name”, “extension”);

Concurrency and Threads :-

Thread :-

Every java process has atleast one thread, the main thread

New Thread() to create the thread

Thread.start() starts the thread

Overriding the run method ( it is the method that starts to run upon the thread.start() execution)

Also thread can be done using the Runnable() interface i.e. :-

Thread t = new Thread(new Runnable() {

//override the run method

});

t.start();

2 ways are there :- either override the run method in thread class and do ur job. Or implement the runnable interface and get ur job. Both of them are fine just that developers prefer to use runnable interface as it is more flexible

U need to start the thread by calling the thread’s start method and not the run method

If run method is called, it doesnot create a new thread but calls the run method in the same thread where new thread was supposed to start

Thread.setName(“string”) sets the name of the thread

currentThread() in the Thread() class gets the current thread’s parameter

Since everything that should be done is in the run method, over there, we should write Thread.sleep(<time>) i.e which ever thread u want to make it to sleep, put thread.sleep there even in the main thread. Requires try catch block in it and also thread\_instance.join() requires the same.

Instanceofthread.interrupt()

Thread\_instance.join() :- waits for the thread\_instance that has been joined to the thread where it is written to complete its work and only after it exits, the main thread begin’s it’s execution.

Join(<time>) i.e. :- wait for this much time and then start running ur code.

Object methods have their local stack. So If 2 threads have same object, and call the same method, their method local variables differ. But the class variables are the same.

Reference to the local objects are on the thread stack but the objects are present on the heap. But since the threads create their own object reference, they will not point to the same value even though the object is having the same value. Thread stack will contain only primitive values, object references, function references.

Local variables are stored in the thread stack.

Instance variables are stored in the heap.

Object instances are stored in the heap.

Synchronization :-

We can synchronize methods and statements

Meaning if a class has 3 synchronized methods, then only one method can run at a time and only on one thread. Since only one thread can execute a synchronized method at a time, threads can’t interleave when running a synchronized method

Every java object has an intrinsic lock. This lock is used to synchronize the work that is done by 2 threads on that object

wait() – this should always be used in a while block with try catch so that it checks in the while block If the condition it was supposed to wait for has been met and only then it will it will start executing the rest of the code.

Notify()/notifyAll() – This is used to notify the threads that are waiting to aquire the same lock and then do what they are supposed to do.

Drrawbacks of synchronized blocks :-

1. threads that are blocked waiting to execute synchronized code are blocked and they are blocked until they get the lock for synchronized code
2. synchronized block must be withing the same method. i.e. u cannot start a synchronized block in one method and end it in other
3. we can’t test to see if an objects intrinsic lock is available also we can’t time out after some time if the lock is not released.
4. If multiple threads are waiting for the lock, it is not fcfs. Any thread might take the lock at any time.

Re-entrant lock and unlock :-

Like mutex

Java.util.concurrent.locks.ReentrantLock;

ReentrantLock buferLock = new ReentrantLock();

bufferLock.lock(); This makes sure that only one thread is having the lock and only after that thread unlocks, that the other thread can go and lock that critical section.

bufferLock.unlock();

why should we enclose lock/unlock code in try finally block

so that our unlock code is called no matter what the exception may be.

Ie. Lock();

Lock Try { critical section } finally { unlock()} // thus no matter what happens in the try block, unlock is surely called.

If(bufferlock.trylock()) // check if the lock is available and if yes, acquire it.

ReentrantLock class accepts a Boolean parameter, if it is set to true, it will do a fair priority to the threads by giving the lock to the thread that has been waiting for it the longest that are waiting for the lock

ReentrantLock getQueuedLength();

ExecutorService interface (using this allows the jvm to do thread optimization)

ExecutorService executorService = Executors.newFixedThreadPool(3) // creates a service for 3 threads.

executorService.execute(<new Runnable1>);

executorService.execute(<new Runnable2>);

executorService.execute(<new Runnable3>);

the code wont exit unless we call the shoudown method. This will see that all the executing methods have done their work and only then will shutdown the main method.

executorService.shutdown();

shutdownNow method will shutdown even if there is some service still running.

We can get some value from the thread as well. This is done using the future object.

Future<obj> future = executorService.submit(new Callable<obj> {

Public obj call() throws Exception{

// do something //

Return obj;

}

});

Sout(future.get()) to get the object // surround it with try catch(ExecutionException | InterruptedException e);

Since the newFixedPool is only three, only 3 threads can run at a time. Even though there is another thread waiting to run. It will run only after one of the thread has finished it’s job.

ArrayBlockingQueue :-

This is a threadsafe implementation of a Queue but we need to tell it before hand how many elements we need.

Eg:- ArrayBlocingQueue<String> q = new ArrayBlockingQueue<String>(<num\_elements>);

q.put to put the elements.

q.isEmpty() to check if any elements are there

q.peek() to see the element in the queue

q.take() will take the first element in the queue. Still we might have to add synchronization code even though arrayblockingqueue is threadsafe

Thread.setPriority() allows to set the priority of the thread.

Fair lock and live lock :-

Fair Lock :-

Only fairness in acquiring the lock is guaranteed not fairness in thread scheduling.

Also in fairness, there has to be extra processing and hence performance will be impacted.

ReentrantLock lock = new ReentrantLock(true) // true implies fair lock ie FCFS

If the application uses only 2-3 threads and starvation may not be issue then u should use synchronized blocks

Or else if starvation is an issue, then it is better that fair locks are used as it will not allow starvation and will be fair

Live lock :- ??

Atomic operations :-

Reading and writing reference variables is atomic. Ie. Myobject1 = myobject2 is atomic.

Reading and writing primitive variables is atomic. Ie. Myint = 1 is atomic but only long and double are not atomic.

Volatile variable importance :-

It comes only in threads to prevent memory inconsistency error. Ie there might be a case where thread1 is running on cpu1 and thread2 on cpu2. Thread1 updates the value of the variable on cpu1 cache which is atomic but the thread2 on cpu2 reads from main memory and hence gets the old value 0 thus resulting in a mismatch in value. Hence volatile is used which makes sure that if an update takes place of a shared variable, its value is immediately updated on the memory as well.

Using the volatile to long and double makes them atomic but we still have to use synchronization if there are 2 threads which update the common variable as in case of primitive types

// some more to be added //

Java.util.concurrent.atomic package

AtomicInteger counter = new AtomicInteger(0);

Methods : -

.incrementAndGet()

.decrementAndGet()

.get

There are atomic classes for following types :-

Boolean, integer, integer array, long, long array, object referenct and double

Majorly used in counters and loop through an array

compareAndSet() this method compares the value that is being set with the expected value and only then sets it and returns true else returns false.

Some other info :-

getClass().getSimpleName() can be run only in non static methods

Lambda expressions :-

Can only be used with interfaces that has only one method that has to be implemented // why because the lambda expression has to map everything it has to a single function.

Has three parts -- argument list, “->” token, {body}//code we want to run //

Indeed simplyfing the code.. no need to write return as well.

No need to put parathesis if there is only one parameter.

Anonymous classes. why a local variable need to be final when we use it within a anonymous class.

Because java doesnot allow it. i.e a method local variable is directly replaced with some value in the instance of the anonymous class and hence to prevent inconsistency, it is declared as final

Any variables used before the lambda block must not be changed or must be final.

Java.util.function package :-

List.forEach(//lambda expression//) to iterate through the list using lambda expressions

Predicate interface has a test method that returns a Boolean value. // this combined with lambda expressions is very useful in reducing the code. (just that lambda expression reduces code)

Pass the argument to test that u want to test. It will return a Boolean value

And/or/ to concatenate two predicate instances,

Function interface :-

Just declare a function interface with a lambda expression

Apply method to apply ur lambda expression to the code. //same as test.

//// some more info to be added /////

Streams :-

Stream is a setoff object references. Stream method creates a stream for a collection. Each object reference in the stream corresponds to an object in the collection and ordering of the object matches ordering of the collection. The object on which stream is used wont change when it is used.

Any stream must meet 2 requirements :-

1. It must be non interfering ie.they don’t change the stream source in anyway and second they must be stateless. i.e each operation should be an independent step and not depend on previous computation.
2. Eg :- List <String> arr // having some string objects
3. arr.stream.map(String::toUpperCase).filter(s->s.startsWith(‘G’)).sorted().forEach(System.out :: println)map here will return a stream of all the toUpperCase objects
4. filter will filter the objects of the stream with G
5. sorted will sort them
6. for each will iterate through the complete Stream

Stream.concat 🡪 static method for concatenating 2 streams

.distinct to get the distinct objects of the stream

.count -> to count the number of items in the stream

.foreach -> iterates through all the elements of the stream

.peek() -> will see the objects of the stream.

Flatmap() 🡪 used to combine a stream of objects. Eg:-

Employee is a class and Department(hr, accounting) is a class

Departments.add(hr)

Departments.add(accounting)

Hr.addEmployee(john)

Hr.addEmployee(jane)

Hr.addEmployee(snow)

Accounting.addEmployee(jack)

This would print all the employees :-

Departments.stream()

.flatmap(department -> department.getEmployees().stream())

.forEach(System.out :: println); would list all the employees from all the departments. Simple way to solve complex problem.

To store the contents of the stream to a list, u could collect method

Ie. List.stream()

.filter(s->s.startsWith(“G”))

.sorted()

.collect(Collectors.toList())

List<Obj> d = c.stream().sorted((o1, o2) -> o1.name.compareTo(o2.name)).collect(Collectors.toList());

Regular Expressions :-

String.replaceAll(“currentString”, “newString”); // returns a new string with the new replaced result. The old string is intact.

. -- matches all the characters of a string.

^ -- start of the string.

$ -- end of the string.

Eg :- [aei] -- to select a set of characters

[^aei] -- to select all the characters except aei

[a-eA-F0-5] --- to select from a to e, A-E, 0-5

(?i)[1-e3-8] – this will ignore case sensitivity. ( ?i for asci and for Unicode, ?u);

\d -- to get all the digits. ( [\\d](\\\\d) so that u escape the \ and then ull get the \d)

\D -- to replace all the non digits.

\s -- to remove all the white spaces i.e space, \t , \n ..

\t -- to remove \t

\S -- to remove all the non-space characters.

\w -- to match all the alphabets, small and caps, numbers, \_

\W -- opposite of the above.

\b -- to surround each word with the replacement string.

Quantifiers – {} – it comes after the letter it applies to.

Eg – abcDeee can be matched by abcDe{3};

abcDe{2,5} will match abcDe(where e should be atleast 2 and not more that 5)

abcDe+ will match abcDe(one or more e’s)

abcDe\* will match abcD(zero or more e’s

String.trim() – to remove white spaces.

String.matches(regex) returns true if regex is present else false ( for the matches method to return true, the string as a whole has to match to the regex only then will it give true else it will give false.

Pattern pattern = Pattern.compile(“<regex>”, Pattern.CASE\_INSENSITIVE | Pattern.UNICODE\_CASE)

To match in terms of case insensitive and Unicode case.

Matcher matcher = pattern.matcher(<text to be matched to pattern>);

matcher.matches() will match according to the String.matches() method.

Returns true if the complete string matches the regex else false.

Matcher .find() -- gives the locations where it has found the pattern.

matcher.start() – gives the start location in the text where it has found the pattern,

matcher.end -- give the end point in the text where the pattern ends.

matcher.reset() -- you can only use a matcher once. Thus u have to reset to start matching it from the start.

Groupmatcher is declared by giving the regex string to be matched in parenthesis.

Groupmatcher.group(<val>); will give the corresponding group it has matched. Multiple groups can be given in the regex string using the | separator.

Greedy vs Lazy matcher :-

Greedy is \* - it doesnot stop once it has found the first occurance and tries to see if more characters can be included.

Lazy is \*? - it stops once it has found the first occurance of the match.

U can get an individual field by having more than one group in regex. (<h2>)(.+?)(</h2>) here give matcher.group(2) will give the text.

t[^v] – will match anything in the text where t is not followed by v but will not match a t alone. To do that u will have to write.

t(?!v) --- this will match t not followed by v. Can be blank as well.

Databases ( JDBC ) :-

Try {

Connection conn = DriverManager.getConnection(“jdbc:sqlite:<path>”) --- to establish connection to the data base

Statement statement = conn.createStatement();

statement.execute(“CREATE TABLE contacts (name TEXT, phone Number, email TEXT)”)

Catch (SQLExeception)

Better approach would be to write the conn and statement in the try() as it would take care of the closing of the stream as well.

//rather than doing this statement.close(), conn.close();

SQL example to create the table only once

CREATE TABLE IF NOT EXISTS contacts (name TEXT, phone Number, email TEXT)

Conn.setAutoCommit(false) 🡪 this will not commit the changes when the execution is finished. It has to be committed explicitely.

Statement.execute(“some command”);

RessultSet results = statement.getResultSet();

While(results.next()) {

Sout (“result.getString(“name”) + “ “ + results.getInt(“phone”) + “ “ + results.getString(“email”);

}

Results is initially pointing the first record -1 . when we call the results.next(), then it points the first record. And so onn.

Better way to get results is

ResultSet results = statement.executeQuery(“<Query>”);

Using the execute query u can execute the query and then store the result set in a list or any iterable and perform the computation.

ResultSetMetaData meta = results.getMetaData();

Int numColumns = meta.getColumnCount();

For(int i=1; i<numColumns;i++) {

Sout(i + meta.getColumnName());

Networking in java

Client server application :-

ServerSocket object for server socket programming.

Socket object for client side programming.

Socket.getInputStream() // gets the input to the socket

Socket.getOutputStream() // gets the output to the socket

Upto here was for tcp/ip stack

For udp it is

Server Code

DatagramSocket socket

DatagramPacket packet = new DatagramPacket(buffer, buffer.length)

Socket.recieve(packet);

ClientCode :-

InetAddress address = new InetAddress.getLocalHost();

DatagramSocket socket

Datagram Packet packet = new DatagramPacket(buffer, buffer.length, address, portNumber)

Socket.send(packet);

List.parallelstream ??

Stream filter collectors.tolist

Finalize

Java debugging. :-

Step over – go to the next line of execution

Step into – go to the function api at the line of execution.

Force step into – used to force step into on the jdk api’s

Step out – go reverse to the execution time.

Run to cursor – run the program till the cursor line.

Watch point – right click and add the variable as watch point to continually evaluate its value.

Field watch point – alt + click on that line