**Theory Question:**

**1.** **What is JVM and explain me the Java memory allocation**

*JVM is the main reason why the program could run on any OS. It is a compiler that knows how to compile Java code (convert it to Byte code) JVM is known as JRE.*

*Java source file (.java) -- compiler produces (javac file.java) → byte code (.class file) -- run program (java file.class) → allocates memory to class and its variables and process the program to generate the output.*

*Stack: contains local variables & reference variables (variables that hold the address of an object in the heap) uses LIFO order and faster than Heap and smaller in size and more expensive*

*Heap: contains all created objects in runtime, objects only plus their object attributes (instance variables) used arbitrary order (dynamic memory allocation)*

*Code: the segment where that actual compiled java bytecodes resides when loaded.*

*JVM's are not platform independent. JVM's are platform specific run time implementation provided by the vendor.*

**2.** **What is Polymorphism and encapsulation?**

*Polymorphism is ability of an object to take many forms. In java it can be achieved by method overriding and method overloading. Polymorphism is primarily achieved by subclassing or by implementing an interface. The derived classes can have its own unique implementation for certain feature and yet share some of the functionality through inheritance. In short we can also say Polymorphism means one object can take up multiple forms. E.g. “+” can be used for addition as well as concatenation of strings.*

*Encapsulation means data is protected. It is the process that prevents class variables from being read or modified by other classes. The only way to use these variables is by calling methods of the class if they are available.*

*Encapsulation is a technique that encourages abstraction by purposefully hiding information. For example, the mechanical details of a car engine are encapsulated behind a steering wheel and floor pedals. Anyone familiar with this interface could drive a car without knowing what type of engine was under the hood. Java encourages encapsulation through the use of interfaces and by providing access modifiers that limit the visibility of classes, fields, and methods.*

**3.** **What is method overloading and Method overriding?**

*Overriding is a method with the same name and arguments as in a parent, whereas overloading is the same method name but different arguments.*

*Method Overriding → When a child class implements the same method as present in the parent class.*

*Method overriding is redefining the base class method to behave in a different manner than its implementation in the base class. Method overriding is an example of dynamic or runtime polymorphism. In dynamic polymorphism, runtime takes the decision to call an implementation, as compiler does not know what to bind at compile time.*

*Method overriding:*

* *Method arguments and its order must be same in the overriding method.*
* *Overriding method can have same return type or subtype of base class method’s return type.*
* *Access modifier of overridden method cannot be more restrictive than its definition in base class.*
* *Constructor, static and final method cannot be overridden.*
* *Overridden method cannot throw checked exception if its definition in base class does not, though overridden method can still throw unchecked exception.*

*Method overloading → defining 2 or more methods with same name and signature but different type / no of arguments.*

*Method overloading is defining more than one method with the same name, but with different parameters. Method overloading is an example of static or compile time polymorphism. In static polymorphism, it’s while writing the code, decision is made to call a specific implementation.*

*Method Overloading:*

* *Method can be overloaded by defining method with the same name as an existing one, having different number of argument list. Different datatype of arguments. Different order of arguments. Return type of the overloaded method can be different. Method with the same name and exactly same parameters cannot be defined, when they differ only by return type. Overloading method is not required to throw same exception as the method its overloading.*

**4.** **Why string is Immutable?**

*The string object is immutable, which means once constructed, the object which String reference refers to, can never change. Though you can assign same reference to another string object.*

*e.g. String name = “Parul”;*

*name = name + “Mekvan”; (this code creates 3 different string objects, “Parul”, “Mekvan” and “Parul Mekvan”.*

*Though you can not change the value of the string object but you can change the reference variable that is reffering to the object. In the example, the string reference name starts referring the string object “Parul Mekvan”. Note that any operation performed on string results into creation of new string. String class is marked final, it’s not possible to override immutable behavior of the string class. It provides benefits of the string pool and security reasons.*

**5.** **What is the difference between String and String buffer?**

|  |  |
| --- | --- |
| *String* | *StringBuffer* |
| *String class is immutable* | *StringBuffer class is mutable* |
| *String is slow & consumes more memory when you concat too many strings because every time it creates new instance* | *StringBuffer is fast & consumes less memory when you concat strings.* |
| *String class overrides the equals() method of Object class. So you can compare the contents of two string by equals() methods.* | *StringBuffer class doesn’t override the equals() method of Object class.* |
| *Strings are strored in string pool.* | *StringBuffer objects are stored in heap.* |

**6.** **What is the difference between array and arraylist?**

|  |  |
| --- | --- |
| ***Array*** | ***ArrayList*** |
| *Fixed size* | *Dynamic size* |
| *Inbuilt variable length is provided for total no. of elements* | *Inbuilt method size is provided* |
| *Can contain object and primitive data types* | *Can contain only objects* |
| *Default size is mandatory to declare array variable* | *Default size is not mandatory,* |
| *Length of array cannot be changed once declared* | *Size of arraylist can be changed after declaration.* |
| *Cannot use generics with array* | *Can use generics with array list* |

**7.** **What is the difference between hash map and Hash table?**

*HashTable and HashMap are data structures used to keep key-value pair. These maintain an array of buckets and each element is added to a bucket based on the hashcode of the key object. The major difference between these is that the HashMap is non-synchronized. This makes HashMap better for single-threaded applications, as unsynchronized Objects perform better than synchronized ones due to lack of locking overheads. In a multi-threaded application HashTable should be used. A HashMap can also be converted to synchronised collection using following method: Collections.synchronizedMap( idToNameMao);*

|  |  |
| --- | --- |
| ***Hashmap*** | ***Hashtable*** |
| *Not synchronized* | *synchronized* |
| *Allow 1 null key and multiple null values* | *No null key / values allowed* |
| *Traversed by iterator* | *Traversed by enumerator / iterator* |
| *It is fail fast* | *It is not fail fast* |
| *It is fast* | *Its slow* |

**8.** **What is a vector in Java?**

*Vector is a parameterized class, so that you can use types such as Vector<String> and Vector<Integer>, but you will often see it used without the type parameter, which is essentially equivalent to using Vector<Object>.*

*Using Vector is similar to using an ArrayList, except that different names are used for some commonly used instance methods, and some instance methods in one class don’t correspond to any instance method in the other class. Suppose that vec is variable of type Vector<T>. Then we have instance methods:*

*-> vec.size() – a function that returns the number of elements currently in the vector.*

*-> vec.elementAt(N) – returns the N-th element of the vector, for an integer N.N must be in the range 0 to vec.size()-1. This is the same as get(N) for an ArrayList.*

*-> vec.setElementAt(obj, N) – set the N-th element in the vector to be obj.N must be in the range 0 to vec.size()-1. This is the same as set(N,obj) for an ArrayList.*

*-> vec.addElement(obj) – adds the Object, obj, to the end of the vector. This is the same as the add() method of an ArrayList.*

*-> vec.removeElement(obj) – removes obj from the vector, if it occurs. Only the first occurrence is removed. This is the same as remove(obj) for an ArrayList.*

*-> vec.removeElementAt( N) -- removes the N-th element, for an integer N. N must be in the range 0 to vec.size()-1. This is the same as remove( N) for an ArrayList.*

*-> vec.setSize( N) -- sets the size of the vector to N. If there were more than N elements in vec, the extra elements are removed. If there were fewer than N elements, extra spaces are filled with null. The ArrayList class, unfortunately, does not have a corresponding setSize() method.*

**9.** **What is set in java?**

*Set – Is a collection types.*

*Basic Set:*

*- set is collection of unique elements.*

*- elements in the set are stored un-ordered.*

*- only one null element can be added to set.*

*- duplicate elements are ignored.*

*- when ordering is not needed, set is fastest and has smaller memory footprint.*

*Linked Hashset:*

*- LinkedHashSet keeps the set elements in the same order in which they were inserted.*

*- Insertion order is not affected in LinkedHashSet if an element is re-inserted.*

*- Iterator in LinkedHashSet returns elements in the same order in which these were added to the collection.*

*SortedSet:*

- *SortedSet imposes ordering of elements to be either sorted in a natural order by implementing Comparable interface or custom sorted using Comparator object.*

-

- *TreeSet is an implementation class for the SortedSet interface.*

-  *// Create a sorted set of city names*

*SortedSet < String > cityNames = new TreeSet < >();*

- *Use Comparator object to perform custom sorting.*

*SortedSet < City > sortedCitiesByName = new TreeSet < >*

*(Comparator.comparing (City:: getName));*

- *If an element implements Comparable interface, then compareTo() method is used to sort in the natural order.*

*Navigable Set:*

- *NavigableSet inherits from the SortedSet and defines additional methods.*

- *NavigableSet can be traversed in both, ascending and descending order.*

- *TreeSet is one of the implementation classes for NavigableSet interface.*

**10.** **What is an abstract class?**

*Abstract class cannot be instantiated but can be extended. You should extend abstract class when you want to enforce a common design and implementation among derived classes.*

*A class having abstract keyword in its declaration is known as abstract class.*

* *It can’t be instantiated.*
* *May or may not have any abstract methods*
* *A class containing any abstract methods must be declared abstract.*
* *Any class inheriting abstract class must implement all its methods else it will give compilation error.*

*Use: it is useful in hiding the implementation details from the user and only providing the functionality. For ex: the below maths class provides different methods of calculations to user hiding the details as to how they are stored and calculated.*

*Abstract Class Maths{*

*Public int add(int a, int b);*

*Public int subtract(int a, int b);*

*Public int multiply(int a, int b);*

*}*

**11. What is an interface?**

*Interface is a reference type in java which can contain abstract methods, default methods, static methods, constant fields and method signature. It is declared using keyword Interface. Interface names should be adjective. They should end with “able” or “ible” whenever interface provides a capability: otherwise they should be nouns. Interface names follow the same capitalization convention as class names:*

*public interface Serializable {...}*

*Public interface SystemPanel {...}*

**12. Why Java is Platform independent?**

*Platform is anything on which a program is run(in lay man term). A platform is either the machine itself, or a primitive kernel or a fully developed OS. Languages can be platform dependent or independent.*

*Dependent means its executable code and source code varies from platform to platform. That is to say that the programmer has to make some changes in the code if it has to run on other platforms. Languages like C are platform dependent.*

*Platform independent means that the code remains the same irrespective of the platform involved. Java has something called a virtual machine called JVM or Java Virtual Machine. What happens in case of Java is that the JVM once installed on any platform like windows or OS X can run the java code without any alteration. The JVM acts like a virtual platform on which the code is executed. As the platform being JVM remains constant throughout all platforms Java programs can run on any platform irrespective of the real platform.*

*If you want to know the detailed process then it is here: Java source code written in the high level java is first converted to something called the javabyte code. This byte code is not readable by the original platform, but only the JVM can read it and then it executes it. This is the primary reason why Java is much in use even today.*

*Platform independence means that we can write and compile the java code in one platform (eg Windows) and can execute the class in any other supported platform eg (Linux, Mac etc).*

**13. What are access modifiers? Give me an example?**

*Access modifiers help you set the level of access you want for your Class, variables as well as Methods. There are three access modifiers. Not including default access modifier . Default is an access control which will be set when one does not specify any access modifier. Access modifiers (some or all) can be applied on class, variable, methods. Classes in Java uses only public and default access modifiers.*

*Public: When set to public, the given Class will be accessible to all the classes available in the Java world.*

*Private*

*Protected*

*Default (package): When set to default, the given class will be accessible to the classes which are defined in the same package.*

*public class Clock {*

***private*** *long time = 0;*

*}*

*There are a number of modifiers in Java, including access modifiers and non-access modifiers. To use a modifier you should include keywords within the definition of a variable, method or class.*

*public class className {*

*// ...*

*}*

*private boolean myFlag;*

*static final double weeks = 9.5;*

*protected static final int BOXWIDTH = 42;*

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

*// body of method*

*}*

*Access control modifiers:*

*Java provides users with a set of access modifiers for setting the levels of access for constructors, variables, classes and methods. There are four levels in Java, namely:*

* *Visible to the whole package. This is the default and requires no modifiers.*
* *Visible only to the class only, i.e. private.*
* *Visible to the whole world, i.e. public.*
* *Visible to the entire package and all of the subclasses, i.e. protected*

*Non-access modifiers:*

*There are also a number of non-access modifiers in Java:*

* *The static modifier is used for creating class methods and variables.*
* *For finalizing the implementation of variables, classes and methods we use the final modifier.*
* *You can create abstract classes and abstract methods by using the abstract modifier.*
* *For dealing with threads, we use the volatile and synchronized modifiers.*

**14. What are java exceptions? Give me an example**

*An exception can be defined as a problem that occurs during a program’s execution and disrupts its normal flow. There are many reasons that an exception can occur. Some of them are:*

* *Invalid data has been given.*
* *The Java virtual machine has run out of memory.*
* *The network connection is terminated during communication.*
* *The required files cannot be found.*

*Some exceptions are due to user error, others to programmer error, and the rest to the failure of physical resources.*

*There are three categories of exceptions in Java. To understand the concept of exception handling you should know the types of exceptions:*

*Checked exceptions: An exception can be defined as a checked exception if it can’t be foreseen by the programmer or if it is a typical user error. Checked exceptions cannot be ignored during compilation time.*

*Runtime exceptions: Runtime exceptions can be defined as exceptions that probably could have been solved by the programmer. Runtime exceptions, unlike checked exceptions, will be ignored during compilation time.*

*Errors: Errors can be defined as problems that a user or a programmer cannot control. Technically they’re not really exceptions. As they cannot be solved, errors are usually ignored. For example, an error will be raised in case of a stack overflow. This error will be ignored during the compilation of the program.*

*E.g. A try-catch block:*

*Try {*

*// do something*

*} catch (Exception e) {*

*// handle exception*

*}*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*class Test {*

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

*FileInputStream file\_in = null;*

*try {*

*file\_in = new FileInputStream("C:/some\_file.txt");*

*} catch(FileNotFoundException e) {*

*e.printStackTrace();*

*}*

*} // end main()*

*}*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Throwing exceptions:*

*class NotNumeric {*

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

*String str = "a";*

*methodOne(str);*

*} //end main()*

*static void methodOne(String s) throws NumberFormatException {*

*Integer.parseInt(s);*

*}*

*}*

**15. What is the difference between throws and throwable?**

***Throws -*** *It is used to indicate that a method may throw the respective type of exception. It is added in method signature. A caller method must handle the exception thrown by try-catch block or can throw to the higher caller in hierarchy. Failure to catch the exception can result in compilation error.*

***Throwable -*** *It is super class of Error and Exception.. It catches all the subclasses of Throwable i.e. Error and Exception. It is not advisable to catch errors in your application as sometimes it can cause the root cause to be lost.*

**16. What is the difference between Error and exception?**

*An error is an irrecoverable condition occurring at runtime like out of memory error. These kind of jvm errors cannot be handled at runtime. Exceptions are because of condition failures, which can be handled easily at runtime.*

**17. What is the difference between Error, throwable and exception?**

***Error -*** *is caused by the environment in which application is running. An application can not recover from a error. The error are of unchecked type and may or may not be handled in application. It handles only the subclasses of Error.*

***Throwable -*** *It is the superclass of all the error and exception. it catches all the subclasses of Throwable i.e. Error and Exception. It is not advisable to catch errors in your application as sometimes it can cause the root cause to be lost.*

***Exception -*** *It is a problem that occur during the execution of the program which disrupts the normal flow of the program and end the program abnormally. It can be checked or unchecked type. It handles only the subclasses of Exception.*

**18. What are collection APIs, give me an example.**

*Collection API is a set of interfaces and classes provided by Java for manipulating and handling group of objects. Some of the collection allow duplicate values, some do not not. Some are ordered and some are unordered. Collection interface is at the top of collection hierarchy. Some elements in collection hierarchy are - list, set, map, etc*

*Ex:* ***List Interface*** *- This extends collection interface and it is a ordered collection of elements. Implementations of List Interface -* ***LinkedList , ArrayList***

*Ex: ArrayList a = new ArrayList();*

*a.add(1);*

*a.get(0); // 0 is location or index*

**19. What is the difference between final and finally?**

|  |  |
| --- | --- |
| final | finally |
| * Keyword * Variable: constant value cannot be changed * Method: can’t be overridden. * Class: cannot be inherited. | * Block * Always be executed whether an exception has been handled or not |

**20. Will java supports multiple inheritance?**

*No, only single class inheritance is allowed in Java, as multiple inheritance comes with its share of complexity.*

**21. What are the different types of interface? (Ans List, set, Queue)**

***Marker Interface*** *- An empty interface with no body is a markr interface. It is used to distinguish a class / interface for the JVM. For ex: Clonable, if a class implements CLonable interface then JVM will know that this class is clonable and will perform the operations accordingly.*

***Nested INterface -*** *A interface defined in another interface*

*Other types in Java are :*

***Set Interface*** *- A set is a interface which do not allow duplicate elements in it. It does not guarantee the order of the elements. Ex: HashSet, TreeSet*

***List INterface*** *- A List interface represents a ordered collection of objects.Ex: LinkedList, ArrayList*

***Queue Interface -*** *Interface to hold elements in a FIFO (first in first out ) order. It provides additional methods - insertion, removal and inspection other than methods available in collection interface.*

**22. What are wrapper class? Give me an example**

*Each Primitive data type has a class defined for it, which wraps the primitive datatype into object of that class. Wrapper classes provide lots of utility methods to operate on primitive data values. As the wrapper classes enable primitive types to convert into reference types, these can be used with collections too.*

*. Ex int => Integer*

*Integer x = new Integer(10); // wrapper class*

*Int y = x.intValue(); // returns the value of x as int*

**23. What is boxing and unboxing in Java? Explain with an example**

Wrapper classes provide mechanism to convert primitive data type to object and vice-versa. This is also called autoboxing and unboxing.

Autoboxing - converting the primitive to Object

Unboxing - converting from Object to primitive

|  |  |
| --- | --- |
| primitive | Wrapper |
| int | Integer |
| byte | Byte |
| char | Character |
| long | Long |
| double | Double |
| float | Float |
| boolean | Boolean |
| long | Long |
| short | Short |

**Ex: Autoboxing**

Int a = 10;

Integer b = Integer.valueOf(a); => autoboxing

**Unboxing**

Integer a = new Integer(3);

Int b = a.intValue();

**24. Explain for each loop**

*For-each loop can be used to loop map.entrySet() to get key and value both.*

*E.g.:*

*for (Map.Entry<String, Account> accountEntry :*

*map.entrySet()) {*

*print("UserId - " + accountEntry.getKey() + ", " +*

*"Account - " + accountEntry.getValue());*

*}*

* *for -each loop can be used to loop map.keySey() to get keys.*
* *For-each loop can be used to loop map.values() to get values.*
* *While running a for-each loop, collection cannot be modified.*
* *For-each loop can only be used to navigate forward.*
* *For-each loop should be preferred over for loop, as for loops are source of erros, specifically related to index calculations.*

**25. What are iterators, explain with an example**

* *Iterator has an ability to move in both backward and forward directions.*
* *You can remove entries during an iteration when using an Iterator, which is not possible when you use for-each loop.*
* *For-each also uses Iterator internally.*
* *Iterator is considered to be more thread safe because it throws exception if the collection changes while iterating.*

*E.g.:*

*Iterator<Map.Entry<String, Account>> accountIterator =*

*map.entrySet().iterator();*

*while (accountIterator.hasNext()){*

*Map.Entry<String, Account> accountEntry =*

*accountIterator.next();*

*print("UserId - " + accountEntry.getKey() + ", " +*

*"Account - " + accountEntry.getValue());*

*}*

**26. How do you access Private variables in different class?**

* *Bye getter and setter methods*

*Ex: Class person {*

*Private String name;*

*Public String getName() { return name; } // getter method*

*Public void setName(String name) { this.name = name;} //setter method*

*}*

* *By Java reflection*

*Java reflection is a technique to get the details of any java class or interface during runtime even though class is not accessible. To access private variables of any other class, we obtain the class object and then we can get the type, name, value of the private field.*

*Ex:*

*Class Person {*

*Private String name = “abc”;*

*Public String getName() { return name; } // getter method*

*Public void setName(String name) { this.name = name;} //setter method*

*}*

*Class ReflectionPerson {*

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

*Person person = new Person();*

*Field nameField = Person.class.getDeclaredType(“name”);*

*nameField.setAccessible(true); // setting the access of private field*

*System.out.println(nameField.get(person); // will give name*

*}*

*}*

**27. Prepare for one java program to write on the board.**

**28.** **What is Constructor Overloading?**

*Constructor overloading is way of having more than one constructor which does different-2 tasks.*

|  |
| --- |
| *public class TimeOfDay1c {*  *public int hour;*  *public int minute;*  *//==================================================== constructor*  *public TimeOfDay1c(int h, int m) {*  *hour = h;*  *minute = m;*  *}*  *//==================================================== constructor*  *public TimeOfDay1c(int h) {*  *hour = h;*  *minute = 0; // Set minutes to 0.*  *}*  *}* |

**29.** **Without using sync key word how do you perform synchronization?**

**30.** **What is Super keyword? when and where do you use it ?**

*Super keyword has two meanings: to invoke a superclass method and to invoke a superclass constructor.*

*When used to invoke constructors, the this and super keywords are closely related. The constructor calls can only occur as the first statement in another constructor. The constructor parameters are either passed to another constructor of the same class (this) or a constructor of the superclass (super).*

#### *super Usage:*

*1) super.<variable\_name> refers to the variable of variable of parent class.*

*2) super() invokes the constructor of immediate parent class.*

*3) super.<method\_name> refers to the method of parent class.*

**Programing Questions:**

1. **Find out the number of days in between two given dates?**

import java.util.\*;

class dateDiff1 {

public static void main(String[] args) {

Calendar c1 = Calendar.getInstance();

c1.set(2016, 8, 01);

Calendar c2 = Calendar.getInstance();

c2.set(2016, 9, 30);

Date d1 = c1.getTime();

Date d2 = c2.getTime();

long diff = d2.getTime() - d1.getTime();

int noofdays = (int) (diff / (1000 \* 24 \* 60 \* 60));

System.out.println(noofdays);

}

}

1. **How to divide a number by 2 without using / operator?**

**import java.util.Scanner;**

public class divideOrMultiplyNoBy2 {

public static void main(String[] args) {

Scanner n = new Scanner (System.in);

System.out.println("Enter an integer value for division and multiplication: ");

int num = n.nextInt();

int mul = multiplayByTwo(num);

int dv = devideByTwo(num);

System.out.println("Result of multiplication " + num + "\*2 is = " + mul);

System.out.println("Result of division " + num + "/2 is = " + dv);

}

public static int multiplayByTwo(int num) {

return (num << 1);

}

public static int devideByTwo(int num) {

return (num >> 1);

}

}

1. **How to multiply a number by 2 without using \* operator?** -- covered in 2nd program
2. **How to swap two variables, by using pass by reference method ?**

**class** myInteger {

**private** **int** x; // single data member

**public** myInteger(**int** xIn) {

x = xIn;

} // constructor

**public** **int** getValue() {

**return** x;

} // retrieve value

**public** **void** insertValue(**int** xIn) {

x = xIn;

} // insert

}

**public** **class** swapTwoVariables {

// swap: pass references to objects

**static** **void** swap(myInteger rWrap, myInteger sWrap) {

// interchange values inside objects

**int** t = rWrap.getValue();

rWrap.insertValue(sWrap.getValue());

sWrap.insertValue(t);

}

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

**int** a = 23, b = 47;

System.***out***.println("Before. a:" + a + ", b: " + b);

myInteger aWrap = **new** myInteger(a);

myInteger bWrap = **new** myInteger(b);

*swap*(aWrap, bWrap);

a = aWrap.getValue();

b = bWrap.getValue();

System.***out***.println("After. a:" + a + ", b: " + b);

}

}

1. **How to make a list immutable?**

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** listImmutable {

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

List<Integer> arrayList = Arrays.*asList*(1, 2, 3);

List<Integer> scores = **new** LinkedList<>(arrayList);

System.***out***.println("scores: " + scores);

arrayList.set(0, 7);

System.***out***.println("updated array list: " + arrayList);

scores.add(42);

System.***out***.println("update scores: " + scores);

scores = Collections.*unmodifiableList*(scores);

**try** {

scores.add(7);

} **catch** (UnsupportedOperationException e) {

System.***out***.println("While adding: " + e.getClass().getSimpleName());

}

**try** {

scores.set(0, 7);

} **catch** (UnsupportedOperationException e) {

System.***out***.println("While setting: " + e.getClass().getSimpleName());

}

}

}

1. **Write a sample code to reverse Singly Linked List by iterating through it only once.**

**public** **class** SinglyLinkedListImpl<T> {

**private** Node<T> head;

**public** **void** add(T element){

        Node<T> nd = **new** Node<T>();

        nd.setValue(element);

        System.out.println("Adding: "+element);

        Node<T> tmp = head;

**while**(**true**){

**if**(tmp == **null**){

                //since there is only one element, both head and

                //tail points to the same object.

                head = nd;

**break**;

            } **else** **if**(tmp.getNextRef() == **null**){

                tmp.setNextRef(nd);

**break**;

            } **else** {

                tmp = tmp.getNextRef();

            }

        }

    }

**public** **void** traverse(){

        Node<T> tmp = head;

**while**(**true**){

**if**(tmp == **null**){

**break**;

            }

            System.out.print(tmp.getValue()+"\t");

            tmp = tmp.getNextRef();

        }

    }

**public** **void** reverse(){

        System.out.println("\nreversing the linked list\n");

        Node<T> prev = **null**;

        Node<T> current = head;

        Node<T> next = **null**;

**while**(current != **null**){

            next = current.getNextRef();

            current.setNextRef(prev);

            prev = current;

            current = next;

        }

        head = prev;

    }

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

        SinglyLinkedListImpl<Integer> sl = **new** SinglyLinkedListImpl<Integer>();

        sl.add(3);

        sl.add(32);

        sl.add(54);

        sl.add(89);

        System.out.println();

        sl.traverse();

        System.out.println();

        sl.reverse();

        sl.traverse();

    }

}

**class** Node<T> **implements** Comparable<T> {

**private** T value;

**private** Node<T> nextRef;

**public** T getValue() {

**return** value;

    }

**public** **void** setValue(T value) {

**this**.value = value;

    }

**public** Node<T> getNextRef() {

**return** nextRef;

    }

**public** **void** setNextRef(Node<T> ref) {

**this**.nextRef = ref;

    }

    @Override

**public** **int** compareTo(T arg) {

**if**(arg == **this**.value){

**return** 0;

        } **else** {

**return** 1;

        }

    }

}

1. **Write a program to implement ArrayList and Linked list**

**import** java.util.Arrays;

**public** **class** myArrayList {

**private** Object[] myStore;

**private** **int** actSize = 0;

**public** myArrayList(){

myStore = **new** Object[10];

}

**public** Object get(**int** index) {

**if** (index < actSize) {

**return** myStore[index];

} **else** {

**throw** **new** ArrayIndexOutOfBoundsException();

}

}

**public** **void** add(Object obj) {

**if** (myStore.length - actSize <= 5) {

increaseListSize();

}

myStore[actSize++] = obj;

}

**public** Object remove(**int** index) {

**if** (index < actSize) {

Object obj = myStore[index];

myStore[index] = **null**;

**int** tmp = index;

**while** (tmp < actSize) {

myStore[tmp] = myStore[tmp + 1];

myStore[tmp + 1] = **null**;

tmp++;

}

actSize--;

**return** obj;

} **else** {

**throw** **new** ArrayIndexOutOfBoundsException();

}

}

**public** **int** size() {

**return** actSize;

}

**private** **void** increaseListSize() {

myStore = Arrays.*copyOf*(myStore, myStore.length \* 2);

System.***out***.println("\nNew length: " + myStore.length);

}

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

myArrayList mal = **new** myArrayList();

mal.add(**new** Integer(2));

mal.add(**new** Integer(5));

mal.add(**new** Integer(1));

mal.add(**new** Integer(23));

mal.add(**new** Integer(14));

**for** (**int** i = 0; i < mal.size(); i++) {

System.***out***.print(mal.get(i) + " ");

}

mal.add(**new** Integer(29));

System.***out***.println("Element at Index 5:" + mal.get(5));

System.***out***.println("List size: " + mal.size());

System.***out***.println("Removing element at index 2: " + mal.remove(2));

**for** (**int** i = 0; i < mal.size(); i++) {

System.***out***.print(mal.get(i) + " ");

}

}

}

**8.** **Write a program for Insertion Sort in java.**

**public** **class** InsertionSort {

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

String[] stringArray = { "ball", "zoo", "coin", "hat", "apple" };

// test array

System.***out***.println("Unsorted array:");

**for** (String element : stringArray)

System.***out***.print(element + " ");

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

*insertionSort*(stringArray);

System.***out***.println("Sorted array:");

**for** (String element : stringArray)

System.***out***.print(element + " ");

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

}

**static** **void** insertionSort(Comparable[] array) {

Comparable temp;

**for** (**int** i = 1; i < array.length; i++) // ar[i] is element to insert

{

temp = array[i];

**int** j = 0;

**for** (j = i; j > 0; j--)

**if** (temp.compareTo(array[j - 1]) < 0)

array[j] = array[j - 1];

**else**

**break**;

array[j] = temp;

}

}

}

**9.** **Write a program to get distinct word list from the given file.**

**10.** **Find longest substring without repeating characters.**

**public** **class** lengthOfLongestSubstring {

**public** **int** lengthOfLongestSubstring(String s) {

**int** n = s.length(), ans = 0;

**int**[] index = **new** **int**[128]; // current index of character

// try to extend the range [i, j]

**for** (**int** j = 0, i = 0; j < n; j++) {

i = Math.*max*(index[s.charAt(j)], i);

ans = Math.*max*(ans, j - i + 1);

index[s.charAt(j)] = j + 1;

}

**return** ans;

}

}

**11.** **Write a program to remove duplicates from sorted array**

**public** **class** removeDupsFromSortedArray {

**public** **static** **int**[] removeDupsFromSortedArray(**int**[] input) {

**int** j = 0;

**int** i = 1;

// return if the array length is less than 2

**if** (input.length < 2) {

**return** input;

}

**while** (i < input.length) {

**if** (input[i] == input[j]) {

i++;

} **else** {

input[++j] = input[i++];

}

}

**int**[] output = **new** **int**[j + 1];

**for** (**int** k = 0; k < output.length; k++) {

output[k] = input[k];

}

**return** output;

}

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

**int**[] input1 = { 0,-1, 2, 3, 4, 4, 8, 9, 10, 10, 10, 12, 12 };

**int**[] output = *removeDupsFromSortedArray*(input1);

**for** (**int** i : output) {

System.***out***.print(i + " ");

}

}

}

**12.** **Write a program to print fibonacci series.**

**import** java.util.Scanner;

**public** **class** FibonacciSeriesRecursive {

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

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter the value of n: ");

**int** n = s.nextInt();

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

System.***out***.print(*fibonacci*(i) + " ");// i=0; i=1; i=2; i=3

}

}

**public** **static** **int** fibonacci(**int** n) {

**if** (n == 0) {

**return** 0;

} **else** **if** (n == 1) {

**return** 1;

} **else** {

**return** *fibonacci*(n - 1) + *fibonacci*(n - 2);

}

}

}

**13.** **Write a program to find out duplicate characters in a string**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** findDuplicateCharacters {

**public** **void** countDupChars(String str) {

// Create a HashMap

Map<Character, Integer> map = **new** HashMap<Character, Integer>();

// Convert the String to char array

**char**[] chars = str.toCharArray();

/\*

\* logic: char are inserted as keys and their count as values. If map

\* contains the char already then increase the value by 1

\*/

**for** (Character ch : chars) {

**if** (map.containsKey(ch)) {

map.put(ch, map.get(ch) + 1);

} **else** {

map.put(ch, 1);

}

}

// Obtaining set of keys

Set<Character> keys = map.keySet();

/\*

\* Display count of chars if it is greater than 1. All duplicate chars

\* would be having value greater than 1.

\*/

**for** (Character ch : keys) {

**if** (map.get(ch) > 1) {

System.***out***.println("Char " + ch + " " + map.get(ch));

}

}

}

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

findDuplicateCharacters obj = **new** findDuplicateCharacters();

System.***out***.println("String: duckduckgo.com");

System.***out***.println("-------------------------");

obj.countDupChars("BeginnersBook.com");

System.***out***.println("\nString: ParulMekvan");

System.***out***.println("-------------------------");

obj.countDupChars("ParulMekvan");

System.***out***.println("\nString: #@$@!#$%!!%@");

System.***out***.println("-------------------------");

obj.countDupChars("#@$@!#$%!!%@");

}

}

**14.** **Write a program to create deadlock between two threads**

// dead lock between two threads

**public** **class** deadLock {

String str1 = "Hello";

String str2 = "World!";

Thread trd1 = **new** Thread("My Thread 1") {

**public** **void** run() {

**while** (**true**) {

**synchronized** (str1) {

**synchronized** (str2) {

System.***out***.println(str1 + str2);

}

}

}

}

};

Thread trd2 = **new** Thread("My Thread 2") {

**public** **void** run() {

**while** (**true**) {

**synchronized** (str2) {

**synchronized** (str1) {

System.***out***.println(str2 + str1);

}

}

}

}

};

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

deadLock mdl = **new** deadLock();

mdl.trd1.start();

mdl.trd2.start();

}

}

**15.** **Find out middle index where sum of both ends are equal**

**public** **class** middleIndexSumEqual {

**public** **static** **int** middleIndexSumEqual(**int**[] numbers) **throws** Exception {

**int** endIndex = numbers.length - 1;

**int** startIndex = 0;

**int** sumLeft = 0;

**int** sumRight = 0;

**while** (**true**) {

**if** (sumLeft > sumRight) {

sumRight += numbers[endIndex--];

} **else** {

sumLeft += numbers[startIndex++];

}

**if** (startIndex > endIndex) {

**if** (sumLeft == sumRight) {

**break**;

} **else** {

**throw** **new** Exception("Please pass proper array to match the requirement");

}

}

}

**return** endIndex;

}

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

**int**[] num = { 2, 4, 4, 5, 4, 1 };

**try** {

System.***out***.println("Starting from index 0, adding numbers till index " + *middleIndexSumEqual*(num) + " and");

System.***out***.println("adding rest of the numbers can be equal");

} **catch** (Exception ex) {

System.***out***.println(ex.getMessage());

}

}

}

**16.** **Write a program to find the given number is Armstrong number or not?**

**import** java.util.Scanner;

**public** **class** armstrongNumber {

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

Scanner ob = **new** Scanner(System.***in***);

System.***out***.println("Enter any number to check whether it is armstrong or not:");

**int** n = ob.nextInt();

**int** r, sum = 0, temp = n;

**while** (n > 0) {

r = n % 10;

n = n / 10;

sum = sum + (r \* r \* r);

}

**if** (sum == temp)

System.***out***.print("Given number " + temp + " is Armstrong");

**else**

System.***out***.println("Given number " + temp + " is not Armstrong");

}

}