**Abstract vs interfaces**

**+** An interface is a description of the behaviour an implementing class will have. The implementing class ensures, that it will have these methods that can be used on it. It is basically a contract or a promise the class has to make.

public interface Payment { void makePayment();//by default it is a abstract method }

public class PayPal implements Payment {

public void makePayment() {

//some logic for PayPal payment

//e.g. Paypal uses username and password for payment }

}

public class CreditCard implements Payment {

public void makePayment() {

//some logic for CreditCard payment

//e.g. CreditCard uses card number, date of expiry etc...}

}

+ An abstract class is a basis for different subclasses that share behaviour which does not need to be repeatedly created. Subclasses must complete the behaviour and have the option to override predefined behaviour (as long as it is not defined as final or private).

public abstract class Burger {

public void packing() {

//some logic for packing a burger }

public abstract void price(); //price is different for different categories of burgers

}

public class VegBerger extends Burger {

public void price() {

//set price for a veg burger. }

}

public class NonVegBerger extends Burger{

public void price() {

//set price for a non-veg burger. }

}

**What is object-oriented programming**?

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

**What are the main principles of OOP?**

**4 main principles of OOP**

+ **Encapsulation** - CricketScorer

In programming languages everything should be properly encapsulated, you should not allow a particular class variable to be modifed outside that particular class. We don't expose the date outside.

--> only call the methods, but not the instance of the class

+ **Abstraction** - Mobile Phone - Hiding the complexity is hiding the complexity, and let the other developers use the methods which we already created.

+ **Inheritance** - Inheritances expresses "is a" relationship between two objects. Using proper inheritance, In derived classes we can reuse the code of existing super classes.

ConstructorExamples : ex: class Animal. Dog extends Animal, but the dog gets name from Animal

+ **Polymorphism** - PolymorphismAbstractClassTest

ex: class Aniaml. Dog extends Animal, Cat extends Animal.

Animal[] animals = {new Dog(), new Cat()};

They all have the same method bark(), but different behavior.

Whatever is the reference the object that is pointed to

the method which is invoked is different. Polymorphism is called same method, but different behavior.

**What is the difference between an object and a class?**

+ A **class** is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object.

+ **Methods** are nothing but members of a class that provide a service for an object or perform some business logic.

**Understanding of java program**

\_ **class** key word is used to decalre a class in java.

\_ **public** keyword is an acess modifier which represents vidibility,

it means it is visible to all.

\_ **static** is a keyword, if we declare any method as static, it is known

as static method. The point is there is no need to create object to invoke

static method. The main method is executed by the JVM (Java virtual machine)

, so it doesn't require to create object to invoke the main method.So it saves memory.

\_ **void** is the return type of the method, it means it doesn't return any value

\_ main represents startup of the program

\_ **String[] args** is used for command line argument.

javac ...

java file.java "input1" "input2" ...

**What happnens at complie time?**

At complie time, java file is complied by Java Compiler and converts the java code into bytecode.

Java code ----> complier ----> Byte code simple.java simple.class

**What happens at runtime?**

class file ----> class loader ---> bytecode verifier ---> interpreter ---> Runtime ---> Hardware

\_**Classloader**: is the subsytem of JVM that is used to load class files.

\_**Bytecode Verifier**: checks the code fragments for illegal code that can violate access right to objects

\_**Interpreter**: read bytecode stream then execute the instructions

**Can you save a java source file by other name than the class name?**

Yes, if the class is not public. Hello.java ---> complier ---> Simple.class

Ex: To **compile**: javac Hello.java

To **execute**: java Simple

**Can you have multiple classes in a java source file? Yes**

Hello.java : classA, classB, ClassC. ----> complier ---> A.class, B.class, and C.class

**Difference between JDK, JRE and JVM**

\_ **JVM(Java Virtual Machine**) is an abstract machine. It is a specification that provides runtime enironment in which java bytecode can be executed.

+ It is runtime instance, whenever we write java command on the command prompt to run java class, an instance of JVM is created.

+ JVMs are availabe for many hardware and software plaforms. JVM, JRE, and JDK are platform dependent because of configuration of each OS differs. But, Java is platform independent

+ The JVM performs following main tasks:

\_ Loads code \_ Verifies code \_ Executes code \_ Provides runtime environment

+ **JVM** provides defintions for the :

\_ Memory area \_ Class file format\_ Register set \_ Garbage-collected heap \_ Fatal error reporting etc.

+ **Internal Architecture of JVM**

\_ It contains classloader, memory area, execution engine etc.

1) **Classloader**:

Classloader is a subsystem of JVM that is used to load class files

2) **Class**(Method) Area:

Class(Method) Area store per-class structres such as the runtime

constant pool, field and method data, the code for methods

3) **Heap**:

It is the runtime data area in which objects are allocated.

4) **Stack**:

+ Java Stack stores frames. It holds local variables and partial

results, and plays a part in method invocation and return.

+ Each thread has a private JVM stack, created at the same time

as thread.

+ A new frame is created each time a method is invoked.

A frame is destroyed when its method invocation completes

5) **Program Counter Register** (PC register)

+ PC (program counter) register. It contains the address of

the Java vritual machine instruction currently being executed.

6) Native Method Stack

+ It contains all the native methods used in the application.

7) Execution Engine

Contains:

+ A virtual processor

+ Interpreter: Read bytecodes stream, then execute the instructions

+ Just-In-Time(JIT) compiler:

It is used to improve the performance. JIT compiles parts of

the byte code that have similar functionality at the same time,

and hence reduces the amount of time needed for compilation.

the term "compiler" here refers to a translator from the

instruction set of a Jav vritual machine (JVM) to the instruction

set of a specific CPU.

\_ **JRE stands** for Java Runtime Enviroment. It is used to provide runtime

environment. It is the implementation of JVM (prodi). It physically exists.

It contains set of libraries + other files that JVM uses at runtime.

\_ **JDK** stands for Java Deveploment Kit. It physically exists. It contains JRE + development tools(javac, java etc...), comilers + debuggers.

**Inheitance:** is a mechanism in which one object accquires all the properties and behaviors of parent object.

The idea of inheritance is that you we can create new classes that are built upon existing classes.

\_When we inherit from an existing class, we can resue methods and fields of parent class, and we can add new methods and fields also.

class Subclass-name extends Superclass-name{ //methods and fields }

\_The **extends** keyword indicates that we are making a new class that derives from an existing class.

\_ In java, a class which is inherited is called parent or super class and the new class is called child or subclass.

**Why multiple inheritance is not supported in java? 🡪** Complie Time Error. Sincde complie time errors are better than runtime errors, java renders complie time error if we inherit 2 classes. So whether we have the same method or different, there will be complie time error.

class A{ void msg(){System.out.println("Hello");} }

class B{ void msg(){System.out.println("Welcome");} }

class C extends A,B{//suppose if it were

Public Static void main(String args[]){ C obj=new C();

obj.msg();// Now which msg() method would be invoked? } }

**Why is Java so popular?** 1. PlatformIndependence 2. Object oriented language.

1. Platform Independence: when we compile a java program and build a jar, we can run the jar (compiled java program) in any Operating System.

On compiling a java file the output is a class file – which contains an internal java representation called bytecode. JVM converts bytecode to executable instructions. The executable instructions are different in different OS. So There are different JVM’s for different OS. However, the JV< understand the bytecode and convert it to the executable code for the respective OS.

**What is ByteCode?**

Java bytecode is the instruction set of JVM. Each bytecode is representing the instruction (opcode), along along with zero or more bytes for passing parameter.

**What are the important differences between C++ and Java?**

1. Java is platform independent. C++ is not.
2. C++ has pointers (access to internal memory). Java has no concept called pointers.
3. Garbage Collection.
4. C++ supports Multiple inheritance. Java does not.

**What are wrapper classes?**

A primitive wrapper class in the Java programming language is one of eight classes provided in the java.lang package to provide object methods for the eight primitive types. All of the primitive wrapper classes in Java are immutable.

**Wrapper**: Boolean,Byte,Character,Double,Float,Integer,Long,Short

**Primitive:** boolean,byte,char ,double, float, int , long,short

**Methods that support Object like creation from other types**

Integer number2=new Integer("55");//String

Integer number = new Integer(55);//int

Integer number2 = new Integer("55");//String

Float number3 = new Float(55.0);//double argument

Float number4 = new Float(55.0f);//float argument

Float number5 = new Float("55.0f");//String

Character c1 = new Character('C');//Only char constructor

//Character c2 = new Character(124);//COMPILER ERROR

Boolean b = new Boolean(true);

//"true" "True" "tRUe" - all String Values give True //Anything else gives false

Boolean b1 = new Boolean("true");//value stored - true

Boolean b2 = new Boolean("True");//value stored - true

Boolean b3 = new Boolean("False");//value stored - false

Boolean b4 = new Boolean("SomeString");//value stored – false

**valueOf Static Methods**

Integer hundred = Integer.valueOf("100");//100 is stored in variable

Integer seven = Integer.valueOf("111", 2);//binary 111 is converted to 7

**What is Casting?** Casting is used when we want to convert on data type to another

Two types of casting: Implicit Casting and explicit Casting.

Implicit Casting: int value = 100; long number value; // Implicit casting, float f = 100; // Implicit casting

Explicit Casting: long number1 = 123; int number2 = (int) number1; // Explicit Casting

// int x = 35.35; // COMPILER ERROR; int x = (int) 35.35; // Explicit Casting

\_ Explicit casting would cause truncation of value if the value stored is greater than the size of the variable.

int bigVale = 280;

byte small = (byte) bigValue;

System.out.println(small); // output 24, only 8 bits remain

**Are all String’s immutable? Yes.**

Value of a String Object once created cannot be modified. Any modification on a String object creates a new String object.

String str1 = “value1”; str1.concat(“value2”); System.out.println(str1); // value1

String newString = str1.concat(“value2”); System.out.println(newString); // value1value2

**Where are string values stored in memory?**

The location where the string values are stored in memory depends on how we create them.

Ex1: String str1 = "value"; this value will be stored in the Heap memory. If if compiler finds a String literal, it checks if it exists. If it exists, it is resued.

Ex2: If new operator is used to create string object, the new object is created on the heap. There will not be any reuse of values. String str2 = new String(“value”)

**What are differences between String and StringBuffer?**

\_ Objects of type String are immutable. StringBuffer is used to represent values that can be modified.

\_ In situations where values are modified a number to times, StringBuffer yields significant performance benefits.

\_ Both String and Stringbuffer are thread-safe.

\_ StringBuffer is implemented by using synchronized keyword on all methods.

**What are differences between StringBuilder and StringBuffer?**

StringBuilder is not thread safe. So, it performs better in situations where thread safety is not required.

**Can you give examples of different utility methods in String class?**

String class defined a number of methods to get information about the string content.

String str = “abcdef”;

str.chartAt(2), str.length(), “abc”.toString(), substring(2), substring(3,7)

**Object Oriented Programming Basics**

**Class:**  is a template for creating multiple objects. A class defines state and behavior that an object can exhibit.

**Object:** An instance of a class.

**What is state of Object?**

Values assigned to instance variables of object. The value of the variable is changed. Ex: score.six(), score.four();

**What is behavior of an Object?**  Methods supported by an object. Ex: six(), four(), getScore()

**What is the supper class of every class in Java?**  Every class in java is a sub class of the class Object. When we create a class we inherit all the methods and properties of Object class.

Ex: String str = “Testing”; System.out.println(str.toString()); str.hashCode(); str.clone();

if (str instanceof Object) { System.out.println(“I extend Object”); }// will be printed

\_ toString, hashCode and clone methods fro String class are inherited from Object class and overridden

**What is the use of equals method in Java?**

Compare 2 objects.

Class Client{

private int id;

public Client (int id) {

this.id = id;

}

@Override

public Boolean equals(Object obj) {Client other = (Client) obj; if(id != other.id) return false; return true}

} ---🡪 client1.equals(client2) OR client1 == client2

== comparison operator checks if the object references are pointing to the same object. It does NOT, then look at the content of the object.

**Modifiers**

**What is default class modifier?**

\_ A class is called a Default Class is when there is no access modifier specified on a class.

\_ Default classes are visible inside the same package only.

\_ Default access is also called Package access.

**What is private access modifier?**

1. Private variables and methods can be accessed only in the class they are declared.
2. Private variables and methods from SuperClass are NOT available in Subclass

**What is default or package access modifier?**

1. Default variables and methods can be accessed in the same package Classes.
2. Default variables and methods from SuperClass are available only to SubClasses in same package.

**What is protected access modifier?**

1. Protected variables and methods can be accessed in the same package Classes.
2. Protected variables and methods from SuperClasss are available to SubClass in any package

**What is the use of a final modifier on a class?**

Final class cannot be extended. Ex of Final class in Java is the String class. Final is used very rarely as it prevents re-use of the class.

**What is the use of a final modifier on a method?** Final methods cannot be overridden.

**What is a Final variable?** Once initialized a final variable, the value of a final variable cannot be changed.

Final variable ex: java.lang.Math.PI

**What is a Static Variable?** Static variables and methods are class level variables and methods. There is only one copy of the static variable for the entire Class. Each instance of the Class (object) will Not have a unique copy of a static variable.

**Ex:** public class Test {private static int count; public Test() { count++; } ; static int getCount(){ return count; }}

main(){ Test a = new Test(); Test b = new Test(); Test c = new Test(); Test d = new Test(); println(Test.getCount()) } -🡪 4

**DEFAULT VALUES**

**Byte, short, int, long: 0 ;; float, double: 0.0 ;; boolean : false ;; object : null**

**Printing a 2D Array:** int[][] matrix = {{1,2,3}, {4,5,6}} 🡪 println(Arrays.toString(matrix)) 🡪 [[@2343],[@24]] -🡪 should be: Arrays.deepToString(maxtrix) [[1,2,3],[4,5,6]]

**Compare two Arrays:** Arrays.equals(array1,array2);

**When is Garbage Collection run?**

Garbage Collection runs when: 1) available memory on the heap is low, CPU is free

**What are best practices on Garbage Collection?**

We can request JVM to run Garbage Collection by calling System.gc() method.

JVM might throw an OutOfMemoryException when memory is full and no objects on the heap are eligible for garbage collection.

**Collections**

**Why do we need Collections in java?**

Array are not dynamic. Once an array of a particular size is declared, the size cannot be modified. To add a new element to the array, a new array has to be created with bigger size and all the elements from the old array copied to new array.

Collections are used in situations where data is dynamic. Collections aalow adding an element, deleting an element and host of other operations. There are a number of Collections in Java allowing to choose the right Collection for the right context.

**What are the important interfaces in the Collection Hierarchy?**

interfact Collection<E> extends Iterable<E> {}

interface Set<E> extends Collection<E> {}

// Unique thing only. Does not allow duplication.

// If obj1.equals(obj2) then only one of them can be in the Set.

Interface List<E> extends Collection<E> {}

**//** list of things. Cares about which position each object is in. Elements can be added in by specifying position – where should it be added in

**//** If element is added without specifying position – it is added at the end

interface Queue<E> extends Collection<E> {}

// Arranged in order of processing – A to-do list for example

// Queue interface extends Collection. So, it supports all Collection Methods

interface map<K,V> {}

// A,C,A, C,E,C,M,D,H,A => {(“A”, 5), (“C”, 2)}

// Key – Value Pair {[“key1”, value1], [“key2”, valu2] ,…}

// Things with unique identifier.

**What are the important methods that are declared in the Collection Interface?**

Most important methods declared in the collection interface are the methods to add and remove and element.

Add method allows adding an element to a collection and delete method allows deleting an element from a collection.

**size()** methods returns number of elements in the collection. Other important methods are:

interface Collection<E> extends Iterable<E> {

boolean add(E paramE);

Boolean remove(Object paramObject);

int size();

boolean isEmpty();

void clear();

boolean contains(Object paramObject);

boolean containsAll(Collection<?> paramCollection);

boolean addAll(Collection<? Extends E> paramCollection);

boolean removeAll(Collection<?> paramCollection);

boolean retainAll(Collection<?> paramCollection);

Iterator<E> iterator();

// A NUMBER OF OTHER METHODS AS WELL…

}

**List Interface?**

List interface extends Collection interface.

Interface List<E> extends Collection<E> {

Boolean addAll(int paramInt, Collection<? Extends E> paramCollection);

E get(int paramInt); E set(int paramInt, E paramE);

Void add(int paramInt, E paramE); E remove(int paramInt);

int indexOf(Object paramObject); int lastIndexOf(Object paramObject);

ListIterator<E> listIterator();

ListIterator<E> listIterator(int paramInt);

List<E> subList(int paramInt1, int paramInt2);

}

**ArrayList**

ArrayList implements the list interface. So, ArrayList stores the lements in insertion order (by default).

Element’s can be inserted into and removed from ArrayList based on their position.

Ex: Instantiate an ArrayList of integers: List<integer> integers = new ArrayList<Integer>();

Add: intergers.add(5); // new Integer(5)

**Can an ArrayList have Duplicate elements?**

ArrayList can have duplicates (since List can have duplicates).

List<String> arrayList = new ArrayList<String>();

// adds at the end of list

arraylist.add(“Sachin”); // [Sachin] arraylist.add(“David”); // [Sachin, David]

println(arrayList.size()) // 2 ; println(arrayList.contains(“David”); // true

**How do you iterate around an ArrayList using Iterator?**

Iterator<String> arrayListIterator = arraylist.interator();

while(arrayListIterator.hasNext()) {

String str = arraylistIterator.next();

Printn(str); // Prints the 4 anmes in the list on separate lines.

}

**How do you sort an ArrayList?**

List<String> numbers = new Ar;rayList<String>();

numbers.add(“one”); numbers.add(“two”) ; numbers.add(“three”); numbers.add(“four”);

println(numbers); // [one, two, three, four]

Collections.sort(numbers); // Strings – By default –are sorted alphabetically

println(numbers); // [four, one, three, two]

**How do you sort elements in an ArrayList using Comparable interface?**

class Test implements comparable<Test> {

int runs; String name;

public Test(String name, int runs) {

super(); this.name = name; this.runs = runs;

}

@Override

public String toString() {return name + “ ” + runs;}

@Override

public int compareTo(Test that) {

if(this.runs > that.runs) return 1;

if(this.run < that.runs) return -1;

return 0;

}

}

-🡪 Collections.sort(…); println(…)

**How do you sort elements in an ArrayList using Comparator interface?**