### What is JVM and explain me the Java memory allocation?

JVM is java virtual machine or abstract machine which enables a computer to run a java program.

JVM includes concepts as: specification, implementation, and instance. The specification is a document that formally describes what is required of a JVM implementation. A JVM implementation is a computer program that meets the requirements of the JVM specification. An instance of a JVM is an implementation running in a [process](https://en.wikipedia.org/wiki/Process_(computing)) that executes a computer program compiled into [Java bytecode](https://en.wikipedia.org/wiki/Java_bytecode). Java bytecode is platform independent whereas JVM is platform dependent which executes bytecode.

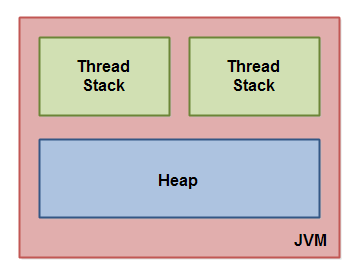
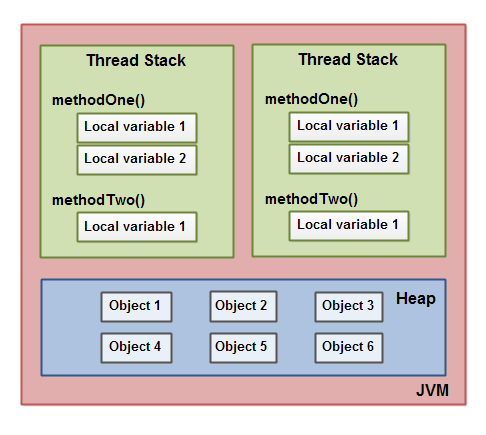
When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.

### JVM internal Architecture:



### Java memory allocation:

The Java memory allocation specifies how the Java virtual machine works with the computer's memory (RAM). JVM divides memory between thread stacks and the heap like:

Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. Also called as ‘call stack". As the thread executes its code, the call stack changes.

The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

A local variable is totally kept on the thread stack.

A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.

An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

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

**Polymorphism**:

Performing a single action by different ways is called Polymorphism in java.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism which is achieved by using method overloading and method overriding.

Method overloading is compile time polymorphism and method overriding is run time polymorphism.

In Runtime Polymorphism is also achieved with multilevel inheritance but Runtime polymorphism can't be achieved by data members.

In compile Time polymorphism, the compiler is able to select and bind the appropriate method to the object for a particular call at compile time itself.

Polymorphism is mainly used for flexibility.

**Encapsulation**:

In java, encapsulation is a process of wrapping code and data together into a single unit.

It is also called as **data hiding.** We can create a fully encapsulated class in java by making all the data members of the class private. Setter and getter methods are used to set and get the data in it. Encapsulation provides control over the data.

Objects encapsulate data and implementation details. To the outside world, an object is a black box that exhibits a certain behavior.

The behavior of this object is what which is useful for the external world or other objects.

An object exposes its behavior by means of public methods or functions.

The set of functions an object exposes to other objects or external world acts as the interface of the object.

**Advantages of Encapsulation:**

1. It improves maintainability and flexibility and re-usability
2. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.
3. **What is method overloading and Method over riding?**

**Method overloading:**

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type (One type is promoted to another implicitly if no matching datatype is found)

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity

## **Advantage:**

Increases the readability of the program.

**Method over riding:**

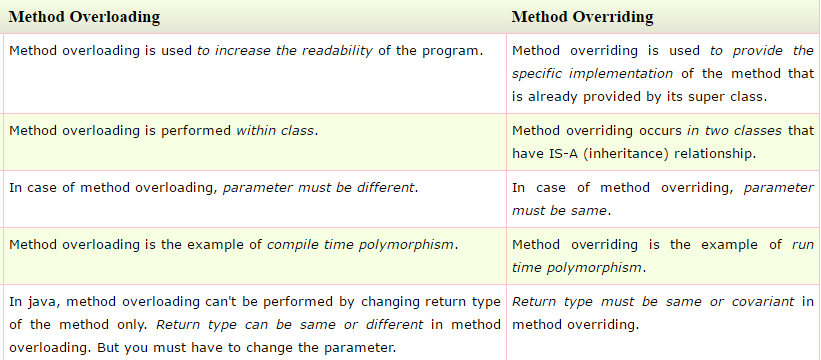
If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding. Means child class has the same as that of parent class.

### Usage of Java Method Overriding

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).



1. **Why string is Immutable?**

Immutable means unchangeable or unmodifiable that means once a string object is created, its data or state cannot be changed but a new string object is created.

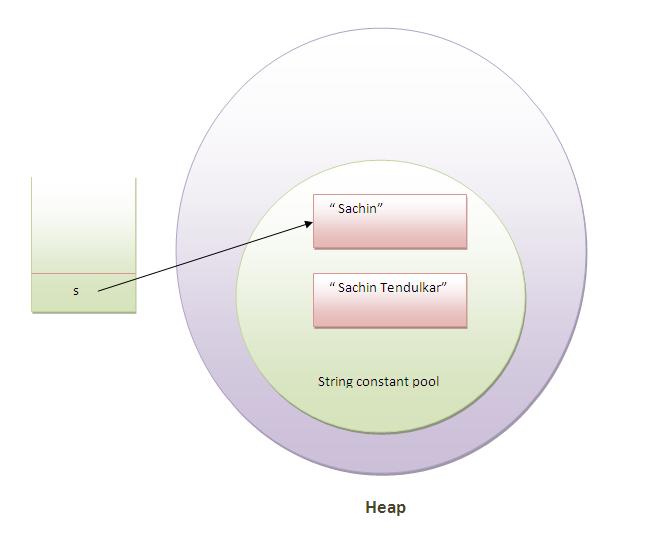
Java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java

e.g.

String s="Sachin";

s.concat(" Tendulkar");

System.out.println(s);



Here, in this example, Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.

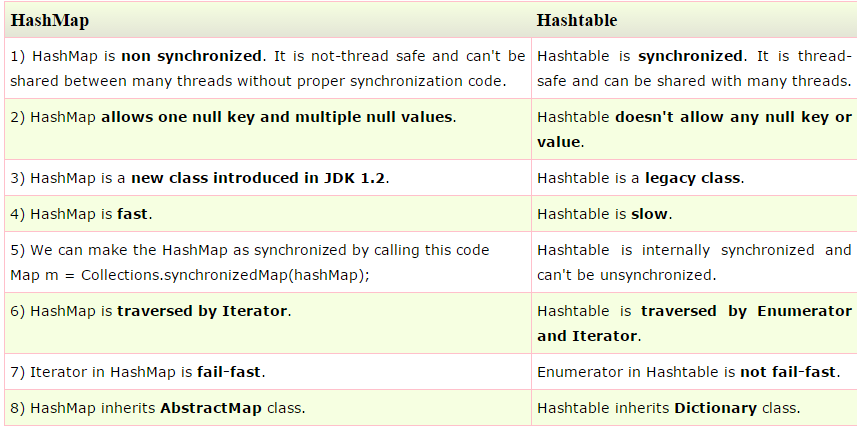
1. **What is the difference between String and String buffer?**

|  |  |  |
| --- | --- | --- |
| Sr. No | **String(String memory)** | **String buffer (Heap memory)** |
| 1 | Performing any operation on String will create new String object i.e. string is immutable | modifying StringBuffer object won't create a new object i.e. StringBuffer is mutable |
| 2 | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| 3 | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |
| 4 | Performance is slow | Performance is fast |

1. **What is the difference between array and array list?**

|  |  |  |
| --- | --- | --- |
| Sr. No | Array | ArrayList |
| 1 | Fixed length data-structure and nature is static | Dynamic i.e. can re-size itself when needed |
| **2** | Cannot use Generics along with Array. | Allows you to use Generics to ensure type-safety.  e.g. list<String> a= new list<String>();  In this ‘String’ is generics. |
| 3 | Uses arrayname.Lenght() method to get the length of array | Uses Size() Method to get length of arraylist |
| 4 | Can contain both primitive & objects in java | Contains only objects |
| 5 | Use assignment operator to store elements into array | Uses Add() method to store elements into arrayList |
| 6 | Its mandatory to provide size of array while creating it | Can create instance of arrayList without specifying size and creates with default size |

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



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

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements. Like an array, elements of a Vector object can be accessed via an index into the vector.

While creating array we need to specify size; but this is not the case with vector. It shrinks and grows automatically when necessary.

Vector implements list. The default capacity is 10. But when it exceeds the limit then capacity increases by 100%.

Vector came in JDK 1.0 First time when Java introduced, it came with Vector

|  |  |  |
| --- | --- | --- |
| Sr. No | Vector | Array List |
| 1 | Checks for the array capacity using capacity (). | Don’t have any method to check capacity |
| 2 | Capacity increased by 100% | Dynamic capacity increases by 50% |
| 3 | Synchronized (Every method in Vector is synchronized) | Unsynchronized |
| 4 | Thread-Safe | Not Thread-Safe |
| 5 | Performance is Slow.  (So when it is asked to use Vector, then only use it. Otherwise use arraylist) | Performance is Fast so preferred over vector |
| 6 | As every time it increases capacity by 100%, vector waste memory | Saves memory as it increases by 50% |

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

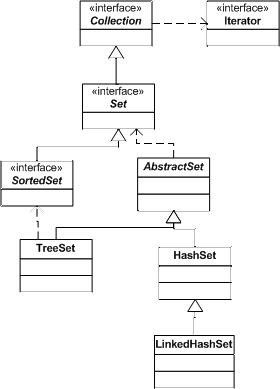
Set is a collection that store only unique values.

Sets allows Set instances to be compared meaningfully even if their implementation types differ.

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet.

 Set can contain no more than one null entry. A Set is not required to maintain the elements of the Set in the order they are added although the LinkedHashSet does maintain order.

We can add elements to set using add (). Also we can find intersection of two sets using retainAll() and difference of two sets using removeAll().



Relationships between the classes and interfaces of the Set interface.

 Key methods of the Set interface:

* **boolean add(Object)** - Ensures the Set holds the argument. The Object is added only if it isn't already in the Set. Returns false if the Object was not added to the Set.
* **void clear()** - Removes all elements from the Set.
* **boolean contains(Object)** - Returns true if the Set contains the argument.
* **boolean isEmpty()**- Returns true if the Set contains no elements.
* **Iterator iterator()** - Returns an Iterator object which can be used to traverse through the Set. (See last month's article for information on the Iterator interface.)
* **boolean remove(Object)** - Removes the argument from the Set. Returns true if the argument was removed.
* **int size()**- Returns the number of elements in the Set.
* **Object[] toArray()** - Returns an Object array containing all the elements in the Set

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

Abstract classes are classes that contain one or more abstract methods and a class that is declared with abstract keyword. It cannot be instantiated(in child class and not in abstract class)and require subclasses to provide implementations for the abstract methods.

Abstract keyword is used to denote both an abstract method, and an abstract class.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

By using abstract classes, you can inherit the implementation of other (non-abstract) methods. You can't do that with interfaces - an interface cannot provide any method implementations

e.g. abstract class A{} ;

The abstract class can also be used to provide some implementation of the interface.

interface A{

void a();

void b();

void c();

void d();

}

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

1. **What is an interface?**

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

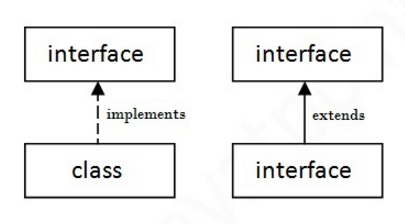
Java Interface also **represents IS-A relationship**. It cannot be instantiated just like abstract class.

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

Interface uses:

* It is used to achieve fully abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

An interface extends another interface but a **class implements an interface**.

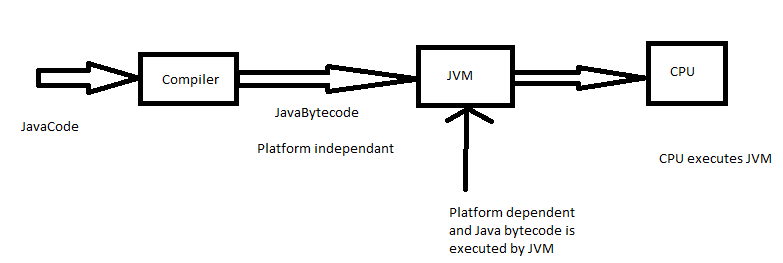


1. **Why Java is Platform independent?**

(Write once and Run anywhere)

The javacode that compiler gets converted to a form known as a bytecode that is present in the “.class” files. Every operating system (Windows. linux etc) has its own JVM.

The JVM takes generated bytecode and then executes the program. So the hardware dependency is taken care of by the JVM. Hence java is **Platform independent.**

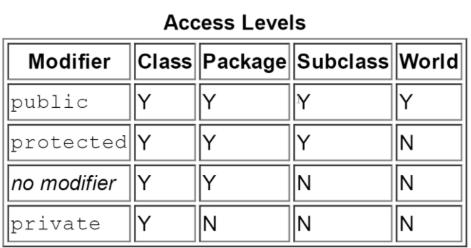


1. **What are access modifiers? Give me an example?**

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public



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

Exception is abnormal condition or exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

As per sun microsystem,there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No | Checked Exception | Unchecked Exception | Error |
| 1 | Occurs at compile time | Occurs at runtime | Problems beyond user control |
| 2 | Cannot be ignored and need to handle | Ignored at the time of compilation | Ignored at the time of compilation |
| 3 | e.g.FilenotFoundException, IOException, SQLException | ArrayIndexOutofBounds | OutOfMemoryError, VirtualMachineError, AssertionError |

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws
6. **What is the difference between throws and throwable?**

**Throwable:** The Throwable class is the superclass of all errors and exceptions in the Java language.  This is the class which you must extend in order to create your own, custom, throwable..

**Throws**: Used when writing methods, to declare that the method in question throws the specified (checked) exception.

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

Copy from notes.

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

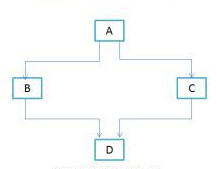
|  |  |  |  |
| --- | --- | --- | --- |
| No. | Errors | Throwable | Exception |
| 1 | Errors in java are of type java.lang.error. | The Throwable class is the superclass of all errors and exceptions in the Java language | Exceptions are type java.lang.Exception |
| 2 | All errors in java are unchecked type. | Checked and Unchecked type | Checked and Unchecked type |
| 3 | Happen at run time | Run & compile time | Run & compile time |
| 4 | Impossible to handle |  | Can be handled using try-catch or finally block |

1. **What are collection APIs, give me an example.**
2. **What is the difference between final and finally?**

|  |  |  |
| --- | --- | --- |
| **No.** | **Final** | **Finally,** |
| 1 | final keyword can be used along with variable, method and Class in Java.  Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally, is used for exception handling along with try and catch Finally is used to place important code, it will be executed whether exception is handled or not |
| 2 | Is a Keyword | Is a block |

1. **Will java support multiple inheritance?**

Java doesn’t support multiple inheritance. It is just to **remove ambiguity,** because **multiple inheritance** can cause ambiguity in few scenarios. One of the most common scenario is **Diamond problem.**   
E.g. Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C. If a method is in class A and class B & C overrides that method in their own way. The problem here is D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). Ambiguity. That’s the main reason why Java doesn’t support multiple inheritance.



### How to achieve multiple inheritance in Java using interfaces?

interface X

{

public void myMethod();

}

interface Y

{

public void myMethod();

}

class Demo implements X, Y

{

public void myMethod()

{

System.out.println(" Multiple inheritance example using interfaces");

}

}

Here the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. This is because, methods in an interface are always [**abstract**](http://beginnersbook.com/2013/05/java-abstract-class-method/) by default, which doesn’t let them to give their implementation (or method definition) in interface itself.

1. **What are the different types of interface?**

The core collection interfaces encapsulate different types of collections. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework.



* Set-
  + Set is a collection that store only unique values.
  + Sets allows Set instances to be compared meaningfully even if their implementation types differ.
  + Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet.
  + Set can contain no more than one null entry. A Set is not required to maintain the elements of the Set in the order
* List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
* Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules.

* Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see [The Deque Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) section.

* Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

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

In Java, a wrapper class is defined as a class in which a primitive value is wrapped up. These primitive wrapper classes are used to represent primitive data type values as objects. The Java platform provides wrapper classes for each of the primitive data types. For example, Integer wrapper class holds primitive ‘int’ data type value. Similarly, Float wrapper class contain ‘float’ primitive values, Character wrapper class holds a ‘char’ type value, and Boolean wrapper class represents ‘boolean’ value.

E.g.

(Wrapping- **Integer Wrapper class**)

int i = 26; // Primitive data type 'int'

Integer obj = new Integer(i);----------------------------------- Wrapping.

// Unwrapping primitive data 'int' from wrapper object

int i1 = obj.intValue();

List of all Wrapper Classes in java

|  |  |
| --- | --- |
| **Primitive data type** | **Wrapper class** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

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

Java 1.5 introduced a special feature of auto conversion of primitive types to the corresponding Wrapper class and vice versa.

**Autoboxing**: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

E.g.

ArrayList<Integer> arrayList = new ArrayList<Integer>();

arrayList.add(11);

**Unboxing**: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

E.g.

Integer inum = new Integer(5);

int num = inum;

1. **Explain for each loop**

‘For each’ loop also called as ‘Advanced or Enhanced For loop’.

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements.

Advantage of for-each loop:

* + 1. It makes the code more readable.
    2. It eliminates the possibility of programming errors.

Syntax:

**for**(data\_type variable : array | collection){}

Example:

**class** A{

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

**int** a1[]={10,20,30,40};

**for**( **int** i:arr){

     System.out.println(i);

   }

 }

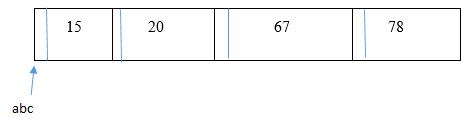
}

Output: 10,20,30,40

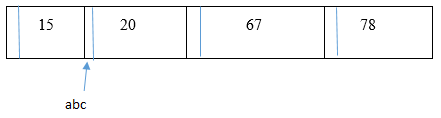
1. **What are iterators, explain with an example**
   * Iterator is an object that implements either the Iterator or the ListIterator interface.
   * Iterator enables to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
   * Each of the collection classes provides an iterator ( ) method that returns an iterator to the start of the collection. By using this iterator object, each element in the collection can be accessed, one element at a time.
   * For creating any iterator we cannot use new iterator syntax as iterator works differently for different collection framework means different implementations for differenet data structure.
   * E.g. iterator for Arraylist:

ArrayList<integer> list = new ArrayList<integer>();

Iterator abc= list.iterator();

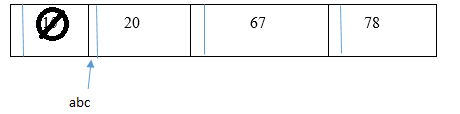


* + - Iterator can not start at the value but before the value.
    - iterator.next() method will move abc to next element and it will consider in between values.
    - E.g.abc.next();



Iterator also have remove method which remove value before that iterator.

e.g. abc.next();



1. **How do you access Private variables in different class.**

To access private variables from class 1 into class 2, we can’t use object reference of class1. Here we can use getter-setter functions. These are public functions and with the help of these functions we can use private variables from another class.

e.g.

class Clock {

String time;

void setTime (String t) {

time = t;

}

String getTime() {

return time;

}

}

class ClockTestDrive {

public static void main (String [] args) {

Clock c = new Clock;

c.setTime("12345")

String tod = c.getTime();

System.out.println(time: " + tod);

}

}

* 1. **What is Constructor Over loading?**

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

class Student5{

int id;

String name;

int age;

Student5(int i,String n){

id = i;

name = n;

}

Student5(int i,String n,int a){

id = i;

name = n;

age=a;

}

void display(){System.out.println(id+" "+name+" "+age);}

public static void main(String args[]){

Student5 s1 = new Student5(111,"Karan");

Student5 s2 = new Student5(222,"Aryan",25);

s1.display();

s2.display();

}

}

Output:

111 Karan 0

222 Aryan 25

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

The **super** keyword in java is a reference variable that is used to refer immediate parent class object. Whenever the instance of subclass is created, an instance of parent class is created implicitly i.e. referred by super reference variable.

## **Usage of java super Keyword**

1. super is used to refer immediate parent class instance variable.

class Vehicle{

int speed=50;

}

class Bike4 extends Vehicle{

int speed=100;

void display(){

System.out.println(super.speed);//will print speed of Vehicle

}

public static void main(String args[]){

Bike4 b=new Bike4();

b.display();

}

}

**Output**:50

1. super() is used to invoke immediate parent class constructor.

class Vehicle{

Vehicle(){System.out.println("Vehicle is created");}

}

class Bike5 extends Vehicle{

Bike5(){

super();//will invoke parent class constructor

System.out.println("Bike is created");

}

public static void main(String args[]){

Bike5 b=new Bike5();

}

}

**Output**: Vehicle is created

Bike is created

1. super is used to invoke immediate parent class method.

class Person{

void message(){System.out.println("welcome");}

}

class Student16 extends Person{

void message(){System.out.println("welcome to java");}

void display(){

message();//will invoke current class message() method

super.message();//will invoke parent class message() method

}

public static void main(String args[]){

Student16 s=new Student16();

s.display();

}

}

**Output**: welcome to java

welcome

---------------------------------------------------------------------------------------------------------------------

