**JAVA**

Getting Started with Java

DAY-1:

* What is Java?

Java is a OOPs programming language that has many advantages.

**Importance** of core java:-

3 billion devices run on the java language only.

1) Java is used to develop Desktop Applications such as Media Player, Antivirus etc.

2) Java is Used to Develop Web Applications such as sravyajobs.com, irctc.co.in etc.

3) Java is Used to Develop Enterprise Application such as Banking applications.

4) Java is Used to Develop Mobile Applications.

5) Java is Used to Develop Embedded System.

6) Java is Used to Develop Smart Cards.

7) Java is Used to Develop Robotics.

8) Java is used to Develop Games …………………… etc.

**Features**: Important features of Java are- Object Oriented, Platform Independent, Simple, Secure, Architecture-neutral, Portable, Robust, Multithreaded, Interpreted, Distributed.





* The Java Platform

**Architecture:**



**Technologies Depends on Core java:-**



* • Obtaining Java
* ***Install the software and set the path :-***
* *1) Download the software.*
* *2) Install the software in your machine.*
* *3) Set the environmental variable.*
* Download the software from internet based on your operating system. The software is different
* from 32-bit operating and 64-bit operating system.
* To download the software open the fallowing web site.
* http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
* for 32-bit operating system please click on
* Windows x86 :- 32- bit operating system
* for 64-bit operating system please click on
* Windows x64 :- 64-bit operating system
* After installing the software the java folder is available in the fallowing location
* Local Disk c: ------program Files--------java----jdk(java development kit),jre(java
* runtime environment)
* **To set the environmental variable:-**
* My Computer (right click on that) ---->properties----->Advanced--->Environment Variables---->
* User variables--new---->variable name : Path
* Variable value: C:\programfiles\java\jdk1.6.0\_11\bin;.;
* -------ok------ok
* • Editions and Versions of Java

**JAVA VERSIONS:-**

***VERSION YEAR***

Java Alpha & beta : 1995

JDK 1.0 : 1996

JDK1.1 : 1997

J2SE 1.2 : 1998

J2SE 1.3 : 2000

J2SE 1.4 : 2002

J2SE 1.5 : 2004

JAVA SE 6 : 2006

JAVA SE 7 : 2011

* + 1. JAVA SE 8 : 2014
* • Java History

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

* • The JDK
  1. Java Runtime Environment
* • Your First Java Program (Note Pad)
* ---------------------------------------------Day-1 End------------------------------------------------------------

DAY-2:

* • Revisiting the JVM
  + 1. JVM Architecture:
* • API documentation

Oracle API docs

**API documentation** is a technical content deliverable, containing instructions about how to effectively use and integrate with an **API**

* • Your First Java Program (IDE)

---------------------------------------------Day-2 End------------------------------------------------------------

DAY-3:

Java Syntax

Class, Method, Variable

* Coding standards
* Valid Identifiers
* Scope

Object Oriented Programming in Java

* Creating (Instantiating) Objects
* Stack and Heap
* Using Objects
* Initializing Data
* Creating Object Types
* Building Constructors
* Default Constructor
* Working Example

---------------------DAY 3 END ------------------------------

Conventions & Primitives

* • Comments
* • Primitive Data Types
* • Primitive Literals
* • Primitive Conversion , type casting
* • Objects vs. Primitives
* Assignments and Operators
  + 1. Incre/Decre, Arithmetic, String Concat(+), Relational, Equality, Instance of, bitwise operator, Boolean Complement, Short circuite, assignment operator(Compound assignment), Conditional operator, new vs newInstance()
* ---------------------DAY 4 END ------------------------------
* • Enums
* • Conditionals
* • Loops
* • Break, Continue, and Labels

---------------------DAY 5 END ------------------------------

Strings

* String Objects
* String Constant Pool vs Heap Area
* String vs StringBuffer
* Immutable Strings
* Immutable and Mutable
* String Operations and Operators
* equals() and toString()
* == vs .equals
* compareTo()
* length()
* ***charAt(int) & split() & trim()***
* ***StringBuffer vs StringBuilder***
* ***StringTokenozer***

---------------------------------------------Day-6 End------------------------------------------------------------

Arrays

* • Arrays
* • Multi-Dimensional Arrays
* • For-Each
* • Sorting and Searching Arrays
* • Common Array Mistakes

---------------------------------------------Day-7 End------------------------------------------------------------

Packaging

* • Packages
* • Classpath
* • Java Archive (JAR)

Static

* Class vs. Object
* Static Variables
* Static Method
* Static Initialization Block
* instance Initialization Block
* “this” Keyword
* Garbage Collection (GC)
* Java Memory Leaks
* • Varargs

Primitive variables:

Primitive variables can be used to represent primitive values.

Example: int x=10;

Reference variables:

Reference variables can be used to refer objects.

Example: Student s=new Student();

**Based on the behavior and position of declaration all variables are divided**

**into the following 3 types.**

**1. Instance variables**

**2. Static variables**

**3. Local variables**

Instance variables:

* If the value of a variable is varied from object to object such type of variables are
* called instance variables.
* For every object a separate copy of instance variables will be created.
* Instance variables will be created at the time of object creation and destroyed at

the time of object destruction hence the scope of instance variables is exactly

same as scope of objects.

* Instance variables will be stored on the heap as the part of object.
* Instance variables should be declared with in the class directly but outside of any

method or block or constructor.

* Instance variables can be accessed directly from Instance area. But cannot be

accessed directly from static area.

But by using object reference we can access instance variables from static area

* Keywords:

***source-file:***

*class*

*extends*

*interface*

*implements*

*package*

*import*

**Data Types**

*byte*

*short*

*int*

*long*

*float*

*double*

*char*

*boolean*

* ***method-level:-***
* void
* return

***Flow-Control:-***

*if*

*else*

*switch*

*case*

*default*

*break*

*for*

*while*

*do*

* *continue*
* ***Object-level:-***
* *new*
* *this*
* *super*
* *instanceof*

---------------------------------------------Day-8 End------------------------------------------------------------

# OOPS: Object Oriented Programming Structure

Encapsulation & Data Hiding

* • Encapsulation & Data Hiding
* • Accessors/Mutators or Getters/Setters
* • Access Modifiers
* • Encapsulation/Data Hiding Resources

---------------------------------------------Day-9 End------------------------------------------------------------

Min Max elements of an Array

Linear Search

Binary Search

Fibonacci series

Swapping of Integers and Strings with and without third variable

---------------------------------------------Day-10 End------------------------------------------------------------

Inheritance

* Chaining Constructors
  1. IS-A and HAS-A -🡪 IS-A ---Using Extends; Has-A🡪 no specific keyword; -- Composition, Aggregation
* Overriding Methods and Revisiting the Super Reference
* The Three Faces of Final
  + - 1. – Final Keyword

– class level –which stops Inherit the class

--- Method level – which stops Overriding

--- instance level -Final variables cannot be overriden

---------------------------------------------Day-11 End------------------------------------------------------------

Abstract Classes and Interfaces

* Abstract Classes
* Interfaces
* Inheritance in Interfaces
* Revisiting Overriding Methods: Covariant Returns
* Marker Interface
* Adaptor Classes

---------------------------------------------Day-12 End------------------------------------------------------------

Pillars of OOPs: Encapsulation , Inheritance, and Polymorphism

Inheritance – Reusability

Encapsulation – Security

Polymorphism - Flexibility

Polymorphism

* • Polymorphism - Many forms – one name – many forms;
  + 1. Ex: Friendship: to start/to end based on the behavior
    2. Types: Compile time == Overloading, Method Hiding – Compile time / Static / early binding

Runtime == Overriding – Runtime / Dynamic / late binding

Overloading 🡪 Two methods with same name

* • Benefits of Polymorphism
* • Object Type Casting
* • The Object Class
* Overloading Methods

---------------------------------------------Day-13 End------------------------------------------------------------

Exception Handling

* • Throwable
* • Catching Exceptions
* • Finally block
* • The Exception Object
* • Runtime vs. Checked Exceptions
* • Creating Exceptions

---------------------------------------------Day-14 End------------------------------------------------------------

**1. Introduction**

**2. Runtime stack mechanism**

**3. Default exception handling in java**

**4. Exception hierarchy**

**Object🡪 Throwable🡪Exception/Error**

**5. Customized exception handling by try catch**

**try-catch-finally-throw-throws**

**6. Control flow in try catch**

**7. Methods to print exception information**

**8. Try with multiple catch blocks**

**9. Finally**

---------------------------------------------Day-15 End------------------------------------------------------------

**10. Difference between final, finally, finalize**

**11. Control flow in try catch finally**

**12. Control flow in nested try catch finally**

**13. Various possible combinations of try catch finally**

**14. throw keyword**

**15. throws keyword**

**16. Exception handling keywords summary 🡪 try – catch – finally -throw - throws**

**17. Various possible compile time errors in exception handling**

**18. Customized exceptions**

**19. Top-10 exceptions**

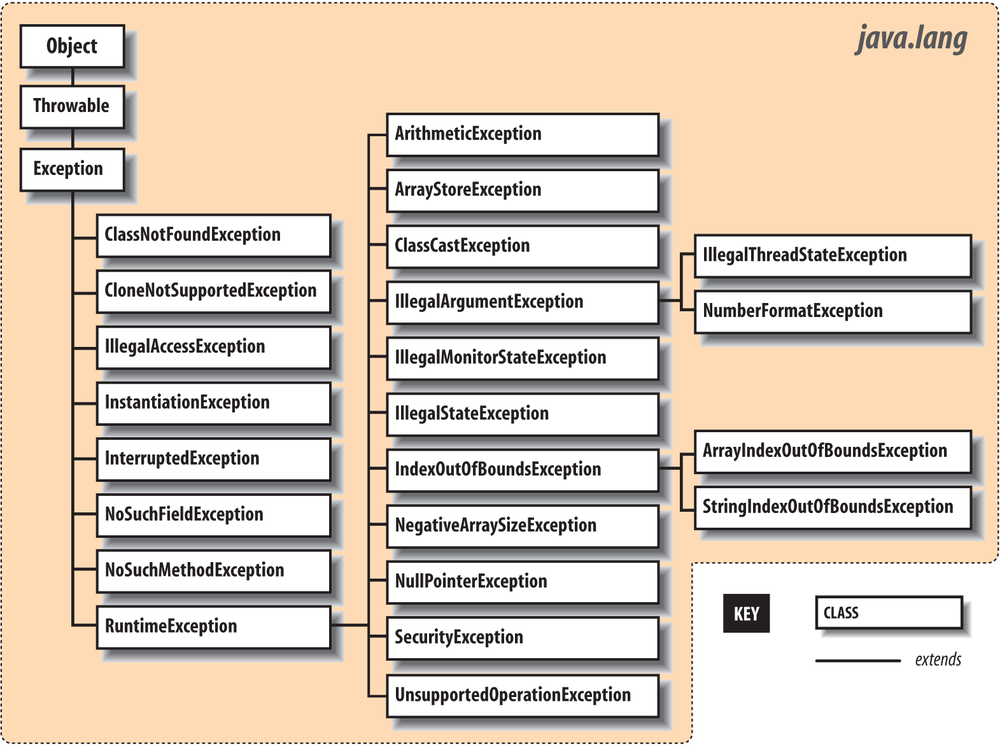
**20. 1.7 Version Enhancements – Try with Resources and Multi catch block**

**1. try with resources**

**2. multi catch block**

**21. Exception Propagation**

**22. Rethrowing an Exception**



---------------------------------------------Day-16 End------------------------------------------------------------

---------------------------------------------Day-17 End------------------------------------------------------------

Collections

Array – Homogeneous – Fixed in size

Collection – Heterogeneous – Dynamic Size

* • When arrays are not enough
* • Collections
* • The Collection Interface
* • Iterator
* • Lists
* • Sets
* • Queue
* • ArrayBlockingQueue
* • PriorityQueue
* • Deque
* • ArrayDeque
* • Maps
* • HashMap



**The key interfaces of collection framework:-**

1. Java.util.Collection

2. Java.util.List

3. Java.util.Set

4. Java.util.SortedSet

5. Java.util.NavigablaSet

6. Java.util.Queue

7. Java.util.Map

8. Java.util.SotedMap

9. Java.util.NavigableMap

10. Map.Entry

11. Java.util.Enumeration

12. Java.util.Iterator

13. Java.util.ListIterator

14. Java.lang.Comparable

15. Java.util.Comparator

**Collection – interface**

**Collections – Class – Utility class for Collection Objects**

**Array----Data Type**

**Arrays🡪 Utility Class**

***Characteristics of Collection frame work classes:-***

1) The collection framework classes are introduced in different Versions.

2) Heterogeneous data allowed or not allowed. 🡪 Normal Definitions and Generic Type Definitions

All classes allowed heterogeneous data except two classes

i. TreeSet ii. TreeMap

3) Null insertion is possible or not possible.

4) Insertion order is preserved or not preserved.

Input --->e1 e2 e3 output --->e1 e2 e3 insertion order is preserved

Input --->e1 e2 e3 output --->e2 e1 e3 insertion order is not-preserved

5) Collection classes’ methods are synchronized or non-synchronized.

6) Duplicate objects are allowed or not allowed.

add(e1)

add(e1)

7) Collections classes underlying data structures.

8) Collections classes supported cursors. 🡪Iterators

---------------------------------------------Day-18 End------------------------------------------------------------

***ArrayList Characteristics:-***

1) ArrayList Introduced in 1.2 version.

2) ArrayList stores Heterogeneous objects(different types).

3) Inside ArrayList we can insert **Null** objects.

4) ArrayList preserved Insertion order it means whatever the order we inserted the data in the same

way output is printed.

***a.*** Input -e1 e2 e3 output -e1 e2 e3 ***insertion order is preserved***

b. Input --e1 e2 e3 output --e1 e3 e2 ***insertion order is not- preserved***

5) ArrayList methods are non-synchronized methods.

6) Duplicate objects are allowed.

7) The under laying data structure is growable array.

***8)*** By using cursor we are able to retrieve the data from ArrayList : ***Iterator , ListIterator***

***New capacity = old capacity/2 + old capacity 🡺 50% increase in current capacity***

***Different ways to initialize values to ArrayList:-***

**Case 1:initializing ArrayList by using asList()**

import java.util.\*;

class ArrayListDemo

{ public static void main(String[] args)

{ **ArrayList<String> al = new ArrayList<String>(**

Arrays.asList("Sadiya","Raju","Shyam"));

System.out.println(al);

}

}

**Case 2:- adding objects into ArrayList by using anonymous inner classes.**

import java.util.ArrayList;

class ArrayListDemo

{ public static void main(String[] args)

{ **ArrayList<String> al = new ArrayList<String>()**

{ {add("Raju");

add("Sadiya");

}

};//semicolan is mandatory

System.out.println(al);

}

}

***Case 3:- normal approach to initialize the data***

import java.util.ArrayList;

class ArrayListDemo

{ public static void main(String[] args)

{ **ArrayList<String> al = new ArrayList<String>();**

al.add("Raju");

al.add("Sadiya");

System.out.println(al);

}

}

***Case 4:-***

***ArrayList<Type> obj = new ArrayList<Type>(Collections.nCopies(count, object));***

import java.util.\*;

class ArrayListDemo

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

{ Emp e1 = new Emp(111,"Raju");

**ArrayList<Emp> al = new ArrayList<Emp>(Collections.nCopies(5,e1));**

for (Emp e:al)

{ System.out.println(e.ename+"---"+e.eid);

}

}

}

***Case 5:-adding Objects into ArrayList by using addAll() method of Collections class.***

import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<String> al = new ArrayList<String>();

String[] strArray={"Raju","Sadiya","Sai"};

Collections.addAll(al,strArray);

System.out.println(al);

}

}

***All collection classes are having 2-versions:-***

1) Normal version(no type safety ).

2) Generic version.(type safety )

***Note :-***

*in java it is recommended to use generic version of collections class to store specified type of data.*

***Syntax:-***

ArrayList**<type-name> al =** new ArrayList**<type-name>**()**;**

**Examples:-**

ArrayList<Integer> al = new ArrayList<Integer>(); **//store only Integer objects**

ArrayList<String> al = new ArrayList<String>(); **//store only String objects**

ArrayList<Student> al = new ArrayList<Student>(); **//store only Student objects**

ArrayList<product> al = new ArrayList<product>()**; //store only product objects**

***Normal version of ArrayList(no type safety)***

1) Normal version is able to hold any type of data(heterogeneous data) hence it is not a type safe..

ArrayList al = new ArrayList();

al.add(10);

al.add(‘a’);

al.add(10.5);

System.out.println(al);

2) Always check the type of the object by using **instanceof** operator.

**3)** In normal it is holding different types of data hence while retrieving data must perform **type casting.**

**4)** If we are using normal version while compilation compiler generate warning message like **unchecked or unsafe operations.**

**Example:- normal version of ArrayList holding different types of Objects.**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ **ArrayList al = new ArrayList();**

al.add(10);

al.add('a');

al.add(10.4);

al.add(true);

System.out.println(al);

}

}

***Generic version of ArrayList(type safety)***

1) Generic version is able to hold specified

type of data hence it is a type safe.

**ArrayList<tye-name> al = new ArrayList<type-name>( );**

**ArrayList<Integer> al = new ArrayList<Integer>();**

**al.add(10);**

**al.add(20);**

**al.add(“Raju”);//compilation error**

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

2) Type checking is not required because it contains only one type of data.

3) It is holding specific data hence at the time of retrieval type casting is not required.

4) If we are using generic version compiler won’t generate warning messages.

***Example :- generic version of ArrayList holding only Integer data.***

import java.util.\*;

class Test

{ public static void main(String[] args)

{**ArrayList<Integer> al = new ArrayList<Integer>();**

al.add(10);

al.add(20);

al.add(30);

al.add(40);

System.out.println(al);

}

}

***Example :- retrieving data from generic version of ArrayList***

import java.util.\*;

class Test

{ public static void main(String[] args)

{ **ArrayList<Emp> al = new ArrayList<Emp>();**

al.add(new Emp(111,"Raju"));

al.add(new Emp(222,"Sadiya"));

……..

***Creation of sub ArrayList & swapping data :-***

Create sub ArrayList by using **subList(int,int)** method of ArrayList.

***public java.util.List<E> subList(int, int);***

to swap the data from one index position to another index position then use **swap()**method of Collections class.

**public static void swap(java.util.List<?>, int, int);**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<String> a1 = new ArrayList<String>();

a1.add("Raju");

a1.add("Sadiya");

a1.add("Sai");

a1.add("Shyam");

ArrayList<String> a2 = new ArrayList<String>(a1.subList(1,3));

System.out.println(a2);

ArrayList<String> a3 = new ArrayList<String>(a1.subList(1,a1.size()));

System.out.println(a3);

**//java.lang.IndexOutOfBoundsException: toIndex = 7**

**//ArrayList<String> a4 = new ArrayList<String>(a1.subList(1,7));**

System.out.println("before swapping="+a1);

Collections.swap(a1,1,3);

System.out.println("after swapping="+a1);

}

}

ArrayList to Array

Array to ArrayList

*String[] a = new String[al.size()];*

*al.toArray(a);*

---------------------------------------------Day-19 End------------------------------------------------------------

**Sorting – Predefined Methods -🡪Custom Sorting**

***Comparable vs Comparator :-***

***Note :- it is possible to sort String and all wrapper objects because these objects are***

***implementing*** Comparable ***interface.***

---------------------------------------------Day-20 End------------------------------------------------------------

LinkedList:

* The Underlying Data Structure is Double LinkedList.
* Insertion Order is Preserved.
* Duplicate Objects are allowed.
* Heterogeneous Objects are allowed.
* null Insertion is Possible.
* Implements Serializable and Cloneable Interfaces but Not RandomAccessInterface.
* ArrayList and Vector Classes Implements RandomAccess Interface. So that we can Access any Random Element with the Same Speed.
* RandomAccess Interface Present in java.utilPackage and it doesn't contain any Methods.
* Hence it is a Marker Interface.
* Best Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.
* Worst Choice if Our Frequent Operation is Retrieval.

Constructors:

1) LinkedList l = new LinkedList(); Creates an Empty LinkedList Object.

2) LinkedList l = new LinkedList(Collection c);

Creates an Equivalent LinkedList Object for the given Collection.

Methods:

Usually we can Use LinkedList to Implement Stacks and Queues. To Provide Support for this

Requirement LinkedList Class Defines the following 6 Specific Methods.

1) void addFirst(Object o)

2) void addLast(Object o)

3) Object getFirst()

4) Object getLast()

5) Object removeFirst()

6) Object removeLast()

Vector: V1.0

* The Underlying Data Structure is Resizable Array OR Growable Array.
* Insertion Order is Preserved.
* Duplicate Objects are allowed.
* Heterogeneous Objects are allowed.
* null Insertion is Possible.
* Implements Serializable, Cloneable and Random Access interfaces.
* Every Method Present Inside Vector is Synchronized and Hence Vector Object is

Thread Safe.

* Vector is the Best Choice if Our Frequent Operation is Retrieval.
* Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.

With Default constructor, Creates an Empty Vector Object with Default Initial Capacity 10.

* Once Vector Reaches its Max Capacity then a New Vector Object will be Created with

New Capacity = Current Capacity \* 2

int capacity()

Stack:

* It is the Child Class of Vector.
* It is a Specially Designed Class for Last In First Out (LIFO) Order.

Constructor:Stack s = new Stack();

Methods:

1) Object push(Object o); To Insert an Object into the Stack.

2) Object pop(); To Remove and Return Top of the Stack.

3) Object peek(); Ro Return Top of the Stack without Removal.

4) boolean empty(); Returns true if Stack is Empty

5) int search(Object o);Returns Offset if the Element is Available Otherwise Returns -1.



we are able to retrieve objects from collection classes in three ways:-

1) By using for-each loop.

2) By using cursors.

3) By using get() method.

---------------------------------------------Day-20 End------------------------------------------------------------

Cursors:-

The 3 Cursors of Java:

* If we want to get Objects One by One from the Collection then we should go for Cursors.
* There are 3 Types of Cursors Available in Java.

1) Enumeration

2) Iterator

3) ListIterator

**1) Enumeration: - Legacy Cursor**

* We can Use Enumeration to get Objects One by One from the Collection.
* We can Create Enumeration Object by using elements().

public Enumeration elements();

Eg:Enumeration e = v.elements(); //v is Vector Object.

Methods:

1) public Boolean hasMoreElements();

2) public Object nextElement();

Enumeration e = v.elements();

Limitations of Enumeration:

* Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal

Cursor.

* By using Enumeration we can Perform Read Operation and we can't Perform Remove

Operation.

To Overcome Above Limitations we should go for Iterator.

Iterator:

* We can Use Iterator to get Objects One by One from Collection.
* We can Apply Iterator Concept for any Collection Object. Hence it is **Universal** Cursor.
* By using Iterator we can Able to Perform Both Read and Remove Operations.
* We can Create Iterator Object by using iterator() of Collection Interface.

public Iterator iterator();

Eg:Iterator itr = c.iterator(); //c Means any Collection Object.

Methods:

1) public Boolean hasNext()

2) public Object next()

3) public void remove()

Limitations:

* By using Enumeration and Iterator we can Move Only towards **Forward Direction** and we

can’t Move Backward Direction.

That is these are Single Direction Cursors but Not Bi-Direction.

* By using Iterator we can Perform Only Read and Remove Operations and we can't Perform

Addition of New Objects and Replacing Existing Objects.

To Overcome these Limitations we should go for ListIterator.

**ListIterator:**

* ListIterator is the Child Interface of Iterator.
* By using ListIterator we can Move Either to the Forward Direction OR to theBackward Direction. That is it is a Bi-Directional Cursor.
* By using ListIterator we can Able to Perform Addition of New Objects and Replacing existing Objects. In Addition to Read and Remove Operations.
* We can Create ListIterator Object by using listIterator().

Public ListIteratorlistIterator();

Eg:ListIterator ltr = l.listIterator(); //l is Any List Object

Methods:

* ListIteratoris the Child Interface of Iterator and Hence All Iterator Methods by Default
* Available to the ListIterator.

Note: The Most Powerful Cursor is ListIterator. But its Limitation is, it is Applicable Only for

List Objects.

**1) Enumeration –v.elements() – only Read – hasME, NE**

**2) Iterator – collectionobj.iterator() – Read and Remove – hN,n,remove**

**3) ListIterator – listobj.listIterator() – Read, remove, replace and delete – 9 methods**

---------------------------------------------Day-21 End------------------------------------------------------------

**1.5 Version Enhancements (Queue Interface):**

* Queue is a Child Interface of Collection.
* If we want to Represent a Group of Individual Objects Prior to processing then we should

go for Queue.

* From 1.5 Version onwards LinkedListalso implements Queue Interface.
* Usually Queue follows FIFO Order. But Based on Our Requirement we can Implement Our

Own Priorities Also (PriorityQueue)

* LinkedList based Implementation of Queue always follows FIFO Order.

Eg: Before sending a Mail we have to Store all Mail IDs in Some Data Structure and for the 1st

Inserted Mail ID Mail should be Sent 1st.For this Requirement Queue is the Best Choice.

Methods:

1) boolean offer(Object o); To Add an Object into the Queue.

2) Object peek();

*  To Return Head Element of the Queue.
*  If Queue is Empty then this Method Returns null.

3) Object element();

*  To Return Head Element of the Queue.
*  If Queue is Empty then this Methodraises RE: NoSuchElementException

4) Object poll();

*  To Remove and Return Head Element of the Queue.
*  If Queue is Empty then this Method Returns null.

5) Object remove();

*  To Remove and Return Head Element of the Queue.
*  If Queue is Empty then this Method raise RE: NoSuchElementException.

***Copying data from Vector to ArrayList:-***

*To copy data from one class to another class use* ***copy()*** *method of Collections class.*

---------------------------------------------Day-22 End------------------------------------------------------------

**Set (I):**

 It is the Child Interface of the Collection.

 If we want to Represent a Group of Individual Objects as a Single Entity where

Duplicates are Not allowed and Insertion Order won't be Preserved. Then we should go

for Set Interface.

**Load Factor:** Initial Capacity 16, Fill Ratio : 0.75

HashSet:

 The Underlying Data Structure is Hashtable.

 Insertion Order is Not Preserved and it isBased on hashCode of the Objects.

 Duplicate Objects are Not Allowed. If we are trying to Insert Duplicate Objects then we

won't get any Compile Time ORRuntime Error.add() Simply Returns false.

 null Insertion is Possible.

 Heterogeneous objects are allowed.

 HashSet implements ***Serializable***and ***Cloneable***Interfaces but Not *RandomAccess.*--> RA—Retrieval increase

 If Our Frequent Operation is Search Operation, then HashSet is the Best Choice.

Default Initial Capacity 16 and Default Fill Ratio :

0.75.

***LinkedHashSet:-***

1. Introduced in 1.4 version and It is a child class of HashSet.

2. Duplicate objects are not allowed if we are trying to insert duplicate values then we won’t get

any compilation errors an won’t get any Execution errors simply add method return false.

3. Null insertion is possible.

4. Heterogeneous objects are allowed

5. The under laying data structure is LinkedList & hashTable.

6. Insertion order is preserved.

7. Methods are non-synchronized.

8. Cursors :- Iterator. 🡪Universal Cursor

SortedSet:

 It is the Child Interface of Set.

 If we want to Represent a Group of Individual Objects without Duplicates and all

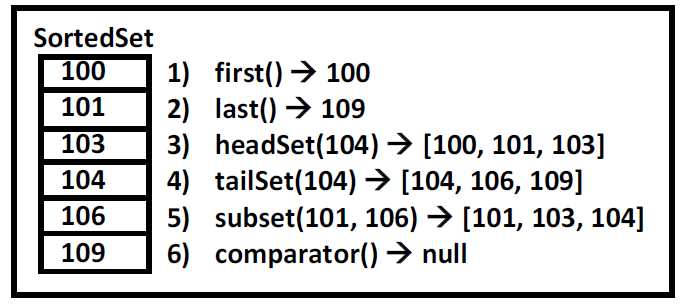
Objects will be Inserted According to Some Sorting Order, then we should go for

SortedSet.

 The Sorting can be Either Default Natural Sorting OR Customized Sorting Order.

 For String Objects Default Natural Sorting is Alphabetical Order.

 For Numbers Default Natural Sorting is Ascending Order.



**TreeSet:-**

1. TreeSet is same as HashSet but TreeSet sorts the elements is ascending order but HashSet does

not maintain any order.

2. The underlying data Structure is BalencedTree.

3. Insertion order is not preserved it is based some sorting order.

4. Heterogeneous data is not allowed.

5. Duplicate objects are not allowed

6. Null insertion is not possible.

*TreeSet<String> t = new TreeSet<String>(new MyComp());*

*Example :-Elimination duplicate objects*

*import java.util.\*;*

*class Test*

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

*{ String[] str={"Sadiya","Raju","Shyam","Sai"};*

*//converion of String[] to List*

*List<String> l = Arrays.asList(str);*

*//conversion of List to Set [it eliminates duplicates]*

*TreeSet<String> t = new TreeSet<String>(l);*

*System.out.println(t);*

*}*

*}*

---------------------------------------------Day-23 End------------------------------------------------------------

**LIST-> Vector🡪Stack**

**SET-> HashSet, LinkedHashSet, TreeSet**

**MAP-> HashMap(1.2), LinkedHashMap(1.5), TreeMap(1.2), HashTable(Legacy class V1.0)**

***Map:-***

1. Map is not a child interface of collection.

2. Up to now we are working with single object and single value where as in the map collections

we are working with two objects and two elements.

3. The main purpose of the collection is to compare the key value pairs and to perform necessary

operation.

4. The key and value pairs we can call it as map Entry.

5. Both keys and values are objects only.

6. In entire collection keys can’t be duplicated but values can be duplicate.

**HashMap:-**

*1) interoduced in 1.2 version*

*2) Heterogeneous data allowed.*

*3) Underlying data Structure is HashTable.*

*4) Duplicate keys are not allowed but values can be duplicated.*

*5) Insertion order is not preserved.*

*6) Null is allowed for key(only once)and allows for values any number of times.*

*7) Every method is non-synchronized so multiple Threads are operate at a time hence permanence*

*is high.*

*8) cursor :- Iterator.*

*Methods: entryset, values, keyset*

***LinkedHashMap:-***

*1) interdicted in 1.4 version*

*2) Heterogeneous data allowed.*

*3) Underlying data Structure is HashTable & linkedlist.*

*4) Duplicate keys are not allowed but values can be duplicated.*

*5) Insertion order is preserved.*

*6) Null is allowed for key(only once)and allows for values any number of times.*

*7) Every method is non-synchronized so multiple Threads are operate at a time hence permanence*

*is high.*

*8) cursor :- Iterator*

**HashTable:-**

1. Introduced in the 1.0 version it’s a legacy class.

2. Every method is synchronized hence only one thread is allowed to access it is a Thread safe but

performance is decreased.

3. Null insertion is not possible if we are trying to insert null values we are getting

NullPointerException.

---------------------------------------------Day-24 End------------------------------------------------------------

Various Possible Implementations of compare():

publicint compare(Object obj1, Object obj2) {

Integer I1 = (Integer)obj1;

Integer I2 = (Integer)obj2;

return I1.compareTo(I2); //[0, 5, 10, 15, 20] Ascending Order

return -I1.compareTo(I2); //[20, 15, 10, 5.0] Descending Order

return I2.compareTo(I1); //[20, 15, 10, 5.0]

return -I2.compareTo(I1); //[0, 5, 10, 15, 20]

return +1; //[10, 0, 15, 5, 20, 20] Insertion Order

return -1; //[20, 20, 5, 15, 0, 10] Reverse of Insertion Order

return 0; //[10] Only 1st Inserted Element Present And All Remaining Elements Treated as Duplicates

}

When we go for Comparable and When we go for Comparator:

Comparable Vs Comparator:

 For Predefined Comparable Classes (Like String) Default Natural Sorting Order is Already

Available. If we are Not satisfied with that we can Define Our Own Sorting by Comparator

Object.

 For Predefine Non- Comparable Classes (Like StringBuffer) Default Natural Sorting Order

is Not Already Available. If we want to Define Our Own Sorting we can Use Comparator

Object.

 For Our Own Classes (Like Employee) the Person who is writing Employee Class he is

Responsible to Define Default Natural Sorting Order by implementing Comparable

Interface.

 The Person who is using Our Own Class if he is Not satisfied with Default Natural Sorting

Order he can Define his Own Sorting by using Comparator Object.

 If he is satisfied with Default Natural Sorting Order then he can Use Directly Our Class.

---------------------------------------------Day-24 End------------------------------------------------------------

---------------------------------------------Day-25 End------------------------------------------------------------

Hashtable internal implementation:

SET,Map 🡪 Hashtable

Equals and Hashcode contract

Working with Collections and Arrays

* • Limitations of Collections
* • Generics and Autoboxing
* • Diamond Operator
* • Sorting and Searching Arrays and Collections
* • Comparable
* • Comparator

---------------------------------------------Day-26 End------------------------------------------------------------

1.0

1.2 🡪 Generics, Enums, Collection frmwrk

1.5🡪 Collection frmwrk enhancements—LinkedList, Linked Hash Set, Treeset

1.7 🡪 Java7

1.8 🡪Java8

Java7->

1. multiple catch stmts🡪 catch(RTE | Exception)
2. Try with resoources 🡪 try{} -🡪 try(BR br = new BR()){ }🡪 avoiding finally block usage🡪 closing files, db connection 🡪 Autoclosable
3. Switch Case with String 🡪 🡪 Switch (ch/int)🡪case 1: case ‘a’:🡪 switch(String): case “SunDay”:

# Java8

Features

1. Lambda expressions 🡪 writing the logic in simpler manner
2. Interfaces 🡪 until java7- interfaces shud nt allow concrete methods—having method definitions. But we can have concrete methods from Java8 onwards 🡪 static and default

Shape->draw(); Circle, Square 🡪 draw(){…}

1. Streams🡪 Enhancements of Collections

Lambda Expressions and Static Imports

* • Marker and Functional Interfaces 🡪 Marker->No methods , Functional 🡪Only One abstract method
* • Lambda Expressions
* • Predicate
* • Static Imports

Lambda Expression:

1. Lambda expressions are nothing but method implementations.
2. Lambda expressions only applied on Functional Interfaces

**Why to go for Lambdas:** Code reducibility**.** Code is less🡪 easy to understand, readability.

{x -> x\*x}

**When to write**: Legacy: Define Interface, Create Impl class and override the unimplemented methods

**class** TestClass **implements** TestInterface{

@Override

**public** **void** method() {

System.***out***.println("Method Implementation");

}

}

**interface** TestInterface{

**void** method();

}

TestClass tc = **new** TestClass();

tc.method();

System.***out***.println("-----Anonymous -No name----------");

**new** TestInterface() {

@Override

**public** **void** method() {

System.***out***.println("Anonymous impltn");

}

}

.method(); -🡪 this can be shorten. To overcome this Lambdas came into the picture

**How to write Lambdas:**

U can replace legacy anonymous impltn with Lambdas.

**new** TestInterface() {

@Override

**public** **void** method() {

System.***out***.println("Anonymous impltn method 1");

}

@Override

**public** **void** method1() {

System.***out***.println("Anonymous impltn method 2");

}

}

.method(); // one method shud be called to get the impltn

In normal interfaces we might have multiple methods on that interfaces we cannot write Lambdas.

Thts why Lambdas applied only Functional Interfaces.

Appendix A – JDBC

• Why JDBC?

* • Driver Manager
* • Connections
* • Statements
* • Inserting Rows
* • Updating Rows
* • Deleting Rows
* • Other Modifying Statements
* • Result Set
* • Mapping Between SQL & Java Data Types
* • PreparedStatement
* • CallableStatement
* • SQLException
* • Metadata
* • Datasource

Appendix B - Java GUI

* • Java Foundation Classes (JFC)
* • Top Level Containers
* • Components
* • Layout Management
* • BorderLayout
* • FlowLayout
* • BoxLayout
* • Event Handling
* • Adapters