Java Collections

java.util.Arrays; ©

Arrays :- `String[] myStringArr={"A","B"};`

`Chr[] myChrArr={'A','B'};`

`int[] myIntArr = new int[]{1,2,3};`

`float[] myFloat = new float[]{459.432f, 348.43f};`

Syntax

`Data-Type[] variableName;`

and

`Data-Type variableName[];`

How many Ways

forEach loop →

```
for (int myInt : myIntArr){
    s.o.p(myInt);
}
```

```
for (int i=0; i<myInt.length; i++){
    myInt[i];
}
```

Elements starts from 0 to n;

→ Copying of Arrays from one Arry to another Arry

① `int[] a={1,2,3};`
`int[] b = a;`

② using for loop `for(int i=0; i<a.length; a++){ b[i]=a[i]; }`

③ `copyOf(objArr[] a, int length);`
`int[] b = Arrays.copyOf(a, a.length);`

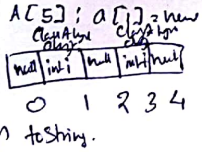
* Using clone:
`int b = a.clone();`

* `arrayCopy()` method of System class;
`System.arrayCopy(a,0,b,0,a.length);`

* Arrays containing Reference to Derived Type of Data:

```
ClassA {
    int i;
}
```

`A[] a; a=new A[5]; a[1]=new A(); a[3]=new A();`



* How to make arrays toString.
`Arrays.toString(a); output [1,2];`

* How to Sort Arrays. by using `Arrays.sort(a);`

* binary Search → Algorithm finds the position of a specified value ("the input key") within a sorted array.
or
half-interval Search
`int[] a={1,2,3,4}` `int Arrays.binarySearch(array, key);`
primitives 0 1 2 3 — return index of the key 3 → 3

* Static `int[] copyOfRange(array, int from, int to);` ②
index index.

* Static void `fill(array, primitive val)`
Assigning the specified primitive val to each element of the array.
`int[] a={1,2,3};` `Arrays.fill(a,5);` → `[5,5,5];`
→ from Index, to Index condition also.

* To Check the Equality of Two Arrays in Java
→ Iterative Method, `Arrays.equal()`, `Arrays.deepEquals()`;

→ `Arrays.deepToString()` method → this is used to print Content of the Array.
Eg:- `String[] twoDim = new Str[]{"one","two"}; Arrays.deepToString(twoDim);`

→ `Arrays.asList(arr)` → this asList (array) return the fixed length of list with specified array type. Eg: `String[] strArr={"one","two"}, List<Str> list = Arrays.asList(strArr);` Size

→ static int hashCode (primitive type Arr) → HashCode based on the Contents of the array.

→ Binary Search – Divided – Conquer – Search.
Half-interval Search Eg: `int[] arr={10,11,20,5,16};`
* step① → before search need to be in sort mode → `Arrays.sort(arr);` `int[] arr={10,11,16,20};`
* `nbs.binarySearch(arr,16);`

logic
```
int, start=0; int end = inputArr.length-1;
while (start<=end){
    int mid = (start+end)/2;
    if(Key == input[mid]){
        return mid;
    }
    if (key<inputArr[mid]){
        end = mid-1; } else { start = mid+1;
```

We can also sort strings in alphabetical order.

```
// A sample Java program to sort an array of strings
// in ascending and descending orders using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
    public static void main(String[] args)
    {
        String arr[] = {"practice.geeksforgeeks.org",
                        "quiz.geeksforgeeks.org",
                        "code.geeksforgeeks.org"
                       };

        // Sorts arr[] in ascending order
        Arrays.sort(arr);
        System.out.printf("Modified arr[] : \n%s\n\n",
                          Arrays.toString(arr));

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : \n%s\n\n",
                          Arrays.toString(arr));
    }
}
```

Run on IDE

Output:

```
Modified arr[] :
[code.geeksforgeeks.org, practice.geeksforgeeks.org, quiz.geeksforgeeks.org]
```

java.lang.String ©

Ternary Operator → booleanExpression ? expression1 : expression2

```
int y = 10;
int x = (y>5) ? (2*y) : (3*y);
```

Immutable class in java

1) Declare the class is final so it can't be extended.

   public final class String.

2) Make all fields private so that direct access is not allowed.

   private final char value[];
   private int hash;

3) Make all mutable fields final so that it's value can be assigned only once.

   private final Char Value[];

4) Initialize all the fields via a Constructor performing deep Copy.

```
public String(){
    this.value = new Char[0]; }
```

5) There is no Setter methods i.e we have no option to change the value of the instance variable.

6) Perform cloning of objects in the getter methods to return a copy rather than returning the actual Object reference.

METHOD SUMMARY

char	charAt (int index)
int	CodePoint (int index) → return unicode at the specified index.
int	codePointBefore (int index)
int	CodePointCount (int beginIndex, int endIndex)
int	CompareTo (String anotherString) → Compares two strings lexicographically. [+, −, 0 / +1 −1]
int	CompareToIgnoreCase (String str)
String	Concat (String str);
boolean	contains (CharSequence s)
boolean	ContentEquals (CharSequence cs)
static String	CopyValueOf (Char[] data);
static String	CopyValueOf (Char[] chr, int offSet, int Count) (offSet – Start point, Count – end Point)
byte[]	getBytes () Encodes this String into a Sequence of bytes using the given @ default Charset storing the result into a new byte array.
int	hashCode () ;
int	indexOf (int ch) → return the index of a Substring.
String	intern → a string that has the same Content as this String. but is guaranteed to be from a Pool of unique strings.
boolean	isEmpty ();
int	length ();
boolean	matches (String regex);
boolean	regionMatches (int toffset , String other, int ooffset, int len);
String[]	split (String regexp);
String []	split (String regex, int limit)
String	subString (int beginIndex) / (int beginIndex, int endIndex)
Char []	toCharArray() → return CharArray of the String.

Eg:- String s1 = "this is index of example");
(this – 0, is – 2, 3, index – 4)

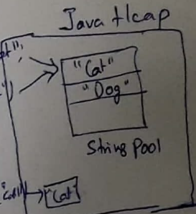
int index1 = s1.indexOf("is"); 2

ignoreCase,
toffset – the starting offset of the other,
ooffset – the starting offset of the len, Subregion in the String,

String Constant Pool

String s1 = "Cat";
String s2 = "Cat";

String s3 = new String("Cat")



s1 == s2 // true.
s1 == s3 // false.
s1.equals(s3) // true.

Difference Between Shallow Copy Vs Deep Copy in java

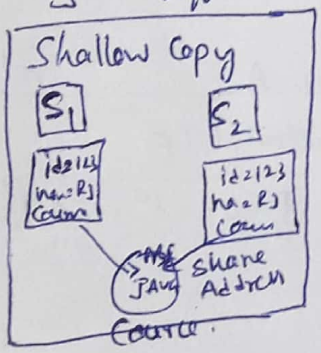
→ cloning is a process of creating an exact copy of an existing object in the memory. java.lang.Object → clone();

→ Note:- Not all the objects in java are eligible for cloning process

? → the objects which implements Cloneable (java.lang.I) interface. (marked interface)

Shallow Copy → the default version of clone() method creates the shallow copy of an object. It is a bit-wise copy of an object.

→ A new object is created that has an exact copy of the values in the original obj. If any of the fields of the object are references to other objects, just reffer addresses are copied. i.e only the memory Address is copied.




```
class Course {
    String cName;
    Course (String cName) {
        this.cName = cName;
    }
}

class D
psvm (String[] args) {
    Student s2 = null;
    Course c1 = new Course();
    Student s1 = new Student();
    s2 = (Student) s1.clone();
}
```

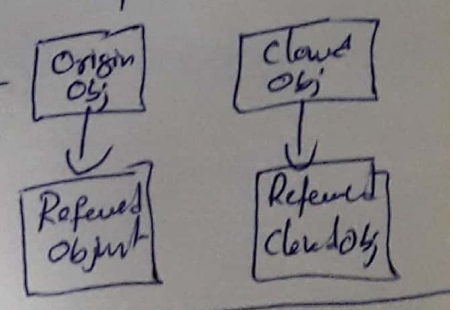
```
class Student implements Cloneable {
    String stName;
    Course c;
    protected Student (String stName, Course c) {
        this.stName = stName;
        this.c = c;
    }
    protected Object clone() {
        return super.clone();
    }
}
```

Deep Copy:-

A Deep Copy copies all fields and makes copies of Dynamically allocated memory pointed by the fields. A deep copy occurs when an Object is copied along with the object to which it refers.

Diff shallow Copy & Deep Copy in java

SC	DC
→ clone obj, orig obj not 100% Disjoint	→ cloned, origin obj 100% Disjoint
→ changes in clone obj will alter the orig obj	→ No reflection b/w clone & orig obj
→ Default version of clone method creates the shallow copy of an obj	→ To create DC of an obj you have to override clone method.
→ it is used if obj has primitive fields	→ it is used if an obj has reference to other objects as field.
→ SC is faster and less expensive	→ DC is slow and very expensive



```
class Student implements Cloneable {
  //Contained object
  private Subject subj;

  private String name;

  public Subject getSubj() {
        return subj;
  }

  public String getName() {
        return name;
  }

  public void setName(String s) {
        name = s;
  }

  public Student(String s, String sub) {
        name = s;
        subj = new Subject(sub);
  }

  public Object clone() {
        //shallow copy
        try {
          return super.clone();
        } catch (CloneNotSupportedException e) {
          return null;
        }
  }
}
```

```
public class CopyTest {

  public static void main(String[] args) {
        //Original Object
        Student stud = new Student("John", "Algebra");

        System.out.println("Original Object: " + stud.getName() + " - "
              + stud.getSubj().getName());

        //Clone Object
        Student clonedStud = (Student) stud.clone();

        System.out.println("Cloned Object: " + clonedStud.getName() + " - "
              + clonedStud.getSubj().getName());

        stud.setName("Dan");
        stud.getSubj().setName("Physics");

        System.out.println("Original Object after it is updated: "
              + stud.getName() + " - " + stud.getSubj().getName());

        System.out.println("Cloned Object after updating original object: "
              + clonedStud.getName() + " - " + clonedStud.getSubj().getName());

  }
}
```

```
Output is:
Original Object: John - Algebra
Cloned Object: John - Algebra
Original Object after it is updated: Dan - Physics
Cloned Object after updating original object: John - Physics
```

```
class Student implements Cloneable {
  //Contained object
  private Subject subj;

  private String name;

  public Subject getSubj() {
        return subj;
  }

  public String getName() {
        return name;
  }

  public void setName(String s) {
        name = s;
  }

  public Student(String s, String sub) {
        name = s;
        subj = new Subject(sub);
  }

  public Object clone() {
        //Deep copy
        Student s = new Student(name, subj.getName());
        return s;
  }
}

Output is:
Original Object: John - Algebra
Cloned Object: John - Algebra
Original Object after it is updated: Dan - Physics
Cloned Object after updating original object: John - Algebra
```

StringBuilder vs StringBuffer

	Thread Safe	Immutability	Class / Interface	How to Concatinate	How to Create	Default Capacity, Length
java.lang. StringBuffer (C)	✓	× (it is mutable)	Class in java 1.0	append ( ) method	StringBuffer sb = new StringBuffer();	16
java.lang. StringBuilder (C)	×	× (it is mutable)	Class in java 5.0	StringBuilder append()	StringBuilder sb = new StringBuilder();	16

public void trimToSize() → Capacity becomes its length of the string inside buffer

String SubStr (int start)
(int start, int end)

StringBuffer/StringBuilder append (primitiveType name)
char[] str
char[] str, int offset, int len

appendCodePoint (int codePoint) → a unicode code point

int
SB/SB capacity() → gives the allowable number of characters this S.Bobject can still accommodate.

SB/SB ensureCapacity (int minimumCapacity) → Ensures that its capacity is at least equal to its specified minimum.

NOTE if minimum is argument is 20 → (minimum capacity) < (argument capacity) → allocated greater capacity
→ its capacity becomes twice its current capacity plus 2
→ its capacity is now (20*2)+2.

SB/SB delete (int start, int end) →
SB/SB deleteCharAt (int index)
SB/SB reverse()
SB/SB indexOf (String str) and indexOf (String str, int fromIndex)
int length()
void setCharAt (int index, char ch)
void setLength (int newLength)
CharSequence subSequence (int start, int end)

Collection FrameWork :- Group of objects stored in well defined manner.

Din-Arrays → Earlier Arrays are used for these group of Objects.
→ Arrays are not re-sizable. Size of the arrays are fixed.
→ Size Can't be Changed once they are defined.

java.util → Contains the collections FW, legacy collection classes event model, date and time facilities, i18n, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)

Collection → java.util
Interface Collection<E> extends Iterable<T> (java.lang) (I)

→ The root Interface in the Collection hierarchy.
→ Do not instantiate a Collection directly
→ but rather a subtype of Collection.
→ You may often treat these Subtypes uniformly as a collection.

Collection Subtypes :- following interfaces (collection types) extends the Collection Interface. (extends)

Collection ← List, Set, SortedSet, NavigableSet, Queue, Deque

Ex: List extends Collection.

Ex:-
```
class MyColl{
  static void doSome(Collection coll){
    for(int i: coll){
      s.o.p(i);
  }
}
```

Adding and Removing Collection
```
Set set = new HashSet();
List list = new ArrayList();
MyColl.doSome(set);
MyColl.doSome(list);
```

Methods
```
boolean     add(E e)
boolean     addAll(Collection<? extends E> c)
void        clear()
int         size();
boolean     isEmpty()
boolean     contains(Object o)
Iterator<E> iterator(); (method of Iterable)
Object[]    toArray();
T[]         toArray(T[] a)

boolean     remove(Object o);
boolean     containsAll(Collection<?> c);
boolean     removeAll(Collection<? extends E> c);
boolean     retainAll(Collection<?> c);
boolean     equal(Object o);
int         hashCode();
```

```
Collections.shuffle(list);
Collections.fill(list, 0.0);
Collection.min(list)
          max(list)
          reverse(list)
Collection.copy(destList, sourceList)

static<T> Collections.synchronizedCollection(Collection)
                                 Map
                                 List
                                 Set
threadSafe
```

java.lang
Interface Iterable<T>

Methods
```
Iterator<T> iterator()                               } since 1.5
default void forEach(Consumer<? super T> action)     }
default Spliterator<T> spliterator();                } 1.8
```

java.util
Interface Spliterator<T>

```
int     characteristics()
long    estimateSize()
default void forEachRemaining(Consumer<? super T>)
default Comparator<? super T> getComparator()
default long getExactSizeIfKnown()
default boolean hasCharacteristics(int characteristics)
boolean tryAdvance(Consumer<? super T> action)
Spliterator<T> trySplit();
```

Ex
```
Cnu cnu1 = new Cnu("MyName");
List<Cnu> list = Collections.singletonCollection(obj1);
list.add(cnu2);
```
Exception
UnsupportedOperationException.

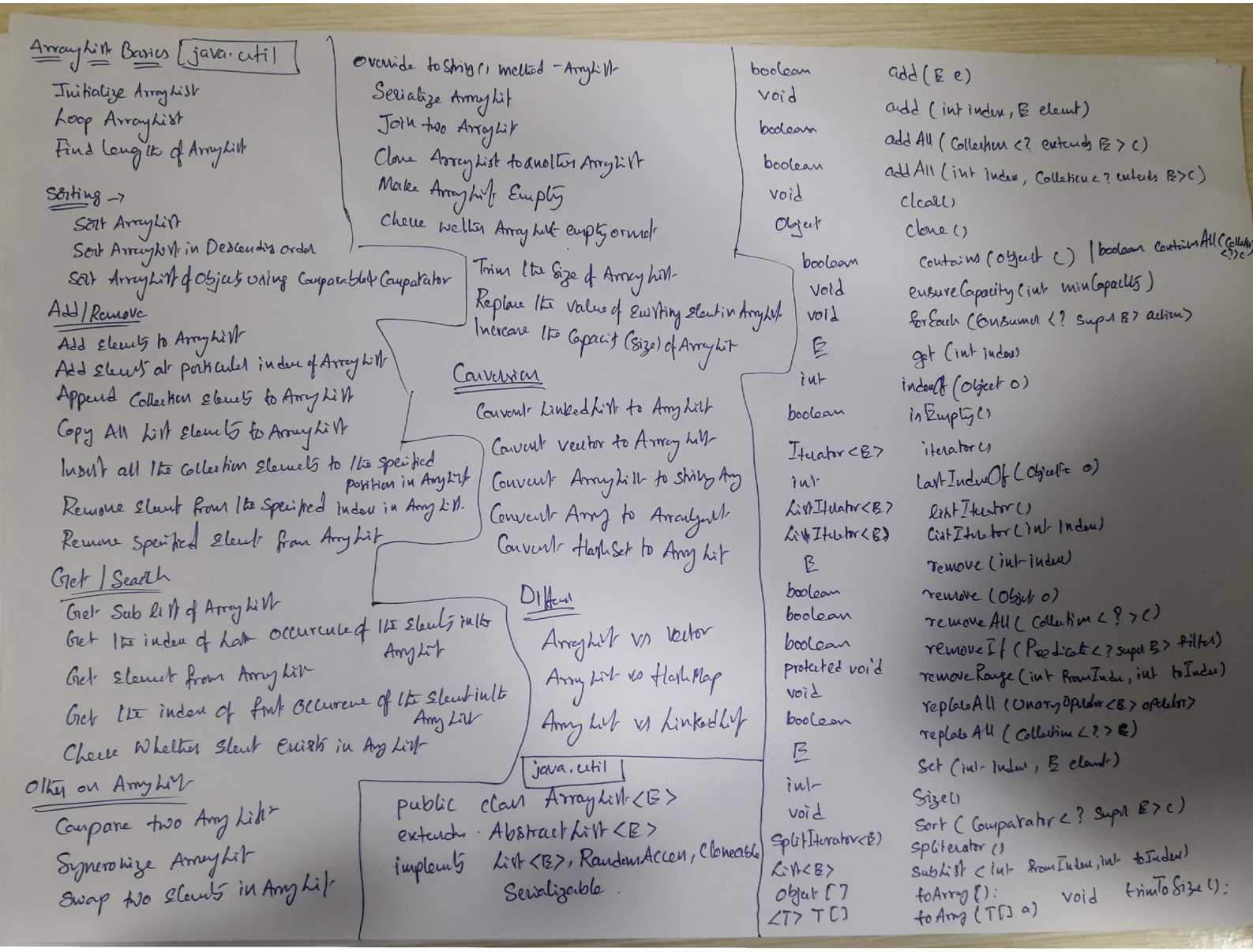
java.util.
Class Collections extends Object. (Since 1.2)

→ This class Consists exclusively of static methods that operate on return Collections. "wrappers" polymorphic algorithms that operate on Collections,
→ It Contains polymorphic algorithms that operate on Collections, "wrappers" which return a new Collection backed by a specified Collection.
→ If Collections class object is null it will throw NullpointerException.

```
static<T> boolean addAll(Collection<? super T> c, T... elements)
static<T> Queue<T> asLifoQueue(Deque<T> deque)
static<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
static<E> Collection<E> checkedCollection(myList, String.class)   (wrapper)
                                   List
                                   Set
                                   Map
static<T> void copy(sourceList, destList)
static boolean disjoint(Collection<?> c1, Collection<?> c2)
static<T> Enumeration<T> emptyEnumeration(), emptyIterator(),
static<T> List   unmodifiableMap(map);   Collection List Map Set  Throws UnSupportedOperation Exception
          Map
void      swap(list, int i, int j)   List Set
```

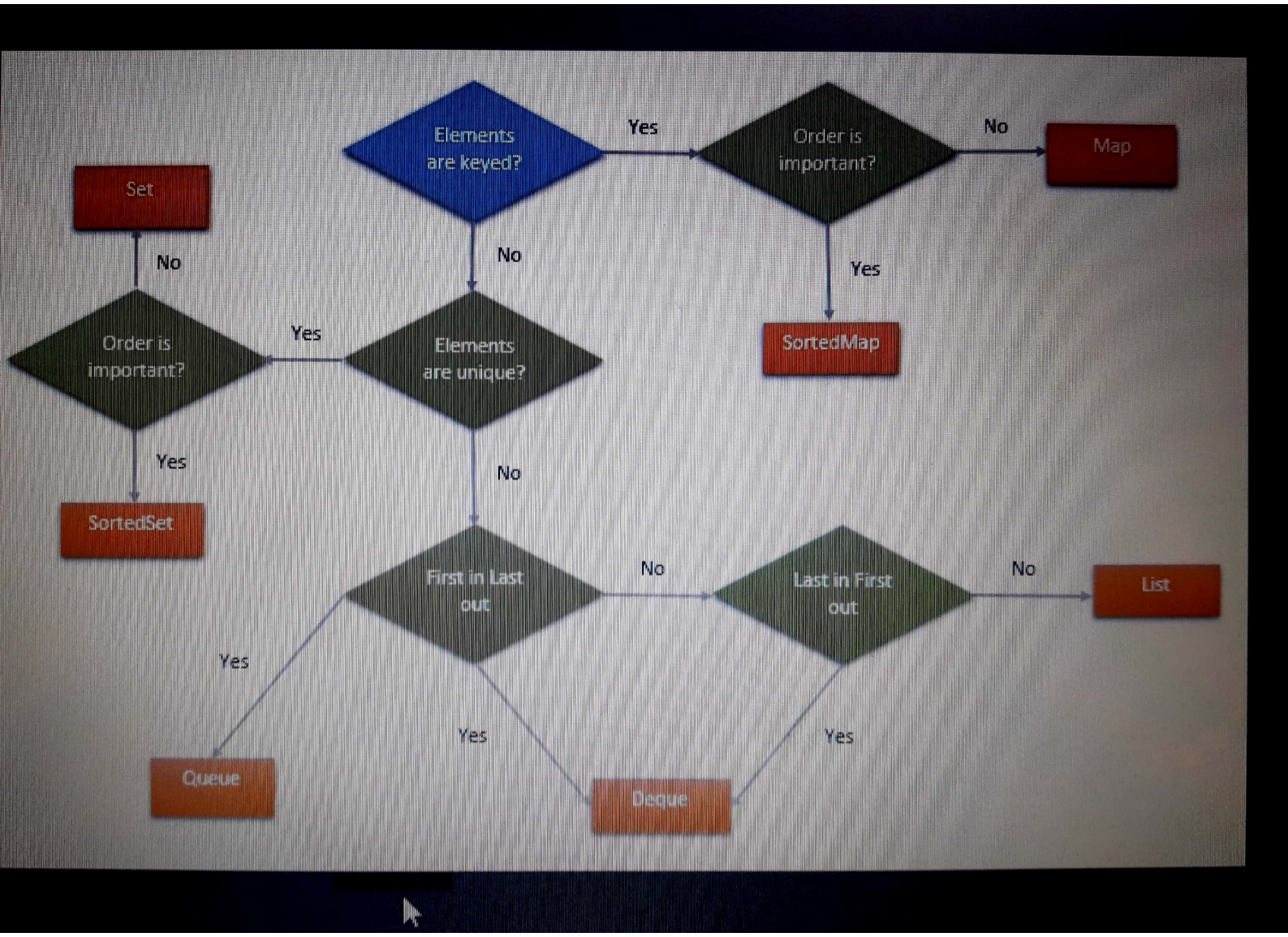
Find common Element count

other	not Exist	true
	Exist	false.



	Size	They can hold	Iteration	How to get size?	Generics	Type Safe	Multi-dimensional	How to add elements?
Array	Fixed	Primitives as well as objects	Only through *for* loop or *for-each* loop	*Length* attribute	Doesn't support	No	Yes	Using assignment operator
ArrayList	Re-sizable	Only objects	Iterators or *for* loop or *for-each* loop	*size()* method	supports	Yes	No	Using *add()* method

	ArrayList	LinkedList	Vector	Stack
Duplicates	Allow	Allow	Allow	Allow
Order	Insertion order	Insertion order	Insertion order	Insertion order
Insert / Delete	Slow	Fast	Slow	Slow
Accessibility	Radom and fast	Sequential and slow	Random and fast	Slow
Traverse	Uses Iterator / ListIterator	Uses Iterator / ListIterator	Enumeration	Enumeration
Synchronization	No	No	Yes	Yes
Increment size	50%	No initial size	100%	100%



	ArrayList	LinkedList
Structure	ArrayList is an index based data structure where each element is associated with an index.	Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.
Insertion And Removal	Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted.	Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed.
Retrieval (Searching or getting an element)	Insertion and removal operations in ArrayList are of order O(n). Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.	Insertion and removal in LinkedList are of order O(1). Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element.
Random Access	Retrieval operation in ArrayList is of order of O(1). ArrayList is of type Random Access. i.e elements can be accessed randomly.	Retrieval operation in LinkedList is of order of O(n). LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element.
Usage	ArrayList can not be used as a Stack or Queue.	LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List.
Memory Occupation	ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it's index.	LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements.
When To Use	If your application does more retrieval than the insertions and deletions, then use ArrayList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

	Vector	ArrayList
Synchronization and thread-safety	Vector is synchronized means that all the method which structurally modifies Vector e.g. add () or remove () are synchronized which makes it thread-safe and allows it to be used safely in a multi-threaded and concurrent environment.	ArrayList methods are not synchronized thus not suitable for use in multi-threaded environment.
Speed and Performance	Since Vector is synchronized and thread-safe it pays price of synchronization which makes it little slow.	ArrayList is way faster than Vector.ArrayList is not synchronized and fast which makes it obvious choice in a single-threaded access environment. You can also use ArrayList in a multi-threaded environment if multiple threads are only reading values from ArrayList or you can create read only ArrayList as well
Capacity	Vector crossed the threshold specified it increases itself by value specified in capacityIncrement field	increase size of ArrayList by calling ensureCapacity () method.
Enumeration and Iterator	Vector can return enumeration of items it hold by calling elements () method	
Legacy	Vector is one of those classes which comes with JDK 1.0 and initially not part of Collection framework but in later version it's been re-factored to implement List interface	

toftheday.com/hashset-vs-linkedhashset-vs-treeset-in-java/

Differences Between HashSet, LinkedHashSet and TreeSet In Java :

	HashSet	LinkedHashSet	TreeSet
How they work internally?	HashSet uses HashMap internally to store it's elements.	LinkedHashSet uses LinkedHashMap internally to store it's elements.	TreeSet uses TreeMap internally to store it's elements.
Order Of Elements	HashSet doesn't maintain any order of elements.	LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted.	TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order.
Performance	HashSet gives better performance than the LinkedHashSet and TreeSet.	The performance of LinkedHashSet is between HashSet and TreeSet. It's performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements.	TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations.

Insertion, Removal And Retrieval Operations	HashSet gives performance of order O(1) for insertion, removal and retrieval operations.	LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations.	TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations.
How they compare the elements?	HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements.	LinkedHashSet also uses equals() and hashCode() methods to compare the elements.	TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn't use equals() and hashCode() methods for comparision of elements.
Null elements	HashSet allows maximum one null element.	LinkedHashSet also allows maximum one null element.	TreeSet doesn't allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException.
Memory Occupation	HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements.	LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements.	TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap.

	HashMap	LinkedHashMap	TreeMap
Implementation	HashMap is implemented as a hash table	LinkedHashMap is Hash table and linked list implementation of the Map interface	TreeMap is implemented based on red-black tree structure
Ordering	HashMap won't maintain ordering on keys or values	LinkedHashMap maintains insertion order of keys, order in which keys are inserted in to LinkedHashMap.	TreeMap is ordered by the key.
Null keys/Values	HashMap allows null keys/values. Allows one null key and many null values.	LinkedHashMap allows null keys/values. Allows one null key and many null values.	TreeMap won't allow null keys and allows many null values.
Performance	A HashMap has a better performance than a LinkedHashMap and TreeMap.	LinkedHashMap has less performance than HashMap and more performance than TreeMap. A LinkedHashMap has a less performance than a HashMap because LinkedHashMap needs the expense of maintaining the doubly-linked list.	TreeMap has less Performance than HashMap and LinkedHashMap. A TreeMap has a less performance because TreeMap arrange the keys in natural sorting order.

Serializable

Property	HashMap	LinkedHashMap	TreeMap
Time Complexity (Big O notation) Get, Put, ContainsKey and Remove method	O(1)	O(1)	O(log n)
Iteration Order	Random	Sorted according to either Insertion Order or Access Order (as specified during construction)	Sorted according to either natural order of keys or comparator (as specified during construction)
Null Keys	allowed	allowed	Not allowed if Key uses Natural Ordering or Comparator does not support comparison on null Keys
Interface	Map	Map	Map, SortedMap and NavigableMap
Synchronization	none, use Collections.synchronizedMap()	None, use Collections.synchronizedMap()	none, use Collections.synchronizedMap()
Data Structure	List of Buckets, if more than 8 entries in bucket then Java 8 will switch to balanced tree from linked list	Doubly Linked List of Buckets	Red-Black Tree (a kind of self-balancing binary search tree) implementation of Binary Tree. This data structure offers O (log n) for Insert, Delete and Search operations and O (n) Space Complexity
Applications	General Purpose, fast retrieval, non-synchronized. ConcurrentHashMap can be used where concurrency is involved.	Can be used for LRU cache, other places where insertion or access order matters	Algorithms where Sorted or Navigable features are required. For example, find among the list of employees whose salary is next to given employee, Range Search, etc.
Requirements for Keys	Equals() and hashCode() needs to be overwritten	Equals() and hashCode() needs to be overwritten	Comparator needs to be supplied for Key implementation, otherwise natural order will be used to

What is the Difference between HashMap and Hashtable? This is one of the frequently asked interview questions for Java/J2EE professionals. HashMap and Hashtable both classes implements Java.util.Map interface, however there are differences in the way they work and their usage. Here we will discuss the differences between these classes.

HashMap vs Hashtable

1) HashMap is non-synchronized. This means if it's used in multithread environment then more than one thread can access and process the HashMap simultaneously.

Hashtable is synchronized. It ensures that no more than one thread can access the Hashtable at a given moment of time. The thread which works on Hashtable acquires a lock on it to make the other threads wait till its work gets completed.

2) HashMap allows one null key and any number of null values.

Hashtable doesn't allow null keys and null values.

3) HashMap implementation LinkedHashMap maintains the insertion order and TreeMap sorts the mappings based on the ascending order of keys.

Hashtable doesn't guarantee any kind of order. It doesn't maintain the mappings in any particular order.

4) Initially Hashtable was not the part of collection framework it has been made a collection framework member later after being retrofitted to implement the Map interface.

HashMap implements Map interface and is a part of collection framework since the beginning.

5) Another difference between these classes is that the iterator of the HashMap is a fail-fast and it throws ConcurrentModificationException if any other Thread modifies the map structurally by adding or removing any element except iterator's own remove() method. In Simple words fail-fast means: When calling iterator.next(), if any modification has been made between the moment the iterator was created and the moment next() is called, a ConcurrentModificationException is immediately thrown.

Enumerator for the Hashtable is not fail-fast.

tps://i.stack.imgur.com/sDoih.png

Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	**Sorted according to natural order of keys**	**Sorted according to the insertion order.**	**Random**
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use Collection.synchronizedMap(new HashMap())			Yes but prefer to use ConcurrentHashMap
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

HashMap Vs HashTable In Java :

HashMap	HashTable
HashMap is not synchronized and therefore it is not thread safe.	HashTable is internally synchronized and therefore it is thread safe.
HashMap allows maximum one null key and any number of null values.	HashTable doesn't allow null keys and null values.
Iterators returned by the HashMap are fail-fast in nature.	Enumeration returned by the HashTable are fail-safe in nature.
HashMap extends AbstractMap class.	HashTable extends Dictionary class.
HashMap returns only iterators to traverse.	HashTable returns both Iterator as well as Enumeration for traversal.
HashMap is fast.	HashTable is slow.
HashMap is not a legacy class.	HashTable is a legacy class.
HashMap is preferred in single threaded applications. If you want to use HashMap in multi threaded application, wrap it using Collections.synchronizedMap() method.	Although HashTable is there to use in multi threaded applications, now a days it is not at all preferred. Because, ConcurrentHashMap is better option than HashTable.

There are five total ways to create objects in Java, which are explained
followed by bytecode of the line which is creating the object.

Using new keyword	} → constructor gets called
Using newInstance() method of Class class	} → constructor gets called
Using newInstance() method of Constructor class	} → constructor gets called
Using clone() method	} → no constructor call
Using deserialization	} → no constructor call

If you will execute program given in end, you will see method 1, 2, 3 use

1. Using new keywords

It is the most common and regular way to create an object and a very simple one also. By using this method we can call whichever constructor we want to call (no-arg constructor as well as parameterized).

```
Employee emp1 = new Employee();
```

```
0: new           #10         // class org/programming/mitra/exercises/Employee
3: dup
4: invokespecial #22         // Method org/programming/mitra/exercises/Employee."<init>":()V
```

2. Using newInstance() method of Class class

We can also use the newInstance() method of a Class class to create an object. This newInstance() method calls the no-arg constructor to create the object.

We can create an object by newInstance() in the following way:

```
Employee emp2 = (Employee) Class.forName("org.programming.mitra.exercises.Employee").newInstance();
```

Or

```
Employee emp2 = Employee.class.newInstance();
```

```
51: invokevirtual    #70    // Method java/lang/Class.newInstance:()Ljava/lang/Object;
```

4. Using newInstance() method of Constructor class

Similar to the newInstance() method of Class class. There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. We can also call parameterized constructor, and private constructor by using this newInstance() method.

```
Constructor<Employee> constructor = Employee.class.getConstructor();
Employee emp3 = constructor.newInstance();
```

```
111: invokevirtual  #88    // Method java/lang/reflect/Constructor.newInstance:([Ljava/lang/Object;)Ljava/lang/Object;
```

4. Using clone() method:

Whenever we call clone() on any object, the JVM actually creates a new object for us and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.

To use clone() method on an object we need to implement Cloneable and define the clone() method in it.

```
Employee emp4 = (Employee) emp1.clone();
```

```
102: invokevirtual #87    // Method org/programming/mitra/exercises/Employee.clone:()Ljava/lang/Object;
```

5. Using deserialization:

Whenever we serialize and deserialize an object, the JVM creates a separate object for us. In deserialization, the JVM doesn't use any constructor to create the object.

To deserialize an object we need to implement a Serializable interface in our class.

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));
Employee emp5 = (Employee) in.readObject();
```

```
261: invokevirtual  #118    // Method java/io/ObjectInputStream.readObject:()Ljava/lang/Object;
```