Java

Mr. Rahul Chauhan

- String
- String Buffer
- String Builder

String :- String is class in java and define in java.lang. It is not primitive data type. It is represent character string.

(i) Difference between String + StringBuffer

String	StringBuffer
(i) immutable	Mutable
exp :- String s = new String("durga") s.concate("Software"); S.op.ln(s); // ⇒ durga	exp :- StringBuffer Sb = new StringBuffer("durga") Sb.append("Software") Sop.ln(Sb); // → durgaSoftware
s → durga → durgaSoftware ↳ Not having reference goes to the garbage collector	Sb → durgasoftware

(ii) == operator in java stands for reference/address comparison

equals() method comes from object class

equals() method comes from object class due to reference comparison / address comparison

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

(2) equals() overridden for content comparison

```
String s1 = new String("durga");
String s2 = new String("durga");

s1 == s2 // false
s1.equals(s2) // True
```

s1 → (durga)

s2 → (durga)

(2) equals() method call from object class

```
StringBuffer sb1 = new StringBuffer("10");
"            sb2 = new "("10")
sb1 == sb2 // false
sb1.equals(sb2) // false
```

sb1 → (10)

sb2 → (10)

2 String Object Creation on Heap and String Constant Pool

* Upto 1.6 version SCP maintain at method Area / PERMGEN but after it will locate on Heap area (1.7 version)

+ String s = new String("Ram")

heap	SCP
s → (Ram)	(Ram) ↓ for reusability purpose

Two object created

String s = "Ram"

SCP

s1 → (Ram)

Only one object created

On SCP object is not eligible for garbage collector because it is maintain by JVM

SCP: it is special storage space in java maintain by JVM that is use to store unique string objects. whenever string object created it first check whether same object with same string is available if yes then it provide the reference of that object otherwise it will create new object.

```
1 String S1 = new String("Ram");
2 String S2 = new String("Ram");
3 String S3 = "Ram";
4 String S4 = "Ram";
```

heap	SCP
S1 → Ram	Ram
S2 → Ram	S3, S4 → Ram

* whenever we create object by using new operator then it will always create new object on heap and on SCP if it is not available.

```
String s = new String("Ram");
s.concat("Software");
s = s.concat("Solutions");
```

Heap	SCP
S1 → Ram	Ram
RamSoftware	Software
S1 → RamSolutions	Solutions

* Any Runtime operation create new object of string only on HEAP area not on SCP concate in runtime operation will create object on heap only.

```
String S1 = "Ram" + "Sita";
```

SCP: S1 = RamSita

* Both Ram and sita is constant so it resolve at compile time and so it will create object on SCP

heap | Scp

+ String S1 = "Ram";

String S2 = S1 + "Sita"; S2 → (RamSita)

S1 → (Ram)

(Sita)

* Here S1 is variable object and "sita" is String constant so S1 is resolve at runtime and it will create object at runtime on heap

* If we make any String variable as FINAL then it is resolve at compile time only.

Importance of String Constant Pool

* If any string object is required to use repeatedly then it is not recomended to create multiple objects with same content it reduces performance of the system and effects memory utilization.

* We can create only one copy and we can reuse the same object for every requirment. We can achieve this by using "scp"

* In Scp several references pointing the same object. If by using one reference, performing any change the remaining references will be impacted. To overcome this problem, immutability concept for string object implemented.

Other immutable object in java
- All wrapper class objects

• Important Constructor of string class

(1) String s = new String()
↳ we can create empty string objects

2 String s = new String(String literal);

3 String s = new String(StringBuffer sb);

4 String s = new String(StringBuilder sb);

5 String s = new String(char[] ch);

6 String s = new String(byte[] b);

exp byte[] b = {97, 98, 99, 100}. -128 to 127
String s = new String(b);
System.out.println(s); // → abcd

• Important methods of String Class

SUNDAY 7

1 Public char charAt(int index) - Returns the character locating at specified index

String s = "RamSita";
s.charAt(3); // → S

2 Public String concate(String str) - (+ or +=)
s.concate("LOVE"); // → RamSitaLove

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

3 Public boolean equals (Object o) → for content comparison
3 Public boolean equalsIgnore (object o) → for content comparison but case is not imperfect

4 Public String subString (int begin) → Return the substring from begin index to End of string

exp

String s = "RamSita";
s.subString(3); // → Sita

5 Public String subString (int begin, int End)

6 Public int length() → returns the length of string
s.length() // → 7

7 Public String toLowerCase() → Convert into lower case
s.toLowerCase() // → ramsita

(8) Public String toUpperCase() → convert into Upper case
s.toUpperCase() // → RAMSITA

9 Public String trim() → Remove blank space at begin and end of the string.
s.trim(); // →

10. Public int indexOf (char ch) → Return the index of 1st occurrence of the specified character

s.indexOf(S) // → 3
s.indexOf(x) // → -1

11. Public int lastIndexOf(char ch) → returns index of last occurrence of specified index.

22 Public String replace(char old, char new)
to replace every old character with new character

String s = "Banana";

s.replace(a, b) // → Bbnbnb

- Creation of Our own immutable Class

```
class Test {
  final private int i;
      int i
    Test ( ) {
        this.i = i;
    }
  final public Test modify (int i)
    {
        if (this.i == i) {
            return this;
        }
        else {
            return new Test(i);
        }
}

    Test t1 = new Test(10);
    Test t2 = t1.modify(100);
    Test t3 = t.modify(10);
```

Final vs immutability

* Final modifier applicable for variables where as immutability concept applicable for objects.

* If the reference variable declared as final. Its mean we cant reasign that variable. It's doesn't mean that we cant change in that object.

* That is by declaring a reference variable as final we won't get any immutability nature.

```
=> class Test
   {
        public void static void main (String [] args)
        {
          final StringBuffer sb = new StringBuffer("Rom");
           sb.append("Software");
           System.out.println(sb) // => RomSoftware

           sb = new StringBuffer("ramSita") // ERROR
        }
   }
```

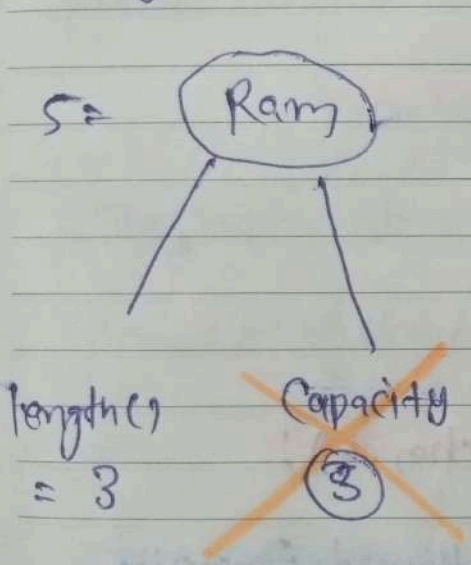
Need of StringBuffer

* If the content will change frequently then never recommended to go for String objects for every change a new object will be created internally.

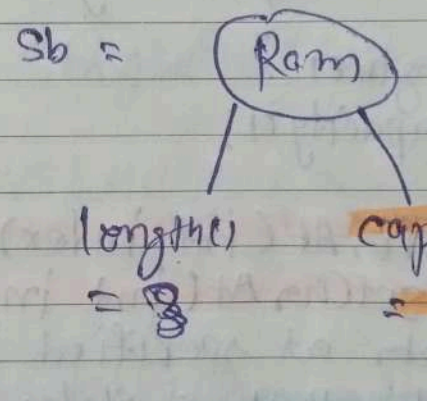
* To handle this type of requirment we should go for string Buffer.

* The main advantage of StringBuffer over string is, all required changes will be performed in existing object.

String s = new String("Ram")



StringBuffer sb = new StringBuffer("Ram");



- StringBuffer class constructor

1. StringBuffer sb = new StringBuffer();
   sb.capacity() // ⇒ 16
   intial capacity = 16
   new capacity = (cc+1) x 2

   (16+1) x 2 = 34
   (34+1) x 2 = 70

2. StringBuffer sb = new StringBuffer(int intialCapacity)

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

3 ***

StringBuffer sb = new StringBuffer(String s);

StringBuffer sb = new StringBuffer("Ram");

sb.capacity() // => 3+16 = 19

Here capacity = s.length() + 16
= 3+16 = 19

- Important methods of StringBuffer

1 Public int length();
2 public int capacity();

3 public char charAt(int index);
4 public void setCharAt(int index, char ch);
↳ replace character at specified index
x There is no StringBufferIndexOutOfBoundsException
it always give StringIndexOutOfException

5 public StringBuffer append(String s);
append(byte b);
append(int i);
append(long l);
append(float f);
} Overloaded method.

append method always add last

(6) public StringBuffer insert (int index, String s)
insert (int index, boolean bl)
...
insert (int index, float f)

Use -> To add at specified index

(7) public StringBuffer delete(int begin, int end);

from begin to end-1 index.

StringBuffer sb = new StringBuffer("abcdefgh")

sb.delete(2,5);
SOPLN (sb) // => abfgh

(8) public StringBuffer deleteCharAt (int index)

9 public StringBuffer reverse()

SB sb = new SB("Ram")
sb.reverse() // => max

SUNDAY 14

10 public void setLength(int length);

SB sb = new SB("abcdefgh");
sb.setLength(4);

SOPLN (Sb.) // => abcd

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

(11) public void ensureCapacity(int capacity)

To increse dynamically capacity of stringbuffer we used it

⇒

```
SB sb = new SB();
SOPLN(sb) // ⇒ 16

sb.ensureCapacity(1000)
SOPLN(sb) // ⇒ 1000
```

(12) public void trimToSize();

To deallocate the extra allocated free memory such that capacity and size are equal

```
SB sb = new SB(1000);
sb.append("Ram");
sb.capacity(); // ⇒ 1000
sb.trimToSize() // ⇒ 3
```

- 8) Need of StringBuilder

- Multithread enviornment stringBuffer is not suitable be because all method inside it synchronized.

  change ⇒

  1. Buffer with Builder
  2. remove synchronized keyword.

FEBRUARY 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

String Buffer	String Builder
1 Every method is Synchronized.	1 No method is Synchronized.
2 Thread Safe, only one thread allowed to operate String Buffer object at a time.	2 Not thread Safe, Multiple thread allowed to operate String Builder object at a time.
3 Performance is low because threads are required to wait to operate on String Buffer object.	3 Performance is high because threads are not required to wait to operate on String Builder objects..
4 Introduced in 1.0 version	4 Introduced in 1.5 version

* String class is final class, inside every method is also final.

Collection Framework

Array

* An array is indexed collection of fixed no of size. Homogenous data elements.
* We can represent Hudge No of elemets as a single unit.
* disadvantage - fixed size. once we create array we can not increase and decrase of size.

Array declaration

(i) 1-D Array -:

```
int[] x;    int arry x
int []x;
int x[];

int[6] x;   // ERROR
```

* At the time of declaration we can't specify size

(ii) 2-D Array -:

```
int[][]   x;
int   [][] x;
int   x[][];
int[]   []x;
int[]   x[];
int   []x[];
```

FEBRUARY 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

int []	a, b ;	a → 1D	b → 1D
int []	a[], b ;	a - 2D	b → 1D
int []	a[], b[];	a - 2D	b - 2D
int []	[]a, b;	a - 2D	b - 2D
int []	[]a, b[];	a - 2D	b - 3D
✗ int []	a[]a, []b;	// ⇒ ERROR	

* Every array in Java is object only, Hence we create an array by using new operator.

```
int [] a = new int[3];

SOPLN (a.getClass().getName());  //⇒ [I    1-D Array
                                 //   [[I   2-D Array.
```

Array Type	Corresponding class Name
int []	[I
int [][]	[[I
double[]	[D
Short[]	[S
byte []	[B
boolean[]	[Z

```
int[] arr = new int[0]
```

exm

```
public static void main (String[] args)
```

* int [] x = new int [-3]; // - Runtime Exception
Here compiler just check only size data is int or not so it give runtime exception when gym see the size in negative.
→ Negative Array Size Exception

byte → short
int → long → float → double
char

Allow data types are byte, short, char, int only in size of array, otherwise we will get compile time error

int data type → 2147483647

Max array size int [] arr = new int [2147483647]; ✓

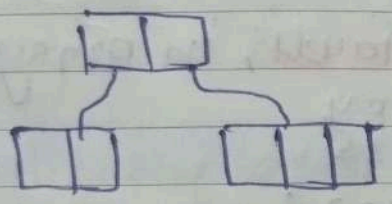
int [] arr = new int [2147483648]; ✗

CE → integer number to large.

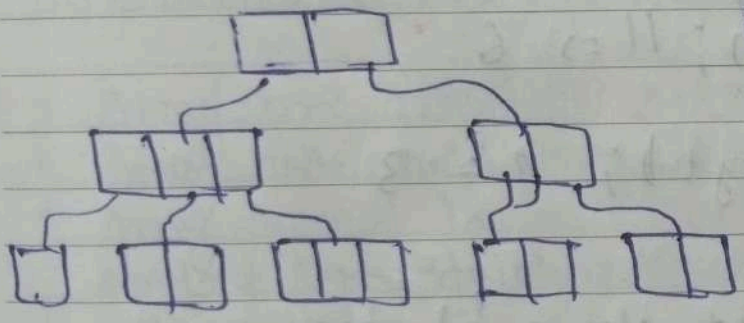
Out of memory error

2-D Array Creation -:

* in Java 2-D Array not implemented by using metrix style, sum people follows Array of Array for multidimensial creation.
  + Memory utilization is improved.



```
int[][] x = new int[2][]
x[0] = new int[2];
x[1] = new int[3];
```



```
int[][][] x = new int[2][][]
x[0] = new int[3][];
x[0][0] = new int[1];
x[0][1] = new int[2];
x[0][2] = new int[3];
x[1] = new int[2][2];
```

in java.lang.Object class toString method return string in the form

SUNDAY 21

className@hashCode-in-hexadecimalform

Array declaration, creation and intialization in a one line

```
int[] x = {10,20,30}
char[] ch = {'a','b','c'}
String[] s = {"Ram","Shyamy","Sita"}
int[][] x = {{10,20},{10,20,30}};
```

- length vs length()

```
int[] x = new int[6];
SOPLN(x.length()) // => gives compile time error

SOPLN(x.length) // => 6
```

* length variable is final by default, in array we can not change length of array.

```
int[][] x = new int[6][3];

SOPLN(x.length); // => 6

SOPLN(x[0].length); // => 3
```

- Anonymous Array - without Name of array.
  Use - One time Use.
  ex- new int[]{10,20,30,40}

- Promotion -:

```
int[] x = {10,20,30,40};
char[] ch = {'a','b','c'};

int[] b = x; ✓

int[] c = ch; X  char → int
                 but
char[] to int[] not possible
```

* Whenever we assign one array to another array just reference variable reasign, element will be not copied

int[] a = {10,20,30,40,50}

int[] b = {70,80}

a = b , b = a

- Collection Framework :
- Need of Collection framework

Limitations of array

+ Array are fixed in size
+ Array can hold only Homogenous data type.
+ Array concept is not implemented based on some standard Data Structure. Hence readymade method support not available.

To overcome the above limitations of arrays, we should go for collections.

+ Collection are growable in nature.
+ Hold Hetrogeneous & Homogeneous elements
+ Every collection is implemented based on some standard Data structure.

- Difference between Arrays and Collections.

Arrays	Collections
1 Arrays are fixed in size	1 Growable in nature
2 Wrt memory, arrays are not recommended to use	2 Wrt memory, collection are recommended to use.
3 Performance is good	3 Performance is not good.
4 Hold Homogeneous type data elements only	4 Hold Homogeneous and Heterogeneous data type.
5 Not based on Standard data structure	5 Based on standard Data Structure
6 Array can hold primitive as well as object type	Collections can hold only objects type.

- What is Collection?

⇒ If we want to represent a group of individual objects as a single entity then we should go to Collections. To define collection required some classes and interfaces. These classes and interfaces available in Collection framework.

Java	C++
1 Collection	Container
2 Collection framework	STL (Standard Template Library)

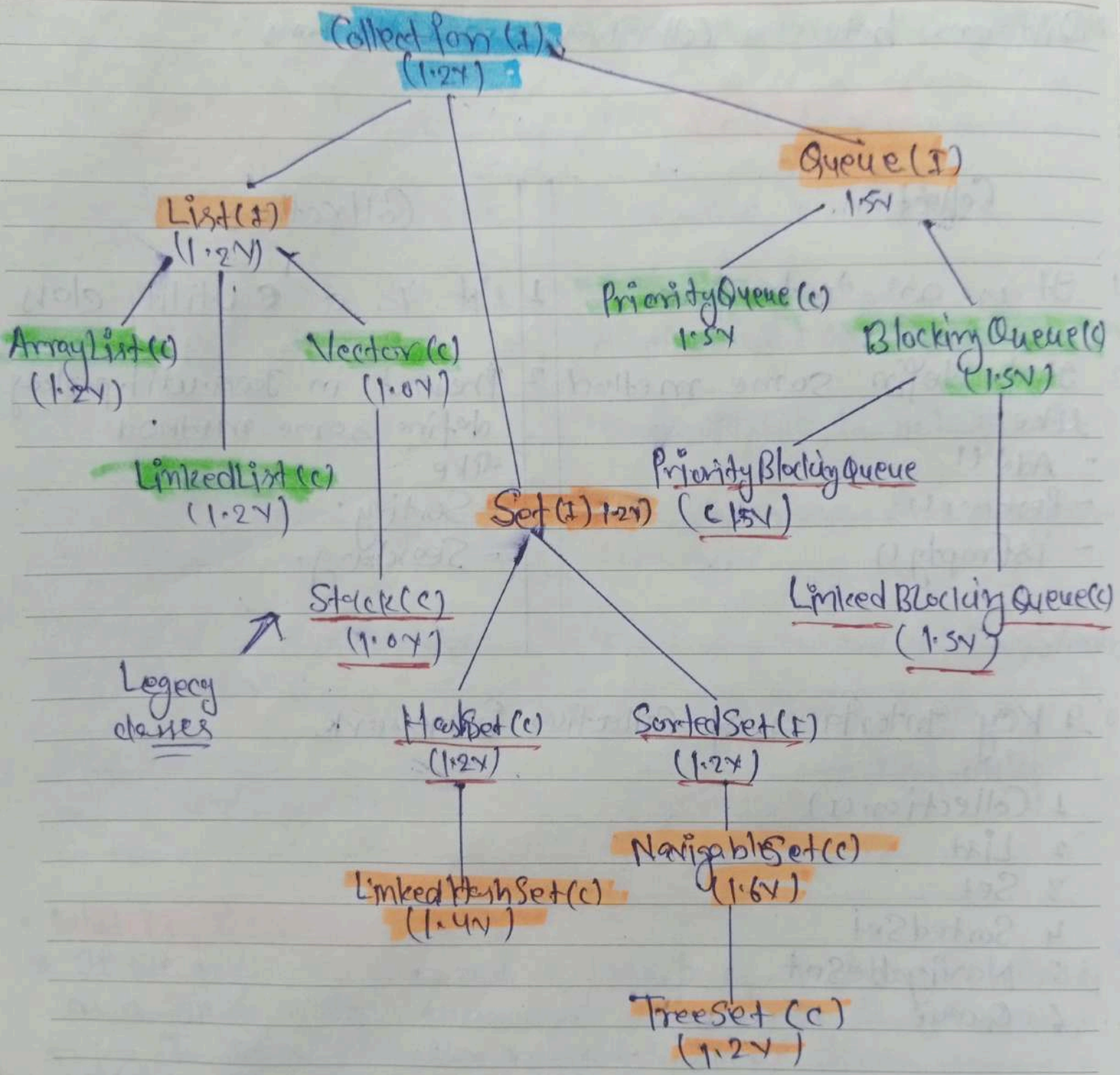
- Difference between Collection and Collections

Collection	Collections
1 It is an interface	1 It is a Utility class
2 It is defin some method like - Add() - Remove() - isEmpty()	2 Present in Java.util.package define some method like - Sorting. - Searching.

- 9 Key Interfaces of Collection framework

1 Collection(I)
2 List
3 Set
4 SortedSet
5 NavigableSet
6 Queue

7 Map
8 SortedMap
9 NavigableMap


Collection (I)
(1.2X)
List (I)
(1.2X)
Queue (I)
1.5X
ArrayList (C)
(1.2X)
Vector (C)
(1.0X)
PriorityQueue (C)
1.5X
BlockingQueue (C)
(1.5X)
LinkedList (C)
(1.2X)
Set (I) 1.2X
PriorityBlockingQueue
(C 1.5X)
Stack (C)
(1.0X)
LinkedBlockingQueue (C)
(1.5X)
Legecy classes
HashSet (C)
(1.2X)
SortedSet (I)
(1.2X)
NavigableSet (C)
(1.6X)
LinkedHashSet (C)
(1.4X)
TreeSet (C)
(1.2X)

* List (I) -:

If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order preserved then we should go for List.

* Set (I) -:

If we want to represent a group of individual objects as a single entity where duplicates are not allowed but insertion order not preserved. then we go for Set.

° Difference between List and Set

List	Set
* Duplicates are allowed	* Duplicates are not allowed.
* Insertion Order preserved	* Insertion Order not preserved.

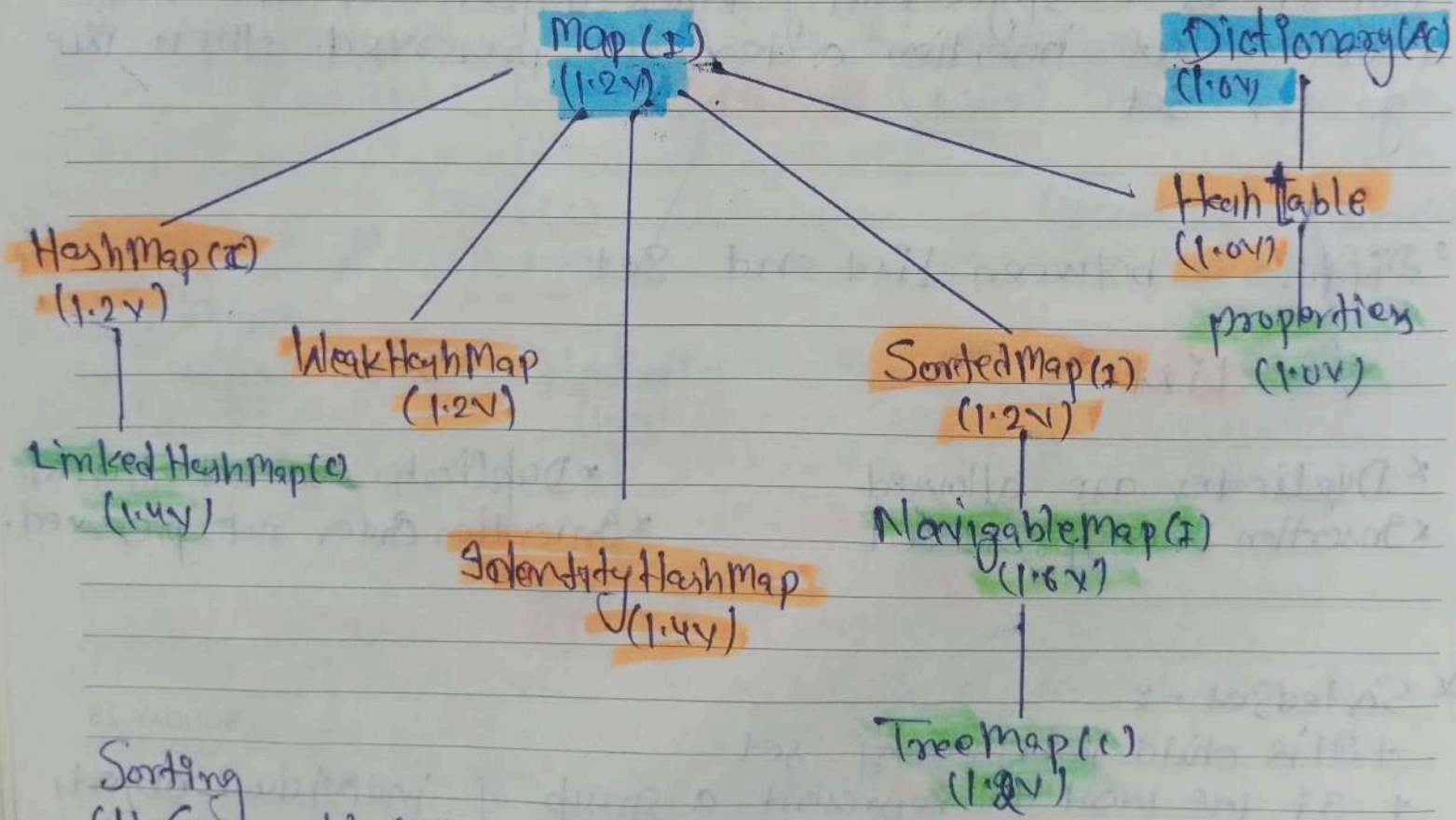
* SortedSet -:

+ It's child interface of set
+ If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting order then we should go for sortedset.

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

* NavigableSet -:
It is the child interface of SortedSet it defines ~~serv~~ methods for navigation purpose.

* Queue -:
if we want to represent group of individual objects prior to processing then we should go for queue.



Sorting
(1) Comparable (I)
(2) Comparator (I)

FEBRUARY 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

Cursors
1 Enumeration (I)
2. Iterator (I)
3. List Iterator (I)

Utility Classes
1 Collections
2 Arrays

* Map :-
  - It is not child interface of collection.
  - If we want to represent ~~the~~ group of objects as key-value pairs then we should go for Map interface.
  - Duplicate keys are not allowed but value can be duplicate.

* SortedMap :-
  - Child interface of Map
  - If we want to represent group of objects as key-value pairs according to some sorting order of key then we should go for SortedMap.
  - Sorting should be based on key but Not based on value.

* NavigableMap :-
  - It defines several utility methods for navigation purpose.

• Collection Framework in Details

* Important methods of collection Interface-

  + boolean add(object o)
  + boolean addAll(Collection c)
  + boolean remove(object o)
  + boolean removeAll(Collection c)
  + boolean retainAll(Collection c)
  + void clear()

+ boolean contains(Object o)
+ boolean containsAll(Collection c)
+ boolean isEmpty()
+ int size()
+ Object [] toArray()
+ Iterator iterator()

Note:-
Collection interface doesn't contain any method to retrieve objects, there is no concrete class which implements collection class directly.

* List Interface
+ ~~Ino~~ Duplicate are allowed
+ Insertion Order preserved.

Methods-
+ void add(int index, Object o)
+ boolean addAll(int Index, Collection c)
+ Object get(int index)
+ Object remove(int index)
+ Object set(int index, Object ~~o~~ me)
+ int indexOf(Object o)
+ int lastIndexOf(Object o)
+ ListIterator listIterator();

Note * In TreeSet and TreeMap Hetrogeneous objects are not allowed

- ArrayList (c)

* The undefined data structure Resizable Array
* Duplicates are allowed
* Insertion order is preserved
* Hetrogeneous objects are allowed
* Null insertion is possible

+ ArrayList Constructor

1. ArrayList al = new ArrayList()

   intial capacity of ArrayList is = 10

   new capacity = $(cc * \frac{3}{2}) + 1$

2. ArrayList al = new ArrayList(int intialCapacity)

3. ArraList al = new ArrayList (Collection c)

=> import java.util.*;
class ArrayList
{
public static void main(String[] args){
ArrayList al = new ArrayList();
al.add ("A");
al.add ("10");
al.add ("A");
al.add (null);
System.out.println (al) // => [A, 10, A, null]

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22
24 25 26 27 28 29

```
al.remove(2);
System.out.println(al); //=> [A,10,null]

al.add("N");
System.out.println(al) //=> [A,10,null,N]

al.add(2,"M");
                        //=> [A,10,M,null,N]
}
}
```

To transfere objects from one place to another place object should be ~~serizable~~ object
seriolizable

So every collection class implements seriolizable interface

This transfered object recieved by one reciever class which directly not performed any operation so it will create ~~object of~~ copy of this recieved objects so thats why every collection class implements from cloneable interface

ArrayList and Vector class implements ~~so~~ RandomAccess so any random element we can able to access with some speed.

RandomAccess - marker interface → java.util package
Best choice → frequent operation is retrieval
Worst choice → Insertion or deletion in the middle

MARCH 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

- Difference between ArrayList + Vector

ArrayList	Vector
1 Every method Non-synchronize	1 Every Method is Synchronized
2 Not Thread Safe	2 Thread Safe
3 Performance is high	3 Performance is Low
4. Introduced in 1.2 version	4 Introduced in 1.0 version

- How to get Synchronize version of ArrayList object.?

public static List synchronizedList (List l)

Non-Synchronized
ArrayList al = new ArrayList();

Synchronized
List l = Collections.synchronizedList(al);

public static Set synchronizedSet (set s);
public static Map synchronizedMap (Map m);

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

LinkedList

In linked list we ↑ store objects in consecutive memory location

* Underlying data structure is doubly linked list.
* Insertion order is preserved
* Duplicates are allowed
* Heterogeneous elements are allowed
* null insertion is possible

* Best choice - insertion and deletion operation in the middle.
* Worst choice - frequent operation is retrieval.

Linked List Method

+ void addFirst(~~obj~~ Object o)
+ void addLast(~~)~~ Object o)
+ Object getFirst()
+ Object getLast()
+ Object removeFirst()
+ Object removeLast()

Constructor in LinkedList

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

```
=> import java.util.*;

class LinkedList
{
        public static void main (String[] args){
         LinkedList l = new LinkedList();
          l.add ("durga");
          l.add ("30");
          l.add (null);
          l.add ("Ram");
         System.out.println (l); //=>
 //=>        [durga, 30, null, Ram]
        l.set (0, "Shyam");
        l.add (0, "Shiva");
        l.removeLast();
        l.addFirst ("AAA")
      System.out.println (l)
        //= [AAA, Shiva, Shyam, 30, null]
    }
}
```

- Difference between ArrayList and LinkedList

ArrayList	LinkedList
1 Best choice -> retrievel operation	1 Best choice -> Insertion + deletion at middle
2 Worst -> insertion + deletion at middle	2 Worst choice -> retrievel operation
3 Underlaying data structure - resizable or growable Array	3 Underlaying Data structure - Doubly linked List
4 Implements RandomAccess interface	4 Not implement RandomAccess

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

- Vector

* Underlaying data structure - Resizable or growable array
* Duplicates objects are allowed
* null insertion allowed
* Insertion order preserved
* Hetrogeneous objects are allowed
* vector class implements, serializable, cloneable and RandomAccess
* Most of method are synchronized. Thread safe.
* Best choice in retrivel operation.

Constructor of vector class

1 Vector v = new Vector();
initial capacity = 10
new capacity = 2 * current capacity.

2 Vector v = new Vector(int initialCapacity)

3 Vector v = new Vector(int initialCapacity, int incrementalCapacity)

4 Vector v = new Vector(Collection c)

- Three Cursors of java

1 Enumeration
2. Iterator
3. ~~Iterator~~ ~~List~~ ~~ListIterator~~

* If we want to retrive objects one by one from the collection then we should go for cursor.

1 Enumeration (I)

We can use enumeration to get objects one by one from a collection (legacy collection)

Public Enumeration elements()

exp

Enumeration e = v.elements()

methods

public boolean hasMoreElements();
public Object nextElements();

```
while (e.hasMoreElements())
    {
        SOPLN (Integer i = (Integer) e.nextElement();
    }
```

2 Iterator (I)

- Enumeration is applicable for only vector.
- In Enumeration we can read only operation.

To overcome above we used iterator

- Iterator is universal cursor
- By using iterator we can perform read and remove operation

To get iterator Object

public Iterator iterator()

eg

Iterator itr = c.iterator()
↳ any collection object

MAY 2010
M T W T F S S
1 2
4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

methods.

(1) public boolean hasNext();
(2) public Object next();
(3) public void remove();

⇒ import java.util.*;

```
class Demo
{       public static void main(String[] args) {
        ArrayList al = new ArrayList();

        for (int i = 0; i <= 10; i++) {
            al.add(i);
        }
        SOPLN(al); // => [0, 1, 2 ..... 10]
        Iterator itr = al.iterator();
        while (itr.hasNext()){
            Integer I = (Integer) itr.next();
            if( I%2 == 0){
                SOPLN(I);
            else
            itr.remove();
            }
        }
}
```

- Limitations :-
  * Enumeration and iterator are single direction (forward) cursor
  * perform read and remove, not replace

3 ListIterator (I)

* ListIterator are Bidirectional cursor
* provide method to read, remove, replacement and addition of new object method.

In List(I).

-> public ListIterator listIterator()

exp

ListIterator itr = l.listIterator();
↳ List object

Methods -

Forward direction

1 public boolean hasNext();
2 public void next();
3 public int nextIndex();

Backward direction

4 public boolean hasPrevious()
5 public void previous()
6 public int previousIndex()

Other capability methods

7 public void remove()
8 public void set(Object new)
9 public void add(Object new)

=> import java.util.*;

class Demo {

public static void main(String[] args) {

```
LinkedList l = new LinkedList();
    l.add("Ram");
    l.add("Soft");
    l.add("Tech");
SOPLN(l) // = [Ram, Soft, Tech].
ListIterator itr = l.listIterator();
while (itr.hasNext()) {
        String s = (String) itr.next();
            if (s.equals("soft")) {
                itr.remove;
            else
                itr.add("India");
        }
        SOPLN(itr); //[Ram, Tech, India].
    }
}
```

Note :- It is applicable only for List Implemented classes.

Collection (I) (1.2V)

Set (I) (1.2V)

HashSet (1.2V)

LinkedHashSet (1.4V)

SortedSet (I) 1.2V

NavigableSet (I) 1.6V

TreeSet (1.2V)

MARCH 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

- Set(I)

If we want to represent group of individual objects as a single entity, where duplicates are not allowed and insertion order is not preserved then we should go for Set.
Set interface doesn't contain any new method.

- HashSet

* The underlying Data Structure is Hashtable.
* Duplicates are not allowed, if we want to add duplicate add() method return false.
* Insertion order is not preserved, Object will be added based on hash-code of objects.
* Heteregeneous objects are allowed
* null insertion is possible
* Implements serializable and cloneable interface.
* Best choice - operation is searching.

Constructor -:

1 HashSet h = new HashSet();
initial capacity = 16, Fillratio/Load factor = 0.75

2 HashSet h = new HashSet(int initialCapacity)

3 HashSet h = new HashSet(int initialCapacity, float loadFactor);

4 HashSet h = new HashSet(Collection c)

```
=> import java.util.*;

class Demo {
        public Static Void main(String[] args) {
OR  LinkedHashSet h = new LinkedHashSet();
        HashSet h = new HashSet();
            h.add("B");
            h.add("C");
            h.add("d");
            h.add("null");
            h.add("10");
        SYOPLN(h.add(C)); //=> false

        SYOPLN(h); // [null, D, B, C, 10]
        }                  // [B, C, D, null, 10]
}
```

- **LinkedHashSet (1.4V)**

* It is child class of HashSet

HashSet	LinkedHashSet
1 Underlaying Data Structure is Hashtable	1 Underlaying data Structure - Hybrid data Structure (Hash table + Linked list)
2 Insertion Order is not preserved	2 Insertion order is preserved.
3 Introduced in 1.2 V	3 Introduced in 1.4 V

MARCH 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

Use- cache based Application
in the cache memory duplicates not allowed but but insertion order is required.

- SortedSet (I)
If we want to represent group of individual object as single entity but according to some sorting order and duplicates are not allowed then we should go for SortedSet.

methods -:
+ object first(); → Return First element
+ object last(); → Return last element

+ SortedSet headSet(Object obj);
↳ return the SortedSet whose elements are < obj.
+ SortedSet tailSet(Object obj);
↳ return the sorteset whose elements are > obj
+ SortedSet subSet(Object obj1, Object obj2)
↳ return the sorted set whose elements are >= obj1 and < obj2

+ Comparator comparator()
↳ return Comparator object that describes underlying sorting technique. If we will using default natural sorting order then we will get null.
Defaut Sorting -
Number - Ascending order
String - Alphabetical order

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15
17 18 19 20 21 22
24 25 26 27 28 29

TreeSet (c)

* Underlayling datastructure - Balanced Tree
* Duplicates objects are not allowed
* Insertion order not preserved, but all objects insert according to some sorting order.
* Hetrogeneous objects are not allowed
  ↳ gives exception → ClassCastException.
* Null insertion is allowed but only once.

Constructors

1 TreeSet t = new TreeSet();
  ↳ Defaulit natural Sorting Order

2 TreeSet t = new TreeSet(comparator c);
  ↳ Customized Sorting Order

3 TreeSet t = new TreeSet(Collection c);

4 TreeSet t = new TreeSet(SortedSet s);

⇒

```
import java.util.*;
class Demo{
    public static void main(String[] args){
        TreeSet t = new TreeSet();
        t.add("A");
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");
```

```
    // t.add(new Integer(10)); -> ClassCastException
    // t.add(null); // NullPointerException - RE
    System.out.println(t); // => [A, B, L, Z, a]
  }
}
```

Null Acceptance.—

1 For empty TreeSet as the first element we can add null ~~but~~ because there is no need of comparison but after adding null we add another object then we get NullPointerException.

2 For Non empty TreeSet, If we trying to insert null we will get NullPointerException.

Case-I

```
=> import java.util.*;
   class Demo {
       public static void main(String[] args) {
           TreeSet t = new TreeSet();

           t.add(new StringBuffer("A"));
           t.add(new StringBuffer("B"));
           SOPLN(t); // ClassCastException
       }
   }
```


1. If we depending upon defaect natural sorting then object class should be implemet comparable interface class. Otherwise gives - CCE
2 All wrapper classes and string class implement comparable interface. but String buffer not.

- Comparable(I) – java.lang. package
  method
  public int compareTo(Object obj)

  To call obj1.compareTo(Obj2);
  if it returns
  -ve → obj1 has to come before Obj2
  +ve → obj1 has to come after obj2
  0 → obj1 and Obj2 are equals.

⇒
```
class demo {
    public static void main(String[] args) {
      SOPLN("A".compareTo("2")); // -ve
      SOPLN("2".compareTo("A")); //+ve
      SOPLN("A".compareTo("A")); // 0
      SOPLN("A".compareTo(null)); // NullPointerException
    }
}
```

Obj1.compareTo(obj2)
↓ which object to be insert
↓ which object allreadey insrted.

- Comparator (I) java.util package
  method
  (1) public int compare(Object obj1, Object Obj2)
  returns
  -ve → obj1 have to come befor Obj 2
  +ve → Obj1 have to come after obj 2
  0 → obj1 and Obj 2 are equals.

MARCH 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

Q) public boolean equals()

* We can use comparator to define our own sorting (customized sorting)
* When ever we are implementing Comparator interface, compulsory we should provide implementation for compare() method, equals() method is optional.
* equals() method available in Object class. so it will inherit from there.

Program to insert integer Objects into the TreeSet where the sorting order is descending.

```
=> import java.util.*;
   class Demo {
    public static void main(String[] args){
TreeSet t = new TreeSet(new MyComp());
  t.add(10);
  t.add(0);
  t.add(15);
  SOPLN(t); //=> [15, 10, 0]
  }
}

class MyComp implements Comparator {
  public int compare(Object obj1, Object obj2){
      Integer I1 = (Integer) obj1;
      Integer I2 = (Integer) obj2;
      if(I1 < I2)
         return +1;
      else if (I1 > I2)
         return -1;
  } else return 0;
}
```

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15
17 18 19 20 21 22
24 25 26 27 28 29

Program to insert string objects into the TreeSet where sorting order is reverse of Alphabetical order.

```
Class MyComp implements Comparator {
    public int compare(Object obj1, Object obj2){
        String S1 = (String) Obj1;
        String S2 = Obj2.toString();
        // return S2.compareTo(S1);
        return -S1.compareTo(S2);
    }
}
```

Note:-

1 If we depending on default sorting order the our objects should be homogeneous and comparable otherwise give CCE

(2) If we defining our own sorting then objects need not be homogenous and comparable

Comparable	Comparator
1 Default natural sorting order	1 Customize sorting order
2 Java.lang package	2 java.util package.
3 Define compareTo() method	3 Define compare() + equals()
4 All wrapper classes and string class implements	4 Collator and RuleBasedCollator implements.

MARCH 2010
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

Comperision table of Set(I) implemented classes

Property	HashSet	LinkedHashSet	TreeSet
1 Underlay Data structure	Hashtable	Hashtable+LinkedList	Balanced Tree
2 Insertion Order	Not Preserved	Preserved	Not Preserved
3 Sorting Order	Not Applicable	Not Applicable	Applicable
4 Hetrogeneous objects	Allowed	Allowed	Not Allowed*
5 Duplicates objects	Not Allowed	Not Allowed	Not Allowed
6 Null Acceptence	Only Once	Only Once	Fore Empty set as first element in other not allowed

Map(I) (1.2V)

* Map is not child interface of collection.
* If we want to represent group of objects in key-value pair then we should go for map
* Both key and value are objects only
* Duplicates key are not allowed but value can be duplicated
* Each key-value pair is called an Entry

Methods

1 Object put (Object key, Object value)
↳ To add any key-value. if key already available then old value will be replaced ~~and~~ with new value and this method return old value. otherwise return null

exp
```
m.put(100, "Ram");
m.put(101, "Sita");
m.put(100, "Rama");
```

2 Void putAll (Map m)
3 Object get (Object key)

4 Object remove (Object key)
5 boolean containsKey (Object key)
6 boolean containsValue (Object value)
7 boolean isEmpty()
8

8. int size()
9. void clear()

Key	Value
101	Ram
102	Shyam
103	Shur

Set — Collection

Entry Set

1 Set keySet()
↳ return all key as set
2 Collection values()
↳ return all values as collection
3 Set entrySet()
↳ return entry as set

} Collection views of Map

- Entry (I)
  - * Each key-value pair called Entry
  - * Without existing Map Objects there is no chance of existing Entry interface.
  - * Entry interface define inside of Map interface.

Methods

1 Object getKey()
2 Object getValue()
3 Object setValue()

- HashMap

* Under Data Structure is Hashtable
* Duplicat key not allowed but value can can be duplicated.
* Hetrogeneous objects are allowed for both keys and value.
* Insertion order not preserved and it is based on hashcode of the key.
* null insertion is allowed for key (only once) and for values (Any No. of time)
* Best choice for searching operation

M	T	W	T	F	S	S
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

* HashMap implements serilizable and clonable but not for RandomAccess.

Constructor

All constructor same as HashSet

1 HashMap m = new Hashmap();
intialCapacity = 16
load factor = 0.75

2 HashMap m = new HashMap(intial intialCapacity)

3 HashMap m = new HashMap(int intialCapacity, float loadfactor)

4 HashMap m = new HashMap(Map m)

⇒
```
import java.util.*;

class demo{
    public static void main(String[] args){

        HashMap m = new HashMap();
            m.put("AB", 700);
            m.put("BC", 800);
            m.put("XY", 1000);
            m.put("MN", 500);
        SOPLN(m); // {BC=800, XY=1000, AB=700, MN=500}

        SOPLN(m.put("MN", 200)); // 500
```

```
Set s = m.keySet();
  SOPLN(s); // => [BC, XY, AB, MN]

  Collection c = m.values();
SOPLN(c); // [800, 1000, 700, 200]

  Set S1 = m.entrySet()
SOPLN(S1); // [BC=800, XY=1000, AB=700, MN=200]

  Iterator itr = S1.iterator();

  while (itr.hasNext()){
      Map.Entry m1 = (Map.Entry)itr.next();
      SOPLN(m1.getKey() + "...." + m1.getValue());
  }
}
}
```

0 Difference between HashMap + Hash table.

	HashMap		Hash table
1	No method is Synchronized	1	Every Method is Synchronized.
2	Not thread safe	2	Thread safe
3	Performance is High	3	Performance is Low
4	Null allowed for Both key and value	4	Null is not allowed
5	Introduced in 1.2 version	5	Introduced in 1.0 version

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

- How to get Synchronized version of HashMap object

```
public static Map SynchronizedMap(Map m)
    HashMap m = new HashMap();
    Map m1 = Collections.synchronizedMap(m);
```

- LinkedHashMap
  * Child class of Hashmap

	HashMap		LinkedHashMap
1	Underlaying data structure Hashtable	1	Underlaying data structure is LinkedList + Hashtable
2	Insertion is not preserved	2	Insertion is preserved.
3	Introduced in 1.2 version	3	Introduced in 1.4 version

- IdentityHashMap
  * Same as Hashmap Except the following difference.
  * In Hashmap JVM use to identify duplicate .equals() method which is meant for content comparison
    But in identityHashMap JVM will use == Operator to identify duplicates key, which is meant for reference comparison.

```
=> import java.util.*;

Class Demo {
    public static void main(String[] args) {

        HashMap m = new HashMap();
// IdentityHashMap = new IdentityHashMap();

        Integer I1 = new Integer(10);
        Integer I2 = new Integer(10);

        m.put(I1, "Ram");
        m.put(I2, "Shyam");

        SOPLN(m); //{10=Shyam}
// SOPLN(m); // => {10=Ram, 10=Shyam}
    }
}
```

WeakHashMap

It is exactly same as HashMap except the following difference.

* HashMap dominates the Garbage Collector, if object doesn't have any reference but still associated with HashMap, so it is not eligible for garbage collector
* If objects doesn't have any reference but still associated with WeakHashMap, So this object is eligible for Garbage Collector. That is Garbage Collector dominate the WeakHashMap.

MAY 2010
M T W T F S S
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

```
=> import java.util.*;

class Demo {
    public static void main(String[] args) {

        HashMap m = new HashMap()
// WeakHashMap m = new WeakHashMap()
        Temp t = new Temp()
        m.put(t, "Ram");

        SOPLN(m); // => {temp = Ram}
        t = null;
        System.gc();
        Thread.sleep(5000);

        SOPLN(m); // => {temp, Ram}
// SOPLN(m); // = { }
    }
}
                                    => {temp = Ram}
                                       finalize() called
                                       {}
class Temp {
    public String toString()
        return "temp"; }
    public void finalize() {
        SOPLN("finalize() called");
    }
}
```

• SortedMap (I)

* It is the child interface of Map
* If we want to represent a group of key-Values pairs According to some sorting order of key then we should go for SortedMap.

Methods

(1) Object firstKey()
(2) Object lastKey()
(3) SortedMap headMap(Object key)
4 SorteMap tailMap(Object key)
5. SortedMap subMap (Object key1, Object key2)
6. Comparator comparator()

• TreeMap

* The underlaying Data Structure - Red Black Tree.
* Duplicate key are not allowed but value can be duplicated.
* Insertion Order is not preserved but based on Sorting Order.
* If Sorting is Default then key Object class implements comparable and should be Homogeneous.
* If we defining our sorting then key need not be Homogeneous and not need to implements comparable.
* Null Acceptance.

SUNDAY 2

1 For Empty TreeMap we can add add one null key but then we can not add any key or get NullPointerException
2 For Non-Empty we can not add null
3 for value we can add null any no of time.

* Same as ~~SortedSet~~ Constructor available in ~~S~~ TreeMap-
TreeSet

1 TreeMap t = new Treemap()
↳ for defaul sorting Order
2 TreeMap t = new TreeMap (comparator c).
↳ Customized Sorting order

3 TreeMap t = new TreeMap (SortedMap m)
4 TreeMap t = new Treemap (Map m)

Concurrent Collection

- Need of Concurrent Collection (java.util.concurrent)

* Tradition Collection object (like ArrayList, HashMap) can be accessed by multiple threads simultaneously and there may be a chance of Data Inconsistency problems and Hence these are not thread Safe.
* Already present thread safe collections performance wise not good.
* Because of every Operation, Even for read operation one thread load whole collection objects at a time so It increases waiting time for other threads.
* Another Big problem, When one thread iterating collection, other thread not allowed to modify collection, If we trying to modify then we will get ConcurrentModificationException
* So to overcome these problems concurrent collection introduced in 1.5 version

ThreadSafe tradition collection
Vector, Hashtable
SynchronizedList
SynchronizedSet
SynchronizeMap
} — performance wise

⇒
```
import java.util.*;
class Demo extends Thread {
    Static ArrayList l = new ArrayList();
    public void run()
    {
        try { Thread.sleep(2000); }
        catch (InterruptedException e) { }
```

```
        SOPLN("child thread updating List");
            l.add("D");
    }
    public static void main(String[] args) {
            l.add("A");
            l.add("B");
            l.add("C");

        Demo t = new Demo();  // t is child thread
            t.start();

        Iterator itr = l.iterator();
        while(itr.hasNext()) {
            String s1 = (String) itr.next()
        SOPLN("Main thread iterating collection"+s1);
                Thread.sleep(3000);
        }
        SOPLN(l);
    }
```

- Difference between Concurrent collection & tradition Collection

Tradition Collection	Concurrent collection
1 Not alway Thread safe	1 Always Trade safe.
2 Performance is not good	2 Performance is good.
3 While One Thread iterating collection other thread are not allowed to modify collection otherwise get ConcurrentModificationException.	3 One thread iterating collection then other thread allowed to modify collection in safe manner, because of different locking mechanism * Block level locking * Segment level locking. * Bucket level locking.

- The important concurrent collection classes.

1. ConcurrentHashMap
2. CopyOnWriteArrayList
3. CopyOnWriteArraySet.

- ConcurrentMap(I)

Map(I)
|
ConcurrentMap(I)
|
ConcurrentHashMap c

methods.

(1) Object putIfAbsent(Object key, Object value)

```
⇒ Object putIfAbsent(Object key, Object value)
  {
      if (!map.containsKey(key) {
          map.put(key, value);
      }
      else
          return map.get(key);
  }
```

put()	putIfAbsent()
1. If the key is alread available then old value will be replaced with new value.	1. If the key is already available then Entry wont be added and return Old value object. If key is not available then it will be added.

2. boolean remove (Object key, Object value)
↳ remove the Entry if the key associated with specified value only.

3 boolean replace (Object key, Object oldValue, Object newValue)

○ ConcurrentHashMap

* Underlying data structure is Hashtable
* It allow concurrent read and Thread Safe update operation
* To perform read operation thread must require any lock but to perform update operation, thread require lock but it is the lock of particular part of map (Bucket level lock)
* ~~In concurrentHashMap, instead of whole map locking we lock partially where update operation~~
* Concurrent update achieved by internally dividing map into smaller portions, which is defined by Concurrency Level
* Default concurrency level is 16.
* CHM allows any no of read operation but 16 update operation at a time by default.
* null is not allowed for both key and value
* While one thread performing iteration operation, then other thread can perform update operation, it will not throw ConcurrentModificationException.

Constructor

1) CHM m = new CHM();
2 CHM m = new CHM(int intialCapacity)
3 CHM m = new CHM(int intialCapacity, float fillRatio)
4 CHM m = new CHM(int initialCapacity, float fillRatio, int concurrenceLevel)
5 CHM m = new CHM(Map m)

⇒

```
class MyThread extends Thread {
    public static CHM m = new CHM();
    public void run() {
        try { Thread.sleep(2000); }
        catch (InterruptException e) { }
    SOPLN("Child Thread Updating Map");
        m.put(103, "C");
    }
public static void main(String[] args) throws InterruptException {
    m.put(101, "A");
    m.put(102, "B");
    MyThread t = new MyThread();
    t.start()
    Set s = m.keySet();
    Iterator itr = s.iterator();
```



```
    while (itr.hasNext()) {
        Integer I1 = (Integer) itr.next();
SOPLN("Main Thread iterating key = " + I1 + " value = " + m.get(I1));
        Thread.Sleep(3000);
    }
}
```

Difference between HashMap and ConcurrentHashMap.

HashMap	ConcurrentHashMap
1 It is not thread safe	1 It is thread safe.
2 Performance is high because threads are not require to wait	2 Relatively Performance is low because some time threads are required to wait.
3 While one thread iterating map other threads are not allowed to perform update operation. otherwise we will get RE ConcurrentModificationException	3 While one thread iterating CHM other threads are allowed to modify CHM. in safe manner id wont throw → RuntimeException
4 null is allowed for both keys and value	4 null is not allowed for both keys and value.
5 Iterator of HashMap is fail-fast and it throw CME	5 Iterator of CHM is fail-safe and it wont throw → CME
6 Introduce in 1.2 version	6 introduce in 1.5 version

* Fail-fast iterator -: When one thread iterating HashMap then other thread perform any modification, ~~the~~ At that time iterator imidiatly throw exceptin CME, and iterator fails this type of iterator called fail-fast iterator

Exception Handling

Exception

A unwanted unexpected ~~that~~ event that disturb normal flow of program called exception.

For normal termination of program requir'd Exception Handling. It doesn't mean repairing an exception.

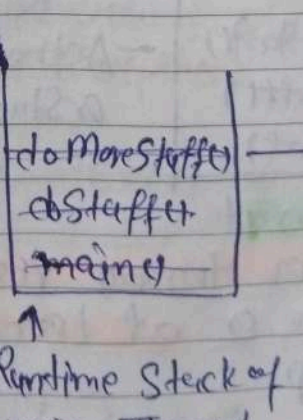
Exception Handling means, we have to define alternative way to continue rest of program normally.

exp
- Read data from Remote file locate at London
- If the locate file is not available
- then read local file
- Use that file

```
try {
    Read data from London file
} catch (FileNotFoundException e)
{
    Use local file and continue
    rest of program normally
}
```

Runtime stack mechanism

```
class Test {
public static void main(String[] args){
        doStuff();
    }
public static void doStuff(){
        doMoreStuff();
    }
public static void doMoreStuff(){
        SOPLN("Hello");
    }
}
```



Default Exception Handling

```
class Test {
public static void main(String[]args){
        doStuff();
}
public static void doStuff(){
        doMoreStuff();
}
public static void doMoreStuff(){
    // SOPLN("Hello");
    SOPLN(1010);
}
```

Called by JVM ← JVM has Default Exception Handler throw Exception Object

Does Have Handler? Not terminate

Who call this method

does you have handler for exception

Who call this method

↳ JVM ask do you have any exception Handling if not JVM terminated

← Exception Raised & create exception object

~~doMoreStuff();~~
~~doStuff()~~
~~main()~~

— Activation Record or Stack Frame

Exception Format

exception in thread main : Exception Name : Discription
stack trace or at Location

exp

exception in thread main : AE : Divide by zero
at ~~at~~ at doMoreStuff
at ~~at~~ at doStuff
at ~~at~~ at main

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
27 28 29 30

Exception Hierarchy

Throwable
- Exception
- Error

Exception	Error
1 Caused by programmer	1 Caused by lack of System resources
2 Recoverable ↳ we can handle the exception	2 Non-Recoverable exp virus attack ~~then~~ and JVM System crashed then we have to start again

Exception
- RuntimeException
  - ArithmeticException
  - NullPointerException
  - classCastException
  - IndexOutOfBoundsException
    - ArrayIndexOutOfBoundsException
    - StringIndexOutOfBoundsException
  - IllegalArgumentException
    - NumberFormatException
- IOException
  - EOFException
  - FileNotFoundException
- ~~IntruptException~~ InturptedException
- RemoteException
- ...

Error
- VirtualMachinError
- StackOverflowError
- OutOfMemoryError
- LinkageError
  - VerifyError
- AssertionError
- ExceptionInInitializerError

Checked vs UnChecked Exception

* The exception which are checked by the compiler whether programmer handle or not, for smooth execution of program at runtime, are called checked exception

```
=> import java.io.*;
   class Test {
        public static void main(String[] args) {

            PrintWriter pw = new PrintWriter("abc.txt");
            pw.println("Hell");

   // Hence compiler will check that you provide any
   // Handler for FileNotFoundException, because may be
   // file "abc.text" is available or not and it give
   // compiltime error
   }
```

* Rarely occured exception which are not checked by the compiler whether programmer handling or not, are called unchecke exception
exp Arithmetic Exception, all Runtime exceptions and its classes and Error and its child classes, All ~~customized exception~~

* There are three method to print exception
Throwable class contain this method.
1 e.printStackTrace() → full details of exception
2 e.getMessage() → For discription
3 e.toString → Name of exception.

• Throw keyword
Some times we have to create Exceptian object and we we have to handover to the JVM manually by using Throw keyword. In general we use Throw key word for customized exceptions but not for predefined exceptions.

exp
```
withdraw (double amount) {
    if (amount > balance)
    {
        throw new InSufficiantFundException;
    }
}
```

Throws keyword

In our program if there is any chance of raising checked exception then compulsory we should handle either by try catch or by throws keyword otherwise the code wont compile.

Notes:

* The main objective of "throws" keyword is to delegate the responsibility of exception handling to caller method.
* throws keyword required for only checked exception
* throw keyword is required to convince compilor. It does not prevent abnormal termination of the program.

* Thread *

Multitasking

Executing several tasks simultaneously is called multitasking. exp. class student.

There are two types of multitasking

1 Process-based multitasking
2 Thread-based multitasking

(1) Process based multitasking

Executing several tasks simultaneously, where each task ~~is~~ have seperate process ~~an~~ ~~and~~ or every task is seperate program. is called process based multitasking

exp while

1 Typing code in text editor
2 At that time listining songs
3 Some downloading file

} Process based multitasking

*this best suitable at OS level

(2) Thread-based multitasking

Executing several tasks simultaneously where each task is ~~part of~~ seperate independent part of the same program. is called thread based multitasking. and each independent part called thread. it is best suitable at programatic level

exp

Here thousand line of program take n time to execute but this program divided into two seperate independent ~~time~~ program so two thread will execute this program simultaneously and decrease execution time

thousand line of program

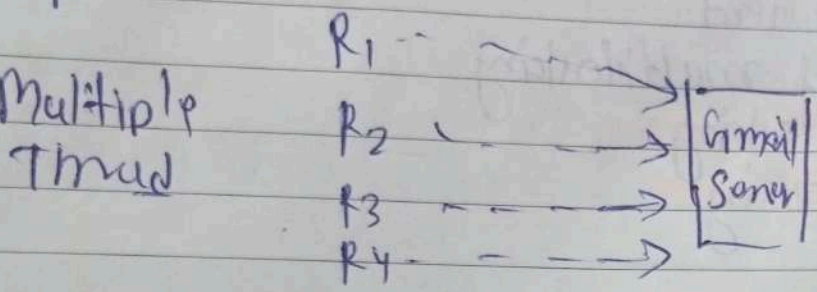
} ← thread 1
..... dividing
} - thread 2 program into two independent programs

Multitasking, reduces responce time of system and improved performance of system

JUNE 2010
M T W T F S S
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

* We used multithreading to developed web/application server

exp



⇒ Handling by using multithreading concept.

* Tomcat can recieve by defaut request = 60,

Thread :- ~~Independent flow~~, seperate flow of flow of execution, independent job, light weight process, seperate flow of execution, if there is only main thread, called single thread operation
For every thread there is a job

* There are two way to defin Thread

o Defining a Thread

1 By extending thread class
2 By implementing Runable interface.

o By extending Thread class

⇒
```
class MyThread extends Thread {
    public void run() {
        for(int i=0, i<10, ++i){
            SOPLN("Child thread");
        }
    }
}
```

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

```
class ThreadDemo {
    public static void main(String[] args) {

        MyThread t = new MyThread();
          t.start();
        for (int i = 0; i < 10; ++i){
            System.out.println("Main thread");
        }
    }
}
```

- Thread Scheduler

If there are many thread, then which order thread will be execute, first or last is done by thread scheduler. JVM can't expect exact algorithm followed by thread scheduler, it is varied from JVM to JVM

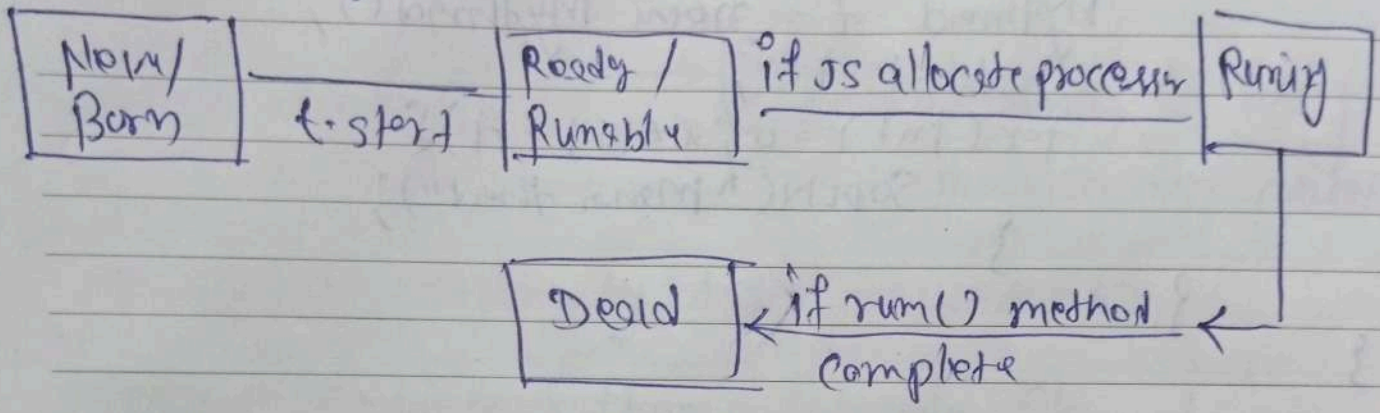
- t.start() it will start a child thread and multithread which is responsible to execute run() method
  t.run() → it is just normal behavior of calling method on object

Inside start() {
1. Register this thread with thread scheduler
2. Performs all other mandatory activities
3. Invoke run()
}

start() method always called No arg run() method.

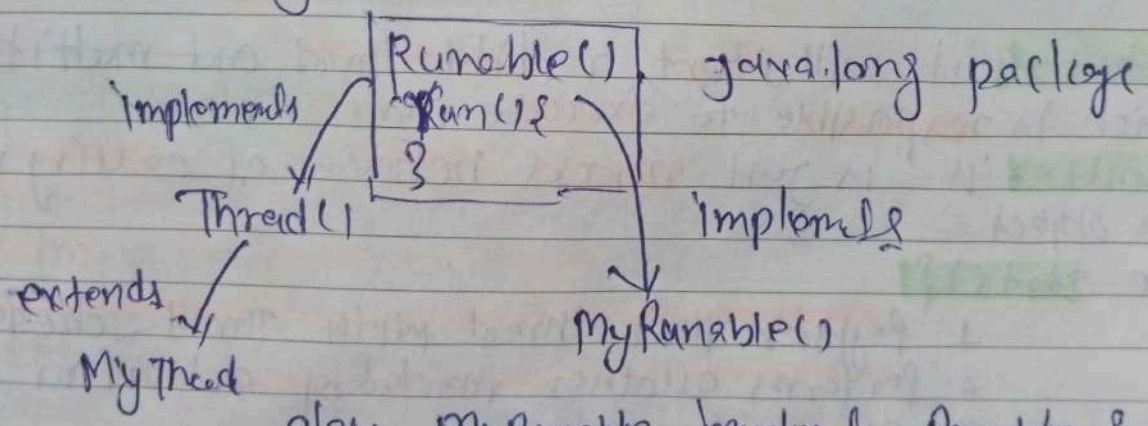
① Basic life cycle of thread.

MyThread t = new MyThread()



* After start a thread if we are trying to restart same thread then we will get RE → IllegalThreadStateException.

② By Implementing Runable interface :



⇒

```
class MyRunable implements Runable {
    public void run() {
        for(int i = 0; i<10; ++i) {
            SOPLN("child class");
        }
    }
}
```

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

```
class ThreadDemo {

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

        MyRunable r = new MyRunable();
        Thread t = new Thread(r);
        t.start();
        for (int i=0; i<10; i++) {
            SOPLN("Main Thread");
        }
    }
}
```

Q Which approch is best to define Thread.

⇒ Implements runable approch is recommended becaun in Thread our class is extended, so ~~we cant i~~ there is no chance of extending other classes hence we are missing inhertance benfit but in Runable interface we can extend any other class.

Java Code

- String Code

- Reverse String

There are four way to reverse string

1 Using toCharArray()
2 Using charAt(int index)
3 In StringBuffer class reverse()
4. In StringBuilde class reverse().

String str = "RAHUL"

1 ⇒ 0 1 2 3 4
R A H U L

```
int length = str.length()
char[] chArr = str.toCharArray();

for (int i=0; i<chArr.length-1; +

for (int i= chArr.length-1; i>=0 ; --i) {

        SOPLN( chArr[i]);
}
```

2 ⇒

```
~~int l~~ = for (int i= str.length()-1; i>=0; --i) {
                SOPLN( charAt(i));
            }
```

3 ⇒

```
StringBuffer sb = new StringBuffer(str)
sb.reverse();
SOPLN (sb);
```

- Write a program to remove all all special characters from a given string.
  input : $jav%va$tst%
  ouput : javastr

⇒ Regular Expression = "[^a-zA-Z0-9]"
  use to show ^ ↵
  is anything other then mention character

Here we used replaceAll() by using regex

str.replaceAll("[^a-zA-Z0-9]", "");
↳ Empty

= To remove white space (\s)

str.trim() ⇒ use to remove space before and After string

str.replaceAll("\\s", "");

- Remove duplicate character in given string.
  1 Using Java 8
  2 Using indexOf() in String class
  3 using character Array
  4 using set interface method.

1 Java 8

```
StringBuffer sb = new StringBuffer()

// str.chars() → return input stream
// distinct() → predefined method use to ~~return~~ prevent duplicates

str.chars().distinct().forEach(c -> sb.append((char) c));
```

2 indexOf()

```
StringBuffer sb = new StringBuffer()
for(int i=0; i<str.length(); ++i) {
char ch = str.charAt(i);
int idx = str.indexOf(ch, i+1) // from i+1 find ch in string
  if (idx==-1)
        sb.append(ch);
}
```

3 char[] arr

```
char[] arr = str.toCharArray();
~~for~~ StringBuffer sb = new StringBuffer();

for (int i=0; i<~~str~~.length; ++i) {
        boolean flag = true
        for (int j = i+1; j<arr.length; j++) {

                if (arr[i]==arr[j]) {
                        flag = false;
                        break;
                }
        }  if (flag)
        sb.append(arr[i]);
}
```

JUNE 2010
M T W T F S S
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26
28 29 30

4 Set

```
StringBuffer sb = new StringBuffer()
Set<Character> set = new LinkedHashSet()

for(int i=0; i<str.length; ++i){
        set.add(str.charAt(i));
    }
for(Character c : set){
        sb.append(c);
    }
```

- String Sorting

1 Without using sorting method

```
String str = "rock";
char[] arr = str.toCharArray();
char temp;
for(int i=0; i<arr.length; ++i){
        for(int j=i+1; j<arr.length; ++j){
                if(arr[i] > arr[j]){
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
        }
}
```

2 With using sort() method

```
char[] arr = str.toCharArray();
Arrays.sort(arr)
```

APRIL 2010
M T W T F S S
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

- Replace character with its Occurance in String

⇒
```
String str = "openterext";
char charToReplace = 't';
if( str.indexOf(charToReplace) == -1) {
    SOPLN("char not found")
    System.exit(0);
}
int count = 1;
for (int i = 0; i < str.length(); ++i) {

    if( charToReplace == str.charAt(i)) {
    str = str.replaceFirst( StringValueOf(charToReplace), String.
                                valueOf(count));
    ++count;
}
SOPLN(str);
```

- To find non repeated string character in string
1- without collection.

⇒
```
String str = "aabbcdeff";
StringBuffer sb = new StringBuffer();
    for(int i = 0; i < str.length(); ++i){
        boolean unique = true;
        for (int j = 0; j < str.length(); ++j){
            if (i != j && str.charAt(i) == str.charAt(j)) {
                unique = false;
                break;
            }
        if (unique) {
            sb.append(str.charAt(i));
        }   // break; // -> to find first unique character
```

- To print the occurance of character in string.

```
Map<character, Integer> map = new HashMap();
for (int i = 0 ; i < str.length(); ++i) {

    if ( map.containsKey (ch)) {
        map.put (ch, map.get (ch) + 1) ;
    else
        map.put (ch, 1);
}
SOPLN (map);
```

- Reverse each word in String.

```
=> String str = "Rahul chouhan";
   String[] words = str.split(" ");
   String output = "";

   for ( String word : words) { String revWord = "";
       for ( int i = word.length() - 1; i >= 0 ; --i)

           revWord = revWord + word.charAt(i);
       }
       output = output + revWord + " ";
   }
   SOPLN (output);
```

- longest substring, not repeating character in String
- Palindrome
  - String.

* Convert to array to ArrayList

```
String[] str = { "Red", "Blue", "white" }
List al = Arrays.asList(str)
```