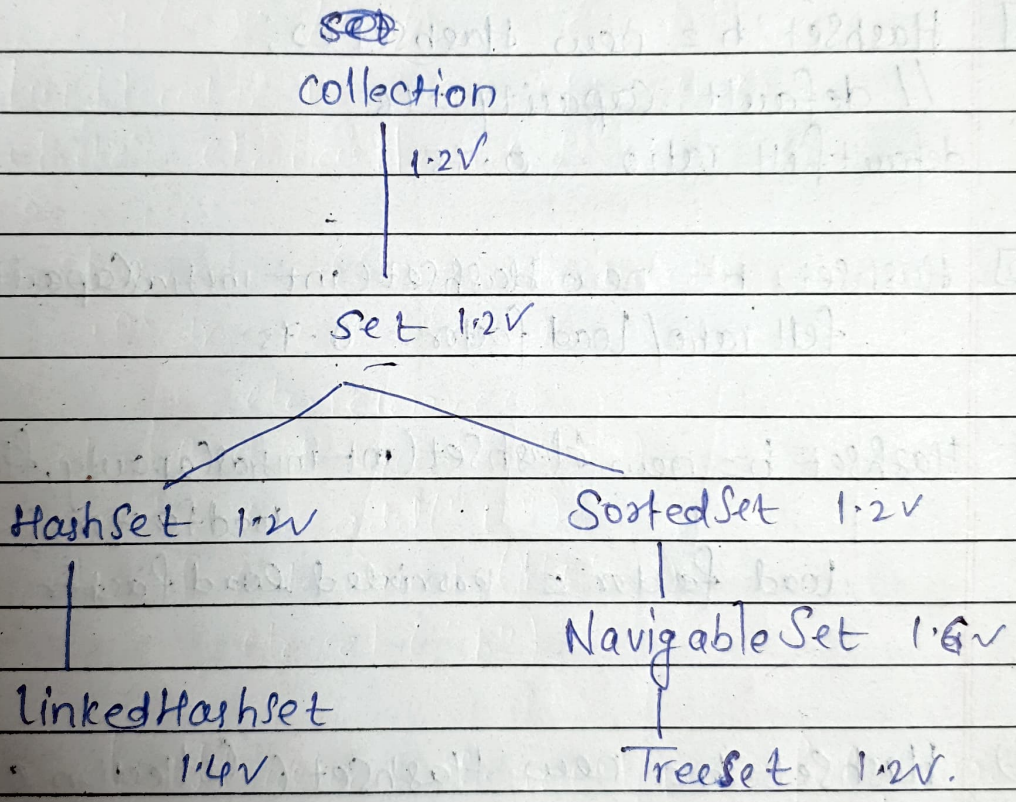
HashSet

* Set
→ child interface of collection
→ Duplicates are not allowed.
→ Insertion order not preserved.
→ it doesn't contain any new methods.
So we have to use only Collection Interface Methods.



HashSet

→ Underlying DS is HashTable.
→ Duplicates – x
→ Insertion Order – Not preserved.
→ Heterogenous – ✓
→ null insertion – ✓
→ Implements – Serializable & clonable.
→ Search operation – Best choice

Constructors

1] HashSet h = new HashSet();
// default capacity = 16
default fill ratio – 0.75.

2] HashSet h = new HashSet(int initialCapacity);
fill ratio / load factor = 0.75

3] HashSet h = new HashSet(int initialCapacity, float loadfactor);
load factor = provided load factor.

(4) HashSet h = new HashSet(Collection c);

Load factor / fill Ratio:
After loading how much factor a new HashSet object will be created, that factor is called as load factor or fill Ratio.

Page No.
Date

LinkedHashSet → 1.4v

* child class of HashSet
* Duplicates are not allowed.
* Insertion order - Preserved.
* Underlying DS - HashTable + LinkedList.

LinkedHashSet is best choice to develop cache based applications, where duplicates are not allowed and Insertion order preserved.

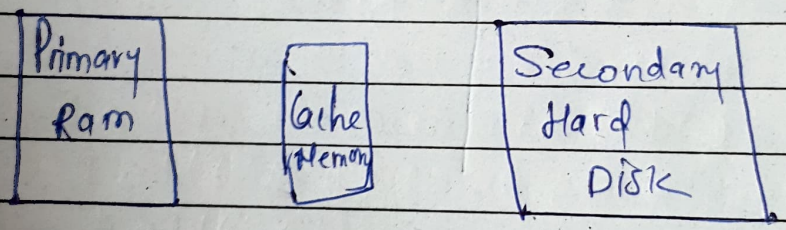
	HashSet	LinkedHashSet
(i)	HashTable	LinkedList + HashTable.
(ii)	1.2v	1.4v
(iii)	Insertion order Not preserved	preserved
(iv)		

```
Class HashSetDemo
{ p s v m (String[] args)
  {
     HashSet h = new HashSet();
     h.add("B");
     h.add("C");
     h.add("D");
     h.add("z");
     h.add(null);
     h.add(10);
  SOP( h.add("z")); // false
```

If Insertion Order is necessary then go for LinkedHashSet.

```
class LinkedHashSetDemo{
    p s v m(String[] args)
{
    LinkedHashSet h = new LinkedHashSet();
    h.add("A");
    h.add("B");
    h.add("c");
    h.add(null);
    sop(h); // A, B c null
    sop(h.add("A")); // false.
```

Cache Baied Applications



Page No.
Date

SortedSet

* Duplicate Not Allowed
* Insertion according to some sorting order.
* child Interface of Set Interface.
* Came in 1.2V

SortedSet Specific Methods

(i) first() - 100
(ii) last() - 115
(iii) headSet(104) - 100 101 103 (element less than Passed element)
(iv) tailSet(104) - 104, 107, 110, 115 (>= 104)
(v) subSet(103, 110) - 103, 104, 107 (>= 103 && < 110)
(vi) comparator() → null (for default Sorting order)
returns Comparator Object that describes Underlying Sorting Technique.

100 101 103 104 107 110 115

Default Sorting Order
for Numbers - Asc
for Strings - Alphabetic Order.

TreeSet

* Implementation class for SortedSet Interface.
* Came in 1.4V.
* Underlying DS - Balanced Tree.
* Duplicates are not allowed.
* Insertion order not preserved.
* Insertion takes place according to Some Sorted order.
* Homogenous Object are allowed only.
Heterogenous object - Not Allowed.
* null insertion allowed only once.

A-65
a-97

TreeSet Constructors

(1) TreeSet t = new TreeSet();
- Inserted into Natural Sorting order.

(2) TreeSet t = new TreeSet (Comparator c)
- Insertion according to Sorted order C

(3) TreeSet t = new TreeSet (Collection c).

4] TreeSet t = new TreeSet (SortedSet s).

ex:-

```
class TreeSetDemo
{
  p s v m (String[] args)
  {
    TreeSet t = new TreeSet();
    t.add("A");
    t.add("C");
    t.add("D");
    t.add("B");
    sop(t); // A B C D.
    sop(t.add(A)); // false
    t.add(new Integer(10)); // Class Cast Exception
    // Heterogenous object not allowed.
    t.add(null); // NullPointer Exception
    // bcoz for non-empty Treeset, null insertion
    // not allowed
    // for empty Treeset, we can Insert null
  }
```

Null Acceptance:

(1) for empty TreeSet as the first element null insertion is possible but After Inserting that null if we are trying to insert any another element, we will get NullPointerException.

(2) for Non-empty, TreeSet, If we are trying to insert Null, we will get NullPointerException.

demo Program

```
class Demo{
  P S V m(String[] args)
  {
    TreeSet t = new TreeSet();
    t.add(new StringBuffer("A"));
    t.add(new StringBuffer("B"));
     sop(t);
}}
```

// we will get ClassCastException saying StringBuffer Object is not Comparable.

If we are depending on natural sorting order, Then Object compulsory be
(1) Homogenous
(2) Comparable.
In demo program, all objects are of StringBuffer i.e. Homogeneous but StringBuffer object is not Comparable

** we can say a object is comparable if and only if Object Implements Comparable Interface.

Comparable Interface

* present in java.lang package.
* contain only one method.
  compareTo().

public int compareTo(Object obj)

obj1.compareTo(obj2)
- returns -ve iff obj1 comes before obj2
- returns +ve " " " After "
- returns 0 " obj1 & obj2 are equal.

ex:-

```
sop("A".compareTo("Z")); //-ve
sop("Z".compareTo("B")); //+ve
sop("A".compareTo("A")); //0
sop("A".compareTo(null)); //RE:NPE
```

* If we depending on default natural sorting order internally JVM will call compareTo() method while inserting objects to TreeSet.
Hence objects should be Comparable.

* If we are not satisfied with default natural sorting order or if the default natural sorting not already available then we can define our own customized sorting by using Comparator.

* Comparble Meant - Natural sorting order.
* Comparator " -> Customized " "

Comparator Interface

* Meant for Customized Sorting Order.
* java.util package.
* Contains two method.!-
  (i) compare() (ii) equals().

(i) public int compare(Object obj1, Object obj2)
- → -ve obj1 before obj2
- → +ve " After "
- → 0ve obj1 & obj2 are equal.

(ii) public boolean equals().

```
class myComparator Implements Comparator
{
    public int compare()
    {
        ---// we should Compulsory provide Implementation
    }       // for compare() method.
  // equals() implementation is optional.
  // bcoz object class equals() method already available.
}.
```

```
class
public myComparator implements Comparator
{
    public int Compare(Object obj1, Object obj2)
    {
        Integer I1 = (Integer) obj2;
        Integer I2 = (Integer) obj2;
        if(I1<I2) return 1;
        else if(I1>I2) return -1;
        else return 0;
    }
}
class demo {
    psvm(String[] args)
    {
        TreeSet t = new TreeSet(new myComparator());
        t.add(10);
        t.add(0);
        t.add(5);
        t.add(20);
        Sop(t);
    }
}
// 20, 10, 5, 0

// p.s compare(Object obj1, Object obj2)
{
    Integer I1 = (Integer) obj1;
    Integer I2 = (Integer) obj2;
    return I1.CompareTo(I2); // Asce
    return -I1.CompareTo(I2); // Des
    return I2.CompareTo(I2); // Des
```

```
return -I2.compareTo(I1); // Asce
return +1; // Insertion order preserved and duplicate Allowed.
return -1; // Reverse of Insertion order and duplicate Allowed.
return 0;  // only first element inserted and all other
           // elements are considered as duplicates.
```

* Note :- If we are defining our own sorting by Comparator, the objects need not be comparable.

** If we are depending upon natural sorting order. Then comparing Objects must be
(i) Homogeneous
(ii) Comparable.

But if we are defining our own customized sorting order using Comparator, then comparing Objects need not be:
(i) Homogeneous
(ii) Comparable.
We can insert heterogeneous non-comparable objects also.

Comparable vs Comparator

(I) for predefined Comparable classes like String, default natural sorting order already available if we are not satisfied with that, we can define our own sorting by Comparator Object.

(II) for predefined non-Comparable classes like StringBuffer, default natural sorting is not already available. we can define required sorting by Implementing Comparator Interface.

(III) for our own classes like Employee Student etc. the person who is writing our own class, is responsible to define default natural sorting order by Implementing Comparable Interface.

The person who is using our class, if he is not satisfied with default natural sorting order, then he can define his own sorting by Using Comparator interface.

Page No.
Date

Demo program for Customized sorting for Employee class:

```
import java.util.*;
class Employee implements Comparable
{
  String name; int eid;
 Employee(String name, int eid){
      this.name = name;
      this.eid = eid;
 }
public String toString()
 {
   return name + "-" + eid;
 }
public int compareTo(Object obj)
 {
   int eid1 = this.eid;
  Employee e = (Employee) obj;
  int eid2 = e.eid;
  if(eid1 < eid2) return -1;
  else if(eid > eid2) return 1;
  else return 0;
 }
}

class CompCompDemo
{
  p s v m(String[] args)
  {
    Employee e1 = new Employee("nag", 100);
    Employee e2 = new Employee("balaih", 200);
    Employee e3 = new Employee("chiru", 50);
```

```
Employee e4 = new Employee ("venki", 150);
Employee e5 = new Employee ("nag", 100);
TreeSet t = new TreeSet();
t.add(e1);
t.add(e2);
t.add(e3);
t.add(e4);
t.add(e5);
sop(t);
// prints elements sorted in natural sorting order
according to eId.
// chiru-50, nag-100, venki-150, balaih-200

TreeSet t1 = new TreeSet(new MyComparator());
t1.add(e1);
t1.add(e2);
t1.add(e3);
t1.add(e4);
t1.add(e5);
sop(t1);
// prints according to customized sorting order.
}    // according to name in alphabetical order.
}

class MyComparator implements Comparator
{p i compare(Object obj1, Object obj2) {
   Employee e1 = (Employee) obj1;
   Employee e2 = (Employee) obj2;
   String s1 = e1.name;
   String s2 = e2.name;
   return s1.compareTo(s2); // customized
                            sorting
}
}
```

String

→ java.lang.String

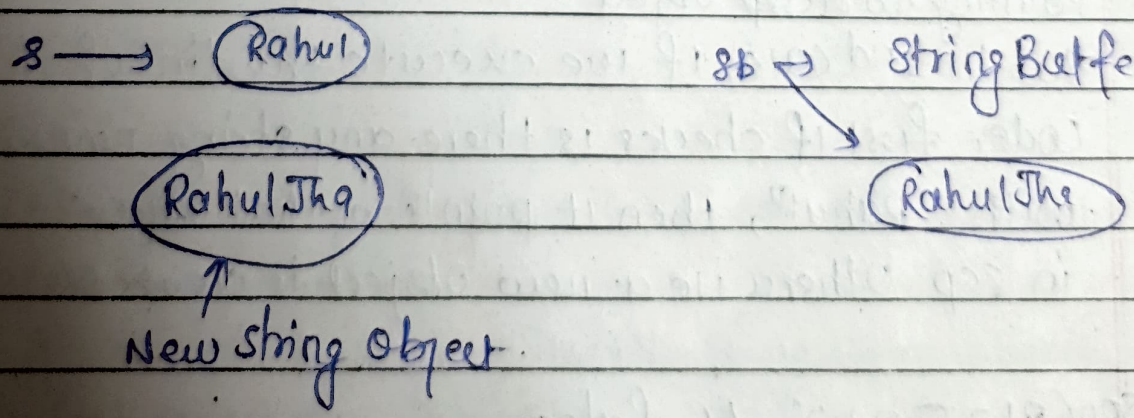
- Present in java.lang Package.
- Most commonly used data type / Object in Java.

```
String s = new String("Rahul");
s.concat("Jha");
sop(s); // Rahul
```

- Once we create a string object, we can't perform any changes in existing object
  If we are trying to perform any changes with those changes a new object will be created.
  This Non-changeble is nothing but Immutability of string.

```
StringBuffer sb = new SB("Rahul");
sb.append("Jha");
sop(sb); // Rahul Jha.
```

→ Once we create a SB object, we can perform any changes on existing object. This changable behaviour is nothing but mutability.



(2)

```
String s1 = new String("Rahul");
String s2 = new String("Rahul");
sop(s1==s2); // false
sop(s1.equals(s2)); // true

SB sb1 = new SB("Rahul");
SB sb2 = new SB("Rahul");
sop(sb1==sb2) // false
sop(sb1.equals(sb2)) // false
```

In case of SB, object class equals method is executed. But Object class equals method meant for Reference comparison.
But, string class equals method meant for content comparison. bcoz equals method is overriden in string.

Case3]

```
String s1 = new String("Rahul");
string s2 = "Rahul";
```

In first case, string object is created in Heap Memory and scp (string constant pool) but Pointing to Heap object.
But in 2nd case, if we execute this line of code, first if cheeks is there any string named with "Rahul", then it points to that object in scp otherwise a new object is created.

```
Sop(s1==s2); // false
sop(s1.equals(s2)); // true
```

* Garbage Collector is not allowed for accessing SCP.

SCP → Heap, Method, stack, Native Method stacks, pc registers

→ GB applicable for Heap Area only.

→ All SCP objects are destroyed when JVM is stopped. At production level, server restart will be there.

```
String s1 = new String("Rahul");
   s1.concat("Jha");
String s2 = s1.concat("kumar");
s1 = s1.concat("J");
sop(s1); // RahulJ
sop(s2); // Rahulkumar.
```

	Heap	SCP
	s_1 ~~→~~ Rahul	Rahul
due to Runtime	→ RahulJha	Jha
creation of Obj	s_2 → Rahulkumar	kumar
	s_1 → RahulJ	J

Total Heap Object = 4
" SCP object = 4.
8

Note :- for every string constant, one object will be placed in SCP Area.

(iv) Because of some Runtime Operation, if an object is required to create, that object will be placed only in the Heap Area But not in SCP Area.

Page No.
Date

String class

Constructor

(1) String s = new String();
An empty string will be created with length = 0.

(2) String s = new String(string Literal);
create a string with given Literal.
in Heap as well as scp.

(3) String s = new String(StringBuffer sb);
create string of StringBuffer Object sb.

(4) for given CharArray.
String s = new String(char[] cb);
ex:-
char cb[] = {'a','b','c','d'};
String s = new String(cb);
sop(s); // "abcd"

(5) for given byte Array.
String s = new String(byte[] b);
ex:-
byte[] b = {100,101,102,103};
String s = new String(b);
sop(s); // defg

(6)

Important Methods of String class

→ charAt(int idx);

```
String s = "Rahul";
sop(s.charAt(3)); // u
sop(s.charAt(10)); // SIOOBE
```

→ pubic String concat(String s)

The overloaded + and += operators also meant for concatenation purpose only.

```
String s = "Rahul";
s = s.concat("Jha");
// s = s + "Softwaare";
// s += "Jha";
sop(s); // Rahul Jha.
```

→ public boolean equals(Object o)

To perform content comparison where case is important.

Overriding verson of Object class equals method.

→ public boolean equalsIgnoreCase(String s)

Case is not Important.

→ substring(int begin, int end);

It starts from start index but neglect end.

Include start but exclude end.

```
String s = "Rahul";
sop(s.subString(1,4)); // ahu
sop(s.subString(2)); // hul
```

begin index to end-1 index

→ length() - Method but Not length Variable.

→ public String replace(char old, char new)
String s = "ababa";
sop(s.replace('a', 'b')); // bbbbb

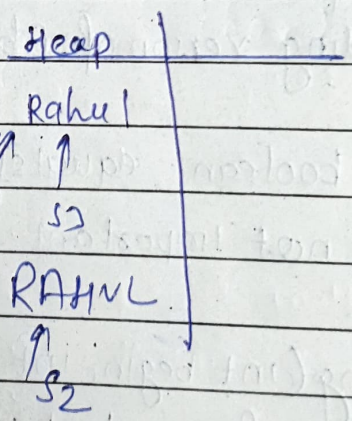
→ public String toUpperCase();
→ " " toLowerCase();

→ public String trim();
Remove all blank spaces at begining and ending.

→ indexOf(char c)
lastIndexOf(char c)

→ S1 = "Rahul";
S2 = S1.toUpperCase();
S3 = S1.toLowerCase();

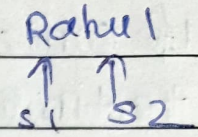
sop(S1 == S2); // f
sop(S1 == S3); // T



At Runtime if there is any change in original Content then only a new object created else existing object is reused.

String s1 = "Rahul";
String s2 = s1.toString();

Heap | SCP



* How to create our own Immutable class?

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

---

final vs Immutability

↑ for variable

↑ for objects

StringBuffer

* If content is not fixed and changes over time to time, it is advisable to go for StringBuffer. All required changes will be performed in the existing object only.

Constructors :-

(1) SB sb = new SB();
default initial Capacity = 16.
New Capacity = (oldCapacity+1)*2
(2) SB sb = new SB(int initialCapacity);
(3) SB sb = new SB(String s);

↑

Capacity = s.length() + 16

Imp Methods SB

(i) append(string s) // Applicable for int, boolean, long, char)
concat the paired arguments.
At last appended

(ii) setCharAt(int idx, char ch);
(iii) insert(int idx, String s);
(int idx, int i);
(int idx, double d);
- - - -
(iv) delete(int begin, int end)
begin to end-1.
(v) deleteCharAt(int idx)
(vi) reverse();

(VI) setLength(int length);
(VII) ensureCapacity(int capacity); // to increase capacity.
(VIII) trimToSize(); // to deallocate extra allocated space Memory

Every Method present in SB is synchronized.
Hence only one Thread is allowed to operate on SB Object at a time.

* StringBuilder
Content is not fixed. it changing frequently.
But I don't want Thread safety.
Multiple Thread should be access to the Object Simultaneously.
Then we should go for StringBuilder class.

→ Introduced in 1.5 version.
→ Non-synchronized StringBuffer version is nothing but StringBuilder.
→ StringBuilder is exactly same as StringBuffer except the following differences:-

StringBuffer	StringBuilder
(i) All methods are synchronized.	Non-synchronized.
(ii) Thread safe.	Not Thread-Safe.
(iii) performance decreases.	Performance increases relatively.
(iv) 1.0 version	1.5 version.

	String	StringBuffer	StringBuilder
(I)	Immutable ✓	✗	✗
(II)	Thread-Safe ✗	✓	✗
(IV)	Thread-Safety Not required ✗	✗	✓