MODULE IV
ADVANCED FEATURES OF JAVA

Syllabus

- Java Library -String Handling – String Constructors, String Length, Special String Operations - Character Extraction, String Comparison, Searching Strings, Modifying Strings, using valueOf(), Comparison of StringBuffer and String.
- Collections framework - Collections overview, Collections Interfaces- Collection Interface, List Interface.
- Collections Class – ArrayList class. Accessing a Collection via an Iterator.

Java Library

- The java programming supports different type of classes as User defines class [A class which is created by user is known as user defined class.] and also there are some classes available with java system that provide some important support to the java programmer for developing their programming logic as well as their programming architecture with very smooth and very fine way. These classes are called **Library Classes**.

String Class

- **String** is a class in **Java's class library**.

- In Java, string is an object that can be represented with a sequence of char values.
- An array of characters works same as Java string.
  - Eg: char[] ch={'h','e','l','l','o'};

    String s= new String(ch);

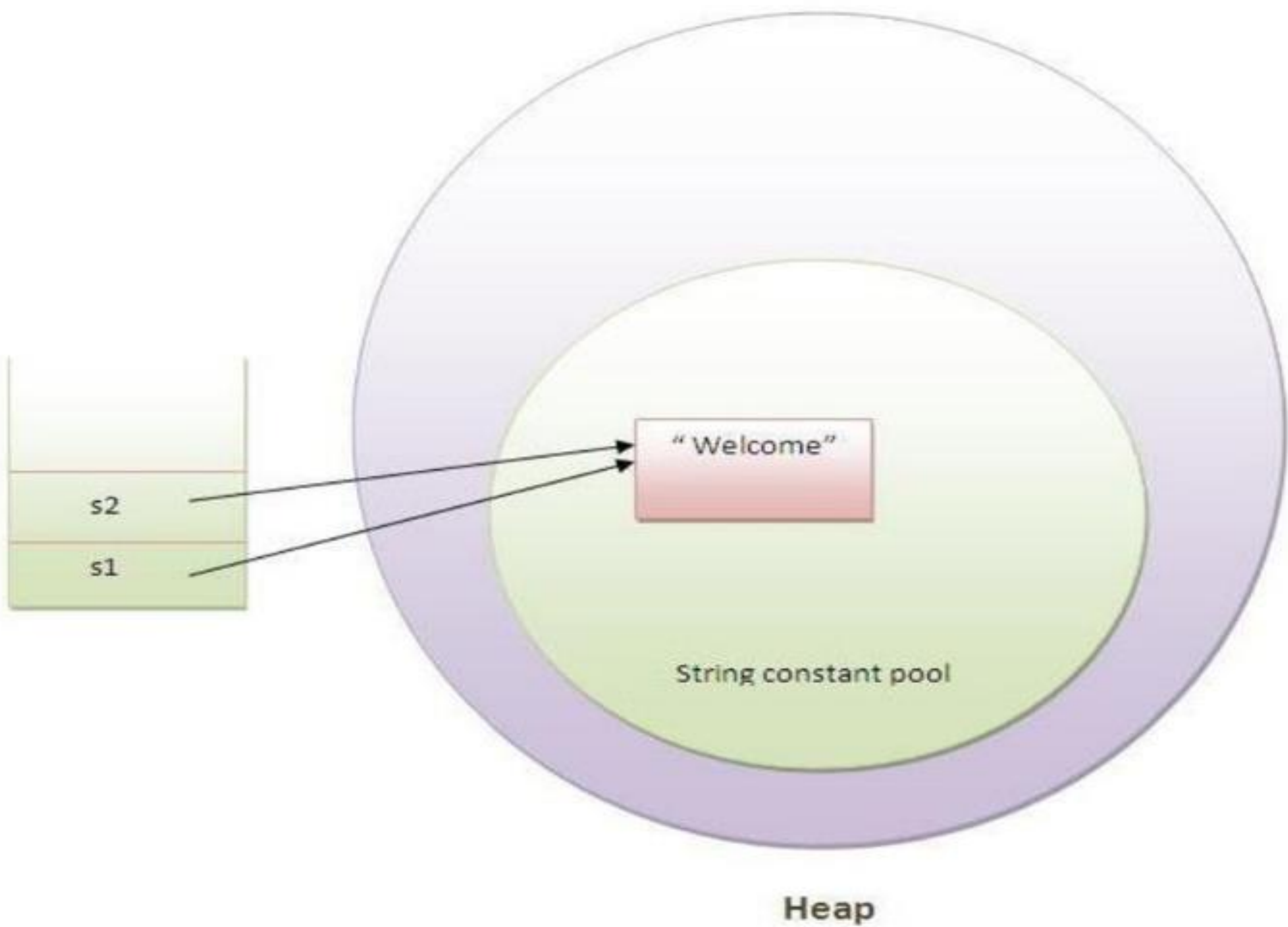
  is same as **String** s="hello";
- Java String class provides a lot of methods to perform operations on strings. Some of them are length(), equals(), concat(), split(), compareTo(), replace(), substring() etc

How to create String object?

- There are two ways to create String object:
  - **By string literal** Eg: **String s = “Hello”;**
  - **By new keyword** Eg: **String s= new String(“Hello”);**

1. By String literal

- String literal is created by double quote. For Example:
  String s="Hello";
- Each time you create a string literal, the JVM checks the string constant pool first. **String constant pool** is a temporary memory which never stores any duplicate strings. **Heap memory** stores all java objects.
- If the string already exists in the pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object instantiates, then is placed in the pool. For example:
  - **String** s1="Welcome";
  - **String** s2="Welcome" ; // no new object will be created


s2
s1
" Welcome"
String constant pool
Heap

- In the above example only one object will be created. First time JVM will find no string object with the name "Welcome" in string constant pool, so it will create a new object. Second time it will find the string with the name "Welcome" in string constant pool,so it will not create new object whether will return the reference to the same instance.
- Note: String objects are stored in a special memory area known as string constant pool inside the Heap memory.

- Why java uses concept of string literal?
  - To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

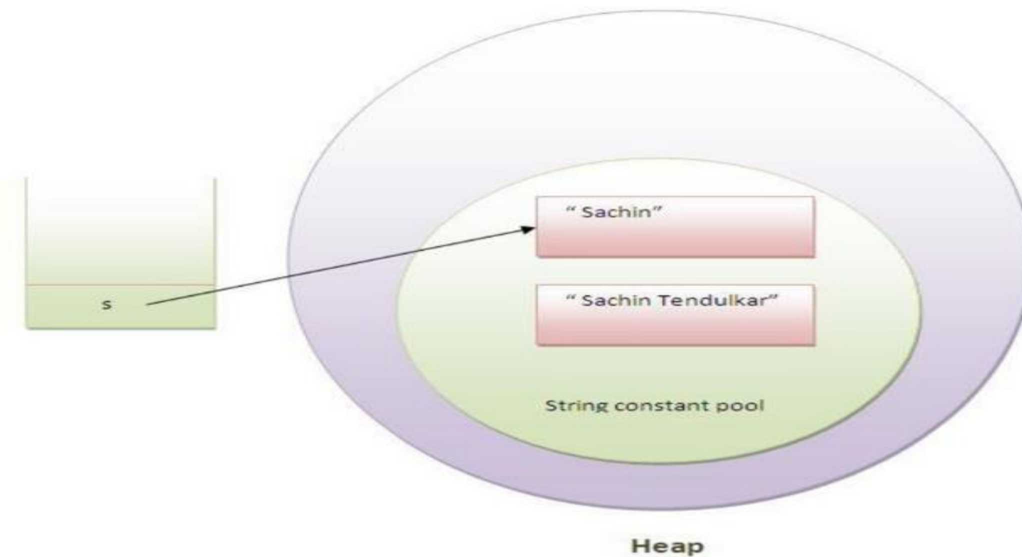
2. By new keyword

- String s=new String("Welcome");//creates two objects and one reference variable .
- In such case, JVM will create a new String object in Heap memory and the literal "Welcome" will be placed in the String constant pool.
- The variable s will refer to the object in Heap memory.

Immutable String in Java

- In java, string objects are **immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.
- **Example**

```
class Simple{
public static void main(String args[]){
String s="Sachin";
 s.concat(" Tendulkar"); //concat() method appends the string
                          //at the end
 System.out.println(s);//will print Sachin because strings are
        } }  //o/p  Sachin                       immutable objects
```



- But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

```
class Simple{
 public static void main(String args[]) {
String s="Sachin";
  s = s.concat(" Tendulkar");
System.out.println(s);   }  }    // o/p  SachinTendulkar
```

Why string objects are immutable in java

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

STRING CONSTRUCTORS

- The string class supports several types of constructors in Java APIs. The most commonly used constructors of String class are as follows:

1. **String() :** To create an empty String, we will call a default constructor.

   For example: String s = new String();

- It will create a string object in the heap area with no value

2. **String(String str) :** It will create a string object in the heap area and stores the given value in it.For example:

```
String s2 = new String("Hello Java");
```

Now, the object contains Hello Java.

3. **String(char chars[ ]) :** It will create a string object and stores the array of characters in it. For example:

```
char chars[ ] = { 'a', 'b', 'c', 'd' };
String s3 = new String(chars);
```

The object reference variable s3 contains the address of the value stored in the heap area.

- Let's take an example program where we will create a string object and store an array of characters in it.

```
package stringPrograms;
public class Science
{
public static void main(String[] args)
{
 char chars[] = { 's', 'c', 'i', 'e', 'n', 'c', 'e' };
 String s = new String(chars);
 System.out.println(s);
 }
}

Output:
       science
```

- **4. String(char chars[ ], int startIndex, int count)**
  - It will create and initializes a string object with a subrange of a character array.
  - The argument **startIndex** specifies the index at which the subrange begins and **count** specifies the number of characters to be copied.
  - For example:

```
char chars[ ] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' };
String str = new String(chars, 2, 3);
```

  - The object str contains the address of the value "ndo" stored in the heap area because the starting index is 2 and the total number of characters to be copied is 3

Example

```
package stringPrograms;
public class Windows
{
public static void main(String[] args)
{
 char chars[] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' };
 String s = new String(chars, 0,4);
 System.out.println(s);
 }
}

Output:
       wind
```

5. String(byte byteArr[ ]) :

- It constructs a new string object by decoding the given array of bytes (i.e, by decoding ASCII values into the characters) according to the system's default character set.

```
package stringPrograms;
public class ByteArray
{
public static void main(String[] args)
{
 byte b[] = { 97, 98, 99, 100 }; // Range of bytes: -128 to 127. These byte
values will be converted into corresponding characters.
 String s = new String(b);
 System.out.println(s);
 }
}

Output:
       abcd
```

6. String(byte byteArr[ ], int startIndex, int count)

This constructor also creates a new string object by decoding the ASCII values using the system's default character set.

```
package stringPrograms;
public class ByteArray
{
public static void main(String[] args)
{
 byte b[] = { 65, 66, 67, 68, 69, 70 }; // Range of bytes: -128 to 127.
 String s = new String(b, 2, 4); // CDEF
 System.out.println(s);
 }
}

Output:
       CDEF
```

STRING LENGTH

- The java string length() method gives length of the string. It returns count of total number of characters.
- Internal implementation

```
public int length() {
return value.length;
}
```

Signature - The signature of the string length() method is given below:

public int length()

String length() method : Example 1

```
public class LengthExample{
public static void main(String args[]){
String s1="javatpoint";
String s2="python";
System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
System.out.println("string length is: "+s2.length());//6 is the length of python string
}}
```

Output

```
string length is: 10
string length is: 6
```

String length() method : Example 2

```
public class LengthExample2 {
    public static void main(String[] args) {
        String str = "Javatpoint";
        if(str.length()>0) {
            System.out.println("String is not empty and length is: "+str.length());
        }
        str = "";
        if(str.length()==0) {
            System.out.println("String is empty now: "+str.length());
        }
    }
```

Output

```
String is not empty and length is: 10
String is empty now: 0
```

String Comparison

- We can compare string in java on the basis of content and reference .
- There are three ways to compare string in java:
  - By **equals()** method
  - By = = operator
  - By **compareTo()** method
- String compare by equals() method
  - The string equals() method compares the original content of the string.
  - It compares values of string for equality. String class provides two methods

public boolean equals(Object another) compares this string to the specified object.

public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

Example 1

```
String s1="javatpoint";
String s2="javatpoint";
String s3="JAVATPOINT";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is
//same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
```

Example 2

```
String s1="javatpoint";
String s2="javatpoint";
String s3="JAVATPOINT";
String s4="python";
System.out.println(s1.equalsIgnoreCase(s2));//true because
 //content and case both are same
System.out.println(s1.equalsIgnoreCase(s3));//true because
 //case is ignored
System.out.println(s1.equalsIgnoreCase(s4));//false because
 //content is not same
```

String compare by == operator

- The = = operator **compares references** not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true (because both refer to same instance)
   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
 }
}
```

```
Output:true
       false
```

String compare by compareTo() method

- The string **compareTo()** method compares values **lexicographically** and returns an integer value that describes if first string is less than, equal to or greater than second string.

Signature

public int compareTo(String anotherString)

Suppose s1 and s2 are two string variables.

If: s1 == s2 : 0

s1 > s2 : positive value

s1 < s2 : negative value

Example 1

```
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
System.out.println(s1.compareTo(s3));
                     //-5 because "h" is 5 times lower than "m"
System.out.println(s1.compareTo(s4));
                     //-1 because "l" is 1 times lower than "m"
System.out.println(s1.compareTo(s5));//2 because "h" is 2 times
                                     // greater than "f"
```

SEARCHING STRINGS

- **String contains()**
  - The java string **contains()** method searches the sequence of characters in this string.
  - It returns true if sequence of char values are found in this string otherwise returns false.

Signature

- The signature of string contains() method is given below:

  public boolean contains(CharSequence sequence)

Example 1

```
class ContainsExample{
public static void main(String args[]){
String name="what do you know about me";
System.out.println(name.contains("do you know"));
System.out.println(name.contains("about"));
System.out.println(name.contains("hello"));
}}      // o/p  true
                true
                false
```

- The **contains()** method searches case sensitive char sequence. If the argument is not case sensitive, it returns false. Let's see an example below.
- **Example 2**

```
public class ContainsExample2 {
 public static void main(String[] args) {
    String str = "Hello Javatpoint readers";
    boolean isContains = str.contains("Javatpoint");
    System.out.println(isContains);  // Case Sensitive , so true
    System.out.println(str.contains("javatpoint")); // false
      }   // Output
}               true
                false
```

- The **contains()** method is helpful to find a char-sequence in the string. We can use it in control structure to produce search based result. Let us see an example below.
- **Example 3**

```
public class ContainsExample3 {
public static void main(String[] args) {
    String str = "To learn Java visit Javapoint.com";
    if(str.contains("Javapoint.com")) {
        System.out.println("This string contains javapoint.com");
    }else
        System.out.println("Result not found");
  }
 }     //   o/p :  This string contains javapoint.com
```

CHARACTER EXTRACTION

- String charAt()
  - The java string **charAt()** method returns a char value at the given index number.
  - The index number starts from 0 and goes to n-1, where n is length of the string.
  - It returns StringIndexOutOfBoundsException if given index number is greater than or equal to this string length or a negative number.
  - **Signature** - The signature of string **charAt()** method is given below: **public char** charAt(int index)

Example 1

```
public class CharAtExample{
public static void main(String args[]){
String name="javatpoint";
char ch=name.charAt(4);//returns the char value at the 4th index
System.out.println(ch);
}}
```

Output

t

StringIndexOutOfBoundsException with charAt()

- Let's see the example of charAt() method where we are passing greater index value.
- In such case, it throws StringIndexOutOfBoundsException at run time.

```
public class CharAtExample{
public static void main(String args[]){
String name="javatpoint";
char ch=name.charAt(10);//returns the char value at the 10th index
System.out.println(ch);
}}
```

Output:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 10
at java.lang.String.charAt(String.java:658)
at CharAtExample.main(CharAtExample.java:4)
```

Example 2

- Let's see a simple example where we are accessing first and last character from the provided string.

```
public class CharAtExample3 {
    public static void main(String[] args) {
    String str = "Welcome to Javatpoint portal";
    int strLength = str.length();
    // Fetching first character
    System.out.println("Character at 0 index is: "+ str.charAt(0));
    // The last Character is present at the string length-1 index
    System.out.println("Character at last index is: "+ str.charAt(strLength-1));
    }
}
```

//o/p Character at 0 index is: W
Character at last index is: l

Example 3

- Let's see an example where we are accessing all the elements present at odd index.

```
public class CharAtExample4 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        for (int i=0; i<=str.length()-1; i++) {
            if(i%2!=0) {
                System.out.println("Char at "+i+" place "+str.charAt(i));
            }
        }
    }
}
```

Output

```
Char at 1 place e
Char at 3 place c
Char at 5 place m
Char at 7 place
Char at 9 place o
Char at 11 place J
Char at 13 place v
Char at 15 place t
Char at 17 place o
Char at 19 place n
Char at 21 place
Char at 23 place o
Char at 25 place t
Char at 27 place l
```

Example 4

- Let's see an example where we are counting frequency of a character in the string.

```
public class CharAtExample5 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        int count = 0;
        for (int i=0; i<=str.length()-1; i++) {
            if(str.charAt(i) == 't') {
                count++;
            }
        }
        System.out.println("Frequency of t is: "+count);
    }
}
```

Output

Frequecy of t is: 4

MODIFY STRINGS

- The java string replace() method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Signature

- There are two type of replace methods in java string.
  - public String replace(char oldChar, char newChar) and
  - public String replace(CharSequence target, CharSequence replacement)
- The second replace method is added since JDK 1.5.

String replace(char old, char new) method

Example

```
public class ReplaceExample1{
public static void main(String args[]){
 String s1="java is a very good language";
// replaces all occurrences of 'a' to 'e'
 String replaceString=s1.replace('a','e');
   System.out.println(replaceString);
   }
}
```

//Output jeve is e very good lenguege

String replace(CharSequence target, CharSequence replacement)

Example

```
public class ReplaceExample2{
public static void main(String args[]){
String s1="my name is khan my name is java";
String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"
System.out.println(replaceString);
}}
```

Output

my name was khan my name was java

Example

```
public class ReplaceExample3 {
  public static void main(String[] args) {
    String str = "oooooo-hhhh-oooooo";
    String rs = str.replace("h","s"); // Replace 'h' with 's'
    System.out.println(rs);
    rs = rs.replace("s","h"); // Replace 's' with 'h'
    System.out.println(rs);
  }
}
```

Output

```
oooooo-ssss-oooooo
oooooo-hhhh-oooooo
```

String replaceAll()

- The java **String replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.
- regex is any regular exxpression
- **Signature**
  - **public String replaceAll(String regex, String replacement)**

Example

```
public class ReplaceAllExample1{
public static void main(String args[]){
 String s1="java is a very good language";
  String replaceString=s1.replaceAll("a","e");
   System.out.println(replaceString); }}
//Output jeve is e very good lenguege
```

/***Note: String replaceAll(String regex, String replacement)** and **String replace(CharSequence target, CharSequence replacement)** work similarly. Here **regex** means regular expressions.*/

String replaceAll() example: remove white spaces

Let's see an example to remove all the occurrences of white spaces.

Example

```
public class ReplaceAllExample3{
public static void main(String args[]){
String s1="My name is Khan. My name is Bob. My name is Sonoo.";
String replaceString=s1.replaceAll("\\s","");
System.out.println(replaceString);
}}
```

Output

MynameiskhanMynameisBobMynameisSonoo

STRING VALUE OF ( )

- The java **String valueOf()** method converts different types of values into string.
- By the help of string valueOf() method, we can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Signature

- The signature or syntax of string **valueOf()** method is given below:

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(char[] c)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
public static String valueOf(double d)
public static String valueOf(Object o)
```

valueOf() method example

Example

```
public class StringValueOfExample{
public static void main(String args[]){
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);//concatenating string with 10
}}
```

Output
3010

valueOf(char ch) Method

- This is a char version of overloaded valueOf() method. It takes char value and returns a string.

Example

```
public class StringValueOfExample3 {
    public static void main(String[] args) {
        // char to String
        char ch1 = 'A';
        char ch2 = 'B';
        String s1 = String.valueOf(ch1);
        String s2 = String.valueOf(ch2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output

A

B

valueOf(float f) and valueOf(double d)

- This is a float version of overloaded valueOf() method. It takes float value and returns a string.

Example

```
public class StringValueOfExample4 {
    public static void main(String[] args) {
        // Float and Double to String
        float f  = 10.05f;
        double d = 10.02;
        String s1 = String.valueOf(f);
        String s2 = String.valueOf(d);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output

10.05

10.02

valueOf(boolean bol) Method

- This is a boolean version of overloaded valueOf() method. It takes boolean value and returns a string.

Example

```
public class StringValueOfExample2 {
    public static void main(String[] args) {
        // Boolean to String
        boolean bol = true;
        boolean bol2 = false;
        String s1 = String.valueOf(bol);
        String s2 = String.valueOf(bol2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output

true

false

Other String methods

- Let String s= "HelloWorld"
- **String substring (int i):** Return the substring from the ith index character to end.

  s.substring(3); **// returns "loWorld"**

- **String substring (int i, int j):** Returns the substring from i to j-1 index.

  s.substring(2, 5); **// returns "llo"**

- **String concat(String str):** Concatenates specified string to the end of this string. Let String s1 = "Hello"; String s2 = "World";

  String output = **s1.concat(s2);** **// returns "HelloWorld"**

  - **Note:** Java string **concatenation operator (+)** is used to add strings. For Example: String s="Sachin"+"Tendulkar";

    System.out.println(s);//SachinTendulkar

Other String methods

- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.
- String s = ”Learn Share Learn”; int output = s.indexOf(“Share”);

  // returns 6
- **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index. String s = ”Learn Share Learn”;

  int output = s.indexOf("ea",3); **// returns 13**
- **Int lastIndexOf(String s):** Returns the index within the string of the last occurrence of the specified string. String s = ”Learn Share Learn”;

  int output = s.lastIndexOf("a"); **// returns 14**

Other String methods

- **String toLowerCase():** Converts all the characters in the String to lower case.String word1 = “HeLLo”;

  String word3 = word1.toLowerCase(); **// returns “hello"**

- **String toUpperCase():** Converts all the characters in the String to upper case.String word1 = “HeLLo”;

  String word2 = word1.toUpperCase(); **// returns “HELLO”**

- **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.String word1 = “ Learn Share Learn “;

  String word2 = word1.trim(); **// returns “Learn Share Learn”**

String and StringBuffer

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

- Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

- **Mutable string** - A string that can be modified or changed is known as mutable string. **StringBuffer** and **StringBuilder** classes are used for creating mutable string.

StringBuffer append() method

- The append() method concatenates the given argument with this string.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
//o/p   HelloJava
```

StringBuffer insert() method

- The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
// o/p   HJavaello
```

StringBuffer replace() method

- The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
 //  o/p   HJavalo
```

StringBuffer delete() method

- The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
  //o/p   Hlo
```

StringBuffer reverse() method

- The reverse() method of StringBuilder class reverses the current string.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

//o/p olleH

StringBuffer capacity() method

- The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

String array is also possible

Example

```
class StringDemo3
{
    public static void main(String args[])
    {
        String str[] = { "one", "two", "three" };
        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " + str[i]);
    }
}
//      o/p     str[0]:  one
                str[1]: two
                str[2]: three
```

Write a java program to read 3 strings from user and display the strings in uppercase

```
import java.io.*;
class Stringeg
{       public static void main(String args[])
        {
            String str[]=new String[5];
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            try
            {
                for(int i=0;i<3;i++)
                {
                    str[i]=br.readLine();
                }
            }
            catch(IOException e)
            {
                System.out.println("Exception caught "+e);
            }
            System.out.println("The Strings are");
            for(int i=0;i<3;i++)
            {
                 System.out.println(str[i].toUpperCase());
            }
        }
}
```

COLLECTIONS IN JAVA

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection in Java - Represents a single unit of objects, i.e., a group.

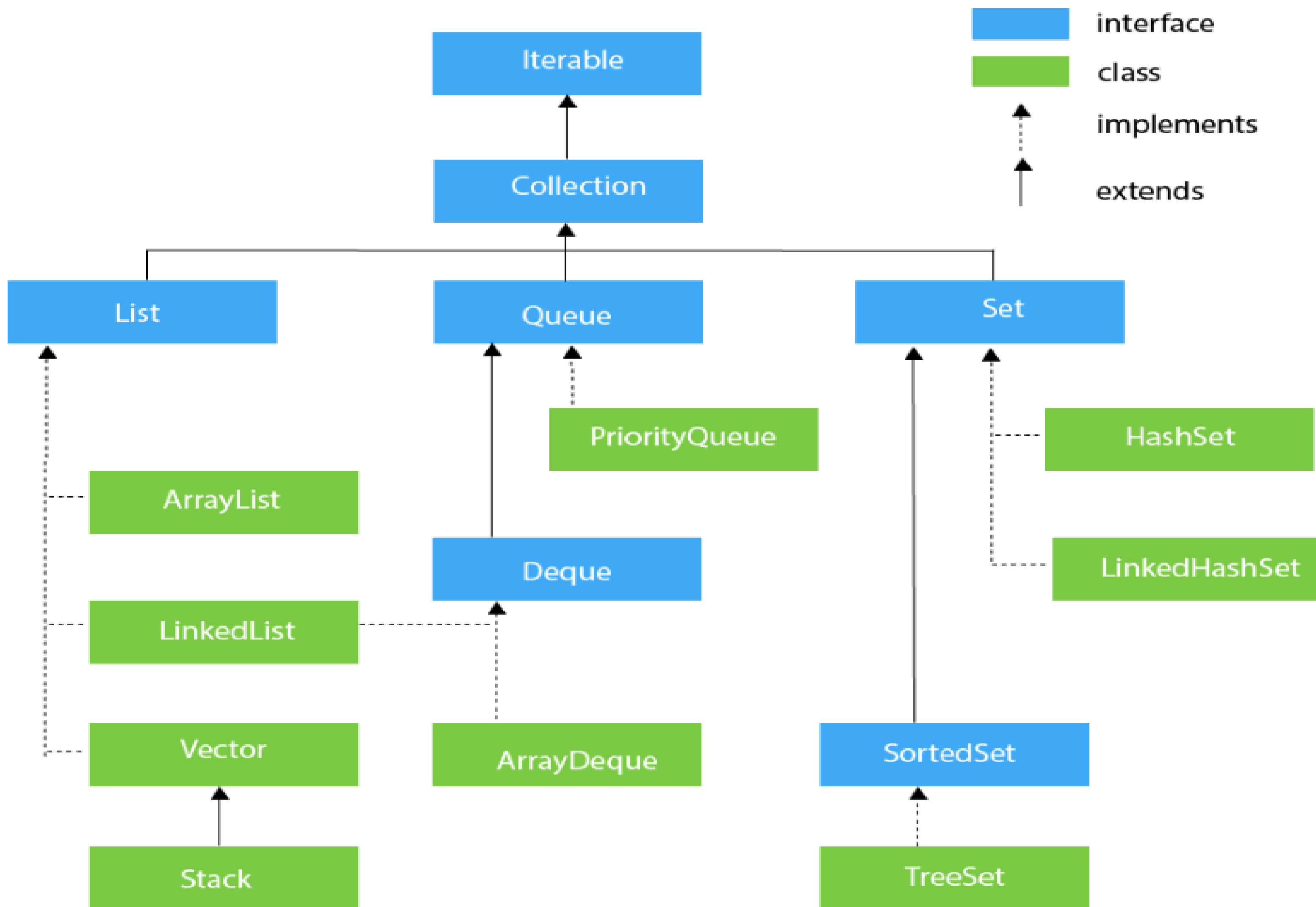
framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

Collection framework

- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:
  - • Interfaces and its implementations, i.e., classes
  - • Algorithm

Hierarchy of Collection Framework



- The **java.util** package contains all the classes and interfaces for the Collection framework.

Collection Interface

- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

LIST INTERFACE

- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

- To instantiate the List interface, we must use :

```
List list1= new ArrayList();
List list2 = new LinkedList();
List list3 = new Vector();
List list4 = new Stack();
```

- There are various methods in List interface like **add**(Object o), **remove**(int index), **get**(int index) etc that can be used to insert, delete, and access the elements from the list.
- The classes that implement the List interface are given below.

ArrayList

- The ArrayList class implements the List interface. Java ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime.

Array Vs ArrayList

	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

Constructor & Description of ArrayList

1**ArrayList( )**

- This constructor builds an empty array list.

2ArrayList(Collection c)

- This constructor builds an array list that is initialized with the elements of the collection **c**.

3ArrayList(int capacity)

- This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

- ArrayList is a generic class that has the following declaration: **class ArrayList<E>**.Here, **E** specifies the **type of objects** that the list ill hold. Consider the following example.

Example

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();/*Creating
                                                    arraylist*/
     list.add("Mango");//Adding object in arraylist
     list.add("Apple");
     list.add("Banana");
     list.add("Grapes");
     //Printing the arraylist object
     System.out.println(list);
 }   //            Output
}   //     [Mango, Apple, Banana, Grapes]
```

- Java ArrayList class uses a dynamic array for storing the elements.
- It is like an array, but there is no size limit. We can add or remove elements anytime.
- So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.
- The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here.
- The ArrayList maintains the insertion order internally.
- ArrayList inherits the AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Iterating ArrayList using Iterator

- An iterator is an interface.
- It can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.
- Iterator object can be created by calling **iterator()** method present in Collection interface.
- Iterator interface defines **three** methods:

1. public boolean **hasNext();** /* Returns true if the iteration has more elements*/
2. public Object **next();** **/*** Returns the next element in the iteration*/
3. public void **remove();** **/*** Remove the next element in the iteration*/

Iterating ArrayList using Iterator

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating ArrayList
  list.add("Mango");//Adding object in ArrayList
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }   }   }
```

//Output:

Mango
Apple
Banana
Grapes

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();/*Creating
                                    arraylist of String type objects  */
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator      // Output
Iterator itr=list.iterator();               Ravi
while(itr.hasNext()){                       Vijay
System.out.println(itr.next());             Ravi
}  }  }                                     Ajay
```