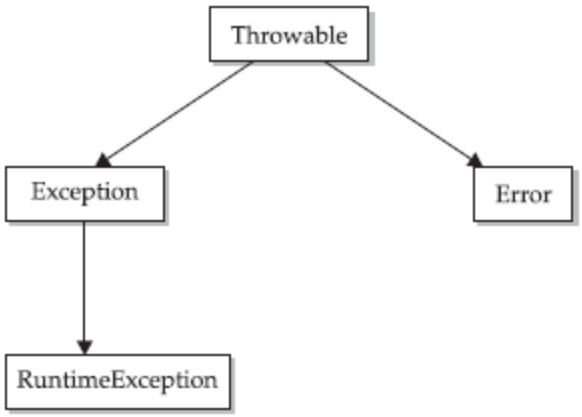
UNIT-IV

Exception Handling:

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a runtime error.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exception hierarchy:



All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
From Throwable class, two classes are derived.

1) Exception
2) Error

Exception class is used for exceptional conditions that user programs should catch.
Error class defines exceptions that are not expected to be caught under normal circumstances by your program.

Consider the following program

```
class A
{
   public static void main(String args[])
   {
      int a=0;
      int b=5/a;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **A** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when the above program is executed:

```
java.lang.ArithmeticException: / by zero
        at A.main(A.java:6)
```

In JAVA language, exceptions are handled by using the following keywords

try
catch
throw
throws
finally

- Program statements that we want to monitor for exceptions are contained within a **try** block.
- Exceptions thrown in try block can be caught and handled using **catch** block. Catch block is also called as Exception Handler.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Following is the general form of exception handling block

```
try
{
   //block of code to monitor for exceptions
}
catch(ExceptionType1 exobj)
{
   //code to handle ExceptionType1
}
catch(ExceptionType2 exobj)
{
    //code to handle ExceptionType2
}
finally
{
    //code that must be executed after try/catch block
}
```

Example:

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

```
  {
    int a,b;
    try
    {
       a=0;
       b=5/a;
    }
    catch(ArithmeticException ae)
    {
          System.out.println("can't perform division by zero");
    }
    System.out.println("After catch statement");
  }
}
```

Displaying a description of an exception:

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. We can display this description in a **println( )** statement by simply passing the exception as an argument.

Example:

```
class A
{
  public static void main(String args[])
  {
    int a,b;
    try
    {
       a=0;
       b=5/a;
    }
    catch(ArithmeticException ae)
    {
          System.out.println(ae);
    }
    System.out.println("After catch statement");
  }
}
```

Multiple catch blocks:

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.

- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try /catch** block.

Example:

```
class MultiplecatchDemo
{
   public static void main(String args[])
   {
     try
    {
        int a=args.length;
        System.out.println("a="+a);
        int b=5/a;
        int c[]={1};
        c[3]=2;
    }
   catch(ArithemticException ae)
   {
      System.out.println(ae);
    }
  catch(ArrayIndexOutOfBoundsException aie)
  {
     System.out.println(aie);
   }
 }
}
```

When we use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Example:

```
class MultiplecatchDemo
{
    public static void main(String args[])
    {
        try
       {
          int a=args.length;
      int b=5/a;
     }
     catch(ArithemeticException ae)
     {
        System.out.println(ae);
     }
    catch(Exception e)
```

```
    {
        System.out.println(e);
    }
  }
}
```

Nested try statements:

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- If an inner **try** statement does not have a **catch** handler for a particular exception, then the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

Example:

```
class NestedTryDemo
{
   public static void main(String args[])
   {
        try
        {
           int a=args.length;
           int b=5/a;
           System.out.println("a="+a);
           try
           {
              if(a==1)
                   a=a/(a-a);
              if(a==2)
              {
                   int c[]={1};
                    c[3]=2;
               }
            }
          catch(ArrayIndexOutOfBoundsException aie)
          {
              System.out.println(aie);
           }
      }
    catch(ArithmeticException ae)
    {
          System.out.println(ae);
     }
  }
}
```

We can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method.

Example:

```
class NestedTryDemo
{
   static void display(int a)
   {
         try
          {
             if(a==1)
                  a=a/(a-a);
             if(a==2)
             {
                  int c[]={1};
                   c[3]=2;
              }
           }
         catch(ArrayIndexOutOfBoundsException aie)
         {
             System.out.println(aie);
          }
    }
   public static void main(String args[])
   {
        try
        {
           int a=args.length;
           int b=5/a;
           System.out.println("a="+a);
           display(a);
     }
    catch(ArithmeticException ae)
    {
         System.out.println(ae);
    }
 }
}
```

throw:

So far, we have only been catching exceptions that are thrown by the Java run-time system. However, it is possible to throw an exception explicitly, using the **throw** statement.

Syntax:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type **Throwable** or a subclass of **Throwable**.

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Example:

```
class ThrowDemo
{
   static void show()
   {
        try
        {
           throw new NullPointerException("demo");
         }
        catch(NullPointerException ne)
        {
              System.out.println(ne);
         }
    }
   public static void main(String args[])
   {
       show();
   }
}
```

It is possible to re throw an exception from catch block.

Example:

```
class ThrowDemo
{
   static void show()
   {
        try
        {
           throw new NullPointerException("demo");
         }
        catch(NullPointerException ne)
        {
```

```
            System.out.println(ne);
            throw ne;
        }
    }
    public static void main(String args[])
    {
        try
        {
        show();
        }
        catch(NullPointerException ne)
        {
            System.out.println(ne);
        }
    }
}
```

throws:

If a method causes an exception and the method don't want to handle that exception, then we use throws keyword to throw the exception away from the method.

Syntax:

```
returntype method_name(parameter_list) throws exception_list
{
    //body
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Example:

```
class ThrowsDemo
{
  static void display() throws IllegalAccessException
   {
      System.out.println("Inside display");
      throw new IllegalAccessException();
    }
  public static void main(String args[])
  {
     try
     {
         display();
     }
    catch(IllegalAccessException ie)
    {
        System.out.println(ie);
     }
  }
```

}

finally:

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- Each **try** statement requires at least one **catch** or a **finally** clause.

Example:

```
class FinallyDemo
{
        static void methodA()
        {
            try
            {
                System.out.println("inside method A");
                throw new RuntimeException("demo");
             }
            finally
           {
                System.out.println("finally in method A");
            }
        }
       static void methodB()
      {
           try
           {
                  System.out.println("inside method B");
                      return;
                 }
            finally
           {
                System.out.println("finally in method B");
            }
          }
        static void methodC()
        {
              try
             {
                   System.out.println("inside method C");
              }
            finally
            {
                 System.out.println("finally in method C");
```

```
            }
        }
        public static void main(String args[])
        {
            try
            {
                methodA();
            }
            catch (RuntimeException re)
            {
                System.out.println(re);
            }
                methodB();
                methodC();
        }
}
```

User defined Exceptions:

- Although Java's built-in exceptions handle most common errors, we want to create our own exception types to handle situations specific to our applications.
- Exceptions created by user are called as user defined exceptions.
- In order to create user defined exceptions, we have to create sub class for Exception class.
- In that class, we have to override toString() method to provide the description about the exception.

Example:

```
class MyException extends Exception
{
        private int detail;
        MyException(int a)
        {
            detail = a;
        }

        public String toString()
        {
            return "MyException[" + detail + "]";
        }
}
class ExceptionDemo
{
    static void compute(int a) throws MyException
    {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
                throw new MyException(a);
        System.out.println("Normal exit");
    }
    public static void main(String args[])
    {
        try
```

```
        {
            compute(1);
           compute(20);
         }
        catch (MyException e)
       {
            System.out.println("Caught " + e);
        }
   }
}
```

Checked Exceptions:

The exceptions that are checked during the compile-time are termed as Checked exceptions in Java.

The Java compiler checks the checked exceptions during compilation to verify that a method that is throwing an exception contains the code to handle the exception with the try-catch block or not.

And, if there is no code to handle them, then the compiler checks whether the method is declared using the throws keyword. And, if the compiler finds neither of the two cases, then it gives a compilation error.

A checked exception extends the Exception class.

Following are the some of the checked exceptions defined in java.lang package

Exception	Meaning
ClassNotFoundException	Class not found
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Unchecked Exceptions:

- An exception that occurs during the execution of a program is called an unchecked or a runtime exception.
- The main cause of unchecked exceptions is mostly due to programming errors like attempting to access an element with an invalid index, calling the method with illegal arguments, etc.
- In Java, the direct parent class of Unchecked Exception is RuntimeException.

Following are the some of the unchecked exceptions defined in java.lang package.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format

Streams:

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices.
- Java implements streams within class hierarchies defined in the **java.io** package.

JAVA defines two types of streams. They are

- Byte Streams
- Character Streams

Byte Streams:

Byte streams provide a convenient means for handling input and output of bytes.
Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.
Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.
Following are the some of the byte stream classes in **java.io** package.

Stream class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
RandomAccessFile	It contains methods for accessing file in random manner

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data.

Character Streams:

- Character streams provide a convenient means for handling input and output of characters.
- Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.
- Following are the some of the character stream classes in **java.io** package.

Stream class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
InputStreamReader	Input stream that translates bytes to characters
OutputStreamWriter	Output stream that translates characters to bytes

File Handling:

To handle the file I/O operations in terms of bytes, the classes FileInputStream and FileOutputStream are used.

These 2 classes provide several constructors. One of the constructor provided by these two classes are

FileInputStream(String filename) throws FileNotFoundException

FileOutputStream(String filename) throws FileNotFoundException

In order to write the data to a file, write method is used.

void write(byte byteval) throws IOException

To read the contents of a file, read method is used.

int read() throws IOException

read() method returns -1 when the end of the file reached.

After performing writing or reading operation, a file must be closed by using close method.

void close() throws IOException

Writing a file:

//Program to write the data to a file

```
import java.io.*;
class WriteDemo
{
   public static void main(String args[]) throws Exception
   {
      FileOutputStream fout=new FileOutputStream("abc.txt");
      String s1="Vishnu institute of Technology";
       byte b[]=s1.getBytes();
      fout.write(b);
      fout.close();
    }
}
```

Reading a file:

//Program to read the data from a file

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[]) throws Exception
   {
          FileInputStream fin=new FileInputStream("abc.txt");
          int i;
          while((i=fin.read())!=-1)
          {
             System.out.println((char)i);
          }
        fin.close();
```

```
    }
}
```

Write a program to copy the contents of one file to another file.

```
//Program to copy from one file another file
import java.io.*;
class CopyDemo
{
  public static void main(String args[]) throws Exception
 {
    FileInputStream fin=new FileInputStream("A.java");
    FileOutputStream fout=new FileOutputStream("B.java");
     int i;
      while((i=fin.read())!=-1)
      {
          fout.write((byte)i);
       }
    fin.close();
    fout.close();
    }
}
```

To perform file I/O operations in terms of characters, JAVA language provides two classes such as FileWriter and FileReader.

FileWriter creates a writer that is used to write the data to a file.

One of the constructors provided by FileWriter class is

FileWriter (String filename)

FileReader creates a reader that is used to read the data from a file.

One of the constructors provided by FileReader class is

FileReader (String filename)

To perform reading and writing operations, the following methods are used.

int read()

void write(String s)

void write(char ch)

Write a program for writing data to a file.

```
import java.io.*;
class WriteDemo
{
  public static void main(String args[]) throws Exception
  {
      FileWriter fw=new FileWriter("simple.txt");
      String s1="Information Technology";
      fw.write(s1);
      fw.close();
   }
}
```

Write a program for reading data from a file.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
        FileReader fr=new FileReader("simple.txt");
        int i;
       while((i=fr.read())!=-1)
       {
           System.out.println((char)i);
       }
```

```
   fr.close();
    }
}
```

Write a program for reading the data from a file line by line.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("simple.txt");
      BufferedReader br=new BufferedReader(fr);
      String s;
       while((s=br.readLine())!=null)
      {
        System.out.println(s);
       }
     fr.close();
  }
}
```

Write a program to read a file and displays the contents of a file on the screen with line number before each line.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("simple.txt");
      BufferedReader br=new BufferedReader(fr);
```

```
        String s;
        int i=1;
         while((s=br.readLine())!=null)
        {
          System.out.println(i+" "+s);
           i++;
         }
       fr.close();
    }
}
```

Write a program to count the number of characters, words and lines in a text file.

```
import java.io.*;
class CountDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("abc.txt");
      BufferedReader br=new BufferedReader(fr);
      String s;
      int noc=0,now=0,nol=0;
     while((s=br.readLine())!=null)
      {
            nol++;
            String words[]=s.split(" ");
            now=now+words.length;
            for(String word:words)
                noc=noc+word.length();
       }
```

```
   System.out.println("Number of characters: "+noc);
   System.out.println("Number of words: "+now);
   System.out.println("Number of lines: "+nol);
   fr.close();
    }
}
```

RandomAccessFile:

To access the file in random manner, JAVA language provides one class named as RandomAccessFile.

One of the constructors provided by RandomAccessFile class is

RandomAccessFile(String filename,String access)

Here access specifies in which type of mode the file is to be opened.

Access can take any one of the following values

r-file is opened for reading only

rw-file is opened for both reading and writing

To move from one location to another location in a file, the following method is used

void seek(long pos)

To know the current position of the file pointer in a file, the following method is used

long getFilePointer()

Example:

```
import java.io.*;
class RandomDemo
{
  public static void main(String args[]) throws Exception
 {
    RandomAccessFile rf=new RandomAccessFile("simple.txt","rw");
     rf.seek(6);
    String s1="afternoon";
```

```
    byte b[]=s1.getBytes();
    rf.write(b);
    rf.close();
   }
}
```

CharArrayReader:

CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

CharArrayReader(char c [ ])
CharArrayReader(char c [ ], int *start*, int *numChars*)

Here, c is the input source. The second constructor creates a Reader from a subset of character array that begins with the character at the index specified by *start* and is *numChars* long.

Example:

```
import java.io.*;
class CharArrayReadDemo
{
    public static void main(String args[]) throws Exception
    {
        String s1="Vishnu institute of technology";
        int length=s1.length();
        char c[]=new char[length];
         s1.getChars(0,length,c,0);
        CharArrayReader cr=new CharArrayReader(c);
        int i;
        while((i=cr.read())!=-1)
       {
          System.out.println((char)i);
        }
```

```
    System.out.println();
    CharArrayReader cr1=new CharArrayReader(c,0,6);
    while((i=cr1.read())!=-1)
    {
        System.out.println((char)i);
    }
  }
}
```

CharArrayWriter:

CharArrayWriter is an implementation of an output stream that uses an array as the destination. CharArrayWriter has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*.

Example:

```
import java.io.*;
class CharArrayWriteDemo
{
 public static void main(String args[]) throws Exception
 {
  CharArrayWriter cw=new CharArrayWriter();
  String s1="vishnu institute of technology";
  int length=s1.length();
  char c[]=new char[length];
  s1.getChars(0,length,c,0);
  cw.write(c);
  FileWriter fw=new FileWriter("sample.txt");
  cw.writeTo(fw);
  fw.close();
```

```
 }
}
```

Strings:

- String is a group of characters.
- In JAVA language, string is implemented as object of type String.
- String is a one of the pre defined class available in java.lang package.

To create a string object, String class provides several constructors.
Some of the constructors are as follows:
1.To create a string objects with no characters, the following constructor is used
Syntax:

```
String()
```

Example:

```
String s1=new String();
```

2. To create a string object from character array, the following constructor is used.
Syntax:

```
String(char ch[])
```

Example:

```
char x[]={'a', 'b', 'c', 'd'};

String s1=new String(x);
```

3. To create a sub range of characters from character array, the following constructor is used.

Syntax:

```
String(char ch[],int startIndex,intnumChars)
```

- Here, startIndex specifies the starting index of subrange in character array.
- Here, numChars represents, number of characters starting from starting index to be stored in string object.

Example:

```
char x[]={'a', 'b', 'c', 'd', 'e'}

String s1=new String(x,2,3);
```

4. To create string object from the existing string object, the following constructor is used.

Syntax:

String(String str)

Example:

```
char x[]={'v', 'i', 's', 'h', 'n', 'u'}

    String s1=new String(x);

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

We can create a string object by passing string value.

```
String s1=new String("VIT");
```

Reading a string as input from the keyboard:

In order to read a string asinput from the keyboard, Scanner class provides two methods.

1.next()
2.nextLine()

next() method reads a string without white spaces.

Example:

```
String s1=new String();
Scanner sc=new Scanner(System.in);
System.out.println("Enter a string");
s1=sc.next();
```

nextLine() method accepts a string with white spaces.

Example:

```
String s1=new String();
Scanner sc=new Scanner(System.in);
System.out.println("Enter a string");
s1=sc.nextLine();
```

NOTE: String objects are immutable objects. That means, once a value is assigned to a string object it cannot be modified.

String Handling Methods:

Java language provides various String Handling methods to perform operations on strings.

1.int length():

This method returns number of characters available in a string.

Example:

```
String s1=new String("Vishnu");
 int n=s1.length();
```

2.char charAt(int index):

This method returns a character at a specified index value.

Example:

```
String s1=new String("Vishnu");
int n=s1.charAt(5);
```

3.char[] getChars(int start,intend,char target[],int targetindex):

- This method returns a group of characters in a string starting from start and up to end-1.
- Group of characters will be stored in array called as target. target index represents the starting index in target array.

Example:

```
String s1="Vishnu Institute of Technology";
int start=7,end=16;
char x[]=new char[end- start];
s1.getChars(start,end,x,0);
System.out.println(x);
```

4.boolean equals(String str):

This method is used to check whether the two strings are identical or not. This method returns true if strings are identical, otherwise returns false.

Example:

```
String s1=new String("Vishnu institute of technology");
String s2=new String("Vishnu institute of technology");
if(s1.equals(s2))
        System.out.println("equal")
else
        System.out.println("not equal");
```

5.boolean equalsIgnoreCase(String str):

This method is used to check whether the two strings are identical or not by ignoring sensitive letters.

Example

```
String s1=new String("vishnu college");
String s2=new String("VISHNU COLLEGE");
if(s1.equalsIgnoreCase(s2))
         System.out.println("equal");
  else
         System.out.println("not equal");
```

6.int indexOf(char ch):

This method gives the index of first occurrence of given character in a string.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.indexOf('i'));
```

7.int lastIndexOf(char ch):

This method returns the index of last occurrence of given character in a string.

Example:

```
String s1=new String("Vishnu Institute of Technology");
System.out.println(s1.lastIndexOf('i')) ;
```

8.int indexOf(String str):

This method returns the index of first occurrence of specified string in the invoked string.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.indexOf("is"));
```

9.int lastIndexOf(String str):

This method returns the index of last occurrence of specified string in the invoked string.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.indexOf("is"));
```

10.boolean startsWith(String str):

This method determines whether the invoking string starts with the specified string or not. This method returns true, if the invoking string starts with the specified string otherwise false.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.startsWith("This"));
```

11.boolean endsWith(String str):

This method determines whether the invoking string ends with the specified string or not. This method returns true, if the invoking string ends with the specified string otherwise false.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.endsWith("Java"));
```

12.string substring(int startindex):

This method extracts a substring from the invoking string starting from start index to till the end of the string.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.substring(7));
```

13.string substring(int startindex,intendindex):

This method also extracts the substring from the invoking string starting from start index to end index-1.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.substring(7,16));
```

14.int compareTo(string str):

This method is used to compare two strings.This method returns integer values such as positive or negative or zero value.

return value	Meaning
<0	Invoking string is lesser than str.
>0	Invoking string is lesser than str.
=0	equal

Example:

```
String s1="Vishnu";
String s2="Java";
System.out.println(s1.compareTo(s2));
```

15.int compareToIgnoreCase(String str):

This method is used to compare the two strings by ignoring case sensitive letters. This method also gives integer values such as positive or negative or zero values.

Example:

```
String s1="vishnu";
String s2="VISHNU;
System.out.println(s1.compareToIgnoreCase(s2));
```

16.string concat(String str):

This method is used to concatenate the specified string with invoking string and returns a string object.

Example:

```
String s1="Vishnu";
String s2=s1.concat("college");
System.out.println(s2);
```

17.string replace(char original,char replacement):

This method is used to replace a character in a string object with new character and returns a string object.

Example:

```
String s1="hello";
System.out.println(s1.replace('e', 'a'));
```

18.string replace(string replace, string original):

This method replaces a group of characters in a string object with another group of characters and returns a string object.

Example:

```
String s1="Vishnu Institute of Technology";
System.out.println("Vishnu", "B V Raju");
System.out.println(s1);
```

19.string join(charSequence delimiter, charSequence String):

This method concatenates two or more strings with specified delimiter.

Example:

```
String s1=new String();
System.out.println(s1.join(" ", "This", "is", "Java", "class"));
```

20.string toLowerCase():

This method is used to convert all the characters in the given string into lowercase letters.

Example:

```
String s1=new String("VISHNU");
System.out.println(s1.toLowerCase());
```

21.string toUpperCase():

This method is used to convert all the characters in the given string into uppercase letters.

Example:

```
String s1=new String("vishnu");
System.out.println(s1.toUpperCase());
```

22.string trim():

This method is used to remove any leading or trailing white spaces in the string object.

Example:

```
String s1=new String("        vishnu        ");
System.out.println(s1.trim());
```

String Buffer:

- String objects are always fixed length and are immutable objects.
- In order to modify the string objects, we use StringBuffer class.
- StringBuffer class provides string objects as growable and mutable objects.
- StringBuffer class is one of the pre defined class in java.lang package.
- StringBuffer class provides the following constructors.

1.StringBuffer():

This constructor is used to create StringBuffer object with capacity of 16 characters without any reallocation.

Example:

```
StringBuffer sb=new StringBuffer();
```

2.StringBuffer(int size):

This constructor is used to create StringBuffer object with specified size as capacity.

Example:

```
StringBuffer sb=new StringBuffer(30);
```

3.StringBuffer(string str):

This constructor is used to create StringBuffer object from string object and it reserves 16 more characters without reallocation.

Example:

```
StringBuffer sb=new StringBuffer("hello");
```

Methods:

1.int length():

This method returns the length of the StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
```

2.int capacity():

This method returns the capacity of the StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
```

3.char charAt(int index):

This method returns a character at a specified index value.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
int n=sb.charAt(5);
```

4.void getChars(int startindex,int endindex,char target[],char targetstart)

5.void setCharAt(int index,charch)

This method is used to set a new character at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello");
sb.setChar(1,'a');
System.out.println(sb);
```

6.StringBuffer append(String str):

This method is used to append the specified string to the invoking StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Hello");
sb.append(" Java");
System.out.println(sb);
```

7.StringBuffer append(int num):

- This method appends string representation of integer type to the StringBuffer object.

Example :

```
StringBuffer sb=new StringBuffer("hello");
sb.append(123);
System.out.println(sb);
```

8.StringBuffer insert(int index, char ch):

This method is used to insert a character at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello");
sb.insert(1,'a');
System.out.println(sb);
```

9.StringBuffer insert(int index, string str):

This method is used to insert a string in StringBuffer at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("I Java");
sb.insert(2,"like");
System.out.println(sb);
```

10.StringBuffer reverse():

This method reverses all the characters in the invoking StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("hello");
 sb.reverse();
System.out.println(sb);
```

11.StringBuffer replace(int startindex,intendindex,string str):

This method replaces the characters from startindex to endindex-1 with the specified str.

Example:

```
StringBuffer sb=new StringBuffer("This is  test");
sb.replace(5,7, "was");
System.out.println(sb);
```

12.StringBuffer delete(int startindex, int endindex):

This method deletes the characters from startindex to endindex-1.

Example:

```
StringBuffer sb=new StringBuffer("hello JAVA");
sb.delete(0,5);
System.out.println(sb);
```

13.StringBuffer deleteCharAt(int index):

This method deletes a single character available at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello JAVA");
System.out.println(sb.deleteCharAt(4));
```

14.String substring(int startindex):

15.String substring(int startindex, int endindex):

Programs

1. **Write a program to check whether the given string is palindrome or not.**

```
//Program to check the string is palindrome or not
import java.util.*;
class Palindrome
{
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String s1=new String();
		System.out.println("Enter a string");
		s1=sc.next();
		String s2=new String();
		for(int i=s1.length()-1;i>=0;i--)
		{
			s2=s2+s1.charAt(i);
		}
		if(s1.equals(s2))
			System.out.println("Given string is palindrome");
		else
			System.out.println("Given string is not a palindrome");
	}
}
```

2.Write a program for sorting a list of names in ascending order.

```
//Program for sorting a list of names in ascending order
import java.util.*;
class Sorting
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number of strings");
        int n=sc.nextInt();
        String s1=new String[n];
        System.out.println("Enter strings");
        for(int i=0;i<n;i++)
        {
            s1[i]=sc.next();
        }
        for(i=0;i<n;i++)
        {
            for(int j=i+1;j<=n;j++)
            {
                if(s1[i].compareTo(s1[j]>0))
                {
                    String temp=s1[i];
```

```
                    s1[i]=s1[j];
                    s1[j]=temp;
                }
            }
        }
    }
}
```