UNIT IV I/O, GENERICS, STRING HANDLING

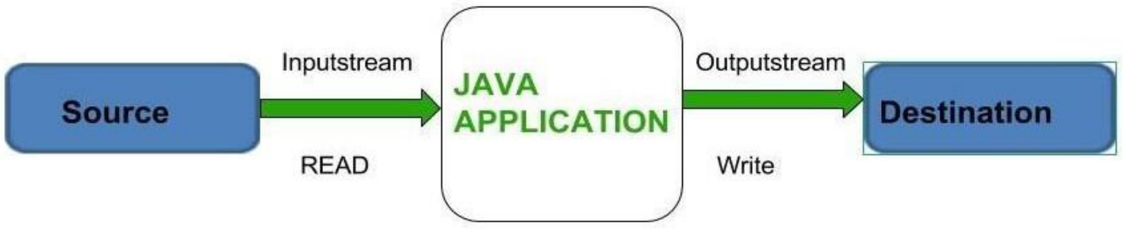
I/O Basics – Reading and Writing Console I/O – Reading and Writing Files. Generics: Generic Programming – Generic classes – Generic Methods – Bounded Types – Restrictions and Limitations. Strings: Basic String class, methods and String Buffer Class.

4.1: INPUT / OUTPUT BASICS

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

3(i)

- ✓ Java uses the concept of stream to make I/O operation fast**.**
- ✓ These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.
- ✓ **The java.io package** contains all the classes required for input and output operations.



➢ Java Input

There are several ways to get input from the user in Java. To get input by using Scanner object, import Scanner class using:

import java.util.Scanner;

Then, we will create an object of Scanner class which will be used to get input from the user.

Scanner input = new Scanner (System.in);
int number = input.nextInt();

Example : Get Integer Input From the User

```
import java.util.Scanner;
class Input{
public static void main(String[] args){
    Scanner input =newScanner (System.in);
    System.out.print ("Enter an integer: ");
    int number =input.nextInt ();
```

```
    System.out.println ("You entered"+ number);
}
```

```
}
```

<u>Output</u>

```
Enter an integer: 23
You entered 23
```

➢ <u>Java Output</u>

Simply use System.out.println(), System.out.print() or System.out.printf() to send output to standard output (screen).system is a class and out is a public static field which accepts output data.

Example to output a line:

```
class Test
{
  public static void main(String[] args)
  {
        System.out.println("Java programming is interesting.");
  }
}
```

<u>Output:</u>

```
Java programming is interesting.
```

What's the difference between println (), print () and printf ()?

- print () - prints string inside the quotes.
- println () - prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- printf () - it provides string formatting.

)3(i)

4.1.1: STREAMS

A **Stream** is a sequence of data or it is an abstraction that either produces or consumes information. In other simple words it is a flow of data from which you can read or write data to it. It's called a stream because it's like a stream of water that continues to flow.

3(i)

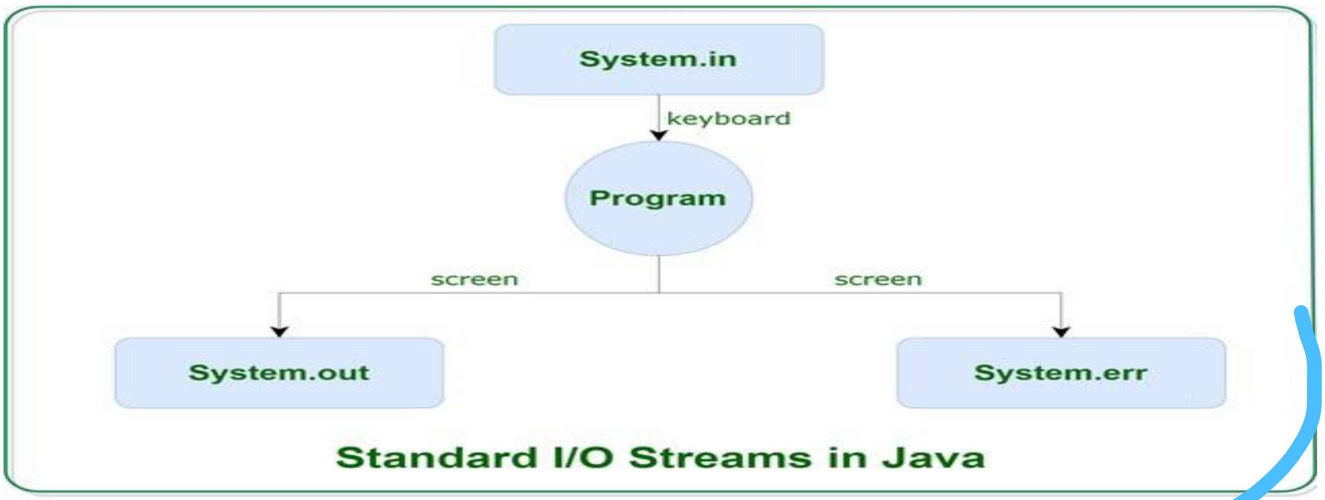
➢ **PREDEFINED STREAMS:**

In java, 3 streams are created for us automatically. All these streams are attached with console.

System.in: This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**. – It is an object of type InputStream.

1) System.out: This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**. - It is an object of type PrintStream

2) System.err: This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**. - It is an object of type PrintStream



➢ **TYPES OF STREAMS:**

1. Byte Stream – Byte Streams provide a convenient means of handling input andoutput in terms of bytes. Byte streams are used when reading or writing binary data.

2. Character Stream – Character streams provide a convenient means of handling input or output in terms of characters. In some cases, character streams are more efficient than byte streams.

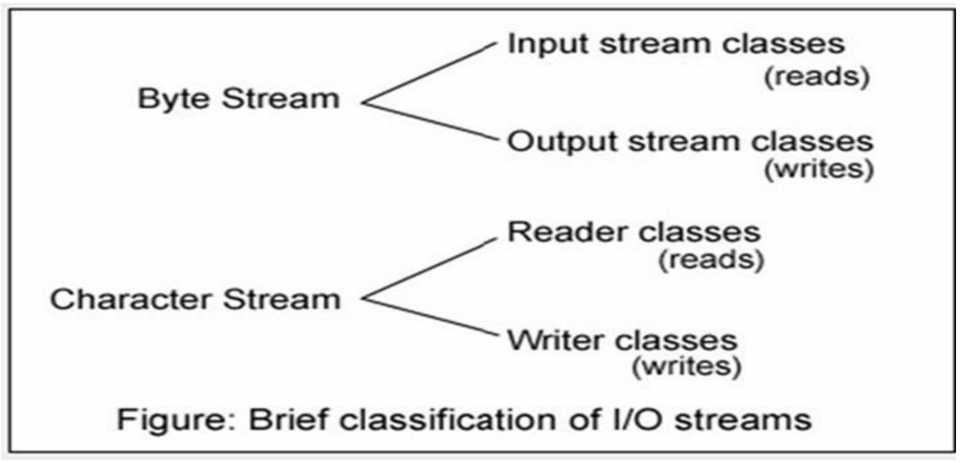


Figure: Brief classification of I/O streams

Some important Byte stream classes:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

Some important Charcter stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

➢ INPUTSTREAM AND OUTPUTSTREAMS:

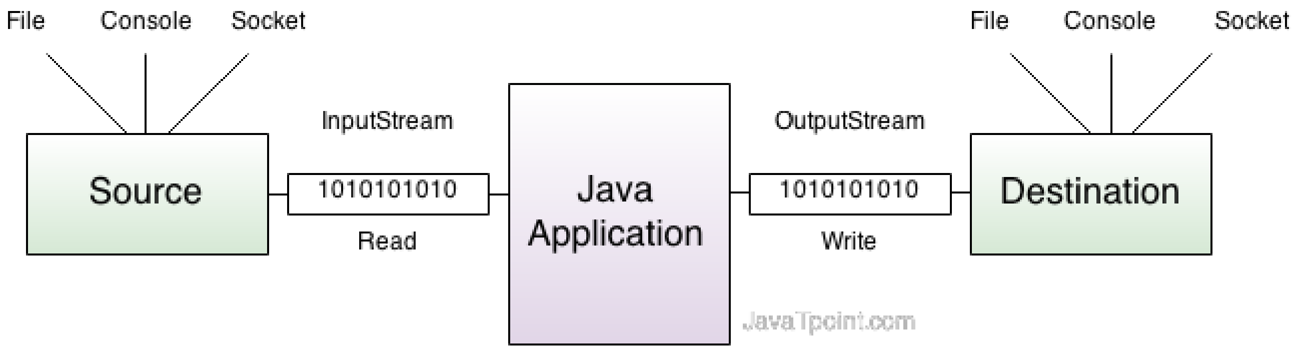
✓ **OutputStream**

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

✓ **InputStream**

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

Working of Java OutputStream and InputStream by the figure given below.



OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Commonly used methods of OutputStream class

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of InputStream class

Method	Description
1) public abstract int read() throws IOException	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available() throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

1. FileInputStream and FileOutputStream (File Handling):

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

✓ **FileOutputStream class**

Java FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
void close()	It is used to closes the file output stream.

Example of Java FileOutputStream class

```
import java.io.*;
class Test{
 public static void main(String args[]){
  try{
    FileOutputstream fout=new FileOutputStream("abc.txt");
    String s="java is my favourite language";
    byte b[]=s.getBytes();//converting string into byte array
    fout.write(b);
    fout.close();
    System.out.println("success...");
  }catch(Exception e){system.out.println(e);}
}
}
```

Output:success...

✓ **FileInputStream class**

Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc.

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

Example of FileInputStream class

```
import java.io.*;
class SimpleRead{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("abc.txt");
    int i=0;
    while((i=fin.read())!=-1){
    System.out.println((char)i);
    }
    fin.close();
  }catch(Exception e){system.out.println(e);}
 }
}
```

Output: java is my favourite language

2. BufferedOutputStream and BufferedInputStream

✓ **BufferedOutputStream class**

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
import java.io.*;
class Test{
 public static void main(String args[])throws Exception{
   FileOutputStream fout=new FileOutputStream("f1.txt");
   BufferedOutputStream bout=new BufferedOutputStream(fout);
   String s="Java is my favourite language";
   byte b[]=s.getBytes();
   bout.write(b);
   bout.flush();
   bout.close();
   fout.close();
   System.out.println("success");
 }
}
```

Output: success...

✓ **BufferedInputStream class**

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.

int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

<u>Example of Java BufferedInputStream</u>

```
import java.io.*;
class SimpleRead{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("f1.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.println((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){system.out.println(e);}
 }
}
```

Output: Java is my favourite language

3. DataInputStream and DataOutputStream:

✓ **DataInputStream class**

DataInputStream class allows the programmer to read primitive data from the input source.

Method	Description
int read(byte[] b)	It is used to read the number of bytes from the input stream.
int readInt()	It is used to read input bytes and return an int value.
byte readByte()	It is used to read and return the one input byte.
char readChar()	It is used to read two input bytes and returns a char value.
double readDouble()	It is used to read eight input bytes and returns a double value.
boolean readBoolean()	It is used to read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	It is used to skip over x bytes of data from the input stream.
void readFully(byte[] b)	It is used to read bytes from the input stream and store them into the buffer array.
void readFully(byte[] b, int off, int len)	It is used to read len bytes from the input stream.

✓ **DataOutputStream class**

The DataOutputStream stream let you write the primitives to an output source.

Example:

Following is the example to demonstrate DataInputStream and DataInputStream. This example reads 5 lines given in a file test.txt and converts those lines into capital letters and finally copies them into another file test1.txt.

Method	Description
int size()	It is used to return the number of bytes written to the data output stream.
void write(int b)	It is used to write the specified byte to the underlying output stream.
void writeChar(int v)	It is used to write char to the output stream as a 2-byte value.
void writeChars(String s)	It is used to write string to the output stream as a sequence of characters.

void writeByte(int v)	It is used to write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	It is used to write string to the output stream as a sequence of bytes.
void writeInt(int v)	It is used to write an int to the output stream
void writeShort(int v)	It is used to write a short to the output stream.
void writeShort(int v)	It is used to write a short to the output stream.
void writeLong(long v)	It is used to write a long to the output stream.
void flush()	It is used to flushes the data output stream.

Test.txt

```
this is test 1 ,
this is test 2 ,
this is test 3 ,
this is test 4 ,
this is test 5  ,
```

test.java

```java
import java.io.*;
public class Test{
  public static void main(String args[])throws IOException{
    DataInputStream d = new DataInputStream(new FileInputStream("test.txt"));
    DataOutputStream out = new DataOutputStream(new FileOutputStream("test1.txt"));
    String count;
    while((count = d.readLine()) != null){
      String u = count.toUpperCase();
      System.out.println(u);
      out.writeBytes(u + "  ,");}
    d.close();
    out.close();
  }}
```

Output:

```
THIS IS TEST 1 ,
THIS IS TEST 2 ,
THIS IS TEST 3 ,
THIS IS TEST 4 ,
THIS IS TEST 5  ,
```

4. PrintStream

The **PrintStream** class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class:

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.

- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class:

In this example, we are simply printing integer and string values.

```
import java.io.*;
class PrintStreamTest{
 public static void main(String args[])throws Exception{
   FileOutputStream fout=new FileOutputStream("mfile.txt");
   PrintStream pout=new PrintStream(fout);
   pout.println(1900);
   pout.println("Hello Java");
   pout.println("Welcome to Java");
   pout.close();
   fout.close();
 }
}
```

Example of printf() method of java.io.PrintStream class:

Example of printing integer value by format specifier:

```
class PrintStreamTest{
 public static void main(String args[]){
   int a=10;
 System.out.printf("%d",a);//Note, out is the object of PrintStream class
 }
}
```

Output:10

➢ **<u>CHARACTER STREAMS (READER & WRITER):</u>**

Java IO's Reader and Writer work much like the InputStream and OutputStream with the exception that Reader and Writer are character based. They are intended for reading and writing text. The InputStream and OutputStream are byte based.

Reader class:

The **Java.io.Writer** class is a abstract class for writing to character streams.

Methods defined by Reader class:

Method	Description
abstract void close()	This method closes the stream and releases any system resources associated with it.
void mark(int numChars)	This method marks the present position in the stream.
boolean markSupported()	This method tells whether this stream supports the mark() operation.
int read()	This method reads a single character.
int read(char buffer[])	This method reads characters into an array.
abstract int read(char buffer[],int offset,int numChars)	This method reads characters into a portion of an array.
boolean ready()	This method tells whether this stream is ready to be read.
void reset()	This method resets the stream.
long skip(long numChars)	This method skips characters.

Writer class:

The Java.io.Writer class is a abstract class for writing to character streams

Methods defined by Writer class:

Method	Description
Writer append(char ch)	This method appends the specified character to this writer.
Writer append(CharSequence chars)	This method appends the specified character sequence to this writer.
Writer append(CharSequence chars, int begin, int end)	This method appends the specified character sequence to this writer.
abstract void close()	This method loses the stream, flushing it first.
abstract void flush()	This method flushes the stream.
void write(int ch)	This method writes a single character.
void write(char buffer[])	This method writes an array of characters.

1. Java FileWriter and FileReader (File Handling in java)

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.
Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

➢ **Java FileWriter class**

Java FileWriter class is used to write character-oriented data to the file.
Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	creates a new file. It gets file name in string.
FileWriter(File file)	creates a new file. It gets file name in File object.

Methods of FileWriter class

Method	Description
1) public void write(String text)	writes the string into FileWriter.
2) public void write(char c)	writes the char into FileWriter.
3) public void write(char[] c)	writes char array into FileWriter.
4) public void flush()	flushes the data of FileWriter.
5) public void close()	closes FileWriter.

➢ **Java FileReader class**

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

Constructors of FileWriter class

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Methods of FileReader class

Method	**Description**
public int read()	returns a character in ASCII form. It returns -1 at the end of file.
public void close()	closes FileReader.

2. BufferedReader and BufferedWriter classes:

➢ **BufferedWriter class:**

This can be used for writing character data to the file.

Constructors

BufferedWriter bw = new BufferedWriter(writer w)
BufferedWriter bw = new BufferedWriter(writer r, int size)

BufferedWriter never communicates directly with the file. It should be communicate through some writer object only.

Important methods of BufferedWriter Class

void write(int ch) thorows IOException
void write(String s) throws IOException
void write(char[] ch) throws IOException
void newLine() for inserting a new line character.
void flush()
void close()

➢ **BufferedReader**

BufferedReader class can read character data from the file.

Constructors

1. BufferedReader br = new BufferedReader(Reader r)
2. BufferedReader br = new BufferedReader(Reader r, int buffersize)
3. BufferedReader never communicates directly with the file. It should Communicate through some reader object only.

Important methods of BufferedReader Class

1. int read()
2. int read(char [] ch)
3. String readLine(); - Reads the next line present in the file. If there is no nextline this method returns null.
4. void close()

Example for Java FileWriter and FileReader , BufferedReader and BufferedWriter classes:

```
import java.io.*;
class Simple{
public static void main(String args[]){
try{
        FileWriter fw=new FileWriter("d:/archana/abc.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write(" Java");
        bw.close();
        fw.close();
         FileReader fr=new FileReader("d:/archana/abc.txt");
         BufferedReader br = new BufferedReader(fr);
          int i;
          while((i=br.read())!=-1)
          System.out.print((char)i);
          br.close();
         fr.close();
         }catch(Exception e){System.out.println(e);}
         System.out.println("success");
        } }
```

Output

```
Java
success
```

3. InputStreamReader and OutputStreamWriter classes:

➢ **OutputStreamWriter**

OutputStreamWriter behaves as a bridge to transfer data from character stream to byte stream. It uses default charset or we can specify charset for change in character stream to byte stream.

Constructors

1. OutputStreamWriter(OutputStream out)
2. OutputStreamWriter(OutputStream out, Charset cs)
3. OutputStreamWriter(OutputStream out, CharsetEncoder enc)
4. OutputStreamWriter(OutputStream out, String charsetName)

Important methods of OutputStreamWriter

1. void close()
2. void flush()
3. String getEncoding()
4. void write(int c)
5. void write(String str, int off, int len)

OutputStreamWriterDemo.java

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class OutputStreamWriterDemo {
        public static void main(String[] args) {
    String str = "Hello World! \nThis is OutputStreamWriter Code Example.”
    BufferedWriter bw = null;
    try {
     Writer w = new OutputStreamWriter(System.out);
     bw = new BufferedWriter(w);
     bw.write(str);
    } catch (IOException e) {
      e.printStackTrace();
    }finally {
       try {
         bw.close();
        } catch (IOException ex) {
         ex.printStackTrace();
        }
     }
  }
}
```

Output

```
Hello World!
This is OutputStreamWriter Code Example.
```

➢ <u>InputStreamReader</u>

InputStreamReader behaves as bridge from bytes stream to character stream. It also uses charset to decode byte stream into character stream.

Constructors

1. InputStreamReader(InputStream in_strm)
2. InputStreamReader(InputStream in_strm, Charset cs)
3. InputStreamReader(InputStream in_strm, CharsetDecoder dec)
4. InputStreamReader(InputStream in_strm, String charsetName)

Important methods of InputStreamReader

1. public boolean ready() – tells whether the character stream is ready to be read or not.
2. public void close() – closes InputStreamReader and releases all the Streams associated with it.
3. public int read() – returns single character after reading.
4. public String getEncoding() – returns the name of the character encoding being used by this stream.

InputStreamReaderDemo.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class InputStreamReaderDemo {
        public static void main(String[] args) {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                int a=0;
                int b=0;
                try {
                        System.out.println("Enter a number..");
                        a = Integer.parseInt(br.readLine());
                        System.out.println("Enter another number..");
                         b = Integer.parseInt(br.readLine());
                } catch (NumberFormatException e) {
                                e.printStackTrace();
                } catch (IOException e) {
                                e.printStackTrace();
                }
                System.out.println("you entered "+a+" and  "+b); }}
```

Output

```
Enter a number..
10
Enter another number..
14
you entered 10 and  14
```

4. PrintWriter Class

The **Java.io.PrintWriter** class prints formatted representations of objects to a text-output stream.

✓ **PrintWriter** defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a newline ('**\\n**') character is output. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

✓ **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**. Thus, you can use these methods in the same way as they have been used with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString( )** method and then print the result.

✓ To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline. For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d); } }
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

4.2: READING AND WRITING CONSOLE

3 Ways to read input from console in Java

1. Using Buffered Reader Class

Advantages

The input is buffered for efficient reading.

Drawback:

The wrapping code is hard to remember.

2. Using Scanner Class

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.
- Regular expressions can be used to find tokens.

Drawback:

The reading methods are not synchronized

3. Using Console Class

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

Does not work in non-interactive environment (such as in an IDE).

Java Console Class

The **Java Console class** is be used to get input from console. It provides methods to read texts and passwords.
If you read password using Console class, it will not be displayed to the user.
The java.io.Console class is attached with system console internally.

Let's see a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console class declaration

public final class Console **extends** Object **implements** Flushable

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

public static Console console(){}

Let's see the code to get the instance of Console class.

Console c=System.console();

Java Console Example

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

Output

```
Enter your name: abcd
Welcome abcd
```

Java Console Example to read password

```
import java.io.Console;
class ReadPasswordTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter password: ");
char[] ch=c.readPassword();
String pass=String.valueOf(ch);//converting char array into string
System.out.println("Password is: "+pass);
}
}
```

Output

```
Enter password:
Password is: 123
```

4.3: READING AND WRITING FILES

What is File Handling in Java?

- ✓ File handling in Java implies reading from and writing data to a file.
- ✓ The File class from the **java.io package**, allows us to work with different formats of files.
- ✓ In order to use the File class, you need to create an object of the class and specify the filename or directory name.

For example:

1) // Import the File class

2) import java.io.File
3) // Specify the filename
4) File obj = new File("filename.txt");

Java uses the concept of a stream to make I/O operations on a file.

The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

➢ **<u>File Operations in Java</u>**

Basically, you can perform four operations on a file. They are as follows:

1) Create a File
2) Get File Information
3) Write To a File
4) Read from a File

1) <u>Create a File</u>

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists.

Example:

```
import java.io.File;
import java.io.IOException;

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
File created: filename.txt
```

2) Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when we are done writing to the file, we should close it with the close() method:

Example:

```
import java.io.FileWriter;  // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors
public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
```

```
        System.out.println("Successfully wrote to the file.");
      } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
      }
    }
  }
```

Output:

```
Successfully wrote to the file.
```

3) Read a File

In the following example, we use the Scanner class to read the contents of the text file we created in the previous example:

Example:

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files
public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }}}
```

Output:

```
Files in Java might be tricky, but it is fun enough!
```

4) Get File Information

To get more information about a file, use any of the File methods:

Example:

```
import java.io.File; // Import the File class
public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

Output:

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```

4.4: Generic Programming

Generic programming is a style of computer programming in which algorithms are written in terms of "**to-be-specified-later"** types that are then instantiated when needed for specific types provided as parameters.
Generic programming refers to writing code that will work for many types of data.

NON-GENERICS:

In java, there is an ability to create generalized classes, interfaces and methods by operating through Object class.

Example:

```
class NonGen
{
    Object ob;
    NonGen(Object o)
    {
                                            ob=o;
    }
    Object getob()
    {

      return ob;
    }
    void showType()
    {
       System.out.println("Type of ob is "+ob.getClass().getName());
    }
}
public class NonGenDemo
{
    public static void main(String[] arg)
    {
        NonGen integerObj;
        integerObj=new NonGen(88);
        integerObj.showType();
        int v=(Integer)integerObj.getob(); // casting required
        System.out.println("Value = "+v);
        NonGen strObj=new NonGen("Non-Generics Test");
        strObj.showType();
        String str=(String)strObj.getob(); // casting required
        System.out.println("Vlaue = "+str);
    }
}
```

Output:

```
Type of ob is java.lang.Integer
Value = 88
Type of ob is java.lang.String
Vlaue = Non-Generics Test
```

Limitation of Non-Generic:

1) Explicit casts must be employed to retrieve the stored data.
2) Type mismatch errors cannot be found until run time.

Need for Generic:

1) It saves the programmers burden of creating separate methods for handling data belonging to different data types.
2) It allows the code reusability.
3) Compact code can be created.

4 (i)

Advantage of Java Generics (Motivation for Java Generics):

1) **Code Reuse:** We can write a method/class/interface once and use for any type we want.
2) **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
3) **Elimination of casts:** There is no need to typecast the object.

   The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

   When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

4) **Stronger type checks at compile time:**

   A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);                   //Compile Time Error
```

5) **Enabling programmers to implement generic algorithms.**

   By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

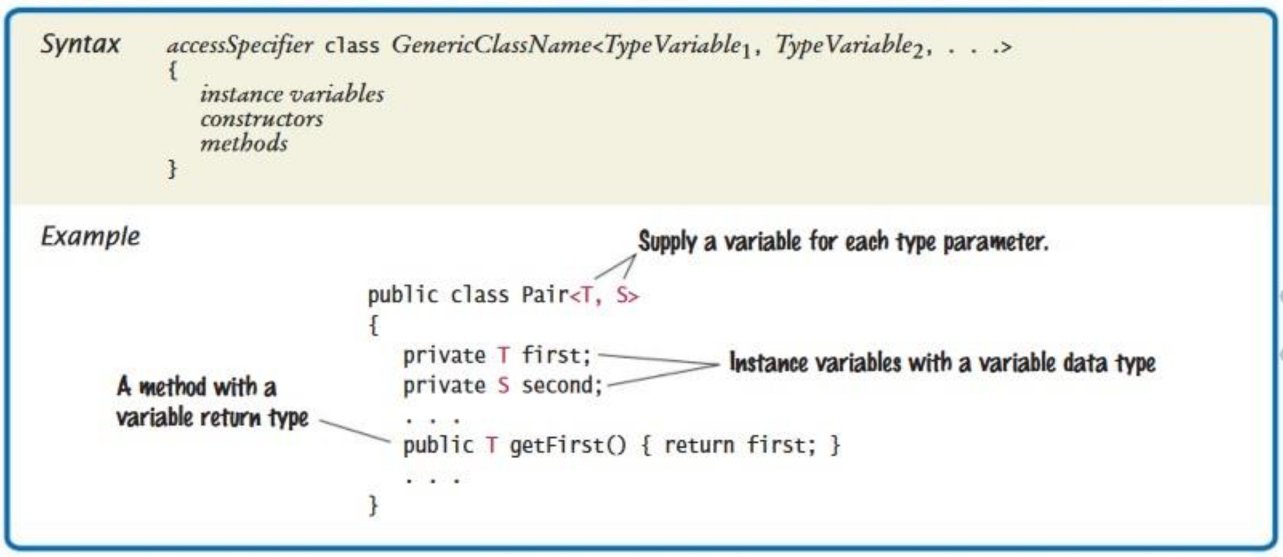
2 (b(i))

4.5: GENERIC CLASSES

A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

Syntax Declaring a Generic Class



Syntax

```
accessSpecifier class GenericClassName<TypeVariable1, TypeVariable2, . . .>
{
   instance variables
   constructors
   methods
}
```

Example

```
public class Pair<T, S>
{
   private T first;
   private S second;
   . . .
   public T getFirst() { return first; }
   . . .
}
```

Supply a variable for each type parameter.
Instance variables with a variable data type
A method with a variable return type

Where, the type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*)

<u>Example:</u>

public class Pair<T, S>
{
. . .
}

Purpose: To define a generic class with methods and fields that depends on type variables.

<u>Class reference declaration:</u>

To instantiate this class, use the new keyword, as usual, but place **<type_parameter>** between the class name and the parenthesis:

class_name<type-arg-list> var-name=new class_name<type-arg-list>(cons-arg-list);

Type Parameter Naming Conventions:

- ✓ Type parameter is a place holder for a type argument.
- ✓ By convention, type parameter names are single, uppercase letters.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Example: Generic class with single type parameter

```
class Gen <T>
{
T obj;
Gen(T x)
{
    obj= x;
}

T show()
{
    return obj;
}

void disp()
{
    System.out.println(obj.getClass().getName());
}
}

public class Test
{
public static void main (String[] args)
{
  Gen < String> ob = new Gen<>("java programming with Generics");
  ob.disp();
  System.out.println("value :  " +ob.show());

  Gen < Integer> ob1 = new Gen<>(550);
  ob1.disp();
  System.out.println("value  :" +ob1.show());
}
}
```

Output:

```
java.lang.String
value :  java programming with Generics
java.lang.Integer
value  :550
```

Example: Generic class with more than one type parameter

In Generic parameterized types, we can pass more than 1 data type as parameter. It works the same as with one parameter Generic type.

```
class Gen <T1,T2>
{
T1 obj1;
T2 obj2;
Gen(T1 o1,T2 o2)
{
  obj1 = o1;
  obj2 = o2;
}
 T1 get1()
{
  return obj1;
}
T2 get2()
{
    return obj2;
}
void disp()
{
   System.out.println(obj1.getClass().getName());
  System.out.println(obj2.getClass().getName());
}
}

public class Test
{
public static void main (String[] args)
{
  Gen < String, Integer> obj = new Gen<>("java programming with Generics",560);
  obj.disp();
  System.out.println("value 1 : " +obj.get1());
  System.out.println("value 2:  "+obj.get2());

Gen < Integer, Integer> obje = new Gen<>(1000,560);
  obje.disp();
  System.out.println("value 1 : " +obje.get1());
  System.out.println("value 2:  "+obje.get2());
}
}
```

Output:

```
java.lang.String
java.lang.Integer
value 1 :  java programming with Generics
value 2:  560
java.lang.Integer
java.lang.Integer
value 1 :  1000
value 2:  560
```

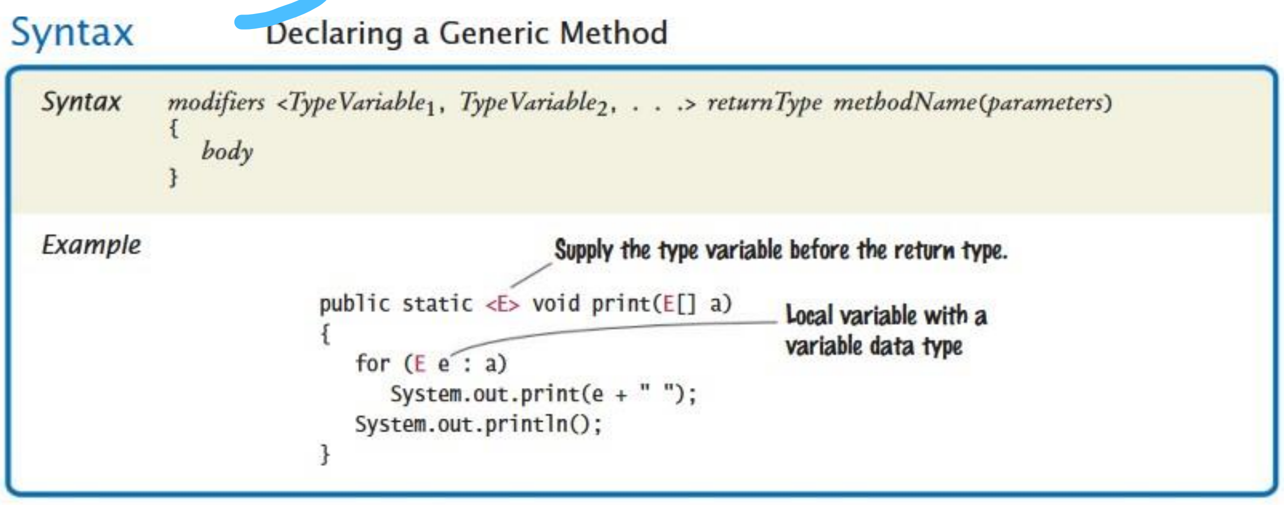
4.6: GENERIC METHODS

A Generic Method is a method with type parameter. We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

6(ii)

Rules to define Generic Methods

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.
- ✓ Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- ✓ The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- ✓ A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).



Syntax Declaring a Generic Method

Syntax *modifiers <TypeVariable$_1$, TypeVariable$_2$, . . .> returnType methodName(parameters)*

```
{
   body
}
```

Example

Supply the type variable before the return type.

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Local variable with a variable data type

<u>Example:</u> (To iterate through the list and display the element using generic method)

```
class a < T >
{
            <T> void show(T[] el)
            {
              for(T x:el)
        System.out.println(x);
  }
}
```

```
public class GenMethod
{
   public static void main(String arg[])
   {
                System.out.println("Integer array");
                a<Integer> o1=new a<Integer>();
                Integer[] ar={10,67,23};
                o1.show(ar);

                System.out.println("String array");
                a<String> o2=new a<String>();
                String[] ar1={"Hai","Hello","Welcome","to","Java  programming"};
                o2.show(ar1);

              System.out.println("Boolean array");
              a<Boolean> o3=new a<Boolean>();
              Boolean[] ar2={true,false};
              o3.show(ar2);

              System.out.println("Double array");
              a<Double> o4=new a<Double>();
              Double[] ar3={10.234,67.451,23.90};
              o4.show(ar3);
   }
}
```

Output:

```
Integer array
10
67
23
String array
Hai
Hello
Welcome
to
Java  programming
```

```
Boolean array
true
false
Double array
10.234
67.451
23.9
```

[illegible]

4.7: GENERICS WITH BOUNDED TYPES

GENERICS WITH BOUNDED TYPE PARAMETERS:

Bounded Type Parameter is a type parameter with one or more bounds. Thebounds restrict the set of types that can be used as type arguments and give accessto the methods defined by the bounds.
For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

Syntax:

```
<T extends superclass>
```

Example:

The following example creates a generic class that contains a method that returns the average of array of any type of numbers. The type of the numbers is represented generically using Type Parameter.

```
public class GenBounds<T extends Number>
{
   T[]  nums;
   GenBounds(T[] obj)
   {
      nums=obj;
    }
   double average()
   {
      double sum=0.0;
      for(int i=0;i<nums.length;i++)
```

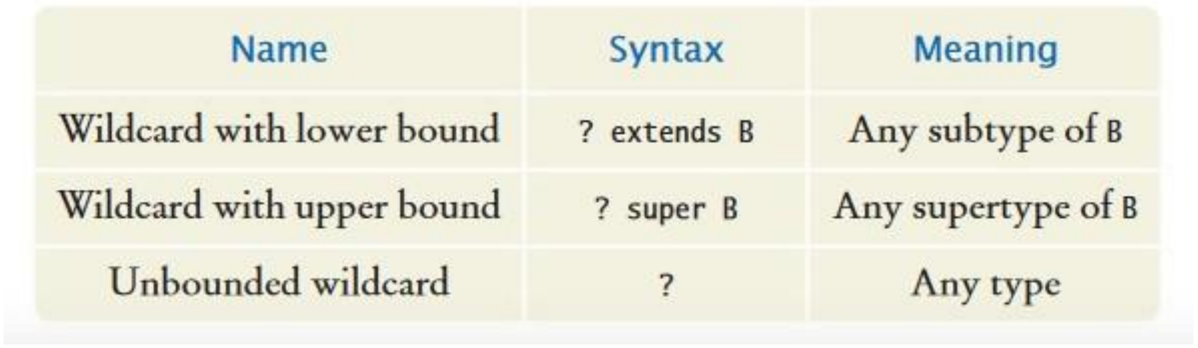
```
        sum+=nums[i].doubleValue();
      double avg=sum/nums.length;
      return avg;
   }
   public static void main(String[] args)
   {
      Integer inum[]={1,2,3,4,5};
      GenBounds<Integer> iobj=new GenBounds<Integer>(inum);
      System.out.println("Average of Integer Numbers : "+iobj.average());

      Double dnum[]={1.1,2.2,3.3,4.4,5.5};
      GenBounds<Double> dobj=new GenBounds<Double>(dnum);
      System.out.println("Average of Double Numbers : "+dobj.average());

       /* Error: java,lang.String not within bound
      String snum[]={"1","2","3","4","5"};
      GenBounds<String> sobj=new GenBounds<String>(snum);
      System.out.println("Average of Integer Numbers : "+iobj.average()); */
  }
}
```

Output:

```
F:\>java GenBounds
Average of Integer Numbers : 3.0
Average of Double Numbers : 3.3
```

Wild Card Arguments:

Question mark (?) is the wildcard in generics and represents an unknown type. The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.



Name	Syntax	Meaning
Wildcard with lower bound	? extends B	Any subtype of B
Wildcard with upper bound	? super B	Any supertype of B
Unbounded wildcard	?	Any type

Example: BOUNDED WILDCARDS:-

A bounded wildcard is a wildcard with either an upper or a lower bound.

The following program illustrates the use of wildcards with upper bound. In below method we can use all the methods of upper bound class Number.

```
import java.util.ArrayList;
import java.util.List;
public class GenericsWildcards
{
      public static void main(String[] args)
       {
          List<Integer> ints = new ArrayList<Integer>();
                         ints.add(3);
      ints.add(5);
      ints.add(10);
          double sum = sum(ints);
          System.out.println("Sum of ints="+sum);
      }

      // here Number is the upper bound for the type parameter
      public static double sum(List<? extends Number>  list)
       {
          double sum = 0;
          for(Number n : list)
      {
                         sum += n.doubleValue();
          }
          return sum;
      }
}
```

Output:

```
F:\>java GenericsBounds
Sum of ints=18.0
```

Example: UNBOUNDED WILDCARD:-

Sometimes we have a situation where we want our generic method to be working with all types; in this case unbounded wildcard can be used. The wildcard "?" simply matches any valid objects.

- ✓ Its same as using <? extends Object>.

```
import java.util.*;
public class GenUBWildcard
{
   public static void main(String[] args)
   {
     List<Integer> ints = new ArrayList<Integer>();
         ints.add(3);
        ints.add(5);
        ints.add(10);
     printData(ints);

     List<String> str = new ArrayList<String>();
         str.add("\nWelcome");
        str.add(" to ");
        str.add(" JAVA ");
     printData(str);
   }
   public static void void printData(List<?> list)
   {
         for(Object obj : list)
            {
                  System.out.print(obj + "\n");
         }}}
```

Output:

```
F:\>java GenUBWildcard
3
5
10
Welcome
 to
 JAVA
```

4.8: RESTRICTIONS AND LIMITATIONS OF GENERICS

1) In Java, generic types are compile time entities. The runtime execution is possible only if it is used along with raw type.
2) Primitive type parameters are not allowed for generic programming.

   For example:

   Stack<int> is not allowed.
3) For the instances of generic class throw and catch keywords are not allowed.

   For example:

```
public class Test<T> extends Exception
{
// code                // Error: can't extend the Exception class
}
```

4) Instantiation of generic parameter T is not allowed.

   For Example:

```
new T();      // Error
new T[10];    // Error
```

5) Arrays of parameterized types are not allowed.

   For Example:

```
New Stack<String>[10]; // Error
```

4.9: STRINGS

Definition:

String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

There are two ways to create a String object:

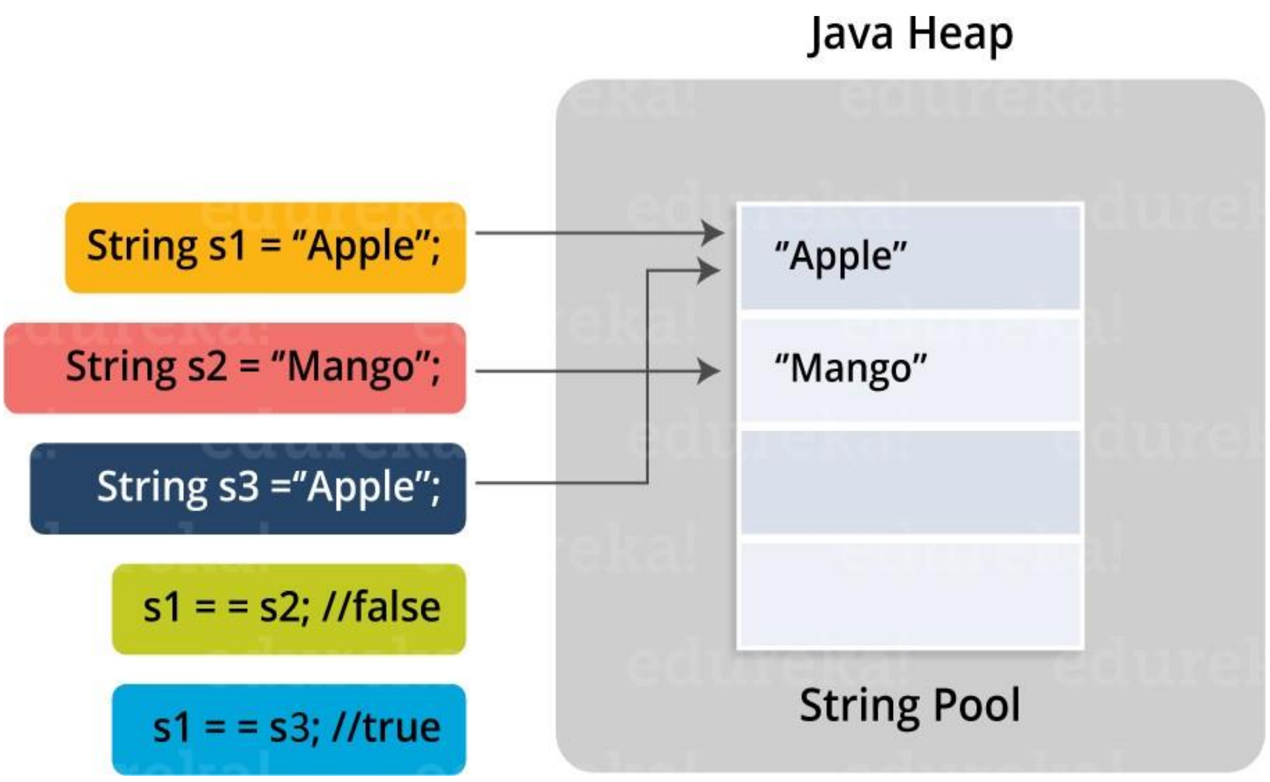
1. **By string literal**: Java String literal is created by using double quotes. For Example: String s="Welcome";

2. **By new keyword**: Java String is created by using a keyword "new".
   For example: String s=new String("Welcome");
   It creates two objects (in String pool and in heap) and one reference variable where the variable 's' will refer to the object in the heap.

Java String Pool:

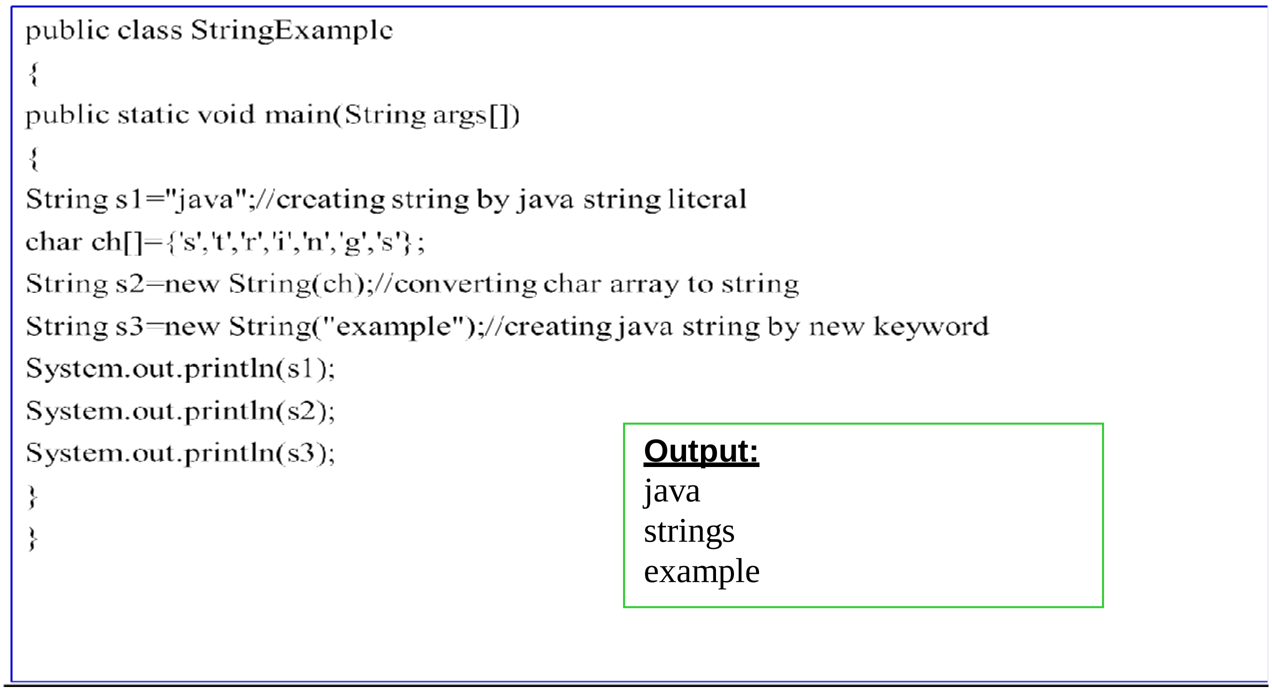
Java String pool refers to collection of Strings which are stored in heap memory.
In this, whenever a new object is created,

1) String pool first checks whether the object is already present in the pool or not.
2) If it is present, then same reference is returned to the variable
3) else new object will be created in the String pool and the respective reference will be returned.



Refer to the diagrammatic representation for better understanding: In the above image, two Strings are created using literal i.e "Apple" and "Mango". Now, when third String is created with the value "Apple", instead of creating a new object, the already present object reference is returned.

Example: Creating Strings



```
public class StringExample
{
public static void main(String args[])
{
String s1="java";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}
```

Output:

```
java
strings
example
```

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means un-modifiable or unchangeable.

- ✓ Once string object is created its data or state can't be changed but a new string object is created.
- ✓ Let's try to understand the immutability concept by the example given below:

```
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 immutable objec
ts
 }
}
```

Output: Sachin

4.10: METHODS

Methods of String class in Java

java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting strings etc.

Important methods of String class.

S.No.	Method	Description
1)	public boolean equals(Object anObject)	Compares this string to the specified object.
2)	public boolean equalsIgnoreCase(String another)	Compares this String to another String, ignoring case.
3)	public String concat(String str)	Concatenates the specified string to the end of this string.
4)	public int compareTo(String str)	Compares two strings and returns int
5)	public int compareToIgnoreCase(String str)	Compares two strings, ignoring case differences.
6)	public String substring(int beginIndex)	Returns a new string that is a substring of this string.
7)	public String substring(int beginIndex,int endIndex)	Returns a new string that is a substring of this string.
8)	public String toUpperCase()	Converts all of the characters in this String to upper case
9)	public String toLowerCase()	Converts all of the characters in this String to lower case.
10)	public String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
11)	public boolean startsWith(String prefix)	Tests if this string starts with the specified prefix.
12)	public boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.

13)	public char charAt(int index)	Returns the char value at the specified index.
14)	public int length()	Returns the length of this string.
15)	public String intern()	Returns a canonical representation for the string object.
16)	public byte[] getBytes()	Converts string into byte array.
17)	public char[] toCharArray()	Converts string into char array.
18)	public static String valueOf(int i)	converts the int into String.
19)	public static String valueOf(long i)	converts the long into String.
20)	public static String valueOf(float i)	converts the float into String.
21)	public static String valueOf(double i)	converts the double into String.
22)	public static String valueOf(boolean i)	converts the boolean into String.
23)	public static String valueOf(char i)	converts the char into String.
24)	public static String valueOf(char[] i)	converts the char array into String.
25)	public static String valueOf(Object obj)	converts the Object into String.
26)	public void replaceAll(String firstString,String secondString)	Changes the firstString with secondString.

➢ String comparison in Java

- ✓ We can compare two given strings on the basis of content and reference.
- ✓ It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.
- ✓ There are three ways to compare String objects:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) By equals() method

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: equals() method

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

   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";

   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 }
}
```

Output:

```
true
true
false
```

Example: equalsIgnoreCase(String) method

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

   String s1="Sachin";
   String s2="SACHIN";

   System.out.println(s1.equals(s2));//false
   System.out.println(s1.equalsIgnoreCase(s3));//true
 }
}
```

Output:

```
false
 true
```

2) By == operator

The = = operator compares references not values.

Example: == operator:

```
class Simple{
 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 nonpoo
l)
 }

}
```

Output:

```
true
false
```

3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two string variables.If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

Example: compareTo() method:

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

   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
```

```
    System.out.println(s1.compareTo(s2));//0
    System.out.println(s1.compareTo(s3));//1(because s1>s3)
    System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
  }
 }
```

Output:

```
0
1
-1
```

- **String Concatenation in Java**

Concating strings form a new string i.e. the combination of multiple strings.

There are two ways to concat string objects:

1. By + (string concatenation) operator
2. By concat() method

1) By + (string concatenation) operator
String concatenation operator is used to add strings.For Example:

```
//Example of string concatenation operator

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

   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 }
}
```

Output: Sachin Tendulkar

The compiler transforms this to:

String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();

String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For

Example:

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

   String s=50+30+"Sachin"+40+40;
   System.out.println(s);//80Sachin4040
 }
}
```

Output: 80Sachin4040

Note: If either operand is a string, the resulting operation will be string concatenation. If both operands are numbers, the operator will perform an addition.

2) By concat() method

concat() method concatenates the specified string to the end of current string.

Syntax: public String concat(String another){}

Example of concat(String) method

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

   String s1="Sachin ";
   String s2="Tendulkar";

   String s3=s1.concat(s2);

   System.out.println(s3);//Sachin Tendulkar
  }
}
```

Output: Sachin Tendulkar 21

➢ **<u>Example Program: Using all the methods of String class</u>**

```
class Simple{
public static void main(String args[])
{
String s="Sachin Tendulkar";
System.out.println("Substring 1: "+s.substring(6));
System.out.println("Substring2: "+s.substring(0,6));
System.out.println("Uppercase: "+s.toUpperCase());
System.out.println("Lowercase: "+s.toLowerCase());
System.out.println("Trim: "+s.trim());
System.out.println("Start With: "+s.startsWith("Sa"));
System.out.println("End with: "+s.endsWith("n"));
System.out.println("Char at Position 0: "+s.charAt(0));
System.out.println("Char at Position 3: "+s.charAt(3))
System.out.println("Length: "+s.length());
String s2=s.intern();
System.out.println("Intern: "+s2);
System.out.println("Replace: "+s.replace('a','q'));
System.out.println("Index 1: "+s.indexOf('I'));
System.out.println("Index 2: "+s.indexOf('I',5));
 }
}
```

```
Substring 1: Tendulkar
Substring2: Sachin
Uppercase: SACHIN TENDULKAR
Lowercase: sachin tendulkar
Trim: Sachin Tendulkar
Start  With:  true
End  with:  false
Char at Position 0: S
Char at Position 3: h
Length: 16
Intern: Sachin Tendulkar
Replace: Sqchin Tendulkqr
Index 1: -1
Index 2: -1
```

4.11: STRING BUFFER CLASS

❖ StringBuffer CLASS

The StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class is same as String except it is mutable i.e. it can be changed.
StringBuffer can be changed dynamically. String buffers are preferred when heavy modification of character strings is involved (appending, inserting, deleting, modifying etc).

Difference between String class and StringBuffer class:

S.No	String Class	StingBuffer Class
1	String objects are constants and immutable(cannot change the content)	StringBuffer objects are not constants and mutable(can change the content)
2	String class supports constant strings	StringBuffer class supports growable and modifiable strings
3	The methods in the String class are not synchronized	The methods of the StringBuffer class can be synchronized

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16. **Example:** StringBuffer s=**new** StringBuffer();
StringBuffer(String str)	creates a string buffer with the specified string. **Example:** StringBuffer s=**new** StringBuffer("Alice in Wonderland");
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length. **Example:** StringBuffer s=**new** StringBuffer(20);

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.

public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e.total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Example:

```
public class StringBufferFunctionsDemo {

        public static void main(String[] args) {
//      Examples of Creation of Strings
        StringBuffer strBuf1 = new StringBuffer("Bobby");
        StringBuffer strBuf2 = new StringBuffer(100); //With capacity 100
        StringBuffer strBuf3 = new StringBuffer(); //Default Capacity 16
        System.out.println("strBuf1 : " + strBuf1);
        System.out.println("strBuf1 capacity : " + strBuf1.capacity());
        System.out.println("strBuf2 capacity : " + strBuf2.capacity());
        System.out.println("strBuf3 capacity : " + strBuf3.capacity());
        System.out.println("strBuf1 length : " + strBuf1.length());
        System.out.println("strBuf1 charAt 2 : " + strBuf1.charAt(2));
        // A StringIndexOutOfBoundsException is thrown if the index is not valid.
        strBuf1.setCharAt(1, 't');
        System.out.println("strBuf1 after setCharAt 1 to t is : "+ strBuf1);
        System.out.println("strBuf1 toString() is : " + strBuf1.toString());
        strBuf3.append("beginner-java-tutorial");
        System.out.println("strBuf3 when appended with a String : "+ strBuf3.toString());
        strBuf3.insert(1, 'c');
        System.out.println("strBuf3 when c is inserted at 1 : "+ strBuf3.toString());
        strBuf3.delete(1, 'c');
        System.out.println("strBuf3 when c is deleted at 1 : "+ strBuf3.toString());
        strBuf3.reverse();
```

```
		System.out.println("Reversed strBuf3 : " + strBuf3);
		strBuf2.setLength(5);
		strBuf2.append("jdbc-tutorial");
		System.out.println("strBuf2 : " + strBuf2);
		//	We can clear a StringBuffer using the following line
		strBuf2.setLength(0);
		System.out.println("strBuf2 when cleared using setLength(0): "+ strBuf2);
		}
}
```

Output:

```
strBuf1 : Bobby
strBuf1 capacity : 21
strBuf2 capacity : 100
strBuf3 capacity : 16
strBuf1 length : 5
strBuf1 charAt 2 : b
strBuf1 after setCharAt 1 to t is : Btbby
strBuf1 toString() is : Btbby
strBuf3 when appended with a String : beginner-java-tutorial
strBuf3 when c is inserted at 1 : bceginner-java-tutorial
strBuf3 when c is deleted at 1 : b
Reversed strBuf3 : b
strBuf2 : jdbc-tutorial
strBuf2 when cleared using setLength(0):
```