CST281
Object Oriented Programming

MODULE 4

Advanced features of Java

Syllabus

- Input/Output –
  - I/O Basics, Reading Console Input, Writing Console Output, PrintWriter Class, Object Streams and Serialization, Reading and Writing Files.
- Java Library
  - String Handling – String Constructors, String Length, Special String Operations Character Extraction, String Comparison, Searching Strings, Modifying Strings, Using valueOf(), Comparison of StringBuffer and String.
- Collections framework
  - Collections overview, Collections Class – ArrayList. Accessing Collections via an Iterator.

I/O Basics

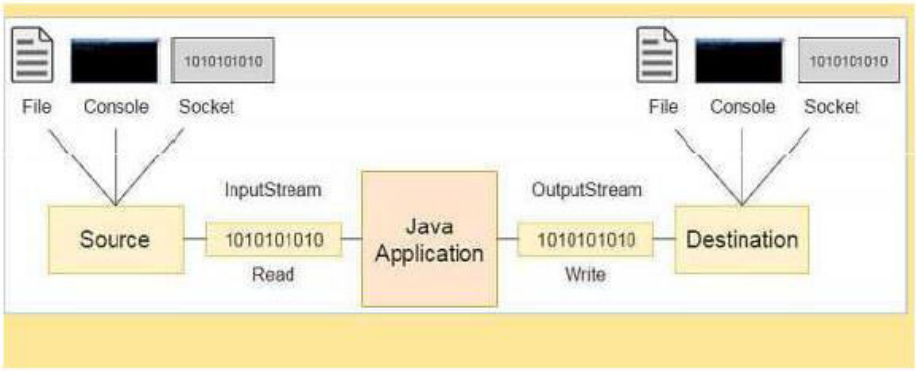
- Only **print( ) and println( ) are used frequently. All** other I/O methods are not used significantly.
  - Because most real applications of Java are not text-based, console programs.
- Java's support for console I/O is limited

Streams

- Java programs perform I/O through streams.
- A stream is an abstraction that either produces or consumes information.
- A stream is a sequence of objects that supports various methods.
- A stream is linked to a physical device by the Java I/O system.
  - Input stream may refer to different kinds of input: from a disk file, a keyboard, or a network socket
  - Output stream may refer to the console, a disk file, or a network connection.

Working of Java I/O stream

- Stream is like a flow of data.



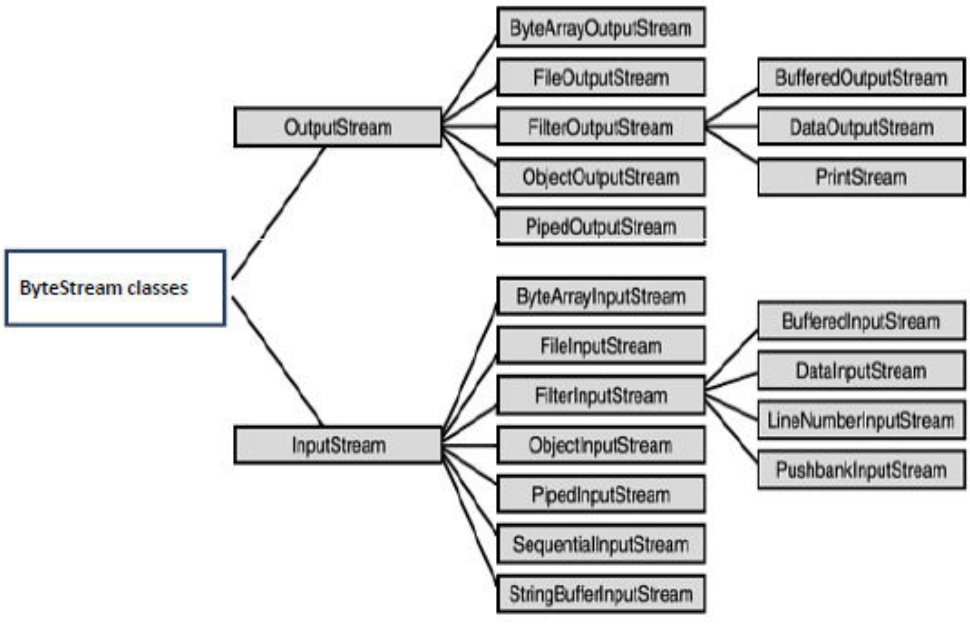
Streams

- The java.io package contains all the classes required for input and output operations.
- Java defines two types of streams: byte and character.
- Byte streams provides a means for handling input and output of bytes.
  - Byte streams are used when reading or writing binary data.
- Character streams provide a means for handling input and output of characters.
  - they use Unicode.
  - in some cases, character streams are more efficient than byte streams.

ByteStream classes

- 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 between various devices, such as disk files, network connections, and even memory buffers.
- Two of the most important are read( ) and write( ),
  - These methods are overridden by derived stream classes.

ByteStream classes

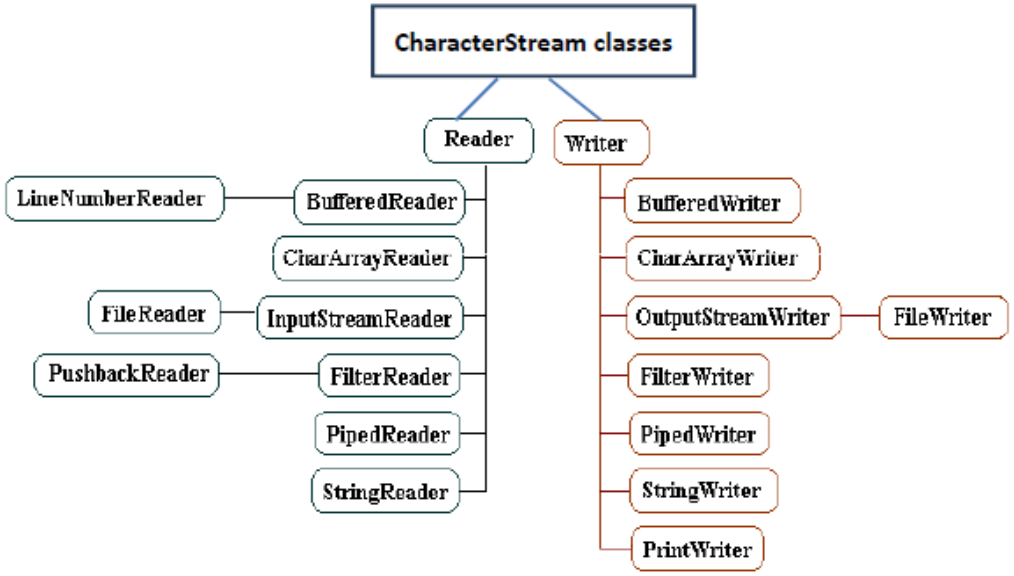


Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the 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
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

TABLE 13-1 The Byte Stream Classes

Character streams

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes,
  - Reader and Writer.
- Java has several concrete subclasses of each of these.
- Two of the most important methods are read( ) and write( ).
  - These methods are overridden by derived stream classes.



11

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
FilterReader	Filtered reader
FilterWriter	Filtered writer

TABLE 13-2 The Character Stream I/O Classes

12

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

TABLE 13-2 The Character Stream I/O Classes *(continued)*

The Predefined Streams

- All Java programs automatically import the java.lang package. This package defines a class called System.
- System contains three predefined stream variables:
  - in, out, and err.
  - These fields are declared as public, static, and final within System.
- System.out refers to the standard output stream.
- System.in refers to standard input, which is the keyboard by default.
- System.err refers to the standard error stream, which is the console by default.
- System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.

Reading Console Input

- The preferred method of reading console input is to use a character-oriented stream.
- In Java, console input is accomplished by reading from System.in.
  - To obtain a character based stream that is attached to the console, **wrap System.in in a BufferedReader object**.

- BufferedReader supports a **buffered input stream.**
  - Its most commonly used constructor is:

BufferedReader(Reader inputReader)

- Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created.
- Reader is an abstract class.

- One of the concrete subclasses of Reader is InputStreamReader.
- InputStreamReader converts bytes to characters.
  - It reads bytes and decodes them into characters using a specified charset.
- To obtain an InputStreamReader object that is linked to System.in, the constructor that can be used is :

```
InputStreamReader(InputStream  inputStream)
```


- Following line of code creates a BufferedReader that is connected to the keyboard:

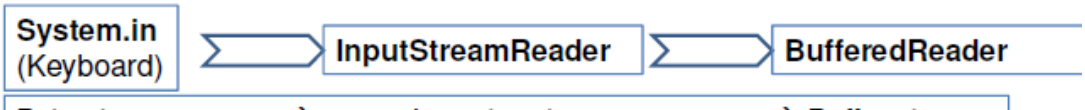
```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

- By wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.
- After this statement executes, br is a character-based stream that is linked to the console through System.in

Reading console-summary

- To accept data from keyboard, we use System.in. (bytestream)
- We need to connect keyboard to an input stream object.
- Here we can use InputStreamReader that can read data from the keyboard
  - convert byte stream to character stream
- Now our data reaches InputSreamReader
- BufferedReader class is used to read the text from a character-based input stream.

- To make program run fast and to make reading efficient , buffering can be done using BufferedReader class. It can read data from stream.
- Create BufferedReader object and connect InputStreamReader object to it




```
InputStreamReaer in = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(in)
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

- **System.in** → keyboard(Byte stream)
- Convert **byte steam** to **character stream** using **InputStreamReader**.
- Then wrap that character stream in a **buffered stream BufferedReader**

Reading Characters

- To read a character from a BufferedReader, use read( ).
- The version of read( ) is

int read( ) throws IOException

- Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value.
- It returns –1 when the end of the stream is encountered

```
import java.io.*;
class Readinp
{
public static void main(String a[]) throws IOException
{
char c;
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter a letter");
c=(char)br.read();
System.out.println("Letter="+c);

}
}
```

OUTPUT
Enter a letter
a
Letter=a

Enter characters one by one when you type q it will stop reading

```
class BRRead {
public static void main(String args[])
throws IOException
{
char c;
BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
    c = (char) br.read();
    System.out.println(c);
    } while(c != 'q');
}
}
```

OUTPUT
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q

No input is actually passed to the program until you press ENTER.

Reading Strings

- To read a string from the keyboard, use the version of **readLine( ) that is a member of the BufferedReader class.**
- Its general form is
  - **String readLine( ) throws IOException**
- This returns a **String object.**


```
import java.io.*;
class Readinp
{
public static void main(String a[]) throws IOException
{
char c;
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter a line of text");
c=(char)br.read();
s=br.readLine();
System.out.println("Line is "+s);
}
}
```

OUTPUT
Enter a line of text
How are you
Line is How are you

Enter lines of text one by one when you type stop it will stop reading


```
import java.io.*;
class Readlinetillstop
{
public static void main(String a[]) throws IOException
{
String s[] = new String[100];

BufferedReader br=new BufferedReader(new  InputStreamReader(System.in));
System.out.println("Enter a line of text");
System.out.println("Enter 'stop' to quit.");
for(int i=0; i<100; i++)
{
s[i] = br.readLine();
if(s[i].equals("stop")) break;
}
System.out.println("Lines are ");
for(int i=0; i<100; i++)
{

if(s[i].equals("stop")) break;
System.out.println(s[i]);
} }}
```

OUTPUT

```
Enter a line of text
Enter 'stop' to quit.
what
how are you
ok
Stop
Lines are
what
how are you
ok
```

25

Read console inputs

```
BufferedReader br = new BufferedReader(new InputStreamReader (System.in));
```

Single Character →	`char ch = br .read ();`
String →	`String str = br .readLine ();`

Integer →	`String str = br .readLine ();`
	`int n = Integer.parseInt (str);`
	`int n = Integer.parseInt (br .readLine ());`

Float →	`Float f = Float.parseFloat (br .readLine ());`
Double →	`double d = Double.parseDouble (br .readLine ());`
Byte →	`byte t = Byte.parseByte (br .readLine ());`
short →	`short s = Short.parseShort (br .readLine ());`
long →	`long l= Long.parseLong (br .readLine ());`
Boolean →	`boolean b = Boolean.parseBoolean (br .readLine ());`

Writing Console Output

- Console output is usually done through **print( ) and println().**
- These methods are defined by the class **PrintStream.**
  - It is the type of object **referenced by System.out.**
  - **System.out is a byte stream,**
- **PrintStream is an output stream derived from OutputStream,**
  - **PrintStream also implements the low-level method write( ).**
  - write( ) can be used to write to the console.

- The simplest form of write( ) defined by PrintStream is
- void write(int byteval)
  - This method writes the byte specified by byteval to the stream
  - byteval is declared as an integer, only the low-order eight bits are written.

```
// Demonstrate System.out.write(). Write letter 'A' to console.
class WriteDemo {
public static void main(String args[]) {
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

OUTPUT
A

PrintWriter class

- For real-world programs, the recommended method of writing to the console using Java is through a PrintWriter stream.
- PrintWriter is one of the character-based classes.
- PrintWriter defines several constructors.

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 println( ) method is called.
  - If flushOnNewline is true, flushing automatically takes place.
  - If false, flushing is not automatic.

- PrintWriter supports the print and println( ) methods
- 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 new line.

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



```
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);
    }
}
```

OUTPUT
This is a string
-7
4.5E-7

System.out

- System.out is a byte stream.
- **System.out** refers to the standard output stream(monitor).
- **System:** It is a final class defined in the java.lang package.
- **out:** This is an instance of PrintStream type, which is a public and static member field of the System class.

PrintWritter

- **PrintWriter** should be used to write a stream of characters
- PrintWriter is a subclass of **Writer** (character stream class)
- It is used in real world programs to make it easier to internationalize the program

Object Streams and Serialization

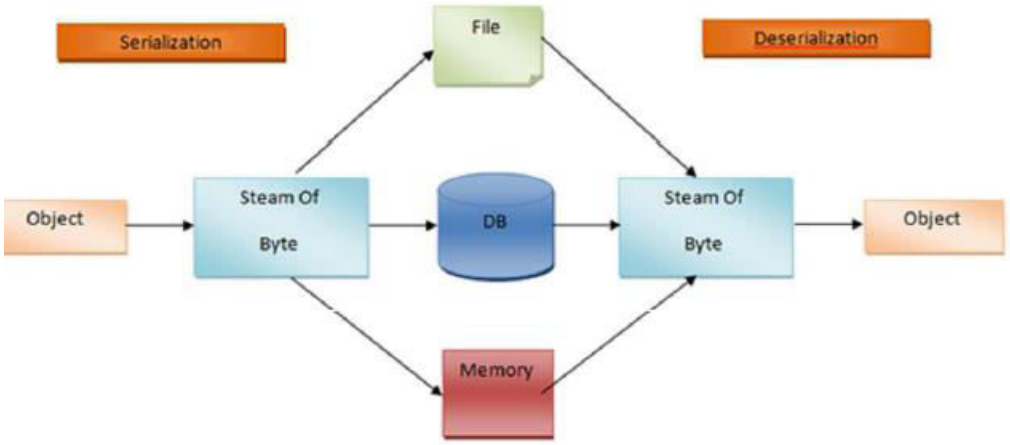
- Object streams support I/O(input-output) of objects
- The object stream classes are

– ObjectInputStream

– ObjectOutputStream

Serialization

- Serialization is the process of writing(converting) the state of an object to a byte stream.
  - This is useful when we want to save(store) the state of the program to a persistent(permanent) storage area,
  - such as a file or when we want to send it over network.
- Later we can restore these objects by using the process of deserialization.
  - Deserialization converts byte streams into object.



- Serialization is also needed to implement Remote Method Invocation (RMI).
  - RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
  - The sending machine serializes the object and transmits it.
  - The receiving machine deserializes it.

- If we attempt to serialize an object at the top of an object graph,
  - all of the other referenced objects are recursively located and serialized.
- Similarly, during the process of deserialization, all of these objects and their references are correctly restored when deserialization is done at the top.

- Interfaces and classes that support serialization are:
  - **Serializable**
  - **Externalizable**

Serializable

- Only an object that implements the Serializable interface can be saved and restored by the serialization facilities.
- The Serializable interface defines no members.
- It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.
- Variables that are declared as transient and static variables are not saved by the serialization facilities.

Externalizable

- Much of the work to save and restore the state of an object occurs automatically.
  - The programmer may need to have control over these processes.
  - it may be desirable to use compression or encryption techniques.
- The **Externalizable interface is designed for these situations.**
- The **Externalizable interface defines two methods:**

void **readExternal**(ObjectInput *inStream)* throws IOException, ClassNotFoundException

void **writeExternal**(ObjectOutput *outStream)* throws IOException

- *inStream* is the byte stream from which the object is to be read
- *outStream* is the byte stream to which the object is to be written

ObjectOutput

- The **ObjectOutput interface extends the DataOutput interface and supports object serialization.**
- It defines the methods such as **writeObject( )**
- **writeObject( ) method is called to serialize an object.**
- All of these methods will throw an **IOException on error** conditions

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

The Methods Defined by **ObjectOutput**

ObjectOutputStream

- The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface.
- It is responsible for writing objects to a stream.
- A constructor of this class is

ObjectOutputStream(OutputStream outStream) throws IOException

  - The argument outStream is the output stream to which serialized objects will be written.
- Methods in this class will throw an IOException on error conditions.
- There is also an inner class to ObjectOuputStream called PutField.
  - It facilitates the writing of persistent fields.

ObjectOutputStream-methods

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

ObjectInput

- The **ObjectInput interface extends the DataInput** interface and defines the method such as **readObject( )** method.
- This is called to **deserialize an object.**
- All of these methods will throw an **IOException on error** conditions.
- The **readObject( ) method can also throw ClassNotFoundException**

ObjectInput-methods

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

ObjectInputStream

- The ObjectInputStream class extends the InputStream class and implements the ObjectInput interface.
- ObjectInputStream is responsible for reading objects from a stream.
- A constructor of this class is

ObjectInputStream(InputStream inStream) throws IOException

  - The argument inStream is the input stream from which serialized objects should be read.
- The methods will throw an IOException on error conditions.
- The readObject() method can also throw ClassNotFoundException.
- There is also an inner class to ObjectInputStream called GetField. It facilitates the reading of persistent fields

Working with Files

- In Java, all files are **byte-oriented.**
- Java provides methods to
– read bytes from a file and
– write bytes to a file.

- Two of the most often-used file stream classes are
- FileInputStream
  – FileInputStream is an input stream to read data
  – from a file in the form of sequence of bytes
- FileOutputStream
  – FileOutputStream class is an output stream for
  – writing data to a file

OPEN a file

- To open a file,
  - create an object of one of these classes
  - specify the name of the file as an argument to the constructor.
- If we want to open a file for reading
  - Create object of FileInputStream class
- If we want to open a file for writing
  - Create object of FileOutputStream class

- Main constructors are

FileInputStream(String *fileName*) throws FileNotFoundException

FileOutputStream(String *fileName*) throws FileNotFoundException

- Here, fileName specifies the name of the file (as String i.e. enclose in double quotes) that we want to open.
  - When we create an input stream, if the file does not exist, then FileNotFoundException is thrown.
  - For output streams, if the file cannot be created, then FileNotFoundException is thrown.
    - When an output file is opened, any file that is already existing with the same name as output file is destroyed

To open a file for reading

- We have to create FileInputStream class object and pass filename as the parameter to the constructor.
- E.g. to open the file Sample.txt for reading

```
FileInputStream fileobject;
fileobject = new FileInputStream(“Sample.txt");
```


To open a file for writing

- We have to create FileOutputStream class object and pass filename as the parameter to the constructor.
- E.g. to open the file Sample.txt for writing

```
FileOutputStream fileobject;
fileobject = new FileOutputStream(“Sample.txt");
```

Closing a file

- After completing file read or write operations, we should close the file by calling close( ).
- It is defined by both FileInputStream and FileOutputStream :

```
void close( ) throws IOException
```


E.g. to close file Sample.txt opened for reading

```
FileInputStream fileobject;
fileobject = new FileInputStream("Sample.txt");
//statements for reading the file
fileobject.close();
```

read a file

- To read data from a file,
  - First, we have to create FileInputStream class object and pass filename as the parameter to the constructor.

  E.g.
  ```
  FileInputStream fileobject;
  fileobject = new FileInputStream("Sample.txt");
  ```
  - Next,we can use a version of read( ) that is defined within FileInputStream.
  ```
  int read( ) throws IOException
  ```
  E.g. `int c=fileobj.read();`
- Each time read() called, it reads a single byte from the file and returns the byte as an integer value.
  - read( ) returns –1 when the end of the file is encountered.
  - read() can throw an IOException.

59

Write a program to read &display the contents in the file Sample.txt

```
import java.io.*;
class Readfile
{
public static void main(String arg[]) throws IOException
{
FileInputStream f;
try
{
f= new FileInputStream("Sample.txt");
int c;
    do   {
         c=f.read();
         if(c!=-1)
         {System.out.print((char)c);}
         } while(c!=-1);
}
```

```
catch(FileNotFoundException e)
{
System.out.println("File not found");
return;
}
f.close();

}
}
```

60

- In this program, to read the file, an object of **FileInputStream class is created.**

```
FileInputStream f;
f= new FileInputStream(“Sample.txt");
```

- Here the argument of the constructor in FileInputStream is the name of the file to be read. Here “Sample.txt”
- The following statement **read one byte from the file** and store in integer variable c

```
c=f.read();
```


- The following statement converts the integer variable c into character using char(c) and print that character on the output screen (console)

```
System.out.print((char)c);
```

- This continues until c is equal to -1(end of file)

- Exceptions(run time error) lik FileNtotFoundException (if the given filename is not in the system path) may occur during file operations.
- So it is better to enclose file operation statements within try block for handling exceptions.
- If the file we try to read a file that does not exist, then that exception is caught by the following catch block and the corresponding action in it is done.

```
try{
  //File Operation statements
  }
  catch(FileNotFoundException e)
  {
  System.out.println("File not found");  //print this message if file is not found
  }
```

15

63

- To write to a file, we can use the write( ) method defined by FileOutputStream.

void write(int byteval) throws IOException

  - This method writes the byte specified by byteval to the file.
  - Although byteval is declared as an integer, only the loworder eight bits are written to the file.
  - If an error occurs during writing, an IOException is thrown

64

Steps to write data to a file

- To write data from a file,
  - First, we have to create FileOutputStream class object and pass filename as the parameter to the constructor.
  - Using write function store the byte value in file.eg:

```
int c=65;
FileOutputStream fileobject;
fileobject = new FileOutputStream("Sample.txt");
fileobject.write(c);
```

- Here lower order will be stored. So this will store ASCII value of 65 that is letter A in file Sample.txt

FILE COPY –copy contents from test.txt to cp.txt

```
import java.io.*;
class Rdfcopy
{ public static void main(String a[]) throws IOException
{
FileInputStream f1=null;
FileOutputStream f2=null;
try
{
f1= new FileInputStream("test.txt");
f2= new FileOutputStream("cp.txt");
int c;
do
{
c=f1.read();
if(c!=-1)
{
f2.write((char)c);
System.out.print((char)c);
}
}while(c!=-1);
}
catch(FileNotFoundException e)
        {
System.out.println("File not found");
return;   }
f1.close();
f2.close();
}
}
```

18

Working of file copy program

- For reading a file, FileInputStream object need to created.
  - Here f1

```
FileInputStream f1=null;
f1= new FileInputStream("test.txt");
```

- For writing to a file, FileOutputStream object need to created
  - Here f2

```
FileOutputStream f2=null;
f2= new FileOutputStream("cp.txt");
```



```
c=f1.read();
if(c!=-1)
{
f2.write((char)c);
System.out.print((char)c);
}
```

- This means that integer **f1.read() reads a single byte from file** pointed by f1(test.txt) and store in integer variable c.
- If c is not -1 (end-of-file) then c is converted int character using(char) casting.

f2.write((char)c);

- This statement writes the character equivalent of c into file pointed by f2(cp.txt)
- This continues until c==-1

FILE READ - file name given as command line argument

Execution: java Readcommandline test.txt

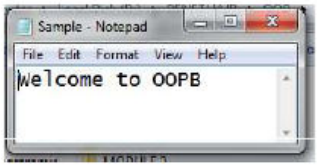
```
import java.io.*;
class Readcommandline
{
public static void main(String arg[]) throws IOException
{
FileInputStream f;
        try
        {
        f= new FileInputStream(arg[0]);
        int c;
                do
                {
                c=f.read();
                if(c!=-1)
                   {System.out.print((char)c);}
                }while(c!=-1);
        }
        catch(FileNotFoundException e)
        {
        System.out.println("File not found");
        return;
}
f.close();}}
```



```
import java.io.*;
 class Writesentencefile
{
public static void main(String arg[]) throws IOException
{
FileOutputStream f;
String s;
try
{
f= new FileOutputStream("Sample.txt");
     s="Welcome to OOP";
byte b[]=s.getBytes();          //converting string into byte array
f.write(b);

 f.write(66);       // write lower bytes. Here we will get ASCII vlue of 66 i.e. letter  B

}
catch(FileNotFoundException e)
{
System.out.println("File not found");
return;
}
f.close();
} }
```



FileReader

- The FileReader class creates a Reader that we can use to read the contents of a file.
- Its two most commonly used constructors are shown here:

FileReader(String filePath)

FileReader(File fileObj)

  - They can throw a FileNotFoundException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file.
- The following example shows how to. It reads its own source file, which must be in the current directory.

Read lines from a file and print these to the standard output stream using FileReader

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

FileWriter

- FileWriter creates a Writer that you can use to write to a file.
- Its most commonly used constructors are:

FileWriter(String filePath)

FileWriter(String filePath, boolean append)

FileWriter(File fileObj)

FileWriter(File fileObj, boolean append)

  - They can throw an IOException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file.
  - If append is true, then output is appended to the end of the file.

- FileWriter will create the file before opening it for output when you create the object.
  - In the case where we attempt to open a read-only file, an IOException will be thrown.
- getChars( ) method is used to extract the character array equivalent.

public void getChars(int srhStartIndex, int srhEndIndex, char[] destArray, int destStartIndex)

Parameters:

srhStartIndex : Index of the first character in the string to copy.

srhEndIndex : Index after the last character in the string to copy.
destArray : Destination array where chars wil get copied.

destStartIndex : Index in the array starting from where the chars will be pushed into the array. **Return:** It does not return any value.

Write a string to file using FileWriter

```
import java.io.*;
class FileWriterSimple
{
public static void main(String args[]) throws IOException
{
String source = "Welcome to OOP class\n" + " Study well";
char buffer[] = new char[source.length()]; // allocate space equal to length of string
source.getChars(0, source.length(), buffer, 0);
//copy the characters from position 0 to whole length(end) to buffer at position 0.
FileWriter f1 = new FileWriter("file1.txt");
f1.write(buffer);
f1.close();
}
}
```

27

75

Append a string to file using FileWriter

```
import java.io.*;
class FileWriterSimple
{
public static void main(String args[]) throws IOException
{
String source = "Welcome to OOP class\n" + " Study well";
char buffer[] = new char[source.length()]; // allocate space equal to length of string
source.getChars(0, source.length(), buffer, 0);
    //copy the characters from position 0 to whole length(end) from source
    //to buffer at /position 0.
FileWriter f1 = new FileWriter("file1.txt",true); //append the contents
f1.write(buffer);
f1.close();
}
}
```

76

Syllabus

- Input/Output –
  - I/O Basics, Reading Console Input, Writing Console Output, PrintWriter Class, Object Streams and Serialization, Reading and Writing Files.
- Java Library
  - String Handling – String Constructors, String Length, Special String Operations Character Extraction, String Comparison, Searching Strings, Modifying Strings, Using valueOf(), Comparison of StringBuffer and String.
- Collections framework
  - Collections overview, Collections Class – ArrayList. Accessing Collections via an Iterator.

String Handling

- String is a class in Java.
- Java implements strings as objects of type String.
  - The String type is used to declare string variables
- Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- A quoted string constant(E.g. “hello”) can be assigned to a String variable.
- A variable of type String can be assigned to another variable of type String.

String Constructors

- The String class supports several constructors.
- To create an empty String, call the default constructor.

String()

  – For example,

```
String s = new String();
```

  – This will create an instance of String with no characters in it.

- To create a String initialized by an array of characters, use the constructor

```
String(char chars[ ])
```

- Example:

```
char letters[] = { 'a', 'b', 'c' };
String s = new String(letters);
```

- This constructor initializes s with the string “abc”.

- To initialize a string with a subrange of a character array(substring) the following constructor is used:

`String(char chars[ ], int startIndex, int numChars)`

  - Here, startIndex specifies the startindex at which the subrange begins, and
  - numChars specifies the number of characters to use.
- E.g.

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
                    0   1   2   3   4   5
String s = new String(chars, 2, 3);
```

- This initializes s with the characters starting from index 2 and number of letters =3. i.e. s will contain cde.

- We can construct a String object that contains the same character sequence as another String object using this constructor:

`String(String strObj)`

```
// Construct one String from another.
class MakeString {
public static void main(String args[])
{
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);
    System.out.println(s1);
    System.out.println(s2);
}
}
```

OUTPUT
Java
Java

- String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

String(byte *asciiChars[ ]*)

String(byte *asciiChars[ ], int startIndex, int numChars*)

- Here asciiChars specifies the array of bytes.
  - In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.


```
class SubStringCons {
public static void main(String args[])
{
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

OUTPUT
ABCDEF
CDE

- We construct a String from a **StringBuffer** by using the constructor :

```
String(StringBuffer strBufObj)
```

- J2SE 5 added two constructors to String.

➢ The first supports the ***extended Unicode character set*** :

```
String(int codePoints[ ], int startIndex, int numChars)
```

– Here, *codePoints is an array that contains Unicode code points*

➢ The second new constructor supports the new **StringBuilder class:-**

```
String(StringBuilder strBuildObj)
```

– This constructs a **String** from the **StringBuilder** passed in *strBuildObj*.

String Length

- The length of a string is the number of characters in the string
- E.g. length of the string “hello” is 5
- The method length() is used to find the length of the string.

int length( )

```
class Stringlen
{
   public static void main(String args[])
   {
   String s="Hello";
   System.out.println("Length="+s.length());
   }
}
```

OUTPUT
Length=5

Special String Operations

- These operations include
  - the automatic creation of new String instances(object) from string literals
  - concatenation of multiple String objects by use of the + operator, and
  - the conversion of other data types to a string representation.

String Literals

- Java automatically constructs a String object for each string literal in our program,.
  - So we can use a string literal to initialize a String object

```
String s2 = "abc";
```

is same as

```
char chars[] = { 'a', 'b', 'c' };
String s2 = new String(chars);
```

- We can use a string literal at any place where we use a String object.
- String literals can call the length( ) method on the string

E.g.

```
System.out.println("abc".length());
```

String Concatenation

- String concatenation is used to join two strings
- Method 1:The + operator can be used between strings to combine them. This is called concatenation.
- Operator + can be chained to concatenate many strings

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

- This fragment displays the string He is 9 years old.
- Instead of letting long strings wrap around within our source code, we can break them into smaller pieces, using the + to concatenate them

String Concatenation with Other Data Types

- We can concatenate strings with other types of data.
- If one of the operand of the + is an instance of String then compiler will convert other operand to its string equivalent.

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

  - This fragment displays **four: 22**
  - Operator precedence causes the concatenation of "four" with 2. So 2is converted into string and "four: " concatenates with string equivalent of 2.
  - Then this result is then concatenated with the string equivalent of 2.
- Parentheses can be used for grouping integers and + to perform addition.

```
String s = "four: " + (2 + 2);
```

  - Here parentheses is first computed. So (2+2) is 4 then string "four: " is concatenated with that. So s contains the string **"four: 4"**

String Conversion and toString( )

- When Java converts data into its string representation during concatenation, it calls one of the overloaded versions of the string conversion method valueOf( ) by class String.
- valueOf( ) is overloaded for all the simple types and for type Object
  - For the simple types, valueOf( ) returns a string that contains the human-readable equivalent of the value with which it is called.
  - For objects, valueOf( ) calls the toString( ) method on the Object.

String Conversion and toString( )

The valueOf() returns the string representation of the corresponding argument. Different overloaded form of valueOf() in String class.

- **valueOf(boolean b)** – Returns the string representation of boolean argument.
- **valueOf(char c)** – char argument.
- **valueOf(char[] data)** char array argument.
- **valueOf(char[] data, int offset, int count)** – specific subarray of the char array argument.
- **valueOf(double d)** – double argument.
- **valueOf(float f)** – float argument.
- **valueOf(int i)** – int argument.
- **valueOf(long l)** – long argument.
- **valueOf(Object obj)** – Object argument. (calls toString() method of the class Object(parent class of all classes n Java)

- The **toString( )** method has this general form:

```
String toString( )
```

- When we try to print an object of a class, it will call method valueOf(object) which calls toString( ) function :-
  - if toString( ) is present (overridden) in the class, then it is called.
  - If there is no toString( ) function in the class, when we try to print an object of that class, it prints **clasname@the memory location of the object**(the hexidecimal address of where that **object** is stored in memory.)

Without using toString()

```
class Box {
double width;
double height;
double depth;
Box(double w, double d, double h,) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +depth + " by " + height + ".";
}
}
```

```
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 14,12);
String s = "Box b: "+ b;

System.out.println(b);
System.out.println(s);
}
}
```

OUTPUT
Box@106d69c
Box b: Box@1db9742

Here when we print the object **b** ,since there is no toString( ) function in the class it will call toString() in class Object and prints **clasname@the memory location of the object**
(Here it prints Box@106d69c)

Using toString()

```
class Box {
double width;
double height;
double depth;
Box(double w, double d, double h,)
{
width = w;
depth = d;
height = h;
}
public String toString()
{
return "Dimensions are " + width + " by " + depth + " by " + height + ".";
}
}
```

```
class StringDemo {
public static void main(String args[])
 {
Box b = new Box(10, 14,12);
String s = "Box b: "+ b;

System.out.println(b);
System.out.println(s);
}
}
```

OUTPUT
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0

Class **Box's toString( )** method is **automatically invoked** when a **Box object** is used in a concatenation expression or used in **println( ).**

Character Extraction

- The String class provides the following methods through which characters can be extracted from a String object.
- charAt( )
- getChars( )
- getBytes( )
- toCharArray( )

charAt( )

- Used to **extract a single character from a String**
- we can refer directly to an individual character via the charAt( ) method.
- General form:

char **charAt(int *where)***

- Here *where is the index of the character that you want to obtain.* e.g.

```
char ch;
ch = "abc".charAt(1);
```

```
abc
index 012
```

- This assigns the value "**b" to variable ch.**

getChars( )

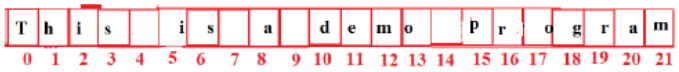
- Used to extract more than one character at a time.
- General form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

- Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index up to which character need to be extracted.
  - (the extracted substring contains the characters from sourceStart through sourceEnd–1.)
- This extracted substring is stored at target array at location targetStart.

Example program- getChars( )

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo program";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```



This program will extract characters in string **s** from index 10 to 14-1(13) and store in character array **buf** and prints it.

OUTPUT
demo

getBytes( )

- Used to **extract the characters in an array of bytes.**
- it uses the default character-to-byte conversions.
- General form

byte[ ] **getBytes( )**

- Most Internet protocols and text file formats use 8-bit
- ASCII for all text interchange.

toCharArray( )

- Used to convert all the characters in a String object into a character array.
- It returns an array of characters for the entire string.
- General form:

char[ ] toCharArray( )


```
public class CharArrayEg{
  public static void main(String args[]){
     String str = new String("Welcome to OOP");
     char[] a= str.toCharArray();
     System.out.print("Content of a is:");
     for(char c: a){
        System.out.print(c);
     }
  }
}
```

OUTPUT
Content of a is: Welcome to OOP

String Comparison

- The String class includes several methods that compare strings or substrings within strings.
- **equals( )**
- **equalsIgnoreCase( )**
- **regionMatches( )**
- **startsWith( )**
- **endsWith( )**
- **equals( ) Versus ==**
- **compareTo( )**

equals( )

- To compare two strings for equality, use equals( )
- General form:

boolean equals(Object str)

- Here, String object str is compared with the invoking String object.
- It returns true if the strings contain the same characters in the same order, and false otherwise.
- The comparison is case-sensitive.

equalsIgnoreCase( )

- This perform a comparison that **ignores case differences(not** case sensitive)
- When it compares two strings, it considers A-Z to be the same as a-z.
- General form:

boolean **equalsIgnoreCase(String *str)***


```
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " is " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " is " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " is " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " is " +s1.equalsIgnoreCase(s4));
}
}
```

```
Hello equals Hello is true
Hello equals Good-bye is false
Hello equals HELLO is false
Hello equalsIgnoreCase HELLO is true
```

regionMatches( )

- The **regionMatches( ) method compares a specific region** inside a string with another specific region in another string.
- General forms:

```
boolean regionMatches(int startIndex, String str2,
                      int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase, int startIndex,
String str2, int str2StartIndex, int numChars)
```


- startIndex specifies the index at which the region begins within the invoking String. The String to be compared is specified by str2.
- The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars.
- In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )

- The startsWith( ) method determines whether a given String begins with a specified string.
- Conversely, endsWith( ) determines whether the String in question ends with a specified string.

General forms:

boolean **startsWith**(String *str)*
boolean **endsWith**(String *str)*

System.out.println("Football".endsWith("ball"));

This prints **true**. (because *bar* comes at the end of string *Football*

System.out.println(" Football ".startsWith("Foo"));

This prints **true**. (because *Foo* comes at the beginning of string *Football*

- A second form of startsWith( ), specify a starting point:

boolean **startsWith**(String *str, int startIndex)*

- Here, *startIndex specifies the index into the invoking string at which point the search will* begin.
- For example,

System.println("Football".startsWith("ball", 4));

  - This prints **true**.

equals( ) Versus ==

- equals( ) method and the == operator perform two different operations.
- the equals( ) method compares the characters inside a String object.
- The == operator compares two object references to see whether they refer to the same instance.


```
class EqualsNotEqualTo
{
public static void main(String args[])
{
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " is " + s1.equals(s2));
System.out.println(s1 + " = = " + s2 + " is " + (s1 ==s2));
}
}
```

OUTPUT
Hello equals Hello is true
Hello = = Hello is false

compareTo( )

- A string is less than another if it comes before the other in dictionary order.
  - E.g. “ant”<“bat” (ant comes before bat in dictionary)
- A string is greater than another if it comes after the other in dictionary order.
  - E.g. “bat”>“ant” (bat comes after ant in dictionary)
- compareTo( ) method in String is used for comparing two strings. General form:

int compareTo(String *str*)

Value	Meaning
Less than zero	The invoking string is less than ***str***.
Greater than zero	The invoking string is greater than ***str***.
Zero.	The two strings are equal


```
class CompareToEg
{
public static void main(String args[]) {
    String s1="ant";
    String s2="bat";
    if(s1.compareTo(s2) < 0)
    {System.out.println(s1 + " comes before "+s2);
    }
    else if(s1.compareTo(s2) > 0)
    {System.out.println(s1 + " comes after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);
}
}
```

OUTPUT
ant comes before bat

Bubblesort to sort strings

```
class SortString {
static String arr[] = {"This", "is", "best", "time", "for", "all"};
public static void main(String args[]) {
for(int i = 0; i < arr.length; i++)
{
    for(int j = i + 1; j < arr.length; j++)
    {
        if(arr[j].compareTo(arr[i]) < 0)
        {
        String temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        }
    }
System.out.println(arr[i]);
}
}
}
```

OUTPUT
This
all
best
for
is
time

compareToIgnoreCase( )

- compareToIgnoreCase( ) method is not case sensitive.

int compareToIgnoreCase(String str)

- This method returns the same results as compareTo( ), except that case differences are ignored.

Using compareToIgnoreCase

```
class CompareToIgnoreEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareToIgnoreCase(s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToIgnoreCase(s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);

}
}
```

OUTPUT

ant is before Hat

Using compareTo

```
class CompareToEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareTo (s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToI (s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is  same as "+s2);
}
}
```

OUTPUT

ant is after Hat

Searching Strings

- The String class provides two methods to search a string for a specified character or substring:
- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.
  - These two methods are overloaded in several different ways.
  - In all cases, the methods return the index at which the character or substring was found. If the character or substring is not found then these method returns -1.

- To search for the **first occurrence of a *character***, use

  int **indexOf**(int *ch)*

- To search for the **last occurrence of a *character***, use

  int **lastIndexOf**(int *ch)*

  – Here, *ch is the character being searched.*

- To search for the **first or last occurrence of a substring,** use

  int **indexOf**(String *str)*

  int **lastIndexOf**(String *str)*

  – Here, *str specifies the substring*.

- We can specify a **starting point for the search** using :

  int indexOf(char *ch, int startIndex)*

  int lastIndexOf(char *ch, int startIndex)*

  int indexOf(String *str, int startIndex)*

  int lastIndexOf(String *str, int startIndex)*

- Here *startIndex* specifies the index at which point the search begins
- For **indexOf( ), the** search runs from *startIndex to the end of the string.*
- *For lastIndexOf( ), the search runs from startIndex to zero.*

```
class indexOfDemo {
public static void main(String args[]) {
String s = “This is a pen. This is a pencil.";
System.out.println(s);
System.out.println("indexOf(i) = " +s.indexOf('i'));
System.out.println("lastIndexOf(i) = " +s.lastIndexOf('i'));
System.out.println("indexOf(This) = " +s.indexOf("This"));
System.out.println("lastIndexOf(This) = " +s.lastIndexOf("This"));
System.out.println("indexOf(i, 10) = " +s.indexOf('i', 10));
System.out.println("lastIndexOf(i, 23) = " + s.lastIndexOf('i', 23));
System.out.println("indexOf(This, 10) = " + s.indexOf("This", 10));
System.out.println("lastIndexOf(This, 13) = " + s.lastIndexOf("This", 13));
}
}
```


This is a pen. This is a pencil.
indexOf(i) = 2
lastIndexOf(i) = 29
indexOf(This) = 0
lastIndexOf(This) = 15
indexOf(i, 10) = 17
lastIndexOf(i, 23) = 20
indexOf(This, 10) = 15
lastIndexOf(This, 13) = 0

Modifying a String

- **String objects are immutable(cannot change a string.)**
- To modify a String, we must either
  - copy it into a StringBuffer or StringBuilder, or
  - use one of the following String methods:

substring( )
concat( )
replace()
trim( )

substring( )

- We can **extract a substring** using **substring( ).**
- It has two forms.
  - The first is

String substring(int *startIndex)*

- Here, startIndex **specifies the index at which the substring will begin**. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

  - The second form of **substring( )** allows to specify both the beginning and ending index of the substring:

String substring(int *startIndex, int endIndex)*

- Here, *startIndex* specifies the beginning index, and endIndex specifies the stopping point.
- The string returned contains all the characters from the beginning index, up to, but **not including, the ending index.**

Modifying a String(contd.)

```
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do {
System.out.println(org);
i = org.indexOf(search);
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result;
}
} while(i != -1);
}
```

This is a test. This is, too.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

OUTPUT

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```


concat( )

- We can use **concat()** method to concatenate two strings.

  String concat(String *str)*

- This method ***creates a new object*** that contains the invoking string with the value of *str appended to the end of it.*
- ***concat( ) performs the same function as* +.**
- ***For example,***

```
String s1 = "one";      // string s1 contains"one"
String s2 = s1.concat("two");
```

  - Here s1 is the invoking string that call the function concat() .
  - s1 contains "one" and is concatenated with argument string value "two" and form the string "onetwo". This result is stored in the String object **s2**

replace( )

replace( ) -The replace( ) method has two forms.

1. The first form **replaces all occurrences of one character in** the invoking string with another character.

String replace(char *original, char replacement)*

   - Here, *original* specifies the character that will be replaced by the character specified by *replacement*.
   - The resulting string is returned.

String s = "Hello".replace('l', 'w');

   - Here letter l is replaced by w. So "Hewwo" is put into String object s.

2. The second form of **replace( ) replaces one character sequence with another.**

String replace(CharSequence *original, CharSequence replacement)*

This form was added by J2SE 5.

trim( )

- The **trim( )** method returns a copy of the invoking string after **removing any leading and trailing whitespace**
- General form:

String trim( )

String s = " Hello World ".trim();

   - This puts the string "Hello World" into **s.**

- The **trim( )** method is quite useful when we process user commands.

- **E.g.** Write a program that prompts the user to **enter the name** of a state(Assam,Goa etc) and then **displays that state's capital**. Use **trim( )** to *remove any leading or trailing whitespace* that may have inadvertently been entered by the user.


```
import java.io.*;
class UseTrim {
public static void main(String args[]) throws
                                    IOException
{
BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
do {
System.out.println("Enter the State: ");
str = br.readLine();
str = str.trim();
if(str.equals("Assam"))
     System.out.println("Capital is  Dispur");
else if(str.equals("Goa"))
     System.out.println("Capital is Panaji");
else if(str.equals("Bihar"))
     System.out.println("Capital is Patna.");
else
System.out.println("Capital is not entered");
     } while(!str.equals("stop"));
  }
}
```

Data Conversion Using valueOf( )

- The valueOf( ) method **converts** data from its internal format **into a human-readable form.**
- It is a **static** method.
- **valueOf( )** is overloaded for all the simple types and for type Object
  - For the simple types, **valueOf( ) returns a string that contains the human-readable equivalent of the value** with which it is called.
  - For objects, **valueOf( ) calls the toString( )** method on the Object.

The valueOf() **returns the string representation of the corresponding argument.** Different overloaded form of valueOf() in String class.

- **valueOf(boolean b)** – Returns the string representation of boolean argument.
- **valueOf(char c)** – char argument.
- **valueOf(char[] data)** char array argument.
- **valueOf(char[] data, int offset, int count)** – specific subarray of the char array argument.
- **valueOf(double d)** – double argument.
- **valueOf(float f)** – float argument.
- **valueOf(int i)** – int argument.
- **valueOf(long l)** – long argument.
- **valueOf(Object obj)** – Object argument. (calls toString() method of the class Object(parent class of all classes n Java)

- **valueOf( )** is called when **a string representation of some other type of data is needed**
  - example, during concatenation operations
- Any object that we pass to **valueOf( )** will return the result of a call to the object's **toString( )** method**.**
- For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type.
- For arrays of **char,** however, a String object is created that contains the characters in the char array

Changing the Case of Characters Within a String

- String toLowerCase( )
- String toUpperCase( )

```
class ChangeCase {
public static void main(String args[])
{
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

OUTPUT
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.

Comparison of String Buffer and String.

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- String represents fixed-length, immutable character sequences.
- StringBuffer represents growable and writeable character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

135

String	StringBuffer
• **String** is immutable.	• **StringBuffer** is **mutable.**
• **String** represents **fixed-length, immutable character** sequences.	• **StringBuffer** represents **growable** and **writeable** character sequences
• Concatenation using String is slow.	• Concatenation using StringBuffer is fast.
• String class can override equals() method.	• StringBuffer class doesnot override equals() method.

136

String str = "Hello World";

str = "Hi World!";

- Here an object is created using string literal "Hello World".
- In second statement when we assigned the new string literal "Hi World!" to str, the **object itself didn't change** instead a **new object got created in memory** using string literal "Hi World!" and the reference to it is assigned to str.

StringBuffer Constructors

- **StringBuffer defines these four constructors:**

StringBuffer( )

StringBuffer(int *size)*

StringBuffer(String *str)*

StringBuffer(CharSequence *chars)*

- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.

length( ) and capacity( )

- The current length of a **StringBuffer** can be found via the **length( )** method.
- The total allocated capacity can be found through the **capacity( )** method.

```
int length( )
int capacity( )
```

```
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

OUTPUT
buffer = Hello
length = 5
capacity = 21

Here capacity is 21 because room for 16 additional characters is automatically added to value Hello

ensureCapacity( )

- **ensureCapacity( )** is used to set the size of the buffer.
- This is useful if we know in advance that we will be appending a large number of small strings to a **StringBuffer.**

```
void ensureCapacity(int capacity)
```

- Here, *capacity specifies the size of the buffer.*

setLength( )

- Used to set the length of the buffer within a **StringBuffer object.**

```
void setLength(int len)
```

Here *len* specifies the length of the buffer. This value must be nonnegative.

- When we increase the size of the buffer, **null characters** are added to the end of the existing buffer.
- If we call **setLength( )** with a value **less than the current value returned by length( ),** then the characters stored **beyond the new length will be lost.**

- **charAt( )** and **setCharAt( )**
- The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method.
- We can set the value of a character within a **StringBuffer** using **setCharAt( ).**

```
char charAt(int where)
```

```
void setCharAt(int where, char ch)
```

```
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1));
}
}
```

OUTPUT
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i

- **getChars( )**
- Used to copy a substring of a **StringBuffer.**

void getChars(int *sourceStart, int sourceEnd, char target[ ],*
int *targetStart)*

append( )

- The **append( )** method **concatenates** the string representation of any other type of data to the **end of the invoking StringBuffer object.**

StringBuffer append(String *str)*
StringBuffer append(int *num)*
StringBuffer append(Object *obj)*

- **String.valueOf( )** is called for each parameter to obtain its string representation. The
- The result is appended to the current **StringBuffer object.**
- The buffer itself is returned by each version of **append( ).**
  - **append() calls can be chained**


```
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s);
}
}
```

Output

a = 42!

insert( )

- The **insert( )** method inserts one string into another.
- It calls **String.valueOf( ).**
- This string is then inserted into the invoking **StringBuffer object.**

StringBuffer insert(int *index, String str)*
StringBuffer insert(int *index, char ch)*
StringBuffer insert(int *index, Object obj)*


```
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);
}
}
```

OUTPUT

I like Java!

reverse( )

- We can reverse the characters within a **StringBuffer** object using **reverse( ):**

StringBuffer **reverse**( )

```
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

OUTPUT

```
abcdef
fedcba
```


delete( ) and deleteCharAt( )

- We can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**

StringBuffer delete(int *startIndex, int endIndex)*

StringBuffer deleteCharAt(int *loc)*

- **delete()** deletes from *startIndex to endIndex–1.*
- The **deleteCharAt( )** method deletes the character at the index specified by *loc*

```
class deleteDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.delete(4, 7);
System.out.println("After delete: " + sb);
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);
}
}
```

The following output is produced:

After delete: This a test.

After deleteCharAt: his a test.

replace( )

- We can replace **one set of characters** with another set inside a **StringBuffer** object by calling **replace( ).**

StringBuffer **replace**(int *startIndex, int endIndex, String str)*

  - The substring at *startIndex through endIndex–1 is replaced.*

```
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
}
}
```

OUTPUT

After replace: This was a test

substring( )

- We can obtain a portion of a **StringBuffer** by calling **substring( ).**

String substring(int *startIndex)*

String substring(int *startIndex, int endIndex)*

- The first form returns the substring that starts at *startIndex and runs to the end of the* invoking **StringBuffer object.**
- The second form returns the substring that starts at *startIndex* and runs through *endIndex–1*.

Syllabus

- Input/Output –
  - I/O Basics, Reading Console Input, Writing Console Output, PrintWriter Class, Object Streams and Serialization, Reading and Writing Files.
- Java Library
  - String Handling – String Constructors, String Length, Special String Operations Character Extraction, String Comparison, Searching Strings, Modifying Strings, Using valueOf(), Comparison of StringBuffer and String.
- Collections framework
  - Collections overview, Collections Class – ArrayList. Accessing Collections via an Iterator.

Collections Framework

- The **java.util** package contains one of Java's most powerful subsystems: The ***Collections Framework.***
- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology(best possible technlogy) for **managing groups of objects.**

  – The **Collection** in **Java** is a **framework** that provides an architecture to **store and manipulate the group of objects.**

  – **Java Collection framework** provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)

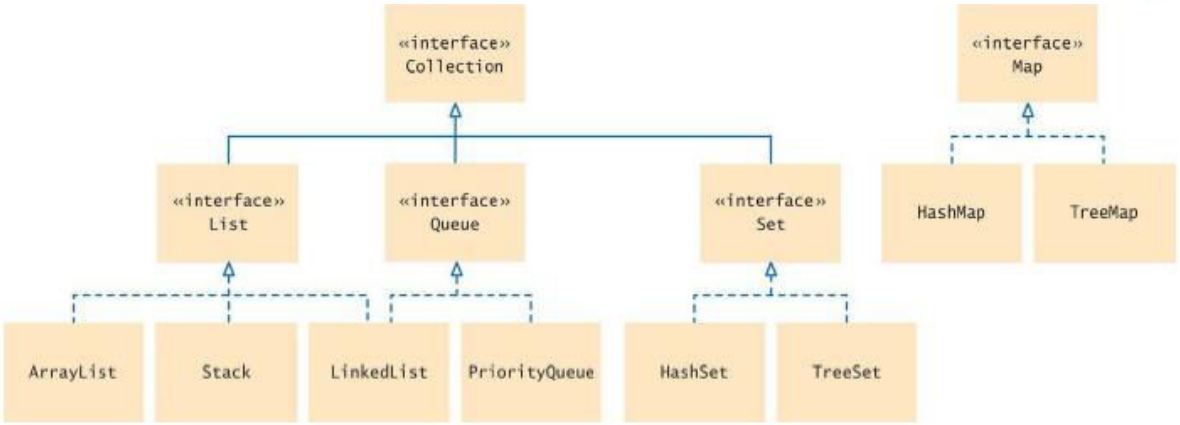


Figure 1 Interfaces and Classes in the Java Collections Framework

Collections Overview

- The Java Collections Framework standardizes the way in which groups of objects are handled by our programs.
- The entire Collections Framework is built upon a set of standard interfaces.
- Mechanisms were added that allow the integration of standard arrays into the Collections Framework.

- The Collections Framework was designed to meet several goals.
  - First, the framework had to be high-performance.
- The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
  - Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Third, extending and/or adapting a collection had to be easy.

- Algorithms are an important part of the collection mechanism.
  - Algorithms operate on collections and are defined as static methods within the Collections class.
- The algorithms provide a standard means of manipulating collections.
- Java Collections Framework provides algorithm implementations that are commonly used such as sorting, searching etc.
  - void sort(List list)
  - int binarySearch(List list, Object value)

- Another item closely associated with the Collections Framework is the Iterator interface.
- An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.
  - An iterator provides a means of enumerating the contents of a collection.
- Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator

- The framework defines several map interfaces and classes.
  - Maps store key/value pairs.
- A map cannot contain duplicate keys.
- Although maps are part of the Collections Framework, they are not “collections” in the strict use of the term

Recent Changes to Collections

- Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use.
  - The changes were caused by the addition of
    - generics
    - autoboxing/unboxing, and
    - for-each style for loop.

- **Generics** add the one feature : **type safety**.
  - With generics, it is possible to **explicitly state the type of data** being stored, and run-time type mismatch errors can be avoided.
- **Autoboxing/unboxing** facilitates the **storing of primitive types** in collections.
  - IN THE PAST, if we wanted ***to store a primitive value, such as an int, in a collection*, we had to manually box it into its type wrapper**.
  - When *the value was* ***retrieved***, it needed to be **manually unboxed** (by using an explicit cast) into its proper primitive type.
  - Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types.

The Collection Framework

- The Collections Framework defines several interfaces.

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

The Collection Classes

- The collection classes **implement collection interfaces.**
- Some of the collection classes provide **full implementations that can be used as-is.**
- Some of the collection classes are **abstract, providing skeletal implementations that are used as starting points** for creating concrete collections.

The standard collection classes are

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. (Added by Java SE 6.)
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

ArrayList Class

- The **ArrayList class extends AbstractList** and implements the **List** interface.
- ArrayList is a generic class that has declaration:

```
class ArrayList<E>
```

  - Here, E specifies the type of objects that the list will hold.
- **ArrayList** supports **dynamic arrays** that can grow as needed.
  - This is needed because in some cases we may not know how large an array we need precisely until run time.

- An **ArrayList** is a **variable-length array** of object references.
  - So **ArrayList** can dynamically increase or decrease in size.
- Array lists are created with an initial size.
  - When this size is exceeded, the collection is **automatically enlarged.**
  - When objects are removed, the array can be **shrunk**.

- **ArrayList** has following **constructors**:

ArrayList( )

- This constructor builds an **empty** array list.

ArrayList(Collection<? extends E> *c)*

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

ArrayList(int *capacity)*

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


```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size=" +al.size());
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size now=" +al.size());
System.out.println("Contents : " + al);
al.remove("F");
al.remove(2);
System.out.println("Size=" +al.size());
System.out.println("Contents=" +al);
}
}
```

```
Initial size=0
Size now=7
Contents : [C, A2, A, E, B, D, F]
Size=5
Contents=[C, A2, E, B, D]
```

- The contents of a collection are displayed using the default conversion provided by **toString( )**, which was inherited from **AbstractCollection**.
- We can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( ).**

```
void ensureCapacity(int cap)
```

- If we want to **reduce** the size of the array that of**ArrayL ist** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**:

```
void trimToSize( )
```


Accessing Collections via an Iterator

- To **cycle through the elements in a collection** (e.g. display each element, sum of elements etc.), we can use *iterator, which is an object* that **implements** either
  - **Iterator** or
  - **ListIterator**

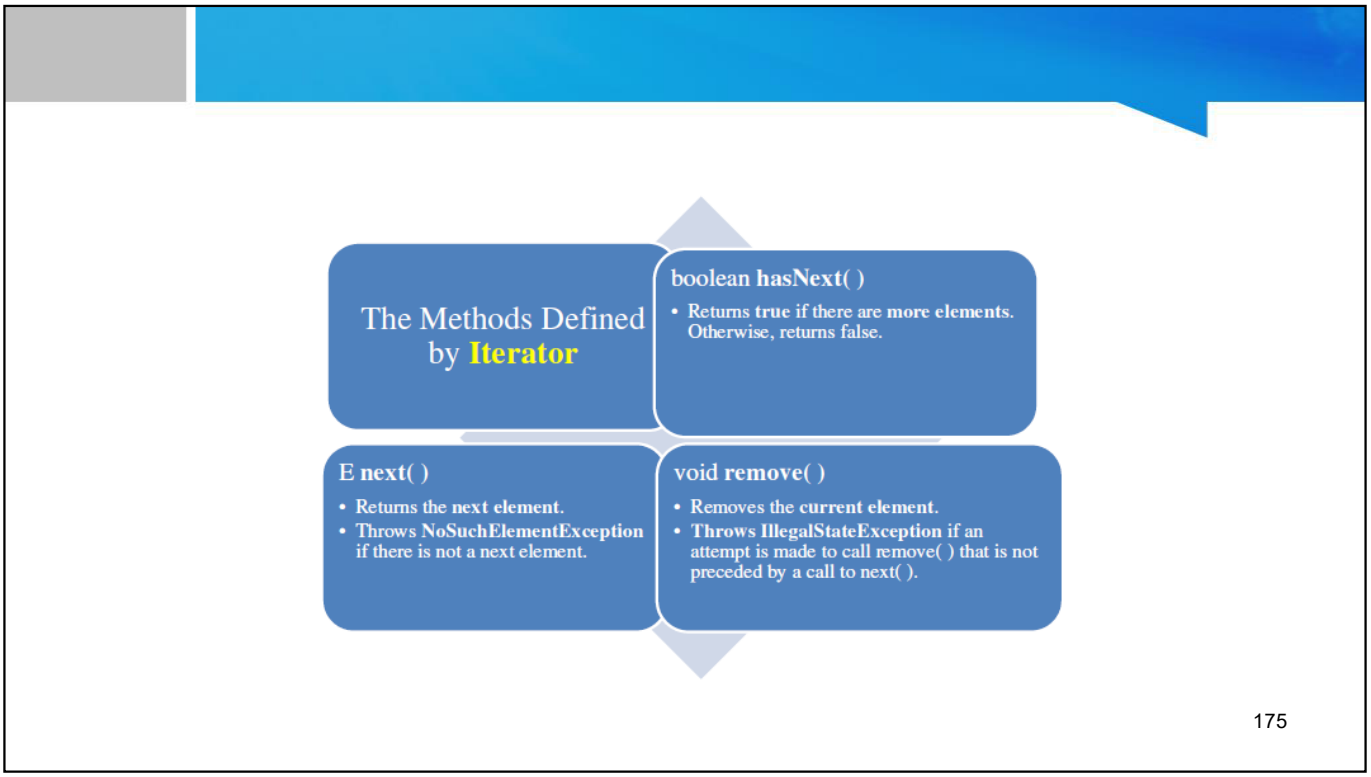
- **Iterator** and **ListIterator** are **generic interfaces** which are declared as :

```
interface Iterator<E>
```

```
interface ListIterator<E>
```

  - Here, **E** specifies the type of objects being iterated

- **Iterator** enables you to
  - cycle through a collection
  - obtaining or removing elements.
- **ListIterator extends Iterator** to allow
  - **bidirectional traversal** of a list,
  - the **modification** of elements



Method Defined by ListIterator	
void add(E obj)	• **Inserts obj into the list in front** of the element that will be returned by the next call to next( ).
boolean hasNext( )	• Returns **true** if there is a **next element**. Otherwise, returns false.
boolean hasPrevious( )	• Returns **true** if there is **a previous element**. Otherwise, returns false.
E next( )	• **Returns the next element.** NoSuchElementException is thrown if there is not a next element.
int nextIndex( )	• Returns the **index of the next element**. If there is not a next element, returns the size of the list.
E previous( )	• Returns the **previous element**. **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	• Returns the **index of the previous element**. If there is not a previous element, returns −1.
void remove( )	• Removes the **current element** from the list. An **IllegalStateException** is thrown if remove( ) is called before next( ) or previous( ) is invoked.
void set(E obj)	• **Assigns obj to the current element.** This is the element last returned by a call to either next( ) or previous( ).

Exceptions in methods

- Exceptions in the Methods Defined by **Iterator**
  - **NoSuchElementException**
  - **IllegalStateException**
- Exceptions in the Methods Defined by **ListIterator**
  - **NoSuchElementException**
  - **IllegalStateException**
  - **UnsupportedOperationException**

Using an Iterator

- Each of the **collection classes** provides an **iterator( ) method that returns an iterator to the start** of the collection.
  - By using this iterator object, we can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection,
  - 1. **Obtain an iterator** to the start of the collection by calling the collection's **iterator( )** method.
  - 2. Set up a loop that makes a call to **hasNext( ).**
    - Have the loop iterate as long as hasNext( ) returns true.
  - 3. Within the loop**, obtain each element** by calling **next( ).**

```
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext())
{
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
ListIterator<String> litr = al.listIterator();
while(litr.hasNext())
 { String element = litr.next();
litr.set(element + "+");  }
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " "); }
System.out.println();
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
} System.out.println();  }  }
```

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```


END OF MODULE 4