Java Iterator Interface

The Iterator interface of the Java collections framework allows us to access elements of a collection. It has a subinterface ListIterator.

All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.

Methods of Iterator

The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections.

hasNext() - returns true if there exists an element in the collection

next() - returns the next element of the collection

remove() - removes the last element returned by the next()

forEachRemaining() - performs the specified action for each remaining element of the collection

Example: Implementation of Iterator

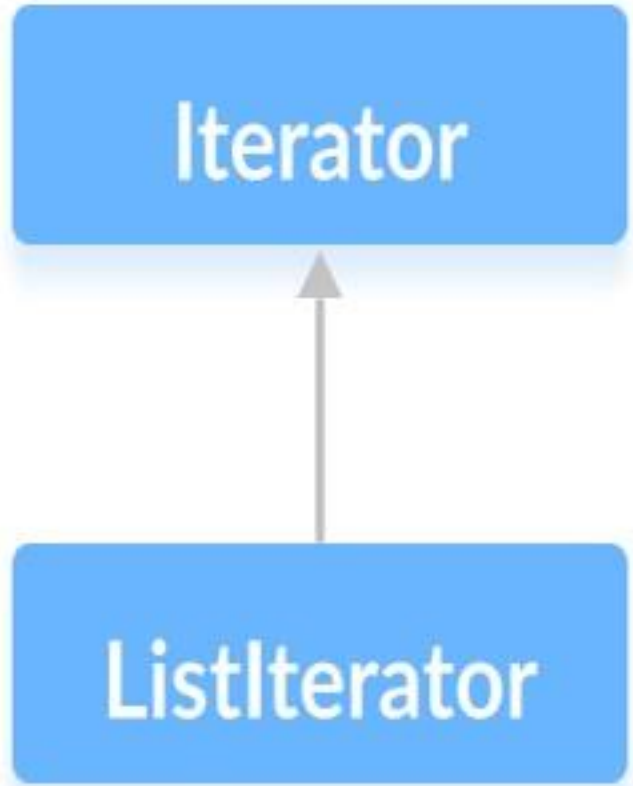
```
import java.util.ArrayList;
import java.util.Iterator;
class Main {
   public static void main(String[] args) {
      // Creating an ArrayList
      ArrayList<Integer> numbers = new ArrayList<>();
      numbers.add(1);
      numbers.add(3);
      numbers.add(2);
      System.out.println("ArrayList: " + numbers);
      // Creating an instance of Iterator
      Iterator<Integer> iterate = numbers.iterator();
      // Using the next() method
      int number = iterate.next();
      System.out.println("Accessed Element: " + number);
      // Using the remove() method
      iterate.remove();
      System.out.println("Removed Element: " + number);
      System.out.print("Updated ArrayList: ");
      // Using the hasNext() method
      while(iterate.hasNext()) {
         // Using the forEachRemaining() method
         iterate.forEachRemaining((value) -> System.out.print(value + ", "));
      }
   }
}
```

Java ListIterator Interface

The ListIterator interface of the Java collections framework provides the functionality to access elements of a list.

It is bidirectional. This means it allows us to iterate elements of a list in both the direction.

It extends the Iterator interface.



Methods of ListIterator

The ListIterator interface provides methods that can be used to perform various operations on the elements of a list.

hasNext() - returns true if there exists an element in the list
next() - returns the next element of the list
nextIndex() returns the index of the element that the next() method will return
previous() - returns the previous element of the list
previousIndex() - returns the index of the element that the previous() method will return
remove() - removes the element returned by either next() or previous()
set() - replaces the element returned by either next() or previous() with the specified element

```
import java.util.ArrayList;
import java.util.ListIterator;

class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(3);
    numbers.add(2);
    System.out.println("ArrayList: " + numbers);
    // Creating an instance of ListIterator
    ListIterator<Integer> iterate = numbers.listIterator();
    // Using the next() method
    int number1 = iterate.next();
    System.out.println("Next Element: " + number1);
    // Using the nextIndex()
    int index1 = iterate.nextIndex();
    System.out.println("Position of Next Element: " + index1);

    // Using the hasNext() method
    System.out.println("Is there any next element? " + iterate.hasNext());
  }
}
```

Java I/O Streams

In Java, streams are the sequence of data that are read from the source and written to the destination.

An input stream is used to read data from the source. And, an output stream is used to write data to the destination.

```
class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

For example, in our first Hello World example, we have used System.out to print a string. Here, the System.out is a type of output stream.

Types of Streams

Depending upon the data a stream holds, it can be classified into:

- Byte Stream
- Character Stream

Byte Stream

Byte stream is used to read and write a single byte (8 bits) of data.

All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

Character Stream

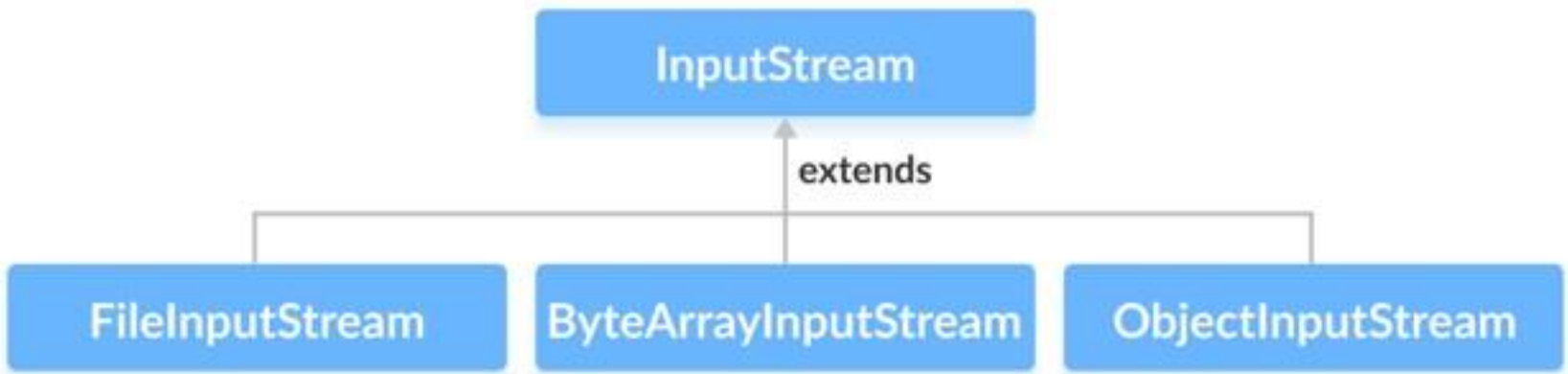
Character stream is used to read and write a single character of data.

All the character stream classes are derived from base abstract classes Reader and Writer.

Java InputStream Class

The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.

Since InputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.



Create an InputStream

In order to create an InputStream, we must import the java.io.InputStream package first. Once we import the package, here is how we can create the input stream.

```
// Creates an InputStream
InputStream object1 = new FileInputStream();
```

Here, we have created an input stream using FileInputStream. It is because InputStream is an abstract class. Hence we cannot create an object of InputStream.

Note: We can also create an input stream from other subclasses of InputStream.

Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

read() - reads one byte of data from the input stream
read(byte[] array) - reads bytes from the stream and stores in the specified array
available() - returns the number of bytes available in the input stream
mark() - marks the position in the input stream up to which data has been read
reset() - returns the control to the point in the stream where the mark was set
markSupported() - checks if the mark() and reset() method is supported in the stream
skips() - skips and discards the specified number of bytes from the input stream
close() - closes the input stream

```
import java.io.FileInputStream;
import java.io.InputStream;

public class Main {
   public static void main(String args[]) {
      byte[] array = new byte[100];
      try {
         InputStream input = new FileInputStream(“FILE path");
         System.out.println("Available bytes in the file: " + input.available());
         // Read byte from the input stream
         input.read(array);
         System.out.println("Data read from the file: ");
         // Convert byte array into string
         String data = new String(array);
         System.out.println(data);
         // Close the input stream
         input.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java OutputStream Class

The OutputStream class of the java.io package is an abstract superclass that represents an output stream of bytes.

Since OutputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Create an OutputStream

In order to create an OutputStream, we must import the java.io.OutputStream package first. Once we import the package, here is how we can create the output stream.

```
// Creates an OutputStream
OutputStream object = new FileOutputStream();
```

Here, we have created an object of output stream using FileOutputStream. It is because OutputStream is an abstract class, so we cannot create an object of OutputStream.

Note: We can also create the output stream from other subclasses of the OutputStream class.

Methods of OutputStream

The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

write() - writes the specified byte to the output stream

write(byte[] array) - writes the bytes from the specified array to the output stream

flush() - forces to write all data present in output stream to the destination

close() - closes the output stream

```
import java.io.FileOutputStream;
import java.io.OutputStream;
public class Main {
   public static void main(String args[]) {
      String data = "This is a line of text inside the file.";
      try {
         OutputStream out = new FileOutputStream("output.txt");
         // Converts the string into bytes
         byte[] dataBytes = data.getBytes();
         // Writes data to the output stream
         out.write(dataBytes);
         System.out.println("Data is written to the file.");
         // Closes the output stream
         out.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java FileInputStream Class

The FileInputStream class of the java.io package can be used to read data (in bytes) from files.

Create a FileInputStream

In order to create a file input stream, we must import the java.io.FileInputStream package first. Once we import the package, here is how we can create a file input stream in Java.

1. Using the path to file

```
FileInputStream input = new FileInputStream(stringPath);
```

Here, we have created an input stream that will be linked to the file specified by the path.

2. Using an object of the file

```
FileInputStream input = new FileInputStream(File fileObject);
```

Here, we have created an input stream that will be linked to the file specified by fileObject.

read() Method

read() - reads a single byte from the file

read(byte[] array) - reads the bytes from the file and stores in the specified array

read(byte[] array, int start, int length) - reads the number of bytes equal to length from the file and stores in the specified array starting from the position start

available() Method

To get the number of available bytes, we can use the available() method. For example,

```
import java.io.FileInputStream;
public class Main {
  public static void main(String args[]) {
    try {
      // Suppose, the input.txt file contains the following text
      // This is a line of text inside the file.
      FileInputStream input = new FileInputStream("input.txt");
      // Returns the number of available bytes
      System.out.println("Available bytes at the beginning: " + input.available());
      // Reads 3 bytes from the file
      input.read();
      input.read();
      input.read();
      // Returns the number of available bytes
      System.out.println("Available bytes at the end: " + input.available());
      input.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
```

skip() Method

To discard and skip the specified number of bytes, we can use the skip() method. For example,

```
import java.io.FileInputStream;
public class Main {
  public static void main(String args[]) {
    try {
      // Suppose, the input.txt file contains the following text
      // This is a line of text inside the file.
      FileInputStream input = new FileInputStream("input.txt");
      // Skips the 5 bytes
      input.skip(5);
      System.out.println("Input stream after skipping 5 bytes:");
      // Reads the first byte
      int i = input.read();
      while (i != -1) {
        System.out.print((char) i);
       // Reads next byte from the file
        i = input.read();
      }
      // close() method
      input.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java FileOutputStream Class

The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.

Create a FileOutputStream

In order to create a file output stream, we must import the java.io.FileOutputStream package first. Once we import the package, here is how we can create a file output stream in Java.

1. Using the path to file

```
// Including the boolean parameter
FileOutputStream output = new FileOutputStream(String path, boolean value);
// Not including the boolean parameter
FileOutputStream output = new FileOutputStream(String path);
```

Here, we have created an output stream that will be linked to the file specified by the path. Also, value is an optional boolean parameter. If it is set to true, the new data will be appended to the end of the existing data in the file. Otherwise, the new data overwrites the existing data in the file.

2. Using an object of the file

FileOutputStream output = new FileOutputStream(File fileObject);

Here, we have created an output stream that will be linked to the file specified by fileObject

write() Method

write() - writes the single byte to the file output stream

write(byte[] array) - writes the bytes from the specified array to the output stream

write(byte[] array, int start, int length) - writes the number of bytes equal to length to the output stream from an array starting from the position start

```
import java.io.FileOutputStream;

public class Main {
   public static void main(String[] args) {
            String data = "This is a line of text inside the file.";
      try {
         FileOutputStream output = new FileOutputStream("output.txt");
         byte[] array = data.getBytes();
         // Writes byte to the file
        output.write(array);
         output.close();
      }
      catch(Exception e) {
         e.getStackTrace();
      }
   }
}
```

Note: The getBytes() method used in the program converts a string into an array of bytes.

flush() Method

To clear the output stream, we can use the flush() method. This method forces the output stream to write all data to the destination. For example,

```
import java.io.FileOutputStream;
import java.io.IOException;
public class Main {
   public static void main(String[] args) throws IOException {
      FileOutputStream out = null;
      String data = "This is demo of flush method";
      try {
         out = new FileOutputStream(" flush.txt");
         // Using write() method
         out.write(data.getBytes());
         // Using the flush() method
         out.flush();
         out.close();
      }
      catch(Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java ByteArrayInputStream Class

The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes).

Note: In ByteArrayInputStream, the input stream is created using the array of bytes. It includes an internal array to store data of that particular byte array.

Create a ByteArrayInputStream

In order to create a byte array input stream, we must import the java.io.ByteArrayInputStream package first. Once we import the package, here is how we can create an input stream.

```
// Creates a ByteArrayInputStream that reads entire array
ByteArrayInputStream input = new ByteArrayInputStream(byte[] arr);
```

Here, we have created an input stream that reads entire data from the arr array. However, we can also create the input stream that reads only some data from the array.

```
// Creates a ByteArrayInputStream that reads a portion of array
ByteArrayInputStream input = new ByteArrayInputStream(byte[] arr, int start, int length);
```

Here the input stream reads the number of bytes equal to length from the array starting from the start position.

```
import java.io.ByteArrayInputStream;
public class Main {
  public static void main(String[] args) {
    // Creates an array of byte
    byte[] array = {1, 2, 3, 4};
    try {
      ByteArrayInputStream input = new ByteArrayInputStream(array);
      System.out.print("The bytes read from the input stream: ");
      for(int i= 0; i < array.length; i++) {
        // Reads the bytes
        int data = input.read();
        System.out.print(data + ", ");
      }
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java ByteArrayOutputStream Class

The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes).

Create a ByteArrayOutputStream

In order to create a byte array output stream, we must import the java.io.ByteArrayOutputStream package first. Once we import the package, here is how we can create an output stream.

```
// Creates a ByteArrayOutputStream with default size
ByteArrayOutputStream out = new ByteArrayOutputStream();
```

Here, we have created an output stream that will write data to an array of bytes with default size 32 bytes. However, we can change the default size of the array.

```
// Creating a ByteArrayOutputStream with specified size
ByteArrayOutputStream out = new ByteArrayOutputStream(int size);
```

Here, the size specifies the length of the array.

```
import java.io.ByteArrayOutputStream;

class Main {
  public static void main(String[] args) {
    String data = "This is a line of text inside the string.";
    try {
      // Creates an output stream
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      byte[] array = data.getBytes();
      // Writes data to the output stream
      out.write(array);
      // Retrieves data from the output stream in string format
      String streamData = out.toString();
      System.out.println("Output stream: " + streamData);
      out.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java ObjectInputStream Class

The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream.

Working of ObjectInputStream

The ObjectInputStream is mainly used to read data written by the ObjectOutputStream.

Basically, the ObjectOutputStream converts Java objects into corresponding streams. This is known as **serialization.** Those converted streams can be stored in files or transferred through networks.

Now, if we need to read those objects, we will use the ObjectInputStream that will convert the streams back to corresponding objects. This is known as **deserialization.**

Create an ObjectInputStream

In order to create an object input stream, we must import the java.io.ObjectInputStream package first. Once we import the package, here is how we can create an input stream.

```
// Creates a file input stream linked with the specified file
FileInputStream fileStream = new FileInputStream(String file);

// Creates an object input stream using the file input stream
ObjectInputStream objStream = new ObjectInputStream(fileStream);
```

In the above example, we have created an object input stream named objStream that is linked with the file input stream named fileStream.

Now, the objStream can be used to read objects from the file.

Methods of ObjectInputStream

read() Method

read() - reads a byte of data from the input stream

readBoolean() - reads data in boolean form

readChar() - reads data in character form

readInt() - reads data in integer form

readObject() - reads the object from the input stream

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
class Main {
  public static void main(String[] args) {
    int data1 = 5;
    String data2 = "Talent Battle";
    try {
      FileOutputStream file = new FileOutputStream("file.txt");
      ObjectOutputStream output = new ObjectOutputStream(file);
      // Writing to the file using ObjectOutputStream
      output.writeInt(data1);
      output.writeObject(data2);
      FileInputStream fileStream = new FileInputStream("file.txt");
      // Creating an object input stream
      ObjectInputStream objStream = new ObjectInputStream(fileStream);
      //Using the readInt() method
      System.out.println("Integer data :" + objStream.readInt());
      // Using the readObject() method
      System.out.println("String data: " + objStream.readObject());
      output.close();
      objStream.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java ObjectOutputStream Class

The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream.

Working of ObjectOutputStream

Basically, the ObjectOutputStream encodes Java objects using the class name and object values. And, hence generates corresponding streams. This process is known as serialization.

Those converted streams can be stored in files and can be transferred among networks.

Note: The ObjectOutputStream class only writes those objects that implement the Serializable interface. This is because objects need to be serialized while writing to the stream

Create an ObjectOutputStream

In order to create an object output stream, we must import the java.io.ObjectOutputStream package first. Once we import the package, here is how we can create an output stream.

```
// Creates a FileOutputStream where objects from ObjectOutputStream are written
FileOutputStream fileStream = new FileOutputStream(String file);

// Creates the ObjectOutputStream
ObjectOutputStream objStream = new ObjectOutputStream(fileStream);
```

In the above example, we have created an object output stream named objStream that is linked with the file output stream named fileStream.

write() Method

write() - writes a byte of data to the output stream

writeBoolean() - writes data in boolean form

writeChar() - writes data in character form

writeInt() - writes data in integer form

writeObject() - writes object to the output stream

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

class Main {
   public static void main(String[] args) {
      int data1 = 5;
      String data2 = “Talent Battle";
      try {
         FileOutputStream file = new FileOutputStream("file.txt");
         // Creates an ObjectOutputStream
         ObjectOutputStream output = new ObjectOutputStream(file);
         // writes objects to output stream
         output.writeInt(data1);
         output.writeObject(data2);
         // Reads data using the ObjectInputStream
         FileInputStream fileStream = new FileInputStream("file.txt");
         ObjectInputStream objStream = new ObjectInputStream(fileStream);
         System.out.println("Integer data :" + objStream.readInt());
         System.out.println("String data: " + objStream.readObject());
         output.close();
         objStream.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java BufferedInputStream Class

The BufferedInputStream class of the java.io package is used with other input streams to read the data (in bytes) more efficiently.

Working of BufferedInputStream

The BufferedInputStream maintains an internal buffer of 8192 bytes.

During the read operation in BufferedInputStream, a chunk of bytes is read from the disk and stored in the internal buffer. And from the internal buffer bytes are read individually.

Hence, the number of communication to the disk is reduced. This is why reading bytes is faster using the BufferedInputStream.

Create a BufferedInputStream

In order to create a BufferedInputStream, we must import the java.io.BufferedInputStream package first. Once we import the package here is how we can create the input stream.

```
// Creates a FileInputStream
FileInputStream file = new FileInputStream(String path);

// Creates a BufferedInputStream
BufferedInputStream buffer = new BufferInputStream(file);
```

In the above example, we have created a BufferdInputStream named buffer with the FileInputStream named file.

Here, the internal buffer has the default size of 8192 bytes. However, we can specify the size of the internal buffer as well.

```
// Creates a BufferedInputStream with specified size internal buffer
BufferedInputStream buffer = new BufferInputStream(file, int size);
```

The buffer will help to read bytes from the files more quickly.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;

class Main {
  public static void main(String[] args) {
    try {
      // Creates a FileInputStream
      FileInputStream file = new FileInputStream("input.txt");
      // Creates a BufferedInputStream
      BufferedInputStream input = new BufferedInputStream(file);
      // Reads first byte from file
      int i = input .read();
      while (i != -1) {
        System.out.print((char) i);
        // Reads next byte from the file
        i = input.read();
      }
      input.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java BufferedOutputStream Class

Working of BufferedOutputStream

The BufferedOutputStream maintains an internal buffer of 8192 bytes.

During the write operation, the bytes are written to the internal buffer instead of the disk. Once the buffer is filled or the stream is closed, the whole buffer is written to the disk.

Hence, the number of communication to the disk is reduced. This is why writing bytes is faster using BufferedOutputStream.

Create a BufferedOutputStream

In order to create a BufferedOutputStream, we must import the java.io.BufferedOutputStream package first. Once we import the package here is how we can create the output stream.

```
// Creates a FileOutputStream
FileOutputStream file = new FileOutputStream(String path);

// Creates a BufferedOutputStream
BufferedOutputStream buffer = new BufferOutputStream(file);
```

In the above example, we have created a BufferdOutputStream named buffer with the FileOutputStream named file.

Here, the internal buffer has the default size of 8192 bytes. However, we can specify the size of the internal buffer as well.

```
// Creates a BufferedOutputStream with specified size internal buffer
BufferedOutputStream buffer = new BufferOutputStream(file, int size);
```

The buffer will help to write bytes to files more quickly.

```
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;

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

    String data = "This is a line of text inside the file";
    try {
      // Creates a FileOutputStream
      FileOutputStream file = new FileOutputStream("output.txt");
      // Creates a BufferedOutputStream
      BufferedOutputStream output = new BufferedOutputStream(file);

      byte[] array = data.getBytes();

      // Writes data to the output stream
      output.write(array);
      output.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java PrintStream Class

The PrintStream class of the java.io package can be used to write output data in commonly readable form (text) instead of bytes.

Working of PrintStream

Unlike other output streams, the PrintStream converts the primitive data (integer, character) into the text format instead of bytes. It then writes that formatted data to the output stream.

And also, the PrintStream class does not throw any input/output exception. Instead, we need to use the checkError() method to find any error in it.

- Note: The PrintStream class also has a feature of auto flushing. This means it forces the output stream to write all the data to the destination under one of the following conditions:

- if newline character \n is written in the print stream
- if the println() method is invoked
- if an array of bytes is written in the print stream

Create a PrintStream

In order to create a PrintStream, we must import the java.io.PrintStream package first. Once we import the package here is how we can create the print stream.

1. Using other output streams

```
// Creates a FileOutputStream
FileOutputStream file = new FileOutputStream(String file);

// Creates a PrintStream
PrintStream output = new PrintStream(file, autoFlush);
```

Here,

we have created a print stream that will write formatted data to the file represented by FileOutputStream

the autoFlush is an optional boolean parameter that specifies whether to perform auto flushing or not

2. Using filename

```
// Creates a PrintStream
PrintStream output = new PrintStream(String file, boolean autoFlush);
```

Here,
we have created a print stream that will write formatted data to the specified file
autoFlush is an optional boolean parameter that specifies whether to perform autoflush or not

Methods of PrintStream

The PrintStream class provides various methods that allow us to print data to the output.

print() Method

print() - prints the specified data to the output stream

println() - prints the data to the output stream along with a new line character at the end

Example: print() method with PrintStream class

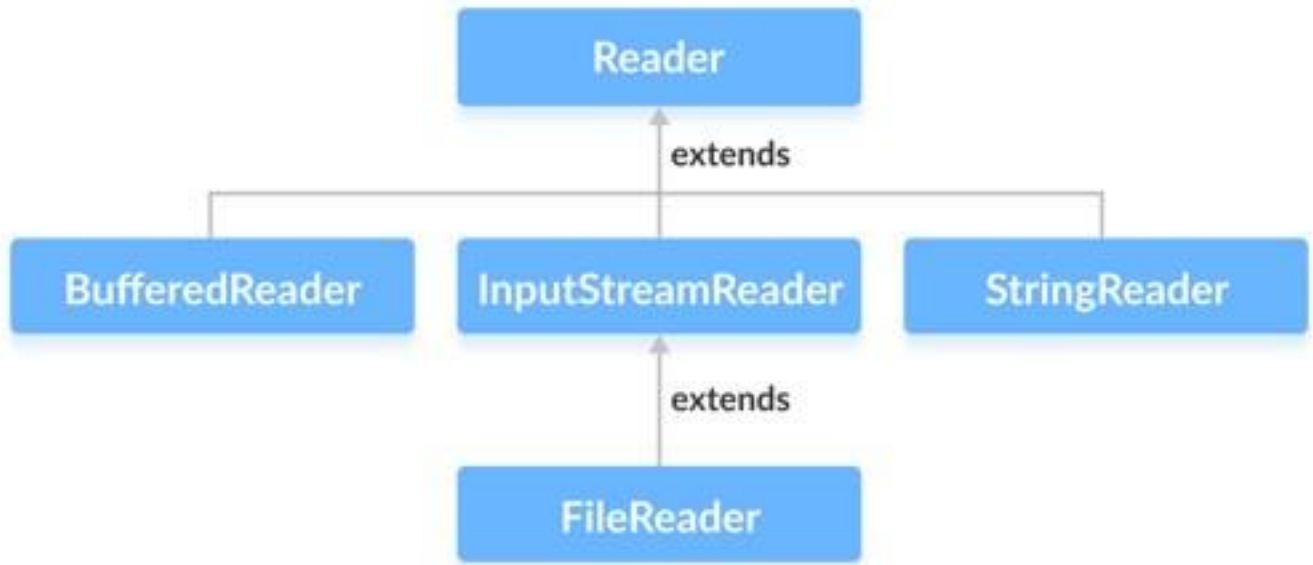
```
import java.io.PrintStream;

class Main {
   public static void main(String[] args) {
      String data = "This is a text inside the file.";
      try {
         PrintStream output = new PrintStream("output.txt");
         output.print(data);
         output.close();
      }
      catch(Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java Reader Class

The Reader class of the java.io package is an abstract superclass that represents a stream of characters.

Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.



Create a Reader

\In order to create a Reader, we must import the java.io.Reader package first. Once we import the package, here is how we can create the reader.

```
// Creates a Reader
Reader input = new FileReader();
```

Here, we have created a reader using the FileReader class. It is because Reader is an abstract class. Hence we cannot create an object of Reader.

Note: We can also create readers from other subclasses of Reader.

Methods of Reader

The Reader class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- ready() - checks if the reader is ready to be read
- read(char[] array) - reads the characters from the stream and stores in the specified array
- read(char[] array, int start, int length) - reads the number of characters equal to length from the stream and stores in the specified array starting from the start
- mark() - marks the position in the stream up to which data has been read
- reset() - returns the control to the point in the stream where the mark is set
- skip() - discards the specified number of characters from the stream

```
import java.io.Reader;
import java.io.FileReader;

class Main {
   public static void main(String[] args) {
      // Creates an array of character
      char[] array = new char[100];
      try {
         // Creates a reader using the FileReader
         Reader input = new FileReader("input.txt");
         // Checks if reader is ready
         System.out.println("Is there data in the stream?  " + input.ready());
        // Reads characters
         input.read(array);
         System.out.println("Data in the stream:");
         System.out.println(array);
         // Closes the reader
         input.close();
      }
      catch(Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java Writer Class

Create a Writer

In order to create a Writer, we must import the java.io.Writer package first. Once we import the package, here is how we can create the writer.

```
// Creates a Writer
Writer output = new FileWriter();
```

Here, we have created a writer named output using the FileWriter class. It is because the Writer is an abstract class. Hence we cannot create an object of Writer.

Note: We can also create writers from other subclasses of the Writer class.

Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

- write(char[] array) - writes the characters from the specified array to the output stream
- write(String data) - writes the specified string to the writer
- append(char c) - inserts the specified character to the current writer
- flush() - forces to write all the data present in the writer to the corresponding destination
- close() - closes the writer

```
import java.io.FileWriter;
import java.io.Writer;

public class Main {
   public static void main(String args[]) {
      String data = "This is the data in the output file";
      try {
         // Creates a Writer using FileWriter
        Writer output = new FileWriter("output.txt");
         // Writes string to the file
         output.write(data);
         // Closes the writer
         output.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Java InputStreamReader Class

The InputStreamReader class of the java.io package can be used to convert data in bytes into data in characters.

Create an InputStreamReader

In order to create an InputStreamReader, we must import the java.io.InputStreamReader package first. Once we import the package here is how we can create the input stream reader.

```
// Creates an InputStream
FileInputStream file = new FileInputStream(String path);

// Creates an InputStreamReader
InputStreamReader input = new InputStreamReader(file);
```

In the above example, we have created an InputStreamReader named input along with the FileInputStream named file.

```
import java.io.InputStreamReader;
import java.io.FileInputStream;

class Main {
 public static void main(String[] args) {
  // Creates an array of character
  char[] array = new char[100];
  try {
   // Creates a FileInputStream
   FileInputStream file = new FileInputStream("input.txt");
   // Creates an InputStreamReader
   InputStreamReader input = new InputStreamReader(file);
   // Reads characters from the file
   input.read(array);
   System.out.println("Data in the stream:");
   System.out.println(array);
   // Closes the reader
   input.close();
  }
  catch(Exception e) {
   e.getStackTrace();
  }
 }
}
```

Java OutputStreamWriter Class

Create an OutputStreamWriter

In order to create an OutputStreamWriter, we must import the java.io.OutputStreamWriter package first. Once we import the package here is how we can create the output stream writer.

```
// Creates an OutputStream
FileOutputStream file = new FileOutputStream(String path);

// Creates an OutputStreamWriter
OutputStreamWriter output = new OutputStreamWriter(file);
```

In the above example, we have created an OutputStreamWriter named output along with the FileOutputStream named file.

```
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;

public class Main {
  public static void main(String args[]) {
    String data = "This is a line of text inside the file.";
    try {
      // Creates a FileOutputStream
      FileOutputStream file = new FileOutputStream("output.txt");
      // Creates an OutputStreamWriter
      OutputStreamWriter output = new OutputStreamWriter(file);
      // Writes string to the file
      output.write(data);
      // Closes the writer
      output.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java StringReader Class

The StringReader class of the java.io package can be used to read data (in characters) from strings.

Note: In StringReader, the specified string acts as a source from where characters are read individually.

Create a StringReader

In order to create a StringReader, we must import the java.io.StringReader package first. Once we import the package here is how we can create the string reader.

```
// Creates a StringReader
StringReader input = new StringReader(String data);
```

Here, we have created a StringReader that reads characters from the specified string named data.

```
import java.io.StringReader;

public class Main {
  public static void main(String[] args) {
    String data = "This is the text read from StringReader.";
    // Create a character array
    char[] array = new char[100];
    try {
      // Create a StringReader
      StringReader input = new StringReader(data);
      //Use the read method
      input.read(array);
      System.out.println("Data read from the string:");
      System.out.println(array);
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java StringWriter Class

Note: In Java, string buffer is considered as a mutable string. That is, we can modify the string buffer. To convert from string buffer to string, we can use the toString() method.

Create a StringWriter

In order to create a StringWriter, we must import the java.io.StringWriter package first. Once we import the package here is how we can create the string writer.

```
// Creates a StringWriter
StringWriter output = new StringWriter();
```

Here, we have created the string writer with default string buffer capacity. However, we can specify the string buffer capacity as well.

```
// Creates a StringWriter with specified string buffer capacity
StringWriter output = new StringWriter(int size);
```

Here, the size specifies the capacity of the string buffer.

```
import java.io.StringWriter;

public class Main {
  public static void main(String[] args) {
    String data = "This is the text in the string.";
   try {
      // Create a StringWriter with default string buffer capacity
      StringWriter output = new StringWriter();
      // Writes data to the string buffer
      output.write(data);
     // Prints the string writer
      System.out.println("Data in the StringWriter: " + output);
      output.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Note: We have used the toString() method to get the output data from string buffer in string form.

```
import java.io.StringWriter;

public class Main {
 public static void main(String[] args) {
  String data = "This is the original data";
  try {
   // Create a StringWriter with default string buffer capacity
   StringWriter output = new StringWriter();
   // Writes data to the string buffer
   output.write(data);
   // Returns the string buffer
   StringBuffer stringBuffer = output.getBuffer();
   System.out.println("StringBuffer: " + stringBuffer);
   // Returns the string buffer in string form
   String string = output.toString();
   System.out.println("String: " + string);
   output.close();
  }
  catch(Exception e) {
   e.getStackTrace();
  }
 }
}
```