# Java I/O Streams

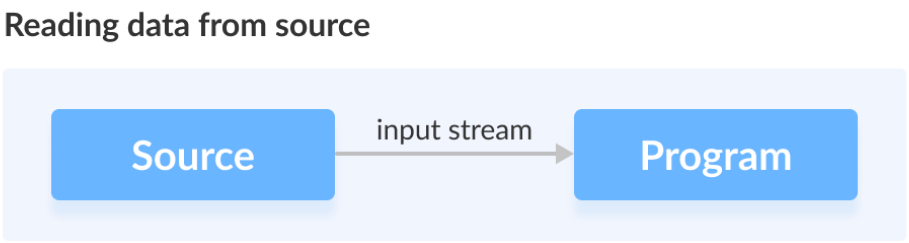
In Java, streams are the sequence of data that are read from the source and written to the destination.

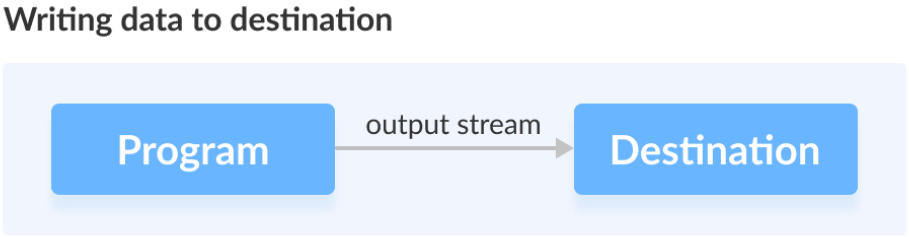
An **input stream** is used to read data from the source. And, an **output stream** is used to write data to the destination.

```
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.

Similarly, there are input streams to take input.




We will learn about input streams and output streams in detail in the later tutorials.

## 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

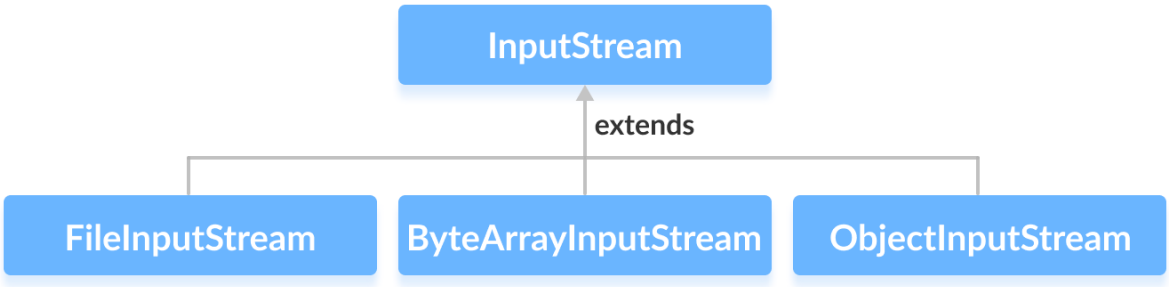
The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.

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

## Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

- FileInputStream
- ByteArrayInputStream
- ObjectInputStream




## 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

### Example: InputStream Using FileInputStream

Here is how we can implement `InputStream` using the `FileInputStream` class.

Suppose we have a file named **input.txt** with the following content.

```
This is a line of text inside the file.
```

Let's try to read this file using FileInputStream (a subclass of InputStream).

```
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("input.txt");

            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();
        }
    }
}
```

### Output

```
Available bytes in the file: 35
Data read from the file:
This is a line of text inside the file
```

In the above example, we have created an input stream using the FileInputStream class. The input stream is linked with the file input.txt.

```
InputStream input = new FileInputStream("input.txt");
```

To read data from the input.txt file, we have implemented these two methods.

```
input.read(array);      // to read data from the input stream
input.close();             // to close the input stream
```

## Java OutputStream Class

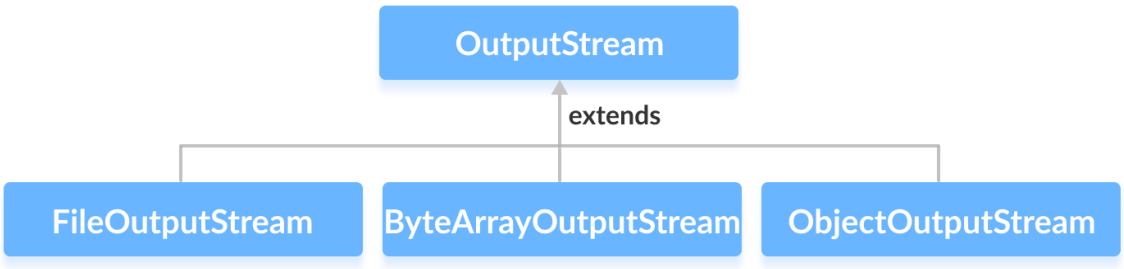
The OutputStream class of the java.io package is an abstract superclass that represents an output stream of bytes.

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

### Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

- FileOutputStream
- ByteArrayOutputStream
- ObjectOutputStream



## 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

## Example: OutputStream Using FileOutputStream

Here is how we can implement `OutputStream` using the `FileOutputStream` class.

```
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();
        }
    }
}
```

In the above example, we have created an output stream using the `FileOutputStream` class. The output stream is now linked with the file **output.txt**.

```
OutputStream out = new FileOutputStream("output.txt");
```

To write data to the **output.txt** file, we have implemented these methods.

```
output.write();      // To write data to the file
output.close();      // To close the output stream
```

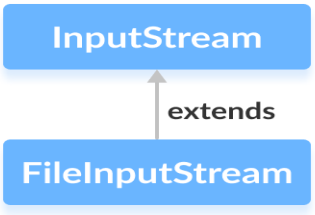
When we run the program, the **output.txt** file is filled with the following content.

```
This is a line of text inside the file.
```

## Java FileInputStream Class

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

It extends the InputStream abstract class.



## 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`.

## Methods of FileInputStream

The FileInputStream class provides implementations for different methods present in the InputStream class.

### 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*

Suppose we have a file named **input.txt** with the following content.

```
This is a line of text inside the file.
```

Let's try to read this file using `FileInputStream`.

```
import java.io.FileInputStream;

public class Main {
```

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

     try {
        FileInputStream input = new FileInputStream("input.txt");

        System.out.println("Data in the file: ");

        // 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();
        }
        input.close();
     }

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

## Output

```
Data in the file:
This is a line of text inside the file.
```

In the above example, we have created a file input stream named *input*. The input stream is linked with the **input.txt** file.

```
FileInputStream input = new FileInputStream("input.txt");
```

To read data from the file, we have used the read() method inside the while loop.

## 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();
      }
   }
}
```

## Output

```
Available bytes at the beginning: 39
Available bytes at the end: 36
```

In the above example,

1. We first use the `available()` method to check the number of available bytes in the file input stream.
2. We then have used the `read()` method 3 times to read 3 bytes from the file input stream.
3. Now, after reading the bytes we again have checked the available bytes. This time the available bytes decreased by 3.

## 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();
      }
   }
}
```

## Output

```
Input Stream after skipping 5 bytes:
is a line of text inside the file.
```

In the above example, we have used the skip() method to skip 5 bytes of data from the file input stream. Hence, the bytes representing the text "This " is not read from the input stream.

### close() Method

To close the file input stream, we can use the close() method. Once the close() method is called, we cannot use the input stream to read data.

In all the above examples, we have used the close() method to close the file input stream.

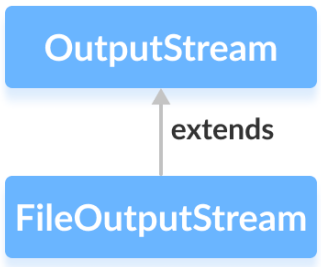
## Other Methods Of FileInputStream

| Methods | Descriptions |
|---|---|
| `finalize()` | ensures that the `close()` method is called |
| `getChannel()` | returns the object of `FileChannel` associated with the input stream |
| `getFD()` | returns the file descriptor associated with the input stream |
| `mark()` | mark the position in input stream up to which data has been read |
| `reset()` | returns the control to the point in the input stream where the mark was set |

# Java FileOutputStream Class

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

It extends the OutputStream abstract class.



## 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.

## Methods of FileOutputStream

The `FileOutputStream` class provides implementations for different methods present in the `OutputStream` class.

### 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*

### Example: FileOutputStream to write data to a File

```
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();
        }
    }
}
```

In the above example, we have created a file output stream named *output*. The file output stream is linked with the file **output.txt**.

```
FileOutputStream output = new FileOutputStream("output.txt");
```

To write data to the file, we have used the `write()` method.

Here, when we run the program, the **output.txt** file is filled with the following content.

```
This is a line of text inside the file.
```

**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();
        }
    }
}
```

When we run the program, the file **flush.txt** is filled with the text represented by the string `data`.

### close() Method

To close the file output stream, we can use the close() method. Once the method is called, we cannot use the methods of FileOutputStream.

## Other Methods Of FileOutputStream

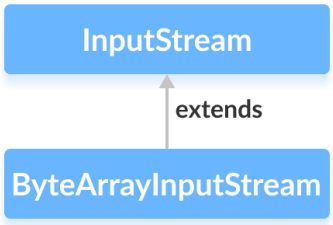
| Methods | Descriptions |
|---|---|
| `finalize()` | ensures that the `close()` method is called |
| `getChannel()` | returns the object of `FileChannel` associated with the output stream |

`getFD()` returns the file descriptor associated with the output stream

# Java ByteArrayInputStream Class

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

It extends the InputStream abstract class.




**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.

## Methods of ByteArrayInputStream

The `ByteArrayInputStream` class provides implementations for different methods present in the `InputStream` class.

### read() Method

- **`read()`** - reads the single byte from the array present in the input stream
- **`read(byte[] array)`** - reads bytes from the input stream and stores in the specified array
- **`read(byte[] array, int start, int length)`** - reads the number of bytes equal to *length* from the stream and stores in the specified array starting from the position *start*

### Example: ByteArrayInputStream to read data

```
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();
    }
  }
}
```

### Output

```
The bytes read from the input stream: 1, 2, 3, 4,
```

In the above example, we have created a byte array input stream named `input`.

```
ByteArrayInputStream input = new ByteArrayInputStream(array);
```

Here, the input stream includes all the data from the specified array. To read data from the input stream, we have used the `read()` method.

## available() Method

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

```
import java.io.ByteArrayInputStream;

public class Main {

  public static void main(String args[]) {

    // Creates an array of bytes
    byte[] array = { 1, 2, 3, 4 };

    try {
      ByteArrayInputStream input = new ByteArrayInputStream(array);

      // Returns the available number of bytes
      System.out.println("Available bytes at the beginning: " +
input.available());

      // Reads 2 bytes from the input stream
      input.read();
      input.read();

      // Returns the available number of bytes
      System.out.println("Available bytes at the end: " +
input.available());

      input.close();
    }

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

**Output**

```
Available bytes at the beginning: 4
Available bytes at the end: 2
```

In the above example,

1. We have used the available() method to check the number of available bytes in the input stream.
2. We have then used the read() method 2 times to read 2 bytes from the input stream.
3. Now, after reading the 2 bytes, we have checked the available bytes. This time the available bytes decreased by 2.

## skip() Method

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

```
import java.io.ByteArrayInputStream;

public class Main {

  public static void main(String args[]) {

    // Create an array of bytes
    byte[] array = { 1, 2, 3, 4 };
    try {
      ByteArrayInputStream input = new ByteArrayInputStream(array);

      // Using the skip() method
      input.skip(2);
      System.out.print("Input stream after skipping 2 bytes: ");

      int data = input.read();
      while (data != -1) {
        System.out.print(data + ", ");
        data = input.read();
      }

      // close() method
      input.close();
    }

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

**Output**

```
Input stream after skipping 2 bytes: 3, 4,
```

In the above example, we have used the skip() method to skip 2 bytes of data from the input stream. Hence 1 and 2 are not read from the input stream.

### close() Method

To close the input stream, we can use the `close()` method.

However, the `close()` method has no effect in `ByteArrayInputStream` class. We can use the methods of this class even after the `close()` method is called.

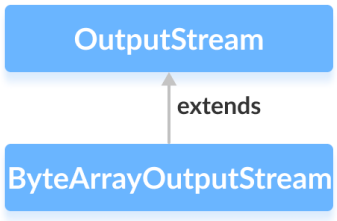
## Other Methods Of ByteArrayInputStream

| Methods | Descriptions |
|---|---|
| `finalize()` | ensures that the `close()` method is called |
| `mark()` | marks the position in input stream up to which data has been read |
| `reset()` | returns the control to the point in the input stream where the mark was set |
| `markSupported()` | checks if the input stream supports `mark()` and `reset()` |

# Java ByteArrayOutputStream Class

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

It extends the OutputStream abstract class.




**Note:** In ByteArrayOutputStream maintains an internal array of bytes to store the data.

## 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.

## Methods of ByteArrayOutputStream

The `ByteArrayOutputStream` class provides the implementation of the different methods present in the `OutputStream` class.

### write() Method

- **`write(int byte)`** - writes the specified byte to the output stream
- **`write(byte[] array)`** - writes the bytes from the specified array to the output stream
- **`write(byte[] arr, int start, int length)`** - writes the number of bytes equal to *length* to the output stream from an array starting from the position *start*
- **`writeTo(ByteArrayOutputStream out1)`** - writes the entire data of the current output stream to the specified output stream

### Example: ByteArrayOutputStream to write data

```
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();
    }
  }
}
```

### Output

```
Output stream: This is a line of text inside the string.
```

In the above example, we have created a byte array output stream named *output*.

ByteArrayOutputStream output = new ByteArrayOutputStream();

To write the data to the output stream, we have used the write() method.

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

## Access Data from ByteArrayOutputStream

- **`toByteArray()`** - returns the array present inside the output stream
- **`toString()`** - returns the entire data of the output stream in string form

For example,

```
import java.io.ByteArrayOutputStream;

class Main {
  public static void main(String[] args) {
    String data = "This is data.";

    try {
      // Creates an output stream
      ByteArrayOutputStream out = new ByteArrayOutputStream();
```

```
      // Writes data to the output stream
      out.write(data.getBytes());

      // Returns an array of bytes
      byte[] byteData = out.toByteArray();
      System.out.print("Data using toByteArray(): ");
      for(int i=0; i<byteData.length; i++) {
        System.out.print((char)byteData[i]);
      }

      // Returns a string
      String stringData = out.toString();
      System.out.println("\nData using toString(): " + stringData);

      out.close();
    }

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

## Output

```
Data using toByteArray(): This is data.
Data using toString(): This is data.
```

In the above example, we have created an array of bytes to store the data returned by the toByteArray() method.

We then have used the for loop to access each byte from the array. Here, each byte is converted into the corresponding character using typecasting.

## close() Method

To close the output stream, we can use the close() method.

However, the close() method has no effect in ByteArrayOutputStream class. We can use the methods of this class even after the close() method is called.

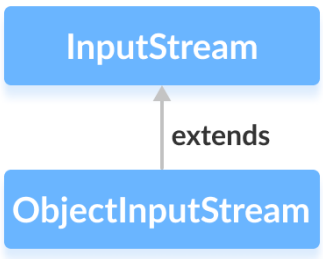
## Other Methods of ByteArrayOutputStream

| Methods | Descriptions |
|---|---|
| `size()` | returns the size of the array in the output stream |
| `flush()` | clears the output stream |

## Java ObjectInputStream Class

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

It extends the InputStream abstract class.




## 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

The ObjectInputStream class provides implementations of different methods present in the InputStream class.

### 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

### Example 1: Java ObjectInputStream

Let's see how we can use the `ObjectInputStream` class to read objects written by the `ObjectOutputStream` class.

```
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 = "This is programiz";

        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();
        }
    }
}
```

## Output

```
Integer data: 5
String data: This is programiz
```

In the above example, we have used the readInt() and readObject() method to read integer data and object data from the file.

Here, we have used the ObjectOutputStream to write data to the file. We then read the data from the file using the ObjectInputStream.

## Example 2: Java ObjectInputStream

Let's see another practical example,

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

class Dog implements Serializable {

    String name;
    String breed;

    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }
}

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

        // Creates an object of Dog class
        Dog dog = new Dog("Tyson", "Labrador");

        try {
            FileOutputStream file = new FileOutputStream("file.txt");

            // Creates an ObjectOutputStream
            ObjectOutputStream output = new ObjectOutputStream(file);

            // Writes objects to the output stream
            output.writeObject(dog);

            FileInputStream fileStream = new
FileInputStream("file.txt");

            // Creates an ObjectInputStream
            ObjectInputStream input = new
ObjectInputStream(fileStream);

            // Reads the objects
            Dog newDog = (Dog) input.readObject();

            System.out.println("Dog Name: " + newDog.name);
            System.out.println("Dog Breed: " + newDog.breed);

            output.close();
```

```
            input.close();
        }

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

### Output

```
Dog Name: Tyson
Dog Breed: Labrador
```

In the above example, we have created

- ObjectOutputStream named output using the FileOutputStream named file
- ObjectInputStream named input using the FileInputStream named fileStream
- An object dog of the Dog class

Here, we have then used the object output stream to write the object to the file. And, the object input stream to read the object from the file.

Note: The Dog class implements the Serializable interface. It is because the ObjectOutputStream only writes the serializable objects to the output stream.

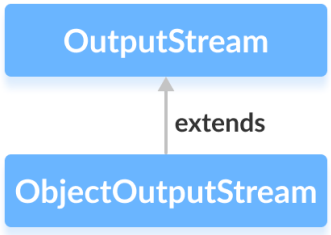
## Other Methods Of ObjectInputStream

| Methods | Descriptions |
|---|---|
| `available()` | returns the available number of bytes in the input stream |
| `mark()` | marks the position in input stream up to which data has been read |
| `reset()` | returns the control to the point in the input stream where the mark was set |
| `skipBytes()` | skips and discards the specified bytes from the input stream |
| `close()` | closes the object input stream |

# Java ObjectOutputStream Class

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

It extends the OutputStream abstract class.




## 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*.

## Methods of ObjectOutputStream

The ObjectOutputStream class provides implementations for different methods present in the OutputStream class.

### 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

### Example 1: Java ObjectOutputStream

Let's see how we can use `ObjectOutputStream` to store objects in a file and `ObjectInputStream` to read those objects from the files

```
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 = "This is programiz";

        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();
        }
    }
}
```

## Output

```
Integer data: 5
String data: This is programiz
```

In the above example, we have used the readInt() method and readObject() method to read an integer data and object data from the files.

Here, we have used the ObjectOutputStream to write data to the file. We then read the data from the file using the ObjectInputStream.

## Example 2: Java ObjectOutputStream

Let's take another example,

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

class Dog implements Serializable {

    String name;
    String breed;

    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }
}
```

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

        // Creates an object of Dog class
        Dog dog1 = new Dog("Tyson", "Labrador");

        try {
            FileOutputStream fileOut = new
FileOutputStream("file.txt");

            // Creates an ObjectOutputStream
            ObjectOutputStream objOut = new
ObjectOutputStream(fileOut);

            // Writes objects to the output stream
            objOut.writeObject(dog1);

            // Reads the object
            FileInputStream fileIn = new FileInputStream("file.txt");
            ObjectInputStream objIn = new ObjectInputStream(fileIn);

            // Reads the objects
            Dog newDog = (Dog) objIn.readObject();

            System.out.println("Dog Name: " + newDog.name);
            System.out.println("Dog Breed: " + newDog.breed);

            objOut.close();
            objIn.close();
        }

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

## Output

```
Dog Name: Tyson
Dog Breed: Labrador
```

In the above example, we have created

- ObjectOutputStream named objOut using the FileOutputStream named fileOut
- ObjectInputStream named objIn using the FileInputStream named fileIn.
- An object dog1 of the Dog class.

Here, we have then used the object output stream to write the object to the file. And, the object input stream to read the object from the file.

Note: The Dog class implements the Serializable interface. It is because the ObjectOutputStream only writes objects that can be serialized to the output stream.

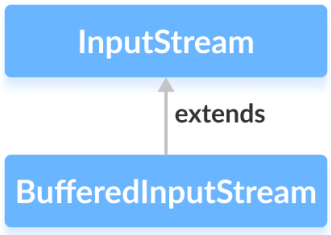
## Other Methods Of ObjectOutputStream

| Methods | Descriptions |
|---|---|
| `flush()` | clears all the data from the output stream |
| `drain()` | puts all the buffered data in the output stream |
| `close()` | closes the output stream |

# 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.

It extends the InputStream abstract class.




### 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.

## Methods of BufferedInputStream

The BufferedInputStream class provides implementations for different methods present in the InputStream class.

### read() Method

- **`read()`** - reads a single byte from the input stream
- **`read(byte[] arr)`** - reads bytes from the stream and stores in the specified array
- **`read(byte[] arr, int start, int length)`** - reads the number of bytes equal to the *length* from the stream and stores in the specified array starting from the position *start.*

Suppose we have a file named **input.txt** with the following content.

```
This is a line of text inside the file.
```

Let's try to read the file using `BufferedInputStream`.

```
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();
        }
    }
}
```

## Output

```
This is a line of text inside the file.
```

In the above example, we have created a buffered input stream named *buffer* along with `FileInputStream`. The input stream is linked with the file **input.txt**.

```
FileInputStream file = new FileInputStream("input.txt");
BufferedInputStream buffer = new BufferedInputStream(file);
```

Here, we have used the `read()` method to read an array of bytes from the internal buffer of the buffered reader.

### available() Method

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

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

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 file = new FileInputStream("input.txt");

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

         // Returns the available number of bytes
         System.out.println("Available bytes at the beginning: " +
buffer.available());

         // Reads bytes from the file
         buffer.read();
         buffer.read();
         buffer.read();

         // Returns the available number of bytes
         System.out.println("Available bytes at the end: " +
buffer.available());

         buffer.close();
      }

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

### Output

```
Available bytes at the beginning: 39
```

```
Available bytes at the end: 36
```

In the above example,

1. We first use the `available()` method to check the number of available bytes in the input stream.
2. Then, we have used the `read()` method 3 times to read 3 bytes from the input stream.
3. Now, after reading the bytes we again have checked the available bytes. This time the available bytes decreased by 3.

## skip() Method

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

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

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 file = new FileInputStream("input.txt");

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

         // Skips the 5 bytes
         buffer.skip(5);
         System.out.println("Input stream after skipping 5 bytes:");

         // Reads the first byte from input stream
         int i = buffer.read();
         while (i != -1) {
            System.out.print((char) i);

            // Reads next byte from the input stream
            i = buffer.read();
         }

         // Closes the input stream
         buffer.close();
      }
```

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

### Output

```
Input stream after skipping 5 bytes: is a line of text inside the
file.
```

In the above example, we have used the `skip()` method to skip 5 bytes from the file input stream. Hence, the bytes `'T'`, `'h'`, `'i'`, `'s'` and `' '` are skipped from the input stream.

### close() Method

To close the buffered input stream, we can use the `close()` method. Once the `close()` method is called, we cannot use the input stream to read the data.

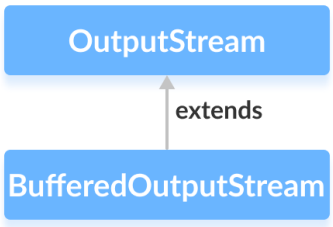
## Other Methods Of BufferedInputStream

| Methods | Descriptions |
|---|---|
| `mark()` | mark the position in input stream up to which data has been read |
| `reset()` | returns the control to the point in the input stream where the mark was set |

# Java BufferedOutputStream Class

The BufferedOutputStream class of the java.io package is used with other output streams to write the data (in bytes) more efficiently.

It extends the OutputStream abstract 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.

## Methods of BufferedOutputStream

The BufferedOutputStream class provides implementations for different methods in the OutputStream class.

## write() Method

- **`write()`** - writes a single byte to the internal buffer of the output stream
- **`write(byte[] array)`** - writes the bytes from the specified array to the output stream
- **`write(byte[] arr, int start, int length)`** - writes the number of bytes equal to *length* to the output stream from an array starting from the position *start*

## Example: BufferedOutputStream to write data to a File

```
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();
        }
    }
}
```

In the above example, we have created a buffered output stream named *output* along with FileOutputStream. The output stream is linked with the file **output.txt**.

```
FileOutputStream file = new FileOutputStream("output.txt");
BufferedOutputStream output = new BufferedOutputStream(file);
```

To write data to the file, we have used the write() method.

Here when we run the program, the **output.txt** file is filled with the following content.

```
This is a line of text inside the file.
```

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

### flush() Method

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

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

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

        String data = "This is a demo of the flush method";

        try {
            // Creates a FileOutputStream
            FileOutputStream file = new FileOutputStream("
flush.txt");

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

            // Writes data to the output stream
            buffer.write(data.getBytes());
```

```
            // Flushes data to the destination
            buffer.flush();
            System.out.println("Data is flushed to the file.");
            buffer.close();
        }

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

### Output

```
Data is flushed to the file.
```

When we run the program, the file **flush.txt** is filled with the text represented by the string *data*.

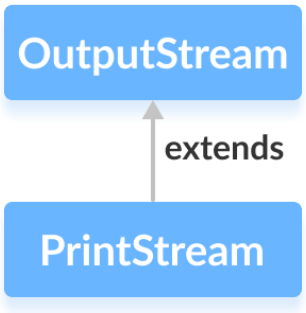
### close() Method

To close the buffered output stream, we can use the `close()` method. Once the method is called, we cannot use the output stream to write the data.

## 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.

It extends the abstract class OutputStream.



## 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

**Note**: In both the case, the PrintStream write data to the file using some default character encoding. However, we can specify the character encoding (**UTF8** or **UTF16**) as well.

```
// Creates a PrintStream using some character encoding
PrintStream output = new PrintStream(String file, boolean autoFlush,
Charset cs);
```

Here, we have used the Charset class to specify the character encoding.

## 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 System class

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

        String data = "Hello World.";
        System.out.print(data);
    }
}
```

### Output

```
Hello World.
```

In the above example, we have not created a print stream. However, we can use the print() method of the PrintStream class.

You might be wondering how is this possible. Well, let me explain what is happening here.

Notice the line,

```
System.out.print(data);
```

Here,

- System is a final class that is responsible to perform standard input/output operation
- out is a class variable of PrintStream type declared in System class

Now since out is of PrintStream type, we can use it to call all the methods of PrintStream class.

**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();
        }
    }
}
```

In the above example, we have created a print stream named *output*. The print stream is linked with the **output.txt** file.

```
PrintStream output = new PrintStream("output.txt");
```

To print data to the file, we have used the print() method.

Here, when we run the program, the **output.txt** file is filled with the following content.

```
This is a text inside the file.
```

### printf() Method

The printf() method can be used to print the formatted string. It includes 2 parameters: formatted string and arguments. For example,

```
printf("I am %d years old", 25);
```

Here,

- I am %d years old is a formatted string
- %d is integer data in the formatted string
- 25 is an argument

The formatted string includes both text and data. And, the arguments replace the data inside the formatted string.

Hence the **%d** is replaced by **25**.

### Example: printf() method using PrintStream

```
import java.io.PrintStream;

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

        try {
            PrintStream output = new PrintStream("output.txt");

            int age = 25;

            output.printf("I am %d years old.", age);
            output.close();
        }
        catch(Exception e) {
            e.getStackTrace();
        }
    }
}
```

In the above example, we have created a print stream named *output*. The print stream is linked with the file **output.txt**.

```
PrintStream output = new PrintStream("output.txt");
```

To print the formatted text to the file, we have used the `printf()` method.

Here, when we run the program, the **output.txt** file is filled with the following content.

```
I am 25 years old.
```

## Other Methods Of PrintStream

| Methods | Descriptions |
|---|---|
| `close()` | closes the print stream |
| `checkError()` | checks if there is an error in the stream and returns a boolean result |
| `append()` | appends the specified data to the stream |