Java Reader Class

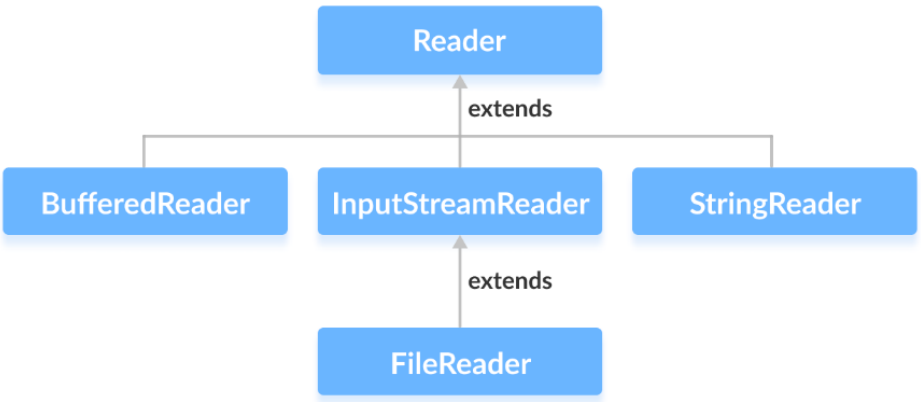
The `Reader` class of the `java.io` package is an abstract superclass that represents a stream of characters.

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

Subclasses of Reader

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

- BufferedReader
- InputStreamReader
- FileReader
- StringReader



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

Example: Reader Using FileReader

Here is how we can implement `Reader` using the `FileReader` 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 FileReader (a subclass of Reader).

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

Output

```
Is there data in the stream?  true
Data in the stream:
This is a line of text inside the file.
```

In the above example, we have created a reader using the `FileReader` class. The reader is linked with the file **input.txt**.

```
Reader input = new FileReader("input.txt");
```

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

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

Java Writer Class

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

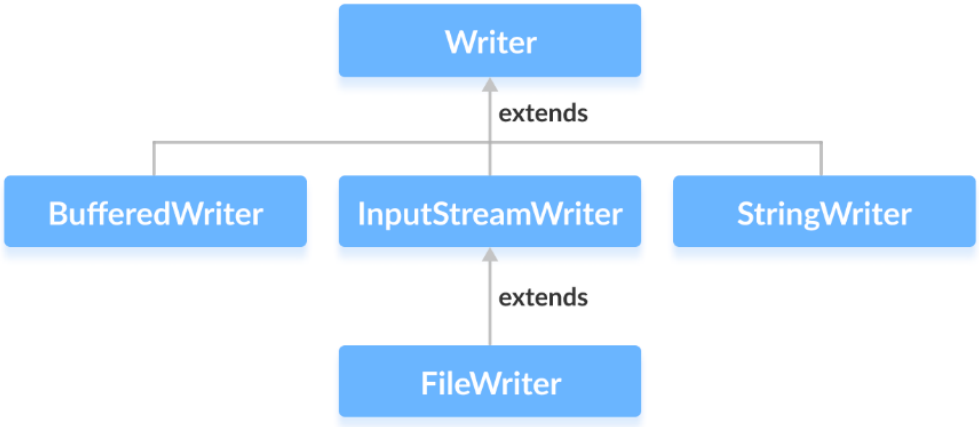
Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of Writer

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

- BufferedWriter
- OutputStreamWriter
- FileWriter

- StringWriter



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

Example: Writer Using FileWriter

Here is how we can implement the `Writer` using the `FileWriter` class.

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

In the above example, we have created a writer using the `FileWriter` class. The writer is linked with the file **output.txt**.

```
Writer output = new FileWriter("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 writer
```

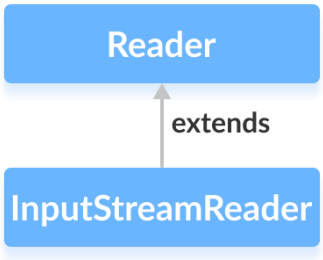
When we run the program, the **output.txt** file is filled with the following content.

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

Java InputStreamReader Class

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

It extends the abstract class Reader.



The InputStreamReader class works with other input streams. It is also known as a bridge between byte streams and character streams. This is because the InputStreamReader reads bytes from the input stream as characters.

For example, some characters required 2 bytes to be stored in the storage. To read such data we can use the input stream reader that reads the 2 bytes together and converts into the corresponding character.

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*.

Here, the data in the file are stored using some default character encoding.

However, we can specify the type of character encoding (**UTF8** or **UTF16**) in the file as well.

```
// Creates an InputStreamReader specifying the character encoding
InputStreamReader input = new InputStreamReader(file, Charset cs);
```

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

Methods of InputStreamReader

The InputStreamReader class provides implementations for different methods present in the Reader class.

read() Method

- **`read()`** - reads a single character from the reader
- **`read(char[] array)`** - reads the characters from the reader and stores in the specified array
- **`read(char[] array, int start, int length)`** - reads the number of characters equal to *length* from the reader and stores in the specified array starting from the *start*

For example, 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 `InputStreamReader`.

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

Output

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

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

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

To read characters from the file, we have used the read() method.

getEncoding() Method

The `getEncoding()` method can be used to get the type of encoding that is used to store data in the input stream. For example,

```
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.io.FileInputStream;

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

    try {
      // Creates a FileInputStream
      FileInputStream file = new FileInputStream("input.txt");

      // Creates an InputStreamReader with default encoding
      InputStreamReader input1 = new InputStreamReader(file);

      // Creates an InputStreamReader specifying the encoding
      InputStreamReader input2 = new InputStreamReader(file,
Charset.forName("UTF8"));
```

```
      // Returns the character encoding of the input stream
      System.out.println("Character encoding of input1: " +
input1.getEncoding());
      System.out.println("Character encoding of input2: " +
input2.getEncoding());

      // Closes the reader
      input1.close();
      input2.close();
    }

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

Output

```
The character encoding of input1: Cp1252
The character encoding of input2: UTF8
```

In the above example, we have created 2 input stream reader named input1 and input2.

- input1 does not specify the character encoding. Hence the getEncoding() method returns the canonical name of the default character encoding.
- input2 specifies the character encoding, UTF8. Hence the getEncoding() method returns the specified character encoding.

Note: We have used the Charset.forName() method to specify the type of character encoding.

close() Method

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

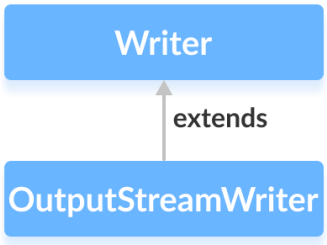
Other Methods of InputStreamReader

Method	Description
`ready()`	checks if the stream is ready to be read
`mark()`	mark the position in stream up to which data has been read
`reset()`	returns the control to the point in the stream where the mark was set

Java OutputStreamWriter Class

The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form.

It extends the abstract class Writer.



The OutputStreamWriter class works with other output streams. It is also known as a bridge between byte streams and character streams. This is because the OutputStreamWriter converts its characters into bytes.

For example, some characters require 2 bytes to be stored in the storage. To write such data we can use the output stream writer that converts the character into corresponding bytes and stores the bytes together.

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.

Here, we are using the default character encoding to write characters to the output stream.

However, we can specify the type of character encoding (**UTF8** or **UTF16**) to be used to write data.

```
// Creates an OutputStreamWriter specifying the character encoding
OutputStreamWriter output = new OutputStreamWriter(file, Charset cs);
```

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

Methods of OutputStreamWriter

The OutputStreamWriter class provides implementations for different methods present in the Writer class.

write() Method

- **`write()`** - writes a single character to the writer
- **`write(char[] array)`** - writes the characters from the specified array to the writer
- **`write(String data)`** - writes the specified string to the writer

Example: OutputStreamWriter to write data to a 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();
    }
  }
}
```

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

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

getEncoding() Method

The `getEncoding()` method can be used to get the type of encoding that is used to write data to the output stream. For example,

```
import java.io.OutputStreamWriter;
import java.nio.charset.Charset;
import java.io.FileOutputStream;

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

    try {
      // Creates an output stream
      FileOutputStream file = new FileOutputStream("output.txt");
```

```
      // Creates an output stream reader with default encoding
      OutputStreamWriter output1 = new OutputStreamWriter(file);

      // Creates an output stream reader specifying the encoding
      OutputStreamWriter output2 = new OutputStreamWriter(file,
Charset.forName("UTF8"));

      // Returns the character encoding of the output stream
      System.out.println("Character encoding of output1: " +
output1.getEncoding());
      System.out.println("Character encoding of output2: " +
output2.getEncoding());

      // Closes the reader
      output1.close();
      output2.close();
    }

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

Output

```
The character encoding of output1: Cp1252
The character encoding of output2: UTF8
```

In the above example, we have created 2 output stream writer named output1 and output2.

- output1 does not specify the character encoding. Hence the getEncoding() method returns the default character encoding.
- output2 specifies the character encoding, UTF8. Hence the getEncoding() method returns the specified character encoding.

Note: We have used the Charset.forName() method to specify the type of character encoding.

close() Method

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

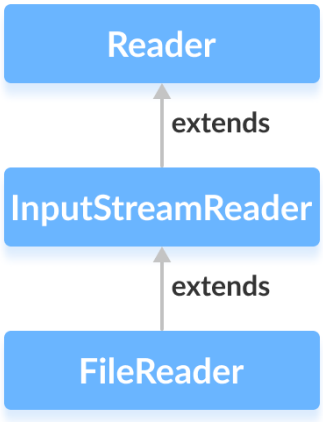
Other methods of OutputStreamWriter

Method	Description
`flush()`	forces to write all the data present in the writer to the corresponding destination
`append()`	inserts the specified character to the current writer

Java FileReader Class

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

It extends the InputSreamReader class.



Create a FileReader

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

1. Using the name of the file

FileReader input = new FileReader(String name);

Here, we have created a file reader that will be linked to the file specified by the *name*.

2. Using an object of the file

```
FileReader input = new FileReader(File fileObj);
```

Here, we have created a file reader that will be linked to the file specified by the object of the file.

In the above example, the data in the file are stored using some default character encoding.

However, since Java 11 we can specify the type of character encoding (**UTF-8** or **UTF-16**) in the file as well.

```
FileReader input = new FileReader(String file, Charset cs);
```

Here, we have used the `Charset` class to specify the character encoding of the file reader.

Methods of FileReader

The FileReader class provides implementations for different methods present in the Reader class.

read() Method

- **`read()`** - reads a single character from the reader
- **`read(char[] array)`** - reads the characters from the reader and stores in the specified array
- **`read(char[] array, int start, int length)`** - reads the number of characters equal to *length* from the reader and stores in the specified array starting from the position *start*

For example, 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 `FileReader`.

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

      // Reads characters
      input.read(array);
      System.out.println("Data in the file: ");
      System.out.println(array);

      // Closes the reader
      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 reader named *input*. The file reader is linked with the file **input.txt**.

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

getEncoding() Method

The `getEncoding()` method can be used to get the type of encoding that is used to store data in the file. For example,

```
import java.io.FileReader;
import java.nio.charset.Charset;

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

    try {
      // Creates a FileReader with default encoding
      FileReader input1 = new FileReader("input.txt");

      // Creates a FileReader specifying the encoding
      FileReader input2 = new FileReader("input.txt",
Charset.forName("UTF8"));
```

```
      // Returns the character encoding of the file reader
      System.out.println("Character encoding of input1: " +
input1.getEncoding());
      System.out.println("Character encoding of input2: " +
input2.getEncoding());

      // Closes the reader
      input1.close();
      input2.close();
    }

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

Output

```
The character encoding of input1: Cp1252
The character encoding of input2: UTF8
```

In the above example, we have created 2 file reader named *input1* and *input2*.

- *input1* does not specify the character encoding. Hence the getEncoding() method returns the default character encoding.
- *input2* specifies the character encoding, **UTF8**. Hence the getEncoding() method returns the specified character encoding.

Note: We have used the Charset.forName() method to specify the type of character encoding.

close() Method

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

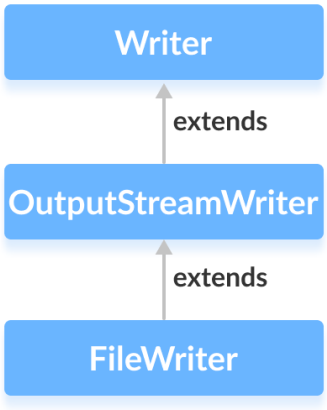
Other Methods of FileReader

Method	Description
`ready()`	checks if the file reader is ready to be read
`mark()`	mark the position in file reader up to which data has been read
`reset()`	returns the control to the point in the reader where the mark was set

Java FileWriter Class

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

It extends the OutputStreamWriter class.



Create a FileWriter

In order to create a file writer, we must import the `Java.io.FileWriter` package first. Once we import the package, here is how we can create the file writer.

1. Using the name of the file

```
FileWriter output = new FileWriter(String name);
```

Here, we have created a file writer that will be linked to the file specified by the *name*.

2. Using an object of the file

```
FileWriter  input = new FileWriter(File fileObj);
```

Here, we have created a file writer that will be linked to the file specified by the object of the file.

In the above example, the data are stored using some default character encoding.

However, since Java 11 we can specify the type of character encoding (**UTF8** or **UTF16**) as well.

```
FileWriter input = new FileWriter(String file, Charset cs);
```

Here, we have used the `Charset` class to specify the character encoding of the file writer.

Methods of FileWriter

The FileWriter class provides implementations for different methods present in the Writer class.

write() Method

- **`write()`** - writes a single character to the writer
- **`write(char[] array)`** - writes the characters from the specified array to the writer
- **`write(String data)`** - writes the specified string to the writer

Example: FileWriter to write data to a File

```
import java.io.FileWriter;

public class Main {

  public static void main(String args[]) {

    String data = "This is the data in the output file";

    try {
      // Creates a FileWriter
      FileWriter output = new FileWriter("output.txt");

      // Writes the string to the file
      output.write(data);
```

```
      // Closes the writer
      output.close();
    }

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

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

```
FileWriter output = new FileWriter("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.
```

getEncoding() Method

The `getEncoding()` method can be used to get the type of encoding that is used to write data. For example,

```
import java.io.FileWriter;
import java.nio.charset.Charset;

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

    String file = "output.txt";

    try {
      // Creates a FileReader with default encoding
      FileWriter output1 = new FileWriter(file);

      // Creates a FileReader specifying the encoding
      FileWriter output2 = new FileWriter(file,
Charset.forName("UTF8"));

      // Returns the character encoding of the reader
      System.out.println("Character encoding of output1: " +
output1.getEncoding());
      System.out.println("Character encoding of output2: " +
output2.getEncoding());
```

```
      // Closes the reader
      output1.close();
      output2.close();
    }

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

Output

```
The character encoding of output1: Cp1252
The character encoding of output2: UTF8
```

In the above example, we have created 2 file writer named output1 and output2.

- output1 does not specify the character encoding. Hence the getEncoding() method returns the default character encoding.
- output2 specifies the character encoding, UTF8. Hence the getEncoding() method returns the specified character encoding.

Note: We have used the `Charset.forName()` method to specify the type of character encoding.

close() Method

To close the file writer, we can use the close() method. Once the close() method is called, we cannot use the writer to write the data.

Other methods of FileWriter

Method	Description
`flush()`	forces to write all the data present in the writer to the corresponding destination
`append()`	inserts the specified character to the current writer

Java BufferedReader Class

The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently.

It extends the abstract class Reader.

Working of BufferedReader

The BufferedReader maintains an internal **buffer of 8192 characters**.

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

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

Create a BufferedReader

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

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

// Creates a BufferedReader
BufferedReader buffer = new BufferedReader(file);
```

In the above example, we have created a `BufferedReader` named *buffer* with the `FileReader` named *file*.

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

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

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

Methods of BufferedReader

The BufferedReader class provides implementations for different methods present in Reader.

read() Method

- **`read()`** - reads a single character from the internal buffer of the reader
- **`read(char[] array)`** - reads the characters from the reader and stores in the specified array
- **`read(char[] array, int start, int length)`** - reads the number of characters equal to *length* from the reader and stores in the specified array starting from the position *start*

For example, 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 `BufferedReader`.

```
import java.io.FileReader;
import java.io.BufferedReader;

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

    // Creates an array of character
    char[] array = new char[100];

    try {
      // Creates a FileReader
      FileReader file = new FileReader("input.txt");

      // Creates a BufferedReader
      BufferedReader input = new BufferedReader(file);

      // Reads characters
      input.read(array);
      System.out.println("Data in the file: ");
      System.out.println(array);

      // Closes the reader
      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 buffered reader named *input*. The buffered reader is linked with the **input.txt** file.

```
FileReader file = new FileReader("input.txt");
BufferedReader input = new BufferedReader(file);
```

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

skip() Method

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

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {

  public static void main(String args[]) {

    // Creates an array of characters
    char[] array = new char[100];

    try {
      // Suppose, the input.txt file contains the following text
      // This is a line of text inside the file.
      FileReader file = new FileReader("input.txt");

      // Creates a BufferedReader
      BufferedReader input = new BufferedReader(file);

      // Skips the 5 characters
      input.skip(5);

      // Reads the characters
      input.read(array);
```

```
      System.out.println("Data after skipping 5 characters:");
      System.out.println(array);

      // closes the reader
      input.close();
    }

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

Output

```
Data after skipping 5 characters:
is a line of text inside the file.
```

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

close() Method

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

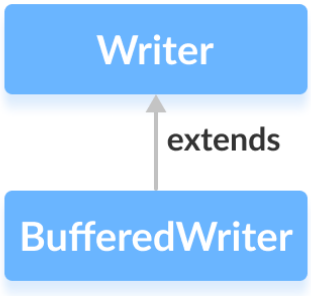
Other Methods of BufferedReader

Method	Description
`ready()`	checks if the file reader is ready to be read
`mark()`	mark the position in reader up to which data has been read
`reset()`	returns the control to the point in the reader where the mark was set

Java BufferedWriter Class

The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently.

It extends the abstract class Writer.



Working of BufferedWriter

The `BufferedWriter` maintains an internal **buffer of 8192 characters**.

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

Hence, the number of communication to the disk is reduced. This is why writing characters is faster using `BufferedWriter`.

Create a BufferedWriter

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

```
// Creates a FileWriter
FileWriter file = new FileWriter(String name);

// Creates a BufferedWriter
BufferedWriter buffer = new BufferedWriter(file);
```

In the above example, we have created a `BufferedWriter` named *buffer* with the `FileWriter` named *file*.

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

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

The buffer will help to write characters to the files more efficiently.

Methods of BufferedWriter

The BufferedWriter class provides implementations for different methods present in Writer.

write() Method

- **`write()`** - writes a single character to the internal buffer of the writer
- **`write(char[] array)`** - writes the characters from the specified array to the writer
- **`write(String data)`** - writes the specified string to the writer

Example: BufferedWriter to write data to a File

```
import java.io.FileWriter;
import java.io.BufferedWriter;

public class Main {

  public static void main(String args[]) {

    String data = "This is the data in the output file";

    try {
      // Creates a FileWriter
      FileWriter file = new FileWriter("output.txt");

      // Creates a BufferedWriter
      BufferedWriter output = new BufferedWriter(file);

      // Writes the string to the file
      output.write(data);

      // Closes the writer
      output.close();
    }
```

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

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

```
FileWriter file = new FileWriter("output.txt");
BufferedWriter output = new BufferedWriter(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.
```

flush() Method

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

For example, suppose we have an empty file named **output.txt**.

```
import java.io.FileWriter;
import java.io.BufferedWriter;

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

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

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

      // Creates a BufferedWriter
      BufferedWriter output = new BufferedWriter(file);

      // Writes data to the file
      output.write(data);

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

```
      output.close();
    }

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

Output

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

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

close() Method

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

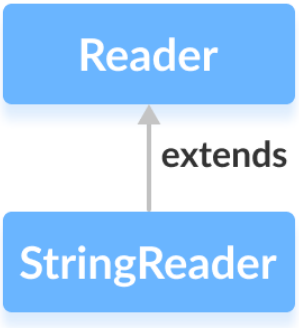
Other Methods of BufferedWriter

Method	Description
`newLine()`	inserts a new line to the writer
`append()`	inserts the specified character to the current writer

Java StringReader Class

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

It extends the abstract class Reader.



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*.

Methods of StringReader

The StringReader class provides implementations for different methods present in the Reader class.

read() Method

- **`read()`** - reads a single character from the string reader
- **`read(char[] array)`** - reads the characters from the reader and stores in the specified array
- **`read(char[] array, int start, int length)`** - reads the number of characters equal to *length* from the reader and stores in the specified array starting from the position *start*

Example: Java StringReader

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

Output

```
Data read from the string:
This is the text read from StringReader.
```

In the above example, we have created a string reader named *input*. The string reader is linked to the string *data*.

```
String data = "This is a text in the string.";
```

```
StringReader input = new StringReader(data);
```

To read data from the string, we have used the read() method.

Here, the method reads an array of characters from the reader and stores in the specified array.

skip() Method

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

```
import java.io.StringReader;

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

    String data = "This is the text read from StringReader";
    System.out.println("Original data: " + data);

    // Create a character array
    char[] array = new char[100];

    try {
      // Create a StringReader
      StringReader input = new StringReader(data);

      // Use the skip() method
      input.skip(5);

      //Use the read method
      input.read(array);
      System.out.println("Data after skipping 5 characters:");
      System.out.println(array);

      input.close();
    }

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

Output

```
Original data: This is the text read from the StringReader
Data after skipping 5 characters:
is the text read from the StringReader
```

In the above example, we have used the skip() method to skip 5 characters from the string reader. Hence, the characters 'T', 'h', 'i', 's' and ' ' are skipped from the original string reader.

close() Method

To close the string reader, we can use the `close()` method. Once the `close()` method is called, we cannot use the reader to read data from the string.

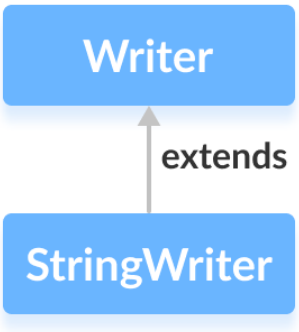
Other Methods of StringReader

Method	Description
`ready()`	checks if the string reader is ready to be read
`mark()`	marks the position in reader up to which data has been read
`reset()`	returns the control to the point in the reader where the mark was set

Java StringWriter Class

The StringWriter class of the java.io package can be used to write data (in characters) to the string buffer.

It extends the abstract class Writer.



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.

Methods of StringWriter

The StringWriter class provides implementations for different methods present in the Writer class.

write() Method

- **`write()`** - writes a single character to the string writer

- **`write(char[] array)`** - writes the characters from the specified array to the writer
- **`write(String data)`** - writes the specified string to the writer

Example: Java StringWriter

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

Output

```
Data in the StringWriter: This is the text in the string.
```

In the above example, we have created a string writer named *output*.

```
StringWriter output = new StringWriter();
```

We then use the `write()` method to write the string data to the string buffer.

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

Access Data from StringBuffer

- **`getBuffer()`** - returns the data present in the string buffer
- **`toString()`** - returns the data present in the string buffer as a string

```
For example, 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();
    }
  }
}
```

Output

```
StringBuffer: This is the original data
String: This is the original data
```

Here we have used the getBuffer() method to get the data present in the string buffer. And also the method toString() returns the data present in the string buffer as a string.

close() Method

To close the string writer, we can use the `close()` method.

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

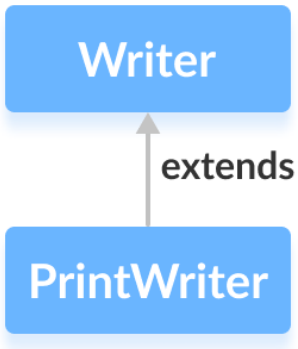
Other methods of StringWriter

Method	Description
`flush()`	forces to write all the data present in the writer to the string buffer
`append()`	inserts the specified character to the current writer

Java PrintWriter Class

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

It extends the abstract class Writer.



Working of PrintWriter

Unlike other writers, `PrintWriter` converts the primitive data (`int`, `float`, `char`, etc.) into the text format. It then writes that formatted data to the writer.

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

Note: The `PrintWriter` class also has a feature of auto flushing. This means it forces the writer to write all data to the destination if one of the `println()` or `printf()` methods is called.

Create a PrintWriter

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

1. Using other writers

```
// Creates a FileWriter
FileWriter file = new FileWriter("output.txt");

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

Here,

- we have created a print writer that will write data to the file represented by the FileWriter
- *autoFlush* is an optional parameter that specifies whether to perform auto flushing or not

2. Using other output streams

```
// Creates a FileOutputStream
FileOutputStream file = new FileOutputStream("output.txt");

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

Here,

- we have created a print writer that will write data to the file represented by the FileOutputStream
- the *autoFlush* is an optional parameter that specifies whether to perform auto flushing or not

3. Using filename

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

Here,

- we have created a print writer that will write data to the specified file
- the *autoFlush* is an optional boolean parameter that specifies whether to perform auto flushing or nor

Note: In all the above cases, the `PrintWriter` writes data to the file using some default character encoding. However, we can specify the character encoding (**UTF8** or **UTF16**) as well.

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

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

Methods of PrintWriter

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

print() Method

- **`print()`** - prints the specified data to the writer
- **`println()`** - prints the data to the writer along with a new line character at the end

For example,

```
import java.io.PrintWriter;

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

    String data = "This is a text inside the file.";

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

      output.print(data);
      output.close();
```

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

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

```
PrintWriter output = new PrintWriter("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 PrintWriter

```
import java.io.PrintWriter;

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

    try {
```

```
      PrintWriter output = new PrintWriter("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 writer named *output*. The print writer is linked with the file **output.txt**.

```
PrintWriter output = new PrintWriter("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 PrintWriter

Method	Description
`close()`	closes the print writer
`checkError()`	checks if there is an error in the writer and returns a boolean result
`append()`	appends the specified data to the writer