COMP3015
Data Communication and Networking
Socket Programming 1: Streams, Sockets, and Client Program

Streams

Streams are the underlying abstraction behind communications in Java. A stream represents an endpoint of a one-way communication channel. The streams provide communication channels between a program and a particular file or device. For example, `System.in` (an input stream) enables a program to input bytes from the default input device (i.e., keyboard). `System.out` (a print stream) enables a program to output data to the default output device (i.e., screen). `System.err` (a print stream) enables a program to output error message to the default output device. Each of these streams can be redirected to another output device.

`InputStream` and `OutputStream` are abstract classes that define methods for performing input and output respectively.

The useful methods defined in the `InputStream` class:

Method	Description
available()	Returns an **estimate** of the number of bytes that can be read.
read()	Returns the next byte of data from the stream in integer format.
read(byte[] b, int off, int len)	Reads up to **len** bytes starting from the stream into the byte array **b**. The first byte will be stored into **b[off]**.
close()	Closes the stream and releases system resources associated with the stream.

The useful methods defined in the `OutputStream` class:

Method	Description
write(int b)	Writes the specified byte **b** to the stream.
write(byte[] b, int off, int len)	Writes **len** bytes from the byte array **b** starting at offset off to the stream.
flush()	Forces any buffered bytes to be written out to the stream.
close()	Closes the stream and releases system resources associated with the stream.

Read and Write

Again, `System.in` is an `InputStream` object, which reads data from the default input device. Without any redirection, the default input device is keyboard – the keyboard input through the console. `System.out` is a `PrintStream` object, which writes data to the default output device. Without any redirection, the default output device is the console screen. `PrintStream` is a derived class of the `OutputStream` class.

In the following example program, `System.out` is upcasted from a `PrintStream` type to an `OutputStream` type. In addition, the statement inside the while-loop outputs the console inputs to the console.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Stream1 {
    public static void main(String[] args) throws IOException {

        InputStream in = System.in;
        OutputStream out = System.out;

        int count = 0;
        System.out.println("Input something and preses ENTER");

        while (true) {
            out.write(in.read());
            System.out.println(++count);
        }
    }
}
```

When we run the program above, the `read()` method blocks the program until the user presses `ENTER`.

For example, after running the code, the user types "HELLO WORLD" and press **ENTER**. Then, 13 bytes of the data are put in the input stream. Note that the `ENTER` key is also captured.

0	1	2	3	4	5	6	7	8	9	10	11	12	
H	E	L	L	O		W	O	R	L	D	\n	\r	Input stream

The `read()` method reads and returns a byte from the input stream. The first round, it reads 'H'. Then, the `write()` method writes the byte 'H' to the output stream. The while-loop repeats total 13 times to handle the string because the `read()` and `write()` methods handle only one byte each time.

The following example program uses `read(byte[] b, int off, int len)` method and `write(byte[] b, int off, int len)` method to handle input and output.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Stream2 {
    public static void main(String[] args) throws IOException {
        InputStream in = System.in;
        OutputStream out = System.out;

        byte[] buffer = new byte[10];
        int count = 0;

        System.out.println("Input something and press ENTER");
        while(true) {
            int len = in.read(buffer, 0, buffer.length);

            System.out.printf("\n# %d:\t", ++count);

            out.write(buffer, 0, len);
        }
    }
}
```

With the buffer (byte array) of size 10, the while-loop repeats total 2 times only to handle the same user inputted string.

Comparing with the previous version, the overhead of the new version is less, but additional memory space is required.

Overhead means the extra activities that are not directly related to the product creation, such as the expression validation in the `while` statement, the internal data handling activities of the stream, etc.

Advanced I/O Stream – DataInputStream & DataOutputStream

`DataInputStream` class and `DataOutputStream` class provides different methods for different primitive data types. With these methods, we can read data from the stream or write data to the stream using the specific data types.

The followings are commonly used methods provided by `DataInputStream`: *readByte(), readInt(), readFloat(), readDouble(), readLong(), readShort(), readBoolean(), readChar(),* and *read()*.

The followings are commonly used methods provided by `DataInputStream`: *write(), writeInt(), writeFloat(), writeDouble(), writeLong(), writeShort(), writeBoolean(), writeChar(),* and *writeBytes()*.

Consider the following code:

```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class Stream3 {

    public static void println(String str) {
        System.out.println(str);
    }

    public static void printf(String str, Object...o) {
        System.out.printf(str, o);
    }

    public static void main(String[] args) throws IOException {
        InputStream iStream = System.in;
        OutputStream oStream = (OutputStream) System.out;

        DataInputStream dataIn = new DataInputStream(iStream);
        DataOutputStream dataOut = new DataOutputStream(oStream);


        println("Input 8 digits and press ENTER");

        byte[] buffer = new byte[4];
        dataIn.read(buffer, 0, 4);

        int num1 = Integer.parseInt(new String(buffer));
        int num2 = dataIn.readInt();

        printf("\nThe retrieved inputs are %d and %d respectively.",
                num1, num2);

        println("\n\nOutput using the write() method of DataOutputStream: ");
        dataOut.write(buffer);

        println("\n\nOutput using the writeInt() method of DataOutputStream: ");

        dataOut.writeInt(num2);

        dataOut.flush();

        dataIn.close();
        dataOut.close();
        iStream.close();
        oStream.close();
    }
}
```

The program above asks us to input 8 digits (12341234) using the keyboard. It then uses the `read()` and `readInt()` methods to read the input. The first 4 digits will be read by the `read()` method and remaining 4 digits will be read by the `readInt()` method.

- The `read()` method stores the input in the byte array. Then, the input is converted in int format.

- The `readInt()` method directly returns an int value.

After running the program above, you should find that the returned value of the `readInt()` method is different from our expectation.

Text input	'1'	'2'	'3'	'4'
	↓	↓	↓	↓
Bytes (ASCII)	49	50	51	52
	↓	↓	↓	↓
Sum them up	49×256^3	$+ 50 \times 256^2$	$+ 51 \times 256^1$	$+ 52 \times 256^0$
	822083584	+ 3276800	+ 13056	+ 52
Actual value	825373492			
	(returned value of the readInt() method)			

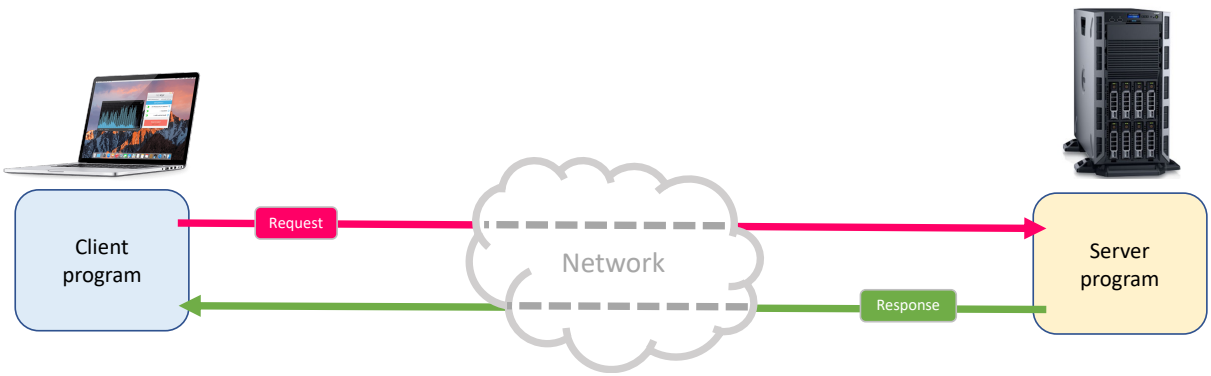
Note that the keyboard input is in text format (e.g., "1234") and finally be transferred in bytes (49, 50, 51, and 52). So, the `read()` and `readInt()` read the bytes but `readInt()` converts the bytes into an integer value. Finally, the `readInt()` changed "1234" to 825373492.

Similarly, the `write()` methods outputs bytes. `System.out` converts the bytes to characters and display on the console. The `writeInt()` method outputs an integer as 4 bytes, so 825373492 is changed back to "1234".

Sockets

A network connects computers, mobile phones, peripherals, network devices, etc. Devices connected to your network can communicate with one another.

In a client-server application, the two main components are server and client. The server runs the server program that passively waits for and responds to client programs. The client runs the client program that initiates the communication to the server program and requests services actively.



The socket is an abstraction through which an application may send and receive data. A socket is uniquely identified by the internet address (IP address in IP network), end-to-end protocol (e.g., TCP or UDP), and a port number. Java API (`java.net`) provides two types of sockets – Stream sockets (TCP) and Datagram sockets (UDP). In our labs and project will cover the Stream sockets only.

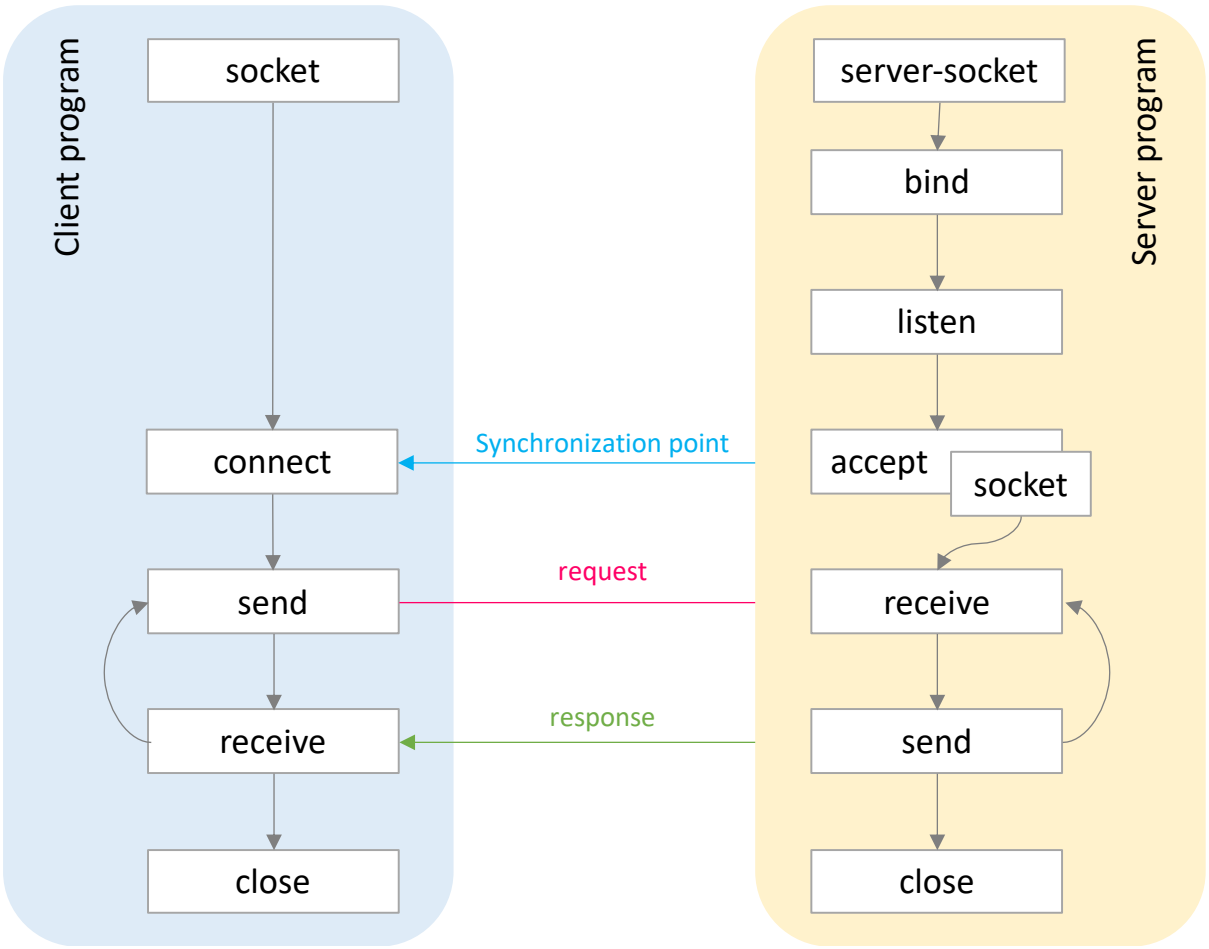
The socket classes, `Socket` class and `ServerSocket` class provided by `java.net`, are used to represent the connection between a client program and a server program.

Server

The server program has a server socket that is bound to a specific port number. This socket just waits and listening to the connection request sent by a client to make a connection. If the connection established successfully, a new socket object is used for communicating with the client. Then, the output stream and input stream of the newly created socket will be used for sending and receiving messages to and from the client respectively.

Client

The client program must know IP address (perhaps hostname if DNS is available) and port the number of the server. Then, the client program sends a connection request to the server and establish a connection. If the connection established successfully, the output and input streams of the socket will be used for communicating with the server program.



Client Program - Connection Establishment

Imagine that a server program transmits a character to its client every 1 second through a TCP connection. The IP address of the machine running the program is *zzz.yyy.xxx.www*, and the program is listening to the TCP port *vvv*. We need to write a client program to receive the characters transmitted from the server program.

Firstly, the client program needs to establish a connection to the server program. The **Socket** class will be used in the program.

```
Socket socket = new Socket("zzz.yyyy.xxx.www", vvv);
```

Receiving Data

When successfully establishing the connection, we can then obtain its input stream for receiving data.

```
InputStream in = socket.getInputStream();
```

Using read()

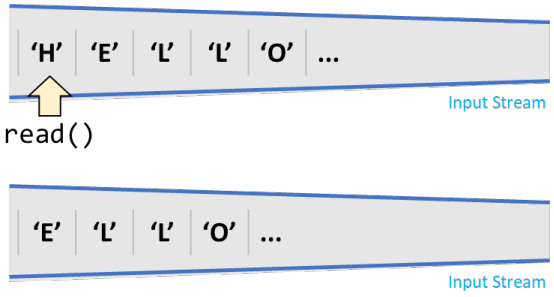
Consider the following code segment:

```
Scanner scanner = new Scanner(System.in);
print("Server IP: ");
String ip = scanner.nextLine();

print("TCP Port: ");
int port = scanner.nextInt();

Socket socket = new Socket(ip, port);
InputStream in = socket.getInputStream();

while(true) {
    char c = (char) in.read();
    println(c);
}
```

Check the following files:

- Client1.java
- Server1.java

In the code above, we use the `read()` method to retrieve a byte from the input stream of the socket. When the input stream is empty but we execute the `read()` method, the program will be blocked until the input stream becomes not empty. If the input stream is not empty, the `read()` method removes the first byte of the data from the input stream and returns that byte in the integer data type.



Using read(byte[], int, int)

If the client program uses the `read()` method to read bytes one by one, and the amount of data stored in the input stream is large, there will be a lot of overhead. Instead, we use the `read(byte[], int, int)` method to read multiple bytes at once.

Consider the following code segment:

```
Scanner scanner = new Scanner(System.in);
print("Server IP: ");
String ip = scanner.nextLine();

print("TCP Port: ");
int port = scanner.nextInt();

byte[] buffer = new byte[1024];

Socket socket = new Socket(ip, port);
InputStream in = socket.getInputStream();

while(true) {
    int len = in.read(buffer, 0, buffer.length);
    println(new String(buffer, 0, len));
}
```

Check the following files:

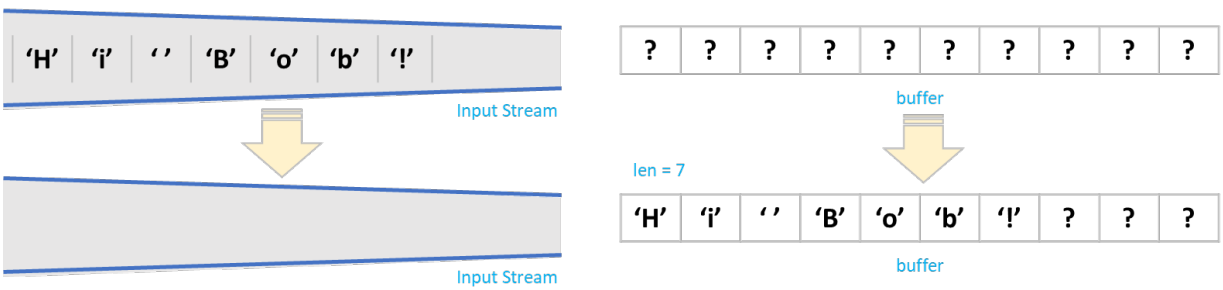
- Client2.java
- Server1.java

Because we do not know the length of the message sent by the server program, we use a large-enough byte array as a buffer for reading messages from the input stream. We assume that the length of each message is less than 1024.

In addition, we use `in.read(buffer, 0, buffer.length)` to read the input stream. Same as the `read()` method, the `read(byte[], int, int)` method blocks the client program if the input stream is empty. If the input stream is not empty, the bytes will be read from the input stream to the buffer.

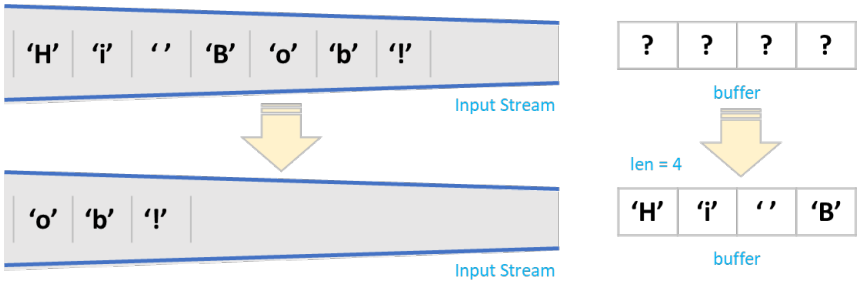
If the buffer is greater than the data stored in the input stream, all bytes will be read to the buffer and the `read` method returns the number of bytes read.

```
int len = read(buffer, 0, buffer.length)
```



If the buffer is smaller than the data, the part of the data will be read and fill the buffer until the buffer becomes full. The remaining part will be kept in the input stream. The `read` method returns the number of bytes read, same as the size of the buffer.

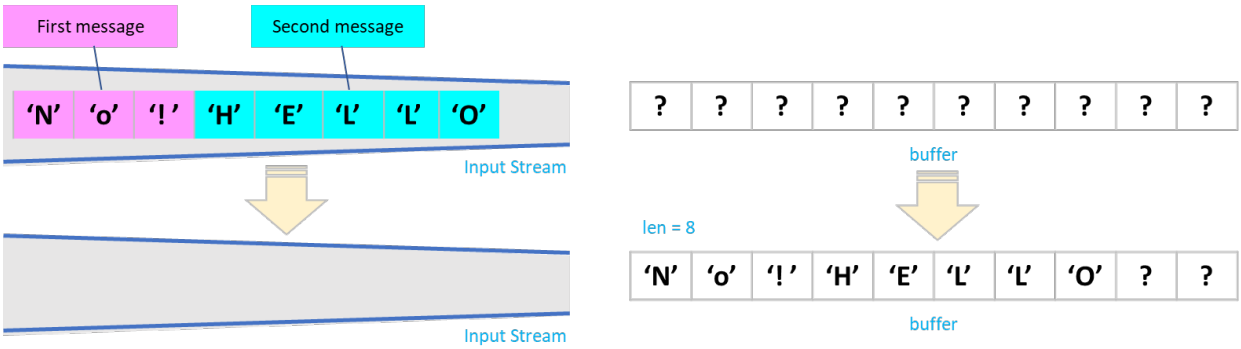
```
int len = read(buffer, 0, buffer.length)
```



Application Layer Protocol

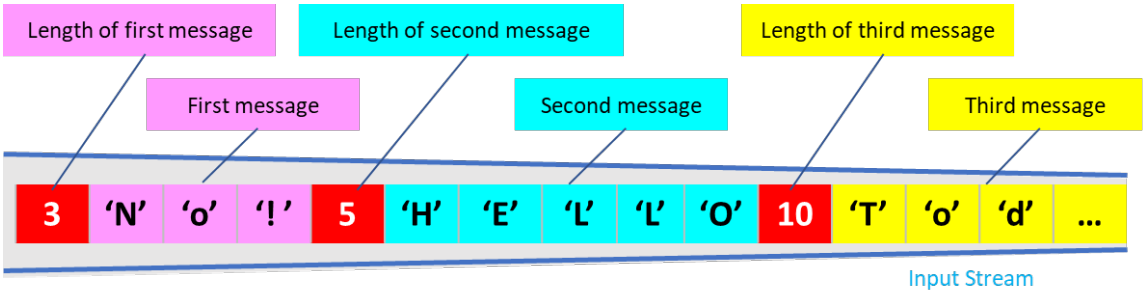
Imagine that the server program sends multiple dynamic-length messages to the client program. The client program needs to process the individual messages. Therefore, the client program must be able to recognize where the beginning and the end of the individual messages are.

```
int len = read(buffer, 0, buffer.length)
```



When the buffer is large enough, the client program will read multiple messages to the buffer. So, we cannot process the messages individually because they are mixed up.

To avoid mixing multiple messages together, we need to set an application-level protocol (the communication method) to the both server and client programs. For example, the server program every time sends the length of the message in integer data type before sending the message. When the client program receives the length of a message, it will know how many following bytes belong to that message.



Receiving Data through DataInputStream

If the server program sends an integer to the client program, the client program must have a way to receive the integer. So, we use the `DataInputStream`.

```
DataInputStream in = new DataInputStream(socket.getInputStream());
int len = in.readInt();
in.read(buffer, 0, len);
```

We use `in.readInt()` to read an integer that represents the length of the following message. In the `in.read(buffer, 0, len)` statement, we specify the number of the bytes for reading the data from the input stream. So, the client program now will not mix the messages together.

Sending Data through DataOutputStream

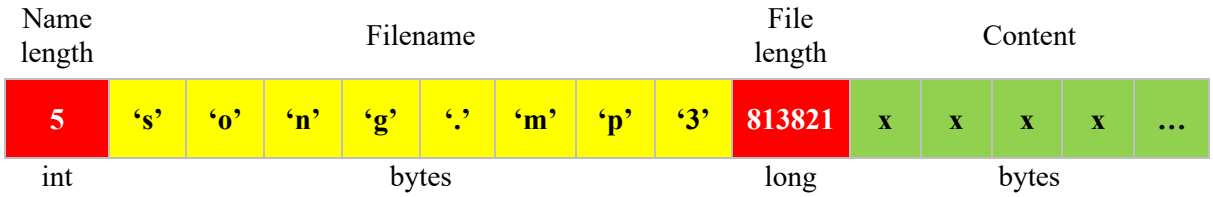
To send the data to the remote side, we use the `write()` method provided by the output stream, just like what we learned from the Streams section. As mentioned above, we usually define the application layer protocols. The server and client programs follow our defined protocols to transmit data.

Let's reuse the protocol defined in the previous example. We first send an integer representing the length of the data, then send the data. Imagine that a server program is running for handling file submissions.

Check the following files:

- Client3.java
- Server2.java

If the client program uploads a file named *song.mp3*, the data transmitted are as follows:



The client program runs for uploading a file by performing the following procedure:

1. Establish a connection to the server.

```
Socket socket = new Socket(serverIP, port);
```

2. Open an output stream to the socket.

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

3. Open an input stream to the file being uploaded.

```
FileInputStream in = new FileInputStream(file);
```

4. Send an integer that represents the length of the name of the file.

```
byte[] filename = file.getName().getBytes();
out.writeInt(filename.length);
```

5. Send the file name in bytes.

```
out.write(filename, 0, filename.length);
```

6. Send a long that represents the file size.

```
long size = file.length();
out.writeLong(size);
```

Check the following files:
- FileUploader.java
- FileServer.java

7. Send the file content in bytes.

```
while(size > 0) {
int len = in.read(buffer, 0, (int) Math.min(size, buffer.length));
    out.write(buffer, 0, len);
    size -= len;
    print(".");
}
out.flush();
```

8. Close the file stream and drop the connection.

```
in.close();
out.close();      /// the socket will be closed too.
```

Supplementary – File Access

File

In Java, the `File` class provides file and directory manipulation. It has many methods for retrieving information of files or directories.

Consider the following program:

```
import java.io.File;
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

public class FileInfo {

    public static void println(String str) { System.out.println(str); }

    public static void main(String[] args) throws IOException {
        println("Input a file/folder path and press ENTER");

        Scanner scanner = new Scanner(System.in);
        String filename = scanner.next();
        scanner.close();

        File file = new File(filename);

        Date date = new Date(file.lastModified());
        DateFormat dateFormat = new SimpleDateFormat("dd MMMM, YYYY hh:mm:ss");
        String dateTime = dateFormat.format(date);

        println("name : " + file.getName());
        println("size (bytes) : " + file.length());
        println("exists? : " + file.exists());
        println("dir? : " + file.isDirectory());
        println("modified: " + dateTime);
        println("canonical path : " + file.getCanonicalPath());
    }
}
```

The following is the sample output from a Mac:

```
Input a file/folder path and press ENTER
/var/log
name : log
size (bytes) : 1568
exists? : true
dir? : true
modified: 03 September, 2022 01:16:23
canonical path : /private/var/log
```

File I/O

For the file Input/output, we can use the `FileInputStream` class and `FileOutputStream` class. `FileInputStream` is a subclass of `InputStream`, and `FileOutputStream` is a subclass of `OutputStream`.

Reading from File

The following example program uses `FileInputStream` to read and print the content of a text file to the console. It works for the text file of any size. If the length of the file is larger than the buffer, the while-loop repeats until the last byte of the file is read. The variable `size` is used as a counter to determine when the loop should stop.

Note that we should not always use the file size to create the buffer, because the size of a file can be up to 16 TB (terabytes) in Windows NTFS, 4GB (gigabytes) in FAT32, and 8EB (exabytes) in MacOS.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Scanner;

public class TextFileDisplayer {

    public static void println(String str) {
        System.out.println(str);
    }

    public static void main(String[] args) throws IOException {

        println("Input a text file path and press ENTER");

        Scanner scanner = new Scanner(System.in);
        String filename = scanner.next();

        byte[] buffer = new byte[1024];

        File file = new File(filename);
        long size = file.length();

        FileInputStream in = new FileInputStream(file);

        while (size > 0) {
            int len = in.read(buffer, 0, buffer.length);
            size -= len;

            println(new String(buffer, 0, len));
        }

        in.close();
        scanner.close();
    }
}
```

Writing to File

When an output stream is opened and the file object is associated with a non-existent file, a new file will be created automatically. When an output stream is opened on an existing file, the file content will be overwritten. Change the second parameter of `FileOutputStream`'s constructor call to `TRUE` if you want to append the new content to the file.

```
FileOutputStream out = new FileOutputStream(file, true);
```

Consider the following example program that uses `FileOutputStream` to write the text to a file:

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

public class TextSaver {
    public static void println(String str) { System.out.println(str); }

    public static void main(String[] args) throws IOException {
        println("Enter the file name and Press ENTER");

        InputStream in = System.in;
        byte[] buffer = new byte[1024];
        int len = in.read(buffer, 0, buffer.length);

        String filename = new String(buffer, 0, len);
        filename = filename.replace("\n", "").replace("\r", "");

        File file = new File(filename);

        println(
            "Please enter the content. (enter @@quit in a new line to quit)");

        FileOutputStream out = new FileOutputStream(file, false);

        while (true) {
            len = in.read(buffer, 0, buffer.length);

            String str = new String(buffer, 0, len);

            if (str.contains("@@quit")) break;

            out.write(buffer, 0, len);
        }

        out.flush();
        out.close();
        in.close();
        println("bye");
    }
}
```