

به نام خدا



برنامه‌سازی پیشرفته

دانشگاه شهید بهشتی . دانشکده مهندسی کامپیوتر

دکتر مجتبی وحیدی اصل

برنامه‌نویسی ورودی/خروجی (I/O) بخش دوم

پارسا حمزه ئى

فهرست مطالب

IO . 1

- انواع ورودی خروجی برنامه و ذخیره سازی آنها
- جریان های ورودی خروجی (**java IO : streams**)

2. کار با کلاس **RandomAccessFile**

3. ایجاد و کار با کلاس های **input/output Stream**

4. آشنایی با **SequenceInputStream**

5. آشنایی با کلاس های **Reader / Writer**

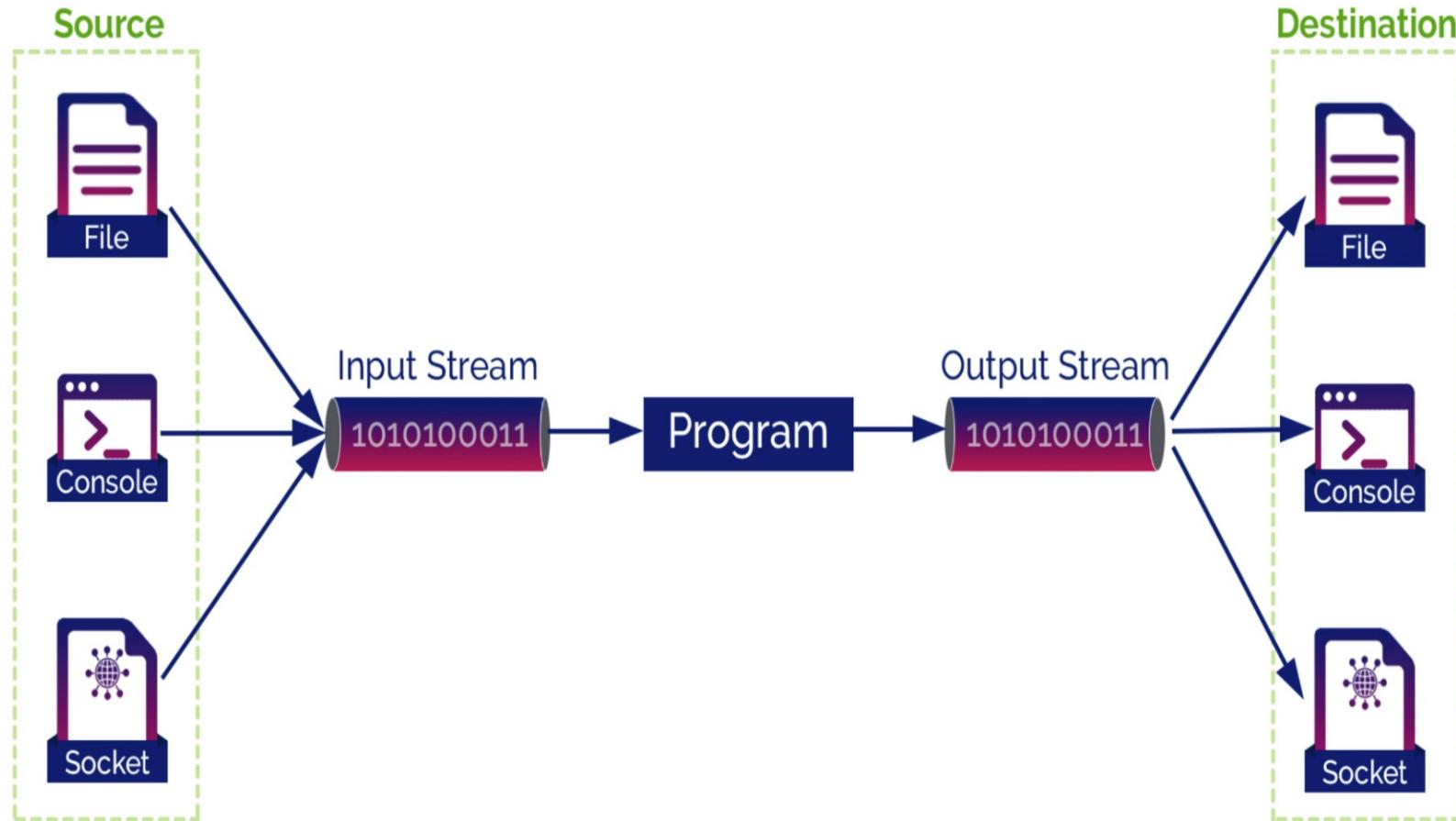
6. مدیریت خط

فایل و جریان

- برای ذخیره کردن دیتا، آنها را در فایل ذخیره میکنیم که معمولاً در حافظه جانبی ذخیره میشود.
در برنامه میتوان عمل اصلی CRUD را روی فایل ها انجام داد:
 - ایجاد کردن و ساخت فایل :**Create**
 - خواندن دیتا از فایل :**Read**
 - تغییر دیتای فایل :**Update**
 - حذف کردن فایل :**Delete**
- انواع ذخیره سازی فایل :
 - متنی (**text files**) : از کاراکتر ها تشکیل شده اند، مانند فایل های txt و html
 - باینری (**binary files**) : مانند فایل های zip,pdf,exe

چیست Stream

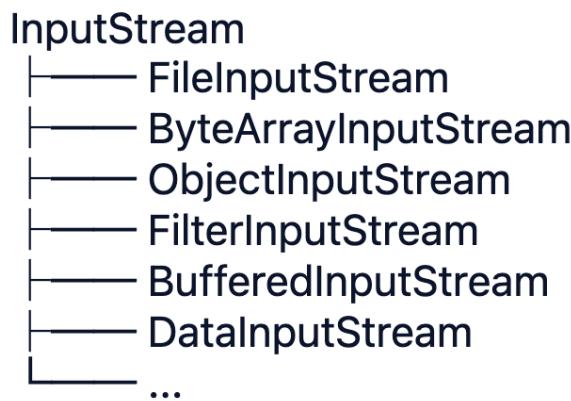
یک جریان از داده ها میباشد که مثل خط لوله داده میتواند از طریق آن جریان یابد، درواقع یک دنباله پیوسته از داده هاست. در برنامه میتواند ورودی یا خروجی باشد، همچنین جریان داده های متنی، یا داده های باینری باشد. به عنوان مثال در تصویر زیر، جریان داده سمت چپ ورودی برنامه ما، و جریان داده سمت راست خروجی برنامه است. و همانطور که مشخص است هم میتواند به فایل متصل شود یا به پایپ یا به شبکه متصل شود.



قبل تر با برخی کلاس ها و زیر کلاس های IO در جاوا آشنا شدید، در این قسمت با تفاوت و کاربرد آنها بیشتر آشنا می شوید:

کتابخانه `java.io` به ما کلاس های متفاوتی برای کار با فایل و جریان ها می دهد.

به عنوان مثال برای جریان های ورودی برنامه کلاس `InputStream` را می دهد که زیر کلاس های آن به صورت زیر است:



و برای جریان های خروجی برنامه کلاس **OutputStream** را در اختیار ما می‌گذارد که زیر کلاس های آن به صورت زیر است:

OutputStream

- |---- FileOutputStream
- |---- ByteArrayOutputStream
- |---- ObjectOutputStream
- |---- FilterOutputStream
- |---- BufferedOutputStream
- |---- DataOutputStream
- |---- PrintStream
- |---- ...

همچنین برای کار با فایل، به طور دقیق تر برای خواندن از فایل کلاس **Reader** را در اختیار ما قرار می‌دهد که زیر کلاس های آن به صورت زیر است:

Reader

- └── InputStreamReader
- └── FileReader
- └── BufferedReader
- └── StringReader
- ...
...

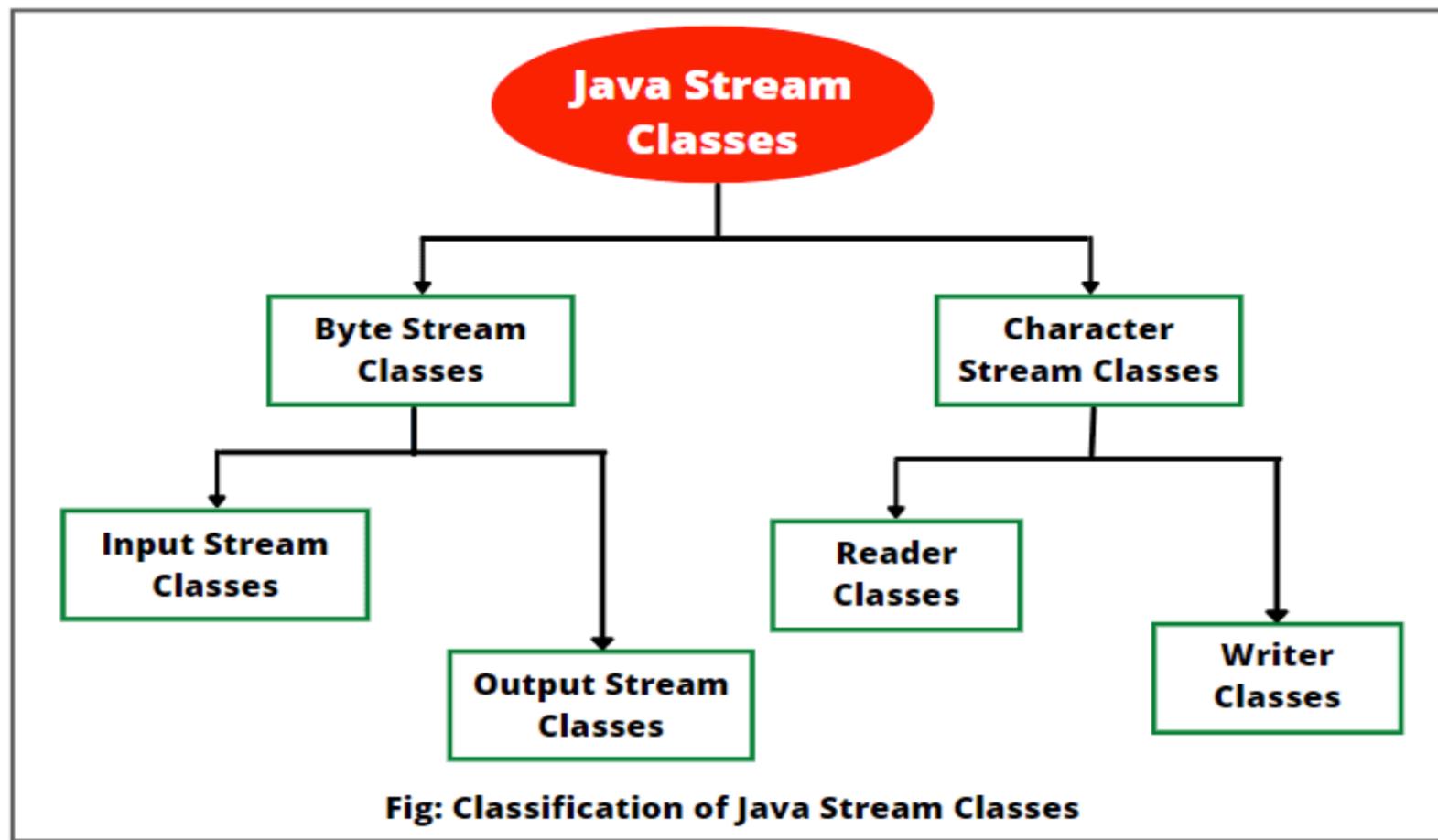
همچنین برای نوشتن در فایل کلاس **Writer** را در اختیار ما می‌گذارد که زیر کلاس های آن به صورت زیر می‌باشد:

Writer

- └── OutputStreamWriter
- └── FileWriter
- └── BufferedWriter
- └── PrintWriter
- └── StringWriter
- ...
...

نکته: کلاس های Reader و Writer (برای کار با فایل) و همچنین InputStream و OutputStream (برای کار با stream ها) همگی کلاس های انتزاعی هستند (abstract class)

تصویر زیر توصیف کلی از نحوه کلاس بندی برای خواندن و نوشتن جریان در جاوا ارائه میدهد:



RandomAccessFile

این کلاس به ما این امکان را می‌دهد که درون فایل حرکت کرده و آن را پیمایش کنیم و بتوانیم از آن بخوانیم یا در آن بنویسیم.

همچنین با استفاده از **FileOutputStream** و **FileInputStream** میتوان بخش‌هایی از فایل را با بخش‌هایی جدید جایگزین کرد.

نکته: در جریان (stream) برخلاف آرایه، اندیس برای خواندن و نوشتمن معنی ندارد! و معمولاً نمیتوان روی جریان به عقب و جلو حرکت کرد در حالی که در آرایه RandomAccessFile امکان پذیر است.

اگرچه برخی از پیاده سازی‌های زیرکلاس stream مانند **pushbackInputStream** به ما این امکان را می‌دهد داده‌هارا در جریان push back کنیم تا بعداً دوباره بخوانیم، اما بخش محدودی از داده‌ها قابل push back هستند و نمیتوان پیمایش کاملی روی تمام داده‌ها داشت.

نحوه ایجاد یک : RandomAccessFile

```
public class Test {  
    public static void main(String[] args) {  
        RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt","rw");  
    }  
}
```

پارامتر دوم (rw) مُد باز کردن فایل را مشخص میکند، و "rw" به معنای مُد خواندن/نوشتن است.

برای خواندن و نوشتن در محل خاص از یک `RandomAccessFile`, باید ابتدا اشاره گر فایل را در محل مورد نظر قرار دهیم. این کار را با تابع `seek()` انجام می‌دهیم، همچنین با تابع `getFilePointer()` از محل فعلی اشاره گر آگاه می‌شویم:

```
public class Test {
    public static void main(String[] args) {
        RandomAccessFile file = new RandomAccessFile("c:\\\\data\\\\file.txt","rw");
        long pointer = file.getFilePointer();
        file.close();
    }
}
```

سوال : اگر بعد از آخرین دستور، تابع `getFilePointer()` را فراخوانی کنیم، خروجی چه خواهد بود؟

خواندن از : RandomAccessFile

برای خواندن از یک RandomAccessFile میتوان یکی از متدهای `read()` را انتخاب کرد، به عنوان مثال :

```
public class Test {  
    public static void main(String[] args) {  
        RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt","rw");  
        int aByte = file.read();  
        file.close();  
    }  
}
```

متدهای `read()` بایتی را که در محل اشاره گر فایل در شیئ ایجاد شده اس **RandomAccessFile** قرار دارد، می خواند. همچنین به یاد داشته باشید که این متدهای از عمل خواندن، اشاره گر فایل را درون فایل مربوطه یک بایت به جلو می برد. یعنی نیاز نیست به صورت دستی اشاره گر را به جلو ببرید.

نوشتن در فایل : RandomAccessFile

برای نوشتن در این نوع فایل ها هم متدهای مختلفی وجود دارد که میتوان از آنها استفاده کرد، به عنوان مثال میتوان از متدهای `write()` استفاده کرد که آن هم مانند متدهای `read()` پس از اجرا، اشاره گر را به یک بایت جلوتر حرکت می‌دهد:

```
public class Test {  
    public static void main(String[] args) {  
        RandomAccessFile file = new RandomAccessFile("c:\\\\data\\\\file.txt","rw");  
        file.write("Hello World".getBytes());  
        file.close();  
    }  
}
```

کلاس InputStream

این کلاس یک کلاس پایه و والد (همچنین انتزاعی) برای تمام جریان های ورودی در جاوا می باشد، اغلب برای خواندن داده ها از یک منبع داده ای از این کلاس استفاده می شود.
برای خواندن داده از فرزندان این کلاس، از متدهای `read()` استفاده می شود، این متدهای `int` برمی گردانند که حاوی مقدار باشد خوانده شده است.

نکته : اگر داده ای برای خواندن باقی نماند باشد، مقدار `-1` را برمی گرداند.

مثال :

```
public class Test {
    public static void main(String[] args) {
        InputStream input = new FileInputStream("c:\\\\data\\\\input-file.txt");
        int data = input.read();
        while( data != -1 )
        {
            data = input.read();
        }
    }
}
```

همانطور که دیدید، **java InputStream** یک از فرزندان کلاس انتزاعی **FileInputStream** میباشد. جاوا برای خواندن داده های بایتی از فایل ها از این کلاس استفاده میکند.

به مثال زیر توجه کنید:

```
public class Test {  
    public static void main(String[] args) {  
        InputStream inputstream = new FileInputStream("c:\\data\\input-file.txt");  
        int data = inputstream.read();  
        while (data != -1 ){  
            //do something with data ....  
            data = inputstream.read();  
        }  
        inputstream.close();  
    }  
}
```

مثال بالا یک شیء جدید از **FileInputStream** ایجاد می‌کند و متدهای `read()` یک مقدار `int` برمی‌گرداند که حاوی مقدار `Byte` خوانده شده است که به صورت زیر میتوان آن را به `char` تبدیل کرد:

```
char aChar = (char) data;
```

اگر به پایان فایل رسیده باشیم و دیتایی برای خواندن باقی نمانده باشد، متدهای `read()` مقدار ۱- برمی‌گرداند.

نکته: زیرکلاس‌های `InputStream` ممکن است دارای متدهای `read()` مختلفی باشند. به عنوان مثال این متدهای در کلاس **DataInputStream** به شما این امکان را می‌دهد در هر لحظه یک مقدار `int`, `long`, `float` و `double` با متدهای `readBoolean()`, `readDouble()` و ... بخوانید.

همچنین در کلاس **InputStream** دو متد (`read()`) دیگر هم وجود دارد که می‌تواند داده‌ها را از منبع داده‌ای `InputStream` به درون آرایه از نوع بایت بريزد، خواندن یک آرایه از بایت‌ها در هر لحظه به مراتب سریع‌تر از خواندن بایت‌داده‌ها می‌باشد.
این متد‌ها به صورت زیر هستند:

```
int read(byte[]);
int read(byte[], int offset, int length)
```

متد (`int read(byte[], int offset, int length)`) مانند قبلی یک آرایه از بایت‌ها را می‌خواند که از بایت `offset` شروع می‌شود و به طول `length` در آرایه قرار می‌دهد. متد (`int read(byte[])`) تلاش می‌کند تا حداقل امکان بایت‌های بیشتری را بخواند و در آرایه پاس داده شده ذخیره کند.
این متد یک مقدار `int` بر می‌گرداند که می‌گویند چند Byte خوانده شده است، در موقعي که تعداد Byte خوانده شده از `InputStream` کمتر از اندازه واقعی آرایه باشد، مقادیر سایر خانه‌ها تغییر نخواهد کرد و همان مقادیر قبلی باقی می‌مانند.

نکته : هر دو متد زمانی که به انتهای جریان میرسیم، مقدار 1- بر می‌گردانند.

مثال

```
public class Test {  
    public static void main(String[] args) {  
        InputStream input_stream = new FileInputStream("c:\\\\data\\\\input-text.txt");  
        byte[] data = new byte[1024];  
        int bytesRead = input_stream.read(data);  
        while(bytesRead != -1) {  
            for(int i=0;i<bytesRead;i++){  
                System.out.print(" "+data[i]);  
            }  
            bytesRead = input_stream.read(data);  
        }  
        input_stream.close();  
    }  
}
```

کلاس OutputStream

این کلاس هم یک کلاس پایه و والد (همچنین انتزاعی) برای تمام جریان های خروجی در جاوا می باشد. برای نوشتنداده ها در یک مقصد داده ای اغلب از این کلاس استفاده می شود.

برای نوشتنداده ها در اشیای ساخته شده از فرزندان این کلاس از متدهای [Write](#) استفاده می شود.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        OutputStream output = new FileOutputStream("c:\\data\\output-file.txt");  
        output.write("Hello World".getBytes());  
        output.close();  
    }  
}
```

ادهای و مقاصد OutputStream

این کلاس هم یک کلاس پایه و والد (همچنین انتزاعی) برای تمام جریان های خروجی در جاوا می باشد. برای نوشتن داده ها در یک مقصد داده ای اغلب از این کلاس استفاده می شود.

یک OutputStream به مقصد داده ای نظیر فایل، پایپ یا اتصال شبکه ای وصل می شود.
مقصد داده ای یک شیء از کلاس OutputStream است که داده نوشته شده در شیء سرانجام به آنجا منتقل می شود.

متدهای Write(Byte)

این متدهای برای نوشتن یک بایت در OutputStream می باشد، همچنین زیر کلاس های OutputStream دارای متدهای Write() دیگر هم هستند به عنوان مثال به شما این امکان را می دهد داده هایی از نوع int, float, Double, Boolean, long و غیره را در قالب متدهای DataOutputStream در یک مقصد داده ای بنویسید.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        OutputStream output = new FileOutputStream("c:\\data\\output-file.txt");  
        while(hasMoreData()){  
            int data = getMoreData();  
            output.write(data);  
        }  
        output.close();  
    }  
}
```

در ابتدا یک شیء از `FileOutputStream` ساخته میشود تا داده ها درون آن نوشته شوند، سپس در یک حلقه `while` تا زمانی که داده ای برای نوشتمن وجود دارد، داده های در جریان نوشته میشوند.
در واقع در حلقه `while` داده از جای دیگر گرفته میشود و درون `output` ریخته میشود.
سر انجام با بستن جریان، محتوای آن به مقصد داده ای که در این مثال یک فایل است، منتقل خواهد شد.

:Write(Byte[])

کلاس OutputStream دارای دو متد Write دیگر هم هست که می‌توانند آرایه‌ای از نوع بایت (یا بخشی از آرایه را) به درون شیء OutputStream ببریزد. متد‌ها به صورت زیر هستند:

```
write(byte[] bytes);  
write(byte[] bytes , int offset , int length)
```

متد اول، تمام بایت‌ها را به درون outputStream می‌ریزد.
متد دوم تعداد bytes را از اندیس offset به درون outputStream می‌ریزد.

:flush()

این متد در کلاس OutputStream، تمام داده‌های قرار داده شده در شیء از نوع OutputStream را به مقصد داده‌ای منتقل می‌کند. به عنوان مثال اگر باشد تا قبل از فراخوانی متد flush() داده‌های نوشته شده در آن به دیسک منتقل نمی‌شوند، بلکه در بخشی از FileOutputStream یک flush() اطمینان حاصل می‌کنیم که همه داده‌های بافر شده در مقصد نوشته خواهند شد.

مثال

میخواهیم یک برنامه بنویسیم که داده هارا از یک فایل بخواند و همزمان در فایلی دیگر بنویسد:
با استفاده از کلاس `FileInputStream` میتوانیم از هر فایلی داده هارا بخوانیم.

این فایل میتواند حتی تصویر یا ویدیو باشد!

در این مثال داده هارا از یک فایل به نام `C.java` میخوانیم و در فایل دیگری به نام `M.java` مینویسیم:

```
public class Test {  
    public static void main(String[] args) {  
        FileInputStream fin = new FileInputStream("C.java");  
        FileOutputStream fout = new FileOutputStream("M.java");  
        int i = 0;  
        while((i=fin.read())!=-1){  
            fout.write((byte)i);  
        }  
        fin.close();  
    }  
}
```

SequenceInputStream

هرگاه بخواهیم محتویات چند فایل را پشت سر هم بخوانیم، میتوانیم از کلاس SequenceInputStream استفاده کنیم

به مثال زیر دقت کنید :

```
public class Test {  
    public static void main(String[] args) {  
        FileInputStream fin1 = new FileInputStream("f1.txt");  
        FileInputStream fin2 = new FileInputStream("f2.txt");  
  
        SequenceInputStream sis = new SequenceInputStream(fin1,fin2);  
        int i;  
        while((i=sis.read())!=-1){  
            System.out.println((char)i);  
        }  
        sis.close();  
        fin1.close();  
        fin2.close();  
    }  
}
```

همچنین برای خواندن از دو فایل و نوشتن در یک فایل میتوان مانند زیر عمل کرد:

```
public class Test {  
    public static void main(String[] args) {  
        FileInputStream fin1 = new FileInputStream("f1.txt");  
        FileInputStream fin2 = new FileInputStream("f2.txt");  
  
        FileOutputStream fout = new FileOutputStream("M.java");  
  
        SequenceInputStream sis = new SequenceInputStream(fin1,fin2);  
        int i;  
        while((i=sis.read())!=-1){  
            fout.write(i);  
        }  
        sis.close();  
        fin1.close();  
        fin2.close();  
        fout.close();  
    }  
}
```

کلاس های Reader / Writer

این دو کلاس مشابه کلاس های `InputStream` و `OutputStream` هستند.
همانطور که پیش تر آموختید تفاوت آن است که Reader و Writer مبتنی بر کاراکتر می باشند. بنابراین این دو کلاس با هدف خواندن و نوشتن متن به کار می روند.

کلاس Reader

این کلاس که قبلاً تر با آن آشنا شده اید، یک کلاس انتزاعی برای تمامی کلاسهای Reader در API جاوا می باشد.
زیر کلاس های Reader , `StringReader` , `InputStreamReader` , `BufferedReader` , `PushbackReader` همگی از کلاس Reader ارثبری می کنند.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        Reader reader = new FileReader("c:\\data\\myfile.txt");  
        int data = reader.read();  
        while(data != -1){  
            char dataChar = (char)data;  
            data = reader.read();  
        }  
    }  
}
```

توجه داشته باشید `InputStream` در هر لحظه یک بایت برمی‌گرداند که مقدارش بین ۰ و ۲۵۵ می‌باشد (منفی یک برای زمانی که به انتهای فایل می‌رسد) اما کلاس `Reader` یک کاراکتر را در هر لحظه برمی‌گرداند که مقدار آن بین ۰ و ۶۵۵۳۵ می‌باشد (منفی یک برای انتهای فایل) یعنی در هر لحظه دو بایت می‌خواند.

کلاس `FileReader`

این کلاس برای خواندن داده‌های کاراکتری از یک فایل استفاده می‌شود. `Read()` در این کلاس یک مقدار `int` برمی‌گرداند که حاوی مقدار کاراکتری کاراکتری خوانده شده است. اگر این متد مقدار ۱- را برگرداند یعنی به انتها فایل رسیده ایم.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        FileReader fr = new FileReader("abc.txt");  
        int i;  
        while((i=fr.read())!=-1){  
            System.out.println((char)i);  
            fr.close();  
        }  
    }  
}
```

Writer کلاس

این کلاس یک کلاس انتزاعی برای تمامی کلاسهای Writer در API جاوا می‌باشد.
زیر کلاس های PrintWriter و BufferedWriter ارثبری می‌کنند.

مثال:

توجه داشته باشید inputStream در هر لحظه یک بایت برمی‌گرداند که مقدارش بین 0 و 255 می‌باشد (منفی یک برای زمانی که به انتهای فایل می‌رسد) اما کلاس Reader یک کاراکتر را در هر لحظه برمی‌گرداند که مقدار آن بین 0 و 65535 می‌باشد (منفی یک برای انتهای فایل) یعنی در هر لحظه دو بایت می‌خواند.

FileReader کلاس

این کلاس برای خواندن داده‌های کاراکتری از یک فایل استفاده می‌شود.
متده Read() در این کلاس یک مقدار int برمی‌گرداند که حاوی مقدار کاراکتری کاراکتر خوانده شده است.
اگر این متده مقدار 1- را برگرداند یعنی به انتها فایل رسیده ایم.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        Writer writer = new FileWriter("c:\\data\\file-output.txt");  
        writer.write("Hello World Writer");  
        write.close();  
    }  
}
```

کلاس **FileWriter**

این کلاس برای نوشتن داده های کاراکتری درون یک فایل استفاده می شود.
همانطور که قبل تر تاکید کردیم، شرکت **میکرو سیستم sun** پیشنهاد کرده در مواردی که فایل های مورد استفاده متنی هستند، از این کلاس و FileReader برای خواندن و نوشتن استفاده کنیم.

نکته : دقت کنید اگر فایل موجود نباشد، این فایل ایجاد می شود و اگر هم موجود باشد محتوای آن پاک می شود.

مثال:

```
public class Test {  
    public static void main(String[] args) {  
        try{  
            FileWriter fw = new FileWriter("abc.txt");  
            fw.write("my name is sashin");  
            fw.flush();  
            fw.close();  
        }catch(Exception e){  
            System.out.println(e);  
        }  
        System.out.println("success");  
    }  
}
```

برای اینکه محتوا پاک نشود و صرفا دیتا به فایل اضافه شود، می‌توان یک ارگومان true به آن پاس داد، به صورت زیر:

```
FileWriter fw = new FileWriter("abc.txt",true);
```

همچنین برای اضافه کردن (append) به یک فایل می‌توان مانند زیر عمل کرد:

```
public class Test {
    public static void main(String[] args) {
        try {
            String filename= "MyFile.txt";
            FileWriter fw = new FileWriter(filename,true); //the true will append the
            new data
            fw.write("add a line\n");//appends the string to the file
            fw.close();
        }
        catch(IOException ioe)
        {
            System.err.println("IOException: " + ioe.getMessage());
        }
    }
}
```

مدیریت خطا ۱۰

بعد از اتمام کار با `writer` ها باید جریان ها را ببندیم، این کار با فراخوانی متد `close()` انجام میشود. بسیاری از کلاس های `IO` متد `close()` را دارا میباشند. این عمل بسیار مهم است زیرا امکان بروز استثنای `IOException` در حین کار با `writer()` وجود دارد، زیرا سیستم عامل فایل را به برنامه تخصیص میدهد و اگر آن را نبندیم، فایل آزاد نمیشود.

نکته: تعداد فایل های قابل باز کردن در برنامه محدود است و آزاد نکردن فایل باعث بروز مشکل میشود، همچنین ممکن است امکان باز کردن فایل در برنامه دیگر نباشد!

ممکن است خطا های دیگری همچون نقض مجوز دسترسی به فایل یا پیدا نشدن فایل (`FileNotFoundException`) بروز دهد! به همین دلیل بهتر است از بلاک های `try / catch` استفاده کنید.

همچنین بهتر است متد `close()` را در بلاک `finally` قرار دهیم، به عنوان مثال:

```
public class Test {
    public static void main(String[] args) {
        try{
            output = new FileOutputStream("c:\\data\\output-text.txt");
            while(hasMoreData()) {
                int data = getMoreData();
                output.write(data);
            }
            finally { if(output != null) { output.close(); } }
        }
    }
}
```

خودمون رو بسنجیم

این بخش برای این طراحی شده که در پایان مطالعه این اسلاید، بتونی خودت رو محک بزنی و ببینی آیا مفاهیم رو به خوبی یاد گرفتی یا نه. سوالات زیر را مرور کن و سعی کن بدون نگاه کردن به متن درس، به اون ها پاسخ بدی.

- انواع ذخیره سازی فایل ها چیست؟ کلاس های خواندن و نوشتن هر کدام را نام ببرید.
- چرا یک تابع (`read()`) مقدار عددی برمیگرداند و نه یک کاراکتر؟

[مطالعه بیشتر](#):

- انواع استاندارد ها برای کاراکتر ها و انواع کدگذاری ها
- کلاس های **closeable**
- امکانات `nio` و `nio.2`

پایان

در صورت هرگونه سوال یا پیشنهاد میتوانید با من
در ارتباط باشید:)

gmail: parsahamzeiii@gmail.com
telegram: @ParsaHami