

This site uses cookies to improve the user experience.

Java IO

1. [Java IO Tutorial](#)
2. [Java IO Overview](#)
3. [Java IO: Files](#)
4. [Java IO: Pipes](#)
5. [Java IO: Networking](#)
6. [Java IO: Byte & Char Arrays](#)
7. [Java IO: System.in, System.out, and System.error](#)
8. [Java IO: Streams](#)
9. [Java IO: Input Parsing](#)
10. [Java IO: Readers and Writers](#)
11. [Java IO: Concurrent IO](#)
12. [Java IO: Exception Handling](#)
13. [Java IO: InputStream](#)
14. [Java IO: OutputStream](#)
15. [Java IO: FileInputStream](#)
16. [Java IO: FileOutputStream](#)
17. [Java IO: RandomAccessFile](#)
18. [Java IO: File](#)
19. [Java IO: PipedInputStream](#)
20. [Java IO: PipedOutputStream](#)
21. [Java IO: ByteArrayInputStream](#)
22. [Java IO: ByteArrayOutputStream](#)
23. [Java IO: FilterInputStream](#)
24. [Java IO: FilterOutputStream](#)
25. [Java IO: BufferedInputStream](#)
26. [Java IO: BufferedOutputStream](#)
27. [Java IO: PushbackInputStream](#)
28. [Java IO: SequenceInputStream](#)
29. [Java IO: DataInputStream](#)
30. [Java IO: DataOutputStream](#)
31. [Java IO: PrintStream](#)

Java IO: SequenceInputStream

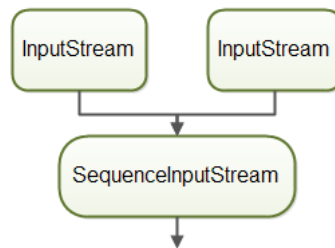
- [SequenceInputStream Example](#)
- [Combining More Than Two InputStreams](#)
- [Closing a SequenceInputStream](#)



Jakob Jenkov
Last update: 2015-08-31



The Java `SequenceInputStream` combines two or more other `InputStream`'s into one. First the `SequenceInputStream` will read all bytes from the first `InputStream`, then all bytes from the second `InputStream`. That is the reason it is called a *SequenceInputStream*, since the `InputStream` instances are read in sequence.



SequenceInputStream Example

It is time to see an example of how to use a `SequenceInputStream`.

Before you can use the `SequenceInputStream` you must import it in your Java class. Here is how to import `SequenceInputStream`:

```
import java.io.SequenceInputStream;
```

This import statement should be at the top of your Java class, right under the package declaration.

Now let us see how to use the `SequenceInputStream`. Here is a simple Java `SequenceInputStream` example:

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");
InputStream input2 = new FileInputStream("c:\\data\\file2.txt");

SequenceInputStream sequenceInputStream =
    new SequenceInputStream(input1, input2);

int data = sequenceInputStream.read();
while(data != -1){
    System.out.println(data);
    data = sequenceInputStream.read();
}
```

This Java code example first creates two `FileInputStream` instances. The `FileInputStream` extends the `InputStream` class, so they can be used with the `SequenceInputStream`.

Second, this example creates a `SequenceInputStream`. The `SequenceInputStream` is given the two `FileInputStream` instances as constructor parameters. This is how you tell the `SequenceInputStream` to combine two `InputStream` instances.

The two `InputStream` instances combined with the `SequenceInputStream` can now be used as if it was a coherent stream. When there is no more data to read from the second `InputStream`, the `SequenceInputStream` `read()` method will return -1, just like any other `InputStream` does.

Combining More Than Two InputStreams

You can combine more than two `InputStream` instances with the `SequenceInputStream` in two ways. The

- 31. [Java IO: FileInputStream](#)
- 32. [Java IO: ObjectInputStream](#)
- 33. [Java IO: ObjectOutputStream](#)
- 34. [Java IO: Serializable](#)
- 35. [Java IO: Reader](#)
- 36. [Java IO: Writer](#)
- 37. [Java IO: InputStreamReader](#)
- 38. [Java IO: OutputStreamWriter](#)
- 39. [Java IO: FileReader](#)
- 40. [Java IO: FileWriter](#)
- 41. [Java IO: PipedReader](#)
- 42. [Java IO: PipedWriter](#)
- 43. [Java IO: CharArrayReader](#)
- 44. [Java IO: CharArrayWriter](#)
- 45. [Java IO: BufferedReader](#)
- 46. [Java IO: BufferedWriter](#)
- 47. [Java IO: FilterReader](#)
- 48. [Java IO: FilterWriter](#)
- 49. [Java IO: PushbackReader](#)
- 50. [Java IO: LineNumberReader](#)
- 51. [Java IO: StreamTokenizer](#)

first way is to put all the `InputStream` instances into a `Vector`, and pass that `Vector` to the `SequenceInputStream` constructor. Here is an example of how passing a `Vector` to the `SequenceInputStream` constructor looks:

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");
InputStream input2 = new FileInputStream("c:\\data\\file2.txt");
InputStream input3 = new FileInputStream("c:\\data\\file3.txt");

Vector<InputStream> streams = new Vector<>();
streams.add(input1);
streams.add(input2);
streams.add(input3);

SequenceInputStream sequenceInputStream =
    new SequenceInputStream(streams.elements());

int data = sequenceInputStream.read();
while(data != -1){
    System.out.println(data);
    data = sequenceInputStream.read();
}
sequenceInputStream.close();
```

The second method is to combine the `InputStream` instances two and two into `SequenceInputStream` instances, and then combine these again with another `SequenceInputStream`. Here is how combining more than two `InputStream` instances with multiple `SequenceInputStream` instances look:

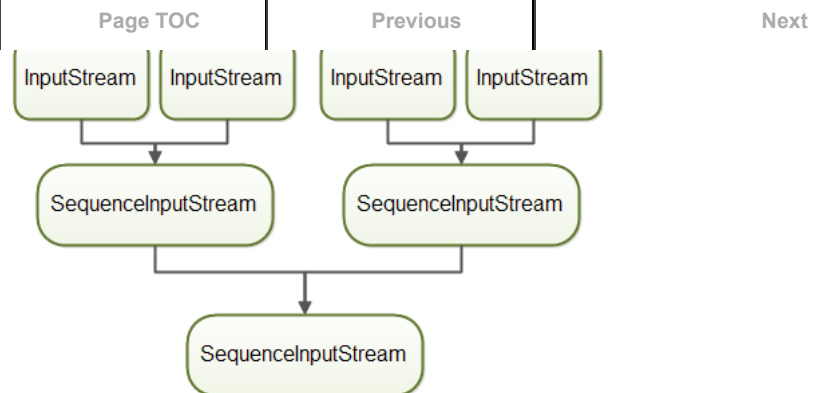
```
SequenceInputStream sequenceInputStream1 =
    new SequenceInputStream(input1, input2);

SequenceInputStream sequenceInputStream2 =
    new SequenceInputStream(input3, input4);

SequenceInputStream sequenceInputStream =
    new SequenceInputStream(
        sequenceInputStream1, sequenceInputStream2){

int data = sequenceInputStream.read();
while(data != -1){
    System.out.println(data);
    data = sequenceInputStream.read();
}
sequenceInputStream.close();
```

The resulting object graph looks like this:


[All Trails](#)
[Trail TOC](#)

- 53. [Java IO: StringReader](#)
- 54. [Java IO: StringWriter](#)

Closing a SequenceInputStream

When you are finished reading data from the `SequenceInputStream` you should remember to close it. Closing a `SequenceInputStream` will also close the `InputStream` instances which the `SequenceInputStream` is reading.

Closing a `SequenceInputStream` is done by calling its `close()` method. Here is how closing a `SequenceInputStream` looks:

```
sequenceInputStream.close();
```

You can also use the [try-with-resources](#) construct introduced in Java 7. Here is how to use and close `SequenceInputStream` looks with the `try-with-resources` construct:

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");
InputStream input2 = new FileInputStream("c:\\data\\file2.txt");

try(SequenceInputStream sequenceInputStream =
    new SequenceInputStream(input1, input2)){
```

```
int data = sequenceInputStream.read();
while(data != -1){
    System.out.println(data);
    data = sequenceInputStream.read();
}
}
```

Notice how there is no longer any explicit `close()` method call. The try-with-resources construct takes care of that.

Notice also that the two `FileInputStream` instances are not created inside the try-with-resources block. That means that the try-with-resources block will not automatically close these two `FileInputStream` instances. However, when the `SequenceInputStream` is closed it will also close the `InputStream` instances it reads from, so the two `FileInputStream` instances will get closed when the `SequenceInputStream` is closed.

Next: [Java IO: DataInputStream](#)



Tweet



Jakob Jenkov

