Module 10

I/O Fundamentals

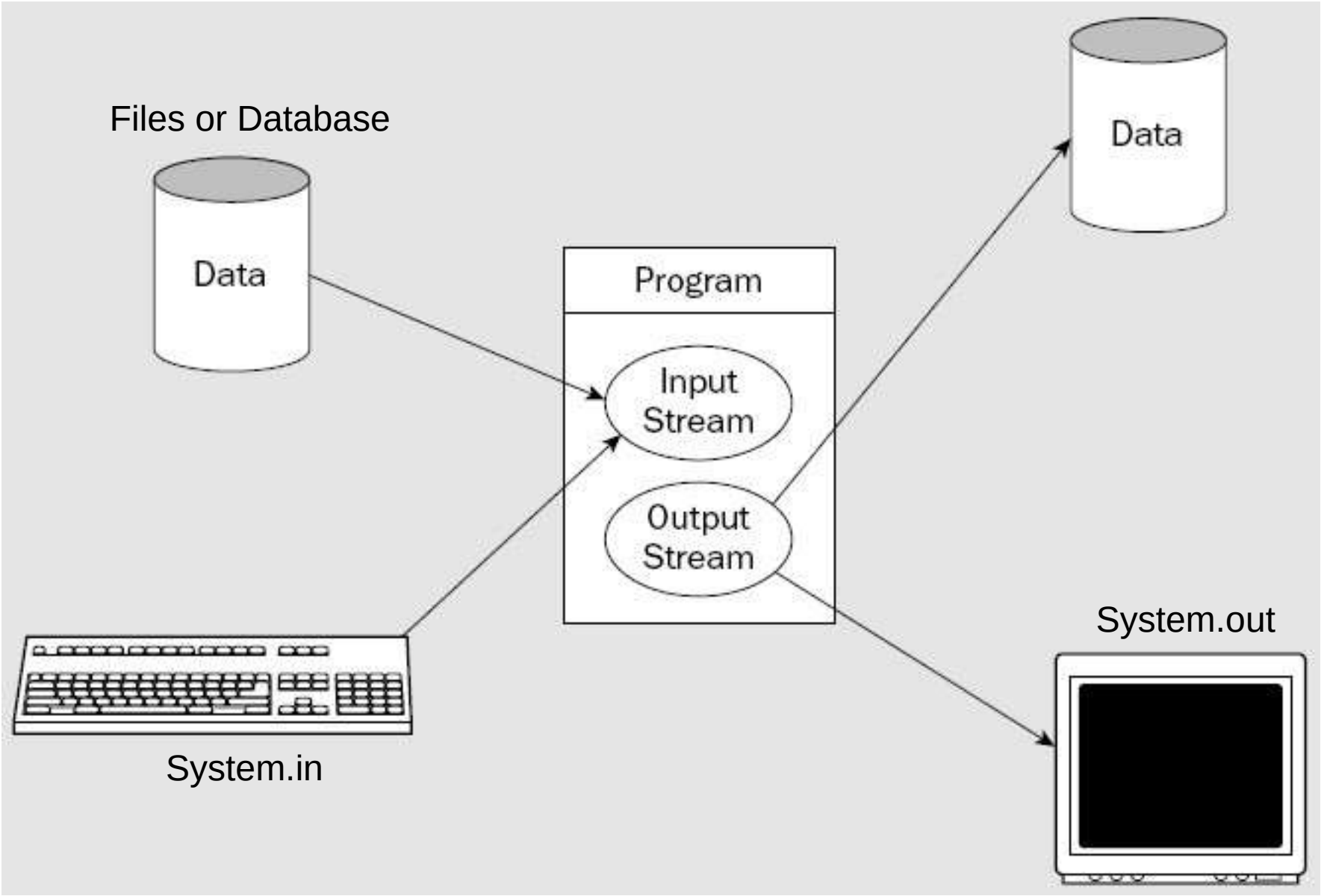
Objectives

- What is a *stream*?
- Class *InputStream* and *OutputStream*
- Class *Reader* and *Writer*
- Node and Processing streams
- Serialize and deserialize objects
- Distinguish readers and writers from streams, and select appropriately between them

Stream

I/O Stream Fundamentals

- A *stream* is a flow of data from a *source* or to a *sink*.
- A *source* stream initiates the flow of data, also called an input stream.
- A *sink* stream terminates the flow of data, also called an output stream.
- Sources and sinks are both *node streams*.
- Types of node streams are files, memory, and pipes between threads or processes.



Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

Fundamental Stream Classes

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

These classes are abstract class, need to define subclasses for usage.

The `InputStream` Methods

- The three basic `read` methods are:

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

- Other methods include:

```
void close()
int available(): Returns an estimate of the number of bytes that can be read
long skip(long n)
boolean markSupported()
void mark(int readlimit) : Marks the current position in this input stream
void reset(): Repositions this stream to the position the mark method was last called
```

The `OutputStream` Methods

- The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

The `Reader` Methods

- The three basic `read` methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

The `Writer` Methods

- The basic `write` methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

Node Streams
vs.
Processing Stream

Node Streams

Type	Character Streams	Byte Streams
File	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Memory: array	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Memory: string	`StringReader` `StringWriter`	N / A
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

A Simple Example

This program performs a copy file operation using a manual buffer:

`java TestNodeStreams file1 file2`

```
1   import java.io.*;
2   public class TestNodeStreams {
3     public static void main(String[] args)
4       { try {
          5         FileReader input = new FileReader(args[0]);
6         try {
7           FileWriter output = new FileWriter(args[1]);
8           try {
9             char[] buffer = new char[128];
10            int charsRead;
11
12            // read the first buffer
13            charsRead = input.read(buffer);
14            while ( charsRead != -1 ) {
15              // write buffer to the output file
```

A Simple Example

```
16              output.write(buffer, 0, charsRead);
17
18              // read the next buffer
19              charsRead = input.read(buffer);
20            }
21
22          } finally
23            { output.close(
24            );}
25        } finally
26      }   { input.close(      {
27            );}
28      }
29    }
30  }
```

Buffered Streams

This program performs a copy file operation using a built-in buffer:

java TestBufferedStreams file1 file2

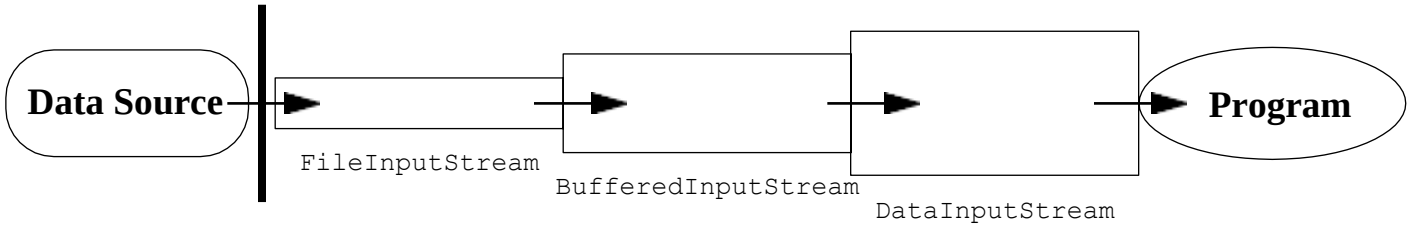
```
1   import java.io.*;
2   public class TestBufferedStreams {
      3     public static void main(String[] args) {
4       try {
5         FileReader input = new FileReader(args[0]);
6         BufferedReader bufInput = new BufferedReader(input);
7         try {
8           FileWriter output = new FileWriter(args[1]);
9           BufferedWriter bufOutput= new BufferedWriter(output);
10          try {
11            String line;
12            // read the first line
13            line = bufInput.readLine();
14            while ( line != null ) {
15              // write the line out to the output file
```

Buffered Streams

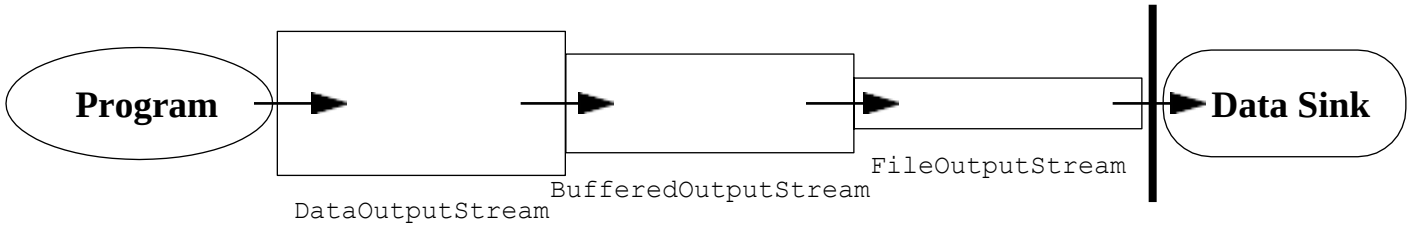
```
16              bufOutput.write(line, 0, line.length());
17              bufOutput.newLine();
18              // read the next line
19              line = bufInput.readLine();
20            }
21          } finally
22            { bufOutput.close(
23            );
24          }
25        } finally
26          { bufInput.close
27      } catch (IOException e) {
28        e.printStackTrace();
29      }
30    }
31  }
32
33
```

I/O Stream Chaining

Input Stream Chain



Output Stream Chain



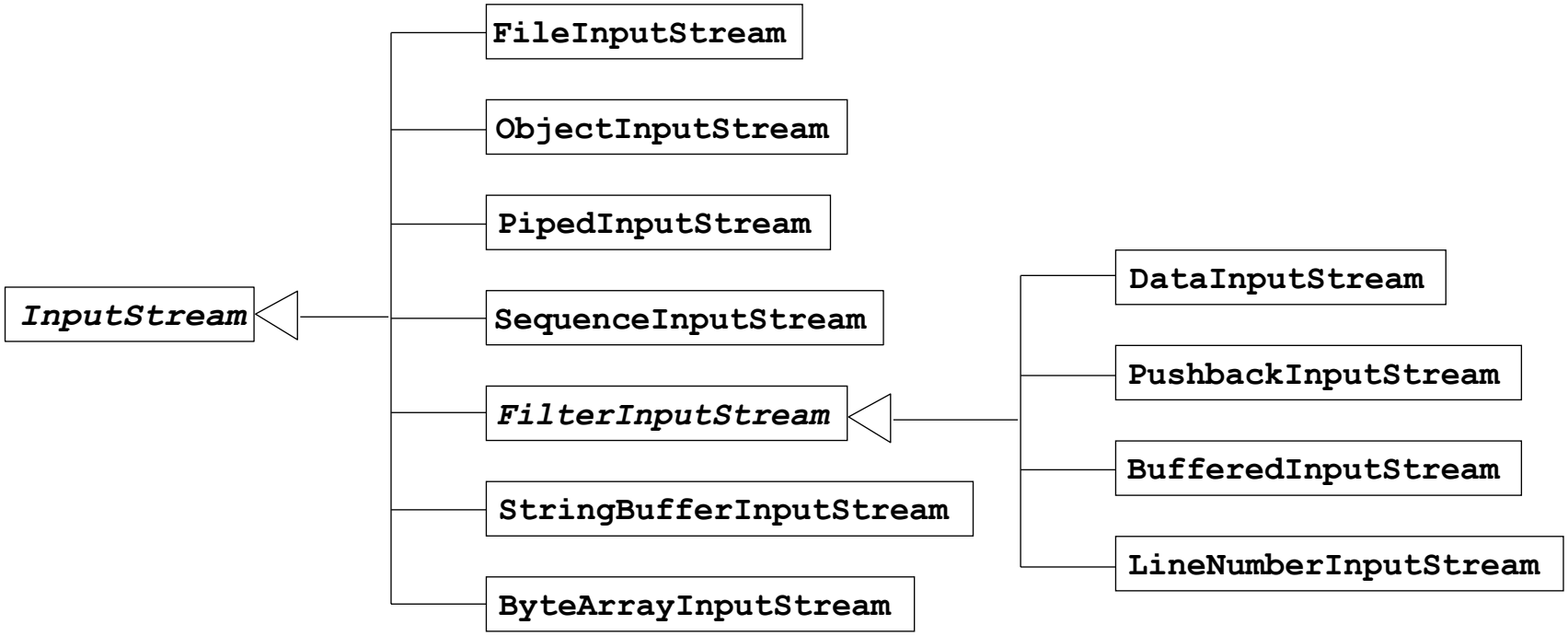
Processing Streams

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	*`FilterReader`* *`FilterWriter`*	*`FilterInputStream`* *`FilterOutputStream`*
Converting between bytes and character	`InputStreamReader` `OutputStreamWriter`	
Performing object serialization		`ObjectInputStream` `ObjectOutputStream`

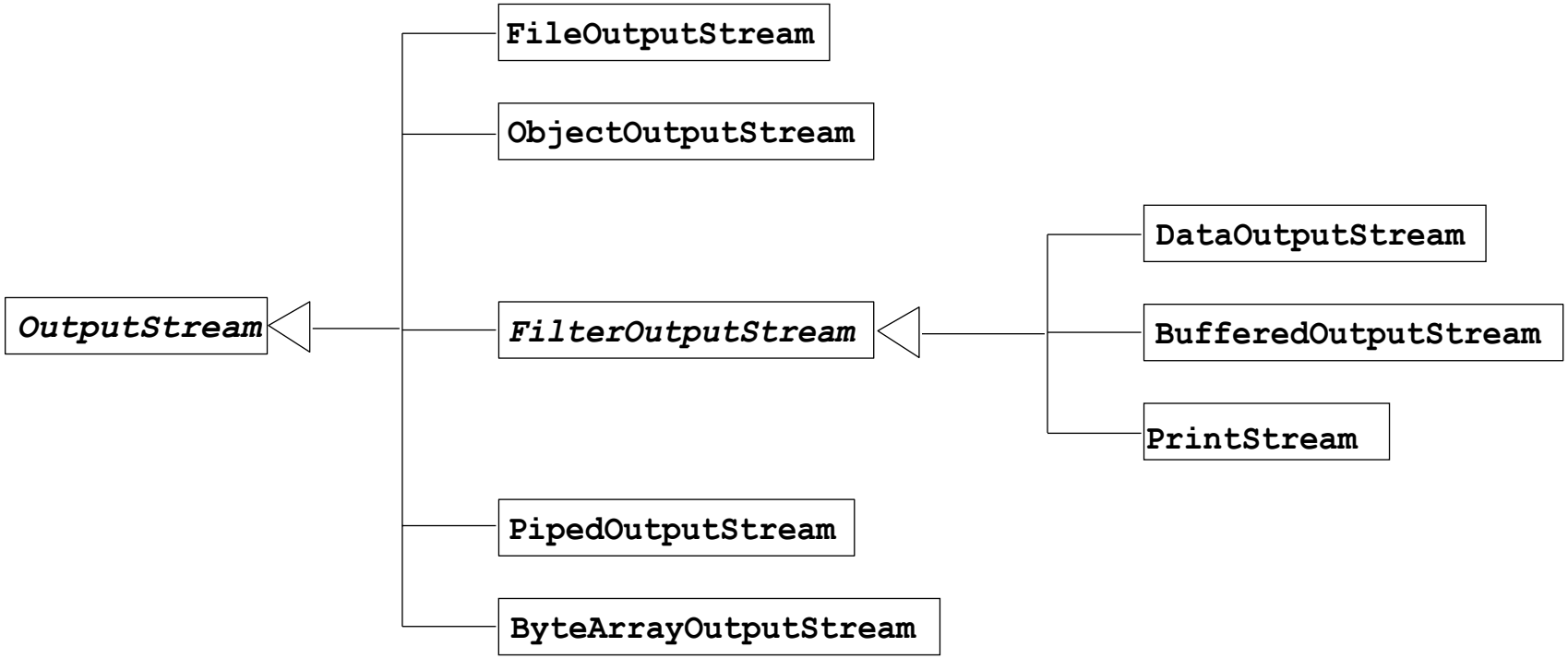
Processing Streams

Type	Character Streams	Byte Streams
Performing data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

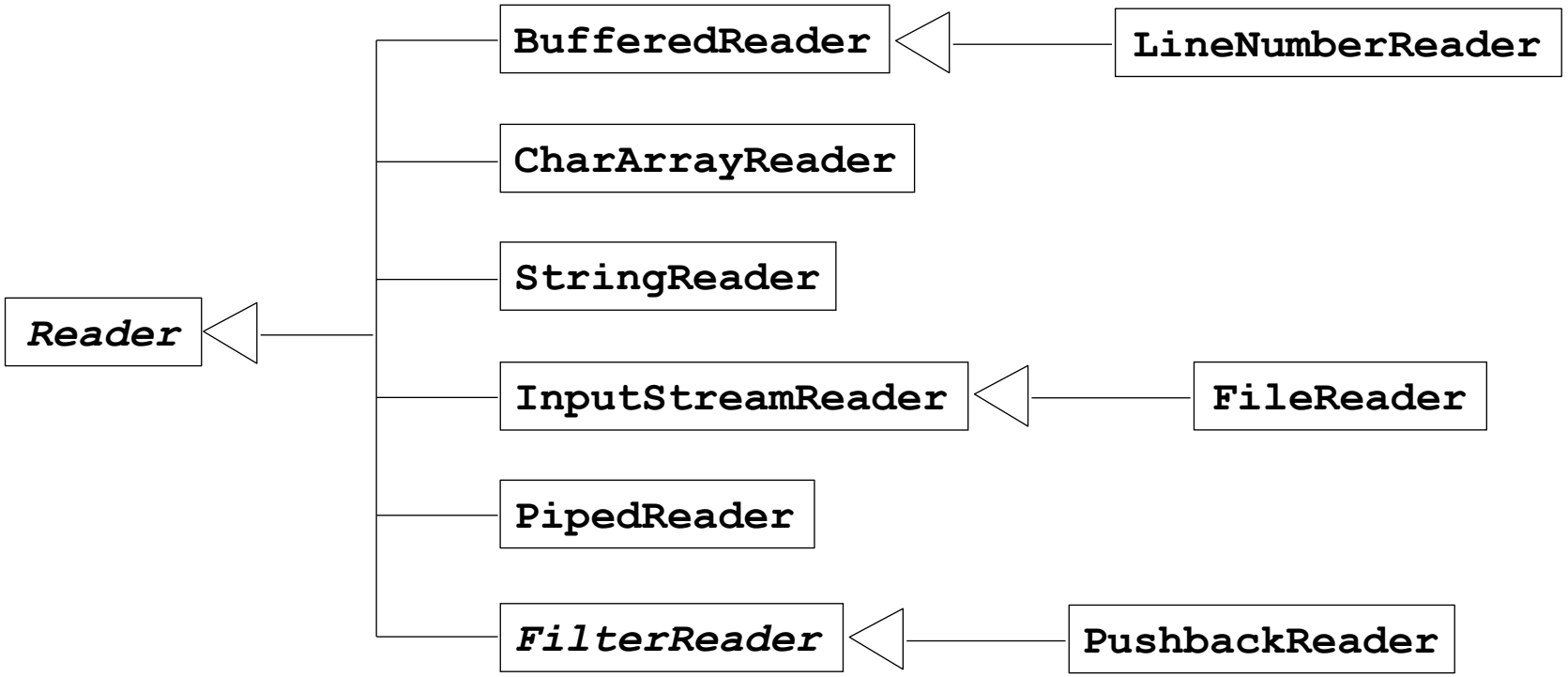
The `InputStream` Class Hierarchy



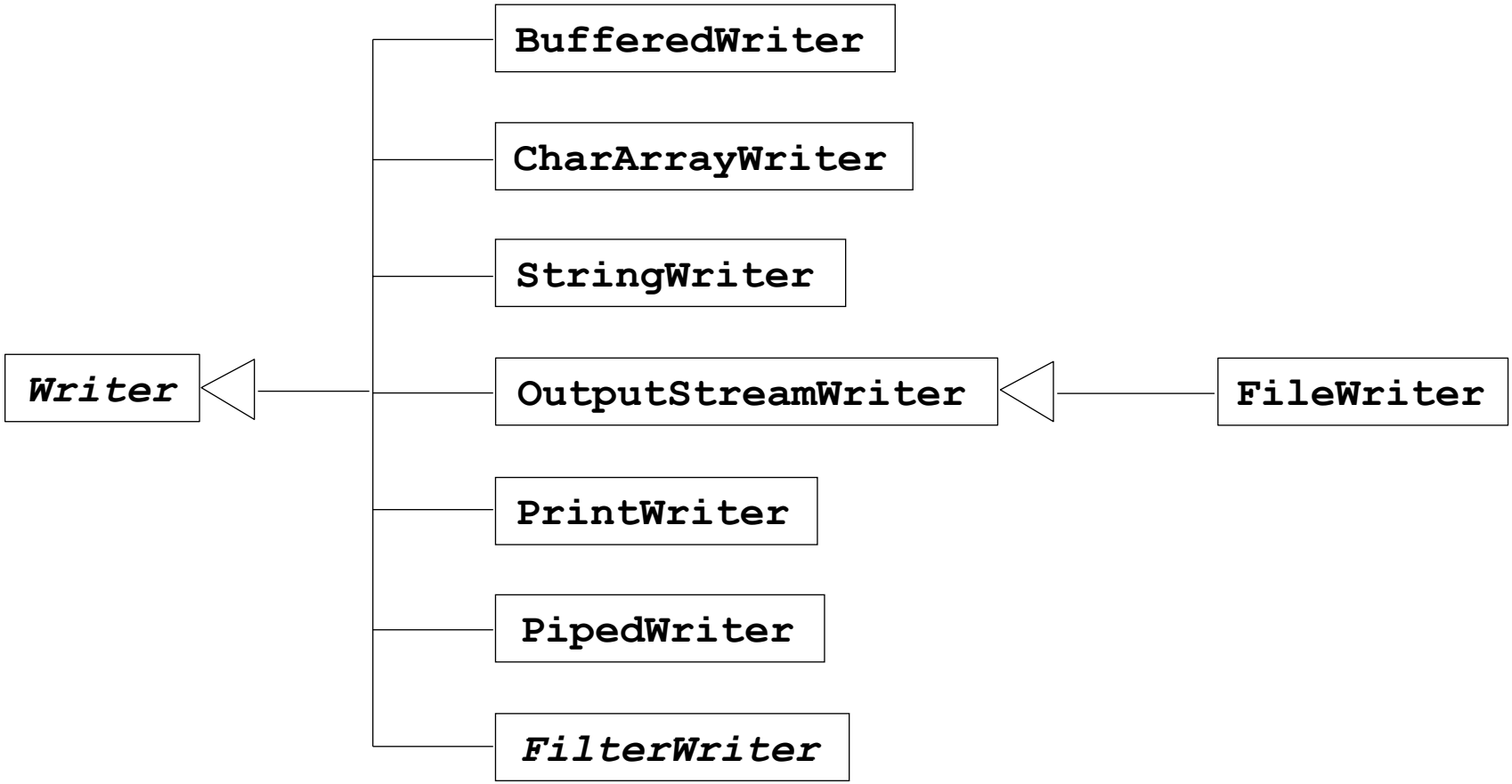
The `OutputStream` Class Hierarchy



The `Reader` Class Hierarchy



The `Writer` Class Hierarchy



Serialization

The `ObjectInputStream` and The `ObjectOutputStream` Classes

- ObjectSerialization is a mechanism for saving the objects as a sequence of bytes and rebuilding them later when needed
  - writing and reading objects from streams.
- Writing an object
  - the object output stream writes the class name, followed by a description of the data members of the class, in the order they appear in the stream, followed by the values for all the fields on that object.
- Reading an object
  - the object input stream reads the name of the class and the description of the class to match against the class in memory, and it reads the values from the stream to populate a newly allocation instance of that class.

Serialization

- Persistent storage of objects can be accomplished if files (or other persistent storage) are used as streams.
- Only the object's data are serialized
  - only the *fields* of the object are preserved
  - When a field references an object, the fields of the referenced object are also serialized
- Serializability of a class is enabled by the class implementing the *java.io.Serializable* interface.
  - The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

Serialization

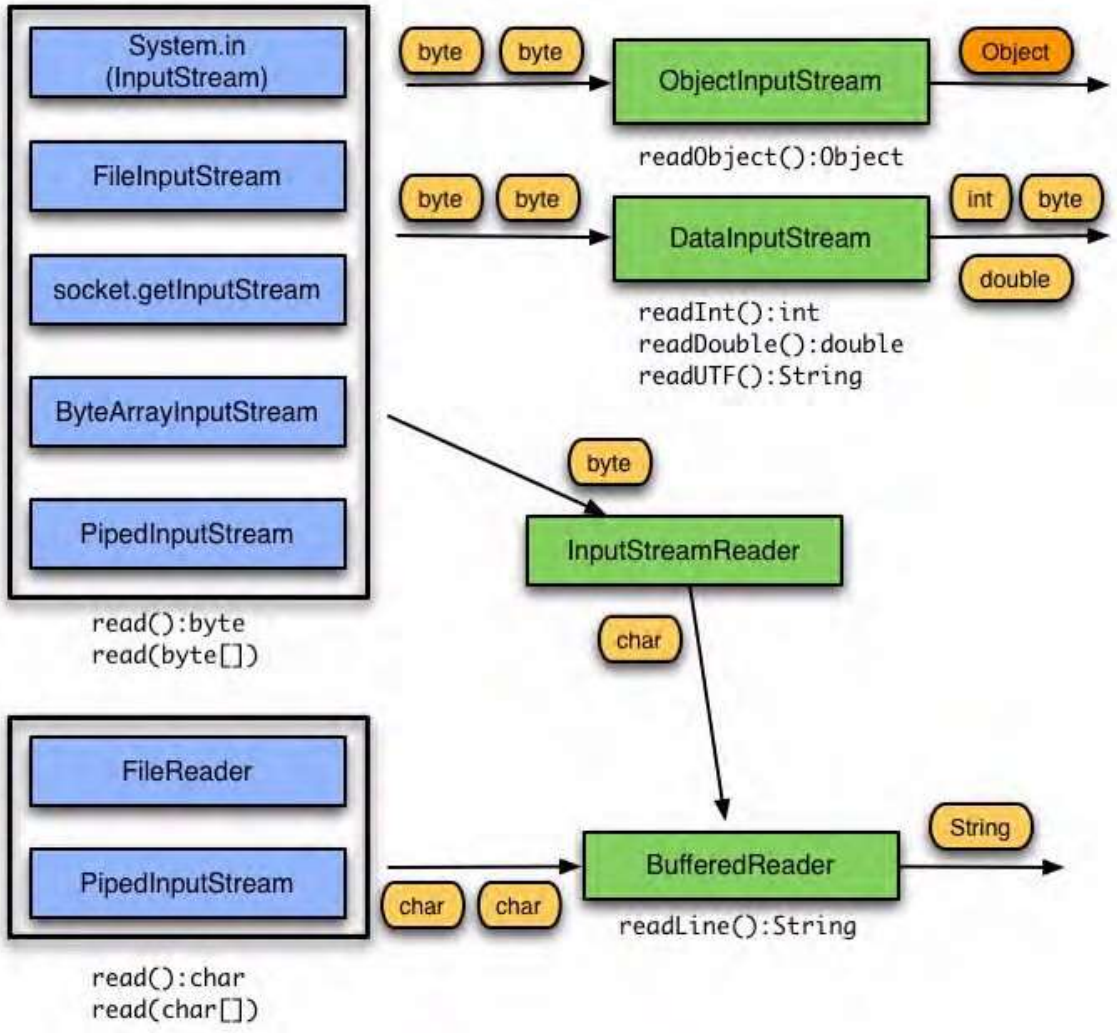
- Data marked with the `transient` keyword are not serialized

```
1   public class MyClass implements Serializable  {
2     public transient Thread myThread;
3     private String customerID;
4     private int total;
5   }
```

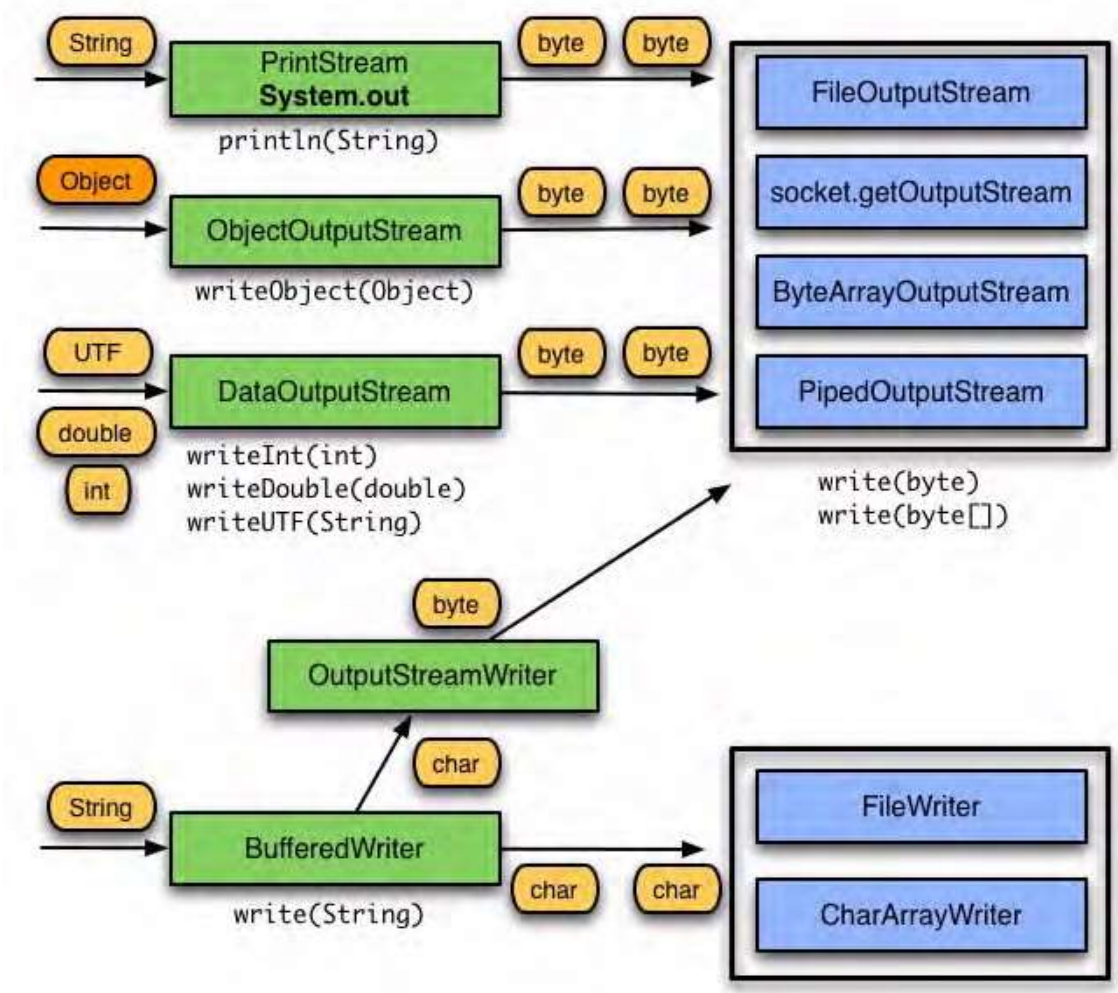
```
1   public class MyClass implements Serializable  {
2     public transient Thread myThread;
3     private transient String customerID;
4     private int total;
5   }
```

- Serialization stores the state of an object to a file; storing the state of an object is called *persistence*

Input Chaining Combinations: A Review



Output Chaining Combinations: A Review



Module 11

Console I/O and File I/O

Objectives

- Read data from the console
- Write data to the console
- Describe files and file I/O

Console I/O

Console I/O

- The variable `System.out` enables you to write to *standard output*.

  `System.out` is an object of type `PrintStream`.
- The variable `System.in` enables you to read from *standard input*.

  `System.in` is an object of type `InputStream`.
- The variable `System.err` enables you to write to *standard error*.

  `System.err` is an object of type `PrintStream`.

Writing to Standard Output

- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Reading From Standard Input

```
1   import java.io.*;
2
3   public class KeyboardInput {
4     public static void main (String args[])
5       { String s;
6       // Create a buffered reader to read
7       // each line from the keyboard.
8       InputStreamReader ir
9         = new InputStreamReader(System.in);
10      BufferedReader in = new BufferedReader(ir);
11
12      System.out.println("Unix: Type ctrl-d to exit." +
13                         "\nWindows: Type ctrl-z to exit");
```

Reading From Standard Input

```
14      try {
15        // Read each input line and echo it to the screen.
16        s = in.readLine();
17        while ( s != null ) {
18          System.out.println("Read: " + s);
19          s = in.readLine();
20        }
21
22        // Close the buffered reader.
23        in.close();
24      } catch (IOException e) { // Catch any IO exceptions.
25        e.printStackTrace();
26      }
27    }
28  }
```

Simple Formatted Output

- You can use the formatting functionality as follows:

```
out.printf("name count\n");
String s = String.format("%s %5d%n", user, total);
```

- Common formatting codes are listed in this table.

Code	Description
`%s`	Formats the argument as a string, usually by calling the `toString` method on the object.
`%d %o %x`	Formats an integer, as a decimal, octal, or hexadecimal value.
`%f %g`	Formats a floating point number. The `%g` code uses scientific notation.
`%n`	Inserts a newline character to the string or stream.
`%%`	Inserts the `%` character to the string or stream.

Simple Formatted Input

- The `Scanner` class provides a formatted input function.
- A `Scanner` class can be used with console input streams as well as file or network streams.
- You can read console input as follows:

```
1   import java.io.*;
2   import java.util.Scanner;
3   public class ScanTest {
       4     public static void main(String [] args)  {
5        Scanner s = new Scanner(System.in);
6        String param = s.next();
7        System.out.println("the param 1" + param);
8        int value = s.nextInt();
9        System.out.println("second param" + value);
10       s.close();
11     }
12  }
```

File I/O

Files and File I/O

The `java.io` package enables you to do the following:

- Create `File` objects
- Manipulate `File` objects
- Read and write to file streams

Creating a New `File` Object

The `File` class provides several utilities:

- `File myFile;`
- `myFile = new File("myfile.txt");`
- `myFile = new File("MyDocs", "myfile.txt");`

Directories are treated like files in the Java programming language. You can create a `File` object that represents a directory and then use it to identify other files, for example:

```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

The `File` Tests and Utilities

- File information:

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
long lastModified()
long length()
```

- File modification:

```
boolean renameTo(File newName)
boolean delete()
```

- Directory utilities:

```
boolean mkdir()
String[] list()
```

The `File` Tests and Utilities

- File tests:

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute();
boolean is Hidden();
```

File Stream I/O

- For file input:
  - Use the `FileReader` class to read characters.
  - Use the `BufferedReader` class to use the `readLine` method.
- For file output:
  - Use the `FileWriter` class to write characters.
  - Use the `PrintWriter` class to use the `print` and `println` methods.

File Output Example

```
1   import java.io.*;
2
3   public class WriteFile {
4     public static void main (String[] args) {
5       // Create file
6       File file = new File(args[0]);
7
8       try {
9         // Create a buffered reader to read each line from standard in.
10        InputStreamReader isr
11          = new InputStreamReader(System.in);
12        BufferedReader in
13          = new BufferedReader(isr);
14        // Create a print writer on this file.
15        PrintWriter out
16          = new PrintWriter(new FileWriter(file));
17        String s;
```

File Output Example

```
18
19        System.out.print("Enter file text.  ");
20        System.out.println("[Type ctrl-d to stop.]");
21
22        // Read each input line and echo it to the screen.
23        while ((s = in.readLine()) != null) {
24          out.println(s);
25        }
26
27        // Close the buffered reader and the file print writer.
28        in.close();
29        out.close();
30
31      } catch (IOException e) {
32      // Catch any IO exceptions.
33        e.printStackTrace();
34      }
35    }
36  }
```

File Input Example

A file input example is:

```
1   import java.io.*;
2   public class ReadFile {
3     public static void main (String[] args) {
4       // Create file
5       File file = new File(args[0]);
6
7       try {
8         // Create a buffered reader
9         // to read each line from a file.
10        BufferedReader in
11          = new BufferedReader(new FileReader(file));
12        String s;
13
```

Printing a File

```
14          // Read each line from the file and echo it to the screen.
15          s = in.readLine();
16          while ( s != null )
17            { System.out.println("Read: " +
18            s); s = in.readLine();
19          }
20          // Close the buffered reader
21          in.close();
22
23        } catch (FileNotFoundException e1) {
24          // If this file does not exist
25          System.err.println("File not found: " + file);
26
27        } catch (IOException e2) {
28          // Catch any other IO exceptions.
29          e2.printStackTrace();
30        }
31      }
32  }
```

Scan from a File

- The *Scanner* class breaks its input into tokens using a delimiter pattern which by default matches *whitespace*.
- The resulting tokens may be converted into values of different types using the various *next* methods.
- Read long types from a file:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
  long aLong = sc.nextLong();
.
.
```

- The *Scanner* can also use delimiters other than *whitespace*.

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).
            useDelimiter("\\s*fish\\s*");
will read :  1  2  red  blue
```

Creating a Random Access File

- With the file name:

```
myRAFile = new RandomAccessFile( Stri
                   ng name, String     mode);
```

- With a `File` object:

```
myRAFile = new RandomAccessFile( Fi
                   le file, String   mode);
```

Random Access Files

- `long getFilePointer()`
- `void seek(long pos)`
- `long length()`

URL Input Streams*

URL（Uniform Resource Locator）

- protocol://host:port/path?query#fragment
- http://www.runoob.com/index.html?language=cn#j2se
  - protocol：http
  - host：www.runoob.com
  - port：80
  - path：/index.html
  - query parameters：language=cn
  - fragment position：j2se

URL InputStreams

- java.net.URL
  - public *URL*(String url)
  - public final InputStream *openStream*()
- URLReader.java
- ToyWebBrowser.java

File I/O (Featuring NIO.2)

- **java.nio.file**
- **The Paths Class(**Since JDK 7**)**

A Path instance contains the information used to specify the location of a file or directory.

To create a Path object :

Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));

- **The Files Class(**Since JDK 7**)**

```
Path file = ...;
boolean isRegularExecutableFile =
    Files.isRegularFile(file) &
    Files.isReadable(file) &
    Files.isExecutable(file);
Files.delete(file);
Files.copy(source, target,REPLACE_EXISTING);
```

File Operations with NIO.2

- Create File

```
Path target = Paths.get("D:/Backup/MyStuff.txt");
Path file = Files.createFile(target);
```

- Delete File

```
Path target = Paths.get("D:/Backup/MyStuff.txt");
Files.delete(target);
```

- Copy File

```
Path source  = Paths.get("D:/Backup/MyStuff.txt");
Path target = Paths.get("D:/Backup/MyStuff.txt");
Files.copy(source, target);
```

- Move File

```
import static java.nio.file.StandardCopyOption.*;
Path source = Paths.get("C:/My Documents/Stuff.txt");
Path target = Paths.get("D:/Backup/MyStuff.txt");
Files.move(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

Summary

- I/O Stream
- *InputStream*, *OutputStream* and *Reader*, *Writer*
- Node Streams and Processing Stream
- Serialize and DeSerialize
- Console I/O
- File I/O
- Random Access Files
- URL Input Streams*
- File Operations with NIO.2*