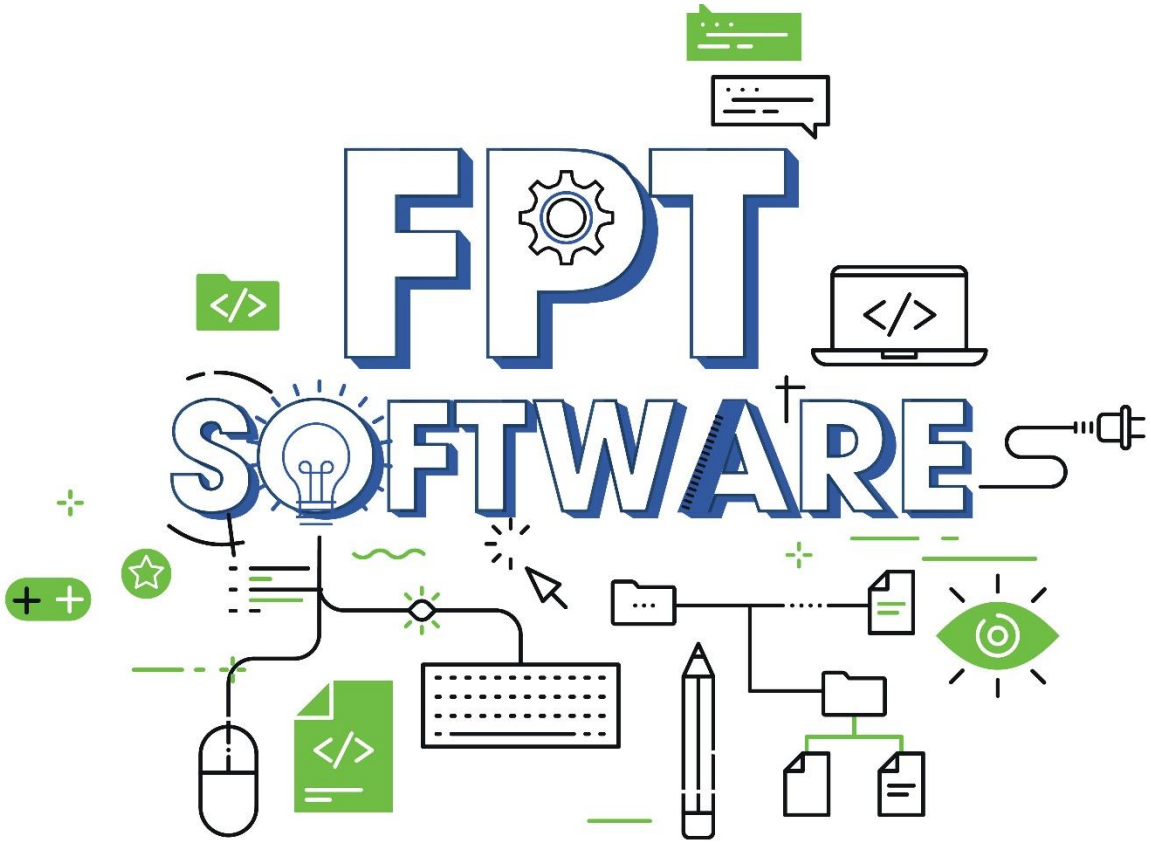
Basic Java IO

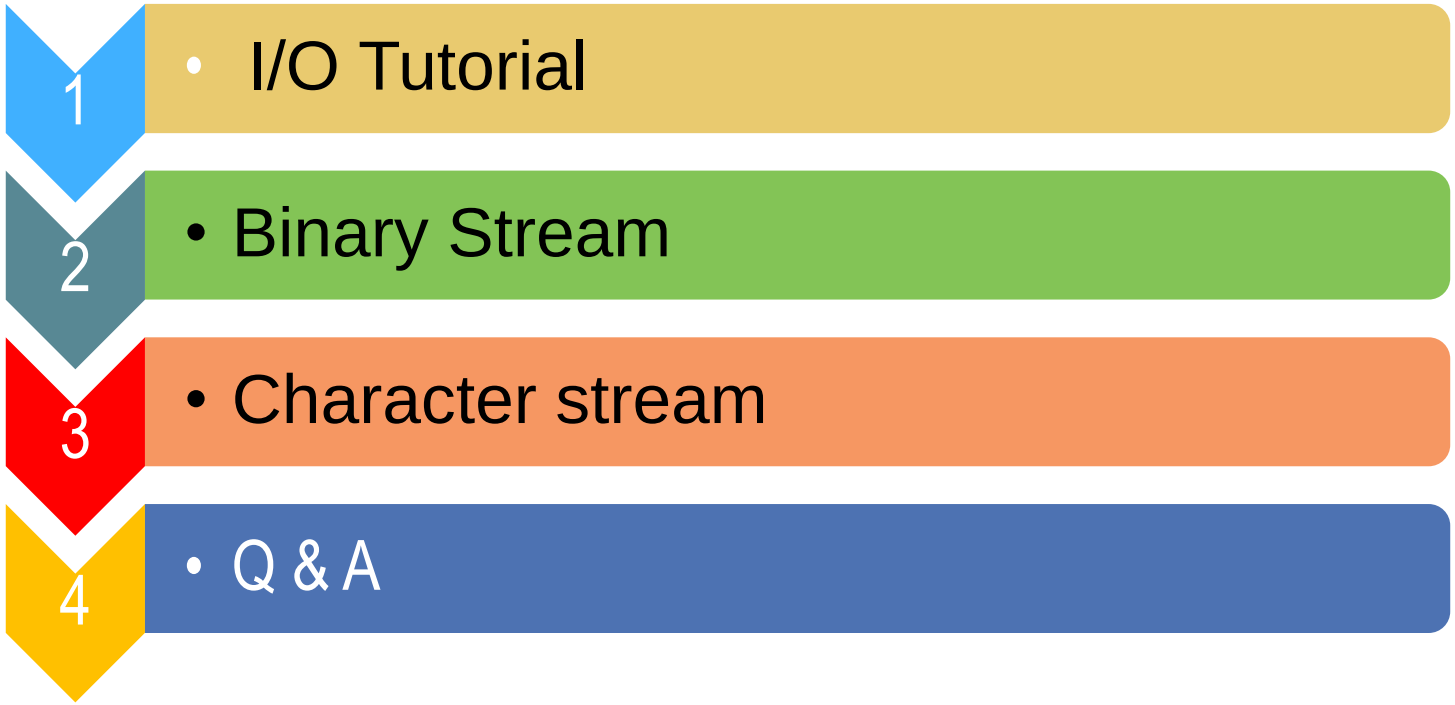
Fsoft Academy



Lesson Objectives

- ***Understand the Java IO Package*** *and* ***its key classes and interfaces****, including File, InputStream, OutputStream, Reader, and Writer.*
- ***Understand the differences*** *between byte-oriented and character-oriented IO.*
- ***Understand the advantages*** *of using character streams for text processing.*
- ***Able to read input*** *from the user and write output to the console.*
- ***Able to read and write binary data*** *using classes such as FileInputStream, FileOutputStream, BufferedInputStream, and BufferedOutputStream.*
- ***Understand how to handle IO-related exceptions*** *effectively.*
- ***Understand the role of the Serializable interface*** *and how to make a class serializable.*

Agenda



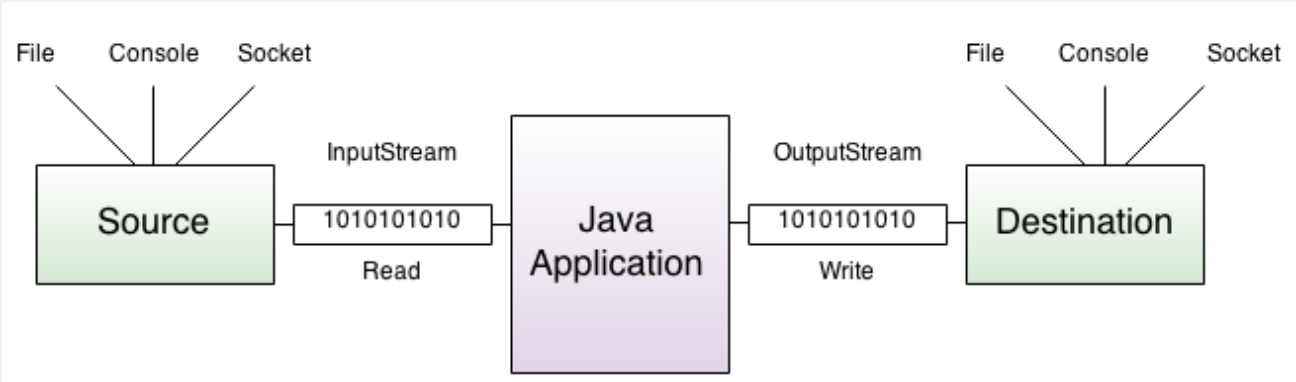
Section 1

I/O Tutorial

09e-BM/DT/FSOFT ©FPT SOFTWARE - Fresher Academy - Internal Use

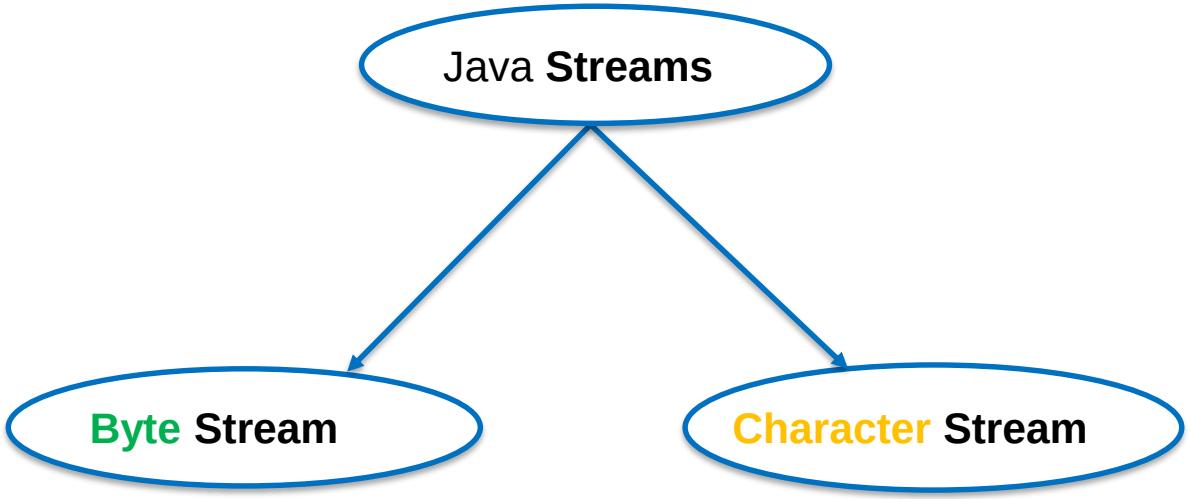
Overview

- **Java I/O** is used for *input and output processing*.
- Java uses streams to make I/O fast. The `java.io package` has I/O classes.
- File handling is done with Java I/O API.
- **Stream**: Sequence of bytes, like a stream of water. There are two stream types:
  - ✓ **InputStream**: Used to read data from a source.
  - ✓ **OutputStream**: Used to write data to a destination.



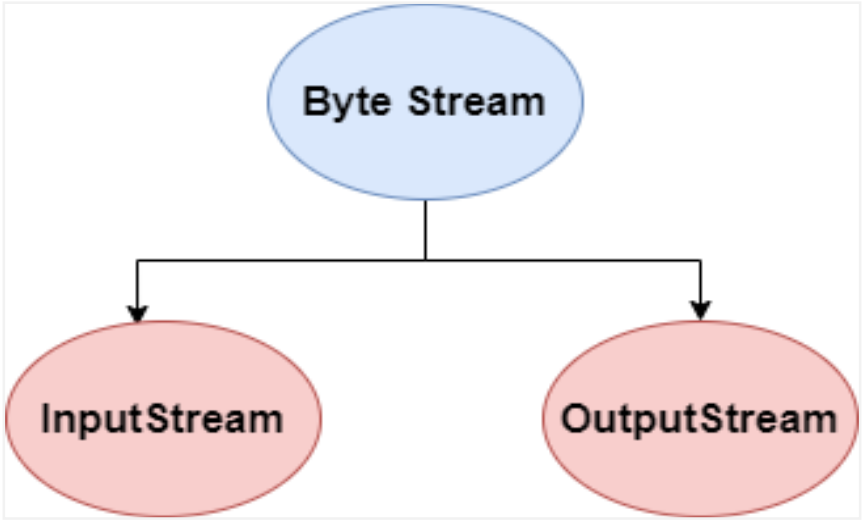
Java Streams

- **Types of streams:**
  - ✓ **Byte Stream:** It provides a convenient means for handling input and output of byte.
  - ✓ **Character Stream:** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

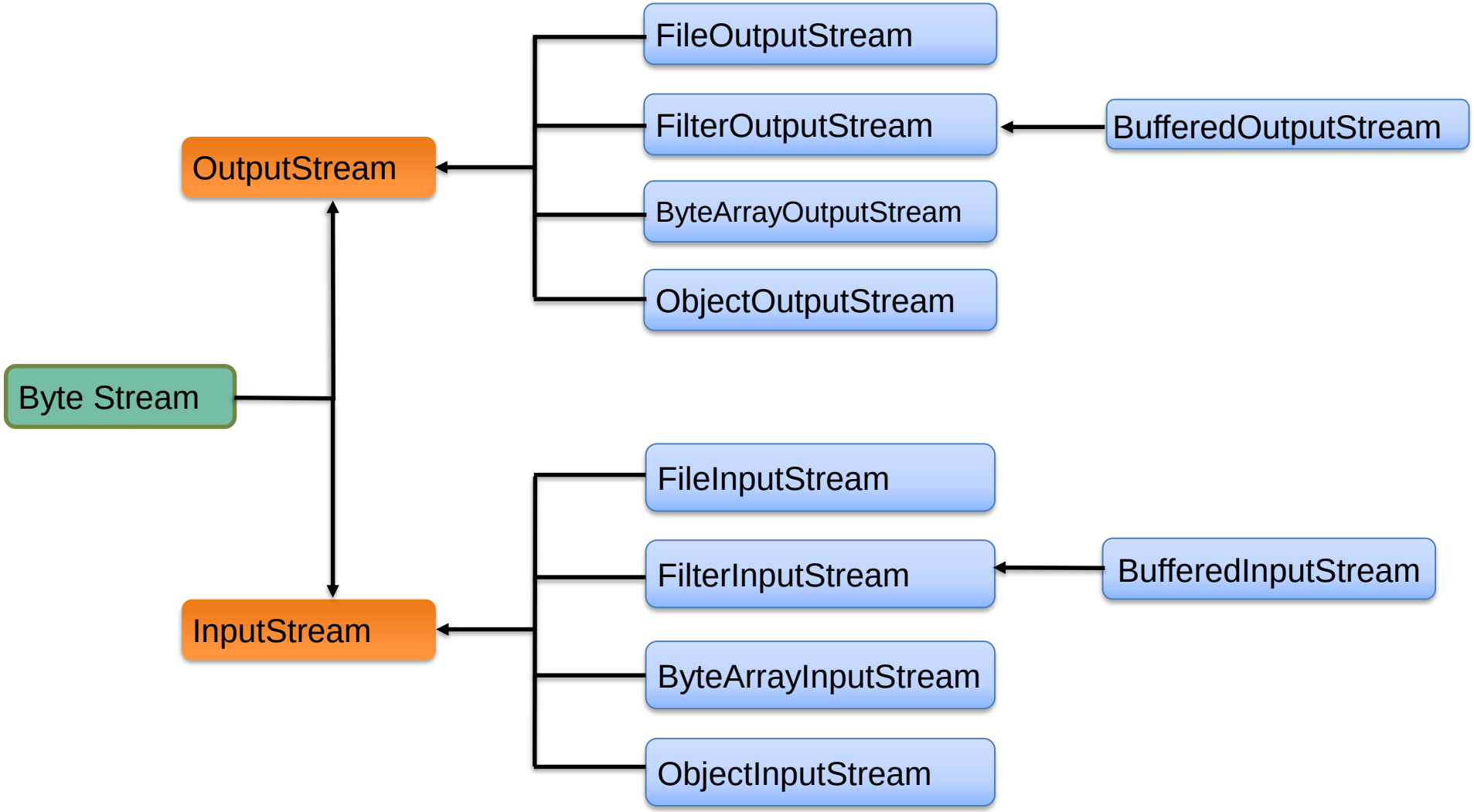


Byte Stream

- Byte Stream classes are used to read **bytes** from an input stream and write **bytes** to an output stream..
- Most common byte stream classes are **FileInputStream** and **FileOutputStream**.

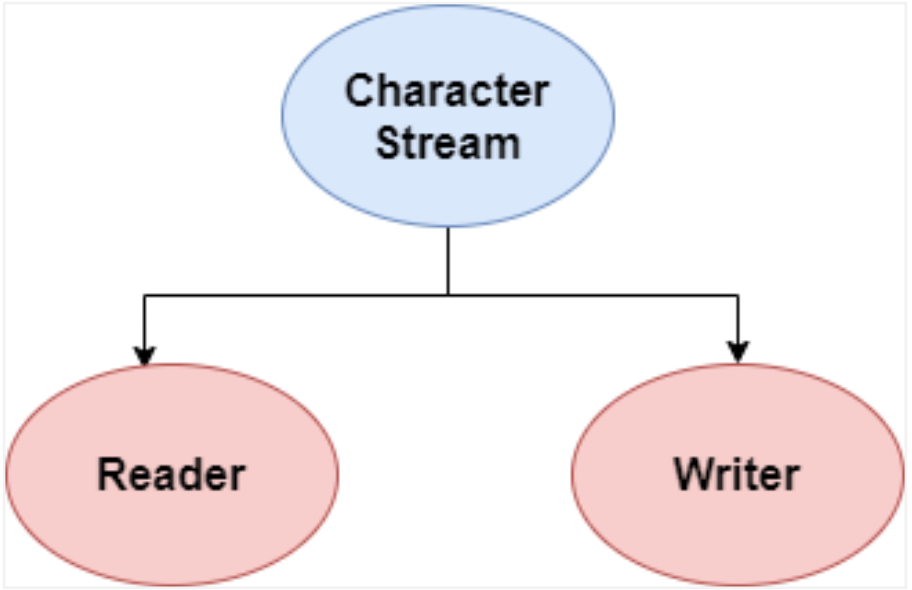


Byte Stream


FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

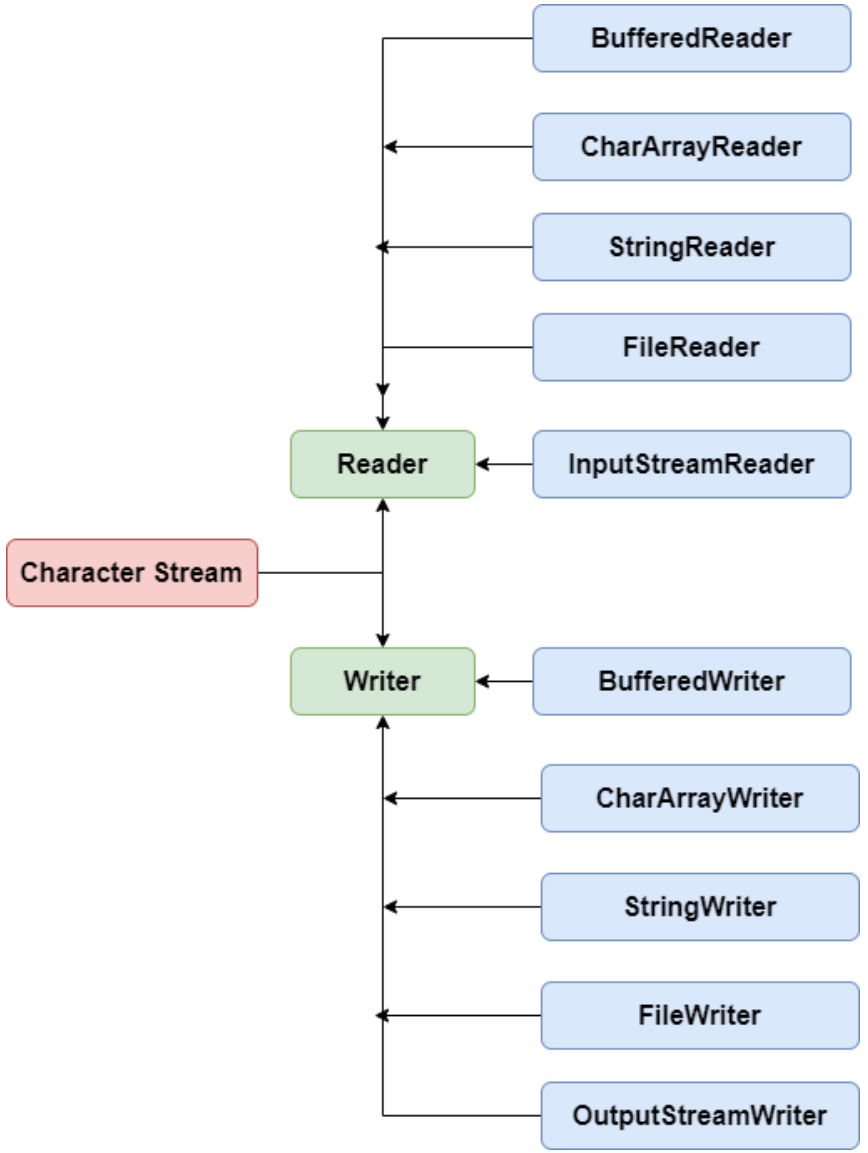
Character Stream

- Java Byte streams handle 8-bit bytes, **Character streams** handle 16-bit Unicode chars.



Character Stream

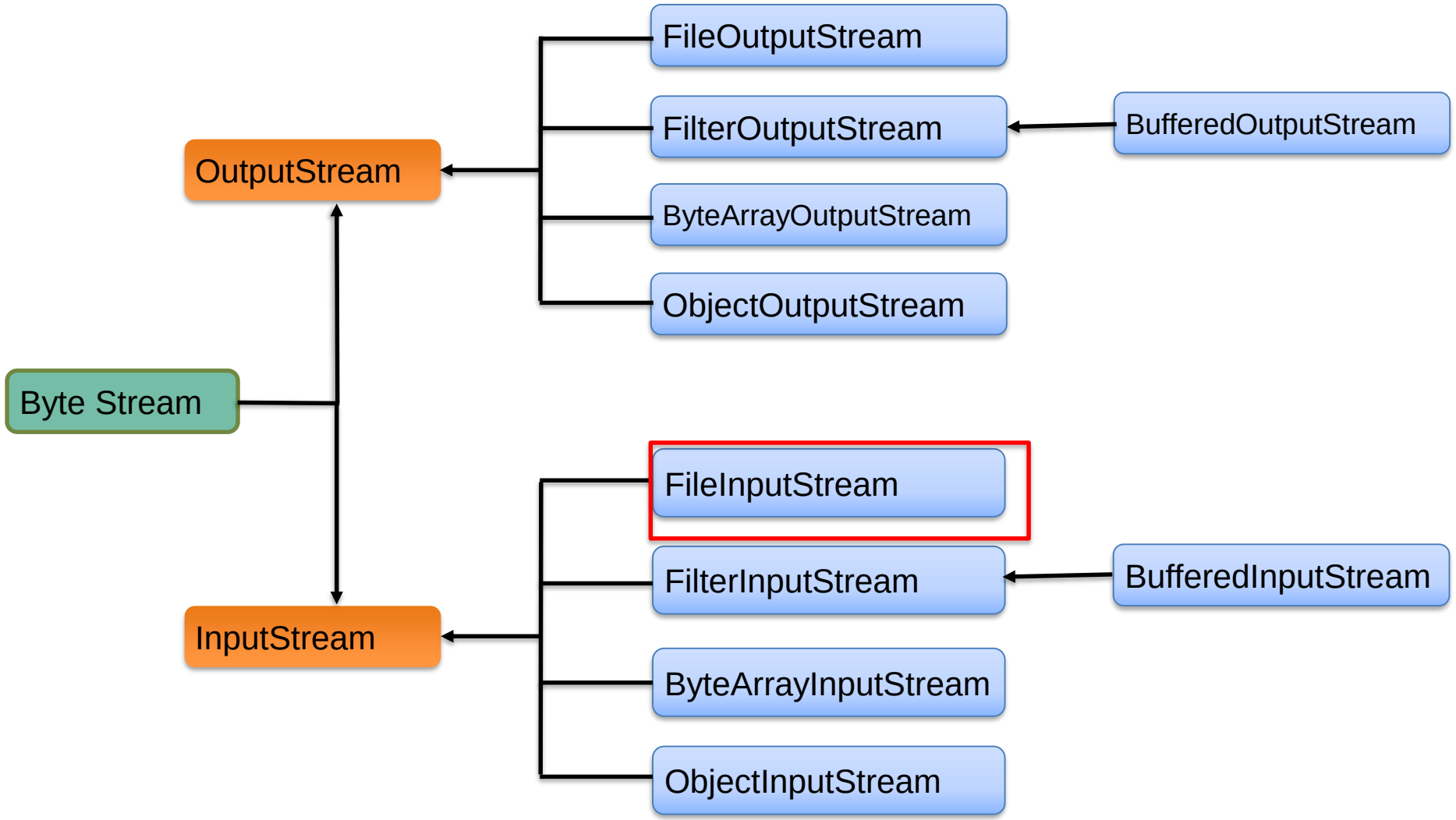
- **Hierarchy class:**



Section 2

Binary Stream

Byte Stream


FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

FileInputStream class

- **FileInputStream** obtains input bytes from a file. Used for reading byte data like *images*, *audio*, *video*.
- **Constructors:**
  - ✓ **FileInputStream**(File file): Opens file in filesystem.
  - ✓ **FileInputStream**(FileDescriptor fdObj): Uses file descriptor for existing filesystem connection.
  - ✓ **FileInputStream**(String name): Opens file connection using pathname.

FileInputStream methods

- **close**(): Closes stream, releases resources.
- **read**(): Reads a byte, returns -1 at end of file.
- **read**(byte[] b): Reads bytes into byte array b, returns num bytes or -1.
- **read**(byte[] b, int off, int len): Reads up to len bytes into array b at offset off, returns num bytes or -1.
- **skip**(long x): Skips and discards x bytes from stream.

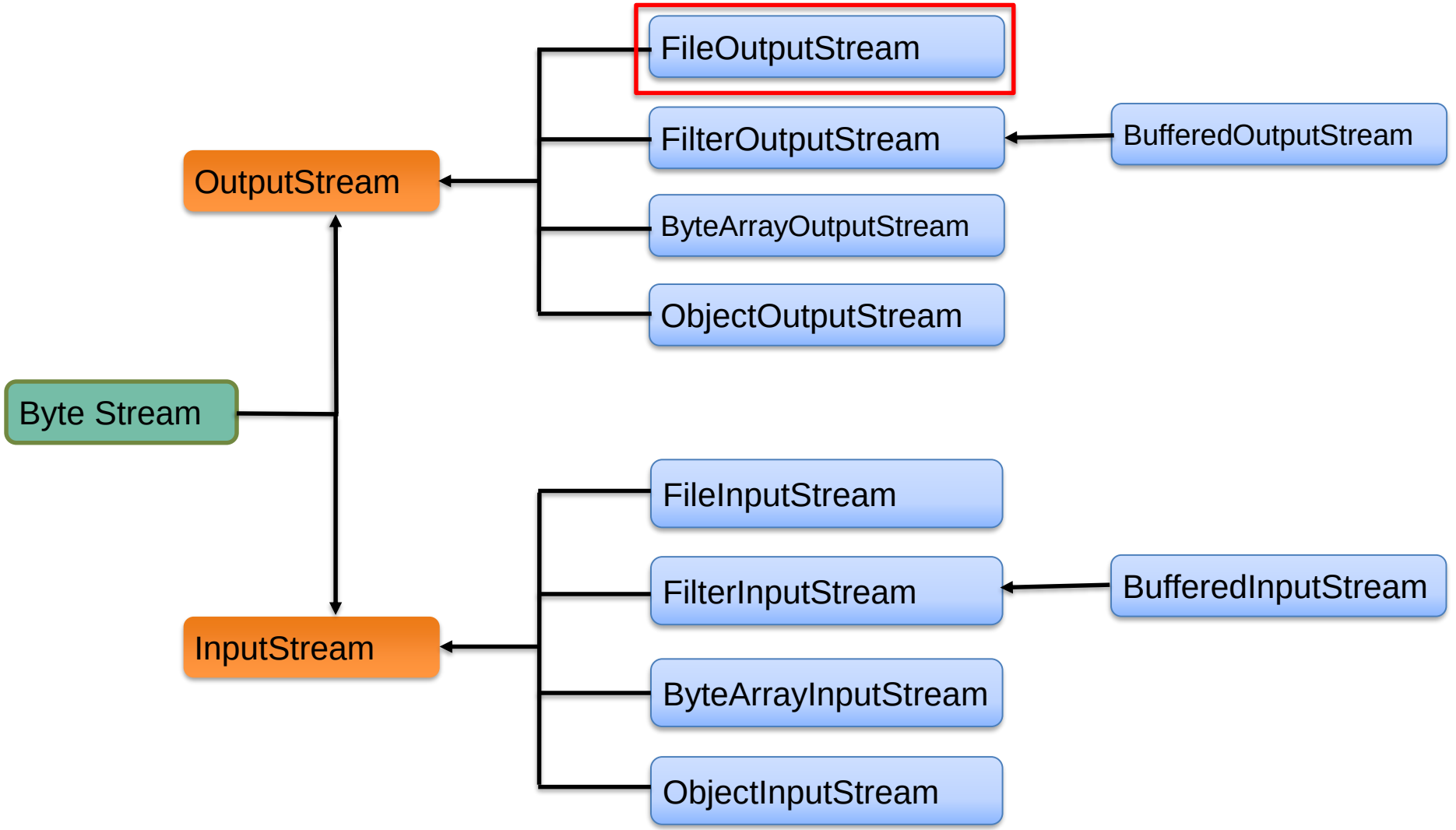
FileInputStream methods

- int **available**(): Returns estimate of readable bytes without blocking.

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

        // attach the file to FileInputStream
        FileInputStream fin = new FileInputStream("src/main/resources/data.txt");

        // illustrating available method
        System.out.println("Number of remaining bytes:" + fin.available());

        // illustrating skip method
        /*Original File content:
         * This is my first line
         * This is my second line*/
        fin.skip(5);
        System.out.println("FileContents :");
        // read characters from FileInputStream and write them
        int ch;
        while ((ch = fin.read()) != -1)
            System.out.print((char) ch);

        // close the file
        fin.close();
    }
}
```

```
Number of remaining bytes:46
FileContents :
is my first line
This is my second line
```

Byte Stream


FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

FileOutputStream class

- **FileOutputStream** class belongs to byte stream and stores the data in the form of individual bytes.
- **FileOutputStream** constructors:
  - ✓ **FileOutputStream**(File file) Writes to File object.
  - ✓ **FileOutputStream**(File file, boolean append): Appends to File object.
  - ✓ **FileOutputStream**(FileDescriptor fdObj): Writes to file descriptor.
  - ✓ **FileOutputStream**(String name): Writes to file by name.
  - ✓ **FileOutputStream**(String name, boolean append) Appends to file by name.

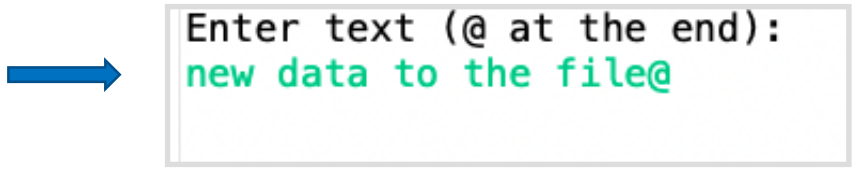
FileOutputStream class

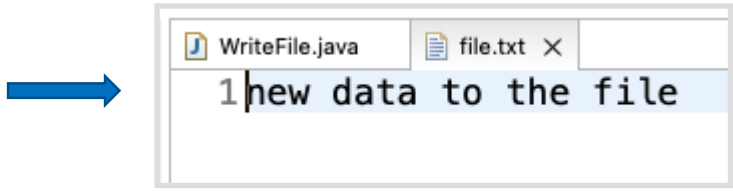
- Important **FileOutputStream** methods:
  - ✓ **close**() Closes stream, releases resources.
  - ✓ **finalize**() Cleans up connection, calls close() on garbage collection.
  - ✓ **write**(byte[] b) Writes byte array to stream.
  - ✓ **write**(byte[] b, int off, int len) Writes bytes from array at offset and length to stream.
  - ✓ **write**(int b) Writes byte to stream.

FileOutputStream class

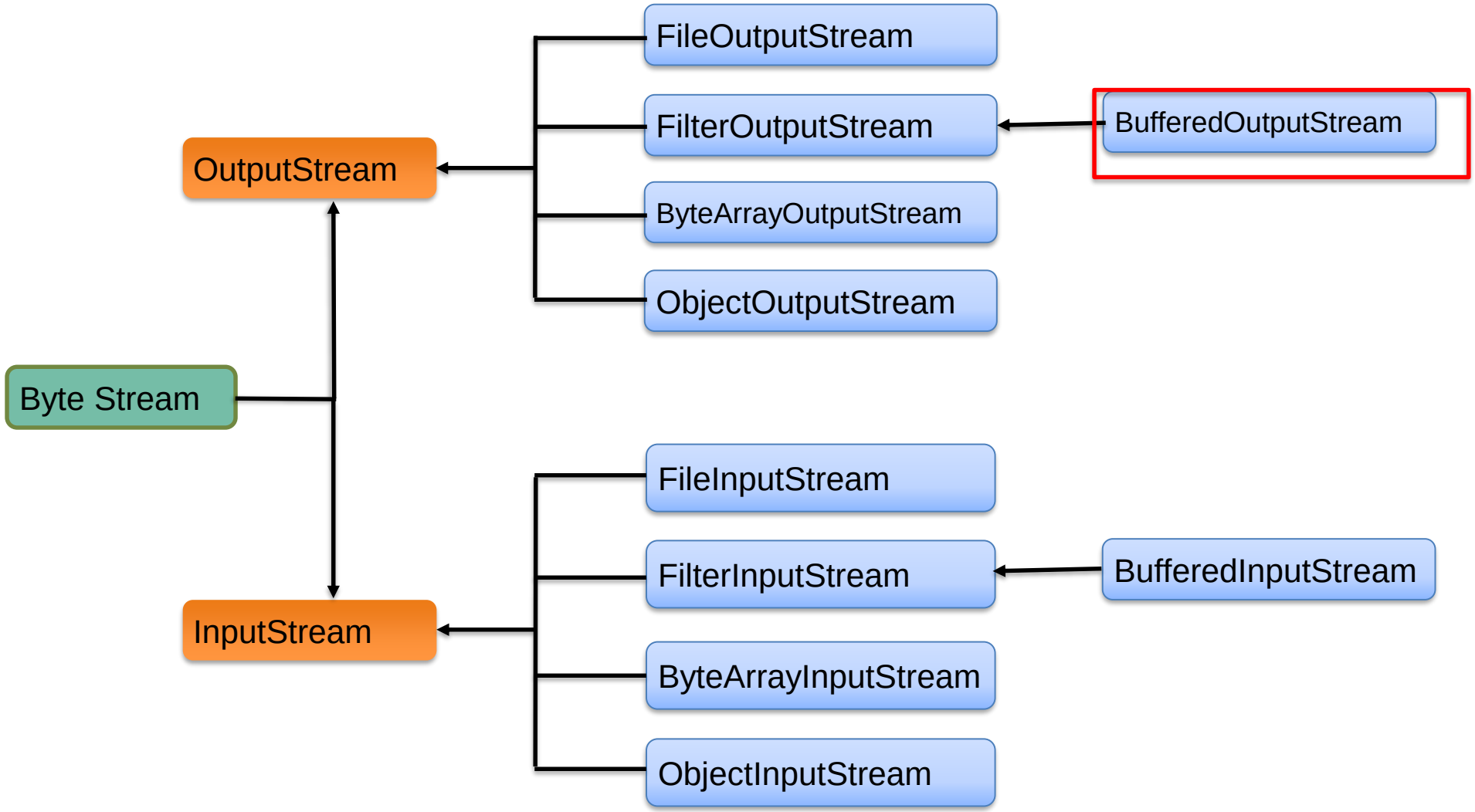
- **Examples:**

```
public class WriteFile {
    public static void main(String[] args) throws IOException {
        //attach keyboard to DataInputStream
        DataInputStream dis=new DataInputStream(System.in);

        // attach file to FileOutputStream
        FileOutputStream fout=new FileOutputStream("src/main/resources/file.txt");

        //attach FileOutputStream to BufferedOutputStream
        //BufferedOutputStream bout=new BufferedOutputStream(fout,1024);
        System.out.println("Enter text (@ at the end):");
        char ch;

        //read characters from dis into ch. Then write them into bout.
        //repeat this as long as the read character is not @
        while((ch=(char)dis.read())!='@') {
            //bout.write(ch);
            fout.write(ch);
        }
        //close the file
        fout.close();
    }
}
```





Byte Stream


FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

BufferedOutputStream Class

- **BufferedOutputStream** buffers output stream:
  - ✓ Uses internal buffer to store data.
  - ✓ More efficient than writing directly to stream.
  - ✓ Faster performance.
- **Constructors:**
  - ✓ **BufferedOutputStream**(OutputStream os): Creates buffered stream writing to given OutputStream.
  - ✓ **BufferedOutputStream**(OutputStream os, int size): Creates buffered stream with specified buffer size.

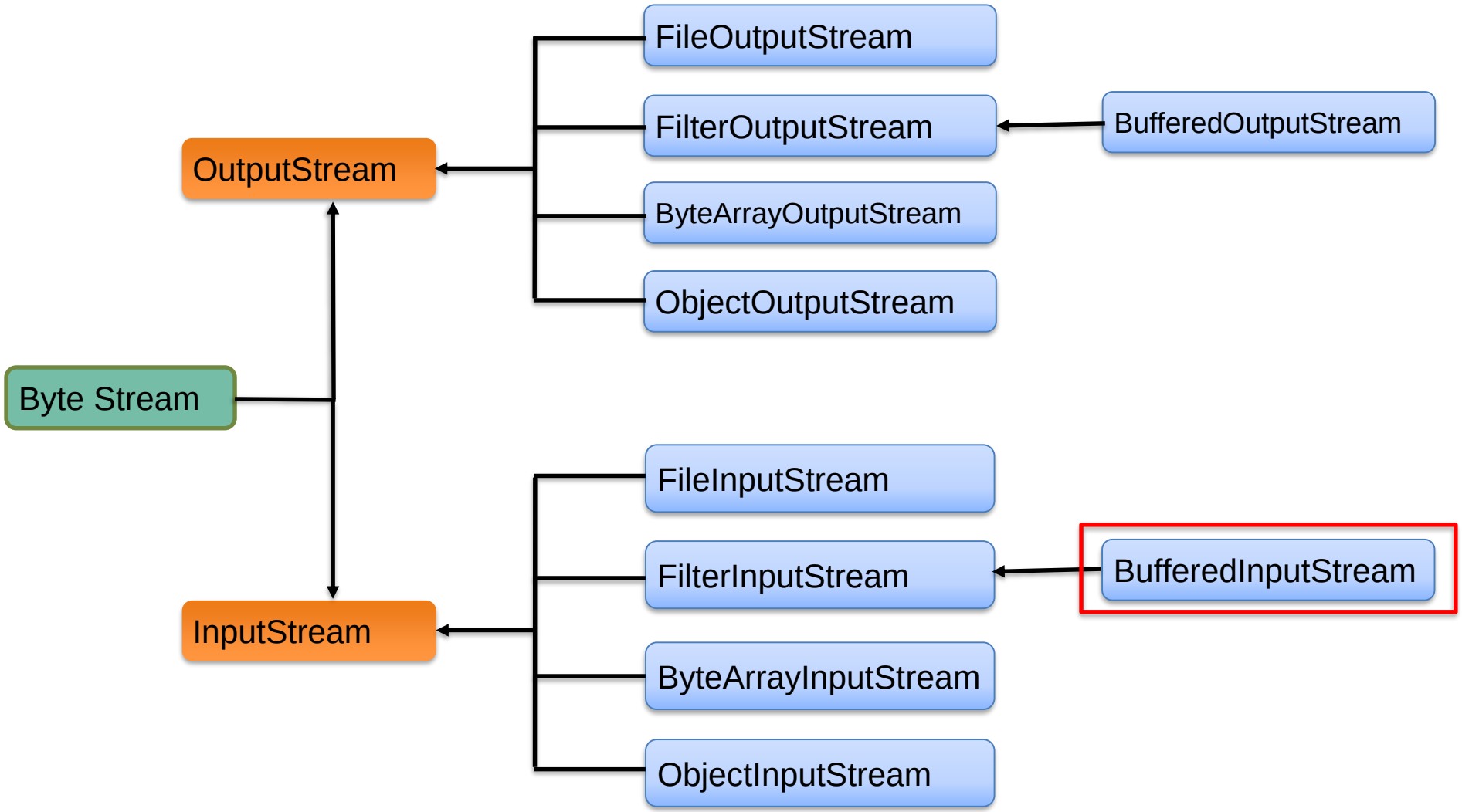
BufferedOutputStream Class

- **BufferedOutputStream** methods:
  - ✓ **write**(int b) Writes byte to buffer.
  - ✓ **write**(byte[] b, int off, int len) Writes bytes from array at offset and length to buffer.
  - ✓ **flush**() Flushes buffer contents to underlying stream.

```
FileOutputStream fout = null; BufferedOutputStream bout = null;
try {
    fout = new FileOutputStream("testout.txt");
    bout = new BufferedOutputStream(fout);
    String s = "Welcome to FPT Software Academy.";
    byte b[] = s.getBytes();
    bout.write(b); bout.flush();
    bout.close(); fout.close();
    System.out.println("Success");
} catch (FileNotFoundException e) {
    System.err.println("File testout not found!");
    e.printStackTrace();
} catch (IOException e) {
    System.err.println("Cannot write to file!");
    e.printStackTrace();
}
```

```
finally {
    if (fout != null) {
        try {
            fout.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (bout != null) {
        try {
            bout.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Byte Stream


FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

BufferedInputStream Class

- Java **BufferedInputStream** class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.
- The important points about **BufferedInputStream** are:
  - ✓ When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
  - ✓ When a BufferedInputStream is created, an internal buffer array is created.

BufferedInputStream Class

- **Constructors**:
  - ✓ **BufferedInputStream**(InputStream is): Saves given InputStream.
  - ✓ **BufferedInputStream**(InputStream is, int size): Specifies buffer size.
- **Methods**:
  - ✓ **available**(): Bytes available without blocking.
  - ✓ **read**(): Reads next byte.
  - ✓ **read**(byte[] b, int off, int len): Reads bytes into array.
  - ✓ **close**(): Closes stream, releases resources.
  - ✓ **skip**(long x): Skips x bytes.

Example 1

```
public class BufferedInputStreamExample {
    public static void main(String[] args) {
        try {
            // Step 1: Create a FileInputStream to open the file for reading.
            FileInputStream fileInputStream = new
                                FileInputStream("src/main/resources/example.txt");

            // Step 2: Create a BufferedInputStream, which wraps the FileInputStream.
            BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);

            // Step 3: Read and display the data from the BufferedInputStream.
            int data;
            while ((data = bufferedInputStream.read()) != -1) {
                System.out.print((char) data); // Convert and print the byte as a character.
            }

            // Step 4: Close the BufferedInputStream (it will also close the underlying FileInputStream).
            bufferedInputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example 2 - Unicode

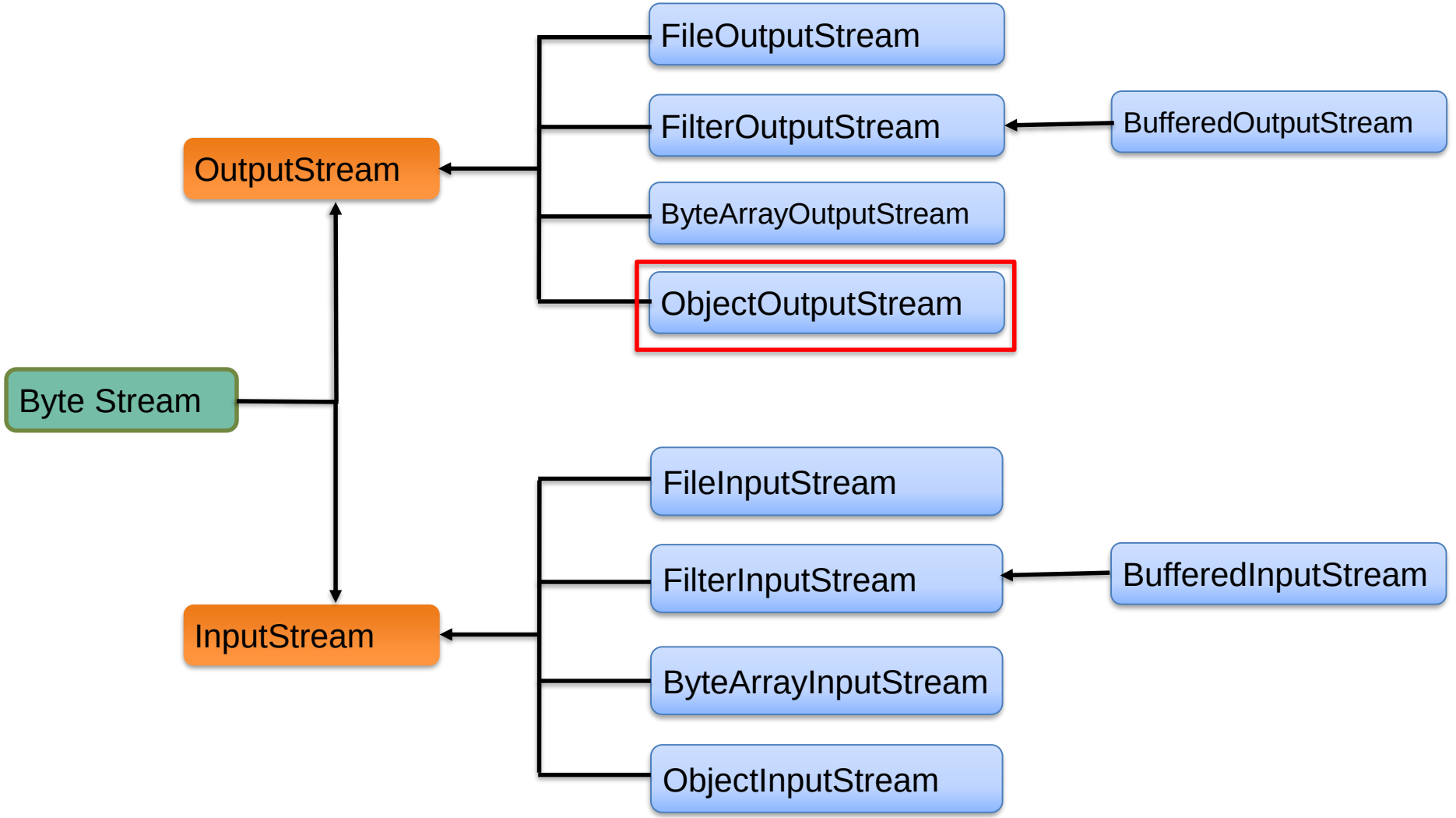
```
public class ReadUTF8File {
    public static void main(String[] args) {
        try {
            // Step 1: Create a FileReader to open the file using UTF-8 encoding.
            FileReader fileReader = new FileReader("src/main/resources/utf8file.txt", StandardCharsets.UTF_8);

            // Step 2: Create a BufferedReader, which wraps the FileReader.
            BufferedReader bufferedReader = new BufferedReader(fileReader);

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

            // Step 3: Close the BufferedReader (it will also close the underlying FileReader).
            bufferedReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

try-with-resources statement

- The **try-with-resources** statement declares one or more resources.
  - ✓ Resources must be closed after use.
  - ✓ Try-with-resources ensures each resource is closed after statement execution.
- Any **AutoCloseable/Closeable** object can be used.
- **Example**: **BufferedInputStream** must be closed after use try-with-resources closes it automatically.

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

        try (FileInputStream fin = new FileInputStream("testout.txt");
             BufferedInputStream bfin = new BufferedInputStream(fin);) {

            int data;
            while ((data = bfin.read()) != -1) {
                System.out.print((char) data);
            }

        } catch (FileNotFoundException exception) {
            System.err.println("File not found!");
        } catch (IOException exception) {
            System.err.println("Cannot read from file!");
        }
    }
}
```

Byte Stream

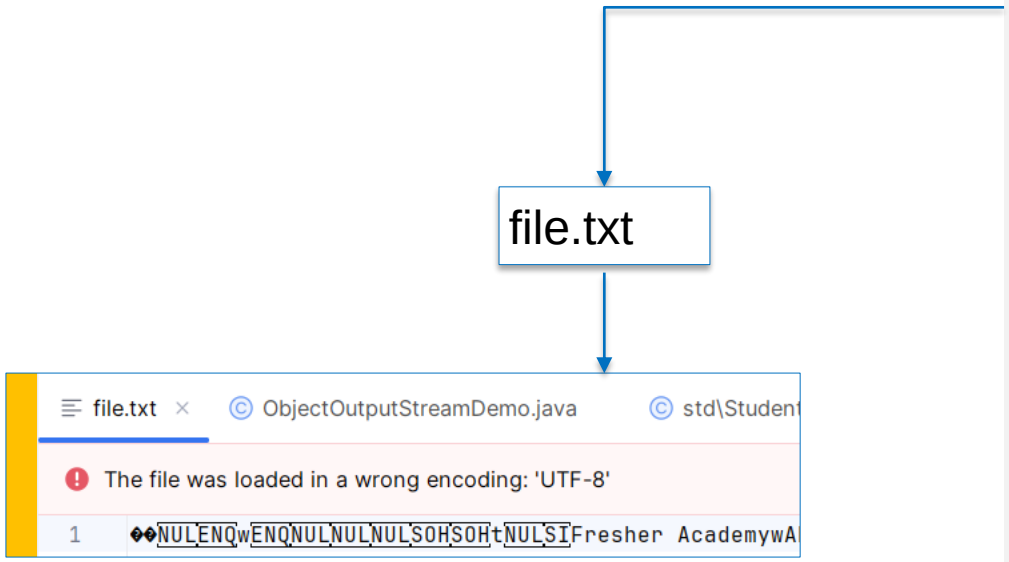

FileOutputStream
FilterOutputStream
BufferedOutputStream
OutputStream
ByteArrayOutputStream
ObjectOutputStream
Byte Stream
FileInputStream
FilterInputStream
BufferedInputStream
InputStream
ByteArrayInputStream
ObjectInputStream

ObjectOutputStream class

- **ObjectOutputStream** *writes objects and primitives to OutputStream*.
  - ✓ Only Serializable objects can be written.
  - ✓ Often used with **ObjectInputStream** to read back objects.
  - ✓ **ObjectOutputStream** writes objects, **ObjectInputStream** reads them.
- **Constructors:**
  - ✓ protected **ObjectOutputStream**(): For subclasses to avoid allocating private data.
  - ✓ **ObjectOutputStream**(OutputStream out): Writes to given OutputStream.
- **Exception:**
  - ✓ *InvalidClassException* − Something is wrong with a class used by serialization.
  - ✓ *NotSerializableException* − Some object to be serialized does not implement the `java.io.Serializable` interface.
  - ✓ *IOException* − Any exception thrown by the underlying OutputStream.

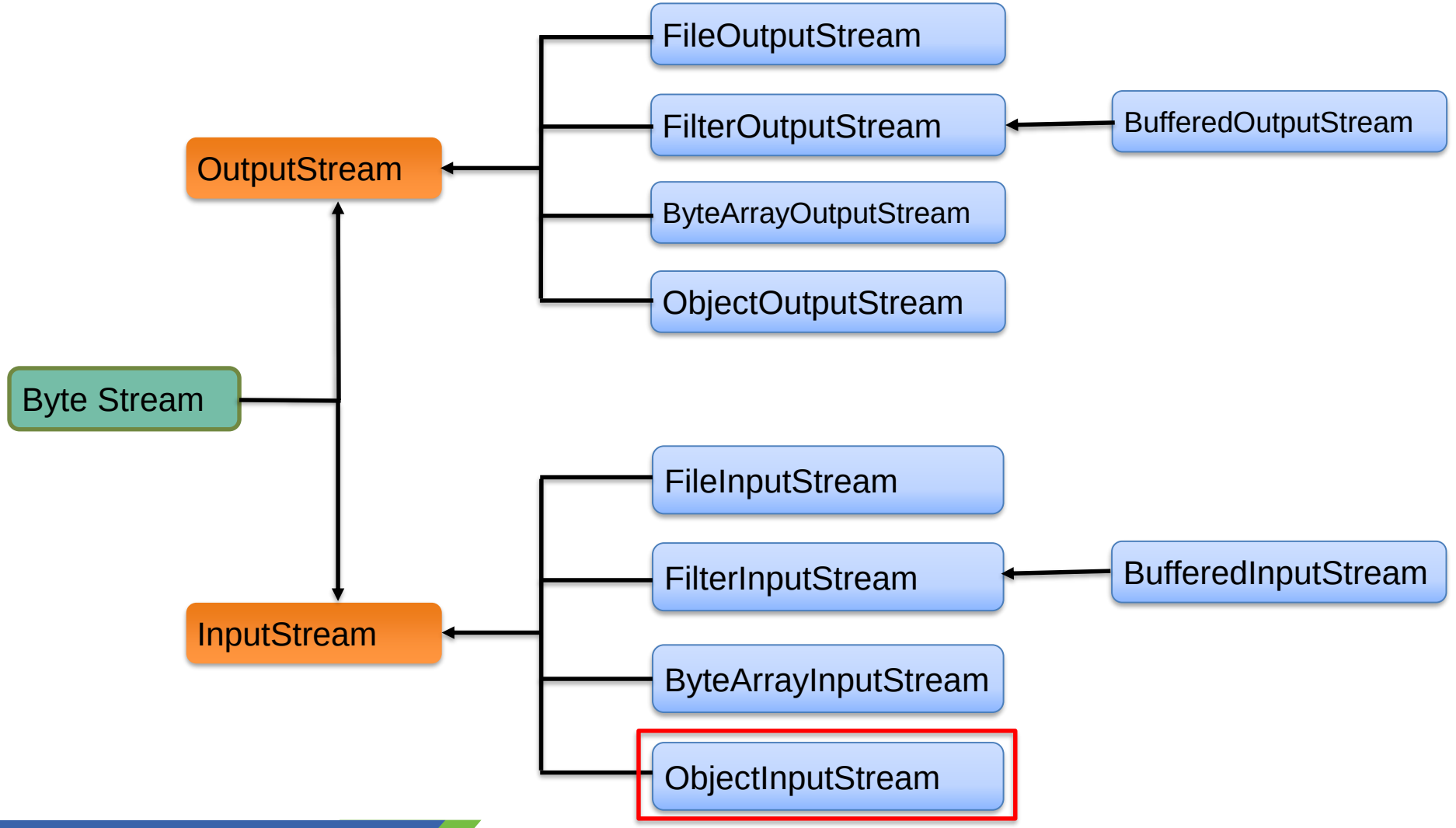
ObjectOutputStream class

- **Important methods:**
  - ✓ void **writeObject**(Object obj): Write the specified object to the ObjectOutputStream.




```
public static void main(String[] args) throws IOException,
ClassNotFoundException {
    FileOutputStream fout = new FileOutputStream("file.txt");
    ObjectOutputStream oot = new ObjectOutputStream(fout);
    String a = "Fresher Academy“, b = "Fresher";
    byte[] be = { 'A', 'B', 'C' };
    // illustrating writeInt(int i)
    oot.writeInt(1);
    // illustrating writeBoolean(boolean a)
    oot.writeBoolean(true);
    // illustrating writeObject(Object x)
    oot.writeObject(a);
    // illustrating writeBytes(String b)
    oot.writeBytes(b);
    // illustrating writeDouble(double d)
    oot.writeDouble(2.3);
    // illustrating writeUTF(String str)
    oot.writeUTF(a);
    // illustrating writeChars(String a)
    oot.writeChars(a);
    // illustrating write(byte[] buff)
    oot.write(be);
    // flushing the stream
    oot.flush(); oot.close();
}
```

Byte Streams



ObjectInputStream class

- **ObjectInputStream** deserializes objects and primitives previously written by ObjectOutputStream.
  - ✓ Used to recover serialized objects. Ensures object types match JVM classes.
  - ✓ Classes loaded as needed using standard mechanisms.

```
public static void main(String[] args)throws IOException,
                ClassNotFoundException {
    byte[] be = { 'A', 'B', 'C' };
    byte c[] = new byte[4];

    FileInputStream fin = new FileInputStream("file.txt");
    ObjectInputStream oit = new ObjectInputStream(fin);

    System.out.println(oit.readInt());
    System.out.println(oit.readBoolean());
    System.out.println(oit.readObject());
    oit.read(c);

    for (int i = 0; i < 4; i++) {
        System.out.print((char) c[i]);
    }

    System.out.println();
    System.out.println(oit.readDouble());
    for (int i = 0; i < 13; i++) {
        System.out.print(oit.readChar());
    }
    System.out.println();
    System.out.println(oit.readShort());
    oit.readFully(be);

    for (int i = 0; i < 3; i++) {
        System.out.print((char) be[i]);
    }
    oit.close();
}
```

Example: write/read a user-defined class

- To serialize an object, call **writeObject**() on **ObjectOutputStream**.
- To deserialize, call **readObject**() on **ObjectInputStream**.

```
public class Student implements Serializable {
    private String ssn;
    private String firstName;
    private String lastName;
    private char mi;
    private LocalDate birthDate;
    private String street;
    private String phone;
    private String zipCode;

    public Student() {

    }
```

```
    public Student (String ssn, String firstName, char mi, String lastName,
                    LocalDate birthDate, String street, String phone, String zipCode) {
        super();
        this.ssn = ssn;
        this.firstName = firstName;
        this.mi = mi;
        this.lastName = lastName;
        this.birthDate = birthDate;
        this.street = street;
        this.phone = phone;
        this.zipCode = zipCode;
    }
     //getter, setter methods
}
```

The object that we want to serialize should implement java.io.Serializable interface.
***Serializable** is just a marker interface and doesn't have any abstract method that we have to implement. We will get `java.io.NotSerializableException` if the class doesn't implement Serializable interface.*

Example: write/read a user-defined class

- Create a class **StudentService**:

```
public class StudentService {

    public boolean write(Student student)
            throws FileNotFoundException, IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.txt"))) {
            oos.writeObject(student);
        }

        return true;
    }

    public Student read() throws FileNotFoundException, IOException, ClassNotFoundException {

        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.txt"))) {
            Student student = (Student) ois.readObject(); // Upcasting
            return student;
        }
    }
}
```

Example: write/read a user-defined class

- **Create a main method:**

```
public static void main(String[] args) {
    LocalDate birthDate = LocalDate.of(1999, 1, 1);

    Student s1 = new Student("12345", "Nguyen Manh", 'K', "Truong", birthDate, "Duytan", "0987654321", "084");

    StudentService studentServices = new StudentService();
    try {
        studentServices.write(s1);
        System.out.println("Complete write a student to file!");

        System.out.println("Reading file:");
        System.out.println(studentServices.read());

    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

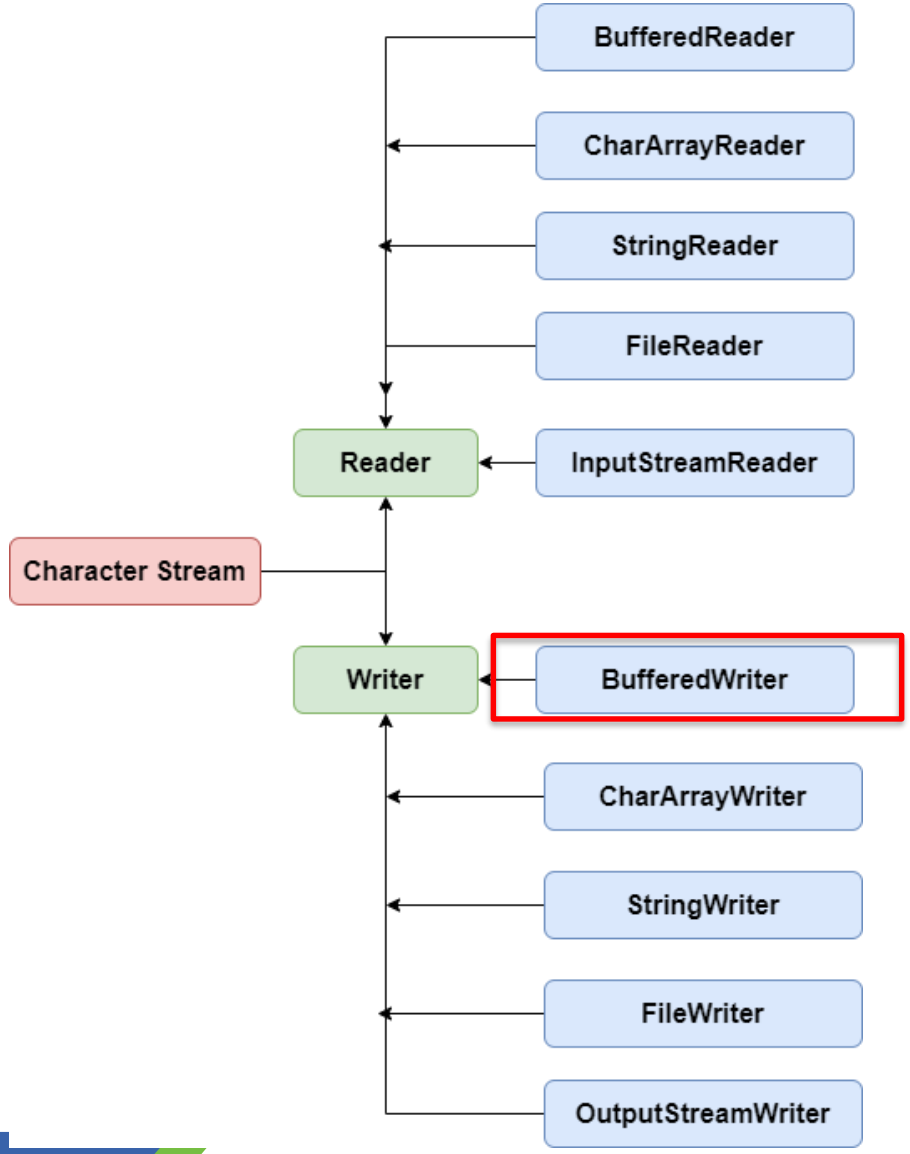
- **Output:**

```
Complete write a student to file!
Reading file:
Student [ssn=12345, firstName=Nguyen Manh, mi=K, lastName=Truong, birthDate=1999-01-01,
street=Duytan, phone=0987654321, zipCode=084]
```

Section 3

Character Stream

Character Stream



BufferedWriter Class

- **BufferedWriter** class: is used to provide buffering for Writer instances
  - ✓ Provides buffering for faster performance.
  - ✓ Inherits from Writer.
  - ✓ The buffering characters are used for providing the efficient writing of *single arrays*, *characters*, and *strings*.
- **Constructors**:
  - ✓ **BufferedWriter**(Writer wrt): to create a buffered character output stream that uses the default size for an output buffer.
  - ✓ **BufferedWriter**(Writer wrt, int size): Specifies buffer size.

BufferedWriter Class

- **Methods**:

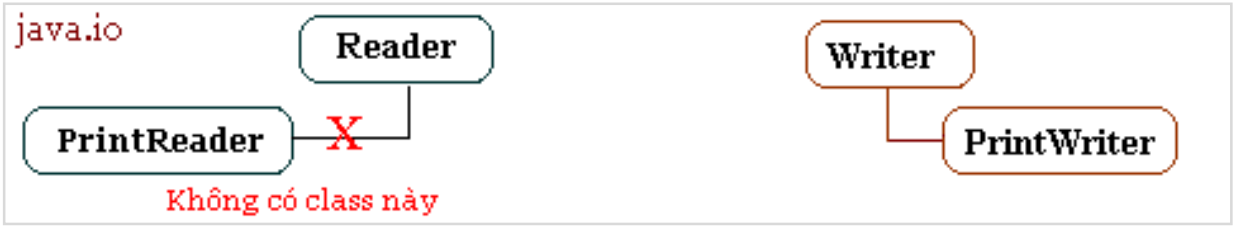
Method	Description
void newLine()	It is used to add a new line by writing a line separator.
void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

BufferedWriter Example

```
public class WriteFile {
   public static void main(String[] args) throws IOException {
      String[] list = { "one", "two", "three", "fo" };
      BufferedWriter bufferedWrite = null;

      try {
         File file = new File("src/main/resources/file.txt");
         FileWriter fileWriter = new FileWriter(file); // A stream that connects to the text file
         bufferedWrite = new BufferedWriter(fileWriter); // Connect the FileWriter to the BufferedWriter

         for (String s : list) {
            bufferedWrite.write(s + "\n");
         }
         System.out.println("Write done!");

      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         if (bufferedWrite != null) {
           bufferedWrite.close();
      }
      }
   }
}
```

PrintWriter class

- Java **PrintWriter** prints formatted representations of objects to text output stream.
- Implements Writer.



- **Constructors:**
  - ✓ **PrintWriter**(Writer out) Appending
  - ✓ **PrintWriter**(Writer out, boolean autoFlush)
  - ✓ **PrintWriter**(OutputStream out)
  - ✓ **PrintWriter**(OutputStream out, boolean autoFlush)
  - ✓ **PrintWriter**(String fileName) No appending

PrintWriter class

- **To open a text file for output: connect it to PrintWriter stream:**

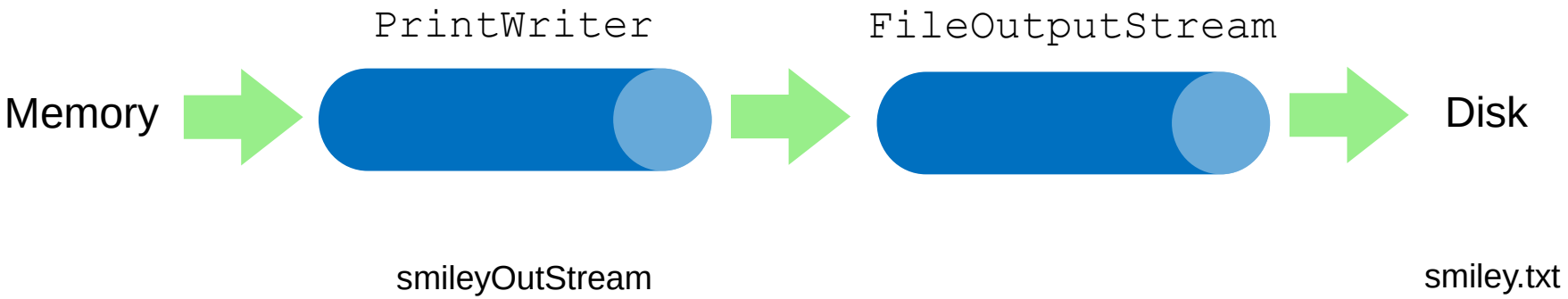
```
PrintWriter outputStream = new PrintWriter(new FileOutputStream("out.txt"));
```

- **Same as:**

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: is to create **PrintWriter** using **FileOutputStream** to open text file.
- **FileOutputStream** connects **PrintWriter** to the file.

Output File Streams




```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream("smiley.txt") );
```

Java Tip: Appending to a Text File

- To append instead of overwrite, use FileOutputStream constructor.

```
outputStream = new PrintWriter(new FileOutputStream("out.txt", true));

System.out.println("A for append or N for new file:");
char ans = Scanner.next().charAt(0);

boolean append = (ans == 'A' || ans == 'a');
outputStream = new PrintWriter(new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- Close output file after writing.
- Close input file after reading.
- Use **close**() on **PrintWriter**, **BufferedReader**.
  - ✓Example: outputStream.close();
- Files are closed automatically when program ends normally.

BufferReader class

- To open a text file for input: connect it to BufferedReader stream
  - ✓Goal is a **BufferedReader**: Using **FileReader** to open the text file
  - ✓**FileReader** connects **BufferedReader** to the file

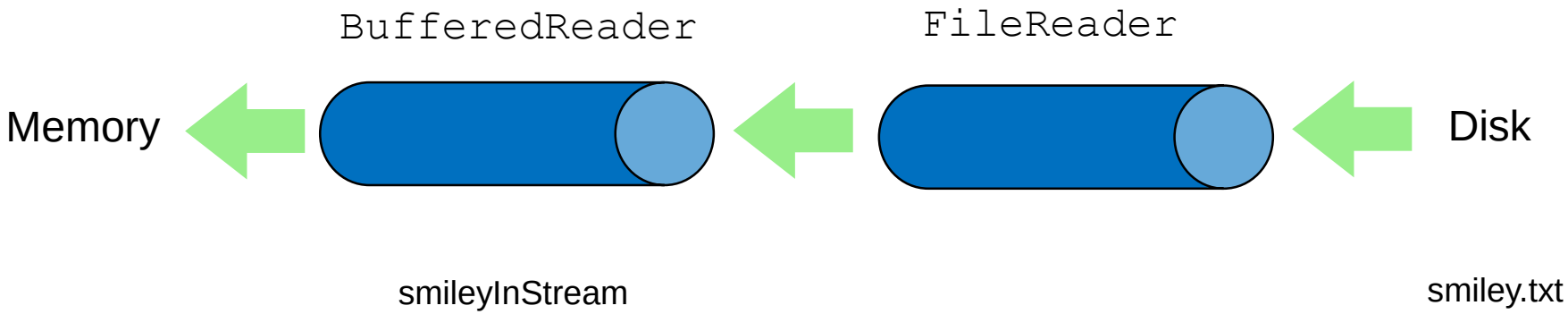
- **Example:**

```
BufferedReader smileyInStream =  new BufferedReader(new FileReader("smiley.txt"));
```

Same as:

```
FileReader s = new FileReader("smiley.txt");
BufferedReader smileyInStream = new BufferedReader(s);
```

Input File Streams




```
BufferedReader smileyInStream = new BufferedReader( new FileReader("smiley.txt") );
```

BufferReader Example

```java
public class ReadFile {
   public static void main(String[] args) throws IOException {
      BufferedReader bufferedReader = null;
      try {
         File file = new File("src/main/resources/file.txt");

         // Connect InputStreamReader to a BufferedReader
         bufferedReader = new BufferedReader(new FileReader(file)); String line = null;

         while ((line = bufferedReader.readLine())!=null) {
           System.out.println(line);
         }

      } catch (Exception e) {
         e.printStackTrace();
      }

      finally {
         if(bufferedReader!=null) {
            bufferedReader.close();
         }
      }
   }
}
```

Output:

```
one
two
three
fo
```

Summary

- I/O Tutorial
- Binary Stream
- Character stream

References

- *https://docs.oracle.com/javase/tutorial/essential/io/*
- *https://www.javatpoint.com/java-io*
- *https://www.digitalocean.com/community/tutorials/objectoutputstream-java-write-object-file*

THANK YOU!

Any questions?