Object Oriented Programming with Java

Zheng Chen

Input & Output

1. Java Stream
2. Java File
3. Pipe
4. Serialization

I/O

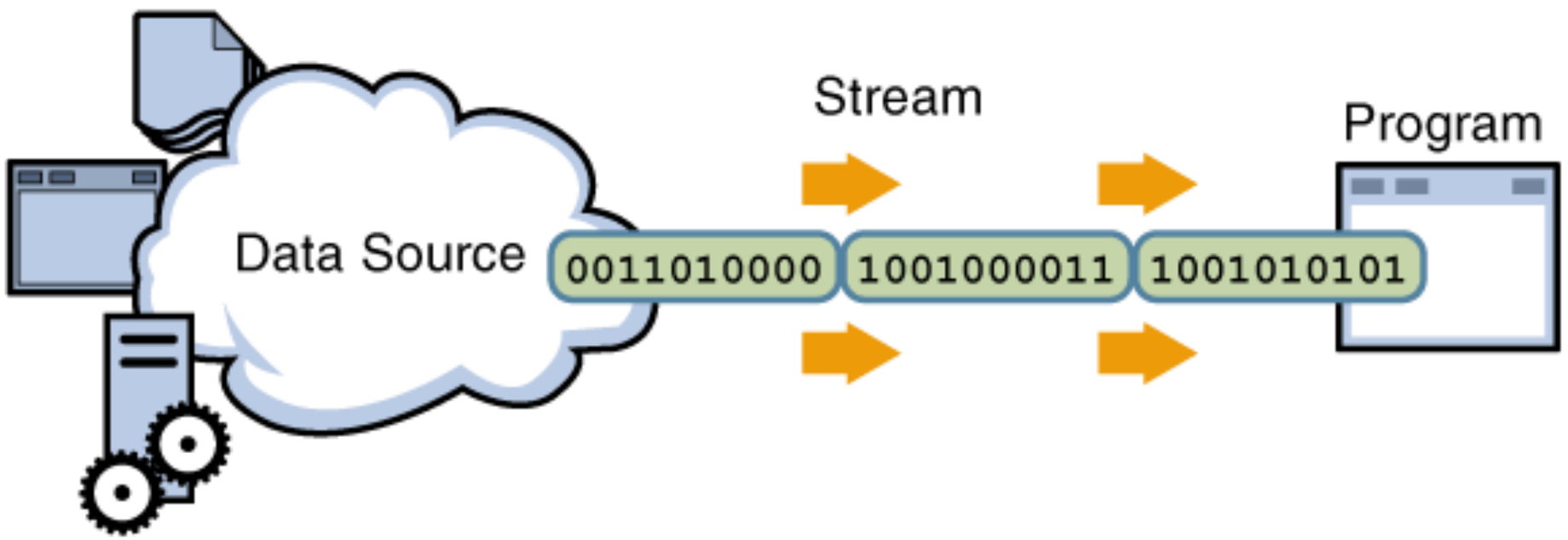
- I/O (Input and Output) is used to process the input and produce the output.

- Everything besides information process is I/O, including file, networking, devices, etc.

I/O in Java

- Java uses the concept of a **stream** to to process the input and produce the output.

- A stream is a sequence of data. It's called a stream because it is like a stream of water that continues to flow.



Standard streams

- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
  - System.out: standard output stream

```
System.out.println("simple message");
```

  - System.in: standard input stream

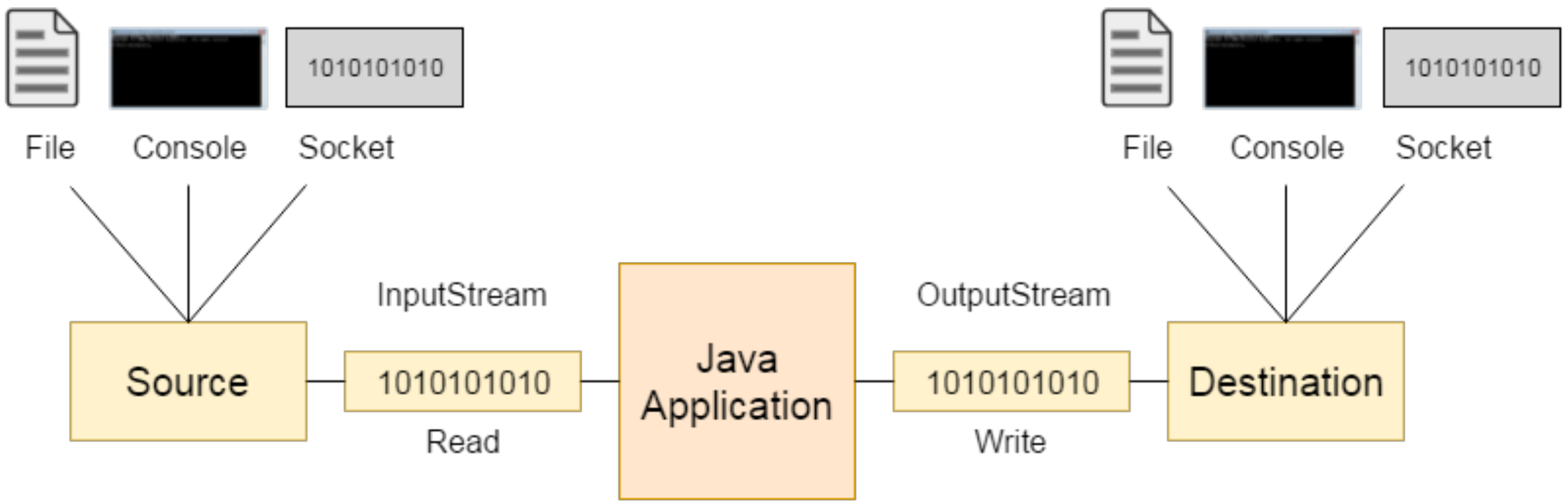
```
int i=System.in.read();
```

  - System.err: standard error stream

```
System.err.println("error message");
```

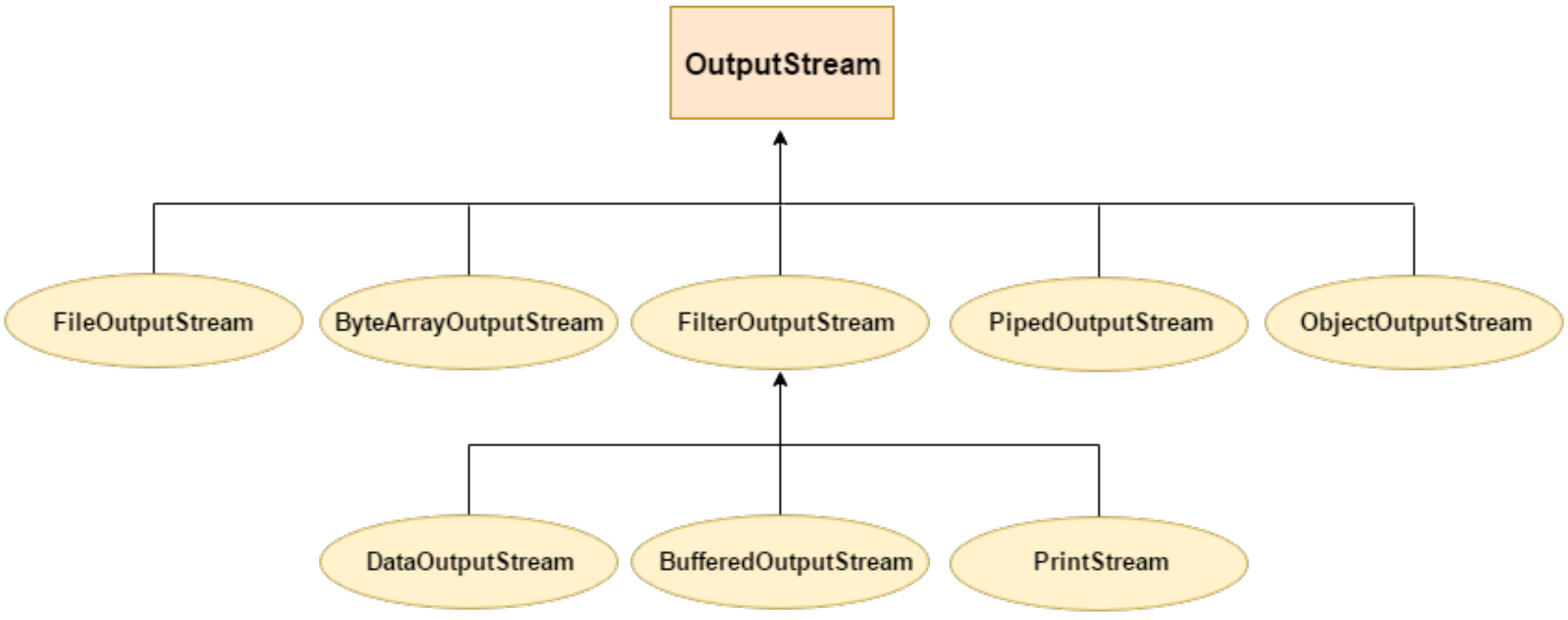
OutputStream and InputStream

- A stream can be defined as a sequence of data. There are two kinds of Streams
  - InputStream: The InputStream is used to read data from a source.
  - OutputStream: The OutputStream is used for writing data to a destination.



OutputStream class

- OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

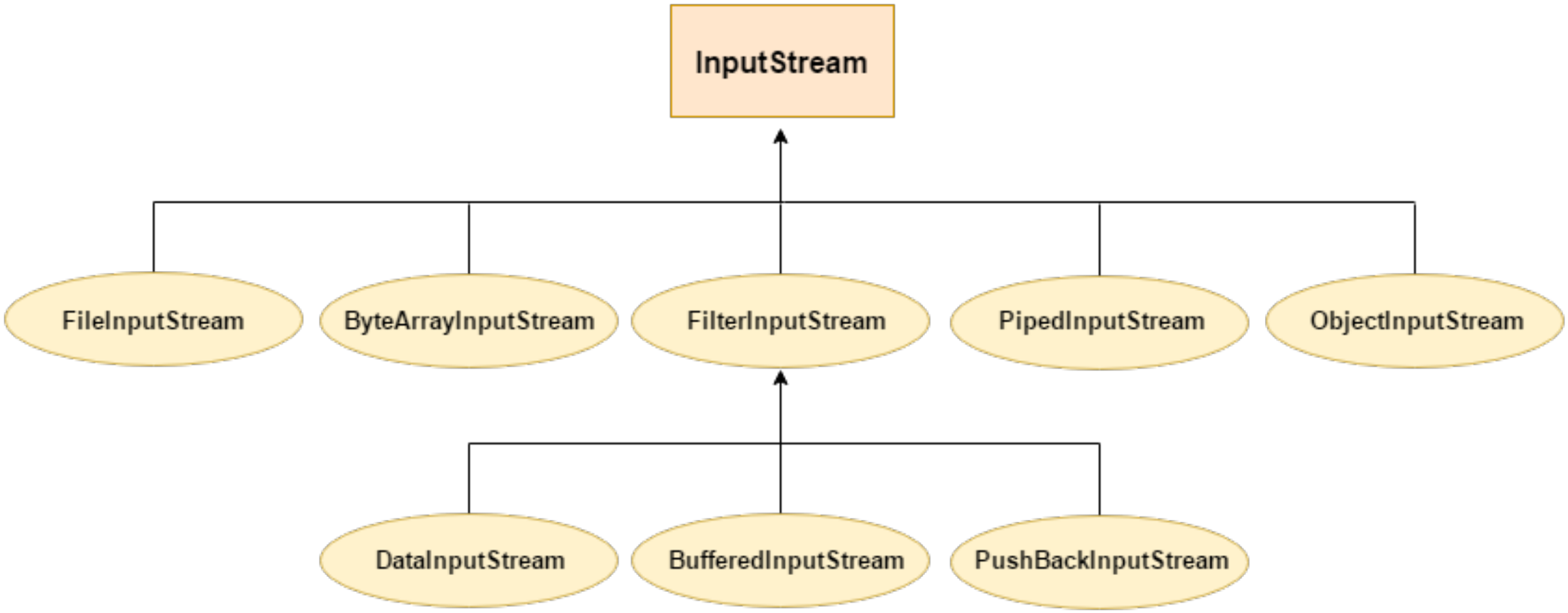


Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

InputStream class

- InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.



Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

FileOutputStream

- Java FileOutputStream is an output stream used for writing data to a file.

```
import java.io.*;

public class FileOutputStreamExample {
  public static void main(String args[]) {
    try {
      OutputStream fout = new FileOutputStream(
                  "testout.txt");
      fout.write(65); //'A'==65
      fout.close();
      System.out.println("success...");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Example: write string to a file

```
import java.io.*;

public class FileOutputStreamExample {
  public static void main(String args[]) {
    try {
      OutputStream fout = new FileOutputStream(
          "testout.txt");
      String s = "Welcome to earth.";
      byte b[] = s.getBytes();
      //converting string into byte array
      fout.write(b);
      fout.close();
      System.out.println("success...");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

FileInputStream

- Java FileInputStream class obtains input bytes from a file.

```
import java.io.*;

public class InputStreamExample {
  public static void main(String args[]) {
    try {
      InputStream fin = new FileInputStream(
            "testout.txt");
      int i = fin.read();
      System.out.print((char) i);
      fin.close();
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Example: read all characters

```
import java.io.*;

public class InputStreamExample {
  public static void main(String args[]) {
    try {
      InputStream fin = new FileInputStream(
          "testout.txt");
      int i = 0;
      while ((i = fin.read()) != -1) {
        System.out.print((char) i);
      }
      fin.close();
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Performance problem with FileOutputStream

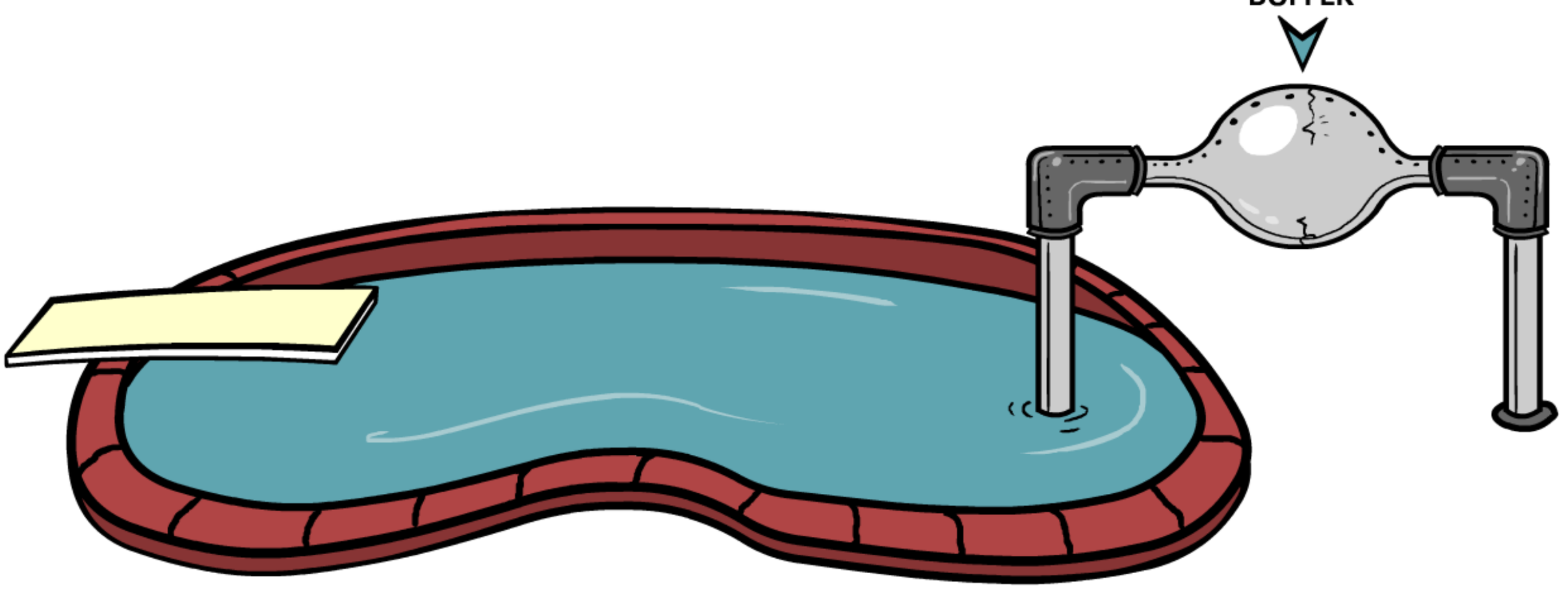
```
import java.io.*;

public class FileOutputStreamExample {
  public static void main(String args[]) throws Exception {
    OutputStream fout = new FileOutputStream(
        “testout.txt");
    long timeS = System.currentTimeMillis();
    for (int i = 0; i < 1024 * 1024; i++) {
      fout.write(65); //'A'==65
    }
    fout.close();
    System.out.println(
      "success within " + (System.currentTimeMillis()
            - timeS) + "ms."
    );
  }
}
```

>>>success within 5259ms.

BufferedOutputStream

- BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.



Performance with BufferedOutputStream

```
import java.io.*;
public class BufferedOutputStreamExample {
  public static void main(String args[]) throws Exception {
    OutputStream fout = new FileOutputStream("testout.txt");
    OutputStream bout = new BufferedOutputStream(fout);
    long timeS = System.currentTimeMillis();
    for (int i = 0; i < 1024 * 1024; i++) {
      bout.write(65); //'A'==65
    }
    bout.flush();
    bout.close();
    fout.close();
    System.out.println("success within "
            + (System.currentTimeMillis() - timeS) + "ms."
    );
  }
}
```

>>>success within 32ms.

BufferedInputStream

- Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

```
import java.io.*;

public class BufferedInputStreamExample {
  public static void main(String args[]) throws Exception {
    InputStream fin = new FileInputStream("testout.txt");
    InputStream bin = new BufferedInputStream(fin);
    int i;
    while ((i = bin.read()) != -1) {
      System.out.print((char) i);
    }
    bin.close();
    fin.close();
  }
}
```

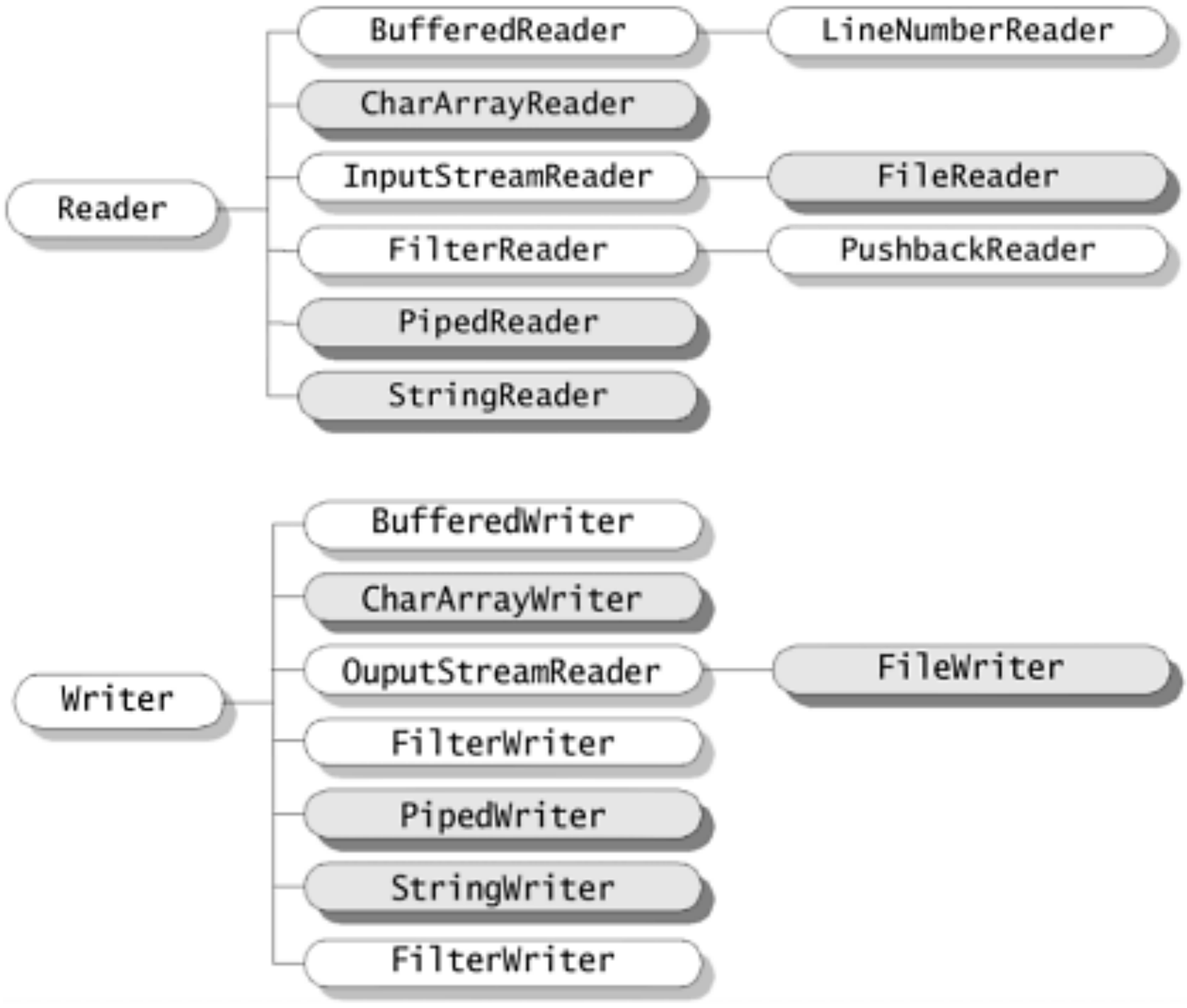
Character Streams

- Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode.

- **InputStream** and **OutputStream** is the abstract super class of all **Byte streams**.

- **Reader** and **Writer** is the is the abstract super class of all **Character streams**.

Reader and Writer class hierarchy



Useful methods of Writer

Type	Method	Description
Writer	append(char c)	It appends the specified character to this writer.
abstract void	close()	It closes the stream, flushing it first.
abstract void	flush()	It flushes the stream.
void	write(char[] cbuf)	It writes an array of characters.
void	write(int c)	It writes a single character.
void	write(String str)	It writes a string.

Useful methods of Reader

Type	Method	Description
abstract void	close()	It closes the stream and releases any system resources associated with it.
void	mark(int readAheadLimit)	It marks the present position in the stream.
boolean	markSupported()	It tells whether this stream supports the mark() operation.
int	read()	It reads a single character.
int	read(char[] cbuf)	It reads characters into an array.
boolean	ready()	It tells whether this stream is ready to be read.
void	reset()	It resets the stream.
long	skip(long n)	It skips characters.

Example of FileWriter

```
import java.io.*;

public class WriterExample {
  public static void main(String[] args)
                throws Exception {
    Writer w = new FileWriter("output.txt");
    String content = "可以写中文哦~";
    w.write(content);
    w.close();
    System.out.println("Done");
  }
}
```

Example of FileReader

```
import java.io.*;

public class ReaderExample {
  public static void main(String[] args)
              throws Exception {
    Reader reader = new FileReader("output.txt");
    int data = reader.read();
    while (data != -1) {
      System.out.print((char) data);
      data = reader.read();
    }
    reader.close();
  }
}
```

Input & Output

1. Java Stream
2. **Java File**
3. Pipe
4. Serialization

Java File

- The File class is an abstract representation of **file and directory** pathname. A pathname can be either absolute or relative.

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.

Useful Methods in File class

Type	Method	Description
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.

Useful Methods in File class

Type	Method	Description
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Example — create new file

```
import java.io.*;

public class FileDemo {
  public static void main(String[] args)
              throws IOException {
    File file = new File("javaFile123.txt");
    if (file.createNewFile()) {
      System.out.println("New File is created!");
    } else {
      System.out.println("File already exists.");
    }
  }
}
```

Example — get file info

```
import java.io.*;

public class FileDemo2 {
  public static void main(String[] args) {
    String path = "";
    boolean bool = false;
    File file = new File("javaFile123.txt");
    System.out.println(file);
    System.out.println(file.isFile());
    System.out.println(file.isDirectory());
    System.out.println(file.canRead());
    System.out.println(file.canWrite());
    System.out.println(file.getName());
    System.out.println(file.getParent());
    System.out.println(file.getPath());
    System.out.println(file.getAbsolutePath());
    System.out.println(file.length());
  }
}
```

Example — listing directories

```
import java.io.File;

public class ReadDir {
  public static void main(String[] args) {
    File file = null;
    File[] paths;
    file = new File("/tmp");
    paths = file.listFiles();
    for (File f : paths) {
      System.out.println(f);
      if (f.isFile()) {
        System.out.println(f.length());
      } else {
        System.out.println(f.listFiles().length);
      }
    }
  }
}
```

RandomAccessFile

- This class is used for reading and writing to random access file.

Constructor	**Description**
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Useful Methods in RandomAccessFile class

Type	Method	Method
void	close()	It closes this random access file stream and releases any system resources associated with the stream.
int	readInt()	It reads a signed 32-bit integer from this file.
String	readUTF()	It reads in a string from this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

Example

```
import java.io.*;

public class RandomAccessFileExample {

  static final String FILEPATH = "myFile.TXT";

  public static void main(String[] args) throws IOException {
    System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
    writeToFile(FILEPATH, "I love my country and my people", 31);
  }

  private static byte[] readFromFile(String filePath, int position, int size) throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "r");
    file.seek(position);
    byte[] bytes = new byte[size];
    file.read(bytes);
    file.close();
    return bytes;
  }

  private static void writeToFile(String filePath, String data, int position)
    throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "rw");
    file.seek(position);
    file.write(data.getBytes());
    file.close();
  }
}
```

Input & Output

1. Java Stream
2. Java File
3. Pipe
4. Serialization

Pipe

- A pipe connects an input stream and an output stream.

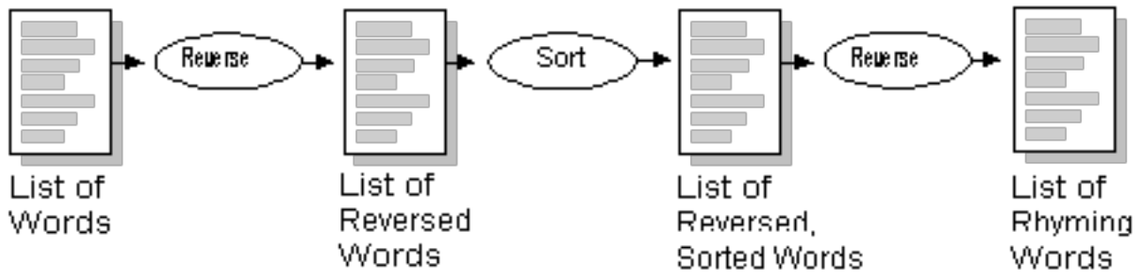
- A piped I/O is based on the producer-consumer pattern, where the producer produces data and the consumer consumes the data.

- In a piped I/O, we create two streams representing two ends of the pipe. A **PipedOutputStream** object represents one end and a **PipedInputStream** object represents the other end. We connect the two ends using the **connect()** method on the either object.

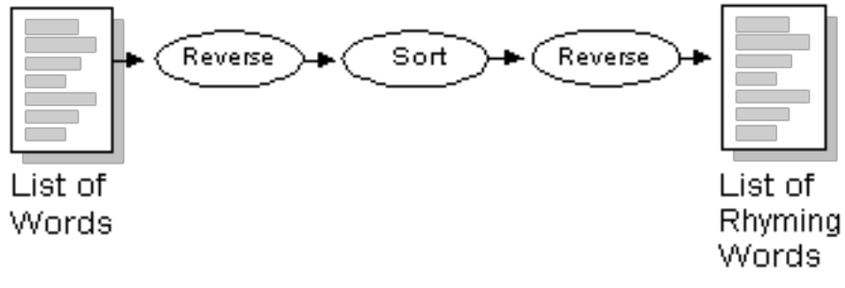
When to use Pipe

- Consider a class that implements various string manipulation utilities, For example, you could reverse each word in a list, sort the words, and then reverse each word again to create a list of rhyming words.

- Without pipe streams, the program would have to store the results somewhere (such as in a file or in memory) between each step:



- With pipe streams, the output from one method could be piped into the next:



How to use Pipe

- There’re two ways to create and connect the two ends of a pipe:
  - The first method creates a piped input and output streams and connect them. It connects the two streams using connect method.

```
PipedInputStream pis  = new PipedInputStream();
PipedOutputStream pos  = new PipedOutputStream();
pis.connect(pos); /* Connect  the   two  ends  */
```

  - The second method creates piped input and output streams and connect them. It connects the two streams by passing the input piped stream to the output stream constructor.

```
PipedInputStream pis  = new PipedInputStream();
PipedOutputStream pos  = new PipedOutputStream(pis);
```

Example

```
import java.io.*;

public class PipeExample {

    public static void main(String[] args)
            throws Exception {
        PipedInputStream pis =
                new PipedInputStream();
        PipedOutputStream pos =
                new PipedOutputStream(pis);

        Runnable producer =
                () -> produceData(pos);
        Runnable consumer =
                () -> consumeData(pis);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
```

```
    public static void produceData(PipedOutputStream pos) {
        try {
            for (int i = 1; i <= 50; i++) {
                pos.write((byte) i);
                pos.flush();
                System.out.println("Writing: " + i);
                Thread.sleep(500);
            }
            pos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void consumeData(PipedInputStream pis) {
        try {
            int num = -1;
            while ((num = pis.read()) != -1) {
                System.out.println("\tReading: " + num);
            }
            pis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Input & Output

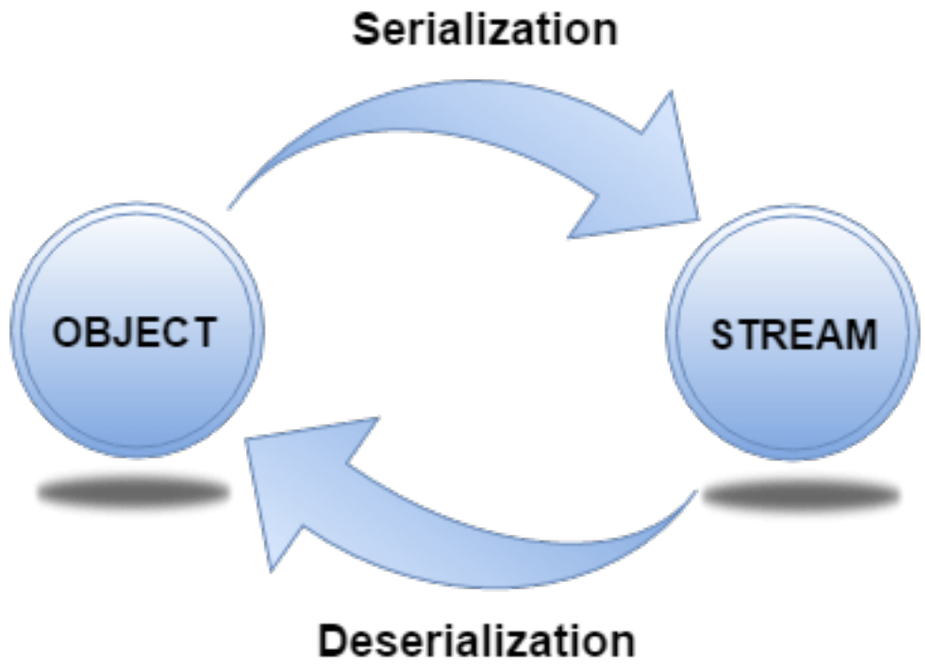
1. Java Stream
2. Java File
3. Pipe
4. **Serialization**

Serialization and Deserialization

- Serialization in Java is a mechanism of writing the state of an object into a byte-stream.

- The reverse operation of serialization is called deserialization where byte-stream is converted into an object.

- The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.



How to Serialization

- For serializing the object, we call the **writeObject()** method of **ObjectOutputStream**, and for deserialization we call the **readObject()** method of **ObjectInputStream** class.

- We must have to implement the **Serializable** interface for serializing the object.

- Serializable is a **marker interface** (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.

Important Methods

Methods of ObjectOutputStream	Description
public final void writeObject(Object obj)	writes the specified object to the ObjectOutputStream.
public void flush()	flushes the current output stream.
public void close()	closes the current output stream.

Method of ObjectInputStream	Description
public final Object readObject()	reads an object from the input stream.
public void close()	closes ObjectInputStream.

Example

```
import java.io.Serializable;

public class Student implements Serializable {
   int id;
   String name;

   public Student(int id, String name) {
      this.id = id;
      this.name = name;
   }
}
```

Example

```
import java.io.*;

class Persist {

  public static void main(String args[]) throws Exception {
    //Creating the object
    Student s1 = new Student(211, "ravi");
    //Creating stream and writing the object
    FileOutputStream fout = new FileOutputStream("f.txt");
    ObjectOutputStream out = new ObjectOutputStream(fout);
    out.writeObject(s1);
    out.flush();
    //closing the stream
    out.close();
    System.out.println("success");
  }
}
```

Example

```
import java.io.*;

class Depersist {

  public static void main(String args[]) throws Exception {
    //Creating stream to read the object
    ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("f.txt"));
    Student s = (Student) in.readObject();
    //printing the data of the serialized object
    System.out.println(s.id + " " + s.name);
    //closing the stream
    in.close();
  }
}
```

Homework

1. Load a txt file,
2. Count each character in that file,
3. Write the result into another file.