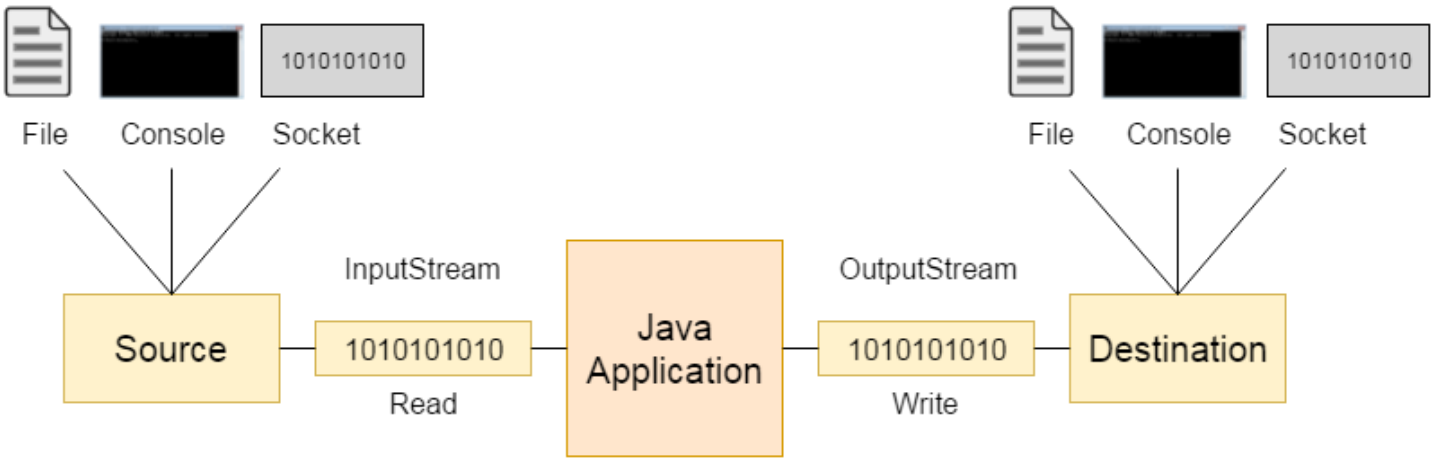
Java I/O

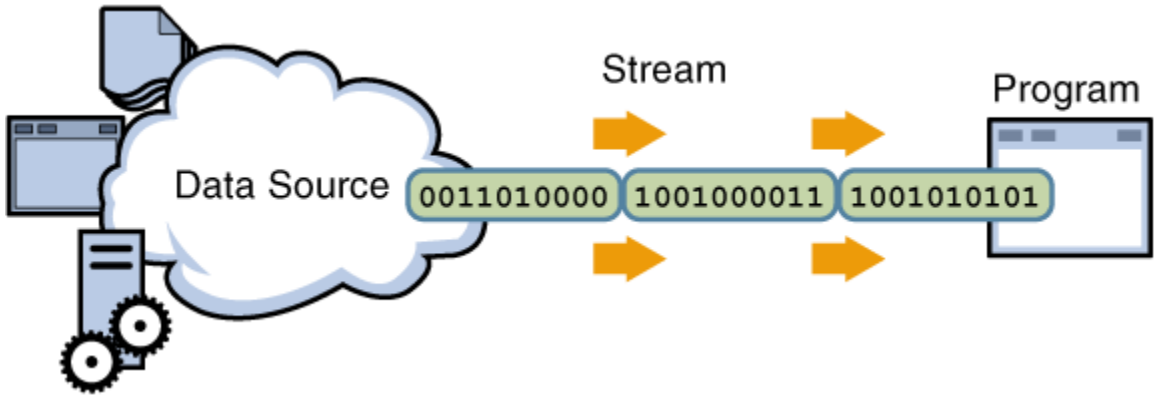
- Java I/O (Input and Output) is used to process the input and produce the output.

- Java uses the concept of a stream to make I/O operation fast. The **java.io** package contains all the classes required for input and output operations.



Stream

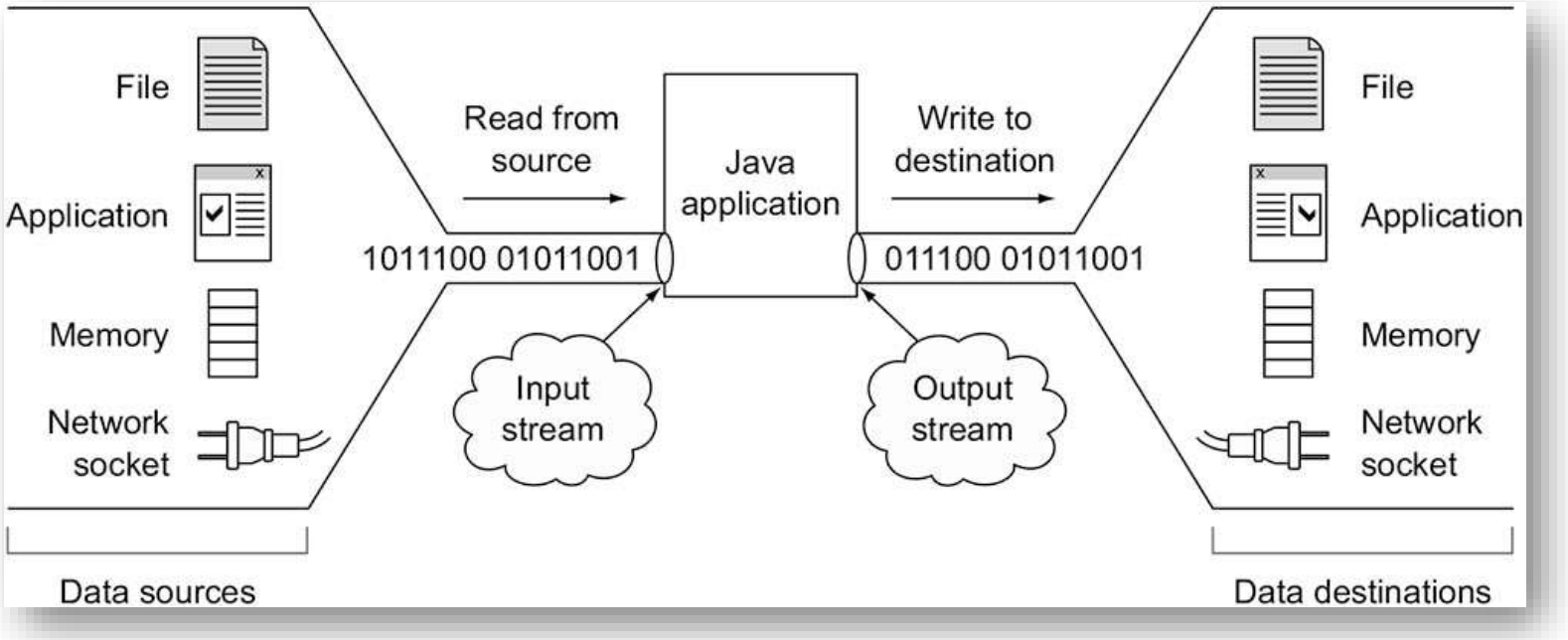
- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.



Streams

- Streams represents a **Source** (which generates the data in the form of Stream) and a **destination** (which consumes or read data available as Stream).

- Streams supports a huge range of source and destinations including disk file, arrays, other devices, other programs etc.



File Handling

- File handling is an important part of any application.
- File Handling implies how to read from and write to file in Java.
- Java provides the basic I/O package for reading and writing streams.
- **Java.io** package allows to do all Input and Output tasks in Java.
- It provides several methods for creating, reading, updating, and deleting files.

File Class

- The File class from the java.io package, allows us to work with files.
- To use the File class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

File Class

- A File object can refer to either a file or directory

  File file1 = new File(“data.txt”);

  File file2 = new File(“C:\Java”);

- To obtain the path to the current working directory use

  System.getProperty(“user.dir”);

- To obtain the file or path separator use

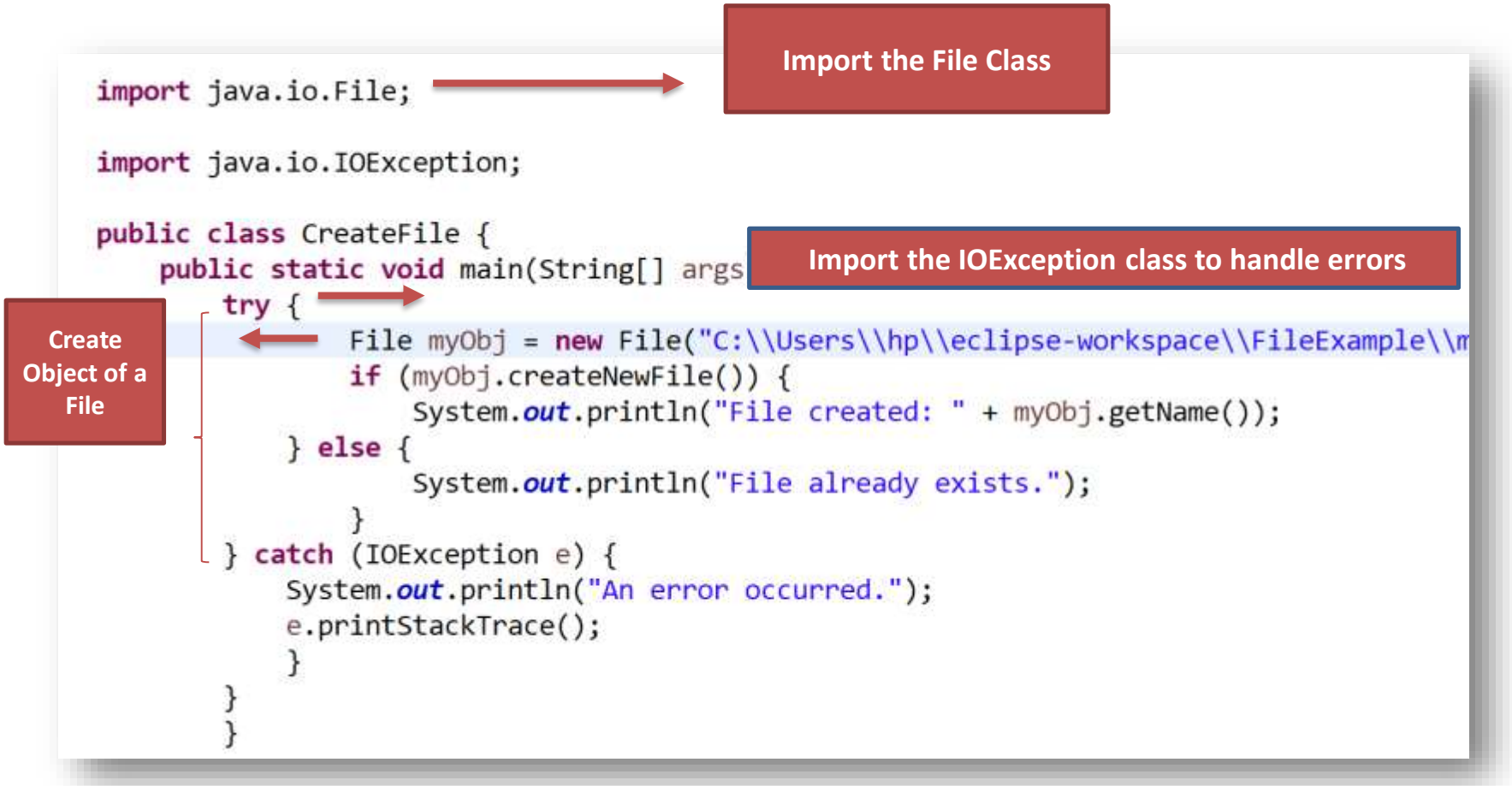
  System.getProperty (“file.separator”);

  System.getProperty (“path.separator”);

Useful File Methods

Method	Type	Description
`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

Create a File



In **try block**, write a code that has to be executed and **catch block** will handle the errors occur in try block. In this case, the most expected error is IOExpection and that will be handled by the catch block.

Get File Information

A path can be **absolute** or **relative**. An absolute path contains the full path from the root of the file system down to the file or directory it points to. A relative path contains the path to the file or directory relative to some other path.

```
import java.io.File; // Import the File class

public class FileInformation {

    public static void main(String[] args) {
        // Creating an object of a file
        File myObj = new File("../FileExample/myFiles/input.txt"); //relative path
        if (myObj.exists()) {
            // Returning the file name
            System.out.println("File name: " + myObj.getName());

            // Returning the path of the file
            System.out.println("Absolute path: " + myObj.getAbsolutePath());

            // Displaying whether the file is writable
            System.out.println("Writeable: " + myObj.canWrite());

            // Displaying whether the file is readable or not
            System.out.println("Readable " + myObj.canRead());

            // Returning the length of the file in bytes
            System.out.println("File size in bytes " + myObj.length());
        } else {
            System.out.println("The file does not exist.");
```

Output

```
File name: input.txt
Absolute path: C:\Users\hp\eclipse-workspace\
Writeable: true
Readable true
File size in bytes 0
```

Directory Listing Example

```
import java.io.*;

public class DirListing {
  public static void main(String[] args) {
    File dir = new File(System.getProperty("user.dir"));

    if (dir.isDirectory())
       {
      System.out.println("Directory of " + dir);
      String[] listing = dir.list();
      for (int i=0; i < listing.length; i++) {
        System.out.println("\t" + listing[i]);
    }

    }
  }
}
```

```
Direcotry of c:\Java\
        DirListing.class
        DirListing.java
        Test
        TryCatchExample.class
        TryCatchExample.java
        XslTransformer.class
        XslTransformer.java
```

Directory Listing, Result

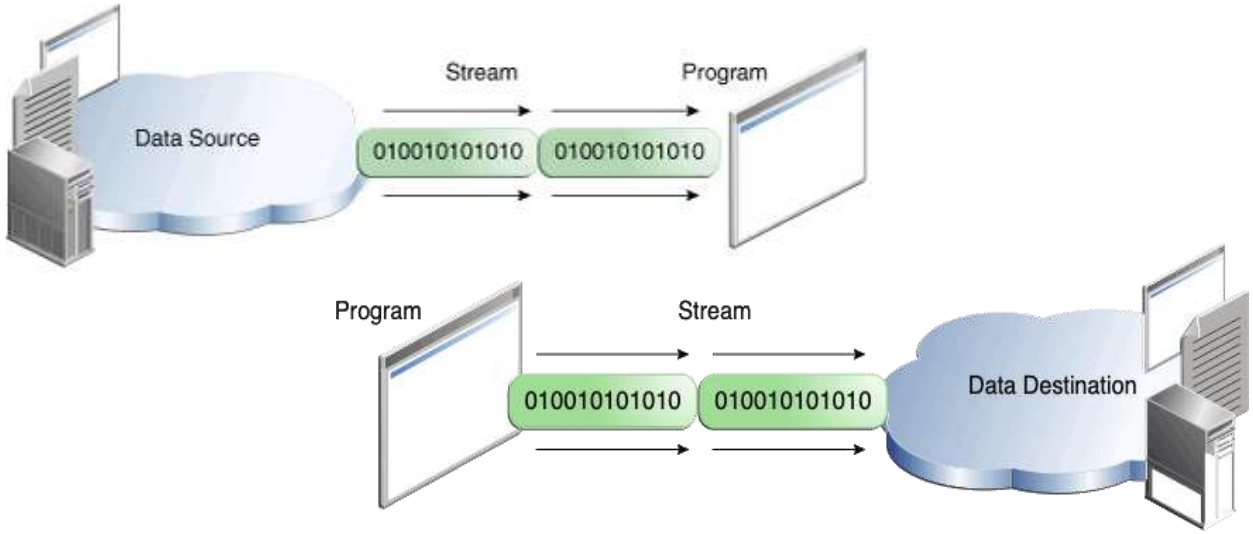
> java DirListing

Direcotry of c:\Java

DirListing.class
DirListing.java
Test
TryCatchExample.class
TryCatchExample.java
XslTransformer.class
XslTransformer.java

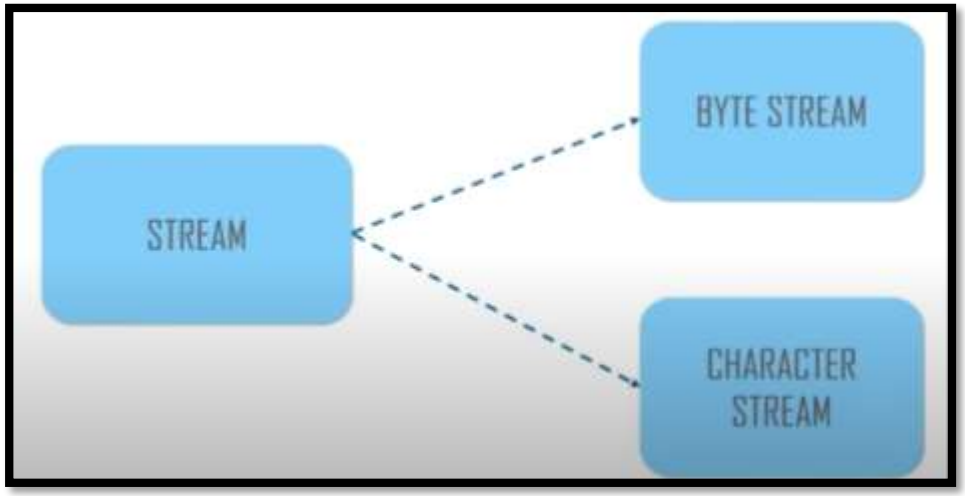
Input/Output Streams

- Java IO package provides over 60 input/output classes (streams)
- Streams are ordered sequence of data that have a source (input), or a destination (output)



Streams

- Java uses the concept of a stream to make I/O operations on a file.



- These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can **store characters, videos, audios, images** etc.

- These handle data in 16 bit Unicode. Using these you **can read and write text data** only.

Basic IO Algorithm

- **Reading**

open a stream
while more information
    read information

close the stream

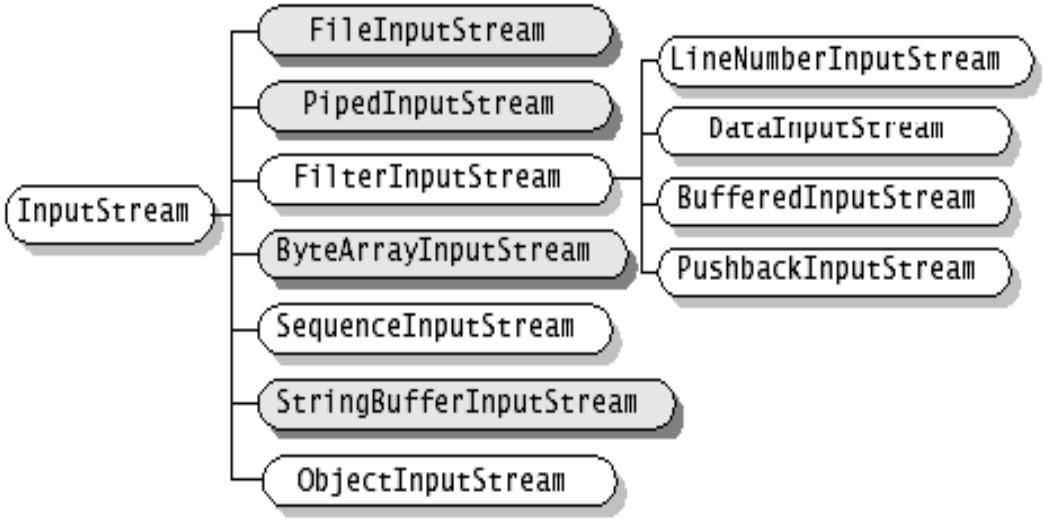
- **Writing**

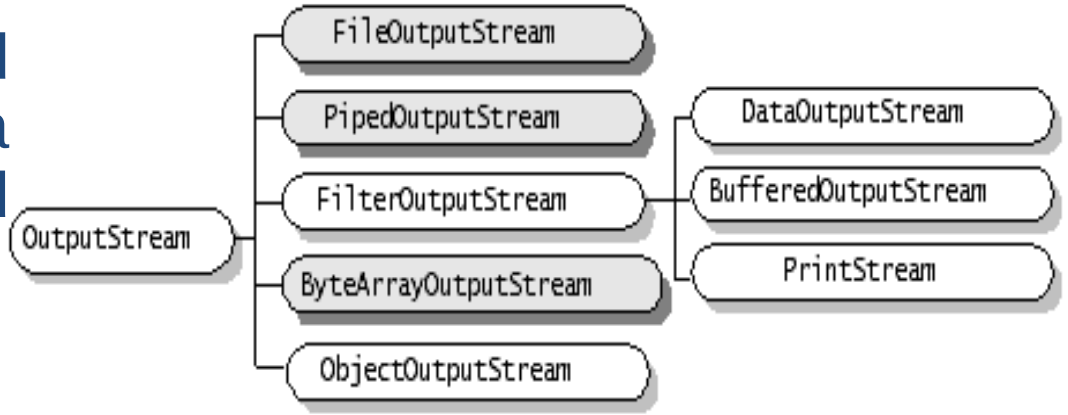
open a stream
while more information
    write information

close the stream

Byte Streams

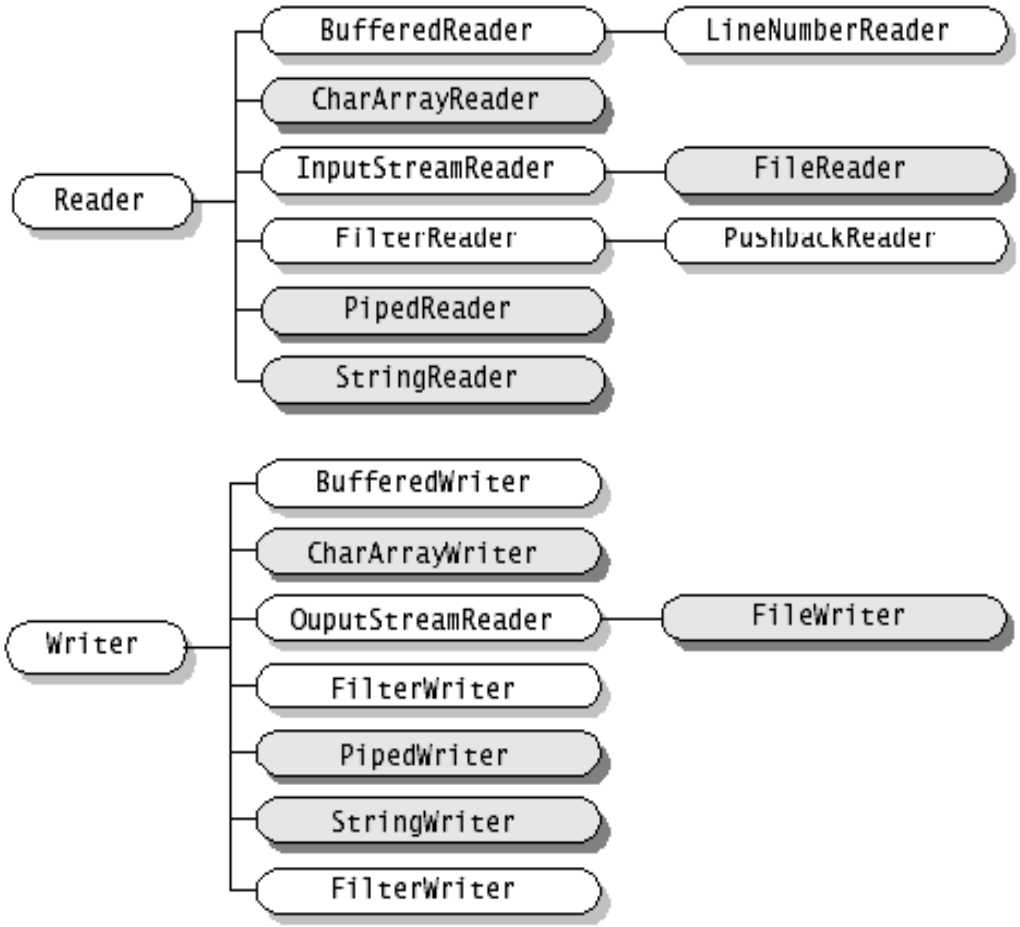
- Read and write 8-bit bytes
- **`InputStream`** and **`OutputStream`** are abstract super classes
- Typically used to read and write binary data such as images and sounds





Character Streams

- Read and write 16-bit characters
- **`Reader`** and **`Writer`** are the abstract classes
- Use readers and writers to read and write textual information



I/O Super Classes

Reader

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

InputStream

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

Writer

```
int write(int c);
int write(char cbuf[]);
int write(char cbuf[],
          int offset,
        int length)
```

OutputStream

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

FileOutputStream Example

```
import java.io.FileOutputStream;

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

        String data = "This is a line of text inside the file.";

        try {
            FileOutputStream output = new FileOutputStream("output.txt");

            byte[] array = data.getBytes();

            // Writes byte to the file
            output.write(array);

            output.close();
        }

        catch(Exception e) {
            e.getStackTrace();
        }
    }
}
```

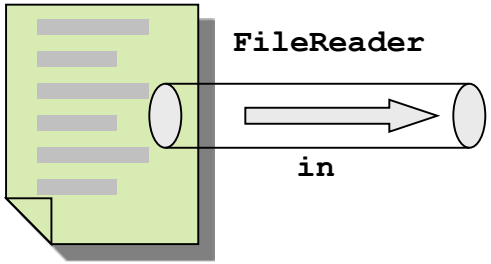
FileInputStream Example

```
This is a line of text inside the file.
```

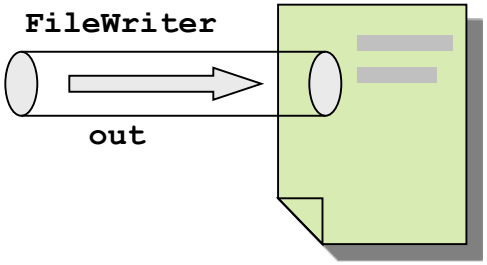
```java
import java.io.FileInputStream;

public class Main {

  public static void main(String args[]) {

     try {
        FileInputStream input = new FileInputStream("input.txt");

        System.out.println("Data in the file: ");

        // Reads the first byte
        int i = input.read();

       while(i != -1) {
           System.out.print((char)i);

           // Reads next byte from the file
           i = input.read();
        }
        input.close();
     }

     catch(Exception e) {
        e.getStackTrace();
     }
  }
}
```

Character File Streams

- FileReader



- FileWriter



Write to a file

```java
// Import the FileWriter class
import java.io.FileWriter;

public class WriteToFile {
    public static void main(String[] args) {
        try {
            FileWriter myWriter = new FileWriter("../FileExample/myFiles/TestFile.txt");
            // Writes this content into the specified file
            myWriter.write("Java is the prominent programming language of the millenium!");

            // Closing is necessary to retrieve the resources allocated
            myWriter.close();
            System.out.println("Successfully wrote to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
```

Java FileWriter class is used to write data to a file.

write() method to write some text into the file

The close() method is used to close the file output stream and releases all system resources associated with this stream.

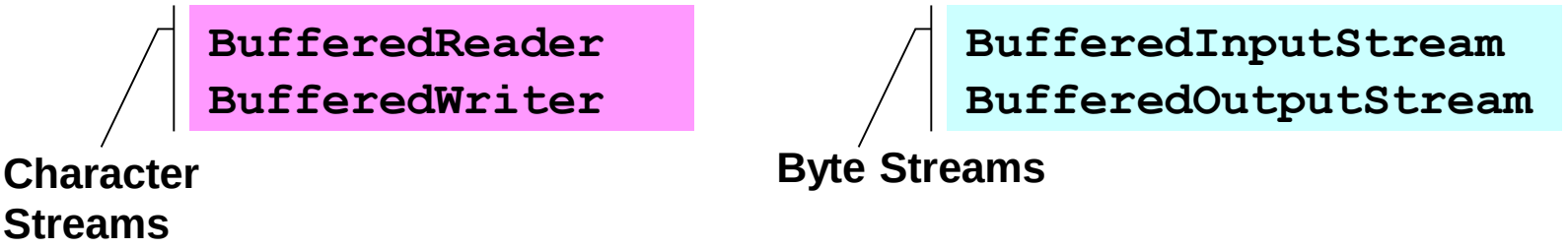
Read a File

FileReader is used for reading streams of characters.

```
// Import the File class
import java.io.File;

public class ReadFromFile {
    public static void main(String[] args) {
        try {
            // Creating an object of the file for reading the data
            FileReader fr=new FileReader("../FileExample/myFiles/TestFile.txt");
            Scanner myReader = new Scanner(fr);
            while (myReader.hasNextLine()) {
                    String data = myReader.nextLine();
                    System.out.println(data);
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }}
```

Buffer Streams

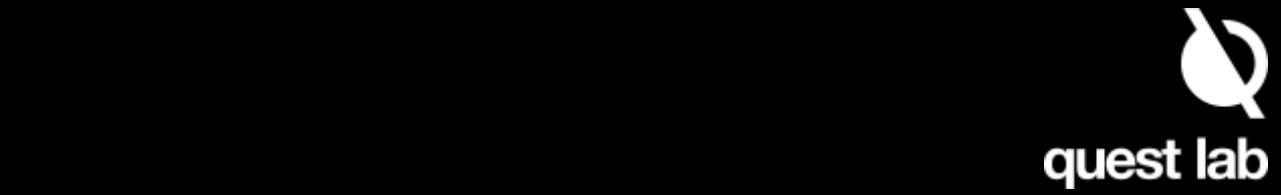
- Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source.
- More efficient than similar non-buffered streams and are often used with other streams
- The buffer size may be specified, or default size may be accepted



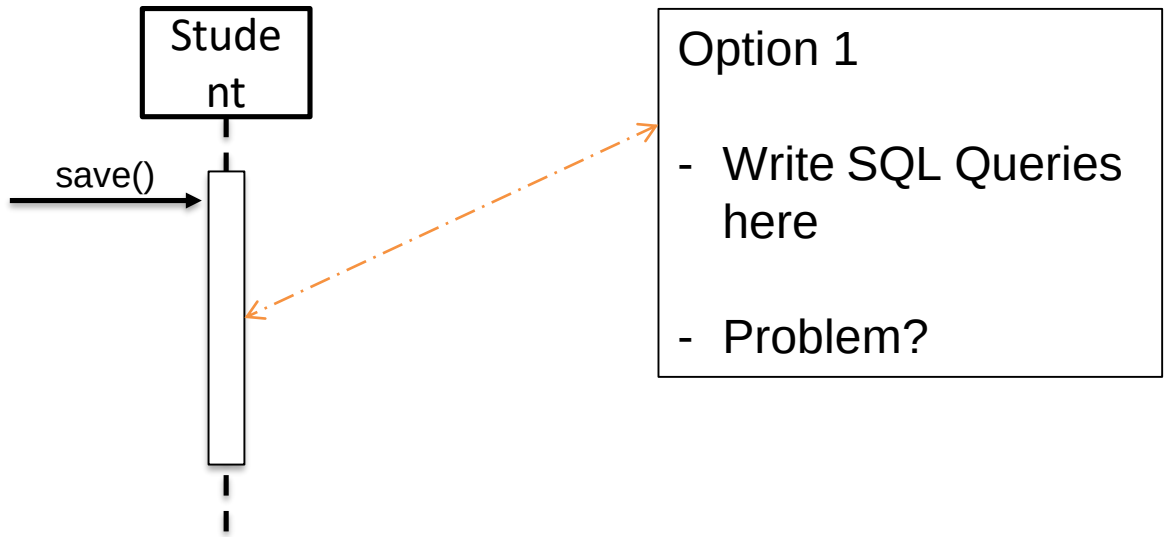
Using Buffer Streams

```
import java.io.*;

public class Copy {
   public static void main(String[] args) throws IOException {
      // opening the streams
      FileReader in = new FileReader (“infile.txt");
      BufferedReader br = new BufferedReader(in);

      FileWriter out = new FileWriter ("outfile.txt");
      BufferedWriter bw = new BufferedWriter(out);

      // processing the streams
      String aLine = null;
      while ((aLine = br.readLine()) != null) {
         bw.write(aLine, 0, aLine.length());
      }
      // closing the streams
      in.close();  out.close();
   }
}
```

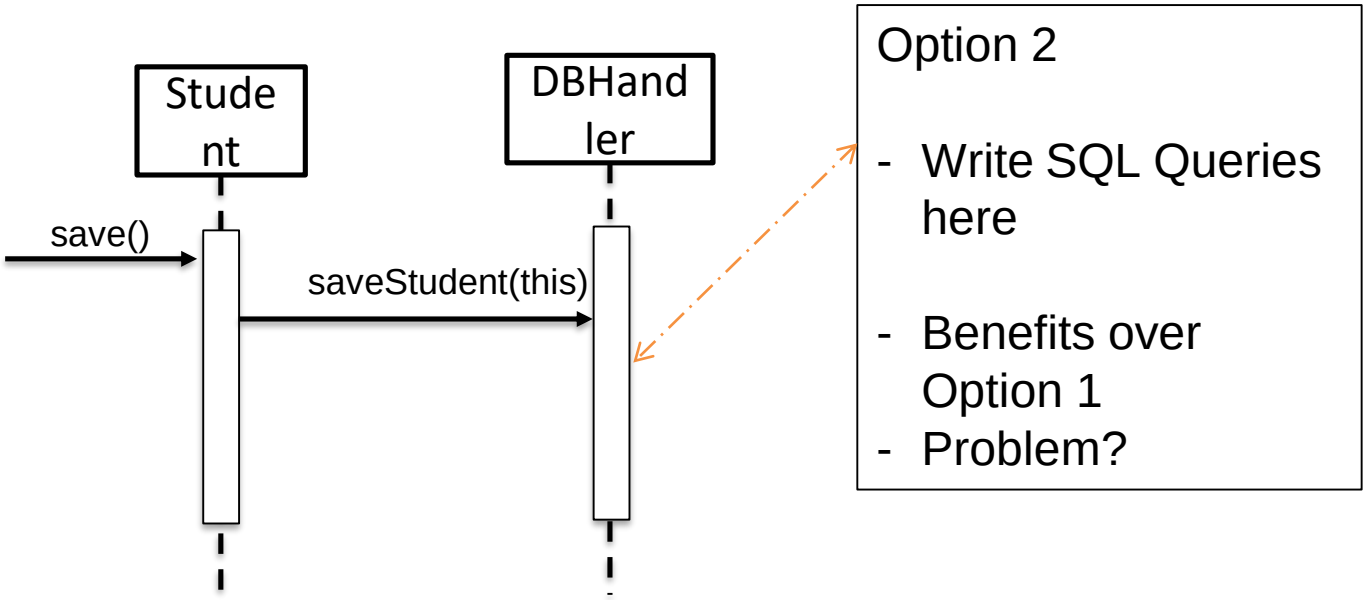
Database Connectivity



Example – Handling Persistence



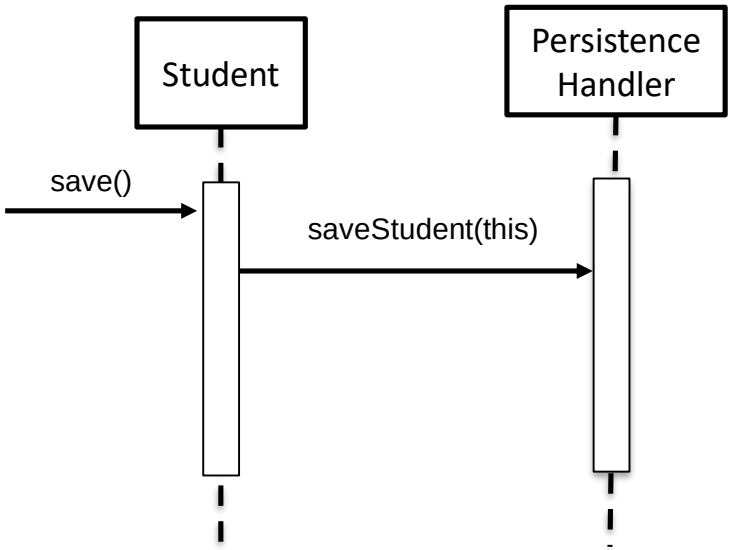
Example – Handling Persistence

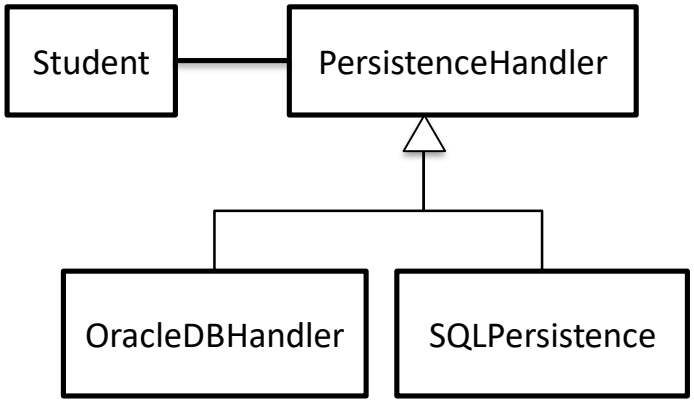


Example – Write to Interfaces

Option 3
- Write to interfaces

- Benefits over other options
- Problem? When to initialize and how?





Implementation

```
//Student
    Class Student{
      PersitenceHandler persHandler;

          void save(){
             persHandler.saveStudent(this);
                   }

          void setPersitenceHandler (PersitenceHandler ph)
                   {
              this.persHandler=ph;
                   }
```

Implementation

```
// PersistenceHandler
Class PersistenceHandler{
      abstract void saveStudent(Student s);
}
```

Implementation

```
class OracleDBHandler extends PersistenceHandler{

    void saveStudent(Student s){
        //connection
        //insert query formulation
        //execute query
    }
    }
```

Implementation

```
class SQLHandler extends PersistenceHandler{

    void saveStudent(Student s){
    //connection
    //insert query formulation
    //execute query
    }
    }
```

Implementation

Main

```
Void main()
{
    PersitenceHandler handler= new OracleHandler();

    University uni= new University();

    Uni. setPersitenceHandler(handler);
}
```