https://github.com/kkh3363/bsu_java2025

# Java Programming
# Chapter 11

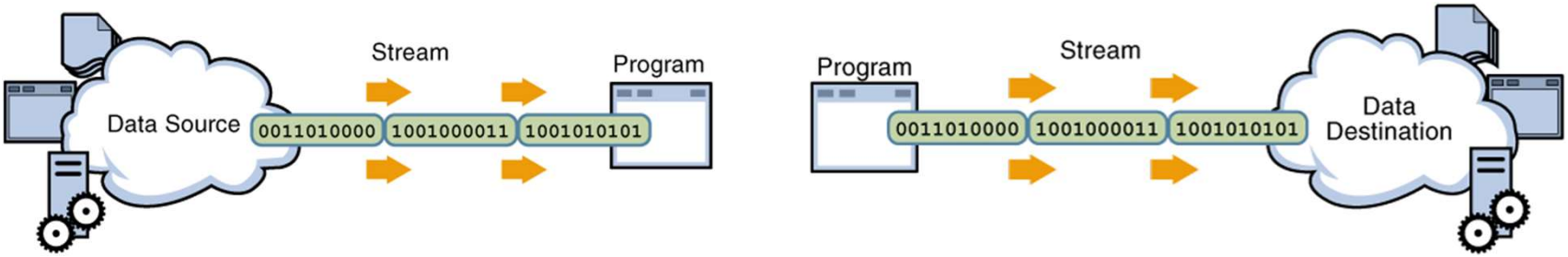
Java I/O Stream

https://github.com/kkh3363/bsu_java2025

https://github.com/kkh3363/bsu_java2025

# I/O Stream

- Stream
  - A "stream" is an abstraction that represents a continuous flow of data for reading or writing .
  - That is, it is a concept that allows data to be processed sequentially rather than all at once.
- Streams are divided into input streams and output streams .
  - An input stream is responsible for reading data from some source (e.g., a file, a memory array, a network, a console, etc.).
  - An output stream is responsible for sending data to some destination (e.g., a file, console, network, memory, etc.).



https://github.com/kkh3363/bsu_java2025

# I/O Stream

- I/O Streams
  - In Java, there is an important difference between working with the File class and working with I/O Streams (Input/Output Stream):
  - The File class (from java.io) is used to get information about files and directories:
    - Does the file exist?
    - What is its name or size?
    - Create or delete files and folders
  - But: the File class does not read or write the contents of the file.
  - So far, we have used FileWriter for writing text and Scanner for reading text. These are easy to use, but they are mainly designed for simple text files.
- I/O Streams are more flexible, because they work with text and binary data (like images, audio, PDFs).

https://github.com/kkh3363/bsu_java2025

# I/O Stream

- Types of Streams
  - Byte Streams
    - Work with raw binary data (like images, audio, and PDF files).
    - Examples: FileInputStream, FileOutputStream.
  - Character Streams
    - Work with text (characters and strings). These streams automatically handle character encoding.
    - Examples: FileReader, FileWriter, BufferedReader, BufferedWriter.

https://github.com/kkh3363/bsu_java2025

# FileInputStream

- FileInputStream
  - So far, you have used the Scanner class to read text files.
  - Scanner is very convenient for text because it can split input into lines, words, or numbers.
  - However, sometimes you need more control. For example, when reading binary data (like images, audio, or PDFs), or when you need full control of raw bytes.
  - In those cases, you use FileInputStream.

https://github.com/kkh3363/bsu_java2025

# FileInputStream

- Read a Text File (Basic Example)
  - to read a text file, one byte at a time, and print the result as characters:

```java
import java.io.FileInputStream;  // Import FileInputStream
import java.io.IOException;      // Import IOException

public class Main {
  public static void main(String[] args) {
    // try-with-resources: FileInputStream will be closed automatically
    try (FileInputStream input = new FileInputStream("filename.txt")) {

      int i;  // variable to store each byte that is read

      // Read one byte at a time until end of file (-1 means "no more data")
      while ((i = input.read()) != -1) {
        // Convert the byte to a character and print it to the console
        System.out.print((char) i);
      }

    } catch (IOException e) {
      // If an error happens (e.g. file not found), print an error message
      System.out.println("Error reading file.");
    }
  }
}
```

https://github.com/kkh3363/bsu_java2025

# FileInputStream

- Copy a Binary File (Real-World Example)

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFile {
  public static void main(String[] args) {
    // Copy image.jpg into copy.jpg
    try (FileInputStream input = new FileInputStream("image.jpg");
         FileOutputStream output = new FileOutputStream("copy.jpg")) {

      int i;
      while ((i = input.read()) != -1) {
        output.write(i);  // write the raw byte to the new file
      }

      System.out.println("File copied successfully.");

    } catch (IOException e) {
      System.out.println("Error handling file.");
    }
  }
}
```

- Choosing the Right Class
  - Java gives you several ways to read files. Here's when to pick each one:
    - **Scanner** - best for simple text and when you want to parse numbers or words easily.
    - **BufferedReader** - best for large text files, because it is faster and reads line by line.
    - **FileInputStream** - best for binary data (images, audio, PDFs) or when you need full control of raw bytes.

https://github.com/kkh3363/bsu_java2025

# FileOutputStream

- The FileOutputStream class works in a similar way, but it writes data as raw bytes. That means you can use it not only for text files, but also for binary files (like images, PDFs, or audio).
- Write a Text File (Basic Example)

```java
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    // The text we want to write
    String text = "Hello World!";

    // try-with-resources: stream will be closed automatically
    try (FileOutputStream output = new FileOutputStream("filename.txt")) {
      output.write(text.getBytes());  // convert text to bytes and write
      System.out.println("Successfully wrote to file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
      e.printStackTrace();
    }
  }
}
```

https://github.com/kkh3363/bsu_java2025

# FileOutputStream

- Copy a Binary File

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFile {
  public static void main(String[] args) {
    // Copy image.jpg into copy.jpg
    try (FileInputStream input = new FileInputStream("image.jpg");
         FileOutputStream output = new FileOutputStream("copy.jpg")) {

      int b;
      while ((b = input.read()) != -1) {
        output.write(b);  // write each raw byte to the new file
      }
      System.out.println("File copied successfully.");
    } catch (IOException e) {
      System.out.println("Error handling file.");
    }
  }
}
```

https://github.com/kkh3363/bsu_java2025

# FileOutputStream

- Append to a File

```java
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    String text = "\nAppended text!";

    // true = append mode (keeps existing content)
    try (FileOutputStream output = new FileOutputStream("filename.txt", true)) {
      output.write(text.getBytes());
      System.out.println("Successfully appended to file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
      e.printStackTrace();
    }
  }
}
```

# BufferedReader

- BufferedReader and BufferedWriter
  - BufferedReader and BufferedWriter make reading and writing text files faster.
    - BufferedReader lets you read text line by line with readLine().
    - BufferedWriter lets you write text efficiently and add new lines with newLine().
  - These classes are usually combined with FileReader and FileWriter, which handle opening or creating the file. The buffered classes then make reading/writing faster by using a memory buffer.

https://github.com/kkh3363/bsu_java2025

# BufferedReader

- Read a Text File (Line by Line)

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new FileReader("filename.txt"))) {
      String line;
      while ((line = br.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException e) {
      System.out.println("Error reading file.");
    }
  }
}
```

- Java gives you several ways to read files. Here's when to pick each one:

  - Scanner - best for simple text. It can split text into lines, words, or numbers (e.g., nextInt(), nextLine()).
  - BufferedReader - best for large text files. It is faster, uses less memory, and can read full lines with readLine().
  - FileInputStream - best for binary files (like images, PDFs, or audio)

https://github.com/kkh3363/bsu_java2025

# BufferedWriter

- The BufferedWriter class is used to write text to a file, one line or one string at a time.
- If the file already exists, its contents will be replaced (overwritten).
- Write to a Text File

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("filename.txt"))) {
      bw.write("First line");
      bw.newLine();  // add line break
      bw.write("Second line");
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
    }
  }
}
```

- Append to a Text File

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    // true = append mode
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("filename.txt", true))) {
      bw.newLine();                    // move to a new line
      bw.write("Appended line");       // add new text at the end
      System.out.println("Successfully appended to the file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
    }
  }
}
```

- Comparing File Writing Classes
  - Java gives you several ways to write to files. Here's when to pick each one:

  - FileWriter - best for simple text writing. Quick and easy to use.
  - BufferedWriter - better for larger text files, because it is faster and lets you easily add line breaks with newLine().
  - FileOutputStream - best for binary files (like images, PDFs, or audio)