Unit 6: Stream in Java

Introduction to stream

In Java, streams are introduced in Java 8 and provide a functional way to process collections of elements.

They offer a concise and readable syntax for common operations like filtering, mapping, and reducing data.

Streams don't store data themselves but instead provide a way to process elements from a source like a collection or an array.

Benefits of Java 8 Streams

- Improved Readability: Streams allow you to write more concise and readable code, making it easier to understand and maintain.
- Lazy Evaluation: Streams perform operations lazily, which means that they only execute as needed and only on the elements of the stream that are actually used, reducing memory usage and improving performance.
- Parallel Processing: Streams can perform operations in parallel, which can lead to significant speedups in performance, especially for large datasets.

Creating Java 8 Streams

Java 8 Streams can be created from various sources, including:

- Collections: Using the stream() method on a collection, such as a List or Set.
- Arrays: Using the Arrays.stream() method on an array.
- Files: Using the Files.lines() method on a file.

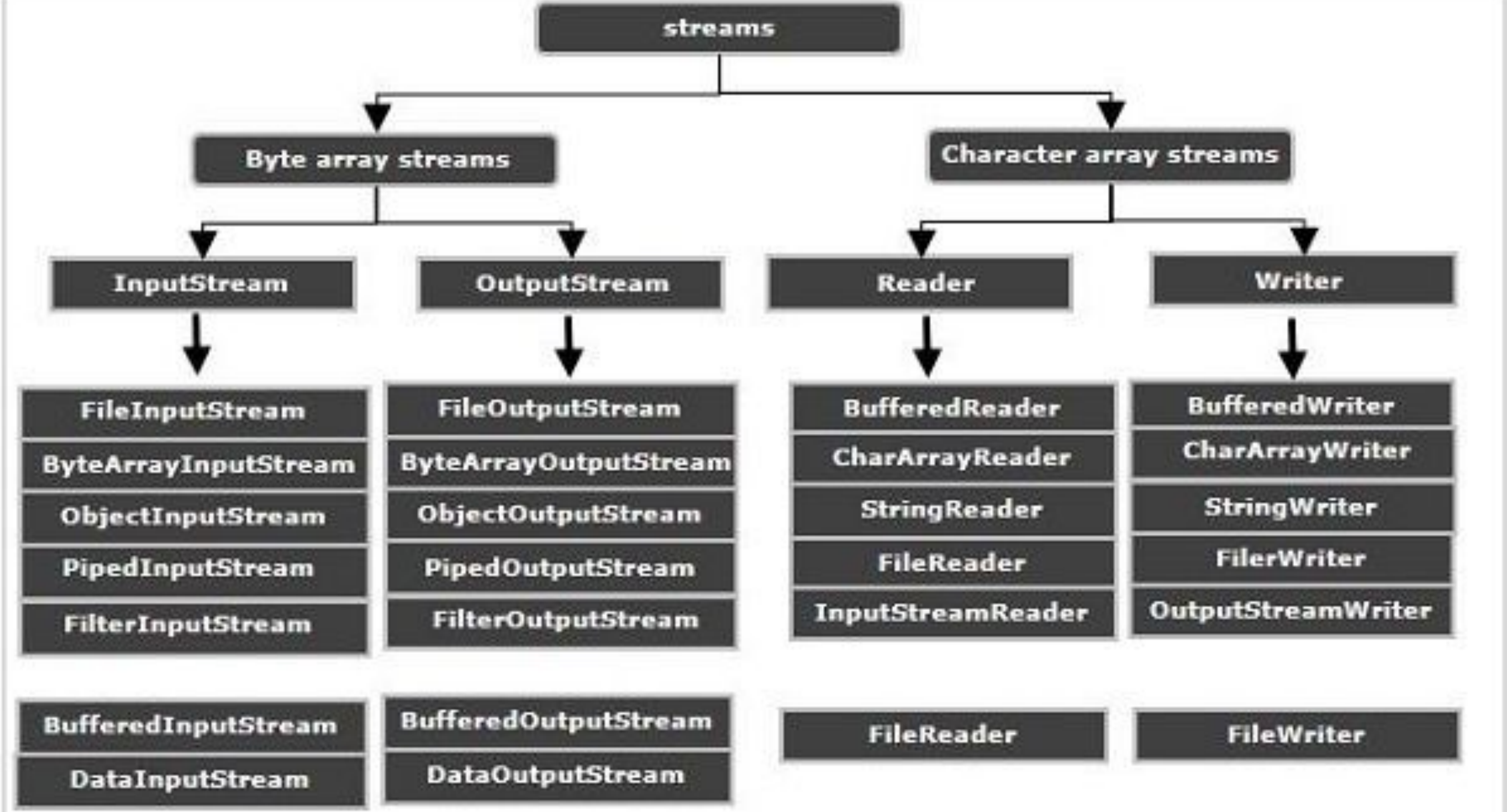
Java 8 Stream Operations

Some common operations on Java 8 Streams include:

- Filtering: Using the filter() method to select elements that match a predicate.
- Mapping: Using the map() method to transform elements from one type to another.
- Reducing: Using the reduce() method to combine elements into a single value.
- Collecting: Using the collect() method to accumulate elements into a collection or array.

Types of Streams

1. Byte Stream
2. Character Stream


streams
Byte array streams
Character array streams
InputStream
OutputStream
Reader
Writer
FileInputStream
ByteArrayInputStream
ObjectInputStream
PipedInputStream
FilterInputStream
FileOutputStream
ByteArrayOutputStream
ObjectOutputStream
PipedOutputStream
FilterOutputStream
BufferedReader
CharArrayReader
StringReader
FileReader
InputStreamReader
BufferedWriter
CharArrayWriter
StringWriter
FilerWriter
OutputStreamWriter
BufferedInputStream
DataInputStream
BufferedOutputStream
DataOutputStream
FileReader
FileWriter

Byte Streams

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.

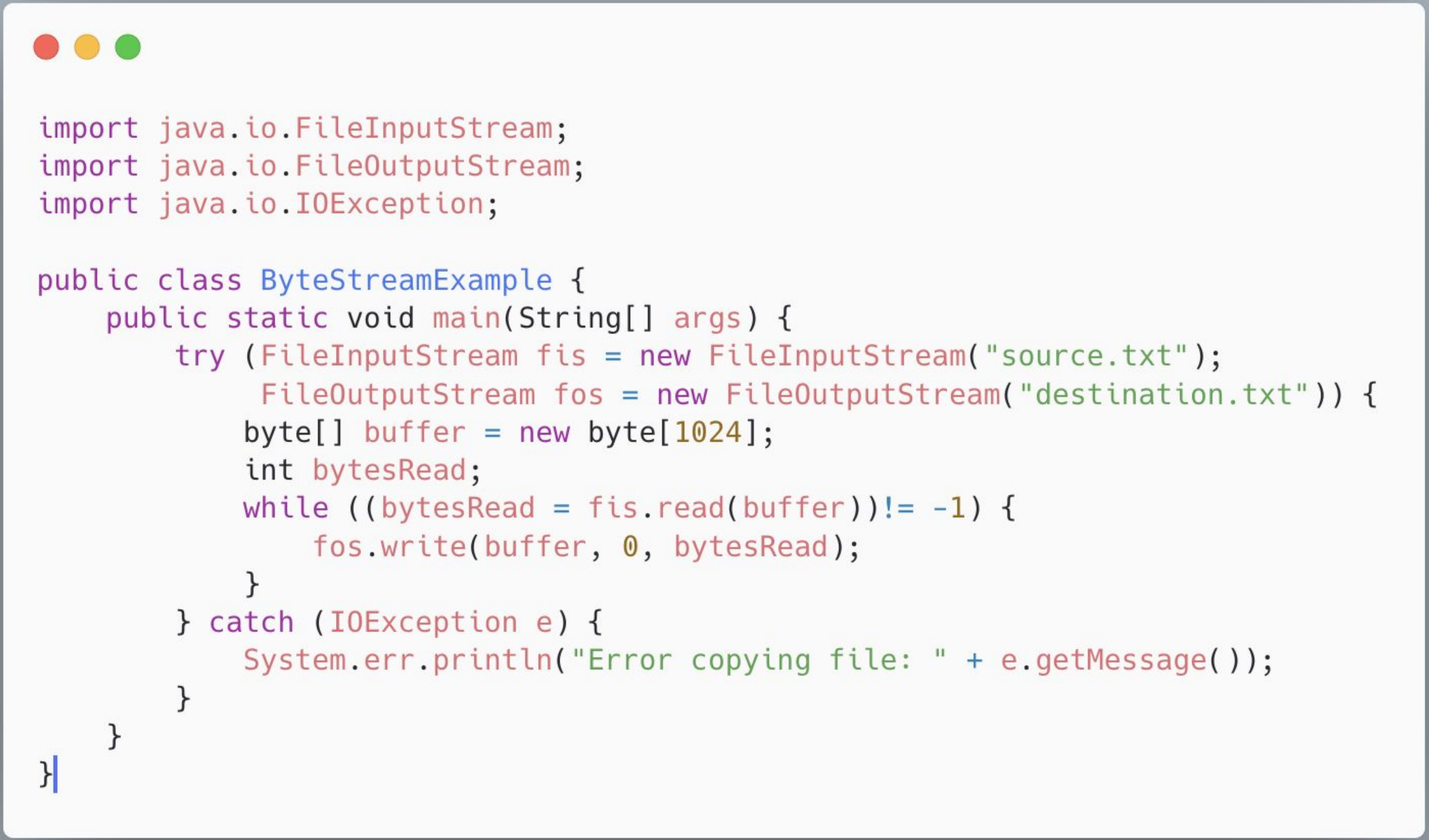
- Processes data byte by byte (8 bits at a time)
- Used for binary data, such as images, audio, and video files
- Common classes: FileInputStream, FileOutputStream
- Example: Reading and writing binary data, such as an image file

Pros:

- Can be used for binary data, such as images, audio, and video files
- More efficient for large files, as it processes data in bytes (8 bits at a time)

Cons:

- Not suitable for text files, as it may not handle Unicode characters correctly
- May require additional processing to handle character encoding



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

public class ByteStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("source.txt");
             FileOutputStream fos = new FileOutputStream("destination.txt")) {
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = fis.read(buffer))!= -1) {
                fos.write(buffer, 0, bytesRead);
            }
        } catch (IOException e) {
            System.err.println("Error copying file: " + e.getMessage());
        }
    }
}
```

Character Streams

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

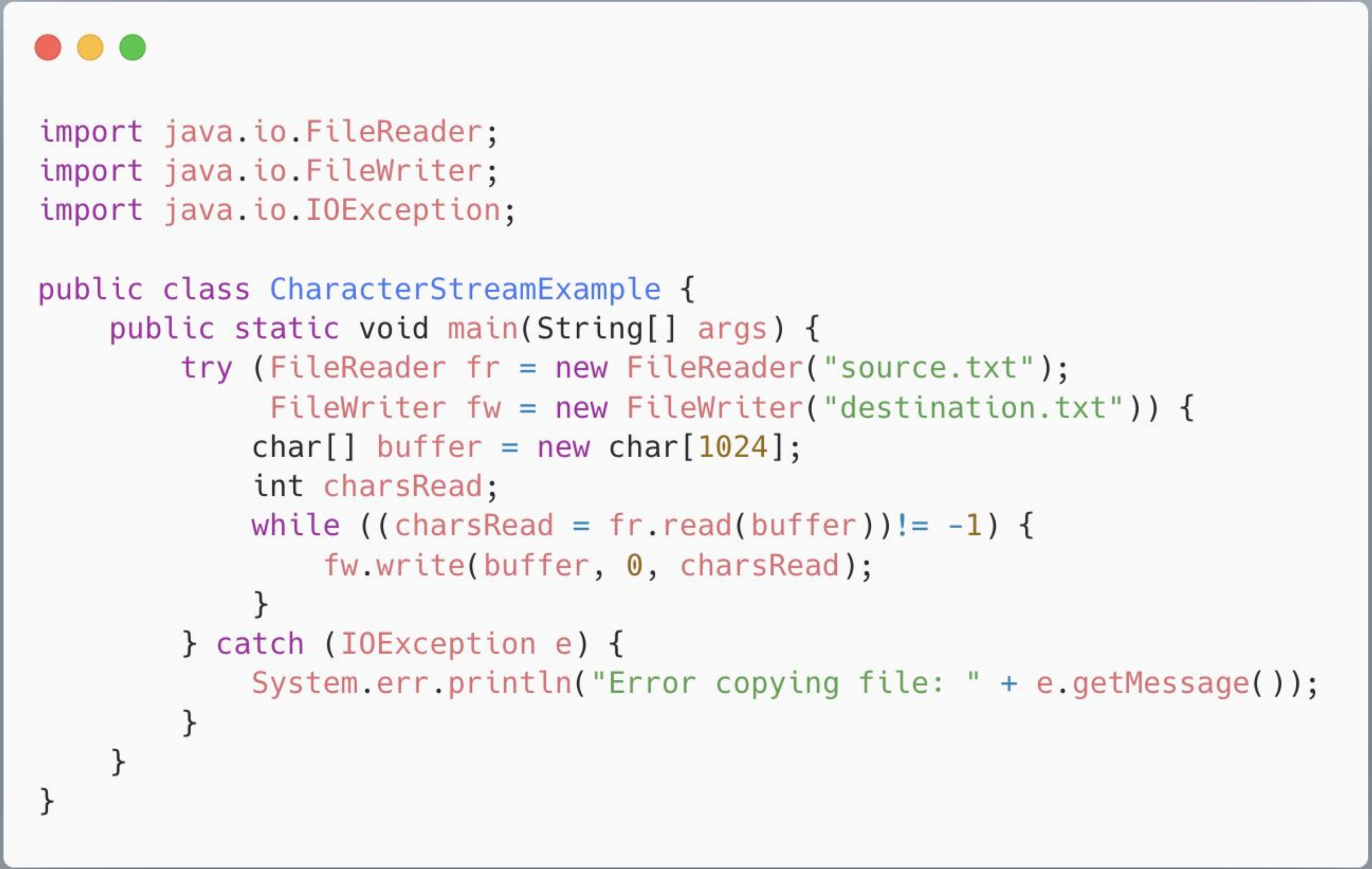
- Processes data character by character (16 bits at a time)
- Used for text files, such as reading and writing characters
- Common classes: FileReader, FileWriter
- Example: Reading and writing text files, such as a .txt file

Pros:

- More efficient for text files, especially when the file contains Unicode characters
- Provides a convenient way to handle character-based inputs and outputs
- Can handle Unicode characters correctly, without requiring additional processing

Cons:

- Not suitable for binary data, such as images, audio, and video files
- May be less efficient for large files, as it processes data in characters (16 bits at a time)



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

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("source.txt");
             FileWriter fw = new FileWriter("destination.txt")) {
            char[] buffer = new char[1024];
            int charsRead;
            while ((charsRead = fr.read(buffer))!= -1) {
                fw.write(buffer, 0, charsRead);
            }
        } catch (IOException e) {
            System.err.println("Error copying file: " + e.getMessage());
        }
    }
}
```

Key Differences

1. Byte streams process data in bytes, while character streams process data in characters
2. Byte streams are used for binary data, while character streams are used for text data
3. Character streams are more efficient when dealing with text files, as they can handle Unicode characters

When to use Character Stream over Byte Stream

1. When processing text files, such as reading and writing characters
2. When dealing with Unicode characters, which require 16 bits to represent

I/O class hierarchy

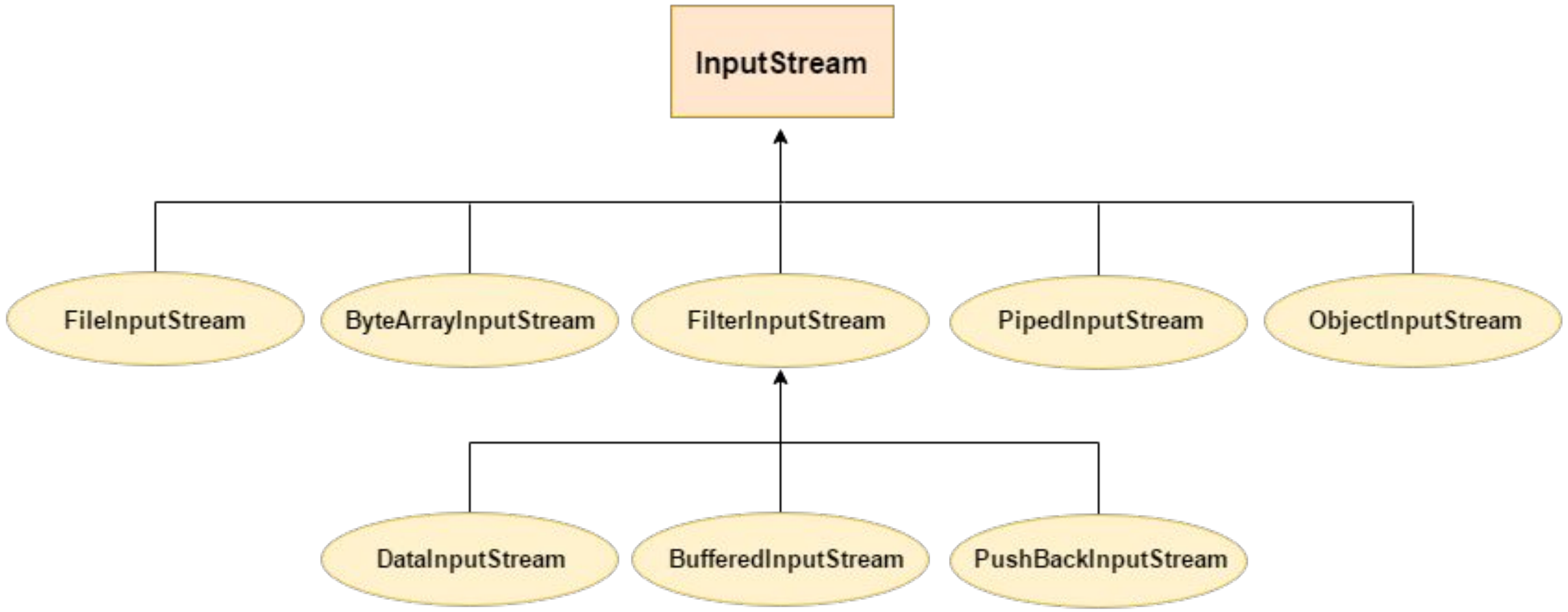
Java I/O (Input and Output) is used to process the input and produce the output.

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.

InputStream Class

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

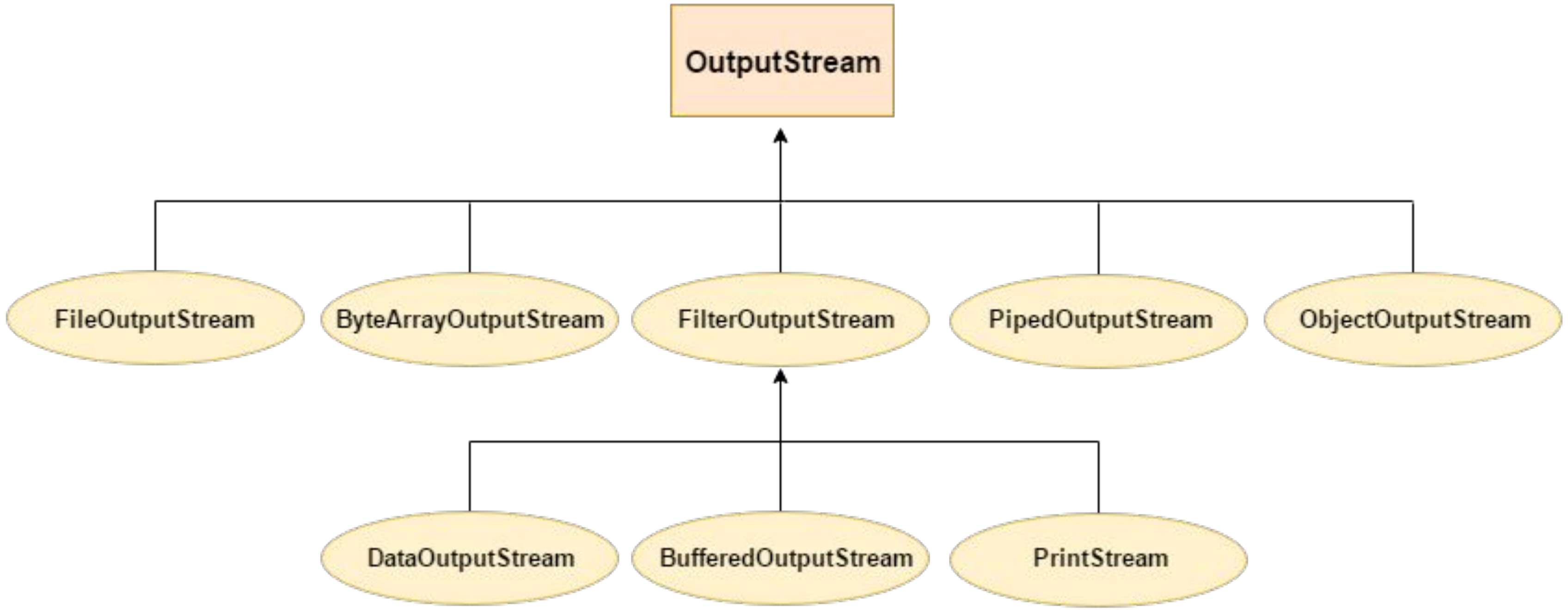
InputStream Hierarchy



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.

OutputStream Hierarchy



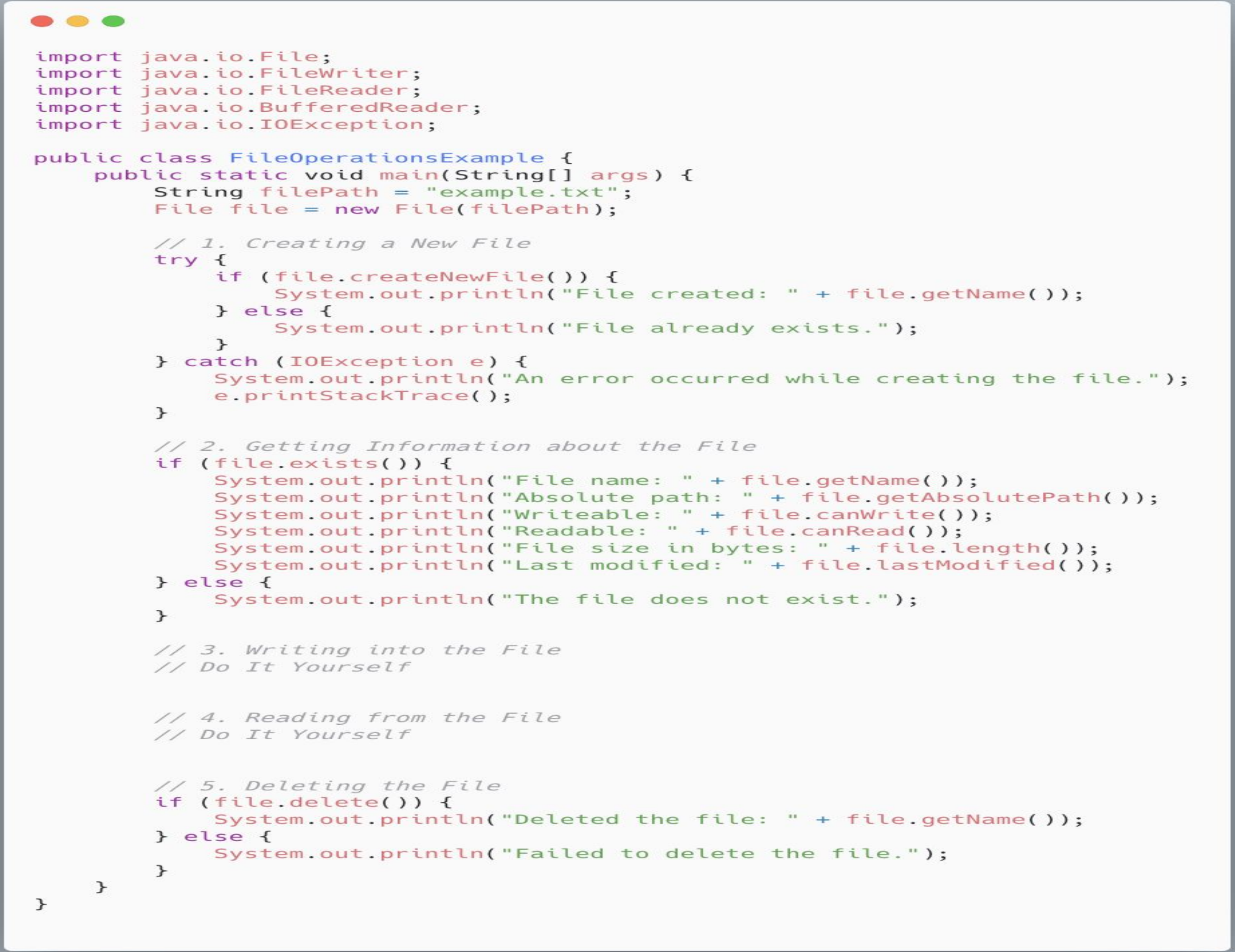
Manipulating file

File Operations in Java

File Operations: creating a new File, getting information about File, writing into a File, reading from a File and deleting a File.

- Create a File: Use File.createNewFile().
- Get File Information: Use methods like getName(), getAbsolutePath(), canWrite(), canRead(), length(), and lastModified().
- Write to a File: Use FileWriter.
- Read from a File: Use FileReader and BufferedReader.
- Delete a File: Use File.delete().

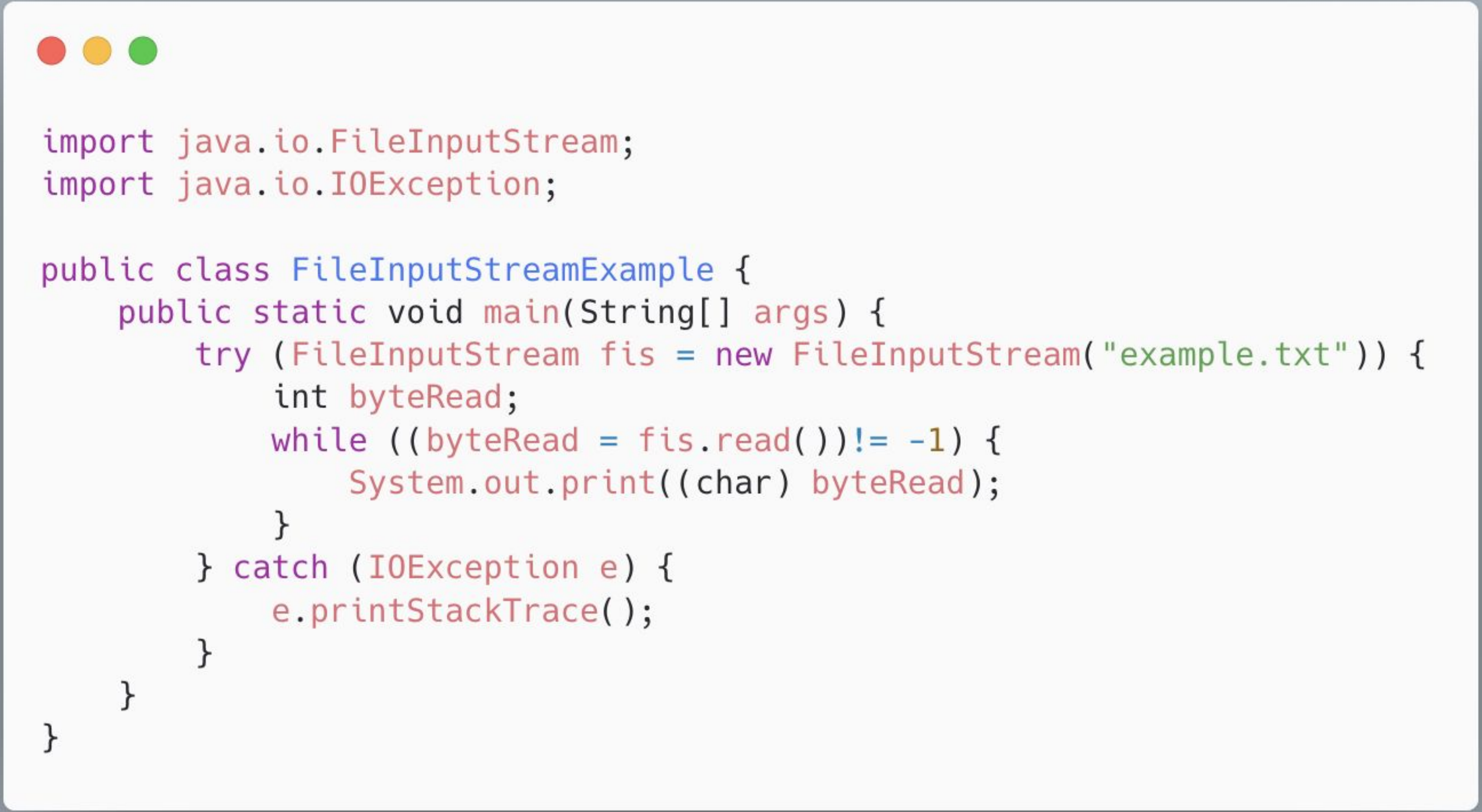
References:https://javatpoint.com/file-operations-in-java



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

public class FileOperationsExample {
    public static void main(String[] args) {
        String filePath = "example.txt";
        File file = new File(filePath);

        // 1. Creating a New File
        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred while creating the file.");
            e.printStackTrace();
        }

        // 2. Getting Information about the File
        if (file.exists()) {
            System.out.println("File name: " + file.getName());
            System.out.println("Absolute path: " + file.getAbsolutePath());
            System.out.println("Writeable: " + file.canWrite());
            System.out.println("Readable: " + file.canRead());
            System.out.println("File size in bytes: " + file.length());
            System.out.println("Last modified: " + file.lastModified());
        } else {
            System.out.println("The file does not exist.");
        }

        // 3. Writing into the File
        // Do It Yourself


        // 4. Reading from the File
        // Do It Yourself


        // 5. Deleting the File
        if (file.delete()) {
            System.out.println("Deleted the file: " + file.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

FileInputStream

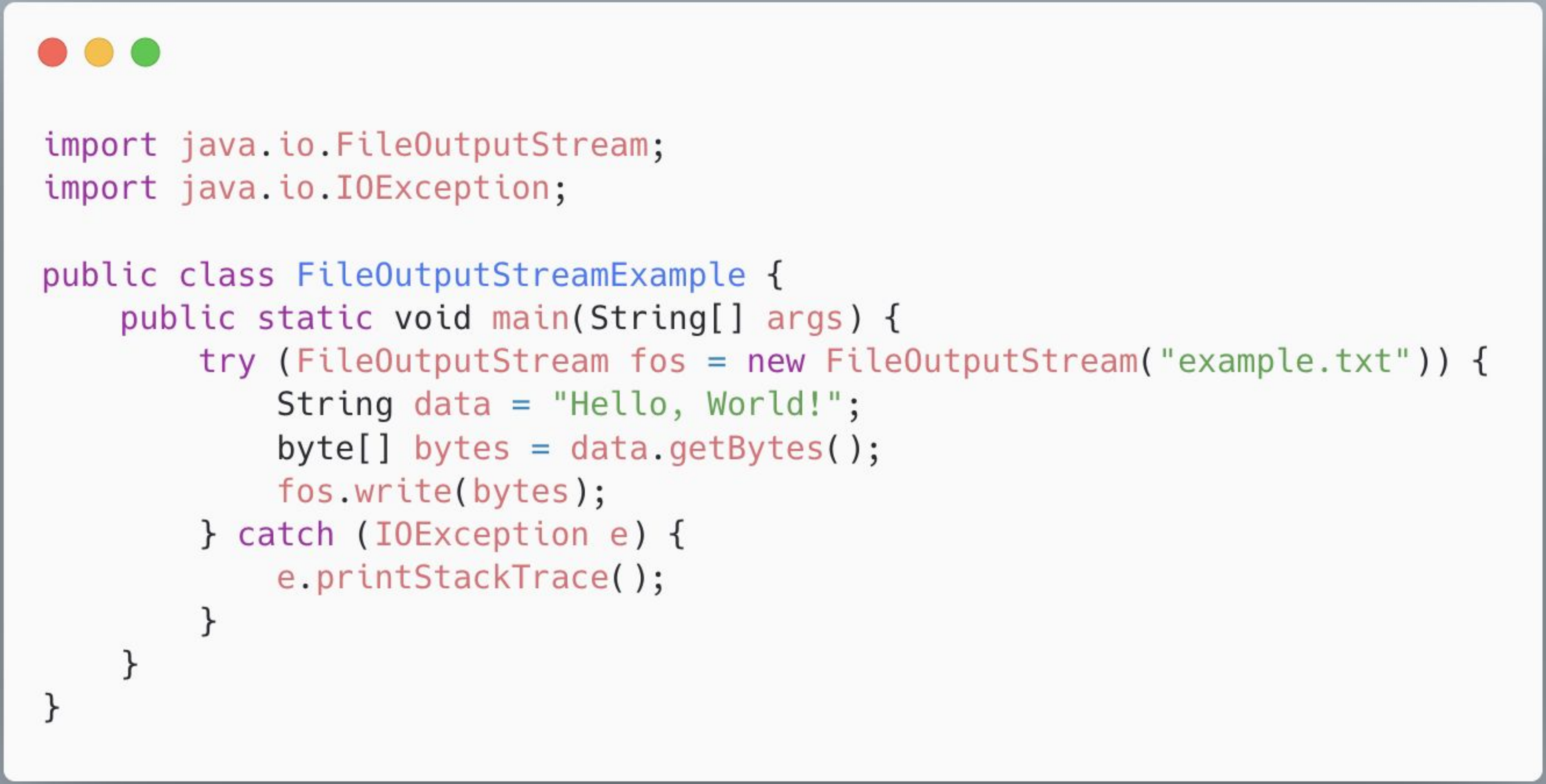
- Used for reading byte-oriented data (streams of raw bytes) from a file, such as image data, audio, video, etc.
- Can also read character-stream data, but it's recommended to use FileReader for that purpose.
- Obtains input bytes from a file.



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

public class FileInputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int byteRead;
            while ((byteRead = fis.read())!= -1) {
                System.out.print((char) byteRead);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FileOutputStream

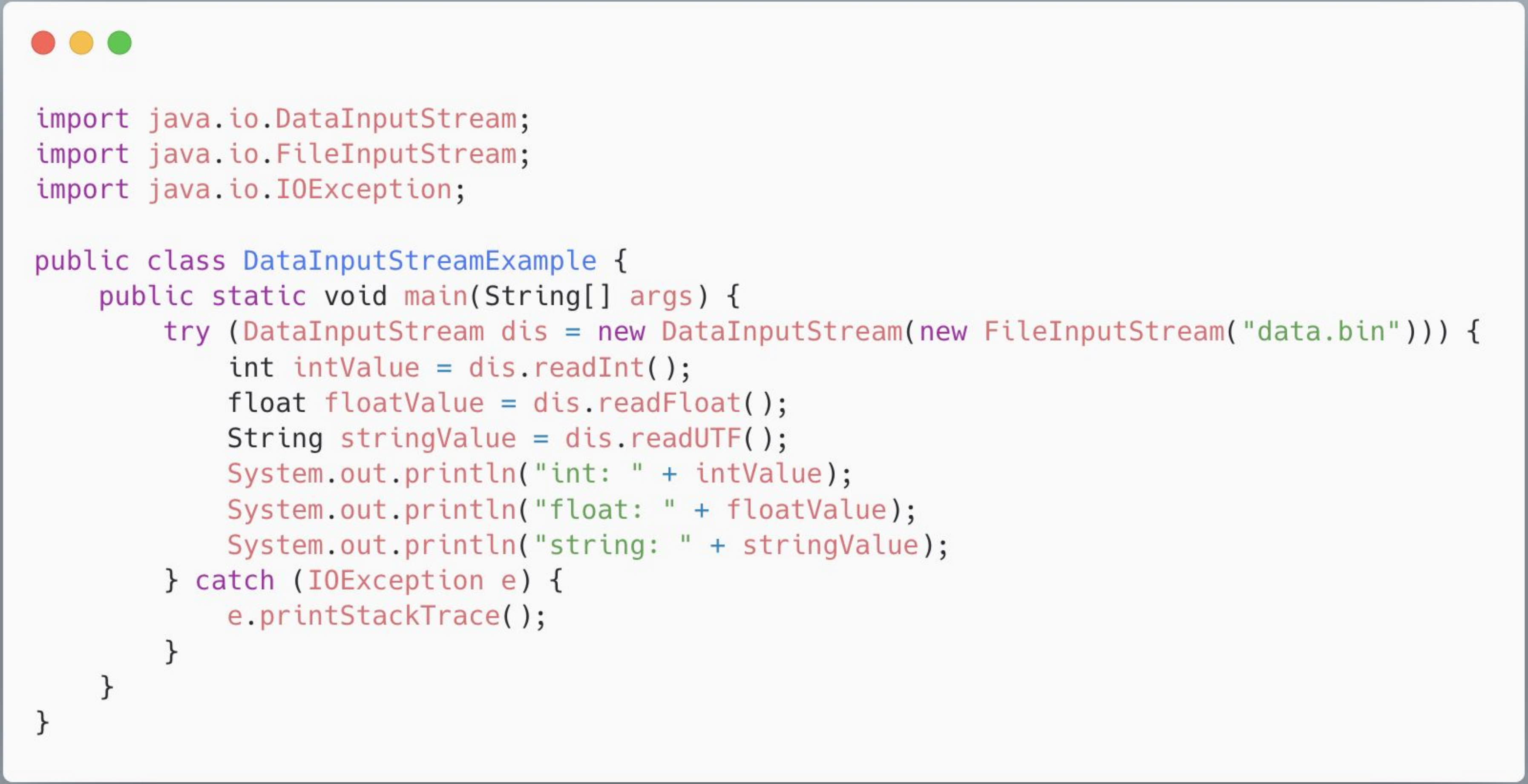
- An output stream for writing data/streams of raw bytes to a file or storing data to a file.
- It is a subclass of OutputStream.
- It can be used to write primitive values into a file, and is suitable for writing byte-oriented and character-oriented data



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

public class FileOutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "Hello, World!";
            byte[] bytes = data.getBytes();
            fos.write(bytes);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

DataInputStream

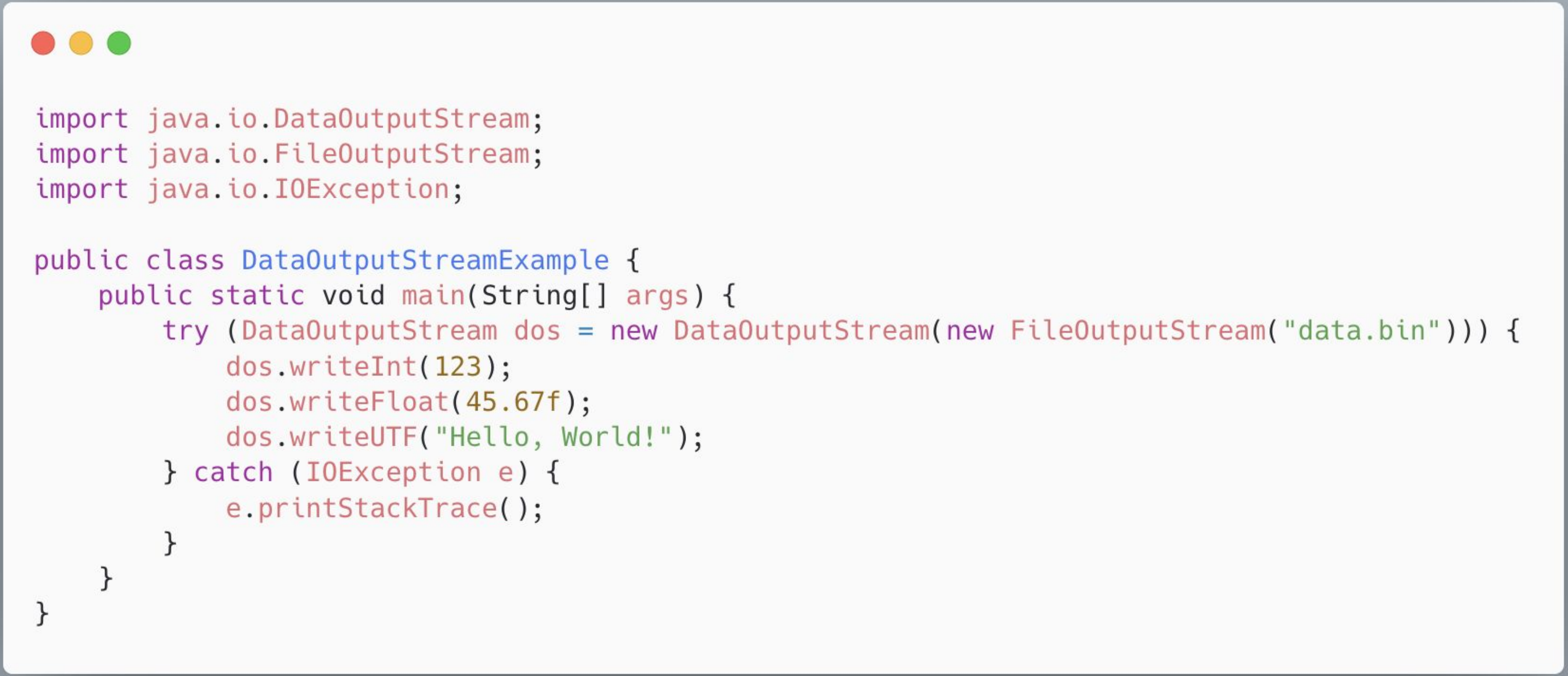
- Used to read structured data from a file, such as primitive types (e.g., int, float, etc.) and strings.
- Provides methods to read specific types of data, such as readInt(), readFloat(), etc.



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

public class DataInputStreamExample {
    public static void main(String[] args) {
        try (DataInputStream dis = new DataInputStream(new FileInputStream("data.bin"))) {
            int intValue = dis.readInt();
            float floatValue = dis.readFloat();
            String stringValue = dis.readUTF();
            System.out.println("int: " + intValue);
            System.out.println("float: " + floatValue);
            System.out.println("string: " + stringValue);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

DataOutputStream

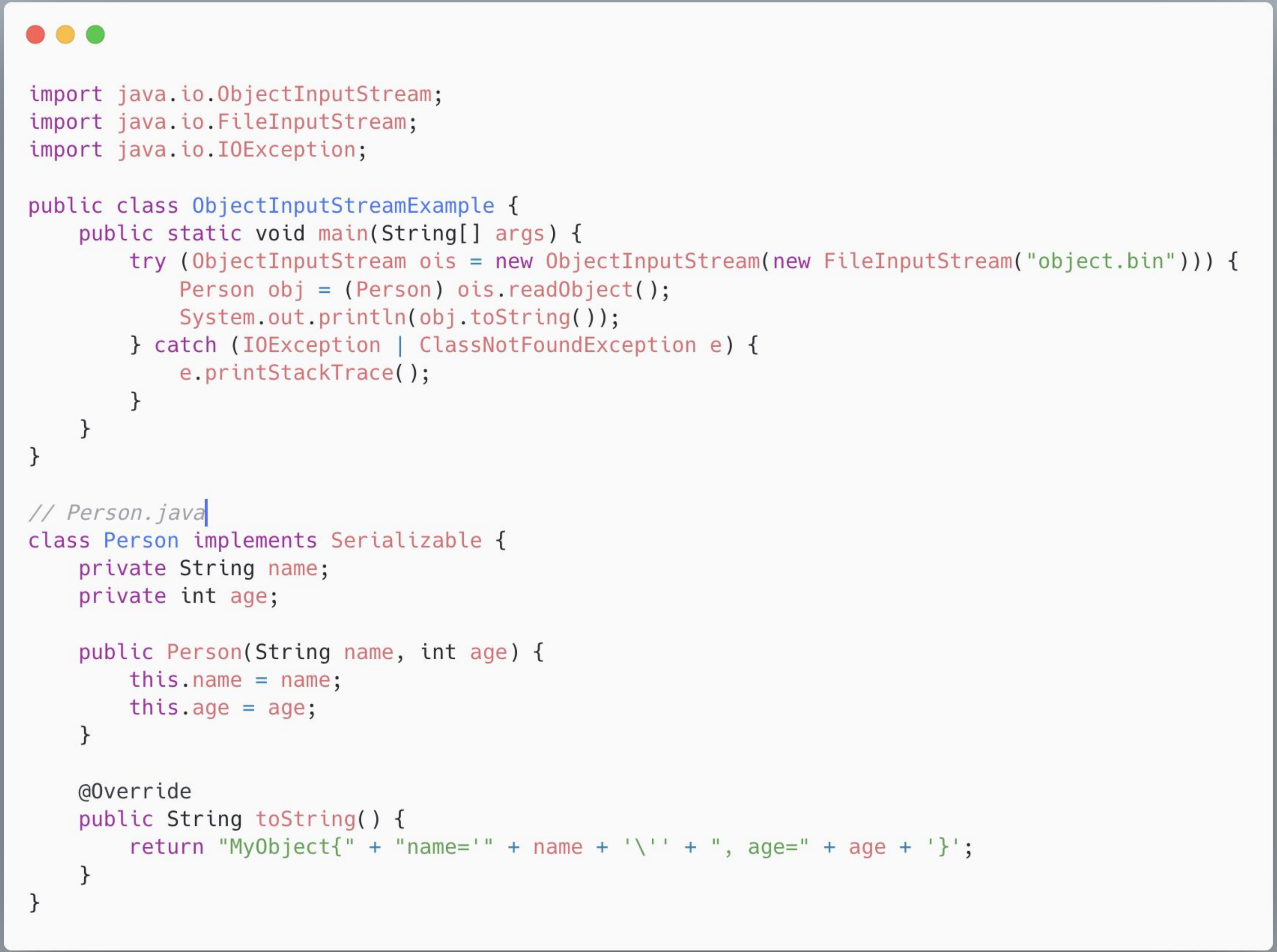
- Used to write structured data to a file, such as primitive types (e.g., int, float, etc.) and strings.
- Provides methods to write specific types of data, such as writeInt(), writeFloat(), etc.



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

public class DataOutputStreamExample {
    public static void main(String[] args) {
        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"))) {
            dos.writeInt(123);
            dos.writeFloat(45.67f);
            dos.writeUTF("Hello, World!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ObjectInputStream

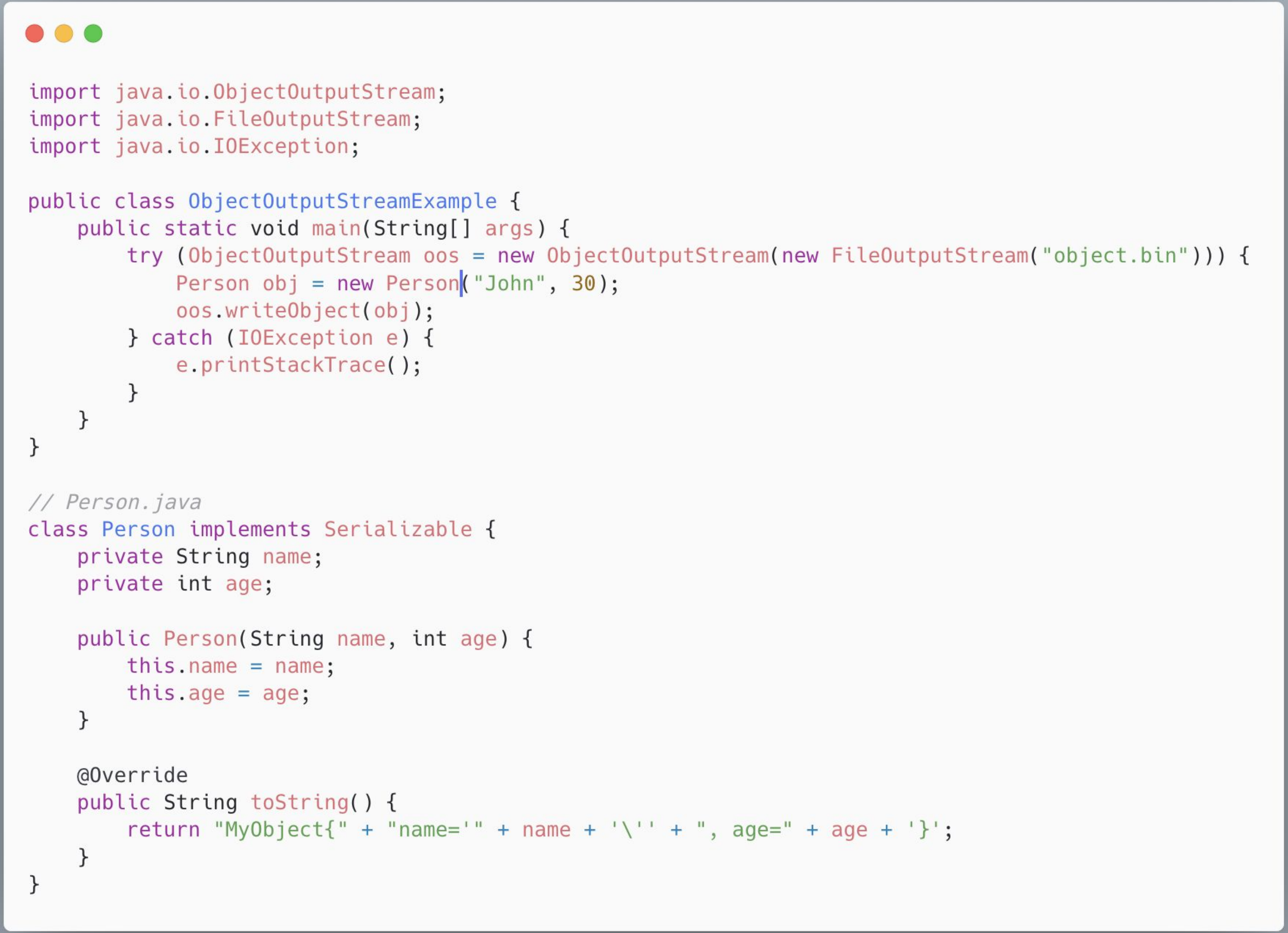
- Used for deserialization, which is the process of converting an input stream to an object.
- Used to perform object read operations.
- Can be used to read objects from a file.



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

public class ObjectInputStreamExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("object.bin"))) {
            Person obj = (Person) ois.readObject();
            System.out.println(obj.toString());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

// Person.java
class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "MyObject{" + "name='" + name + '\'' + ", age=" + age + '}';
    }
}
```

ObjectOutputStream

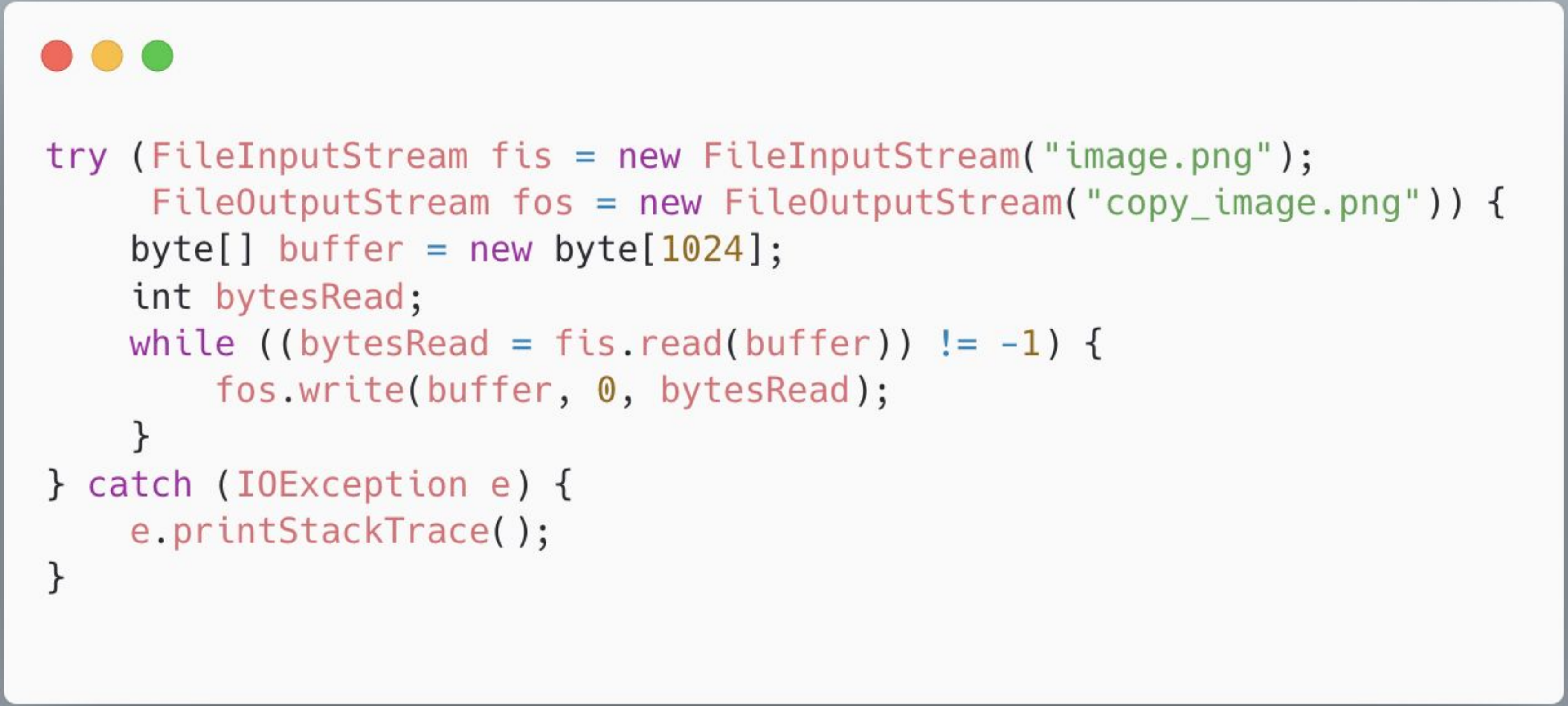
- Used for serialization, which is the process of converting an object to an output stream.
- Used to write objects to a file.



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

public class ObjectOutputStreamExample {
    public static void main(String[] args) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("object.bin"))) {
            Person obj = new Person("John", 30);
            oos.writeObject(obj);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Person.java
class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "MyObject{" + "name='" + name + '\'' + ", age=" + age + '}';
    }
}
```

Real-World Use Cases

- FileInputStream/FileOutputStream: Use when you need to work with raw binary data, such as copying files or dealing with media files.
- DataInputStream/DataOutputStream: Use when you need to read/write structured binary data, such as primitive data types, in a compact and efficient manner.
- ObjectInputStream/ObjectOutputStream: Use when you need to persist and retrieve Java objects, allowing for complex data structures to be easily saved and restored.



```java
try (FileInputStream fis = new FileInputStream("image.png");
     FileOutputStream fos = new FileOutputStream("copy_image.png")) {
    byte[] buffer = new byte[1024];
    int bytesRead;
    while ((bytesRead = fis.read(buffer)) != -1) {
        fos.write(buffer, 0, bytesRead);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Practical

1. Write a Java program to read a text file named input.txt using FileReader and print its content to the console.
2. Write a Java program to copy content from one binary file source.dat to another destination.dat using FileInputStream and FileOutputStream.
3. Write a Java program to write and then read primitive data types (int, float, string) to/from a file named data.dat using DataOutputStream and DataInputStream.
4. Write a Java program to serialize an object of a class Person (with fields name and age) to a file named person.ser and then deserialize it back using ObjectOutputStream and ObjectInputStream.