CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

Lecture 7 – IO Files

Files and Streams

- Java views each file as a sequential **stream of bytes** (Fig. 17.1).
- Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that is recorded in a system-maintained administrative data structure.
- A Java program simply receives an indication from the operating system when it reaches the end of the stream

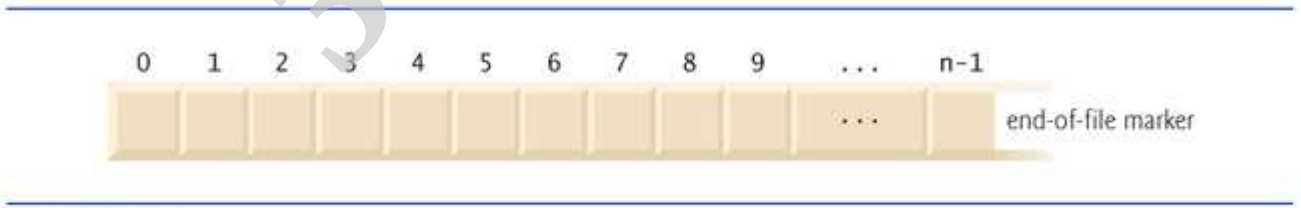


Fig. 17.1 | Java's view of a file of *n* bytes.

Files and Streams (cont.)

- File streams can be used to input and output data as bytes or characters.
- Streams that input and output bytes are known as **byte-based streams**, representing data in its binary format.
- Streams that input and output characters are known as **character-based streams**, representing data as a sequence of characters.
- Files that are created using byte-based streams are referred to as **binary files.**
- Files created using character-based streams are referred to as **text files.** Text files can be read by text editors.
- Binary files are read by programs that understand the specific content of the file and the ordering of that content.

Files and Streams (cont.)

- A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it.
  - Can also associate streams with different devices.
- Java creates three stream objects when a program begins executing
  - `System.in` (the standard input stream object) normally inputs bytes from the keyboard
  - `System.out` (the standard output stream object) normally outputs character data to the screen
  - `System.err` (the standard error stream object) normally outputs character-based error messages to the screen.

Files and Streams (cont.)

- Java programs perform file processing by using classes from package **java.io**.
- Includes definitions for stream classes
  - **FileInputStream** (for byte-based input from a file)
  - **FileOutputStream** (for byte-based output to a file)
  - **FileReader** (for character-based input from a file)
  - **FileWriter** (for character-based output to a file)
- You open a file by creating an object of one these stream classes. The object's constructor opens the file.

Files and Streams (cont.)

- Can perform input and output of objects or variables of primitive data types without having to worry about the details of converting such values to byte format.
- To perform such input and output, objects of classes **ObjectInputStream** and **ObjectOutputStream** can be used together with the byte-based file stream classes `FileInputStream` and `FileOutputStream`.
- The complete hierarchy of classes in package `java.io` can be viewed in the online documentation at
- http://download.oracle.com/javase/6/docs/api/java/io/package-tree.html

Files and Streams (cont.)

- Class **File** provides information about files and directories.
- Character-based input and output can be performed with classes `Scanner` and **Formatter**.
  - Class `Scanner` is used extensively to input data from the keyboard. This class can also read data from a file.
  - Class `Formatter` enables formatted data to be output to any text-based stream in a manner similar to method `System.out.printf`.

Class `File`

- Class `File` provides four constructors.
- The one with a `String` argument specifies the `name` of a file or directory to associate with the `File` object.
  - The `name` can contain **path information** as well as a file or directory name.
  - A file or directory's path specifies its location on disk.
  - An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory.
  - A **relative path** normally starts from the directory in which the application began executing and is therefore "relative" to the current directory.

Class `File` (cont.)

- The constructor with two `String` arguments specifies an absolute or relative path and the file or directory to associate with the `File` object.
- The constructor with `File` and `String` arguments uses an existing `File` object that specifies the parent directory of the file or directory specified by the `String` argument.
- Figure 17.2 lists some common `File` methods. The
- http://download.oracle.com/javase/6/docs/api/java/io/File.html

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

Problem: Explore File Properties

- Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix.

TestFileClass

```java
import java.io.File;
public class Main {
  public static void main(String[] args) {
        File file = new File("C:\\Users\\ahmed\\Desktop\\a.txt");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " +
            file.getAbsolutePath());
        System.out.println("Last modified on " +
            new java.util.Date(file.lastModified()));
    }
  }
```

```
Does it exist? true
The file has 373 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? true
Is it hidden? false
Absolute path is C:\Users\ahmed\Desktop\a.txt
Last modified on Thu Dec 08 23:56:05 EET 2022
```

Text I/O

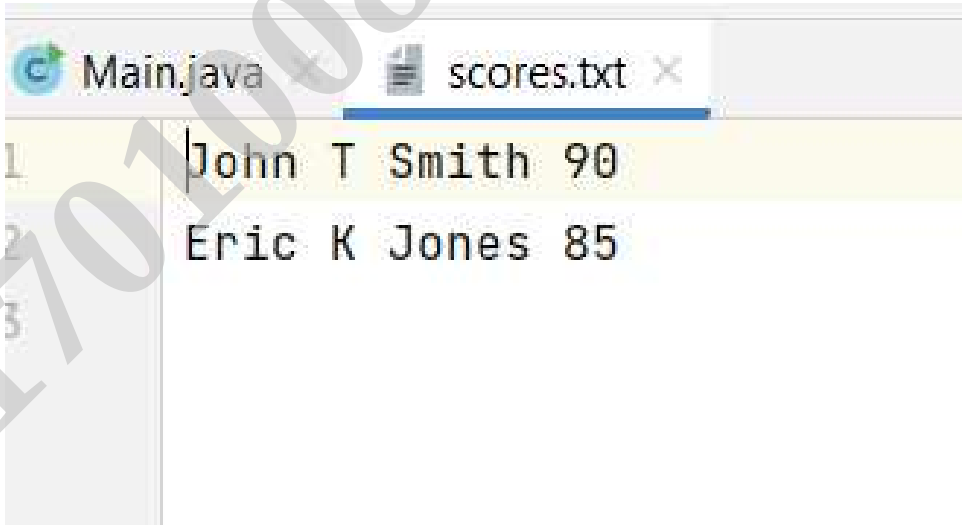
- A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes.
- The objects contain the methods for reading/writing data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output and Strings."

WriteData

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class Main {
   public static void main(String[] args) {
      File file = new File("scores.txt");
      if (file.exists()) {
         System.out.println("File already exists");
         System.exit(0);
      }
      PrintWriter output = null;
      try {
         output = new PrintWriter(file);
         // Write formatted output to the file
         output.print("John T Smith ");
         output.println(90);
         output.print("Eric K Jones ");
         output.println(85);
         // Close the file
         output.close();
      } catch (FileNotFoundException e) {
         throw new (e);
      } RuntimeException
   }}
```



Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

ReadData

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class Main {
  public static void main(String[] args) throws FileNotFoundException {
    File file = new File("scores.txt");
    Scanner input = new Scanner(file);
    while (input.hasNext()) {
      String firstName = input.next();
      String mi = input.next();
      String lastName = input.next();
      int score = input.nextInt();
      System.out.println(firstName + " " + mi + " " + lastName + " " + score);
    }
    input.close();
  }
}
```

John T Smith 90
Eric K Jones 85

Case Study(CRUD Operations)

- **CRUD** refers to the four basic **operations** a software application should be able to perform – Create, Read, Update, and Delete
- Create mini system to add, delete , update , search , list ,student to files
- There admin class that control the system

Main Class

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String  Fname, Lname;
        int id, oldID, age, level;
        double GPA;
        Admin admin=new Admin();
        Scanner in=new Scanner(System.in);
        int choice;
        while(true) {
            System.out.println("1- Add student" + "\n" +
                    "2- Delete Student" + "\n" +
                    "3- Update Student" + "\n" +
                    "4- Search Student" + "\n" +
                    "5- List all Students" + "\n" +
                    "6- Exit" + "\n");
            System.out.println("Enter your Choice");
            choice=in.nextInt();
```

Main Class

```
switch(choice){
   case 1:
      System.out.println("Enter Student Info ... ");
      System.out.print("Student First Name : ");
      Fname = in.next();
      System.out.print("Student Last Name : ");
      Lname = in.next();
      System.out.print("Student ID : ");
      id = in.nextInt();
      System.out.print("Student Age : ");
      age = in.nextInt();
      System.out.print("Student Level : ");
      level = in.nextInt();
      System.out.print("Student GPA : ");
      GPA = in.nextDouble();
      admin.addNewStudent(id, Fname, Lname, age, level, GPA);
      break;
   case 2:
      System.out.print("\nDelete Student info ...!\nEnter Student ID : ");
      id = in.nextInt();
      admin.deleteStudent(id);
      break;
```

Main Class

```
case 3:
    System.out.print("\nUpdate Student info ...!\nEnter Student OldID : ");
    oldID = in.nextInt();
    System.out.println("\nEnter Student New Info ... ");
    System.out.print("Student First Name : ");
    Fname = in.next();
    System.out.print("Student Last Name : ");
    Lname = in.next();
    System.out.print("Student ID : ");
    id = in.nextInt();
    System.out.print("Student Age : ");
    age = in.nextInt();
    System.out.print("Student Level : ");
    level = in.nextInt();
    System.out.print("Student GPA : ");
    GPA = in.nextDouble();
    Student x = new Student(id, Fname, Lname, age, level, GPA);
    admin.updateStudent(oldID, x);
    break;
```

Main Class

```
case 4:
                System.out.print("\nSearch for Student ...!\nEnter Student ID : ");
                id = in.nextInt();
                admin.searchForStudent(id);
                break;

            case 5:
                admin.displayStudents();
                break;
            case 6:
                System.exit(0);

        }

    }

  }
}
```

```java
public class Admin  {
   String user; pass, fname, lname;
   int id, age;
   public Admin() {   }
   public Admin(int id, String fname, String lname, int age) {
      this.id = id;
      this.fname = fname;
      this.lname = lname;
      this.age = age;
   }
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getFname() {
      return fname;
   }
   public void setFname(String fname) {
      this.fname = fname;
   }
   public String getLname() {
      return lname;
   }
   public void setLname(String lname) {
      this.lname = lname;
   }
   public int getAge() {
      return age;    }
```

```java
public void addNewStudent(int id, String fname, String lname, int age, int level, double GPA) {
   Student x = new Student( id, fname, lname, age, level, GPA);
   if (x.addStudent()) {
      System.out.println(x.toString() + "Added Successfully ... !");
   } else {
      System.out.println("Failed to insert ... !");
   }
}
public void displayStudents() {
   Student x = new Student();
   System.out.println(x.displayAllStudents());
}
public void searchForStudent(int id) {
   Student x = new Student();
   System.out.println(x.searchStudent(id));
}
public void updateStudent(int oldID, Student newStudentValues) {
   Student x = new Student();
   x.updateStudent(oldID, newStudentValues);
   System.out.println("Updated Successfully ... !");
}
public void deleteStudent(int Id) {
   Student x = new Student();
   x.deleteStudent(Id);
   System.out.println("deleted Successfully ... !");
}
}
```

Admin Class

FileManger Class

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Scanner;
public class FileManger {
    public boolean write(String Query, String FilePath, boolean appendType) {
        PrintWriter writter = null;
        try {
            System.out.print("\nwritting in ! " + FilePath);
            writter = new PrintWriter(new FileWriter(new File(FilePath), appendType));
            writter.println(Query);
            System.out.println(" ... Done ! ");
            return true;
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            writter.close();
        }
        return false;
    }
```

Go To binary Version

```
public ArrayList<Object> read(String FilePath) {
    Scanner Reader = null;
    try {
       System.out.println("Reading ! From " + FilePath);
       Reader = new Scanner(new File(FilePath));
    } catch (FileNotFoundException e) {
       System.out.println(e+" Cann't find file");
    }
       ArrayList<Student> Students = new ArrayList<Student>();
       Student x;
       while (Reader.hasNext()) {
          x = new Student();
          String Line = Reader.nextLine();
          String[] seprated = Line.split(",");
          x.setId(Integer.parseInt(seprated[0]));
          x.setFname(seprated[1]);
          x.setLname(seprated[2]);
          x.setAge(Integer.parseInt(seprated[3]));
          x.setLevel(Integer.parseInt(seprated[4]));
          x.setGPA(Double.parseDouble(seprated[5]));
          Students.add(x);
       }
       return (ArrayList<Object>) (Object) Students;
  }
}
```

FileManger Class

```java
import java.util.ArrayList;
public class Student {
   private int level;
   private int age;
   private double GPA;
   private String fname;
   private String lname;
   private int id;
   private final String studentFileName = "Students.txt";
   public static ArrayList<Student> Students = new ArrayList<Student>();
   private FileManger fileManger=new FileManger();
   public Student(){}
   public Student(int id, String fname, String lname, int age, int level, double GPA) {
      this.id=id;
      this.fname=fname;
      this.lname=lname;
      this.age=age;
      this.level = level;
      this.GPA = GPA;
   }
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getFname() {
      return fname;
   }
public void setFname(String fname) {
      this.fname = fname;
   }
   public String getLname() {
      return lname;
   }
public void setLname(String lname) {
   this.lname = lname;
}
public void setLevel(int level) {
   this.level = level;
}

public int getAge() {
   return age;
}

public void setAge(int age) {
   this.age = age;
}

public void setGPA(double GPA) {
   this.GPA = GPA;
}

public int getLevel() {
   return this.level;
}

public double getGPA() {
   return this.GPA;
}
```

Student Class

```
public boolean addStudent() {
    if (fileManger.write(getStudentData(), studentFileName, true)) {
        return true;
    } else {
        return false;
    }
}

private String getStudentData() {
    return this.id + "," + this.fname + "," + this.lname + "," + this.age + "," + this.level + "," + this.GPA;
}

private void commitToFile() {
    fileManger.write(Students.get(0).getStudentData(), studentFileName, false);
    for (int i = 1; i < Students.size(); i++) {
        fileManger.write(Students.get(i).getStudentData(), studentFileName, true);
    }

}
```

Student Class

Go To binary Version

```
private int getStudentIndex(int id){
    for (int i = 0; i < Students.size(); i++)
        if(Students.get(i).getId() == id)
            return i;

    return -1;
}

private void loadFromFile() {
    Students = (ArrayList<Student>) (Object) fileManger.read(studentFileName);
}

public String displayAllStudents() {
    loadFromFile();
    String S = "\nAll Student Data:\n";
    for (Student x : Students) {
        S = S + x.toString();
    }
    return S;
}
```

Student Class

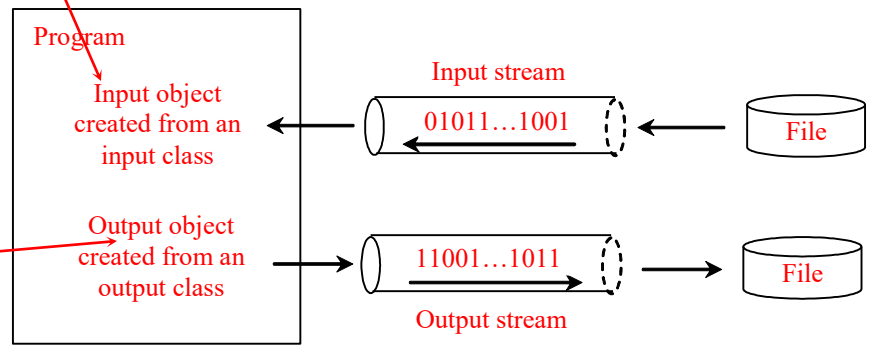
```
public String searchStudent(int id){
    loadFromFile();
    int index = getStudentIndex(id);
    if(index != -1)
        return "\nFound ...!" + Students.get(index).toString();
    else
        return "\nNot Found ...!";
  }
  public void updateStudent(int oldID, Student x){
    loadFromFile();
    int index = getStudentIndex(oldID);
    Students.set(index, x);
    commitToFile();
  }
  public void deleteStudent(int id){
    loadFromFile();
    int index = getStudentIndex(id);
    Students.remove(index);
    commitToFile();
  }
  @Override
  public String toString() {
    return "\nI'm Eng : " + fname + " " + lname + "\n" + "ID : " + id + " Age : " + age + "\n" + "Level : " + level + " GPA : " + GPA+ "\n";
  }
}
```

Student Class

How is I/O Handled in Java?

- A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes.

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```



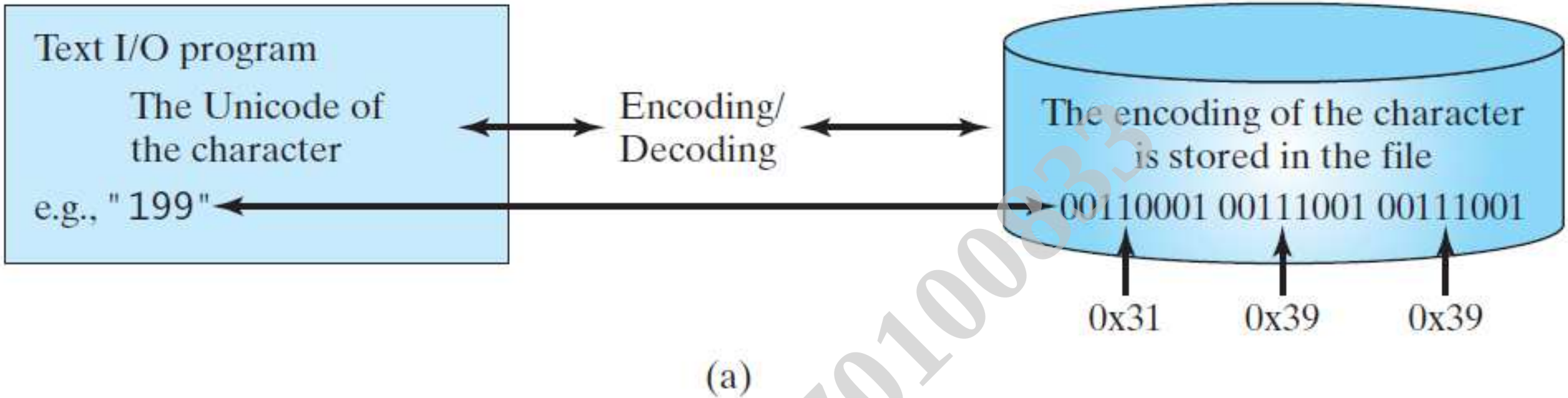

```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```

Text File vs. Binary File

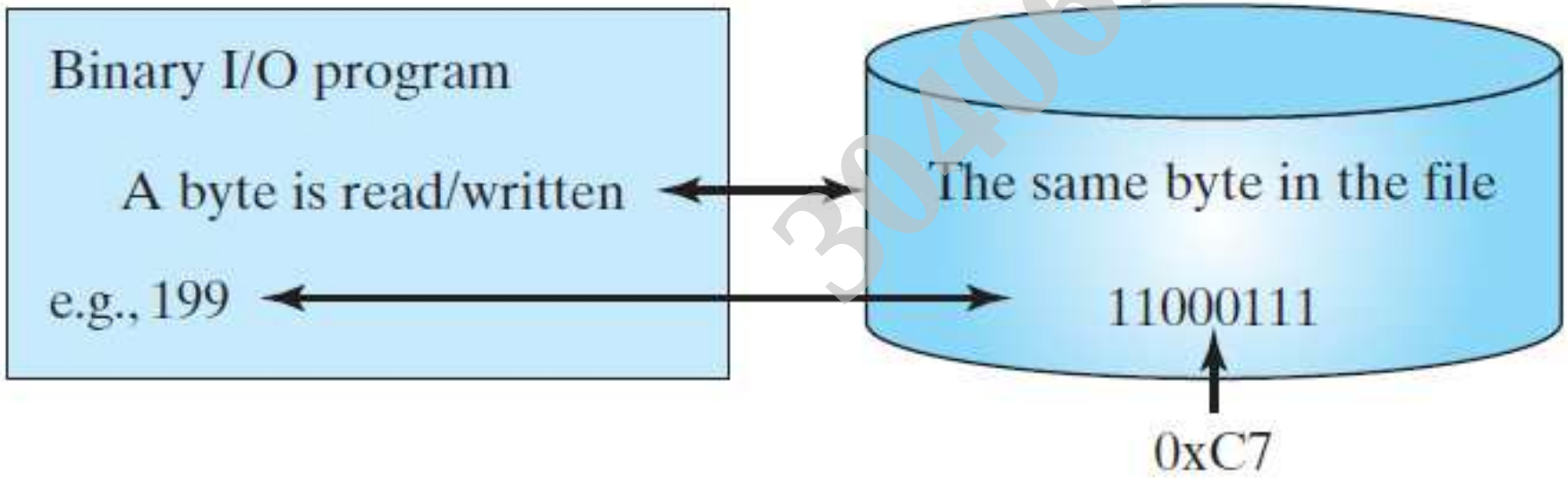
- Data stored in a text file are represented in human-readable form.
- Data stored in a binary file are represented in binary form. You cannot read binary files. Binary files are designed to be read by programs.
- For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.
- Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters and a binary file consists of a sequence of bits. For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.

Binary I/O

- Text I/O requires encoding and decoding.
- The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character.
- Binary I/O does not require conversions. When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.

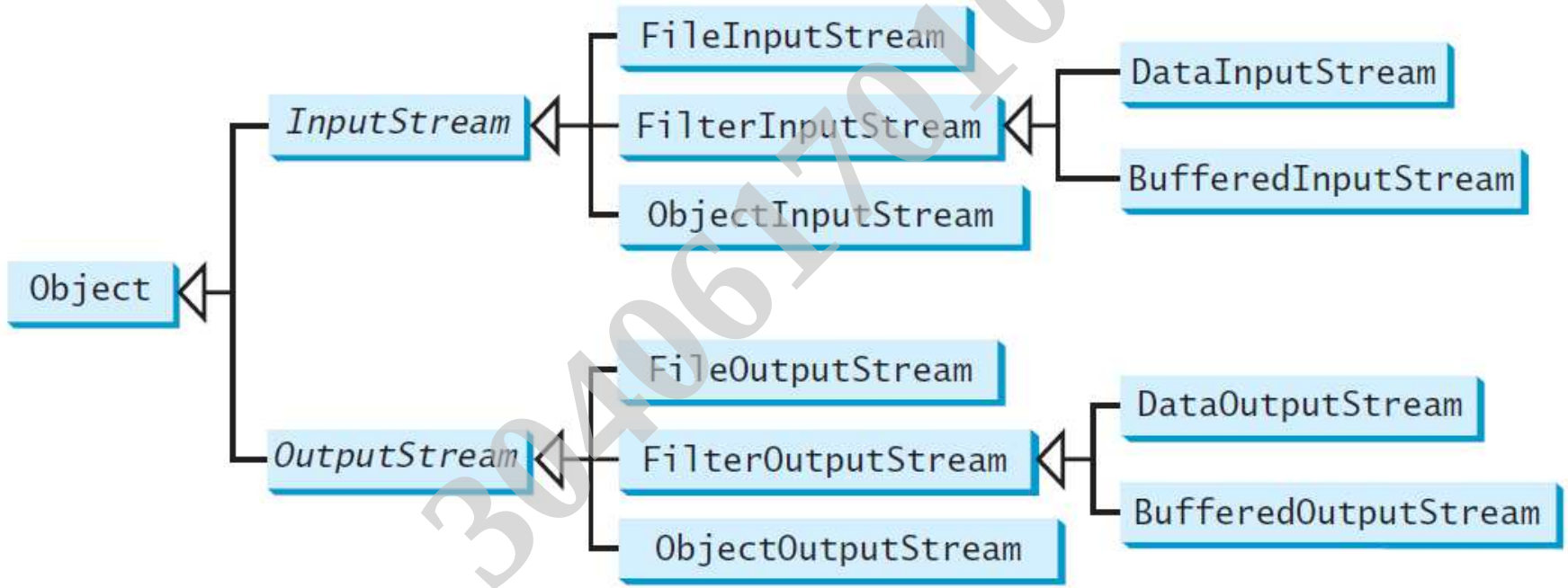


(a)



(b)

Binary I/O Classes



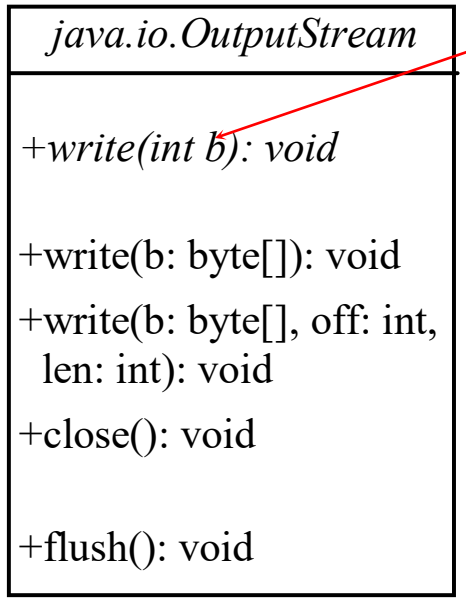
InputStream

The value returned is a byte as an int type.

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

OutputStream

The value is a byte as an int type.



Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.

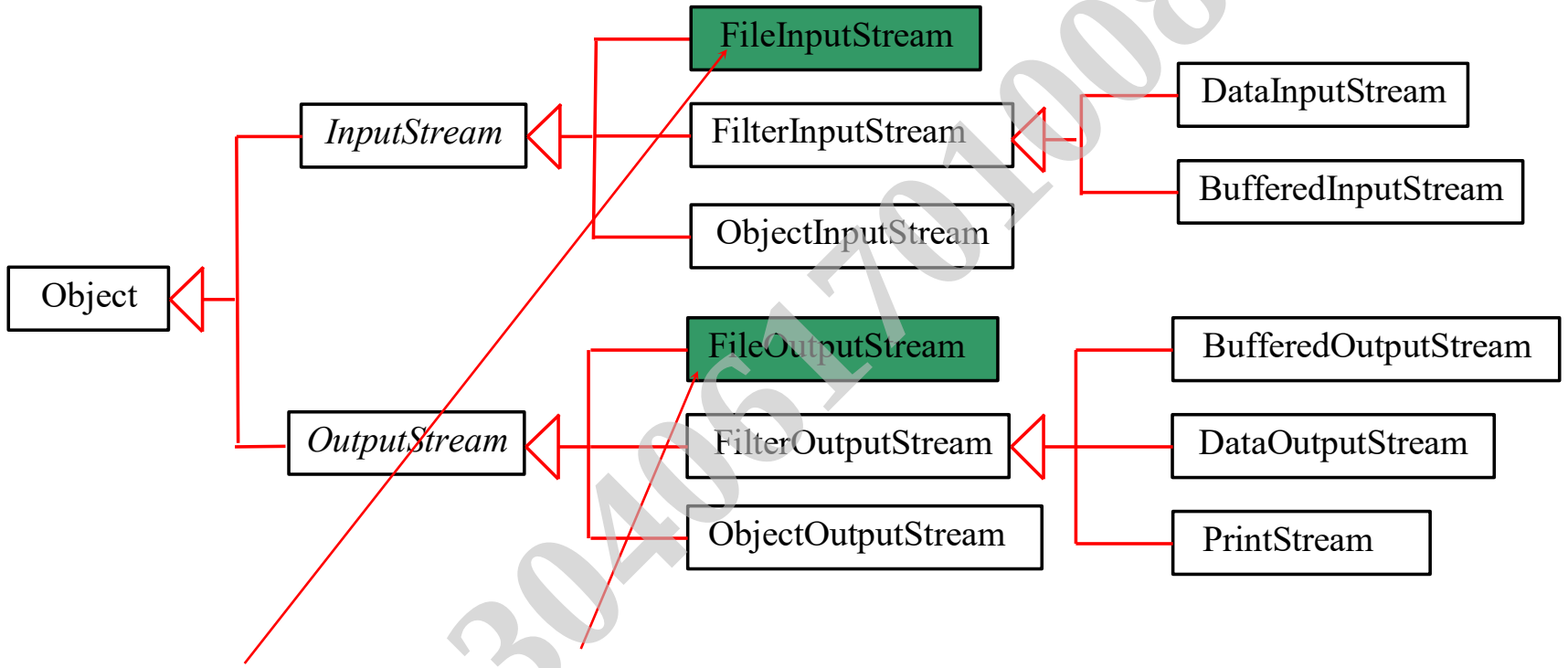
Writes all the bytes in array b to the output stream.

Writes b[off], b[off+1], …, b[off+len-1] into the output stream.

Closes this output stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream/FileOutputStream



FileInputStream/FileOutputStream associates a binary input/output stream with an external file. All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses.

FileInputStream

To construct a FileInputStream, use the following constructors:

public FileInputStream(String filename)

public FileInputStream(File file)

A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.

FileOutputStream

To construct a FileOutputStream, use the following constructors:

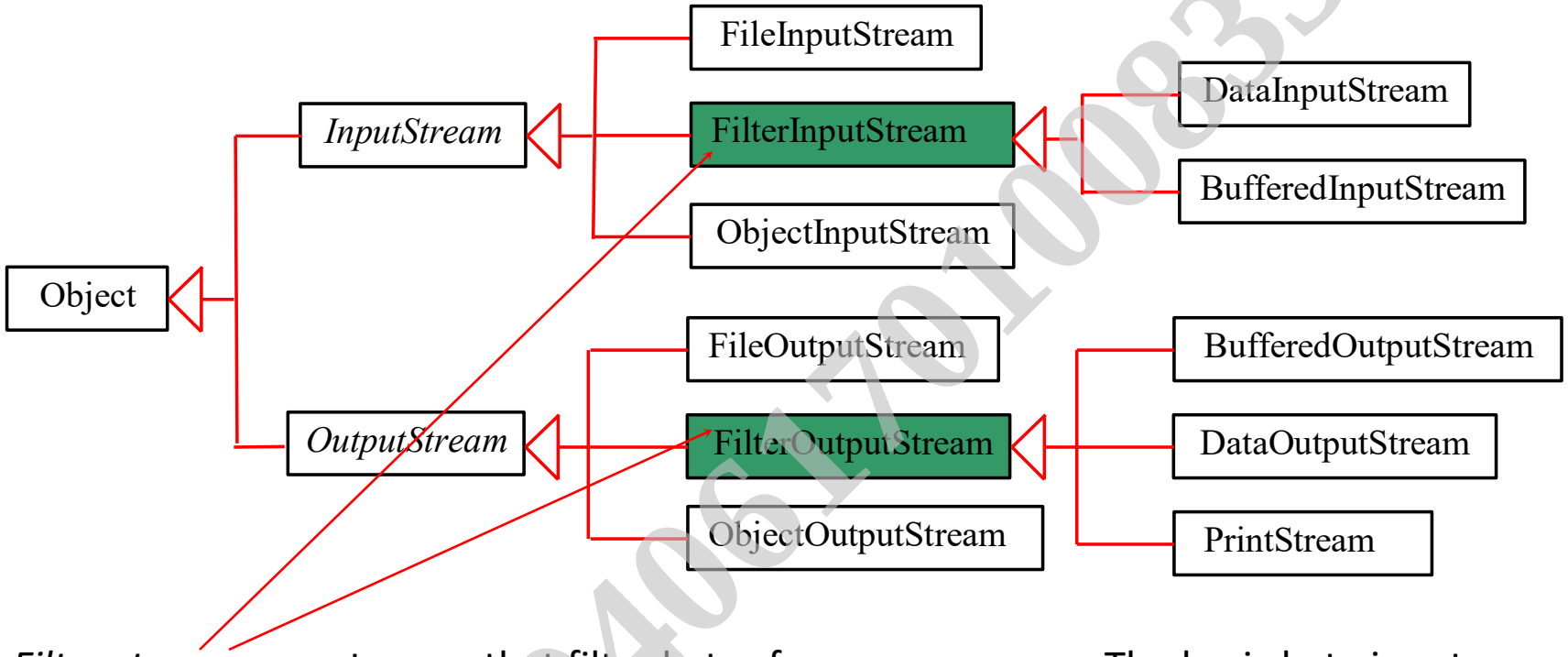
```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)
```

TestFileStream Run

If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

```java
import java.io.*;
public class TestFileStream {
  public static void main(String[] args) throws IOException {
    try (
      // Create an output stream to the file
      FileOutputStream output = new
FileOutputStream("temp.dat");
    ) {
      // Output values to the file
      for (int i = 1; i <= 10; i++)
        output.write(i);
    }
    try (
      // Create an input stream for the file
      FileInputStream input = new FileInputStream("temp.dat");
    ) {
      // Read values from the file
      int value;
      while ((value = input.read()) != -1)
        System.out.print(value + " ");
    }
  }
}
```

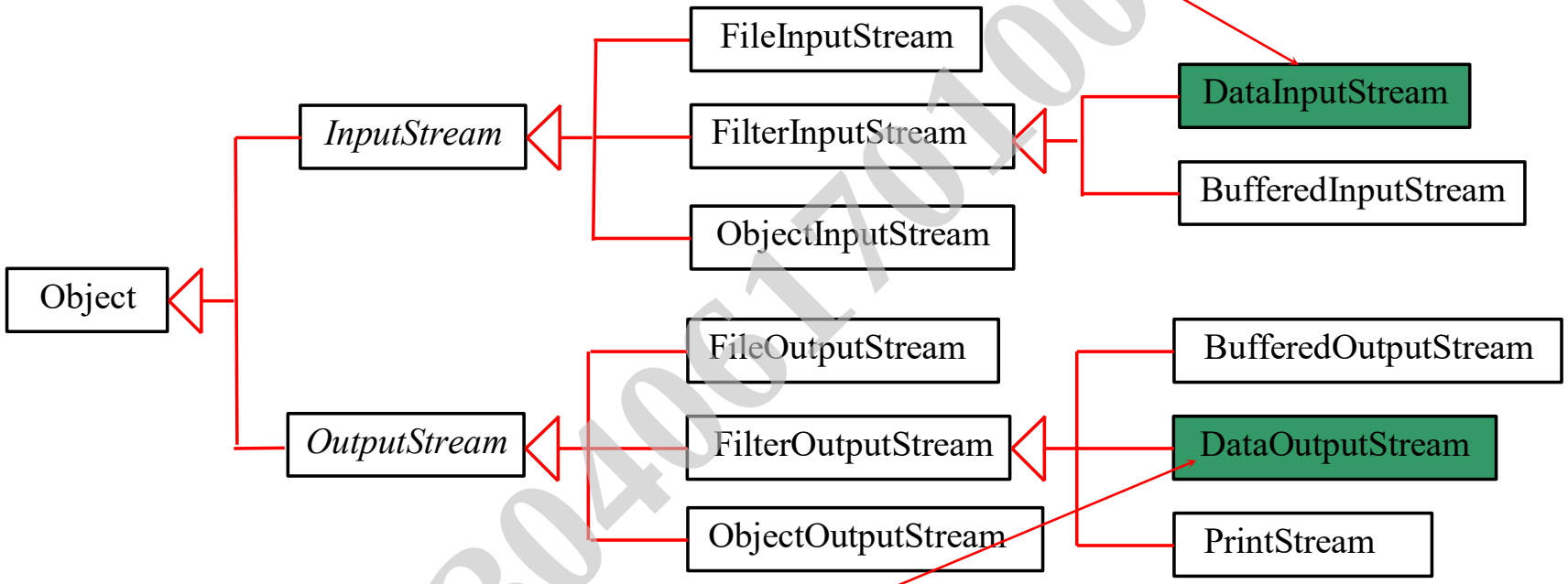
FilterInputStream/FilterOutputStream



Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. FilterInputStream and FilterOutputStream are the base classes for filtering data. When you need to process primitive numeric types, use DatInputStream and DataOutputStream to filter bytes.

DataInputStream/DataOutputStream

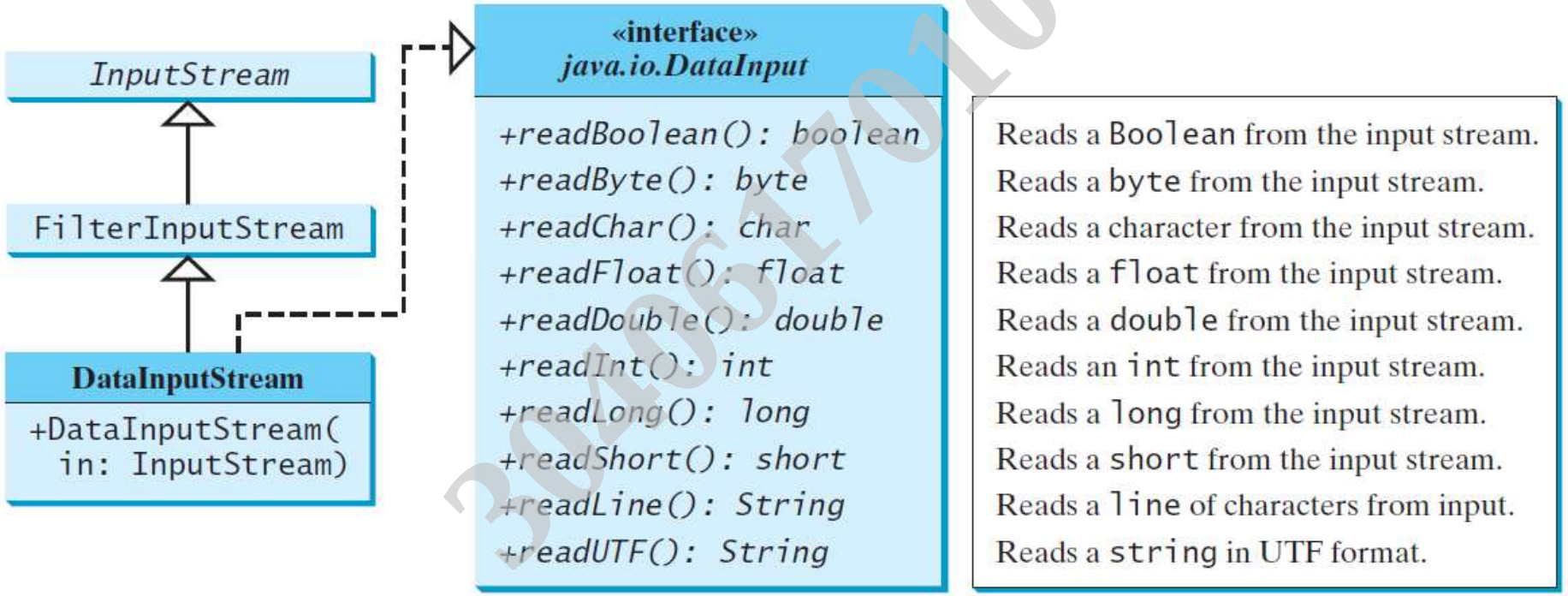
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

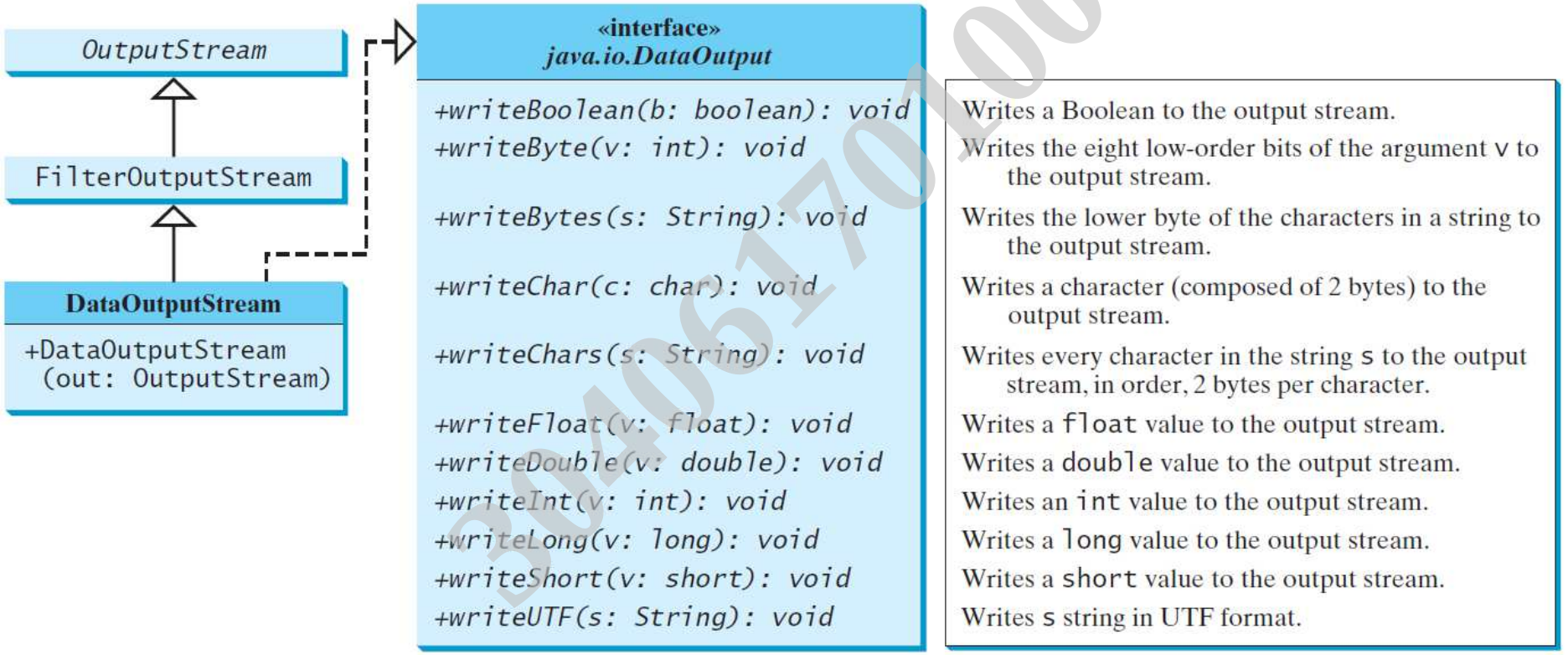
DataInputStream

DataInputStream extends FilterInputStream and implements the DataInput interface.



DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



What is UTF

- **UTF** in Java refers to **Unicode Transformation Format**, which is a standard for encoding characters into a sequence of bytes. Java uses UTF encoding for character representation, particularly **UTF-8** and **Modified UTF-8**, in its various classes and APIs.
- **UTF-8**:
  - A variable-length encoding that represents every Unicode character.
  - It uses 1 to 4 bytes per character.
  - Java supports UTF-8 for reading and writing text files and network communication.
  - Widely used because of its backward compatibility with ASCII (characters in the range 0-127 use one byte).

What is UTF

- Modified UTF-8:
  - A variation of UTF-8 used internally in Java for serialization (e.g., in DataInput and DataOutput streams).
  - Differs from standard UTF-8 in two ways:It uses two bytes for null (\u0000) instead of one.
  - It encodes surrogate pairs directly as three bytes, unlike UTF-8, which encodes them as six bytes.
  - Used for efficiency in JVM-internal operations like .class file representation.
  - Example: In a serialized form or when using Java DataInputStream and DataOutputStream

Where is UTF Used in Java?

- String Encoding and Decoding:
  - String.getBytes() and new String(byte[], Charset) support UTF encoding.
- File I/O:
  - Classes like InputStreamReader, OutputStreamWriter, and Files use UTF-8 by default in many cases.
- Serialization:
  - Modified UTF-8 is used in DataInputStream and DataOutputStream

Why UTF-8? What is UTF-8?

- UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently. Most operating systems use ASCII. Java uses Unicode.

1.Networking: Protocols and web services often rely on UTF-8 for text-based communication

- The ASCII character set is a subset of the Unicode character set. Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character. The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes.
- ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.

Characters and Strings in Binary I/O

- A Unicode consists of two bytes. The writeChar(char c) method writes the Unicode of character c to the output. The writeChars(String s) method writes the Unicode for each character in the string s to the output.
- The writeBytes(String s) method writes the lower byte of the Unicode for each character in the string s to the output. The high byte of the Unicode is discarded. The writeBytes method is suitable for strings that consist of ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode. If a string consists of non-ASCII characters, you have to use the writeChars method to write the string.
- The writeUTF(String s) method writes a string using the UTF coding scheme. UTF is efficient for compressing a string with Unicode characters.

Using DataInputStream/DataOutputStream

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

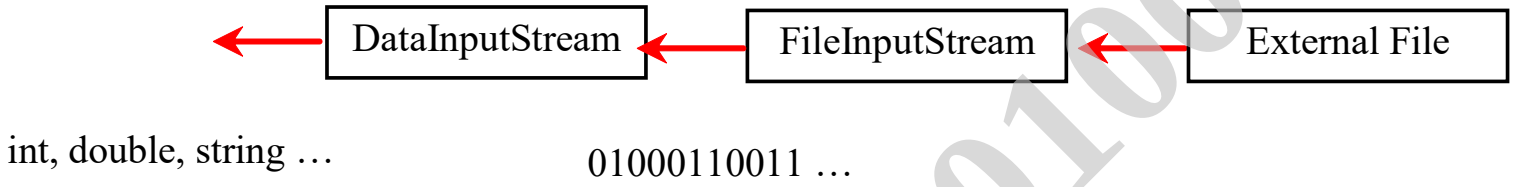
```
DataInputStream infile =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

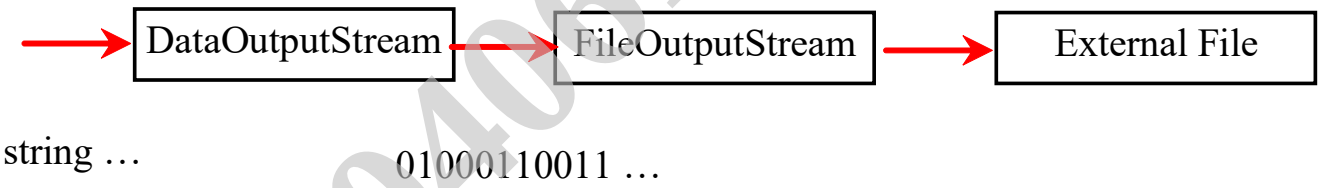
TestDataStream Run

```java
import java.io.*;
public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file temp.dat
      DataOutputStream output =
        new DataOutputStream(new FileOutputStream("temp.dat"));
    ) {
      // Write student test scores to the file
      output.writeUTF("Liam");
      output.writeDouble(85.5);
      output.writeUTF("Susan");
      output.writeDouble(185.5);
      output.writeUTF("Chandra");
      output.writeDouble(105.25);
    }
    try ( // Create an input stream for file temp.dat
      DataInputStream input =
        new DataInputStream(new FileInputStream("temp.dat"));
    ) {
      // Read student test scores from the file
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
    }  }}
```

Liam 85.5
Susan 185.5
Chandra 105.25

Concept of pipeline





Order and Format

- CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.
- Checking End of File
- TIP: If you keep reading data at the end of a stream, an EOFException would occur. So how do you check the end of a file? You can use input.available() to check it. input.available() == 0 indicates that it is the end of a file.

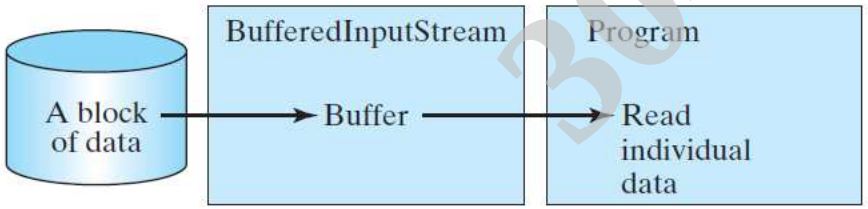
BufferedInputStream/BufferedOutputStream

- can be used to speed up input and output by reducing the number of disk reads and writes.
- Using BufferedInputStream, the whole block of data on the disk is read into the buffer in the memory once.
- The individual data are then loaded to your program from the buffer.
- Using BufferedOutputStream, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk once

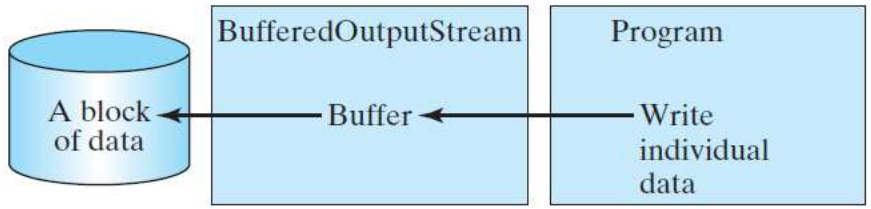
Constructing BufferedInputStream/BufferedOutputStream

```
// Create a BufferedInputStream
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStreamr out, int bufferSize)
```

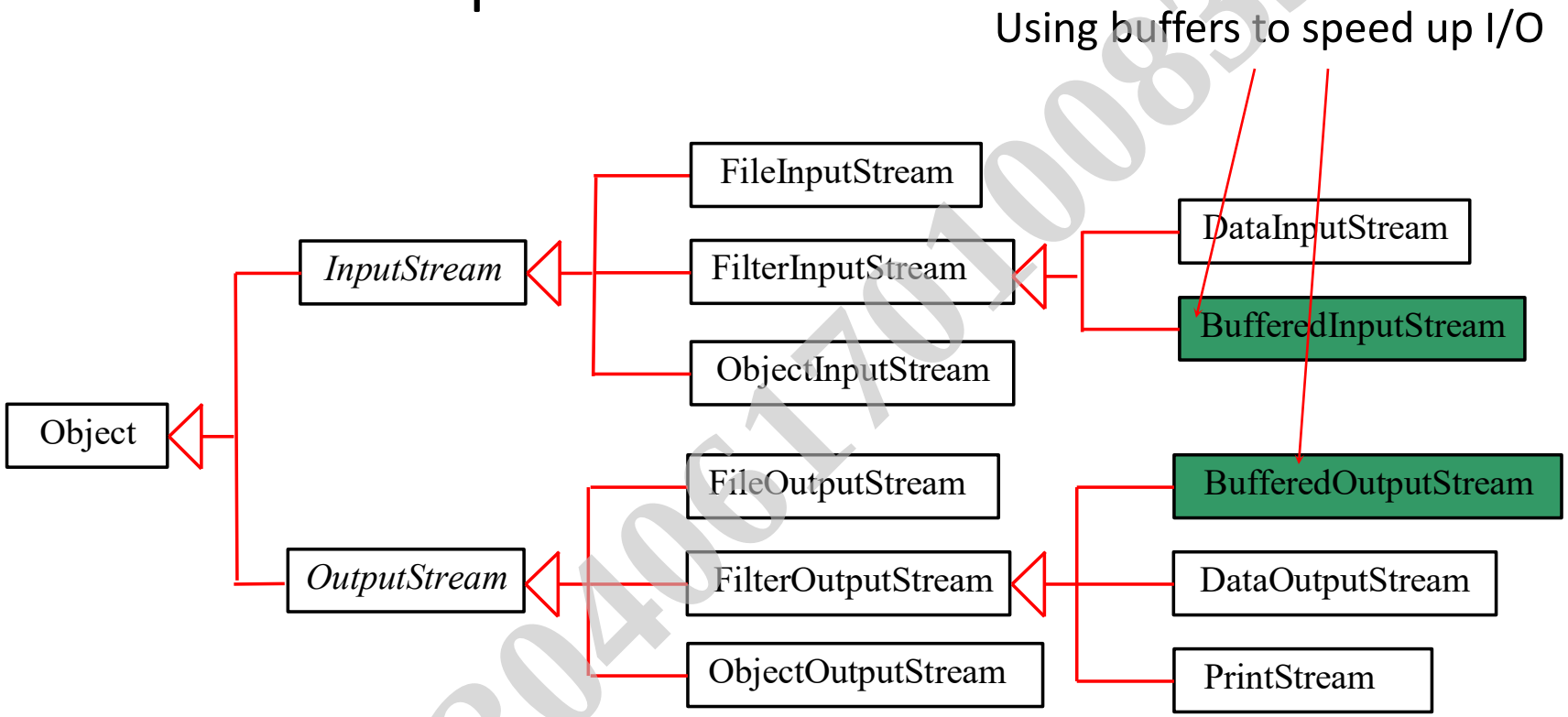


(a)



(b)

BufferedInputStream/ BufferedOutputStream

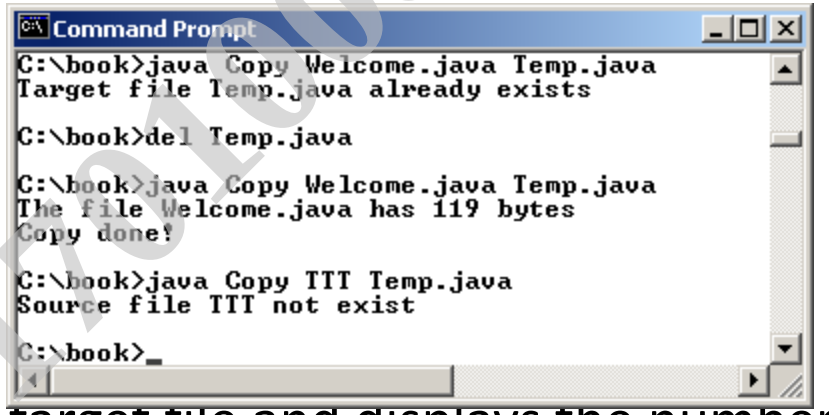


BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

Case Studies: Copy File

This case study develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

java Copy source target



The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, tell the user the file is not found. If the target file already exists, tell the user the file already exists.

Copy Run

```
c:\book>java Copy Welcome.java Temp.java
179 bytes copied
```

```java
import java.io.*;
public class Copy {
  /** Main method
     @param args[0] for sourcefile
     @param args[1] for target file    */
  public static void main(String[] args) throws IOException {
    // Check command-line parameter usage
    if (args.length != 2) {
      System.out.println(
        "Usage: java Copy sourceFile targetfile");
      System.exit(1);
    }
    // Check if source file exists
    File sourceFile = new File(args[0]);
    if (!sourceFile.exists()) {
       System.out.println("Source file " + args[0] + " does not exist");
       System.exit(2);
    }
    // Check if target file exists
    File targetFile = new File(args[1]);
    if (targetFile.exists()) {
      System.out.println("Target file " + args[1]  + " already exists");
      System.exit(3);
    }
```

```
  try (
      // Create an input stream
      BufferedInputStream input =
        new BufferedInputStream(new FileInputStream(sourceFile));

      // Create an output stream
      BufferedOutputStream output =
        new BufferedOutputStream(new FileOutputStream(targetFile));
    ) {
      // Continuously read a byte from input and write it to output
      // The input value of -1 signifies the end of a file.
      int r, numberOfBytesCopied = 0;
      while ((r = input.read()) != -1) {
        output.write((byte)r);
        numberOfBytesCopied++;
      }

      // Display the file size
      System.out.println(numberOfBytesCopied + " bytes copied");
    }
  }
}
```

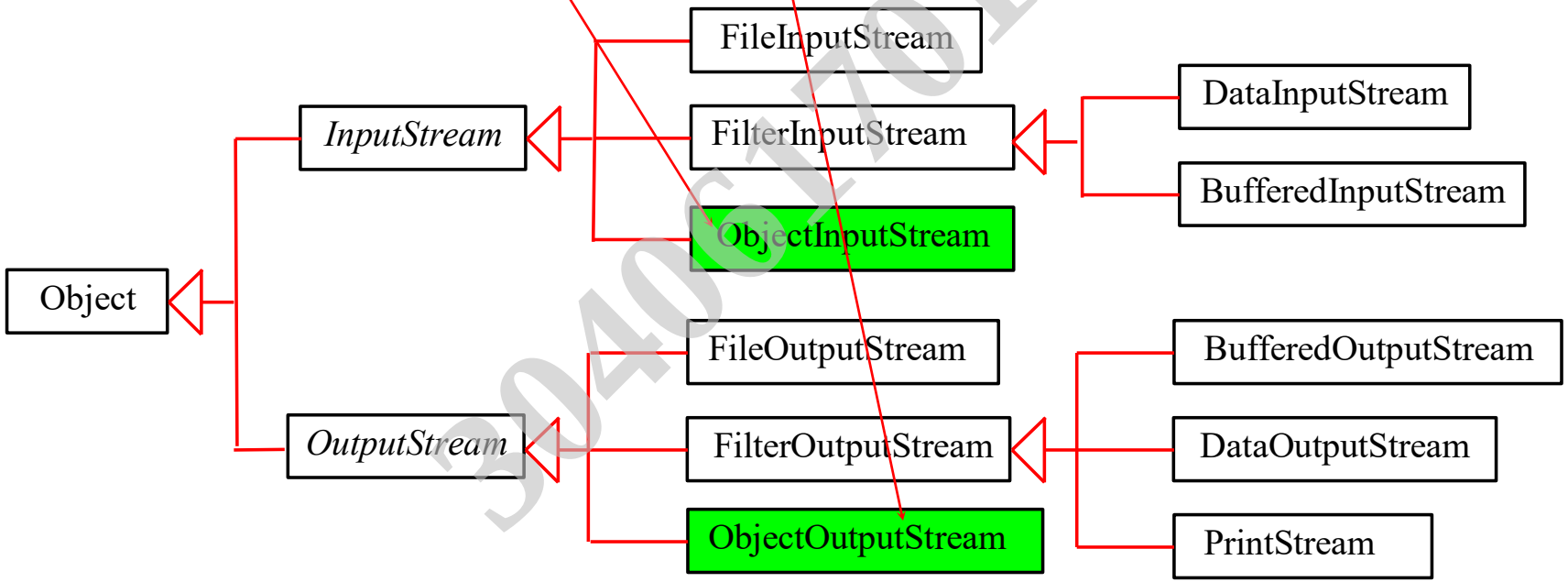
InputStream

The value returned is a byte as an int type.

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

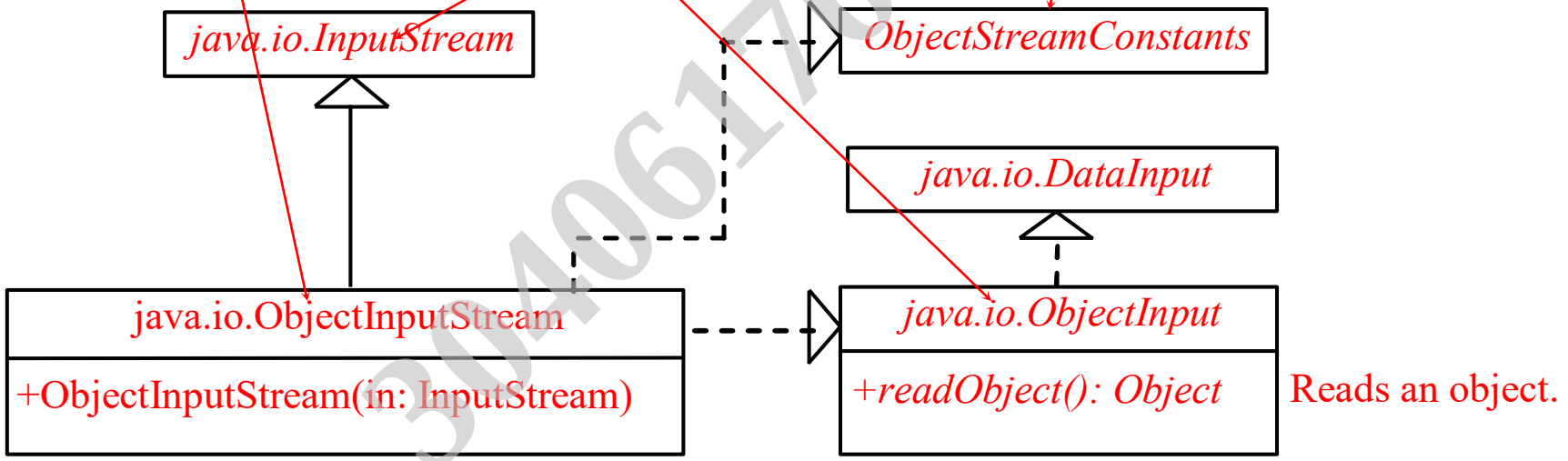
Object I/O

DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings.
ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition for primitive type values and strings.



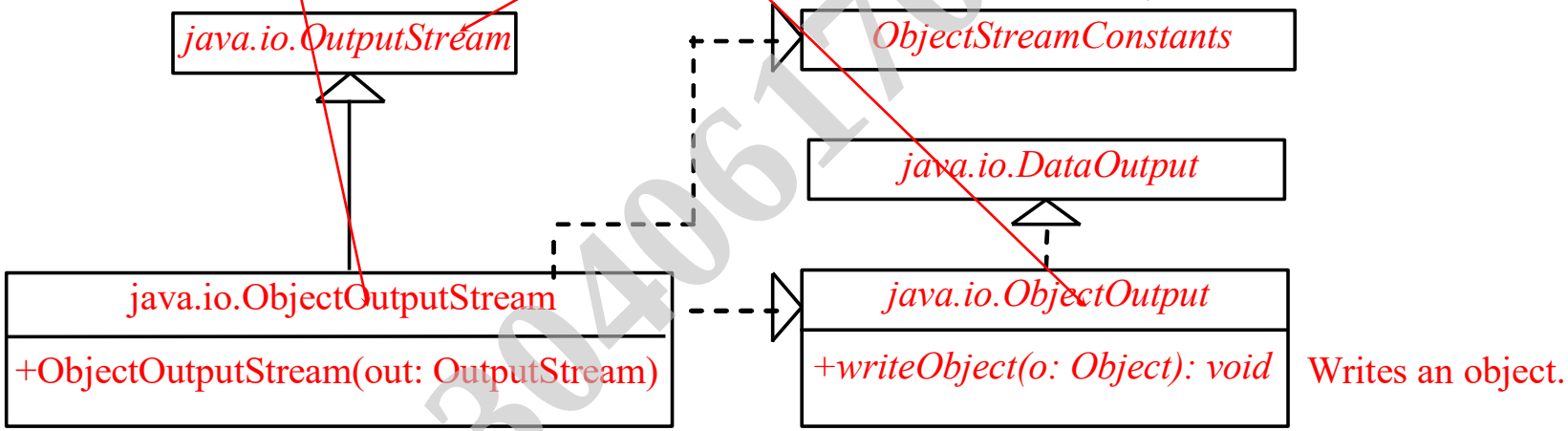
ObjectInputStream

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



ObjectOutputStream

ObjectOutputStream extends OutputStream and implements ObjectOutput and ObjectStreamConstants.



Using Object Streams

You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

TestObjectOutputStream Run

TestObjectInputStream Run

```java
import java.io.*;

public class TestObjectOutputStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file object.dat
      ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat"));
    ) {
      // Write a string, double value, and object to the file
      output.writeUTF("Jamal");
      output.writeDouble(85.5);
      output.writeObject(new java.util.Date());
    }
  }
}
```

```
1 2 3 4 5
John Susan Kim
```

```java
import java.io.*;

public class TestObjectInputStream {
  public static void main(String[] args)
    throws ClassNotFoundException, IOException {
    try ( // Create an input stream for file object.dat
      ObjectInputStream input =
        new ObjectInputStream(new FileInputStream("object.dat"));
    ) {
      // Read a string, double value, and object from the file
      String name = input.readUTF();
      double score = input.readDouble();
      java.util.Date date = (java.util.Date)(input.readObject());
      System.out.println(name + " " + score + " " + date);
    }
  }
}
```

```
John 85.5 Sun Dec 04 10:35:31 EST 2011
```

The `Serializable` Interface

- Not all objects can be written to an output stream. Objects that can be written to an object stream is said to be *serializable*. A serializable object is an instance of the java.io.Serializable interface. So the class of a serializable object must implement Serializable.
- The Serializable interface is a marker interface. It has no methods, so you don't need to add additional code in your class that implements Serializable.
- Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

The `Serializable` Interface

- To appreciate this automation feature, consider what you otherwise need to do in order to store an object. Suppose that you wish to store an ArrayList object. To do this, you need to store all the elements in the list.
- Each element is an object that may contain other objects. As you can see, this would be a very tedious process. Fortunately, you don't have to go through it manually.
- Java provides a built-in mechanism to automate the process of writing objects. This process is referred as object serialization, which is implemented in ObjectOutputStream. In contrast, the process of reading objects is referred as object deserialization, which is implemented in ObjectInputStream.

The `Serializable` Interface

- Many classes in the Java API implement Serializable. All the wrapper classes for primitivetype values, **java.math.BigInteger, java.math.BigDecimal, java.lang.String, java.lang.StringBuilder, java.lang.StringBuffer, java.util.Date, and java.util.ArrayList implement java.io.Serializable.**
- Attempting to store an object that does not support the Serializable interface would cause a NotSerializableException.
- To make the object is **`Serializable , its class must implement the Serializable interface`**
- **Note that the** in Java static field are not serialized and The values of the object's static variables are not stored..

The `transient` Keyword

- If an object is an instance of Serializable, but it contains non-serializable instance data fields, can the object be serialized?
- The answer is no. To enable the object to be serialized, you can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream.

The `transient` Keyword

- When we de-serialized an object only, instance variables are saved and will have same values after the process.
- Transient variables – The values of the transient variables are never considered (they are excluded from the serialization process). i.e. When we declare a variable transient, after de-serialization its value will always be null, false, or, zero (default value).
- Static variables – The values of static variables will not be preserved during the de-serialization process. In-fact static variables are also not serialized but since these belongs to the class. After de-serialization they get their current values from the class.

The `transient` Keyword, cont.

Consider the following class:

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}
class A { } // A is not serializable
```

When an object of the Foo class is serialized, only variable v1 is serialized. Variable v2 is not serialized because it is a static variable, and variable v3 is not serialized because it is marked transient. If v3 were not marked transient, a java.io.NotSerializableException would occur.

```
class Student implements Serializable{
    private String name;
    private transient int age;
    private static int year = 2018;
    public Student(){
        System.out.println("This is a constructor");
        this.name = "Krishna";
        this.age = 25;
    }
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
    public void display() {
        System.out.println("Name: "+this.name);
        System.out.println("Age: "+this.age);
        System.out.println("Year: "+Student.year);
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setYear(int year) {
        Student.year = year;
    }
}
```

```
public class SerializeExample{
   public static void main(String args[]) throws Exception{
       Student std = new Student("Vani", 27);
      //Serializing the object
      FileOutputStream fos = new FileOutputStream("e:\student.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(std);
      oos.close();  fos.close();
      //Printing the data before de-serialization
      System.out.println("Values before de-serialization");
      std.display();
      std.setYear(2019);   //Changing the static variable value
      std.setName("Varada"); //Changing the instance variable value
      std.setAge(19); //Changing the transient variable value
      System.out.println("Object serialized.......");
      //De-serializing the object
      FileInputStream fis = new FileInputStream("e:\student.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      Student deSerializedStd = (Student) ois.readObject();
      System.out.println("Object de-serialized.......");
      ois.close();  fis.close();
      System.out.println("Values after de-serialization");
      deSerializedStd.display();
   }}
```

Output

- Values before de-serialization:
- Name: Vani
- Age: 27
- Year: 2018
- Object serialized.....
- Object de-serialized.....
- Values after de-serialization:
- Name: Vani
- Age: 0
- Year: 2019

After the process the value of the instance variables will be same. The transient variables display the default values, and the static variables print the new (current) value from the class.

Serializing Arrays

- An array is serializable if all its elements are serializable. So an entire array can be saved using writeObject into a file and later restored using readObject. Here is an example that stores an array of five int values and an array of three strings, and reads them back to display on the console.

TestObjectStreamForArray Run

```java
import java.io.*;
public class TestObjectStreamForArray {
  public static void main(String[] args)
      throws ClassNotFoundException, IOException {
    int[] numbers = {1, 2, 3, 4, 5};
    String[] strings = {"John", "Susan", "Kim"};
    try ( // Create an output stream for file array.dat
      ObjectOutputStream output = new ObjectOutputStream(new
        FileOutputStream("array.dat", true));
    ) {
      // Write arrays to the object output stream
      output.writeObject(numbers);
      output.writeObject(strings);
    }
 try ( // Create an input stream for file array.dat
ObjectInputStream input =new ObjectInputStream(new FileInputStream("array.dat"));
) {
      int[] newNumbers = (int[])(input.readObject());
      String[] newStrings = (String[])(input.readObject());
        // Display arrays
      for (int i = 0; i < newNumbers.length; i++)
        System.out.print(newNumbers[i] + " ");
      System.out.println();
        for (int i = 0; i < newStrings.length; i++)
        System.out.print(newStrings[i] + " ");
    }
  }
}
```

Reimplementing The Case study in Slide 18 but using Binary Objects

```
public class FileMangerBinary implements Serializable {
    public boolean write(String FilePath, Object data) {
        try {
            System.out.print("\nwritting in ! " + FilePath);

            ObjectOutputStream writter = new ObjectOutputStream(new
                                                FileOutputStream(FilePath));

            writter.writeObject(data);

            System.out.println(" ... Done ! ");
            writter.close();
            return true;

        } catch (IOException e) {
            System.out.println("Can't write ...!\n" + e);
        }
        return false;
    }
```

FileManger Class

Go To text Version

FileManger Class

```
public Object read(String FilePath) {
        Object Result = null;
        try {
            System.out.println("Reading ! From " + FilePath);
            ObjectInputStream Reader = new ObjectInputStream(new
                                FileInputStream(FilePath));
            Result = Reader.readObject();
        } catch (IOException e) {
            System.out.println(e);
        }
        return Result;
    }
}
```

```
public boolean addStudent() {
        loadFromFile();
        Students.add(this);
        return commitToFile();
    }

    public boolean commitToFile() {
        return FManger.write(studentFileName, Students);
    }

    public void loadFromFile() {
        Students = (ArrayList<Student>) FManger.read(studentFileName);
    }
```

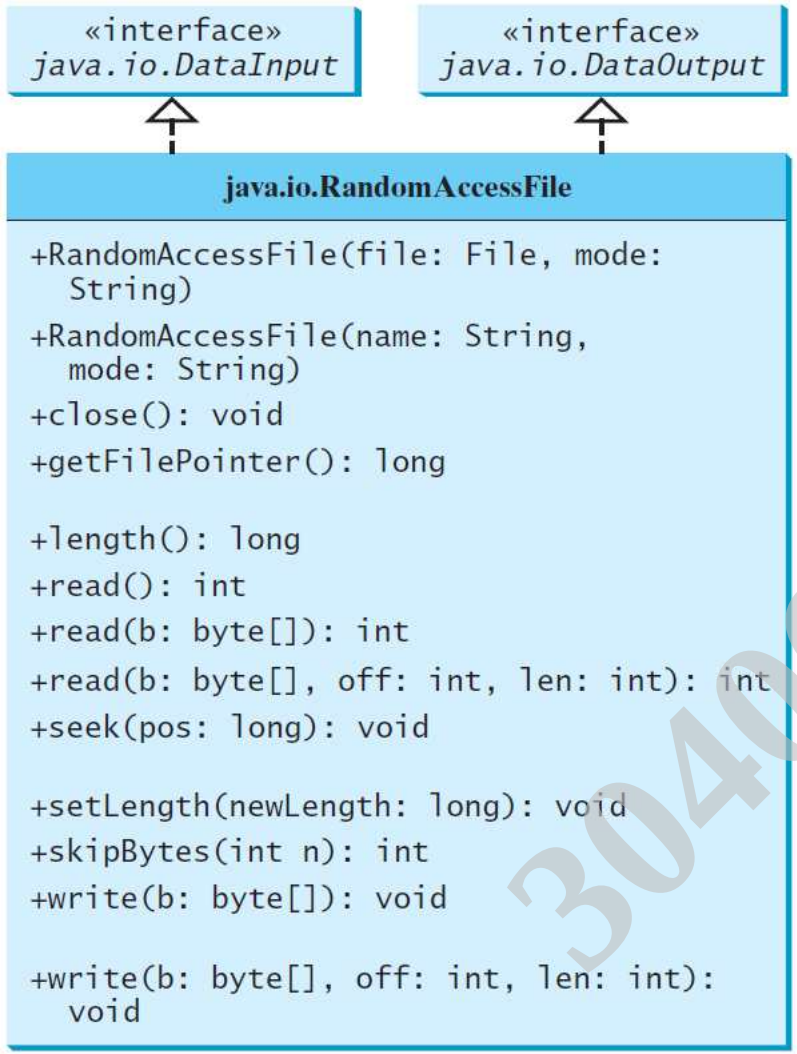
Some Methods from Student Class

Go To text Version

Random Access Files

- All of the streams you have used so far are known as *read-only* or *write-only* streams.
- The external files of these streams are *sequential* files that cannot be updated without creating a new file.
- It is often necessary to modify files or to insert new records into files.
- Java provides the RandomAccessFile class to allow a file to be read from and write to at random locations.

RandomAccessFile

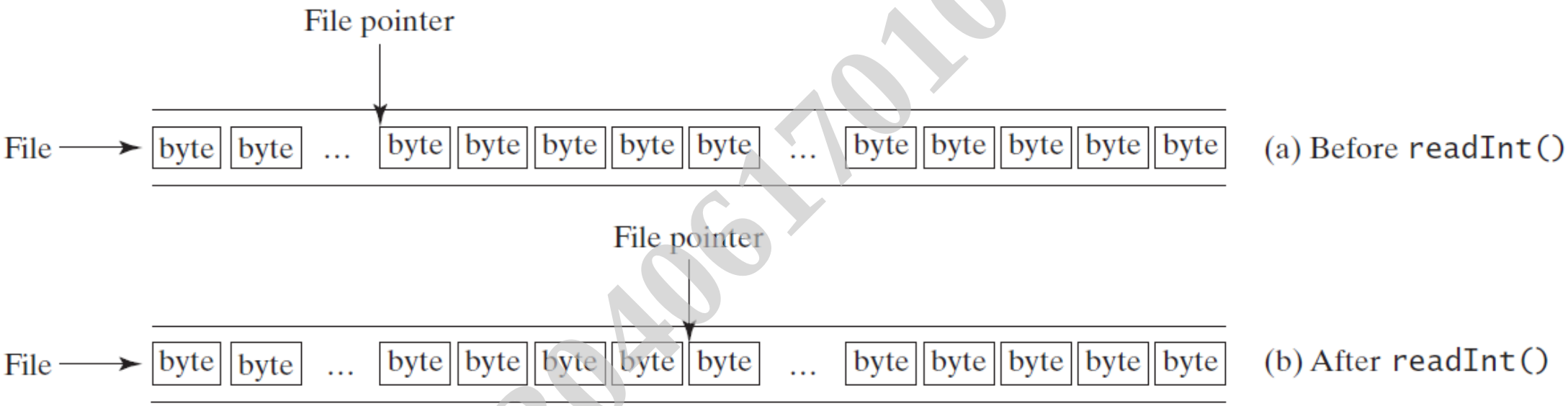


`+RandomAccessFile(file: File, mode: String)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`+RandomAccessFile(name: String, mode: String)`	Creates a `RandomAccessFile` stream with the specified file name string and mode.
`+close(): void`	Closes the stream and releases the resource associated with it.
`+getFilePointer(): long`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`+length(): long`	Returns the number of bytes in this file.
`+read(): int`	Reads a byte of data from this file and returns –1 at the end of stream.
`+read(b: byte[]): int`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`+read(b: byte[], off: int, len: int): int`	Reads up to `len` bytes of data from this file into an array of bytes.
`+seek(pos: long): void`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`+setLength(newLength: long): void`	Sets a new length for this file.
`+skipBytes(int n): int`	Skips over `n` bytes of input.
`+write(b: byte[]): void`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`+write(b: byte[], off: int, len: int): void`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

RandomAccessFile

- A random access file consists of a sequence of bytes. There is a special marker called *file pointer* that is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer sets at the beginning of the file.
- When you read or write data to the file, the file pointer moves forward to the next data.
- For example, if you read an int value using readInt(), the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.

File Pointer



`RandomAccessFile` Methods

- Many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`. For example, `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()`, and `writeLong()` can be used in data input stream or data output stream as well as in `RandomAccessFile` streams.

`RandomAccessFile` Methods, cont.

```
void seek(long pos) throws IOException;
```

Sets the offset from the beginning of the `RandomAccessFile` stream to where the next read or write occurs.

```
long getFilePointer() IOException;
```

Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.

`RandomAccessFile` Methods, cont.

`long length()IOException`

Returns the length of the file.

`final void writeChar(int v) throws IOException`

Writes a character to the file as a two-byte Unicode, with the high byte written first.

`final void writeChars(String s) throws IOException`

Writes a string to the file as a sequence of characters.

`RandomAccessFile` Constructor

```
RandomAccessFile raf =
 new RandomAccessFile("test.dat", "rw"); //
 allows read and write
```

```
RandomAccessFile raf =
 new RandomAccessFile("test.dat", "r"); // read
 only
```

A Short Example on RandomAccessFile

```java
import java.io.*;

public class TestRandomAccessFile {
  public static void main(String[] args) throws IOException {
    try ( // Create a random access file
      RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
    ) {
      // Clear the file to destroy the old contents if exists
      inout.setLength(0);

      // Write new integers to the file
      for (int i = 0; i < 200; i++)
        inout.writeInt(i);

      // Display the current length of the file
      System.out.println("Current file length is " + inout.length());

      // Retrieve the first number
      inout.seek(0); // Move the file pointer to the beginning
      System.out.println("The first number is " + inout.readInt());
        // Retrieve the second number
      inout.seek(1 * 4); // Move the file pointer to the second number
      System.out.println("The second number is " + inout.readInt());
```

```
// Retrieve the tenth number
        inout.seek(9 * 4); // Move the file pointer to the tenth number
        System.out.println("The tenth number is " + inout.readInt());
          // Modify the eleventh number
        inout.writeInt(555);

        // Append a new number
        inout.seek(inout.length()); // Move the file pointer to the end
        inout.writeInt(999);

        // Display the new length
        System.out.println("The new length is " + inout.length());

        // Retrieve the new eleventh number
        inout.seek(10 * 4); // Move the file pointer to the eleventh number
        System.out.println("The eleventh number is " + inout.readInt());
      }
    }
}
```

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

Thanks

References

- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- Tamer AbdElaziz Yassen, Free Object-Oriented Programming (OOP) Tutorial - Object Oriented Programming using Java in Arabic (Free) | Udemy
- This slides based on slides provided by Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang

- php - What is the advantage of using try {} catch {} versus if {} else {} - Stack Overflow