320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2014

Lecture 10: The Java I/O System

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Objectives

JACOBS UNIVERSITY

This lecture introduces the following

- Java Streams
- Object Serialization in Java
- File Management

Introduction

JACOBS UNIVERSITY

Storage of data in *variables* and *collections* is temporary

- Data is lost when a local variables goes out of scope or when the program terminates

- Computers use files for long term retention of large amounts of data

- Data that is maintained in files is called **persistent data**

- Computers store files on **secondary storage devices** such as hard disks

Introduction

JACOBS UNIVERSITY

Java I/O provides communication with devices

- Example devices are **files**, **console**, **networks, memory blocks** etc.

There are various types of communication

- Examples include: **sequential**, **random-access**, **binary**, **char**, **lines**, **words**, **objects**, ...

Java provides a "mix and match" solution based on:

- **Byte-oriented** I/O streams (ordered sequences of bytes)
- **Character-oriented** I/O streams (ordered sequences of characters)

Introduction

JACOBS UNIVERSITY

Input stream

- An object from which a sequence of bytes can be *read* (`InputStream`)

Output stream

- An object to which a sequence of bytes can be written (`OutputStream`)

Example

- System streams `System.in` (`out` & `err`) are available to all Java programs
  - `System.in` is an instance of the **BufferedInputStream** class
  - `System.out` is an instance of **PrintStream** class

I/O involves creating appropriate stream objects for your task

Reading Bytes

JACOBS UNIVERSITY

Streams *read* and *write* **8-bit** values to/from various data sources:

Examples of streams: **Files**, **Network connections**, **Memory Blocks**

There are more than 60 different stream types

Generality of Processing

- Files, network connections, memory blocks, etc are handled in the same way

Byte-Oriented vs Unicode-Oriented

JACOBS UNIVERSITY

Streams that input and output bytes to files are called **byte-oriented streams**

- The int value 5 would be stored using the binary format of 5:
  00000000 00000000 00000000 00000101

- The numeric value can be used as an **int** in calculations

- Files created using byte-oriented streams are called **binary files**

- Binary files are read by a program that converts the data to a human-readable format

Bytes-Oriented vs Unicode-Oriented

JACOBS UNIVERSITY

Streams that input and output characters to files are called **character-oriented (Unicode-oriented) streams**

- Ex. value 5 would be stored using the binary format of character 5 or 00000000 00110101 (character 5 in Unicode character set)
- The character 5 is a character that can be used in a string
- Files created using character-oriented streams are called **text files**
- Text files can be read by text editors

Bytes-Oriented vs Unicode-Oriented

JACOBS UNIVERSITY

Byte-oriented streams versus Unicode-oriented characters

- Byte-oriented streams are inconvenient for processing Unicode data
- Classes inheriting from **Reader** and **Writer** abstract classes are used for processing Unicode data

Byte-Oriented Streams Processing	Unicode-Character Oriented Processing
InputStream	**Reader**
OutputStream	**Writer**

The Java I/O system is based on these **four** classes

- A zoo of classes inherit from the four **abstract classes**

InputStream and OutputStream Classes

JACOBS UNIVERSITY

The **InputStream** class has an abstract method

```
abstract int read() throws IOException
```

Returns one byte, or -1 if **end of input source** is encountered

- The **OutputStream** class also has an abstract method

```
abstract void write(int b) throws IOException
```

Writes one byte to an output location

Reader and Writer Classes

JACOBS UNIVERSITY

Basic methods are similar to ones for the **InputStream** & **OutputStream**

```
abstract int read() throws IOException
```

Returns a Unicode code unit, or -1 if **end of input source** is encountered

- The **Writer** class has an abstract method

```
abstract void write(int b) throws IOException
```

Writes one Unicode code unit to an output location

Basic File Processing

JACOBS UNIVERSITY

The typical pattern for processing a file is:

1. Open a file
2. Check if the file is opened
3. If the file is opened, read/write from/to the file
4. Close the file

- Input & output streams have **close()** method (output also uses **flush()**)

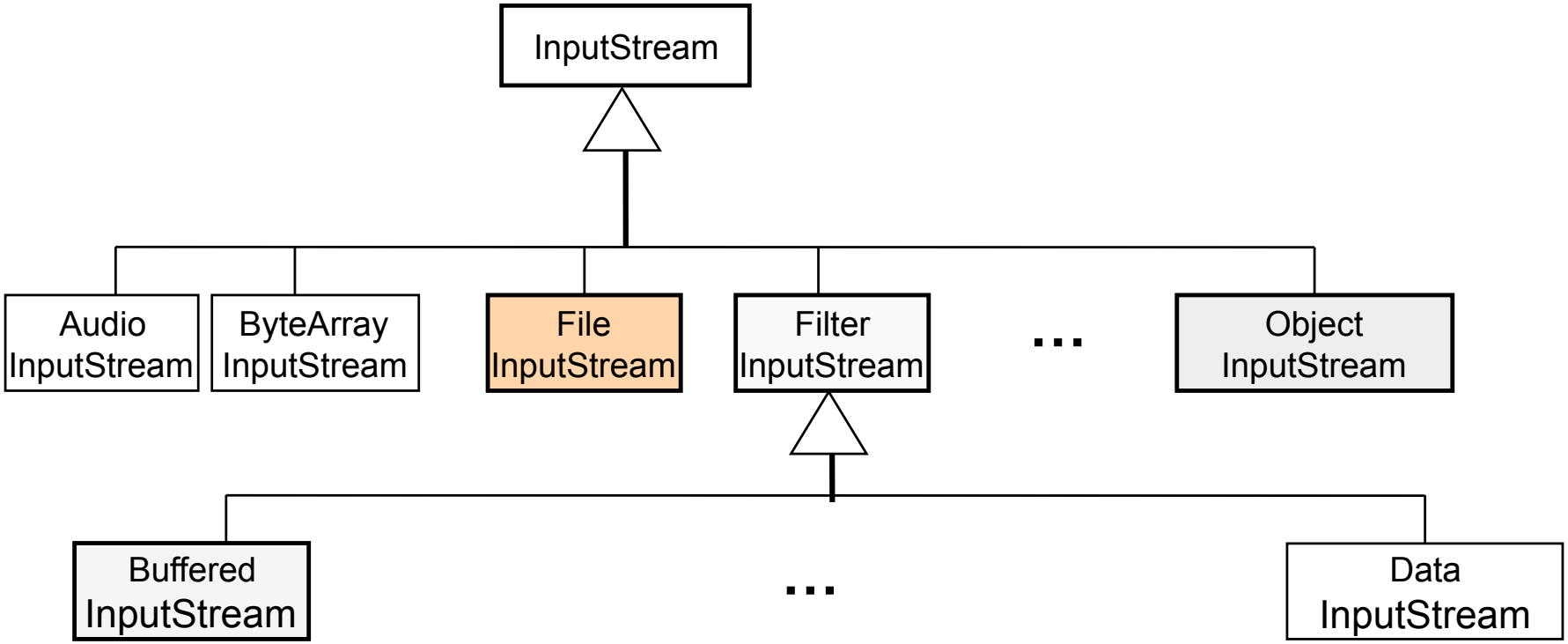
Closing a File

- Closing a file releases system resources
- Closing a file also **flushes the buffer to the output stream**

I/O Class Hierarchy

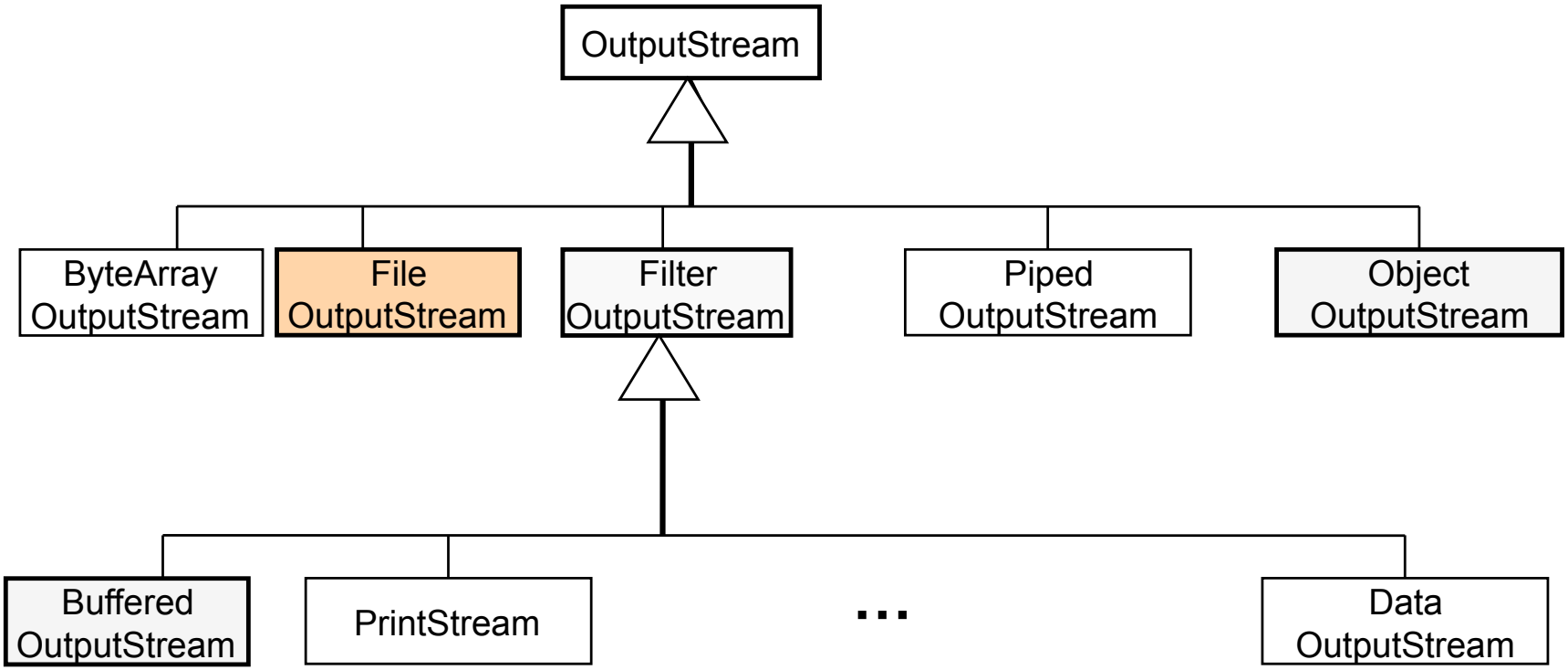
Java I/O system is based on four abstract classes

- **InputStream, OutputStream, Reader, Writer**



FileInputStream represents an input stream that is attached to a disk file

I/O Class Hierarchy



FileOutputStream represents an output stream that is attached to a disk file

Example 1

JACOBS UNIVERSITY

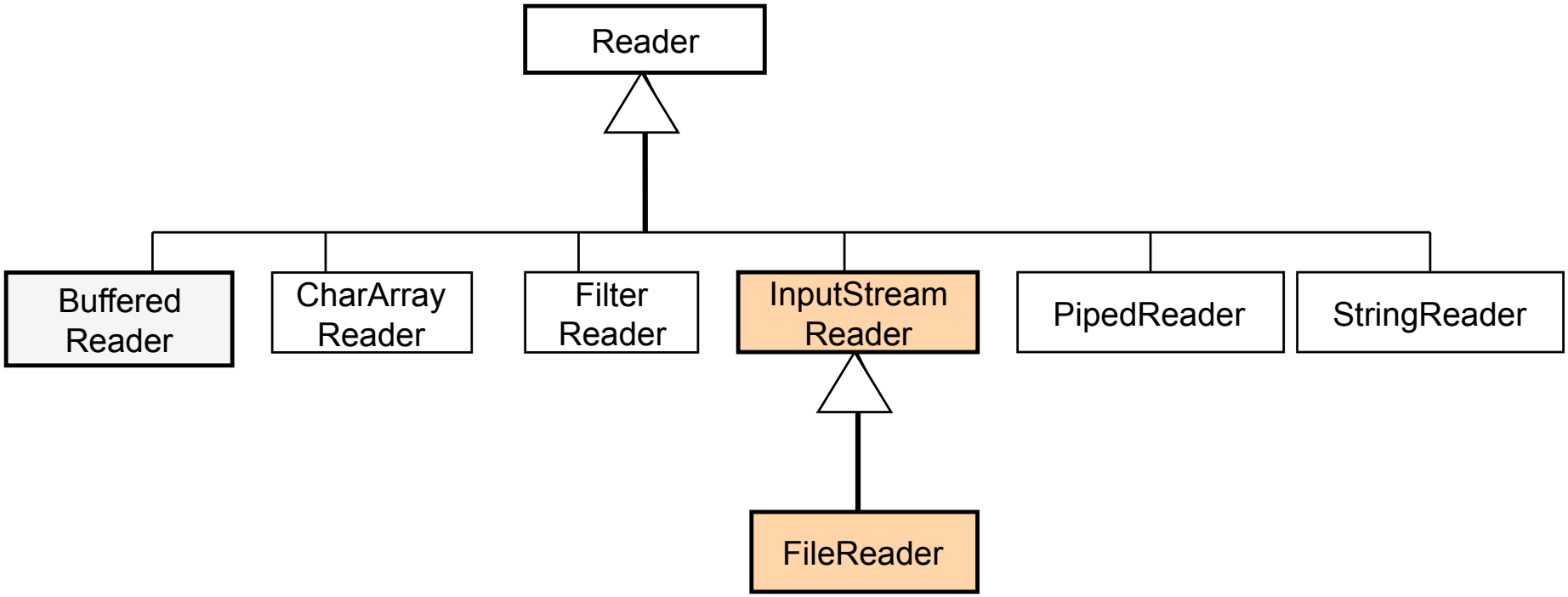
```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
/** souce: The Java tutorial textbook, 5th edition */
public class CopyBytes {
    public static void main(String [] argv) throws IOException {
            FileInputStream in      = null;
            FileOutputStream out = null;

            try {
                    in   = new FileInputStream("xanadu.txt");
                    out = new FileOutputStream("Outagain.txt");
                    int c;
                    while ( (c = in.read()) != -1)
                            out.write(c);
            } finally {
                    if (in != null) in.close();
                    if (out != null) out.close();
            } // end of finally
   } // end of main

} // end of CopyBytes
```

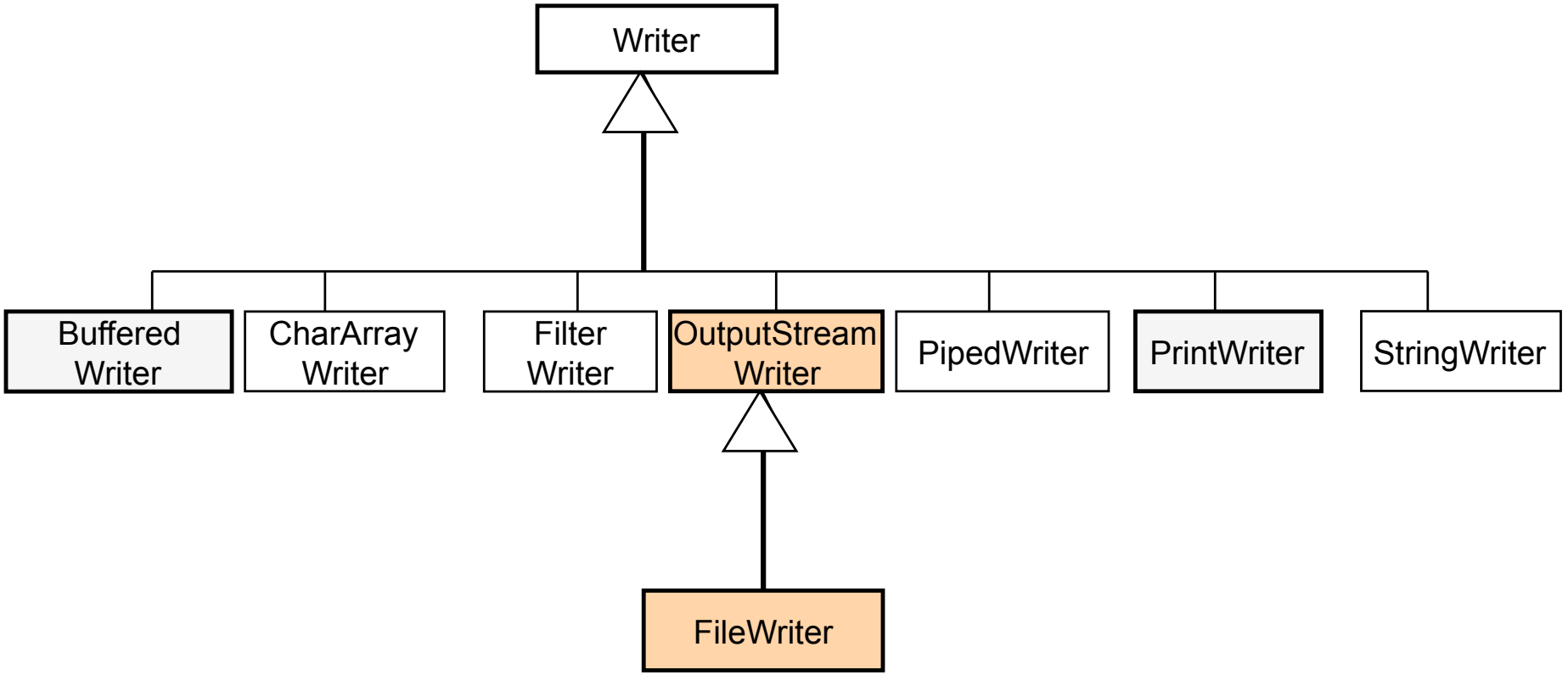
I/O Class Hierarchy

Reader Hierarchy



I/O Class Hierarchy

Writer Hierarchy



Example 2

JACOBS UNIVERSITY

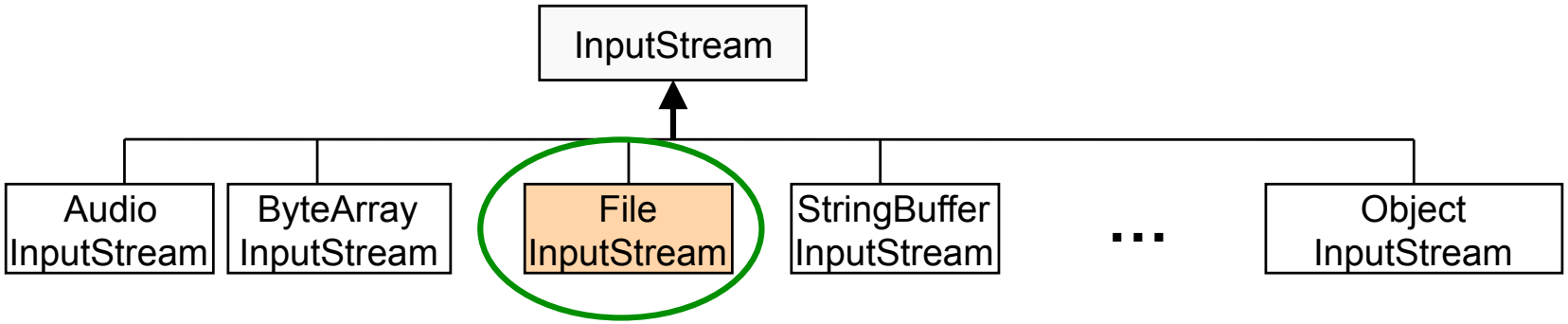
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
/** souce: The Java tutorial textbook, 5th edition */
public class CopyCharacter {
    public static void main(String [] argv) throws IOException {
            FileReader inputStream = null;
            FileWriter outputStream = null;

            try {
                    inputStream   = new FileReader("xanadu.txt");
                    outputStream = new FileWriter("Characteroutput.txt");
                    int c;
                    while ( (c = inputStream.read()) != -1)
                            outputStream.write(c);
            } finally {
                    if (inputStream != null) inputStream.close();
                    if (outputStream != null) outputStream.close();
            } // end of finally
    } // end of main

} // end of CopyCharacter
```

File Stream Processing

FileInputStream and **FileOutputStream** give I/O streams attached to a disk file



Give the filename or full path name of the file in a constructor

- Use the constant string **`File.`**`separator` as a file separator

```
FileInputStream fin = new FileInputStream("employee.dat");
```

- OR

```
File f = new File("employee.dat");
FileInputStream fin = new FileInputStream(f);
```

Buffered Streams

JACOBS UNIVERSITY

Unbuffered I/O is inefficient because:

- Each read/write is handled directly by the operating system
- Each request triggers disk access, network activity etc which is expensive

- Java implements buffered I/O streams to read data from a buffer

With buffered I/O

- The native API is called only when the buffer is full (writing) or the buffer is flushed or the buffered stream is closed
- The native API is called only when the buffer is empty (reading)

Example 3

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
/** souce: The Java tutorial textbook, 5th edition */
public class CopyBytesBuffered {
    public static void main(String [] argv) throws IOException {
            FileInputStream in     = null;
            FileOutputStream out = null;

            try {
                    in   = new BufferedInputStream
                            new (FileInputStream (“xanadu.txt”) );
                    out = new BufferedOutputStream (
                            new FileOutputStream (“Characteroutput.txt”) );
                    int c;
                    while ( (c = in.read()) != -1)
                            out.write(c);
            } finally {
                    if (in != null) in.close();
                    if (out != null) out.close();
            } // end of finally
   } // end of main

} // end of CopyBytesBuffered
```

Example 4

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
/** souce: The Java tutorial textbook, 5th edition */
public class CopyCharacterBuffered {
    public static void main(String [] argv) throws IOException {
            FileReader inputStream = null;
            FileWriter outputStream = null;

            try {
                    inputStream   = new BufferedReader
                                        new (FileReader("xanadu.txt") );
                    outputStream = new BufferedWriter(
                                        new FileWriter("Characteroutput.txt") );
                    int c;
                    while ( (c = inputStream.read()) != -1)
                            outputStream.write(c);
            } finally {
                    if (inputStream != null) inputStream.close();
                    if (outputStream != null) outputStream.close();
            } // end of finally
   } // end of main

} // end of CopyCharacterBuffered
```

Filtered Streams

JACOBS UNIVERSITY

Java's IO package is built on the principal that

- Each class should have a very focused responsibility (**cohesion**)

- **FileInputStream** interacts with files: its job is to get bytes, not to analyze them

Filtered Streams

JACOBS UNIVERSITY

- To read numbers, strings, objects etc., combine **FileInputStream** with other classes whose responsibility is to group bytes or characters together

- The combination is done by ***feeding an existing stream into the constructor of another to get the additional functionality***

- The combined streams are called **filtered streams**

Filtered Streams

We have seen examples in previous slides, here are more:

- To be able to *read numbers* from a file, first create a **FileInputStream**
- Pass the **FileInputStream** reference to the constructor of a **DataInputStream**

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

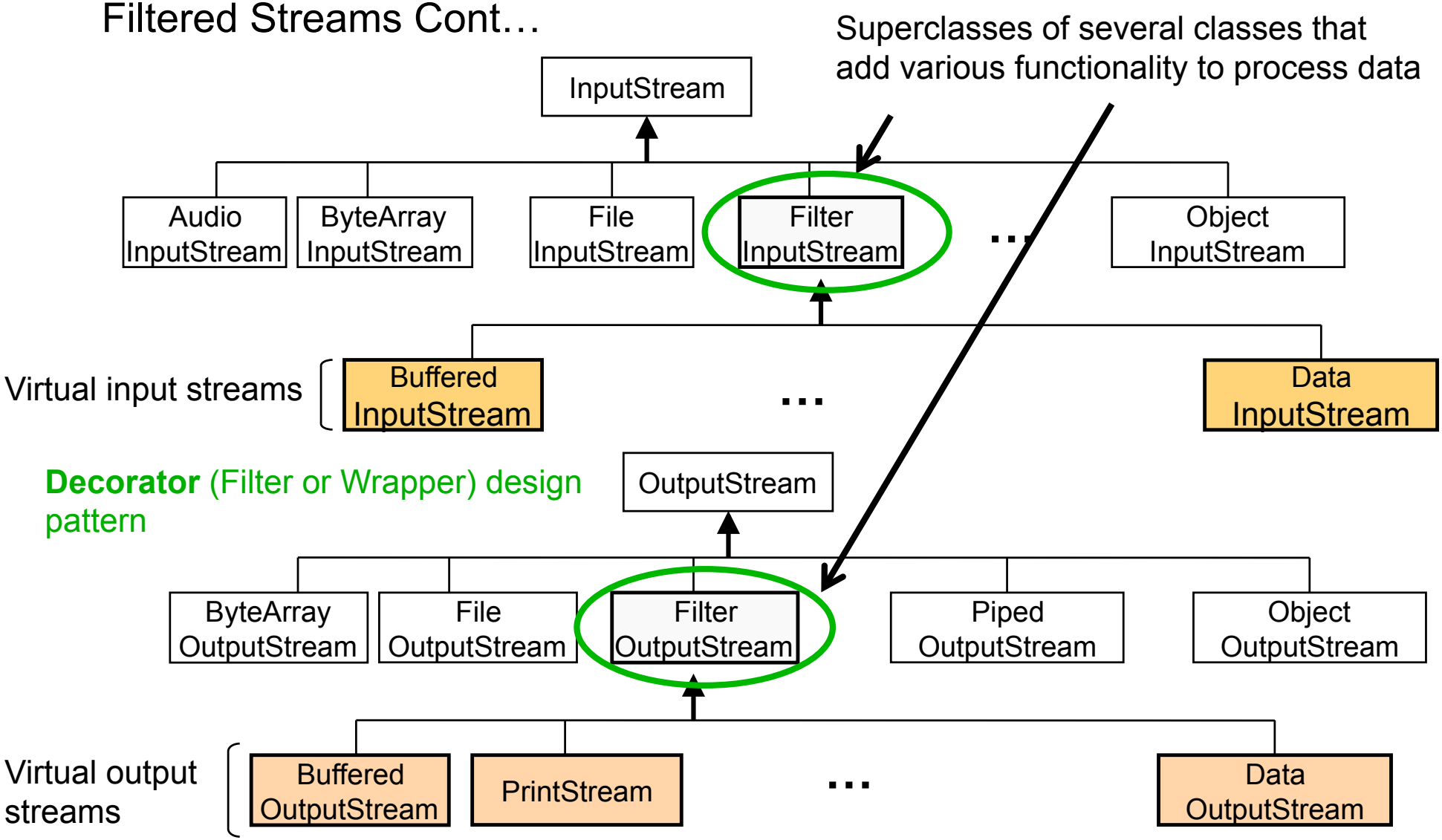
New stream with more capable interface

DataInputStream/ DataOutputStream

- Has interface that allows to read/write all the basic Java types

Filtered Streams

Filtered Streams Cont…



Filtered Streams

JACOBS UNIVERSITY

Filtered Streams Example

- To support buffering and data input methods when reading files

Data input methods
Buffer stream
Obtain stream

```
DataInputStream din = new DataInputStream(
                          new BufferedInputStream(
                           new FileInputStream("employee.dat")));
```

Reading numbers from compressed zip file

```
ZipInputStream zin = new ZipInputStream(
                          new FileInputStream("employee.zip"));

DataInputStream din = new DataInputStream(zin);
```

Data Streams

DataInput/ **DataOutput** interface

- Data streams support binary I/O of primitive data values in Java
- Data streams implement the **DataInput** and **DataOutput** interfaces
- **DataInputStream** and **DataOutputStream** are the most widely used implementations of the **DataInput** and **DataOutput** interfaces respectively

DataInput interface	**DataOutput** interface
`readDouble`	`writeDouble`
`readShort`	`writeShort`
`readInt`	`writeInt`
`readLong`	`writeLong`
`readFloat`	`writeFloat`
`readBoolean`	`writeBoolean`
`readChar`	`writeChar`

Text Streams

JACOBS UNIVERSITY

We have so far looked at **binary I/O**

- It is fast and efficient, but is not easily readable by humans
- Humans can better comprehend text I/O

Unicode

- Java uses Unicode *code units* to represent texts
- Local systems use their own encoding
- Java provides stream **filters** that bridge the gap between local system and Unicode

- The Unicode processing classes all inherit from abstract classes
  - ❑ **Reader** and **Writer** classes

Text Streams

JACOBS UNIVERSITY

Example

- An input reader that reads keystrokes from the console & converts to Unicode

```
InputStreamReader in = new InputStreamReader(System.in);
```

FileReader and **FileWriter**

- Convenience classes for processing text strings from a file

```
FileWriter out = new FileWriter("output.txt")
```

- Is equivalent to

```
FileWriter out = new FileWriter(new FileOutputStream("output.txt"));
```

Text Streams

JACOBS UNIVERSITY

The **PrintWriter** class

- The class is used for *text output*

- A print writer can print strings and numbers in text format

- A print write must be combined with a **destination writer**

Text Streams

JACOBS UNIVERSITY

Writing to a **PrintWriter**

- Use *print* and *println* used with **System**.out

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt"));
```

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

Harry Hacker 75000

Text Streams

JACOBS UNIVERSITY

Reading Text Input

- Use the **BufferedReader** class and its **`readLine`** method to read data
- Use the **Scanner** class to read text data

BufferedReader Example

```
BufferedReader in = new BufferedReader(new FileReader("employee.txt"));
String line;

while ((line = in.readLine()) != null)  {
        do something with line
}
```

Use of Streams

JACOBS UNIVERSITY

Writing Delimited Output

- Delimited format imply each record is stored in a separate line
- Instance fields are separated by delimiters

```
#Here is a sample set of records (firstname lastname|salary|year|month|day):

Harry Hacker|35500|1989|10|1
Carl Cracker|75000|1987|12|15
Tony Tester|38000|1990|3|15
```

Use of Streams

JACOBS UNIVERSITY

Write records using **PrintWriter** class

```
public void writeData(PrintWriter out) throws IOException {

      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(hireDay);
      out.println(name + "|"
                  + salary + "|"
                  + calendar.get(Calendar.YEAR) + "|"
                  + (calendar.get(Calendar.MONTH) + 1) + "|"
                  + calendar.get(Calendar.DAY_OF_MONTH));
}
```

Example

JACOBS UNIVERSITY

Reading Delimited Input

- Read-in a line of text using the **`readLine`** method of **BufferedReader**

```
public void readData(BufferedReader in) throws IOException {

      String s = in.readLine();
      StringTokenizer t = new StringTokenizer(s, "|");

      String name = t.nextToken();
      double = salary = Double.parseDouble(t.nextToken());
      int y = Integer.parseInt(t.nextToken());
      int m = Integer.parseInt(t.nextToken());
      int d = Integer.parseInt(t.nextToken());

      GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
      hireDay = calendar.getTime();
}
```

Object Serialization

JACOBS UNIVERSITY

Object serialization:

- An object is represented as a sequence of bytes
- The serialized representation includes the object's data as well as object type information and the types of data stored in the object

A **serialized object** can be written to a file or send over a network

Object deserialization:

- After a **serialized object** has been written to a file, it can be read from the file and be **deserialized**
- The type information and bytes that represent the object and its data can be used to recreate the object in memory

Object Serialization

JACOBS UNIVERSITY

ObjectInputStream

- Enables an entire objects to be read from a stream (e.g., file)

- Implements the **ObjectInput** interface which contains a method *readObject*

- The method *readObject* reads and returns an **Object** from an **InputStream**

- Use the **FileInputStream** class to read from files

Object Serialization

JACOBS UNIVERSITY

ObjectOutputStream

- Enables entire objects to be written to a stream (e.g., file)

- Implements the **ObjectOutput** interface which contains a method *writeObject*

- The method *writeObject* takes an **Object** as parameter and writes its information to an **OutputStream**

- Use **FileOutputStream** to write serialized objects to files

Object Serialization

JACOBS UNIVERSITY

Serialization Process:

1. A class must implement the **Serializable** interface to be serialized

2. Open an **ObjectOutputStream**

3. Call the ***writeObject*** method of **ObjectOutputStream** to save the object

Object Serialization

JACOBS UNIVERSITY

Example: Saving an object

```
// The class to be saved must implement the Serializable interface
// The Serializable interface has no methods to be implemented

class Employee implements Serializable { … }
```

```
// first open an ObjectOutputStream object
ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream(“employee.dat”));
//create the objects
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);

//save the objects by calling writeObject method
out.writeObject(harry);
out.writeObject(boss);
```

Object Serialization

To read the objects back

1. First get an **ObjectInputStream** object

2. Retrieve the objects in the order in which they were written

```
ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream("employee.dat"));

Employee e1 = (Employee) in.readObject(); //
Employee e2 = (Employee) in.readObject();
```

Note that saving a network of objects is a challenge see textbook, (Horstmann & Cornell, 2013, Core Java, Vol II, 9th edition, Chapter 1)

File Management

JACOBS UNIVERSITY

The **File** class

- A **File** can represent either a *file* or a *directory*
- Example **File** constructors

```
// associates name of file or directory to File object
// name can contain path info – absolute or relative
public File (String name)
// example
File file = new File("test.txt");
```

```
public File (String pathToName, String name) // locates directory
```

```
// uses existing File object directory to locate file or directory
public File (File directory, String name)
```

```
public File (URI uri) // uses URI object to locate a file
```

File Management

JACOBS UNIVERSITY

Common **File** methods (see API for complete listing)

Method	Description
boolean *exists()*	True if name specified as argument to File constructor is a file or directory; false otherwise
boolean *isFile()*	True if name specified as argument to File constructor is a file;. false otherwise
boolean *isDirectory()*	True if name specified as argument to File constructor is a directory;. false otherwise
String *getAbsolutePath()*	Returns absolute path of file or directory
String *getName()*	Returns name of file or directory
String *getPath()*	Returns path of file or directory
String *getParent()*	Returns parent directory of file or directory
long *length()*	Returns length of file in bytes; 0 returned if object represents dir
String[] *list()*	Returns array of strings representing the contents of a directory

Reading Assignment

- Core Java, Volume II, Chapter 1. Streams and Files by Horstmann and Cornell, 2013.