Day 4

I/O Streams

Introduction

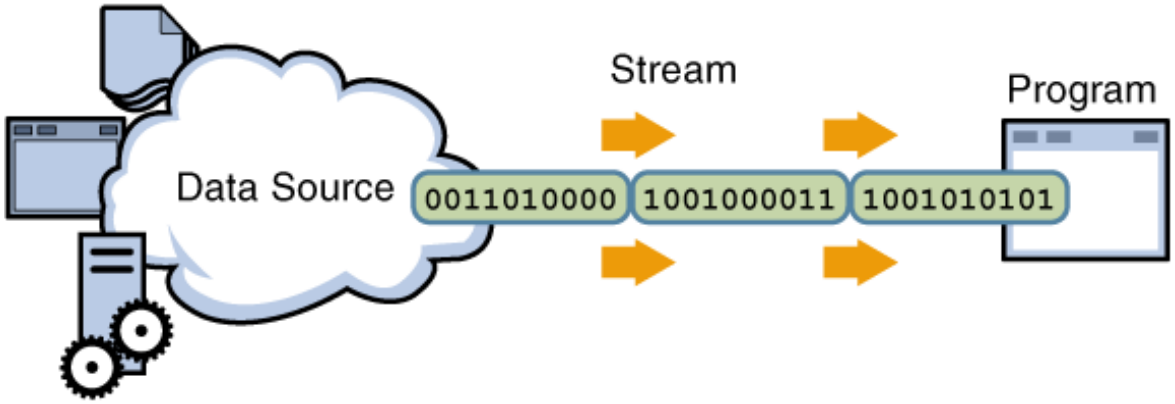
- Often a program needs to
  - bring in information from an external source
  - send out information to an external destination
- The information can be anywhere:
  - in a file, on disk, somewhere on the network, in memory, or in another program
- The information can be of any type:
  - objects, characters, images, or sounds
- Our programs can use Java I/O Stream classes to read and to write data

Introduction

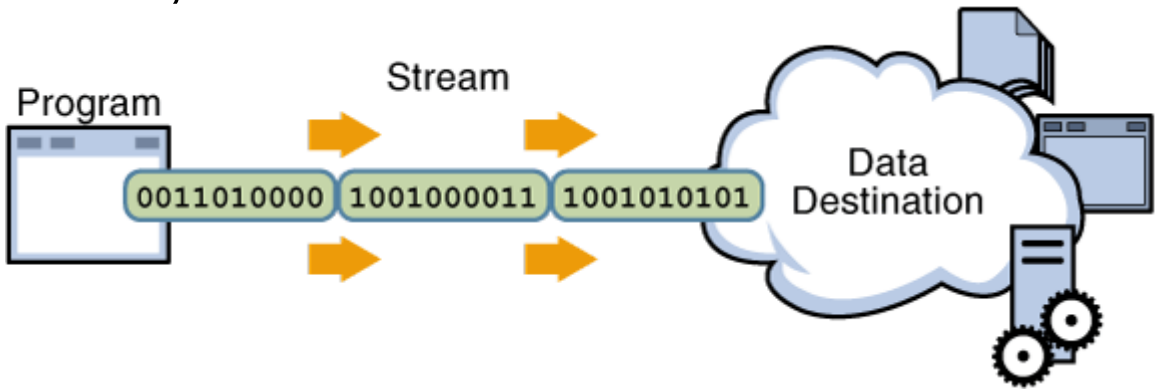
- Stream
  - A stream is a sequence of data.
  - Abstraction of a file or a device that allows a series of items to be read or written
- The **`java.io`** package contains I/O Stream classes that your programs can use to read and write data
- Most of the classes implement sequential access streams

Introduction

- A program uses an *input stream* to read data from a source, one item at a time



- A program uses an *output stream* to write data to a destination, one item at time



Stream Categories

- Streams can be divided into two groups
  - Byte Streams
    - read and write bytes (8-bit data)
  - Character Streams
    - that read and write Unicode characters (16-bit data)

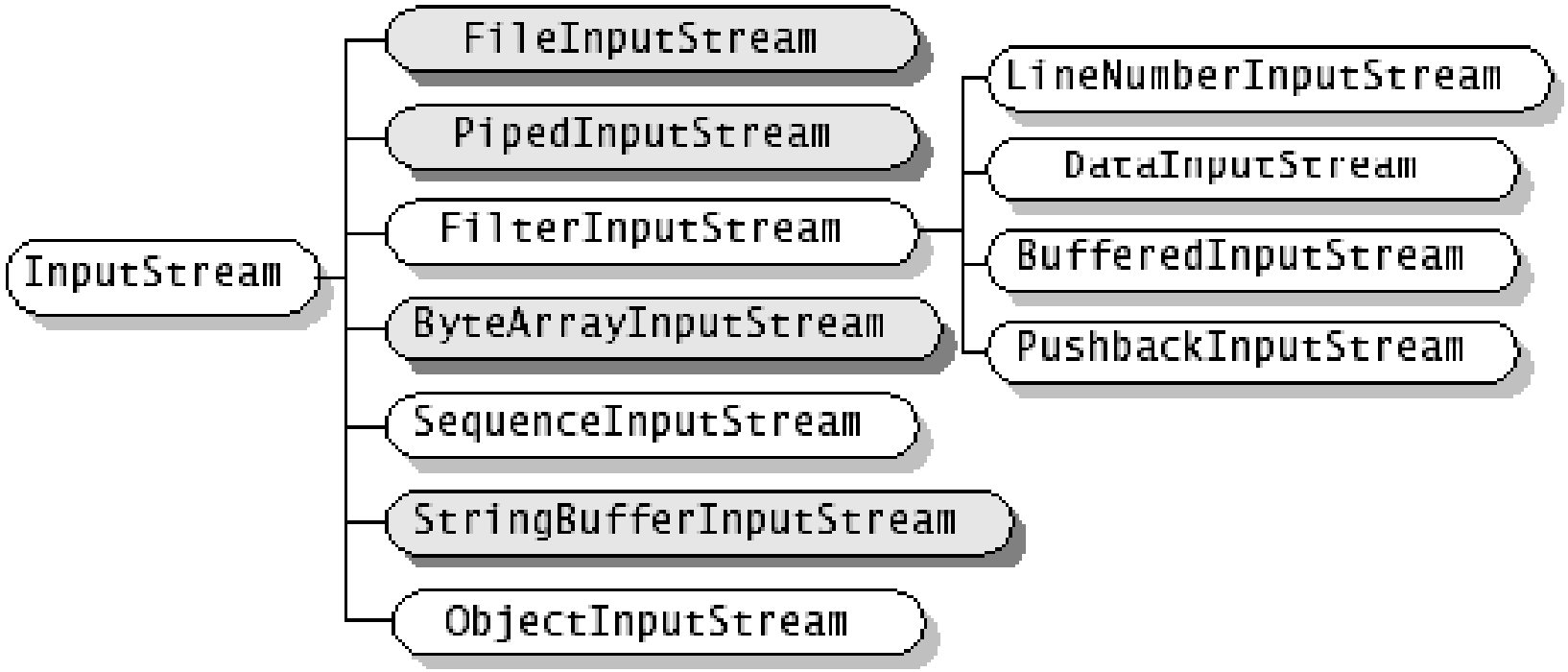
- Each sequential access stream has a speciality
  - such as reading from or writing to a file, filtering data as its read or written, or serializing an object.

Byte Streams

- Used For binary data
  - These streams are typically used to read and write binary data such as images and sounds
- Root classes for byte streams
  - *InputStream* Class
  - *OutputStream* Class
- Both of these classes are abstract
  - provide the API and partial implementation for
    - *input streams* (streams that read 8-bit bytes)
    - *output* streams (streams that write 8-bit bytes)

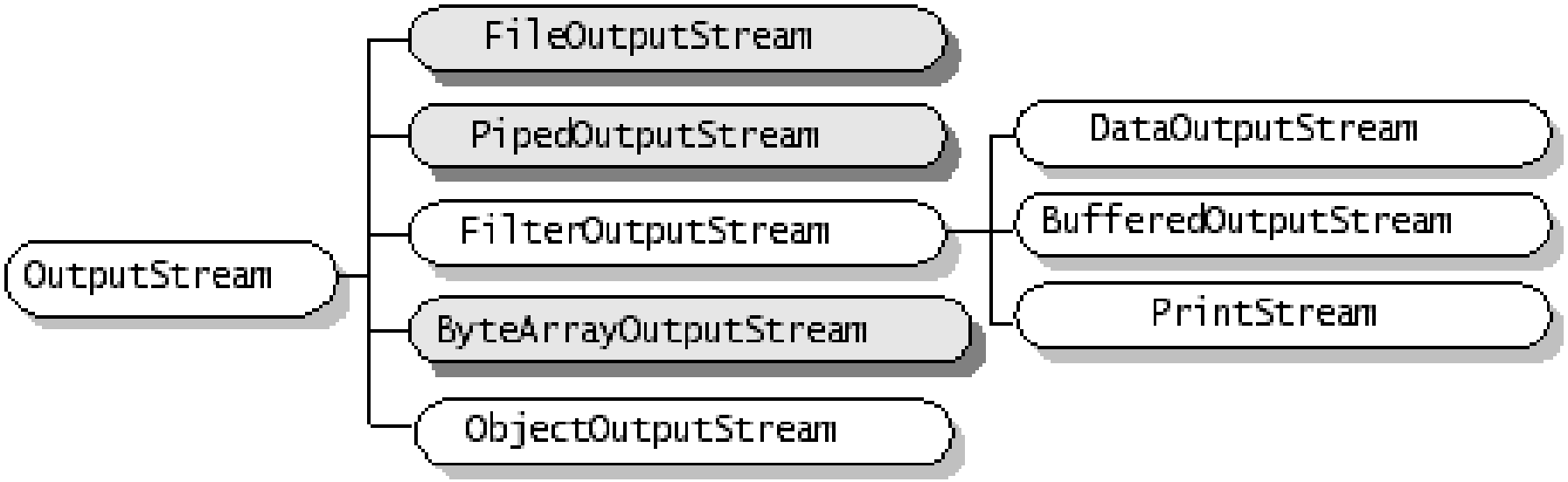
Byte Streams

- Class hierarchy of Input Stream classes



Byte Streams

- Class hierarchy of Output Stream classes

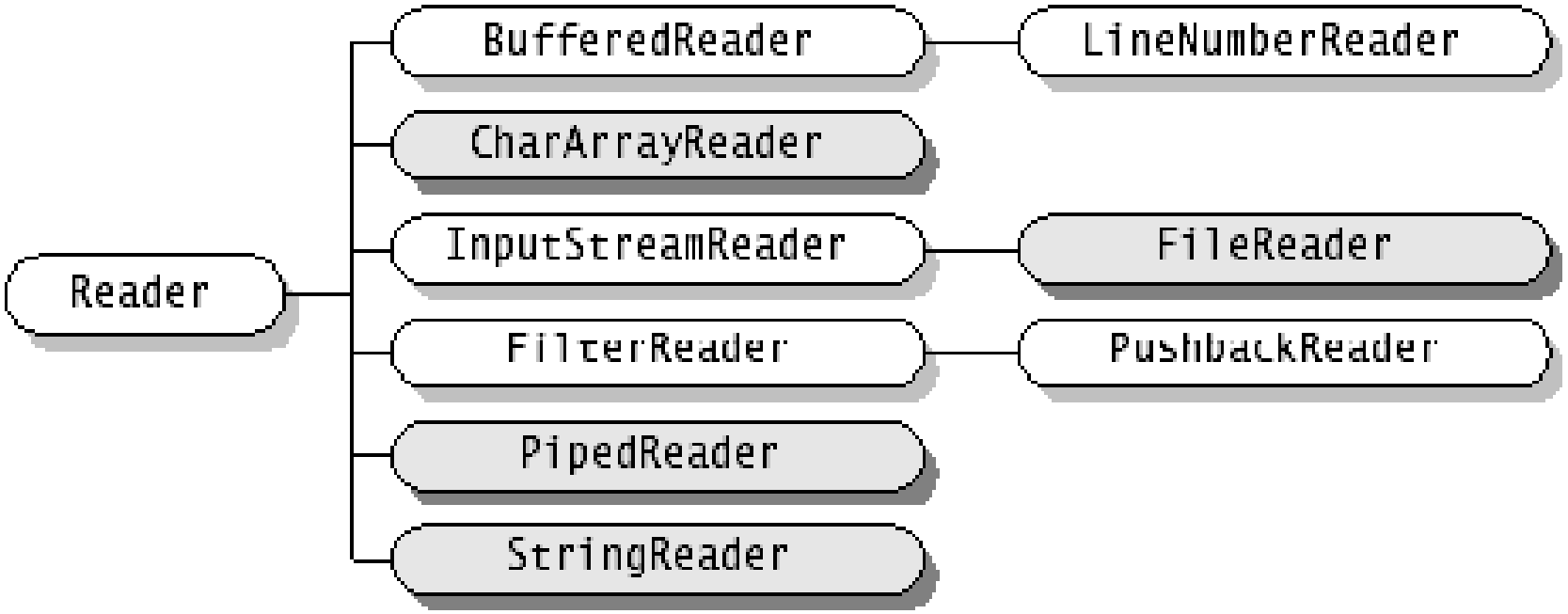


Character Streams

- Used For Unicode Character data
  - File or device abstractions for Unicode characters
  - These streams are typically used to read and write Unicode Character data
- Root classes for Character streams
  - *Reader* Class
  - *Writer* Class
- Both of these classes are abstract
  - provide the API and partial implementation for
    - *Reader* (streams that read 16-bit data)
    - *Writer* (streams that write 16-bit data)

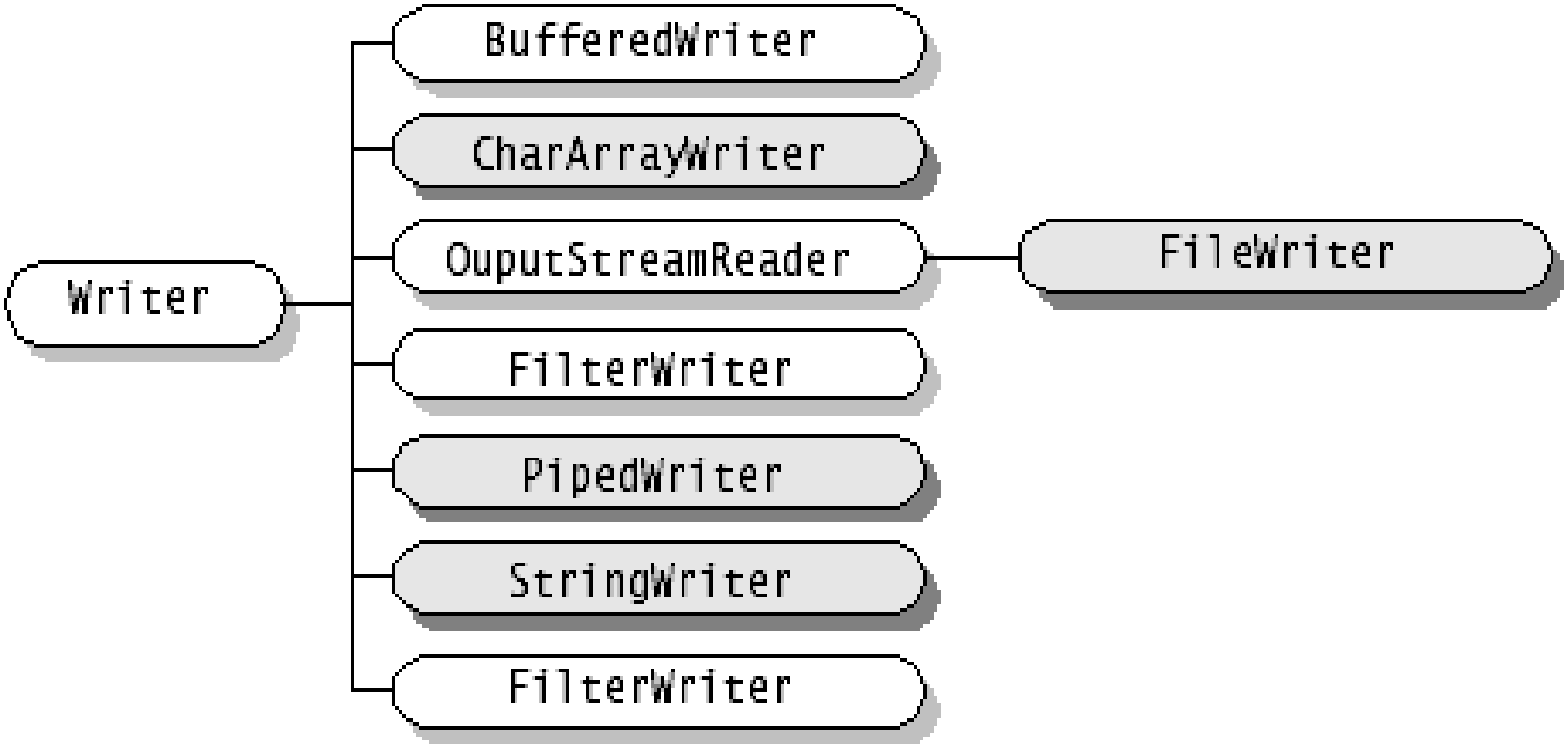
Character Streams

- Class hierarchy of *Reader* classes



Character Streams

- Class hierarchy of *`Writer`* classes



Character Streams

- The Java platform stores character values using Unicode conventions
- Character stream I/O automatically translates this internal format to and from the local character set.
- In Western locales, the local character set is usually an 8-bit superset of ASCII.
- For most applications, I/O with character streams is no more complicated than I/O with byte streams.
- Input and output done with stream classes automatically translates to and from the local character set.
- A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer

The *File* Class

- Not a stream class
- Important since stream classes manipulate *File* objects
- Abstract representation of actual files and directory pathnames

Constructors of File Class

Constructor Summary
File(File parent, String child) Creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.

Useful Methods of File Class

Method Summary	
boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
boolean	**canWrite**() Tests whether the application can modify to the file denoted by this abstract pathname.
boolean	**exists**() Tests whether the file denoted by this abstract pathname exists.
String	**getAbsolutePath**() Returns the absolute pathname string of this abstract pathname.
String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.
String	**getPath**() Converts this abstract pathname into a pathname string.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.

Useful Methods of File Class

boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
boolean	**mkdir**() Creates the directory named by this abstract pathname.
boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	**renameTo**(File dest) Renames the file denoted by this abstract pathname.
boolean	**setLastModified**(long time) Sets the last-modified time of the file or directory named by this abstract pathname.
boolean	**setReadOnly**() Marks the file or directory named by this abstract pathname so that only read operations are allowed.

File class: Example(1)

- Checking the Attributes of a File

```
import java.io.File;

class FileDemo {
 public static void main(String args[]) {
    File f1 = new File("f:\\");
    System.out.println("File Name: " + f1.getName());
    System.out.println("Path: " + f1.getPath());
    System.out.println("Abs Path: " + f1.getAbsolutePath());
    System.out.println("Parent: " + f1.getParent());
    System.out.println("File exists : "+f1.exists());
    System.out.println("The File is writable :"+f1.canWrite());
    System.out.println("The file is readable : "+f1.canRead());
    System.out.println("The File is a directory : "+f1.isDirectory());
    System.out.println("File last modified: " + f1.lastModified());
    System.out.println("File size: " + f1.length() + " Bytes");
  }
}
```

File class: Example(2)

- Listing the contents of a Directory

```
import java.io.*;
class DirListing
{
        public static void main(String[] args)
        {
                File dir = new File(System.getProperty("user.dir"));
                if (dir.isDirectory())
                {
                        System.out.println("Directory of "+dir);
                        String[] list = dir.list();
                        for (int i=0;i<list.length;i++ )
                        {
                                System.out.println("\t"+list[i]);
                        }
                }
                System.out.println("--End of list--");
        }
}
```

Using Streams

The I/O Superclasses

InputStream class

Method Summary	
int	**available**() Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.
void	**close**() Closes this input stream and releases any system resources associated with the stream.
void	**mark**(int readlimit) Marks the current position in this input stream.
boolean	**markSupported**() Tests if this input stream supports the mark and reset methods.
abstract int	**read**() Reads the next byte of data from the input stream.
int	**read**(byte[] b) Reads some number of bytes from the input stream and stores them into the buffer array b.
int	**read**(byte[] b, int off, int len) Reads up to len bytes of data from the input stream into an array of bytes.
void	**reset**() Repositions this stream to the position at the time the mark method was last called on this input stream.
long	**skip**(long n) Skips over and discards n bytes of data from this input stream.

The I/O Superclasses

- All of the streams--readers, writers, input streams, and output streams--are automatically opened when created.
- You can close any stream explicitly by calling its close method.
- Or the garbage collector can implicitly close it, which occurs when the object is no longer referenced

Reading and Writing Process

Reading	**Writing**
Open a Stream While more information read information close the Stream	Open a Stream While more information write information close the Stream

- InputStream Object is a source of information to your program
- OutputStream is a Sink for your program

Stream Example: reading from a file

```
import java.io.*;

class DisplayTxtDemo{
public static void main( String[] args) throws Exception{
FileInputStream fis = new FileInputStream("abc.txt");
int i;

while((i=fis.read())!=-1){
    char c = (char)i;
System.out.print(c);
}
fis.close();
}
};
```

Reader Example: reading from a file

```
import java.io.*;

class DisplayTxtDemo1{
public static void main( String[] args) throws Exception{
FileIReader fis = new FileReader("abc.txt");
int i;

while((i=fis.read())!=-1){
    char c = (char)i;
System.out.print(c);
}
fis.close();
}
};
```

Copying a file using Byte Stream classes

```
import java.io.*
public class CopyBytes {
public static void main(String[] args) 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();
                }
        }
 }
}
```

Copy a file :Character Stream classes

```
import java.io.*;

public class CopyCharacters {
    public static void main(String[] args) 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();
            }
        }
    }
}
```

Bridging I/O Stream Classes

- *InputStreamReader* and *OutputStreamWriter*
  - *A reader and writer pair that forms the bridge between byte streams and character streams*
- An InputStreamReader reads bytes from an InputStream and converts them to characters,
  - uses the default character encoding or a character encoding specified by name
- An OutputStreamWriter converts characters to bytes,
  - uses the default character encoding or a character encoding specified by name and then writes those bytes to an OutputStream

Buffered Streams

- The examples we've seen so far use *unbuffered* I/O
- In Unbuffered I/O each read or write request is handled directly by the underlying OS
- This can make a program much less efficient
  - since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

Buffered Streams

- To reduce the overhead due to unbuffered I/O, the Java platform implements *buffered* I/O streams.

- Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty.

- Buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Buffering a Stream

- A program can convert a unbuffered stream into a buffered stream by wrapping the unbuffered Stream Object
- To wrap, the unbuffered stream object is passed to the constructor for a buffered stream class

```
inputStream
        = new BufferedReader(new FileReader("xanadu.txt"));
outputStream
        = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Flushing a Buffer

- Writing out a buffer at critical points, without waiting for it to fill is known as *flushing* the buffer.
- Some buffered output classes support *autoflush*
  - specified by an optional constructor argument
- When autoflush is enabled, certain key events cause the buffer to be flushed
  - an autoflush PrintWriter object flushes the buffer on every invocation of println or format
- To flush a stream manually, invoke its flush method
  - The flush method is valid on any output stream, but has no effect unless the stream is buffered.

Data Streams

(DataInputStream and DataOutputStream)

Data Streams

- Data streams support binary I/O of primitive data type values
  - boolean, char, byte, short, int, long, float, and double as well as String values
- All data streams implement either the DataInput interface or the DataOutput interface

- The most widely-used implementations of these interfaces are DataInputStream and DataOutputStream.

DataOutputStream

- An application uses a data output stream to write data that can later be read by a data input stream.

```
FileOutputStream fos=new FileOutputStream("test.txt");
DataOutputStream dos=new DataOutputStream(fos);
```

DataInputStream

- A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way

```
FileInputStream fis =new FileInputStream("test.txt");
DataInputStream dis =new DataInputStream(fis);
```

DataOutputStream : example

```
import java.io.*;
class DataOutputStreamDemo
{
      public static void main(String[] args) throws Exception
      {
                double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
                int[] units = { 12, 8, 13, 29, 50 };
                String[] descs = { "Java T-shirt",  "Java Mug",  "Duke Juggling Dolls",
                                          "Java Pin", "Java Key Chain" };
               DataOutputStream out = new DataOutputStream(new
            BufferedOutputStream(new FileOutputStream("data.dat")));

               for (int i = 0; i < prices.length; i ++) {
               out.writeDouble(prices[i]);
               out.writeInt(units[i]);
               out.writeUTF(descs[i]);
               }
               out.flush();
               out.close();
      }
}
```

DataInputStream : example

```
import java.io.*;
class DataInputStreamDemo
{
	public static void main(String[] args) throws Exception
	{
		DataInputStream in = new DataInputStream(new
	    BufferedInputStream(new FileInputStream("data.dat")));

		double price;
		int unit;
		String desc;
		double total = 0.0;

		try {
			while (true) {
				price = in.readDouble();
				unit = in.readInt();
				desc = in.readUTF();
				System.out.format("You ordered %d units of %s at $%.2f%n",
					unit, desc, price);
				 total += unit * price;
			}
			} catch (EOFException e) {  }
		System.out.println("Hello World!");
	}
}
```

Command Line Input

- The Java platform supports Command line input through the Standard Streams
  - By default, they read input from the keyboard and write output to the display
- The Java platform supports three Standard Streams:
  - ***Standard Input***, accessed through System.in
  - ***Standard Output***, accessed through System.out
  - ***Standard Error***, accessed through System.err
- These objects are defined automatically and do not need to be opened

Command Line Input

- Standard Streams are not character streams, but, for historical reasons, they are byte streams
- System.out and System.err are defined as PrintStream objects
  - PrintStream utilizes an internal character stream object to emulate many of the features of character streams.
- System.in is a byte stream with no character stream features
- To use Standard Input as a character stream, we need to wrap System.in in InputStreamReader.

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

Keyboard Input:Example(1)

```
import java.io.*;

class BRRead {
  public static void main(String args[])
    throws IOException
  {
    char c;
     BufferedReader br = new BufferedReader(new
     InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");

    // read characters
    do {
              int i =  br.read();
      c = (char)i;
      System.out.print(c);
    } while(c != 'q');
  }
}
```

Keyboard Input:Example(2)

```
import java.io.*;

class BRReadLines {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str;

    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    while(true) {
      str = br.readLine();
     if ( str.equals("stop")) System.exit(0);
      System.out.println(str);
    }
  }
}
```

Keyboard Input:Example(2)

```
import java.io.*;

class BRReadLines {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str;

    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    while(!((str=br.readLine()).equals(“stop”))) {
      str = br.readLine();
          System.out.println(str);
    }
  }
}
```

Serialization

(ObjectInputStream and ObjectOutputStream)

Serialization

- Serialization is the process of writing the state of an object to a byte stream
  - This is useful when you want to save the state of your program to a persistent storage area, such as a file
- These Objects may be restored by using the process of deserialization
- Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities
  - The **Serializable** interface defines no members

Use of Object Serialization

- Remote Method Invocation (RMI)
  - communication between objects via sockets
- Lightweight persistence
  - the archival of an object for use in a later invocation of the same program

Streams for Serialization

- Streams for serialization
  - ObjectInputStream
    - for Deserialization
  - ObjectOutputStream
    - for serialization
- To allow an object to be serializable:
  - Its class should implement the *Serializable* interface
  - Its class should also provide a default constructor or a constructor with no arguments
  - Serializability is inherited
    - Don’t have to implement *Serializable* on every class
    - Can just implement *Serializable* once along the class heirarchy

A Serializable Class

```
import java.io.*;

public class Email implements Serializable
{
    private String to;
    private String from;
    private String message;

    public Email(String t, String f, String m)
    {
        to=t;
        from=f;
        message=m;
    }

public void getMail()
    {
    System.out.println(to+"  "+from+"  "+message);
    }
};
```

Serialization : example

```
import java.io.*;
class  SerializerDemo
{
    public static void main(String[] args) throws Exception
    {
        Email e = new
    Email("abc@yahoo.com","xyz@gmail.com","Hello, this IO Stream");
        System.out.println("Before Serialization info is");
        e.getMail();

        FileOutputStream fos = new FileOutputStream("Email.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(e);
        oos.flush();
        oos.close();
        System.out.println(" The Object is Serialized...");
    }
}
```

Deserialization : Example

```
import java.io.*;
class DeSerializerDemo
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = new FileInputStream("Email.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Object o = ois.readObject();
        System.out.println(o.getClass().getName());
        Email mail = (Email)o;
        mail.getMail();
        System.out.println("Hello World!");
    }
}
```

Thank You