10.3: Java input and output basics in java.io package

IT1406 – Introduction to Programming

Level I - Semester 1

10.3.1. Input and output classes, interfaces, files and directories

Input/Output: Exploring java.io

- Most programs cannot accomplish their goals without accessing external data.
- Data is retrieved from an *input* source and the results of a program are sent to an *output* destination.
- In Java, these sources or destinations are defined very broadly.
- For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these devices are all handled by the same abstraction: the *stream*.
- An I/O stream is linked to a physical device by the Java I/O system.
- All I/O streams behave in the same manner, even if the actual physical devices they are linked to differ.

10.3.1. Input and output classes, interfaces, files and directories

The I/O Classes and Interfaces

Some of the I/O classes defined by java.io are listed here

BufferedInputStream	FileWriter	PipedOutputStream	CharArrayWriter
BufferedOutputStream	FilterInputStream	PipedReade	LineNumberReader
BufferedReader	FilterOutputStream	PipedWriter	RandomAccessFile
BufferedWriter	FilterReader	PrintStream	Console
ByteArrayInputStream	FilterWriter	PrintWriter	ObjectInputStream
ByteArrayOutputStream	InputStream	PushbackInputStream	Reader
CharArrayReader	InputStreamReader	PushbackReader	File

10.3.1. Input and output classes, interfaces, files and directories

The I/O Classes and Interfaces

Following interfaces are defined by java.io

Closeable	ObjectInputValidation	FileFilter	DataInput
FilenameFilter	ObjectOutput	Flushable	DataOutput
ObjectStreamConstants	Externalizable	Serializable	ObjectInput

10.3.1. Input and output classes, interfaces, files and directories

File

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not.
- It deals directly with files and the file system. That is, the **File** class does not specify how
- information is retrieved from or stored in files; it describes the properties of a file itself.
- A **File** object is used to obtain or manipulate the information (permission, time, date, path) associated with a disk file
- The following constructors can be used to create **File** objects:
  - File(String *directoryPath*)
  - File(String *directoryPath*, String *filename*)
  - File(File *dirObj*, String
- Example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.
- `File f1 = new File("/");`
- `File f2 = new File("/","autoexec.bat");`
- `File f3 = new File(f1,"autoexec.bat");`

10.3.1. Input and output classes, interfaces, files and directories

File

- **File** defines many methods that obtain the standard properties of a **File** object.
- For example,
  - **getName( )** returns the name of the file;
  - **getParent( )** returns the name of the parent directory;
  - exists( ) returns true if the file exists, false if it does not.

Example demonstrates several of the **File** methods. It assumes that a directory called **java** exists off the root directory and that it contains a file called **COPYRIGHT**

```
import java.io.File;
class FileDemo {
  static void p(String s) {
   System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
```

10.3.1. Input and output classes, interfaces, files and directories

```
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
```

10.3.1. Input and output classes, interfaces, files and directories

File

- We'll discuss about some **File** methods.
- **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.
- The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.
- According to the example given below to the method **renameTo( )**,

  boolean renameTo(File *newName*)

- Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).
- Another method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

  boolean delete( )

- You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns**true** if it deletes the file and **false** if the file cannot be removed.

10.3.1. Input and output classes, interfaces, files and directories

File Methods

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
long getUsableSpace( )	Returns the number of usable free bytes of storage available on the partition associated with the invoking object
boolean isHidden( )	Returns **true** if the invoking file is hidden. Returns **false** otherwise.
boolean setLastModified(long *millisec*)	Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from sJanuary 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

10.3.1. Input and output classes, interfaces, files and directories

File

- Methods also exist to mark files as readable, writable, and executable.
- Because **File** implements the **Comparable** interface, the method **compareTo( )** is also supported.
- **toPath( )** returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath( )** converts a **File** into a **Path**.)
- **Path** is packaged in **java.nio.file** and is part of NIO.
- Thus, **toPath( )** forms a bridge between the older **File** class and the newer **Path** interface.

10.3.1. Input and output classes, interfaces, files and directories

Directories

- A directory is a **File** that contains a list of other files and directories.
- When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**.
- In this case, you can call **list( )** on that object to extract the list of other files and directories inside.
- It has twoforms. The first is shown here:

  String[ ] list( )

- The list of files is returned in an array of **String** objects.
- The program shown here illustrates how to use **list( )** to examine the contents of a
- directory:

```
import java.io.File;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
```

10.3.1. Input and output classes, interfaces, files and directories

```
    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();
      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
    System.out.println(dirname + " is not a directory");
   }
  }
}
```

10.3.1. Input and output classes, interfaces, files and directories

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

10.3.2. I/O exceptions and two ways to close a stream

File I/O exception

- There are 2 exceptions play an important role in I/O handling.

IOException.

- If an I/O error occurs, an **IOException** is thrown. In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**.

SecurityException

- In situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file.
- By default, applications run via **java** do not use a security manager.
- For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, applets will use the security manager provided by the browser, and file I/O performed by an applet could generate a **SecurityException.**

10.3.2. I/O exceptions and two ways to close a stream

Two ways to close a Stream

- In general, a stream must be closed when it is no longer needed.
- Failure to do so can lead to memory leaks and resource starvation.
- First way, is to explicitly call **close( )** on the stream.
- This is the traditional approach that has been used since the original release of Java. With this approach, **close( )** is typically called within a **finally** block.
- The traditional approach is shown here

```
try {
  // open and access file
} catch( I/O-exception) {
  // ...
} finally {
  // close the file
}
```

10.3.2. I/O exceptions and two ways to close a stream

Two ways to close a Stream

- The second approach to closing a stream is to automate the process by using the **try**-with-resources statement that was added by JDK 7 (and, of course, supported by JDK 8).
- The **try**-with-resources statement is an enhanced form of **try** that has the following form:

```
try (resource-specification) {
  // use the resource
}
```

- Here, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource.
- It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. Thus, there is no need to call **close( )** explicitly.
- key points about the **try**-with-resources statement:
  - Resources managed by try-with-resources must be objects of classes that implement **AutoCloseable**.
  - The resource declared in the **try** is implicitly **final**.
  - You can manage more than one resource by separating each declaration by a semicolon.
- Remember that the scope of the declared resource is limited to the try-with-resources statement.

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – InputStream

- The byte stream classes provide a rich environment for handling byte-oriented I/O.
- A byte stream can be used with any type of object, including binary data.
- Byte stream classes are topped by **InputStream** and **OutputStream**
- **InputStream** is an abstract class that defines Java's model of streaming byte input.
- It implements the **AutoCloseable** and **Closeable** interfaces.
- Most of the methods in this class will throw an **IOException** when an I/O error occurs.
- Following table shows the methods in **InputStream**

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an**IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – FileOutputStream

Method	Description
boolean markSupported( )	Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – OutputStream

- **OutputStream** is an abstract class that defines streaming byte output.
- It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors.
- Following table sjhows methods in **OutputStream**.

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*,int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – FileInputStream

- The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file.
- Two commonly used constructors are shown here:
  - FileInputStream(String *filePath*)
  - FileInputStream(File fileObj).
- The following example creates two **FileInputStream**s that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream
```

- Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream.
- When a **FileInputStream** is created, it is also opened for reading.
- **FileInputStream** overrides six of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**. `(f);`

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – FileInputStream

- Following example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes.

```
import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    try ( FileInputStream f =newFileInputStream("FileInputStreamDemo.java"))    {
        System.out.println("Total Available Bytes: " +(size = f.available()));
        int n = size/40;
        System.out.println("First " + n +" bytes of the file one read() at a time");
        for (int i=0; i < n; i++) {
           System.out.print((char) f.read());
        }
        System.out.println("\nStill Available: " + f.available());
        System.out.println("Reading the next " + n + " with one read(b[])");
        byte b[] = new byte[n];
        if (f.read(b) != n) {
          System.err.println("couldn't read " + n + " bytes.");
        }
```

10.3.3. The Byte stream – FileInputStream and FileoutputStream

```
            System.out.println(new String(b, 0, n));
            System.out.println("\nStill Available: " + (size = f.available()));
            System.out.println("Skipping half of remaining bytes with skip()");
            f.skip(size/2);
            System.out.println("Still Available: " + f.available());
            System.out.println("Reading " + n/2 + " into the end of array");
            if (f.read(b, n/2, n/2) != n/2) {
               System.err.println("couldn't read " + n/2 + " bytes.");
            }
            System.out.println(new String(b, 0, b.length));
            System.out.println("\nStill Available: " + f.available());
        } catch(IOException e) {
        System.out.println("I/O Error: " + e);
      }
    }
}
```

Here is the output produced by this program:

Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
Still Available: 1741
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J
Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
if (
Still Available: 827

10.3.3. The Byte stream – FileInputStream and FileoutputStream

The Byte Streams – FileOutputStream

- **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file.
- It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- Four of its constructors are shown here:
  - FileOutputStream(String *filePath*)
  - FileOutputStream(File *fileObj*)
  - FileOutputStream(String *filePath*, boolean *append*)
  - FileOutputStream(File *fileObj*, boolean *append*)
- Creation of a **FileOutputStream** is not dependent on the file already existing.
- **FileOutputStream** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an exception will be thrown.
- The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

10.3.3. The Byte stream – FileInputStream and FileoutputStream

```
import java.io.*;
  class FileOutputStreamDemo {
    public static void main(String args[]) {
      String source = "Now is the time for all good men\n‟ + " to come to the
aid of their country\n‟ + " and pay their due taxes.";
      byte buf[] = source.getBytes();
      FileOutputStream f0 = null;
      FileOutputStream f1 = null;
      FileOutputStream f2 = null;
      try {
        f0 = new FileOutputStream("file1.txt");
        f1 = new FileOutputStream("file2.txt");
        f2 = new FileOutputStream("file3.txt");
       for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);
          f1.write(buf);
          f2.write(buf, buf.length-buf.length/4, buf.length/4);
     } catch(IOException e) {
       System.out.println("An I/O Error Occurred");
     } finally {
```

10.3.3. The Byte stream – FileInputStream and FileoutputStream

```
        try {
        if(f0 != null) f0.close();
        } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
       }
       try {
        if(f1 != null) f1.close();
       } catch(IOException e) {
         System.out.println("Error Closing file2.txt");
    }
    try {
      if(f2 != null) f2.close();
   } catch(IOException e) {
      System.out.println("Error Closing file3.txt");
    }
  }
 }
}
```

10.3.4. The character stream – FileReader and FileWriter

The Charahcter Streams

- While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.
- Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.

The Charahcter Streams – Reader

- **Reader** is an abstract class that defines Java's model of streaming character input.
- It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods inthis class (except for **markSupported( )**) will throw an **IOException** on error conditions.
- Table below shows a synopsis of the methods in **Reader**.

10.3.4. The character stream – FileReader and FileWriter

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
int read(CharBuffer *buffer*)	Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

10.3.4. The character stream – FileReader and FileWriter

The Charahcter Streams – Writer

- **Writer** is an abstract class that defines streaming character output.
- It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in
- Table below shows a synopsis of the methods in **Writer**.

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.

10.3.4. The character stream – FileReader and FileWriter

Method	Description
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*.

10.3.4. The character stream – FileReader and FileWriter

The Charahcter Streams – FileWriter

```
import java.io.*;
  class FileWriterDemo {
    public static void main(String args[]) throws IOException {
      String source = "Now is the time for all good men\n" + " to come to the aid
of their country\n" + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);
    try ( FileWriter f0 = new FileWriter("file1.txt");
      FileWriter f1 = new FileWriter("file2.txt");
      FileWriter f2 = new FileWriter("file3.txt") )
   {
   for (int i=0; i < buffer.length; i += 2) {
     f0.write(buffer[i]);
   }
   write to second file
   f1.write(buffer);
   // write to third file
   f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4
   } catch(IOException e) {
     System.out.println("An I/O Error Occurred");
  }
 }
}
```

10.3.4. The character stream – FileReader and FileWriter

The Charahcter Streams – FileReader

- The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Two commonly used constructors are shown here:
  - FileReader(String *filePath*)
  - FileReader(File *fileObj*)
- The following example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

```
import java.io.*;
  class FileReaderDemo {
    public static void main(String args[]) {
      try ( FileReader fr = new FileReader("FileReaderDemo.java") ){
       int c;
       while((c = fr.read()) != -1) System.out.print((char) c);
     } catch(IOException e) {
       System.out.println("I/O Error: " + e);
     }
   }
}
```

10.3.5. The character stream – BufferedReader and PrintWriter

BufferedReader

- **BufferedReader** improves performance by buffering input.
- It has two constructors:
  - BufferedReader(Reader *inputStream*) : creates a buffered character stream using a default buffer size.
  - BufferedReader(Reader *inputStream*, int *bufSize*): creates a buffered character stream using the size of the buffer is passed in *bufSize*
- Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.
- As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer.
- To support this, **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**.
- JDK 8 adds a new method to **BufferedReader** called **lines( )**, that returns a **Stream** reference to the sequence of lines read by the reader.
- The following example reworks the **BufferedInputStream** example:

10.3.5. The character stream – BufferedReader and PrintWriter

```
import java.io.*;
  class BufferedReaderDemo {
    public static void main(String args[]) throws IOException {
      String s = "This is a &copy; copyright symbol " + "but this is &copy not.\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
    int c;
    boolean marked = false;
    try ( BufferedReader f = new BufferedReader(in){
       while ((c = f.read()) != -1) {
         switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;} else {
          marked = false;}
     break;
     case ';':
      if (marked) {
        marked = false;
          System.out.print("(c)");} else
         System.out.print((char) c);
      break;
      case ' ':
        if (marked) {
          marked = false;
          f.reset();
          System.out.print("&");} else
     System.out.print((char) c);
     break;
     default:
       if (!marked)
        System.out.print((char) c);
      break;
}}} catch(IOException e) {
    System.out.println("I/O Error: " + e);
}}}
```

10.3.5. The character stream – BufferedReader and PrintWriter

PrintWriter

- **PrintWriter** is essentially a character-oriented version of **PrintStream**.
- It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- **PrintWriter** has several constructors.
  - PrintWriter(OutputStream *outputStream*)
  - PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
  - PrintWriter(Writer *outputStream*)
  - PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)
- The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.
  - PrintWriter(File *outputFile*) throws FileNotFoundException
  - PrintWriter(File *outputFile*, String *charSet*)
  - PrintWriter(String *outputFileName*) throws FileNotFoundException
  - PrintWriter(String *outputFileName*, String *charSet*)
- These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file.
- In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file.
- You can specify a character encoding by passing its name in *charSet*.

10.3.5. The character stream – BufferedReader and PrintWriter

PrintWriter

- **PrintWriter** supports the **print( )** and **println( )** methods for all types, including **Object**.
- If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.
- **PrintWriter** also supports the **printf( )** method. It works the same way it does in the **PrintStream** class described earlier: It allows you to specify the precise format of the data.
- Here is how **printf( )** is declared in **PrintWriter**:
  - PrintWriter printf(String *fmtString*, Object … *args*) : writes *args* to standard output in the format specified by *fmtString* using the default locale
  - PrintWriter printf(Locale *loc*, String *fmtString*, Object …*args*): lets you specify a locale
- Both return the invoking **PrintWriter**.
- The **format( )** method is also supported. It has these general forms:
  - PrintWriter format(String *fmtString*, Object … *args*)
  - PrintWriter format(Locale *loc*, String *fmtString*, Object … *args*)
- It works exactly like **printf( )**.

10.3.6. 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.
- At a later time, you may restore these objects by using the process of **deserialization**.
- Serialization is also needed to implement *Remote Method Invocation (RMI)*; RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
- An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.
- Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects.
- This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph.
- That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves.
- The object serialization and deserialization facilities have been designed to work correctly in these scenarios.
- If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.
- Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

10.3.6. Serialization

serializable

- Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities.
- The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.
- Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

Externalizable

- There are cases in which the programmer may need to have control over these processes of serialization and deseraillization.
- For example, it may be desirable to use compression or encryption techniques.
- The **Externalizable** interface is designed for these situations.
- The **Externalizable** interface defines these two methods:
  - void readExternal(ObjectInput *inStream*) ; *inStream* is the byte stream from which the object is to be read
  - void writeExternal(ObjectOutput *outStream*) ; *outStream* is the byte stream to which the object is to be written.

10.3.6. Serialization

ObjectOutput

- The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization.
- It defines the methods shown in below Table.
- Note especially the **writeObject( )** method. This is called to serialize an object.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

10.3.6. Serialization

ObjectOutputStream

- The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface.
- It is responsible for writing objects to a stream.
- One constructor of this class is:
  - ObjectOutputStream(OutputStream *outStream*) throws IOException
- Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.
- Several commonly used methods in this class are shown in following Table given in the next slide.

10.3.6. Serialization

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

10.3.6. Serialization

ObjectInput

- The **ObjectInput** interface extends the DataInput and AutoCloseable interfaces and defines the methods shown in Table below table.
- It supports object serialization.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

10.3.6. Serialization

ObjectInputStream

- The ObjectInputStream class extends the InputStream class and implements the ObjectInput interface.
- ObjectInputStream is responsible for reading objects from a stream.
- One constructor of this class is shown here:
  - ObjectInputStream(InputStream inStream) throws IOException: inStream is the input stream from which serialized objects should be read.
- Closing an ObjectInputStream automatically closes the underlying stream specified by inStream.
- Several commonly used methods in this class are shown in Table given in the next slide.

10.3.6. Serialization

ObjectInputStream

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
Boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.

10.3.6. Serialization

Example

- **MyClass** is defined to implement the **Serializable** interface.
- If this is not done,a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
class MyClass implements Serializable {
  String s;
  int i;
  double d;
  public MyClass(String s, int i, double d) {
   this.s = s;
   this.i = i;
   this.d = d;
  }
  public String toString() {
  return "s=" + s + "; i=" + i + "; d=" + d;
 }
}
```

The output is shown here:
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10

Summary

Stream Benefits

- The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task.
- The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.
- Java programs written to adhere to the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented.