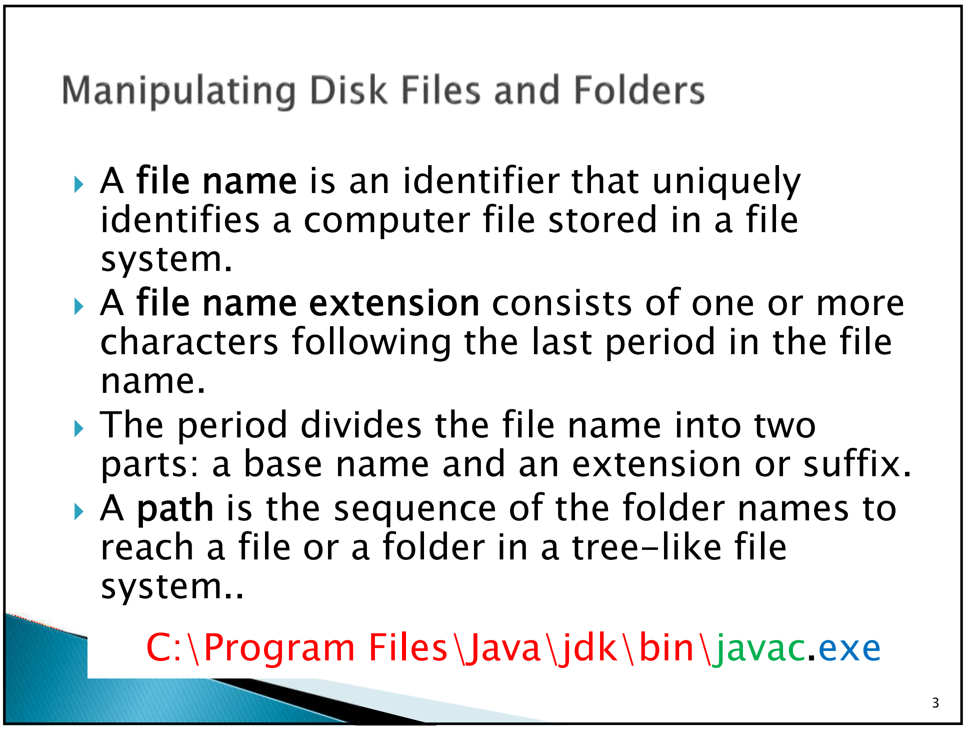
Chapter 7 Stream I/O

Manipulating Disk Files and Folders

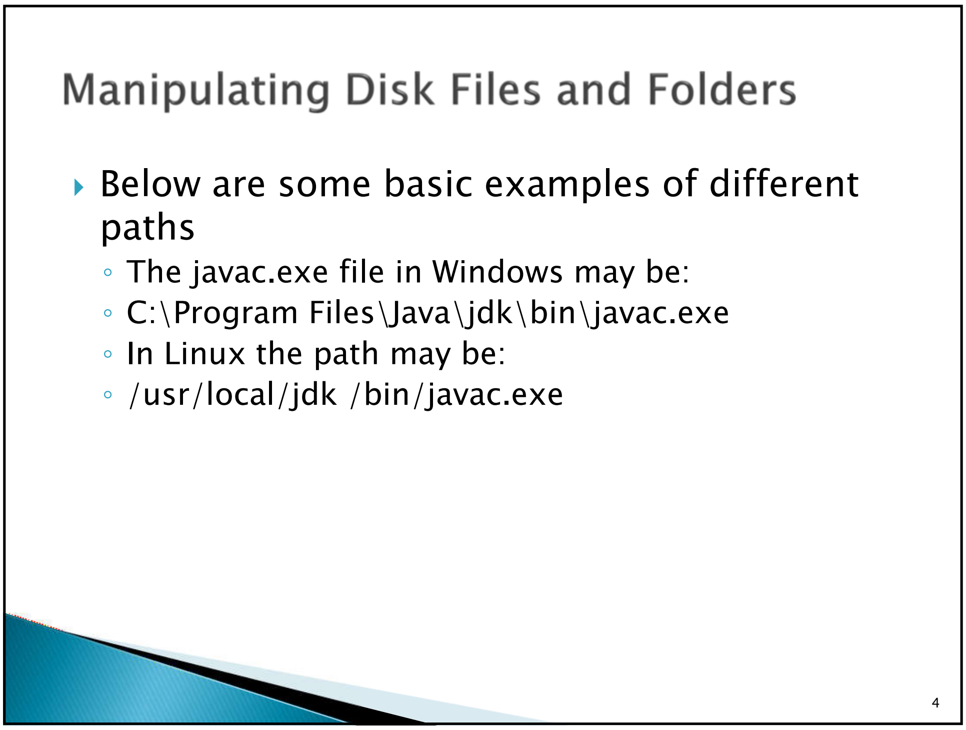
- A file is a logical grouping of related data.
- Files are organized into folder (sometimes called directories, we use those terms alternately).
- A folder is a hierarchical collection of folders and files.
- A volume is a collection of folders and files.



Manipulating Disk Files and Folders

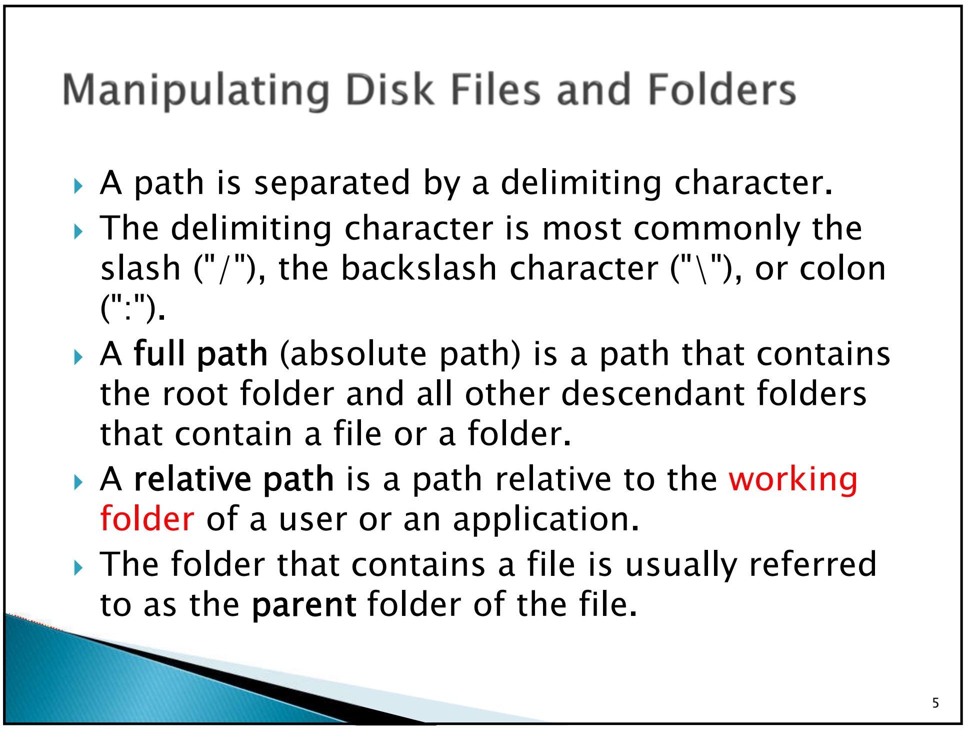
- A **file name** is an identifier that uniquely identifies a computer file stored in a file system.
- A **file name extension** consists of one or more characters following the last period in the file name.
- The period divides the file name into two parts: a base name and an extension or suffix.
- A **path** is the sequence of the folder names to reach a file or a folder in a tree-like file system..

C:\Program Files\Java\jdk\bin\javac.exe



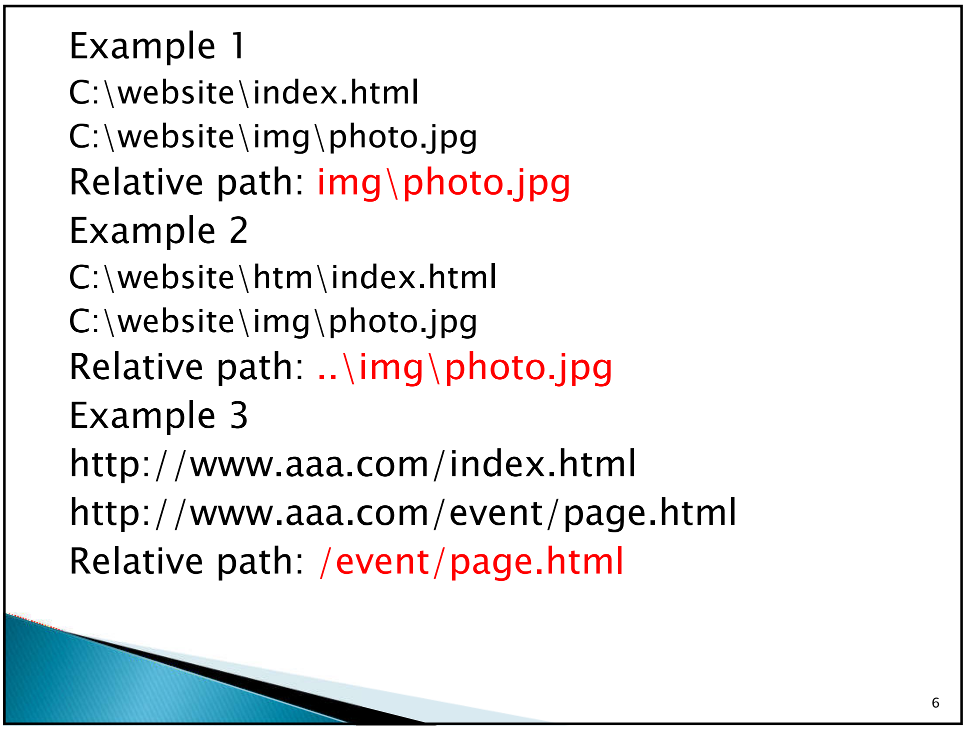
Manipulating Disk Files and Folders

- Below are some basic examples of different paths
  - The javac.exe file in Windows may be:
  - C:\Program Files\Java\jdk\bin\javac.exe
  - In Linux the path may be:
  - /usr/local/jdk /bin/javac.exe



Manipulating Disk Files and Folders

- A path is separated by a delimiting character.
- The delimiting character is most commonly the slash ("/"), the backslash character ("\"), or colon (":").
- A **full path** (absolute path) is a path that contains the root folder and all other descendant folders that contain a file or a folder.
- A **relative path** is a path relative to the working folder of a user or an application.
- The folder that contains a file is usually referred to as the **parent** folder of the file.



Example 1

C:\website\index.html

C:\website\img\photo.jpg

Relative path: img\photo.jpg

Example 2

C:\website\htm\index.html

C:\website\img\photo.jpg

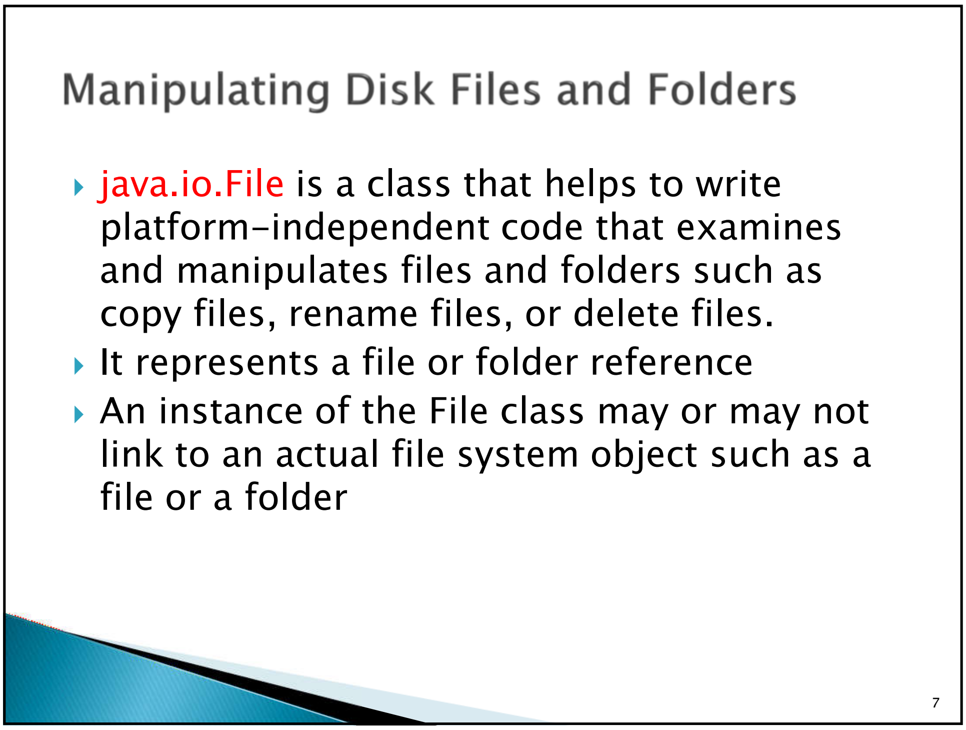
Relative path: ..\img\photo.jpg

Example 3

http://www.aaa.com/index.html

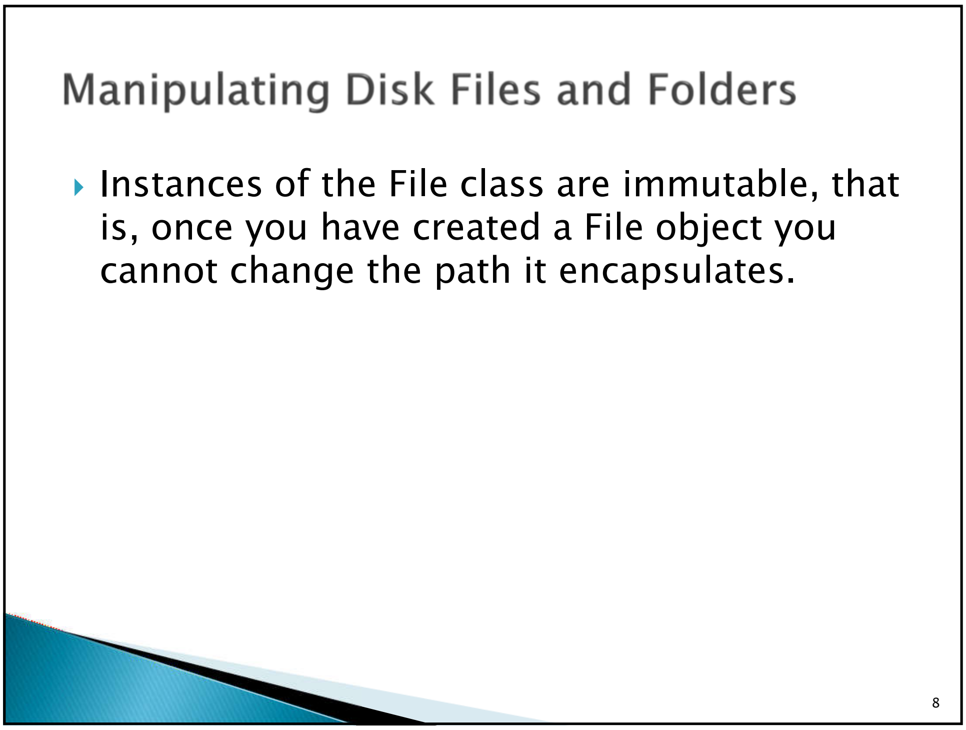
http://www.aaa.com/event/page.html

Relative path: /event/page.html



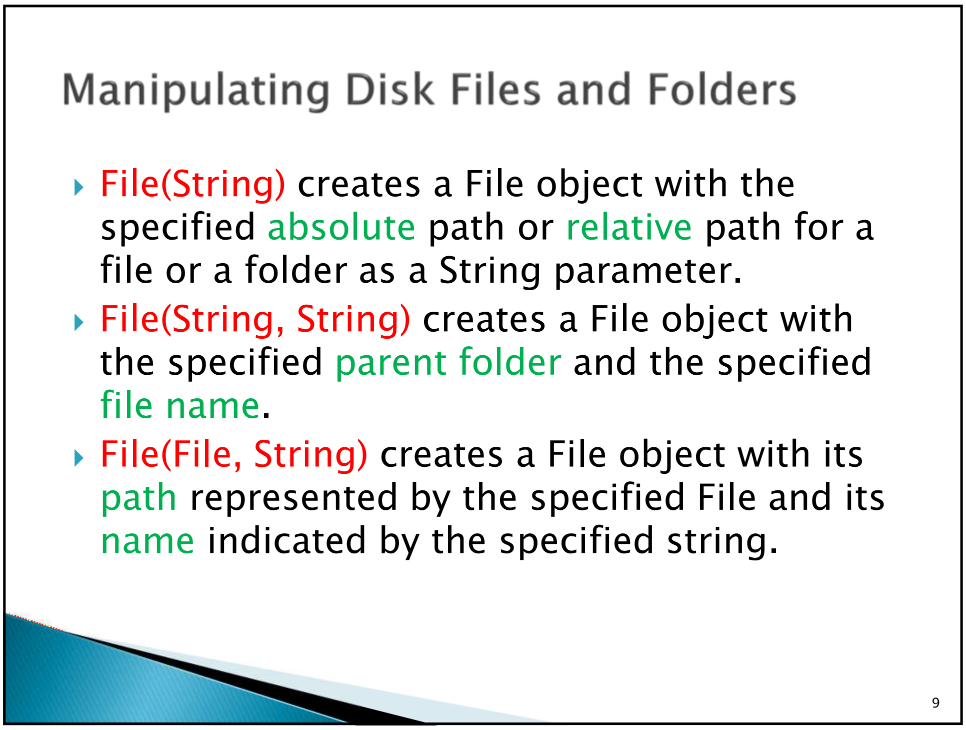
Manipulating Disk Files and Folders

- java.io.File is a class that helps to write platform-independent code that examines and manipulates files and folders such as copy files, rename files, or delete files.
- It represents a file or folder reference
- An instance of the File class may or may not link to an actual file system object such as a file or a folder



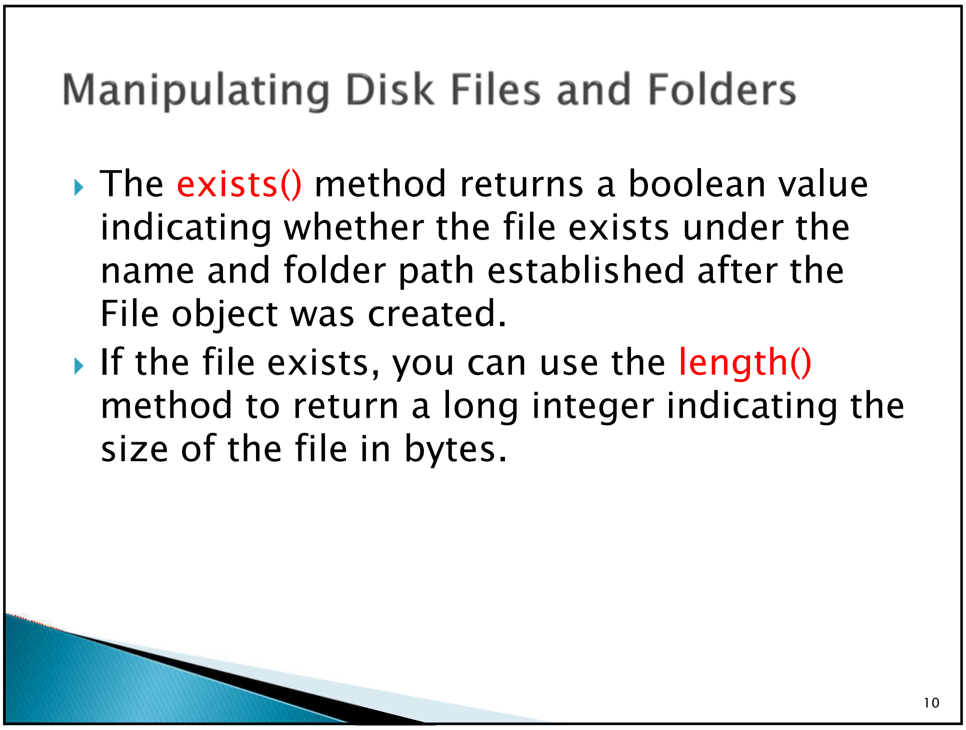
Manipulating Disk Files and Folders

- Instances of the File class are immutable, that is, once you have created a File object you cannot change the path it encapsulates.



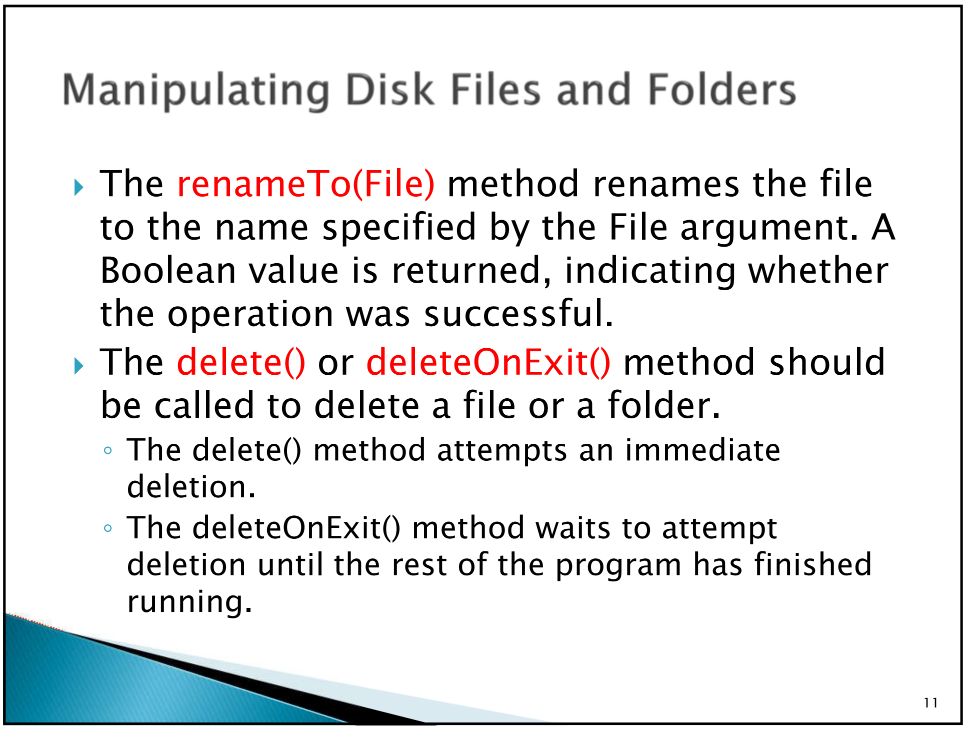
Manipulating Disk Files and Folders

- File(String) creates a File object with the specified absolute path or relative path for a file or a folder as a String parameter.
- File(String, String) creates a File object with the specified parent folder and the specified file name.
- File(File, String) creates a File object with its path represented by the specified File and its name indicated by the specified string.



Manipulating Disk Files and Folders

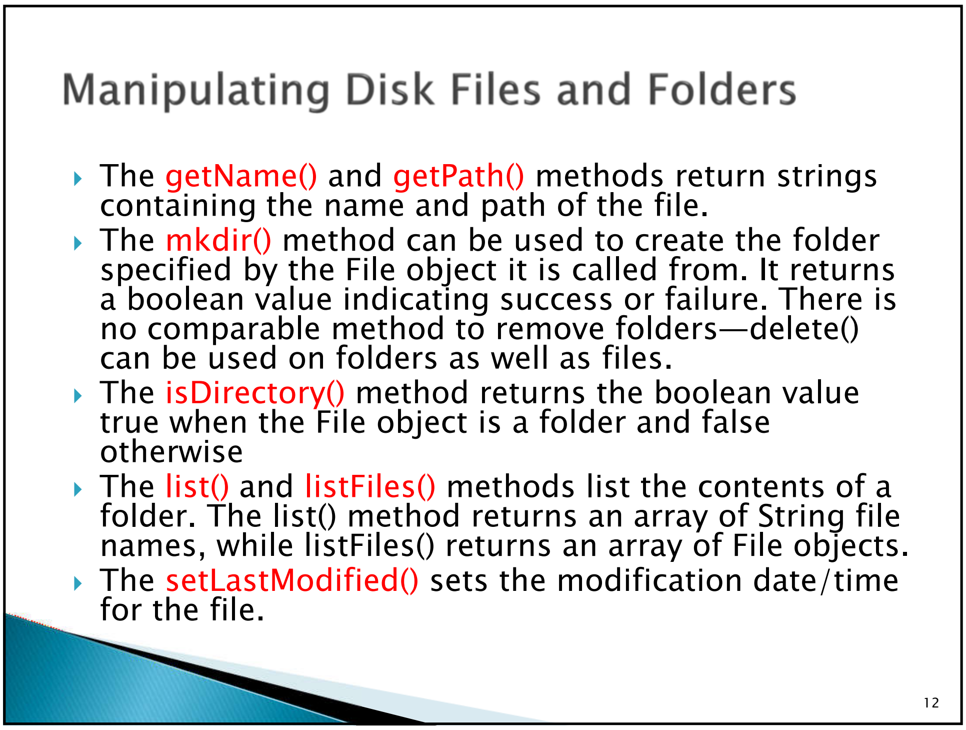
- The exists() method returns a boolean value indicating whether the file exists under the name and folder path established after the File object was created.
- If the file exists, you can use the length() method to return a long integer indicating the size of the file in bytes.



Manipulating Disk Files and Folders

- The renameTo(File) method renames the file to the name specified by the File argument. A Boolean value is returned, indicating whether the operation was successful.
- The delete() or deleteOnExit() method should be called to delete a file or a folder.
  - The delete() method attempts an immediate deletion.
  - The deleteOnExit() method waits to attempt deletion until the rest of the program has finished running.

11



Manipulating Disk Files and Folders

- The getName() and getPath() methods return strings containing the name and path of the file.
- The mkdir() method can be used to create the folder specified by the File object it is called from. It returns a boolean value indicating success or failure. There is no comparable method to remove folders—delete() can be used on folders as well as files.
- The isDirectory() method returns the boolean value true when the File object is a folder and false otherwise
- The list() and listFiles() methods list the contents of a folder. The list() method returns an array of String file names, while listFiles() returns an array of File objects.
- The setLastModified() sets the modification date/time for the file.

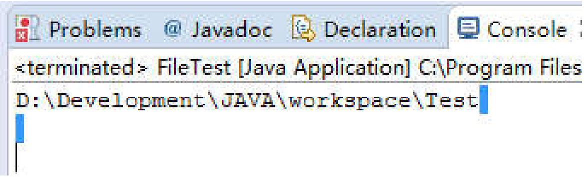
12

Manipulating Disk Files and Folders

```
import java.io.File;
public class NewFileTest {
    public static void main(String[] args) {
        File f = new File("D:\\work\\Sayyou.txt");
        try {
            if (f.exists()) {
                f.delete();
            } else {
                f.createNewFile();
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- specify the path for Sayyou.txt in a slightly more system-independent way, like this:
- File f = new File("E:" + File.separator + "work" + File.separator + "Sayyou.txt");

```
public class FileTest {
    public static void main(String[] args) {
        File file = new File("");
        System.out.println(file.getAbsolutePath());
        System.out.println(file);
    }
}
```



```
<terminated> FileTest [Java Application] C:\Program Files
D:\Development\JAVA\workspace\Test
```

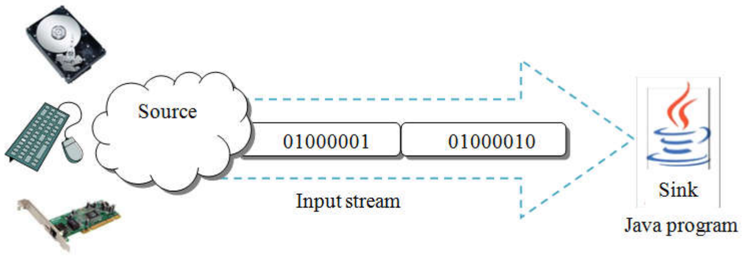
- File file = new File(“\files\setting.xml");

Streams

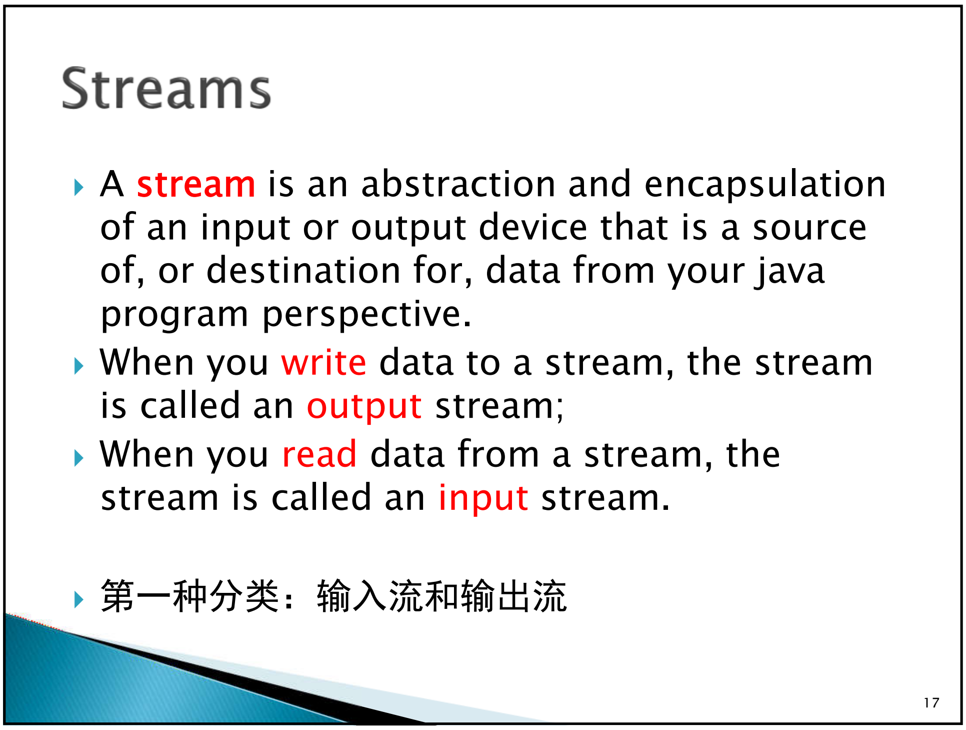
- A stream is a sequential and contiguous one-way flow of data (just like cars on highway).
- Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program.
- A Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream

15

Streams



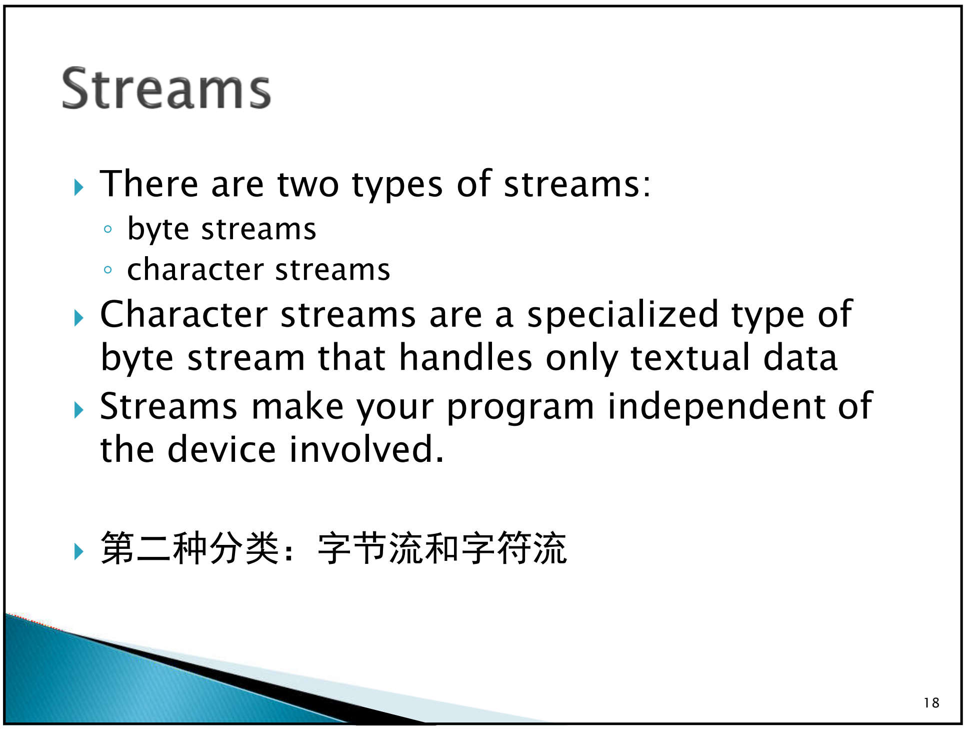
16



Streams

- A stream is an abstraction and encapsulation of an input or output device that is a source of, or destination for, data from your java program perspective.
- When you write data to a stream, the stream is called an output stream;
- When you read data from a stream, the stream is called an input stream.

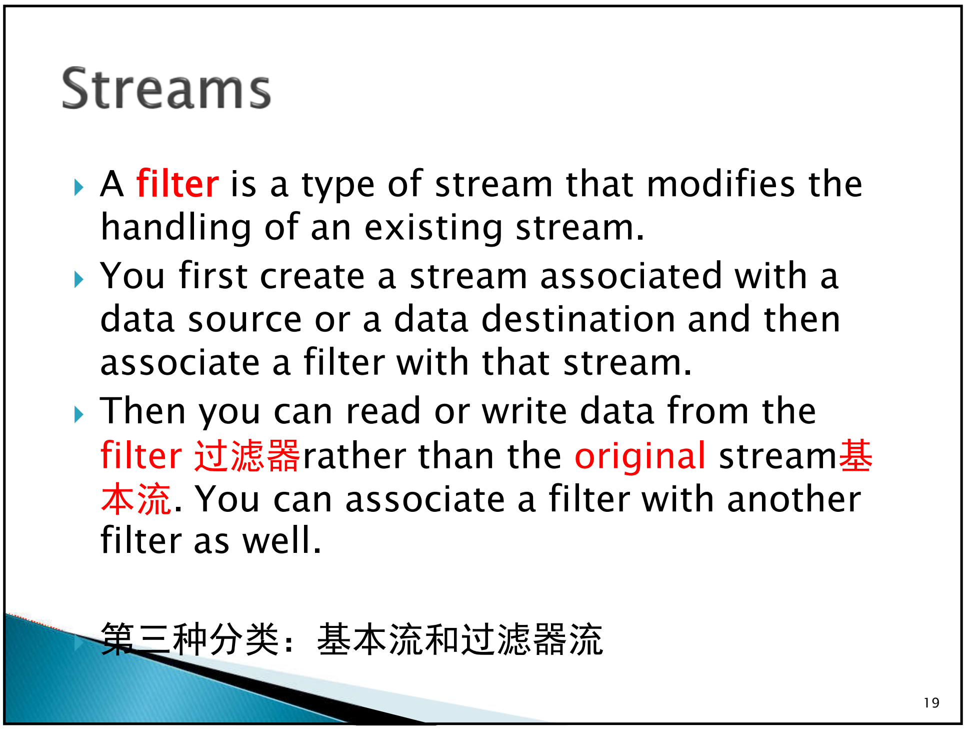
- 第一种分类：输入流和输出流



Streams

- There are two types of streams:
  - byte streams
  - character streams
- Character streams are a specialized type of byte stream that handles only textual data
- Streams make your program independent of the device involved.

- 第二种分类：字节流和字符流

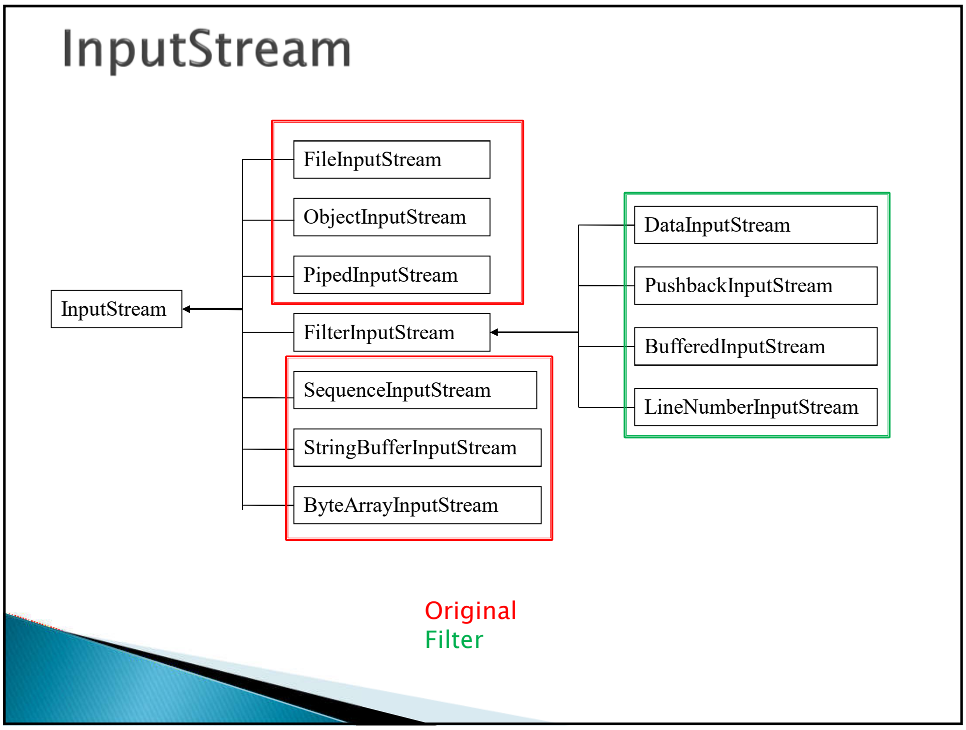


Streams

- A filter is a type of stream that modifies the handling of an existing stream.
- You first create a stream associated with a data source or a data destination and then associate a filter with that stream.
- Then you can read or write data from the filter 过滤器rather than the original stream基本流. You can associate a filter with another filter as well.

第三种分类：基本流和过滤器流

19



InputStream

InputStream

FileInputStream

ObjectInputStream

PipedInputStream

FilterInputStream

SequenceInputStream

StringBufferInputStream

ByteArrayInputStream

DataInputStream

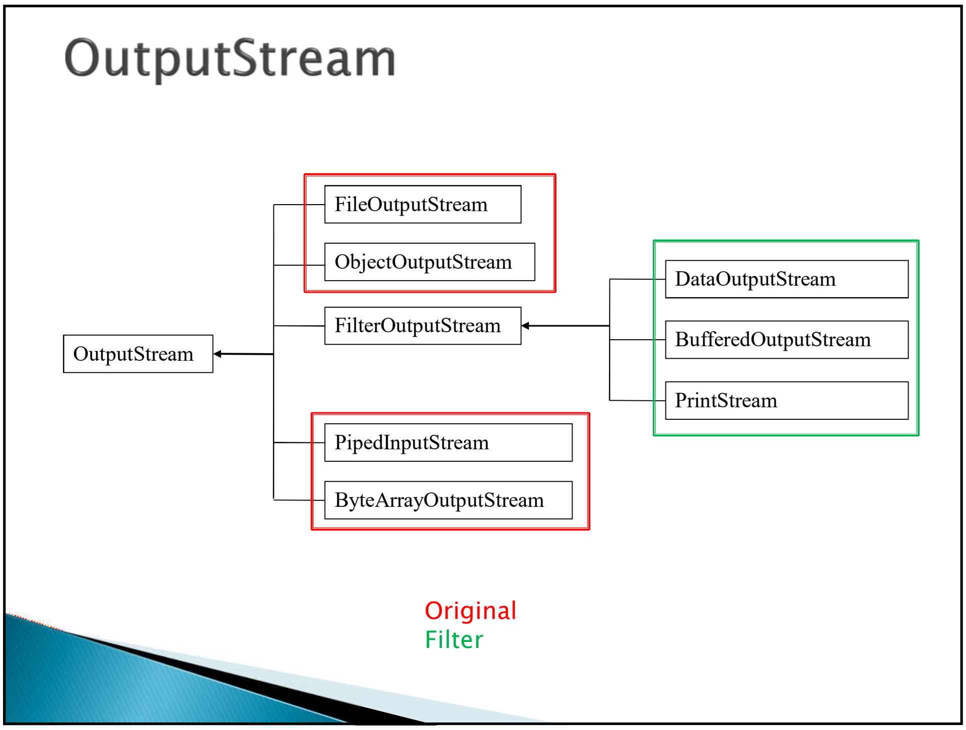
PushbackInputStream

BufferedInputStream

LineNumberInputStream

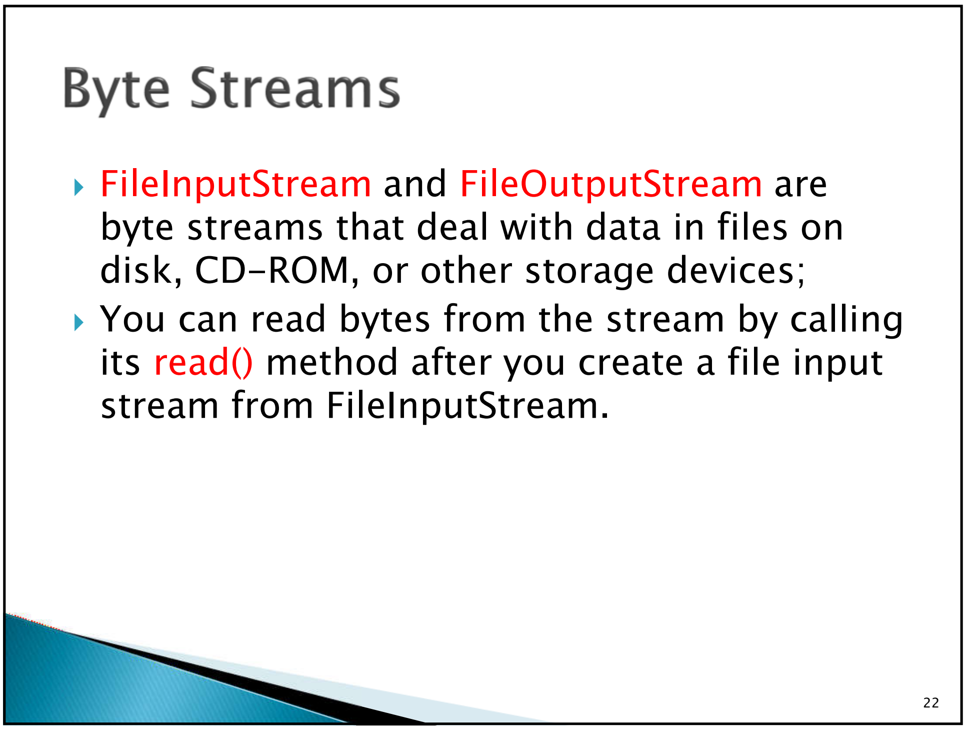
Original

Filter



OutputStream

Original
Filter



Byte Streams

- FileInputStream and FileOutputStream are byte streams that deal with data in files on disk, CD-ROM, or other storage devices;
- You can read bytes from the stream by calling its read() method after you create a file input stream from FileInputStream.

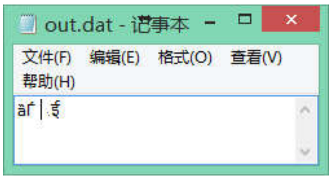
Byte Streams

- A file output stream can be created with the FileOutputStream(String) constructor.
- If the argument is the same as an existing file, the original will be wiped out when you start writing data to the stream.
- If you plan to append data after the end of an existing file, you can create a file output stream with the FileOutputStream(String, boolean) constructor.

Byte Streams

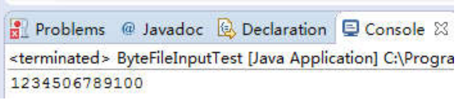
```
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteFileOutputTest {

        public static void main(String[] args) {
                byte[] data = { 1, 2, 3, 4, 5, 0, 6, 7, 8, 9, 10, 0 };
                File outFile = new File("out.dat");
                try {
                        FileOutputStream fout
                                = new FileOutputStream(outFile);
                        for (byte i : data)
                                fout.write(i);
                        System.out.println("Total bytes: " + outFile.length());
                        fout.close();
                } catch (IOException e) {
                        System.err.println(e.getMessage());
                }
        }
}
```

- Total bytes: 12



Byte Streams

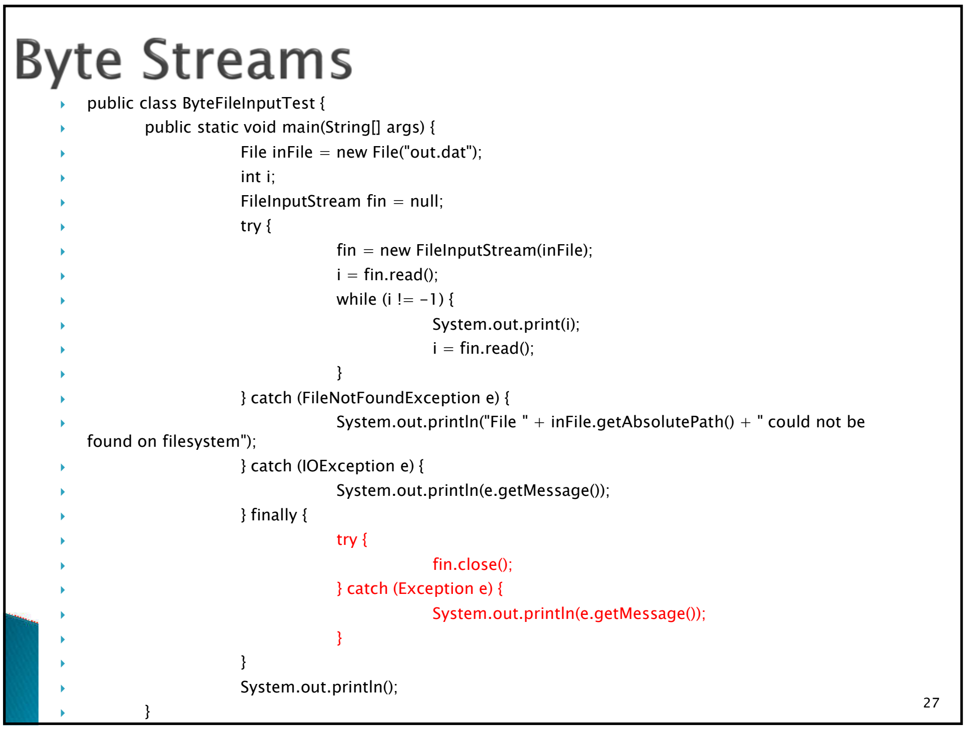
```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
public class ByteFileInputTest {
        public static void main(String[] args) {
                File inFile = new File("out.dat");
                int i;
                FileInputStream fin = null;
                try {
                        fin = new FileInputStream(inFile);
                        i = fin.read();
                        while (i != -1){
                                System.out.print(i);
                                i = fin.read();                         }
                        fin.close();
                } catch (FileNotFoundException e) {
        System.out.println("File " + inFile.getAbsolutePath() + " could not be found on filesystem");
                } catch (IOException e) {
                        System.out.println(e.getMessage());                     }
                System.out.println(); }
}
```



```
<terminated> ByteFileInputTest [Java Application] C:\Progra
1234506789100
```

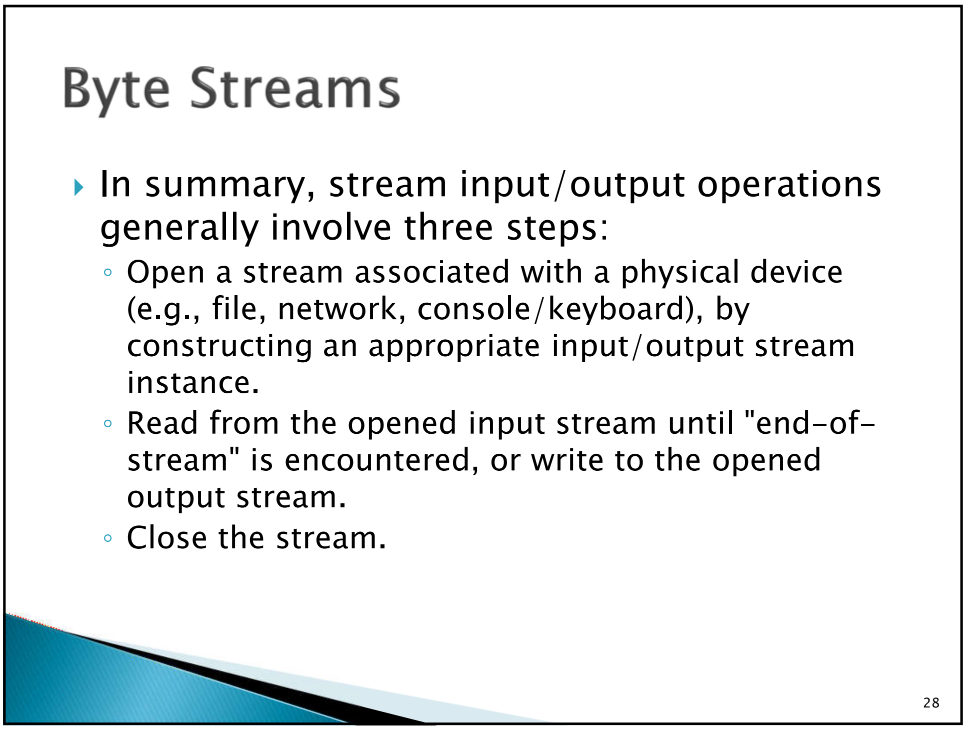
Byte Streams

- Streams occupy system resources which you must always clean up explicitly, by calling the close() method.
- If an I/O exception occurs while reading the stream, fin, the close() on the stream will not work.
- This problem can be solved by putting the close() call in a finally clause.



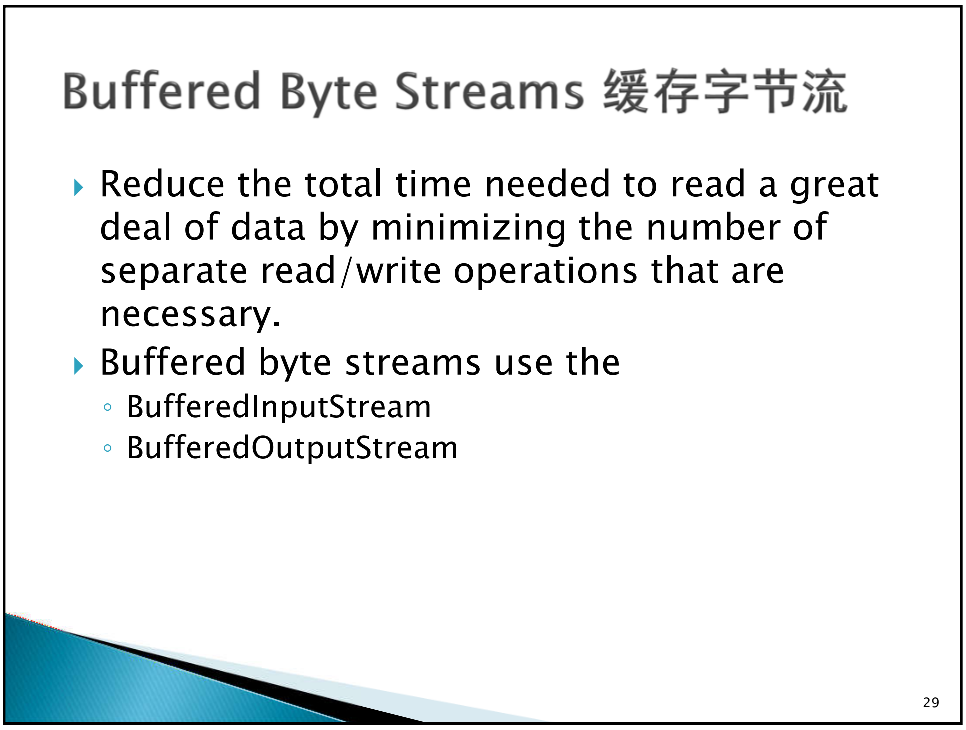
Byte Streams

```
public class ByteFileInputTest {
        public static void main(String[] args) {
                File inFile = new File("out.dat");
                int i;
                FileInputStream fin = null;
                try {
                        fin = new FileInputStream(inFile);
                        i = fin.read();
                        while (i != -1) {
                                System.out.print(i);
                                i = fin.read();
                        }
                } catch (FileNotFoundException e) {
                        System.out.println("File " + inFile.getAbsolutePath() + " could not be
found on filesystem");
                } catch (IOException e) {
                        System.out.println(e.getMessage());
                } finally {
                        try {
                                fin.close();
                        } catch (Exception e) {
                                System.out.println(e.getMessage());
                        }
                }
                System.out.println();
        }
```



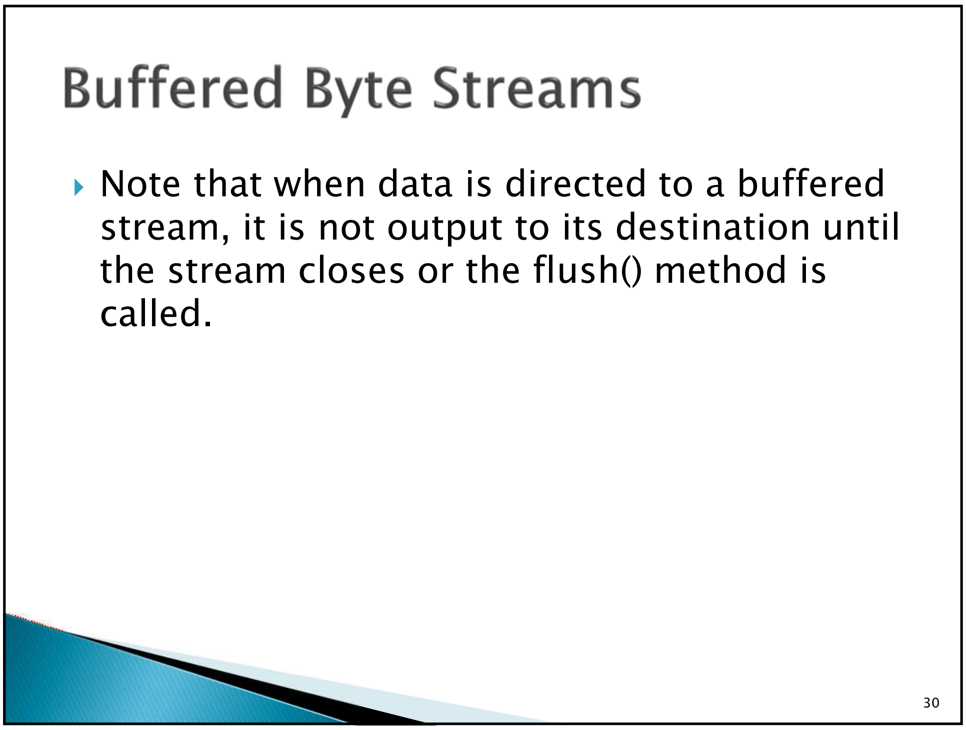
Byte Streams

- In summary, stream input/output operations generally involve three steps:
  - Open a stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate input/output stream instance.
  - Read from the opened input stream until "end-of-stream" is encountered, or write to the opened output stream.
  - Close the stream.



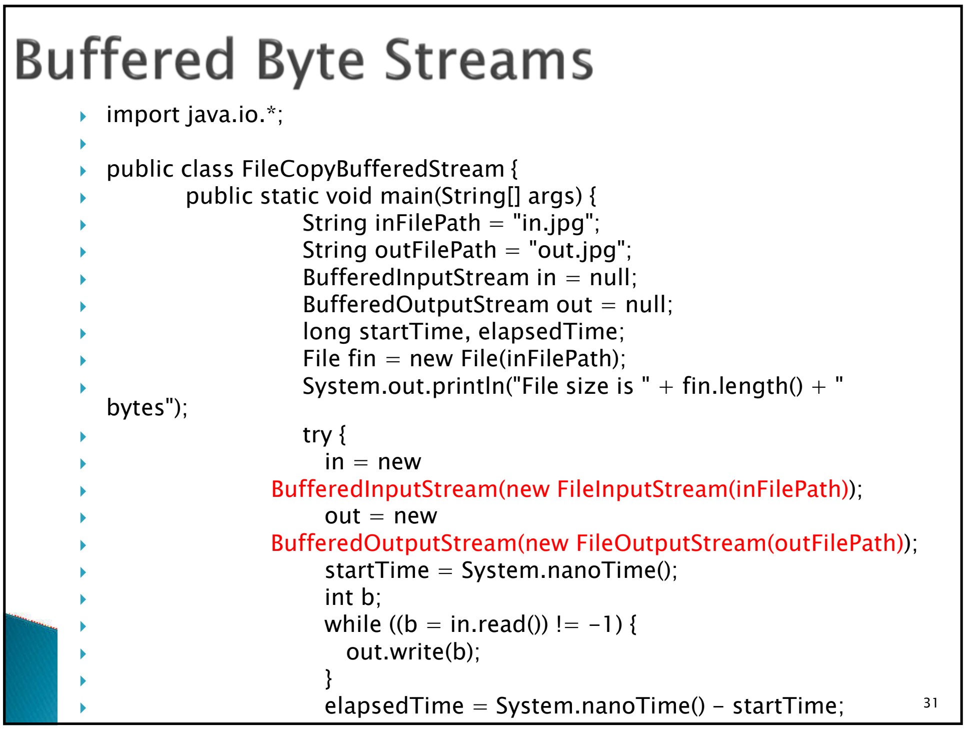
Buffered Byte Streams 缓存字节流

- Reduce the total time needed to read a great deal of data by minimizing the number of separate read/write operations that are necessary.
- Buffered byte streams use the
  - BufferedInputStream
  - BufferedOutputStream



Buffered Byte Streams

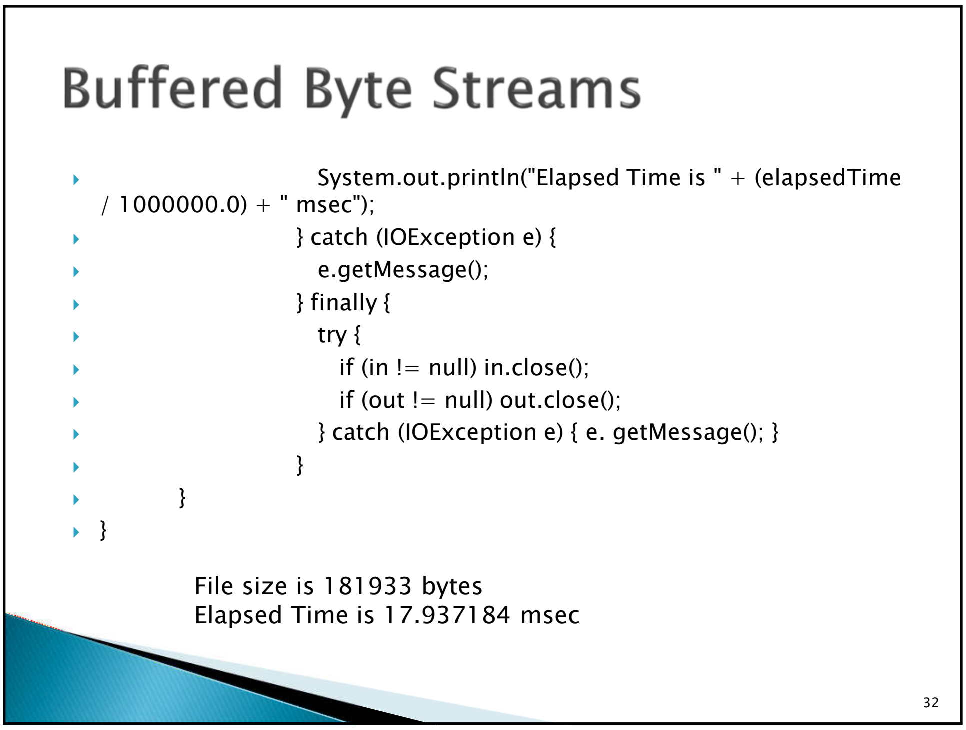
- Note that when data is directed to a buffered stream, it is not output to its destination until the stream closes or the flush() method is called.



Buffered Byte Streams

```
import java.io.*;

public class FileCopyBufferedStream {
          public static void main(String[] args) {
                         String inFilePath = "in.jpg";
                         String outFilePath = "out.jpg";
                         BufferedInputStream in = null;
                         BufferedOutputStream out = null;
                         long startTime, elapsedTime;
                         File fin = new File(inFilePath);
                         System.out.println("File size is " + fin.length() + "
bytes");
                         try {
                           in = new
                     BufferedInputStream(new FileInputStream(inFilePath));
                           out = new
                     BufferedOutputStream(new FileOutputStream(outFilePath));
                           startTime = System.nanoTime();
                           int b;
                           while ((b = in.read()) != -1) {
                              out.write(b);
                           }
                           elapsedTime = System.nanoTime() - startTime;
```

31

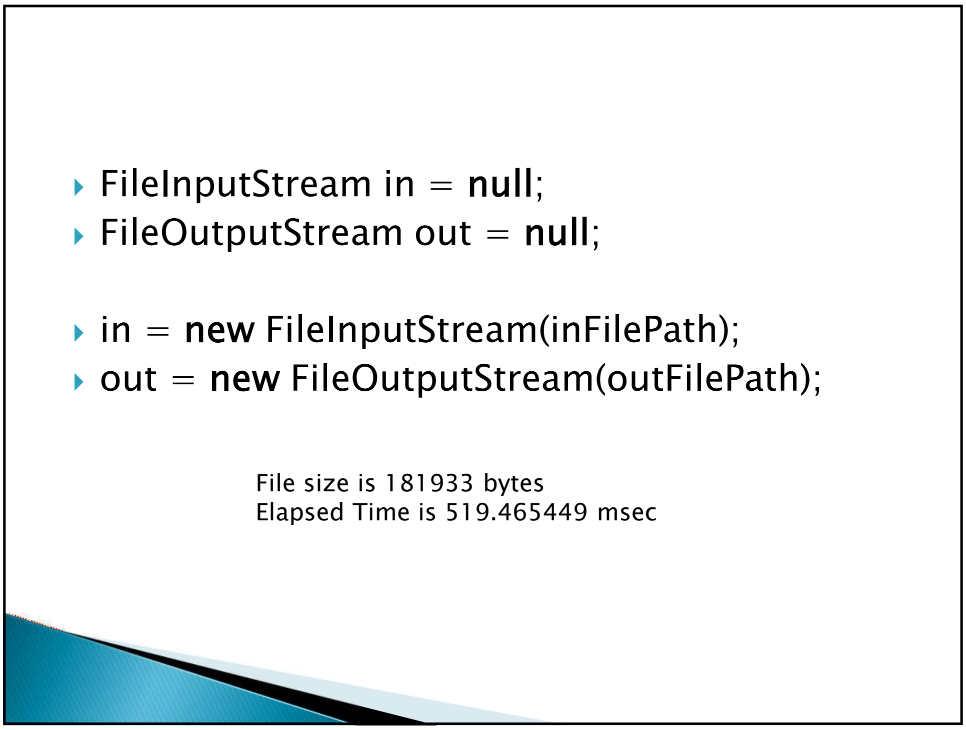


Buffered Byte Streams

```
                           System.out.println("Elapsed Time is " + (elapsedTime
/ 1000000.0) + " msec");
                        } catch (IOException e) {
                           e.getMessage();
                        } finally {
                           try {
                              if (in != null) in.close();
                              if (out != null) out.close();
                           } catch (IOException e) { e. getMessage(); }
                        }
          }
}
```

```
File size is 181933 bytes
Elapsed Time is 17.937184 msec
```

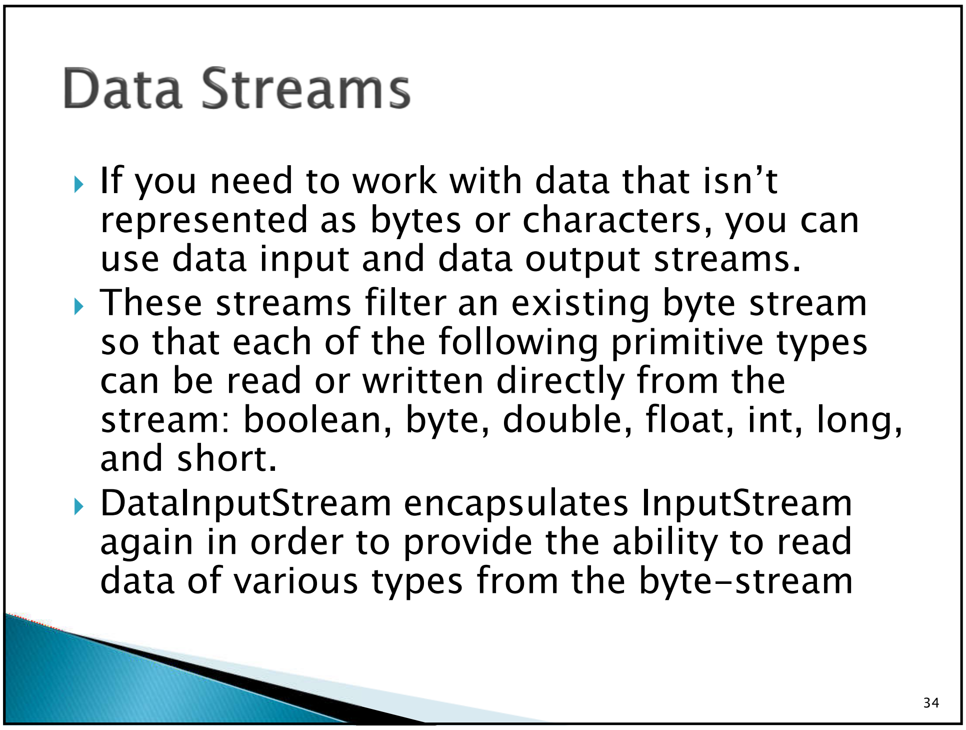
32



- FileInputStream in = **null**;
- FileOutputStream out = **null**;

- in = **new** FileInputStream(inFilePath);
- out = **new** FileOutputStream(outFilePath);

```
File size is 181933 bytes
Elapsed Time is 519.465449 msec
```



Data Streams

- If you need to work with data that isn’t represented as bytes or characters, you can use data input and data output streams.
- These streams filter an existing byte stream so that each of the following primitive types can be read or written directly from the stream: boolean, byte, double, float, int, long, and short.
- DataInputStream encapsulates InputStream again in order to provide the ability to read data of various types from the byte-stream

Data Streams

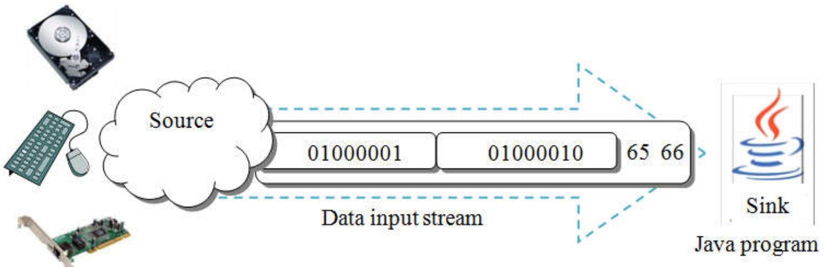
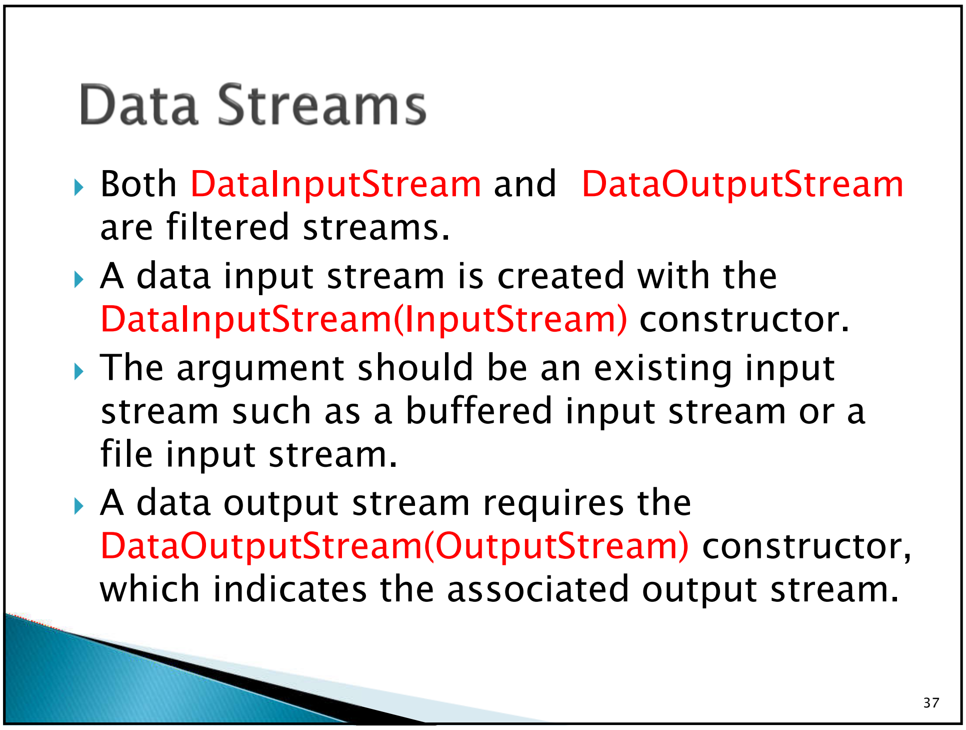


Fig. 6.12 DataInputStream is an encapsulation of InputStream

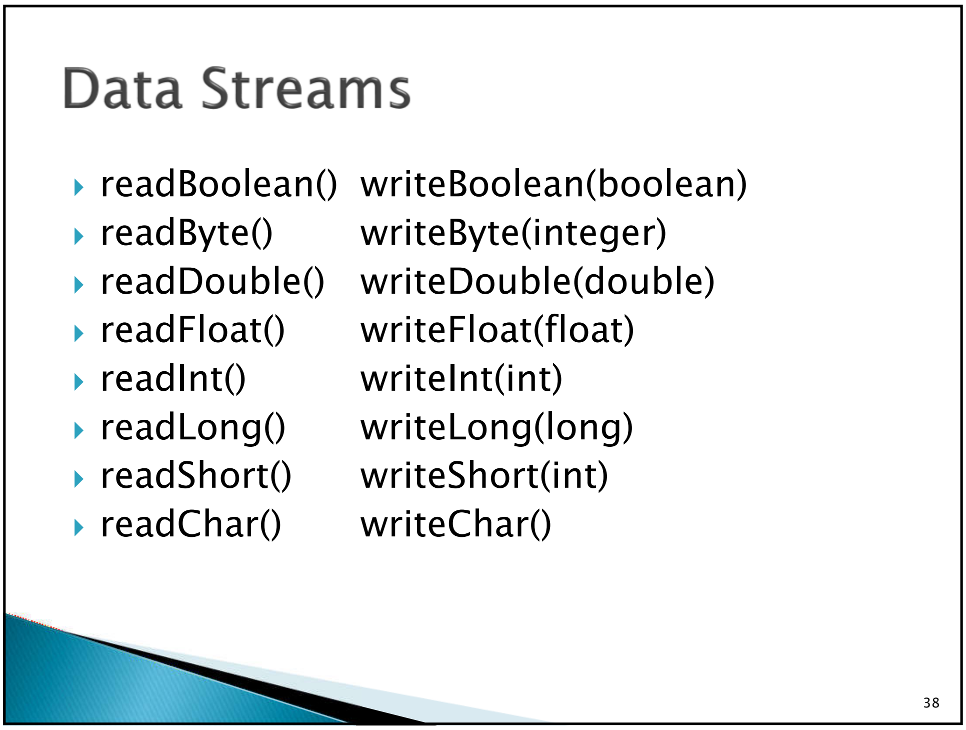
Data Streams

- OutputStream only has methods for outputting bytes, DataOutputStream has methods writeDouble(double x) for outputting values of type double, writeInt(int x) for outputting values of type int, and so on.



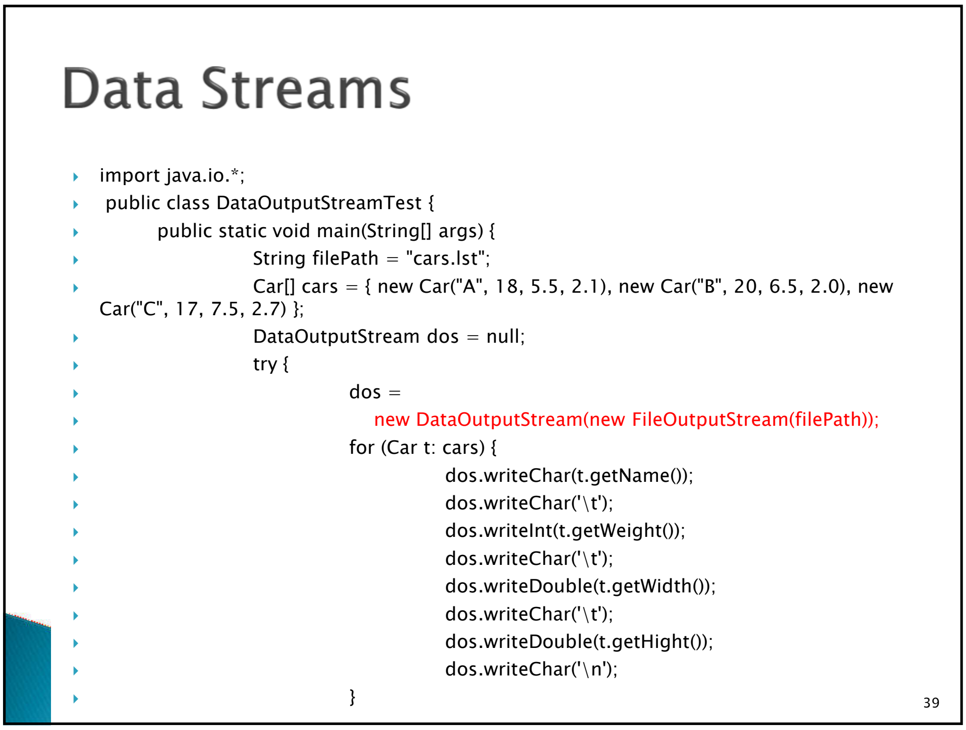
Data Streams

- Both DataInputStream and DataOutputStream are filtered streams.
- A data input stream is created with the DataInputStream(InputStream) constructor.
- The argument should be an existing input stream such as a buffered input stream or a file input stream.
- A data output stream requires the DataOutputStream(OutputStream) constructor, which indicates the associated output stream.



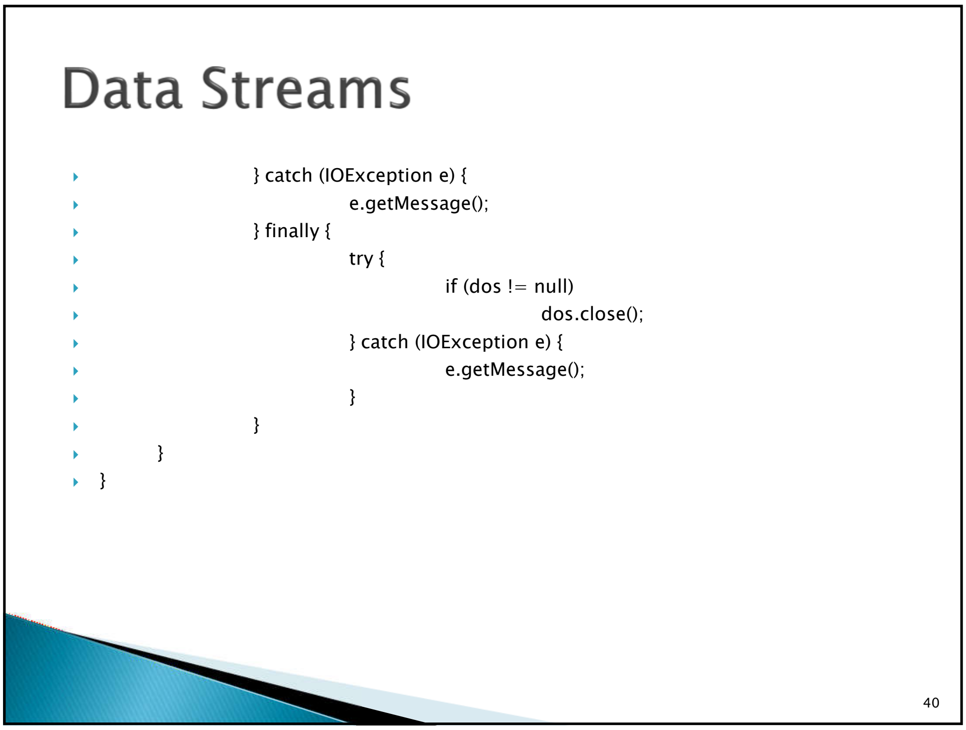
Data Streams

readBoolean()	writeBoolean(boolean)
readByte()	writeByte(integer)
readDouble()	writeDouble(double)
readFloat()	writeFloat(float)
readInt()	writeInt(int)
readLong()	writeLong(long)
readShort()	writeShort(int)
readChar()	writeChar()



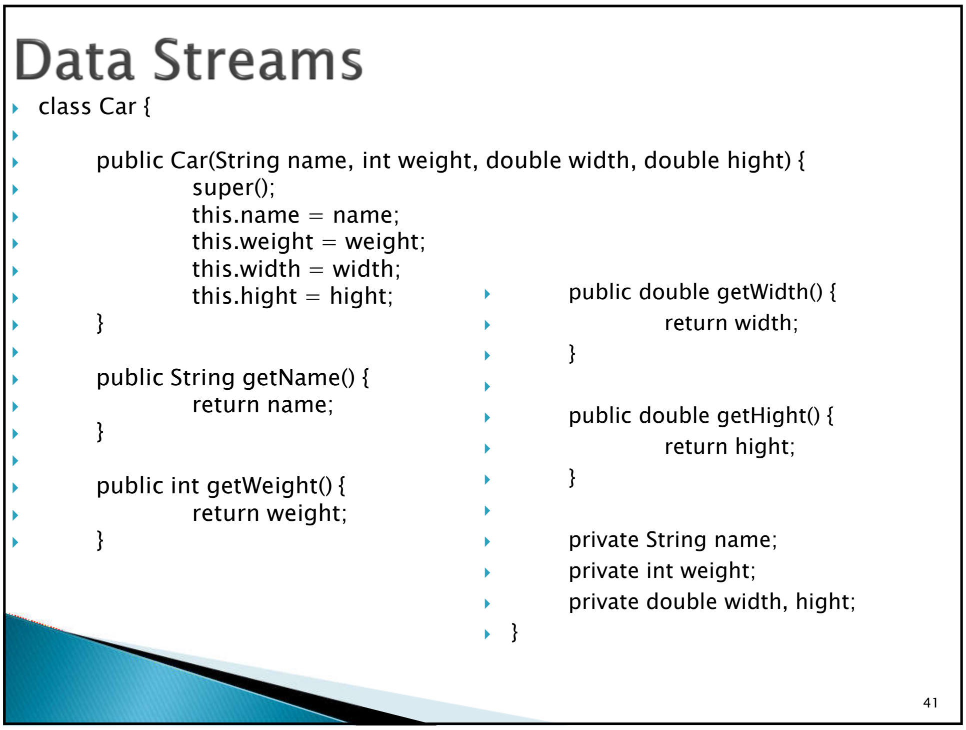
Data Streams

```
import java.io.*;
 public class DataOutputStreamTest {
        public static void main(String[] args) {
                String filePath = "cars.lst";
                Car[] cars = { new Car("A", 18, 5.5, 2.1), new Car("B", 20, 6.5, 2.0), new
Car("C", 17, 7.5, 2.7) };
                DataOutputStream dos = null;
                try {
                        dos =
                            new DataOutputStream(new FileOutputStream(filePath));
                        for (Car t: cars) {
                                dos.writeChar(t.getName());
                                dos.writeChar('\t');
                                dos.writeInt(t.getWeight());
                                dos.writeChar('\t');
                                dos.writeDouble(t.getWidth());
                                dos.writeChar('\t');
                                dos.writeDouble(t.getHight());
                                dos.writeChar('\n');
                        }
```



Data Streams

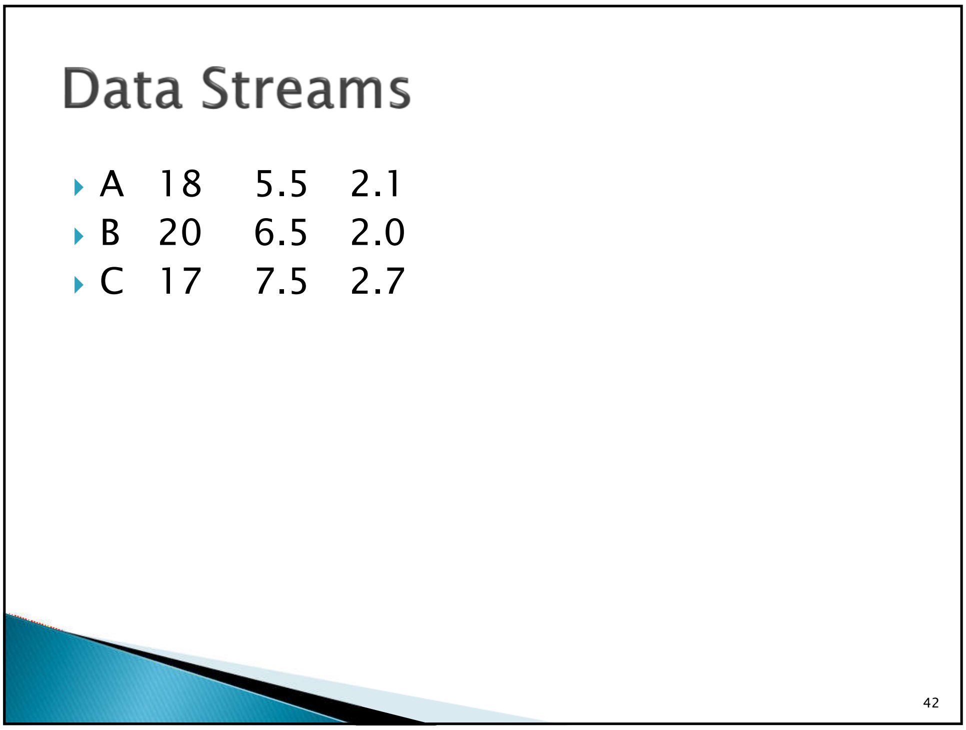
```
                } catch (IOException e) {
                        e.getMessage();
                } finally {
                        try {
                                if (dos != null)
                                        dos.close();
                        } catch (IOException e) {
                                e.getMessage();
                        }
                }
        }
}
```



Data Streams

```
class Car {

        public Car(String name, int weight, double width, double hight) {
                super();
                this.name = name;
                this.weight = weight;
                this.width = width;
                this.hight = hight;
        }

        public String getName() {
                return name;
        }

        public int getWeight() {
                return weight;
        }
        public double getWidth() {
                return width;
        }

        public double getHight() {
                return hight;
        }

        private String name;
        private int weight;
        private double width, hight;
}
```

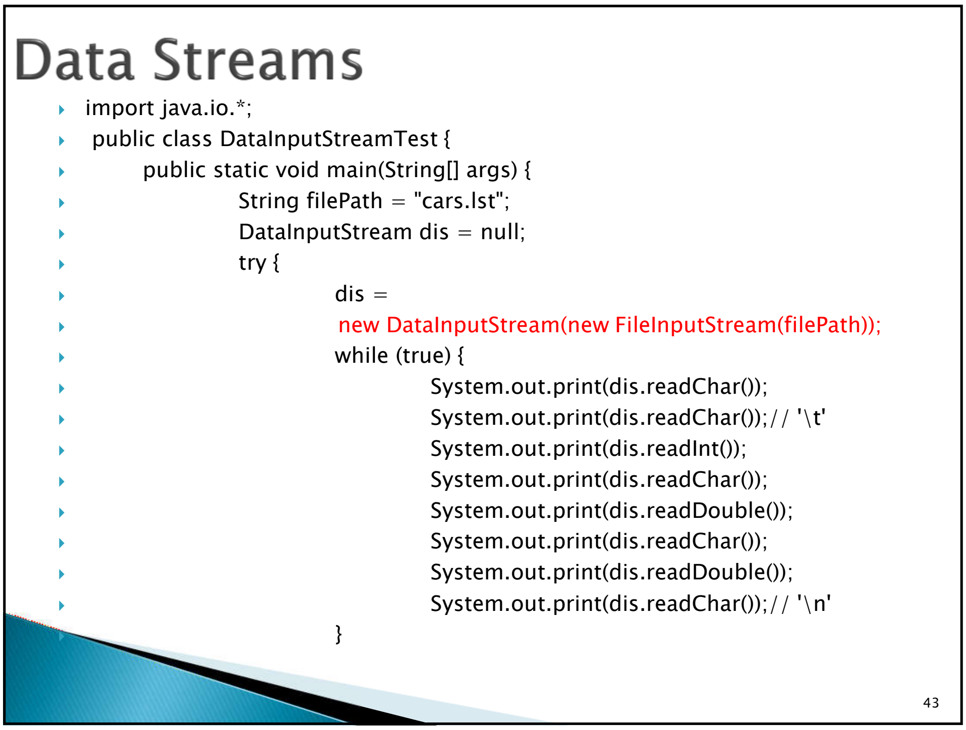
41



Data Streams

A	18	5.5	2.1
B	20	6.5	2.0
C	17	7.5	2.7

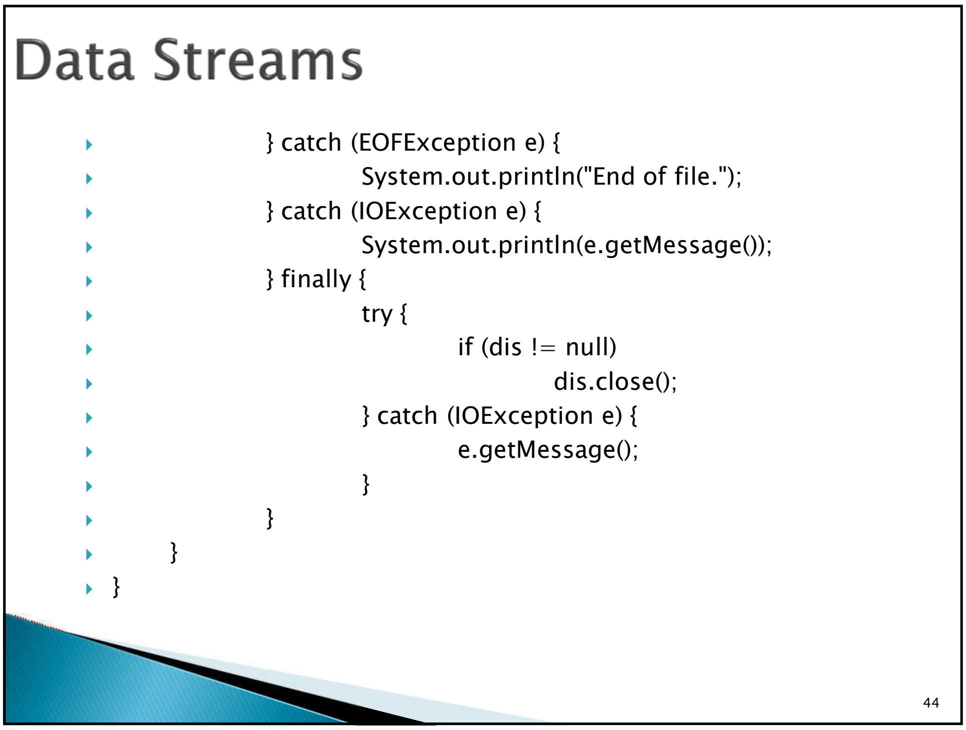
42



Data Streams

```
import java.io.*;
 public class DataInputStreamTest {
        public static void main(String[] args) {
                String filePath = "cars.lst";
                DataInputStream dis = null;
                try {
                        dis =
                         new DataInputStream(new FileInputStream(filePath));
                        while (true) {
                                System.out.print(dis.readChar());
                                System.out.print(dis.readChar());// '\t'
                                System.out.print(dis.readInt());
                                System.out.print(dis.readChar());
                                System.out.print(dis.readDouble());
                                System.out.print(dis.readChar());
                                System.out.print(dis.readDouble());
                                System.out.print(dis.readChar());// '\n'
                        }
```

43



Data Streams

```
                } catch (EOFException e) {
                        System.out.println("End of file.");
                } catch (IOException e) {
                        System.out.println(e.getMessage());
                } finally {
                        try {
                                if (dis != null)
                                        dis.close();
                        } catch (IOException e) {
                                e.getMessage();
                        }
                }
        }
 }
```

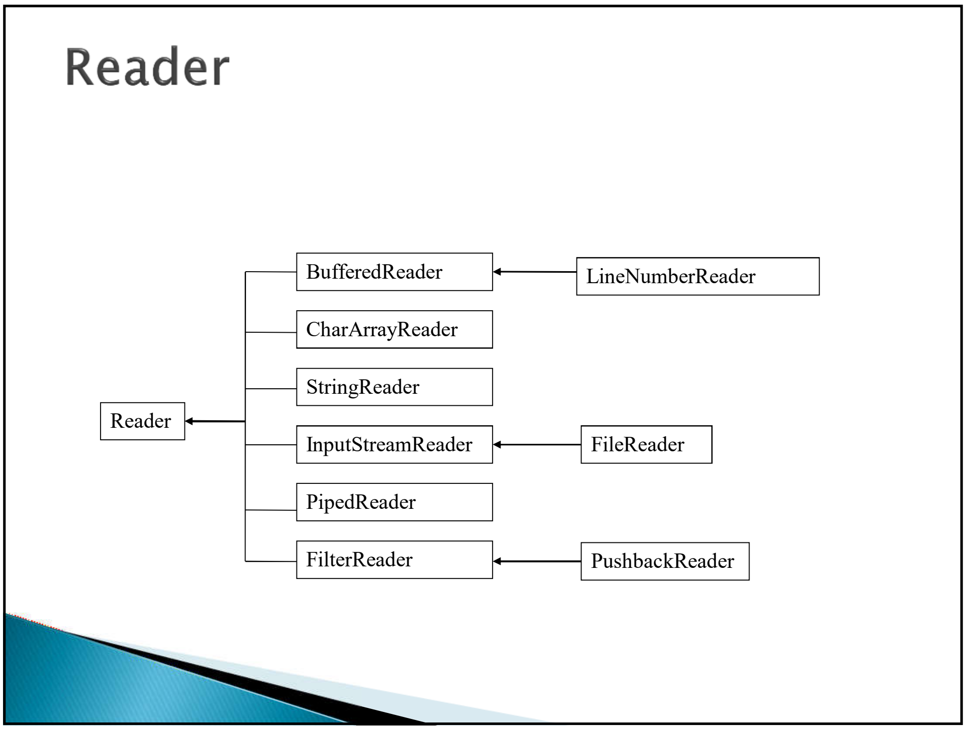
44

Character Streams

- Java IO's Reader and Writer work much like the InputStream and OutputStream except that Reader and Writer are character based while the InputStream and OutputStream are byte based.

Character Streams

- A text file consists of characters while a byte file consists of bytes.
- For example, an integer 12 may be
- 00110001 00110010 in a text file
- 00000000 00000000 00000000 00001100 in a byte file.



Reader

Reader

BufferedReader

LineNumberReader

CharArrayReader

StringReader

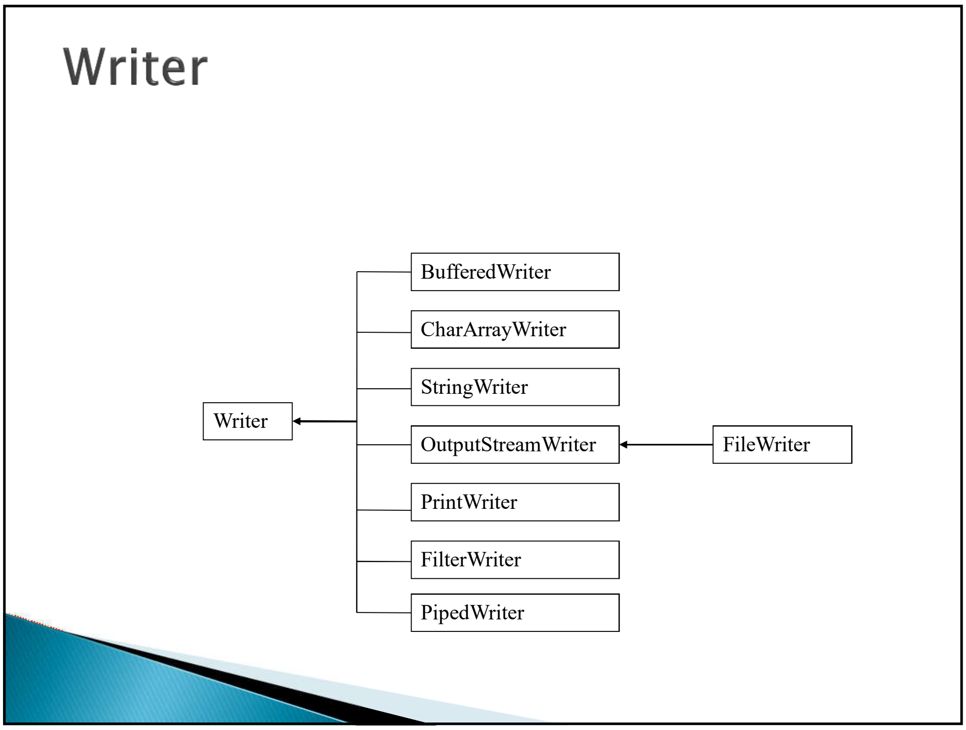
InputStreamReader

FileReader

PipedReader

FilterReader

PushbackReader



Writer

Writer

BufferedWriter

CharArrayWriter

StringWriter

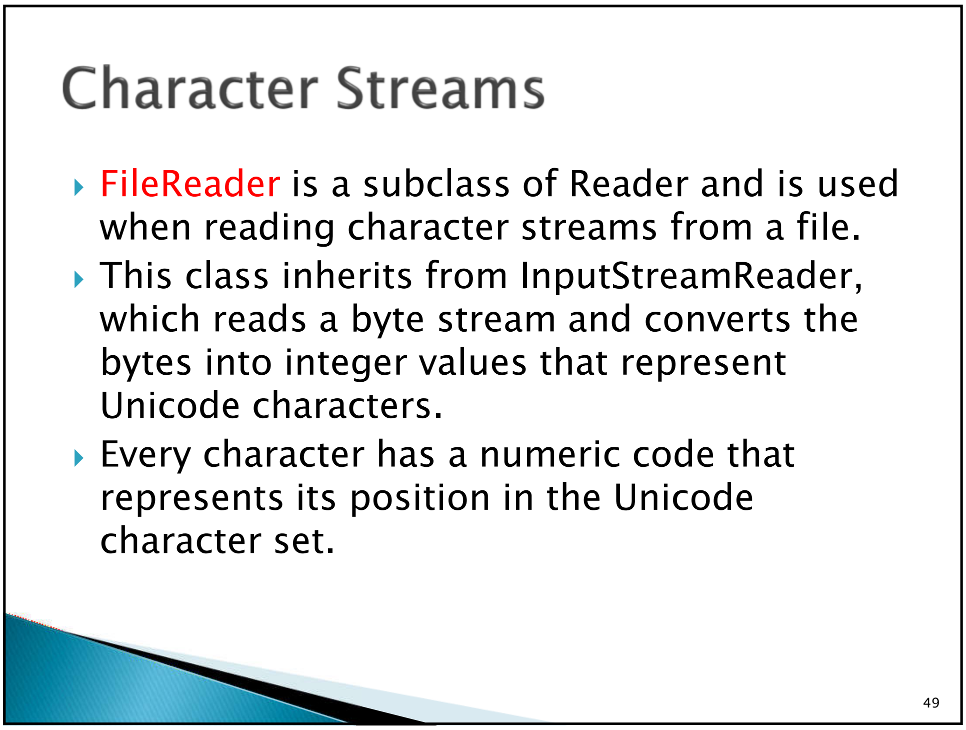
OutputStreamWriter

FileWriter

PrintWriter

FilterWriter

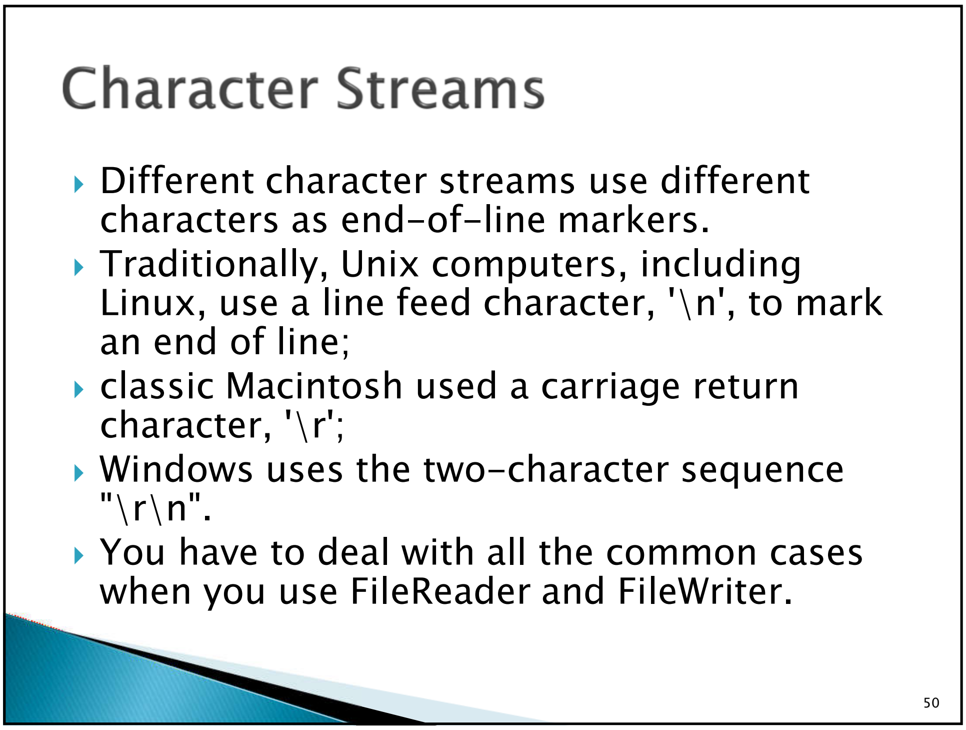
PipedWriter



Character Streams

- FileReader is a subclass of Reader and is used when reading character streams from a file.
- This class inherits from InputStreamReader, which reads a byte stream and converts the bytes into integer values that represent Unicode characters.
- Every character has a numeric code that represents its position in the Unicode character set.

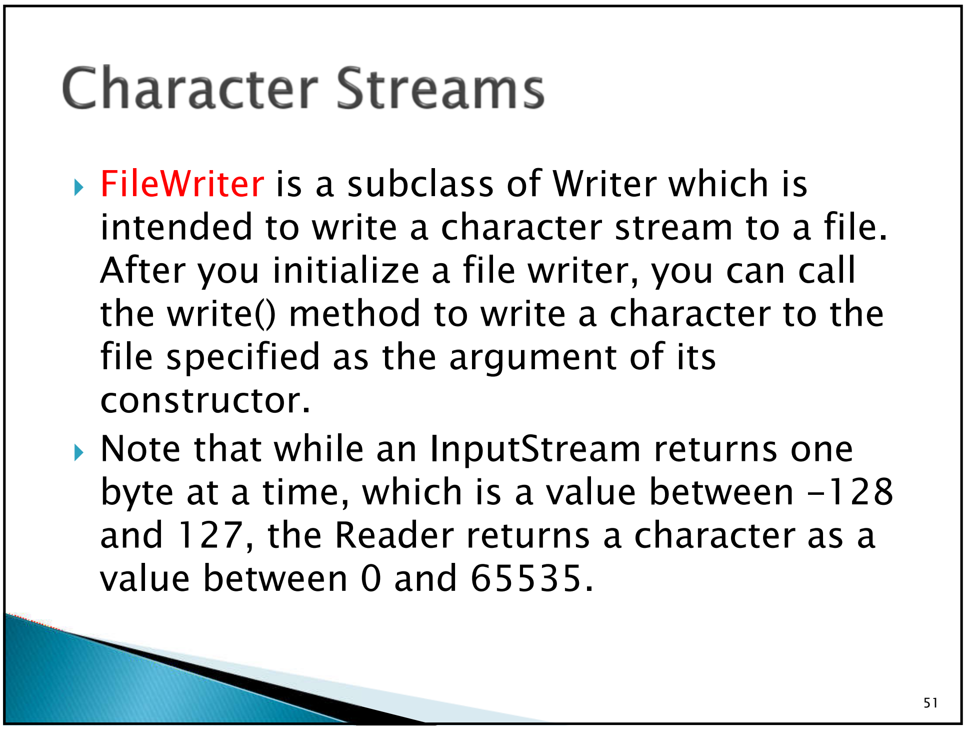
49



Character Streams

- Different character streams use different characters as end-of-line markers.
- Traditionally, Unix computers, including Linux, use a line feed character, '\n', to mark an end of line;
- classic Macintosh used a carriage return character, '\r';
- Windows uses the two-character sequence "\r\n".
- You have to deal with all the common cases when you use FileReader and FileWriter.

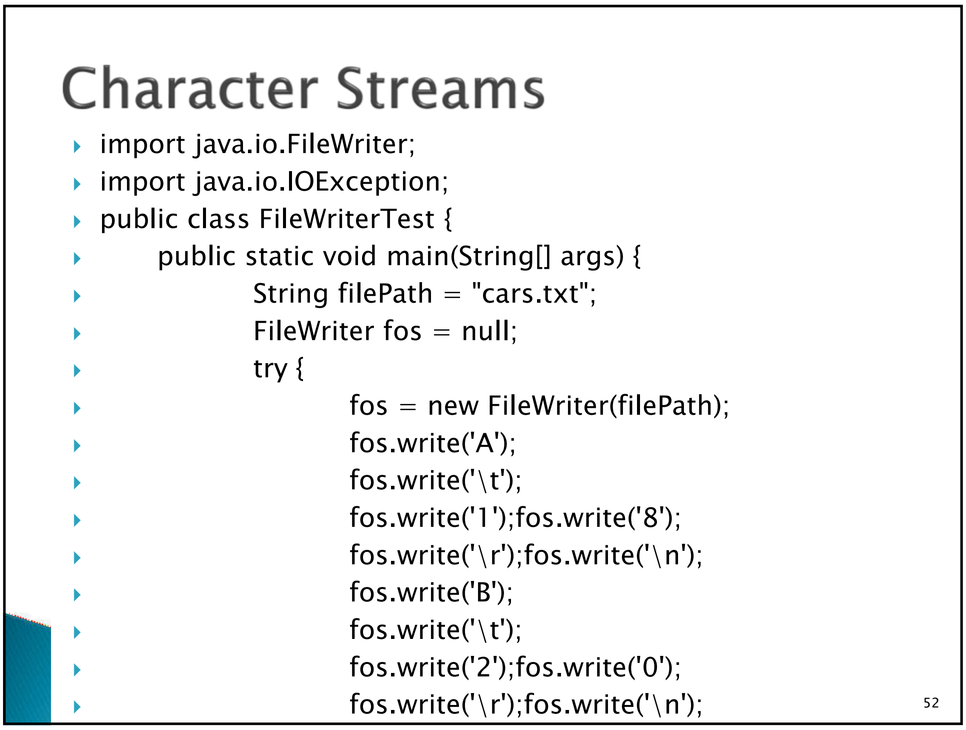
50



Character Streams

- FileWriter is a subclass of Writer which is intended to write a character stream to a file. After you initialize a file writer, you can call the write() method to write a character to the file specified as the argument of its constructor.
- Note that while an InputStream returns one byte at a time, which is a value between -128 and 127, the Reader returns a character as a value between 0 and 65535.

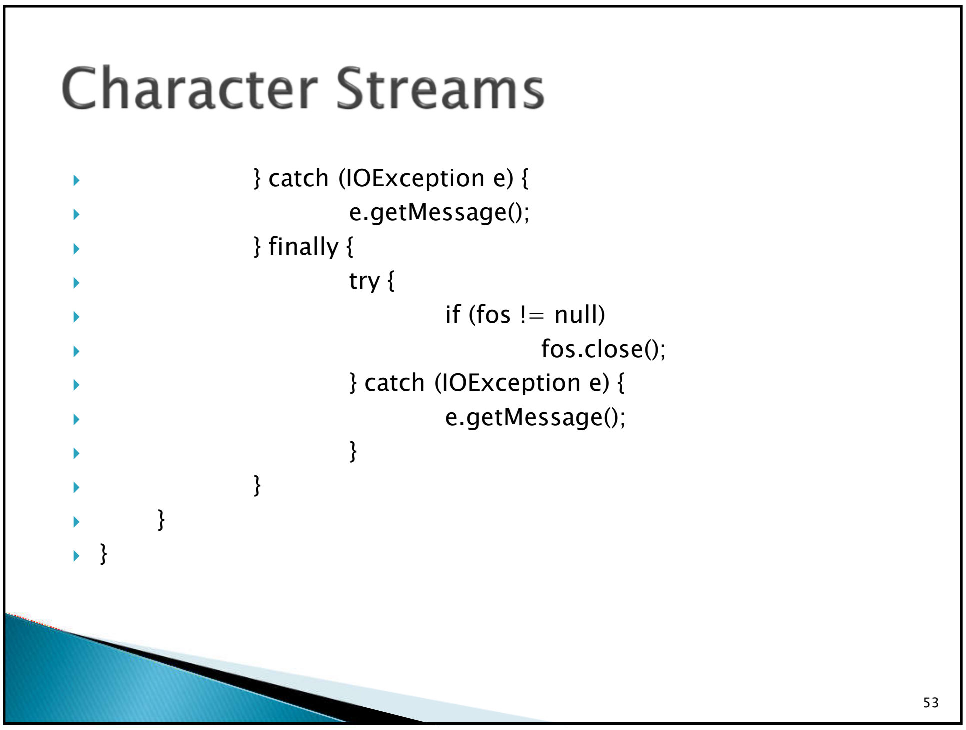
51



Character Streams

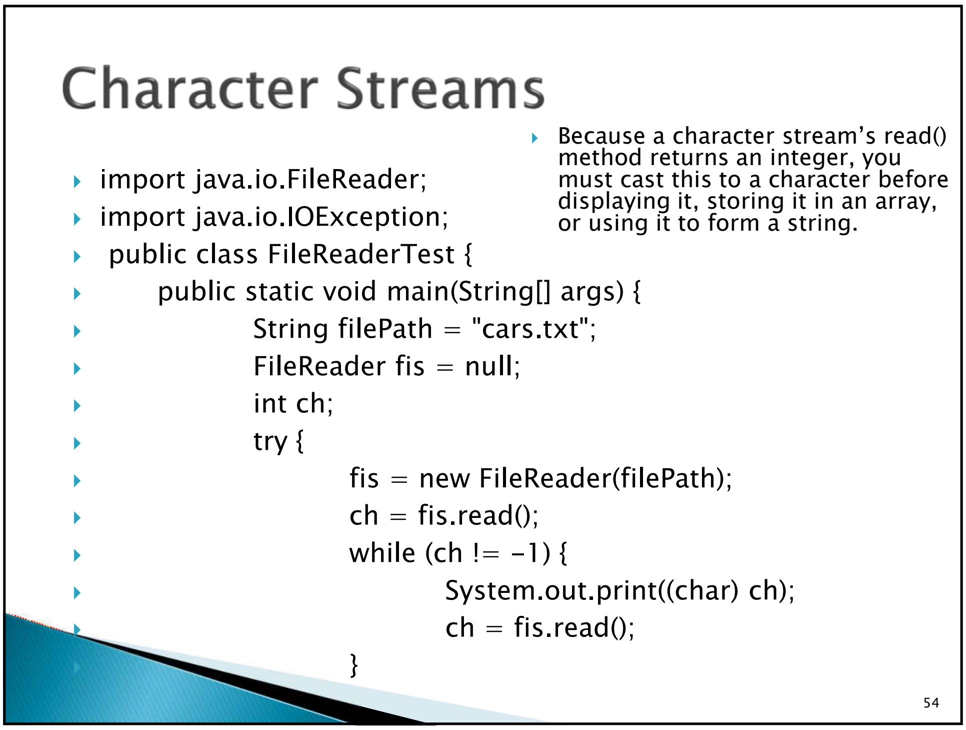
```
import java.io.FileWriter;
import java.io.IOException;
public class FileWriterTest {
    public static void main(String[] args) {
            String filePath = "cars.txt";
            FileWriter fos = null;
            try {
                    fos = new FileWriter(filePath);
                    fos.write('A');
                    fos.write('\t');
                    fos.write('1');fos.write('8');
                    fos.write('\r');fos.write('\n');
                    fos.write('B');
                    fos.write('\t');
                    fos.write('2');fos.write('0');
                    fos.write('\r');fos.write('\n');
```

52



Character Streams

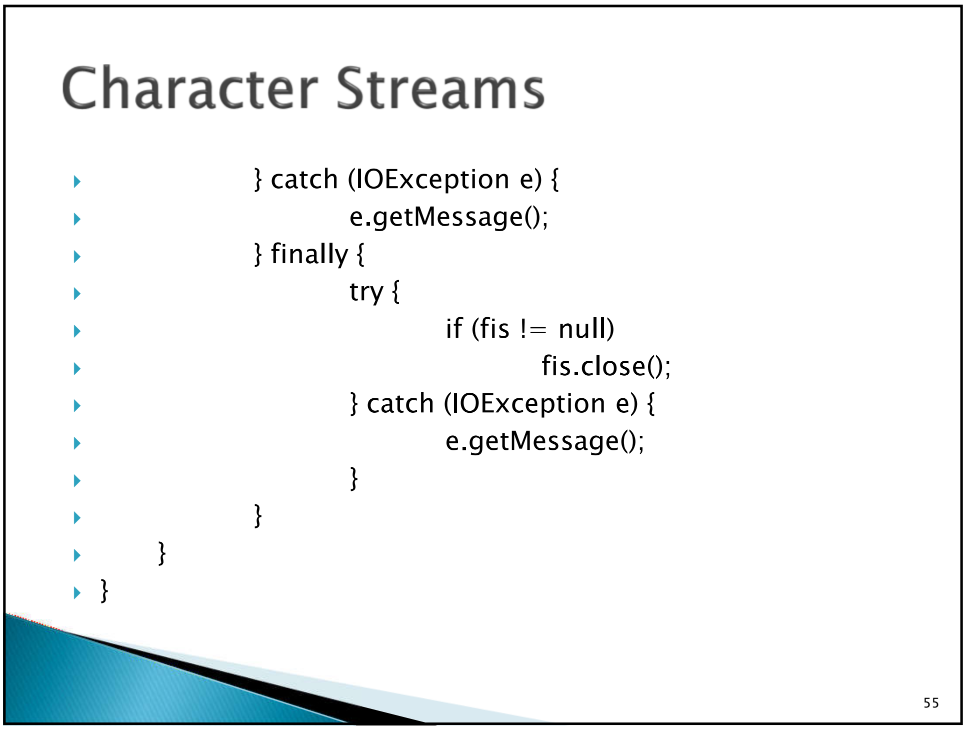
```
                } catch (IOException e) {
                        e.getMessage();
                } finally {
                        try {
                                if (fos != null)
                                        fos.close();
                        } catch (IOException e) {
                                e.getMessage();
                        }
                }
        }
}
```



Character Streams

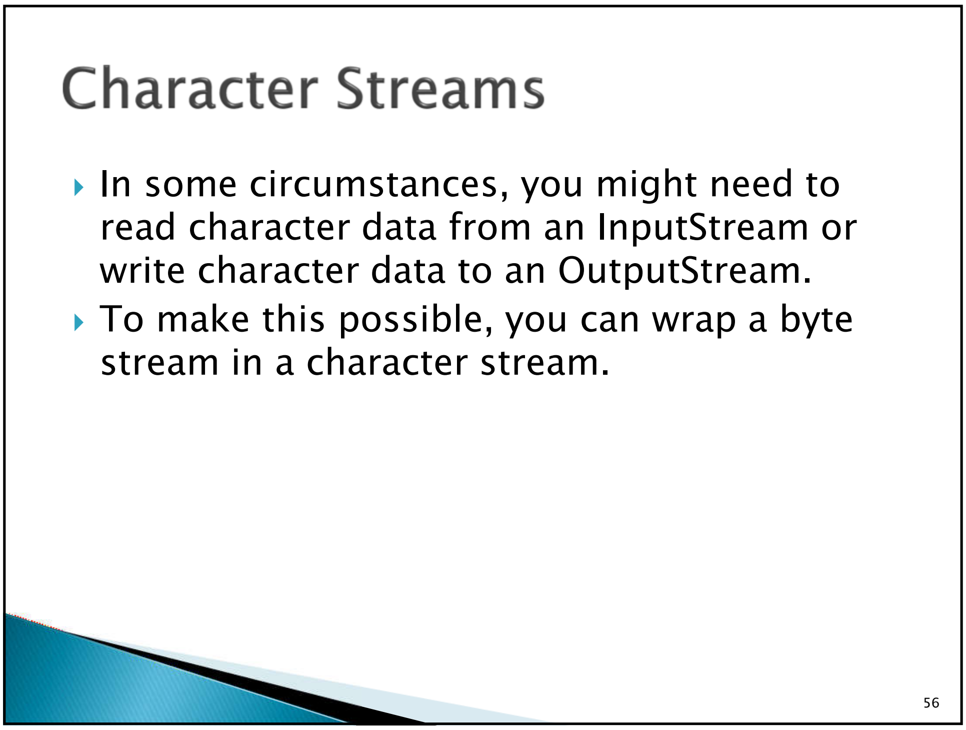
- Because a character stream’s read() method returns an integer, you must cast this to a character before displaying it, storing it in an array, or using it to form a string.

```
import java.io.FileReader;
import java.io.IOException;
 public class FileReaderTest {
        public static void main(String[] args) {
                String filePath = "cars.txt";
                FileReader fis = null;
                int ch;
                try {
                        fis = new FileReader(filePath);
                        ch = fis.read();
                        while (ch != -1) {
                                System.out.print((char) ch);
                                ch = fis.read();
                        }
```



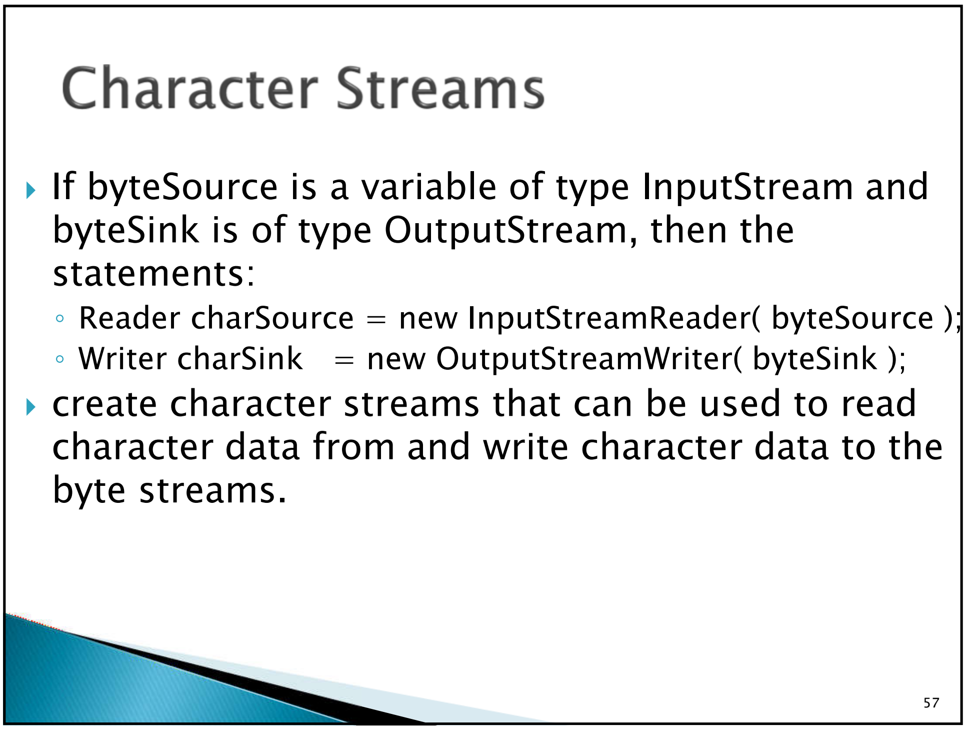
Character Streams

```
            } catch (IOException e) {
                    e.getMessage();
            } finally {
                    try {
                            if (fis != null)
                                    fis.close();
                    } catch (IOException e) {
                            e.getMessage();
                    }
            }
    }
}
```



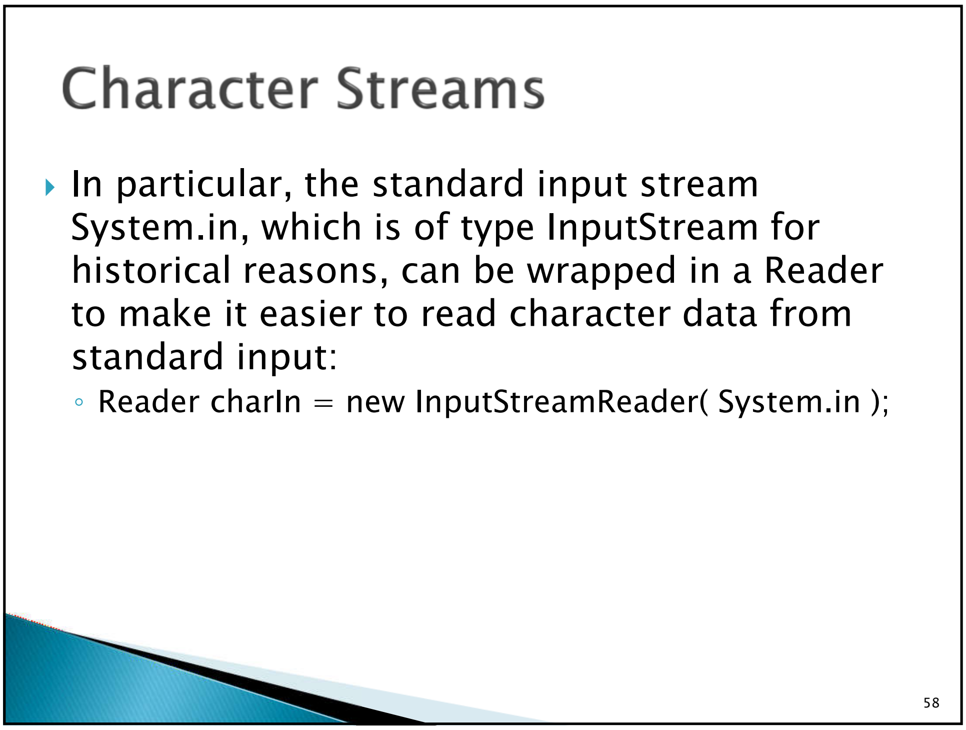
Character Streams

- In some circumstances, you might need to read character data from an InputStream or write character data to an OutputStream.
- To make this possible, you can wrap a byte stream in a character stream.



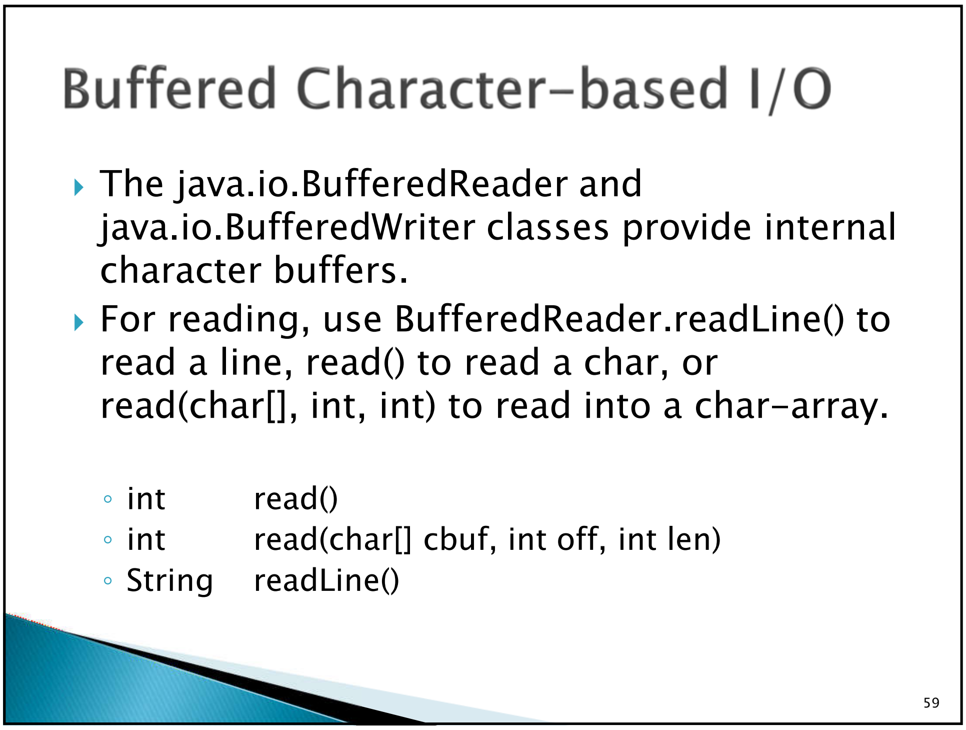
Character Streams

- If byteSource is a variable of type InputStream and byteSink is of type OutputStream, then the statements:
  - Reader charSource = new InputStreamReader(byteSource);
  - Writer charSink   = new OutputStreamWriter(byteSink);
- create character streams that can be used to read character data from and write character data to the byte streams.



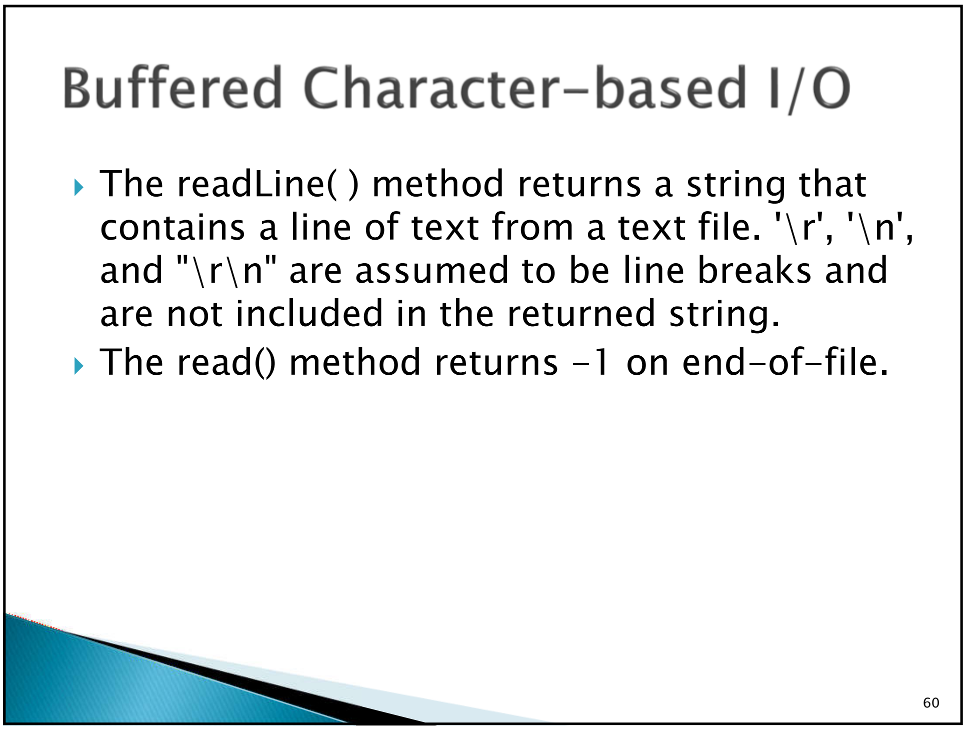
Character Streams

- In particular, the standard input stream System.in, which is of type InputStream for historical reasons, can be wrapped in a Reader to make it easier to read character data from standard input:
  - Reader charIn = new InputStreamReader(System.in);



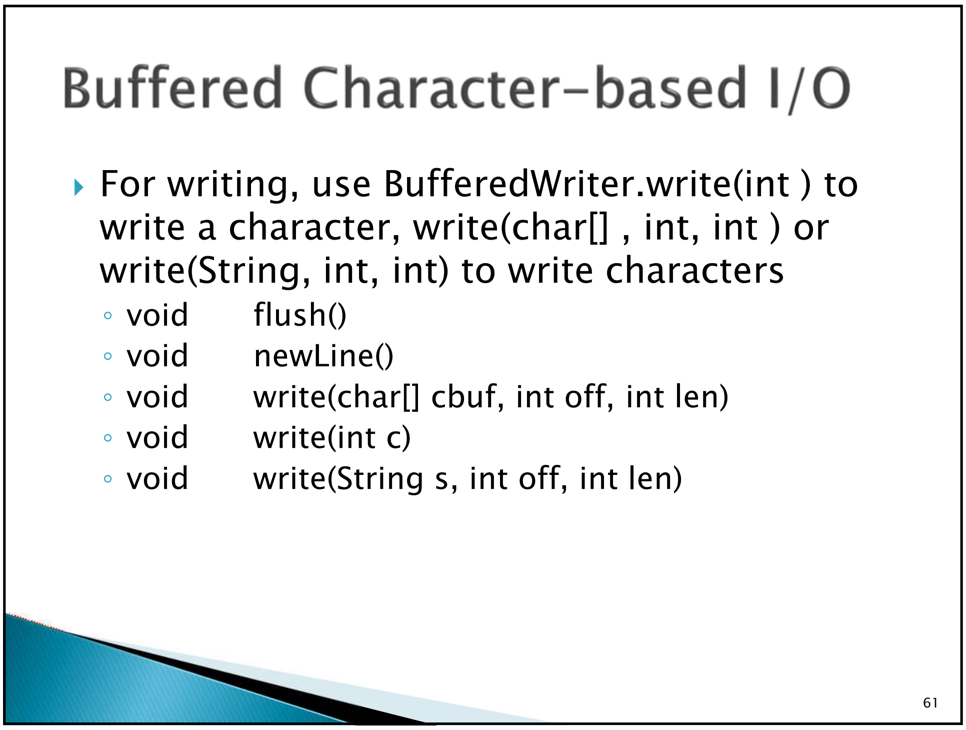
Buffered Character-based I/O

- The java.io.BufferedReader and java.io.BufferedWriter classes provide internal character buffers.
- For reading, use BufferedReader.readLine() to read a line, read() to read a char, or read(char[], int, int) to read into a char-array.
    - int   read()
    - int   read(char[] cbuf, int off, int len)
    - String   readLine()



Buffered Character-based I/O

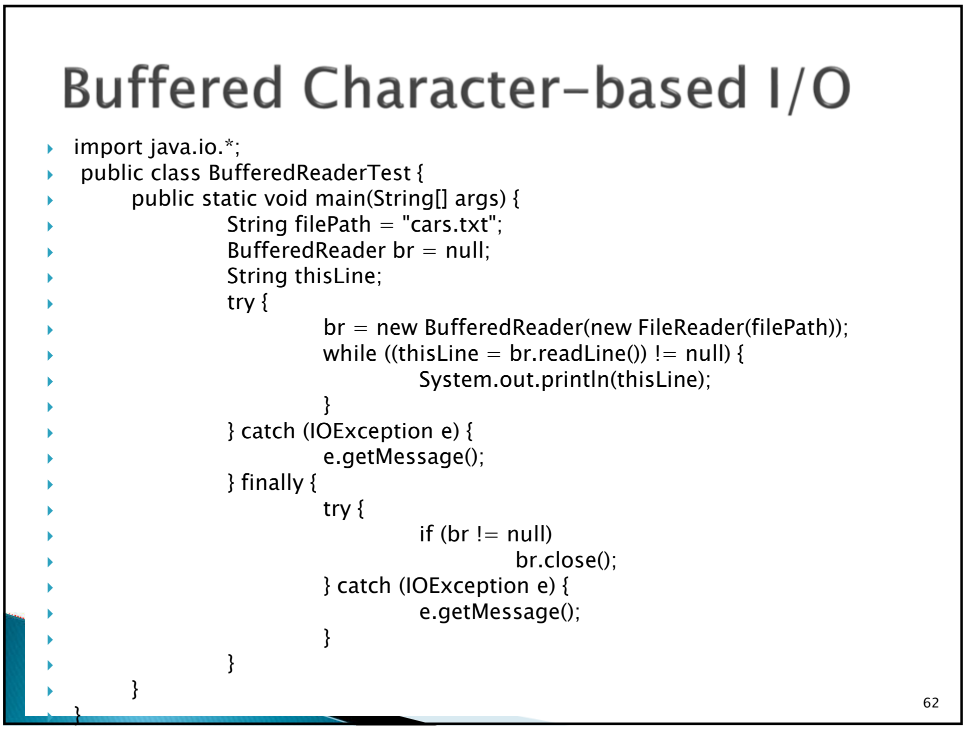
- The readLine( ) method returns a string that contains a line of text from a text file. '\r', '\n', and "\r\n" are assumed to be line breaks and are not included in the returned string.
- The read() method returns -1 on end-of-file.



Buffered Character-based I/O

- For writing, use BufferedWriter.write(int) to write a character, write(char[] , int, int) or write(String, int, int) to write characters
  - void flush()
  - void newLine()
  - void write(char[] cbuf, int off, int len)
  - void write(int c)
  - void write(String s, int off, int len)

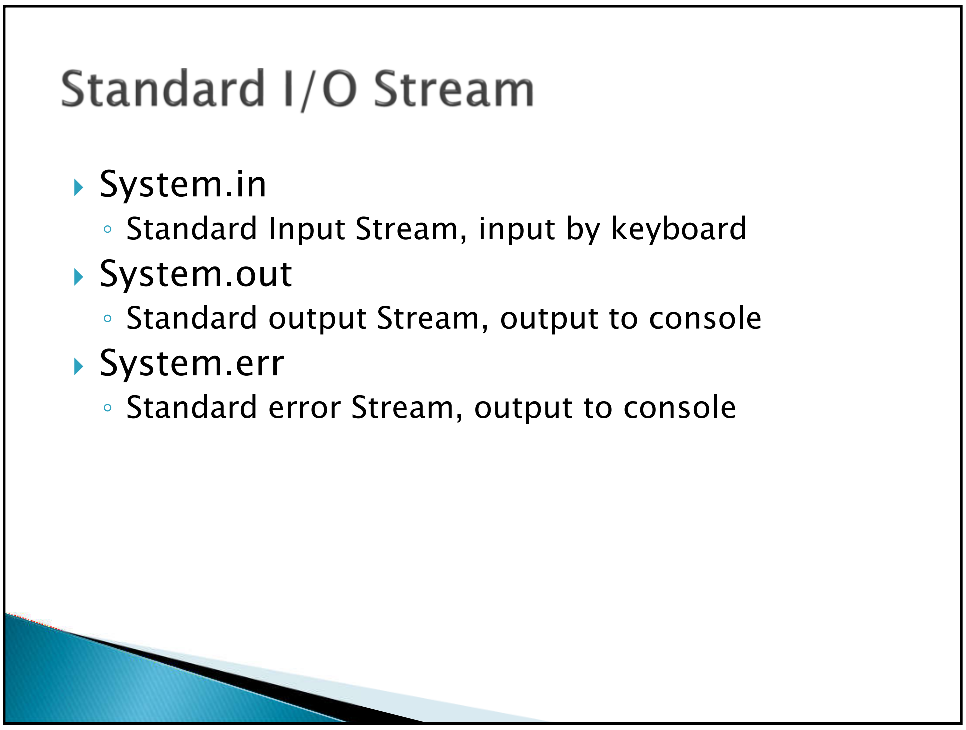
61



Buffered Character-based I/O

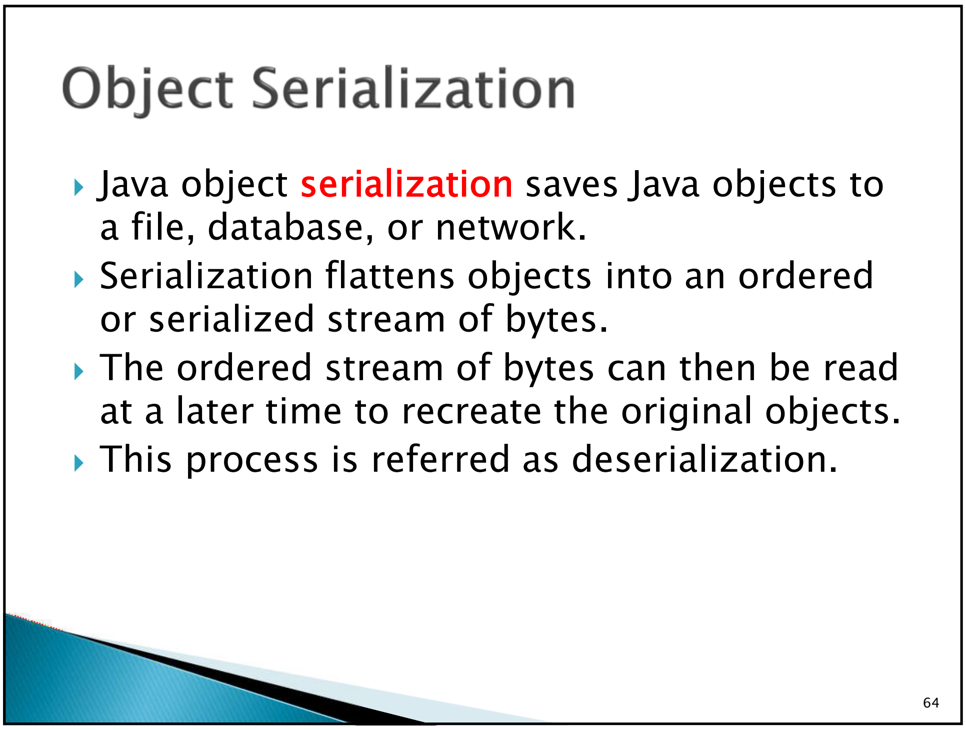
```
import java.io.*;
 public class BufferedReaderTest {
        public static void main(String[] args) {
                String filePath = "cars.txt";
                BufferedReader br = null;
                String thisLine;
                try {
                        br = new BufferedReader(new FileReader(filePath));
                        while ((thisLine = br.readLine()) != null) {
                                System.out.println(thisLine);
                        }
                } catch (IOException e) {
                        e.getMessage();
                } finally {
                        try {
                                if (br != null)
                                        br.close();
                        } catch (IOException e) {
                                e.getMessage();
                        }
                }
        }
}
```

62



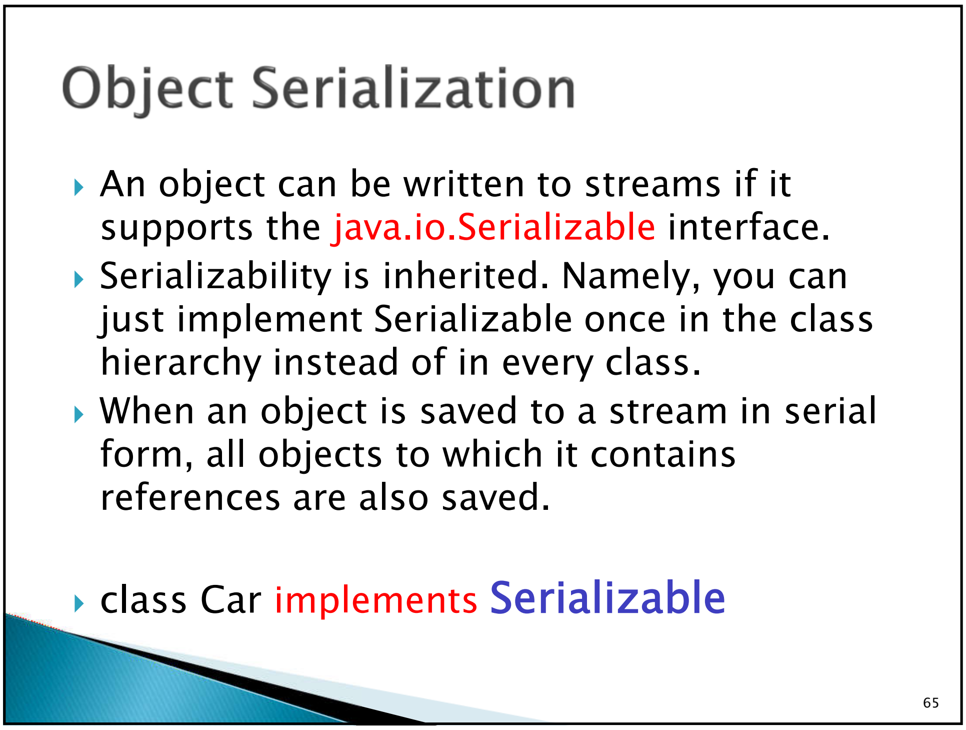
Standard I/O Stream

- System.in
  - Standard Input Stream, input by keyboard
- System.out
  - Standard output Stream, output to console
- System.err
  - Standard error Stream, output to console



Object Serialization

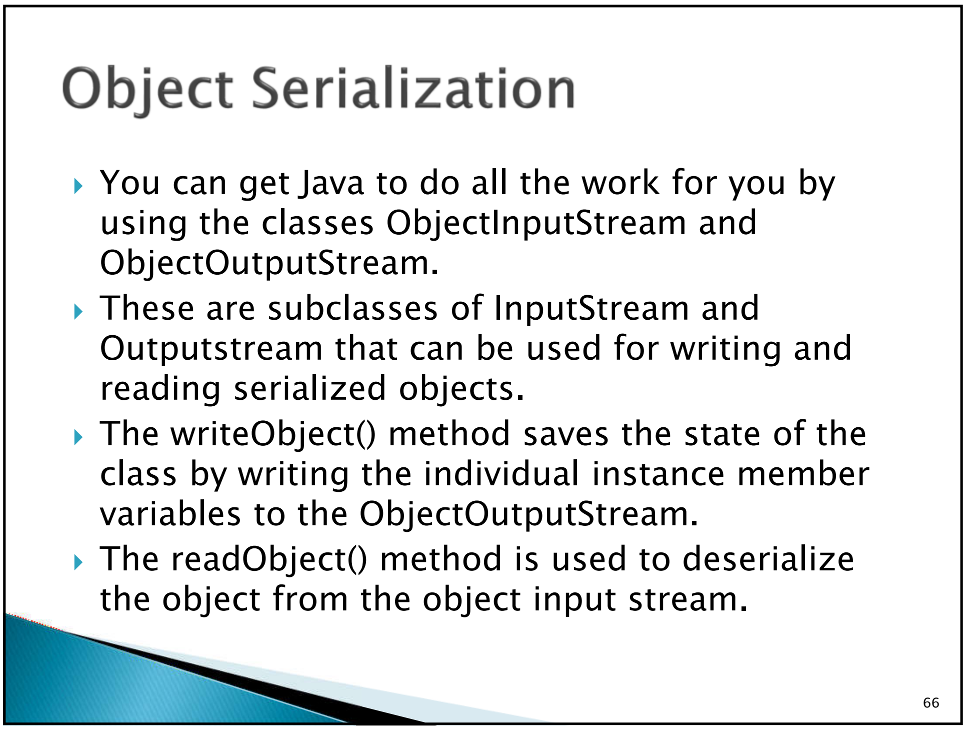
- Java object serialization saves Java objects to a file, database, or network.
- Serialization flattens objects into an ordered or serialized stream of bytes.
- The ordered stream of bytes can then be read at a later time to recreate the original objects.
- This process is referred as deserialization.



Object Serialization

- An object can be written to streams if it supports the java.io.Serializable interface.
- Serializability is inherited. Namely, you can just implement Serializable once in the class hierarchy instead of in every class.
- When an object is saved to a stream in serial form, all objects to which it contains references are also saved.

- class Car implements Serializable



Object Serialization

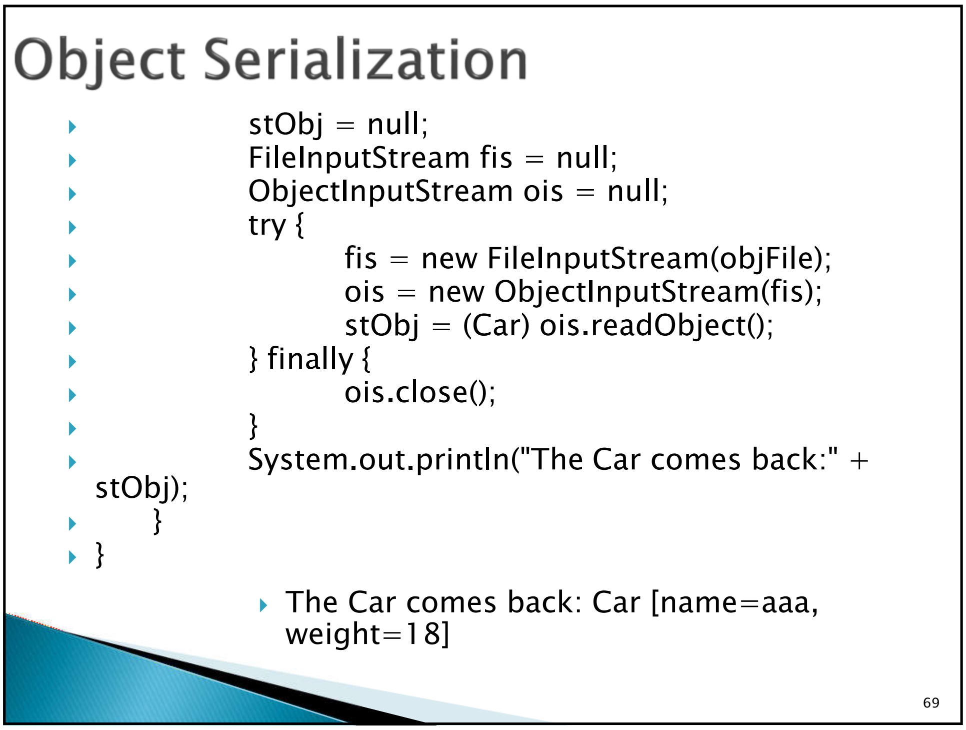
- You can get Java to do all the work for you by using the classes ObjectInputStream and ObjectOutputStream.
- These are subclasses of InputStream and Outputstream that can be used for writing and reading serialized objects.
- The writeObject() method saves the state of the class by writing the individual instance member variables to the ObjectOutputStream.
- The readObject() method is used to deserialize the object from the object input stream.

Object Serialization

- When several objects contain references to the same object, Java automatically ensures that only one copy of that object is serialized.
- Each object is assigned an internal serial number; successive attempts to save that object store only that number.

Object Serialization

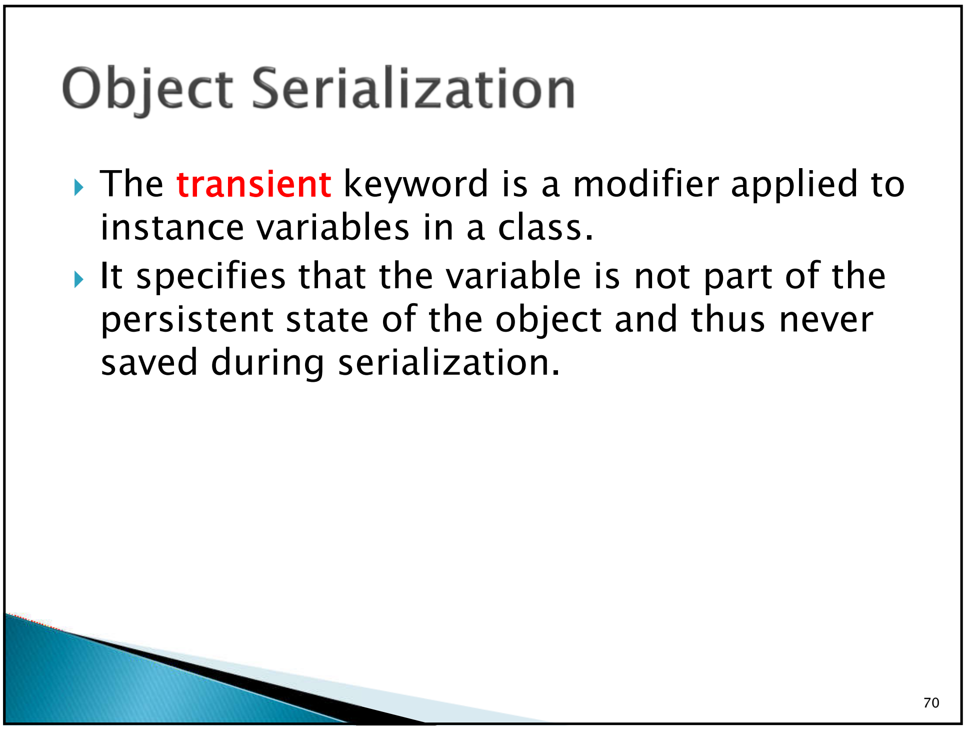
```
import java.io.*;

public class SerializationTest {
        public static void main(String[] args) throws IOException,
ClassNotFoundException {
                Car stObj = new Car("aaa", 18);
                File objFile = new File("cars.ser");

                FileOutputStream fos = null;
                ObjectOutputStream oos = null;
                try {
                        fos = new FileOutputStream(objFile);
                        oos = new ObjectOutputStream(fos);
                        oos.writeObject(stObj);
                } finally {
                        oos.close();
                }
```



Object Serialization

```
                stObj = null;
                FileInputStream fis = null;
                ObjectInputStream ois = null;
                try {
                        fis = new FileInputStream(objFile);
                        ois = new ObjectInputStream(fis);
                        stObj = (Car) ois.readObject();
                } finally {
                        ois.close();
                }
                System.out.println("The Car comes back:" +
stObj);
        }
}
```

- The Car comes back: Car [name=aaa, weight=18]



Object Serialization

- The transient keyword is a modifier applied to instance variables in a class.
- It specifies that the variable is not part of the persistent state of the object and thus never saved during serialization.