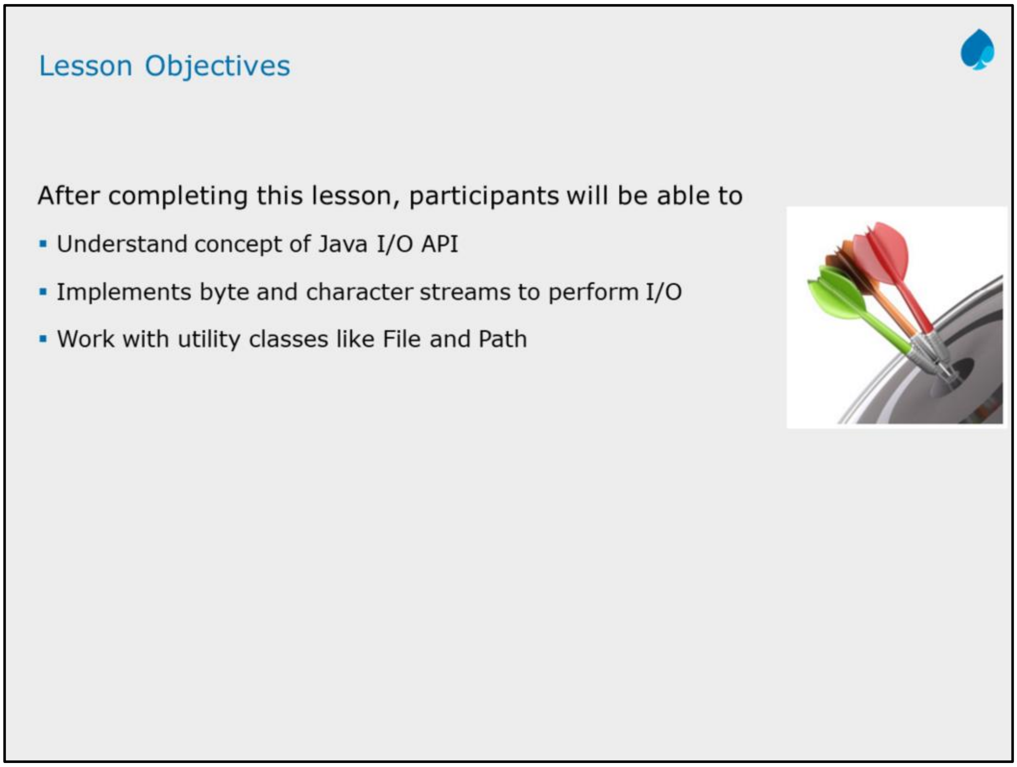
Core Java 8

Lesson 17: File IO



Lesson Objectives

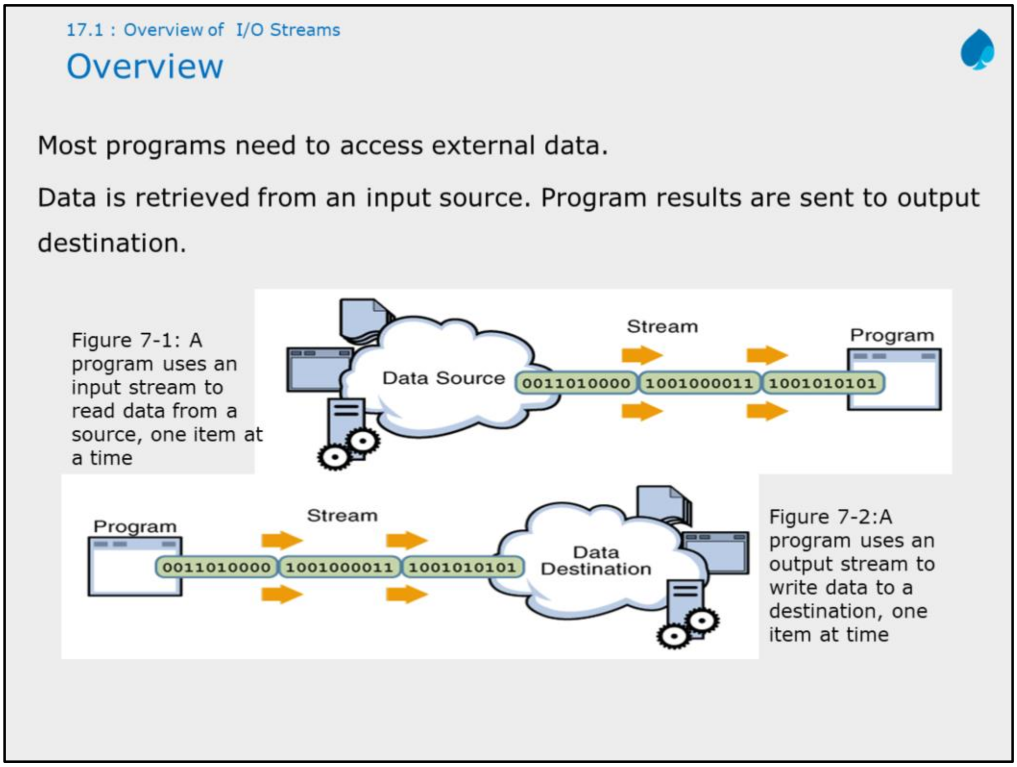
After completing this lesson, participants will be able to

- Understand concept of Java I/O API
- Implements byte and character streams to perform I/O
- Work with utility classes like File and Path

This lesson covers the Java platform classes used for basic I/O. It focuses primarily on I/O Streams, a powerful concept that greatly simplifies I/O operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again. Most of the classes covered are in the java.io package.

Lesson outline:

17.1: Overview of I/O Streams
17.2: Types of Streams
17.3: The Byte-stream I/O hierarchy
17.4: Character Stream Hierarchy
17.5: Buffered Stream
17.6: The File class
17.7: Exploring NIO
17.8: Object Stream
17.9: Best Practices



17.1 : Overview of I/O Streams

Overview

Most programs need to access external data.

Data is retrieved from an input source. Program results are sent to output destination.

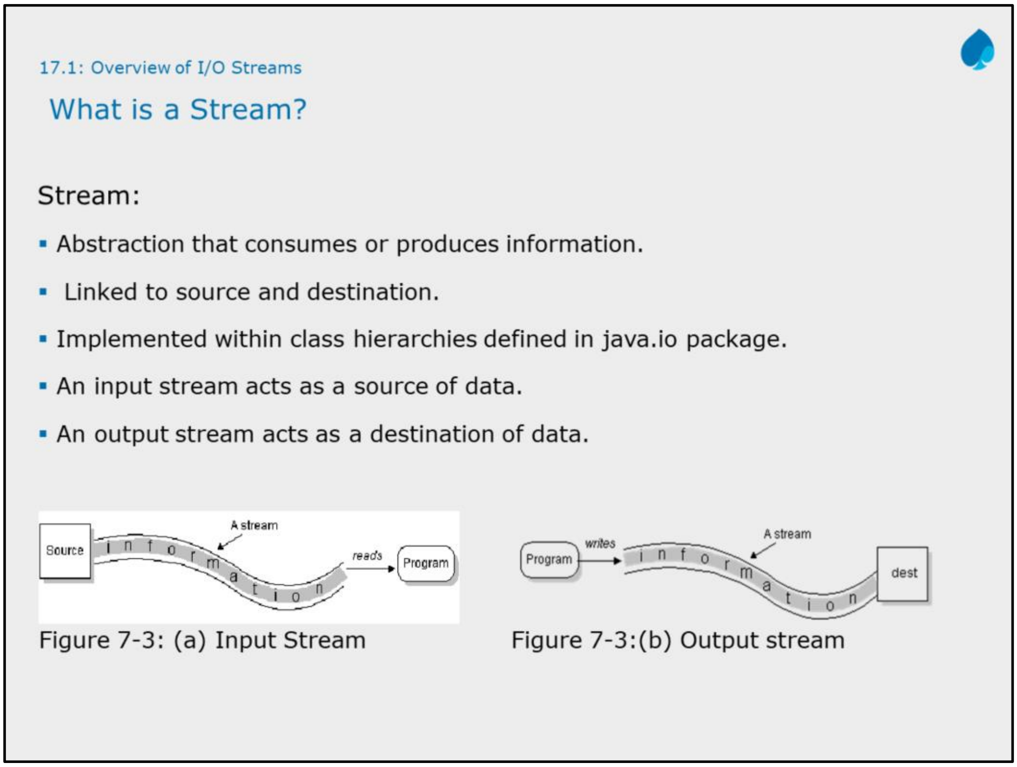
Figure 7-1: A program uses an input stream to read data from a source, one item at a time

Figure 7-2:A program uses an output stream to write data to a destination, one item at time

Most programs need to use data. To read some data, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. To write some data, a program opens a stream to a data source and writes to it in a serial fashion.

Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same.

The java.io package provides an extensive library of classes dealing with input and output. Each class has a variety of member variables & methods. java.io is layered. i.e. it does not attempt to put too much capability into one class. Instead, you can get the features you want, by layering (chaining streams) one class over another.



17.1: Overview of I/O Streams

What is a Stream?

Stream:

- Abstraction that consumes or produces information.
- Linked to source and destination.
- Implemented within class hierarchies defined in java.io package.
- An input stream acts as a source of data.
- An output stream acts as a destination of data.

Figure 7-3: (a) Input Stream

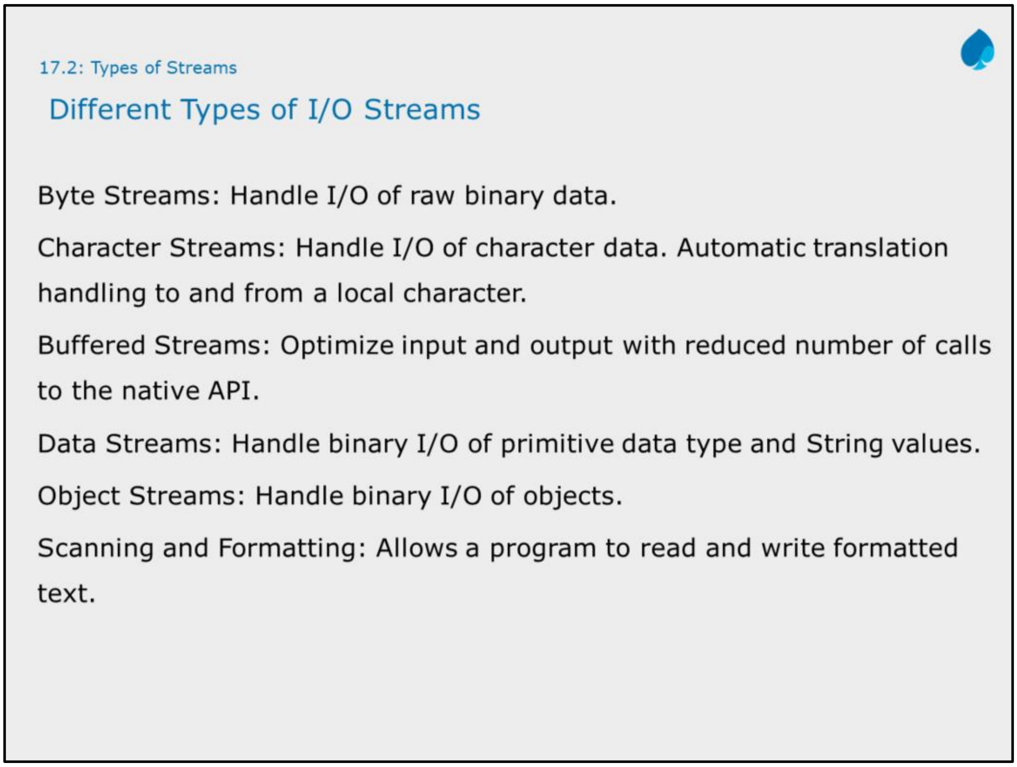
Figure 7-3:(b) Output stream

The source and destination of data for a java program can be anything – a network connection, local files, memory buffer etc. All are handled in the same way using streams. Streams implement sequential access of data. Java implements streams within class hierarchies defined in the java.io package.

An input stream is an object that an application can use, to read a sequence of data. An output stream is an object that an application can use to write a sequence of data.

An input stream acts as a source of data, and an output stream acts as a destination of data.

Some streams simply pass on data; others manipulate and transform the data in useful ways.



17.2: Types of Streams

Different Types of I/O Streams

Byte Streams: Handle I/O of raw binary data.

Character Streams: Handle I/O of character data. Automatic translation handling to and from a local character.

Buffered Streams: Optimize input and output with reduced number of calls to the native API.

Data Streams: Handle binary I/O of primitive data type and String values.

Object Streams: Handle binary I/O of objects.

Scanning and Formatting: Allows a program to read and write formatted text.

There are different types of I/O (Input/Output) Streams:

Byte Streams: They provide a convenient means for handling input and output of bytes. Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes descend from InputStream and OutputStream class.

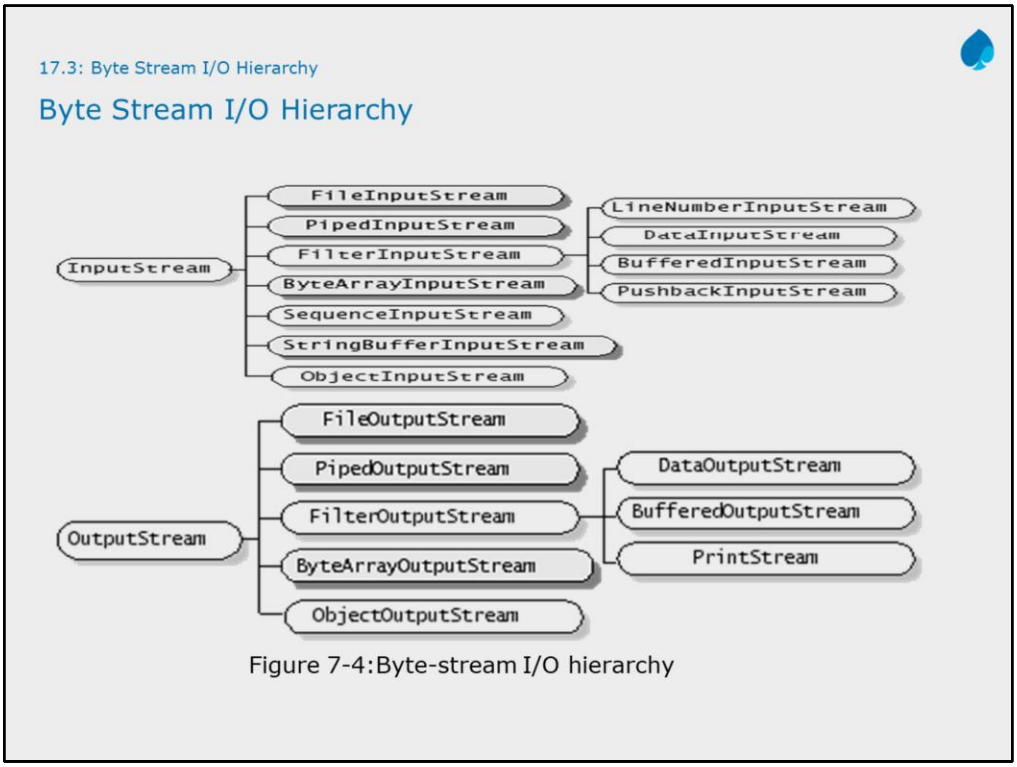
Character streams: They provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

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

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.

Object Streams: Just as data streams support I/O of primitive data types, object streams support I/O of objects.

Scanning and Formatting: It allows a program to read and write formatted text.



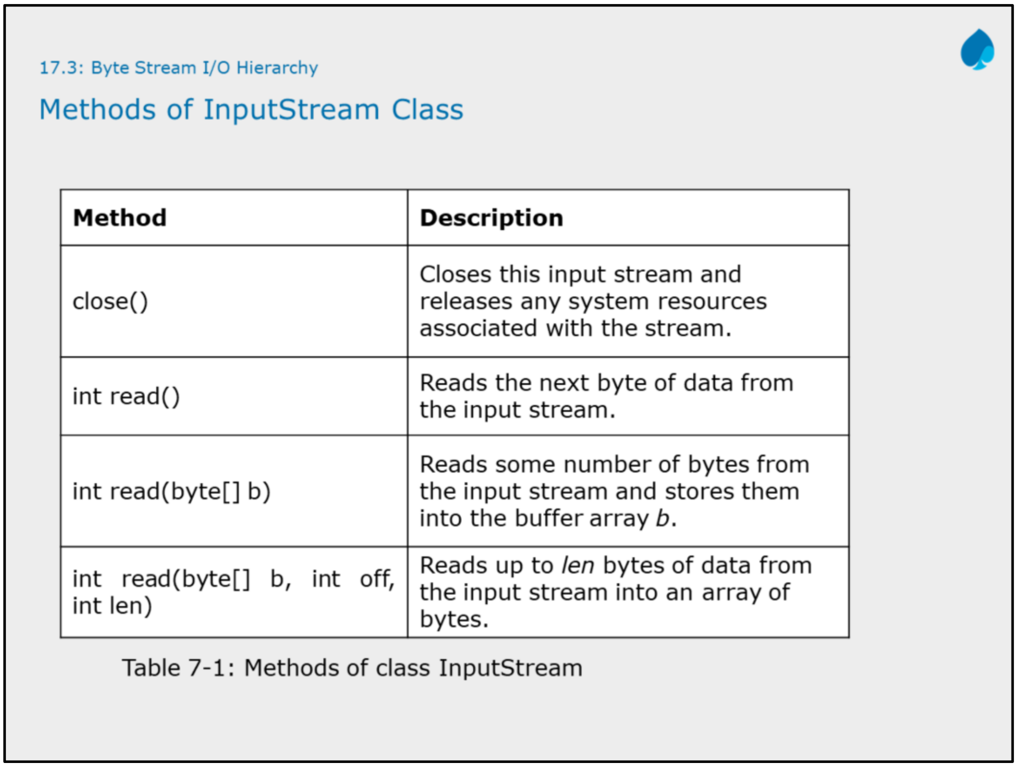
17.3: Byte Stream I/O Hierarchy

Byte Stream I/O Hierarchy

Figure 7-4:Byte-stream I/O hierarchy

At the top of the hierarchy are two abstract classes: InputStream and OutputStream.

Each of these abstract classes serves as base class for all other concretely implemented I/O classes. Each of the abstract classes defines several key methods that the other stream classes implement.



17.3: Byte Stream I/O Hierarchy

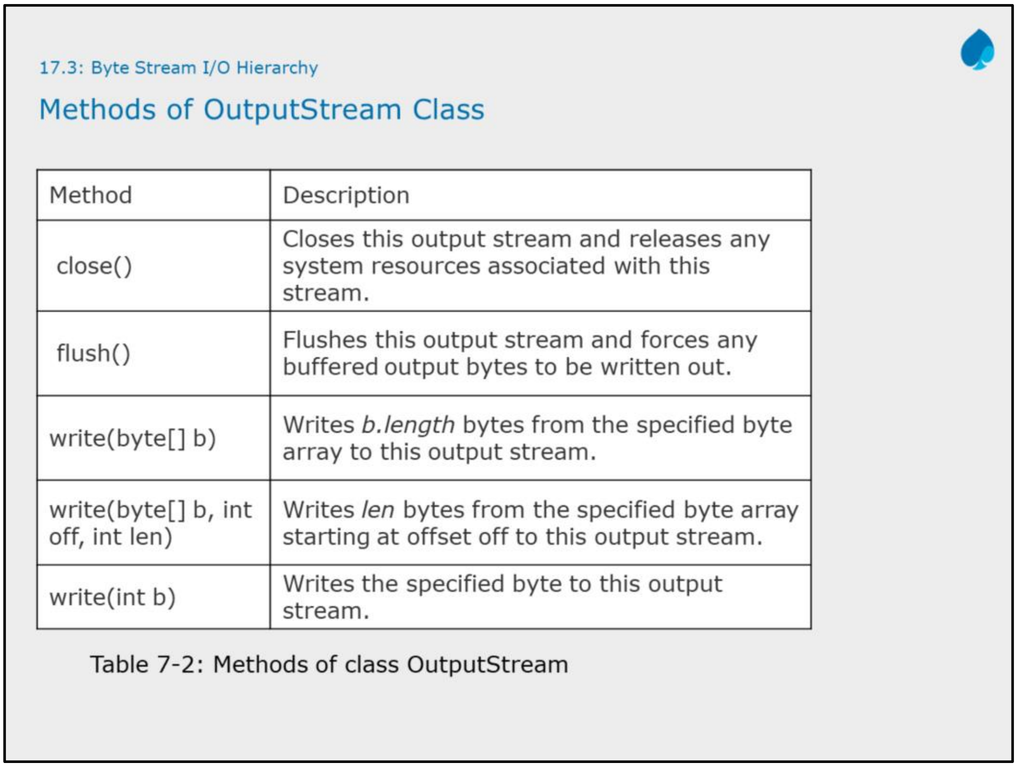
Methods of InputStream Class

Method	Description
close()	Closes this input stream and releases any system resources associated with the stream.
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.

Table 7-1: Methods of class InputStream

All byte stream classes are descended from InputStream and OutputStream.

Note: Refer to Java documentation for more methods.



17.3: Byte Stream I/O Hierarchy

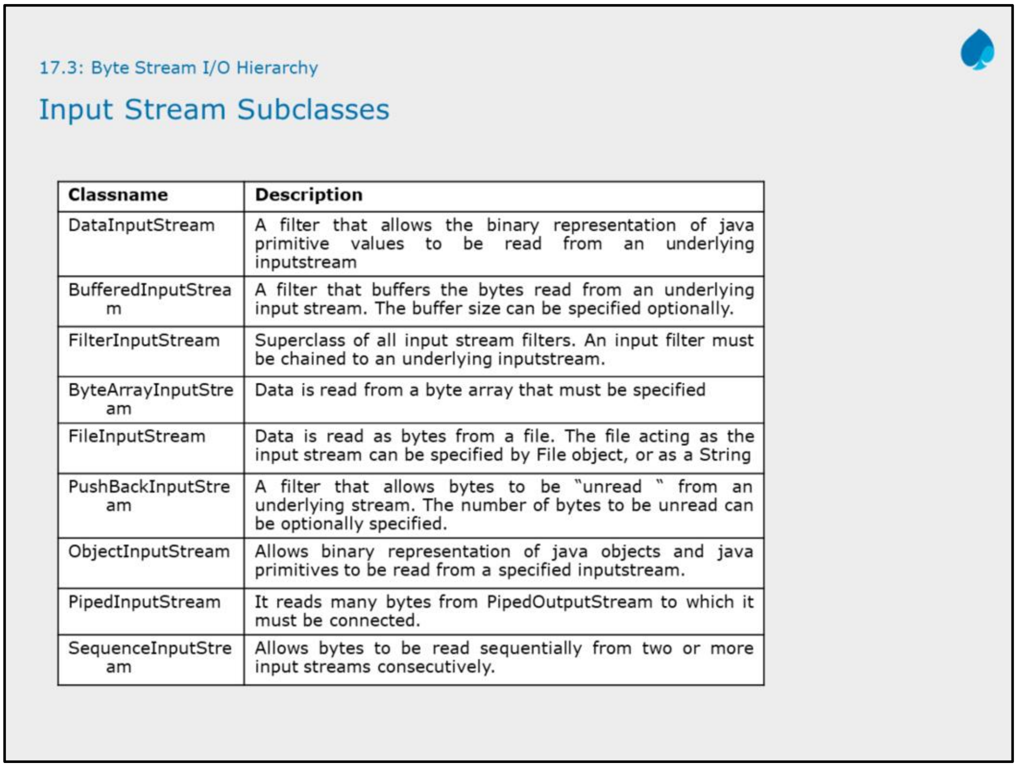
Methods of OutputStream Class

Method	Description
close()	Closes this output stream and releases any system resources associated with this stream.
flush()	Flushes this output stream and forces any buffered output bytes to be written out.
write(byte[] b)	Writes *b.length* bytes from the specified byte array to this output stream.
write(byte[] b, int off, int len)	Writes *len* bytes from the specified byte array starting at offset off to this output stream.
write(int b)	Writes the specified byte to this output stream.

Table 7-2: Methods of class OutputStream

Closing an output stream automatically flushes the stream, meaning that data in its internal buffer is written out. An output stream can also be manually flushed by calling the flush() method.

Note: Refer to Java documentation for more methods.



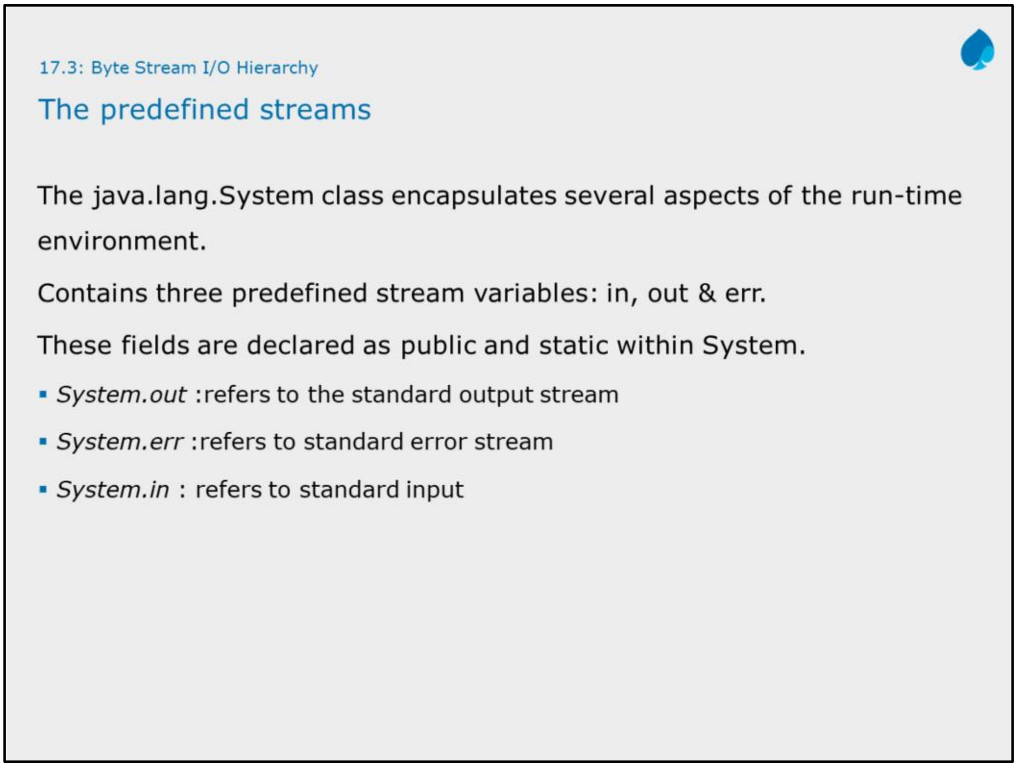
17.3: Byte Stream I/O Hierarchy

Input Stream Subclasses

Classname	Description
DataInputStream	A filter that allows the binary representation of java primitive values to be read from an underlying inputstream
BufferedInputStream	A filter that buffers the bytes read from an underlying input stream. The buffer size can be specified optionally.
FilterInputStream	Superclass of all input stream filters. An input filter must be chained to an underlying inputstream.
ByteArrayInputStream	Data is read from a byte array that must be specified
FileInputStream	Data is read as bytes from a file. The file acting as the input stream can be specified by File object, or as a String
PushBackInputStream	A filter that allows bytes to be “unread ” from an underlying stream. The number of bytes to be unread can be optionally specified.
ObjectInputStream	Allows binary representation of java objects and java primitives to be read from a specified inputstream.
PipedInputStream	It reads many bytes from PipedOutputStream to which it must be connected.
SequenceInputStream	Allows bytes to be read sequentially from two or more input streams consecutively.

The InputStream class allows several classes to derive from it. Some of these classes are described in the table above.

Note: All of the above mentioned classes have corresponding output stream classes except SequenceInputStream.



17.3: Byte Stream I/O Hierarchy

The predefined streams

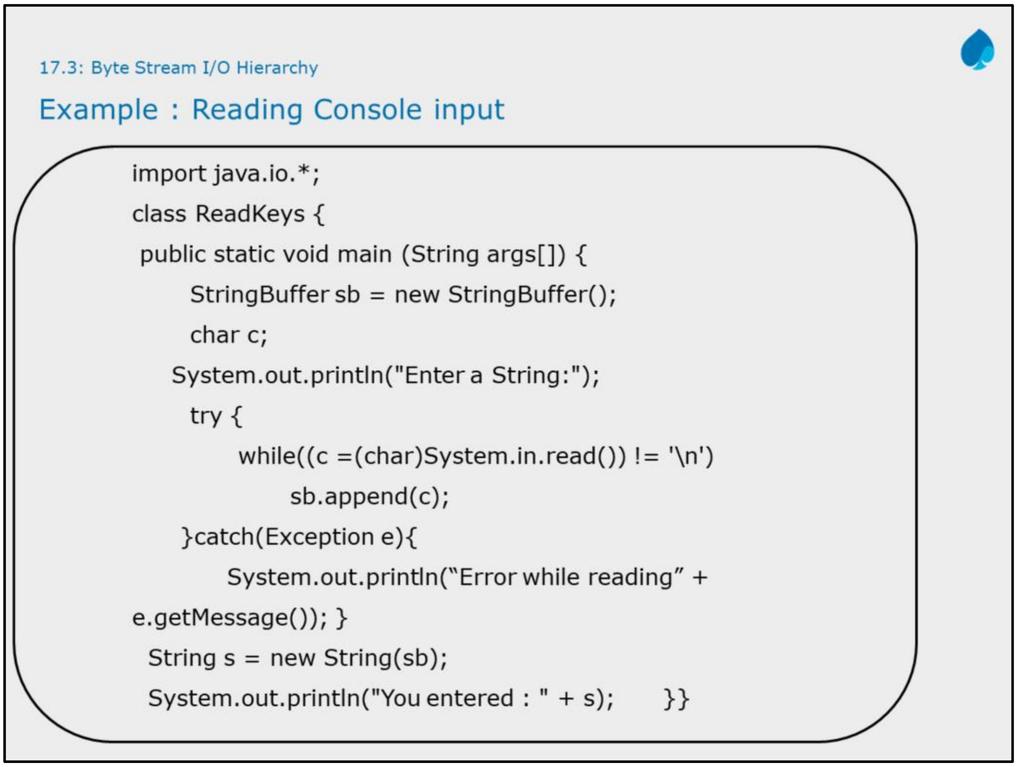
The java.lang.System class encapsulates several aspects of the run-time environment.

Contains three predefined stream variables: in, out & err.

These fields are declared as public and static within System.

- *System.out* :refers to the standard output stream
- *System.err* :refers to standard error stream
- *System.in* : refers to standard input

System.out refers to the standard output stream and System.err refers to standard error stream (Both, by default, the console). These are objects of type PrintStream class.
System.in refers to standard input (keyboard by default). This is an object of the InputStream class.

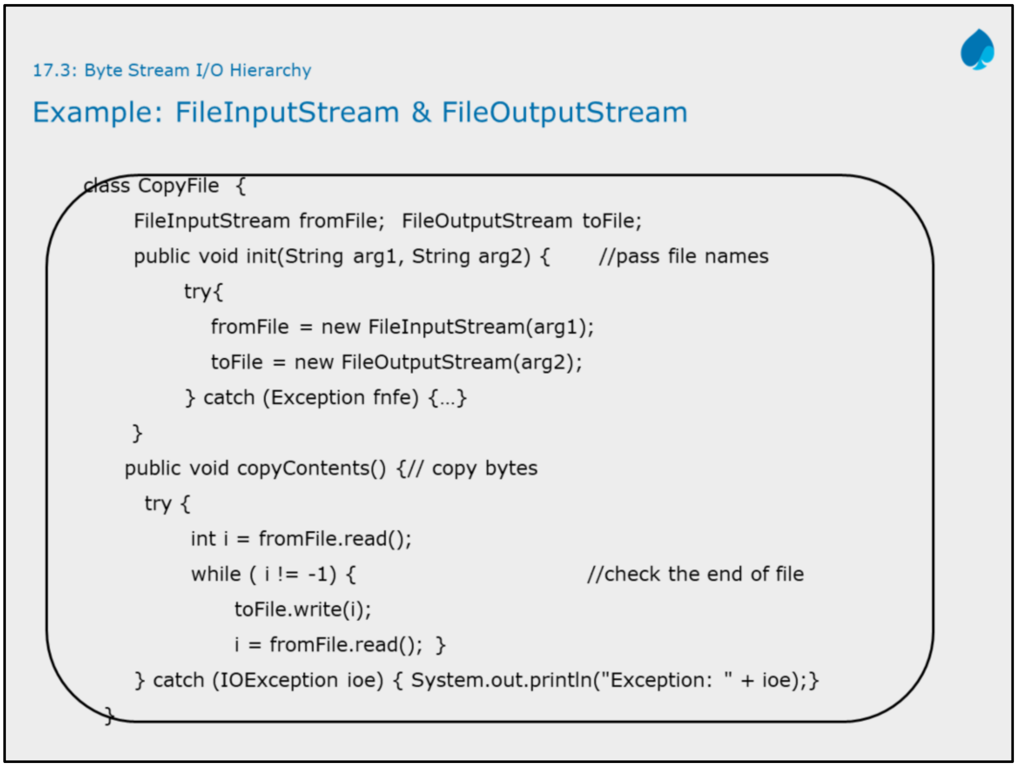


17.3: Byte Stream I/O Hierarchy

Example : Reading Console input

```
import java.io.*;
class ReadKeys {
 public static void main (String args[]) {
       StringBuffer sb = new StringBuffer();
       char c;
     System.out.println("Enter a String:");
       try {
             while((c =(char)System.in.read()) != '\n')
                   sb.append(c);
      }catch(Exception e){
            System.out.println(“Error while reading” +
e.getMessage()); }
  String s = new String(sb);
  System.out.println("You entered : " + s);      }}
```

In Java, console input is accomplished by reading from System.in. In the above example, read() methods reads a byte from the input stream (here, the keyboard), and returns an integer. Therefore, casting to char type need to be done.



17.3: Byte Stream I/O Hierarchy

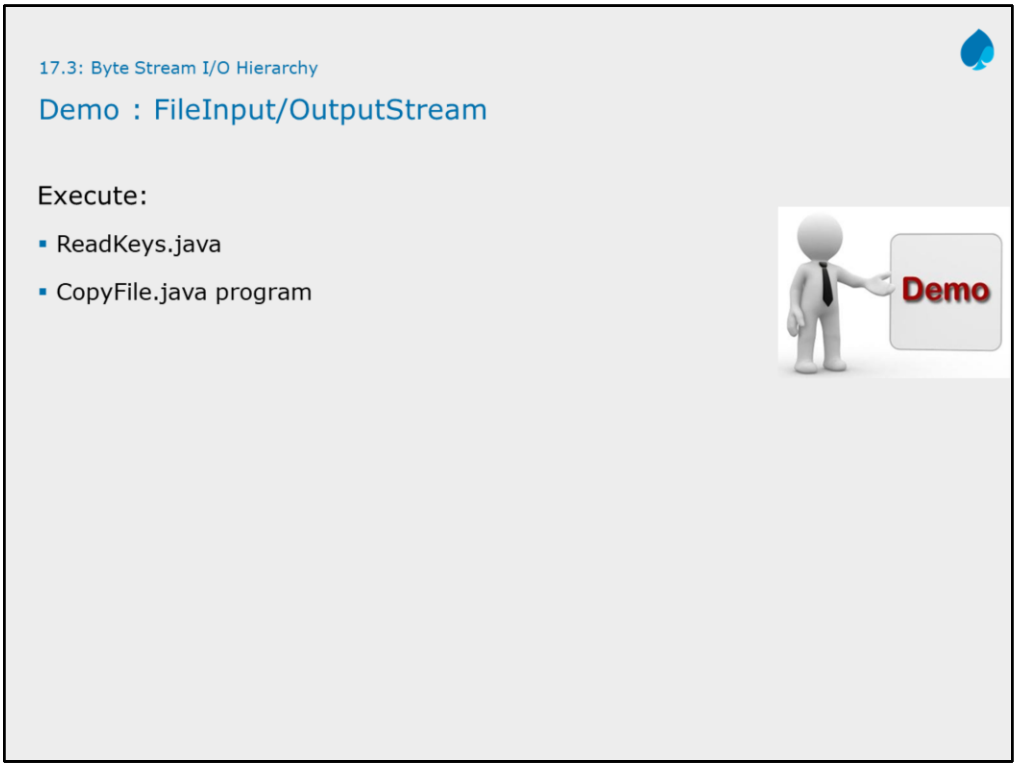
Example: FileInputStream & FileOutputStream

```
class CopyFile  {
    FileInputStream fromFile;  FileOutputStream toFile;
    public void init(String arg1, String arg2) {      //pass file names
        try{
          fromFile = new FileInputStream(arg1);
          toFile = new FileOutputStream(arg2);
        } catch (Exception fnfe) {…}
    }
   public void copyContents() {// copy bytes
     try {
         int i = fromFile.read();
         while ( i != -1) {                        //check the end of file
            toFile.write(i);
            i = fromFile.read();  }
    } catch (IOException ioe) { System.out.println("Exception:  " + ioe);}
 }
```

The remainder of the code follows:

```
public void closeFiles() {                    //close the file
   try{
          fromFile.close();
          toFile.close();
    } catch (IOException ioe){
          System.out.println("Exception: " + ioe);
          }
}
public static void main(String[] args){
         CopyFile c1 = new CopyFile();
         c1.init(args[0], args[1]);
         c1.copyContents();
         c1.closeFiles();
} }
```

The FileInputStream and FileOutputStream classes define byte input and output streams that are connected to files. Data can only be read or written as a sequence of bytes. The above example demonstrates the use of File InputStream and FileOutputStream.

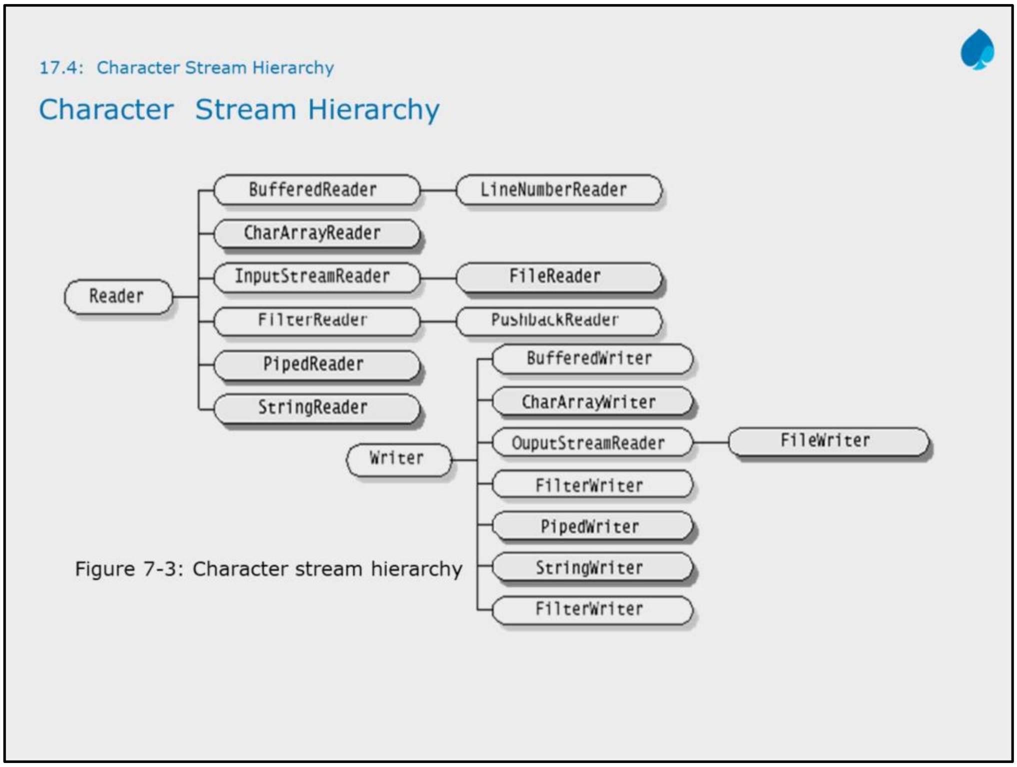


17.3: Byte Stream I/O Hierarchy

Demo : FileInput/OutputStream

Execute:

- ReadKeys.java
- CopyFile.java program



17.4: Character Stream Hierarchy

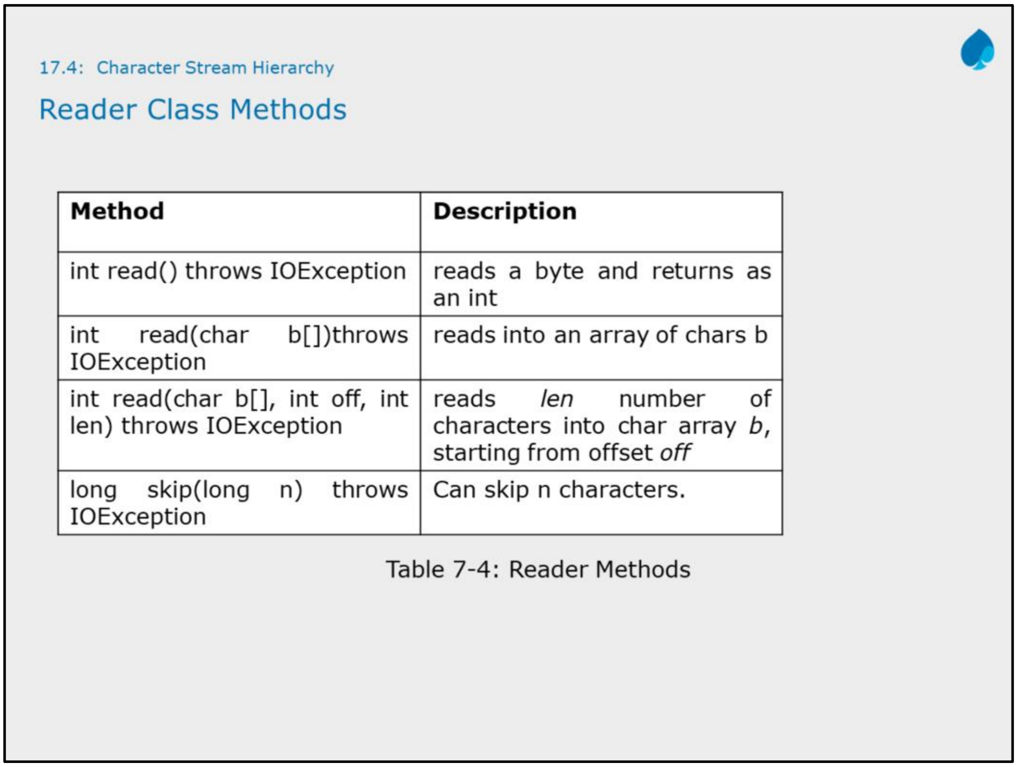
Character Stream Hierarchy

Figure 7-3: Character stream hierarchy

The byte stream classes support only 8-bit byte streams and doesn’t handle 16-bit Unicode characters well. A character encoding is a scheme for representing characters. Java represents characters internally in the 16-bit Unicode character encoding, but the host platform might use different character encoding.

The abstract classes Reader and Writer are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding.

A reader is an input character stream that reads a sequence of Unicode characters, and a writer is an output character stream that writes a sequence of Unicode characters.



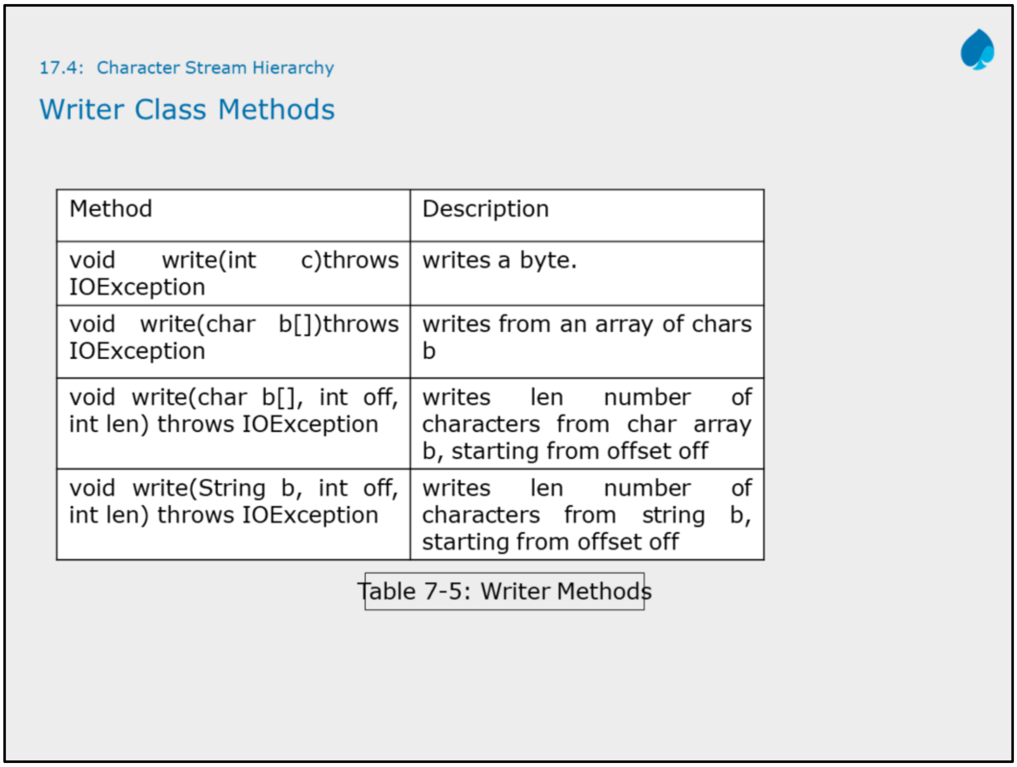
17.4: Character Stream Hierarchy

Reader Class Methods

Method	Description
int read() throws IOException	reads a byte and returns as an int
int read(char b[])throws IOException	reads into an array of chars b
int read(char b[], int off, int len) throws IOException	reads *len* number of characters into char array *b*, starting from offset *off*
long skip(long n) throws IOException	Can skip n characters.

Table 7-4: Reader Methods

Note: Refer to Java documentation for more methods.



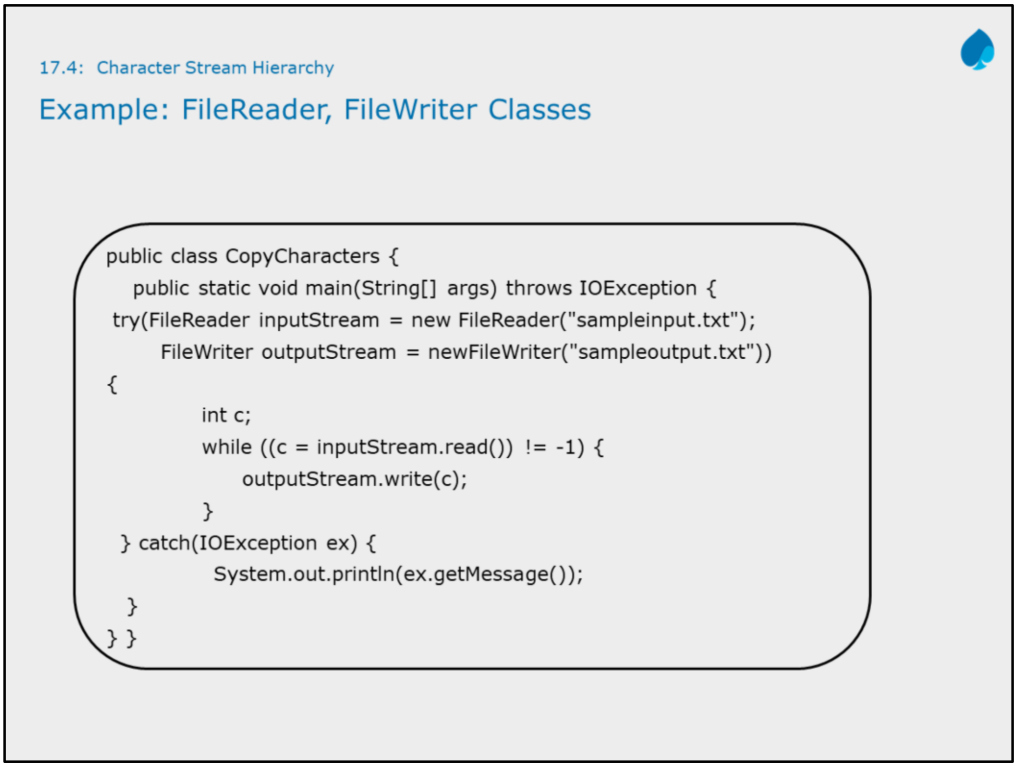
17.4: Character Stream Hierarchy

Writer Class Methods

Method	Description
void write(int c)throws IOException	writes a byte.
void write(char b[])throws IOException	writes from an array of chars b
void write(char b[], int off, int len) throws IOException	writes len number of characters from char array b, starting from offset off
void write(String b, int off, int len) throws IOException	writes len number of characters from string b, starting from offset off

Table 7-5: Writer Methods

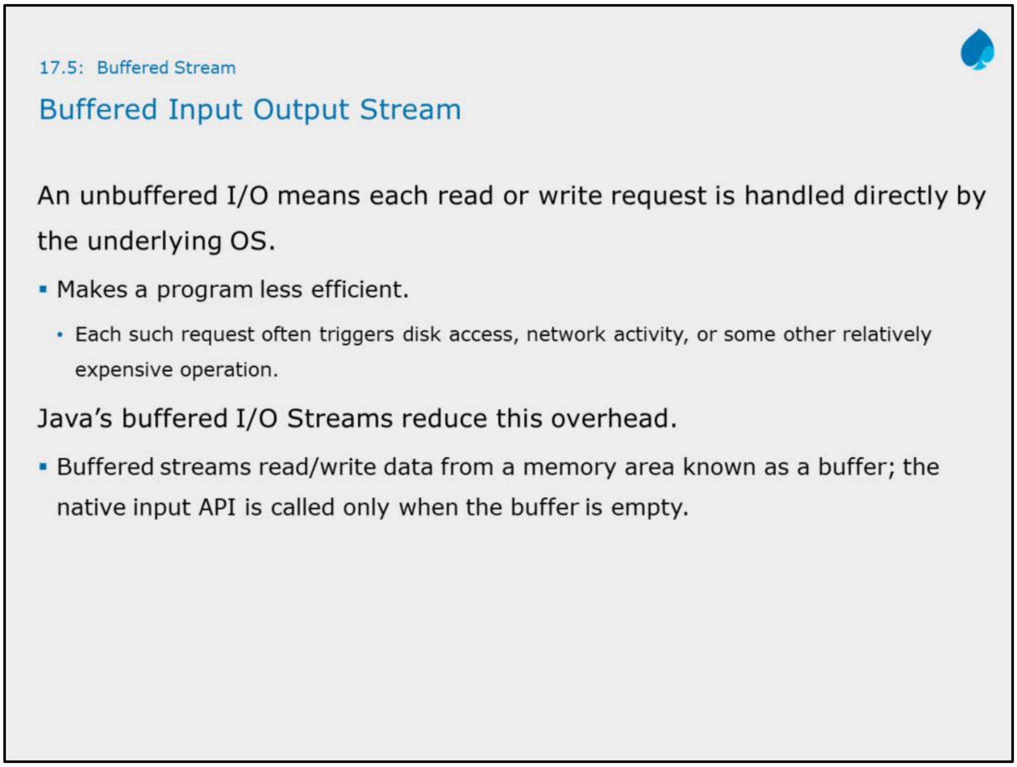
Note: Refer to Java documentation for more methods.



17.4: Character Stream Hierarchy

Example: FileReader, FileWriter Classes

```
public class CopyCharacters {
   public static void main(String[] args) throws IOException {
 try(FileReader inputStream = new FileReader("sampleinput.txt");
      FileWriter outputStream = newFileWriter("sampleoutput.txt"))
{
           int c;
           while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
           }
  } catch(IOException ex) {
            System.out.println(ex.getMessage());
  }
} }
```



17.5: Buffered Stream

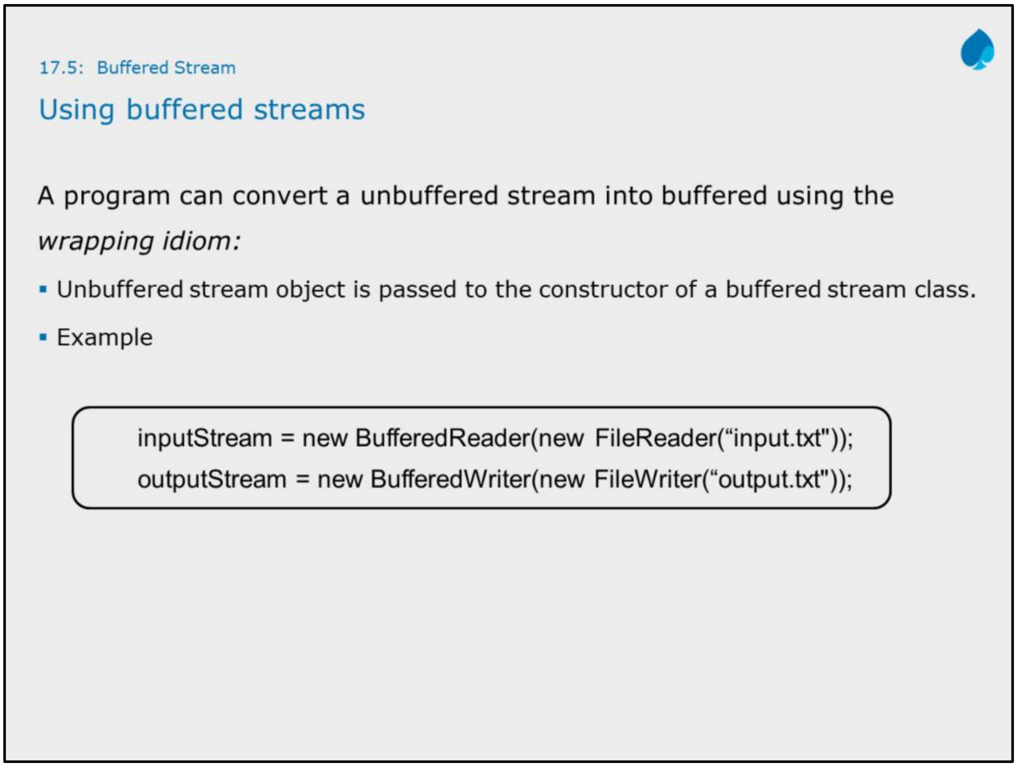
Buffered Input Output Stream

An unbuffered I/O means each read or write request is handled directly by the underlying OS.

- Makes a program less efficient.
  - Each such request often triggers disk access, network activity, or some other relatively expensive operation.

Java’s buffered I/O Streams reduce this overhead.

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



17.5: Buffered Stream

Using buffered streams

A program can convert a unbuffered stream into buffered using the *wrapping idiom:*

- Unbuffered stream object is passed to the constructor of a buffered stream class.
- Example

```
inputStream = new BufferedReader(new FileReader(“input.txt"));
outputStream = new BufferedWriter(new FileWriter(“output.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams.

BufferedInputStream and BufferedOutputStream - create buffered byte streams.

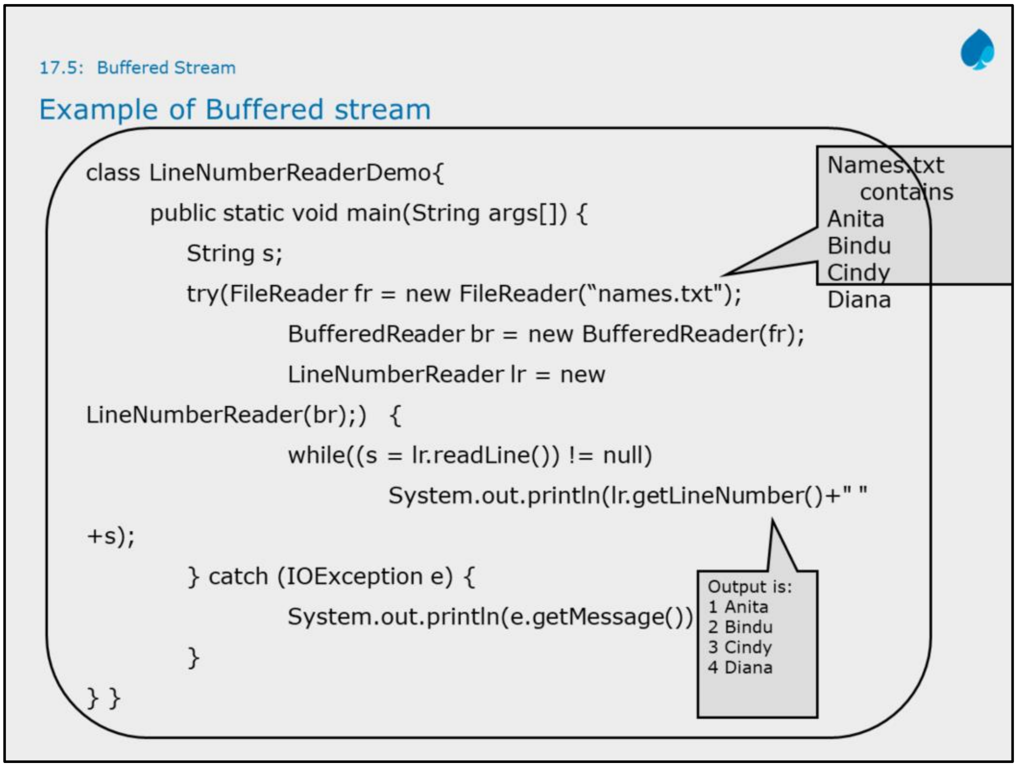
BufferedReader and BufferedWriter - create buffered character streams.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This 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. For example, 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.



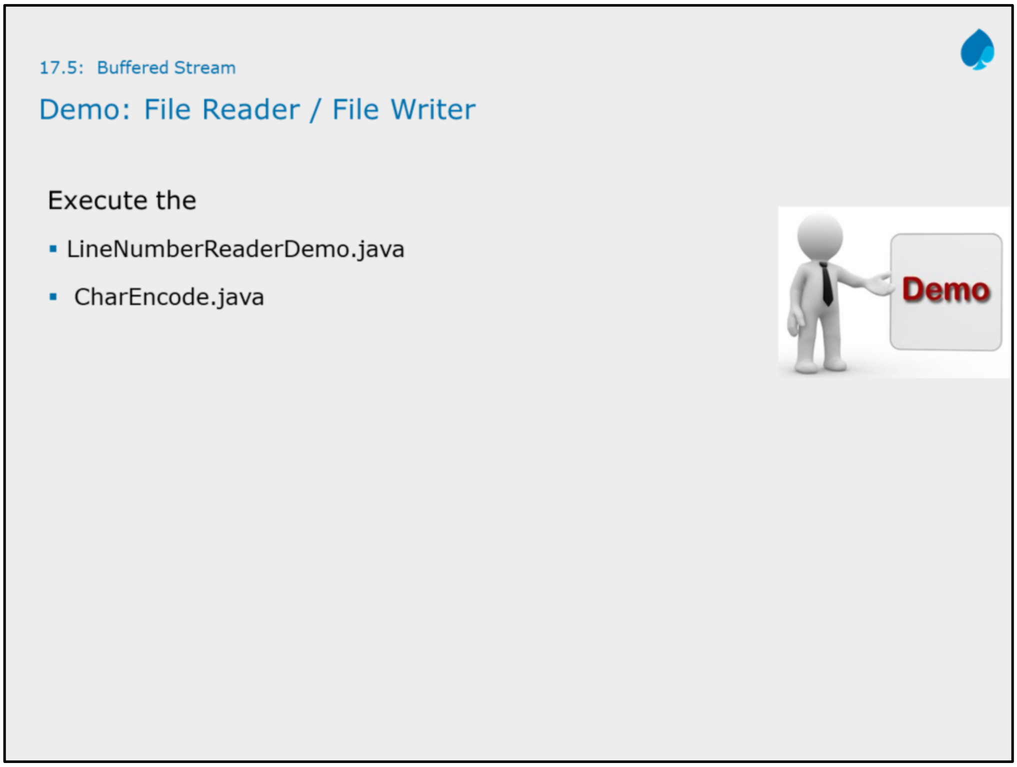
17.5: Buffered Stream

Example of Buffered stream

```
class LineNumberReaderDemo{
        public static void main(String args[]) {
            String s;
            try(FileReader fr = new FileReader(“names.txt");
                        BufferedReader br = new BufferedReader(fr);
                        LineNumberReader lr = new
LineNumberReader(br);)  {
                        while((s = lr.readLine()) != null)
                                    System.out.println(lr.getLineNumber()+" "
+s);
            } catch (IOException e) {
                        System.out.println(e.getMessage())
            }
} }
```

Names.txt contains
Anita
Bindu
Cindy
Diana

```
Output is:
1 Anita
2 Bindu
3 Cindy
4 Diana
```

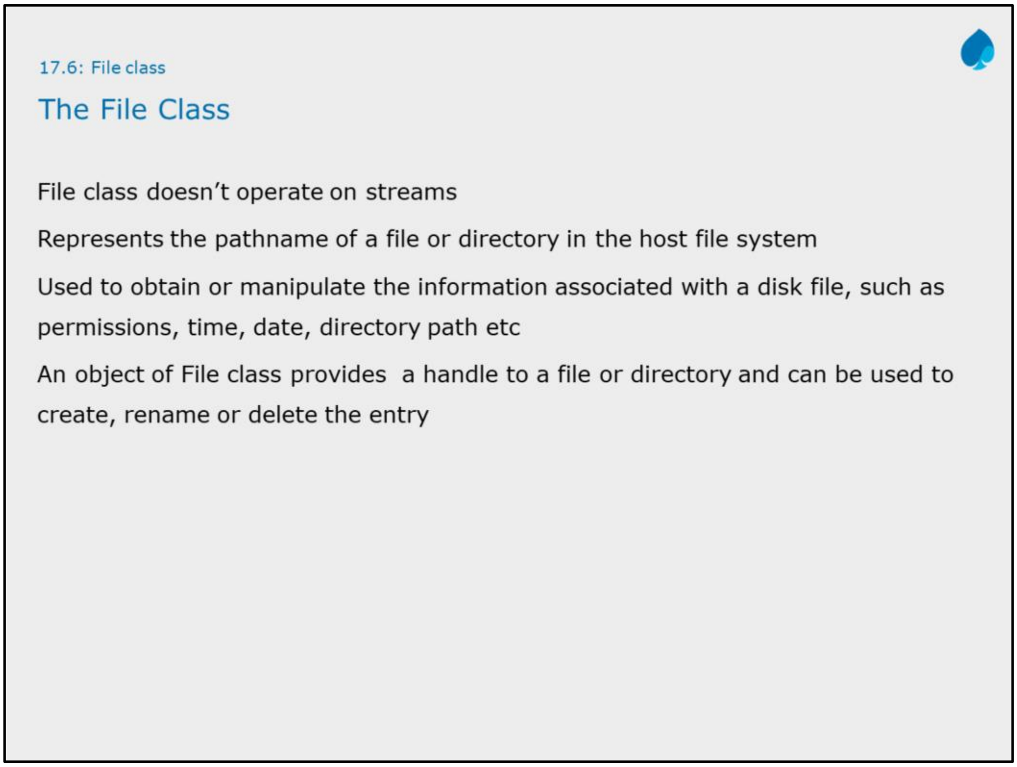


17.5: Buffered Stream

Demo: File Reader / File Writer

Execute the

- LineNumberReaderDemo.java
- CharEncode.java



17.6: File class

The File Class

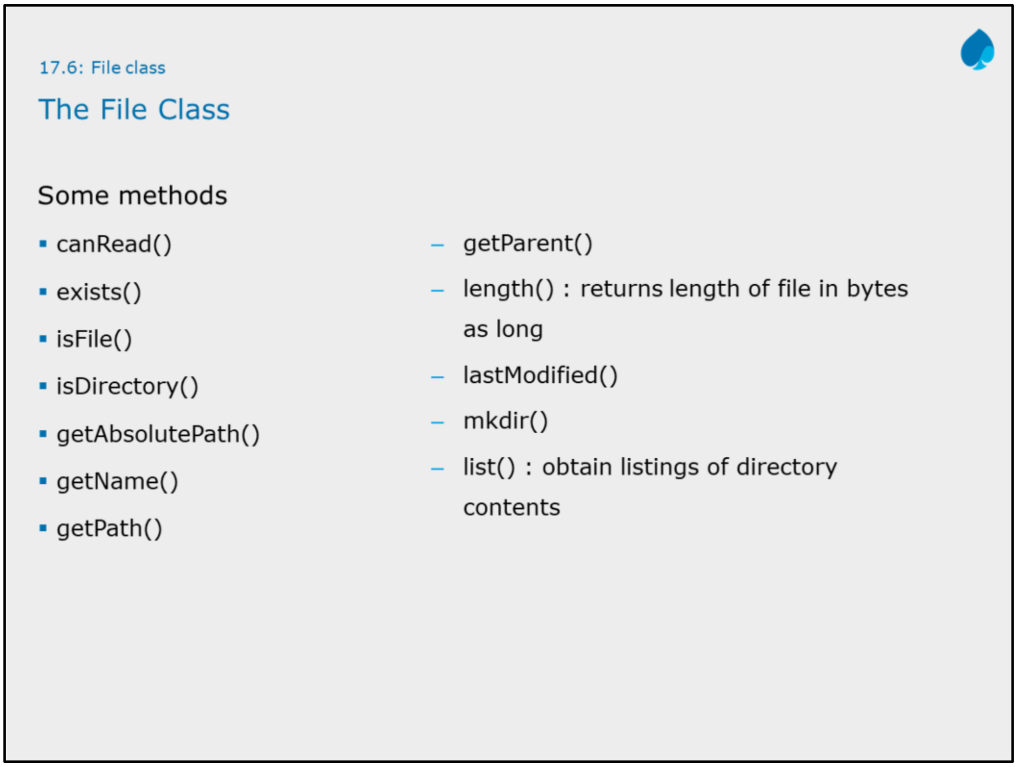
File class doesn’t operate on streams

Represents the pathname of a file or directory in the host file system

Used to obtain or manipulate the information associated with a disk file, such as permissions, time, date, directory path etc

An object of File class provides a handle to a file or directory and can be used to create, rename or delete the entry

Support for File/Directory Operations are provided by java.io.File. This class makes it easier to write platform-independent code that examines and manipulates files.
Provides methods
To obtain basic information about the file/directory
To Create / Delete Files and Directories



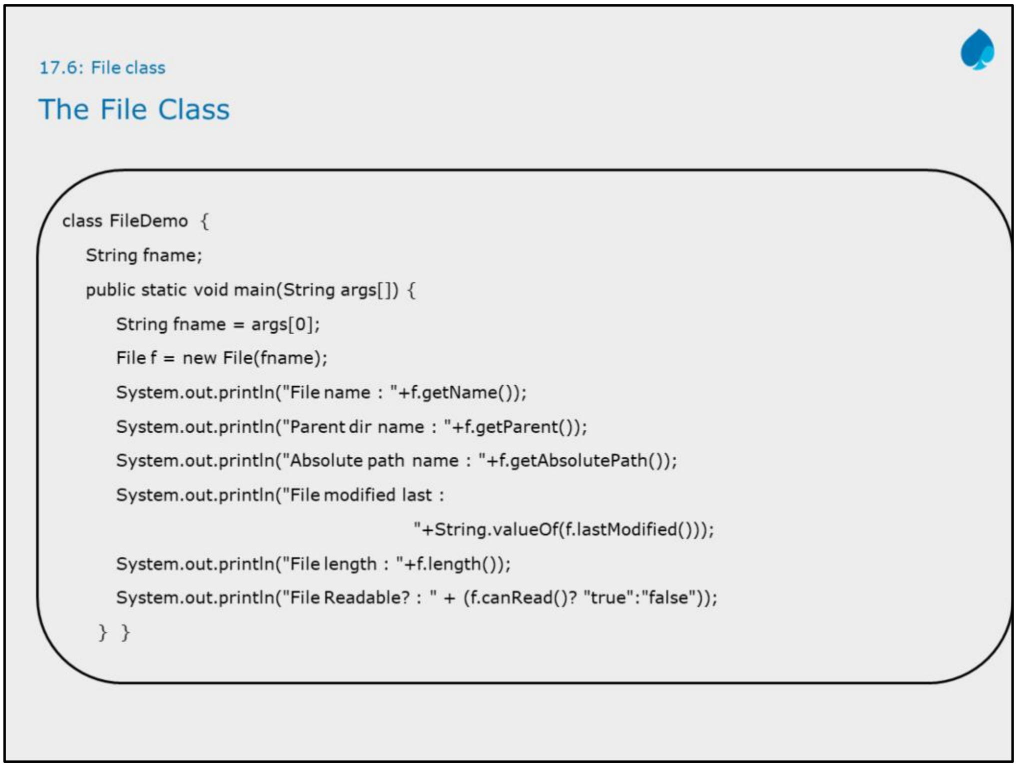
17.6: File class

The File Class

Some methods

- canRead()
- exists()
- isFile()
- isDirectory()
- getAbsolutePath()
- getName()
- getPath()

- getParent()
- length() : returns length of file in bytes as long
- lastModified()
- mkdir()
- list() : obtain listings of directory contents

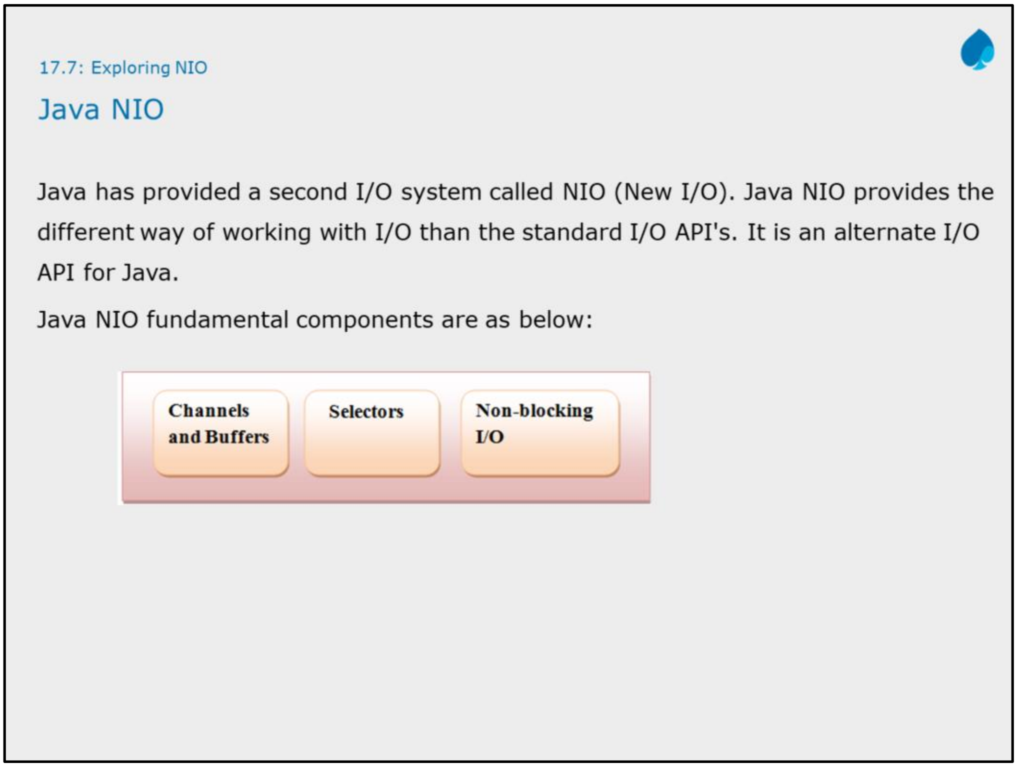


17.6: File class

The File Class

```
class FileDemo  {
   String fname;
   public static void main(String args[]) {
       String fname = args[0];
       File f = new File(fname);
       System.out.println("File name : "+f.getName());
       System.out.println("Parent dir name : "+f.getParent());
       System.out.println("Absolute path name : "+f.getAbsolutePath());
       System.out.println("File modified last :
                                         "+String.valueOf(f.lastModified()));
       System.out.println("File length : "+f.length());
       System.out.println("File Readable? : " + (f.canRead()? "true":"false"));
     } }
```

```
Output :
File name : books.xml
Parent directory name :    null
Absolute path name :    D:\G-drive contents\java-demo\day5(filesIO)\demo\file
handling\books.xml
File modified last on :    0
File length :    0
File Readable?  :    false
```



17.7: Exploring NIO

Java NIO

Java has provided a second I/O system called NIO (New I/O). Java NIO provides the different way of working with I/O than the standard I/O API's. It is an alternate I/O API for Java.

Java NIO fundamental components are as below:

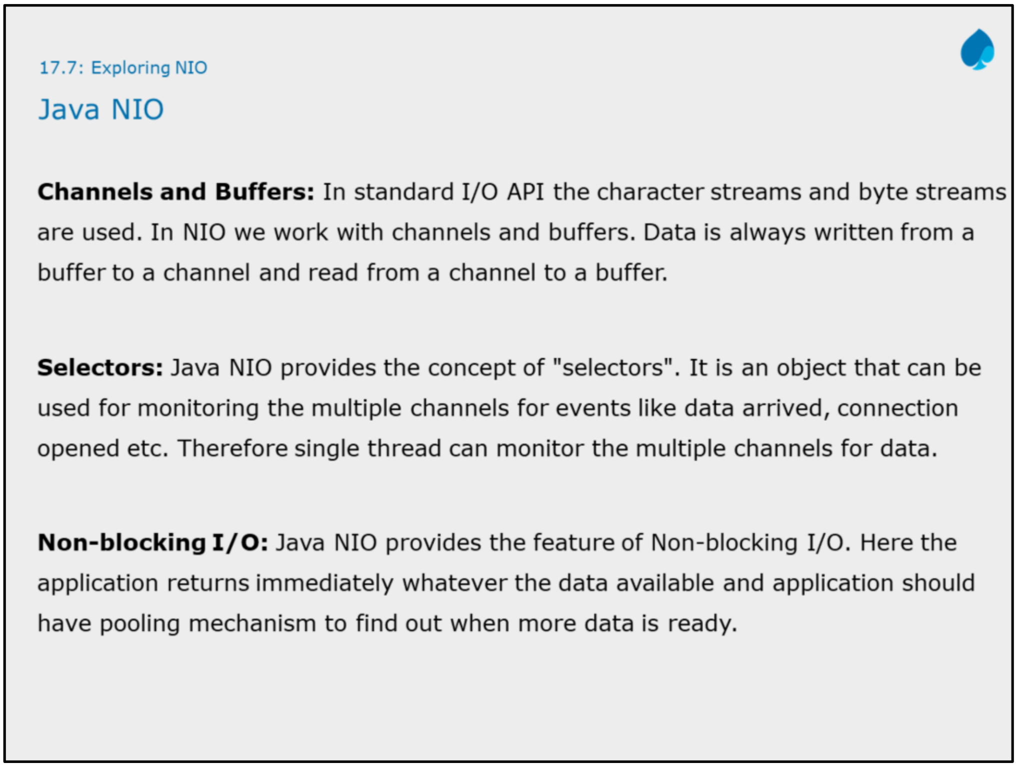
Channels and Buffers

Selectors

Non-blocking I/O

It supports a buffer-oriented, channel based approach for I/O operations. With the introduction of JDK 7, the NIO system is expanded, providing the enhanced support for file system features and file-handling. Due to the capabilities supported by the NIO file classes, NIO is widely used in file handling.

NIO was developed to allow Java programmers to implement high-speed I/O without using the custom native code. NIO moves the time-taking I/O activities like filling, namely and draining buffers, etc back into the operating system, thus allows for great increase in operational speed.



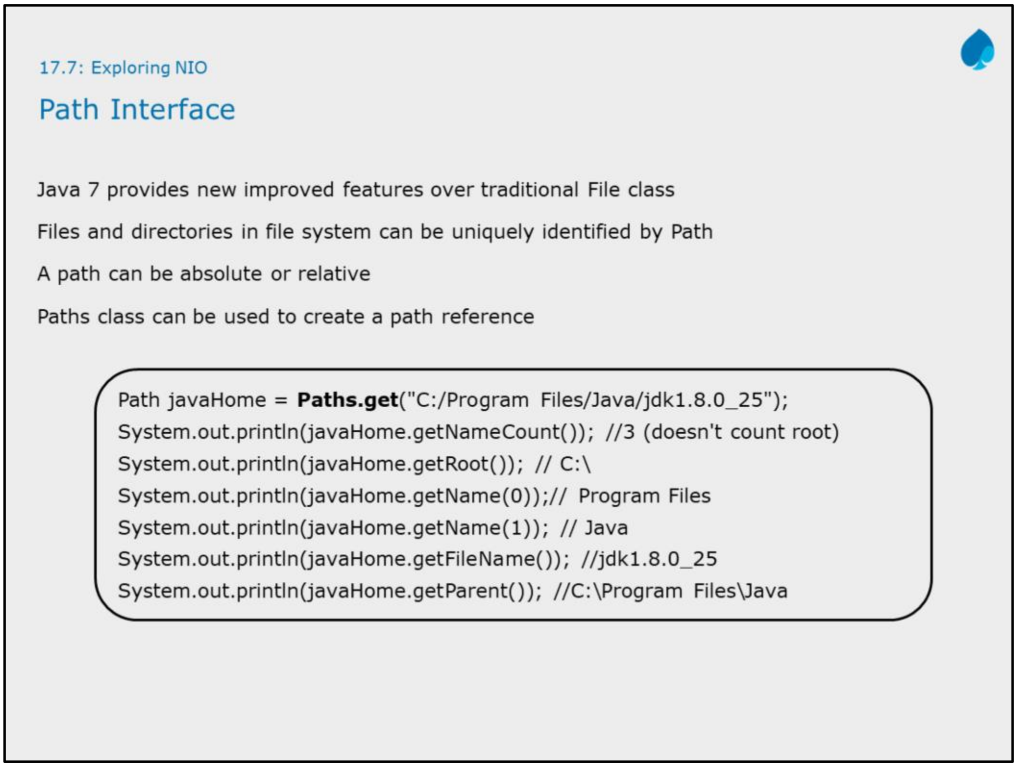
17.7: Exploring NIO

Java NIO

Channels and Buffers: In standard I/O API the character streams and byte streams are used. In NIO we work with channels and buffers. Data is always written from a buffer to a channel and read from a channel to a buffer.

Selectors: Java NIO provides the concept of "selectors". It is an object that can be used for monitoring the multiple channels for events like data arrived, connection opened etc. Therefore single thread can monitor the multiple channels for data.

Non-blocking I/O: Java NIO provides the feature of Non-blocking I/O. Here the application returns immediately whatever the data available and application should have pooling mechanism to find out when more data is ready.



17.7: Exploring NIO

Path Interface

Java 7 provides new improved features over traditional File class

Files and directories in file system can be uniquely identified by Path

A path can be absolute or relative

Paths class can be used to create a path reference

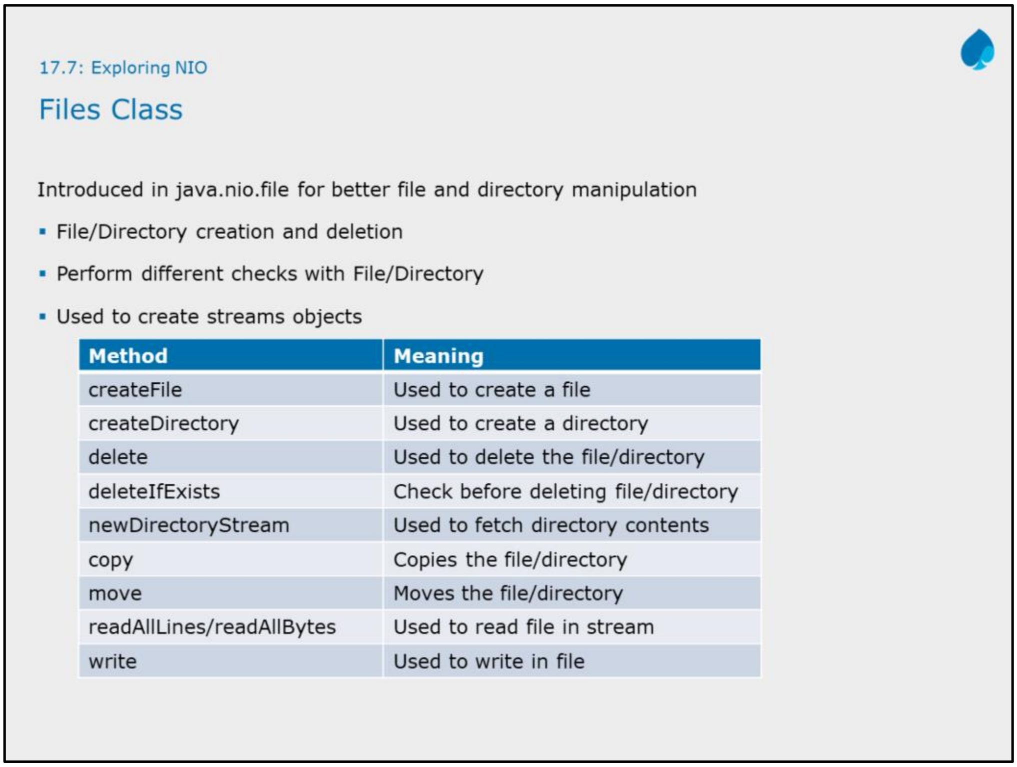
```
Path javaHome = Paths.get("C:/Program Files/Java/jdk1.8.0_25");
System.out.println(javaHome.getNameCount()); //3 (doesn't count root)
System.out.println(javaHome.getRoot()); // C:\
System.out.println(javaHome.getName(0));// Program Files
System.out.println(javaHome.getName(1)); // Java
System.out.println(javaHome.getFileName()); //jdk1.8.0_25
System.out.println(javaHome.getParent()); //C:\Program Files\Java
```

Path interface introduced in java.nio.file package to support better file handling and to overcome few drawbacks of traditional File class.

Path instance is used as reference to File or Directories as either relative or absolute path. Paths class is used to create a non-existance reference to file or directory. It means, creating reference of Path doesn’t create new file or directory.

Few methods of Path interface are shown in slide, the getNameCount() method is used to return count of path parts. Individual part of path can be retrieved by using getName(index); the index start from 0.

The getFileName() returns last part of path and getParent() returns parent path.



17.7: Exploring NIO

Files Class

Introduced in java.nio.file for better file and directory manipulation

- File/Directory creation and deletion
- Perform different checks with File/Directory
- Used to create streams objects

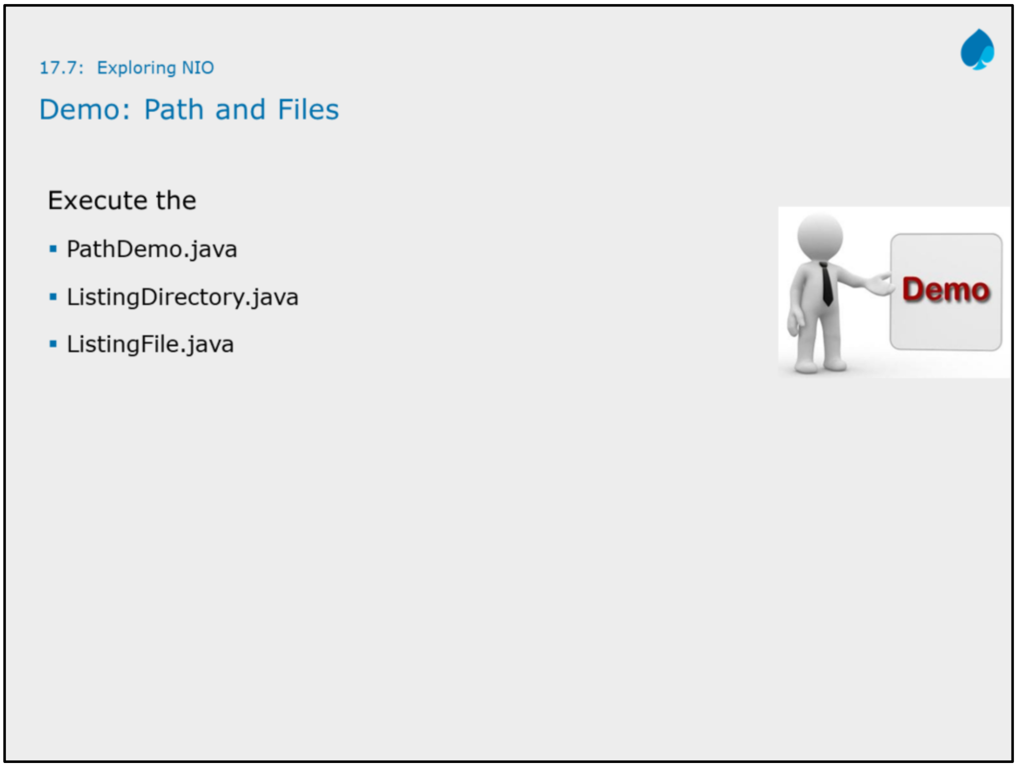
Method	Meaning
createFile	Used to create a file
createDirectory	Used to create a directory
delete	Used to delete the file/directory
deleteIfExists	Check before deleting file/directory
newDirectoryStream	Used to fetch directory contents
copy	Copies the file/directory
move	Moves the file/directory
readAllLines/readAllBytes	Used to read file in stream
write	Used to write in file

Files class contains lots of static methods to perform manipulation on files and directories. It also helps to retrieve streams from file/directory for reading or writing. The code snippet shown below is used to list all contents of directory.

```
Path javaHome= Paths.get("C:/Program Files/Java/jdk1.8.0_25");
DirectoryStream<Path> contents = Files.newDirectoryStream(javaHome);
for(Path content: contents) {
        System.out.println(content.getFileName());
}
contents.close();
```

Below listed snippet shows how to read the contents of a textual file with ease.

```
Path file = Paths.get("D:/output.txt");
List<String> lines = Files.readAllLines(file);
for(String line:lines) {
     System.out.println(line);
}
System.out.println("End of File....");
```

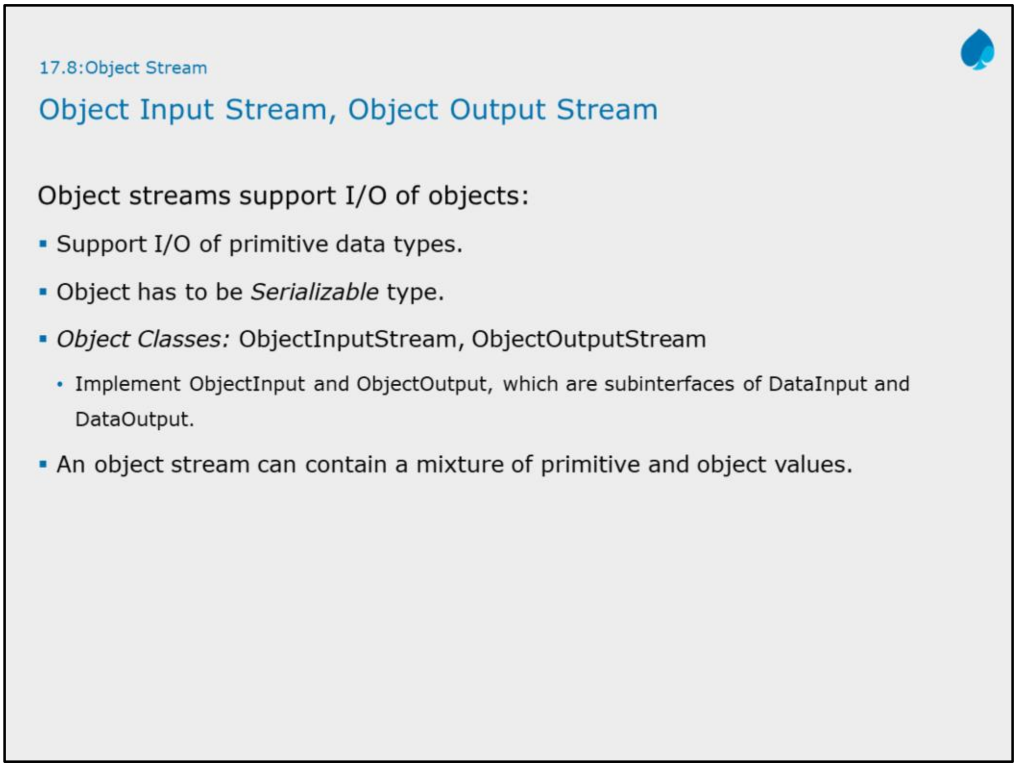


17.7: Exploring NIO

Demo: Path and Files

Execute the

- PathDemo.java
- ListingDirectory.java
- ListingFile.java



17.8:Object Stream

Object Input Stream, Object Output Stream

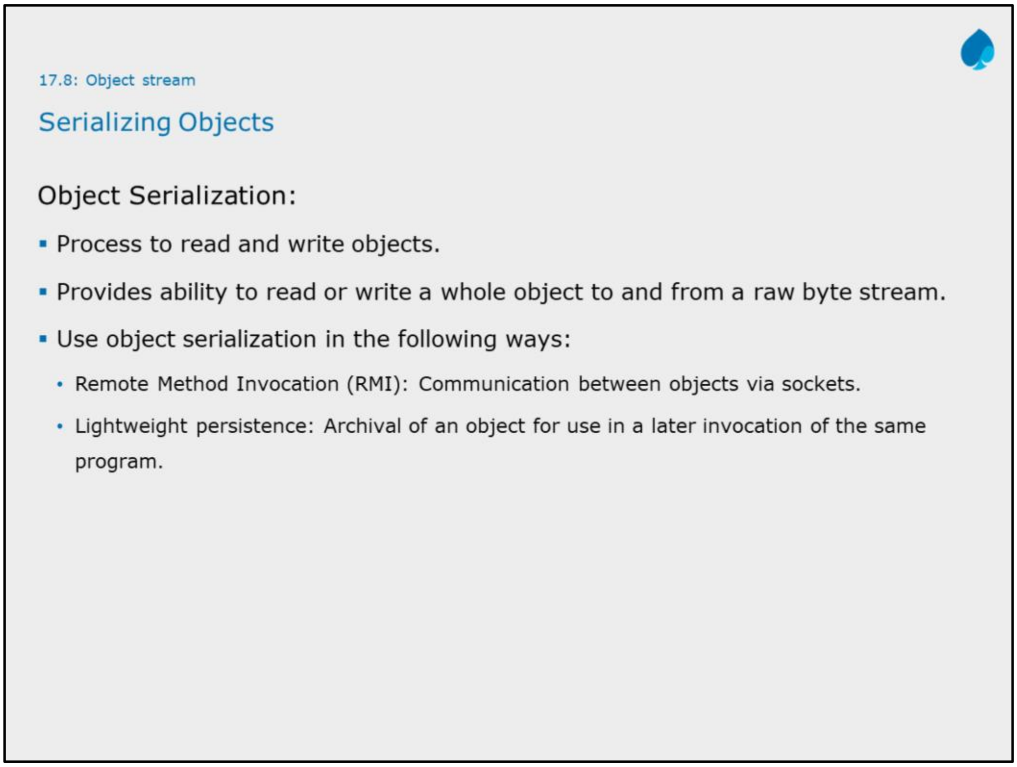
Object streams support I/O of objects:

- Support I/O of primitive data types.
- Object has to be *Serializable* type.
- *Object Classes:* ObjectInputStream, ObjectOutputStream
  - Implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.
- An object stream can contain a mixture of primitive and object values.

Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams.

The method readObject is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.

Primitive data types can be read from the stream using the appropriate method on DataInput.



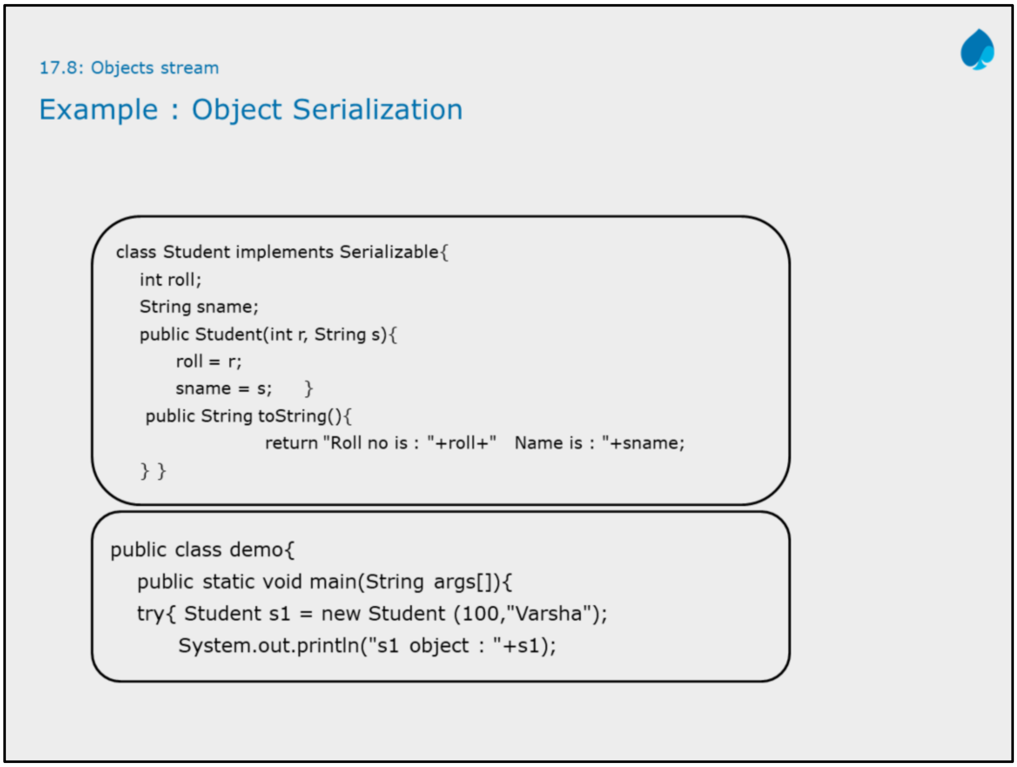
17.8: Object stream

Serializing Objects

Object Serialization:

- Process to read and write objects.
- Provides ability to read or write a whole object to and from a raw byte stream.
- Use object serialization in the following ways:
  - Remote Method Invocation (RMI): Communication between objects via sockets.
  - Lightweight persistence: Archival of an object for use in a later invocation of the same program.

Object Serialization allows an object to be transformed into a sequence of bytes that can be later re-created (deserialized) into an original object.
Java provides this facility through ObjectInput and ObjectOutput interfaces, which allow the reading and writing of objects from and to streams. These interfaces extend DataInput and DataOutput respectively
The concrete implementation of ObjectOutput and ObjectInput interfaces is provided in ObjectOutputStream and ObjectInputStream classes respectively.
These two interfaces have the following methods:
final void writeObject(Object obj) throws IOException.
final Object readObject() throws IOException, ClassNotFoundException
The writeObject() method can be used to write any object to a stream, including strings and arrays, as long as an object supports java.io.Serializable interface, which is a marker interface with no methods.
Serializing an object requires only that it meets one of two criteria. The class must either implement the Serializable interface (java.io.Serializable) which has no methods that you need to write or the class must implement the Externalizable interface which defines two methods. As long as you do not have any special requirements, making a serializable is as simple as adding the ``implements Serializable" clause.

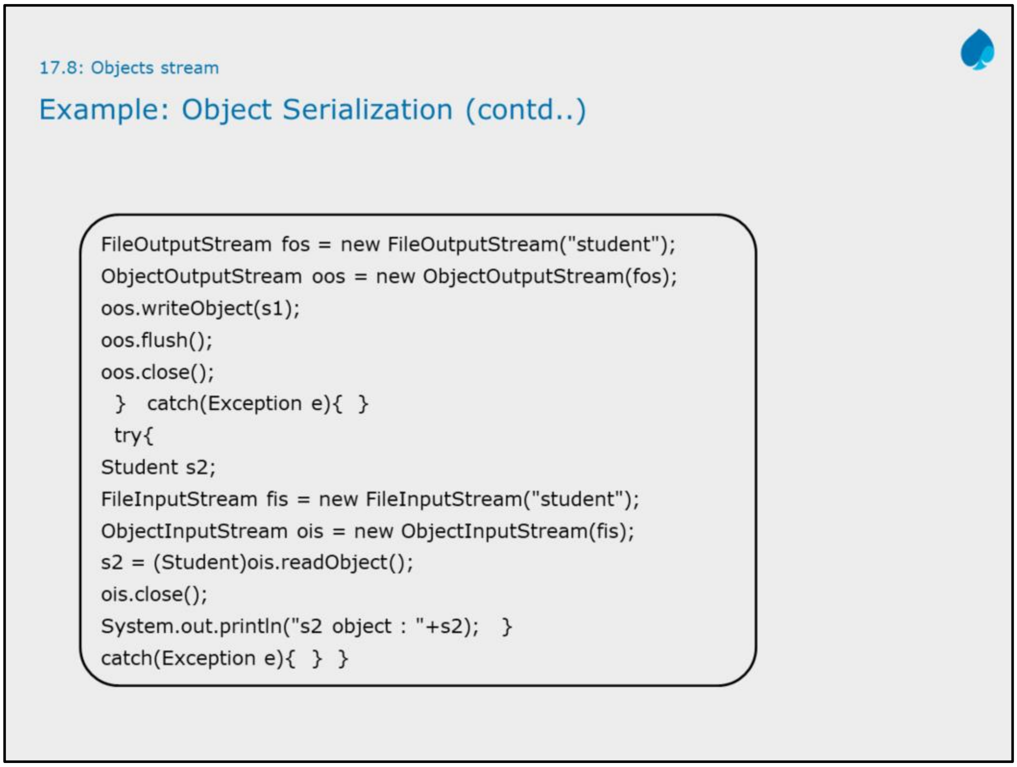


17.8: Objects stream

Example : Object Serialization

```
class Student implements Serializable{
    int roll;
    String sname;
    public Student(int r, String s){
          roll = r;
          sname = s;     }
     public String toString(){
                    return "Roll no is : "+roll+"   Name is : "+sname;
    } }
```

```
public class demo{
    public static void main(String args[]){
    try{ Student s1 = new Student (100,"Varsha");
          System.out.println("s1 object : "+s1);
```



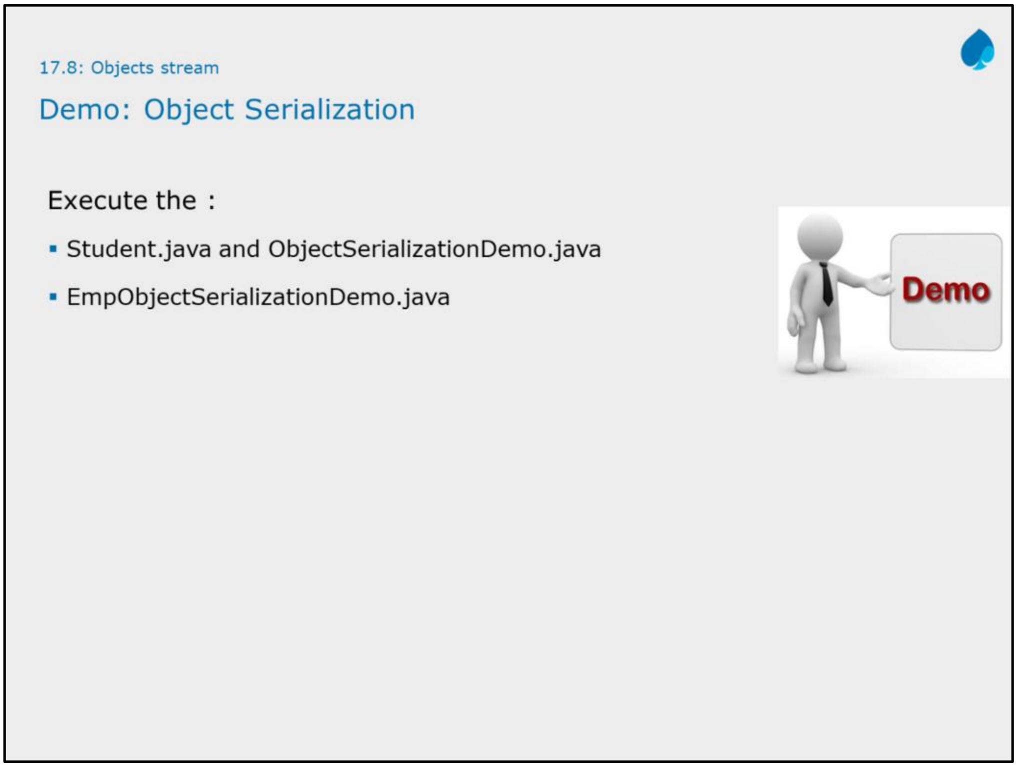
17.8: Objects stream

Example: Object Serialization (contd..)

```
FileOutputStream fos = new FileOutputStream("student");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(s1);
oos.flush();
oos.close();
 }  catch(Exception e){  }
 try{
Student s2;
FileInputStream fis = new FileInputStream("student");
ObjectInputStream ois = new ObjectInputStream(fis);
s2 = (Student)ois.readObject();
ois.close();
System.out.println("s2 object : "+s2);   }
catch(Exception e){  }  }
```

Output :

```
s1 object : Roll no is : 100   Name is : Varsha
s2 object : Roll no is : 100   Name is : Varsha
```

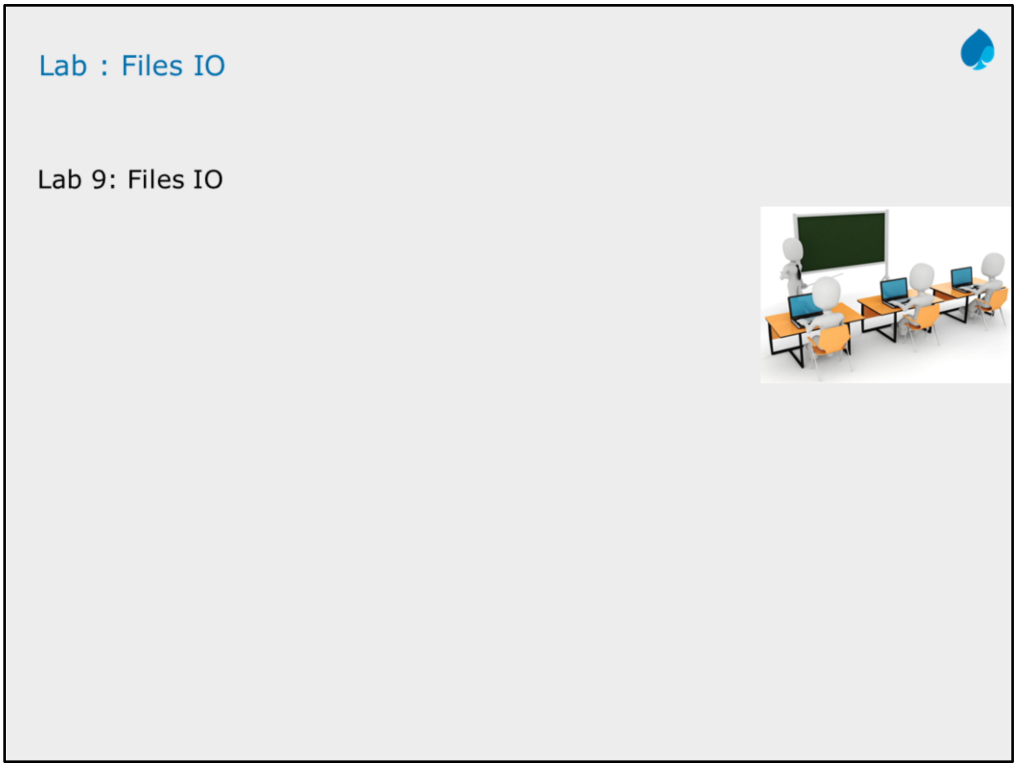


17.8: Objects stream

Demo: Object Serialization

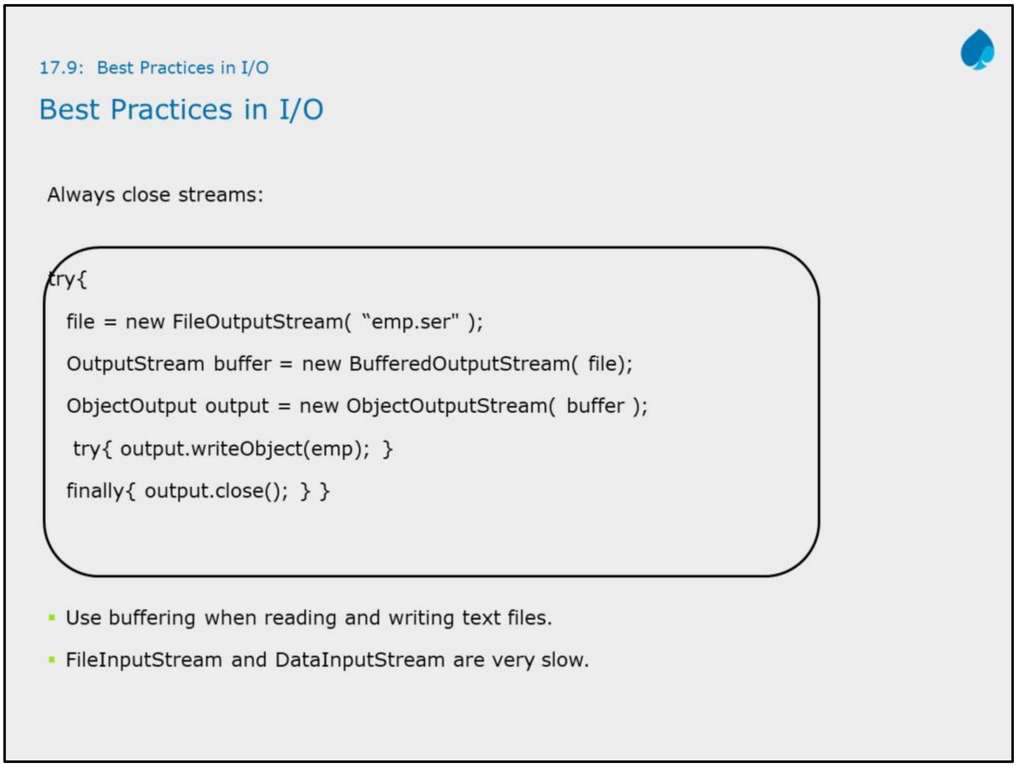
Execute the :

- Student.java and ObjectSerializationDemo.java
- EmpObjectSerializationDemo.java



Lab : Files IO

Lab 9: Files IO



17.9: Best Practices in I/O

Best Practices in I/O

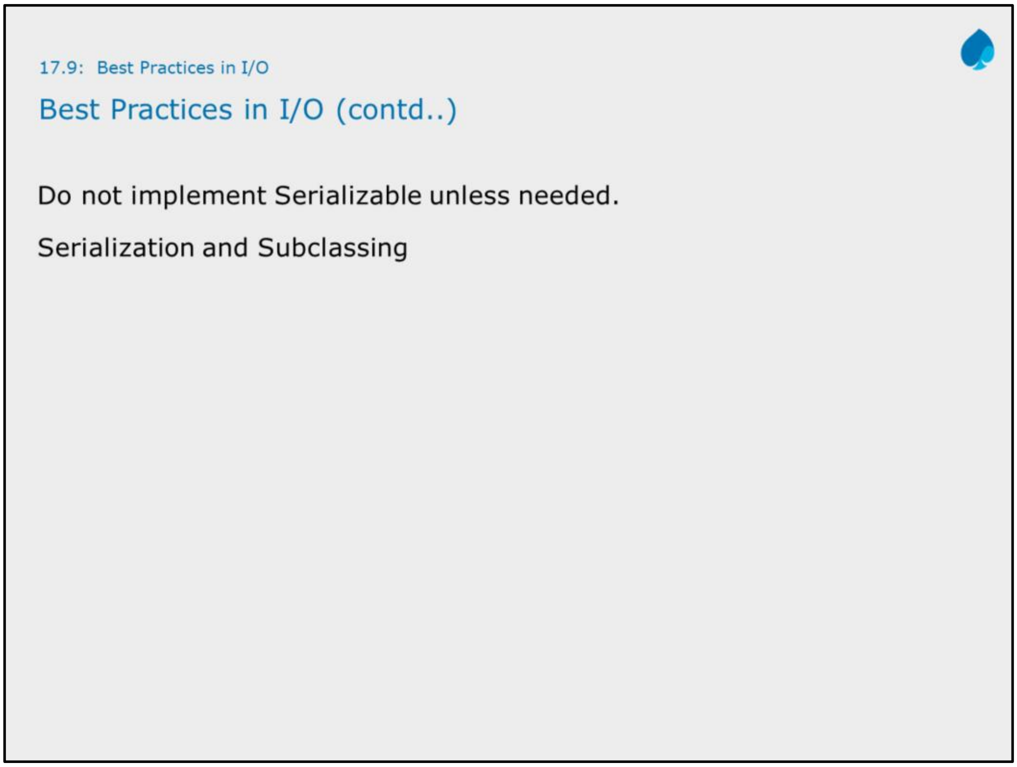
Always close streams:

```
try{
  file = new FileOutputStream( "emp.ser" );
  OutputStream buffer = new BufferedOutputStream( file);
  ObjectOutput output = new ObjectOutputStream( buffer );
   try{ output.writeObject(emp); }
  finally{ output.close(); } }
```

- Use buffering when reading and writing text files.
- FileInputStream and DataInputStream are very slow.

Always close streams
Streams represent resources which you must always clean up explicitly, by calling the close method. Some java.io classes (apparently just the output classes) include a flush method. When a close method is called on a such a class, it automatically performs a flush. There is no need to explicitly call flush before calling close.
One stream is chained to another by passing it to the constructor of some second stream. When this second stream is closed, then it automatically closes the original underlying stream as well.
If multiple streams are chained together, then closing the one which was the last to be constructed, and is thus at the highest level of abstraction, will automatically close all the underlying streams. So, one only has to call close on one stream in order to close (and flush, if applicable) an entire series of related streams.
Reading and Writing text files : When reading and writing text files:
 it is almost always a good idea to use buffering (default size is 8K)
 Unbuffered input and output classes operate only on one byte at a time.
Using a buffer will often increase performance by large factors.

FileInputStream & DataInputStream is very slow : since they call read() for every character. Use FileReader and BufferedReader instead. Reader objects use Large Buffer. Unbuffered input/output classes operate only on one byte at a time. Using a buffer will often increase performance by large factors.



17.9: Best Practices in I/O

Best Practices in I/O (contd..)

Do not implement Serializable unless needed.

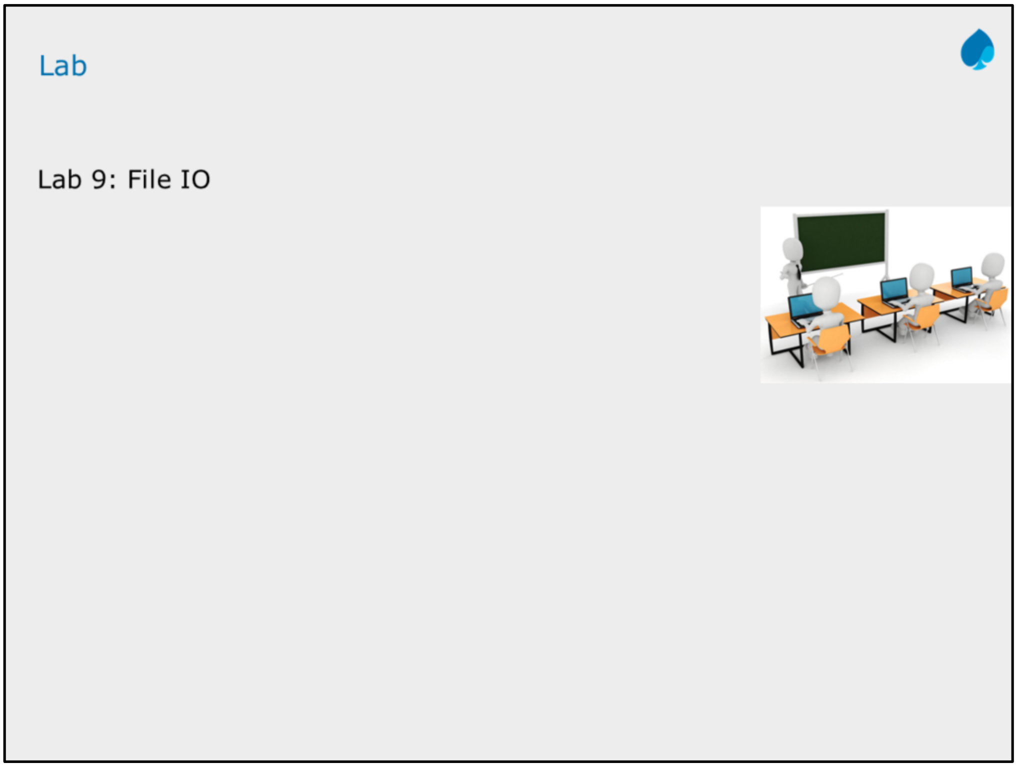
Serialization and Subclassing

Implementing Serializable
Do not implement Serializable lightly, since it restricts future flexibility, and publicly exposes class implementation details which are usually private.

Serialization and Subclassing
Interfaces and classes designed for inheritance should rarely implement Serializable, since this would force a significant and (often unwanted) task on their implementors and subclasses.

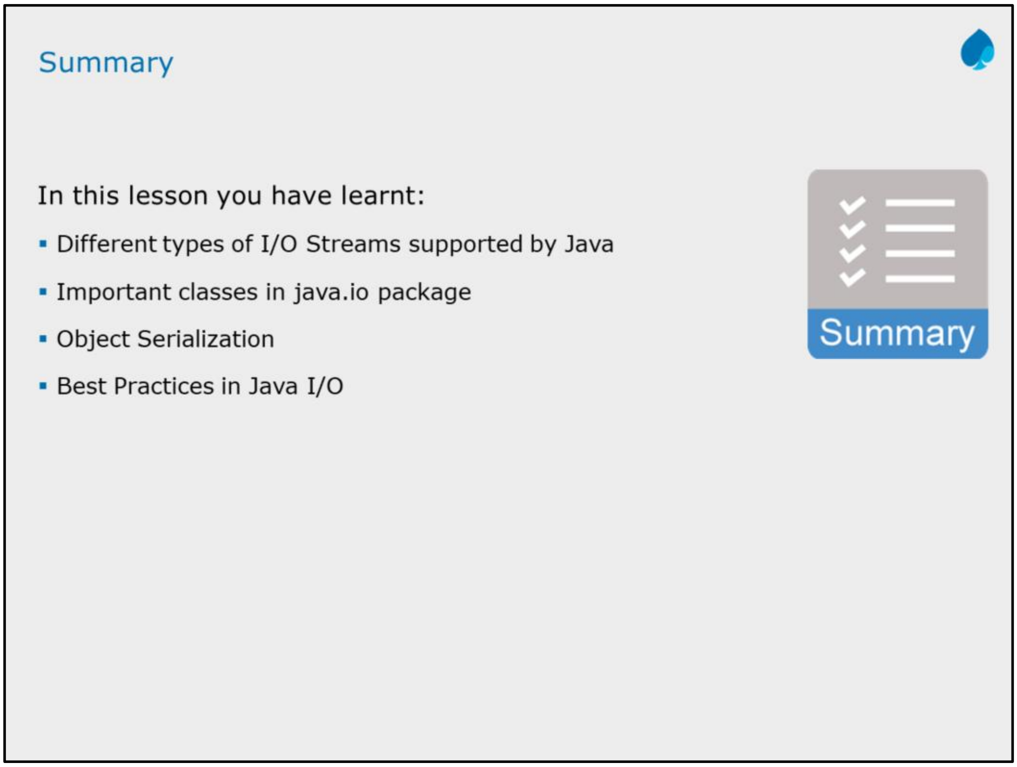
However, even though most abstract classes do not implement Serializable, they may still need to allow their subclasses to do so, if desired.

There are two cases. If the abstract class has no state, then it can simply provide a no-argument constructor to its subclasses. If the abstract class has state, however, then there is more work to be done



Lab

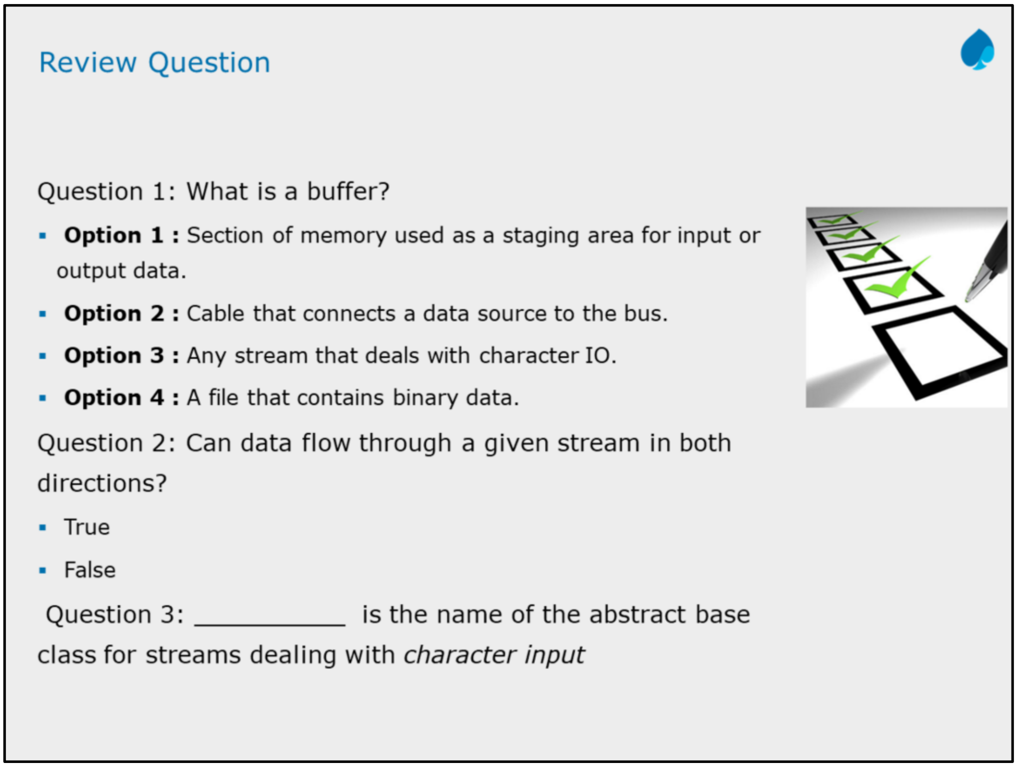
Lab 9: File IO



Summary

In this lesson you have learnt:

- Different types of I/O Streams supported by Java
- Important classes in java.io package
- Object Serialization
- Best Practices in Java I/O



Review Question

Question 1: What is a buffer?

- **Option 1 :** Section of memory used as a staging area for input or output data.
- **Option 2 :** Cable that connects a data source to the bus.
- **Option 3 :** Any stream that deals with character IO.
- **Option 4 :** A file that contains binary data.

Question 2: Can data flow through a given stream in both directions?

- True
- False

Question 3: ____________ is the name of the abstract base class for streams dealing with *character input*