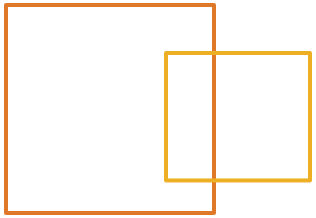
Fast Track to Java

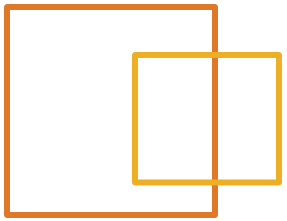
Customized for Starbucks

Delivered by DevelopIntelligence

Java IO

Objectives

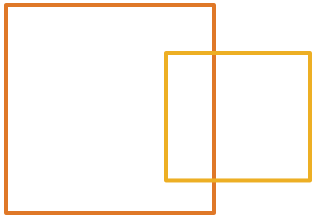


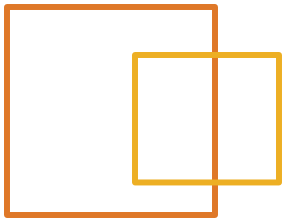


At the end of this module you should be able to

- Describe the architecture of the `java.io` API
- Describe the streams model
- Use implementation streams
- Use filter streams
- Describe the difference between streams, readers and writers
- Use data streams and files
- Use buffered I/O and the `PrintWriter`
- Describe the File class

I/O in Java





Java has two main types of I/O

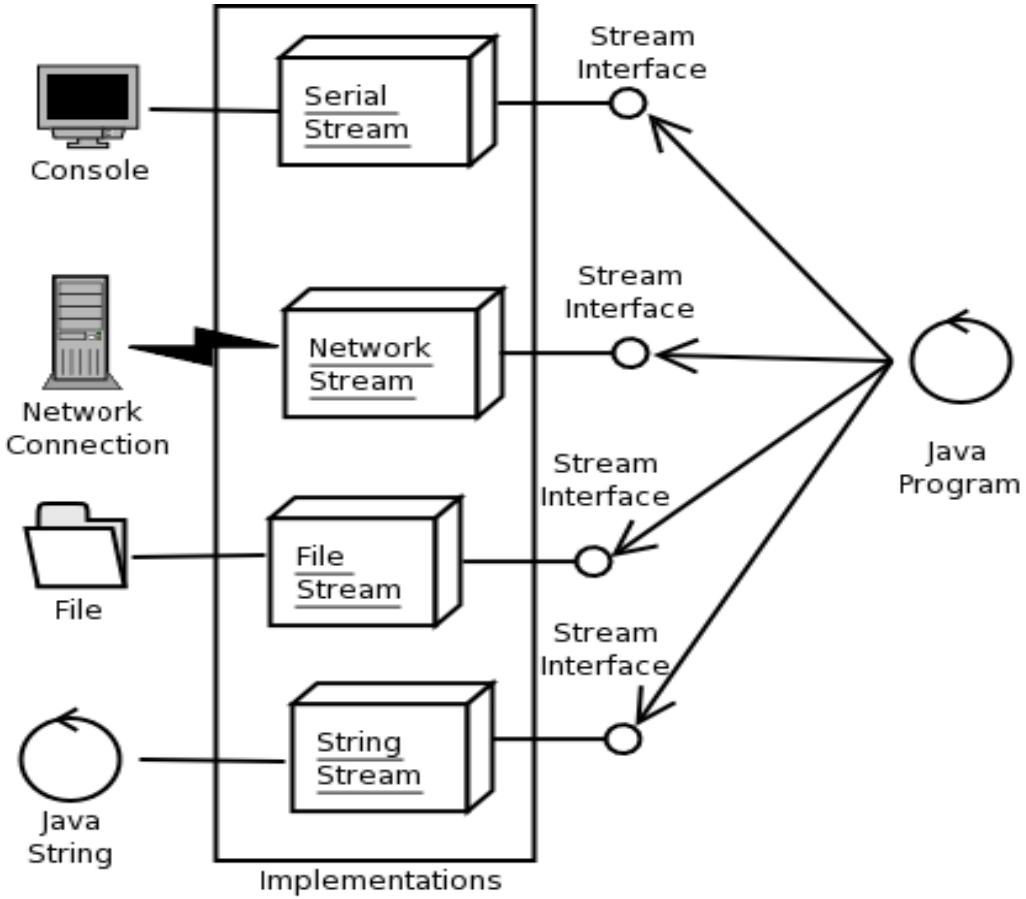
1. Blocking I/O
   - ***`java.io`***
   - Referred to as synchronous I/O
   - Utilizes streams

2. New (fast and non-blocking) I/O
   - ***`java.nio`***
   - Referred to as asynchronous I/O
   - Utilizes channels
   - We will not cover NIO

I/O in Java (cont.)

- Blocking I/O has facilities for two types of streams
  - Binary streams
  - Character streams
- A stream is a reference to a “flowing” sequence of bytes
- Anything can generate a stream
  - Network connection
  - Database connection
  - File connection
  - Even ***`String`***

Architecture of a Stream Approach to I/O

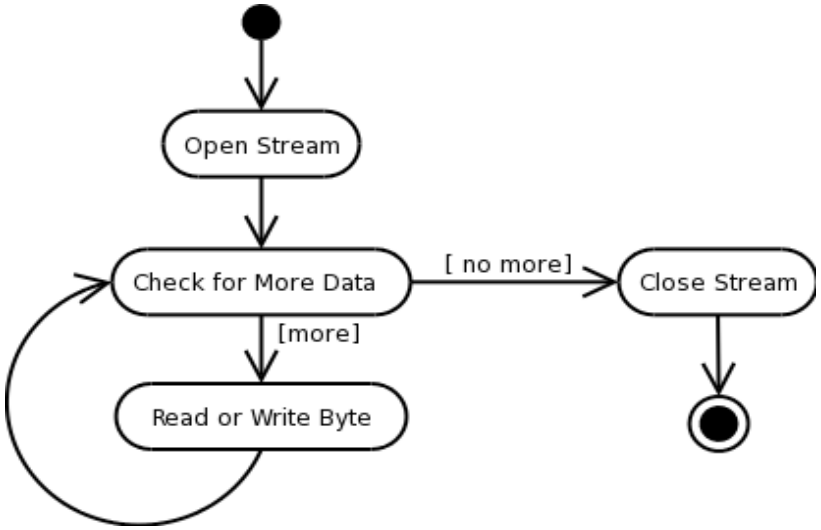


Logical architecture of a Stream approach to I/O

Streams Model

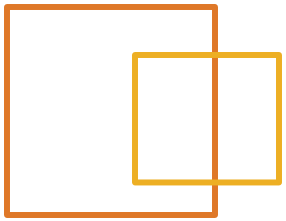
The basic stream programming model is the same for binary and character streams

1. Open the Stream
    1. Create the stream object
    2. Initialize the stream object
2. Perform operations
    1. Read the data
    2. Write the data
3. Close the stream



Basic logic for using an InputStream or OutputStream

The Top Level interfaces



java.io.Reader **Interface**

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

java.io.InputStream **Interface**

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

java.io.Writer **Interface**

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

java.io.OutputStream **Interface**

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

I/O APIs

- The top-level I/O APIs are pretty low-level
  - Good for low-level OS communication
  - Useful when dealing with proprietary protocols
- Java provides higher-level I/O APIs
  - Rely on low-level I/O APIs
  - Provide convenience input and output methods
  - Many variations
    - ***`FileReader / FileWriter`***
    - ***`FileInputStream / FileOutputStream`***
    - ***`PrintWriter / PrintStream`***
    - ***`BufferedReader / BufferedWriter`***
- Both sets of APIs utilize ***`Exception`***s

FileReader and FileWriter Example

```
import java.io.*;
public class CopyTextFile {
  public static void main(String[] args) {
    int count = 0;
    try {
      FileReader in = new FileReader("filesource");
      FileWriter out = new FileWriter("filesink");
      int c;
      while ((c = in.read()) != -1){
        count++;
      }
      out.write(c);
      in.close();
      out.close();
    } catch (IOException e) {
      System.err.println(e.getMessage());
    }
    System.out.println("Copied " + count + " characters");
  }
}
```

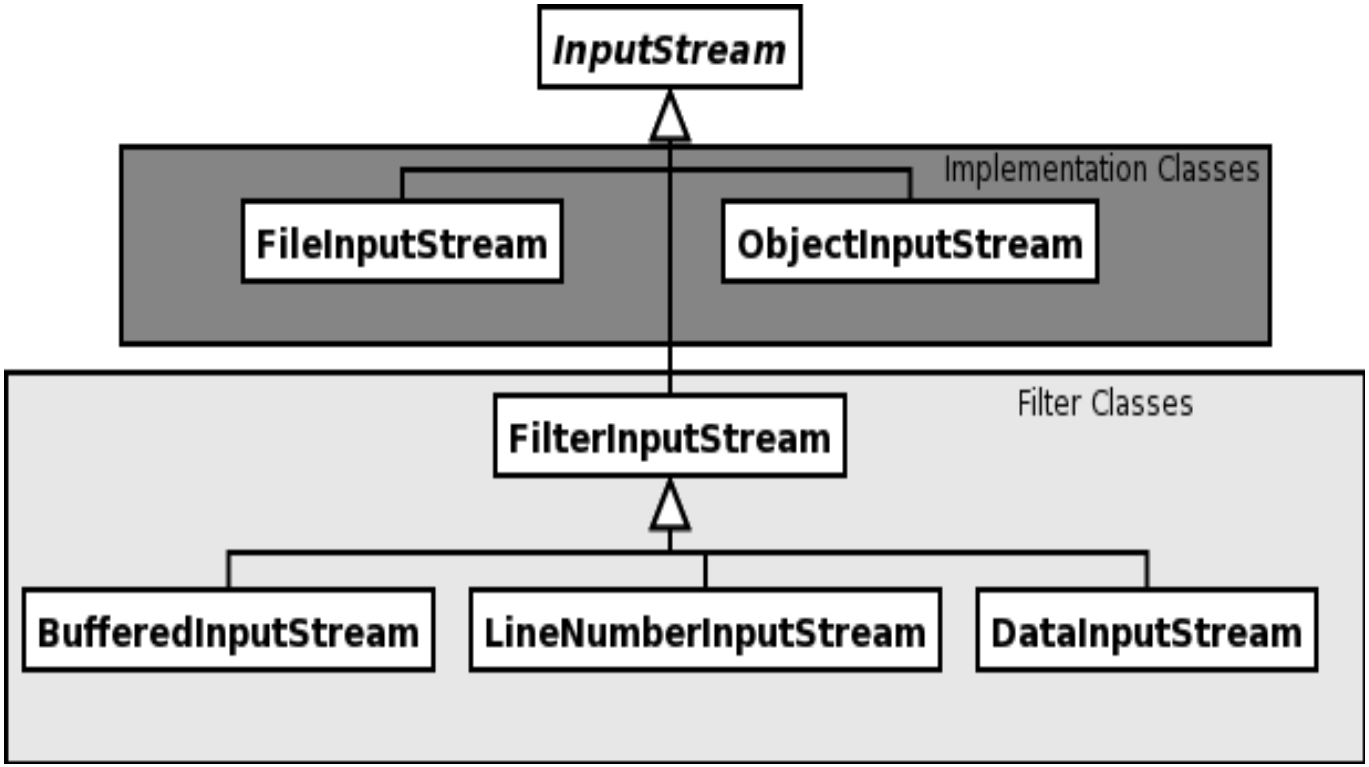
StringReader and FileWriter Example

```
import java.io.*;
public class Ex10_2 {
  public static void main(String[] args) throws IOException {
    String s =
      "This is the string source to be used for input.";
    StringReader in = new StringReader(s);
    FileWriter out = new FileWriter("filesink");
    int c;
    int count = 0;
    while ((c = in.read()) != -1){
      count++;
    }
    out.write(c);
    in.close();
    out.close();
    System.out.println("Copied "+count+" bytes");
  }
}
```

Decorator Pattern

- The I/O package has many classes, creating subclasses for every permutation would be unreasonable
- So, I/O in Java utilizes an object oriented design pattern
  - *Decorator Pattern* adds functionality to an object by *wrapping* it instead of sub-classing it
  - Decorator objects can be wrapped by other decorator objects that can be wrapped by other decorator objects . . .

Filter Streams and the Decorator Pattern

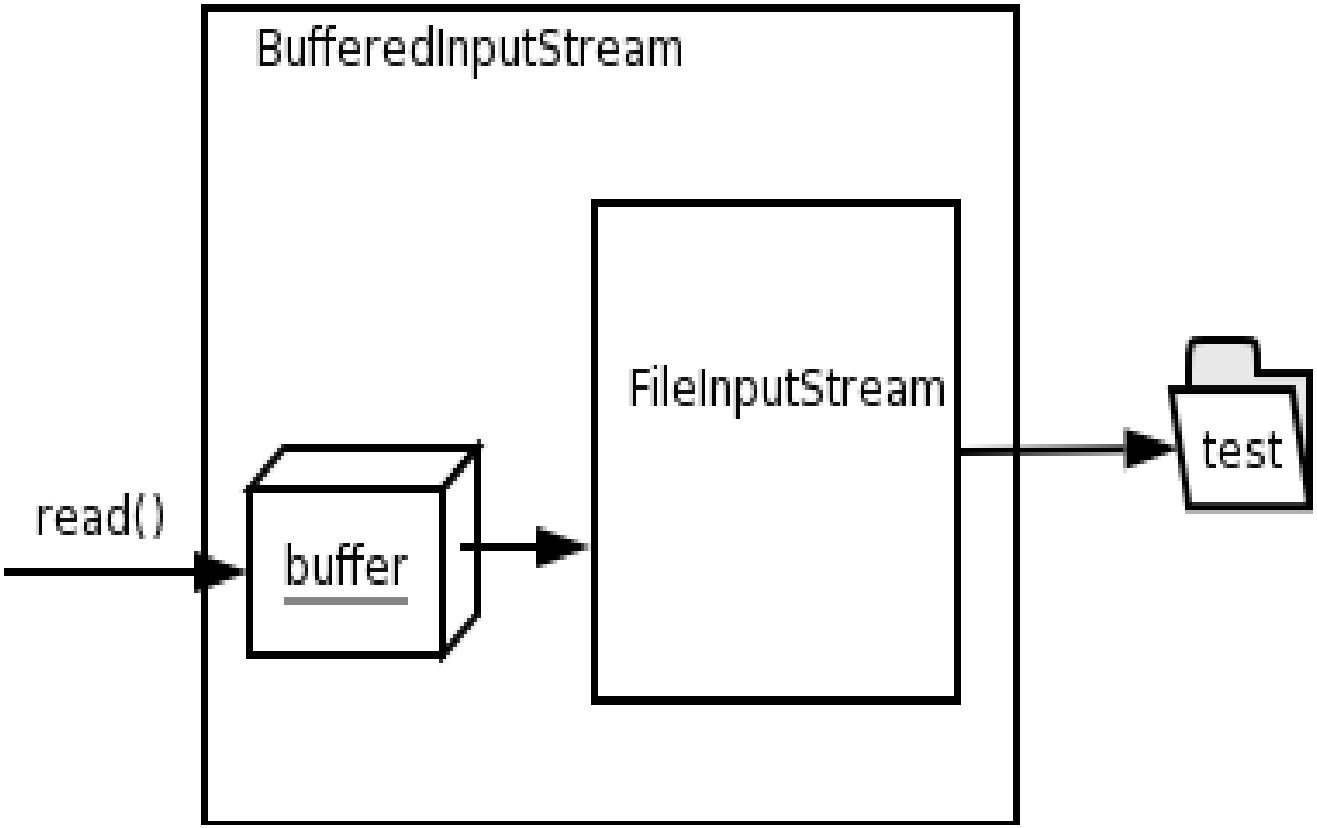


Part of the stream hierarchy

Using the Decorator Pattern

- Create a reference to the low-level stream
- Pass that reference to the decorator upon construction
- Call I/O operations on the decorator
  - Decorator initially performs operation
  - Decorator delegates operation to low-level stream
  - Low-level stream interactions are hidden from you and taken care of for you

Using the Decorator Pattern (cont.)



A decorated FileInputStream

Filter Class Example

```
import java.io.*;
public class Ex10_3 {
  public static void main(String[] args) {
    try {
      BufferedReader input =
        new BufferedReader(
          new FileReader("TestInput.text"));
      String inputLine = new String();
      System.out.println("File output...");
      while((inputLine = input.readLine()) != null){
        System.out.println(inputLine);
      }
      input.close();
    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

Converting Streams

- The two categories of streams are
  - Binary – used with binary data
  - Character – used with Unicode character data
- Classes in the binary category have a different inheritance tree than those of the character category
- Therefore you can't cast a binary stream into a character stream
- If you can't cast it, convert it!
- Two conversion utility classes
  - ***InputStreamReader***
  - ***OutputStreamWriter***

Stream Conversion Example

```
import java.io.*;
public class Ex10_4 {
  public static void main(String[] args) {
    // Create the Reader
    Reader r = new InputStreamReader(System.in);
    // Create the Buffered Reader
    BufferedReader input = new BufferedReader(r);
    try {
      while (true) {
        System.out.print("Enter a line ('end' terminates):");
        String s = input.readLine();
        if (s.equals("end"))
          break;
        System.out.println("You said -- " + s);
      }
      System.out.println("bye");
    } catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

PrintWriter

- Writer is pretty low level
- `FileWriter` allows us to write to a file
- `BufferedWriter` makes `FileWriter` better
- But, what if we want to write out lines of text to a file in a single operation?
- Use `PrintWriter`!

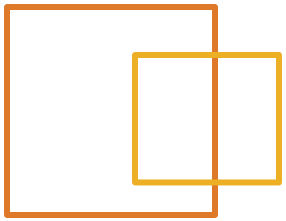
Using a PrintWriter

```
import java.io.*;
public class Ex10_5 {
   public static void main(String[] args) {
      // Create the Buffered Reader
      BufferedReader input = new BufferedReader(
         new InputStreamReader(System.in));
      try {
         // Create the writer (buffered!)
         PrintWriter pw = new PrintWriter(
                              new BufferedWriter(
                                new FileWriter("dialog.text")));
         pw.println("----------- Starting");
         int lineNum = 1;
. . .
```

Using a PrintWriter (cont.)

```
            while (true) {
              System.out.print("Enter a line ('end' terminates):");
              String s = input.readLine();
              if (s.equals("end")) {
                break;
              }
              System.out.println("You said -- " + s);
              pw.print(lineNum++);
              pw.println(")  "+s);
            }
            System.out.println("bye");
            pw.println("----------- Done");
            pw.close();
          } catch (Exception e) {
            System.err.println(e);
          }
      }
  }
```

Writing Java Data



The I/O API provides two sets of classes for reading and writing Java specific data

1. **`DataInputStream`** / **`DataOutputStream`**
   - Used for reading / writing primitive data
   - Method for each primitive type
   - Preserves platform independence
   - Can be used to persist state of an object, but done very manually
2. **`ObjectInputStream`** / **`ObjectOutputStream`**
   - Used for reading / writing objects
   - Utilizes *Object Serialization*
   - Easiest way to persist state of an object

DataStreams Example

```
import java.io.*;
public class Ex10_6 {
  public static void main(String[] args) {
    try {
      DataOutputStream out = new DataOutputStream(
                                 new BufferedOutputStream(
                                  new FileOutputStream("Data.tmp")));
      out.writeDouble(839.829);
      System.out.println("Wrote " + 839.829);
      out.writeInt(-1872);
      System.out.println("Wrote " + (-1872));
      out.close();
 . . .
```

DataStreams Example (cont.)

```
      DataInputStream in = new DataInputStream(
                                 new BufferedInputStream(
                                   new FileInputStream("Data.tmp")));
      double d = in.readDouble();
      System.out.println("Read " + d);
      int i = in.readInt();
      System.out.println("Read " + i);
      in.close();
    } catch (Exception e) {
      //bad practice.. But quick and dirty
      throw new RuntimeException(e);
    }
  }
}
```

Object Serialization

- Introduced as part of the Java Beans specification
- Complex mechanism to persist and restore the state of an object
  - Utilizes something referred to as object graphs
  - Objects within objects within objects are all stored
- Two types
  1. Automatic
     - Follow some rules
     - ***`implement java.io.Serializable`***
     - Persistence and restoration are done for you
  2. Manual
     - Do most everything yourself
     - ***`implement java.io.Externalizable`***
     - Complete control

Automatic Serialization Rules

- Class should be ***`public`***
- Instance variables you don't want saved should be marked ***`transient`***
- Should have a ***`public`*** no-arg constructor
- Must implement ***`java.io.Serializable`***

Object Serialization

```
import java.io.*;
class DataObject implements Serializable {
     private int id;
     public DataObject(int n) {
        id = n;
     }
     public String toString() {
        return " DataObject " + id;
     }
  }
```

Object Serialization

```
public class Ex10_7 implements Serializable {
   private DataObject[] objects = { new DataObject(981),
                                    new DataObject(3),
                                    new DataObject(-98)
                                  };
   private String id = "Container";
   public String toString() {
     String s = "";
     for (int i = 0; i < 3; i++){
       s += objects[i];
     }
     return id + " " + s;
   }

   public static void main(String[] args) {
      Ex10_7 c = new Ex10_7();

. . .
```

Object Serialization

```
        try {
          System.out.println("Created object");
          System.out.println(c);
          ObjectOutputStream out = new ObjectOutputStream(
                                   new FileOutputStream("somewhere"));
          out.writeObject(c);
          out.close();
          c = null; // object is now out of scope.
          System.out.println("Written and destroyed.");
          ObjectInputStream in = new ObjectInputStream(
                                   new FileInputStream("somewhere"));
          Ex10_7 newc = (Ex10_7) in.readObject();
          in.close();
          System.out.println("Read.");
          System.out.println("Recovered  object");
          System.out.println(newc);
        } catch(Exception e) {
          System.err.println(e);
      }
    } //end main
} //end class
```

File Interactions

- Java provides classes to work with underlying file systems in platform independent manner
- The ***`File`*** class allows you to create an object representation for a file
- A ***`File`*** instance is not the file itself; but allows you to
  - Find out information about the underlying file
  - Delete, rename, move the underlying file
  - Check permissions
  - Etc.

The File Class

```
import java.io.*;
public class Ex10_8 {
  public static void main(String[] args) {
     // get the current path
     File pwd = new File(".");
     String[] dirList = pwd.list();
     for(int i = 0; i < dirList.length; i++) {
       System.out.println(dirList[i]);
     }
  }
}
```

The File Class

```
C:\>java Ex10_8
arcldr.exe
arcsetup.exe
AUTOEXEC.BAT
boot.ini
CONFIG.SYS
Documents and Settings
Ex10_8.class
IO.SYS
j2sdk1.4.2_01
MSDOS.SYS
My Documents
NTDETECT.COM
ntldr
pagefile.sys
Program Files
RECYCLER
System Volume Information
WINNT
Work

C:\>_
```

Output from example

Some new I/O classes in JDK 7

- `Path/Paths`
  - Classes to help navigate and manipulate file system paths.
- `Files`
  - Replacement for much of the functionality of the `File` class. Plus a whole lot more.
  - The `Files` API makes extensive use of the `Path` class

Summary

We covered

- Describing the architecture of the ***`java.io`*** API
- Describing the streams model
- Using implementation streams
- Using filter streams
- Describing the difference between streams, readers and writers
- Using data streams and files
- Using buffered I/O and the ***`PrintWriter`***
- Describing the ***`File`*** class

Lab 9

- I/O
  - Write a program that prompts the user for a file name, and attempts to display information about the file (size, read/write permissions, parent directory) and the contents of the file (as text) on the console. If the file does not exist, loop round and prompt the user to re-enter the filename, otherwise exit.
  - Solution: SimpleFileLab/SimpleFileLabToo