CS 5004

LECTURE 9

ART OF DESIGN

JAVA I/O, MODEL-VIEW-CONTROLLER (FOCUS: THE MODEL)

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ **T** 617.373.2462 ▪ **khoury.northeastern.edu**

AGENDA

- Java I/O
- Model-View-Controller & Other Architectures
- MVC: The Model
- Q & A

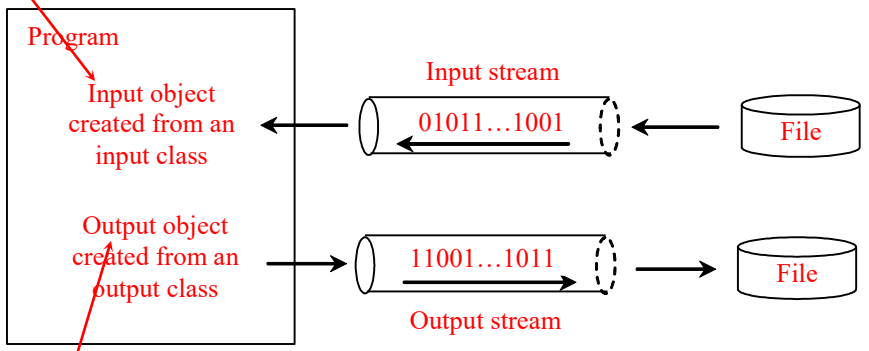
JAVA I/O

- Many slides from this section are taken from Liang © 2020

HOW IS I/O HANDLED IN JAVA?

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes.

```
Scanner input = new Scanner(new
File("temp.txt"));
System.out.println(input.nextLine());
```




```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```

TEXT VS BINARY FILES

- Data stored in a text file are represented in human-readable form.
- Data stored in a binary file are represented in binary form.
  - For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM.
  - Computers process binary files are more efficiently than text files.

THE FILE CLASS

The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The File class is a wrapper class for the file name and its directory path.

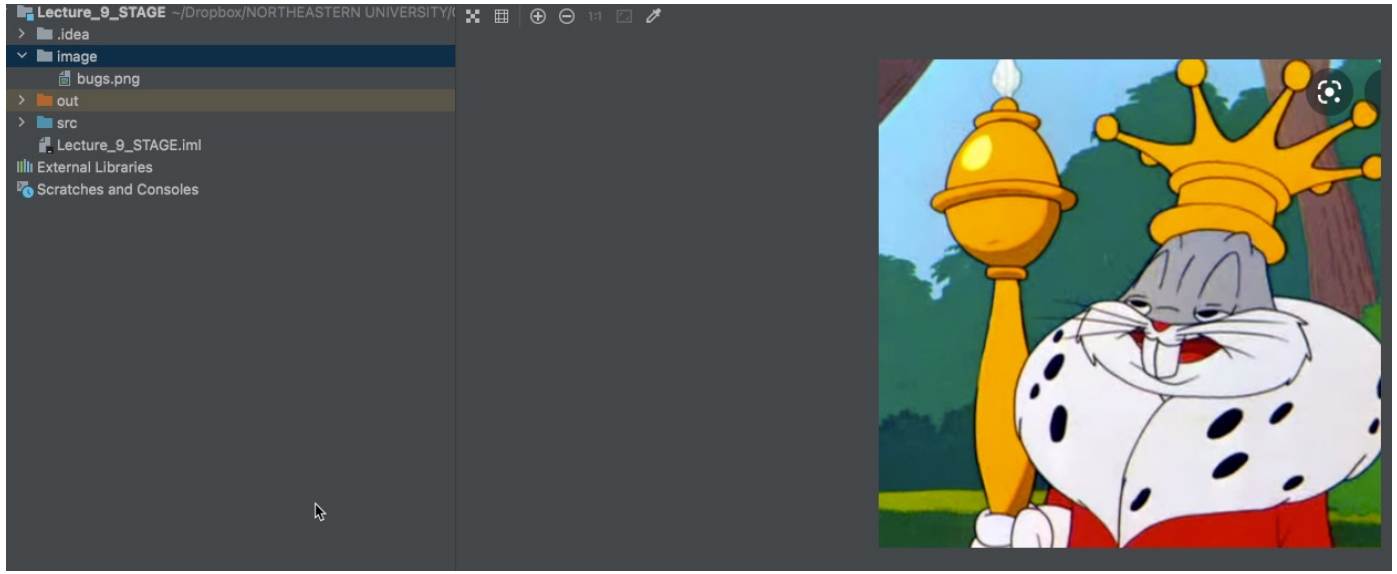
THE FILE CLASS

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

EXAMPLE: EXPLORE FILE PROPERTIES

We'll write a program that demonstrates how to manipulate files in a platform-independent way.

```
package my.io;

public class TestFile {
  public static void main(String[] args) {
    java.io.File file = new java.io.File( pathname: "image/bugs.png");
    System.out.println("Does it exist? " + file.exists());
    System.out.println("The file has " + file.length() + " bytes");
    System.out.println("Can it be read? " + file.canRead());
    System.out.println("Can it be written? " + file.canWrite());
    System.out.println("Is it a directory? " + file.isDirectory());
    System.out.println("Is it a file? " + file.isFile());
    System.out.println("Is it absolute? " + file.isAbsolute());
    System.out.println("Is it hidden? " + file.isHidden());
    System.out.println("Absolute path is " +
            file.getAbsolutePath());
    System.out.println("Last modified on " +
            new java.util.Date(file.lastModified()));
  }
}
```



TEXT I/O

- A File object encapsulates the *properties* of a file or a path but does not contain the methods for reading/writing data from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes.
  - These objects contain the methods for reading/writing data from/to a file.
    - We can read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

WRITING DATA USING PRINTWRITER

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output and Strings."

TRY-WITH-RESOURCES

Files and other resources should be closed after use. Similar to Python's with-open style of resource management, Java provides a try-with-resources syntax that automatically closes the files.

try (declare and create resources) {
  Use the resource to process the file;
}

```
with open('accounts.txt', mode='r') as accounts:
```

TRY-WITH-RESOURCES

```
4  public static void main(String[] args) throws Exception {
5    java.io.File file = new java.io.File( pathname: "scores.txt");
6    if (file.exists()) {
7      System.out.println("File already exists");
8      System.exit( status: 0);
9    }
10
11   try (
12           // Create a file
13           java.io.PrintWriter output = new java.io.PrintWriter(file);
14   ) {
15     // Write formatted output to the file
16     output.print("Peter J Parker ");
17     output.println(90);
18     output.print("Mary J Watson ");
19     output.println(85);
20   }
21 }
```

READING DATA USING SCANNER

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

USING SCANNER TO READ FROM FILES

```java
package my.io;

import java.util.Scanner;

public class ReadData {
  public static void main(String[] args) throws Exception {
    // Create a File instance
    java.io.File file = new java.io.File( pathname: "scores.txt");

    // Create a Scanner for the file
    Scanner input = new Scanner(file);

    // Read data from a file
    while (input.hasNext()) {
      String firstName = input.next();
      String mi = input.next();
      String lastName = input.next();
      int score = input.nextInt();
      System.out.println(
              firstName + " " + mi + " " + lastName + " " + score);
    }

    // Close the file
    input.close();
  }
}
```

REPLACING TEXT IN FILES

Temporary files are useful for replacing text.

Copy new data into the temp file, then replace original

Example: Class that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

java ReplaceText sourceFile targetFile oldString newString

```
public static void main(String[] args) throws Exception {
  // Check command line parameter usage
  if (args.length != 4) {
    System.out.println(
            "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
    System.exit( status: 1);
  }

  // Check if source file exists
  File sourceFile = new File(args[0]);
  if (!sourceFile.exists()) {
    System.out.println("Source file " + args[0] + " does not exist");
    System.exit( status: 2);
  }

  // Check if target file exists
  File targetFile = new File(args[1]);
  if (targetFile.exists()) {
    System.out.println("Target file " + args[1] + " already exists");
    System.exit( status: 3);
  }

  try (
          // Create input and output files
          Scanner input = new Scanner(sourceFile);
          PrintWriter output = new PrintWriter(targetFile);
  ) {
    while (input.hasNext()) {
      String s1 = input.nextLine();
      String s2 = s1.replaceAll(args[2], args[3]);
      output.println(s2);
```

USING SCANNER FOR CONSOLE INPUT

Scanner can be used to direct input from the console (System.in) for interactive I/O

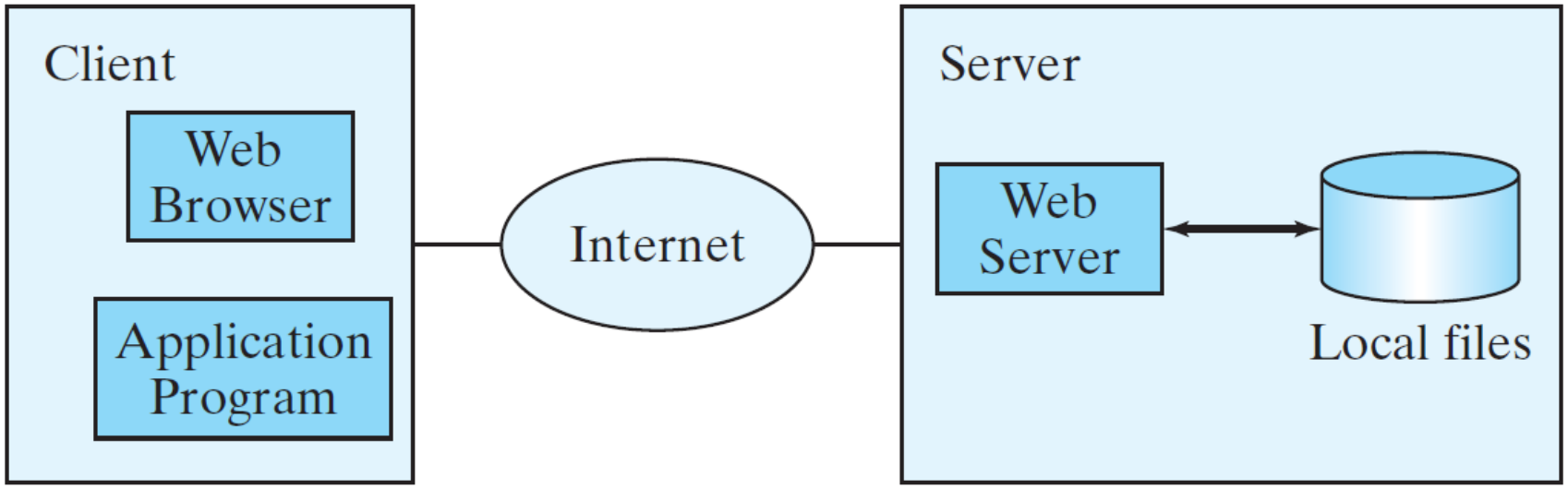
System.in is simply another text stream!

```
ReadFromConsole
/Users/keithbagley/Library/Java/JavaVirtualMachin
Hello! What is your name? Keith
How many courses are you taking this term? 3
3!...Wow Keith! That's a LOT!

Process finished with exit code 0
```

```java
package my.io;

import java.util.Scanner;

public class ReadFromConsole {
  public static void main(String[] args) {
    Scanner console = new Scanner(System.in);

    System.out.print("Hello! What is your name? ");
    String name = console.next();

    System.out.print("How many courses are you taking this term? ");
    int numCourses = console.nextInt();

    System.out.println(numCourses + "!...Wow " + name + "! That's a LOT!");
  }
}
```

READING DATA FROM THE WEB

Similar to reading data from a file on a local machine, we can read data from a file on the internet.



READING DATA FROM THE WEB

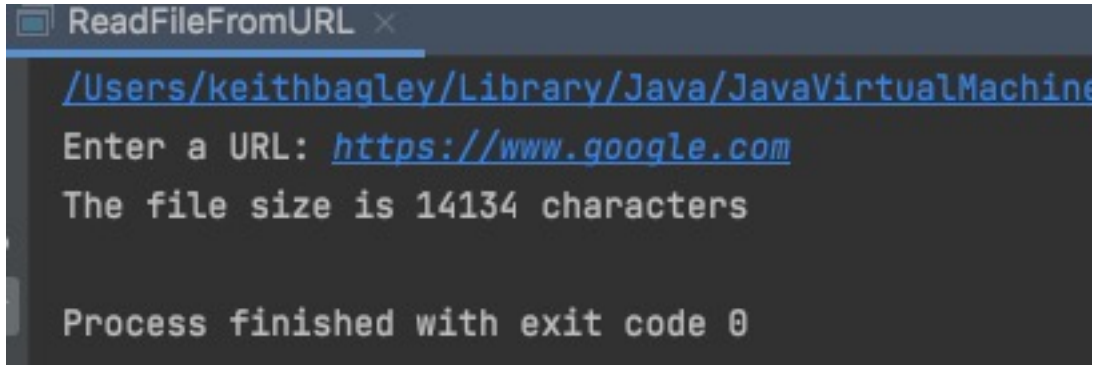
URL url = **new** URL("**www.google.com/index.html**");

After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

Scanner input = **new** Scanner(url.openStream());

READING DATA FROM THE WEB

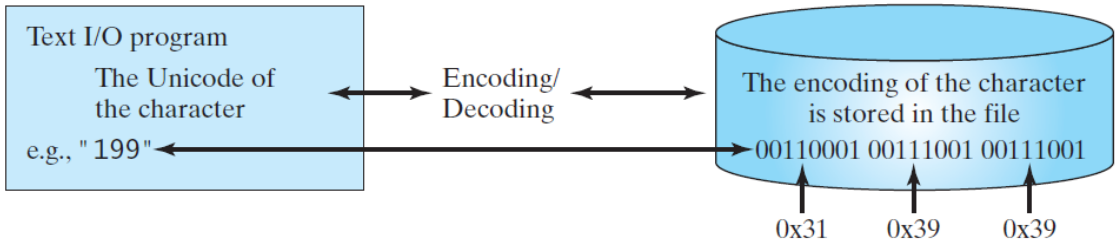
```java
package my.io;

import java.util.Scanner;

public class ReadFileFromURL {
  public static void main(String[] args) {
    System.out.print("Enter a URL: ");
    String URLString = new Scanner(System.in).next();

    try {
      java.net.URL url = new java.net.URL(URLString);
      int count = 0;
      Scanner input = new Scanner(url.openStream());
      while (input.hasNext()) {
        String line = input.nextLine();
        count += line.length();
      }

      System.out.println("The file size is " + count + " characters");
    }
    catch (java.net.MalformedURLException ex) {
      System.out.println("Invalid URL");
    }
    catch (java.io.IOException ex) {
      System.out.println("IO Errors");
    }
  }
}
```



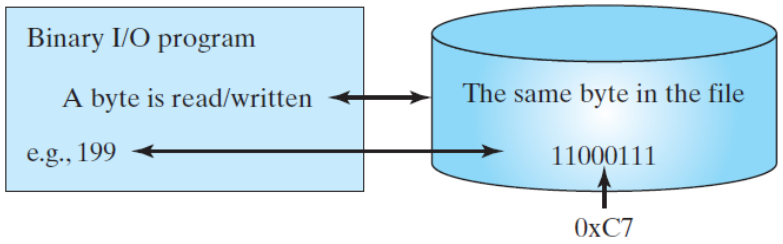
What is the output if we send the data in the variable line to the console?

BINARY I/O

Text I/O requires encoding and decoding. The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character. Binary I/O does not require conversions. When a byte is written to a file, the original byte is copied into the file.

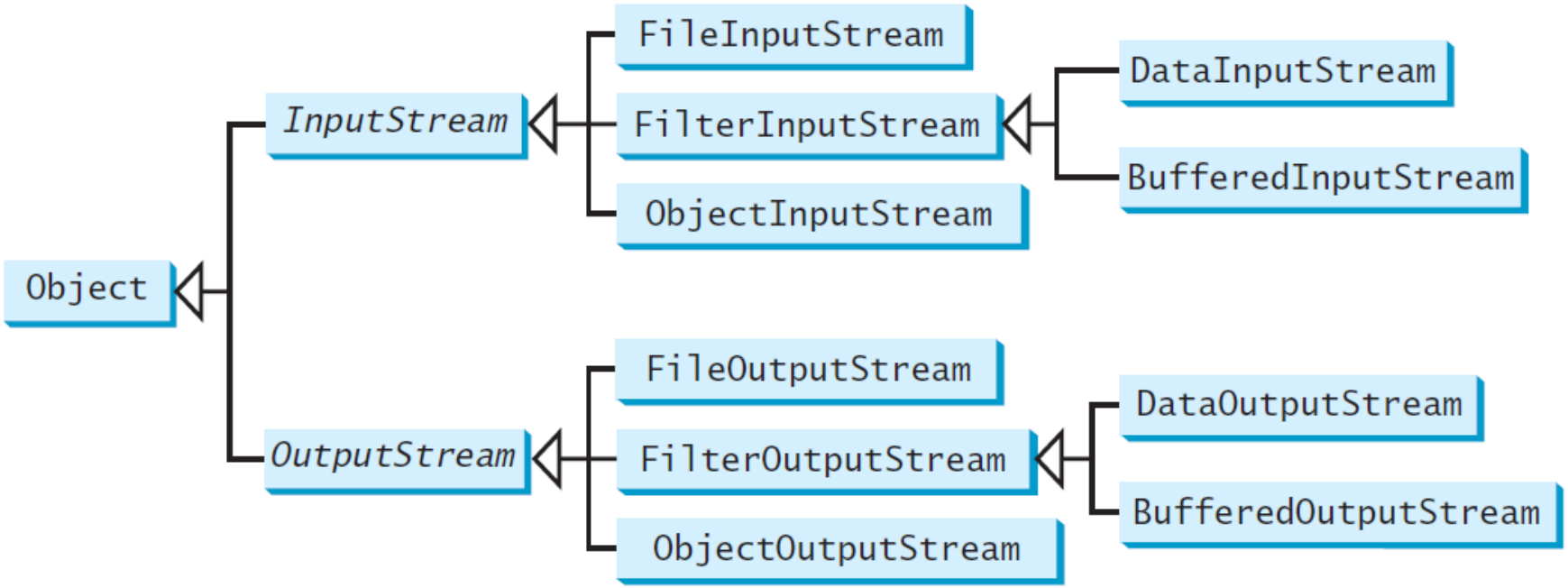


(a)



(b)

BINARY I/O CLASSES



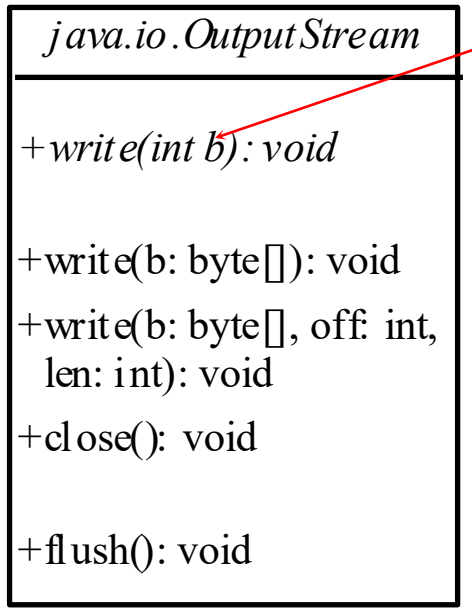
INPUTSTREAM

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

The value returned is a byte as an int.

OUTPUTSTREAM

The value is a byte as an int type.



Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.

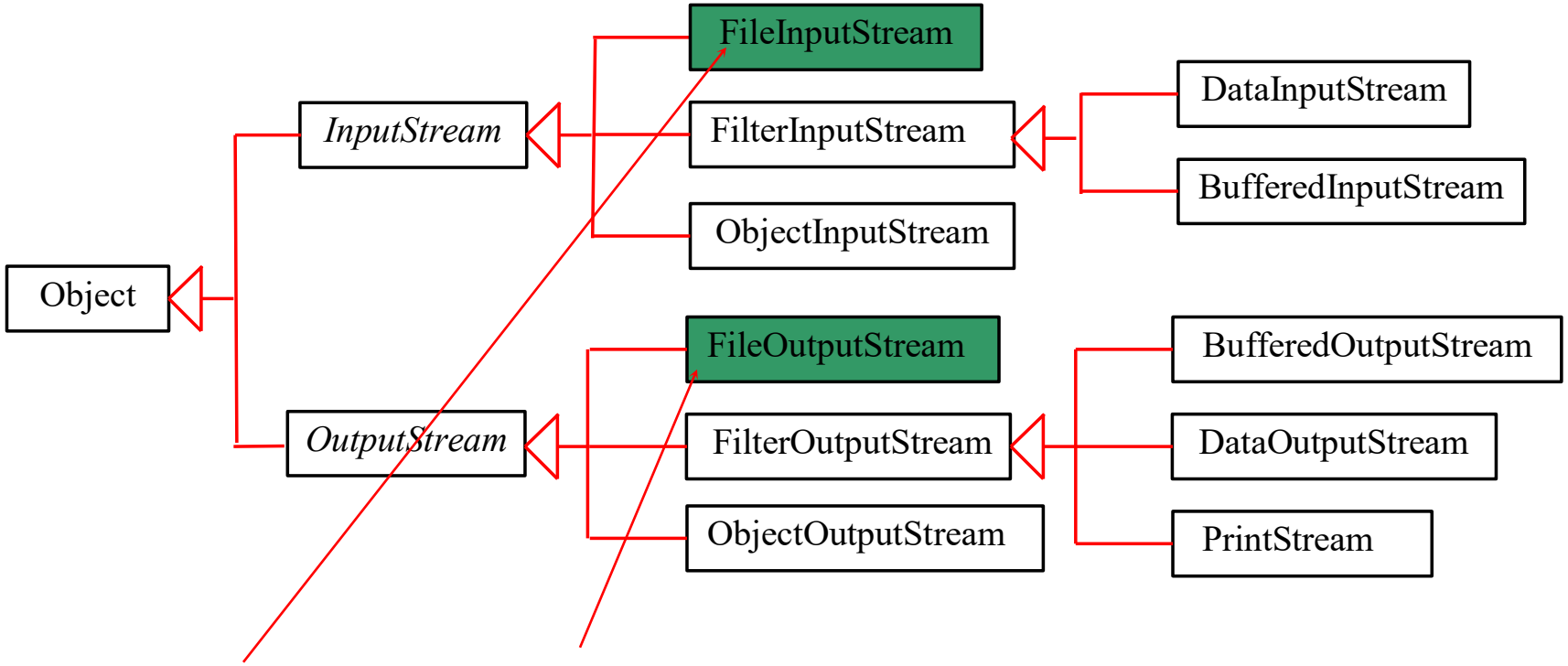
Writes all the bytes in array b to the output stream.

Writes b[off], b[off+1], …, b[off+len-1] into the output stream.

Closes this output stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.

FILEINPUTSTREAM/FILEOUTPUTSTREAM



FileInputStream/FileOutputStream associates a **binary** input/output stream with an external file.

FILEINPUTSTREAM

To construct a FileInputStream, use the following constructors:

public FileInputStream(String filename)

public FileInputStream(File file)

A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.

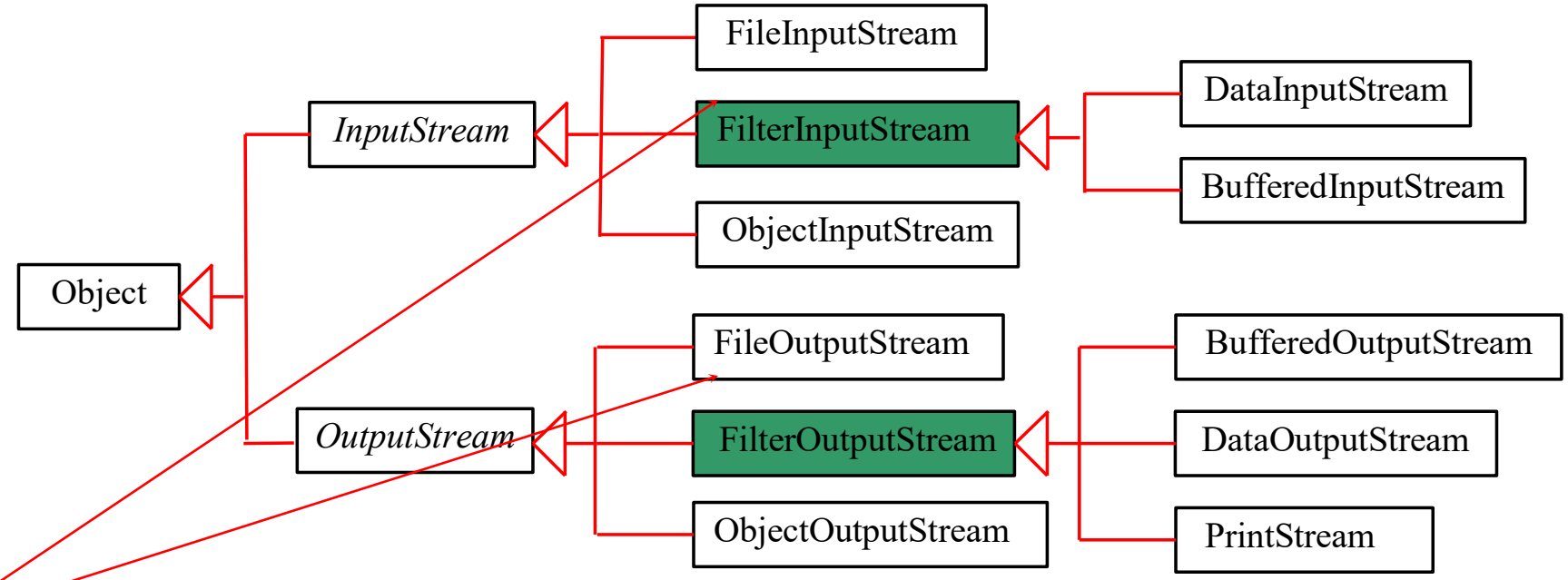
FILEOUTPUTSTREAM

To construct a FileOutputStream, use the following constructors:

```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)
```

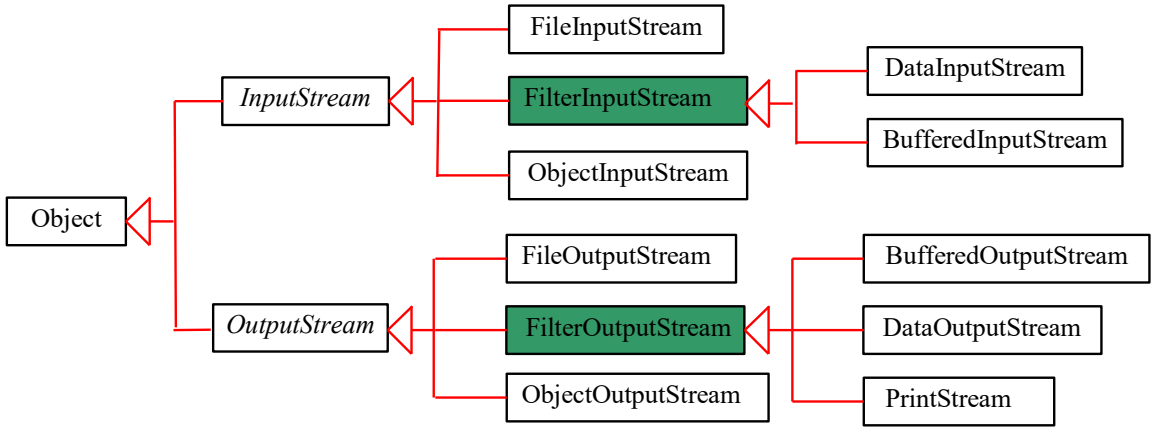
If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

FILTERINPUTSTREAM/FILTEROUTPUTSTREAM



Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. FilterInputStream and FilterOutputStream are the base classes for filtering data. When you need to process primitive numeric types, use DataInputStream and DataOutputStream to filter bytes.

FILTER STREAMS – PROTECTED CONSTRUCTORS



public class **FilterInputStream**
extends InputStream

A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class FilterInp
with versions that pass all requests to the contained input stream. Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

Since:
1.0

Field Summary

Fields

Modifier and Type	Field	Description
protected InputStream	in	The input stream to be filtered.

Constructor Summary

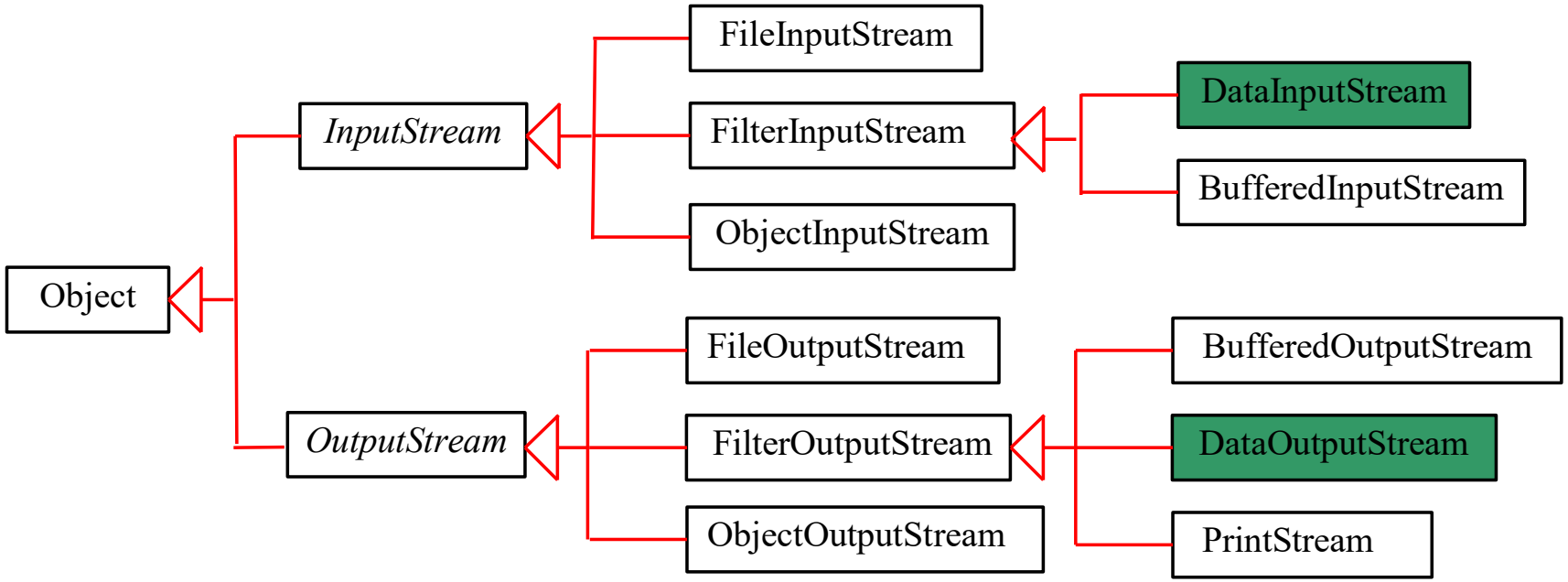
Constructors

Modifier	Constructor	Description
protected	FilterInputStream(InputStream in)	Creates a FilterInputStream by assigning the argument in to the field this.in so as to remember it for later use.

Filter streams are We covered this topic last week. In this case, the idea is that Filter streams form the basic protocol for developers to create their own I/O streams (or use the existing concrete classes). Not expecting clients to create a “raw” FilterInputStream but rather a DataInputStream or BufferedInputStream or other.

DATAINPUTSTREAM/DATAOUTPUTSTREAM

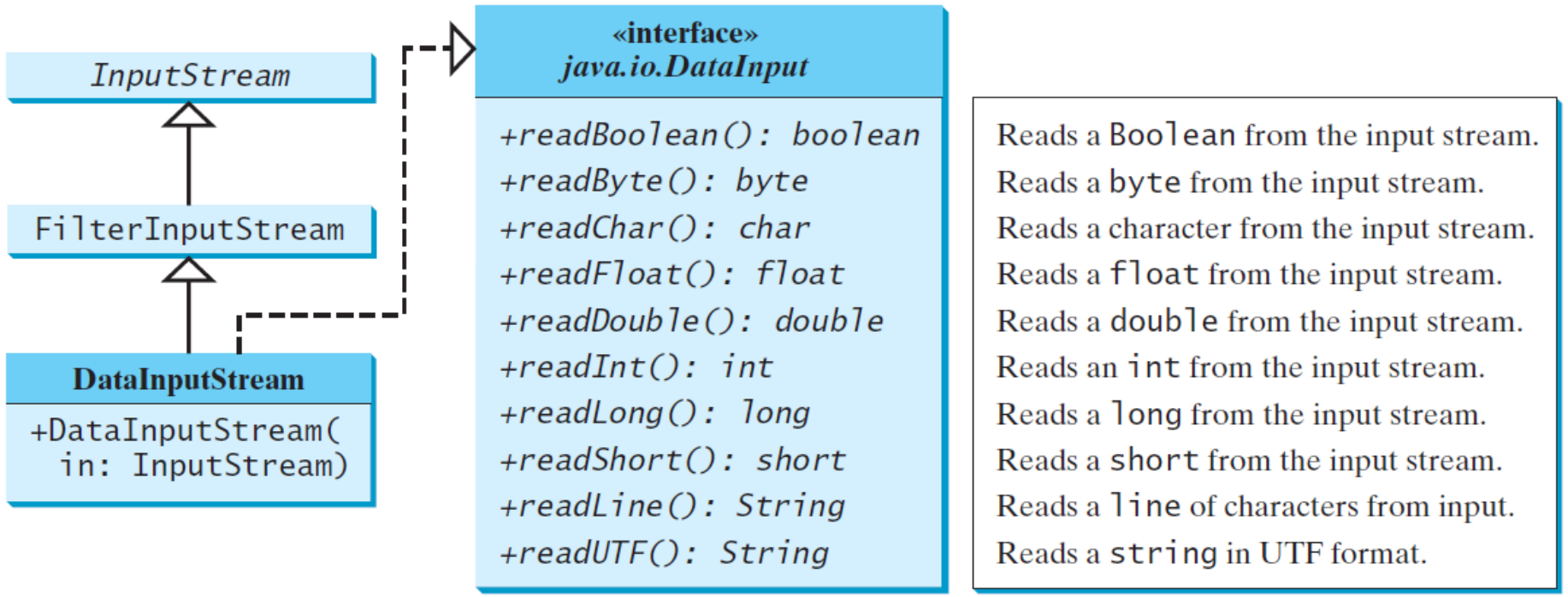
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

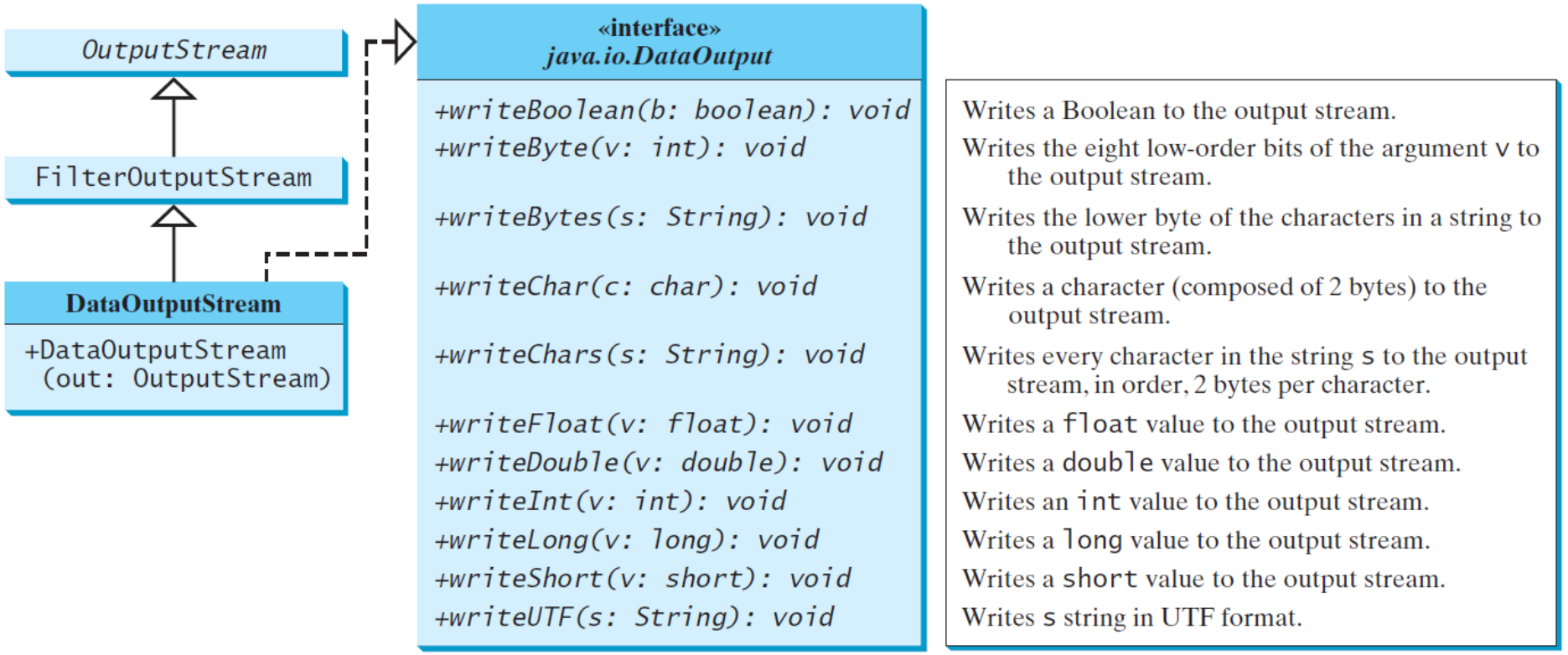
DATAINPUTSTREAM

DataInputStream extends FilterInputStream and implements the DataInput interface.



DATAOUTPUTSTREAM

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



CHARACTERS & STRINGS: BINARY I/O

A Unicode consists of two bytes. The writeChar(char c) method writes the Unicode of character c to the output. The writeChars(String s) method writes the Unicode for each character in the string s to the output.

What is UTF-8?

UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently. The ASCII character set is a subset of the Unicode character set. Since many applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character. The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes. ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.

DATAINPUTSTREAM/DATAOUTPUTSTREAM

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

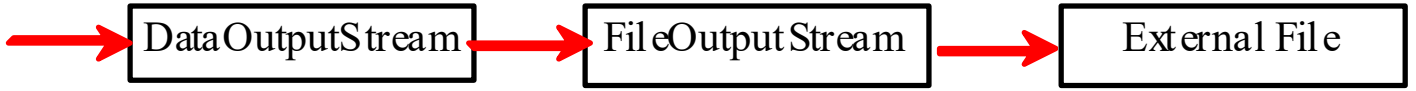
The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream infile =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

DATA PIPE LINE

int, double, string …

01000110011 …



int, double, string …

01000110011 …

USING DATAINPUT/OUTPUT STREAMS

```
import java.io.*;

public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file pinball.dat
          DataOutputStream output =
                  new DataOutputStream(new FileOutputStream( name: "pinball.dat"));
    ) {
      // Write student pinball scores to the file
      output.writeUTF( str: "Annie");
      output.writeDouble( v: 85.5);
      output.writeUTF( str: "Shriya");
      output.writeDouble( v: 185.5);
      output.writeUTF( str: "Lauryn");
      output.writeDouble( v: 105.25);
    }

    try ( // Create an input stream for file temp.dat
          DataInputStream input =
                  new DataInputStream(new FileInputStream( name: "pinball.dat"));
    ) {
      // Read student test scores from the file
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
    }
  }
}
```

```
Run:  TestDataStream
Annie 85.5
Shriya 185.5
Lauryn 105.25

Process finished with exit code 0
```

NOTES

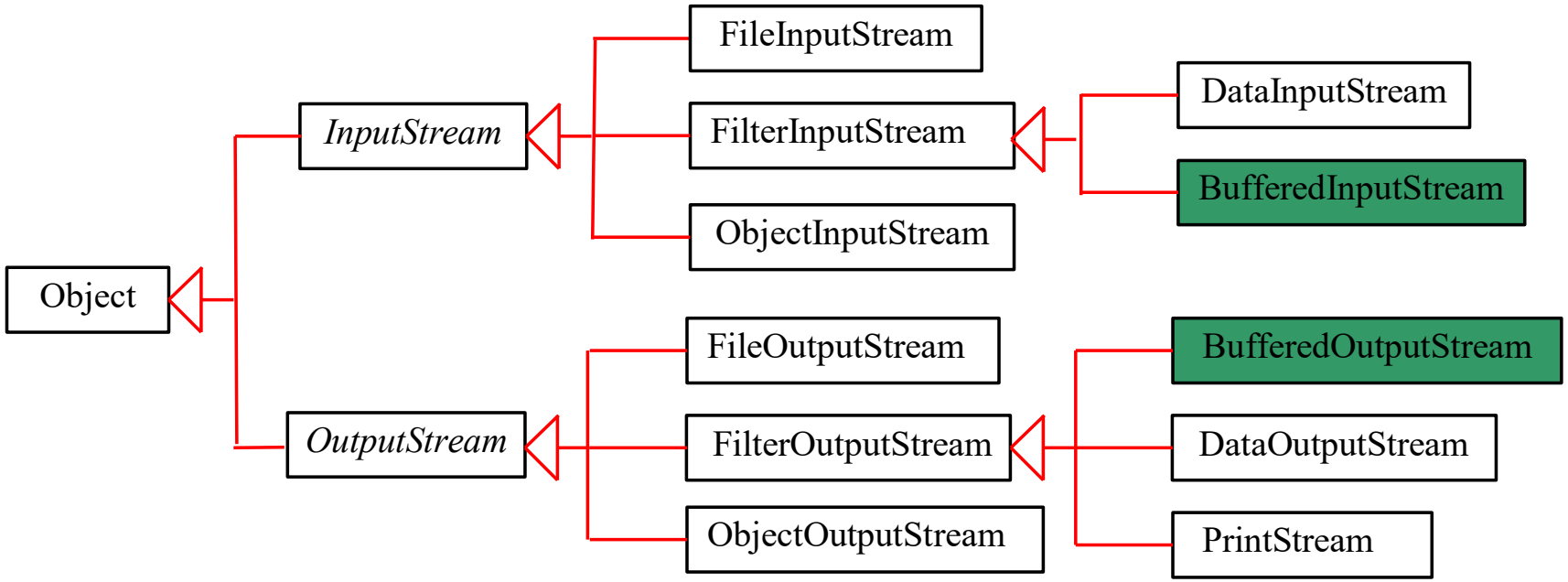
We must read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

TIP: If you keep reading data at the end of a stream, an EOFException will occur.

Use input.available() to check the stream for more data. input.available() == 0 indicates that it is the end of a file.

BUFFERED STREAMS

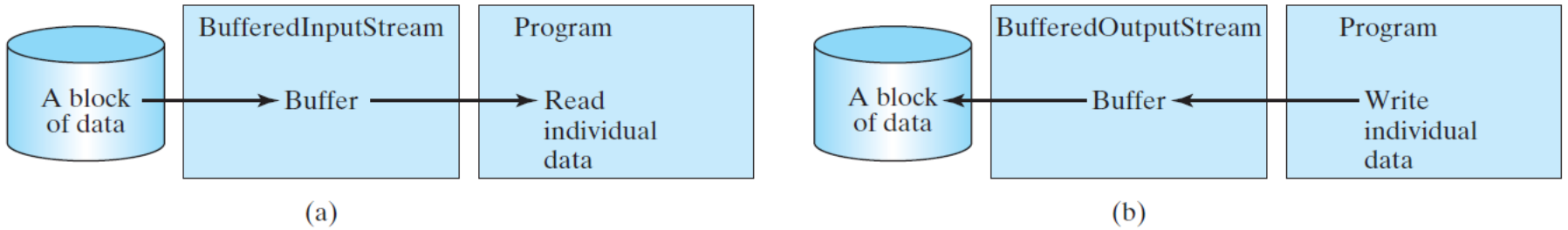
Buffers can speed up I/O



BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

BUFFERED STREAMS

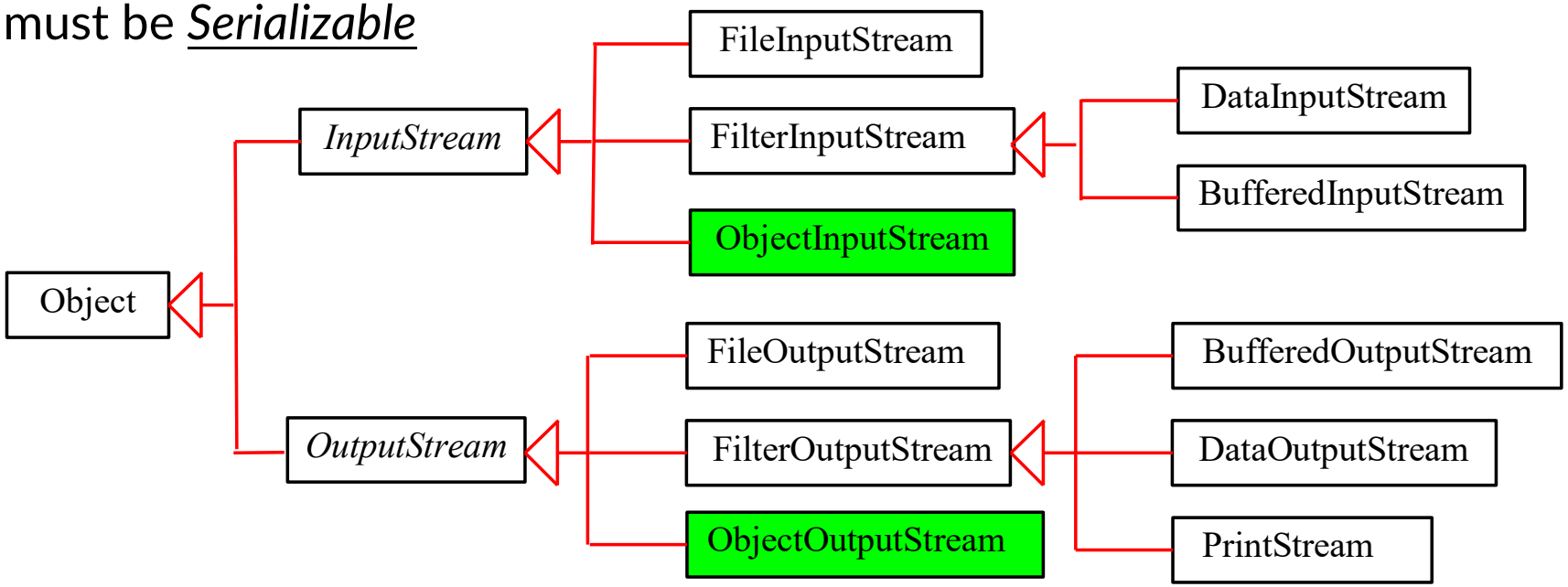
```
// Create a BufferedInputStream
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStreamr out, int bufferSize)
```



OBJECT I/O

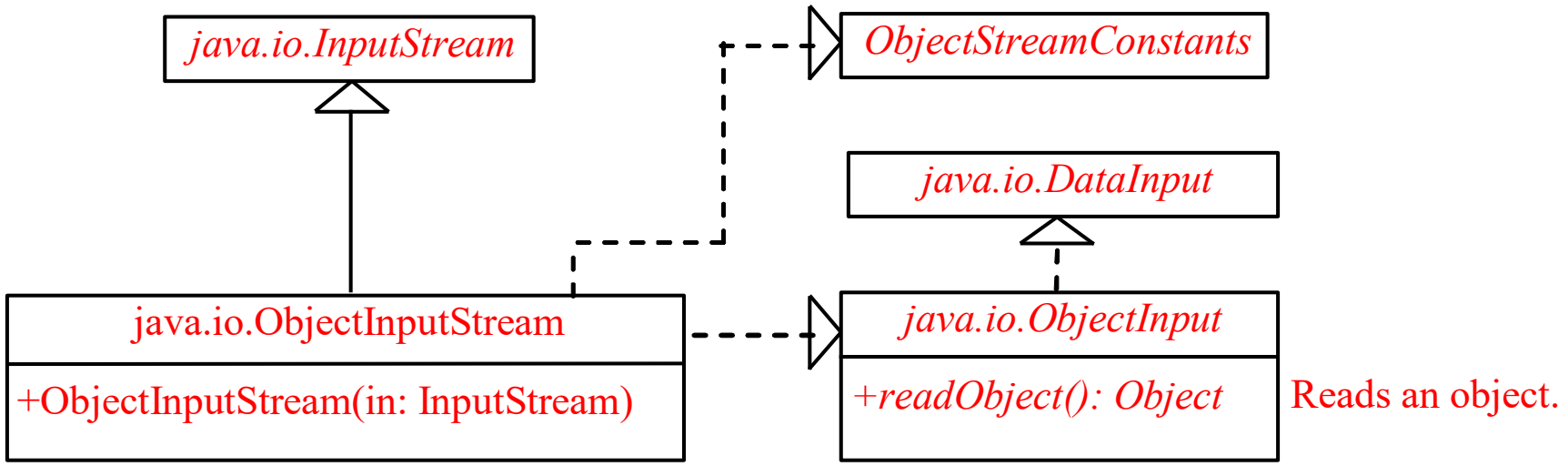
DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings. ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition for primitive type values and strings.

The classes must be *Serializable*



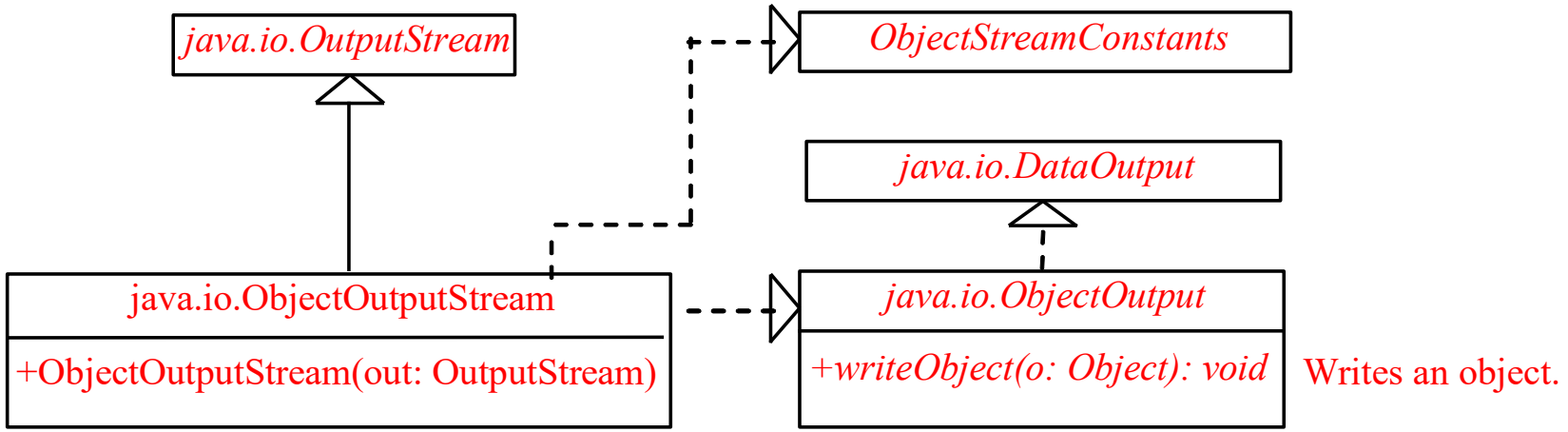
OBJECTINPUTSTREAM

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



OBJECTOUTPUTSTREAM

ObjectOutputStream extends OutputStream and implements ObjectOutput and ObjectStreamConstants.



USING OBJECT STREAMS

You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

USING OBJECT STREAMS

```java
package my.io;

import java.io.*;

public class TestObjectStream {
  public static void writeObject() throws IOException {
    try ( // Create an output stream for file object.dat
          ObjectOutputStream output =
                  new ObjectOutputStream(new FileOutputStream( name: "object.dat"));
    ) {
      // Write a string, double value, and object to the file
      output.writeUTF( str: "Circle");
      output.writeDouble( val: 1.0);
      output.writeObject(new SerialCircle( radius: 2));
    }
  }
```

```java
package my.io;

import java.io.Serializable;

public class SerialCircle implements Serializable {
  private double radius;

  public SerialCircle(double radius) throws IllegalArgumentException {
    if(radius < 0) {
      throw new IllegalArgumentException("Radius cannot be negative");
    }
    this.radius = radius;
  }

  public double getArea() { return Math.PI * Math.pow(this.radius, 2); }
}
```

Using a simpler version of our Circle class

THE `SERIALIZABLE` INTERFACE

Not all objects can be written to an output stream. Objects that can be written to an object stream is said to be *serializable*. A serializable object is an instance of the java.io.Serializable interface. The class of a serializable object must implement Serializable.

The Serializable interface is a marker interface. It has no methods, so you don't need to add additional code in your class that implements Serializable.

Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

THE `TRANSIENT` KEYWORD

If an object is an instance of Serializable, but it contains non-serializable instance data fields, the object cannot be serialized.

We can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream.

THE `TRANSIENT` KEYWORD, CONT.

Consider the following class:

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}
class A { } // A is not serializable
```

When an object of the Foo class is serialized, only variable v1 is serialized. Variable v2 is not serialized because it is a static variable, and variable v3 is not serialized because it is marked transient. If v3 were not marked transient, a java.io.NotSerializableException would occur.

SERIALIZATION CAUTIONS

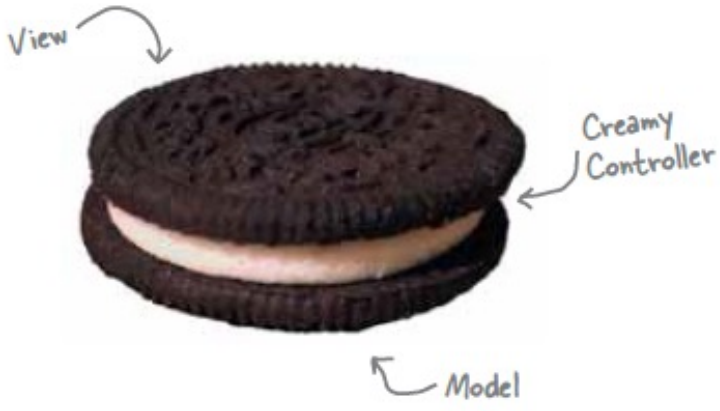
- Serialization is relatively “cheap” since Java has it inherently built in, but…
- Serialization is very brittle
  - As classes change/evolve, serialized data must be morphed or else it becomes unreadable
  - Graphs of serialized objects will encounter problems if even one of the classes is extended/modified
  - If certain data is transient, you must manage the persistence for that information separately
- Serialization can be acceptable for prototypes and small systems but generally speaking, other approaches (JSON, Object DBs, Object-Relational mapping, etc) are better options

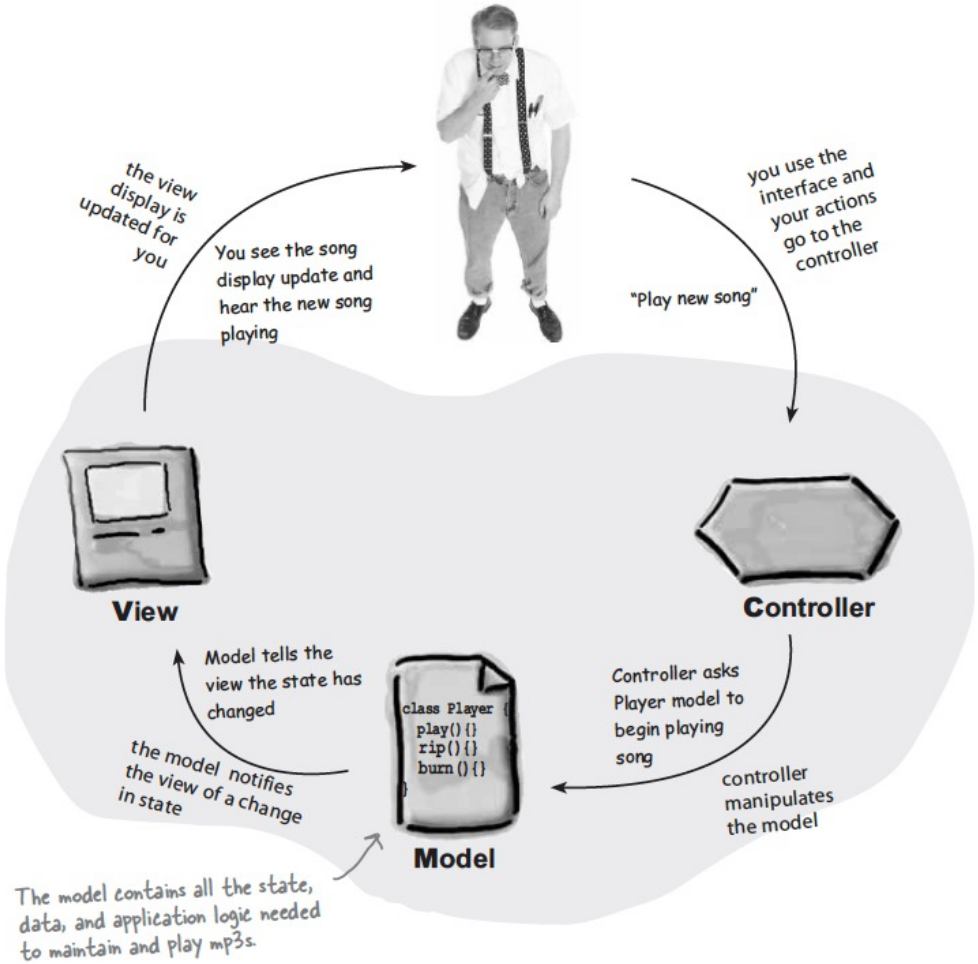
MVC OVERVIEW

MVC's a paradigm for factoring your code into functional segments, so your brain does not explode.

To achieve reusability, you gotta keep those boundaries clean

Model on the one side, View on the other, the Controller's in between.





Source: Head-First Design Patterns, 2014

MVC

- MVC is an architectural approach (or pattern) aimed at connecting an application's user interface to its core data and representation
- Architecture is concerned with how components fit together, and organization
- MVC makes use of a later topic called "Design Patterns", which are smaller, micro-architectural solutions for a given problem in context

MODEL

- Models implement the functionality for the domain objects
  - Does not care when functionality is used
  - Does not care how results are shown to the user

VIEW

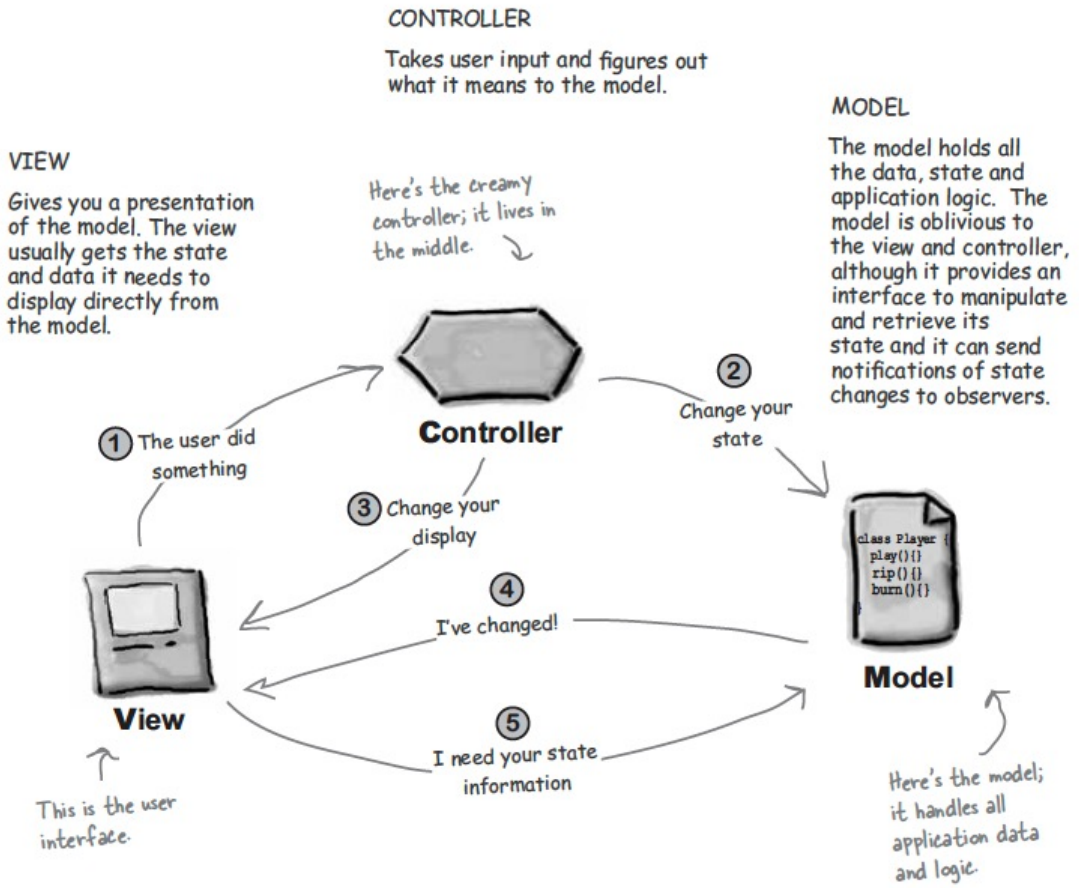
- Displays results to user
- Doesn’t care how the results are produced
- Doesn’t care when it shows the results (it’s told to update accordingly)

CONTROLLER

- Takes user input. Tells model what to do and view what to display
- Does not care how model implements functionality
- Does not care about screen layout, etc.

- *NB: In some simple systems, the View & Controller are merged*

MVC OVERVIEW



Source: Head First Design Patterns, 2014

1. **You're the user — you interact with the view.**
The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.

2. **The controller asks the model to change its state.**
The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. **The controller may also ask the view to change.**
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4. **The model notifies the view when its state has changed.**
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5. **The view asks the model for state.**
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

MVC ALTERNATIVES

- MVC is not the only approach!

Alternatives To MVC

HMVC - Hierarchical Model-View-Controller

This is quite similar to the MVC pattern, except that you can nest the triads together. So you can have one MVC structure for a page, one for navigation and a separate one for the content on the page. So the "top level" dispatches requests down to navigation and content MVC triads.

This makes structuring complex pages easier, since it allows you to create reusable widgets. But it brings all of the problems that MVC has, and solves none of them (it just adds complexity on top).

So HMVC doesn't really solve our problems...

MVVM - Model-View-ViewModel

The difference between MVC and MVVM is a lot more subtle. The basic premise is that in normal MVC, it's bad that the View is doing two jobs: presentation and presentation data logic. Meaning that there's a difference between actual rendering, and dealing with the data that will be rendered. So MVVM splits the MVC View in half. The presentation (rendering) happens in the View. But the data component lives in the ViewModel.

The ViewModel can interact with the rest of the program, and the View is bound to the ViewModel. This means that there's more of a separation between presentation and the application code that lives in the Model.

The controller isn't mentioned, but it's still in there somewhere.

Again, this solves some types of problems with MVC, but doesn't address any of our issues.

MVP - Model View Presenter

MVP is a bit different from MVC in implementation. Instead of having the Controller intercept user interaction and the View render data, MVP structures itself a bit differently. The View is responsible for passive presentation. Meaning that it doesn't bind to the Model, it just renders the data that it's given. But it also receives user interaction events (like the MVC controller). Basically, the View is the only thing that's exposed to the user.

The Presenter sits behind the View, and handles all of the functionality. When the View receives user interaction, it forwards it

Retrieved from: https://blog.ircmaxell.com/2014/11/alternatives-to-mvc.html

MODEL

- Key thing to keep in mind:
  - For MVC, the model is the application model, which in larger applications is not generally the “domain classes” that we’ve worked on thus far (unless we’re using Hierarchical MVC)
    - In some context when people speak of a “model” class, they are talking about the domain entities like Shape, Person, etc.
    - For MVC, groups of these domain assets plus the “application plumbing” for publishing and subscribing to application events is really the “M”.
    - However, for simpler applications, the model actually might be one of the domain classes we worked on.

See: http://www.cs.utsa.edu/~cs3443/mvc-example.html

Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!