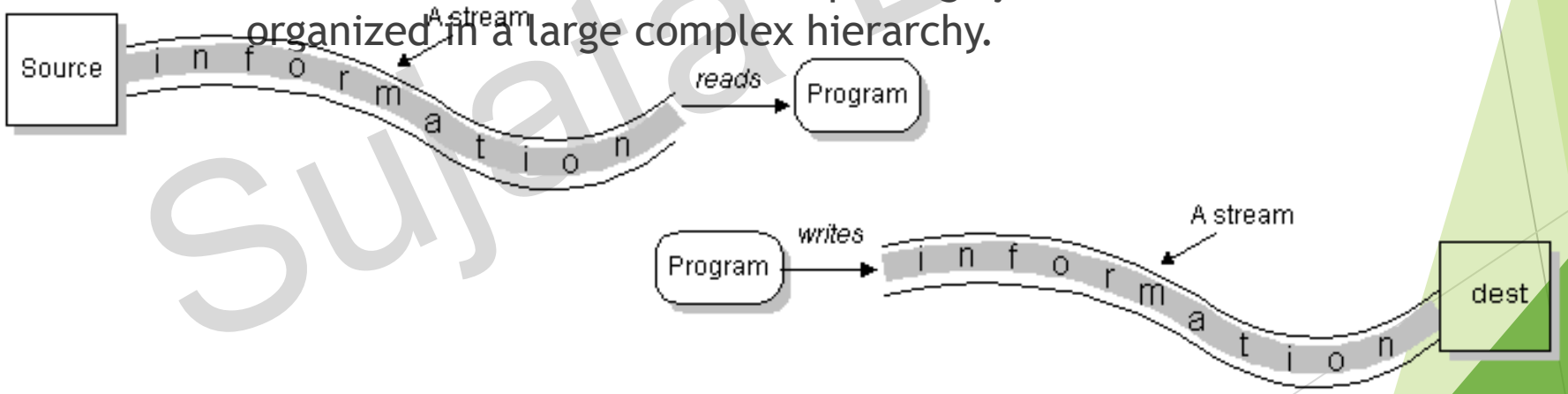
Files & IO

Overview

- At its lowest level, all Java I/O involves a stream of bytes either entering or leaving memory
- Packaged classes exist to make it easy for a program to read and write larger units of data.
- Low-level stream class object are used to handle byte I/O
- High-level stream class object will allow the program to read and write primitive data values and objects

File Classes

- Java views the data in files as a stream of bytes.
- A stream of bytes from which data are read is called an ***input stream***.
- A stream of bytes to which data are written is called an ***output stream***.
- Java provides classes for connecting to and manipulating data in a stream.
- The classes are defined in the package java.io and are organized in a large complex hierarchy.



File Class

- This is an abstract representation file and directory pathnames
  - Not used to read and write data
  - Used for searching and deleting of files, creating directories and working with path and making directories
  - has methods for getting file/directory info.
  - cannot be used to read or write to a file.
  - The path name in the code hence will depend on the underlying OS in which JVM is installed.
  - To make the code portable so that it works on all systems, static member separator defined in the File class can be used.
- To make a **File** object, there are three commonly used constructors `File`
  - `File file1 = new File("C:\\Data\\myFile.dat");`
  - public File(String directory, String filename)

File Class (Contd.)

- The following table list the methods in File class:

Return Type	Method Name	Description
boolean	createNewFile()	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.
boolean	exists()	Tests whether the file or directory denoted by this abstract pathname exists.
String	getName()	Returns the name of the file or directory denoted by this abstract pathname.
boolean	isFile()	Tests whether the file denoted by this abstract pathname is a normal file.
boolean	isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
long	length()	Returns the length of the file denoted by this abstract pathname.

File Class (Contd.)

- The following table list the methods in File class:

Return Type	Method Name	Description
String[]	list()	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
boolean	mkdir()	Creates the directory named by this abstract pathname.
boolean	renameTo(File dest)	Renames the file denoted by this abstract pathname.

Example: Creating a file

*Creates a new file named **`newFile.txt`**. If file exists then it deletes the file and creates a new one*

```
import java.io.*;
class FileOper{
public static void main(String str[]){
try{
File file = new File("newFile.txt");
if(file.exists())
file.delete();
boolean b=file.createNewFile();
System.out.println(b);
}catch(IOException e){ }
}}
```

What are streams

- An IO stream is an abstract term for any type of input or output device.
- There are 2 types of stream
  - Input stream to read data from a source. An input stream may be files, keyboard, console, other programs, a network, or an array!
  - Output stream to read data into a destination. An output stream may be disk files, monitor, a network, other programs, or an array
- Fundamentally stream may be
  - Byte stream : data read or written is in the form of byte
    or
  - Character stream: data read or written is in the form of character
- Stream is a sequence of data

Stream types in Java

- Character stream
  - Character stream writer classes
  - Character stream reader classes

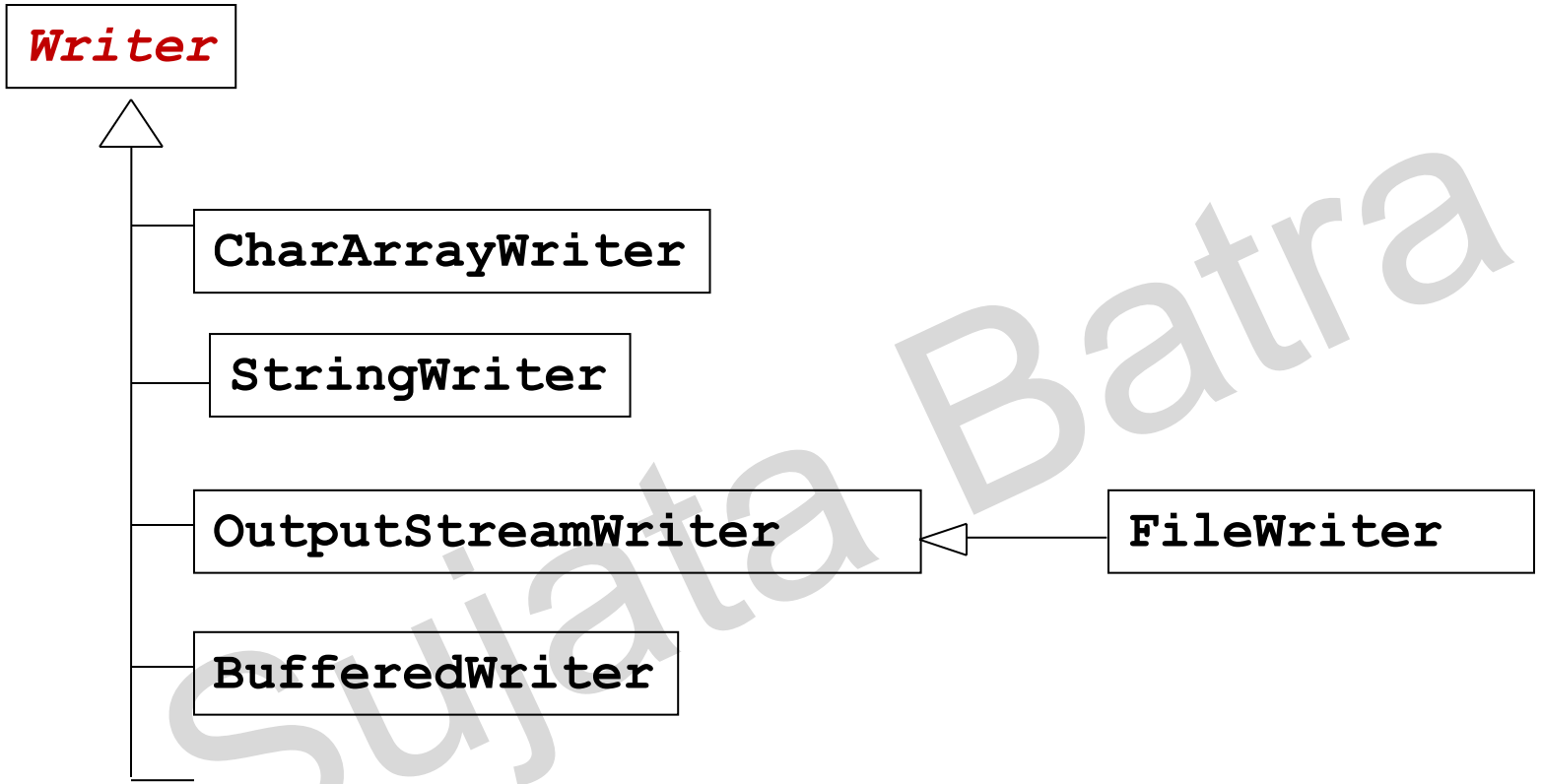
- Byte stream
  - Byte stream writer classes
  - Byte stream reader classes
  - Supports Serialization

Character stream

- As we are aware, the character in java is in the form of unicode.
- Character stream I/O automatically translates unicode to the local character set.
- At the top of the hierarchy we have **`Reader`** and **`Writer`** abstract classes are provided

- *First we will explore Writer classes*

er



Writer

```
void write(char[] cbuf)
void write(char[] cbuf, int off, int len)
void write(String str)
void write(String str, int off, int len)
void write(int c)
void close()
void flush()
```

- It is an abstract class for writing to character streams. Methods are to write or append a character or character array or strings and flush.
- All the methods throw **`IOException`**.

FileWriter

FileWriter inherits from **OutputStreamWriter.**

Constructors:

- **FileWriter(File file)**
- **FileWriter(String fileName)**

Creates an instance of **FileWriter** and also the file if it does not exist. If it exists it overwrites.

If the file exists but is a directory rather than a regular file **IOException** is thrown

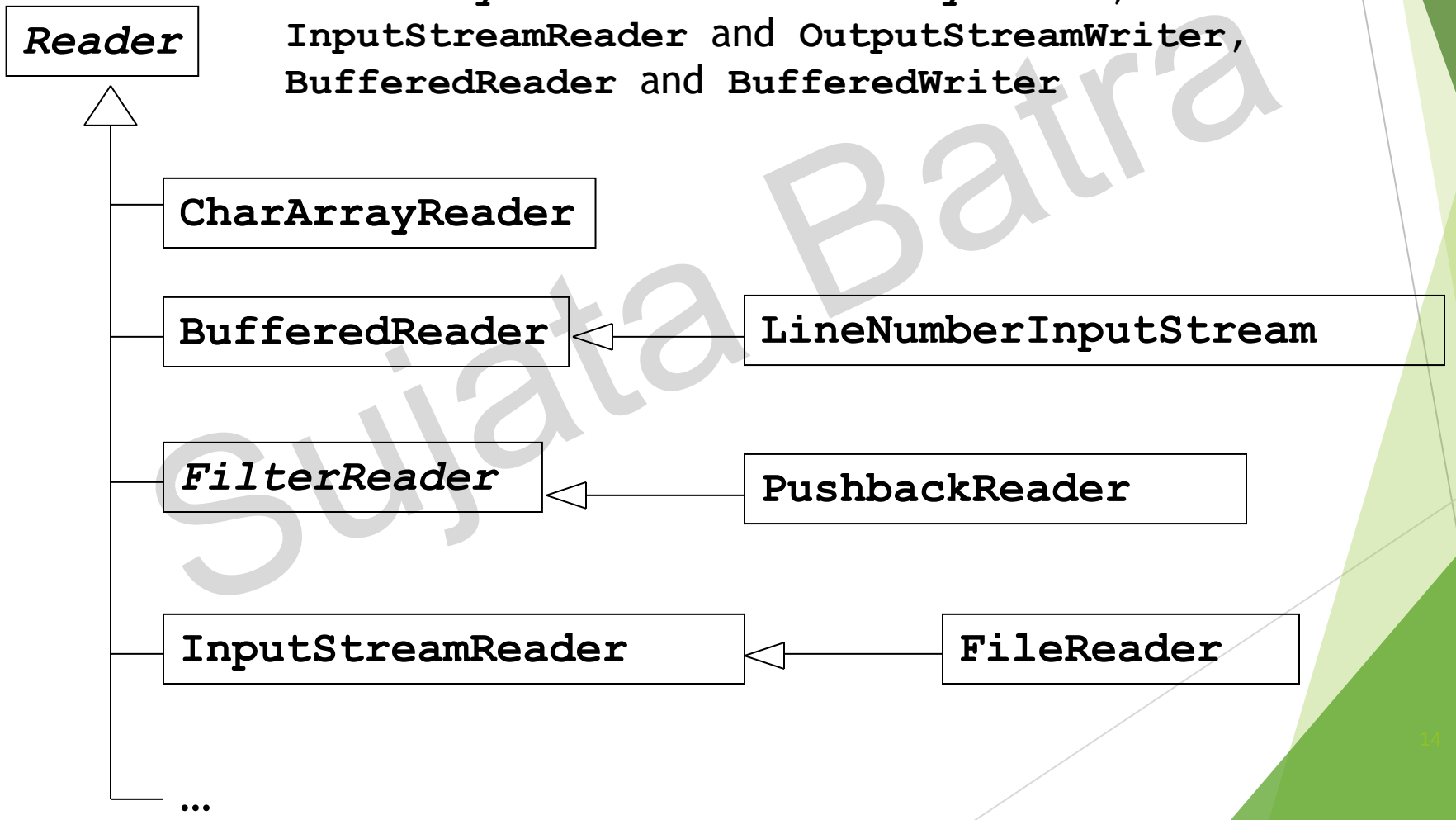
- **FileWriter(File file, boolean append)**
- **FileWriter(String fileName, boolean append)**

Provide same functionalities as that of the previous constructor, if **append** is **true**, then data will be written to the end of the file rather than the beginning.

All constructors throw **IOException**

Hierarchy of character stream reader

Reader class hierarchy is very similar to that of the Writer hierarchy. **`FileReader`** and **`FileWriter`** form pairs where one can be used for reading the text which other has written. Similarly we have **`CharArrayReader`** and **`CharArrayWriter`**, **`InputStreamReader`** and **`OutputStreamWriter`**, **`BufferedReader`** and **`BufferedWriter`**



Reader

Reader is an abstract class for reading character streams.

Methods:

```
void close()
int read()
int read(char[] cbuf, int off, int len)
void mark(int readAheadLimit)
void reset()
```

- Marks the current position in the stream. When **reset()** is called after **mark()** the file pointer is positioned to the marked position.
- **readAheadLimit** is used to specify how many characters can be read further from the marked position so as to retain the marked position. If characters read is greater than what is specified in **readAheadLimit,** then calling reset does not position the file pointer in the marked position.

`long skip(long n)`

`boolean markSupported()`

`mark()` and `reset()` are optional methods that is not all implementing class need to provide the implementation for **`mark()`** and **`reset()`.** Therefore before they are used we must test if they are supported by the implementing class using **`markSupported()`**

All of the methods except **`markSupported()`** throw **`IOException`**.

FileReader

FileReader is subclass of **InputStreamWriter**

This class is used to read from a text file.

Constructors:

```
FileReader(File file) throws FileNotFoundException
```

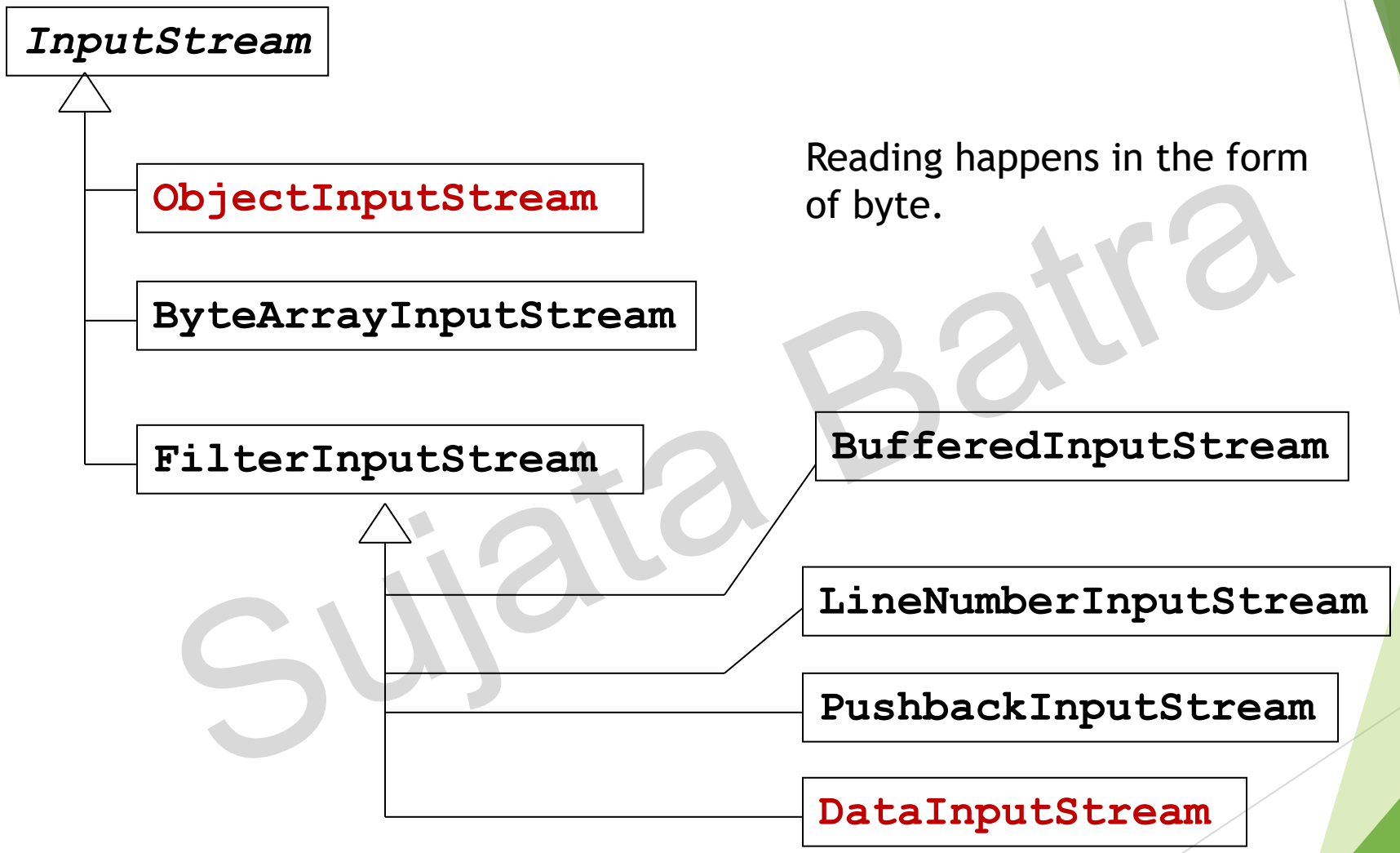
```
FileReader(String fileName) throws FileNotFoundException
```

Either filename can be specified as a String or File object is passed to the **FileReader** constructor.

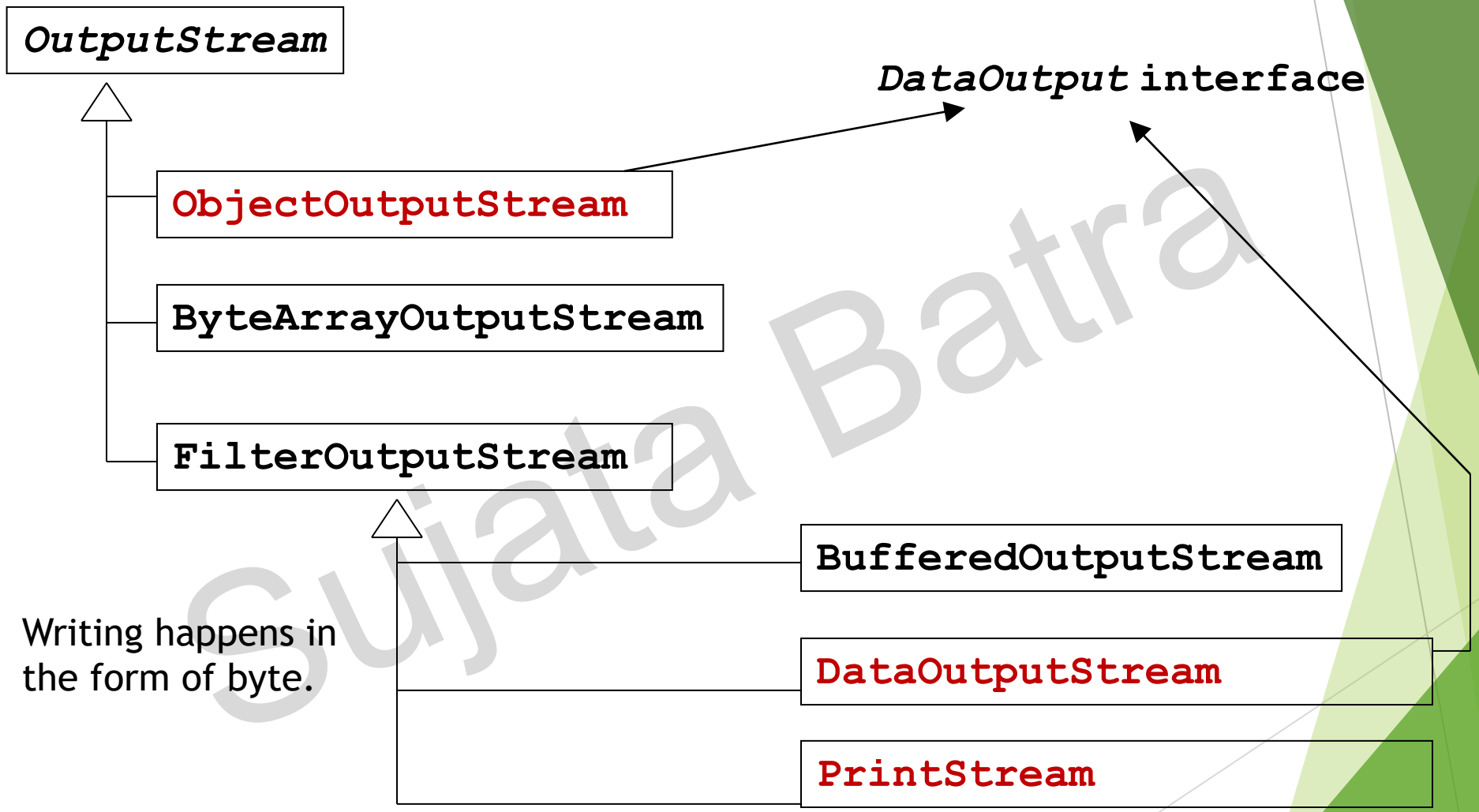
If the file specified by the name does not exist a **FileNotFoundException** is thrown

FileNotFoundException is a subclass of **IOException**

Hierarchy of byte stream



Reading happens in the form of byte.



Writing happens in the form of byte.

Example: using byte stream

- *Classes which are not in red in the last 2 slides are similar/parallel to character stream classes . Only difference is in place of char array we have byte array. So we end with an example for these classes.*
- *Example below copies the content of one file into another file.*

```
import java.io.*;
public class CopyFile {
public static void main(String[] args) throws
IOException {
File file1 = new File("D:"+File.separator+"read.txt");
File file2 = new
File("D:"+File.separator+"write.txt");
FileInputStream fin=null;
FileOutputStream fout=null;
```

```
try    {
fin = new FileInputStream(file1);
fout = new FileOutputStream (file2);
byte fileContent[] = new byte[(int)file1.length()];
fin.read(fileContent);
String strFileContent = new String(fileContent);
 fout.write(fileContent);
  System.out.println(strFileContent);
    }
    catch(FileNotFoundException e)    {
      System.out.println("File not found" + e);
    }
    catch(IOException ioe)    {
      System.out.println("Exception while reading the
file " + ioe);
    }
finally{
if(fin!=null)fin.close();
if(fout!=null)fout.close();}}}
```

DataInputStream and DataOutputStream

- A data input stream and data output stream lets an application read and write primitive Java data types from an underlying input stream and output stream in a machine-independent way.
- An application uses a data output stream to write data that can later be read by a data input stream and vice versa.

DataOutputStream methods

Inherited from **OutputStream**

```
void write(int b)
void write(byte[] b, int off, int len)
void writeXxx(xxx v)
```

Inherited from **InputStream**

DataInputStream methods

```
int read(byte[] b)
int read(byte[] b, int off, int len)

xxx readXxx()
```

where xxx can be byte, short, int, long, char, float, double.

All the above methods throw **IOException**

Example: using DataInputStream and DataOutputStream

Example shows how primitive can be written and read using **DataOutputtream** and **DataInputStream**

```
import java.io.*;
class Test{
public static void main(String[] st) throws Exception{
DataOutputStream out= new DataOutputStream(new
FileOutputStream("a.txt"));
int i=10;
double d= 12.3;
out.writeInt(i);
out.writeDouble(d);
out.close();
DataInputStream in= new DataInputStream(new
FileInputStream("a.txt"));
System.out.println( in.readInt() );
System.out.println( in.readDouble() );
in.close();
}  }
```

Serialization

- What is Serialization
- What is preserved when an object is serialized
- *transient* keyword
- Process of serialization
- Process of deserialization
- Version control

Serializing

- Creating the sequence of bytes from an object ,and Recreating the object from the above generated bytes
- Ability to read or write an object to a stream
  - Process of "flattening" an object
- Used to save object to some permanent storage
  - Its state should be written in a serialized form to a file such that the object can be reconstructed at a later time from that file
- Used to pass on to another object via the *OutputStream* class
  - Can be sent over the network

To use serialization

- Most Java classes are serializable
- Classes need to implement the serializable interface.
  - *Serializable* interface is marker interface
  - Class should also provide a default constructor with no args
- Objects of some system-level classes are not serializable
  - Because the data they represent constantly changes
  - Reconstructed object will contain different value anyway
- A *NotSerializableException* is thrown if you try to serialize non-serializable objects

To use serialization

- Objects are written using an ObjectOutputStream and read using an ObjectInputStream.
- Only the object's data are preserved, Methods and constructors are not part of the serialized stream ,the class information is included
- Marking a field with the *transient* keyword
  - The *transient* keyword prevents the data from being serialized
  - All non-transient fields are considered part of an object
- Have access to the no-argument (or default) constructor of its first nonserializable superclass (or supersuperclass, supersupersuper class)
- Serializability is inherited

Process of Serialization and Deserialization

- **public final void writeObject(Object obj) throws IOException**
  - where, *obj* is the object to be written to the stream
- **public final Object readObject() throws IOException, ClassNotFoundException**
  - *readObject* method of the *ObjectInputStream* class
  - When an object is deserialized, its constructors are **not** called.
- The *Object* type returned should be type casted to the appropriate class name before methods on that class can be executed

Serialization

```
public static void main(String[] args)
   {
     try {
          ObjectOutputStream out =
                 new ObjectOutputStream (
                   new FileOutputStream(new File("abc.ser"));

   Employee eObj = new Employee(100,"ramesh",4500);

     out.writeObject(eObj);

     System.out.println("Object Serialized in abc.ser");

     } catch (FileNotFoundException e) {
          e.printStackTrace();
     } catch (IOException e) {
          e.printStackTrace();
     } }
```

DeSerialization

```
public static void main(String[] args) {
try
{
ObjectInputStream inObj=

   new ObjectInputStream(
   new FileInputStream("abc.ser"));

      Employee eObj = (Employee) inObj.readObject();

      System.out.println(eObj.getEmpId());
}
catch(ClassNotFoundException e)
{
e.printStackTrace();
}
catch(Exception e)
{
e.printStackTrace();
}
```

What if the serialized object has a reference to another object

- In case we have a reference to another object and the referenced class also implements Serializable, that the referenced class will be automatically serialized.
- In case the referenced class does not implement Serializable interface, than there will be runtime error in serializing class. To avoid the error just make reference variable transient.
- If the referenced class can not implement Serializable interface and we still want to persist its states, then we need to override writeObject and readObject method and they will be called during serialization and deserialization .

Inheritance in Java Serialization

- In case super class is Serializable than all its subclasses will be serializable by default.
- In case super class is not Serializable than to serialize the subclass's object we must implement serializable interface in subclass explicitly. In this case the superclass must have a no-argument constructor in it.
- To prevent subclass from being serialized we must implement writeObject() and readObject() method and need to throw NotSerializableException from these methods.

Control the java serialize versioning

- Every Serializable class contains a serialVersionUID.
- This long value is calculated by default from the name and signature of the class and its data members and methods using Sercure Hash algorithm..
- For backward compatibility, you can specify your own public static final long serialVersionUID
- Prior to modify the class, you can use serialver tool to find out the old version ID:
- > serialver app.Rectangle
- It is importent to make sure the changes are both forward and backward compatible.
- Add, remove or modify the methods are normally compatiple, but you should consider the consequences of the change in your business logic.
- To add new data members are compatiple. This means that the new class has to deal with missing data for the new data members, as the old class will ignore unknown data members of the new object.
- Removing fields is incompatiple. (The old class could might trust on non-default values from the fields that are missing in new objects.)
- You may need to implement a customize handling using the readObject() methods to ensure compatibility.
- Or you may need to use a full serializable control implementing the java.io.Externalizable interface to ensure compatibility.

Cloning

- The object cloning is a way to create exact copy of an object.
- By default, java cloning is ‘field by field copy’ i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked.
- JVM when called for cloning, do following things:
  - For primitive data type members of the class a completely new copy of the object will be created and the reference to the new object copy will be returned.
  - If the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.

A clone object should follow basic characteristics

- a.clone() != a, which means original and clone are two separate object in Java heap.
- a.clone().getClass() == a.getClass() and clone.equals(a), which means clone is exact copy of original object,but these are not absolute requirements.

Java infrastructure for cloning

- You must implement Cloneable interface or else you will get CloneNotSupportedException.
- You must override clone() method from Object class.
- Object class has the clone method (protected) you cannot use it in all your classes.
- The class which you want to be cloned should implement clone method and overwrite it.
- It should provide its own meaning for copy or to the least it should invoke the super.clone().

Shallow Cloning

- This is default implementation in java.
- When you invoke the super.clone() then you are dependent on the Object class's implementation and what you get is a shallow copy
- In overridden clone method, if you are not cloning all the object types (not primitives), then you are making a shallow copy.

Deep cloning

- A clone which is independent of original and making changes in clone should not affect original.