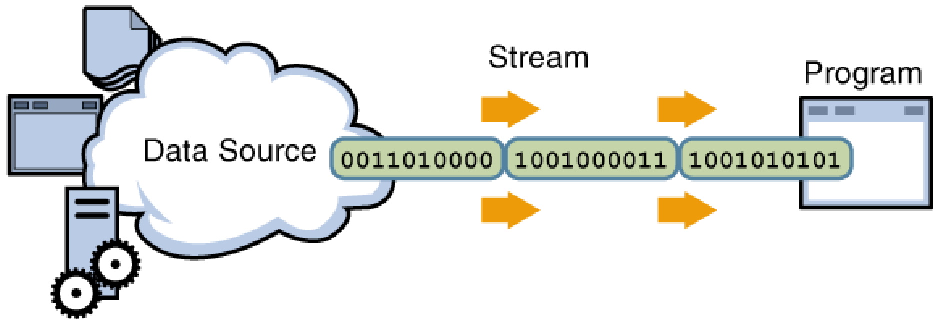
Java IO

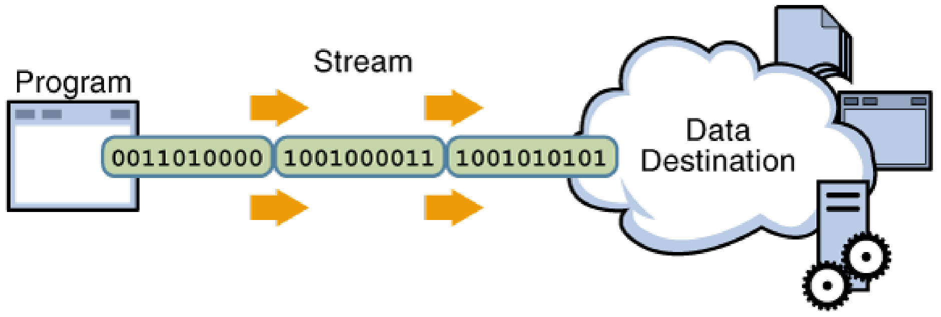
IO Streams

- Pre-defined Streams
- System properties
- Byte Stream
  - InputStream
    - DataInputStream
    - FileInputStream
  - OutputStream
    - DataOutputStream
    - FileOutputStream
  - Buffered IO
- File class
- Character Stream
  - Reader and Writer classes
- Object Serialization & DeSerialization

Java IO Streams

- Java programs perform I/O through streams.
- A stream is:
  - an abstraction that either produces or consumes information
  - linked to a physical device by the Java I/O system
- **A Stream is a channel or a path of communication between programs and source/destination of data.**
  - *Streams hide the details of what happens to the data inside the actual I/O devices.*
- Streams can read data from blocks of memory, files and network connections etc., Similarly, streams can write data to blocks of memory, files , network connections etc.





The Java Input and Output system is designed to make java applications device independent

Java IO Streams

- We have two types of streams:
  - Byte stream
  - Character stream

Java stream classes are defined in the **java.io** package.

- Byte streams:
  - provide a convenient means for handling input and output of bytes
  - are used for reading or writing binary data
- Character streams:
  - provide a convenient means for handling input and output of characters
  - use **Unicode**, and, therefore, can be internationalized

Note : Normally, the term *stream* refers to a byte stream. The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

The Predefined Streams

- System class of the java.lang package contains three predefined stream variables, in, out, and err.

- These variables are declared as public and static within System:
  - **System.out** refers to the standard output stream which is the console.
  - **System.in** refers to standard input, which is the keyboard by default.
  - **System.err** refers to the standard error stream, which also is the console by default.

Difference between System.out and System.err

System.out sends the output to the standard output stream, which is console by default.

System.err sends the output to the standard error stream, which also happens to be console by default.

The reason behind having two separate streams for output and error is that the standard output should be used for regular program outputs while standard error should be used for error messages.

System Properties

System Properties provide information about local system configuration. When the Java Virtual Machine starts, it inserts local System Properties into a System properties list.

We can use methods defined in System class to access or change the values of these properties.

```
public static Properties getProperties()
public static String getProperty(String key)
public static void setProperties(Properties properties)
```

System Properties (Contd.).

Some of the Important Properties are listed below :

Key	**Description of Associated Value**
java.version	Java Runtime Environment version
java.home	Java installation directory
java.class.path	Java class path
os.name	Operating system name
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

System.getProperties()

public static Properties getProperties()

The **System.getProperties()** method returns an object of the type Properties.

You can use this method to list all the System Properties.

public static String getProperty(String key)

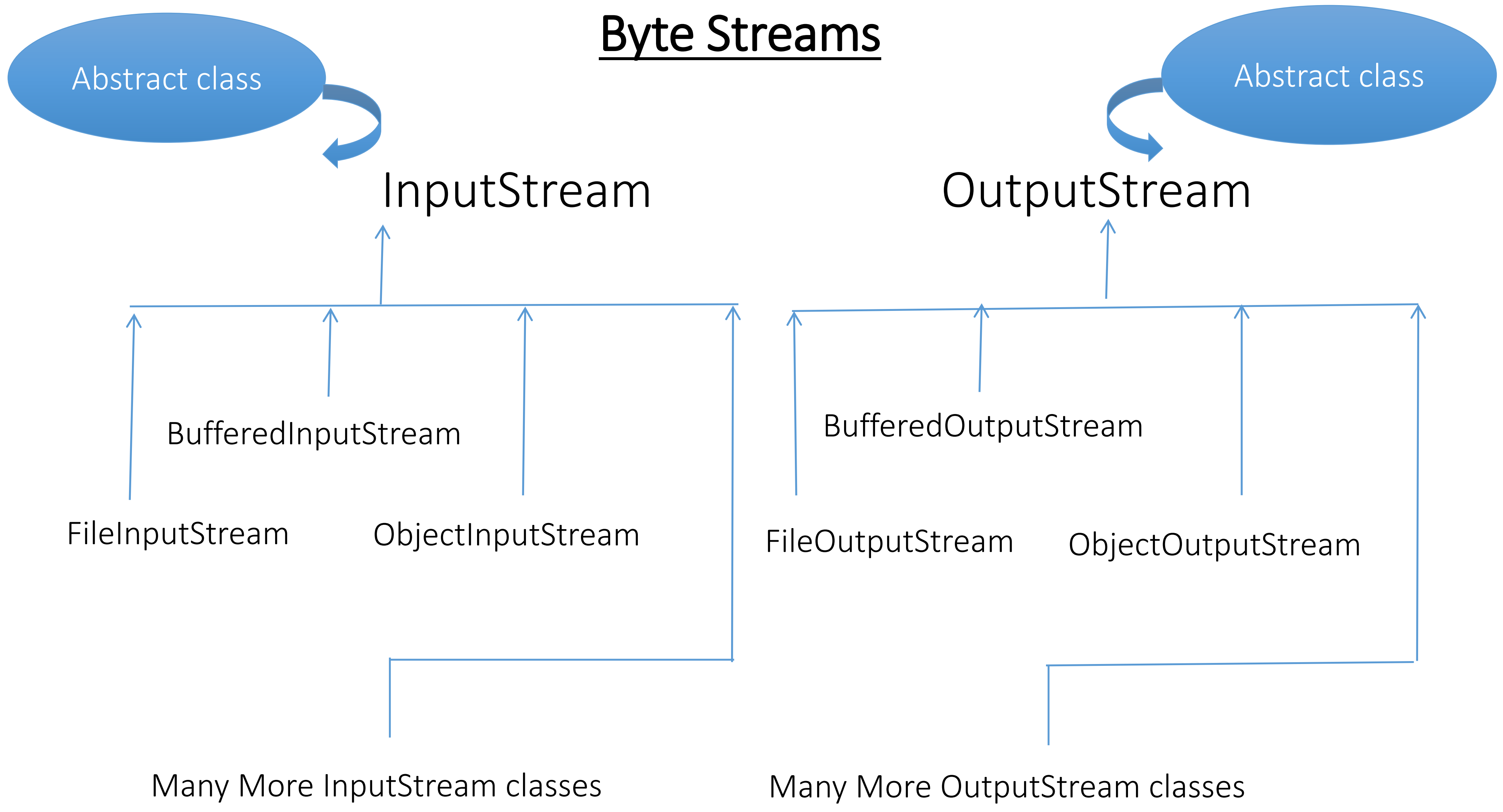
System.getProperty(String Key) method gets the value of a particular property represented with a key.

System.getProperty (String Key) - Demo

Java Project : JavaIOProject
Package: com.deloitte.trg.ui

```
public class GetPropertyDemo{
public static void main(String [] args){
String user_home= System.getProperty("user.home");
String java_version= System.getProperty("java.version");
String java_home = System.getProperty("java.home");
String class_path = System.getProperty("java.class.path");
System.out.println("The user home directory is "+user_home);
System.out.println("The java version is "+java_version);
System.out.println("The Java Home directory is "+java_home);
System.out.println("The class path is set to "+class_path);
}
}
```

I/O Streams hierarchy



Byte Stream classes

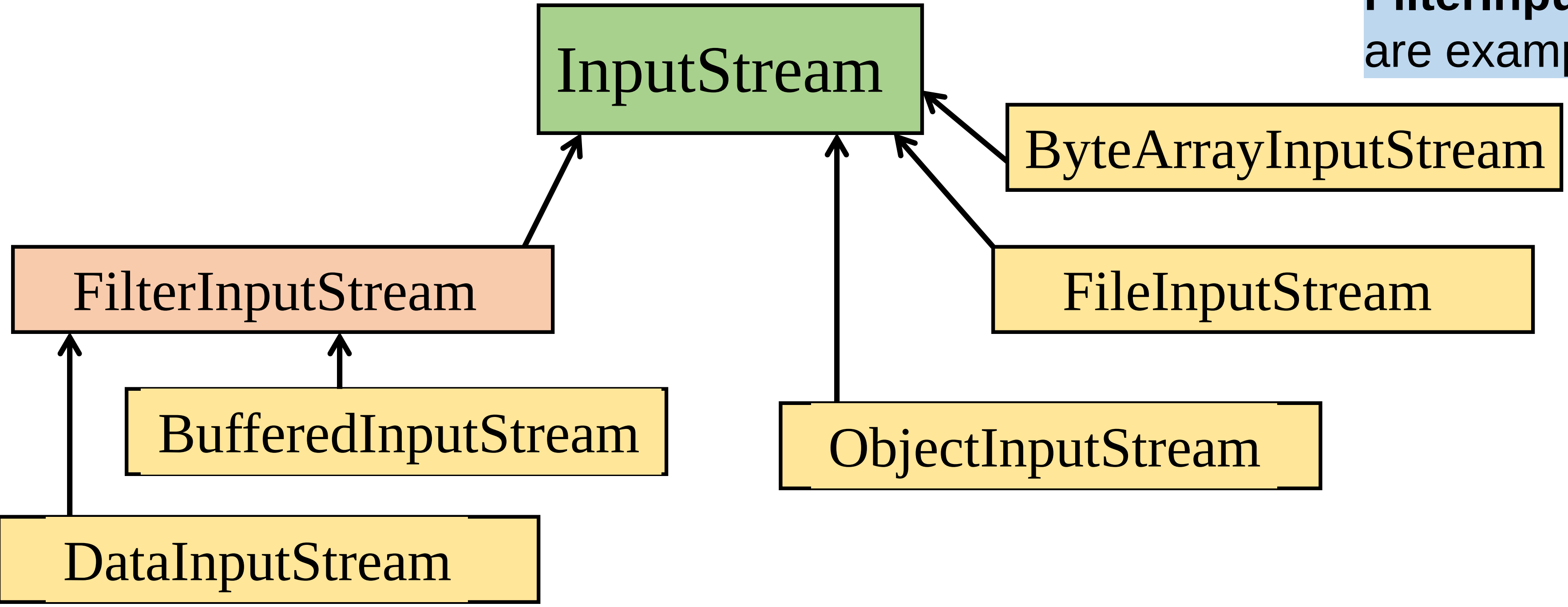
BufferedInputStream **Buffered**OutputStream	To read & write data into buffer
FileInputStream **FIle**OutputStream	To read & write data into file
ObjectInputStream **Object**OutputStream	To read & write object into secondary device (serialization)
DataInputStream **Data**OutputStream	To read & write primitive-type data into file

Byte Stream

- A general stream, which receives and sends information **as bytes,** is called a byte stream.
- Programs use *byte streams* to perform input and output of 8-bit bytes.
- The following are the two abstract classes provided for reading and writing bytes
  - InputStream - Reading bytes
  - OutputStream - Writing bytes
- The goal of InputStream and OutputStream is to abstract different ways to input and output: whether the stream is a file, a web page, or the screen shouldn't matter. All that matters is that you receive information from the stream (or send information into that stream.)
- To perform IO operations, we depend on the sub classes of these two top-level classes

Input Stream hierarchy

FilterInputStream and **FileOutputStream** are examples of *Decorator* design pattern.



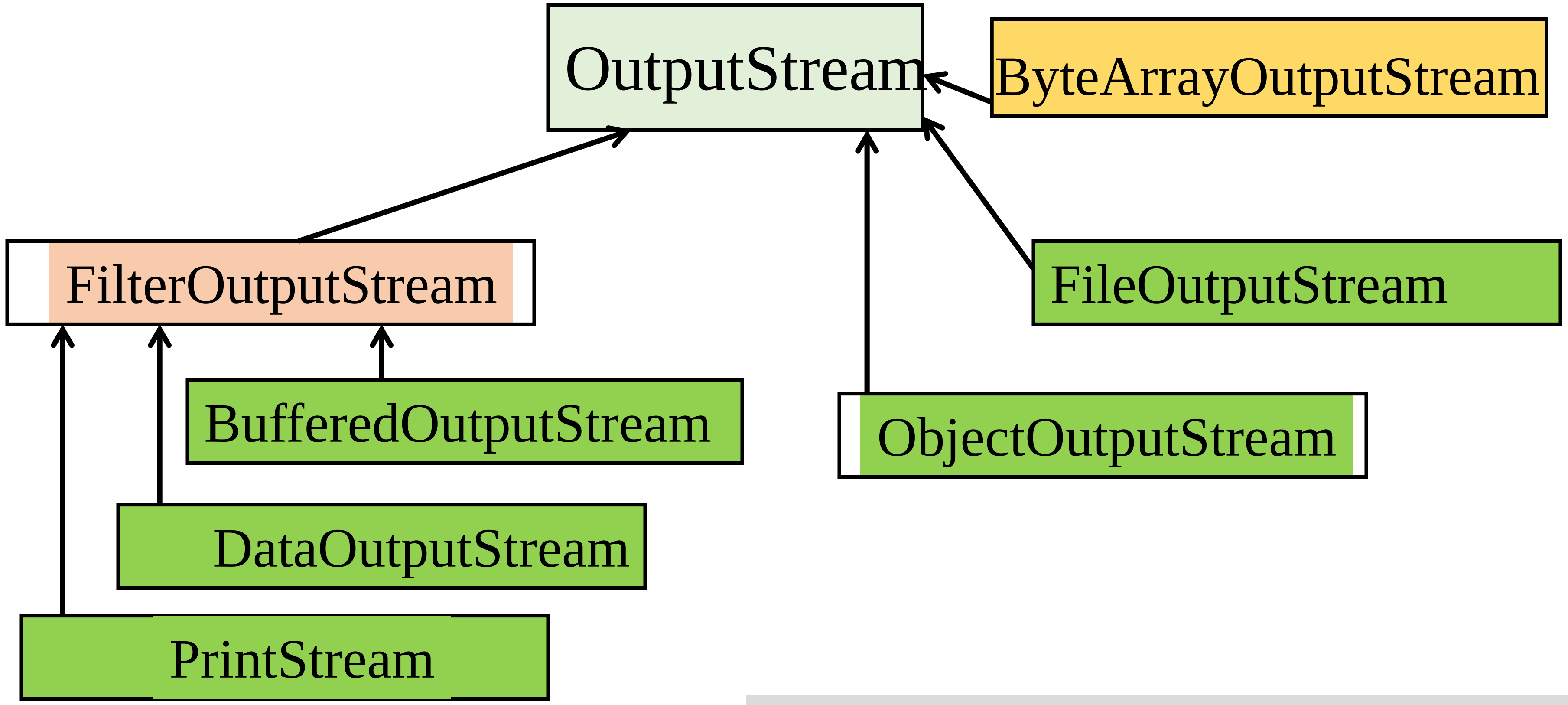
The three basic `read` methods are:

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

Other methods include:

`void close()` : Close an open stream
`int available()` :*Number of bytes available*
`long skip(long n)` :*Discard n bytes from stream*
`void reset()`

OutputStream hierarchy



The three basic `write` methods are:

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

Other methods include:

`void close()` : *Automatically closed in try-with-resources*

`void flush()` : Force a write to the stream

InputStream – The read() method

- The read() method reads a byte from the *input device* and returns an integer value.
- It returns an integer value -1 when the read fails
  - The read() method of **FileInputStream** returns -1 when it reaches end of file condition.

Note : read() returns an int value so as to allow read() to use -1 to end of the stream..

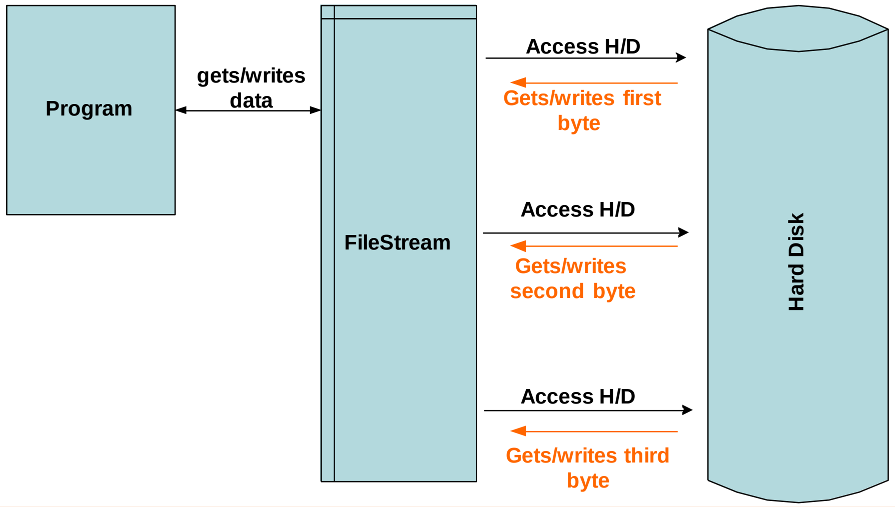
- In case of a successful read, the returned **integer** can be typecast to a **char** to get the data read from the device.

```
try(FileOutputStream out = new FileOutputStream("F:\\data\\deloitte\\demo.txt")){
char[][] words= {{'J','a','v','a','\n'},{'C','D','S'}};
for(int i=0;i<words.length;i++ ){
  for(int j=0; j<words[i].length;j++){
        out.write((int)words[i][j]);
}
}
```

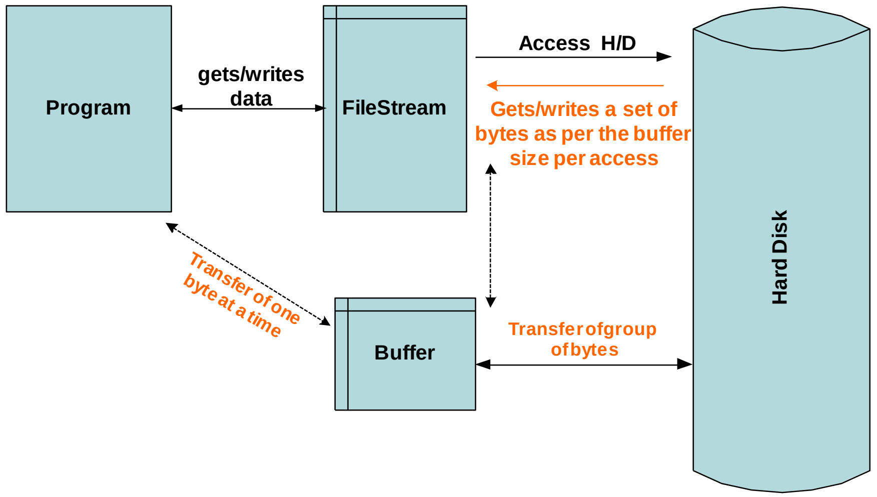
```
try(FileInputStream input = new FileInputStream(" F:\\data\\deloitte\\demo.tx
")){
int ch;
while( (ch=input.read()) != -1){
      System.out.print((char)ch);
}
}
catch(IOException ioe){
  ioe.printStackTrace();
```

Non-Buffered IO Vs Buffered IO

Reading and writing a byte at a time can be expensive in terms of processing time



Buffering helps java to store an entire block of values into a buffer .It never has to go back to the source and ask for more, unless it runs out of data.


Program
gets/writes data
FileStream
Access H/D
Gets/writes a set of bytes as per the buffer size per access
Transfer of one byte at a time
Buffer
Transfer of group of bytes
Hard Disk

File Class

The **File class** in the Java IO API gives you access to the underlying file system.

Using the File class, we can:

- Check if a file or directory exists.
- Create a directory if it does not exist.
- Read the length of a file.
- Rename or move a file.
- Delete a file.
- Check if path is file or directory.
- Read list of files in a directory.

Instantiating a java.io.File

```
File file = new File("pathname");
```

Example:

```
File myFile = new  File("D:\\data\\demo.txt");
System.out.println(myFile.toString());
System.out.println(myFile + (myFile.exists() ?" does" : " does not") + " exist");
System.out.println(myFile + (myFile.isFile() ?" is" : " is not") + " a file");
System.out.println(myFile + (myFile.isHidden() ?" is" : " is not") + " hidden");
System.out.println("You can " + (myFile.canRead() ? " " : " not") + " read" + myFile);
System.out.println("You can " + (myFile.canWrite() ? " " : " not") + " write to" + myFile);
```

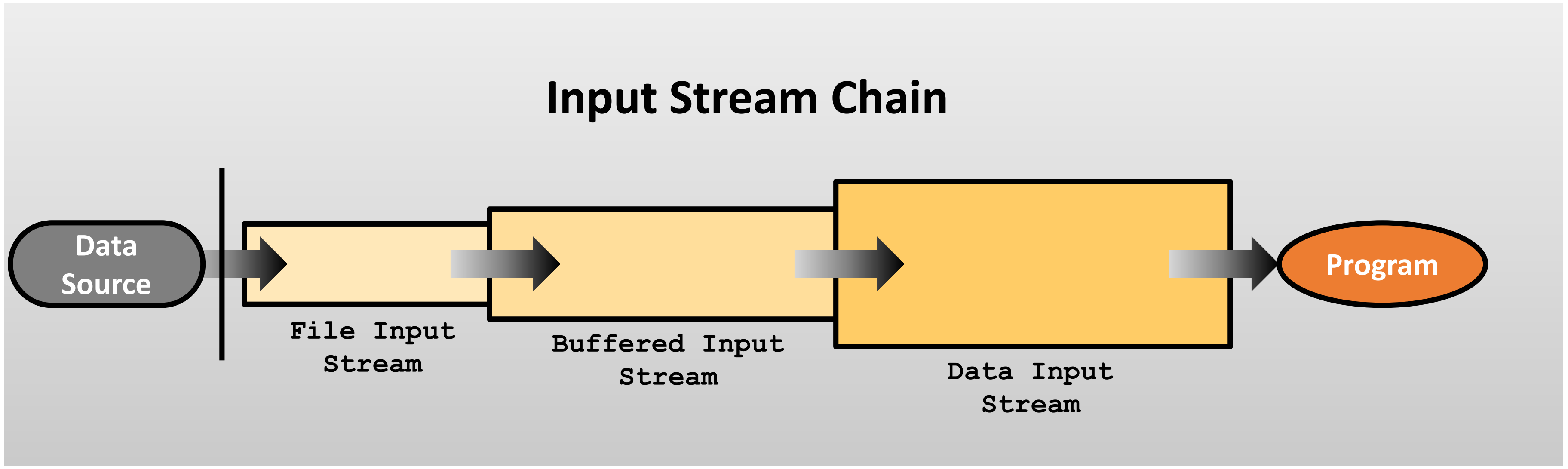
FileOutputStreamDemo

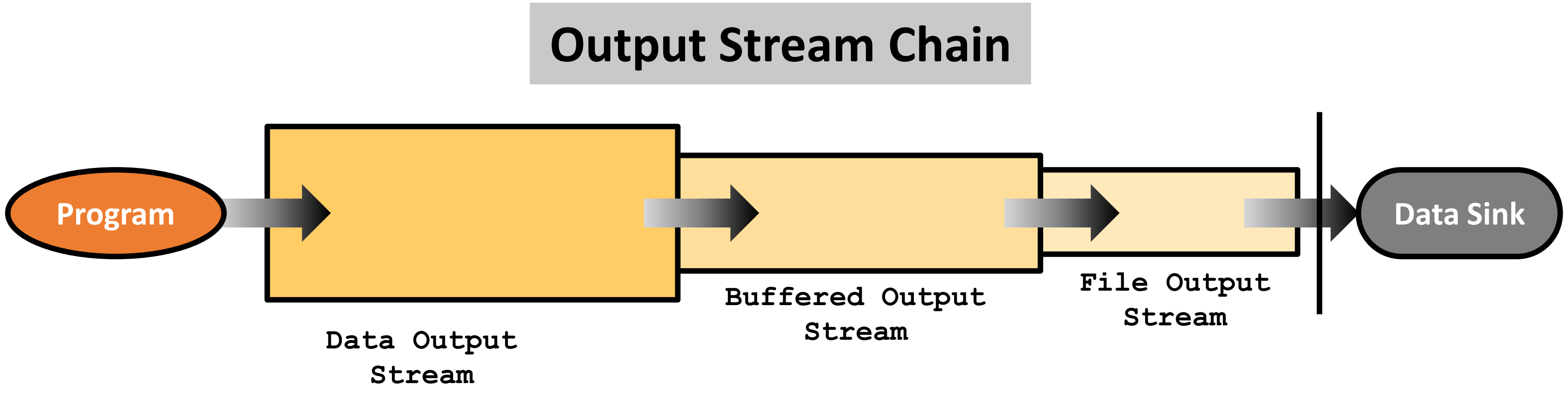
```
public class FileOutputStreamDemo {
public static void main(String[] args) {
File file = new File("F:\\data\\deloitte\\customer.txt");

try(BufferedReader bufferedReader=new BufferedReader(new InputStreamReader(System.in));
FileOutputStream fileOutputStream=new FileOutputStream(file, true);//append mode
BufferedOutputStream bos=new BufferedOutputStream(fileOutputStream);
){
System.out.println("Enter customer names( cntrl-z to stop)\n");
int s;
while((s =bufferedReader.read()) !=-1){
bos.write(s);
}
bos.flush();
System.out.println("\n File created/appended");
}
catch(IOException e){
e.printStackTrace();
}
}
}
```

FileInputStreamDemo

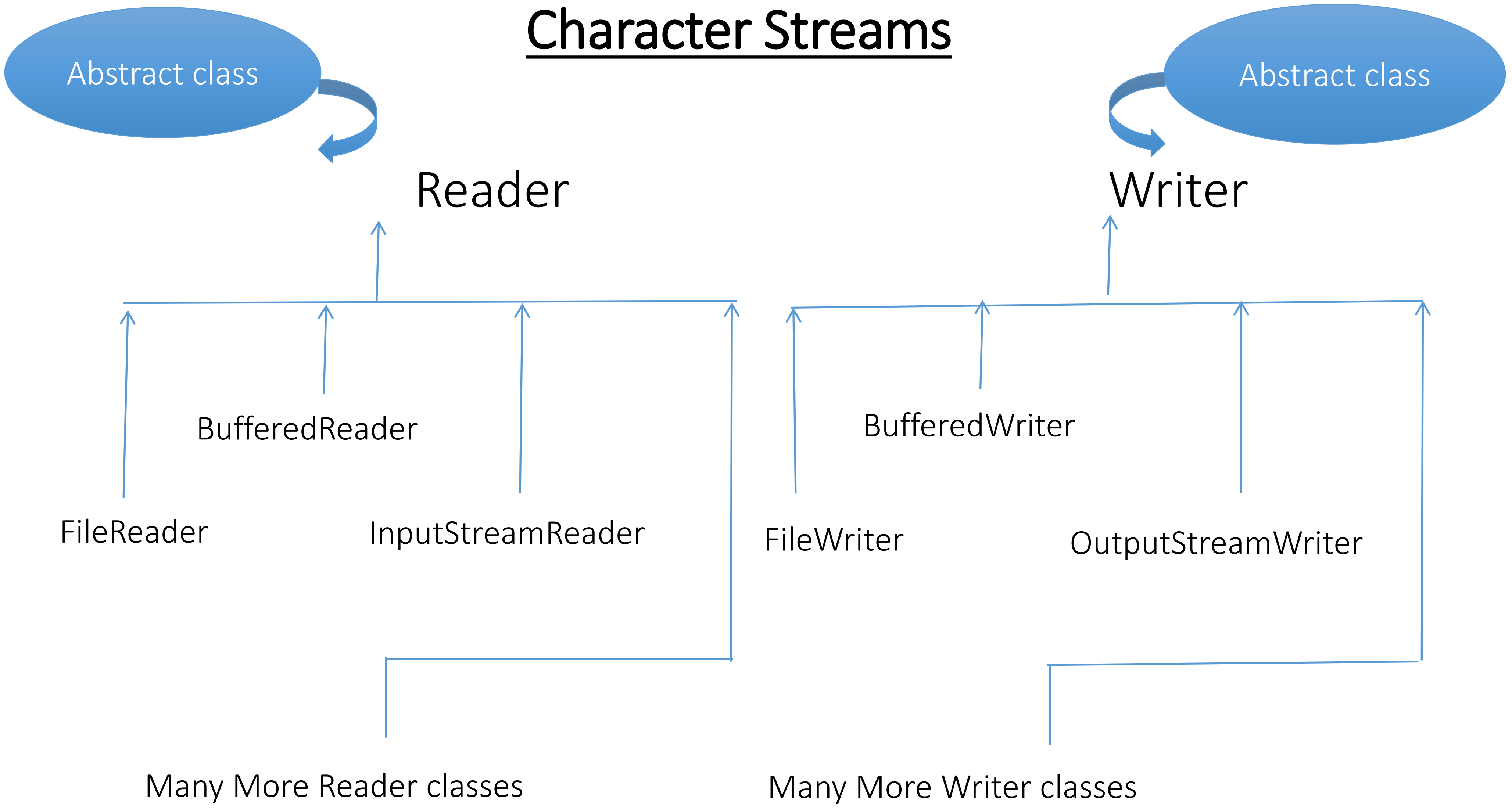
```java
public class FileInputStreamDemo {
public static void main(String[] args) {
//creating File object
File file = new File("F:\\data\\deloitte\\customer.txt");
if(file.exists() && file.canRead()){
try(FileInputStream fileInputStream=new FileInputStream(file);
BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);
){
int n=0;
while( (n=bufferedInputStream.read()) != -1){
System.out.print((char)n);
}
}
catch(IOException e){
e.printStackTrace();
}
}else{
System.out.println("unable to open the file");
}
}
}
```

IO Stream Chaining





Character Streams hierarchy



Character Stream classes

BufferedReader
BufferedWriter

To read & write data into buffer

FileReader
FIleWriter

To read & write data into file

InputStreamReader
OutputStreamWriter

Bridge from character stream to byte stream

Character IO

Character-Oriented System

- The Java platform stores character values using *Unicode conventions*.
- Since byte-oriented streams are inconvenient for Unicode-compliant character- based I/O , Java has the following abstract classes for performing character I/O
  - Reader class
  - Writer class
- Supports *internationalization* of Java I/O
- Subclasses of Reader and Writer are used for handling Unicode characters.

- **Character Streams** handle character data.
- **Reader** and **Writer** classes are to character I/O what **InputStream** and **OutputStream** classes are to byte I/O

Byte IO	Character IO
FileOutputStream	FileWriter
FileInputStream	FileReader
BufferedOutputStream	BufferedWriter
BufferedInputStream	BufferedReader
ByteArrayOutputStream	CharArrayWriter
ByteArrayInputStream	CharArrayReader
etc…	

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

```
void close()
int read()
 int read(char[] c, int len)
 int read(char [] c, int off , int len)
```

Character IO Example

FileWriterDemo

```
public class FileWriterDemo {
public static void main(String[] args) {
File file=new File("f:\\data\\deloitte\\names.txt");
getData(file);
}

private static void getData(File file) {
try(FileWriter fileWriter=new FileWriter(file,true);
PrintWriter printWriter=new PrintWriter( fileWriter);
BufferedReader bufferedReader=new BufferedReader(new InputStreamReader(System.in))
){
System.out.println("Enter number of lines: ");
int size= Integer.parseInt(bufferedReader.readLine());
for(int i=0; i<size; i++){
System.out.println("Enter Line: "+ (i+1));
printWriter.print(bufferedReader.readLine());
printWriter.println();
}
System.out.println("File created");
}catch(IOException ioe){  }}}
```

Character IO Example contd…

FileReaderDemo

```
public class FileReaderDemo {

public static void main(String[] args) {
File file =new File("f:\\data\\deloitte\\names.txt");
if(file.exists() && file.canRead()){
        printData(file);
}else{
        System.out.println("Unable to process the file");
}
}
```

```
private static void printData(File file) {
try(BufferedReader br=new BufferedReader(new FileReader(file))
){
String line="";
while( (line= br.readLine()) !=null ){
        System.out.println(line);
}
}
catch(IOException e){
        e.printStackTrace();
}
catch(Exception e){e.printStackTrace();}}}
```

Object Serialization

Persistence and Object Serialization

Persistence:

- The capability of an object to exist beyond the execution of the program which created it is called *persistence*.

Object Serialization

- The process of **storing objects** of a class that implements Serializable interface into permanent storage device is called **Object Serialization**

Serialization Mechanism

To *serialize* an object means to convert its state to a byte stream/

A Java object is *serializable* if its class or any of its super classes implements either the **java.io.Serializable** interface or its sub-interface, **java.io.Externalizable**.

- *Serializable* objects can be converted into stream of bytes
- This stream of bytes can be written into a file, which is done by **ObjectOutputStream** class methods.
- These bytes can be read back to re-create the objects, which is done by **ObjectInputStream** class methods. This process is called as **Deserialization**.
- Only those objects that implements **java.io.Serializable** or **java.io.Externalizable** interface can be serialized.
- **Serializable interface is a marker interface.**

ObjectOutputStream and ObjectInputStream

- **ObjectOutputStream** wraps around any **OutputStream** and helps to write a Serializable object into the OutputStream
  - **ObjectOutputStream(new OutputStream ())**

- The method **writeObject(Object object**) will write the object to the underlying OutputStream object

  public final void writeObject(Object x) throws IOException

- **ObjectInputStream** wraps around any **InputStream** and helps to read a stream of bytes as an Object from the InputStream
  - **ObjectInputStream(new InputStream())**

- The method **readObject()** reads a stream of bytes, converts them into an Object and returns it

  public final Object readObject() throws IOException, ClassNotFoundException

Student class implementing Serializable interface

```
package com.deloitte.businesstier;
public class Student implements Serializable{
private int admissionCode;
private String studentName;
private GregorianCalendar birthdate;
private String grade;
private static int counter;
public Student(){
this.admissionCode=++counter;
}
........
```

```
package com.deloitte.presentationtier;
......
try{
fileOutputStream=new
FileOutputStream("f:\\data\\deloitte\\student.ser");
objectOutputStream=new ObjectOutputStream(fileOutputStream);
Student student1=new Student("Smith",new
GregorianCalendar(1980,01,10),"B-Grade");
......
objectOutputStream.writeObject(student1);
.......
}
```

```
package com.deloitte.presentationtier;
.....
try{
      fileInputStream=new FileInputStream("f:\\data\\pega\\student.ser");
      objectInputStream=new ObjectInputStream(fileInputStream);
      while(true){
            Student student = (Student)objectInputStream.readObject();
            System.out.println(student);
      }
}catch(EOFException eofException){}
```

Note:

- The values stored in static fields are not serialized.
- When the object is de-serialized, the values of static fields are set to the values declared in the class.

transient field

transient keyword is an indication to the Java virtual machine that the indicated **variable** is not part of the persistent state of the object and hence not to be serialized.

Update Student class by making grade field as *transient*

```
public class Student implements Serializable{
private int admissionCode;
private String studentName;
private GregorianCalendar birthdate;
transient private String grade;
……….
}
```

During De-serialization , the transient fields contains *0* or *space* or *null* based on the field type

Serial Version UID

- The serialVersionUID is a universal version identifier for a Serializable class.
- The serialVersionUID is used as a version control in a Serializable class. If you do not explicitly declare a serialVersionUID, JVM will implicitly compute serialVersionUID.
- De Serialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown.
- A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

```
private static final long serialVersionUID = 1L;
```

To guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value.

Understanding Serial Version UID

```
public class SerializeMe implements Serializable{
private static final long serialVersionUID = 1L;

private int data;

public SerializeMe (int data) {
      this.data = data;
   }
public int getData() {
      return data;
   }
public String toString(){
  return "data:" + data;
}
}
```

- Write a class, ***SerializeDemo*** to serialize objects of the class, **SerializeMe** and another
- Class, **DeSerializeDemo** to de-serialize the objects and display.
- Change the *serialVersionUID* of *SerializeMe* class and save.
- Re-run the DeSerialize class.

We get compilation error and the following exception object is thrown.

Exception in thread "main" java.io.InvalidClassException:
com.deloitte.businesstier.SerializeMe;
local class incompatible: stream classdesc serialVersionUID = 1, local class
serialVersionUID = 2

ObjectOutputStream writes every time the value of *serialVersionUID* to the output stream, even though serialVersionUID is static field.
ObjectInputStream reads it back and if the value read from the stream *does not* agree with the *serialVersionUID* value in the current version of the class, then it throws the *InvalidClassException*.

Thank You!