Java Input/Output library

Agenda

- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Byte streams
- Character streams

Agenda

- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- File class
- Serialization

I/O Streams

An *I/O Stream* represents an input source or an output destination

A stream can represent many different kinds of sources and destinations:

- HDD
- Devices
- Other programs
- Network sockets

I/O Streams

- Streams support many different kinds of data
  - simple bytes, primitive data types, localized, characters, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways.
- No matter how they work internally, all streams present the same simple model to programs that use them
  - A stream is a sequence of data

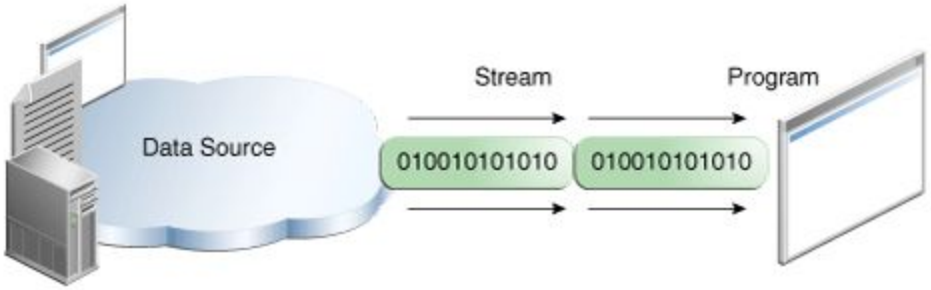
I/O Streams

Stream I/O operations involve three steps:

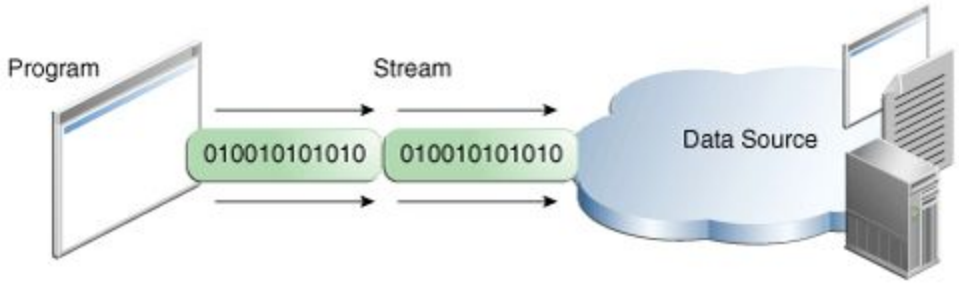
- *Open* a stream with associated source
- *Read* from the opened input stream until "end-of-stream" encountered, or *write* to the opened output.
- *Close* the stream.

I/O Streams

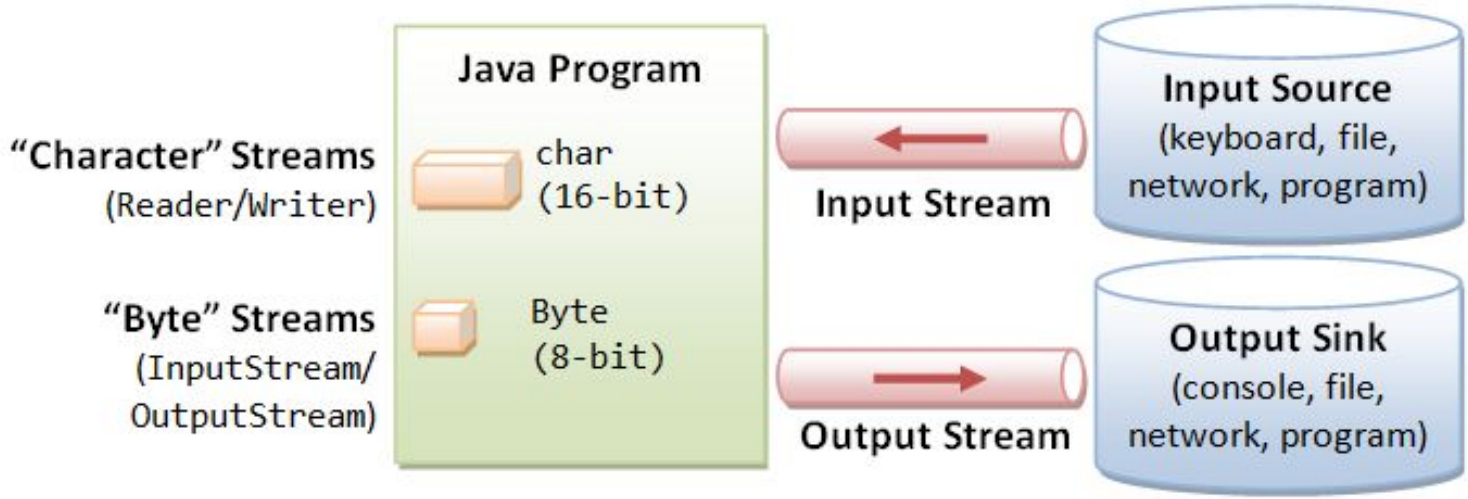
- Reading information into a program (INPUT).



- Writing information from a program (OUTPUT).



I/O Streams types



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Byte Streams

- 8 bits, data-based
- Two parent `abstract` classes:
  - `InputStream`
  - `OutputStream`

InputStream

- Reading bytes:
  - `InputStream` class defines an abstract method

```
public abstract int read() throws
  IOException
```

    - Designer of a concrete input stream class overrides this method to provide useful functionality.
    - E.g. in the `FileInputStream` class, the method reads one byte from a file
  - `InputStream` class also contains nonabstract methods to read an array of bytes or skip a number of bytes

OutputStream

- Writing bytes:
  - `OutputStream` class defines an abstract method

```
public abstract void write(int b) throws
  IOException
```

  - `OutputStream` class also contains nonabstract methods for tasks such as writing bytes from a specified byte array

Example

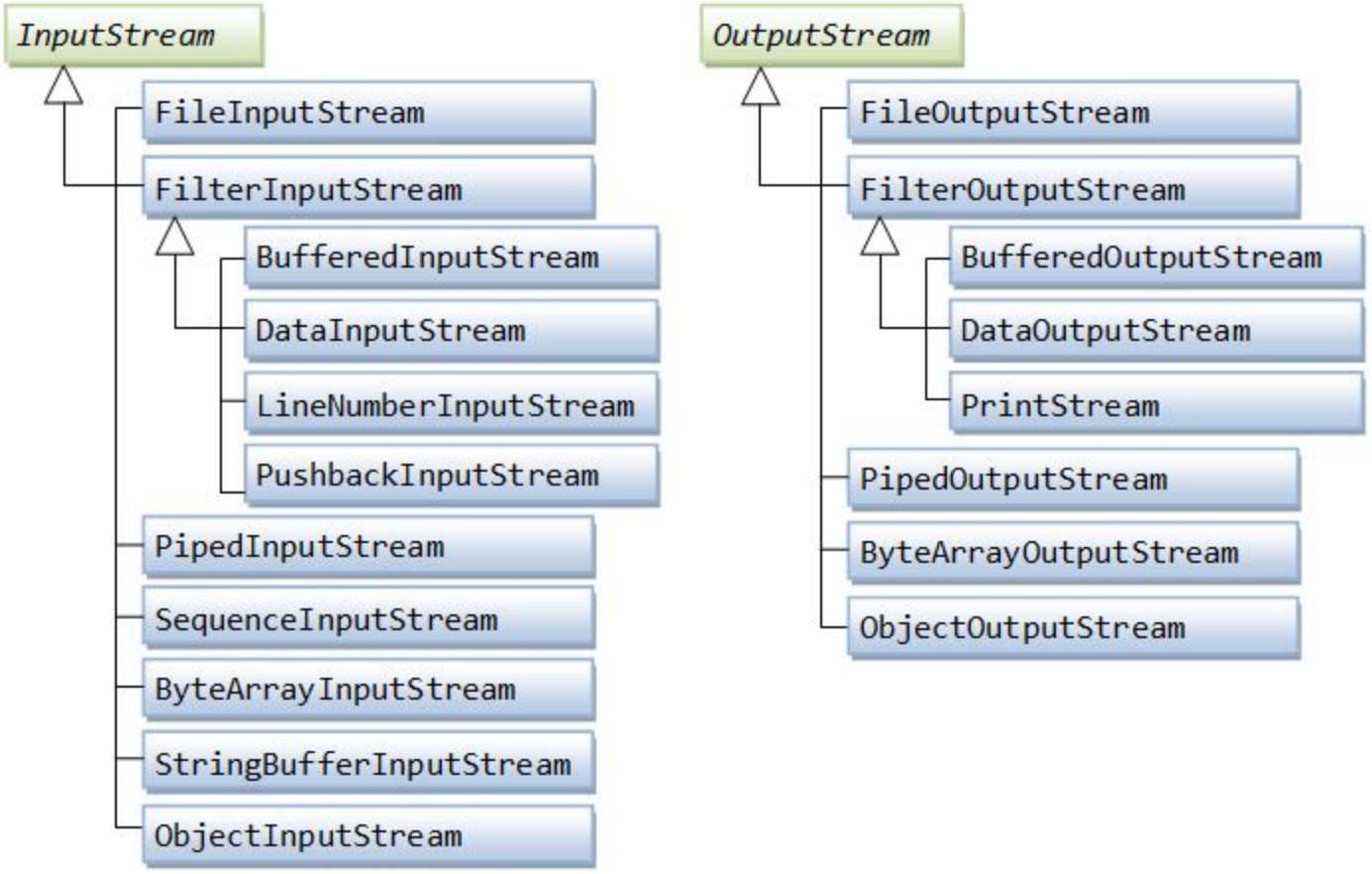
```
FileInputStream in = null;
......
try {
   in = new FileInputStream(...);  // Open stream
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
} finally {  // always close the I/O streams
   try {
      if (in != null) in.close();
   } catch (IOException ex) {
      ex.printStackTrace();
   }
}
```

Example

- JDK 1.7 introduces a new try-with-resources syntax, which automatically closes all the opened resources after try or catch, as follows.

```
try (FileInputStream in = new FileInputStream(...)) {
    ......
    ......
} catch (IOException ex) {
    ex.printStackTrace();
}  // Automatically closes all opened resource in try (...).
```

Byte Streams implementations



File I/O Byte-Streams

FileInputStream and **FileOutputStream** are concrete implementations to the abstract classes **InputStream** and **OutputStream**, to support I/O from disk files.

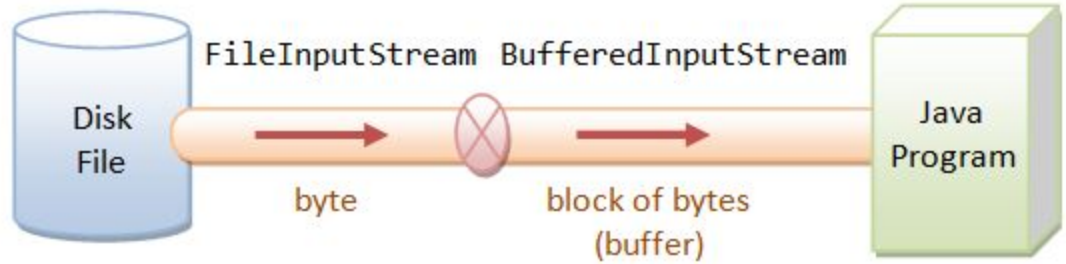
Buffered I/O Byte-Streams

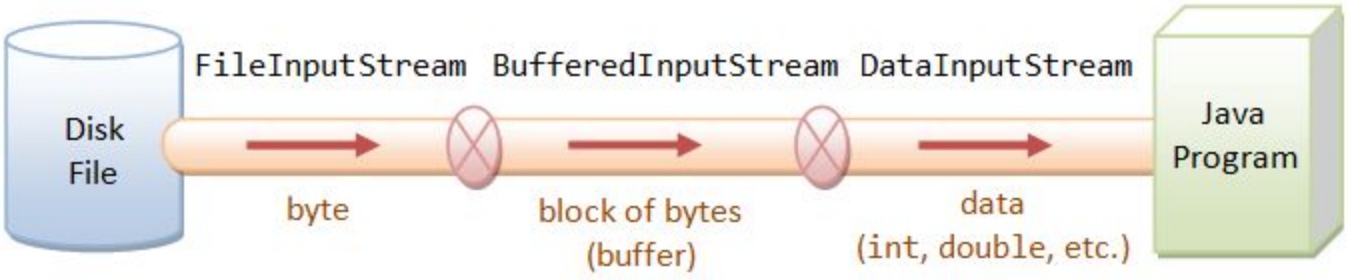
BufferedInputStream & BufferedOutputStream

- *Buffering*, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

Layered (or Chained) I/O Streams

- The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or data-format conversion (between raw bytes and primitive types)

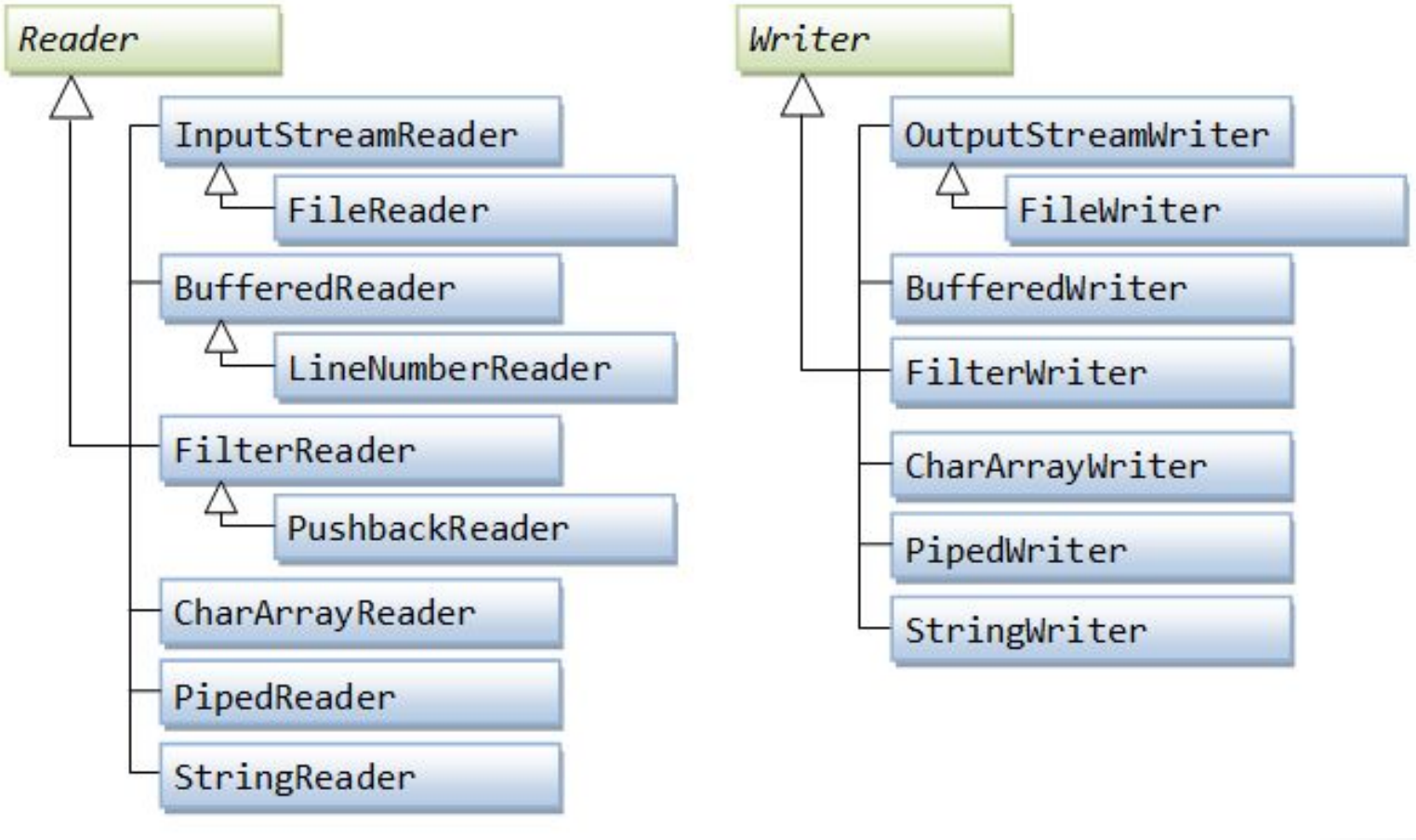




Character Streams

- 16 bits unicode, text-based
- Two parent `abstract` classes for characters: `Reader` and `Writer`.

Character Streams implementations



PrintWriter/PrintStream

- The PrintWriter and PrintStream classes are designed to simplify common text output tasks.
  - The **print()** method is overloaded to print a String representation of all Java primitive types, and to automatically print the toString() representation of all Objects.
  - The **println()** method works in the same way as print(), but add a platform-specific line terminator.
  - The **format()** - formatted representation of one or more Objects
  - The class methods never throw an IOException. Instead, exceptional situations merely set an internal flag that can be tested via the **checkError()** method.

Standard Streams

- Standard Streams are a feature of many operating systems.
  - System.in
  - System.out
  - System.err

File class

- The path may or may not refer to an actual on-disk file or directory.
- Methods on the File class allow you to manipulate the path and perform file system operations.
- The File class is **not** used to read or write file contents.

File class

The File constructor is overloaded, allowing you to create a File object from:

- A single String representing a path
- A String or File representing a parent directory path and a second String argument representing a child directory or file

File class

- The path used to create a File object can be absolute or relative to the present working directory.
- Like String objects, File objects are *immutable*.
- Once you create one, you cannot modify the path it represents.

File class

Methods that modify the file system include:

- createNewFile()
- mkdir()
- mkdirs()
- renameTo()
- delete()
- deleteOnExit()
- setReadOnly()
- setLastModified()

File class

Methods that query the file system include:

- canRead()
- canWrite()
- exists()
- isDirectory()
- isFile()
- isHidden()
- getAbsolutePath()
- lastModified()
- length()
- listFiles()
- listRoots()

Unix & Windows

Unix path name:

- Example: "/user/angela/data/data.txt"
- A BufferedReader input stream connected to this file is created as follows:

```
is = new BufferedReader(new FileReader("/user/sallyz/data/data.txt"));
```

Windows path name:

- Example: C:\dataFiles\data\data.txt
- A BufferedReader input stream connected to this file is created as follows:

```
is = new BufferedReader(new FileReader("C:\\dataFiles\\data\\data.txt"));
```

- Note that in Windows \\ must be used in place of \, since a single backslash denotes an the beginning of an escape sequence

Serialization

- Object serialization is the process of representing a "particular state of an object" in a serialized bit.

Serialization

For an object (class) to be serializable, the class must:

- Implement the java.io.Serializable interface, a *marker interface* with no required methods
- Contain instance fields that are serializable — primitives or other Serializable types — except for any fields marked as **transient**

Serialization

- Have a no-argument constructor
- (Optional but recommended) Implement a static final long field named **serialVersionUID** as a "version number" to identify changes to the class implementation that are incompatible with serialized objects of previous versions of the class.

Public static final long serialVersionUID = 1L;

Serialization

You can then serialize and deserialize objects with the following filter classes:

- **ObjectOutputStream** — Serialize an object to an underlying **OutputStream** with the **writeObject**() method.
- ObjectInputStream — Deserialize an object from an underlying **InputStream** with the **readObject**() method.

Serialization example

```
public class Car implements Serializable{

    public static final long serialVersionUID = 123L;

    private int serialNumber;
    private String model;
    private String manufacturer;
    private Color color;
    private double engineVolume;
    private transient String information;

    //add all getters and setter
}
```

Serialization example - writing

```
public class Main {

    public static void main(String[] args) {
        ObjectOutputStream outputStream = null;
        try {
            Car car = new Car();
            car.setColor(new Color(200, 100, 150));
            car.setEngineVolume(2.0);
            car.setInformation("Some car information");
            car.setManufacturer("Audi");
            car.setModel("A5");
            car.setSerialNumber(123456);
            outputStream = new ObjectOutputStream(
                    new BufferedOutputStream(
                            new FileOutputStream(
                                    "serializable_file.txt")));

            outputStream.writeObject(car);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (outputStream != null) {
                try {
                    outputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Serialization example - reading

```
public class Main {

    public static void main(String[] args) {
        ObjectInputStream inputStream = null;
        Car car = null;

        try {
            File file = new File("serializable_file.txt");
            if (file.exists()) {
                inputStream = new ObjectInputStream(
                        new BufferedInputStream(
                                new FileInputStream(file)));
                car = (Car) inputStream.readObject();
                System.out.println("Color: " + car.getColor());
                System.out.println("Engine: " + car.getEngineVolume());
                System.out.println("Info: " + car.getInformation());
                System.out.println("Manufacturer: " +
car.getManufacturer());
                System.out.println("Model: " + car.getModel());
                System.out.println("Serial: " + car.getSerialNumber());
            } else {
                System.out.println("Cant find file!");
            }
```

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