# Input and Output in Java

Byte- and Character-based I/O

File Objects

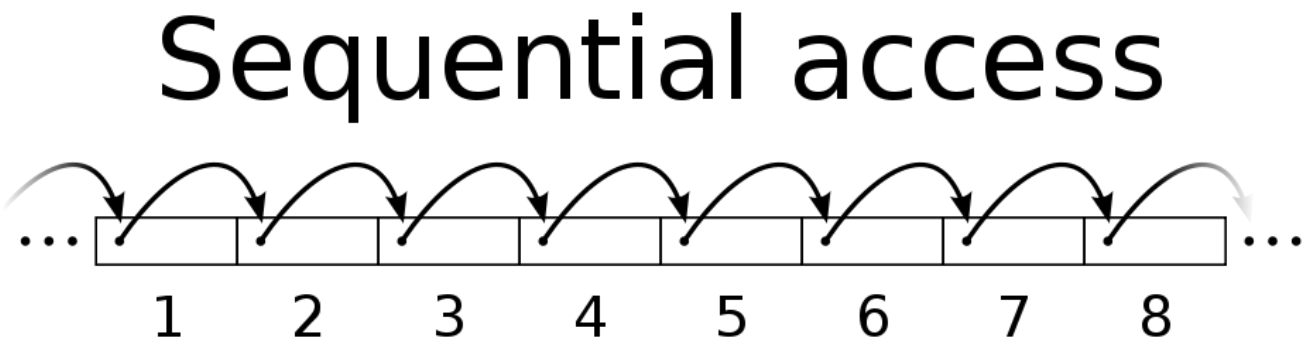
jonas.kvarnstrom@liu.se – 2016

# Introduction

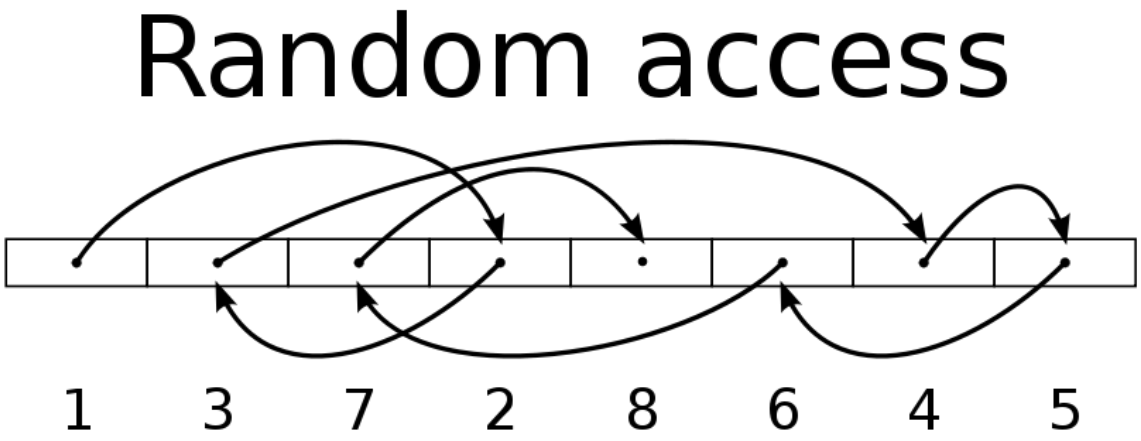


jonkv@ida

- Two forms of I/O:




**Discussed in this lecture**



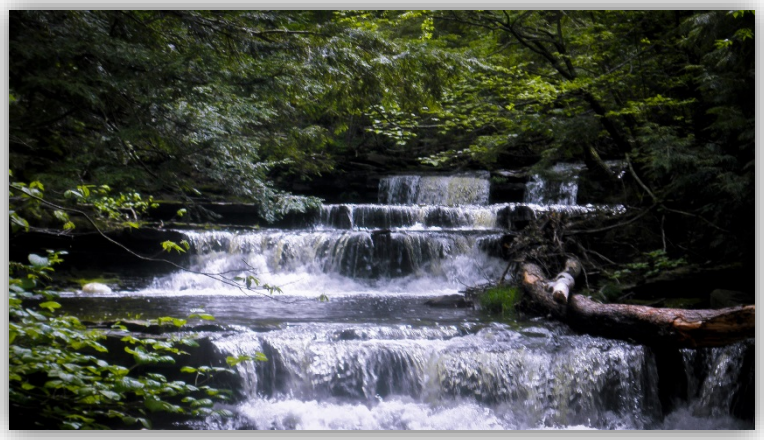

**See RandomAccess File**

# Streams 1: The Concept



jonkv@ida

- Sequential access: Provided by a **stream**
  - In nature:
    Water



  - In industry, "discretized":
    A conveyor belt
    - A **sequence of elements**
    - Arriving **one at a time**

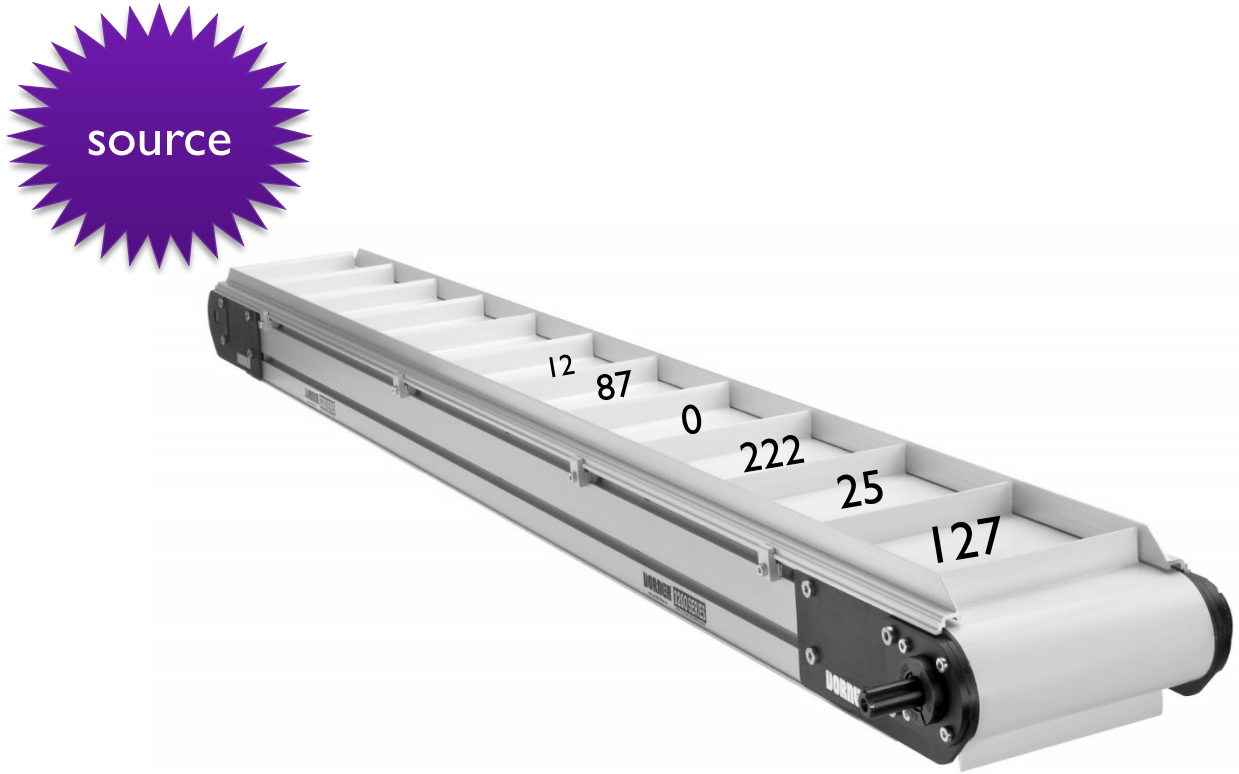
**Used in most of Java's I/O classes**

# Streams 2: Input



jonkv@ida

- Elements can be *bytes*
  - Can come from a file on disk, but also:
    - A *network connection* – a natural stream, can't jump back/forward
    - An *array of bytes* – already in memory, but someone wants a stream from us!
    - …

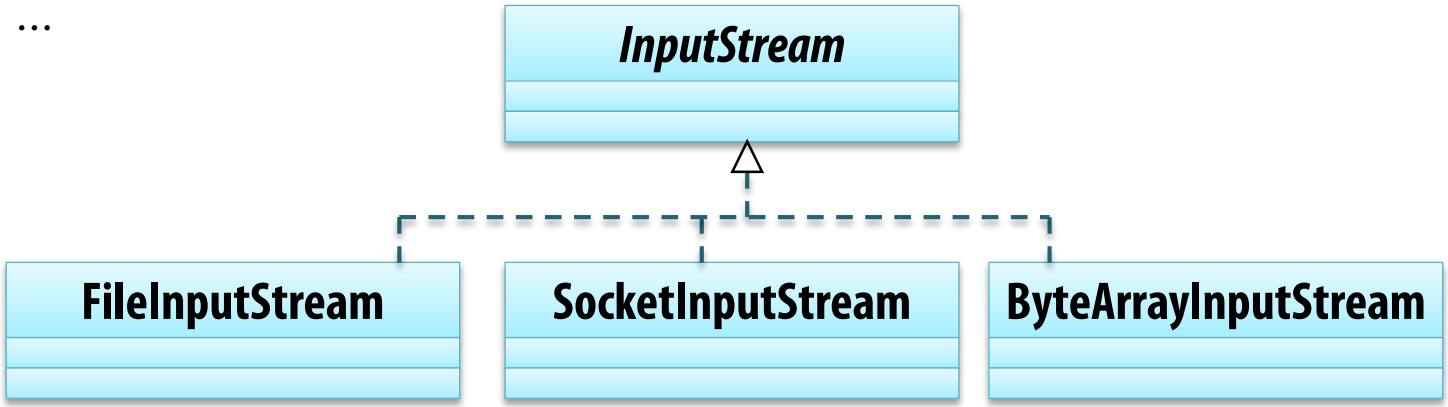


# Streams 3: Sources



jonkv@ida

- Java streams:
  - One abstract "general input stream"
    - abstract class InputStream
  - One concret subclass for each source
    - `InputStream is1 = new FileInputStream("info.dat");`
    - `InputStream is2 = new ByteArrayInputStream(myArray);`
    - `InputStream is3 = socket.getInputStream();`
    - …



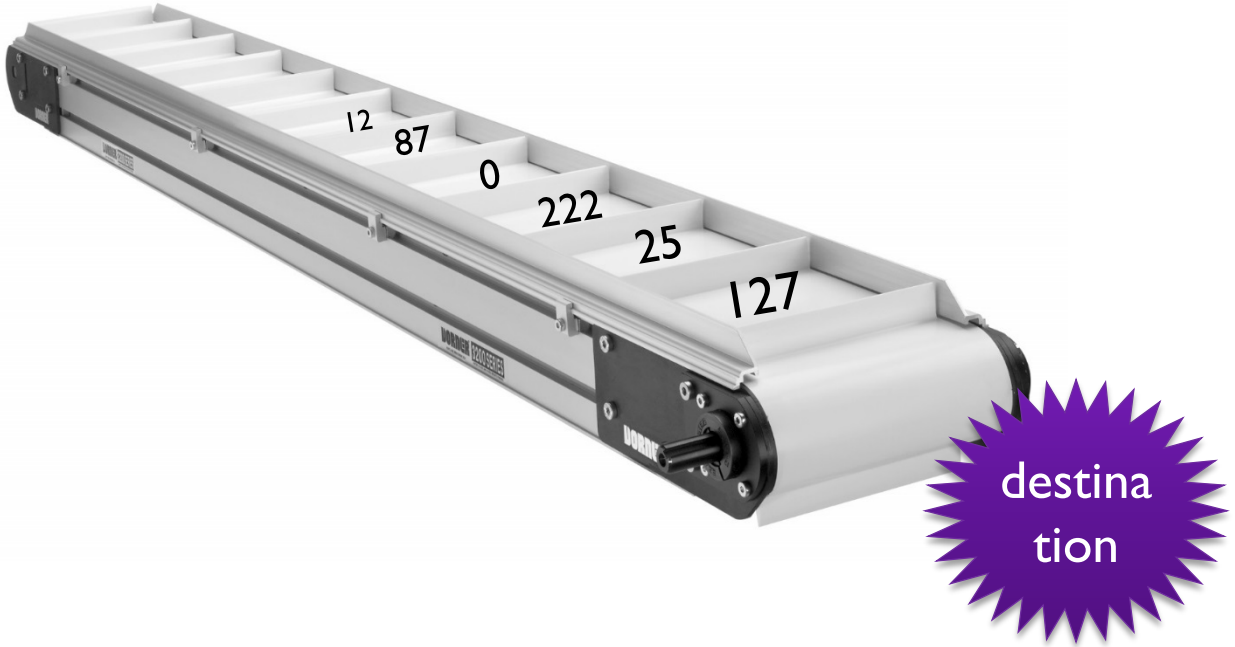
# Streams 4: Output



jonkv@ida

- We can *write* to a file on disk, but also to:
  - A *network connection*
  - An *array of bytes* – someone generates a stream, but we want to *capture* it!
  - …

**Our code, writing**

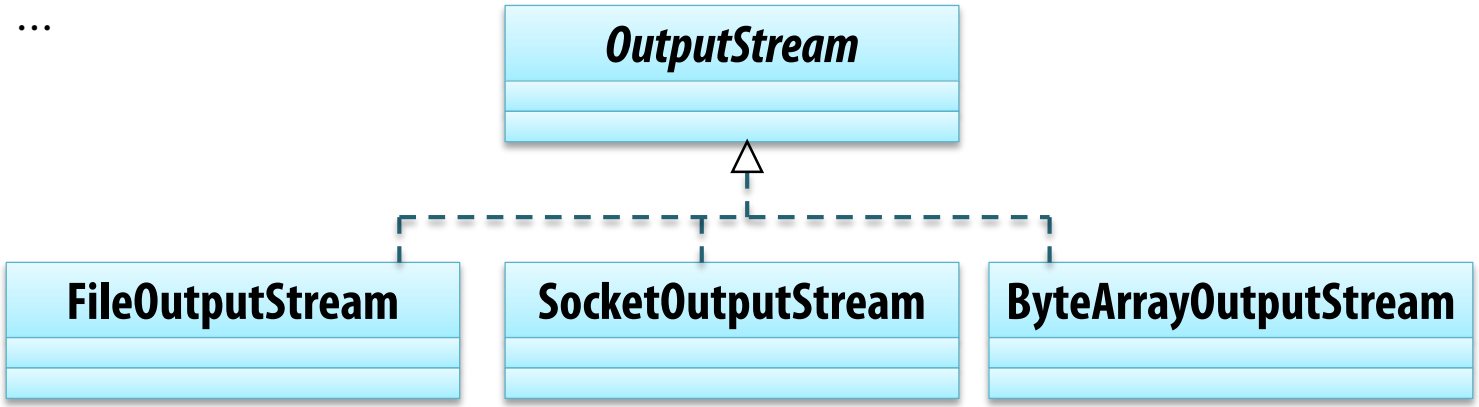


# Streams 5: Destinations



jonkv@ida

- Java streams:
  - One abstract "general output stream"
    - abstract class OutputStream
  - One concret subclass for each source
    - `OutputStream os1 = new FileOutputStream("info.dat");`
    - `OutputStream os2 = new ByteArrayOutputStream(myArray);`
    - `OutputStream os3 = socket.getOutputStream();`
    - …



# Streams 6: Open, use and close



jonkv@ida

- A simple **OutputStream** example
  - Remember that these are *byte streams*
    - ```
      public static void main(String[] args) {
          OutputStream os = new FileOutputStream("info.dat");
          …
          os.write(127);
          os.write(new byte[] { 6, 7, 8 });
          …
          os.close();
      }
      ```

**But file I/O can lead to *exceptions*!**
      ```

# Streams 7: Open, use, close, handle errors



jonkv@ida

- A simple **OutputStream** example
  - With error handling

```
public static void main(String[] args) {
    try {
        OutputStream os = new FileOutputStream("info.dat");
        ...
        os.write(3);
        os.write(new byte[] { 6, 7, 8 });
        ...
        os.close();
    } catch (IOException e) {
        ... handle I/O errors that may arise when opening or using the stream...
    }
}
```

**But if this yields an exception, we won't close the file!**

# Streams 8: Try-with-resources



jonkv@ida

- A simple **OutputStream** example
  - Using "try-with-resources"

```
public static void main(String[] args) {
    try (OutputStream os = new FileOutputStream("info.dat")) {
        ...
        os.write(3);
        os.write(new byte[] { 6, 7, 8 });
        ...
        // No need to close here
    } catch (IOException e) {
        ... handle I/O errors that may arise
            when opening or using the stream...
    }
}
```

Declare an AutoCloseable resource in try()…

Use it inside the block…

… and when you exit the block, the try statement **will** close the resource (file), even if there is an exception!

**Error handling is important! Always make sure that every file is closed!**

# Streams 9: Example



jonkv@ida

- Using more than one resource:
  - ```
    public static void main(String[] args) {
        try ( InputStream is = new FileInputStream("foo.dat");
              OutputStream os = new FileOutputStream("info.dat"))
        {
            …
        } catch (IOException e) {
            … handle it …
        }
    }
    ```
    ```

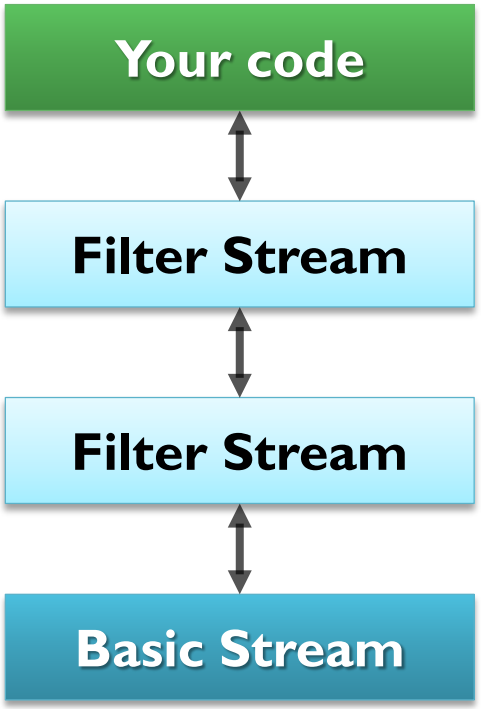
# Stream Filters 1: Introduction



jonkv@ida

## –Very little basic functionality… for a reason!

- Division of responsibilities
  - "**Basic**" streams handle **sources** and **destinations**
    - Only handle bytes, byte arrays
  - **Filter streams** provide additional functionality
    - Add buffering
    - Support other types of data
    - …



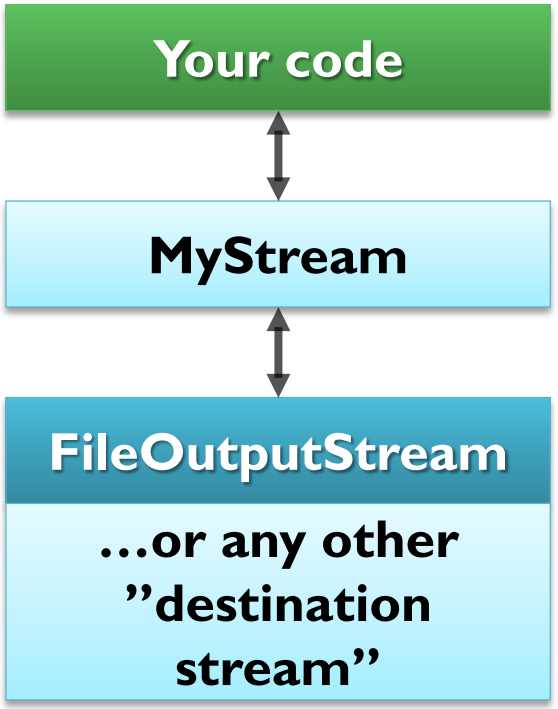
# Stream Filters 2: How do they Work?



jonkv@ida

- A simple filter stream example:
  - ```
    try (MyStream out = new MyStream(new FileOutputStream("foo.dat"))) {
        out.write("Hello, World!");
    }
    ```
  - ```
    public class MyStream {
        private OutputStream out;
        public MyStream(final OutputStream out) {
            this.out = out;
        }

        public void write(String str) {
            char[] chars = str.toCharArray();
            byte[] bytes = convertToUTF8(chars);
            out.write(bytes);
        }
        …
    }
    ```



# Stream Filters 3: Examples



jonkv@ida

## Useful output filters

- **BufferedOutputStream**
  - Buffers I/O:
    Don't write a single byte at a time…
- **PrintStream** (System.out / err)
  - print(), println()
  - Uses platform character encoding to convert chars ➔ bytes
- **DataOutputStream**
  - writeLong(), writeFloat(), …
  - writeUTF() – writes UTF-8 format

```
try (DataOutputStream os =
            new DataOutputStream(
             new BufferedOutputStream(
              new FileOutputStream("info.dat"))))
{
   os.write(new byte[] { 6, 7, 8 });
   os.writeLong(1234567890123L);
   os.writeFloat(2.7f);
} catch (final IOException e) {
   … handle it …
}
```

## Useful input filters

- **BufferedInputStream**
  - Buffers I/O:
    Don't read a single byte at a time…
- **DataInputStream**
  - readLong(), readFloat(), readUTF()

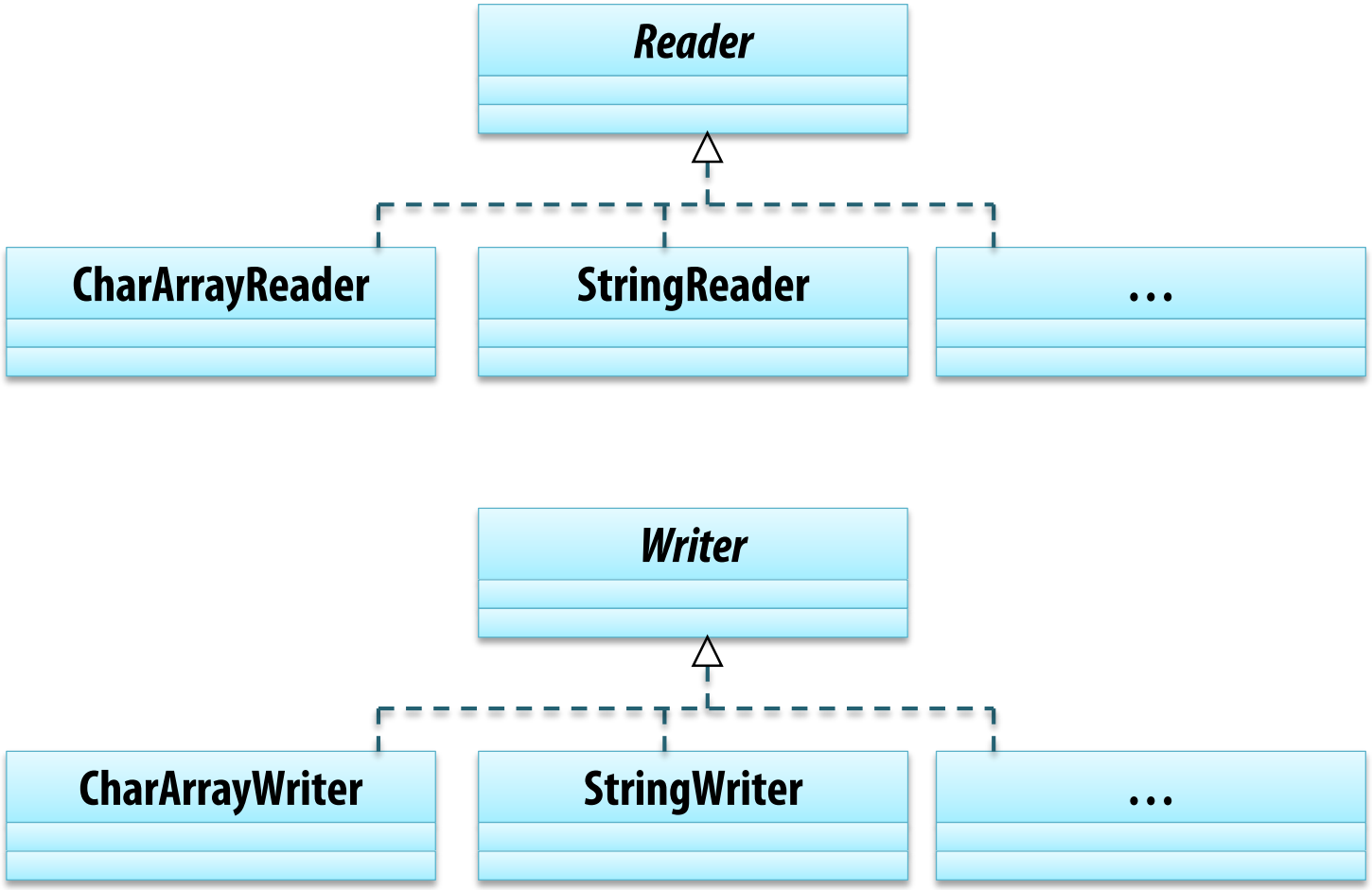
# Text I/O: Readers and Writers

# Readers and Writers



jonkv@ida

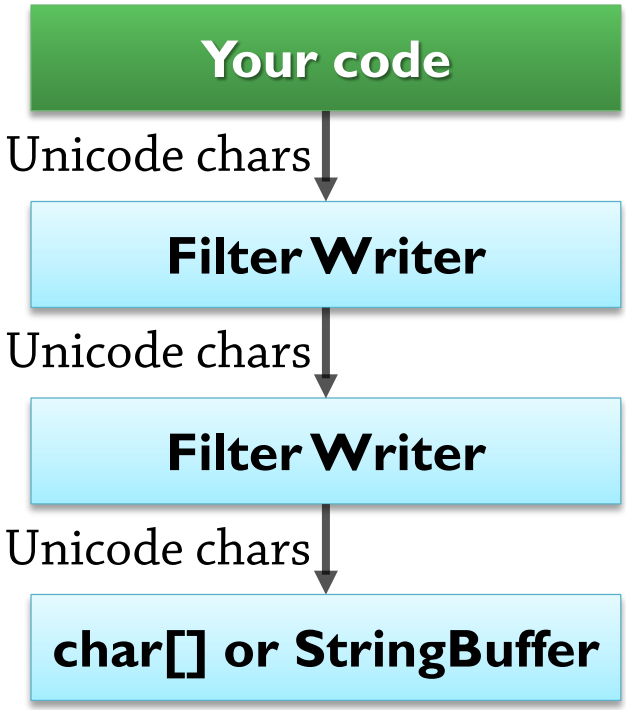
- Distinct subsystem for **text-based I/O**: **Reader**, **Writer**
  - Elements are full 16-bit Unicode characters, not 8-bit bytes

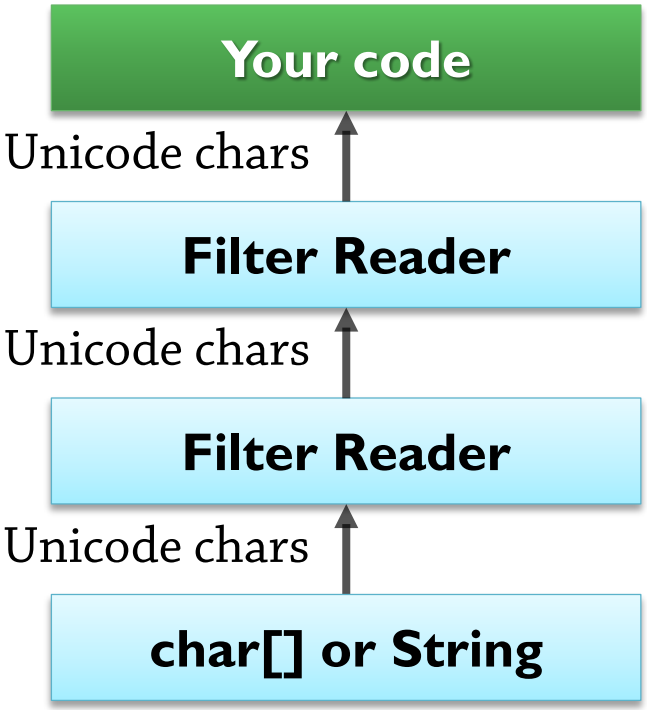


# Readers and Writers 2



- Also supports filters




## Filter Writers

- **PrintWriter**
  - print(), println() methods
- **BufferedWriter** buffers I/O

## Filter Readers

- **BufferedReader** buffers I/O
  - Also: Method for reading a *line*

# Readers and Writers 3: Stream Connections

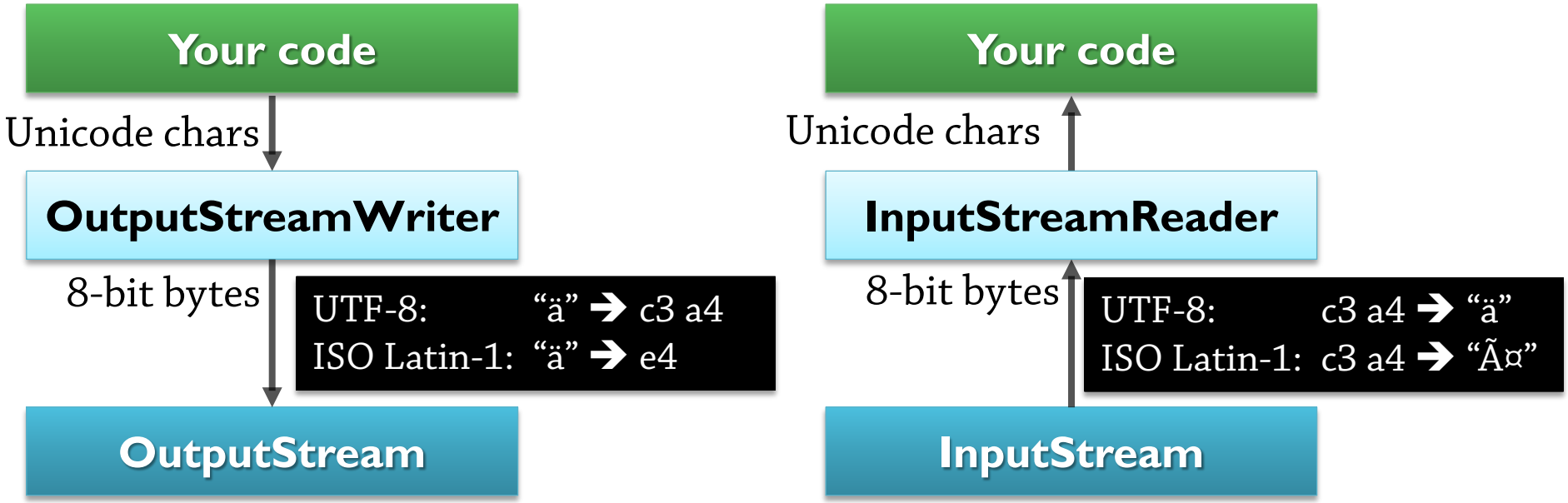


jonkv@ida

- Most destinations only support bytes – files, sockets, …
  - Use adapter classes, tell them how to convert – which *character encoding*?

```
Writer wr = new OutputStreamWriter(
   new FileOutputStream("file.txt"),
   "ISO Latin-1"
);
```

```
Reader re = new InputStreamReader(
   socket.getInputStream(),
   "UTF-8"
);
```



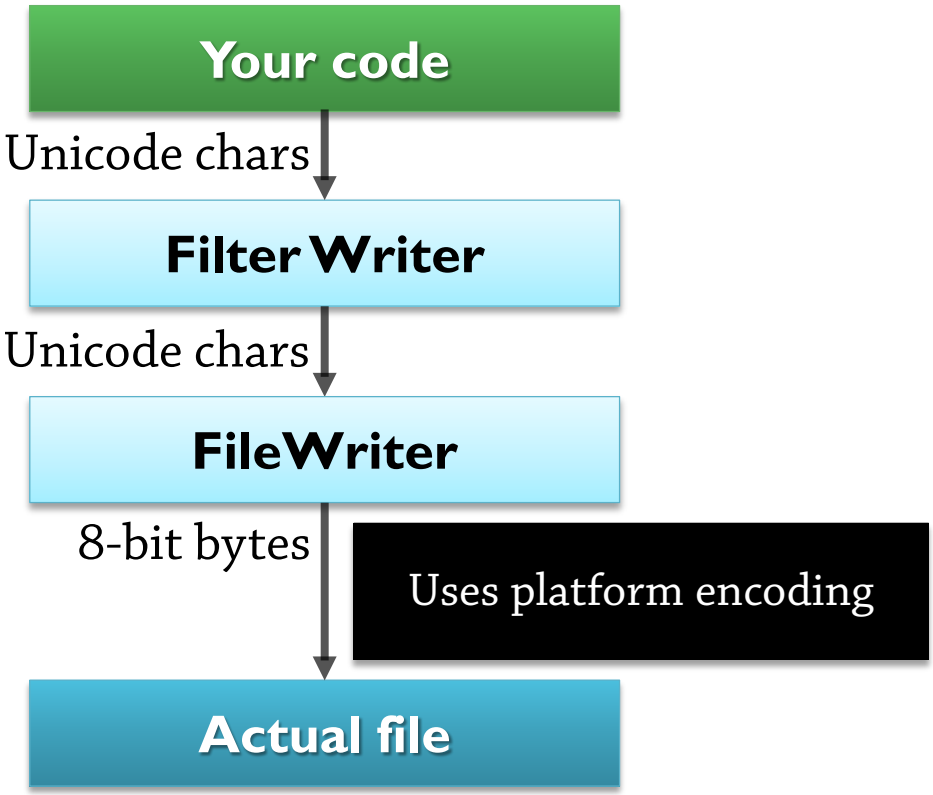

**Use UTF-8 → all characters supported, readable on most systems!**

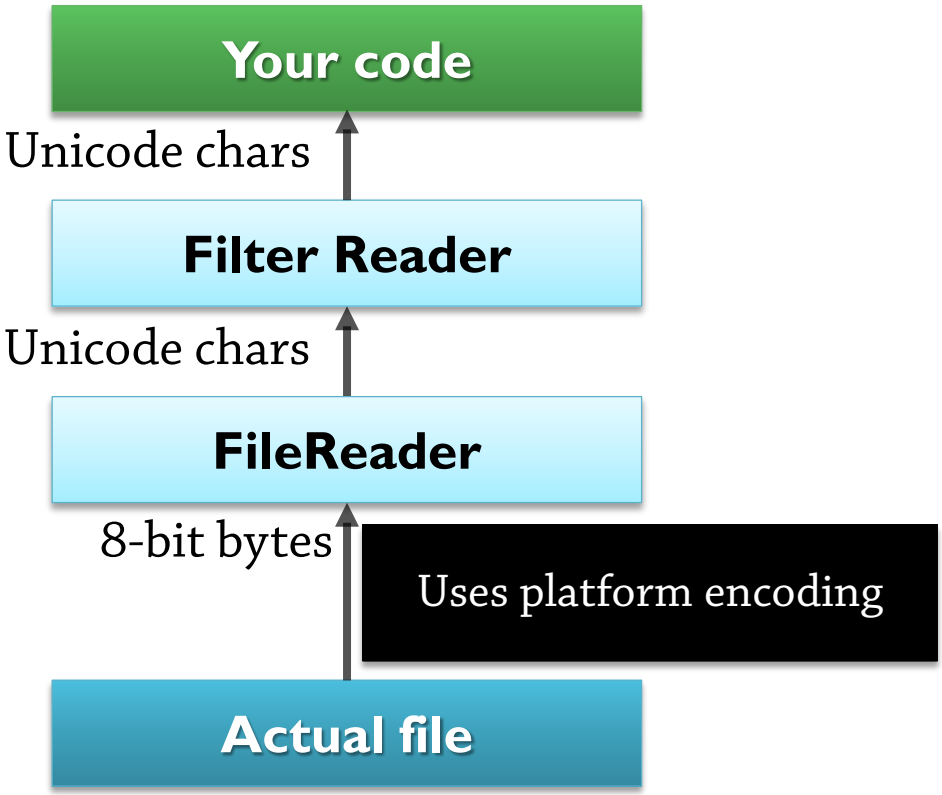
# R/W 4: File I/O, Platform Encoding



jonkv@ida

- Shortcut for file I/O using the platform's *default* encoding
  - **Not** for files that should be machine-readable on *multiple systems*!



# File Objects

# File[name] Objects: The File Class



jonkv@ida

- **File Objects** (java.io.File) represent file and path names
  - They do **not** represent open files!

```
File f = new File("/");
File f2 = new File(f, "etc");
File f3 = new File(f, "passwd");
System.out.print(f3.getAbsolutePath());
if (f3.exists()) {
  System.out.println("You have a password file");
  System.out.println("It's in the directory " + f3.getParent());
  System.out.println("Its length is " + f3.length());
  if (f3.canRead()) System.out.println("I can read it");
  if (f3.canWrite()) System.out.println("I can write to it");
  if (f3.isHidden()) System.out.println("It is hidden");
  if (f3.delete()) System.out.println("I have deleted it!");
  try (OutputStream os = new FileOutputStream(f3)) { ... };
}
```

Also **directory** / **file system** operations:
listFiles(), mkdir(), renameTo(),
find available space, total space,
file system roots (C:\, D:\),

File.createNewFile() is used for atomic locking – just use a FileOutputStream in most cases!

# Object-based I/O: Serialization

jonas.kvarnstrom@liu.se – 2016

# Serialization 1: Intro



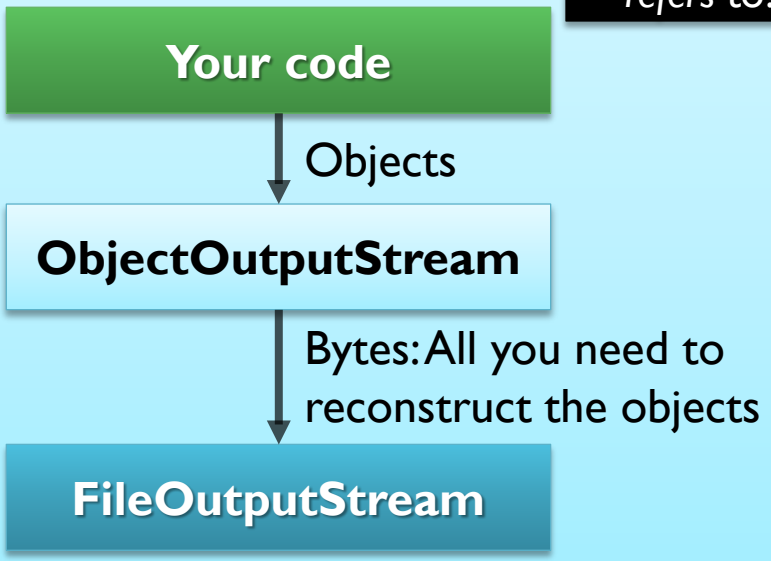
jonkv@ida

## Serialization: Convert objects to/from sequences of bytes

### ObjectOutputStream

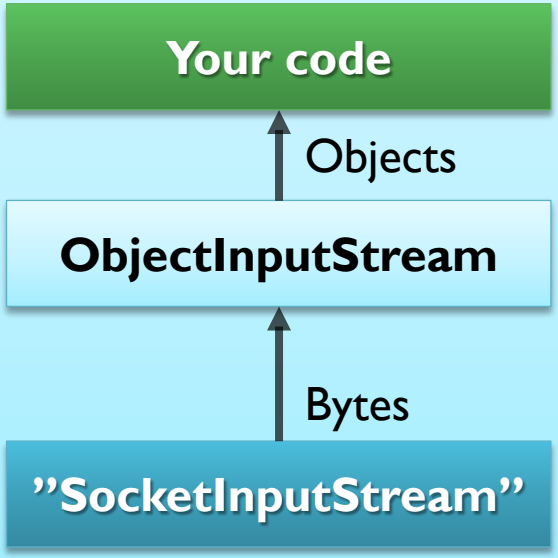
```
OutputStream os =
   new FileOutputStream("file.dat");
ObjectOutputStream out =
   new ObjectOutputStream(os);
out.writeObject(gameBoard);
out.writeObject(highscoreList);
out.close();
```

Writes an object and everything it *refers* to!




### ObjectInputStream

```
Socket sock = ...;
InputStream is = sock.getInputStream();
ObjectInputStream in =
   new ObjectInputStream(is);
List<Score> highscoreList =
   (List<Score>) in.readObject();
in.close();
```



# Serialization 2: Serializable Interface



jonkv@ida

- The objects **must implement java.io.*Serializable***
  - An interface *without methods*, indicating that serialization is allowed
    - By accessing the byte stream you can read private fields!
    - **public class** Pair **implements** *Serializable* {
      **private** Object first;
      **private** Object second;
      **private transient** int hashCodeCache;

      Pair(Object first, Object second) {
      **this**.first = first;
      **this**.second = second;
      }
      }

All *field values* must also be of primitive or Serializable types!

…except *transient* fields, which we assume can be reconstructed or are unnecessary for other reasons

  - The superclass must:
    - Be *Serializable*, so we are allowed to *save* its data to the stream, or
    - Have a constructor without arguments, so it can be *reconstructed* from scratch
  - Many Java classes already implement *Serializable*
    - Strings, Collections subclasses, …

# Serialization 3: Writing An Object Twice



jonkv@ida

- **ObjectOutputStream** must handle **circular references**
  - Node structure example:
    - ```
      Node parent = new Node(null);
      Node child  = new Node(parent); // child points to parent
      parent.addChild(node);          // parent points to child
      ```
  - Remembers which objects were written
    - First time: Write object ID + entire object representation
    - Second time: Write object ID
  - Does not care whether the object was updated!
    - ```
      List list = new ArrayList();
      oos.write(list);                // Writes object ID + entire list
      list.add("Another element");
      oos.write(list);                // Writes the object ID...
      ```
  - To write a new copy of the object: Use reset()
    - ```
      oos.reset();
      ```
      ```

# Serialization 4: Class Versions



jonkv@ida

- Can old saved objects be read after **changing the class**?
  - **Some changes are allowed**
    - Adding fields – if you read an old object, the field will be set to 0/null
    - Changing public/protected/private
    - A few more types of changes
  - **Others are forbidden**
    - Changing the class hierarchy in certain ways
    - Removing Serializable
    - …
  - You must have the same serial version ID
    - By default this is a hash of certain features in the class — **too strict**!
    - To allow adding new fields, declare your own version ID:
      `private final static long serialVersionUID = 1;  // for example`
    - IMPORTANT!  **Change** this if you make incompatible changes to your class!

# Serialization 5: Exceptions



jonkv@ida

- **Error handling** was omitted
  - ClassNotFoundException – received an object of a non-existing class
  - InvalidClassException
  - StreamCorruptedException – bad control information in the stream
  - OptionalDataException – primitive data found instead of objects
  - NotSerializableException – an object was not Serializable
  - IOException – the usual Input/Output related exceptions

- Many more serialization features…
  - http://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html