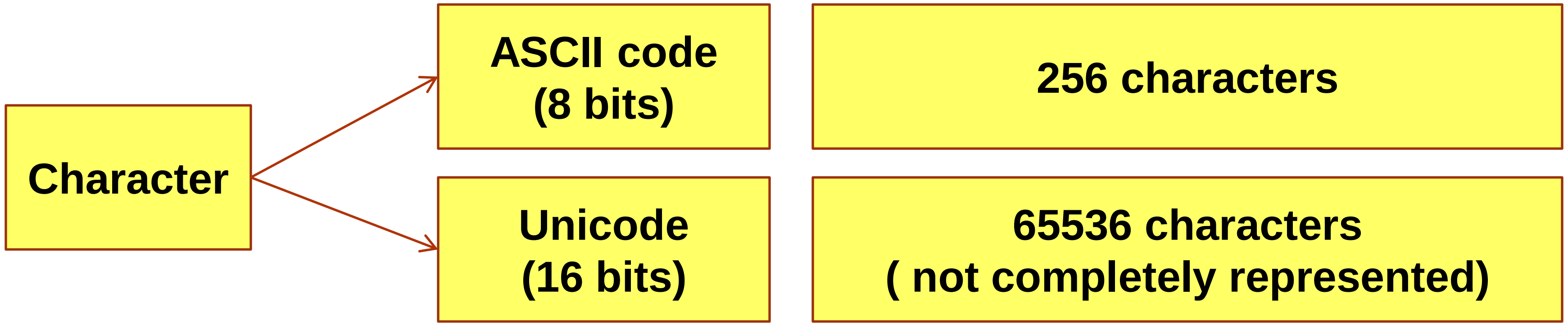
Session 06
IO and Streams

Objectives

- java.io package
  - Text, UTF, and Unicode
  - File Input and Output
  - Streams, Readers, and Writers
  - Object Streams and Serialization

Text, UTF, and Unicode



Unicode character: a character is coded using 16/32 bits

UTF: **<u>U</u>**niversal Character Set – UCS- **<u>T</u>**ransformation **<u>F</u>**ormat

UTF: *Unicode transformation format , a* Standard for compressing strings of Unicode text .

UTF-8: A standard for compressing Unicode text to 8-bit code units.

Refer to: http://www.unicode.org/versions/Unicode7.0.0/

Java :

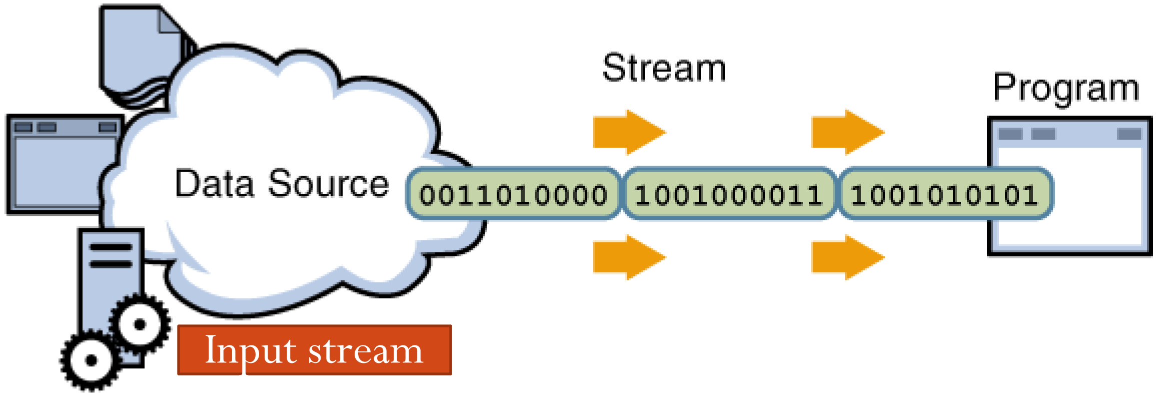
- Uses UTF to read/write Unicode
- Helps converting Unicode to external 8-bit encodings and vice versa.

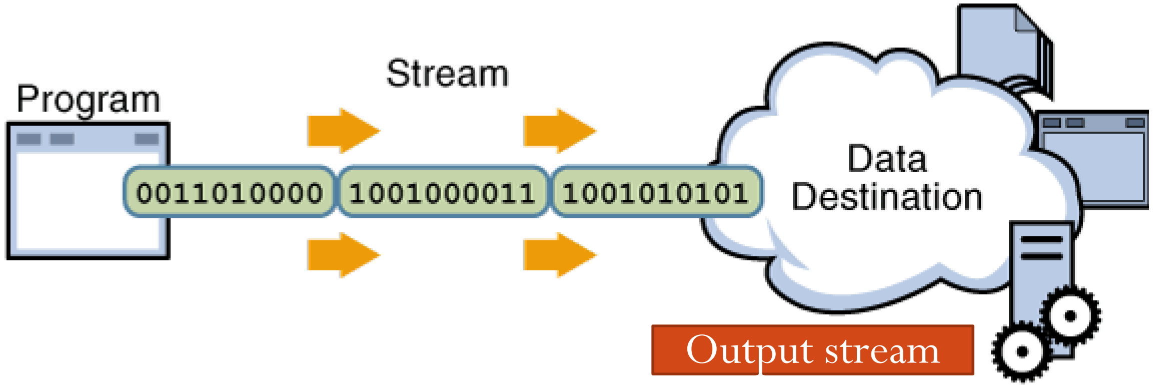
Text files

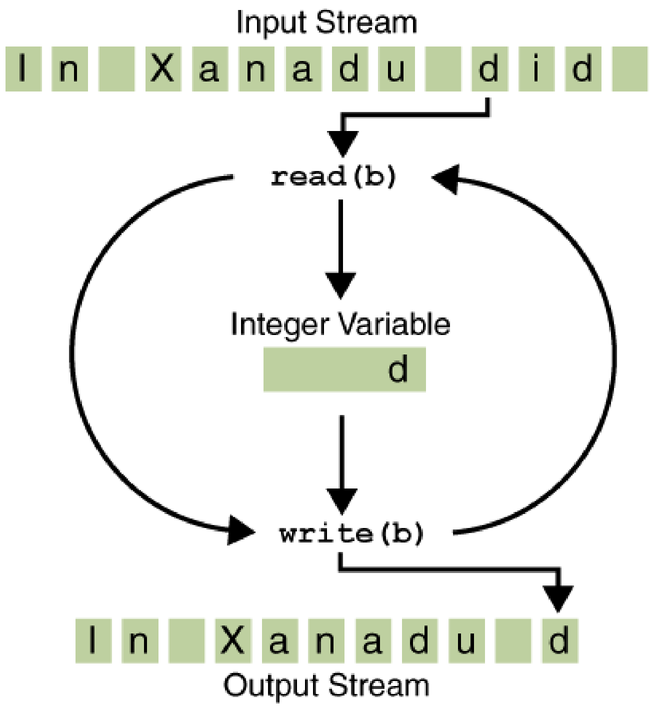
- Text (.txt) files are the simplest kind of files
  - text files can be used by many different programs
- Formatted text files (such as .doc files) also contain binary formatting information
- Only programs that “know the secret code” can make sense formatted text files
- Compilers, in general, work only with text.

Streams (1)

- A stream is an object managing a data source in which operations such as read data in the stream to a variable, write values of a variable to the stream associated with type conversions are performed automatically. These operations treat data as a chain of units (byte/character/data object) and data are processed in unit-by-unit manner.







Streams (2)

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the keyboard
- An input stream or an output stream may be associated with a file
- Different streams have different characteristics:
  - A file has a definite length, and therefore an end
  - Keyboard input has no specific end

How to do I/O

```
import java.io.*;
```

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

Why Java I/O is hard

- Java I/O is very powerful, with an overwhelming number of options
- Any given kind of I/O is not particularly difficult
- The trick is to find your way through the maze of possibilities
- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

Example of opening a stream

- A FileReader is a used to connect to a file that will be used for input:

  FileReader fr = new FileReader(fileName);
- The fileName specifies where the (external) file is to be found
- You never use fileName again; instead, you use fr

```
FileReader fr = new FileReader("test.txt");
int k; char c;
while(true) {
      k = fr.read();
      if(k == -1) break;
      c = (char) k;
       System.out.print(c);
   }
fr.close();
```

- The fr.read() method reads one character and returns it as an integer, or -1 if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

Using a stream

- Some streams can be used only for input, others only for output, still others for both
- *Using* a stream means doing input from it or output to it
- But it’s not usually that simple--you need to manipulate the data in some way as it comes in or goes out

Manipulating the input data

- Reading characters as integers isn’t usually what you want to do
- A BufferedReader will convert integers to characters; it can also read whole lines
- The constructor for BufferedReader takes a FileReader parameter.
- A BufferedReader will return null if there is nothing more to read.

```
FileReader fr = new FileReader("test.txt");
BufferedReader br = new BufferedReader(fr);
String s;
while((s = br.readLine()) != null)  System.out.println(s);
fr.close();
```

Closing

- A stream is an expensive resource
- There is a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You must close a stream before you can open it again
- *Always close your streams!*

How did I figure that out?

- I wanted to read lines from a file
- I found a readLine method in the BufferedReader class
- The constructor for BufferedReader takes a Reader as an argument
- An InputStreamReader is a kind of Reader
- A FileReader is a kind of InputStreamReader

The PrintWriter class

```
PrintWriter write = new
PrintWriter("test.txt");
int k = 5; float x = 12.7f; double y =
23.83;
writer.write("Hello there,\n");
writer.write(" here is some text.\n");
writer.write("We are writing");
writer.write(" the text to the file.");
write.flush();
write.close();
```

- Buffers are automatically flushed when the program ends normally
- Usually it is your responsibility to flush buffers if the program does not end normally
- PrintWriter can do the flushing for you.

File and RandomAccessFile classes

- Java's java.io.File and java.io.RandomAccessFile classes provide functionality for <u>navigating</u> the local file system, <u>describing files</u> and <u>directories</u>, and <u>accessing</u> files in <u>non-sequential</u> order.

The *File* Class

- File class represents the name of a file or directory that <u>might</u> exist on the host machine’s file system.
- Ctor:

  File(String pathname);
- Constructing an instance of File does <u>not</u> create a file on the local file system.

File methods

- boolean exists().
- String getAbsolutePath()
- String getCanonicalPath()
- String getName()
- String getParent()
- boolean isDirectory()
- boolean isFile()
- String[] list()
- boolean canRead()
- boolean canWrite()
- boolean createNewFile()
- boolean delete()
- long length()
- boolean mkdir()
- boolean renameTo(File *newname*)

File examples

```
File f = new File(fname);
if(f.exists())
    System.out.println("The file " + fname + " exists");
     else
      System.out.println("The file " + fname + " does not exist");
```

```
if(!f.exists()) f.createNewFile();
```

```
if(!f.exists()) f.delete();
```

```
File f = new File("data");
if(!f. isDirectory()) f.mkdir();
```

The *RandomAccessFile* Class

- With a random-access file, you can seek to a desired position within a file and then read or write a desired amount of data.
- The RandomAccessFile class provides methods that support seeking, reading, and writing.

RandomAccessFile consructors

- RandomAccessFile(String *file*, String *mode*)
- RandomAccessFile(File *file*, String *mode*)

Where:

- *mode:* should be either “r” or “rw”. Use “r” to open the file for reading only, and use “rw” to open for both reading and writing.

RandomAccessFile...

RAF offer the following functionality:

- Seeking to any position within a file
- Reading and writing single or multiple bytes
- Reading and writing groups of bytes, treated as higher-level data types
- Closing

RandomAccessFile methods

- long getFilePointer()
- long length()
- void seek(long *position*)
- int read()
- int read(byte *dest*[])
- int read(byte *dest*[], int *offset*, int *len*)
- void write(int *b*)
- void write(byte *b*[])
- void write(byte *b*[],int *offset*, int *len*)
- void writeBytes(String s)
- void close() throws IOException

RandomAccessFile methods…

Read Method	Write Method
`boolean readBoolean()`	`void writeBoolean(boolean b)`
`byte readByte()`	`void writeByte(int b)`
`short readShort()`	`void writeShort(int s)`
`char readChar()`	`void writeChar(int c)`
`int readInt()`	`void writeInt(int i)`
`long readLong()`	`void writeLong(long l)`
`float readFloat()`	`void writeFloat(float f)`
`double readDouble()`	`void writeDouble(double d)`
`int readUnsignedByte()`	None
`int readUnsignedShort()`	None
`String readLine()`	None
`String readUTF()`	`void writeUTF(String s)`

RandomAccessFile examples (1)

```
RandomAccessFile f = new
RandomAccessFile("test.txt","rw");
f.writeInt(5);
f.writeDouble(15.5);
f.writeBytes("ABC XYZ");
f.close();
f = new RandomAccessFile("test.txt","r");
int k; double x;String s;
k = f.readInt();
x = f.readDouble();
s = f.readLine();
f.close();
```

RandomAccessFile examples (2)

```
 RandomAccessFile f = new
RandomAccessFile(“vehicle.dat","r");
 String [] a; String s; int year; double cost; Vehicle v;
 while(true)
    {s = f.readLine();
      if(s==null || s.length()<3) break;
      a = s.split("\\s*\\;\\s*");
      if(a==null || a.length<4) break;
      year = Integer.parseInt(a[1]);
      cost = Double.parseDouble(a[2]);
      v = new Vehicle(a[0],year,cost,a[3]);
      list.add(v);
    }
f.close();
```

```
Madaz123; 2016; 1234; red
civic345;2017;234;yellow
toyota23; 2018;456;black
```

Streams, Readers, and Writers

- Java’s stream, reader, and writer classes view input and output as ordered sequences of bytes.
- Dealing strictly with bytes would be tremendously bothersome, because data appears sometimes as bytes, sometimes as ints, sometimes as floats, and so on.

Output

- A low-level output stream receives bytes and writes bytes to an output device.
- A high-level filter output stream receives general-format data, such as primitives, and writes bytes to a low-level output stream or to another filter output stream.
- A **writer** is similar to a filter output stream but is specialized for writing Java strings in units of Unicode characters.

Input

- A low-level input stream reads bytes from an input device and returns bytes to its caller.
- A high-level filter input stream reads bytes from a low-level input stream, or from another filter input stream, and returns general-format data to its caller.
- A reader is similar to a filter input stream but is specialized for reading <u>UTF strings</u> in units of Unicode characters.

Low-Level Streams

- *Low-level input streams*
  - FileInputStream
  - ByteArrayInputStream
  - ..
- *Low-level output streams*
  - FileOutputStream
  - ByteArrayOutputStream
  - ...

FileInputStream..

```
byte b;
byte bytes[] = new byte[100];
byte morebytes[] = new byte[50];
try {
    FileInputStream fis = new FileInputStream(“fname”);
    b = (byte) fis.read(); // Single byte
    fis.read(bytes); // Fill the array
    fis.read(morebytes, 0, 20); // 1st 20 elements
    fis.close();
}
catch (IOException e) { }
```

High-Level Streams

- The most common of these extend from the superclasses FilterInputStream and FilterOutputStream.
- Do not read/write from input/output devices such as files or sockets; rather, they read/write from other streams.

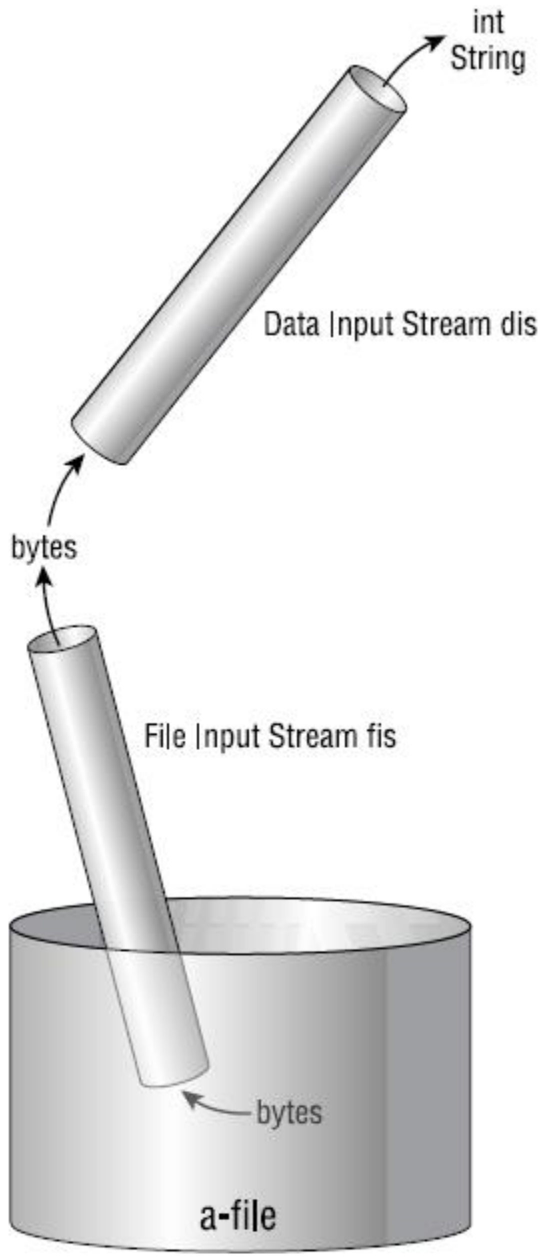
High-Level Streams…

- DataInputStream
- DataOutputStream
- BufferedInputStream
- BufferedOutputStream
- ….

DataInputStream Sample

```
FileInputStream in = null;
      DataInputStream datain = null;
      try {
            in = new FileInputStream(file);
            datain = new DataInputStream(in)
System.out.println(datain.readInt());
System.out.println(datain.readDouble());
  System.out.println(datain.readChar());
      } catch (IOException e) {
```

A chain of input streams



Readers and Writers

- *Readers* and *writers* are like input and output streams: The low-level varieties communicate with I/O devices, and the high-level varieties communicate with low-level varieties.
- ***Readers* and *writers* are exclusively oriented to Unicode characters.**

Readers and Writers..

- FileReader and FileWriter
- CharArrayReader and CharArrayWriter
- PipedReader and PipedWriter
- StringReader and StringWriter
- BufferedReader and BufferedWriter
- InputStreamReader and OutputStreamWriter
- LineNumberReader
- ..

Object Streams and Serialization

- Object streams go one step beyond data streams by allowing you to read and write <u>entire objects</u>.

Serialization

- You can also read and write *objects* to files
- Object I/O goes by the awkward name of serialization
- Serialization in other languages can be *very* difficult, because objects may contain references to other objects
- Java makes serialization (almost) easy

Conditions for serializability

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement **Serializable** interface
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects (static fields and transient fields are also not serialized)

Implementing the Serializable interface

- To “implement” an interface means to define all the methods declared by that interface, but...
- The Serializable interface does not define any methods!
  - Question: What possible use is there for an interface that does not declare any methods?
  - Answer: Serializable is used as flag to tell Java it needs to do extra work with this class

Writing objects to a file

```
ObjectOutputStream ou = new ObjectOutputStream(new
FileOutputStream(filename));
ou.writeObject(serializableObject);
ou.close( );
```

Reading objects from a file (De-serialization)

```
ObjectInputStream ob= new ObjectInputStream(new
FileInputStream(filename));

myObject = (itsType)ob.readObject( );

ob.close( );
```

Example (Vidu3DT)

Summary

- java.io package
  - Text, UTF, and Unicode
  - File Input and Output
  - Streams, Readers, and Writers
  - Object Streams and Serialization

Q&A