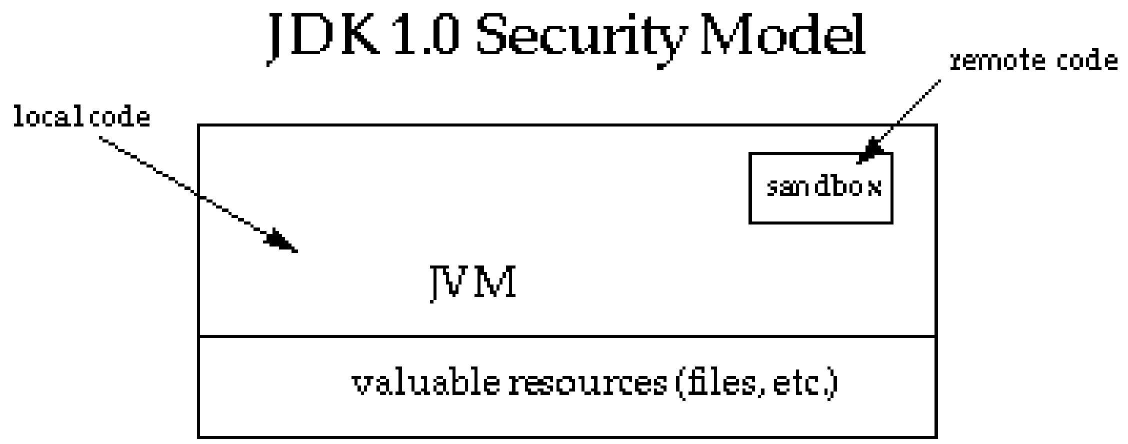
LearnQuest

Presents

Java File I/O

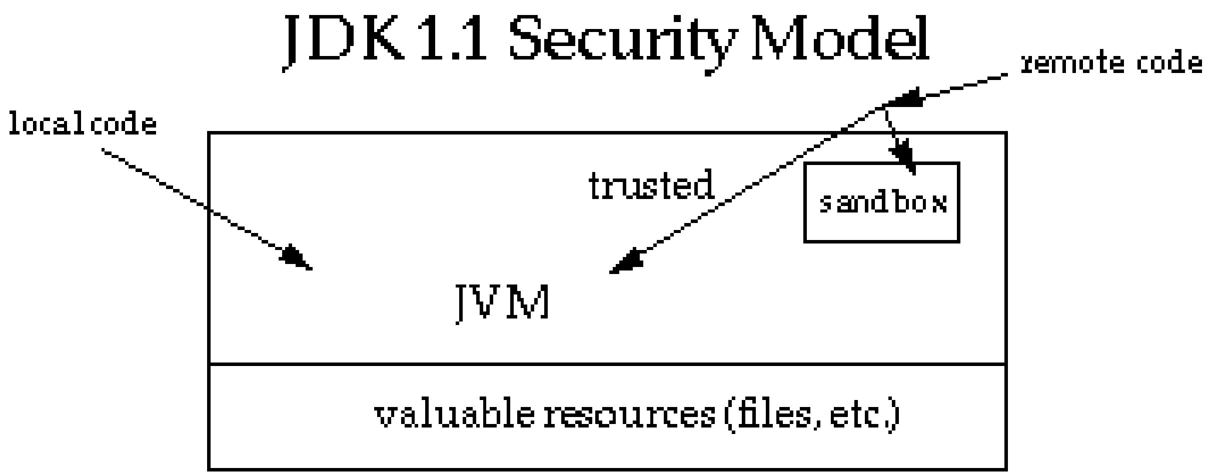
Initial Java I/O

- Java was originally intended as a browser engine
  - ✓ Designed to run in a sandbox in the browser for security
  - ✓ Unable to access the client file system or network
  - ✓ Only code run on the local file system could access local resources



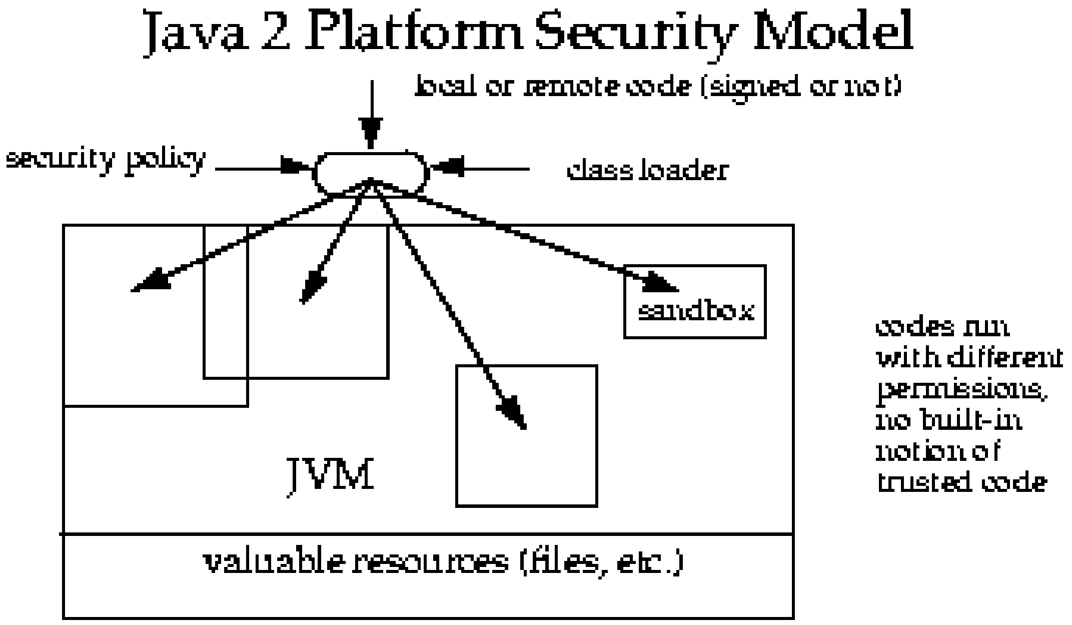
Modified Java I/O

- Added the idea of trusted remote code
  - ✓ Remote code vetted by the security manager could access local resources



Java 2 I/O

- Extended the security manager to check all code, whether local or remote
  - ✓ Local code is now also restricted by security policies
  - ✓ Additional tools to configure security policies


Java 2 Platform Security Model
local or remote code (signed or not)
security policy
class loader
sandbox
codes run with different permissions, no built-in notion of trusted code
JVM
valuable resources (files, etc.)

Streams

- Java uses a basic streams I/O model for most I/O operations
  - ✓ Data is accessed through a stream interface
  - ✓ Sources are places where data is read from
  - ✓ Sinks are places where data is written to
- The streams model is commonly used in many programming languages
  - ✓ Meets most of the needs for I/O
  - ✓ Random access read/write can be done in Java
  - ✓ Most CRUD functionality nowadays is handled by databases and data services instead of flat files

Stream Types

- Five basic streams types
- Byte Streams
  - ✓ Read or writes a file byte by byte – used for arbitrary data
- Character Streams
  - ✓ Reads and writes a file character by character
  - ✓ Characters represented by UTF formats have variable sizes
- Buffered Streams
  - ✓ Line oriented reading and writing
  - ✓ Standard functionality for reading text type files

Stream Types

- Five basic streams types

- Data Streams
  - ✓ Manages binary I/O of primitive data types and strings
  - ✓ Not covered in this class

- Object Streams
  - ✓ Manages the serialization of Java objects

Byte Streams

- Inputs and outputs data in 8-bit chunks
  - ✓ Uses the interfaces File`InputStream` and File`OutputStream`
- Requires files to be open prior to use
  - ✓ Throws IOExceptions if files cannot be accessed
  - ✓ This are checked exceptions and must be handled
- The basic `read()` and `write()` operations move one byte at a time
  - ✓ The `read()` operations returns a -1 on EOF

Byte Streams

```
try {
    infile = new FileInputStream("SampleText.txt");
    outfile = new FileOutputStream("Copy.txt");

    while ((b = (byte)infile.read()) ≠ -1) {
        outfile.write(b);
        byteCount++;
    }

} catch (IOException e) {
    System.out.println(e);

} finally {
    infile.close();
    outfile.close();

}
```

Character Stream

- Inputs and outputs data in single characters
  - ✓ Uses the interfaces File`Reader` and File`Writer`
- Manages conversion of bytes to characters
  - ✓ The type of text encoding is used to compute how many bytes are needed to read a character
  - ✓ The encoding defaults to the whatever the platform default is
  - ✓ As of Java 12, the encoding of the files can be specified

```
infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
outfile = new FileWriter("Copy.txt",StandardCharsets.UTF_8 );
```

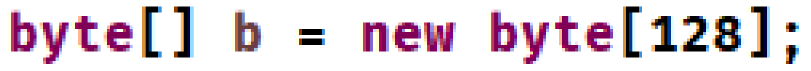
Charset Stream

- As of Java 12, Java understands the following character encodings.

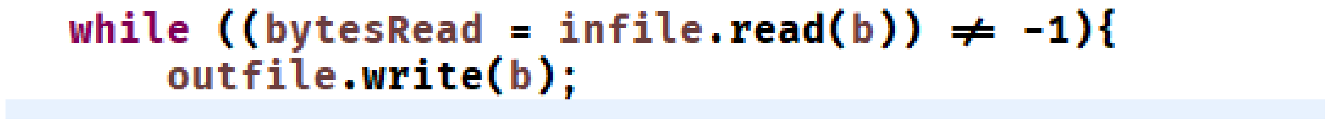
Charset	Description
US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Byte Array Stream

- The bytes streams can be read and written in chunks by defining a sized sixed buffer to be used.
  - ✓ For large files, this is more efficient than reading a single byte at a time
  - ✓ To do this, we use a different form of the read and write methods that take a reference to the buffer to be used.
  - ✓ The read method returns the number of bytes read.



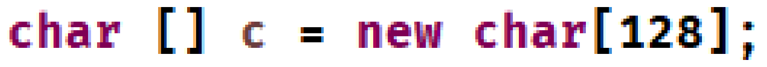
```
byte[] b = new byte[128];
```



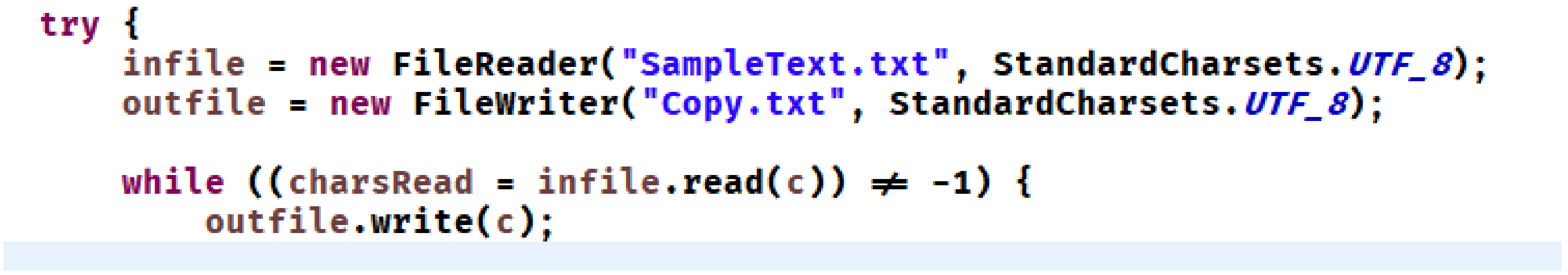
```
while ((bytesRead = infile.read(b)) ≠ -1){
    outfile.write(b);
```

Character Array Stream

- ▶ This is essentially the same a byte array
  - ✓ The only difference is the specification of the characterset and
  - ✓ The user of a char array instead of a byte array



```
char [] c = new char[128];
```



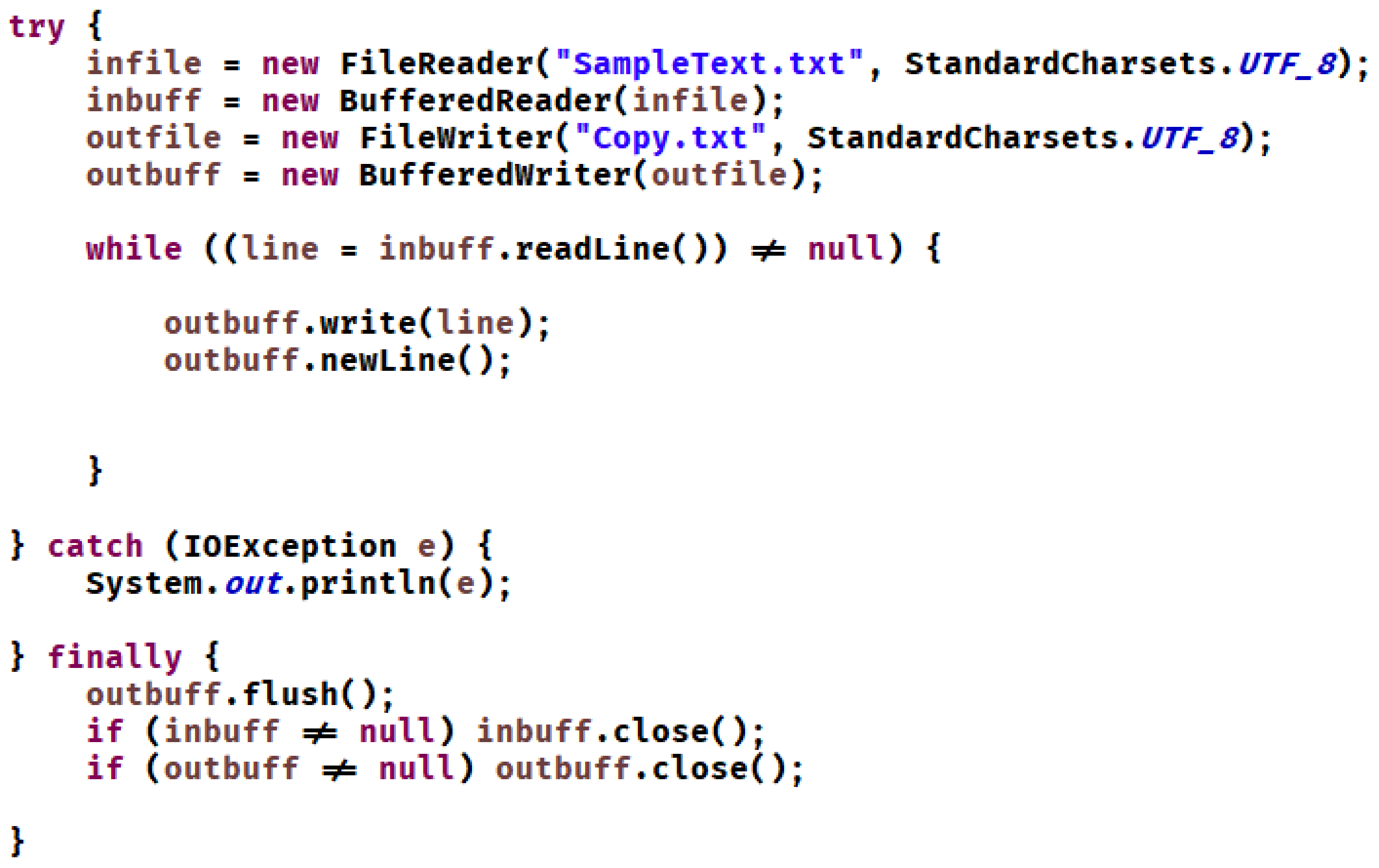
```
try {
    infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
    outfile = new FileWriter("Copy.txt", StandardCharsets.UTF_8);

    while ((charsRead = infile.read(c)) ≠ -1) {
        outfile.write(c);
```

Buffered Streams

- Java can do buffering so we don't have to
  - ✓ The FileReader and File Writer are wrapped in a either a BufferedReader of BufferedWriter
  - ✓ These are generally used for line oriented input
  - ✓ The translation of EOL characters is handled automatically;
- When using BufferedWriter
  - ✓ The buffer has to be flushed to force a write to the file
  - ✓ Otherwise what is in the buffer will not get written to disk

LearnQuest

Buffered IO



```
try {
    infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
    inbuff = new BufferedReader(infile);
    outfile = new FileWriter("Copy.txt", StandardCharsets.UTF_8);
    outbuff = new BufferedWriter(outfile);

    while ((line = inbuff.readLine()) ≠ null) {

        outbuff.write(line);
        outbuff.newLine();


    }

} catch (IOException e) {
    System.out.println(e);

} finally {
    outbuff.flush();
    if (inbuff ≠ null) inbuff.close();
    if (outbuff ≠ null) outbuff.close();

}
```

Questions