CS 251
Intermediate Programming
Java I/O – Streams

Brooke Chenoweth

University of New Mexico

Fall 2026

Basic Input/Output

- I/O Streams – mostly in `java.io` package
- File I/O – mostly in `java.nio.file` package

What is an I/O Stream?

- A stream is a *sequence of data*.
- Represents an input source or output destination. (Files, devices, other programs, etc.)
- Support different kinds of data (Bytes, primitive data types, characters, objects)
- Some streams simply pass on data, others transform it in useful ways.
- An *input stream* reads data from a source, one item at a time.
- An *output stream* writes data to a destination, one item at a time.

Byte Streams

- Byte streams perform input and output of 8-bit bytes.
- All byte stream classes extend `InputStream` or `OutputStream`
- Very primitive – usually won't use directly
- All other stream types are built on byte streams.

Byte Stream Example

```
import java.io.*;
public class CopyBytes {
  public static void main(String[] args)
                    throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("infile.txt");
      out = new FileOutputStream("outfile.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    } finally {
      if (in != null) { in.close(); }
      if (out != null) { out.close(); }
    }
  }
}
```

Character Streams

- Character streams read/write characters.
- Automatically translates to/from local character set.
- Character stream classes descend from `Reader` and `Writer`

Character Stream Example

```
import java.io.*;
public class CopyCharacters {
  public static void main(String[] args)
                              throws IOException {
    FileReader in = null;
    FileWriter out = null;
    try {
      in = new FileReader("infile.txt");
      out = new FileWriter("outfile.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    } finally {
      if (in != null) { in.close(); }
      if (out != null) { out.close(); }
    }
  }
}
```

Wrapping Byte Streams

- Character streams are often "wrappers" for byte streams.
- For example, `FileReader` uses `FileInputStream` to actually read the data, and then translates the bytes into characters.
- If you have a byte stream, but no character stream class (when using sockets for networking, for example), use general-purpose byte-to-character "bridge" streams:
  - `InputStreamReader`
  - `OutputStreamWriter`

Buffered I/O

- With *unbuffered* I/O streams, each read/write request is handled directly by underlying OS.
  - Inefficient
  - Expensive
- Better approach – *buffered* I/O streams.
  - Uses a *buffer* in memory to reduce calls to system.
  - Buffered input streams read data from a buffer, only call the native input API when the buffer is empty.
  - Buffered output streams write data to a buffer, and only call native output API when the buffer is full.
- Buffered streams also let us work with more data at once. Lines instead of characters, for example.

Line I/O Example

```
import java.io.*;

public class CopyLines {
  public static void main(String[] args)
                        throws IOException {
    BufferedReader in = null;
    PrintWriter out = null;
    try {
      in = new BufferedReader(new FileReader("infile.txt"));
      out = new PrintWriter(new FileWriter("outfile.txt"));
      String line;
      while ((line = in.readLine()) != null) {
        out.println(line);
      }
    } finally {
      if (in != null) { in.close(); }
      if (out != null) { out.close(); }
    }
  }
}
```

Always Close Streams

- Always close a stream when it is not longer needed.
- Avoids resource leaks.
- Need to close even if an error occurs. Use `finally` or try-with-resources.

try-with-resources Statement

- Java 7 introduced try-with-resources statement.
- A `try` statement that declares one or more resources.
- A *resource* is an object that must be closed when program is done with it.
- Any object that implements `java.lang.AutoCloseable` can be used as a resource.
- I/O Streams implement `java.io.Closeable`, which extends `AutoCloseable`

Line I/O Example with try-with-resources

```
import java.io.*;

public class CopyLines {
  public static void main(String[] args)
                                throws IOException {
    try (
         BufferedReader in =
           new BufferedReader(
              new FileReader("infile.txt"));
         PrintWriter out =
           new PrintWriter(
              new FileWriter("outfile.txt"))) {
      String line;
      while ((line = in.readLine()) != null) {
        out.println(line);
      }
    }
  }
}
```

Standard Streams

Java automatically defines three standard streams.

- Standard Input – `System.in`
- Standard Output – `System.out`
- Standard Error – `System.err`
- `System.out` and `System.err` are `PrintStream` objects.
- `System.in` is a byte stream. Wrap with `InputStreamReader` if you want to read characters.

Breaking up input with Scanner

- Construct `Scanner` by wrapping an `InputStream`
- The `Scanner` breaks down input into tokens.
- By default, tokens are separated by whitespace.
- Use `useDelimiter` method to specify different delimiter (argument is a regular expression)
- Can recognize and parse primitive types.
- Be sure to close when done!

Split Words Example

```
import java.io.*;
import java.util.Scanner;

public class SplitWords {
  public static void main(String[] args)
                              throws IOException {

    try (Scanner s =
            new Scanner(
              new BufferedReader(
                new FileReader("words.txt"))) {

      while (s.hasNext()) {
        System.out.println(s.next());
      }
    }
  }
}
```

Sum Numbers Example

```
import java.io.*;
import java.util.Scanner;
public class SumNumbers {
  public static void main(String[] args)
                          throws IOException {
    double sum = 0;
    try (Scanner s =
            new Scanner(
              new BufferedReader(
                new FileReader("numbers.txt")) {
      while (s.hasNext()) {
        if(s.hasNextDouble()) {
          sum += s.nextDouble();
        } else {
          s.next();
        }
      }
    }
    System.out.println(sum);
  }
}
```

Formatting

- Wrap a `PrintWriter` around an `OutputStream` to write formatted output.
- The `print` and `println` methods output single value after converting with `toString` method.
- The `format` method formats multiple arguments based on a *format string*.
  - Inspired by C's `printf` function.
  - Can specify numeric precision and alignment.
  - Can also format date/time.
  - See API for details.

Formatting example

```
public class RootDemo {
  public static void main(String[] args) {
    int i = 2;
    double r = Math.sqrt(i);

    System.out.format("Sqrt %d is %f.%n", i, r);
    System.out.format("Sqrt %d is about %.2f.%n", i, r);
  }
}
```

```
Sqrt 2 is 1.414214.
Sqrt 2 is about 1.41.
```

Data Streams

- Data streams support binary I/O of primitive types and Strings.
- All data streams implement `DataInput` or `DataOutput`
- Wrap a byte stream with `DataInputStream` or `DataOutputStream`
- Use `readInt`, `readDouble`, etc. for input.
- Use `writeInt`, `writeDouble`, etc. for output.

Object Streams

- Object streams support binary I/O of objects.
- Objects that can be serialized implement `Serializable` interface.
- Object stream classes are `ObjectInputStream` and `ObjectOutputStream`.
- Use `readObject` and `writeObject` to read/write objects.
- Object stream classes extend data stream classes, so can also read/write primitives.