I/O

Working With Files

Working With Files

- all classes in this lesson are in `java.io.*` package
- Java reads/writes either characters or bytes
- classes with *Stream* in their name read/write **binary**:

  (usually used for images, executable files, etc.)

  - `FileInputStream`
  - `FileOutputStream`
  - `ObjectInputStream`
  - `ObjectOutputStream`

Working With Files (cnt'd)

- Readers and Writers are used to read/write **text**:
  - `FileReader`
  - `BufferedReader`
  - `FileWriter`
  - `BufferedWriter`
  - `PrintWriter`
  - these are used for text files (data which consist of characters or strings)

Using File Classes

- using special class `File` enables one to create File objects
  - from which actual physical files on the hard disk can be created
- many of the classes mentioned in previous slides are intended to be *wrapped*
  - this enables low-level classes to get access to higher-level functionality
  - which results with efficiency (buffering)

Abstract class: InputStream

- Low-level class: `FileInputStream`
  - reads one byte at a time
- High-level class (for efficiency): `BufferedInputStream`
- High-level class (other): `ObjectInputStream`
- Wrapping example:

```
new ObjectInputStream(new FileInputStream("file.dat"));
```

Abstract class: OutputStream

- Low-level class: `FileOutputStream`
- High-level class (for efficiency): `BufferedOutputStream`
- High-level class (other): `ObjectOutputStream`
- Wrapping example:

```
new ObjectOutputStream(new FileOutputStream("file.dat"));
```

Abstract class: Reader

- Low-level class: `FileReader`
- High-level class (for efficiency): `BufferedReader`
- High-level class (other): `InputStreamReader`
  - a bridge between byte streams and character streams
- Wrapping examples:

```
new BufferedReader(new FileReader("in.txt"));
new BufferedReader(new InputStreamReader(System.in));
```

Abstract class: Writer

- Low-level class: `FileWriter`
- High-level class (for efficiency): `BufferedWriter`
- High-level classes (other): `OutputStreamWriter, PrintWriter`
  - a bridge between byte streams and character streams
- Wrapping examples:

```
new BufferedWriter(new FileWriter("out.txt"));
new BufferedWriter(new OutputStreamWriter(System.out));
```

```
// example: copy text file (no buffering)
```

loading physical files into Java objects

```
File srcFile = new File("C:\\Users\\Luka\\MyIOFiles\\source.txt");
File destFile = new File("C:\\Users\\Luka\\MyIOFiles\\destination.txt");
```

```
try(var reader = new FileReader(srcFile);
    var writer = new FileWriter(destFile)) {
    int c;
    while ((c = reader.read()) != -1) {
     writer.write(c);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

using try-with-resources to make sure that `close()` is applied on our objects

reading character by character with `read()` method (in Java, `-1` signals the end of the stream)

writing character by character with `write()` method

catching `IOException`

```
// example: copy text file (with buffering, same files)
try(var reader = new BufferedReader(new FileReader(srcFile));
    var writer = new BufferedWriter(new FileWriter(destFile))) {
   String line;
   while ((line = reader.readLine()) != null) {
     writer.write(line);
     writer.newLine();
   }
} catch (IOException e) {
   e.printStackTrace();
}
```

wrapping `FileReader` into `BufferedReader`

we use `readLine()` method, specific to `BufferedReader`

we use `newLine()` because `readLine()` strips out the end-of-line character

```
// example: copy binary file (no buffering)
File srcFile = new File("C:\\Users\\Luka\\MyIOFiles\\source.dat");
File destFile = new File("C:\\Users\\Luka\\MyIOFiles\\destination.dat");

try(var in = new FileInputStream(srcFile);
    var out = new FileOutputStream(destFile)) {
    int b;
    while ((b = in.read()) != -1) {
      out.write(b);
    }
} catch (IOException e) {
   e.printStackTrace();
}
```

```
// example: copy binary file (with buffering, same files)
try(var in =  new BufferedInputStream(new FileInputStream(srcFile));
    var out = new BufferedOutputStream(new FileOutputStream(destFile))) {
   var buffer = new byte[1024];
   int numBytesRead;
   while ((numBytesRead = in.read(buffer)) > 0) {
      out.write(buffer, 0, numBytesRead);
      out.flush();
   }
} catch (IOException e) {
   e.printStackTrace();
}
```

reads buffer (up to 1024 bytes) and stores it in `buffer`, and returns number of bytes read

write bytes from buffer, from 0 to buffer length

used if we want to ensure the data is written immediately