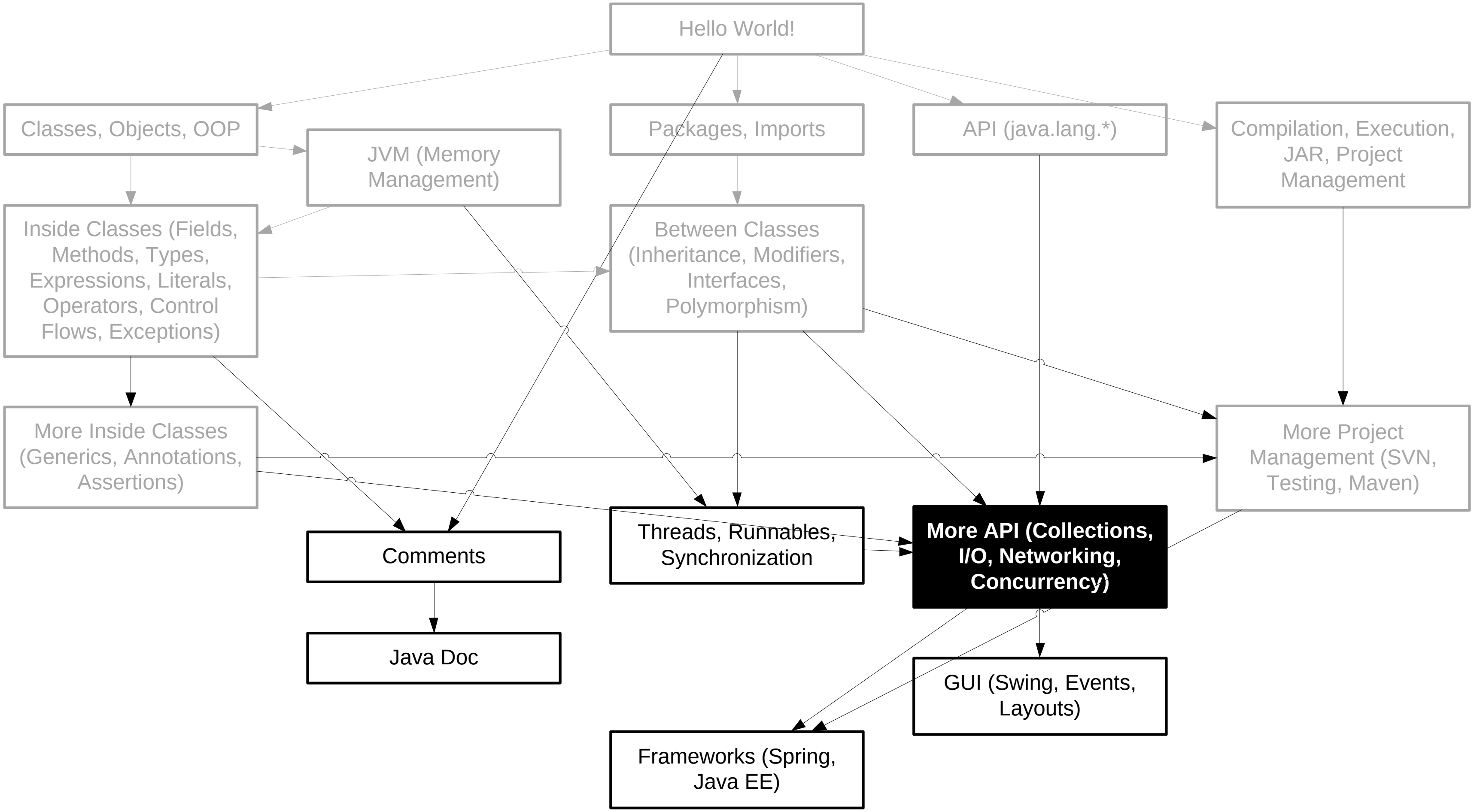
Java I/O

Shan-Hung Wu

CS, NTHU,

Fall, 2013

Where Are We?


Hello World!
Classes, Objects, OOP
JVM (Memory Management)
Packages, Imports
API (java.lang.*)
Compilation, Execution, JAR, Project Management
Inside Classes (Fields, Methods, Types, Expressions, Literals, Operators, Control Flows, Exceptions)
Between Classes (Inheritance, Modifiers, Interfaces, Polymorphism)
More Inside Classes (Generics, Annotations, Assertions)
More Project Management (SVN, Testing, Maven)
Comments
Threads, Runnables, Synchronization
More API (Collections, I/O, Networking, Concurrency)
Java Doc
GUI (Swing, Events, Layouts)
Frameworks (Spring, Java EE)

Why I/O?

- I/O is one of the most common tasks for every programmer
- Main packages:
  - `java.io`: traditional API that provides high-level abstractions for paths, random access files, and streams
  - `java.nio`: complement `java.io` with low-level abstractions and asynchronous access

Assigned Readings

- For each class/method mentioned in this lecture, read the corresponding document in Java API spec

Agenda

- Paths, Files, and Directories
  - java.io.File
  - java.io.RandomAccessFile
- Streams
  - Byte Streams
  - Buffered Streams
  - Character Streams and Formatting
  - Data Streams
  - Object Streams
- New I/O
  - Buffers and Channels
  - More File I/O

Paths

- `java.io.File` provides a system-independent view of hierarchical **paths** (which may not exist)
- Represents either files or directories:

```
File parent = new File("/bin"); // a directory
File child = new File(parent, "java.exe"); // a file
```

  - '/' works on all platforms, including Windows
  - Paths are relative to the working directory if not start with '/'
- Usage:
  - Check for existence and permissions: `exist()`,`canWrite()` etc.
  - Query various info: `isFile()`,`length()`,`lastModified()`,etc.
  - Create, rename or delete files and directories: `createNewFile()`, `createTempFile()`,`mkdir()`,`renameTo()`, `delete()`, `deleteOnExit()`,etc.
- Static fields `File.separator` and `File.pathSeparator` provide quick access to system-dependent separators

Tips for Path Manipulation

- To obtain a temporary file:

```
File temp = File.createTempFile("temp-", ".jpg", "/dir");
temp.deleteOnExit(); // optional
```

  – Each temp file will be guaranteed to have a unique name, e.g., `temp-25640856485650506 4.jpg`

- Enumerate Windows/Unix roots:

```
File[] winDrives = File.listRoots(); // for win
File unixRoot = File.listRoots()[0]; // for unix
```

- Enumerate files in a directory:

```
File[] files = new File("some dir").listFiles();
```

Random Access Files

- `java.io.RandomAccessFile` provides an abstraction to random accessible files (usually on local disks)

```
File path = ...
RandomAccessFile rf = new RandomAccessFile(path, "rws");
rf.seek(10); // set pointer at which next read/write occurs
int b = rf.read(); // pointer is advanced automatically
rf.seek(10);
rf.write(++b);
```

- Mode: "r", "rw", "rws"
  - 's' ensures that every write is synchronized to disk
- Acts like a large array of bytes
  - Has a **file pointer** that can be read and moved: `getFilePointer(),seek()`

RandomAccessCopier

```
public class RandomAccessCopier implements FileCopier {
  @Override
  public void copy(File src, File dest) throws IOException {
    RandomAccessFile srcRa = null, destRa = null;
    try {
      srcRa = new RandomAccessFile(src, "r");
      destRa = new RandomAccessFile(dest, "rws");
      int b = -1;
      while ((b = srcRa.read()) != -1) {
        destRa.write(b);
      }
    } finally {
      if (srcRa != null) srcRa.close();
      if (destRa != null) destRa.close();
    }
  }
}
```

Performance Issues

- `destRa` (with mode "rws") ensures that each call to `write()` is reflected to disk
- However, disks are very slow
  - Typically, the access time of RAM is about 60 ns
  - 1,000 times faster than flash drives (~60 us)
  - 100,000 times faster than magnetic drives (~6 ms)
- The performance of `RandomAccessCopier` is poor
- Can you make it faster?

BufferedRandomAccessCopier

```java
public class BufferedRandomAccessCopier implements FileCopier {
  private static final int BUFFER_SIZE = 8192; // in bytes

  @Override
  public void copy(File src, File dest) throws IOException {
    RandomAccessFile srcRa = null, destRa = null;
    byte[] buffer = new byte[BUFFER_SIZE];
    try {
      srcRa = new RandomAccessFile(src, "r");
      destRa = new RandomAccessFile(dest, "rws");
      int num = -1;
      while ((num = srcRa.read(buffer)) != -1) {
        destRa.write(buffer, 0, num);
      }
    } finally {
      if (srcRa != null) srcRa.close();
      if (destRa != null) destRa.close();
    }
  }
}
```

Agenda

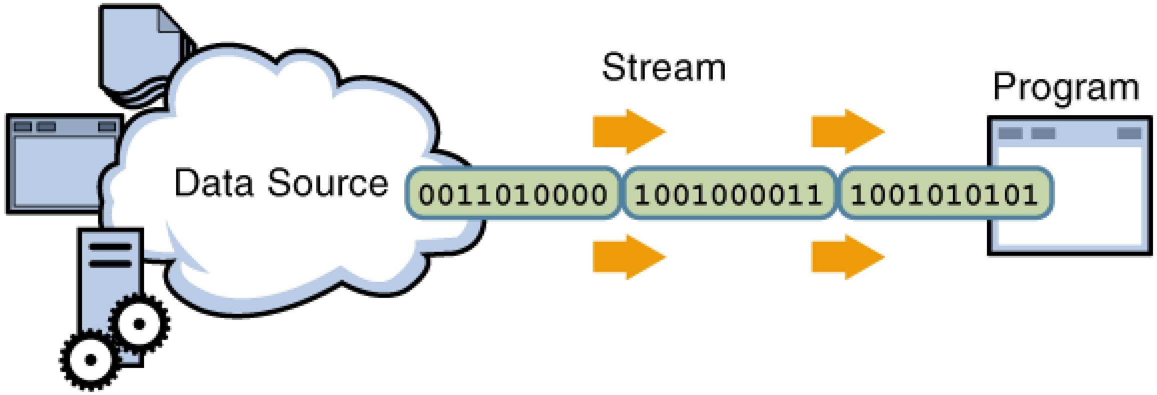
- Paths, Files, and Directories
  - java.io.File
  - java.io.RandomAccessFile
- Streams
  - Byte Streams
  - Buffered Streams
  - Character Streams and Formatting
  - Data Streams
  - Object Streams
- New I/O
  - Buffers and Channels
  - More File I/O

Streams (1/2)

- `File` and `RandomAccessFile` corresponding to paths and files in your file system
  - They are API specific to **file I/O**
- **General I/O** can read/write bytes from/to any device other than disks
  - E.g., network interface card, console, etc.
- Random access may not be possible!

Streams (2/2)

- `java.io.InputStream/OutputStream` provides an abstraction of **streams**



- Streams are independent of the nature of data sources
  - Bytes can be read/written sequentially by calling `read()` and `write()` repeatedly

ByteStreamCopier

```
public class ByteStreamCopier implements FileCopier {
  @Override
  public void copy(File src, File dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
      in = new FileInputStream(src);
      out = new FileOutputStream(dest);
      int b;
      while ((b = in.read()) != -1) {
        out.write(b);
      }
    } finally {
      if (in != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

Types of Streams

- Two main categories in Java:
  - Byte streams: `InputStream`/`OutputStream`
  - Character streams: `Reader`/`Writer`
- Endpoint streams that read/write directly from/to devices:
  - E.g., `FileInputStream`, `FileReader`, `ByteArrayInputStream`, `StringReader`, etc.
- Wrapper streams that read/write from/to endpoint streams:
  - E.g., `BufferedOutputStream`, `BufferedWriter`, `DataOutputStream`, `ObjectOutputStream`, `PrintWriter` etc.

BufferedByteStreamCopier

```
public class BufferedByteStreamCopier implements FileCopier {
  @Override
  public void copy(File src, File dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
      in = new BufferedInputStream(new FileInputStream(src));
      out = new BufferedOutputStream(new FileOutputStream(dest));
      int b;
      while ((b = in.read()) != -1) {
        out.write(b);
      }
      // (optional) make sure data are reflected to disk
      out.flush();
      ...
    } finally {
      if (in != null) in.close();
      if (out != null) out.close(); // flush first
    }
  }
}
```

CharStreamCopier

```
public class CharStreamCopier implements FileCopier {
  private static final String CHAR_SET = "UTF-8";
  @Override
  public void copy(File src, File dest) throws IOException {
    Reader r = null;
    Writer w = null;
    try {
      r = new InputStreamReader(
            new FileInputStream(src), CHAR_SET);
      w = new OutputStreamWriter(
            new FileOutputStream(dest), CHAR_SET);
      int c; // char as int
      while ((c = r.read()) != -1) {
        w.write(c);
      }
    } finally {
      if (r != null) r.close();
      if (w != null) w.close();
    }
  }
}
```

- Java also provides `FileReader/FileWriter`, but they support default encoding only

LinedCharStreamCopier

```java
public class LinedCharStreamCopier implements FileCopier {
  private static final String CHAR_SET = "UTF-8";
  @Override
  public void copy(File src, File dest) throws IOException {
    BufferedReader br = null;
    BufferedWriter bw = null;
    try {
      br = new BufferedReader(
            new InputStreamReader(
             new FileInputStream(src), CHAR_SET));
      bw = new BufferedWriter(
            new OutputStreamWriter(
             new FileOutputStream(dest), CHAR_SET));
      String s;
      while ((s = br.readLine()) != null) { // system independent
        bw.write(s);
      }
    } finally {
      if (br != null) br.close();
      if (bw != null) bw.close();
    }
  }
}
```

Formatting

- Char streams simplify reading/writing lines
  - System independent; works either for "\r", "\r\n", or "\n"
- Another advantage: formatting

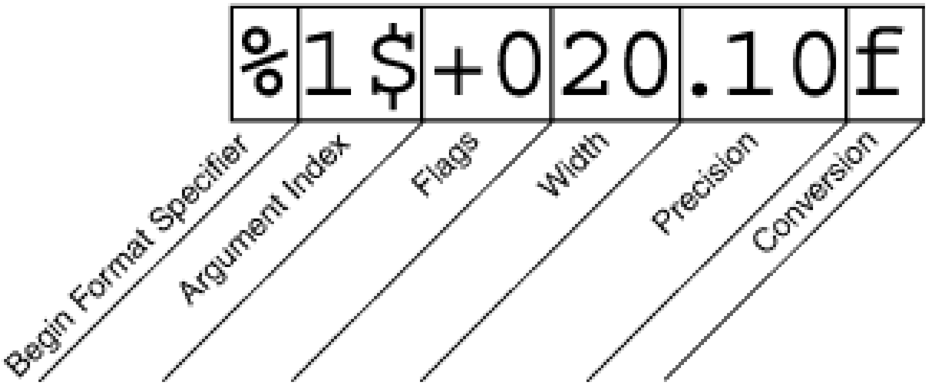
```
PrintWriter pw = new PrintWriter(/* some Writer */);
// or
PrintStream pw = new PrintStream(/* some OutputStream */,
     ..., /* charset */)'
pw.format("The square root of %d is %f. Correct?%n",
     9, Math.sqrt(9));
...
System.out.format("Current time at %s is %tc.%n",
     "Taipei", System.currentTimeMillis());
```

- Format specifier: `%`
- Conversion: `n` (line terminator), `d` (decimal), `f` (floating point), `s` (string), `tH` (hour), `tM` (minute), `tc` (date/time), etc.

Formatting (2/2)

```
1. System.out.format("<%f, %1$+020.10f>)%n", Math.PI);
   // output: <3.141593, +00000003.1415926536>
```

- Flags:
  - `+` (signed)
  - `0/'  '` (0/space-padded)
  - `-` (right padding)
  - `,` (thousand separated)



- Width: minimum width; padded if necessary
- Precision: for floating points; truncated at right
- Read this format syntax API spec for more details

DataStream

- `java.io.DataInputStream`/`DataOutputStream` allows reading/writing primitive and String values from/to streams in a portable way
  - `writeBoolean()`, `writeInt()`, `wrtieDouble()`, `writeChar()`, `writeUTF()`, etc.

ObjectStream

- `java.io.DataInputStream/DataOutputStream` allows reading/writing primitives **and objects** from/to streams

```
ObjectOutputStream oos = new ObjectOutputStream(...);
oos.writeObject("Today");
oos.writeObject(new Date());
ClassA a1 = new ClassA(...);
oos.writeObject(a1);
oos.close();
...
ObjectInputStream ois = new ObjectInputStream(...);
String s = (String) oos.readObject();
Date date = (Date) oos.readObject();
ClassA a2 = (ClassA) oos.readObject();
ois.close();
...
System.out.println(a1 == a2); // true or false?
```

Object Serialization (1/2)

- We say objects are **deserialized**/**serialized** upon reading/writing from streams
  - Unlike cloning, serialization is deep; all fields will be serialized recursively
  - The only exceptions are those fields declared with the `transient` modifier
- `ObjectInput/OutputStream` provides a convenient way to perform deep cloning:

```
ObjectOutputStream oos = new ObjectOutputStream(
      new ByteArrayOutputStream()); // endpoint is memory
ClassA obj = new ClassA(...);
oos.writeObject(obj);
ObjectInputStream ois = new ObjectInputStream(
      new ByteArrayInputStream(oos.toByteArray()));
ClassA deepCloneObj = (ClassA) ois.readObject();
...
oos.close();
ois.close();
```

Object Serialization (2/2)

- An object is **not** serializable unless its class implements `java.io.Serializable`

```
public ClassA implements Serializable {
  private static final long serialVersionUID = 1L;
  ...
}
```

- What happens if you serialize an object of ClassA, modify ClassA, and then deserialize the object?
  - A serializable class should define a field named `serialVersionUID`
  - Increment `serialVersionUID` if you think the modification of `CalssA` makes the previously serialized objects incompatible with the new ones

Agenda

- Paths, Files, and Directories
  - java.io.File
  - java.io.RandomAccessFile
- Streams
  - Byte Streams
  - Buffered Streams
  - Character Streams and Formatting
  - Data Streams
  - Object Streams
- New I/O
  - Buffers and Channels
  - More File I/O

Channels and Buffers

- `java.nio` offers a new way for I/O: `Channel`s and `Buffer`s
- All data operations are performed on Buffers (data blocks in memory)
  - There are buffers for all primitive types, like `ByteBuffer`, `LongBuffer`, `FloatBuffer`, etc.
  - Buffers can be allocated directly by OS, saving memory copying from user space to kernel space before each I/O operation
  - Buffers maintain their capacity, limit, and position, thus simplifying your code
- Channels are like streams, but they
  - Are bi-directional
  - Read/write data from/into Buffers only

Buffer Positions

- Buffers maintain their own positions
  - 0 <= mark <= position <= limit <= capacity
- `clear()`: sets the limit to the capacity and the position to zero
  - Makes the buffer ready for channel-read
- `flip()`: sets the limit to the current position and then sets the position to zero
  - Makes the buffer ready for channel-write
- `rewind()`: leaves the limit unchanged and sets the position to zero
  - Makes the buffer ready for re-reading the data that it already contains

ChannelCopier

```java
public class ChannelCopier implements FileCopier {
  private static final int BUFFER_SIZE = 8192; // in bytes
  private ByteBuffer buffer = ByteBuffer.allocateDirect(BUFFER_SIZE);
  @Override
  public void copy(File src, File dest) throws IOException {
    FileChannel ic = null, oc = null;
    try {
      ic = new FileInputStream(src).getChannel();
      oc = new FileOutputStream(dest).getChannel();
      while (ic.read(buffer) != -1) {
        buffer.flip();
        oc.write(buffer);
        buffer.clear();
      }
    } finally {
      if (ic != null) ic.close();
      if (oc != null) oc.close();
    }
  }
}
```

More File I/O

- In Java 7, `java.nio` is further extended to support advanced file I/O operations
  - Recursive file traversal
  - Manipulating symbolic/hard links
  - Searching files with wildcards, etc.
- Here is a mapping between the new and old file I/O APIs
- Optional reading: File I/O in Java 7