Principles of Software Construction: Objects, Design, and Concurrency

Design Case Study: Stream I/O

Some answers…

Charlie Garrod    Jonathan Aldrich
A challenge for you

• **Identify the design patterns in this lecture**
  - For each design pattern you recognize, write:
    - The class name
    - The design pattern
    - If you have time: At least one design goal or principle achieved by the pattern in this context
  - **Hints:**
    - Use the slides online to review the lecture
    - Design patterns include at least:
      - Adapter
      - Decorator
      - Iterator
      - Marker Interface
      - Template Method
The stream abstraction

- A sequence of **bytes**
- May read 8 bits at a time, and close

```java
java.io.InputStream
    void close();
    abstract int read();
    int read(byte[] b);
```

- May write, flush and close

```java
java.io.OutputStream
    void close();
    void flush();
    abstract void write(int b);
    void write(byte[] b);
```
The stream abstraction

• A sequence of **bytes**

• May read 8 bits at a time, and close

```java
java.io.InputStream
void close();
abstract int read();
int read(byte[] b);
```

• May write, flush and close

```java
java.io.OutputStream
void close();
void flush();
abstract void write(int b);
void write(byte[] b);
```

Template Method

Iterator?
The reader/writer abstraction

- A sequence of **characters** in some encoding
- May read one character at a time and close

**java.io.Reader**
```java
void close();
abstract int read();
int read(char[] c);
```

- May write, flush and close

**java.io.Writer**
```java
void close();
void flush();
abstract void write(int c);
void write(char[] b);
```
The reader/writer abstraction

- A sequence of **characters** in some encoding
- May read one character at a time and close

**java.io.Reader**

```java
void close();
abstract int read();
int read(char[] c);
```

**java.io.Writer**

```java
void close();
void flush();
abstract void write(int c);
void write(char[] b);
```

**Template Method**

**Iterator?**
Implementing streams

- **java.io.FileInputStream**
  - Reads from files, byte by byte

- **java.io.ByteArrayInputStream**
  - Provides a stream interface for a byte[]

- Many APIs provide streams for network connections, database connections, ...
  - e.g., java.lang.System.in, Socket.getInputStream(), Socket.getOutputStream(), ...
Implementing streams

- `java.io.FileInputStream`
  - Reads from files, byte by byte

- `java.io.ByteArrayInputStream`
  - Provides a stream interface for a `byte[]`

- Many APIs provide streams for network connections, database connections, ...
  - e.g., `java.lang.System.in`, `Socket.getInputStream()`, `Socket.getOutputStream()`, ...

Adapter
Implementing readers/writers

- **java.io.InputStreamReader**
  - Provides a Reader interface for any InputStream, adding additional functionality for the character encoding
    - Read characters from files/the network using corresponding streams

- **java.io.CharArrayReader**
  - Provides a Reader interface for a char[]

- Some convenience classes: FileReader, StringReader, ...
Implementing readers/writers

- **java.io.InputStreamReader**
  - Provides a Reader interface for any InputStream, adding additional functionality for the character encoding
  - Read characters from files/the network using corresponding streams

- **java.ioCharArrayReader**
  - Provides a Reader interface for a char[]

- **Some convenience classes**: FileReader, StringWriter, ...
A better design to add functionality to streams

**OutputStream**
- `write()`
- `close()`
- `flush()`

**FileOutputStream**
- `-file`
- `+write()`
- `+close()`
- `+flush()`

**ByteArrayOutputStream**
- `-buffer`
- `+write()`

**FilterOutputStream**
- `+write()`
- `+close()`
- `+flush()`

**GZipOutputStream**
- `+compress()`

**AESEncryptionOutputStream**
- `+encrypt()`

**DataOutputStream**
- `+writeInt()`
- `+writeString()`
- `+writeFloat()`

Delegating all calls to other output stream.
A better design to add functionality to streams

**Decorator**

- delegating all calls to other output stream

**OutputStream**

- «interface»
- +write()
- +close()
- +flush()

**FileOutputStream**

- +write()
- +close()
- +flush()

**ByteArrayOutputStream**

- +write()

**FilterOutputStream**

- +write()
- +close()
- +flush()

**GZipOutputStream**

- +compress()

**AESEncryptionStream**

- +encrypt()

**DataOutputStream**

- +writeInt()
- +writeString()
- +writeFloat()
To read and write arbitrary objects

- Your object must implement the `java.io.Serializable` interface
  - Methods: none

- If all of your data fields are themselves `Serializable`, Java can automatically serialize your class
  - If not, will get runtime `NotSerializableException`

- Can customize serialization by overriding special methods

See QABean.java and FileObjectExample.java
To read and write arbitrary objects

- Your object must implement the `java.io.Serializable` interface
  - Methods: none

- If all of your data fields are themselves `Serializable`, Java can automatically serialize your class
  - If not, will get runtime `NotSerializableException`

- Can customize serialization by overriding special methods

See QABean.java and FileObjectExample.java
The java.util.Scanner

• Provides convenient methods for reading from a stream

java.util.Scanner:

  Scanner(InputStream source);
  Scanner(File source);
  void close();
  boolean hasNextInt();
  int nextInt();
  boolean hasNextDouble();
  double nextDouble();
  boolean hasNextLine();
  String nextLine();
  boolean hasNext(Pattern p);
  String next(Pattern p);
...


The `java.util.Scanner`

- Provides convenient methods for reading from a stream

`java.util.Scanner`:

```
Scanner(InputStream source);
Scanner(File source);
void close();
boolean hasNextInt();
int nextInt();
boolean hasNextDouble();
double nextDouble();
boolean hasNextLine();
String nextLine();
boolean hasNext(Pattern p);
String next(Pattern p);
...
```