Principles of Software Construction: Objects, Design and Concurrency

Design Case Study: Stream I/O

Christian Kästner

Charlie Garrod

© 2014 C Kästner, C Garrod, J Aldrich, and W Scherlis
Learning Goals

• Understand the design aspects of the Stream abstractions in Java

• Understand the underlying design patterns
  ▪ Adapter
  ▪ Decorator
  ▪ Template Method
  ▪ Iterator
Design Challenge

• Identify a generic and uniform way to handle I/O in programs
  ▪ Reading/writing files
  ▪ Reading/writing from/to the command line
  ▪ Reading/writing from/to network connections

• Reading bytes, characters, lines, objects, ...

• Support various features
  ▪ Buffering
  ▪ Encoding (utf8, iso-8859-15, ...)
  ▪ Encryption
  ▪ Compression
  ▪ Line numbers

• Refer to files
  ▪ Paths, URLs, symbolic links, directories, files in .jar containers, searching, ...
Streams and Readers
The Stream abstraction

• A sequence of **bytes**

• May read, 8 bit at a time, and close (see Iterator)

```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 Reader/Writer abstraction

• A sequence of **characters** (given a character encoding)

• May read, one by one (see Iterator), and close

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

• May write, flush and close

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

- A sequence of **characters** (given a character encoding)
- May read, one by one (see Iterator), and close

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

- May write, flush and close

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

Notice the template method pattern
The Template Method design pattern

- **Applicability**
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations

- **Consequences**
  - **Code reuse**
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses

You may have used this in your virtual world
Implementing Streams

- `java.io.FileInputStream`
  - reads from files, returns input byte by byte

- `java.io.ByteArrayInputStream`
  - **Adapter** to provide the stream interface for a byte[] array

- `java.io.StringBufferedInputStream`
  - **Adapter** to provide the stream interface for a string (deprecated, due to character conversion issues, use Reader instead)

- Many network devices provide streams for URLs, database connections, etc

- `java.lang.System.in` provides an `InputStream` for the standard input
Implementing Readers/Writers

- We could provide FileInputStreamReaders and StringReaders and ByteArrayReaders and StandardInputReader, but would replicate functionality of streams.

- Instead provide adapter for stream interface; supports streams from arbitrary sources.

  - java.io.InputStreamReader
    - **Adapter** from any InputStream to Reader (adds additional functionality of the character encoding).
    - Read characters from files/the network using corresponding streams.

  - java.io.CharArrayReader
    - **Adapter** from char[] array to Reader interface.
Readers and Streams

```
Client «interface» Reader
InputStreamReader «interface» InputStream
FileInputStream «interface» InputStream
File
```

```
read (characters)
read (bytes)
```
Adding Features
Common Functionality on Streams

- Encryption
- Compression
- Buffering
- Reading of objects, numbers, lists, ...

```
+compress()
GZipOutputStream
```

```
«interface»
OutputStream
```

```
+encrypt()
AESEncryptionStream
```

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

```
«interface»
OutputStream
```

```
+encrypt()
AESEncryptionStream
```

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

```
+compress()
GZipOutputStream
```

Common Functionality on Streams

- Encryption
- Compression
- Buffering
- Reading of objects, numbers, lists, ...

```
+compress()
GZipOutputStream
«interface»
OutputStream
+encrypt()
AESEncryptionStream
+writeInt()
+writeString()
+writeFloat()
DataOutputStream
```
DataOutputStream (simplified)

class DataOutputStream implements OutputStream {
    private final OutputStream out;
    DataOutputStream(OutputStream o) { this.out=o; }
    public final void writeLong(long v) throws IOException {
        private byte writeBuffer[] = new byte[8];
        writeBuffer[0] = (byte)(v >>> 56);
        writeBuffer[1] = (byte)(v >>> 48);
        writeBuffer[2] = (byte)(v >>> 40);
        writeBuffer[3] = (byte)(v >>> 32);
        writeBuffer[4] = (byte)(v >>> 24);
        writeBuffer[5] = (byte)(v >>> 16);
        writeBuffer[6] = (byte)(v >>> 8);
        writeBuffer[7] = (byte)(v >>> 0);
        out.write(writeBuffer, 0, 8);
        incCount(8);
    }
    public void write(int b) throws IOException { out.write(b); }
    public void flush() throws IOException { out.flush(); }
}
The *Decorator* design pattern
The *Decorator* design pattern

- **Applicability**
  - To add responsibilities to individual objects dynamically and transparently
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical

- **Consequences**
  - More flexible than static inheritance
  - Avoids monolithic classes
  - Breaks object identity
  - Lots of little objects
Decorator Pattern in OutputStreams

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

- ByteArrayOutputStream
  - buffer
  + write()

- FilterOutputStream
  - delegating all calls to other output stream
  + write()
  + close()
  + flush()

- GZipOutputStream
  + compress()

- AESEncryptionStream
  + encrypt()

- DataOutputStream
  + writeInt()
  + writeString()
  + writeFloat()
The Scanner observer

• 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);
  ...

• Implements the *Iterator<String>* interface
Summary

- General abstractions for streams and readers
- Many optional features: compression, encryption, object serialization, ...
- Flexibility with few base implementations through
  - Adapter pattern
  - Delegation pattern