Principles of Software Construction: Objects, Design, and Concurrency

Design Case Study: Stream I/O

Charlie Garrod   Jonathan Aldrich
Administrivia

• No homework due this week
  ▪ Homework 4b due next Tuesday

• TAs have committed to grading Homework 4a by Friday "night"
  ▪ If you want faster feedback to revise your design as you work on Homework 4b, post a Piazza question asking for feedback
    • Please include your Andrew ID in the Piazza note
    • Please ask only if you sincerely need feedback to start Homework 4b
Key concepts from last Thursday
The Iterator design pattern

• Provides a strategy to uniformly access all elements of a container in a sequence
  § Independent of the container implementation
  § Ordering is unspecified, but every element visited once

• The `java.util.Iterator` interface:
  ```java
  public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // removes previous returned item
  } // from the underlying collection
  ```
Creating Iterators

```java
public interface Collection<E> {
    boolean    add(E e);
    boolean    addAll(Collection<E> c);
    boolean    remove(E e);
    boolean    removeAll(Collection<E> c);
    boolean    retainAll(Collection<E> c);
    boolean    contains(E e);
    boolean    containsAll(Collection<E> c);
    void       clear();
    int        size();
    boolean    isEmpty();
    Iterator<E> iterator();
    Object[]   toArray();
    E[]        toArray(E[] a);
    ...
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.
The Factory Method design pattern
Design patterns we have seen so far

- Iterator
- Composite
- Template Method
- Adapter
- Observer
- Strategy
- Marker Interface
- Decorator
- Model-View-Controller
- Factory Method
Learning goals for today

• Understand design aspects of the stream abstractions in Java

• Recognize the underlying design patterns:
  - Adapter
  - Decorator
  - Template Method
  - Marker Interface
  - Iterator
A Java aside

• What is a byte?
  • Answer: a signed, 8-bit integer (-128 to 127)

• What is a char?
  • Answer: a 16-bit Unicode-encoded character
The I/O 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, ...
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 reader/writer abstraction

• A sequence of **characters** in some encoding

• May read one character at a time 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);
```
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 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, ...**
Readers and streams

Client -> «interface» Reader

InputStreamReader -> «interface» InputStream

FileInputStream -> File

Client | InputStreamReader | FileInputStream
read (characters) | read (bytes)
Writers and streams

• See FileExample.java
Adding functionality to streams

- E.g. encryption, compression, buffering, reading formatted data such as objects, numbers, lists, ...
  - Two possible solutions:

```
class OutputStream {
  + encrypt()
}

class GZipOutputStream extends OutputStream {
  + compress()
}

class AESEncryptionStream extends OutputStream {
  + encrypt()
}

class DataOutputStream extends OutputStream {
  + writeInt()
  + writeString()
  + writeFloat()
}

class GZipOutputStream extends DataOutputStream {
  + compress()
}
```
A better design to add functionality to streams

```
«interface» OutputStream
+write()
+close()
+flush()
```

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

- `ByteArrayOutputStream`
  - -buffer
  - +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
The `java.util.Scanner`

- Provides convenient methods for reading from a stream

`java.util.Scanner`:

```java
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);
...```

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
Warning: A subtlety of serializability

- **Implement Serializable judiciously**
  - Making a class Serializable violates the principle of information hiding
  - *(Effective Java by Josh Bloch, 2nd edition, p. 274)*
Summary

- `java.io` provides general abstractions for streams and readers
  - Standard implementations, convenience implementations
- Many optional features: compression, encryption, object serialization, ...
- Convenience and flexibility via the Adapter pattern and Decorator pattern