Advanced Input/Output

Advanced I/O

- Character Encoding
- Decorator pattern
- Tokenizer
- Compressed I/O
- Random access file

Character Encoding

- Characters are not one-to-one mappings of integer values
  - Prevents round trip conversion of characters (or strings) to bytes
- Character to byte encoder and decoder classes
  - CharsetEncoder
  - CharsetDecoder

Character Sets

- `US-ASCII`
  - Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
- `ISO-8859-1`
  - ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
- `UTF-8`
  - Eight-bit UCS Transformation Format
- `UTF-16BE`
  - Sixteen-bit UCS Transformation Format, big-endian byte order
- `UTF-16LE`
  - Sixteen-bit UCS Transformation Format, little-endian byte order
- `UTF-16`
  - Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

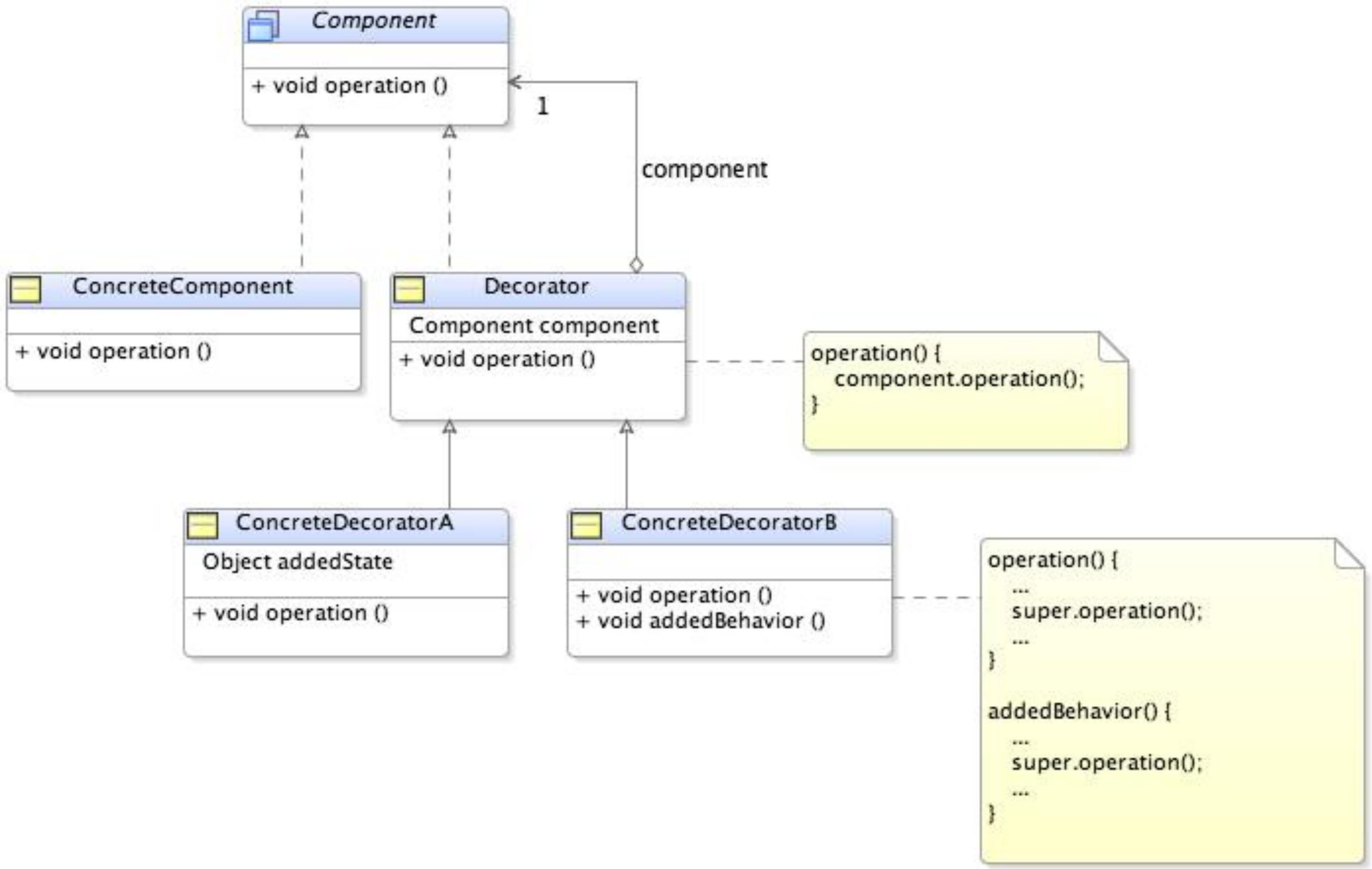
Decorator Problem

- Want to add several discrete behaviors to a single base class
  - But don’t want all of them at once
  - New class for each behavior
  - What if you want combinations of the behaviors
- Need to avoid explosion in the number of classes

Decorator Solution

- Provides an interface, realized by
  - Concrete class implementing base behavior
  - Abstract decorator class which is further extended to add additional behavior
    - Holds reference to instance of class it extends
    - Forwards calls to this instance
- Concrete decorators perform additional work before or after invoking base decorator methods

Decorator Structure



```
operation() {
   component.operation();
}
```

```
operation() {
   ...
   super.operation();
   ...
}

addedBehavior() {
   ...
   super.operation();
   ...
}
```

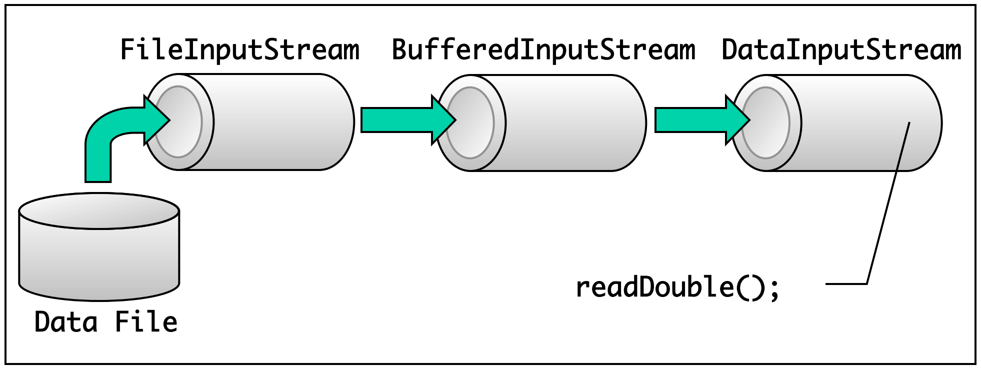
Decorator Solution

- Defining characteristics
  - Component being extended and the decorator share a common interface
  - Decorator maintains a reference to the component it extends and forwards method invocations to it
  - Concrete decorators add functionality by carrying out the additional behavior before or after delegating to extended component
- Discussion
  - Java IO filter example
  - Big pay off when using multiple decorators

Decorator Consequences

- Provides a more flexible way of adding functionality to a class than inheritance
- Eliminates feature laden classes high in the hierarchy
  - May cause there to be many small similar classes which are difficult to learn
- Decorator is not below its concrete component in the hierarchy so tests for the component type will fail

Layered Streams



```
FileInputStream fis;
BufferedInputStream bis;

fis = new FileInputStream( "data.dat" );
bis = new BufferedInputStream( fis );
DataInputStream dis = new DataInputStream( bis );
double x = dis.readDouble();
```

Filter Classes

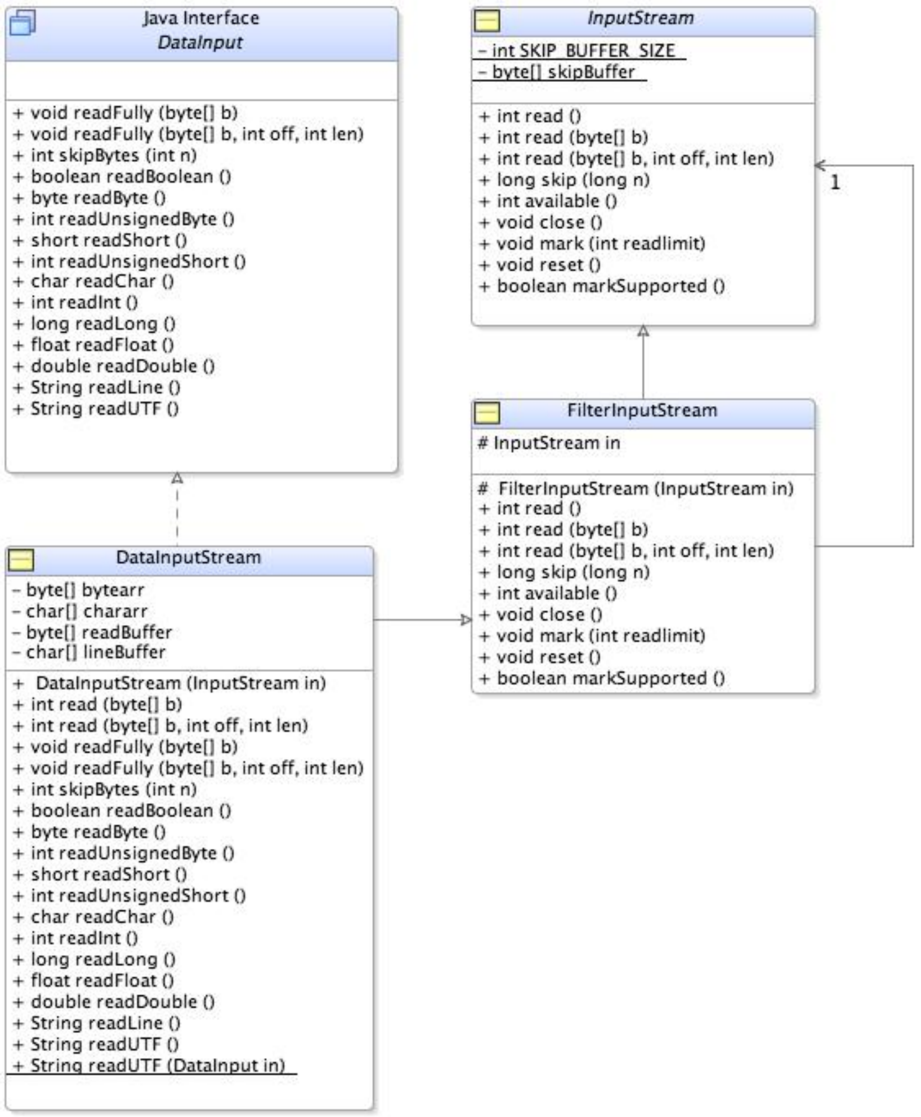
- **Four abstract classes are provided to facilitate the creation of custom filters (layers).**
  - **Hold a reference to the underlying stream.**
  - **Methods pass all requests to the underlying stream.**

```
FilterInputStream & FilterOutputStream
FilterReader & FilterWriter
```

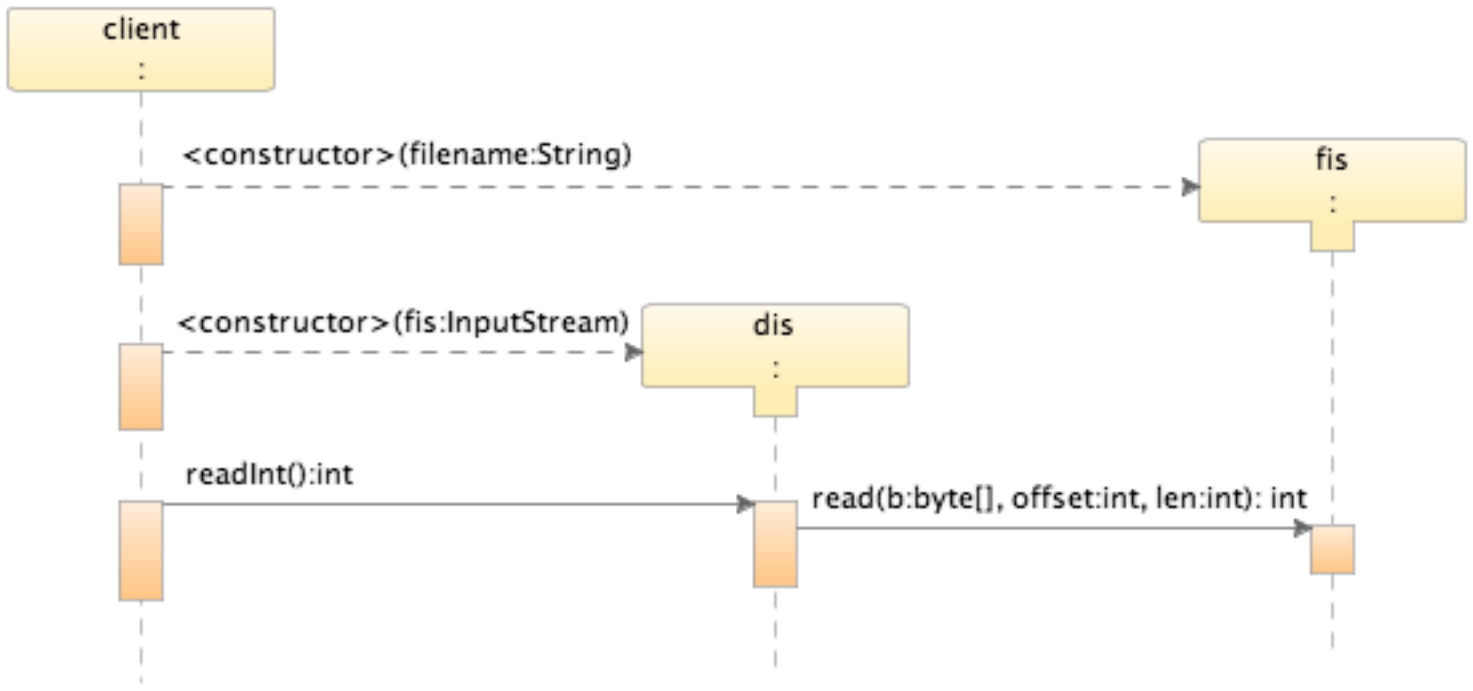
DataInput & DataOutput

- Interfaces for reading and writing primitive types
- Direct implementations
  - DataInputStream
  - DataOutputStream

`DataInputStream` Decorator



DataInputStream Sequence


client
:
<constructor>(filename:String)
fis
:
<constructor>(fis:InputStream)
dis
:
readInt():int
read(b:byte[], offset:int, len:int): int

StreamTokenizer

- Created using Reader object as source
- Recognizes
  - C/C++ style comments
  - String literals
  - Numbers
  - Whitespace
- Scanner loop invokes `nextToken()`
  - Returns token type

Token Type

- Variable `ttype` contains token type
  - `TT_WORD`
  - `TT_NUMBER`
  - `TT_EOL`
  - `TT_EOF`
  - Value of a character token
  - Quote character for quoted strings

Parsed Values

- The variable nval holds the value of numeric tokens
- The variable sval holds a string containing word tokens

Configuration

- **Methods used to configure parser behavior**
  - `void eolIsSignificant( boolean )`
  - `void lowerCaseMode( boolean )`
  - `void slashSlashComments( boolean )`
  - `void slashStarComments( boolean )`
  - `void parseNumbers()`

Configuration

- **Methods used to define token contents**
  - `void commentChar( int )`
  - `void ordinaryChar( int )`
  - `void ordinaryChars( int, int )`
  - `void quoteChar( int )`
  - `void resetSyntax()`
  - `void whitespaceChars( int, int )`
  - `void wordChars( int, int )`

RPN Calculator Script

- Script for RPN calculator

```
# Input file for StreamCalc
2
5
*
7
13
+
*
2
/
3.14
-
```

RPN Calculator

```
// create tokenizer
FileReader dataFile = new FileReader( args[0] );
StreamTokenizer stok = new StreamTokenizer(dataFile);
// configure tokenizer
stok.parseNumbers();
stok.commentChar( '#' );
stok.ordinaryChar( '*' );
stok.ordinaryChar( '/' );
stok.ordinaryChar( '+' );
stok.ordinaryChar( '-' );
// perform calculations
Stack<Double> stack = new Stack<Double>();
int token;
double r;
```

RPN Calculator

```
while((token = stok.nextToken()) != StreamTokenizer.TT_EOF) {
    switch( token ) {
        case StreamTokenizer.TT_NUMBER:
                 stack.push(stok.nval);
                 break;
         case '*':
                 r = stack.pop() * stack.pop();
                 stack.push(r);
                 break;
         ...
    }
}
System.out.println( "Result = " + stack.pop() );
```

Compressed I/O Classes

- Their own package – `java.util.zip`
- Derived from Input/Output streams
- A number of compression algorithms
  - ZIP
  - GZIP

Compressed Streams

- **Simple stream filters**
  - `InflaterInputStream`
    - `GZIPInputStream`
    - `ZipInputStream`
  - `DeflaterOutputStream`
    - `GZIPOutputStream`
    - `ZipOutputStream`

ZipEntry

- Entries, like files or directories
  - Name – immutable
  - Time
  - Size
  - CRC
  - Method
  - Extra bytes
  - Comment
  - Compressed size

ZipInputStream

- **Methods for manipulating entries**
  - `void closeEntry()`
  - `ZipEntry getNextEntry()`

ZipOutputStream

- **Methods for manipulating entries**
  - void setComment(String)
  - void setMethod(int)
  - void setLevel(int)
  - void putNextEntry(ZipEntry)
  - void closeEntry()
  - void finish()

`ZipFile`

- **Allows reading of zip files as a series of entries**
  - `Enumeration entries()`
  - `ZipEntry getEntry( String )`
  - `InputStream getInputStream( ZipEntry )`

Simple Zipper

```
public void zip( String zipFileName, String[] files ) {
   FileOutputStream fos = new FileOutputStream( zipFileName );
   BufferedOutputStream bos = new BufferedOutputStream( fos );
   ZipOutputStream zos = ( new ZipOutputStream( bos ) );
   for( int i = 0; i < files.length; i++ ) {
      addEntry( zos, files[i] );
   }
   zos.close();
}
```

Simple Zipper

```
private void addEntry( ZipOutputStream zipOut, String file ) {
   File f = new File( file );
   if( f.isDirectory() ) {
      // process files in directory
      String[] files = f.list();
      for( int i = 0; i < files.length; i++ )
         addEntry( zipOut, file + "/" + files[i] );
   } else {
      byte buf[] = new byte[BUF_SIZE];
      int bytes = 0;
      FileInputStream in = new FileInputStream( f );
      ZipEntry e = new ZipEntry( file );
      zipOut.putNextEntry( e );
      while( (bytes = in.read( buf ) ) != -1 )
         zipOut.write( buf, 0, bytes );
      zipOut.closeEntry();
   }
}
```

Simple Zipper

```
public void unzip( String zipFile, String targetDir ) {
   boolean hasTargetDir = (targetDir != null &&
                           targetDir.length()>0);
   if( hasTargetDir ) {
      File dir = new File( targetDir );
      if( !dir.exists() ) {
         dir.mkdirs();
      }
   }
   FileInputStream fs = new FileInputStream( zipFile );
   ZipInputStream zin = new ZipInputStream( fs );
   ZipEntry entry;
   byte buf[] = new byte[BUF_SIZE];
   while( (entry = zin.getNextEntry()) != null )
   {
      ...
```

Simple Zipper

```
      String name;
      if( hasTargetDir )
         name = targetDir+"/"+entry.getName();
      else
         name = entry.getName();
      File f = new File( name );
      if( !entry.isDirectory() && !f.exists() ) {
         File dir = f.getParentFile();
         if( dir != null && !dir.exists() )
            dir.mkdirs();
         FileOutputStream fos = new FileOutputStream( name );
         while( (int len=zin.read( buf, 0, BUF_SIZE )) != -1 )
               fos.write( buf, 0, len );
      }
      zin.closeEntry();
   }
}
```

Random File Access

- "Random Access" - ability to set the read/ write file pointer to any position in the file and perform an operation
- Records must be well understood
  - Known Size
  - Known Structure
- Supports "`r`", "`rw`", no support for write-only

Derivation

- No association with Input/Output Stream hierarchy
  - Not a Stream type
- Can not use other Stream classes
  - Only works with Files
  - Implements DataInput and DataOutput

Seeking

- C standard I/O Library, seeking to a position relative to:
  - Beginning of the file
  - End of file
  - Current position
- Java seeks only relative to beginning of file

RandomAccessFile

- **Methods**

  `long length()`

  `long getFilePointer()`

  `seek( long )`

  `void close()`

  read… implements `DataInput`

  write… implements `DataOutput`

Random Read Example

- Randomly access a file
  - File uses ‘$$’ as a record separator
  - Create a record index
  - Randomly seek and read using the index

Random Read Example

```java
private void index() {
   int textChar;
   mIndex.addElement( new Long(0) );
   while( ( textChar = mRandomFile.read() ) != -1 ) {
      if( (char) textChar == '$' ) {
         textChar = mRandomFile.read();
         if( (char)textChar == '$' )
            mIndex.addElement(
              new Long(mRandomFile.getFilePointer()));
         else if( textChar == -1 )
            break;
      }
   }
}
```

Random Read Example

```
public void printOne() {
   int ndx = (int)(Math.random() * mIndex.size());
   long offset = ((Long)mIndex.elementAt(ndx)).longValue();
   System.out.println( "Offset:" + offset );
   mRandomFile.seek( offset );
   System.out.print( "String = '" );
   int textChar;
   while( (textChar = mRandomFile.read()) != -1 ) {
      if( textChar == '$' ) {
         if( (textChar = mRandomFile.read()) == '$' )
            break;
         System.out.print( '$' );
      }
      System.out.print( (char)textChar );
   }
   System.out.println( "'" );
}
```

Properties

- `java.util.Properties`
  - **Subclass of** `Hashtable`
  - **A map of** `String` **keys and values (properties)**
- **Supports persistence**
  - `void store(OutputStream out, String comment)`
  - `void load(InputStream in)`

Line Oriented Text I/O

- **Reading and writing individual lines in a text file**
  - `BufferedReader`
    - `String readLine()`
  - `PrintStream/PrintWriter`
    - `void println(String s)`

Strings in Binary Files

- Strings are variable length, unlike the primitives
- Writing
  - `DataOutputSream.writeUTF`
    - Writes the length, followed by the characters
- Reading
  - `DataInputStream.readUTF`
    - Reads the string