

IO and Javadoc

Lecture 8

Part I: IO

I/O Package Overview

- Package java.io
- Core concept: streams
 - Ordered sequences of data that have a source (for input) or a destination (for output)
- Two major flavors:
 - Byte streams
 - 8 bits, data-based
 - Input streams and output streams
 - Character streams
 - 16 bits, text-based
 - Readers and writers
- See Java API documentation for details

Byte Streams

- Two abstract base classes: InputStream and OutputStream
- InputStream (for reading bytes) defines:
 - An abstract method for reading 1 byte at a time `public abstract int read()`
 - Returns next byte value (0-255) or -1 if end-of-stream encountered
 - Concrete input stream overrides this method to provide useful functionality
 - Methods to read an array of bytes or skip a number of bytes
- OutputStream (for writing bytes) defines:
 - An abstract method for writing 1 byte at a time `public abstract void write(int b)`
 - Methods to write bytes from a specified byte array
- Close the stream after reading/writing `public void close()`
 - Frees up limited operating system resources
- All of these methods can throw IOException

Example 1: Counting Characters

```
import java.io.*;
class CountBytes {
    public static void main(String[] args)
        throws IOException {
        InputStream in = new FileInputStream(args[0]);
        int total = 0;
        while (in.read() != -1) {
            total++;
        }
        in.close();
        System.out.println(total + " bytes");
    }
}
```

Standard Streams

- Three standard streams for console IO
 - System.in
 - Input from keyboard
 - System.out
 - Output to console
 - System.err
 - Output to error (console by default)
- These streams are byte streams!
 - System.in is an InputStream, the others are PrintStreams (inherit from OutputStream)
 - Would be more logical for these to be character streams not byte streams, but they predate the inclusion of character streams in Java

Example 2: Console Streams

```
import java.io.*;
class TranslateBytes {
    public static void main(String[] args)
        throws IOException {
        byte from = (byte)args[0].charAt(0);
        byte to = (byte)args[1].charAt(0);
        int x;
        while((x = System.in.read()) != -1)
            System.out.write(x == from ? to : x);
    }
}
```

- If you run "java TranslateBytes b B" and enter text bigboy via the keyboard the output will be: BigBoy

Character Streams

- Two abstract base classes: Reader and Writer
- Similar methods to byte stream counterparts
- Reader abstract class defines:
 - public int read()
 - Returns value in range 0..65535 (or -1)
 - public int read(char[] cbuf)
 - Returns number of characters read
 - public void skip(int n)
- Writer abstract class defines:
 - public void write(int c)
 - public void write(char[] cbuf)
 - public abstract void flush()
 - Ensures previous writes have been sent to destination
 - Useful for buffered streams
- Both classes define:
 - public void close()

Converting Byte/Character Streams

- Translation supported by conversion streams: InputStreamReader and OutputStreamWriter
 - Subclasses of Reader and Writer respectively
- InputStreamReader

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String encoding)
public int read()
```

 - An encoding is a standard map of characters to bits (eg UTF-16)
 - Reads bytes from associated InputStream and converts them to characters using the appropriate encoding for that stream
- OutputStreamWriter

```
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out, String enc)
public void write(int c)
```

 - Converts argument to bytes using the appropriate encoding and writes these bytes to its associated OutputStream
- Closing the conversion stream also closes the associated byte stream – may not always be desirable

Working with Files

- A file can be identified in one of three ways
 - A String object (file name)
 - A File object
 - A FileDescriptor object
- Sequential-Access file: read/write at end of stream only
 - FileInputStream, FileOutputStream, FileReader, FileWriter
 - Each file stream type has three constructors
- Random-Access file: read/write at a specified location
 - RandomAccessFile
 - A *file pointer* is used to guide the starting position
 - Not a subclass of any of the four basic IO classes (InputStream, OutputStream, Reader, or Writer)
 - Supports both input and output
 - Supports both bytes and characters

Example: A Random Access File

```
class Filecopy {
    public static void main(String args[]) {
        RandomAccessFile fh1 = null;
        RandomAccessFile fh2 = null;

        try {
            fh1 = new RandomAccessFile(args[0], "r");
            fh2 = new RandomAccessFile(args[1], "rw");
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
            System.exit(100);
        }

        try {
            int bufsize = (int) (fh1.length()/2);
            byte[] buffer1 = new byte[bufsize];
            fh1.readFully(buffer1, 0, bufsize);
            fh2.write(buffer1, 0, bufsize);
        } catch (IOException e) {
            System.out.println("IO error occurred!");
            System.exit(200);
        }
    }
}
```

The File Class

- Useful for retrieving information about a file or a directory
 - Represents a *path*, not necessarily an underlying file
 - Does not open/close files or provide file-processing capabilities
- Three constructors

```
public File(String name)
public File(String pathToName, String name)
public File(File directory, String name)
```
- Main methods

```
boolean canRead() / boolean canWrite()
boolean exists()
boolean isFile() / boolean isDirectory()
String getAbsolutePath() / String getPath()
String getParent()
String getName()
long length()
long lastModified()
```

Efficient IO

- Buffering greatly improves IO performance
- Example: BufferedReader for character input streams

```
public BufferedReader(Reader in)
```

 - The buffered stream "wraps" the unbuffered stream
- Example declarations of BufferedReaders
 - An InputStreamReader inside a BufferedReader

```
Reader r = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(r);
```
 - A FileReader inside a BufferedReader

```
Reader fr = new FileReader("fileName");
BufferedReader in = new BufferedReader(fr);
```
 - Then you can invoke `in.readLine()` to read from the stream line by line

Example

```
public class EfficientReader {
    public static void main (String[] args) {
        try {
            Reader fr = new FileReader(args[0]);
            BufferedReader br = new BufferedReader(fr)
            String line = br.readLine();

            while (line != null) {
                System.out.println("Read a line:");
                System.out.println(line);
                line = br.readLine();
            }
            br.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + args[0]);
        } catch (IOException e) {
            System.out.println("Can't read from file: " + args[0]);
        }
    }
}
```

Part II: Javadoc

Motivation

- Over the lifetime of a project, it is easy for documentation and implementation to diverge
 - Usually, documentation and code are not *both* living entities
- Basic principle of software design:
 - Single-point of control over change
- When that's not possible:
 - Make (logical) coupling obvious
 - When things get out of whack, code starts to smell
 - Items need to be co-located and visually linked

Basics

- Convention for formatting source code comments
 - Not *compiler* enforced, but other tools exist
- Place comments between `/**` and `*/`
 - Comment must appear *immediately* before class, interface, method, field
 - Overview and package level comments available too
- Includes standard set of tags
 - `@author`, `@param`, `@return`, `@see`, `@throws...`
 - Each tag begins line, followed by text description
- Process code with javadoc tool
 - Produces linked, html output
 - See JDK API documentation

Javadoc Comments

- Comment = main description + block tags
 - First sentence of main description is "summary"
 - Terminated by "." followed by white space/new line
 - Appears at the top of page
 - Write comments in html (`<p>`, `<pre>...`)
 - Use html character entities (`<` & `>` & `&`)
 - Avoid `<h1>` < `<h2>`
- Block tags
 - `@author`, `@param`, `@return`, `@see`, `@throws`, `@deprecated`, ...
- Inline tags
 - Used within text of a documentation comment
 - `{@link}`, `{@value}`, `{@code}`, `{@literal}`, `{@inheritDoc}`, ...

Example

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <param>
 * This method always returns immediately, whether or not the
 * image exists.
 *
 * @author Sun
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) { ... }
```

Javadoc Tags

- **@param**: documents a single parameter of a method
 - Use one for each parameter of the method
 - Syntax: `@param parameter-name description`
 - Example:
`@param max The maximum number of words to be read`
- **@return**: documents the return value of a method
 - Example:
`@return The number of words actually read`
- **@throws**: documents an exception thrown by the method
 - Use one for each type of exception the method throws
 - Example:
`@throws NullPointerException The name is {@code null}`

Javadoc Tags (cont'd)

- **@see**: creates a cross-reference link to other javadoc documentation
 - Forms a "See also" section at the end of the documentation
 - Qualify the identifier *sufficiently*
 - Specify class/interface members by using a # before the member
 - If a method is overloaded, list its parameters
 - Specify classes/interfaces with their simple names
 - Give full name if class/interface is from another package
 - Examples:
`@see #getName`
`@see Attr`
`@see com.hostname.attr.Attr`
`@see com.hostname.attr.Attr#getName`
`@see com.hostname.attr.Attr#Attr(String, Object)`
`@see com.hostname.attr.Attr#Attr(String)`
`@see Attribute Specification`
 - You can also use a label after an entity reference. The label will be the actual text displayed.
`@see #getName Attribute Names`

Javadoc Tags (cont'd)

- **{@link}**: similar to **@see**, but it embeds a cross reference in the text of your comments
 - Syntax: `{@link package.class#member [label]}`
 - Identifier specification follows the same requirement for **@see**
 - Example:
`Changes the value returned by calls to {@link #getValue}`
- **@deprecated**: marks that an identifier should no longer be used. It should suggest a replacement.
 - Example:
`@deprecated Use {@code setVisible(true)} instead`
- **@author**
 - Only one author name per **@author** paragraph
- **@version**
- **@since**: denote when the tagged entity was added to your system
- Example: Graphics.java Output Documentation
 - `% javadoc Graphics.java`

New in 5.0

- Additions in 1.4 were more significant:
 - User-defined custom tags with `-tag option`
 - `{@inheritDoc}` for fine-grain control over inheritance
 - `-linksource` for producing html version of source code
 - Omitting leading asterisks makes leading white space meaningful
 - Useful for visually formatting cut-and-paste code
- **{@literal}** and **{@code}** inline tags
 - `{@literal xx
xx}` gives `xx
xx` in documentation
 - `{@code yyyy} = <code>{@literal yyyy}</code>`
- Javadoc tags vs Annotations
 - Tags provide information to human
 - Annotations provide information to compiler/library
 - Sometimes need both (eg `@deprecated`, `@Deprecated`)

Package Documentation

- Unlike classes/methods, packages not defined in 1 source file
- To generate package comments, add a package.html file in the package directory
 - The contents of the package.html between `<body>` and `</body>` will be read as if it were a doc comment.
 - `@deprecated`, `@author`, and `@version` are not used in a package comment
 - The first sentence of the body is the summary of the package.
 - Any `@see` and `{@link}` tag must use the fully qualified form of the entity's name, even for classes and interfaces within the package itself
- You can also provide an overview comment for all source files by placing a `overview.html` file in the parent directory
 - The contents between `<body>` and `</body>` is extracted
 - The comment is displayed when the user selects "Overview"

Best Practices: A Uniform Style

- Consistency among team members
 - Omit ()'s from method names
 - exception: overloaded methods: list parameter types
 - Phrase for param's beginning with article + type
 - @param ch the character to be inserted in the selected buffer
 - 3rd person descriptive
 - * Appends the image observer to the queue of active observers.
 - Required vs optional tags
 - Ordering of block tags
 - param, return, throws, exception, author, see, deprecated
- Sun's style guide
 - "How to Write Doc Comments for Javadoc"
 - <http://java.sun.com/j2se/javadoc/writingdoccomments/>
 - Virtually an industry-wide standard

Best Practices: Doc the Contract

- Javadoc comments describe a component's *contract* not its implementation
 - Describe *what* a method does, not *how* it does it
 - What a *client* component needs to know
 - Contract is usually more stable than implementation
- Describe method assumptions
 - Preconditions on arguments
 - eg, observer must be non-Null, list must contain target
 - Preconditions on object state
 - In terms of "public" (ie externally checkable, abstract) state
- Describe method guarantees
 - Postconditions on return value
 - eg, @return true if and only if target is within image boundary
 - Postconditions on object state
- Describe class invariants

Best Practices: Doc Exceptions

- Document every checked exception
 - @throws clause for each, describing reason
- Throw (and document) exceptions at the right level of abstraction
 - Avoid revealing implementation specifics
 - eg `IndexOutOfBoundsException` vs `ArrayIndexOutOfBoundsException`
- Document "some" runtime exceptions
 - The ones the client should reasonably care about (?)
 - Never include these in method signature
 - Danger: no real enforcement mechanism
 - Consistency within project? Client attention?
 - Parent's @throws for *unchecked* exceptions *not* inherited
 - Use { @inheritDoc } to explicitly bring this in
 - Documentation for *checked* exceptions *is* inherited (if child declares)

Worst Practices: Bad Hungarian

- Adding *programming language type* as prefix to variable name
 - eg `fDone` (f for boolean)
 - Obfuscation, inconsistencies, redundancy, concrete coupling
- Adding *semantic information* to variable name, however, can be useful
 - eg `radSunDeclination`
 - Can help to expose unit errors
 - `if (radSunDeclination == degMoonDeclination)...`
 - `inTableCircumference = 2*PI*cmTableRadius`

Worst Practices: Miscellaneous

- End-of-function comments

```
public void setRate (int frequency) {  
    ...  
} //setRate
```

 - Obviated by modern editors with code folding
- Commenting bug fixes
 - Version control is a better place for this than Javadoc
- Comments with no additional value
 - Repeating the parameter name as the description
- Leaving boiler-plate comments in code
 - Automatically generated Javadoc with obvious boiler plate code should *never* appear in repository
 - Don't leave it hanging around your own code