# Programmieren II

## Java-Docs & Deployment

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from Oracle and T. Bögel)

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

July 3, 2014

# Outline

# Outline

# Principles

## Two ways of writing docs

- API specifications (Java Platform API Specification)
- Programming guide documentation

# Writing API Specifications

1. API Specification (most commonly used)
   - Ideally: all assertions required to do **clean-room implementation**
   - API specification: defined by *documentation comments* in source code
   - Extended documentation in *separate* files
   - Describes *contracts*, no implementation details
   - Exceptions must be set apart
   - Clear error behavior
   - API should be enough to write **Unit Tests**

# Writing Programming Guide Documentation

## 2. Programming Guide Documentation

- Programming guide: examples, definition of common terms, metaphors, description of implementation
- Contribute to developer's understanding
- Should be separated from doc comments in the source code
- Example: Java Tutorials

# Terminology

## API docs or API specs

- Descriptions of the API
- Target audience: programmers
- Automatically extractable from the source code

## Doc comments

- Special comments to indicate Java Docs: `/** . . . */`

## Javadoc

- JDK tool that generates API documentation

## Source files

- Source code files for java classes
- Package comment files
- Overview comment files

# Format of doc comments

- Format for doc comments: HTML
- Comments precede corresponding constructor, method or declaration
- Two parts: **description** followed by **tags**

# Doc comment – Example I

```
/**
 * Returns an Image object that can be painted on the screen.
 * The url argument must specify an absolute {@link URL}.
 * The name is a specifier that is relative to the url.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL:  the location of the image
 * @param  name the location of the image, relative to the url
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
```

# Doc comment – Example II

```
            return null;
        }
}
```

# Comments I

- Each line is indented to align with code below the comment
- The first line contains the *begin-comment delimiter* ( `/**`)
- Leading asterisks are optional
- *First sentence*: short summary
- Inline tag `{@link URL}`: hyperlink pointing to documentation of `URL` class
- Paragraphs are separated by <p>
- Blank comment line between description and tags
- First line beginning with an "@" ends description
- Last line: end-comment delimiter (`*/`)
- Limit any doc-comment line to 80 characters

# Descriptions I

## First sentence

- First sentence: summary sentence
- First sentence is used for package/class or member summaries
- “Crisp and informative sentences that can stand on their own”
- Sentence ends at first period that is followed by a white space

## Problematic white spaces

```
/**
 * This is a simulation of Prof. Knuth's MIX computer.
 */
```

→ first sentence ends at “Prof.”

# Descriptions II

Solution: HTML code for white spaces

```
/**
 * This is a simulation of Prof. Knuth's MIX computer.
 */
```

# Descriptions III

## Distinguishing overloaded methods

- First sentence should distinguish overloaded methods
- Example:

```
/**
 * Class constructor.
 */
foo() {
  ...

/**
 * Class constructor specifying number of objects to create.
 */
foo(int n) {
  ...
```

# Descriptions – Hints

## Hints

- Description should be complete enough for conforming implementors
- Specs should be complete, include boundary conditions and value ranges
- Description should be implementation-independent

# Automatic re-use of method comments

## Automatically inherited/duplicated comments

- When a method in a **class** *overrides* a method in a **superclass**
- When a method in an **interface** *overrides* a method in a **superinterface**
- When a method in a **class** *implements* a method in an **interface**
- If java doc is defined in sub-class: docs are not copied

# Style Guide I

## <code> style

- Use <code>. . . </code> for key words and names
- Java key words
- Package names, class names
- Method names, interface names, field names
- Argument names
- Code examples

# Style Guide II

## In-line links

- In-line links: `{@link target}` tag
- Not necessary to add links for all API names in a doc comment
- Use links if user might actually *want to click* on it for more information
- Only for the **first occurrence** of each API name

# Style Guide III

## Some style hints

- Omit parentheses for the general form of methods and constructors
  Example: "The add method enables you to insert items."
- OK to use phrases instead of complete sentences, in the interests of brevity
- Use 3rd person instead of 2nd person
- Method descriptions should begin with a verb phrase
- Add description beyond the API name

# Required tags

## `@param`

- `@param` required for every parameter
- Followed by the name of the parameter
- First noun in the description: data type of the parameter
- Data type starts with a lowercase letter
- Example: `@param ch the character to be tested`

## `@return`

- Required for every method that returns something other than `void`
- Whenever possible: state return values for **special** cases

# Optional tags I

`@author`

- None, one or multiple `@authors`
- Not included in the API specification
- Only visible in the source code

`@deprecated`

- Tell the user when the API was deprecated
- Name possible replacements

# Optional tags II

`@throws`

- Should be included for any checked exception
- Errors should not be documented
- Example:

```
/**
 * @throws IOException  If an input or output
 *                      exception occurred
 */
public void f() throws IOException {
    // body
}
```

# Package-level comments

## Package-level comments

- Each package can have its own package-level doc comment source file
- File name: `package-info.java`
- Location: in the source directory along with all `*.java` files

# Generating Javadocs

## Javadoc

- Documentation generator for generating API docs in HTML format from Java source code
- In Eclipse: `Project` → `Generate Javadoc`
- → `Use standard doclet`
- Adjust `Destination` and additional settings

## Doclet

- Modifies content and format of documentation
- Usually: sufficient to use the built-in doclet
- Documentation for doclets: `http://download.java.net/jdk8/docs/technotes/guides/javadoc/index.html`

# Outline

# Java Archive (JAR) file format

- Bundles multiple files into a single archive file
- Typically: class files & auxiliary resources

## Benefits

- **Security**: JARs can be digitally signed
- **Compression**: content in JAR files is compressed
- Packaging for **extensions**: JAR files can be added to other programs easily

# Using JAR Files: Basics

- JAR files are packaged with ZIP file format
- This allows for compression, archiving, decompression and unpacking
- JAR files can be created with the *Java Archive Tool* (in the JDK)

## Common operations

| **Operation** | **Command** |
|---|---|
| Creating a jar file | `jar cfe jar-file MainClass input-file(s)` |
| Viewing the contents of a JAR file | `jar tf jar-file` |
| Extracting the contents of a JAR file | `jar xf jar-file` |
| Extracting specific files from a JAR file | `jar xf jar-file archived-file(s)` |
| Running application (JAR file)[a] | `java -jar app.jar` |

[a] Uses MainClass

# Creating a jar file

## Basic command format

`jar cfe jar-file MainClass input-file(s)`

- `c`: create a file
- `f`: output should be a *file*
- `e`: entrypoint, the class whose main method should be run (optional)
- `jar-file`: name of the resulting jar file
- `input-files`: space-separated list of one or more files that should be included in JAR file.
  Directories are added recursively.
- Adds a default manifest file to path `META-INF/MANIFEST.MF`

## Parameters

- `0`: do not compress the content
- `v`: verbose
- `m`: include manifest information from an existing manifest file

# Viewing the contents of a jar file

## Basic command format

`jar tf jar-file`

- `t`: view the *table* of contents of the jar file
- `f`: input is a *file*
- `jar-file`: name of the jar file to be read
- `v` (optional): additional information about file size and modification dates

# Extracting the contents of a jar file

## Basic command format

`jar xf jar-file [archived-file(s)]`

- `x`: *extract* files from the jar archive
- `f`: input is a *file*
- `jar-file`: name of the jar file to be extracted
- `archived-file(s)` (optional): space-separated list of the files to be extracted from the archive

# Updating a jar file

## Basic command format

`jar uf jar-file input-file(s)`

- u: *update* existing jar file
- `f`: input that should be updated is a *file*
- `jar-file`: existing jar-file that should be updated
- `input-file(s)`: space-delimited list of one or more files that you want to add to the jar file

# Running jar files as applications

## Basic command

`java -jar jar-file`

- Runtime environment needs to know **which class to execute**
- This is done by adding a `Main-Class: classname` header to the Manifest file with the e parameter when creating the jar file, or by explicitly creating a manifest

# Default manifest

## Manifest file

- Manifest file contains information about files packaged in a jar file
- Meta information about a jar file
- Only one manifest per jar file
- Path of the manifest file: `META-INF/MANIFEST.MF`
- Format of entries: (`header: value`) pairs

## Default manifest (without e option)

```
Manifest-Version: 1.0
Created-By: 1.7.0_09 (Oracle Corporation)
```

# Modifying a manifest file

## Basic command to modify default manifest

`jar cfm jar-file manifest-addition input-file(s)`

- `manifest-addition`: path of existing text file whose contents you want to add to the jar file's manifest
- `manifest-addition` is a plain text file that contains the desired additions

# Setting an application's entry point

## Specifying the start class

- Add this to the manifest file: `Main-Class: classname`
- Class needs to have a *main* method
- Create a jar file with the modified manifest file
- $\rightarrow$ start class is executed with the command `java -jar jar-name`

# Adding Classes to the jar File's Classpath

- Reference classes in other JAR files from within a JAR file
- Add this to the manifest file: `Class-Path: jar1-name jar2-name directory-name/jar3-name`

# Creating jar files with eclipse

## Exporting your java project as a jar file

- `Export` → `Runnable JAR file`
- `Launch configuration`: entry point (starting class with a main method)

# Signing jar files

- Jar files can be signed to verify that the content has not changed

## General workflow

1. Programmer signs a jar file
2. Jar file can be verified by a user
3. Documentation: `http://docs.oracle.com/javase/tutorial/deployment/jar/signing.html`

# What is Apache Ant?

- Java-based build tool
- Acronym for “Another Neat Tool”
- Similar to *make*
- Apache project. Download and information: `http://ant.apache.org/`

# General Ant workflow

- `build.xml` in a directory defines *targets* that can be executed
- Tasks for an Ant script
  - Compiling the source files
  - Creating a jar file for deployment
  - Cleaning up temporary files

# Example: Hello World with Apache Ant

## Preparing your project

- Create your source directory: `mkdir src`
- Create a HelloWorld class in `src/test/HelloWorld.java`

# Basic build file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="jar">

    <target name="clean" description="Delete all generated
        files">
        <delete dir="classes"/>
        <delete file="MyTasks.jar"/>
    </target>

    <target name="compile" description="Compiles the Task">
        <javac srcdir="src" destdir="classes"/>
    </target>

    <target name="jar" description="JARs the Task">
        <jar destfile="MyTask.jar" basedir="classes"/>
    </target>

</project>
```

# Single components of a build.xml file I

## XML header

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

- Standard XML header

## Project element

```
<project name="MyTask" basedir="." default="jar">
```

- Specifies the name of the project
- Base/root directory
- Specifies the *default* target

# Single components of a build.xml file II

## Targets

```
<target name="clean" description="Delete all generated files">
    <delete dir="classes"/>
         <delete file="MyTasks.jar"/>
     </target>
```

- One target represents one task
- Target has a name and description
- Within target element: tasks and operations provided by Ant
- Overview of available ant tasks: https://ant.apache.org/manual/tasksoverview.html

# Ant tasks

## javac

- Compile source files in directory `srcdir` to `destdir`
- `classpath` option: classpath to be used

## jar

- Creates the jar file specified with `destfile`
- `basedir`: directory with files that should be included in the jar file
- `manifest`: the manifest file to use

# Using properties and defining dependencies

- Same value is used repeatedly: we should use variables
- Variables in Ant: properties
- Properties can be used within the build file with `${name}`

## Defining properties

```
<property name="src.dir" value="src"/>
<property name="classes.dir" value="classes"/>
```

## Dependencies

- `target` elements can contain an optional depends attribute to show that another target needs to run before the target
- E.g. `<target name="jar" depends="compile">`

# Updated build file I

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="jar">
    <property name="src.dir" value="src"/>
    <property name="classes.dir" value="classes"/>

    <target name="clean" description="Delete all generated
        files">
        <delete dir="${classes.dir}" failonerror="false"/>
        <delete file="${ant.project.name}.jar"/>
    </target>

    <target name="compile" description="Compiles the Task">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
    </target>

    <target name="jar" description="JARs the Task" depends="
        compile">
```

# Updated build file II

```
        <jar destfile="${ant.project.name}.jar" basedir="${
            classes.dir}"/>
    </target>
</project>
```

# Running ant targets

## Running the build process

`ant [target]`

- `ant` without any target runs the `default` target
- If a target is specified, this target (and optionally dependencies) is executed

# References


The Java Tutorial

*Lesson: Packaging Programs in JAR Files*

http://docs.oracle.com/javase/tutorial/deployment/jar/

Oracle Technology Network

*How to Write Doc Comments for the Javadoc Tool*

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Ullenboom, Ch.

*Java ist auch eine Insel*. (Ch. 19.3 & 19.4)

Galileo Computing, 10th edition, 2012.