Java Code Conventions, Packaging & Delivery

EXTRA MATERIAL

** Indentation / Naming / Comments / Class Definition

** JavaDocs / Packaging / Making JARs

Chapter 4 (sections 4.7-4.10) – "Core Java" book
Chapter 17 – "Head First Java" book
Sections 1.10+1.11 – "Introduction to Java Programming" book
Chapter 7 – "Java in a Nutshell" book

These slides are left as practice and self-study.

Java Code Conventions: why?

- To produce programs that have *good style*.
- Improve *readability*: by author, by others.
- Design for *reusability*: can be reused later in other programs.
- Good *appearance*.
- *Maintenance*: reduce cost.
- Clean and well packaged as a product.
- *How to achieve this?*
  - Indentation
  - Naming
  - Comments
  - Source file organisation

Indentation

- The *blank space(s)* between a margin and the beginning of an indented line.
- Emphasises program structure.
- Unit: 4 spaces (or just *be consistent*).
- Indent every level:
  - when a new set of curly braces *or*
  - (a block) occurs.

Tip: Use IDE to format your source code and this becomes a very simple task.

Eclipse: Source → Format
NetBeans: Source → Reformat Code

Naming: class, method, variable, package names (1/3)

- To be a valid name:
  - Made up of letters, digits and underscore (_).
  - Start with a letter.
  - Can not be Java keywords (e.g. **`char`**, **`transient`**, …).
- To be a good name:
  - Simple.
  - Meaningful.

- Class names:
  - nouns
  - mixed case
  - capitalise 1st letter of each internal word
  - use whole words

```
public class Point {
   // ...
}
```

```
Other good names:
Diary
FileProcessor
CounterGUI
BlackJackGame
```

Naming: class, method, variable, package names (2/3)

- Variable names:
  - short
  - mixed case
  - 1st letter lowercase, and 1st letter of each internal word capitalised

```
Good names:
accountNo
accountName
balance
```

```
Common names for
temporary variables: i, j, k, m
integers: n
characters: c, d, e
```

- Constant name:
  - all uppercase
  - words separated by underscores ("_")
  - `final`

```
Other examples:
LIMIT
MAX_LENGTH
MIN_VALUE
```

```
final double PI = 3.1415926;
```

Naming: class, method, variable, package names (3/3)

- Method name:
  - verb
  - 1st letter lowercase, and 1st letter of each internal word capitalised

```
Examples:
reverse
changeCase
draw
deal
writeToFile
```

- Package name:
  - all-lowercase
  - use top-level domain name
    - `.com`, `.org`, `.gov`, `.net`, …

```
Examples:
cardgame
pontoon
carpark
cdplayer
```

Comments

- Documentation comments:

  `/** ... */` to describe the specification

  – all classes

  – at least all service methods

  – will be written in Java doc

Write comments!!!

- Implementation comments:

  `/* ... */` or `//` to comment out code or comment about the particular implementation.

- Very important when *generating Javadocs*.

- Write comments on top of a block of code.

Source File Organisation

```
/*
 * Beginning comments: File name, version, date, copyright etc ...
 */
package packagename;

import packagename.className;
import packagename.className;
 /**
  * Class documentation comments
  */
public class ClassName {
  static variables (1.public, 2.protected, 3.package level, 4.private)
  instance variables (1.public, 2.protected, 3.package level, 4.private)
  constructors (1.default constructor 2.user-defined constructors)
  methods (write documentation comments for each method)
           1.accessor methods
           2.service and support methods (grouped by functionality)
           3.toString
           4.main
}
```

Java Code Conventions – information in QMplus:

- under the **Writing and Debugging Programs** topic
- under the **Teaching Week 2** topic

Javadocs (Revision)

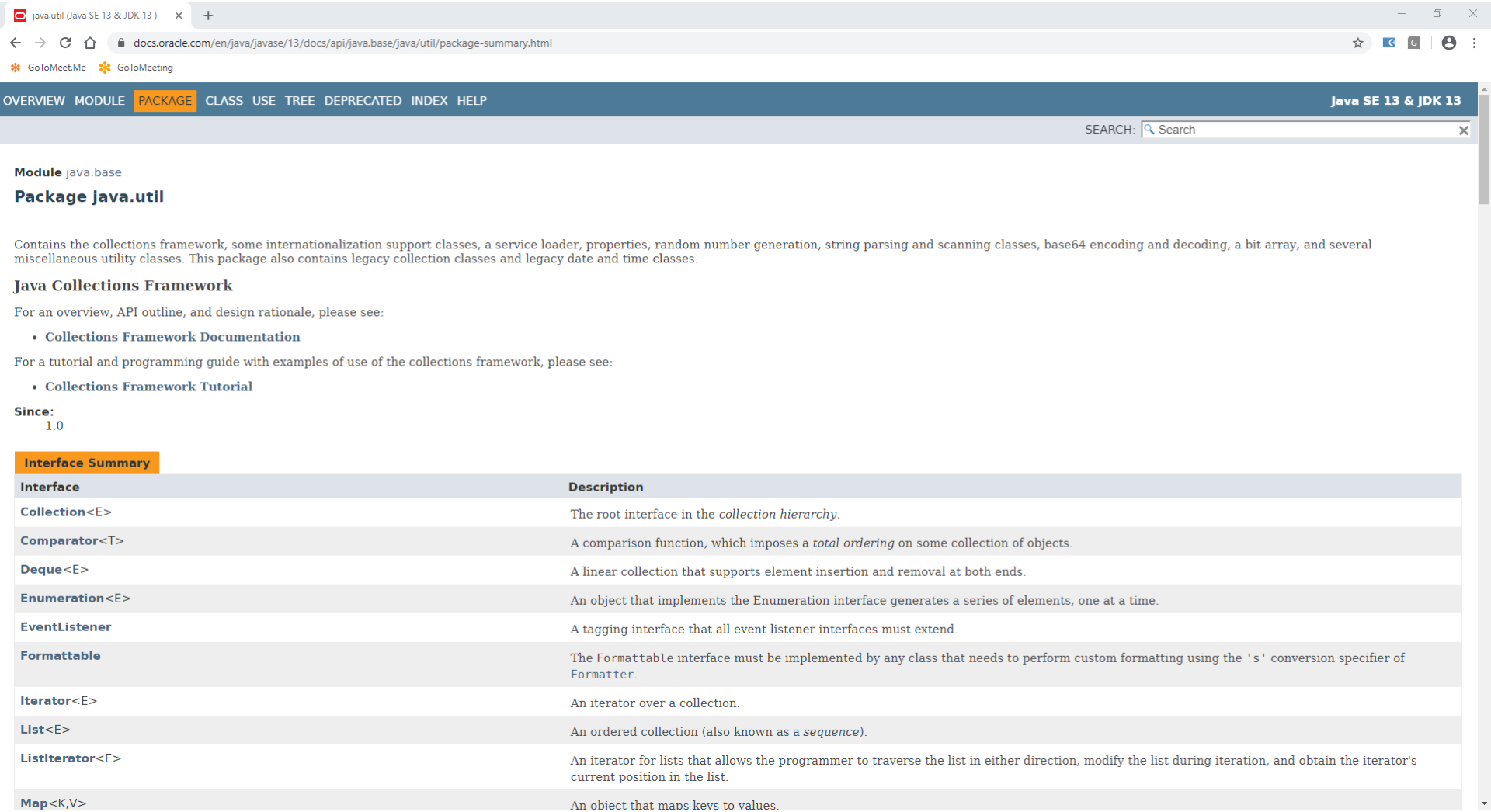
- What is Javadoc:
  - A *tool for generating API documentation* in HTML format from documentation comments in source code.
  - As seen in Java standard library.

- How to generate Javadocs:
  - Command line:

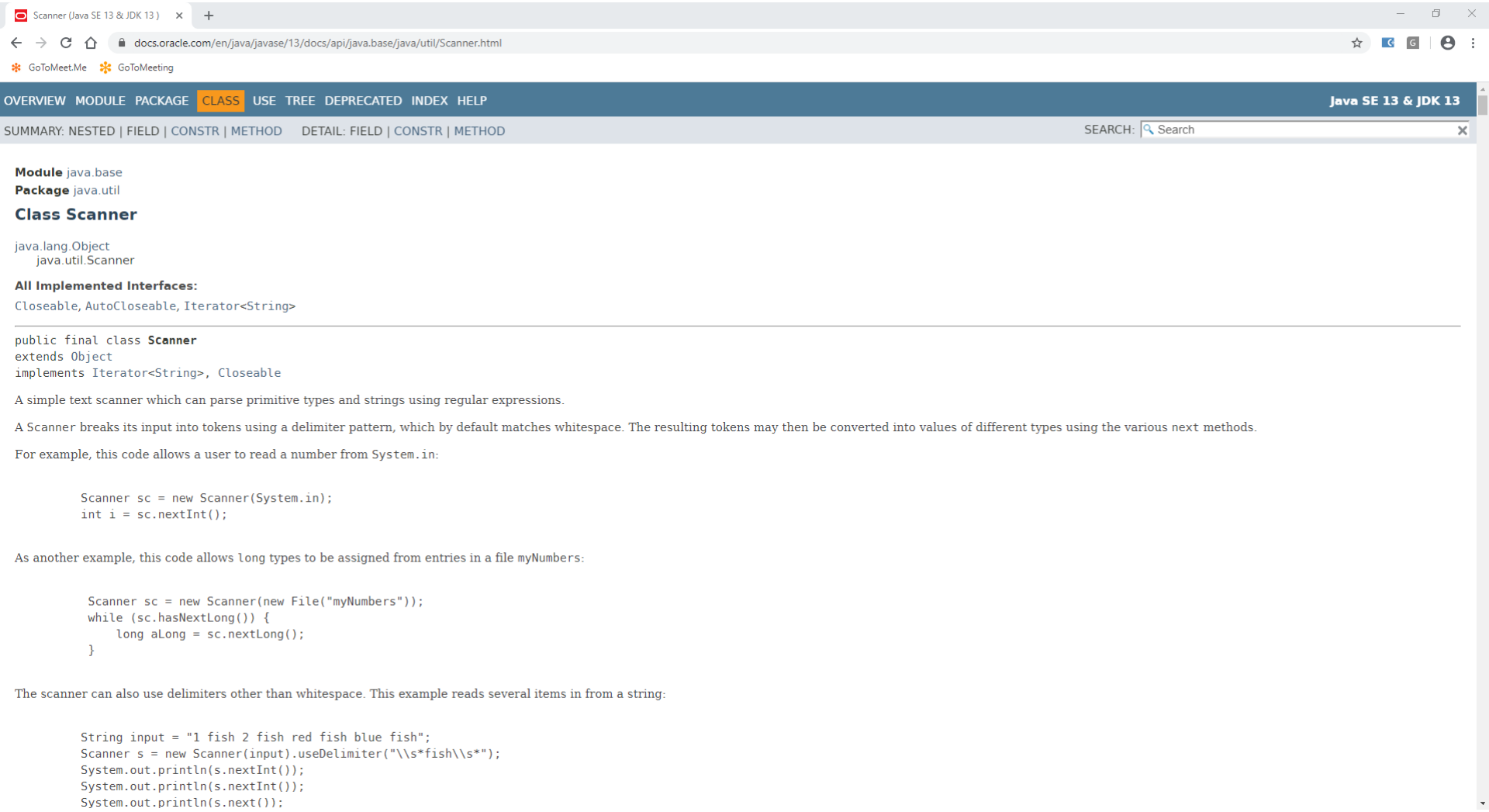
```
javadoc [options] [packagenames] [source files] [@files]
```

  - IDE:
    - Eclipse: Project → generate javadoc
    - NetBeans: Build → generate javadoc

Javadocs: Package level (Revision)



Javadocs: Class level (Revision)



Scanner (Java SE 13 & JDK 13)

docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Scanner.html

GoToMeet.Me GoToMeeting

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Java SE 13 & JDK 13

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH: Search

Module java.base
Package java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

`Closeable`, `AutoCloseable`, `Iterator<String>`

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

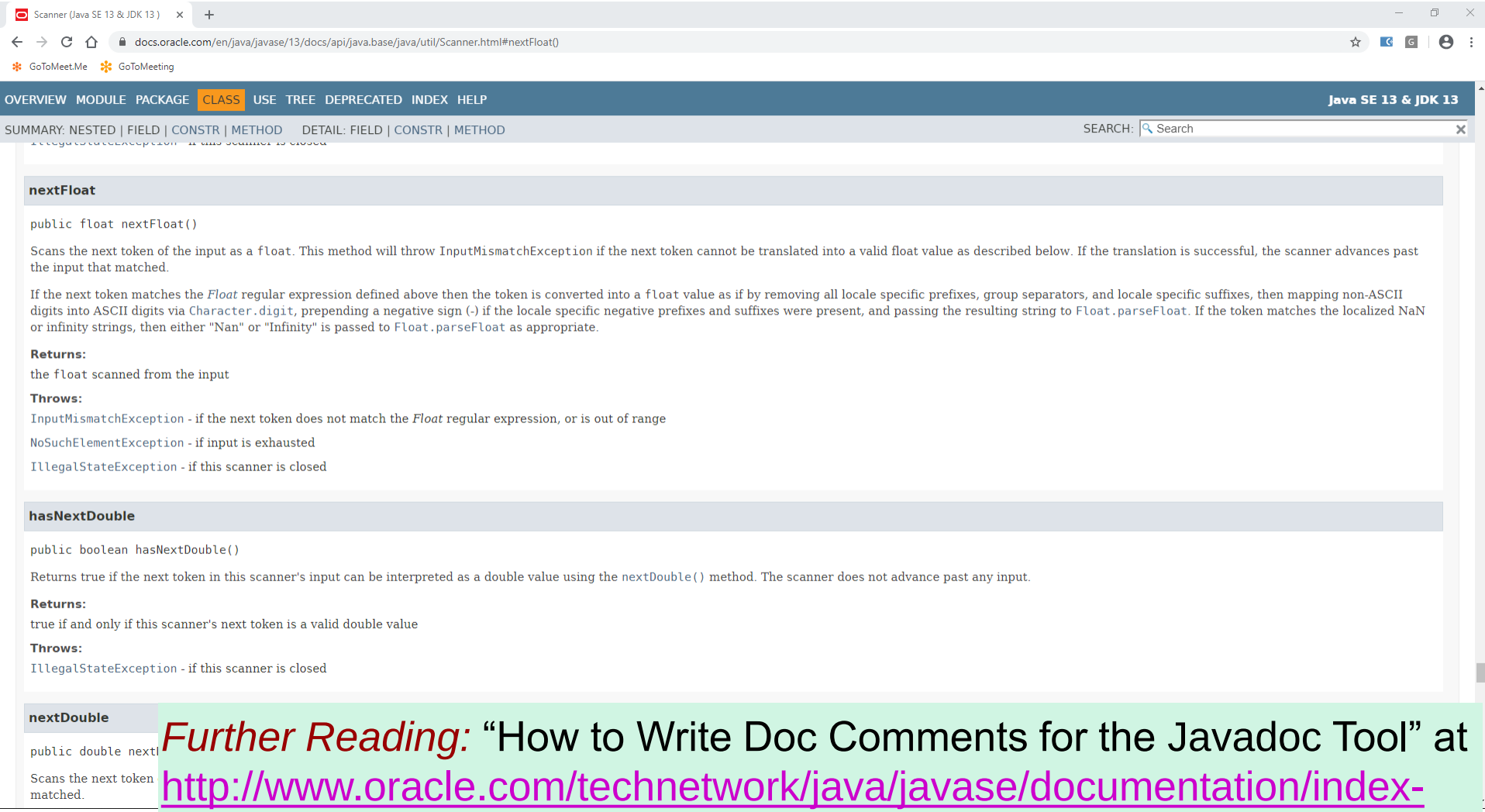
As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
```

Javadocs: method level (Revision)

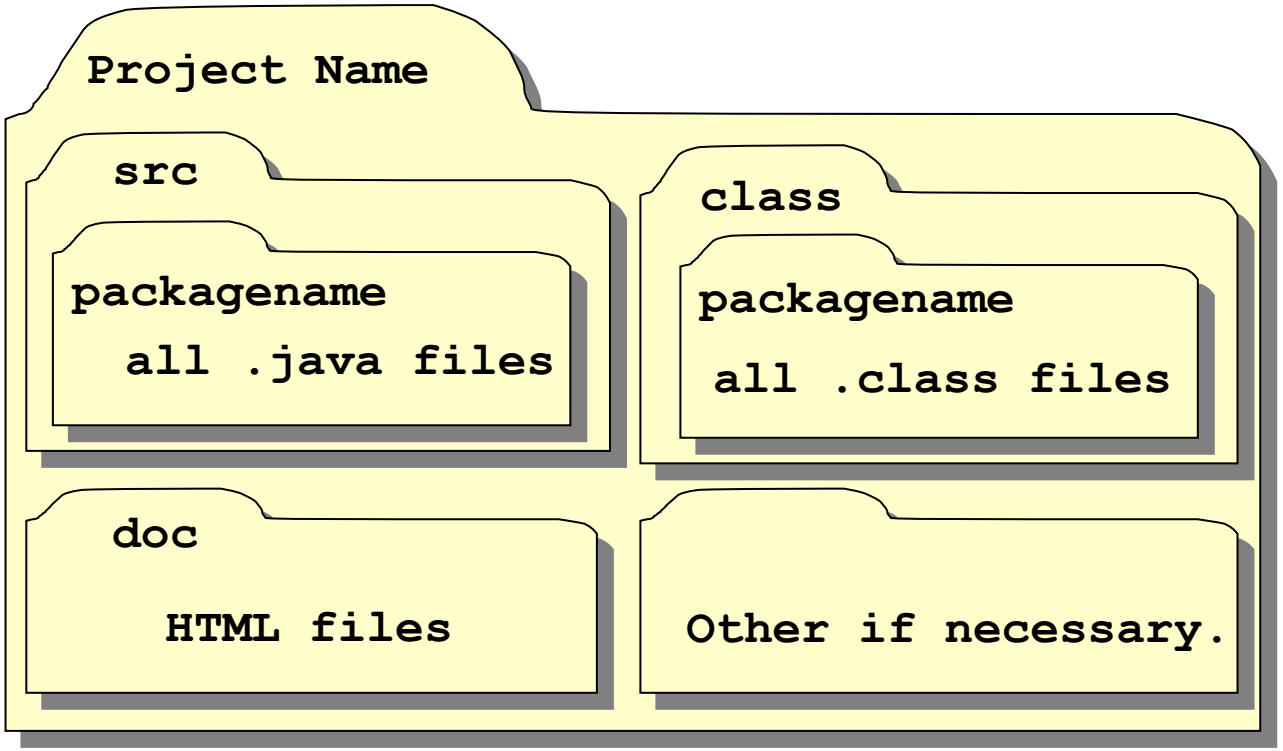


Further Reading: "How to Write Doc Comments for the Javadoc Tool" at http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

Packaging

- Package your code: add package name.
- Separate source files and class files.
- Generate Javadoc.

- Use IDE to manage your files:
  - Separate source and class.
- Finally, deliver it!
  - Make a ZIP of your top project folder. *OR*
  - Make a JAR.



Code Delivery

- JAR:
  - JavaARchive
  - Executable
  - JAR files are packaged with the ZIP file format
- Benefits of JARs:
  - data compression
  - archiving
  - decompression
  - archive unpacking
- Making JAR from an IDE:
  - Eclipse: right click your project, export → Java → JAR
  - Netbeans: right click your project → build
- May include ***`src`***, ***`class`*** and ***`doc`***.

Further reading: "Packaging Programs in JAR Files" at http://docs.oracle.com/javase/tutorial/deployment/jar/index.html.

Example using `jar` commands

Making a JAR from the command line:

- Create → `jar cf jar-file input-file(s)`
- View the contents → `jar tf jar-file`
- Extract the contents → `jar xf jar-file`
- Run application → `java -jar app.jar`