

# 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



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

**Examples:**

`reverse`  
`changeCase`  
`draw`  
`deal`  
`writeToFile`

**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
- 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.

**Write comments!!!**

# 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)

java.util (Java SE 13 & JDK 13) x +

docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/package-summary.html

GoToMeet.Me GoToMeeting

OVERVIEW MODULE **PACKAGE** CLASS USE TREE DEPRECATED INDEX HELP

Java SE 13 & JDK 13

SEARCH: Search

Module java.base

Package java.util

Contains the collections framework, some internationalization support classes, a service loader, properties, random number generation, string parsing and scanning classes, base64 encoding and decoding, a bit array, and several miscellaneous utility classes. This package also contains legacy collection classes and legacy date and time classes.

Java Collections Framework

For an overview, API outline, and design rationale, please see:

- [Collections Framework Documentation](#)

For a tutorial and programming guide with examples of use of the collections framework, please see:

- [Collections Framework Tutorial](#)

Since:  
1.0

Interface Summary

Interface	Description
<b>Collection</b> <E>	The root interface in the <i>collection hierarchy</i> .
<b>Comparator</b> <T>	A comparison function, which imposes a <i>total ordering</i> on some collection of objects.
<b>Deque</b> <E>	A linear collection that supports element insertion and removal at both ends.
<b>Enumeration</b> <E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
<b>EventListener</b>	A tagging interface that all event listener interfaces must extend.
<b>Formattable</b>	The Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter.
<b>Iterator</b> <E>	An iterator over a collection.
<b>List</b> <E>	An ordered collection (also known as a <i>sequence</i> ).
<b>ListIterator</b> <E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
<b>Map</b> <K,V>	An object that maps keys to values.

# Javadocs: Class level (Revision)

Scanner (Java SE 13 & JDK 13) x +

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*f\\s*f\\s*f\\s*f\\s*f");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
```

# Javadocs: method level (Revision)

Scanner (Java SE 13 & JDK 13) x +

docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Scanner.html#nextFloat()

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

**nextFloat**

```
public float nextFloat()
```

Scans the next token of the input as a float. This method will throw `InputMismatchException` if the next token cannot be translated into a valid float value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Float* regular expression defined above then the token is converted into a float value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Float.parseFloat`. If the token matches the localized NaN or infinity strings, then either "Nan" or "Infinity" is passed to `Float.parseFloat` as appropriate.

**Returns:**  
the float scanned from the input

**Throws:**  
`InputMismatchException` - if the next token does not match the *Float* regular expression, or is out of range  
`NoSuchElementException` - if input is exhausted  
`IllegalStateException` - if this scanner is closed

**hasNextDouble**

```
public boolean hasNextDouble()
```

Returns true if the next token in this scanner's input can be interpreted as a double value using the `nextDouble()` method. The scanner does not advance past any input.

**Returns:**  
true if and only if this scanner's next token is a valid double value

**Throws:**  
`IllegalStateException` - if this scanner is closed

**nextDouble**

```
public double nextDouble()
```

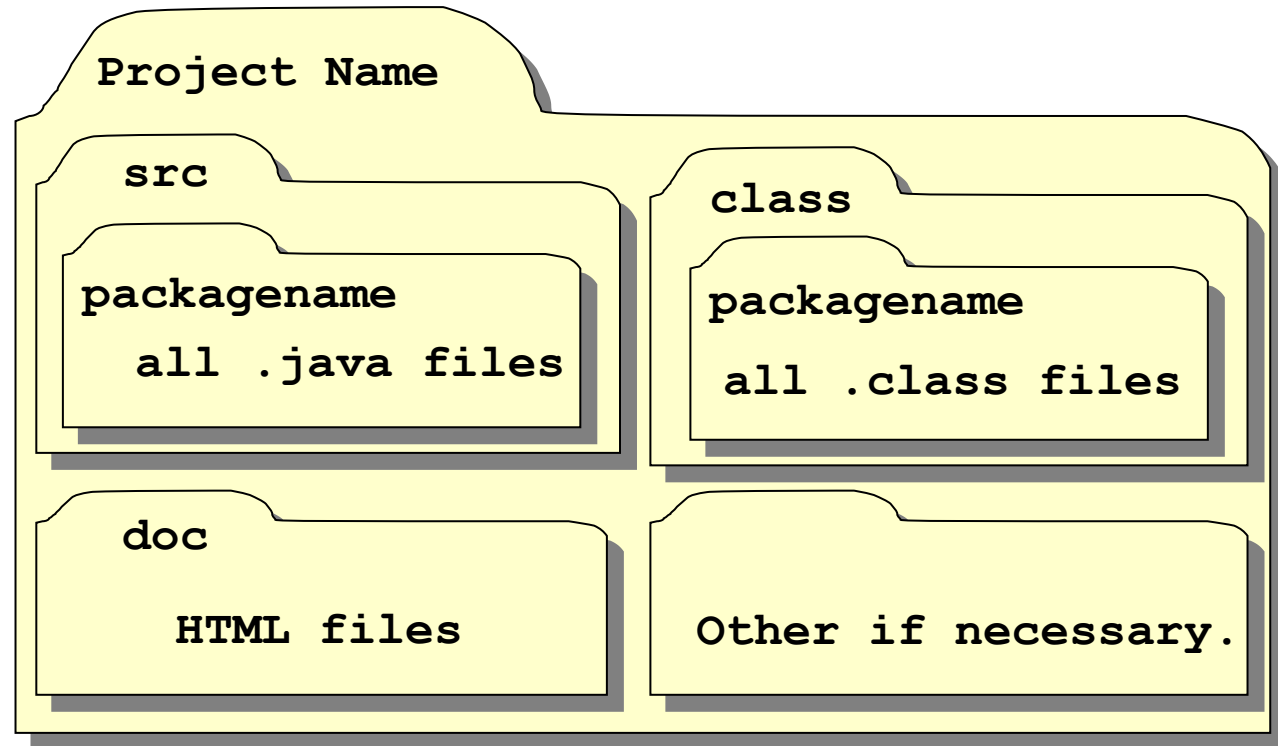
Scans the next token matched.

**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`

```
C:\> CMD

C:\coursework\ELB2222>cd classes

C:\coursework\ELB2222\classes>jar cvf myjar.jar question1/Answer1.class question1/Answer2.class question2/Answer1.class
added manifest
adding: question1/Answer1.class(in = 458) (out= 307)(deflated 32%)
adding: question1/Answer2.class(in = 458) (out= 307)(deflated 32%)
adding: question2/Answer1.class(in = 458) (out= 307)(deflated 32%)

C:\coursework\ELB2222\classes>dir
Volume in drive C has no label.
Volume Serial Number is 116F-B317

Directory of C:\coursework\ELB2222\classes

23/10/2006  02:39    <DIR>          .
23/10/2006  02:39    <DIR>          ..
23/10/2006  02:39             1,680 myjar.jar
23/10/2006  01:44    <DIR>          question1
23/10/2006  01:44    <DIR>          question2
               1 File(s)              1,680 bytes
               4 Dir(s)  16,286,064,640 bytes free

C:\coursework\ELB2222\classes>
```