

Code Conventions & Javadoc

Lab 4

Code Conventions

Java Code Conventions

- Search “java code conventions”
- Open oracle page
 - <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>
- Information in these slides is copyrighted:

Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved. Used by permission.

Why have code conventions?

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

File Suffixes

- Java software uses the following file suffixes:
 - Java source files (.java)
 - Java bytecode (.class)

Java Source Files

- Java source files have the following ordering:
 - Beginning comments
 - Package and Import statements
 - Class and interface declarations

Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

<https://www.jetbrains.com/help/idea/generating-and-updating-copyright-notice.html>

- Tasks
 - Add beginning comments to Tweet.java
 - Create copyright notice

Package and Import Statements

The first non-comment line of most Java source files is a package statement.

After that, import statements can follow.

```
package java.awt;  
  
import java.awt.peer.CanvasPeer;
```

Order of Class/Interface Declarations

- Class/interface documentation comment (`/**...*/`)
- Class/interface statement
- Class/interface implementation comment (`/*...*/`), if necessary
- Class (static) variables
 - Order: public, protected, no access modifier, private
- Instance variables
 - Order: public, protected, no access modifier, private
- Constructors
- Methods
 - Grouped by functionality, rather than scope or accessibility

Lines

- Avoid lines longer than 80 characters.
- When an expression will not fit on a single line, break it according to these general principles:
 - Break after a comma.
 - Break before an operator.
 - Align the new line with the beginning of the expression at the same level on the previous line.
 - If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Lines: break after a comma

```
someMethod(longExpression1, longExpression2, longExpression3,  
          longExpression4, longExpression5);  
  
var = someMethod1(longExpression1,  
                  someMethod2(longExpression2,  
                             longExpression3));
```

Lines: break before an operator

```
longName1 = longName2 * (longName3 + longName4 - longName5)  
          + 4 * longname6; // PREFER  
  
longName1 = longName2 * (longName3 + longName4  
                      - longName5) + 4 * longname6; // AVOID
```

The first is preferred, since the break occurs outside the parenthesized expression.

Lines: indenting method declarations

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
                                                 Object anotherArg, String yetAnotherArg,
                                                 Object andStillAnother) {
    ...
}
```

Tab = 8 spaces

Comments: block comments

- Multi-line comment.

```
/*
 * Here is a block comment.
 */
```

Comments: line comments

- If comment cannot fit on a single line, follow block comment format.
- Indent to the level of the code that follows.

```
if (condition) {  
    /* Handle the condition. */  
    ...  
}
```

Comments: trailing comments

- Short comments on the same line of code as they describe.
- Shift far enough to the right to separate them from code.
- If more than one, indent them all to the same tab setting.

```
if (a == 2) {  
    return TRUE;          /* special case */  
} else {  
    return isPrime(a);    /* works only for odd a */  
}
```

Comments: end-of-line comments

Comment out a complete line or a partial line:

```
if (foo > 1) {  
    // Do a double-flip.  
    ...  
}  
else {  
    return false;           // Explain why here.  
}
```

Comment out sections of code:

```
//if (bar > 1) {  
//  
//    // Do a triple-flip.  
//    ...  
//}  
//else {  
//    return false;  
//}
```

Declarations: number per line

- One declaration per line.

```
int level; // indentation level
int size; // size of table
is preferred over

int level, size;
```

Declarations: initialization & placement

- Instance variables: initialize in constructor.
- Local variables: initialize at declaration, put at beginning of block.

```
void myMethod() {  
    int int1 = 0;          // beginning of method block  
  
    if (condition) {  
        int int2 = 0;      // beginning of "if" block  
        ...  
    }  
}
```

Declarations: initialization & placement

- Avoid local declarations that hide declarations at higher levels.
- Do not declare the same variable name in an inner block.

```
int count;  
...  
myMethod() {  
    if (condition) {  
        int count = 0;          // AVOID!  
        ...  
    }  
    ...  
}
```

Declarations: classes and interfaces

- No space between a method name and the parenthesis "(" starting its parameter list.

```
class Sample extends Object {  
    int ivar1;  
    int ivar2;  
  
    Sample(int i, int j) {  
        ivar1 = i;  
        ivar2 = j;  
    }  
  
    int emptyMethod() {}  
  
    ...  
}
```

Declarations: classes and interfaces

- Open brace "{" appears at the end of the same line as the declaration statement.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{".

```
class Sample extends Object {  
    int ivar1;  
    int ivar2;  
  
    Sample(int i, int j) {  
        ivar1 = i;  
        ivar2 = j;  
    }  
  
    int emptyMethod() {}  
  
    ...  
}
```

Declarations: classes and interfaces

- Methods are separated by a blank line.

```
class Sample extends Object {  
    int ivar1;  
    int ivar2;  
  
    Sample(int i, int j) {  
        ivar1 = i;  
        ivar2 = j;  
    }  
  
    int emptyMethod() {}  
  
    ...  
}
```

Statements: simple statements

- Each line should contain at most one statement.

```
argv++;           // Correct  
argc--;           // Correct  
argv++; argc--; // AVOID!
```

Statements: return statements

- A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

```
return;  
  
return myDisk.size();  
  
return (size ? size : defaultSize);
```

Statements: if, if else, if else-if else statements

If-else statements
should have the
following form:

```
if (condition) {  
    statements;  
}  
  
if (condition) {  
    statements;  
} else {  
    statements;  
}  
  
if (condition) {  
    statements;  
} else if (condition) {  
    statements;  
} else {  
    statements;  
}
```

Note: if statements always use braces, {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!  
    statement;
```

Statements: for and while

```
for (initialization; condition; update) {  
    statements;  
}
```

```
while (condition) {  
    statements;  
}
```

An empty while statement should have the following form:

```
while (condition);
```

Statements: do-while and switch

```
do {  
    statements;  
} while (condition);
```

```
switch (condition) {  
case ABC:  
    statements;  
    /* falls through */  
case DEF:  
    statements;  
    break;  
case XYZ:  
    statements;  
    break;  
default:  
    statements;  
    break;  
}
```

Statements: try-catch

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

White Space: blank lines

- Two blank lines should always be used in the following circumstances:
 - Between sections of a source file
 - Between class and interface definitions
- One blank line should always be used in the following circumstances:
 - Between methods
 - Between the local variables in a method and its first statement
 - Before a block or single-line comment
 - Between logical sections inside a method to improve readability

White Space: blank spaces

- A blank space should appear after commas in argument lists.
- A keyword followed by a parenthesis should be separated by a space.

```
while (true) {  
    ...  
}
```

White Space: blank spaces

- All binary operators except ". " should be separated from their operands by spaces.
- Blank spaces should never separate unary operators:
 - Unary plus operator: +
 - Unary minus operator: -
 - Increment operator: ++
 - Decrement operator: --
 - Logical compliment operator: !

```
a += c + d;  
a = (a + b) / (c * d);  
  
while (d++ = s++) {  
    n++;  
}  
printSize("size is " + foo + "\n");
```

White Space: blank spaces

- The expressions in a for statement should be separated by blank spaces.

```
for (expr1; expr2; expr3)
```

White Space: blank spaces

- Casts should be followed by a blank space.

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
          + 1);
```

Naming Conventions: packages

- All lowercase ASCII letters.
- Top-level domain names, then organization's own internal naming conventions.

```
com.sun.eng  
com.apple.quicktime.v2  
edu.cmu.cs.bovik.cheese
```

Naming Conventions: classes and interfaces

- Should be nouns, in mixed case, with the first letter of each internal word capitalized.
- Keep names simple and descriptive.
- Use whole words; avoid acronyms and abbreviations, unless the abbreviation is more widely used than the long form, such as URL or HTML.

```
class Raster;  
class ImageSprite;
```

Naming Conventions: methods

- Methods should be verbs, in mixed case with the first letter lowercase, and with the first letter of each internal word capitalized.

```
run();  
runFast();  
getBackground();
```

Naming Conventions: variables

- Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed.
- Should be short yet meaningful.
- One-character variable names should be avoided except for temporary "throwaway" variables.
- Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

int	i;
char	c;
float	myWidth;

Naming Conventions: constants

- The names of variables declared class constants should be all uppercase with words separated by underscores ("_").

```
static final int MIN_WIDTH = 4;  
static final int MAX_WIDTH =  
999;  
  
static final int GET_THE_CPU =  
1;
```

Javadoc

What is Javadoc?

- Utility that parses code documentation from Java source files and produces a document, typically in HTML.
- Output format can be customized.

What can be documented?

- Packages
- Classes
- Fields
- Methods

General Syntax

- Start with `/**` and end with `*/`.

```
/**
 * 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.
 * <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 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) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

General Syntax: useful HTML tags

- Comments can contain some HTML tags
 - Paragraph
`<p>`
 - Line break
text `
`
 - Pre-formatted text
`<pre> text </pre>`
 - Lists
``
`Item 1`
`item 2`
``

General Syntax: documenting classes

- Put Javadoc before the class declaration.
- Recommended tags:
 - @author: who wrote the class or interface
 - @see: other related class, package or interface
 - @version: current version of the class
 - @deprecated: indicates that the class is no longer in use

General Syntax: documenting methods

- Put Javadoc before the method declaration.
- Recommended tags:
 - @param: parameters of the method
 - @return: return value of the method
 - @throws: exceptions the method may throw
 - @see: other related class, package or interface

General Syntax: documenting methods

- What to document:
 - Expected behavior: expected or desired behavior of this operation.
 - State transitions: specify what state transitions this operation may trigger.
 - Range of valid argument values
 - Null argument values - for each reference type argument, specify the behavior when null is passed in.
 - Range of return values - specify the range of possible return values, including where the return value may be null.
 - Exception - when exception may be thrown.

General Syntax: summary

- Tags start with @ and should follow this order:
 - @author: who wrote the class or interface (classes and interfaces only, required)
 - @version: current version of the class (classes and interfaces only, required)
 - @param: parameters of the method (methods and constructors only)
 - @return: return value of the method (methods only)
 - @throws: exceptions the method may throw
 - @see: other related class, package or interface
 - @since: when the code was introduced (e.g. version of the file when you added a method)
 - @deprecated: indicates that the class is no longer in use, describe deprecated item and what alternative to use instead