CS-230 Software Engineering

L09: Coding Conventions

Dr. Liam O'Reilly

Semester 1 – 2020

Moving on ...

We are now going to move on to more code focused material:

- Coding Conventions
- Version Control Systems – Git
- Javadoc
- JavaFX (modern way to do Java GUIs)
- Design of Exceptions
- ...

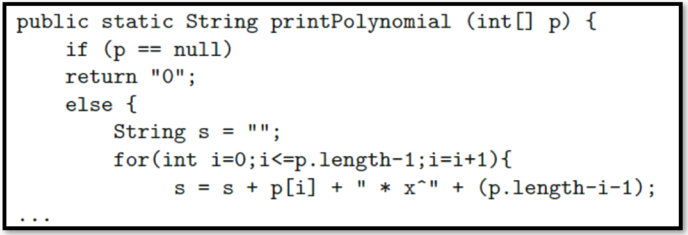
Motivation

- Writing a useful software applications is difficult.
    - For some of you, this may be first time implementing a larger, long-term project.
    - Developing large software application requires different sets of skills.
    - Following guidelines, combined with good practices, facilitates success.
    - Following standards fosters good communication between team members.

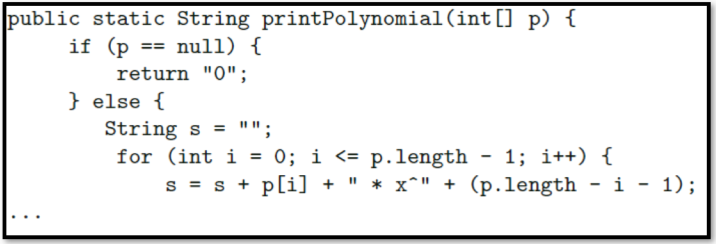
Coding Conventions

The same code but with different formatting.

Which is better? Which is easier to read?



```
public static String printPolynomial (int[] p) {
    if (p == null)
    return "0";
    else {
        String s = "";
        for(int i=0;i<=p.length-1;i=i+1){
             s = s + p[i] + " * x^" + (p.length-i-1);
...
```



```
public static String printPolynomial(int[] p) {
     if (p == null) {
         return "0";
     } else {
        String s = "";
         for (int i = 0; i <= p.length - 1; i++) {
             s = s + p[i] + " * x^" + (p.length - i - 1);
...
```

Why Coding Conventions?

Why should we bother with coding conventions?

- Makes code easier to read once you are used to the standard.
- Communication is made easier and code is more legible.
- Other people can read the coder easily.

- Basic philosophy behind conventions is to maximise legibility.
- Legible software contains fewer bugs, more stable.
- Legible software is more flexible, encourages re-use.

- Coding conventions will be marked as part of A2.

Why Coding Conventions? (2)

- Allows bugs to be spotted by deviations from the standard.
- Good coding conventions can lead to less bugs. E.g., Some code conventions for C forbid:

  ```
  if (x == SOME_CONSTANT) {
  ```

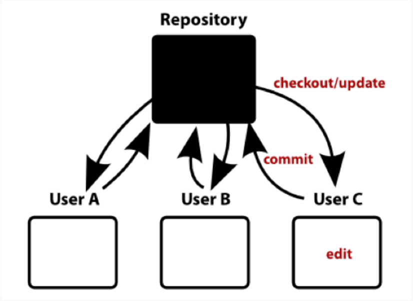
  and require instead

  ```
  if (SOME_CONSTANT == x) {
  ```

  Reason: If you miss out one of the equal signs then the compiler will not be able to assign to a constant. Bug avoided.

Why Coding Conventions? (3)

Version control systems like Git and Subversion (covered later in the module) allow multiple people to work on a project simultaneously.



If all the users don't agree on a specific coding style then style wars can break out.

Even worse, you end up creating needless conflicts when committing code changes back to the repository.

Content of Coding Conventions

Coding conventions are (hard/soft) guidelines for laying out code.

Examples of things you might find:

- Indentation: spaces vs tabs, how many spaces.
- Placement of '{' and '}' characters.
- White space around operators.
- Naming schemes (e.g., prefix variables with type information, e.g., `f_amount`, `i_age`).
- Maximum length of lines.
- Maximum length of methods.
- Maximum depth of nesting.
- What commenting is required.
- Other principals and practices.

Every Company Has Its Own Style

- JavaRanch – a friendly place for Java greenhorns
  `http://www.javaranch.com/style.jsp`
- GeoSoft – Geotechnical Software Services
  `http://geosoft.no/development/javastyle.html`
- Google – Google Java Style
  `https://google.github.io/styleguide/javaguide.html`
- Oracle – Code Conventions for the Java Programming Language
  `http:`
  `//www.oracle.com/technetwork/java/index-135089.html`
- (Old) Sun Microsystems Java Conventions
  `http://web.archive.org/web/20140228225807/http://www.`
  `oracle.com/technetwork/java/codeconventions-150003.pdf`

Soft or Hard Rules?

Some coding conventions can be softer that others.

It is possible to break a rule – for a good reason. However, you must never ignore rules (and pretend they don't exist).

Different standards and companies "enforce" their rules in different ways and have different degrees of leniency when it comes to breaking the rules.

CS-230 Coding Conventions

Why These Coding Conventions?

The conventions here are mostly the Java Sun conventions (most others Java coding conventions are actually based on the Sun conventions).

You will likely come into contact with these in your future careers.

For CS-230's Assignment Two (A2):

- You must follow the coding conventions stipulated from this slide onwards.
- Some rules have exceptions that are clearly marked.
- Rules marked with Soft Rule are soft rules and can be broken for good reasons.

Order Within a Java Source File

Java source files must have the following ordering:

1. Import statements.
2. Javadoc for class.
3. Class declaration.

```
import java.util.ArrayList;
import java.util.Scanner;

/**
 * A menu that is displayed on a console.
 * @author Liam O'Reilly
 * @version 1.6
 */
public class Menu {
    ...
```

Each class must be stored in its own file.
Exception: inner classes (e.g., used for GUI event handlers) may be declared within the class that is using it.

Order Within a Class

The order of a class must be:

1. Javadoc for class
2. Class declaration
3. Class constants (static & final) variables
4. Class (static) variables
5. Instance variables
6. Constructors
7. Methods

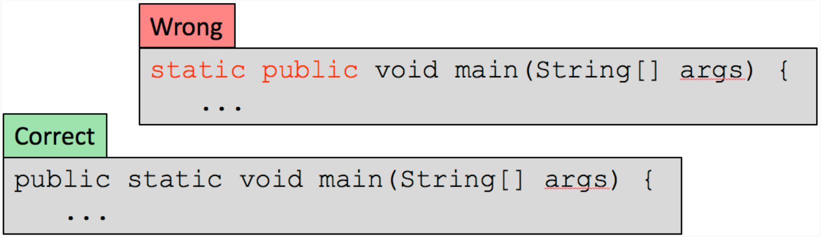
Within each of the above the items must appear in the order:

- First the public, then protected, then package level (no access modifier), and then private.

Order of Modifiers

The modifiers in a single constructor/method declaration must appear in the following order:

- public/protected/private
- abstract
- static
- final



Wrong

```
static public void main(String[] args) {
   ...
```

Correct

```
public static void main(String[] args) {
   ...
```

No Long Lines of Code

- No line of code shall exceed 80 characters in length.

Soft Rule

- Why?
  - Lines that are too long are hard to read and debug.
  - The longer a line is, the more difficult it is on the eyes.
  - Publisher guideline of approximately 66 (so 80 is generally too much) [Oetiker et al, 2008].
    - Reason why most newspapers and magazines are multi-column.
  - When programming multiple windows are usually open simultaneously.
  - Having one window full screen makes the programmer's job much more difficult [Sun Microsystems, 1999].

No Long Methods

- In your code there should be no long methods: Soft Rule
    - Methods must be a maximum of 75 lines.
    - Method should be visible on a single screen/page.
    - Possible to see whole method from start to finish (without scrolling).

Why No Long Methods?

- Why shall we not have long methods?
  - The longer a method is...
    - The less re-usable and more difficult it is to modify.
    - It is more likely to contain bugs and hard to debug.
    - It's role in the design becomes imprecise.
  - By confining method to one screen, the programmer has a chance to keep track of variables from beginning to end.
  - Conformance to this rule facilitates code optimisation [Meyers'96].
- Solution: Break method up into "sub"-methods and use method calls.

Restriction on Number of Nested Statements

- Methods shall use no more than 5 levels of indentation.

Soft Rule

- Why low indentation?
- Many levels
  - make code
    - hard to read.
- Many nested loops make code inefficient.
- Solution: Break method up into “sub”-methods and use method calls.

Restriction on Number of Method Parameters

- Methods should not require more than 5 parameters.

Soft Rule

- Exceptions are rare.
- Why?
    - The more parameters a method takes, the less re-usable it is.
    - Better to have different implementations of same method taking different parameters.
    - A long list of parameters may indicate that changes to design are necessary, e.g., the introduction of a new class(es) or rearrangement of existing classes [Sun Microsystems 1999].

Break and Continue Statements

- Never use `continue` statements.
- Never use `break` statements other than in a `switch` statement.
- Why?
  - They make it difficult to later break a construct into smaller constructs or methods.
  - It also forces the developer to consider more than one end-point for a construct.
  - The use of these leads to hard to understand code that often contains bugs.

Identifiers and Case

- Classes (and Interfaces) must be written in camel case style and start with an upper-case letter. e.g., `SalesOrder`. Exception: acronyms, may be all upper case, e.g., `URLTarget`.
- Constants must be written in block capital letters with underscores between words e.g., `VAT_RATE`.
- All other identifiers, including (but not limited to) fields, local variables, methods and parameters must be written in camel case style and start with a lower-case letter. e.g., `totalCost`.

These rules allow us to easily distinguish classes, variables and constants from each other.

Meaningful Identifiers

- Class names must be singular nouns,
  e.g., `Boat` and not `Sailing`
- Variable names should be meaningful,
  e.g., `totalCost` and not just `tc`.
- Variables should normally be singular, expect for collections such as arrays, these should be named plural e.g.,

```
Person neighbour;
Person[] employees;
```

- Methods use meaningful and "verb-like" names,
  e.g., `calculatePay()`.

These rules allow us to easily identify classes, variables and constants.

Data Encapsulation

- All class variables (instance variables and static variables) must be private.
- Exception: Constants. Constants can be publicly available.
- All class attributes are accessed using get/set methods when accessed externally.
- Why?
    - Enforces encapsulation.
    - Can change the data structure (within the class) without updating code in collaborating classes.
    - Only the class itself should know about the specific implementation details of its own data [Meyers 2005].

No Magic Numbers in Your Code

- Do not use numbers in your code, but rather constants.
- Exceptions: 0 or 1 (sometimes 2).

- What the heck is 4 supposed to mean?
- One 6 may not be same as another 6. [Sutter and Alexandrescu 2005]
- Using constants conveys what you intend the numbers to mean.
- Constants can remind you what you mean (especially for large software).
- Constants makes values in the code easy to change.
- Changing values of numbers directly in the code causes bugs, especially when the number appears in multiple places [Sun Microsystems 1999].

Javadoc Comments

- Javadoc allows for automatic generation of documentation from code.
- All classes require a class level Javadoc comment that describes the overall purpose of the class. The `@author` tag must be used to specify the author.
- All methods must have a Javadoc comment. The parameters must be correctly described using `@param` tags. You must also document the return values using the `@return` tag.
- All Javadoc for public entities must not discuss or expose implementation details.

Note: Javadoc has been taught in CS-135, but we will look at it again later in this module.

Indentation: Spaces vs Tabs

This is a tricky one. Lots of disagreement.

Examples:

- Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).
- Only tabs – 1 tab per level. Allows each programmer to customise in their editor how many spaces should be "displayed" per tab character.

For CS-230 I do not mind which you use – but you must be consistent.

One Declaration p=Per Line

Number declarations per Line: Only one declaration per line.

Wrong:

```
int height, weight;
```

Correct:

```
int height; // in cm
int weight; // in grams
```

Why?

- Allows easier commenting.
- Leads to less bugs if types are changed.

Placement of Declarations

Always declare variables with the minimum scope to get the job done. (This is a hard rule)

Declare variables as close as possible to where they are used.

Soft Rule

- This is another rule with lots of disagreement.
- Some conventions/people state you should only declare variables at the beginning of blocks.
- A block is any code surrounded by curly braces "{" and "}".

Java Classes

- No blank space between a method name and the parenthesis “(“ starting its parameter list.
- Open brace “{ “ appears at the end of the same line as the declaration statement with a space before it.
- A single blank space before the opening brace.
- Closing brace “}” starts a line by itself indented to match its corresponding opening statement.
- Methods are separated by a single blank line.

Java Class Example

```java
public class Sample extends AnotherClass {
    private int ivar1;
    private int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    public int aMethod() {
        return 42;
    }
}
```

Space

No Space

No Space

Lines Up with Start of Opening Statement

`if`, `if-else`, `if-else-if-else` **Statements**

- Blank space after keywords, e.g., if, else and else if.
- No space after open parenthesis and no space before closing.

Space

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

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

No Space

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

Loops

- For loops: Space after the semi-colons.
- Space after keywords, e.g., while and do.

Space

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

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

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

Optional Braces

All `if`, `while` and `for` statements must always use braces even if they control just one statement.

Reason: If they are omitted and code is added to the body of the construct, often programmers do not notice the missing braces.

Method Declarations and Method Calls

- A keyword followed by a parenthesis should be separated by a blank space.

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

- But no space between method name and its opening parenthesis.

```
someVariable = aMethodCall(42);
```

- A space after commas in argument lists.

```
anotherVariable = methodCall(42, 100, 52);
```

Unary and Binary Operators

- All binary operators (except ".") must be separated from their operands by a single space.
- Blank spaces should not be used to separate unary operators from their operands. For example:
  - unary minus (-),
  - increment (++), and
  - decrement (--)

```
n = -9;
a += c - d;
a = (a + b) / (c * d);
while (d++ == s++) {
    n++;
}
System.out.println("size is " + foo);
```

Arrays and Casting

All array access should be immediately followed by a left square bracket.

- Wrong:

```
x = myArray [0];
```

- Correct:

```
x = myArray[0];
```

All casts should be written with a single space.

- Wrong:

```
x = (int)foo(42);
```

- Correct:

```
x = (int) foo(42);
```

Checkstyle

- Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.
- The rules Checkstyle uses can be configured.
- It automates the process of checking Java code to spare humans of this boring (but important) task.
- This makes it ideal for projects that want to enforce a coding standard.

You covered this last year in CS-135.

Summary

- Coding conventions are important:
    - Makes code easier to read.
    - Good conventions can help reduce bugs.
    - Critical for team work when using version control systems.
- CS-230 Coding conventions:
    - Presented in this lecture.
    - Must be used for A2.
    - Are (mostly) based on standard conventions.