LECTURE 3: CODE REVIEW

Dr. Ehab Essa
Computer Science Department,
Mansoura University

CODE REVIEW

Code review is careful, systematic study of source code by people who are not the original author of the code. It's analogous to proofreading a term paper.

Code review really has two purposes:

Improving the code. Finding bugs, anticipating possible bugs, checking the clarity of the code, and checking for consistency with the project's style standards.

Improving the programmer. Code review is an important way that programmers learn and teach each other, about new language features, changes in the design of the project or its coding standards, and new techniques.

Code review is widely practiced in open source projects like Apache and Mozilla. Code review is also widely practiced in industry. At Google, you can't push any code into the main repository until another engineer has signed off on it in a code review.

STYLE STANDARDS

Coding style standards are guidelines for writing computer code.

Most companies and large projects have coding style standards (for example, Google Java Style). it prescribes specific formatting (like brace use and indentation), naming conventions for variables, classes, and methods, and defense for clear, concise comments and Javadoc documentation.

There are some style guide rules that are quite sensible:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Return results, don't print them
- Use whitespace for readability

DON'T REPEAT YOURSELF (DRY)

Duplicated code is a risk to safety. If you have identical or very similar code in two places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other.

Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the block you're copying, the riskier it is.

The DRY principle is rooted in the reduction of repetition. The idea is to encapsulate logic or configuration that is used in more than one place within reusable components, such as functions, classes, modules, etc., to promote ease of maintenance and scalability.

Experienced developers advocate for the DRY principle and caution against habitual copy-pasting.

SMELLY EXAMPLE #1

The dayOfYear example is full of identical code. How would you DRY it out?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

- How many times is the number of days in April written in dayOfYear()?
- Suppose our calendar changed so that February really has 29 days instead of 28. How many numbers in this code have to be changed?
- Another kind of repetition in this code is dayOfMonth+=
  - Assume you have an array: int[] monthLengths = new int[] { 31, 28, 31, 30, ..., 31}
  - How to DRY the code out enough so that dayOfMonth+= appears only once?

COMMENTS WHERE NEEDED

Good software developers write comments in their code, and do it judiciously. Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change.

One kind of crucial comment is a **specification**, which appears above a method or above a class and documents the behavior of the method or class. Specifications document assumptions.

In Java, this is conventionally written as a Javadoc comment, meaning that it starts with /** and includes @-syntax, like @param and @return for methods. Here's an example of a spec:

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture#Statement_of_the_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

COMMENTS WHERE NEEDED

Another crucial comment is one that specifies the source of a piece of code that was copied or adapted from elsewhere. This is vitally important for practicing software developers

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-cod
e
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDeli
miter("\\A").next();
```

One reason for documenting sources is to avoid violations of copyright. Another reason for documenting sources is that the code can fall out of date; for example you can have multiple or improved answer over time on the Stack Overflow

COMMENTS WHERE NEEDED

Some comments are bad and unnecessary. Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java:

```
while (n != 1) { // test whether n is 1   (don't write comments like this!)
   ++i; // increment i
   l.add(n); // add n to l
}
```

But obscure code should get a comment:

```
sendMessage("as you wish"); // this basically says "I love you"
```

COMMENTS WHERE NEEDED

Which comments are useful additions to the code? Consider each comment independently, as if the other comments weren't there. Check all that apply.

```
/** @param month month of the year, where January=1 and December=12  [C1] */
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {      // we're in February  [C2]
        dayOfMonth += 31;  // add in the days of January that already passed  [C3]
    } else if (month == 3) {
        dayOfMonth += 59;  // month is 3 here  [C4]
    } else if (month == 4) {
        dayOfMonth += 90;
    }
    ...
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth; // the answer  [C5]
}
```

FAIL FAST

Failing fast means that code should reveal its bugs as early as possible.

The earlier a problem is observed (the closer to its cause), the easier it is to find and fix.

As we saw in the static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

The dayOfYear function doesn't fail fast — if you pass it the arguments in the wrong order, it will quietly return the wrong answer.

In fact, the way dayOfYear is designed, it's highly likely that a non-American will pass the arguments in the wrong order! It needs more checking — either static checking or dynamic checking.

FAIL FAST

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Suppose the date is January 3, 2019. The correct dayOfYear for this date is 3,

Which of the following are **plausible** ways that a programmer might (**mistakenly**) call dayOfYear? And for each one, does it lead to a static error, dynamic error, or wrong answer?

1. `dayOfYear(1, 3, 2019)`
2. `dayOfYear(0, 3, 2019)`
3. `dayOfYear(3, 1, 2019)`
4. `dayOfYear(1, 2019, 3)`
5. `dayOfYear("January", 3, 2019)`

plausible mistake -- static error
plausible mistake -- dynamic error
plausible mistake -- wrong answer
plausible mistake -- right answer
implausible mistake
not a mistake -- right answer

AVOID MAGIC NUMBERS

There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1. All other constants are called magic, because they appear as if out of thin air with no explanation.

One way to explain a number is with a comment, but a far better way is to declare the number as a named constant with a good, clear name.

dayOfYear is full of magic numbers:

AVOID MAGIC NUMBERS

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

- The months 2, …, 12 would be far more readable as FEBRUARY, …, DECEMBER.
- The days-of-months 30, 31, 28 would be more readable (and eliminate duplicate code) if they were in a data structure like an array, list, or map, e.g. MONTH_LENGTH[month].
- The mysterious numbers 59 and 90 are particularly pernicious examples of magic numbers. They are actually the result of a computation done by hand
- Don’t hardcode constants that you’ve computed by hand. Java is better at arithmetic than you are.
- Explicit computations like 31 + 28 make the provenance of these mysterious numbers much clearer. MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY] would be clearer still.

ONE PURPOSE FOR EACH VARIABLE

In the dayOfYear example, the parameter dayOfMonth is reused to compute a very different value — the return value of the function, which is not the day of the month.

Don't reuse parameters, and don't reuse variables.

Variables are not a scarce resource in programming. Introduce them freely, give them good names, and just stop using them when you stop needing them. You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.

Not only is this an ease-of-understanding question, but it's also a safety-from-bugs and ready-for-change question.

Method parameters, in particular, should generally be left unmodified. (This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn't blow them away while you're computing.)

It's a good idea to use final for method parameters, and as many other variables as you can. The final keyword says that the variable should never be reassigned, and the Java compiler will check it statically. For example:

USE GOOD NAMES

Good method and variable names are long and self-descriptive. Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables.

For example, you can rewrite

```
int tmp = 86400;  // tmp is the number of seconds in a day (don't do this!)
```

as:

```
int secondsPerDay = 86400;
```

In general, variable names like tmp, temp, and data are awful, symptoms of extreme programmer laziness.

Every local variable is temporary, and every variable is data, so those names are generally meaningless.

Better to use a longer, more descriptive name, so that your code reads clearly all by itself.

USE GOOD NAMES

Follow the lexical naming conventions of the language. In Python, classes are typically Capitalized, variables are lowercase, and words_are_separated_by_underscores.

In Java:

- methodsAreNamedWithCamelCaseLikeThis
- variablesAreAlsoCamelCase
- CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES
- ClassesAreCapitalized
- packages.are.lowercase.and.separated.by.dots

Method names are usually verb phrases, like getDate or isUpperCase, while variable and class names are usually noun phrases.

Choose short words, and be concise, but avoid abbreviations. For example, message is clearer than msg, and word is so much better than wd.

ALL_CAPS_WITH_UNDERSCORES is used for static final constants. All variables declared inside a method, including final ones, use camelCaseNames.

USE WHITESPACE TO HELP THE READER

Use consistent indentation. The leap example is bad at this:

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
      if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true;
        else
            return false;
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false;
        }
            if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true;
    }
    return false;
}
```

You'll notice else is not lined up with its if. and if isn't lined up with the other if that's in the same block.

No good reason for having bad indentation, it makes your code harder to read. And your programming editor has an automatic feature that reformats code.

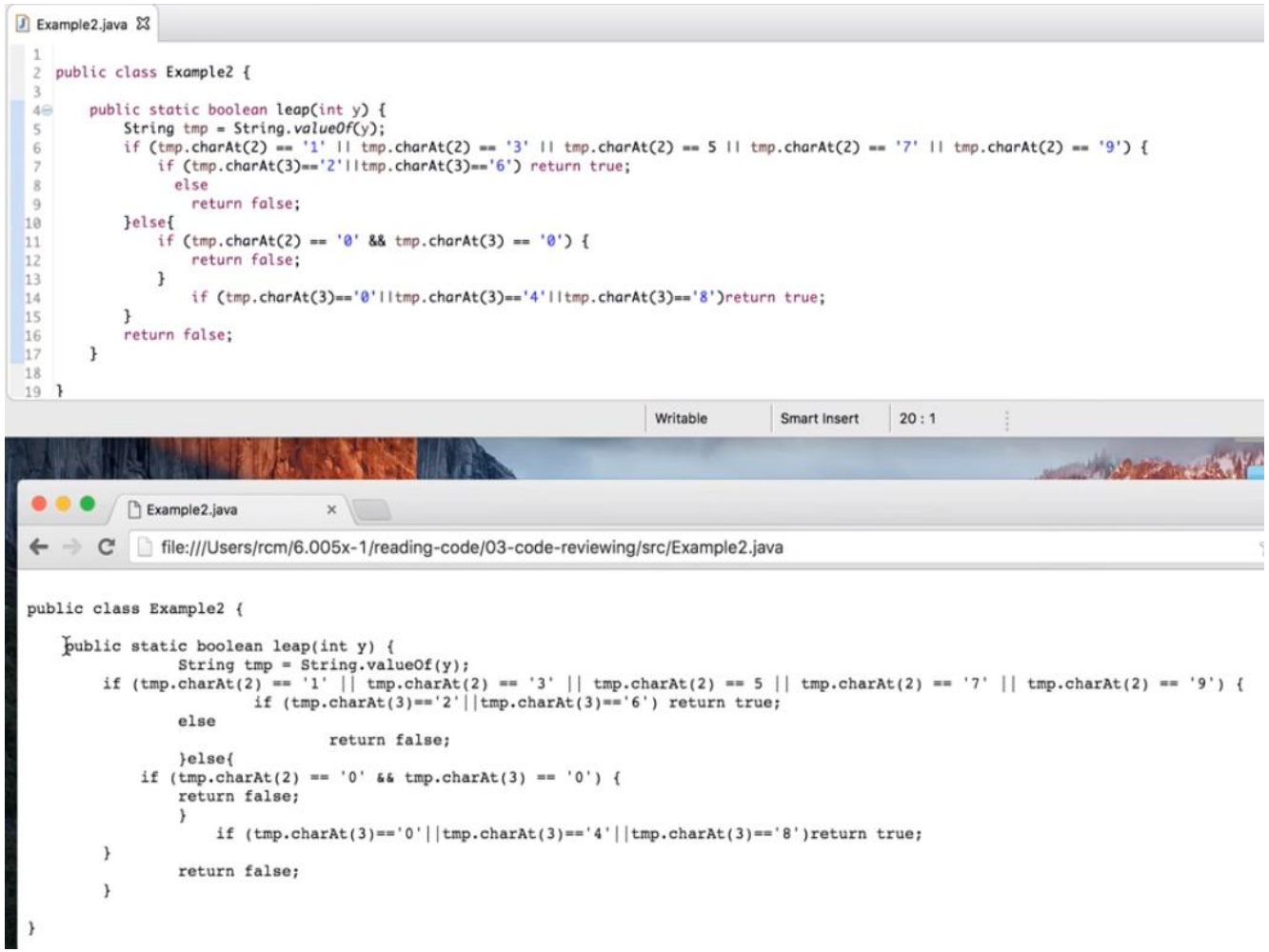
Put spaces within code lines to make them easy to read. The leap example has some lines that are packed together — put in some spaces.

USE WHITESPACE TO HELP THE READER

Never use tab characters for indentation, only space characters.

Note that we say characters, not keys. We're not saying you should never press the Tab key, only that your editor should never put a tab character into your source file in response to your pressing the Tab key.

The reason for this rule is that different tools treat tab characters differently — sometimes expanding them to 4 spaces, sometimes to 2 spaces, sometimes to 8. If another programmer views your source code in a different editor, then the indentation may be completely screwed up. Just use spaces.



DON'T USE GLOBAL VARIABLES; RETURN, DON'T PRINT

Avoid global variables. Let's break down what we mean by global variable. A global variable is:

- a variable, a name whose meaning can be changed
- that is global, accessible and changeable from anywhere in the program.

Why Global Variables Are Bad has a good list of the dangers of global variables.

In Java, a global variable is declared public static. The public modifier makes it accessible anywhere, and static means there is a single instance of the variable.

In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on.

METHODS SHOULD RETURN RESULTS, NOT PRINT THEM

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

countLongWords isn't ready for change. It sends some of its result to the console, System.out. That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten.

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results.

The sole exception here is debugging output, which can of course be printed to the console. But that kind of output shouldn't be a part of your design, only a part of how you debug your design.

CODE REVIEW

Safe from bugs. In general, code review uses human reviewers to find bugs.

- DRY code lets you fix a bug in only one place, without fear that it has propagated elsewhere.
- Commenting your assumptions clearly makes it less likely that another programmer will introduce a bug.
- The Fail Fast principle detects bugs as early as possible.
- Avoiding global variables makes it easier to localize bugs related to variable values, since non-global variables can be changed in only limited places in the code.

Easy to understand. Code review is really the only way to find obscure or confusing code, because other people are reading it and trying to understand it.

- Using judicious comments, avoiding magic numbers, keeping one purpose for each variable, using good names, and using whitespace well can all improve the understandability of code.

Ready for change. Code review helps here when it's done by experienced software developers who can anticipate what might change and suggest ways to guard against it.

- DRY code is more ready for change, because a change only needs to be made in one place.
- Returning results instead of printing them makes it easier to adapt the code to a new purpose.