

# Code Sense

# Topics

- 1) How can we know what good code is?
- 2) What techniques make my code higher quality?
- 3) How to remove duplication?
- 4) How to think about architectures?

# Code Sense

- What to look for during code review?
  - Teammates will look for these things, so look for it in your own code
- Your "code sense" is:
  - 1) Being able to..
  - 2) Knowing..
  - 3) Having the..

(Robert C. Martin, Clean Code 2008)

# Quality in the Code

# Naming

- Use..
  - The name should tell you everything you need to know about the function.
- ..
  - Based on the name of a function, variable, or class, a reviewer should be the least surprised by what they find it does, holds, or manages.

# What does this code do?

- What does this function do?
- What names could we improve?

```
int big(int[] d) {
    if (d.length == 0) {
        throw new IllegalArgumentException();
    }
    int retMe = d[0];
    for (int i = 0; i < d.length; i++) {
        int val2 = d[i];
        if (val2 > retMe) {
            retMe = val2;
        }
    }
    return retMe;
}
```

# Same code, new names

- Is this code less surprising?

```
int findMax(int[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("empty array");
    }
    int max = data[0];
    for (int i = 0; i < data.length; i++) {
        int val = data[i];
        if (val > max) {
            max = val;
        }
    }
    return max;
}
```

# Alt Implementations

- Are these better or worse?

```
int findMax2(int[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("empty array");
    }
    int max = data[0];
    for (int d : data) {
        max = Math.max(d, max);
    }
    return max;
}
```

```
int findMaxBrief(int[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("empty array");
    }

    int max = data[0];
    for (int i = 1;
        i < data.length;
        max = (max > data[i]) ? max : data[i], i++)
        ;
    return max;
}
```



# Comments

- Comments can be..
  - If you have a “code smell”, adding comments masks smell of bad code.
  - Should clean the code instead!

```
int findMaxBrief(int[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("array must not be empty");
    }

    // Loop through all the data, finding the max
    // note the ? and , operators! Don't mess it up!
    // ? operator format is (condition)? <then> : <else>;
    // , operator executes the part before , and part after as one stmt
    int max = data[0];
    for (int i = 1;
        i < data.length;
        max = (max > data[i]) ? max : data[i], i++);
    return max;
}
```

# Long methods

- Long methods reduce maintainability
  - May not fit on screen
  - ..
- Refactor:
  - Extract method
    - Each does one thing
    - Top level method has higher abstraction

```
void makeLeet(char[] msg) {  
    // replace e with 3  
    for (int i = 0; i < msg.length; i++) {  
        if (msg[i] == 'e') {  
            msg[i] = '3';  
        }  
    }  
  
    // Remove duplicate letters  
    int tar = 0;  
    for (int i = 0; i < msg.length - 1; i++) {  
        if (msg[i] != msg[i + 1]) {  
            msg[tar] = msg[i];  
            tar++;  
        }  
    }  
    for (int i = tar; i < msg.length; i++) {  
        msg[i] = '_';  
    }  
  
    // Capitalize every other letter  
    for (int i = 0; i < msg.length; i++) {  
        if (i % 2 == 0) {  
            msg[i] =  
                Character.toUpperCase(msg[i]);  
        }  
    }  
}
```

# Refactored

```
void makeLeet2(char[] msg) {
    replaceEWith3(msg);
    removeDuplicateLetters(msg);
    capitalizeEveryOtherLetter(msg);
}
```

```
private void capitalizeEveryOtherLetter(
    char[] msg)
{
    for (int i = 0; i < msg.length; i++) {
        if (i % 2 == 0) {
            msg[i] =
                Character.toUpperCase(msg[i]);
        }
    }
}
```

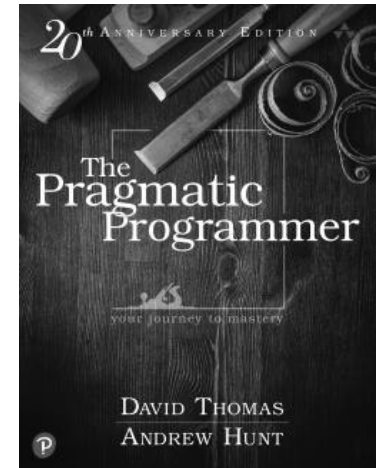
```
private void replaceEWith3(char[] msg) {
    for (int i = 0; i < msg.length; i++) {
        if (msg[i] == 'e') {
            msg[i] = '3';
        }
    }
}
```

```
private void removeDuplicateLetters(
    char[] msg)
{
    int tar = 0;
    for (int i = 0; i < m.length - 1; i++) {
        if (msg[i] != msg[i + 1]) {
            msg[tar] = msg[i];
            tar++;
        }
    }
    for (int i = tar; i < msg.length; i++) {
        msg[i] = '_';
    }
}
```

# Code Duplication

# Copy and Paste

- DRY: Don't repeat yourself
  - “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”  
(Any Hunt and Dave Thomas  
in The Pragmatic Programmer)
  - “When the same thing is done over and over, it's a sign that there is an idea in our mind that is not well represented in the code.”  
(Ron Jeffries)
- Where it applies
  - Constants instead of magic numbers
  - Single routine to manipulate data vs multiple copies
  - (and more!)



```
// Option A
// -----
void displayInPennies(int priceInCents) {
    int dollars = priceInCents / 100;
    int cents = priceInCents % 100;
    System.out.printf("Your $%d.%02d is %d pennies!", dollars, cents, priceInCents);
}
void displayInNickles(int priceInCents) {
    int dollars = priceInCents / 100;
    int cents = priceInCents % 100;
    int numNickles = priceInCents / 5;
    System.out.printf("Your $%d.%02d is %d nickles!", dollars, cents, numNickles);
}
void displayInDimes(int priceInCents) {
    int dollars = priceInCents / 100;
    int cents = priceInCents % 100;
    int numDimes = priceInCents / 10;
    System.out.printf("Your $%d.%02d is %d nickles!", dollars, cents, numDimes);
}
void displayInQuarters(int priceInCents) {
    int dollars = priceInCents / 100;
    int cents = priceInCents % 100;
    int numQuarters = priceInCents / 25;
    System.out.printf("Your $%d.%02d is %d quarters!", dollars, cents, numQuarters);
}
// ..Client
void clientExplicitFunctions(int priceInCents) {
    displayInPennies(priceInCents);
    displayInNickles(priceInCents);
    displayInDimes(priceInCents);
    displayInQuarters(priceInCents);
}
```

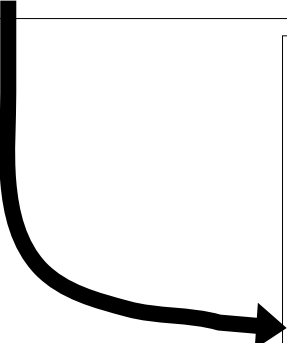
# Parameterize a Function

```
// Option B: Parameterized Function
// -----
void displayInCoins(int priceInCents, int coinAmount, String coinName) {
    int dollars = priceInCents / 100;
    int cents = priceInCents % 100;
    int numQuarters = priceInCents / coinAmount;
    System.out.printf("Your $%d.%02d is %d %s!",
        dollars, cents, numQuarters, coinName);
}

// ..Client
void clientParameterizedFunction(int priceInCents) {
    displayInCoins(priceInCents, 1, "pennies");
    displayInCoins(priceInCents, 5, "nickles");
    displayInCoins(priceInCents, 10, "dimes");
    displayInCoins(priceInCents, 25, "quarters");
}
```

# Encapsulate into object

```
private void clientParameterizedFunction(int priceInCents) {  
    displayInCoins(priceInCents, 1, "pennies");  
    displayInCoins(priceInCents, 5, "nickles");  
    displayInCoins(priceInCents, 10, "dimes");  
    displayInCoins(priceInCents, 25, "quarters");  
}
```



```
enum Coin {  
    PENNIE (1, "pennies"),  
    NICKEL (5, "nickels"),  
    DIME (10, "dime");  
    QUARTER (25, "quarter");  
  
    private int amount;  
    private String name;  
    Coin(int amount, String name) {...}  
    // ...  
}  
  
// ..Client  
void clientCoinsArray(int priceInCents) {  
    for (Coin coin : Coin.values()) {  
        displayInCoins(priceInCents, coin.amount, coin.name);  
    }  
}
```



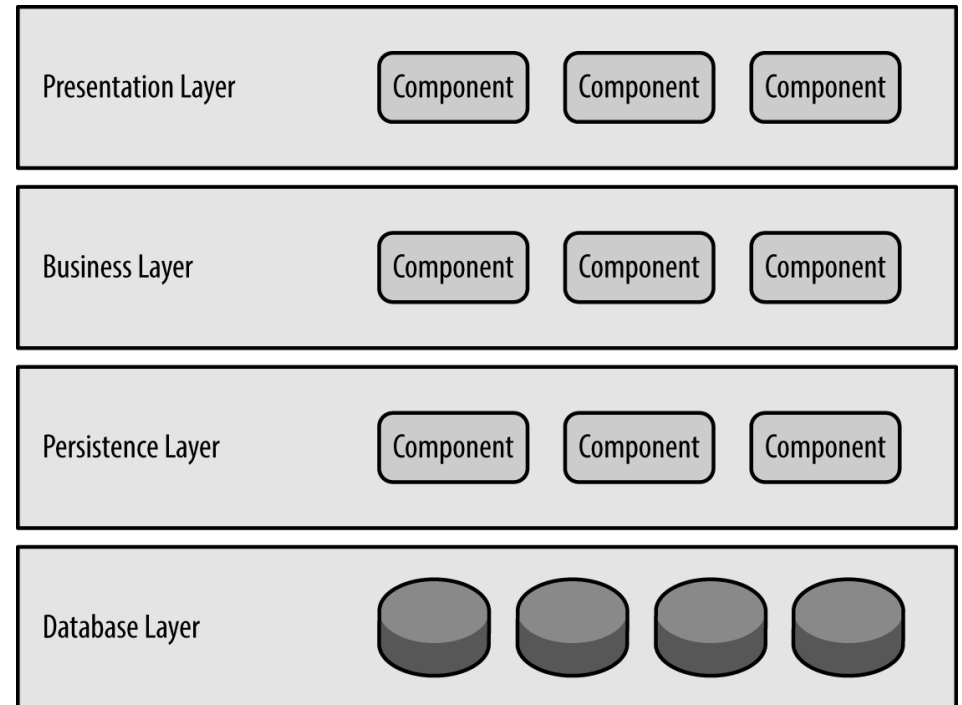
# Quality in Design

# Architecture

- 2 Layer Architecture
  - Separate UI from model package/layer
  - Allows separation of concerns to reduce code complexity
  - Allows decoupling of logic (core) to UI (changes)
- N-Layer architecture (ex: web)
  - UI / Presentation Layer / Front-end
    - May have a UI, model, and communication layer
  - Back-end
    - API endpoint layer
    - Business logic layer
    - Data layer
    - Persistence (database)

# Architecture

- Layers
  - Each horizontal layer has modules with a similar function
  - Provides an abstract view of system
  - Data flows up and down layers (to/from user)
- Layer Separation
  - Layer might (should?) only know about layer above and below:  
decoupling; easy to replace
  - Easy to test a layer in isolation:  
Mock out next lower layer



# OOD Guidelines

- Create fully-formed objects
  - If object has a mandatory `init()` function (or the like), the client must remember to always create then `init()`
  - Called..
- Use encapsulation
  - Make data and methods private when you can
  - Clear interface to reduce cognitive load
  - Prefer fewer setter functions
- Store info in just one place
  - ..  
avoid unnecessary caching

# What's wrong with this class?

```
class ConfigFile {
    File fileName;
    String beforeExtension;
    String extension;

    public void setFileName(File fileName) {
        this.fileName = fileName;
    }

    public void open() {
        if (fileName == null) {
            throw new IllegalStateException();
        }

        // open config file...
    }
}
```

# What changed?

- Is this code better? Why?

```
class ConfigFileSafer {
    private File fileName;

    public ConfigFileSafer(File fileName) {
        if (fileName == null) {
            throw new IllegalArgumentException();
        }
        this.fileName = fileName;
    }

    public void open() {
        // open the config file
    }

    public String getNameBeforeExtension() { ... }
    public String getExtension() { ... }
}
```

# Summary

- Code Sense
  - 1) Being able to recognize poor quality code,
  - 2) Knowing strategies to clean it up,
  - 3) Having the discipline to make all our code clean.
- Clean coding guidelines
  - Intention revealing names
  - Principle of least surprise
  - Don't Repeat Yourself (DRY)
  - Comments can be deodorizers
  - Long methods can be clearer if shorter
  - Create fully formed objects
  - Use encapsulation
- Design: 2 layer, and multi-layer architectures