# Clean code & best practices

•••

Adam Kučera
03/2018

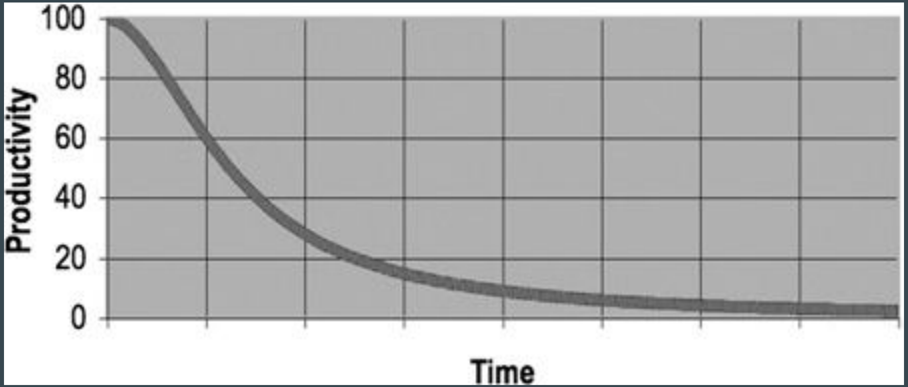
## In this talk...

- Why Clean code matters
- What is (and **is not**) a clean code, shown on simple examples
- Best practices to avoid bad code

WHY IS THIS STRUCTURE HERE?
WHERE COULD THIS BRIDGE POSSIBLY LEAD?
THIS SIGN DOESN'T HELP ME MUCH.
//THIS IS BRIDGE

## Why clean code?

- Bad code makes software maintenance a living hell



- Bad code leads to waste of time and financial losses
- You’re going to read code a lot. Do you remember code you wrote two weeks ago?

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

★

John Woods

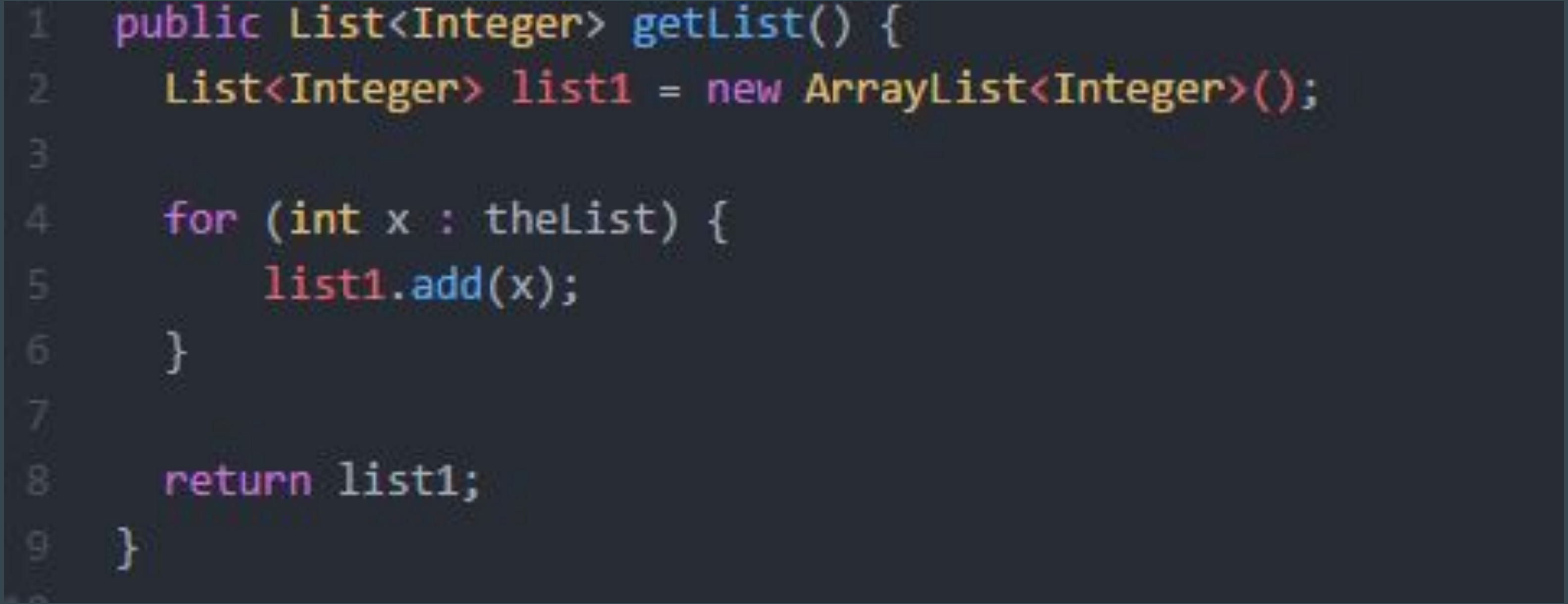
## What is clean code?

*Grady Booch: Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.*

# Few examples...

- Let’s see some examples of bad code and show some of the many clean code principles
- Showing just the **most basic** rules

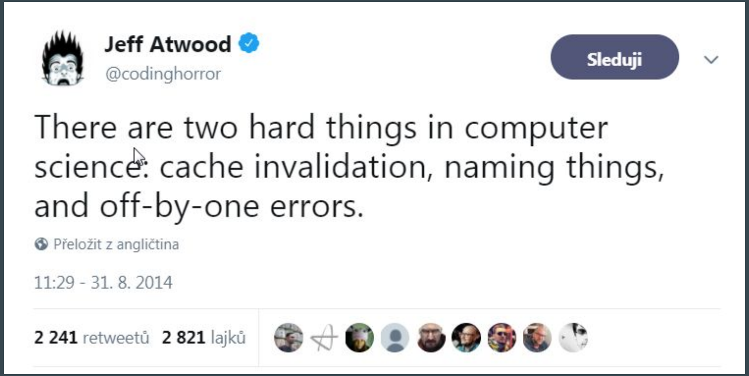
## Case 1



```
public List<Integer> getList() {
  List<Integer> list1 = new ArrayList<Integer>();

  for (int x : theList) {
      list1.add(x);
  }

  return list1;
}
```

# Naming matters!

- Naming is hard
- Names should reveal, what is the intent of variable, function, class...



Jeff Atwood
@codinghorror

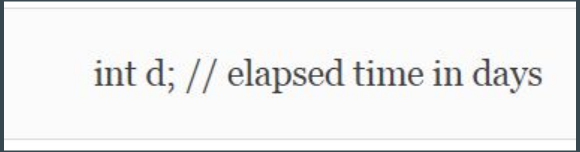
Sleduji

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.

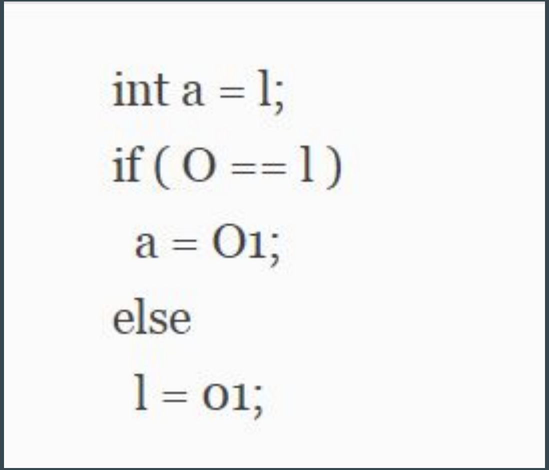
Přeložit z angličtina

11:29 - 31. 8. 2014

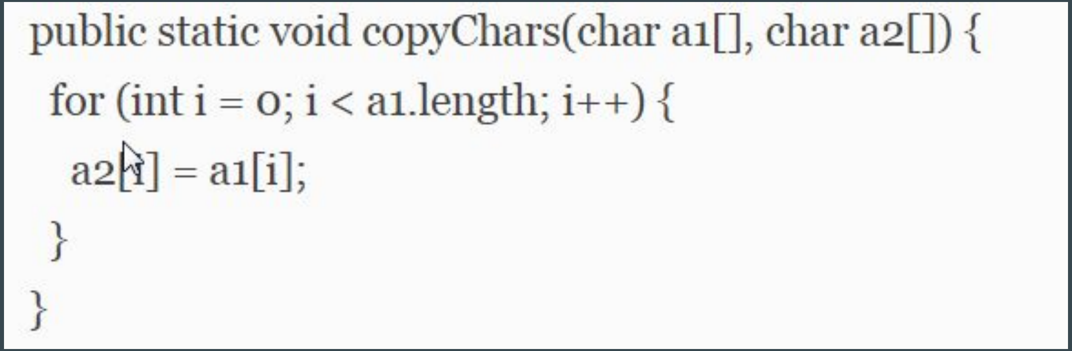
2 241 retweetů 2 821 lajků



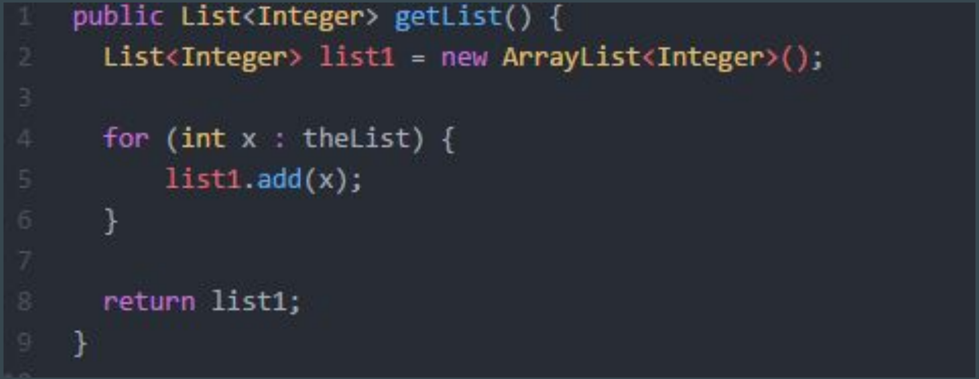
```
int d; // elapsed time in days
```



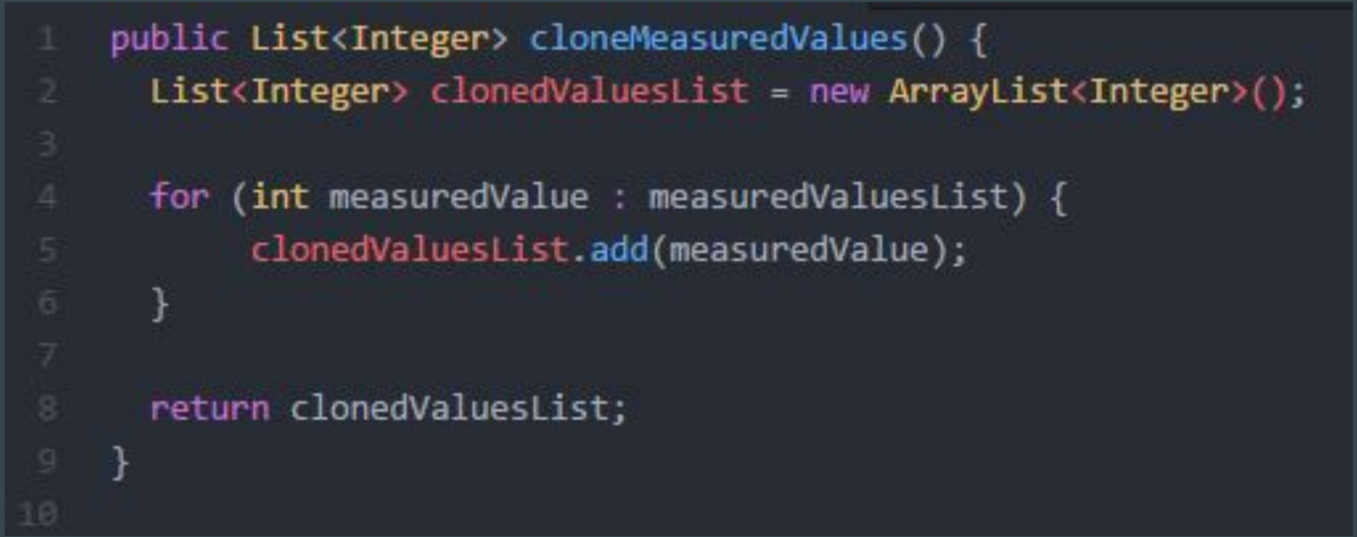
```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```



```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

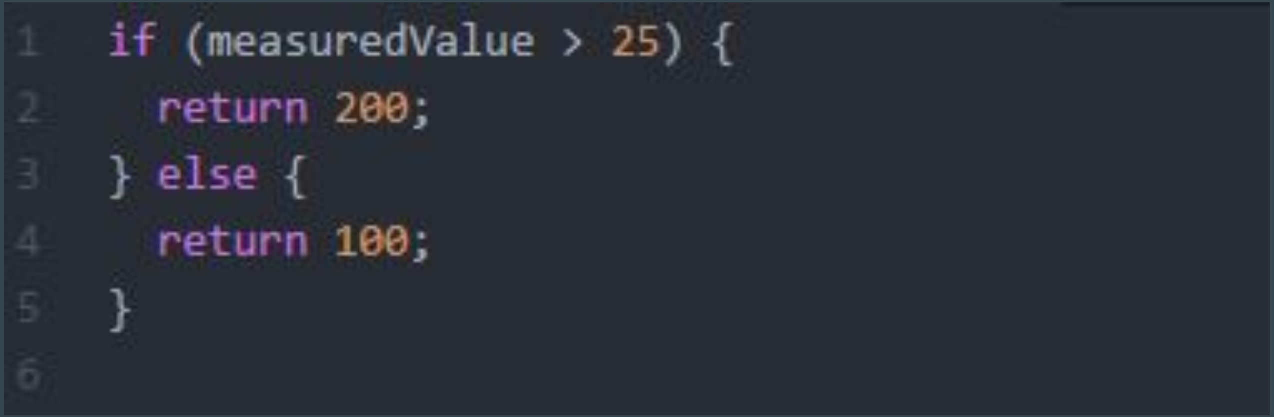


```java
public List<Integer> getList() {
  List<Integer> list1 = new ArrayList<Integer>();

  for (int x : theList) {
      list1.add(x);
  }

  return list1;
}
```



```java
public List<Integer> cloneMeasuredValues() {
  List<Integer> clonedValuesList = new ArrayList<Integer>();

  for (int measuredValue : measuredValuesList) {
        clonedValuesList.add(measuredValue);
  }

  return clonedValuesList;
}
```

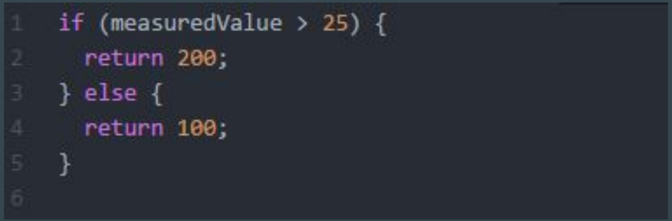
## Case 2



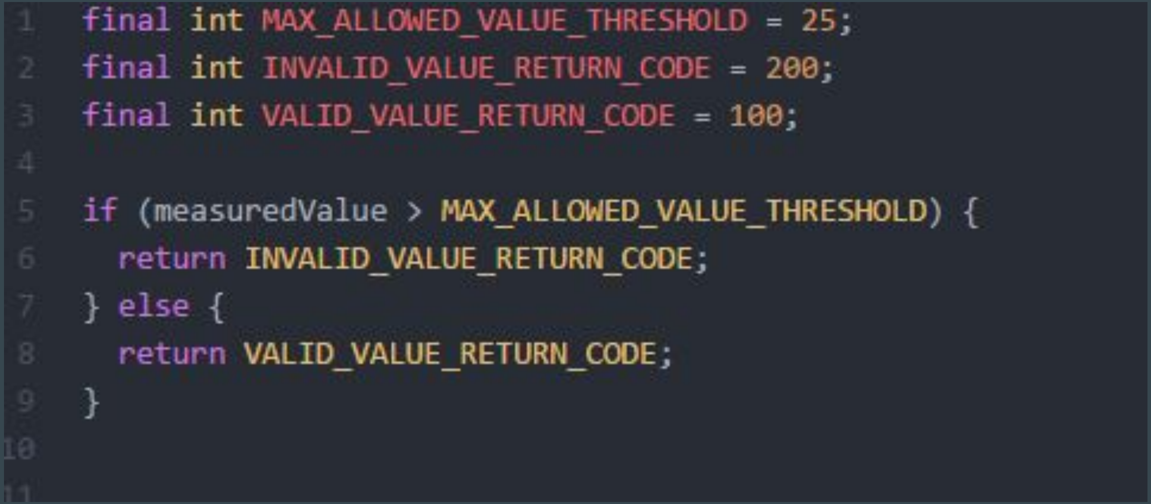
```
if (measuredValue > 25) {
  return 200;
} else {
  return 100;
}
```

## Magic numbers, magic strings

- Constants magically appearing inside your code
- They should be extracted to constants / configuration

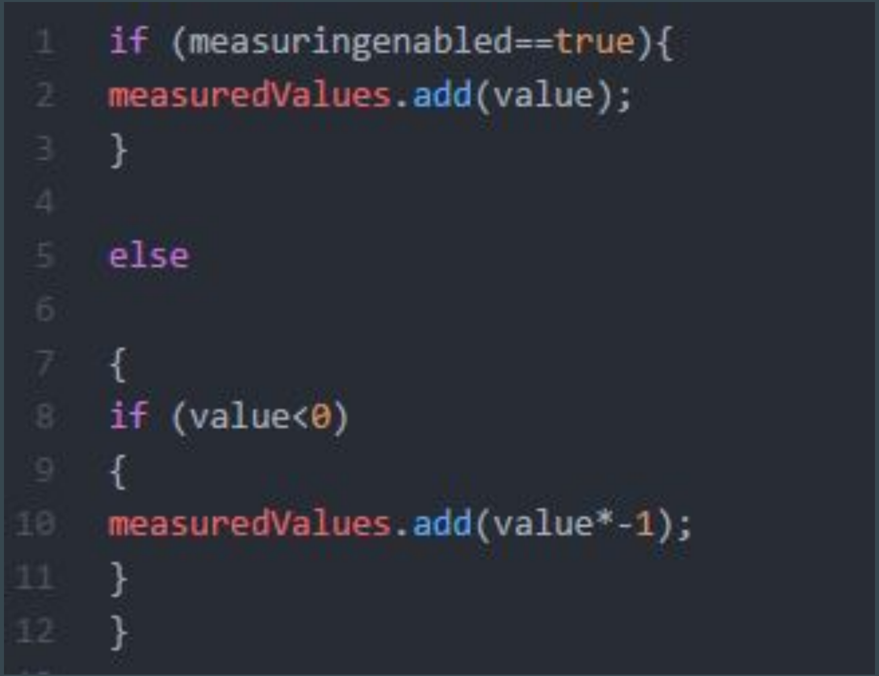


```
if (measuredValue > 25) {
  return 200;
} else {
  return 100;
}
```



```
final int MAX_ALLOWED_VALUE_THRESHOLD = 25;
final int INVALID_VALUE_RETURN_CODE = 200;
final int VALID_VALUE_RETURN_CODE = 100;

if (measuredValue > MAX_ALLOWED_VALUE_THRESHOLD) {
  return INVALID_VALUE_RETURN_CODE;
} else {
  return VALID_VALUE_RETURN_CODE;
}
```

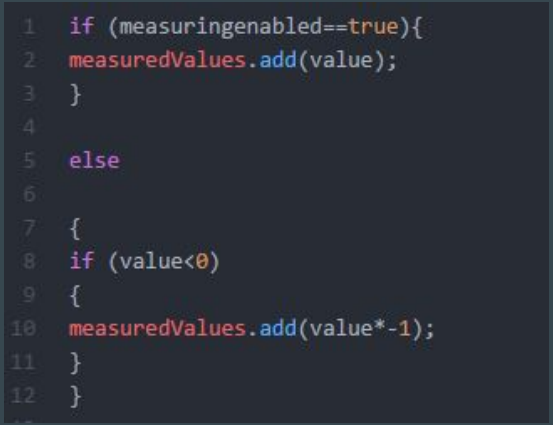
## Case 3



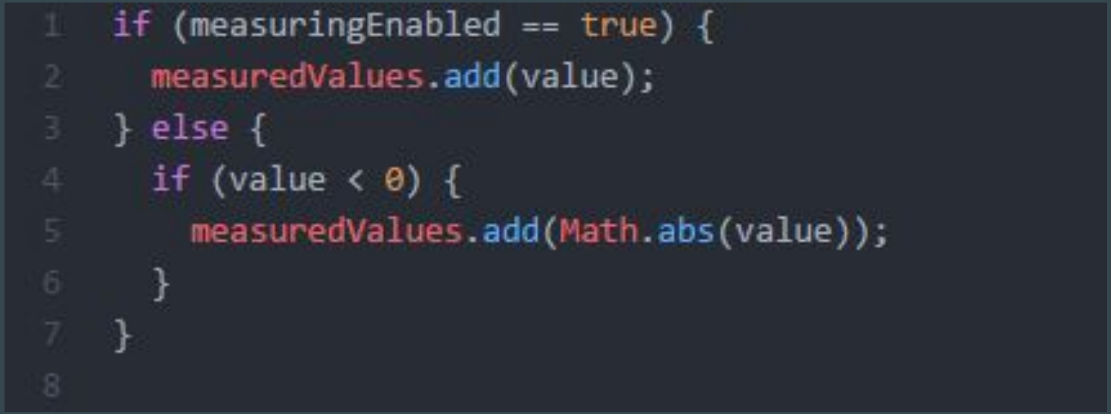
```
if (measuringenabled==true){
measuredValues.add(value);
}

else


{
if (value<0)
{
measuredValues.add(value*-1);
}
}
```

## Structure

- Use same format across the project
- Variable naming
- Use tools

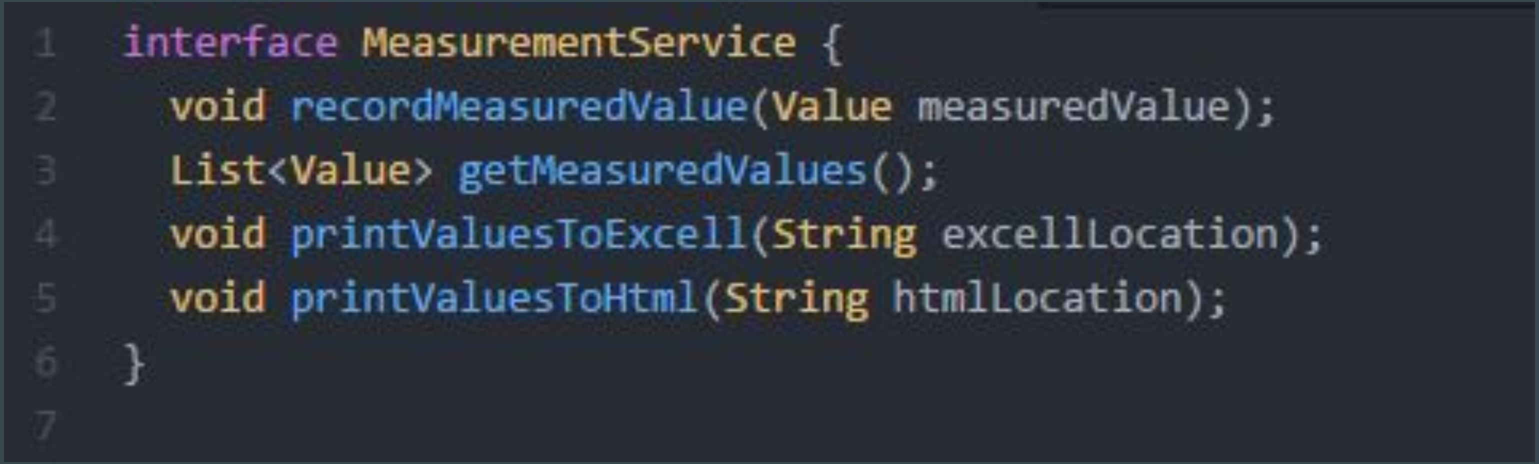


```
if (measuringenabled==true){
measuredValues.add(value);
}

else

{
if (value<0)
{
measuredValues.add(value*-1);
}
}
```



```
if (measuringEnabled == true) {
  measuredValues.add(value);
} else {
  if (value < 0) {
    measuredValues.add(Math.abs(value));
  }
}

```

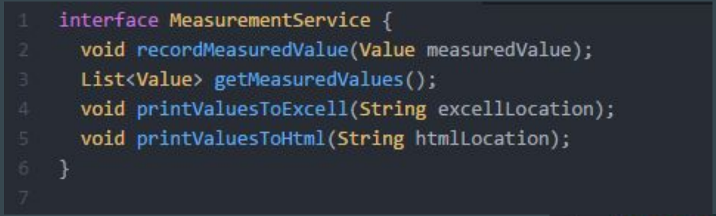
# Case 4



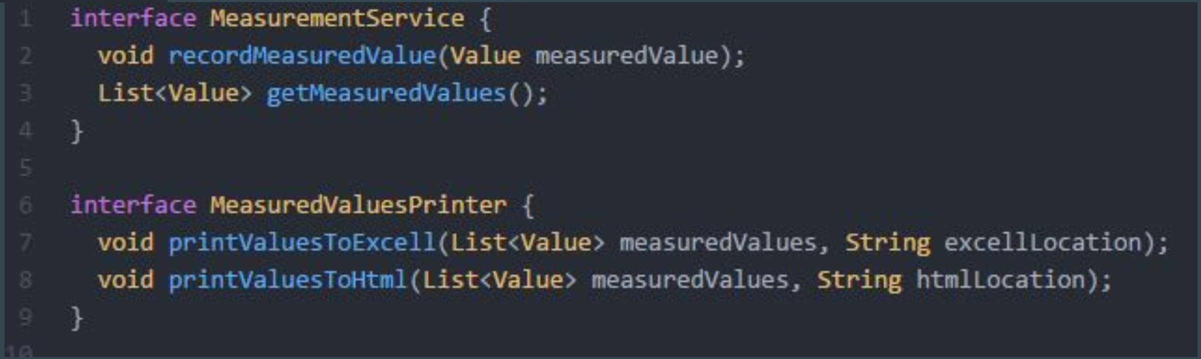
```java
interface MeasurementService {
  void recordMeasuredValue(Value measuredValue);
  List<Value> getMeasuredValues();
  void printValuesToExcell(String excellLocation);
  void printValuesToHtml(String htmlLocation);
}

```

## Single responsibility principle

- Originally OOP concept, but applies everywhere
- **Every class should have one reason change.**
- **Every function should do one thing and it should do it well.**

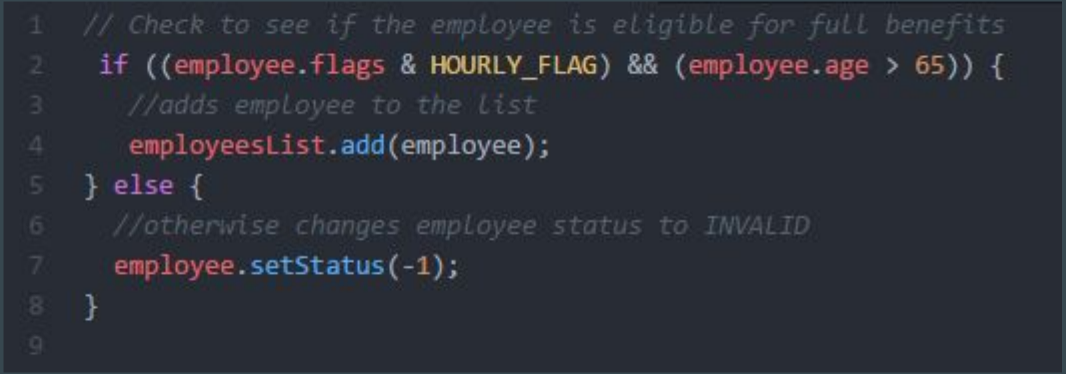


```
interface MeasurementService {
  void recordMeasuredValue(Value measuredValue);
  List<Value> getMeasuredValues();
  void printValuesToExcell(String excellLocation);
  void printValuesToHtml(String htmlLocation);
}
```



```
interface MeasurementService {
  void recordMeasuredValue(Value measuredValue);
  List<Value> getMeasuredValues();
}

interface MeasuredValuesPrinter {
  void printValuesToExcell(List<Value> measuredValues, String excellLocation);
  void printValuesToHtml(List<Value> measuredValues, String htmlLocation);
}
```

# Case 5

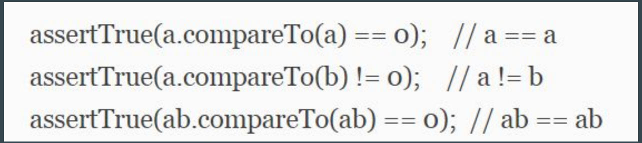

```
// Check to see if the employee is eligible for full benefits
 if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) {
   //adds employee to the list
   employeesList.add(employee);
} else {
  //otherwise changes employee status to INVALID
  employee.setStatus(-1);
}
```

## Comments

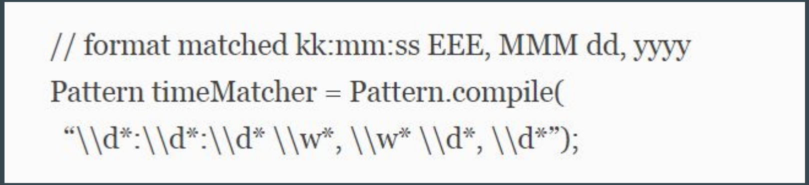
- Good code comments explain why things are done not what is done
- The proper use of comments is to compensate for our *failure* to express ourselves in code

- It’s very hard to keep comments up-to-date
- Document the code by good naming / clean code, not by comments!
- Think before writing a comment.

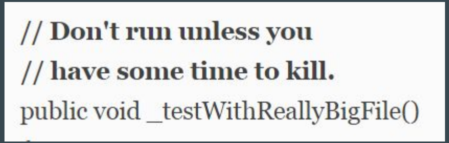
- Some comments can be useful:



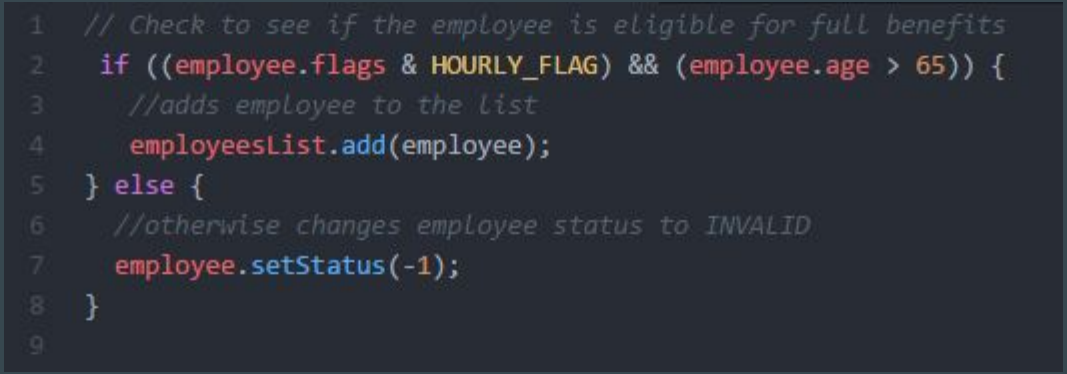
```
assertTrue(a.compareTo(a) == 0);    // a == a
assertTrue(a.compareTo(b) != 0);    // a != b
assertTrue(ab.compareTo(ab) == 0);  // ab == ab
```



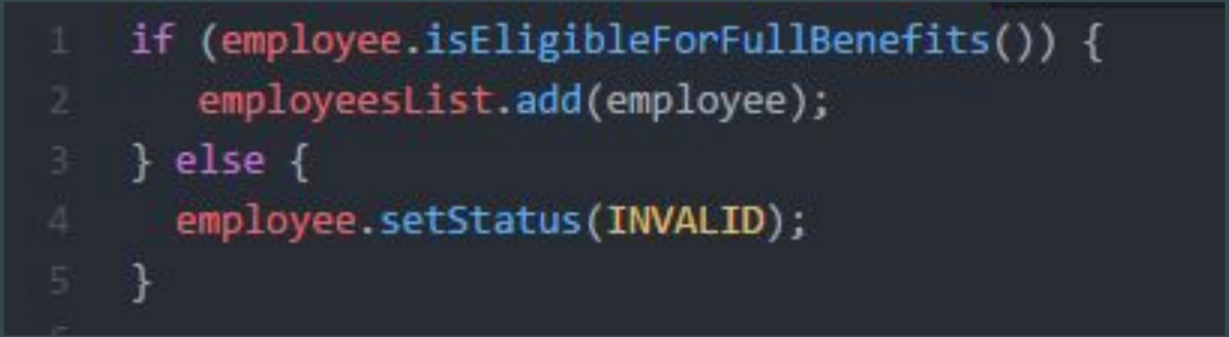
```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
 “\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*”);
```



```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile()
```

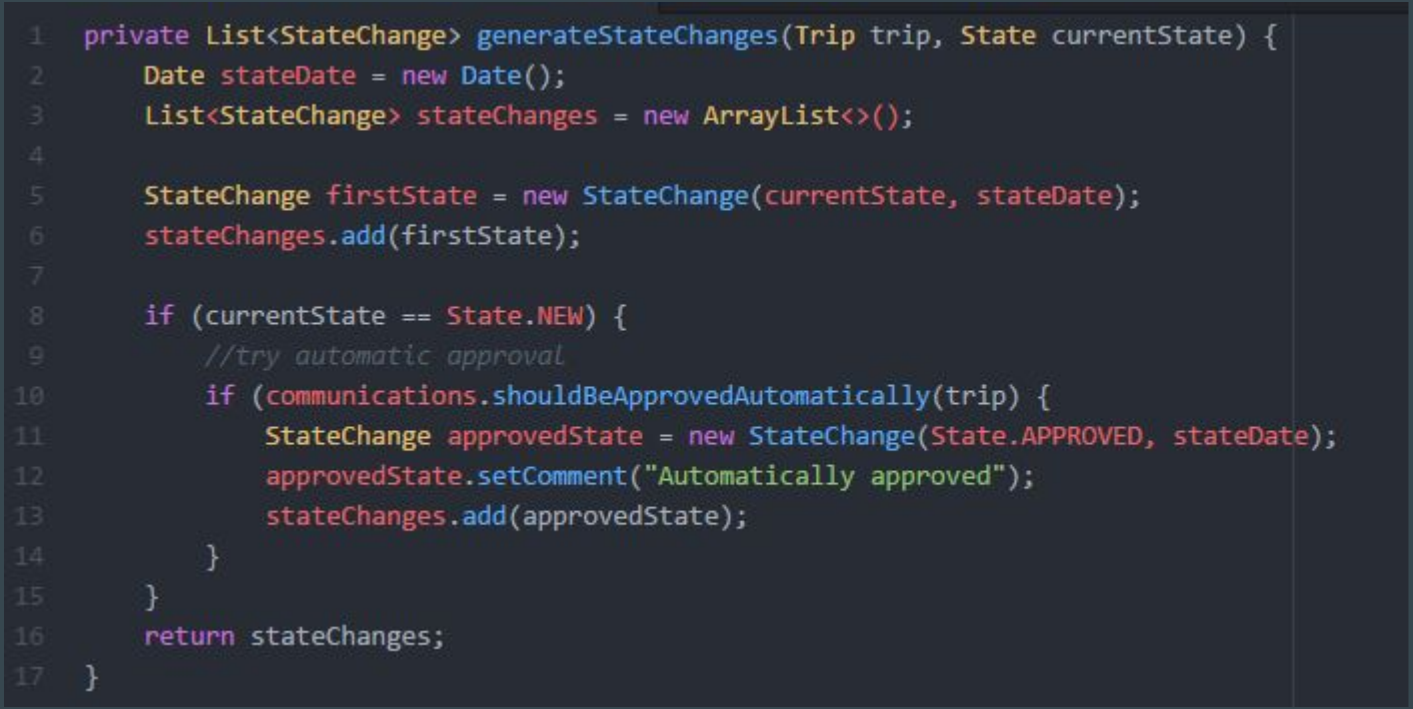


```
// Check to see if the employee is eligible for full benefits
 if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) {
   //adds employee to the list
   employeesList.add(employee);
} else {
  //otherwise changes employee status to INVALID
  employee.setStatus(-1);
}
```



```
if (employee.isEligibleForFullBenefits()) {
   employeesList.add(employee);
} else {
  employee.setStatus(INVALID);
}
```

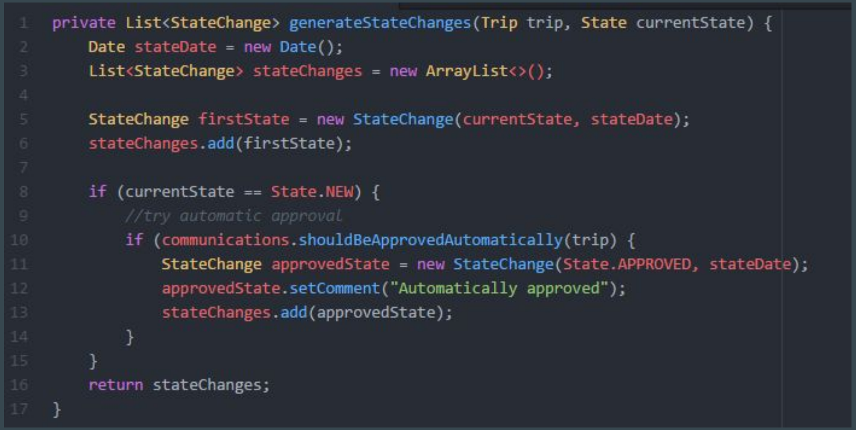
## Case 6



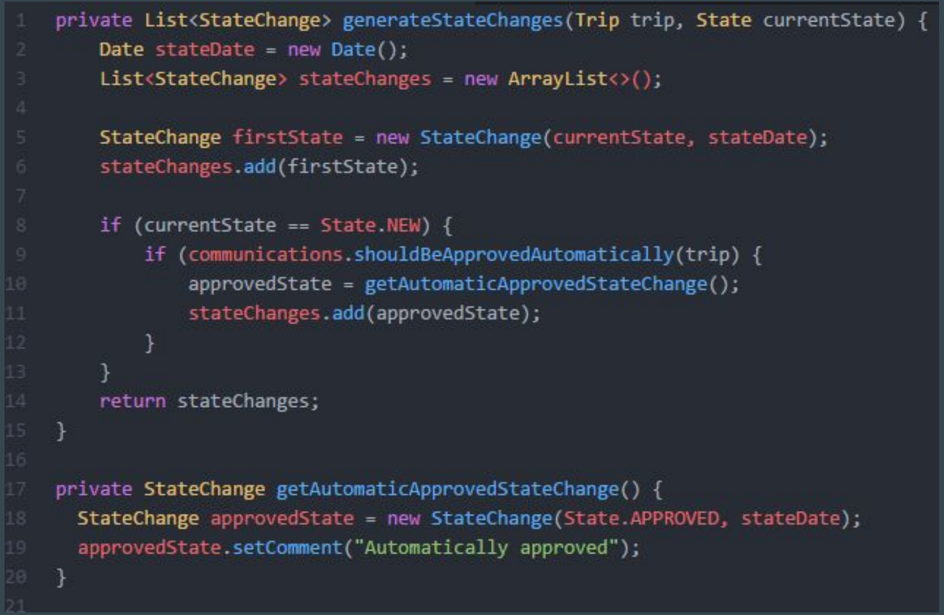
```java
private List<StateChange> generateStateChanges(Trip trip, State currentState) {
    Date stateDate = new Date();
    List<StateChange> stateChanges = new ArrayList<>();

    StateChange firstState = new StateChange(currentState, stateDate);
    stateChanges.add(firstState);

    if (currentState == State.NEW) {
        //try automatic approval
        if (communications.shouldBeApprovedAutomatically(trip) {
            StateChange approvedState = new StateChange(State.APPROVED, stateDate);
            approvedState.setComment("Automatically approved");
            stateChanges.add(approvedState);
        }
    }
    return stateChanges;
}
```

## Long functions

- Try to have your function short
- Extract code to new smaller functions
- Short branches in if-else conditions



```java
private List<StateChange> generateStateChanges(Trip trip, State currentState) {
    Date stateDate = new Date();
    List<StateChange> stateChanges = new ArrayList<>();

    StateChange firstState = new StateChange(currentState, stateDate);
    stateChanges.add(firstState);

    if (currentState == State.NEW) {
        //try automatic approval
        if (communications.shouldBeApprovedAutomatically(trip) {
            StateChange approvedState = new StateChange(State.APPROVED, stateDate);
            approvedState.setComment("Automatically approved");
            stateChanges.add(approvedState);
        }
    }
    return stateChanges;
}
```

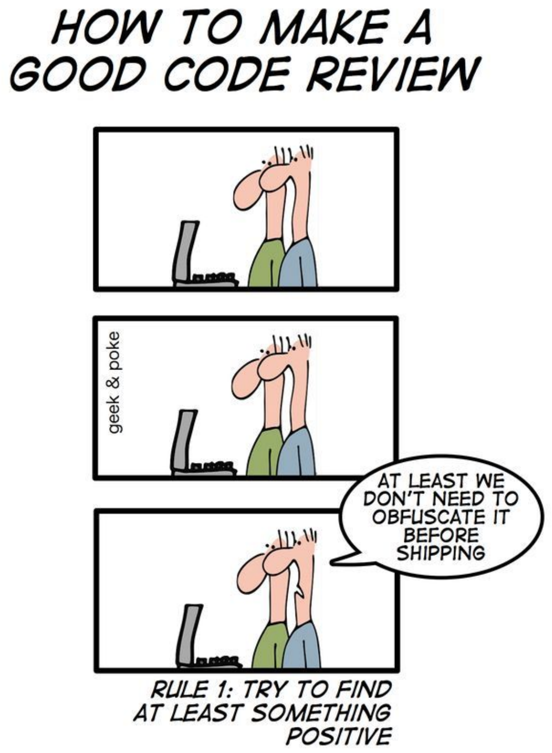


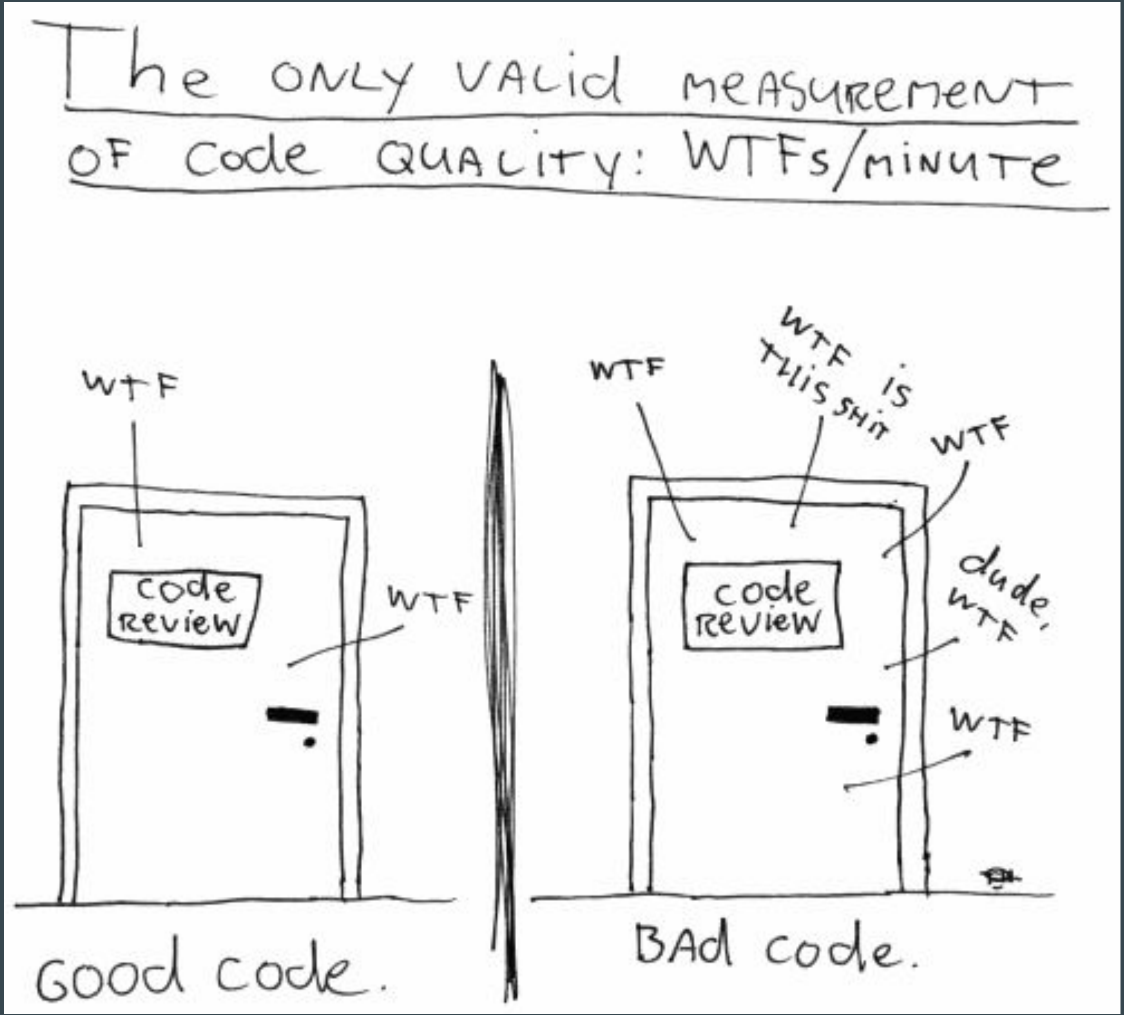
```java
private List<StateChange> generateStateChanges(Trip trip, State currentState) {
    Date stateDate = new Date();
    List<StateChange> stateChanges = new ArrayList<>();

    StateChange firstState = new StateChange(currentState, stateDate);
    stateChanges.add(firstState);

    if (currentState == State.NEW) {
        if (communications.shouldBeApprovedAutomatically(trip) {
            approvedState = getAutomaticApprovedStateChange();
            stateChanges.add(approvedState);
        }
    }
    return stateChanges;
}

private StateChange getAutomaticApprovedStateChange() {
  StateChange approvedState = new StateChange(State.APPROVED, stateDate);
  approvedState.setComment("Automatically approved");
}
```

## Best practices to have clean code

## Code reviews

- Very good practice to keep each others code in a good shape
- There are also best practices during code reviews :)


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE


The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
Bad code.

## Unit tests

- See previous talk :-)

## Boy scout rule and broken windows theory

- Leave the code in better shape than you found it!

- If there are broken windows in your code, there will soon be more and more issues.
- Fix the problems immediately!

## Sources

- Clean code, Robert C. Martin
    - The Bible of software development, definitely find time to read it
- Good code vs Bad code
- 7 reasons clean code matters