



Clean Code by Example

M. Scott Ford
June/2022





Introduction



Write Code for Humans

Not Machines

The primary audience for the code that we write are the people on our team (or the people who will be in the future).

- We *read* code way more often than we *write* code
- Code is a form of documentation



What Does it Mean to Work with Code?

When you are reading or writing code, then you work with code.

- True regardless of the language.
- True for code reviews.
- True when writing pseudo code.
- Your brain works like a computer whenever you are reading code.
 - Source: <https://news.mit.edu/2020/brain-reading-computer-code-1215>



Poll: What About You?

How would you best describe your current role?

- I work with code most of the time.
- I work with code some of the time.
- I rarely work with code.
- I never work with code.



Defining “Clean Code”



Describing “Clean Code”

- Easy to read
- Easy to understand
- Easy to explain
- Easy to modify
- Easy to test



Describing “Messy Code”

- Hard to read
- Hard to understand
- Hard to explain
- Hard to modify
- Hard to test

Poll: “Clean” vs “Messy”

What’s a good description of this block of code?

- It’s clean
- It’s messy
- Some of both
- I’m not really sure

```
unsigned long
elf_hash(const char *name)
{
    unsigned long h, t;
    const unsigned char *s;

    s = (const unsigned char *) name;
    h = t = 0;

    for (; *s != '\0'; h = h & ~t) {
        h = (h << 4) + *s++;
        t = h & 0xF0000000UL;
        if (t)
            h ^= t >> 24;
    }

    return (h);
}
```



Discussion: Reflect on Answers from the Poll

Use the Group Chat to share your thoughts

Share your thoughts about the following:

- Why was there so much (or so little) variation in the answers?
 - Is it because of differences in experience?
 - Is it because of differences in aesthetic preferences?
 - Is it because it's actually hard to say one way or another?
- Anything else interesting that you want to share?



What's Going to be Covered?



A Preview of Today's Topics

Here's what you're going to learn today.

- Clean Names
 - Some rules for naming things cleanly
- Clean Formatting
 - Some rules for laying out your code cleanly
- Clean Logic
 - Some rules for keeping your code's logic clean
- Clean Unit Tests
 - Some rules for writing clean unit tests



A Note About “Rules”

- Use as a starting point
- Have a discussion with your team
- Change and/or break the rules when it makes sense



Structure and Other Notes

- There are some exercises for you to work through
 - Download the PDF from the Resource List
- The code that I'm showing today is available from GitHub
 - Click the GitHub repository link in the Resource List
- There will be time after each “section” to ask questions
 - Feel free to use either the Group Chat or Q&A boxes to ask questions
 - Feel free to answer each other's questions
- We will take breaks after each Q&A session



Q&A

Post your questions in either
the Group Chat or Q&A boxes



Break

5 minutes



Clean Names



Why are clean names important?

Insert subtitle here...

messy.rb

```
def a(b, c)
  b + c
end

puts a(5, 4)
```



Why are clean names important?

Insert subtitle here...

messy.rb

single letter function name
provides no context

```
def a(b, c)
  b + c
end

puts a(5, 4)
```



Why are clean names important?

Insert subtitle here...

messy.rb

single letter function name
provides no context

```
def a(b, c)
  b + c
end

puts a(5, 4)
```

single letter variable names
provides no context



Why are clean names important?

Insert subtitle here...

messy.rb

single letter function name
provides no context

```
def a(b, c)
  b + c
end

puts a(5, 4)
```

single letter variable names
provides no context

no additional context when
the function is called



Why are clean names important?

Insert subtitle here...

messy.rb

```
def a(b, c)
  b + c
end
```

```
puts a(5, 4)
```

single letter function name
provides no context

single letter variable names
provides no context

no additional context when
the function is called

must read function definition
to understand what it does



Why are clean names important?

Insert subtitle here...

messy.rb

```
def a(b, c)
  b + c
end

puts a(5, 4)
```

clean.rb

```
def add(left, right)
  left + right
end

puts add(5, 4)
```



Why are clean names important?

Insert subtitle here...

messy.rb

```
def a(b, c)
  b + c
end

puts a(5, 4)
```

descriptive names in definitions give a hint about usage

clean.rb

```
def add(left, right)
  left + right
end

puts add(5, 4)
```




Why are clean names important?

Insert subtitle here...

messy.rb

```
def a(b, c)
  b + c
end

puts a(5, 4)
```

clean.rb

```
def add(left, right)
  left + right
end

puts add(5, 4)
```

descriptive names in definitions give a hint about usage

usage in context gives clue about behavior without having to read source



Discussion

Answer the following question in the Group Chat box

When have you been frustrated by a name that was unclear?



Prefer clarity over brevity

Insert subtitle here...

loops-messy.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];  
  
for (let i = 0; i < fruits.length; i++) {  
  const f = fruits[i];  
  console.log(f);  
}
```



Prefer clarity over brevity

Insert subtitle here...

loops-messy.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];  
  
for (let i = 0; i < fruits.length; i++) {  
  const f = fruits[i];  
  console.log(f);  
}
```

single letter loop counter does not
give hint about how it's used



Prefer clarity over brevity

Insert subtitle here...

loops-messy.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];  
  
for (let i = 0; i < fruits.length; i++) {  
  const f = fruits[i];  
  console.log(f);  
}
```

single letter loop counter does not
give hint about how it's used

single letter variable provides
no context about meaning



Prefer clarity over brevity

Insert subtitle here...

loops-clean.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];  
  
for (let index = 0; index < fruits.length; index++) {  
  const fruit = fruits[index];  
  console.log(fruit);  
}
```




Prefer clarity over brevity

Insert subtitle here...

loops-clean.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];  
  
for (let index = 0; index < fruits.length; index++) {  
  const fruit = fruits[index];  
  console.log(fruit);  
}
```

lets us know we're indexing into an array with this variable





Prefer clarity over brevity

Insert subtitle here...

loops-clean.js

```
const fruits = [  
  'apple',  
  'banana',  
  'cherry',  
  'date'  
];
```

lets us know we're indexing into an array with this variable

```
for (let index = 0; index < fruits.length; index++) {  
  const fruit = fruits[index];  
  console.log(fruit);  
}
```

we don't need to read the assignment to guess at what is being logged here



Exercise

Use the exercise PDF or a scrap sheet of paper to do the following:

1. Think about the acronyms and abbreviations that are commonly used on your project. Now write down as many as you can in 3 minutes.
1. Put a mark next to each of the acronyms and abbreviations that you listed if you are pretty sure that everyone on your team knows the true meaning of the acronym or abbreviation.
1. Compute the percentage of acronyms and abbreviations that everyone on your team knows. Write the number down; we'll use it later.



Acronyms and Abbreviations

Let's look at an example project to see the impact of acronyms and abbreviations in action.

- `code/clean-names/02-acronyms-and-abbreviations/Messy/Messy.sln`
- `code/clean-names/02-acronyms-and-abbreviations/Clean/Clean.sln`



Poll: Share the results of the previous exercise

What percentage of the acronyms and abbreviations do you think everyone on your team understands the meaning of?

- 80% or higher
- 60% – 79%
- 40% – 59%
- 20% – 39%
- Less than 20%



Class and Type Names

Prefer nouns for class names

class_names.py

```
# clean - nouns and noun forms  
class Performer: pass  
class Performance: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid verb forms  
class Perform: pass  
class Performed: pass  
class Performing: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid verb forms
class Perform: pass
class Performed: pass
class Performing: pass

# cleaner - use adjective prefixes to convey time
class ActivePerformance: pass
class PastPerformer: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid adjectives
class Huge: pass
class Small: pass
class Fast: pass
class Slow: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid adjectives
class Huge: pass
class Small: pass
class Fast: pass
class Slow: pass

# cleaner - adjective as prefix to a noun
class SmallPerformance: pass
class FastPerformer: pass
```




Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid vague prefixes  
class MyPerformer: pass  
class APerformer: pass  
class ThePerformer: pass  
class ThisPerformer: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid single letter class names  
class P: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid single letter class names
```

```
class P: pass
```

```
# messy - avoid single letter prefixes
```

```
class CPerformer: pass
```

```
class TPerforer: pass
```



Class and Type Names

Prefer nouns for class names

exception.cs

```
// exception: languages with templates or type parameters
// T is often used for a single type parameter
class CustomList<T> {}
```



Class and Type Names

Prefer nouns for class names

exception.cs

```
// exception: languages with templates or type parameters
// T is often used for a single type parameter
class CustomList<T> {}

// unique single letters are often used
class Pair<K, V> {}
```



Class and Type Names

Prefer nouns for class names

exception.cs

```
// exception: languages with templates or type parameters
// T is often used for a single type parameter
class CustomList<T> {}

// unique single letters are often used
class Pair<K, V> {}

// T as a prefix with a description is best
class Function<TReturn, TValue> {}
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid all capital acronyms  
class HTTPAPIPerformer: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid all capital acronyms
```

```
class HTTPAPIPerformer: pass
```

```
# cleaner - easier to see boundary between acronyms
```

```
class HttpApiPerformer: pass
```




Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - avoid all capital acronyms
class HTTPAPIPerformer: pass

# cleaner - easier to see boundary between acronyms
class HttpApiPerformer: pass

# messy avoid abbreviations
class Perf: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - plural used on normal class  
class Performers: pass
```



Class and Type Names

Prefer nouns for class names

class_names.py

```
# messy - plural used on normal class
class Performers: pass

# cleaner - plural used for collection class
class Performers:
    def __getitem__(self, key): pass
    def __iter__(self): pass
```



Exercise

View the Exercise PDF: Practice Clean Class and Type Names

1. Identify the class and type names in the following list that violate one of the rules that were presented.



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# prefer present tense verbs for method names
def perform; end
def open; end
def close; end
def validate; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# avoid gerunds
def performing; end
def opening; end
def closing; end
def validating; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# and avoid past tense verb forms
def performed; end
def opened; end
def closed; end
def validated; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# and avoid past tense verb forms
def performed; end
def opened; end
def closed; end
def validated; end
```




Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# better versions start with a verb
# prefix gerunds with `is_`
def is_performing; end
def is_opening; end
def is_closing; end
def is_validating; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# prefix past tense verbs with `has_`  
def has_performed; end  
def has_opened; end  
def has_closed; end  
def has_validated; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# in ruby these forms are typically suffixed with `?`  
def performing?; end  
def performed?; end  
def opening?; end  
def opened?; end  
def closing?; end  
def closed?; end  
def validating?; end  
def validated?; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# also in ruby and many other scripting languages
# accessor methods have noun names
def name; end
def place; end
def address; end
```



Method and Function Names

Prefer present tense verbs for method names

methods_names.rb

```
# some language communities prefer  
# prefixing accessor methods with `get_`  
def get_name; end  
def get_place; end  
def get_address; end
```



Discussion: Reflect on the following question

Use the Group Chat to share your thoughts

Share your thoughts about the following:

- When might it make sense to break the method and function naming rules?



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// prefer singular nouns for primitive types and object instances  
String name = "Alma";  
Performer performer = new Performer(name);
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// prefer singular nouns for primitive types and object instances
String name = "Alma";
Performer performer = new Performer(name);

// use plural nouns for arrays and other collections
String names[] = {"Alex", "Ali", "Aesop"};
List<Integer> years = Arrays.asList(1980, 1999, 2003, 2010);
```




Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid verbs for variables that store primitive types  
int perform = 12;  
boolean create = false;
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid verbs for variables that store primitive types
int perform = 12;
boolean create = false;

// instead make them nouns
int performanceCode = 12;
boolean creationEnabled = false;
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// this is true even when the closure is just being passed as a
// parameter to a method or constructor
BiFunction<Integer, Integer, Integer> adder = (left, right) -> {
    return left + right;
};
Performer addedPerformer = new Performer("The Adder", adder);
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid single letter variable names  
int t = 12;  
int i = 8;
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid single letter variable names
int t = 12;
int i = 8;

// instead spell them out
int testCode = 12;
int index = 8;
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid confusing acronyms and abbreviations  
String dbsqlSelAllNames = "select * from names;";
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid confusing acronyms and abbreviations
String dbSqlSelAllNames = "select * from names;";

// instead separate acronyms and spell out abbreviations
String dbSqlSelectAllNames = "select * from names;";
```



Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid complicated prefixes such as Hungarian notation  
final String f_strFirstName = "Jefferson";  
String firstName = "Jefferson";
```




Method and Function Names

Prefer nouns for variable names

VariablesNames/src/com/example/Main.java

```
// avoid complicated prefixes
final String f_strFirstName = "Jefferson";
String firstName = "Jefferson";

// avoid using the type name as a suffix
String lastNameString = "Amaya";
String lastName = "Amaya";
```



Poll: Think about variable names in your project

How often do you change variable names when you see one in your project that breaks your project's naming rules?

- Always
- Often
- Sometimes
- Rarely
- Never



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// prefer singular nouns for a single value
function add(left, right) {
    return left + right;
}
add(4, 6);

function negate(value) {
    return -value;
}
negate(-10);
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// use plural nouns for arrays and other collections
function sum(values) {
  let result = 0;
  for (let value of values) {
    result += value;
  }
  return result;
}
sum([1, 2, 3, 4]);
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// use plural nouns for arrays and other collections
function sum(values) {
  let result = 0;
  for (let value of values) {
    result += value;
  }
  return result;
}
sum([1, 2, 3, 4]);
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// the -er suffix can help communicate that a parameter contains a closure
function compare(left, right, comparer) {
  return comparer(left, right);
}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// the -er suffix can help communicate that a parameter contains a closure
function compare(left, right, comparer) {
  return comparer(left, right);
}

// avoid single letter variable names even when they are part of a closure
compare(1, 11, (l, r) => { return l - r; });
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// the -er suffix can help communicate that a parameter contains a closure
function compare(left, right, comparer) {
  return comparer(left, right);
}

// avoid single letter variable names even when they are part of a closure
compare(1, 11, (l, r) => { return l - r; });

// instead spell them out
compare(1, 11, (left, right) => { return left - right; });
```




Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid abbreviations  
function squareRoot(val) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid abbreviations
```

```
function squareRoot(val) {}
```

```
// spell them out instead
```

```
function squareRoot(value) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid starting parameters with a capital letter
// doing so makes them look like they are class or type names
function random(SeedGenerator) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid starting parameters with a capital letter
// doing so makes them look like they are class or type names
function random(SeedGenerator) {}

// use a lower case letter instead
function random(seedGenerator) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid confusing compound acronyms  
function postResult(HTTPAPI, value) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid confusing compound acronyms
function postResult(HTTPAPI, value) {}

// clearly separate acronyms
function postResult(httpApi, value) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid complicated prefixes  
function persistName(sName) {}  
function persistPerson(oPerson) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid complicated prefixes
function persistName(sName) {}
function persistPerson(oPerson) {}

// using simple nouns instead
function persistName(name) { }
function persistPerson(person) { }
```




Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid using the type name as a suffix  
function persistName(nameString) {}  
function persistPerson(personObject) {}
```



Parameter Names

Prefer nouns for parameter names

parameter_names.js

```
// avoid using the type name as a suffix
function persistName(nameString) {}
function persistPerson(personObject) {}

// instead drop the suffixes
function persistName(name) {}
function persistPerson(person) {}
```



Exercise

View the Exercise PDF: Practice Clean Parameter Names

1. Open `messy_parameters.js` in a text editor and correct all of the messy parameter names that you see.



Constant Names

Basic Rules

Basic Rules:

- Use an uppercase first letter
- Singular nouns for primitive value or object types
- Plural nouns for collections of values
- Avoid single letters and abbreviations
- Ensure clear separation between acronyms



Constant Names

Some conventions vary by language

constants.rb

```
# typically formatted in all capital letters separated by underscores
```

```
DIRECTORY_NAME = "/code"
```

```
# this convention is also used by Java, JavaScript, Kotlin, PHP, Python,  
# and Rust
```

```
FILE_NAME = "/code/sample.rb"
```



Constant Names

Some conventions vary by language

constants.cs

```
// typically formatted the same as class names
const string TableName = "examples";

// this convention is also used by Go and Scala
const string StatusColumnName = "status";

// some C++ style guides add a prefix `k` to these rules
const float kDaysPerYear = 365.25;
```



Constant Names

Enumerations

enumerations.py

```
from enum import Enum

# singular nouns for the containing type
class Color(Enum):
    # and singular nouns for the enumeration members
    RED = "#FF0000"
    YELLOW = "#FFFF00"
    GREEN = "#008000"
    LIME = "#00FF00"
    PURPLE = "#800080"
```



Constant Names

Enumerations

enumerations.py

```
from enum import Enum

# singular nouns for the containing type
class Color(Enum):
    # and singular nouns for the enumeration members
    RED = "#FF0000"
    YELLOW = "#FFFF00"
    GREEN = "#008000"
    LIME = "#00FF00"
    PURPLE = "#800080"
```




Constant Names

Enumerations

enumerations.py

```
from enum import Enum

# avoid using plural nouns for the containing type
class Statuses(Enum):
    SUBMITTED = 1
    STARTED = 2
    SUCCEEDED = 3
    FAILED = 4
```



Constant Names

Enumerations

enumerations.py

```
from enum import Enum

# avoid logical mismatches between the names used
# by the containing type and the enumeration members
class Activity(Enum):
    # colors aren't usually considered activities
    RED = 1
    YELLOW = 2
    GREEN = 3
```



Enumeration Names

Some conventions vary by language

Language conventions differ on how the containing type and enumeration members should be formatted.

- C# and Rust - Mixed case is used for both outer and inner names
 - `InputEvent::KeyUp`
- Java and Python 3.4 and later - Mixed case is used for outer name and all capital letters is used for inner name
 - `Day.SUNDAY`



Enumeration Names

Some conventions vary by language

For JavaScript, Ruby, Go, and Python prior to 3.4:

- Enumerations are not part of the language
- Emulated with using other types and constants.
- Follow the same rules for constants in those languages



Discussion: Consider the clean naming rules for Variables, Parameters, and Constants

Use the Group Chat to share your thoughts

Share your thoughts about the following:

- Why is it worth having different rules for variables, parameters, and constants?



Q&A

Post your questions in either
the Group Chat or Q&A boxes



Break

5 minutes



Clean Formatting



Why Clean Formatting Matters

Consider your audience

The primary audience of the code you write is other humans.

- Most compilers don't care about consistent formatting.



Why Clean Formatting Matters

Formatting affects readability

How you format your code impacts its readability

- In many languages, a large program can be formatted as a single line of text.
- It is possible to craft code that is difficult to read and understand



Poll: How much do you value consistent formatting?



When reading code that you didn't write, how much do you value consistent formatting?

- 5 - I think it's extremely important
- 4
- 3
- 2
- 1 - I don't value it at all



Indentation and Readability

Some basic rules to follow

- Be consistent
- Mixing and matching styles will cause confusion
- Pick tabs or spaces – never both



Indentation and Readability

Example

Let's look at an example project to see the impact of mixing tabs and spaces

- `code/clean-formatting/01-indentation/quicksort_tabs_and_spaces_mixed.js`



Indentation and Readability

How many spaces is enough?

How many spaces is enough?

- Either 2 or 4 spaces is a good guideline
- Pick a number and be consistent
- Learn how to configure your text editor to help you stay consistent



Indentation and Readability

Grouping blocks of code

Guideline for grouping blocks of code

- Paragraph-like chunks
- Use the “squint test”
- This guideline applies to all code structures



Brackets and Readability

Where should begin and end brackets be placed?

brackets.cs

```
// Common C# convention
if (value)
{
    // ...
}
else
{
    // ...
}
```

brackets.java

```
// Common Java convention
if (value) {
    // ...
} else {
    // ...
}
```




Brackets and Readability

Where should begin and end brackets be placed?

Each language community has different

- Each language community has different rules
- Some language communities have multiple competing styles
- All choices and styles are valid as long as you're consistent



Exercise

Practice cleaning up messy indentation and bracket placement

1. Open `exercises/clean-formatting/MessyIndentsAndBrackets/PascalsTriangle.cs` in a text editor and correct all of the indentation and bracket placement problems that you find.



Line Wrapping

Rules for dealing with long statements

- Choose a maximum line length
 - 80 characters is traditional
 - Many teams today choose 120 or more characters
- There are different style choices for your team to make
- All choices and styles are valid as long as you're consistent



Line Wrapping

Rules for dealing with long statements

- Choose a maximum line length
 - 80 characters is traditional
 - Many teams today choose 120 or more characters
- There are different style choices for your team to make
- All choices and styles are valid as long as you're consistent



Line Wrapping

Where should period characters appear when wrapping long lines?

dots_first.rb

```
"value"  
  .capitalize  
  .rjust(50)  
  .length
```

dots_last.rb

```
"value".  
  capitalize.  
  rjust(50).  
  length
```



Line Wrapping

Where should closing parenthesis characters appear when wrapping long lines?

parenthesis_final_line.js

```
console.log(  
  Math.max(  
    1,  
    2,  
    3,  
    4,  
    5));
```

parenthesis_own_line.js

```
console.log(  
  Math.max(  
    1,  
    2,  
    3,  
    4,  
    5  
  )  
);
```



Discussion

Answer the following question in the Group Chat box

What maximum line length do you prefer and why?



Whitespace

Inserting separation between statements and operators

- Vertical whitespace
 - Improves readability from top to bottom
 - Groups related chunks of code
- Horizontal Whitespace
 - Improves readability from right to left
 - Insert a space after every comma
 - Insert a space before and after all boolean operators such as `+`, `-`, `/`, `*`



Whitespace

Example

Let's look at an example file to see the impact of compressed whitespace and how to remedy it

- `code/clean-formatting/04-whitespace/gaussian_filter.py`



Q&A

Post your questions in either
the Group Chat or Q&A boxes



Break

5 minutes



Clean Logic



Magic Numbers and Constants

Magic Numbers:

- Literal value with no clear meaning

Named Constants:

- Special kind of variable that is not allowed to change
- Provides an opportunity to give context to these values

magic_numbers.rb

```
# what does this number mean?  
puts 1048576
```

```
# constant name provides context  
SIZE_OF_MEGABYTE = 1048576  
puts SIZE_OF_MEGABYTE
```



Magic Numbers and Constants

Simple Example: Template String

mustache.mjs

```
var business = {  
  name: "LinkedIn",  
  url: "https://linkedin.com"  
};  
  
var html = Mustache.render(  
  '<h1><a href="{{url}}">{{name}}</a></h1>',  
  business  
);
```



Magic Numbers and Constants

More Complex Example

Let's look at an example project to see the impact of using magic numbers

- `code/clean-logic/01-magic-numbers-and-constants/exif/jpeg.rb`
- We're going to also practice refactoring to make the meaning of these numbers more clear



Poll: Magic Numbers

How often do you replace magic numbers with named constants?

- Always
- Often
- Sometimes
- Rarely
- Never



Parameter Lists

Ways method definitions provide context:

- Method name
- Parameter names

The longer a parameter list is, the larger the negative impact on code readability.

parameter_list_examples.rb

```
function negate(value) {  
  return -value;  
}  
  
// methods with single parameters  
// are easiest to read  
console.log(  
  `Inverse of 1: ${negate(1)}`  
);
```



Parameter Lists

Method with two parameters

parameter_list_examples.js

```
function collectBook(title, author) {  
  console.log(  
    `Recording that you own '${title}' by ${author}`  
  );  
}  
  
// this method makes it hard to tell what each parameter is for  
collectBook('Anna Karenina', 'Leo Tolstoy');  
collectBook('Oliver Twist', 'Charles Dickens');
```



Parameter Lists

Method name can hint at purpose of first parameter

parameter_list_examples.js

```
function addNameForAge(age, name) {  
  console.log(  
    `Adding ${name} to the ${age} year old group`  
  );  
}
```

```
// here the final word in the method name gives a  
// hint at the purpose of the first parameter
```

```
addNameForAge(10, 'Kenton');  
addNameForAge(12, 'Aimee');
```



Parameter Lists

More Complex Example

Let's look at an example project to see the impact of using long parameter lists

- `code/clean-logic/02-parameters/parameters.js`
- We're going to also practice refactoring to clarify the purpose of the parameters



Discussion

Answer the following question in the Group Chat box

How many items is too many in a parameter list?



Complex Boolean Logic

- Encodes important details and business rules
- Can become complex and difficult to understand
- The meaning is often unclear



Predicate Methods

- Encapsulates boolean logic
- Returns true or false
- Method name is used to convey meaning



Predicate Methods

Example

Let's look at an example project to explore how to use predicate methods

- `code/clean-logic/03-predicate-methods/PredicateMethods/PredicateMethods.csproj`



Exercise

Practice working with predicate methods

1. Open `exercises/clean-logic/practice-predicate-methods/src/index.js` in a text editor and convert all of the complex if statements to use predicate methods.



Utilizing Loops

Three or more? Use a for!

– *Charles Petzold & Don Roberts*

The Rule of Three:

- Don't repeat yourself more than once



Utilizing Loops

A simple comparison

example_without_loops.py

```
print('')  
print('')  
print('')
```

example_with_loops.py

```
for _ in range(3):  
    print('')
```



Utilizing Loops

Example

Let's look at an example to explore how to use loops to clean up repetition

- `code/clean-logic/04-utilizing-loops/loops.py`



Q&A

Post your questions in either
the Group Chat or Q&A boxes



Break

5 minutes



Clean Unit Tests



What's a “unit” test?

The phrase “unit test” means different things to different people.

- Some use “unit test” to refer to any test that's written with a unit testing tool, such as JUnit (Java), NUnit (C#), Jest (JavaScript), PyUnit (Python), Test::Unit (Ruby)
- In this course, a “unit test” evaluates a component in complete isolation from all others



Attributes of a Clean Unit Test

A clean unit test has the following attributes:

- Fast – each test runs in less than a second, and the full test suite runs in a few minutes at most
- Focused – each unit only has one reason to fail
- Independent – each test can be run by itself and the full suite can be run in any order
- Tidy – avoids needless repetition



Discussion

Answer the following question in the Group Chat box

How do you define a “unit” on your project?



Keep your tests fast

External dependencies negatively affect the performance of your application's test suite.

- Avoid writing tests that interact with database engines
 - Move these interactions to integration tests and limit their number
- Avoid writing tests that make HTTP connections to remote servers
 - Move these interactions to integration tests and limit their number
 - Use a record/playback library to make these kinds of tests faster



Keep your tests fast

More Complex Example

Let's look at an example project to see the impact of techniques that can be used to keep your tests fast.

- `code/clean-unit-tests/01-fast-unit-tests/github-client/test/repository.spec.js`



Poll: How fast is your test suite?

How long does your test suite take to run?

- More than one hour
- Between 30 minutes and an hour
- Between 10 minutes and 30 minutes
- Between 1 minute and 10 minutes
- Less than a minute
- I don't know
- We don't have a test suite



Keep your tests focused

Each test should have a single reason to fail

- When a test fails, makes it easier to determine what failure has been introduced in the codebase
- If a failure occurs in the middle of a test, and there are other assertions after the failure point, then those assertions are typically not run



Keep your tests focused

More Complex Example

Let's look at an example project to see the the effect of having multiple assertions in a single unit test

- `code/clean-unit-tests/02-single-assertion-per-test/square/spec/square_spec.rb`



Exercise

Practice working with focused unit tests

1. Open `exercises/clean-unit-tests/practice-single-assertion-per-test/test/repository.spec.js` in a text editor and convert the single unit test with multiple assertions into a multiple unit tests, each with a single assertion.



Keep your tests isolated

- Each test can be run on its own
 - Being able to run a single test is important when investigating failures.
- The entire test suite can be run in any order
 - Subtle changes in the test suite have impacts on the order that tests are run in



Keep your tests isolated

More Complex Example

Let's look at an example project to see the the effect of having tests that can only be run in a particular order

- `code/clean-unit-tests/03-isolated-unit-tests/Calculator/Calculator.sln`



Discussion

Answer the following questions in the Group Chat box

- Will the tests in your test suite still pass if they are run in a random order?
- What has your team done to defend against ordering dependencies in your test suite?



Q&A

Post your questions in either
the Group Chat or Q&A boxes



Stay In Touch

Here are ways to reach out and stay in touch

Corgibytes

<https://corgibytes.com>

Co-founder & CTO

Legacy Code Rocks

<https://legacycode.rocks>

Podcast Co-host

Twitter: [@mscottford](https://twitter.com/mscottford)

LinkedIn: [in/mscottford](https://in.linkedin.com/in/mscottford)

MenderCon 2022

June 8, 2022

<https://mendercon.com/>