**Clean Coding**

Table of Contents

[Writing Good Code 6](#_Toc83393147)

[Naming 7](#_Toc83393148)

[Intention-Revealing Names 7](#_Toc83393149)

[Avoid Disinformation 7](#_Toc83393150)

[Make Meaningful Distinctions 8](#_Toc83393151)

[Pronounceable Names 9](#_Toc83393152)

[Searchable Names 9](#_Toc83393153)

[Avoid ‘Clever’ Names 10](#_Toc83393154)

[One Word Per Concept 10](#_Toc83393155)

[Understanding Over Consistency 10](#_Toc83393156)

[Meaningful Context 11](#_Toc83393157)

[Verbs for Functions 11](#_Toc83393158)

[Nouns for Variables 11](#_Toc83393159)

[Unambiguous 11](#_Toc83393160)

[Comments 12](#_Toc83393161)

[Good Comments 13](#_Toc83393162)

[Bad Comments 15](#_Toc83393163)

[Formatting 19](#_Toc83393164)

[Formatting Rules 19](#_Toc83393165)

[Functions 20](#_Toc83393166)

[Best Practices 20](#_Toc83393167)

[Exception and Error Handling 23](#_Toc83393168)

[Error-Handling Rules 24](#_Toc83393169)

[Other Techniques 25](#_Toc83393170)

The four main goals of coding are:

1. The code must **work**.
2. The code must **solve** the problem.
3. The code must **fit well** into the existing system.
4. The code must be **readable** by other programmers.

The last step is where **clean coding** comes in.

Clean code is:

* Testable and Tested
* Elegant
* Efficient
* Expressive
* Expected
* Readable
* Minimal Dependencies
* Maintainable
* Self-Documenting
* Simple

Formally, clean code is software code that is **formatted correctly** and in an **organized manner** so that other developers can easily read or modify it.

It should be:

* **Simple, but Effective** – The code should solve the problem at hand without introducing unnecessary complexity.
* **Well-Tended** – Time and effort should be spent tending it and improving it.
* **Adaptable** – The code should be easy for colleagues to understand, update and modify. In addition, it should be adaptable when requirements change.
* **Testable** – The code should be designed to be testable. It needs to be covered by tests that run automatically. Using tests that run as we make changes, we can immediately get feedback on whether or not we broke something.

On the flip side, **bad code** has:

* Inappropriate information
* Obsolete, redundant or poorly written comments
* Commented out code
* Too many arguments
* Dead functions, which execute and produce results we are not using
* Incorrect behaviour and boundaries
* Code at the wrong level of abstraction
* Base classes depending on their derivates
* Too much information

Clean code is important because:

* Clean code **saves time**
* Clean code maintains **fast productivity**
* Clean code makes the **next developer** happy
* Messy code is **expensive** to own
* Messy code **decreases productivity**

The obstacles to clean code are:

* **Ignorance** – Not knowing how to write good code.
* **Stubbornness** – Knowing how but refusing to put in the effort to write good code.
* **Short-Timer Syndrome** - People who are leaving suck at writing code.
* **Arrogance** – Thinking their style is better.
* **Job Security** – Writing bad code on purpose to ensure they are needed to understand the code.

The main reason behind unclean code is leaving messy code with the intention of cleaning it up later.

Apparently, the official measurement of code quality is in WTFs per minute. Fewer is better in case it wasn’t obvious.

## Writing Good Code

Writing good code has several parts:

* Naming
* Functions
* Comments
* Source Code Structure
* Objects and Data Structures
* Error Handling
* Tests
* Code Smells

## Naming

### Intention-Revealing Names

We should try to use **variable names** that reveal the **intent** of the variable.

*// Bad*int d; *// elapsed time in days  
  
// Good*int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;

JAVA

If a variable name requires a comment, it is not intention-revealing.

### Avoid Disinformation

We should try to avoid leaving **false clues** that obscure the meaning of code.

For example, we should name a group of accounts accountGroup, not accountList, since the latter indicates that we have used a List data structure, which we have not.

In the same line of thought, we should avoid **encoding datatypes** within our variable names. For example, nameString should just be name and salaryFloat should just be salary.

### Make Meaningful Distinctions

It may be tempting to name two variables that are similar in exactly the same way, but doing this hides the reason those variables exist.

string cust; *// what is stored in cust that is different from customer?*string customer;  
  
class ProductInfo; *// how are these classes different?*class ProductData;  
  
void getActiveAccount(); *// how do you seperate these methods?*void getActiveAccounts();  
void getActiveAccountsInfo();

JAVA

If the work of the variables or functions truly are different, we should name them in a way so that we can distinguish between them.

The same goes for arguments in functions:

*//Bad*public static void copyChars (char a1[], char a2[])  
{  
 for (int i=0; i < a1.length; i++)  
 a2[i] = a1[i];  
}  
  
*//Good*public static void copyChars (char source[], char destination[])  
{  
 for (int i=0; i < source.length; i++)  
 destination[i] = source[i];  
}

JAVA

### Pronounceable Names

*//Bad*class DtaRcrd102

{  
 private Date genymdhms;  
 private Date modymdhms;  
 private final String pszqint = "102";  
};  
  
*//Good*class Customer

{  
 private Date generationTimestamp;  
 private Date modificationTimestamp;  
 private final String recordId = "102";  
};

JAVA

### Searchable Names

We should use variable names that are **easy to find** when reading the code. Single letter names or even direct values may be difficult to find.

*//Bad*for (int j = 0; j < 34; j++)

s += (t[j] \* 4) / 5;   
  
*//Good*int realDaysPerIdealDay = 4;  
const int WORK\_DAYS\_PER\_WEEK = 5;  
int sum = 0;  
for (int j = 0; j < NUMBER\_OF\_TASKS; j++)  
{  
 int realTaskDays = taskEstimate[j] \* realDaysPerIdealDay;  
 int realTaskWeeks = (realdays / WORK\_DAYS\_PER\_WEEK);  
 sum += realTaskWeeks;  
}

JAVA

### Avoid ‘Clever’ Names

DeleteItems is okay. HolyHandGrenade is not. abort() is okay. eatMyShorts() is not. Say what you mean. Mean what you say.

### One Word Per Concept

Functions or variables that do similar things (at different places) should use similar names. **Consistency** in naming will make understanding the code easier.

If we use dataFetcher() in one class to retrieve some data, we should not use dataGetter() in another class to retrieve some other data. We can just use the same function name.

### Understanding Over Consistency

This is the opposite of the last rule. If naming two functions in the same way makes their usage confusing, give them **different names**.

For example, if two classes have an add() method, but one of the methods is actually summing up two numbers while the other adds elements to a list, we should name the latter method insert() or append() instead.

### Meaningful Context

Names should be meaningful in the context they are being used in. If we store a global variable (avoid doing that too, but that’s a different topic) named state, it is difficult to tell what this variable does. If we had a class called address in which there was a variable called state, then it would be obvious. This is why context is important.

We should try to put variables in well-named classes or functions to give them the required context, but if it is not at all possible, then prefixes could be used as a last resort. For example, we can store a global variable named addressState.

### Verbs for Functions

For example, saveOrder(), getDiscounts(), etc.

### Nouns for Variables

For example, currentUser, priceFilter, etc.

### Unambiguous

Names should not suggest multiple meanings. The variable recdptrl is not only unpronounceable, but also suggests two meanings. Does it mean ‘received petrol’ or ‘record department role’?

## Comments

The first and foremost thing is that we should not add comments to explain bad code, we should **rewrite** the code. The best kind of comments are the ones we don’t need.

There are reasons why this ‘write fewer comments’ approach is being pushed:

1. Comments **lie** – Code may be updated in the future, while the corresponding comments are not. If we do make comments, we should make sure to update the comments if we modify our code.
2. Comments do not make up for **bad code** – As mentioned above, we should rewrite bad code instead of using comments to clarify things.
3. Code should be **self-explanatory** – Again, good code does not need to be explained.

Consider we have this code:

*// Check to see if the employee is eligible for full benefits*if ((employee.flags & HOURLY\_FLAG) && (employee.age > 65))

JAVA

Instead of this, if we have a small two-line method in the employee class that checks these conditions, the code will be much cleaner:

if (employee.isEligibleForFullBenefits())

JAVA

The reason this is cleaner is because when the developer is reading this line of code, most likely, he does not need to know what exactly makes the employee eligible for benefits. The details are not relevant right now and forcing the developer to read them will confuse them. All we need to know is whether the employee is eligible for benefits at all.

### Good Comments

The following are examples of good comments:

* **Legal Comments** – Sometimes, we should include legal content inside comments to notify anyone reading the code about where it came from. This is especially important to avoid legal issues over open-source code.

*// Copyright (C) 2003,2004,2005 by Object Mentor, Inc.*

*// All rights reserved.  
// Released under the terms of the GNU General Public License*

*// version 2 or later.*

JAVA

* **Explanation of Intent** – There are cases where it is not possible to make code self-explanatory or clearer. In those cases, we should explain what we attempted to do using comments. The following is not a very good example of this.

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

JAVA

* **Warning of Consequences** – If modifying a piece of code or writing it differently causes any problems, then we should mention this using comments so that future developers do not try to change it.

public static SimpleDateFormat makeStandardHttpDateFormat()  
{  
 *//SimpleDateFormat is not thread safe,  
 //so we need to create each instance independently.* SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");  
 df.setTimeZone(TimeZone.getTimeZone("GMT"));  
 return df;  
}

JAVA

* **TODO Comments** – These are comments that are meant to remind us of future work we plan to do. Some IDEs, like JetBrains, provide methods to keep track of TODO comments.

*// TODO-MdM these are not needed  
// We expect this to go away when we do the checkout model*protected VersionInfo makeVersion() throws Exception  
{  
 return null;  
}

JAVA

* **Amplification** – Sometimes, a piece of code may seem misleadingly unimportant. In those cases, comments can help amplify their importance.

String listItemContent = match.group(3).trim();  
*// the trim is really important. It removes the starting  
// spaces that could cause the item to be recognized  
// as another list.*new ListItemWidget(this, listItemContent, this.level + 1);  
return buildList(text.substring(match.end()));

JAVA

### Bad Comments

The following are examples of bad comments:

* **Mumbling** – We should not write comments just for the sake of writing them when they do not serve any purpose. The comments we do decide to write should be clear and easily understandable.

public void loadProperties()  
{  
 try  
 {  
 String propertiesPath = propertiesLocation + "/" + PROPERTIES\_FILE;  
 FileInputStream propertiesStream = new FileInputStream(propertiesPath);  
 loadedProperties.load(propertiesStream);  
 }  
 catch(IOException e)  
 {  
 *// No properties files means all defaults are loaded* }  
}

JAVA

The comment above is unclear. It is difficult to understand what it means.

* **Redundant Comments** – We should not write comments for code that is self-explanatory.

*// Utility method that returns when this.closed is true.*

*// Throws an exception if the timeout is reached.*public synchronized void waitForClose (final long timeoutMillis) throws Exception  
{  
 if(!closed)  
 {  
 wait(timeoutMillis);  
 if(!closed)  
 throw new Exception("MockResponseSender could not be closed");  
 }  
}

JAVA

The comment above does not serve any purpose. It is not more informative than the code, not easier to read and does not justify the code either. It is redundant.

* **Mandated Comments** – Organizations that have a policy of writing a comment for every variable or every function are dumb.

void hashIt(data)  
{  
 let hash = 0; *// The hash* const length = data.length; *// Length of string  
  
 // Loop through every character in data* for (let i = 0; i < length; i++)  
 {  
 const char = data.charCodeAt(i); *// Get character code.* hash = (hash << 5) - hash + char; *// Make the hash* hash &= hash; *// Convert to 32-bit integer* }  
}

JAVA

* **Journal Comments** – Journal comments are comments that exist to keep track of changes to code. Don’t use them. Version control exists. Use it.

*/\*\*  
 \* 2016-12-20: Removed monads, didn't understand them (RM)  
 \* 2016-10-01: Improved using special monads (JP)  
 \* 2016-02-03: Removed type-checking (LI)  
 \* 2015-03-14: Added combine with type-checking (JR)  
 \*/*void combine(a, b)  
{  
 return a + b;  
}

JAVA

* **Noise Comments** – Again, don’t write unnecessary comments. No clue why this is another comment type. It’s the same as redundant comments.
* **Position Markers** – Don’t use comments to denote ‘sections’. This is a type of Noise Comment.

*// Actions //////////////////////////////////*

JAVA

* **Explaining Code** – We should try to avoid comments that explain our code when there are ways to explain the code with itself.

*// does the module from the global list <mod> depend on the  
// subsystem we are part of?*if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))

JAVA

Instead of using comments to explain the line of code above, we can use variables with appropriate names to make what is happening clearer.

ArrayList moduleDependees = smodule.getDependSubsystems();  
String ourSubSystem = subSysMod.getSubSystem();  
if (moduleDependees.contains(ourSubSystem))

JAVA

* **Closing Brace Comments** – Don’t use comments to signify which brace ends what section.

try  
{  
 while ((line = in.readLine()) != **NULL**)  
 {  
 lineCount++;  
 charCount += line.length();  
 String words[] = line.split("\\W");  
 wordCount += words.length;  
 } *//while* System.out.println("wordCount = " + wordCount);  
 System.out.println("lineCount = " + lineCount);  
 System.out.println("charCount = " + charCount);  
} *// try*catch (IOException e)  
{  
 System.err.println("Error:" + e.getMessage());  
} *//catch*

JAVA

* **Attributions** – Don’t place attributions in comments. Use Version Control.

*/\* Added by Rick \*/*

JAV

* **Commented Out Code** – Don’t comment out old code that you may use again. Use Version Control.

doStuff();  
*// doOtherStuff();*

JAVA

## Formatting

### Formatting Rules

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar function should be close.
6. Place functions in the downward direction, meaning dependent functions should be below the functions they depend on.
7. Keep lines short.
8. Don’t use horizontal alignment.
9. Don’t break indentation.

## Functions

### Best Practices

1. Functions should be really **small**. No. Even smaller. Literally 20 to 30 lines. Shorter functions are more likely to follow SRP and will be easier to understand.
2. If-else **blocks**, while blocks etc. should be kept minimal. Preferably, they should call a separate function that executes the actual code instead of implementing the code inside the block. Similarly, **nested blocks** should be avoided and placed in separate functions.
3. **Indentation** should be kept minimal, not more than two tabs deep.
4. A function should do just one thing. Follow the **Single Responsibility Principle**.
5. Function **names** should say exactly what they do. The name of a function, variable or class should answer the following:

* Why does it exist?
* What does it do?
* How is it used?

1. Avoid **duplicate code**. Duplicate code must be updated in multiple places for each change.
2. Avoid **side effects**. Our functions should not do more than one thing without informing the client. In particular, our functions should avoid **temporal coupling**, where a dependency exists which is not explicitly specified. For example, say we have an object from a Car class and we want to invoke a turnRight method. Before doing this, the startEngine method must be invoked, or else the program will crash. However, we did not provide any hints that this is required. The client using our class may not realize this.
3. Remove **dead code**. If we need it again, we can retrieve it from Version Control.
4. Maintain the **same level** of **abstraction**. This will allow us to realize when a function is doing too much.

For example, consider a function that takes a bunch of items, checks whether those items are available and then calculates their total price. Say we are calling a different function in a loop to check if each of the items are available. This is one level of abstraction, since the code to actually check for availability is hidden from us.

On the other hand, we are calculating the total price directly within this function instead of calling a different function, meaning the code to calculate the price is not hidden from us. This is a different level of abstraction.

Thus, we have multiple levels of abstraction. Coding like this makes the code difficult to read. Instead, we should hide the code to calculate the total price inside a different function and call that function. This would maintain the same level of abstraction and make our code easier to read.

1. Keep the number of **arguments** low, **three** at most. If there are more arguments, it might be better to wrap the arguments into a class and use objects of the class instead.
2. Maintain **command query separation**. This means that a function should either **do something** or **answer something**, but not both. Doing both causes confusion.

The only way to actually get a function to follow all of these rules is to **repeatedly revise**. Every time we go over our function, we can modify it slightly to make the code a little cleaner.

## Exception and Error Handling

Things go wrong. When they do go wrong, we need to make sure our code handles the situation appropriately.

One way to handle the situation when things go wrong is to use **error codes**. If we do use error codes, we will have to map the error codes to the errors and also take appropriate action for the error.

int withdraw (int amount)  
{  
 if (amount > balance) return -1;  
 else balance -= amount;  
 return 0;  
}

JAVA

In these cases, it is better to **throw an exception**. This simplifies the process and also separates error handling from the logic of the code. We simply attempt to run the code and if a problem occurs, we take some action. In this way, our code itself does not deal with the errors at all.

void withdraw (int amount) throws BalanceException  
{  
 if (amount > balance) throw new BalanceException();  
 balance -= amount;  
}

JAVA

### Error-Handling Rules

1. Write the **try-catch-finally** statement first. Try blocks are like transactions, executing a block of code that might cause an exception. The catch block is meant to handle the situation if an exception does occur. The finally block executes regardless of whether or not there was an error.
2. Provide **context** with exceptions. Each exception that is thrown should provide enough context to identify the **source** of an error. Exceptions should contain **informative error messages**. They should mention the operation that failed and the type of failure. If the application logs errors, pass enough information to allow developers to figure out what went wrong from the logs.
3. **Don’t** **return null**. Instead, throw an exception. When using a method from a third-party API that returns null, wrap the method call with a different method that throws an exception.
4. **Don’t pass null**. Returning null is bad, but passing null is worse. Unless a third-party API expects you to pass null, avoid it.

More details regarding exceptions and error handling can be found [here](https://isocpp.org/wiki/faq/exceptions).

## Other Techniques

1. Keep things simple.
2. Understand your code.
3. Don’t repeat yourself.
4. Indent your code.
5. Follow the naming convention.
6. Explore.
7. Use your brain.
8. Avoid hard-coded string and magic numbers.
9. Always write braces around single-line blocks.
10. Good comments are your best friend.
11. Always perform test runs.
12. Write everything code-related in English.
13. Practice your art.