**10 Basic Programming Principles Every Programmer Must Know**

Your code should be clear and easy to maintain. Here are several other programming principles to help you clean up your act.

[**BY ANTHONY GRANT**](https://www.makeuseof.com/author/anthonygrant/)JUL 14, 2020

It's easy to write code. It's challenging to write good code.

Bad code comes in many forms. Messy code, massive if-else chains, programs that break with one adjustment, variables that don't make sense. The program might work once but will never hold up to any scrutiny.

Don't settle for shortcuts. Aim to write code that is easy to maintain. Easy for you to maintain, and easy for any other developer on your team to maintain. How do you write effective code?  You write good code by being disciplined with programming principles.

Here are 10 programming principles that will make you a better coder.

**1. Keep It Simple, Stupid (KISS)**

It sounds a little harsh, but it's a coding principle to live by. What does this mean?

It means you should be writing code as simple as possible. Don't get caught up in trying to be overly clever or showing off with a paragraph of advanced code. If you can write a script in one line, write it in one line.

Here's a simple function:

function addNumbers(num1,num2){

return num1 + num2;

}

Pretty simple. It's easy to read and you know exactly what is going on.

Use clear variable names. Take advantage of coding libraries to use existing tools. Make it easy to come back after six months and get right back to work. Keeping it simple will save you the headache.

**2. Write DRY Code**

The**Don't Repeat Yourself (DRY)** principle means, plainly, not repeating code. It's a [common coding mistake](https://www.makeuseof.com/tag/common-programming-coding-mistakes/). When writing code, avoid duplication of data or logic. If you've ever copied and pasted code within your program, it's not DRY code.

Take a look at this script:

function addNumberSequence(number){  
 number = number + 1;

number = number + 2;

number = number + 3;

number = number + 4;

number = number + 5;

return number;

}

Instead of duplicating lines, try to find an algorithm that uses iteration. For loops, and while loops are ways to control code that needs to run multiple times.

DRY code is easy to maintain. It's easier to debug one loop that handles 50 repetitions than 50 blocks of code that handle one repetition.

**3. Open/Closed**

This principle means you should aim to make your code open to extension but closed to modification. This is an important principle when releasing a library or framework that others will use.

For example, suppose you're maintaining a GUI framework. You could release for coders to directly modify and integrate your released code. But what happens when you release a major update four months later?

Their code will break. This will make engineers unhappy. They won't want to use your library for much longer, no matter how helpful it may be.

Instead, release code that prevents direct modification and encourages extension. This separates core behavior from modified behavior. The code is more stable and easier to maintain.

**4. Composition Over Inheritance**

If you write code using [object-oriented programming](https://www.makeuseof.com/tag/python-object-oriented-programming/) you're going to find this useful. The **composition over inheritance**principle states: objects with complex behaviors should contain instances of objects with individual behaviors. They should not inherit a class and add new behaviors.

Relying on inheritance causes two major issues. First, the inheritance hierarchy can get messy in a hurry. You also have less flexibility for defining special-case behaviors. Let's say you want to implement behaviors to share:

Composition programming is a lot cleaner to write, easier to maintain and allows flexibility defining behaviors. Each individual behavior is its own class. You can create complex behaviors by combining individual behaviors.

**5. Single Responsibility**

The **single responsibility principle** says that every class or module in a program should only provide one specific functionality. As Robert C. Martin puts it, "A class should have only one reason to change."

[Classes and modules](https://www.makeuseof.com/tag/python-instance-static-class-methods/) often start off this way. Be careful not to add too many responsibilities as classes get more complicated. Refactor and break them up into smaller classes and modules.

The consequence of overloading classes is twofold. First, it complicates debugging when you're trying to isolate a certain module for troubleshooting. Second, it becomes more difficult to create additional functionality for a specific module.

**6. Separation Of Concerns**

The **separation of concerns principle** is an abstract version of the single responsibility principle. This idea states that a program should be designed with different containers, and these containers should not have access to each other.

A well-known example of this is the model-view-controller (MVC) design. MVC separates a program into three distinct areas: the data (model), the logic (controller), and what the page displays (view). Variations of MVC are common in today's most popular web frameworks.

For example, the code that handles the database doesn't need to know how to render the data in the browser. The rendering code takes input from the user, but the logic code handles the processing. Each piece of code is completely independent.

The result is code that is easy to debug. If you ever need to rewrite the rendering code, you can do so without worrying about how the data gets saved or the logic gets processed.

**7. You Aren't Going To Need It (YAGNI)**

Thisprinciple means you should never code for functionality on the chance that you may need in the future. Don't try and solve a problem that doesn't exist.

In an effort to write DRY code, programmers can violate this principle. Often inexperienced programmers try to write the most abstract and generic code they can. Too much abstraction causes bloated code that is impossible to maintain.

Only apply the DRY principle only when you need to. If you notice chunks of code written over and over, then abstract them. Don't think too far out at the expense of your current code batch.

**8. Document Your Code**

Any senior developer will stress the importance of documenting your code with proper comments. All languages offer them and you should make it a habit to write them. Leave comments to explain objects, enhance variable definitions, and make functions easier to understand.

Here's a JavaScript function with comments guiding you through the code:

//This function will add 5 to the input if odd, or return the number if even

function evenOrOdd(number){

//Determine if the number is even

if(number % 2 == 0){

return number;

}

//If the number is odd, this will add 5 and return

else {

return number + 5;

}

}

Leaving comments is a little more work while you're coding, and you understand your code pretty well right?

Leave comments anyway!

Try writing a program, leaving it alone for six months, and come back to modify it. You'll be glad you documented your program instead of having to pour over every function to remember how it works. Work on a coding team? Don't frustrate your fellow developers by forcing them to decipher your syntax.

**9. Refactor**

It's hard to accept, but your code isn't going to be perfect the first time. Refactoring code means reviewing your code and looking for ways to optimize it. Make it more efficient while keeping the results exactly the same.

Codebases are constantly evolving. It's completely normal to revisit, rewrite, or even redesign entire chunks of code. It doesn't mean you didn't succeed the first time you wrote your program. You're going to get more familiar with a project over time. Use that knowledge to adjust your existing code to be DRY, or following the KISS principle.

**10. Clean Code At All Costs**

Leave your ego at the door and forget about writing clever code. The kind of code that looks more like a riddle than a solution. You're not coding to impress strangers.

Don't try to pack a ton of logic into one line. Leave clear instructions in comments and documentation. If your code is easy to read, it will be easy to maintain.

Good programmers and readable code go hand-in-hand. Leave comments when necessary. Adhere to style guides, whether dictated by a language or your company.

**What Makes A Good Programmer?**

Learning how to be a good programmer takes quite a bit of work! These 10 coding principles are a roadmap to becoming a professional programmer.

A good programmer understands how to make their apps easy to use, works well within a team, and finishes projects to specification and on time. By following these principles you will set yourself up for success in your programming career. Try out these [10 beginner programming projects](https://www.makeuseof.com/tag/beginner-programming-projects/) and review your code. See if you're sticking to these principles. If not, challenge yourself to improve your code.