# Extract Method

### Problem

You have a code fragment that can be grouped together.

**void** **printOwing**() {

printBanner();

// Print details.

System.out.println("name: " + name);

System.out.println("amount: " + getOutstanding());

}

### Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

**void** **printOwing**() {

printBanner();

printDetails(getOutstanding());

}

**void** **printDetails**(**double** outstanding) {

System.out.println("name: " + name);

System.out.println("amount: " + outstanding);

}

### Why Refactor

The more lines found in a method, the harder it’s to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

### Benefits

* More readable code! Be sure to give the new method a name that describes the method’s purpose: createOrder(), renderCustomerInfo(), etc.
* Less code duplication. Often the code that’s found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.
* Isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

### How to Refactor

1. Create a new method and name it in a way that makes its purpose self-evident.
2. Copy the relevant code fragment to your new method. Delete the fragment from its old location and put a call for the new method there instead.

Find all variables used in this code fragment. If they’re declared inside the fragment and not used outside of it, simply leave them unchanged—they’ll become local variables for the new method.

1. If the variables are declared prior to the code that you’re extracting, you will need to pass these variables to the parameters of your new method in order to use the values previously contained in them. Sometimes it’s easier to get rid of these variables by resorting to [**Replace Temp with Query**](https://refactoring.guru/replace-temp-with-query).
2. If you see that a local variable changes in your extracted code in some way, this may mean that this changed value will be needed later in your main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

# Inline Method

### Problem

When a method body is more obvious than the method itself, use this technique.

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

### Solution

Replace calls to the method with the method’s content and delete the method itself.

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** numberOfLateDeliveries > 5 ? 2 : 1;

}

}

### Why Refactor

A method simply delegates to another method. In itself, this delegation is no problem. But when there are many such methods, they become a confusing tangle that’s hard to sort through.

Often methods aren’t too short originally, but become that way as changes are made to the program. So don’t be shy about getting rid of methods that have outlived their use.

### Benefits

* By minimizing the number of unneeded methods, you make the code more straightforward.

### How to Refactor

1. Make sure that the method isn’t redefined in subclasses. If the method is redefined, refrain from this technique.
2. Find all calls to the method. Replace these calls with the content of the method.
3. Delete the method.

# Extract Variable

### Problem

You have an expression that’s hard to understand.

**void** **renderBanner**() {

**if** ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// do something

}

}

### Solution

Place the result of the expression or its parts in separate variables that are self-explanatory.

**void** **renderBanner**() {

**final** **boolean** isMacOs = platform.toUpperCase().indexOf("MAC") > -1;

**final** **boolean** isIE = browser.toUpperCase().indexOf("IE") > -1;

**final** **boolean** wasResized = resize > 0;

**if** (isMacOs && isIE && wasInitialized() && wasResized) {

// do something

}

}

### Why Refactor

The main reason for extracting variables is to make a complex expression more understandable, by dividing it into its intermediate parts. These could be:

* Condition of the if() operator or a part of the ?: operator in C-based languages
* A long arithmetic expression without intermediate results
* Long multipart lines

Extracting a variable may be the first step towards performing [**Extract Method**](https://refactoring.guru/extract-method) if you see that the extracted expression is used in other places in your code.

### Benefits

* More readable code! Try to give the extracted variables good names that announce the variable’s purpose loud and clear. More readability, fewer long-winded comments. Go for names like customerTaxValue, cityUnemploymentRate, clientSalutationString, etc.

### Drawbacks

* More variables are present in your code. But this is counterbalanced by the ease of reading your code.
* When refactoring conditional expressions, remember that the compiler will most likely optimize it to minimize the amount of calculations needed to establish the resulting value. Say you have a following expression if (a() || b()) .... The program won’t call the method b if the method a returns true because the resulting value will still be true, no matter what value returns b.

However, if you extract parts of this expression into variables, both methods will always be called, which might hurt performance of the program, especially if these methods do some heavyweight work.

### How to Refactor

1. Insert a new line before the relevant expression and declare a new variable there. Assign part of the complex expression to this variable.
2. Replace that part of the expression with the new variable.
3. Repeat the process for all complex parts of the expression.

# Inline Temp

### Problem

You have a temporary variable that’s assigned the result of a simple expression and nothing more.

**boolean** **hasDiscount**(Order order) {

**double** basePrice = order.basePrice();

**return** basePrice > 1000;

}

### Solution

Replace the references to the variable with the expression itself.

**boolean** **hasDiscount**(Order order) {

**return** order.basePrice() > 1000;

}

### Why Refactor

Inline local variables are almost always used as part of [**Replace Temp with Query**](https://refactoring.guru/replace-temp-with-query) or to pave the way for [**Extract Method**](https://refactoring.guru/extract-method).

### Benefits

* This refactoring technique offers almost no benefit in and of itself. However, if the variable is assigned the result of a method, you can marginally improve the readability of the program by getting rid of the unnecessary variable.

### Drawbacks

* Sometimes seemingly useless temps are used to cache the result of an expensive operation that’s reused several times. So before using this refactoring technique, make sure that simplicity won’t come at the cost of performance.

### How to Refactor

1. Find all places that use the variable. Instead of the variable, use the expression that had been assigned to it.
2. Delete the declaration of the variable and its assignment line.

# Replace Temp with Query

### Problem

You place the result of an expression in a local variable for later use in your code.

**double** **calculateTotal**() {

**double** basePrice = quantity \* itemPrice;

**if** (basePrice > 1000) {

**return** basePrice \* 0.95;

}

**else** {

**return** basePrice \* 0.98;

}

}

### Solution

Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

**double** **calculateTotal**() {

**if** (basePrice() > 1000) {

**return** basePrice() \* 0.95;

}

**else** {

**return** basePrice() \* 0.98;

}

}

**double** **basePrice**() {

**return** quantity \* itemPrice;

}

### Why Refactor

This refactoring can lay the groundwork for applying [**Extract Method**](https://refactoring.guru/extract-method) for a portion of a very long method.

The same expression may sometimes be found in other methods as well, which is one reason to consider creating a common method.

### Benefits

* Code readability. It’s much easier to understand the purpose of the method getTax() than the line orderPrice() \* 0.2.
* Slimmer code via deduplication, if the line being replaced is used in multiple methods.

### Good to Know

#### Performance

This refactoring may prompt the question of whether this approach is liable to cause a performance hit. The honest answer is: yes, it is, since the resulting code may be burdened by querying a new method. But with today’s fast CPUs and excellent compilers, the burden will almost always be minimal. By contrast, readable code and the ability to reuse this method in other places in program code—thanks to this refactoring approach—are very noticeable benefits.

Nonetheless, if your temp variable is used to cache the result of a truly time-consuming expression, you may want to stop this refactoring after extracting the expression to a new method.

### How to Refactor

1. Make sure that a value is assigned to the variable once and only once within the method. If not, use [**Split Temporary Variable**](https://refactoring.guru/split-temporary-variable) to ensure that the variable will be used only to store the result of your expression.
2. Use [**Extract Method**](https://refactoring.guru/extract-method) to place the expression of interest in a new method. Make sure that this method only returns a value and doesn’t change the state of the object. If the method affects the visible state of the object, use [**Separate Query from Modifier**](https://refactoring.guru/separate-query-from-modifier).
3. Replace the variable with a query to your new method.

# Split Temporary Variable

### Problem

You have a local variable that’s used to store various intermediate values inside a method (except for cycle variables).

**double** temp = 2 \* (height + width);

System.out.println(temp);

temp = height \* width;

System.out.println(temp);

### Solution

Use different variables for different values. Each variable should be responsible for only one particular thing.

**final** **double** perimeter = 2 \* (height + width);

System.out.println(perimeter);

**final** **double** area = height \* width;

System.out.println(area);

### Why Refactor

If you’re skimping on the number of variables inside a function and reusing them for various unrelated purposes, you’re sure to encounter problems as soon as you need to make changes to the code containing the variables. You will have to recheck each case of variable use to make sure that the correct values are used.

### Benefits

* Each component of the program code should be responsible for one and one thing only. This makes it much easier to maintain the code, since you can easily replace any particular thing without fear of unintended effects.
* Code becomes more readable. If a variable was created long ago in a rush, it probably has a name that doesn’t explain anything: k, a2, value, etc. But you can fix this situation by naming the new variables in an understandable, self-explanatory way. Such names might resemble customerTaxValue, cityUnemploymentRate, clientSalutationString and the like.
* This refactoring technique is useful if you anticipate using [**Extract Method**](https://refactoring.guru/extract-method) later.

### How to Refactor

1. Find the first place in the code where the variable is given a value. Here you should rename the variable with a name that corresponds to the value being assigned.
2. Use the new name instead of the old one in places where this value of the variable is used.
3. Repeat as needed for places where the variable is assigned a different value.

# Remove Assignments to Parameters

### Problem

Some value is assigned to a parameter inside method’s body.

**int** **discount**(**int** inputVal, **int** quantity) {

**if** (quantity > 50) {

inputVal -= 2;

}

// ...

}

### Solution

Use a local variable instead of a parameter.

**int** **discount**(**int** inputVal, **int** quantity) {

**int** result = inputVal;

**if** (quantity > 50) {

result -= 2;

}

// ...

}

### Why Refactor

The reasons for this refactoring are the same as for [**Split Temporary Variable**](https://refactoring.guru/split-temporary-variable), but in this case we’re dealing with a parameter, not a local variable.

First, if a parameter is passed via reference, then after the parameter value is changed inside the method, this value is passed to the argument that requested calling this method. Very often, this occurs accidentally and leads to unfortunate effects. Even if parameters are usually passed by value (and not by reference) in your programming language, this coding quirk may alienate those who are unaccustomed to it.

Second, multiple assignments of different values to a single parameter make it difficult for you to know what data should be contained in the parameter at any particular point in time. The problem worsens if your parameter and its contents are documented but the actual value is capable of differing from what’s expected inside the method.

### Benefits

* Each element of the program should be responsible for only one thing. This makes code maintenance much easier going forward, since you can safely replace code without any side effects.
* This refactoring helps to extract [**repetitive code to separate methods**](https://refactoring.guru/extract-method).

### How to Refactor

1. Create a local variable and assign the initial value of your parameter.
2. In all method code that follows this line, replace the parameter with your new local variable.

# Replace Method with Method Object

### Problem

You have a long method in which the local variables are so intertwined that you can’t apply *Extract Method*.

**class** **Order** {

// ...

**public** **double** price() {

**double** primaryBasePrice;

**double** secondaryBasePrice;

**double** tertiaryBasePrice;

// Perform long computation.

}

}

### Solution

Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class.

**class** **Order** {

// ...

**public** **double** price() {

**return** **new** PriceCalculator(**this**).compute();

}

}

**class** **PriceCalculator** {

**private** **double** primaryBasePrice;

**private** **double** secondaryBasePrice;

**private** **double** tertiaryBasePrice;

**public** PriceCalculator(Order order) {

// Copy relevant information from the

// order object.

}

**public** **double** compute() {

// Perform long computation.

}

}

### Why Refactor

A method is too long and you can’t separate it due to tangled masses of local variables that are hard to isolate from each other.

The first step is to isolate the entire method into a separate class and turn its local variables into fields of the class.

Firstly, this allows isolating the problem at the class level. Secondly, it paves the way for splitting a large and unwieldy method into smaller ones that wouldn’t fit with the purpose of the original class anyway.

### Benefits

* Isolating a long method in its own class allows stopping a method from ballooning in size. This also allows splitting it into submethods within the class, without polluting the original class with utility methods.

### Drawbacks

* Another class is added, increasing the overall complexity of the program.

### How to Refactor

1. Create a new class. Name it based on the purpose of the method that you’re refactoring.
2. In the new class, create a private field for storing a reference to an instance of the class in which the method was previously located. It could be used to get some required data from the original class if needed.
3. Create a separate private field for each local variable of the method.
4. Create a constructor that accepts as parameters the values of all local variables of the method and also initializes the corresponding private fields.
5. Declare the main method and copy the code of the original method to it, replacing the local variables with private fields.
6. Replace the body of the original method in the original class by creating a method object and calling its main method.

# Substitute Algorithm

### Problem

So you want to replace an existing algorithm with a new one?

**String** **foundPerson**(**String**[] people){

**for** (**int** i = 0; i < people.length; i++) {

**if** (people[i].equals("Don")){

**return** "Don";

}

**if** (people[i].equals("John")){

**return** "John";

}

**if** (people[i].equals("Kent")){

**return** "Kent";

}

}

**return** "";

}

### Solution

Replace the body of the method that implements the algorithm with a new algorithm.

**String** **foundPerson**(**String**[] people){

List candidates =

Arrays.asList(**new** **String**[] {"Don", "John", "Kent"});

**for** (**int** i=0; i < people.length; i++) {

**if** (candidates.contains(people[i])) {

**return** people[i];

}

}

**return** "";

}

### Why Refactor

1. Gradual refactoring isn’t the only method for improving a program. Sometimes a method is so cluttered with issues that it’s easier to tear down the method and start fresh. And perhaps you have found an algorithm that’s much simpler and more efficient. If this is the case, you should simply replace the old algorithm with the new one.
2. As time goes on, your algorithm may be incorporated into a well-known library or framework and you want to get rid of your independent implementation, in order to simplify maintenance.
3. The requirements for your program may change so heavily that your existing algorithm can’t be salvaged for the task.

### How to Refactor

1. Make sure that you have simplified the existing algorithm as much as possible. Move unimportant code to other methods using [**Extract Method**](https://refactoring.guru/extract-method). The fewer moving parts in your algorithm, the easier it’s to replace.
2. Create your new algorithm in a new method. Replace the old algorithm with the new one and start testing the program.
3. If the results don’t match, return to the old implementation and compare the results. Identify the causes of the discrepancy. While the cause is often an error in the old algorithm, it’s more likely due to something not working in the new one.
4. When all tests are successfully completed, delete the old algorithm for good!