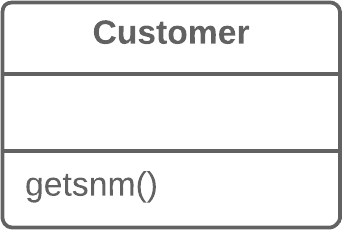
**Rename Method**

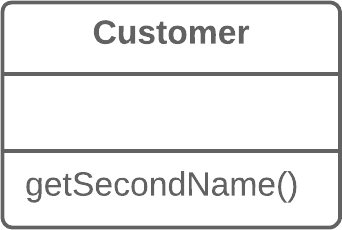
**Problem**

The name of a method doesn’t explain what the method does.



**Solution**

Rename the method.



**Why Refactor**

Perhaps a method was poorly named from the very beginning—for example, someone created the method in a rush and didn’t give proper care to naming it well.

Or perhaps the method was well named at first but as its functionality grew, the method name stopped being a good descriptor.

**Benefits**

* Code readability. Try to give the new method a name that reflects what it does. Something like createOrder(), renderCustomerInfo(), etc.

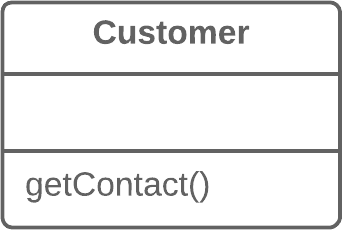
**How to Refactor**

1. See whether the method is defined in a superclass or subclass. If so, you must repeat all steps in these classes too.
2. The next method is important for maintaining the functionality of the program during the refactoring process. Create a new method with a new name. Copy the code of the old method to it. Delete all the code in the old method and, instead of it, insert a call for the new method.
3. Find all references to the old method and replace them with references to the new one.
4. Delete the old method. If the old method is part of a public interface, don’t perform this step. Instead, mark the old method as deprecated.

**Add Parameter**

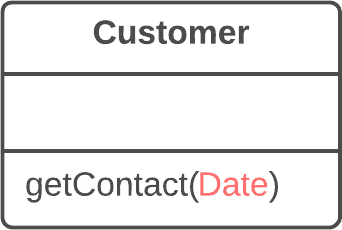
**Problem**

A method doesn’t have enough data to perform certain actions.



**Solution**

Create a new parameter to pass the necessary data.



**Why Refactor**

You need to make changes to a method and these changes require adding information or data that was previously not available to the method.

**Benefits**

* The choice here is between adding a new parameter and adding a new private field that contains the data needed by the method. A parameter is preferable when you need some occasional or frequently changing data for which there’s no point in holding it in an object all of the time. In this case, the refactoring will pay off. Otherwise, add a private field and fill it with the necessary data before calling the method.

**Drawbacks**

* Adding a new parameter is always easier than removing it, which is why parameter lists frequently balloon to grotesque sizes. This smell is known as the [**Long Parameter List**](https://refactoring.guru/smells/long-parameter-list).
* If you need to add a new parameter, sometimes this means that your class doesn’t contain the necessary data or the existing parameters don’t contain the necessary related data. In both cases, the best solution is to consider moving data to the main class or to other classes whose objects are already accessible from inside the method.

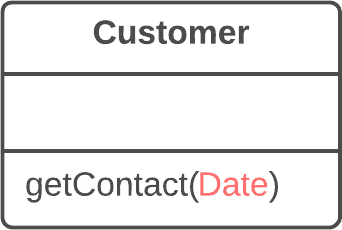
**How to Refactor**

1. See whether the method is defined in a superclass or subclass. If the method is present in them, you will need to repeat all the steps in these classes as well.
2. The following step is critical for keeping your program functional during the refactoring process. Create a new method by copying the old one and add the necessary parameter to it. Replace the code for the old method with a call to the new method. You can plug in any value to the new parameter (such as null for objects or a zero for numbers).
3. Find all references to the old method and replace them with references to the new method.
4. Delete the old method. Deletion isn’t possible if the old method is part of the public interface. If that’s the case, mark the old method as deprecated.

**Remove Parameter**

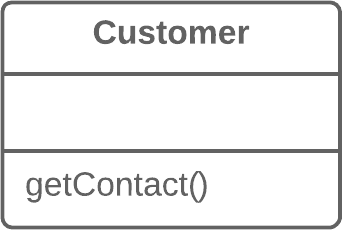
**Problem**

A parameter isn’t used in the body of a method.



**Solution**

Remove the unused parameter.



**Why Refactor**

Every parameter in a method call forces the programmer reading it to figure out what information is found in this parameter. And if a parameter is entirely unused in the method body, this “noggin scratching” is for naught.

And in any case, additional parameters are extra code that has to be run.

Sometimes we add parameters with an eye to the future, anticipating changes to the method for which the parameter might be needed. All the same, experience shows that it’s better to add a parameter only when it’s genuinely needed. After all, anticipated changes often remain just that—anticipated.

**Benefits**

* A method contains only the parameters that it truly requires.

**When Not to Use**

* If the method is implemented in different ways in subclasses or in a superclass, and your parameter is used in those implementations, leave the parameter as-is.

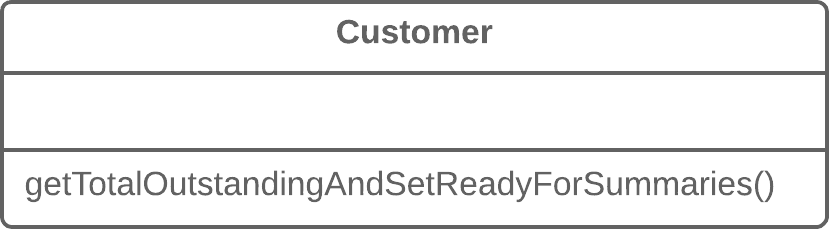
**How to Refactor**

1. See whether the method is defined in a superclass or subclass. If so, is the parameter used there? If the parameter is used in one of these implementations, hold off on this refactoring technique.
2. The next step is important for keeping the program functional during the refactoring process. Create a new method by copying the old one and delete the relevant parameter from it. Replace the code of the old method with a call to the new one.
3. Find all references to the old method and replace them with references to the new method.
4. Delete the old method. Don’t perform this step if the old method is part of a public interface. In this case, mark the old method as deprecated.

**Separate Query from Modifier**

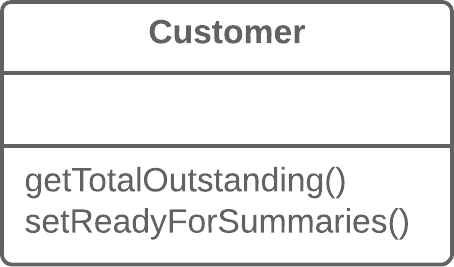
**Problem**

Do you have a method that returns a value but also changes something inside an object?



**Solution**

Split the method into two separate methods. As you would expect, one of them should return the value and the other one modifies the object.



**Why Refactor**

This factoring technique implements *Command and Query Responsibility Segregation*. This principle tells us to separate code responsible for getting data from code that changes something inside an object.

Code for getting data is named a *query*. Code for changing things in the *visible state* of an object is named a *modifier*. When a *query* and *modifier* are combined, you don’t have a way to get data without making changes to its condition. In other words, you ask a question and can change the answer even as it’s being received. This problem becomes even more severe when the person calling the query may not know about the method’s “side effects”, which often leads to runtime errors.

But remember that side effects are dangerous only in the case of *modifiers* that change the **visible** state of an object. These could be, for example, fields accessible from an object’s public interface, entry in a database, in files, etc. If a *modifier* only caches a complex operation and saves it within the private field of a class, it can hardly cause any side effects.

**Benefits**

* If you have a *query* that doesn’t change the state of your program, you can call it as many times as you like without having to worry about unintended changes in the result caused by the mere fact of you calling the method.

**Drawbacks**

* In some cases it’s convenient to get data after performing a command. For example, when deleting something from a database you want to know how many rows were deleted.

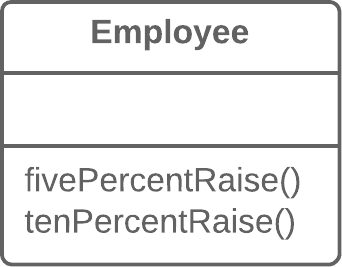
**How to Refactor**

1. Create a new *query method* to return what the original method did.
2. Change the original method so that it returns only the result of calling the new *query method*.
3. Replace all references to the original method with a call to the *query method*. Immediately before this line, place a call to the *modifier method*. This will save you from side effects in case if the original method was used in a condition of a conditional operator or loop.
4. Get rid of the value-returning code in the original method, which now has become a proper *modifier method*.

**Parameterize Method**

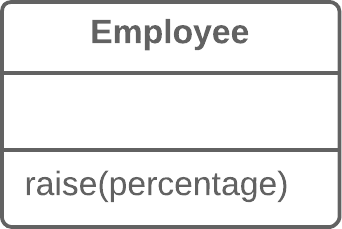
**Problem**

Multiple methods perform similar actions that are different only in their internal values, numbers or operations.



**Solution**

Combine these methods by using a parameter that will pass the necessary special value.



**Why Refactor**

If you have similar methods, you probably have duplicate code, with all the consequences that this entails.

What’s more, if you need to add yet another version of this functionality, you will have to create yet another method. Instead, you could simply run the existing method with a different parameter.

**Drawbacks**

* Sometimes this refactoring technique can be taken too far, resulting in a long and complicated common method instead of multiple simpler ones.
* Also be careful when moving activation/deactivation of functionality to a parameter. This can eventually lead to creation of a large conditional operator that will need to be treated via [**Replace Parameter with Explicit Methods**](https://refactoring.guru/replace-parameter-with-explicit-methods).

**How to Refactor**

1. Create a new method with a parameter and move it to the code that’s the same for all classes, by applying [**Extract Method**](https://refactoring.guru/extract-method). Note that sometimes only a certain part of methods is actually the same. In this case, refactoring consists of extracting only the same part to a new method.
2. In the code of the new method, replace the special/differing value with a parameter.
3. For each old method, find the places where it’s called, replacing these calls with calls to the new method that include a parameter. Then delete the old method.

**Replace Parameter with Explicit Methods**

**Problem**

A method is split into parts, each of which is run depending on the value of a parameter.

**void** **setValue**(**String** name, **int** value) {

**if** (name.equals("height")) {

height = value;

**return**;

}

**if** (name.equals("width")) {

width = value;

**return**;

}

Assert.shouldNeverReachHere();

}

**Solution**

Extract the individual parts of the method into their own methods and call them instead of the original method.

**void** **setHeight**(**int** arg) {

height = arg;

}

**void** **setWidth**(**int** arg) {

width = arg;

}

**Why Refactor**

A method containing parameter-dependent variants has grown massive. Non-trivial code is run in each branch and new variants are added very rarely.

**Benefits**

* Improves code readability. It’s much easier to understand the purpose of startEngine() than setValue("engineEnabled", true).

**When Not to Use**

* Don’t replace a parameter with explicit methods if a method is rarely changed and new variants aren’t added inside it.

**How to Refactor**

1. For each variant of the method, create a separate method. Run these methods based on the value of a parameter in the main method.
2. Find all places where the original method is called. In these places, place a call for one of the new parameter-dependent variants.
3. When no calls to the original method remain, delete it.

**Preserve Whole Object**

**Problem**

You get several values from an object and then pass them as parameters to a method.

**int** low = daysTempRange.getLow();

**int** high = daysTempRange.getHigh();

**boolean** withinPlan = plan.withinRange(low, high);

**Solution**

Instead, try passing the whole object.

**boolean** withinPlan = plan.withinRange(daysTempRange);

**Why Refactor**

The problem is that each time before your method is called, the methods of the future parameter object must be called. If these methods or the quantity of data obtained for the method are changed, you will need to carefully find a dozen such places in the program and implement these changes in each of them.

After you apply this refactoring technique, the code for getting all necessary data will be stored in one place—the method itself.

**Benefits**

* Instead of a hodgepodge of parameters, you see a single object with a comprehensible name.
* If the method needs more data from an object, you won’t need to rewrite all the places where the method is used—merely inside the method itself.

**Drawbacks**

* Sometimes this transformation causes a method to become less flexible: previously the method could get data from many different sources but now, because of refactoring, we’re limiting its use to only objects with a particular interface.

**How to Refactor**

1. Create a parameter in the method for the object from which you can get the necessary values.
2. Now start removing the old parameters from the method one by one, replacing them with calls to the relevant methods of the parameter object. Test the program after each replacement of a parameter.
3. Delete the getter code from the parameter object that had preceded the method call.

**Replace Parameter with Method Call**

**Problem**

Calling a query method and passing its results as the parameters of another method, while that method could call the query directly.

**int** basePrice = quantity \* itemPrice;

**double** seasonDiscount = **this**.getSeasonalDiscount();

**double** fees = **this**.getFees();

**double** finalPrice = discountedPrice(basePrice, seasonDiscount, fees);

**Solution**

Instead of passing the value through a parameter, try placing a query call inside the method body.

**int** basePrice = quantity \* itemPrice;

**double** finalPrice = discountedPrice(basePrice);

**Why Refactor**

A long list of parameters is hard to understand. In addition, calls to such methods often resemble a series of cascades, with winding and exhilarating value calculations that are hard to navigate yet have to be passed to the method. So if a parameter value can be calculated with the help of a method, do this inside the method itself and get rid of the parameter.

**Benefits**

* We get rid of unneeded parameters and simplify method calls. Such parameters are often created not for the project as it’s now, but with an eye for future needs that may never come.

**Drawbacks**

* You may need the parameter tomorrow for other needs... making you rewrite the method.

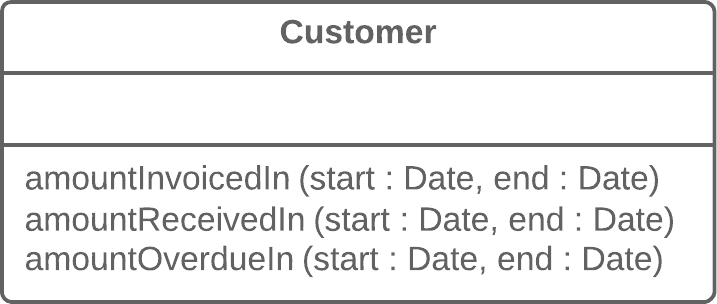
**How to Refactor**

1. Make sure that the value-getting code doesn’t use parameters from the current method, since they’ll be unavailable from inside another method. If so, moving the code isn’t possible.
2. If the relevant code is more complicated than a single method or function call, use [**Extract Method**](https://refactoring.guru/extract-method) to isolate this code in a new method and make the call simple.
3. In the code of the main method, replace all references to the parameter being replaced with calls to the method that gets the value.
4. Use [**Remove Parameter**](https://refactoring.guru/remove-parameter) to eliminate the now-unused parameter.

**Introduce Parameter Object**

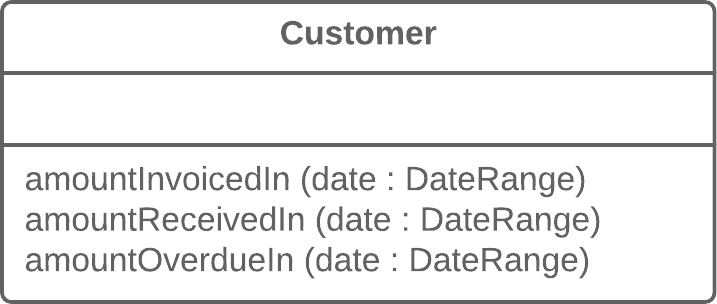
**Problem**

Your methods contain a repeating group of parameters.



**Solution**

Replace these parameters with an object.



**Why Refactor**

Identical groups of parameters are often encountered in multiple methods. This causes code duplication of both the parameters themselves and of related operations. By consolidating parameters in a single class, you can also move the methods for handling this data there as well, freeing the other methods from this code.

**Benefits**

* More readable code. Instead of a hodgepodge of parameters, you see a single object with a comprehensible name.
* Identical groups of parameters scattered here and there create their own kind of code duplication: while identical code isn’t being called, identical groups of parameters and arguments are constantly encountered.

**Drawbacks**

* If you move only data to a new class and don’t plan to move any behaviors or related operations there, this begins to smell of a [**Data Class**](https://refactoring.guru/smells/data-class).

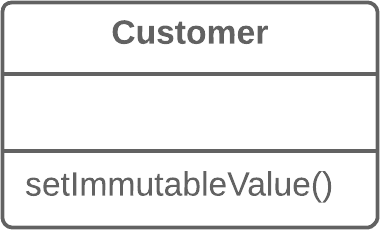
**How to Refactor**

1. Create a new class that will represent your group of parameters. Make the class immutable.
2. In the method that you want to refactor, use [**Add Parameter**](https://refactoring.guru/add-parameter), which is where your parameter object will be passed. In all method calls, pass the object created from old method parameters to this parameter.
3. Now start deleting old parameters from the method one by one, replacing them in the code with fields of the parameter object. Test the program after each parameter replacement.
4. When done, see whether there’s any point in moving a part of the method (or sometimes even the whole method) to a parameter object class. If so, use [**Move Method**](https://refactoring.guru/move-method) or [**Extract Method**](https://refactoring.guru/extract-method).

**Remove Setting Method**

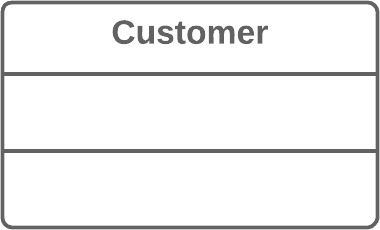
**Problem**

The value of a field should be set only when it’s created, and not change at any time after that.



**Solution**

So remove methods that set the field’s value.



**Why Refactor**

You want to prevent any changes to the value of a field.

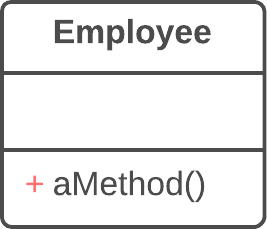
**How to Refactor**

1. The value of a field should be changeable only in the constructor. If the constructor doesn’t contain a parameter for setting the value, add one.
2. Find all setter calls.
   * If a setter call is located right after a call for the constructor of the current class, move its argument to the constructor call and remove the setter.
   * Replace setter calls in the constructor with direct access to the field.
3. Delete the setter.

**Hide Method**

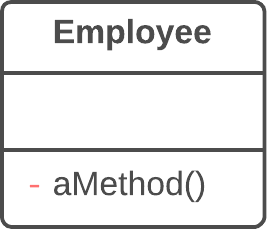
**Problem**

A method isn’t used by other classes or is used only inside its own class hierarchy.



**Solution**

Make the method private or protected.



**Why Refactor**

Quite often, the need to hide methods for getting and setting values is due to development of a richer interface that provides additional behavior, especially if you started with a class that added little beyond mere data encapsulation.

As new behavior is built into the class, you may find that public getter and setter methods are no longer necessary and can be hidden. If you make getter or setter methods private and apply direct access to variables, you can delete the method.

**Benefits**

* Hiding methods makes it easier for your code to evolve. When you change a private method, you only need to worry about how to not break the current class since you know that the method can’t be used anywhere else.
* By making methods private, you underscore the importance of the public interface of the class and of the methods that remain public.

**How to Refactor**

1. Regularly try to find methods that can be made private. Static code analysis and good unit test coverage can offer a big leg up.
2. Make each method as private as possible.

**Replace Constructor with Factory Method**

**Problem**

You have a complex constructor that does something more than just setting parameter values in object fields.

**class** **Employee** {

Employee(**int** type) {

**this**.type = type;

}

// ...

}

**Solution**

Create a factory method and use it to replace constructor calls.

**class** **Employee** {

**static** Employee create(**int** type) {

employee = **new** Employee(type);

// do some heavy lifting.

**return** employee;

}

// ...

}

**Why Refactor**

The most obvious reason for using this refactoring technique is related to [**Replace Type Code with Subclasses**](https://refactoring.guru/replace-type-code-with-subclasses).

You have code in which a object was previously created and the value of the coded type was passed to it. After use of the refactoring method, several subclasses have appeared and from them you need to create objects depending on the value of the coded type. Changing the original constructor to make it return subclass objects is impossible, so instead we create a static factory method that will return objects of the necessary classes, after which it replaces all calls to the original constructor.

Factory methods can be used in other situations as well, when constructors aren’t up to the task. They can be important when attempting to [**Change Value to Reference**](https://refactoring.guru/change-value-to-reference). They can also be used to set various creation modes that go beyond the number and types of parameters.

**Benefits**

* A factory method doesn’t necessarily return an object of the class in which it was called. Often these could be its subclasses, selected based on the arguments given to the method.
* A factory method can have a better name that describes what and how it returns what it does, for example Troops::GetCrew(myTank).
* A factory method can return an already created object, unlike a constructor, which always creates a new instance.

**How to Refactor**

1. Create a factory method. Place a call to the current constructor in it.
2. Replace all constructor calls with calls to the factory method.
3. Declare the constructor private.
4. Investigate the constructor code and try to isolate the code not directly related to constructing an object of the current class, moving such code to the factory method.

**Replace Error Code with Exception**

**Problem**

A method returns a special value that indicates an error?

**int** **withdraw**(**int** amount) {

**if** (amount > \_balance) {

**return** -1;

}

**else** {

balance -= amount;

**return** 0;

}

}

**Solution**

Throw an exception instead.

**void** **withdraw**(**int** amount) **throws** BalanceException {

**if** (amount > \_balance) {

**throw** **new** BalanceException();

}

balance -= amount;

}

**Why Refactor**

Returning error codes is an obsolete holdover from procedural programming. In modern programming, error handling is performed by special classes, which are named exceptions. If a problem occurs, you “throw” an error, which is then “caught” by one of the exception handlers. Special error-handling code, which is ignored in normal conditions, is activated to respond.

**Benefits**

* Frees code from a large number of conditionals for checking various error codes. Exception handlers are a much more succinct way to differentiate normal execution paths from abnormal ones.
* Exception classes can implement their own methods, thus containing part of the error handling functionality (such as for sending error messages).
* Unlike exceptions, error codes can’t be used in a constructor, since a constructor must return only a new object.

**Drawbacks**

* An exception handler can turn into a goto-like crutch. Avoid this! Don’t use exceptions to manage code execution. Exceptions should be thrown only to inform of an error or critical situation.

**How to Refactor**

Try to perform these refactoring steps for only one error code at a time. This will make it easier to keep all the important information in your head and avoid errors.

1. Find all calls to a method that returns error codes and, instead of checking for an error code, wrap it in try/catch blocks.
2. Inside the method, instead of returning an error code, throw an exception.
3. Change the method signature so that it contains information about the exception being thrown (@throws section).

**Replace Exception with Test**

**Problem**

You throw an exception in a place where a simple test would do the job?

**double** **getValueForPeriod**(**int** periodNumber) {

**try** {

**return** values[periodNumber];

} **catch** (ArrayIndexOutOfBoundsException e) {

**return** 0;

}

}

**Solution**

Replace the exception with a condition test.

**double** **getValueForPeriod**(**int** periodNumber) {

**if** (periodNumber >= values.length) {

**return** 0;

}

**return** values[periodNumber];

}

**Why Refactor**

Exceptions should be used to handle irregular behavior related to an unexpected error. They shouldn’t serve as a replacement for testing. If an exception can be avoided by simply verifying a condition before running, then do so. Exceptions should be reserved for real errors.

For instance, you entered a minefield and triggered a mine there, resulting in an exception; the exception was successfully handled and you were lifted through the air to safety beyond the mine field. But you could have avoided this all by simply reading the warning sign in front of the minefield to begin with.

**Benefits**

* A simple conditional can sometimes be more obvious than exception handling code.

**How to Refactor**

1. Create a conditional for an edge case and move it before the try/catch block.
2. Move code from the catch section inside this conditional.
3. In the catch section, place the code for throwing a usual unnamed exception and run all the tests.
4. If no exceptions were thrown during the tests, get rid of the try/catch operator.