**Self Encapsulate Field**

Self-encapsulation is distinct from ordinary [**Encapsulate Field**](https://refactoring.guru/encapsulate-field): the refactoring technique given here is performed on a private field.

**Problem**

You use direct access to private fields inside a class.

**class** **Range** {

**private** **int** low, high;

**boolean** includes(**int** arg) {

**return** arg >= low && arg <= high;

}

}

**Solution**

Create a getter and setter for the field, and use only them for accessing the field.

**class** **Range** {

**private** **int** low, high;

**boolean** includes(**int** arg) {

**return** arg >= getLow() && arg <= getHigh();

}

**int** getLow() {

**return** low;

}

**int** getHigh() {

**return** high;

}

}

**Why Refactor**

Sometimes directly accessing a private field inside a class just isn’t flexible enough. You want to be able to initiate a field value when the first query is made or perform certain operations on new values of the field when they’re assigned, or maybe do all this in various ways in subclasses.

**Benefits**

* *Indirect access to fields* is when a field is acted on via access methods (getters and setters). This approach is much more flexible than *direct access to fields*.
  + First, you can perform complex operations when data in the field is set or received. *Lazy initialization* and *validation of field values* are easily implemented inside field getters and setters.
  + Second and more crucially, you can redefine getters and setters in subclasses.
* You have the option of not implementing a setter for a field at all. The field value will be specified only in the constructor, thus making the field unchangeable throughout the entire object lifespan.

**Drawbacks**

* When *direct access to fields* is used, code looks simpler and more presentable, although flexibility is diminished.

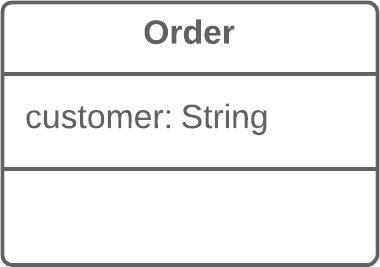
**How to Refactor**

1. Create a getter (and optional setter) for the field. They should be either protected or public.
2. Find all direct invocations of the field and replace them with getter and setter calls.

**Replace Data Value with Object**

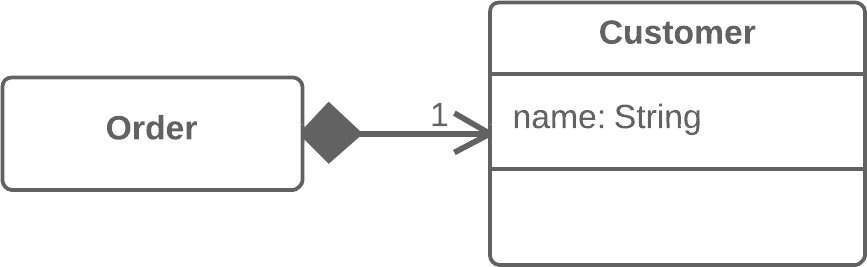
**Problem**

A class (or group of classes) contains a data field. The field has its own behavior and associated data.



**Solution**

Create a new class, place the old field and its behavior in the class, and store the object of the class in the original class.



**Why Refactor**

This refactoring is basically a special case of [**Extract Class**](https://refactoring.guru/extract-class). What makes it different is the cause of the refactoring.

In [**Extract Class**](https://refactoring.guru/extract-class), we have a single class that’s responsible for different things and we want to split up its responsibilities.

With replacement of a data value with an object, we have a primitive field (number, string, etc.) that’s no longer so simple due to growth of the program and now has associated data and behaviors. On the one hand, there’s nothing scary about these fields in and of themselves. However, this fields-and-behaviors family can be present in several classes simultaneously, creating duplicate code.

Therefore, for all this we create a new class and move both the field and the related data and behaviors to it.

**Benefits**

* Improves relatedness inside classes. Data and the relevant behaviors are inside a single class.

**How to Refactor**

Before you begin with refactoring, see if there are direct references to the field from within the class. If so, use [**Self Encapsulate Field**](https://refactoring.guru/self-encapsulate-field) in order to hide it in the original class.

1. Create a new class and copy your field and relevant getter to it. In addition, create a constructor that accepts the simple value of the field. This class won’t have a setter since each new field value that’s sent to the original class will create a new value object.
2. In the original class, change the field type to the new class.
3. In the getter in the original class, invoke the getter of the associated object.
4. In the setter, create a new value object. You may need to also create a new object in the constructor if initial values had been set there for the field previously.

**Next Steps**

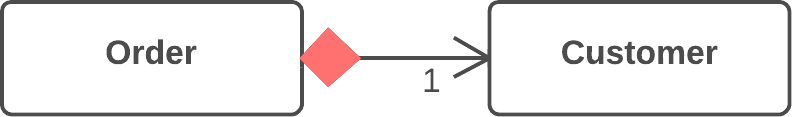
After applying this refactoring technique, it’s wise to apply [**Change Value to Reference**](https://refactoring.guru/change-value-to-reference) on the field that contains the object. This allows storing a reference to a single object that corresponds to a value instead of storing dozens of objects for one and the same value.

Most often this approach is needed when you want to have one object be responsible for one real-world object (such as users, orders, documents and so forth). At the same time, this approach won’t be useful for objects such as dates, money, ranges, etc.

**Change Value to Reference**

**Problem**

So you have many identical instances of a single class that you need to replace with a single object.



**Solution**

Convert the identical objects to a single reference object.



**Why Refactor**

In many systems, objects can be classified as either values or references.

* **References**: when one real-world object corresponds to only one object in the program. References are usually user/order/product/etc. objects.
* **Values**: one real-world object corresponds to multiple objects in the program. These objects could be dates, phone numbers, addresses, colors, and the like.

The selection of reference vs. value isn’t always clear-cut. Sometimes there’s a simple value with a small amount of unchanging data. Then it becomes necessary to add changeable data and pass these changes every time the object is accessed. In this case it becomes necessary to convert it to a reference.

**Benefits**

* An object contains all the most current information about a particular entity. If the object is changed in one part of the program, these changes are accessible from the other parts of the program that make use of the object.

**Drawbacks**

* References are much harder to implement.

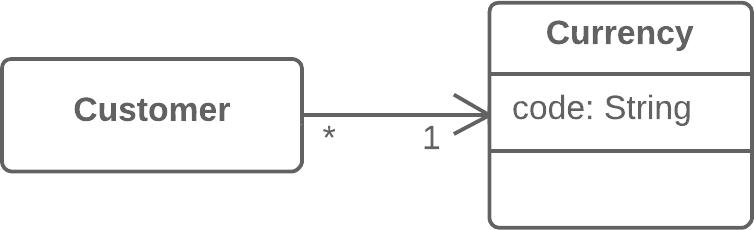
**How to Refactor**

1. Use [**Replace Constructor with Factory Method**](https://refactoring.guru/replace-constructor-with-factory-method) on the class from which the references are to be generated.
2. Determine which object will be responsible for providing access to references. Instead of creating a new object, when you need one you now need to get it from a storage object or static dictionary field.
3. Determine whether references will be created in advance or dynamically as necessary. If objects are created in advance, make sure to load them before use.
4. Change the factory method so that it returns a reference. If objects are created in advance, decide how to handle errors when a non-existent object is requested. You may also need to use [**Rename Method**](https://refactoring.guru/rename-method) to inform that the method returns only existing objects.

**Change Reference to Value**

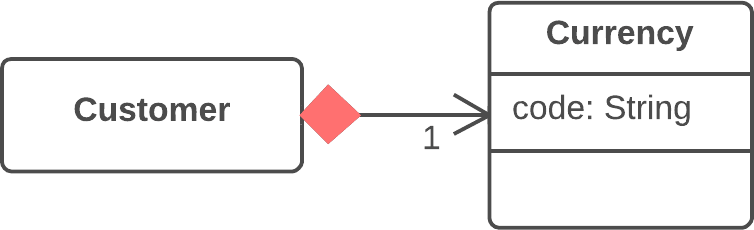
**Problem**

You have a reference object that’s too small and infrequently changed to justify managing its life cycle.



**Solution**

Turn it into a value object.



**Why Refactor**

Inspiration to switch from a reference to a value may come from the inconvenience of working with the reference. References require management on your part:

* They always require requesting the necessary object from storage.
* References in memory may be inconvenient to work with.
* Working with references is particularly difficult, compared to values, on distributed and parallel systems.

Values are especially useful if you would rather have unchangeable objects than objects whose state may change during their lifetime.

**Benefits**

* One important property of objects is that they should be unchangeable. The same result should be received for each query that returns an object value. If this is true, no problems arise if there are many objects representing the same thing.
* Values are much easier to implement.

**Drawbacks**

* If a value is changeable, make sure if any object changes that the values in all the other objects representing the same entity are updated. This is so burdensome that it’s easier to create a reference for this purpose.

**How to Refactor**

1. Make the object unchangeable. The object shouldn’t have any setters or other methods that change its state and data ([**Remove Setting Method**](https://refactoring.guru/remove-setting-method) may help here). The only place where data should be assigned to the fields of a value object is a constructor.
2. Create a comparison method to be able to compare two values.
3. Check whether you can delete the factory method and make the object constructor public.

**Replace Array with Object**

This refactoring technique is a special case of [**Replace Data Value with Object**](https://refactoring.guru/replace-data-value-with-object).

**Problem**

You have an array that contains various types of data.

**String**[] row = **new** **String**[2];

row[0] = "Liverpool";

row[1] = "15";

**Solution**

Replace the array with an object that will have separate fields for each element.

Performance row = **new** Performance();

row.setName("Liverpool");

row.setWins("15");

**Why Refactor**

Arrays are an excellent tool for storing data and collections of a single type. But if you use an array like post office boxes, storing the username in box 1 and the user’s address in box 14, you will someday be very unhappy that you did. This approach leads to catastrophic failures when somebody puts something in the wrong “box” and also requires your time for figuring out which data is stored where.

**Benefits**

* In the resulting class, you can place all associated behaviors that had been previously stored in the main class or elsewhere.
* The fields of a class are much easier to document than the elements of an array.

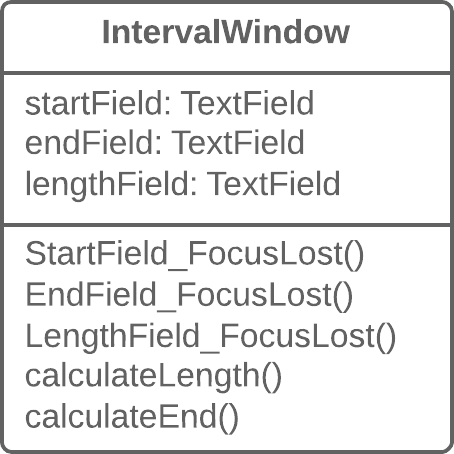
**How to Refactor**

1. Create the new class that will contain the data from the array. Place the array itself in the class as a public field.
2. Create a field for storing the object of this class in the original class. Don’t forget to also create the object itself in the place where you initiated the data array.
3. In the new class, create access methods one by one for each of the array elements. Give them self-explanatory names that indicate what they do. At the same time, replace each use of an array element in the main code with the corresponding access method.
4. When access methods have been created for all elements, make the array private.
5. For each element of the array, create a private field in the class and then change the access methods so that they use this field instead of the array.
6. When all data has been moved, delete the array.

**Duplicate Observed Data**

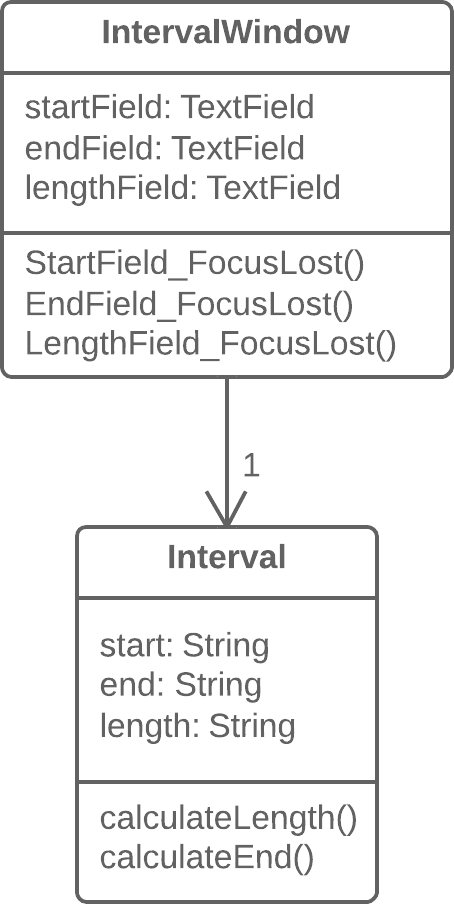
**Problem**

Is domain data stored in classes responsible for the GUI?



**Solution**

Then it’s a good idea to separate the data into separate classes, ensuring connection and synchronization between the domain class and the GUI.



**Why Refactor**

You want to have multiple interface views for the same data (for example, you have both a desktop app and a mobile app). If you fail to separate the GUI from the domain, you will have a very hard time avoiding code duplication and a large number of mistakes.

**Benefits**

* You split responsibility between business logic classes and presentation classes (cf. the *Single Responsibility Principle*), which makes your program more readable and understandable.
* If you need to add a new interface view, create new presentation classes; you don’t need to touch the code of the business logic (cf. the *Open/Closed Principle*).
* Now different people can work on the business logic and the user interfaces.

**When Not to Use**

* This refactoring technique, which in its classic form is performed using the [**Observer**](https://refactoring.guru/design-patterns/observer) template, isn’t applicable for web apps, where all classes are recreated between queries to the web server.
* All the same, the general principle of extracting business logic into separate classes can be justified for web apps as well. But this will be implemented using different refactoring techniques depending on how your system is designed.

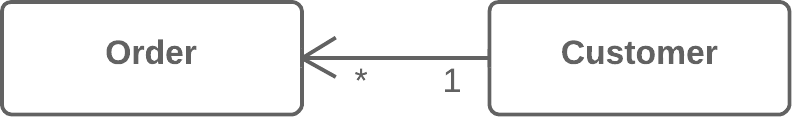
**How to Refactor**

1. Hide direct access to domain data in the *GUI class*. For this, it’s best to use [**Self Encapsulate Field**](https://refactoring.guru/self-encapsulate-field). So you create the getters and setters for this data.
2. In handlers for *GUI class* events, use setters to set new field values. This will let you pass these values to the associated *domain object*.
3. Create a domain class and copy necessary fields from the *GUI class* to it. Create getters and seters for all these fields.
4. Create an Observer pattern for these two classes:
   * In the *domain class*, create an array for storing observer objects (*GUI objects*), as well as methods for registering, deleting and notifying them.
   * In the *GUI class*, create a field for storing references to the *domain class* as well as the update() method, which will be reacting to changes in the object and update the values of fields in the *GUI class*. Note that value updates should be established directly in the method, in order to avoid recursion.
   * In the *GUI class* constructor, create an instance of *domain class* and save it in the field you have created. Register the *GUI object* as an observer in the *domain object*.
   * In the setters for *domain class* fields, call the method for notifying the observer (in other words, method for updating in the *GUI class*), in order to pass the new values to the GUI.
   * Change the setters of the *GUI class* fields so that they set new values in the domain object directly. Watch out to make sure that values aren’t set through a *domain class* setter—otherwise infinite recursion will result.

**Change Unidirectional Association to Bidirectional**

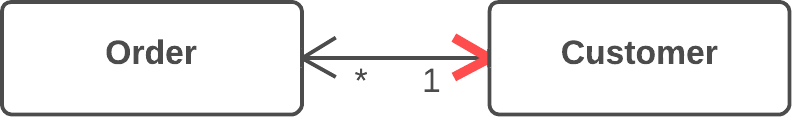
**Problem**

You have two classes that each need to use the features of the other, but the association between them is only unidirectional.



**Solution**

Add the missing association to the class that needs it.



**Why Refactor**

Originally the classes had a unidirectional association. But with time, client code needed access to both sides of the association.

**Benefits**

* If a class needs a reverse association, you can simply calculate it. But if these calculations are complex, it’s better to keep the reverse association.

**Drawbacks**

* Bidirectional associations are much harder to implement and maintain than unidirectional ones.
* Bidirectional associations make classes interdependent. With a unidirectional association, one of them can be used independently of the other.

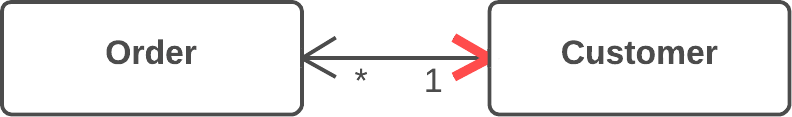
**How to Refactor**

1. Add a field for holding the reverse association.
2. Decide which class will be “dominant”. This class will contain the methods that create or update the association as elements are added or changed, establishing the association in its class and calling the utility methods for establishing the association in the associated object.
3. Create a utility method for establishing the association in the “non-dominant” class. The method should use what it’s given in parameters to complete the field. Give the method an obvious name so that it isn’t used later for any other purposes.
4. If old methods for controlling the unidirectional association were in the “dominant” class, complement them with calls to utility methods from the associated object.
5. If the old methods for controlling the association were in the “non-dominant” class, create the methods in the “dominant” class, call them, and delegate execution to them.

**Change Bidirectional Association to Unidirectional**

**Problem**

You have a bidirectional association between classes, but one of the classes doesn’t use the other’s features.



**Solution**

Remove the unused association.



**Why Refactor**

A bidirectional association is generally harder to maintain than a unidirectional one, requiring additional code for properly creating and deleting the relevant objects. This makes the program more complicated.

In addition, an improperly implemented bidirectional association can cause problems for garbage collection (in turn leading to memory bloat by unused objects).

Example: the garbage collector removes objects from memory that are no longer referenced by other objects. Let’s say that an object pair User-Order was created, used, and then abandoned. But these objects won’t be cleared from memory since they still refer to each other. That said, this problem is becoming less important thanks to advances in programming languages, which now automatically identify unused object references and remove them from memory.

There’s also the problem of interdependency between classes. In a bidirectional association, the two classes must know about each other, meaning that they can’t be used separately. If many of these associations are present, different parts of the program become too dependent on each other and any changes in one component may affect the other components.

**Benefits**

* Simplifies the class that doesn’t need the relationship. Less code equals less code maintenance.
* Reduces dependency between classes. Independent classes are easier to maintain since any changes to a class affect only that class.

**How to Refactor**

1. Make sure that one of the following is true for your classes:
   * No association is used.
   * There’s another way to get the associated object, such through a database query.
   * The associated object can be passed as an argument to the methods that use it.
2. Depending on your situation, use of a field that contains an association with another object should be replaced by a parameter or method call for getting the object in a different way.
3. Delete the code that assigns the associated object to the field.
4. Delete the now-unused field.

# Replace Magic Number with Symbolic Constant

### Problem

Your code uses a number that has a certain meaning to it.

**double** **potentialEnergy**(**double** mass, **double** height) {

**return** mass \* height \* 9.81;

}

### Solution

Replace this number with a constant that has a human-readable name explaining the meaning of the number.

**static** **final** **double** GRAVITATIONAL\_CONSTANT = 9.81;

**double** **potentialEnergy**(**double** mass, **double** height) {

**return** mass \* height \* GRAVITATIONAL\_CONSTANT;

}

### Why Refactor

A magic number is a numeric value that’s encountered in the source but has no obvious meaning. This “anti-pattern” makes it harder to understand the program and refactor the code.

Yet more difficulties arise when you need to change this magic number. Find and replace won’t work for this: the same number may be used for different purposes in different places, meaning that you will have to verify every line of code that uses this number.

### Benefits

* The symbolic constant can serve as live documentation of the meaning of its value.
* It’s much easier to change the value of a constant than to search for this number throughout the entire codebase, without the risk of accidentally changing the same number used elsewhere for a different purpose.
* Reduce duplicate use of a number or string in the code. This is especially important when the value is complicated and long (such as 3.14159 or 0xCAFEBABE).

### Good to Know

#### Not all numbers are magical.

If the purpose of a number is obvious, there’s no need to replace it. A classic example is:

for (i = 0; i < сount; i++) { ... }

#### Alternatives

1. Sometimes a magic number can be replaced with method calls. For example, if you have a magic number that signifies the number of elements in a collection, you don’t need to use it for checking the last element of the collection. Instead, use the standard method for getting the collection length.
2. Magic numbers are sometimes used as type code. Say that you have two types of users and you use a number field in a class to specify which is which: administrators are 1 and ordinary users are 2.

In this case, you should use one of the refactoring methods to avoid type code:

* + [**Replace Type Code with Class**](https://refactoring.guru/replace-type-code-with-class)
  + [**Replace Type Code with Subclasses**](https://refactoring.guru/replace-type-code-with-subclasses)
  + [**Replace Type Code with State/Strategy**](https://refactoring.guru/replace-type-code-with-state-strategy)

### How to Refactor

1. Declare a constant and assign the value of the magic number to it.
2. Find all mentions of the magic number.
3. For each of the numbers that you find, double-check that the magic number in this particular case corresponds to the purpose of the constant. If yes, replace the number with your constant. This is an important step, since the same number can mean absolutely different things (and replaced with different constants, as the case may be).

# Encapsulate Field

### Problem

You have a public field.

**class** **Person** {

**public** **String** name;

}

### Solution

Make the field private and create access methods for it.

**class** **Person** {

**private** **String** name;

**public** **String** getName() {

**return** name;

}

**public** **void** setName(**String** arg) {

name = arg;

}

}

### Why Refactor

One of the pillars of object-oriented programming is Encapsulation, the ability to conceal object data. Otherwise, all objects would be public and other objects could get and modify the data of your object without any checks and balances! Data is separated from the behaviors associated with this data, modularity of program sections is compromised, and maintenance becomes complicated.

### Benefits

* If the data and behavior of a component are closely interrelated and are in the same place in the code, it’s much easier for you to maintain and develop this component.
* You can also perform complicated operations related to access to object fields.

### When Not to Use

* In some cases, encapsulation is ill-advised due to performance considerations. These cases are rare but when they happen, this circumstance is very important.

Say that you have a graphical editor that contains objects possessing x- and y-coordinates. These fields are unlikely to change in the future. What’s more, the program involves a great many different objects in which these fields are present. So accessing the coordinate fields directly saves significant CPU cycles that would otherwise be taken up by calling access methods.

As an example of this unusual case, there’s the [**Point**](http://docs.oracle.com/javase/7/docs/api/java/awt/Point.html) class in Java. All fields of this class are public.

### How to Refactor

1. Create a getter and setter for the field.
2. Find all invocations of the field. Replace receipt of the field value with the getter, and replace setting of new field values with the setter.
3. After all field invocations have been replaced, make the field private.

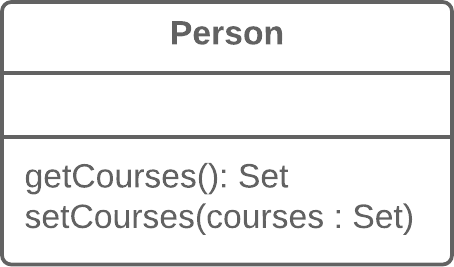
#### Next Steps

Encapsulate Field is only the first step in bringing data and the behaviors involving this data closer together. After you create simple methods for access fields, you should recheck the places where these methods are called. It’s quite possible that the code in these areas would look more appropriate in the access methods.

**Encapsulate Collection**

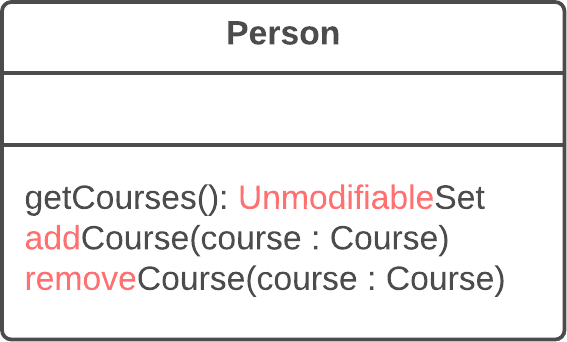
**Problem**

A class contains a collection field and a simple getter and setter for working with the collection.



**Solution**

Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.



**Why Refactor**

A class contains a field that contains a collection of objects. This collection could be an array, list, set or vector. A normal getter and setter have been created for working with the collection.

But the collections should be used by a protocol that’s a bit different from the one used by other data types. The getter method shouldn’t return the collection object itself, since this would let clients change collection contents without the knowledge of the owner class. In addition, this would show too much of the internal structures of the object data to clients. The method for getting collection elements should return a value that doesn’t allow changing the collection or disclose excessive data about its structure.

In addition, there shouldn’t be a method that assigns a value to the collection. Instead, there should be operations for adding and deleting elements. Thanks to this, the owner object gains control over addition and deletion of collection elements.

Such a protocol properly encapsulates a collection, which ultimately reduces the degree of association between the owner class and the client code.

**Benefits**

* The collection field is encapsulated inside a class. When the getter is called, it returns a copy of the collection, which prevents accidental changing or overwriting of the collection elements without the knowledge of the class that contains the collection.
* If collection elements are contained inside a primitive type, such as an array, you create more convenient methods for working with the collection.
* If collection elements are contained inside a non-primitive container (standard collection class), by encapsulating the collection you can restrict access to unwanted standard methods of the collection (such as by restricting addition of new elements).

**How to Refactor**

1. Create methods for adding and deleting collection elements. They must accept collection elements in their parameters.
2. Assign an empty collection to the field as the initial value if this isn’t done in the class constructor.
3. Find the calls of the collection field setter. Change the setter so that it uses operations for adding and deleting elements, or make these operations call client code.

Note that setters can be used only to replace all collection elements with other ones. Therefore it may be advisable to change the setter name ([**Rename Method**](https://refactoring.guru/rename-method)) to replace.

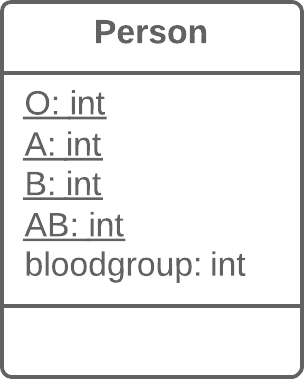
1. Find all calls of the collection getter after which the collection is changed. Change the code so that it uses your new methods for adding and deleting elements from the collection.
2. Change the getter so that it returns a read-only representation of the collection.
3. Inspect the client code that uses the collection for code that would look better inside of the collection class itself.

**Replace Type Code with Class**

**What’s type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

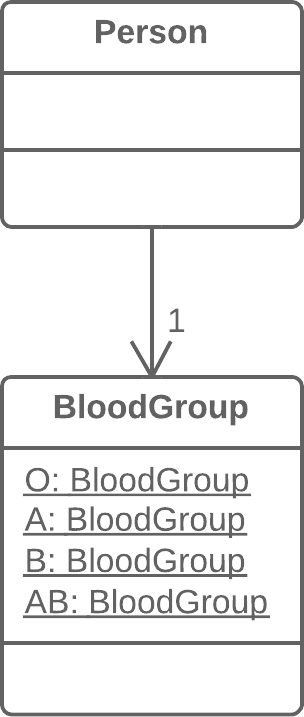
**Problem**

A class has a field that contains type code. The values of this type aren’t used in operator conditions and don’t affect the behavior of the program.



**Solution**

Create a new class and use its objects instead of the type code values.



**Why Refactor**

One of the most common reasons for type code is working with databases, when a database has fields in which some complex concept is coded with a number or string.

For example, you have the class User with the field user\_role, which contains information about the access privileges of each user, whether administrator, editor, or ordinary user. So in this case, this information is coded in the field as A, E, and U respectively.

What are the shortcomings of this approach? The field setters often don’t check which value is sent, which can cause big problems when someone sends unintended or wrong values to these fields.

In addition, type verification is impossible for these fields. It’s possible to send any number or string to them, which won’t be type checked by your IDE and even allow your program to run (and crash later).

**Benefits**

* We want to turn sets of primitive values—which is what coded types are—into full-fledged classes with all the benefits that object-oriented programming has to offer.
* By replacing type code with classes, we allow type hinting for values passed to methods and fields at the level of the programming language.

For example, while the compiler previously didn’t see difference between your numeric constant and some arbitrary number when a value is passed to a method, now when data that doesn’t fit the indicated type class is passed, you’re warned of the error inside your IDE.

* Thus we make it possible to move code to the classes of the type. If you needed to perform complex manipulations with type values throughout the whole program, now this code can “live” inside one or multiple type classes.

**When Not to Use**

If the values of a coded type are used inside control flow structures (if, switch, etc.) and control a class behavior, you should use one of the two refactoring techniques for type code:

* [**Replace Type Code with Subclasses**](https://refactoring.guru/replace-type-code-with-subclasses)
* [**Replace Type Code with State/Strategy**](https://refactoring.guru/replace-type-code-with-state-strategy)

**How to Refactor**

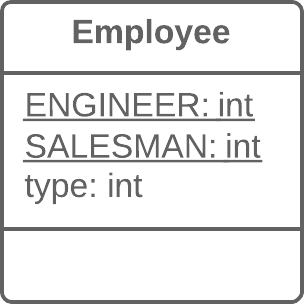
1. Create a new class and give it a new name that corresponds to the purpose of the coded type. Here we’ll call it *type class*.
2. Copy the field containing type code to the *type class* and make it private. Then create a getter for the field. A value will be set for this field only from the constructor.
3. For each value of the coded type, create a static method in *type class*. It’ll be creating a new *type class* object corresponding to this value of the coded type.
4. In the original class, replace the type of the coded field with *type class*. Create a new object of this type in the constructor as well as in the field setter. Change the field getter so that it calls the *type class* getter.
5. Replace any mentions of values of the coded type with calls of the relevant *type class* static methods.
6. Remove the coded type constants from the original class.

**Replace Type Code with Subclasses**

**What’s type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

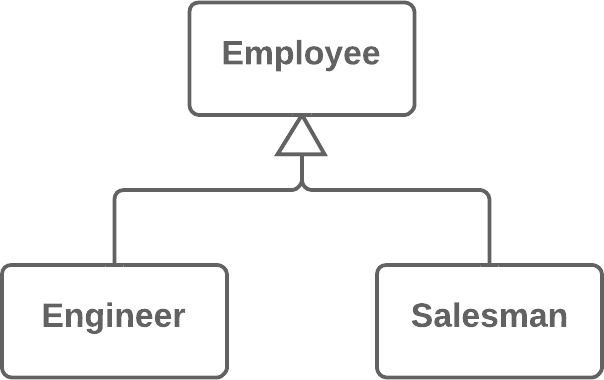
**Problem**

You have a coded type that directly affects program behavior (values of this field trigger various code in conditionals).



**Solution**

Create subclasses for each value of the coded type. Then extract the relevant behaviors from the original class to these subclasses. Replace the control flow code with polymorphism.



**Why Refactor**

This refactoring technique is a more complicated twist on [**Replace Type Code with Class**](https://refactoring.guru/replace-type-code-with-class).

As in the first refactoring method, you have a set of simple values that constitute all the allowed values for a field. Although these values are often specified as constants and have understandable names, their use makes your code very error-prone since they’re still primitives in effect. For example, you have a method that accepts one of these values in the parameters. At a certain moment, instead of the constant USER\_TYPE\_ADMIN with the value "ADMIN", the method receives the same string in lower case ("admin"), which will cause execution of something else that the author (you) didn’t intend.

Here we’re dealing with control flow code such as the conditionals if, switch and ?:. In other words, fields with coded values (such as $user->type === self::USER\_TYPE\_ADMIN) are used inside the conditions of these operators. If we were to use [**Replace Type Code with Class**](https://refactoring.guru/replace-type-code-with-class) here, all these control flow constructions would be best moved to a class responsible for the data type. Ultimately, this would of course create a type class very similar to the original one, with the same problems as well.

**Benefits**

* Delete the control flow code. Instead of a bulky switch in the original class, move the code to appropriate subclasses. This improves adherence to the *Single Responsibility Principle* and makes the program more readable in general.
* If you need to add a new value for a coded type, all you need to do is add a new subclass without touching the existing code (cf. the *Open/Closed Principle*).
* By replacing type code with classes, we pave the way for type hinting for methods and fields at the level of the programming language. This wouldn’t be possible using simple numeric or string values contained in a coded type.

**When Not to Use**

* This technique isn’t applicable if you already have a class hierarchy. You can’t create a dual hierarchy via inheritance in object-oriented programming. Still, you can replace type code via composition instead of inheritance. To do so, use [**Replace Type Code with State/Strategy**](https://refactoring.guru/replace-type-code-with-state-strategy).
* If the values of type code can change after an object is created, avoid this technique. We would have to somehow replace the class of the object itself on the fly, which isn’t possible. Still, an alternative in this case too would be [**Replace Type Code with State/Strategy**](https://refactoring.guru/replace-type-code-with-state-strategy).

**How to Refactor**

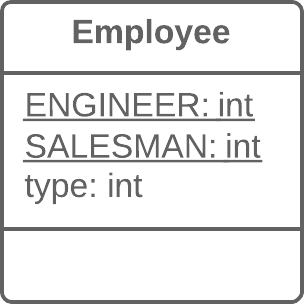
1. Use [**Self Encapsulate Field**](https://refactoring.guru/self-encapsulate-field) to create a getter for the field that contains type code.
2. Make the superclass constructor private. Create a static factory method with the same parameters as the superclass constructor. It must contain the parameter that will take the starting values of the coded type. Depending on this parameter, the factory method will create objects of various subclasses. To do so, in its code you must create a large conditional but, at least, it’ll be the only one when it’s truly necessary; otherwise, subclasses and polymorphism will do.
3. Create a unique subclass for each value of the coded type. In it, redefine the getter of the coded type so that it returns the corresponding value of the coded type.
4. Delete the field with type code from the superclass. Make its getter abstract.
5. Now that you have subclasses, you can start to move the fields and methods from the superclass to corresponding subclasses (with the help of [**Push Down Field**](https://refactoring.guru/push-down-field) and [**Push Down Method**](https://refactoring.guru/push-down-method)).
6. When everything possible has been moved, use [**Replace Conditional with Polymorphism**](https://refactoring.guru/replace-conditional-with-polymorphism) in order to get rid of conditions that use the type code once and for all.

**Replace Type Code with State/Strategy**

**What’s type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

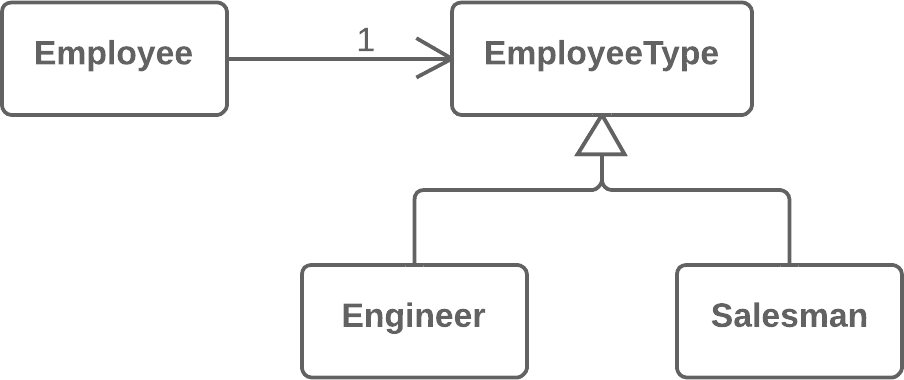
**Problem**

You have a coded type that affects behavior but you can’t use subclasses to get rid of it.



**Solution**

Replace type code with a state object. If it’s necessary to replace a field value with type code, another state object is “plugged in”.



**Why Refactor**

You have type code and it affects the behavior of a class, therefore we can’t use [**Replace Type Code with Class**](https://refactoring.guru/replace-type-code-with-class).

Type code affects the behavior of a class but we can’t create subclasses for the coded type due to the existing class hierarchy or other reasons. Thus means that we can’t apply [**Replace Type Code with Subclasses**](https://refactoring.guru/replace-type-code-with-subclasses).

**Benefits**

* This refactoring technique is a way out of situations when a field with a coded type changes its value during the object’s lifetime. In this case, replacement of the value is made via replacement of the state object to which the original class refers.
* If you need to add a new value of a coded type, all you need to do is to add a new state subclass without altering the existing code (cf. the *Open/Closed Principle*).

**Drawbacks**

* If you have a simple case of type code but you use this refactoring technique anyway, you will have many extra (and unneeded) classes.

**Good to Know**

Implementation of this refactoring technique can make use of one of two design patterns: **State** or **Strategy**. Implementation is the same no matter which pattern you choose. So which pattern should you pick in a particular situation?

If you’re trying to split a conditional that controls the selection of algorithms, use Strategy.

But if each value of the coded type is responsible not only for selecting an algorithm but for the whole condition of the class, class state, field values, and many other actions, State is better for the job.

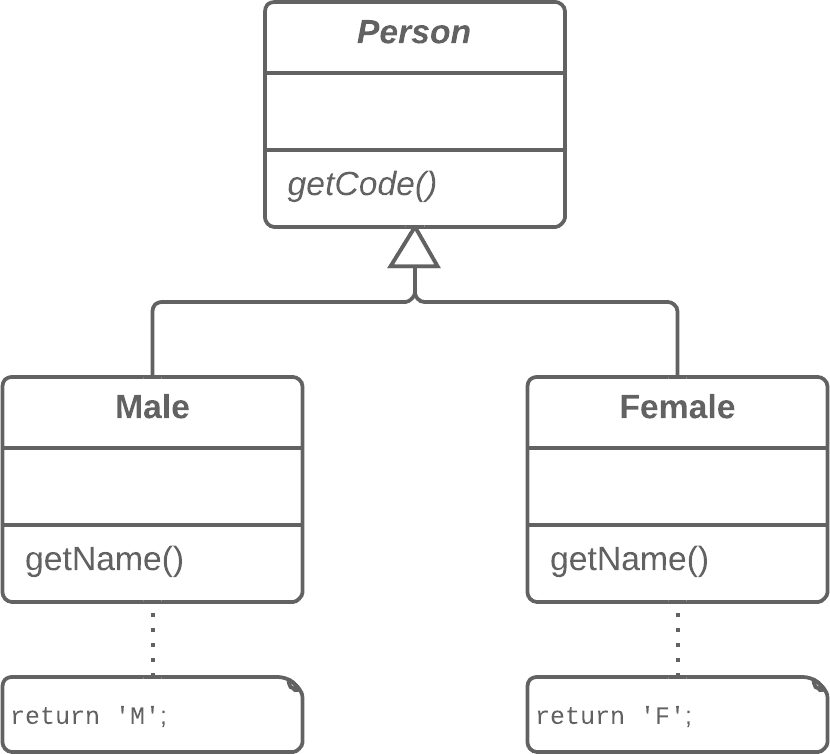
**How to Refactor**

1. Use [**Self Encapsulate Field**](https://refactoring.guru/self-encapsulate-field) to create a getter for the field that contains type code.
2. Create a new class and give it an understandable name that fits the purpose of the type code. This class will be playing the role of *state* (or *strategy*). In it, create an abstract coded field getter.
3. Create subclasses of the state class for each value of the coded type. In each subclass, redefine the getter of the coded field so that it returns the corresponding value of the coded type.
4. In the abstract state class, create a static factory method that accepts the value of the coded type as a parameter. Depending on this parameter, the factory method will create objects of various states. For this, in its code create a large conditional; it’ll be the only one when refactoring is complete.
5. In the original class, change the type of the coded field to the state class. In the field’s setter, call the factory state method for getting new state objects.
6. Now you can start to move the fields and methods from the superclass to the corresponding state subclasses (using [**Push Down Field**](https://refactoring.guru/push-down-field) and [**Push Down Method**](https://refactoring.guru/push-down-method)).
7. When everything moveable has been moved, use [**Replace Conditional with Polymorphism**](https://refactoring.guru/replace-conditional-with-polymorphism) in order to get rid of conditionals that use type code once and for all.

**Replace Subclass with Fields**

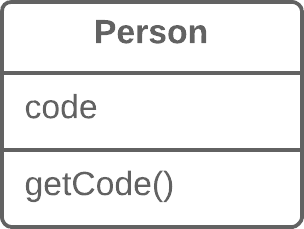
**Problem**

You have subclasses differing only in their (constant-returning) methods.



**Solution**

Replace the methods with fields in the parent class and delete the subclasses.



**Why Refactor**

Sometimes refactoring is just the ticket for avoiding type code.

In one such case, a hierarchy of subclasses may be different only in the values returned by particular methods. These methods aren’t even the result of computation, but are strictly set out in the methods themselves or in the fields returned by the methods. To simplify the class architecture, this hierarchy can be compressed into a single class containing one or several fields with the necessary values, based on the situation.

These changes may become necessary after moving a large amount of functionality from a class hierarchy to another place. The current hierarchy is no longer so valuable and its subclasses are now just dead weight.

**Benefits**

* Simplifies system architecture. Creating subclasses is overkill if all you want to do is to return different values in different methods.

**How to Refactor**

1. Apply [**Replace Constructor with Factory Method**](https://refactoring.guru/replace-constructor-with-factory-method) to the subclasses.
2. Replace subclass constructor calls with superclass factory method calls.
3. In the superclass, declare fields for storing the values of each of the subclass methods that return constant values.
4. Create a protected superclass constructor for initializing the new fields.
5. Create or modify the existing subclass constructors so that they call the new constructor of the parent class and pass the relevant values to it.
6. Implement each constant method in the parent class so that it returns the value of the corresponding field. Then remove the method from the subclass.
7. If the subclass constructor has additional functionality, use [**Inline Method**](https://refactoring.guru/inline-method) to incorporate the constructor into the superclass factory method.
8. Delete the subclass.