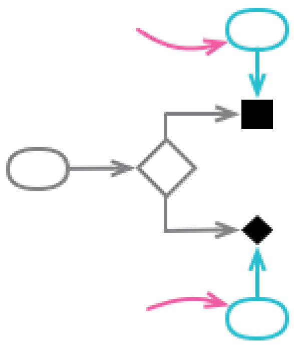
Remove Flag Argument



```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

formerly: Replace Parameter with Explicit Methods

Motivation

A flag argument is a function argument that the caller uses to indicate which logic the called function should execute. I may call a function that looks like this:

```
function bookConcert(aCustomer, isPremium) {
  if (isPremium) {
    // logic for premium booking
  } else {
    // logic for regular booking
  }
}
```

To book a premium concert, I issue the call like so:

```
bookConcert(aCustomer, true);
```

Flag arguments can also come as enums:

```
bookConcert(aCustomer, CustomerType.PREMIUM);
```

or strings (or symbols in languages that use them):

```
bookConcert(aCustomer, "premium");
```

I dislike flag arguments because they complicate the process of understanding what function calls are available and how to call them. My first route into an API is usually the list of available functions, and flag arguments hide the differences in the function calls that are available. Once I select a function, I have to figure out what values are available for the flag arguments. Boolean flags are even worse since they don't convey their meaning to the reader—in a function call, I can't figure out what `true` means. It's clearer to provide an explicit function for the task I want to do.

```
premiumBookConcert(aCustomer);
```

Not all arguments like this are flag arguments. To be a flag argument, the callers must be setting the boolean value to a literal value, not data that's flowing through the program. Also, the implementation function must be using the argument to influence its control flow, not as data that it passes to further functions.

Removing flag arguments doesn't just make the code clearer—it also helps my tooling. Code analysis tools can now more easily see the difference between calling the premium logic and calling regular logic.

Flag arguments can have a place if there's more than one of them in the function, since otherwise I would need explicit functions for every combination of their values. But that's also a signal of a function doing too much, and I should look for a way to create simpler functions that I can compose for this logic.

Mechanics

- Create an explicit function for each value of the parameter.
  If the main function has a clear dispatch conditional, use Decompose Conditional to create the explicit functions. Otherwise, create wrapping functions.
- For each caller that uses a literal value for the parameter, replace it with a call to the explicit function.

Example

Looking through some code, I see calls to calculate a delivery date for a shipment. Some of the calls look like

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

and some look like

```
aShipment.deliveryDate = deliveryDate(anOrder, false);
```

Faced with code like this, I immediately begin to wonder about the meaning of the boolean value. What is it doing?

The body of `deliveryDate` looks like this:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) {
    let deliveryTime;
    if (["MA", "CT"]     .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
  }
  else {
    let deliveryTime;
    if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
  }
}
```

Here, the caller is using a literal boolean value to determine which code should run—a classic flag argument. But the whole point of using a function is to follow the caller's instructions, so it is better to clarify the caller's intent with explicit functions.

In this case, I can do this by using Decompose Conditional, which gives me this:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) return rushDeliveryDate(anOrder);
  else        return regularDeliveryDate(anOrder);
}
function rushDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT"]     .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
}
function regularDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
}
```

The two new functions capture the intent of the call better, so I can replace each call of

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

with

```
aShipment.deliveryDate = rushDeliveryDate(anOrder);
```

and similarly with the other case.

When I've replaced all the callers, I remove `deliveryDate`.

A flag argument isn't just the presence of a boolean value; it's that the boolean is set with a literal rather than data. If all the callers of `deliveryDate` were like this:

```
const isRush = determineIfRush(anOrder);
aShipment.deliveryDate = deliveryDate(anOrder, isRush);
```

then I'd have no problem with `deliveryDate`'s signature (although I'd still want to apply Decompose Conditional).

It may be that some callers use the argument as a flag argument by setting it with a literal, while others set the argument with data. In this case, I'd still use Remove Flag Argument, but not change the data callers and not remove `deliveryDate` at the end. That way I support both interfaces for the different uses.

Decomposing the conditional like this is a good way to carry out this refactoring, but it only works if the dispatch on the parameter is the outer part of the function (or I can easily refactor it to make it so). It's also possible that the parameter is used in a much more tangled way, such as this alternative version of `deliveryDate`:

```
function deliveryDate(anOrder, isRush) {
  let result;
  let deliveryTime;
  if (anOrder.deliveryState === "MA" || anOrder.deliveryState === "CT")
    deliveryTime = isRush? 1 : 2;
  else if (anOrder.deliveryState === "NY" || anOrder.deliveryState === "NH") {
    deliveryTime = 2;
    if (anOrder.deliveryState === "NH" && !isRush)
      deliveryTime = 3;
  }
  else if (isRush)
    deliveryTime = 3;
  else if (anOrder.deliveryState === "ME")
    deliveryTime = 3;
  else
    deliveryTime = 4;
  result = anOrder.placedOn.plusDays(2 + deliveryTime);
  if (isRush) result = result.minusDays(1);
  return result;
}
```

In this case, teasing out `isRush` into a top-level dispatch conditional is likely more work than I fancy. So instead, I can layer functions over the `deliveryDate`:

```
function rushDeliveryDate   (anOrder) {return deliveryDate(anOrder, true);}
function regularDeliveryDate(anOrder) {return deliveryDate(anOrder, false);}
```

These wrapping functions are essentially partial applications of `deliveryDate`, although they are defined in program text rather than by composition of functions.

I can then do the same replacement of callers that I did with the decomposed conditional earlier on. If there aren't any callers using the parameter as data, I like to restrict its visibility or rename it to a name that conveys that it shouldn't be used directly (e.g., `deliveryDateHelperOnly`).

previous:
Parameterize Function

next:
Preserve Whole Object