Move Function

previous: Moving Features

next: Move Field

```
class Account {
  get overdraftCharge() {...}
```

```
class AccountType {
    get overdraftCharge() {...}
```

formerly: Move Method

Motivation

The heart of a good software design is its modularity—which is my ability to make most modifications to a program while only having to understand a small part of it. To get this modularity, I need to ensure that related software elements are grouped together and the links between them are easy to find and understand. But my understanding of how to do this isn't static—as I better understand what I'm doing, I learn how to best group together software elements. To reflect that growing understanding, I need to move elements around.

All functions live in some context; it may be global, but usually it's some form of a module. In an object-oriented program, the core modular context is a class. Nesting a function within another creates another common context. Different languages provide varied forms of modularity, each creating a context for a function to live in.

One of the most straightforward reasons to move a function is when it references elements in other contexts more than the one it currently resides in. Moving it together with those elements often improves encapsulation, allowing other parts of the software to be less dependent on the details of this module.

Similarly, I may move a function because of where its callers live, or where I need to call it from in my next enhancement. A function defined as a helper inside another function may have value on its own, so it's worth moving it to somewhere more accessible. A method on a class may be easier for me to use if shifted to another.

Deciding to move a function is rarely an easy decision. To help me decide, I examine the current and candidate contexts for that function. I need to look at what functions call this one, what functions are called by the moving function, and what data that function uses. Often, I see that I need a new context for a group of functions and create one with Combine Functions into Class or Extract Class. Although it can be difficult to decide where the best place for a function is, the more difficult this choice, often the less it matters. I find it valuable to try working with functions in one context, knowing I'll learn how well they fit, and if they don't fit I can always move them later.

Mechanics

- Examine all the program elements used by the chosen function in its current context. Consider whether they should move too.

  If I find a called function that should also move, I usually move it first. That way, moving a clusters of functions begins with the one that has the least dependency on the others in the group.

  If a high-level function is the only caller of subfunctions, then you can inline those functions into the high-level method, move, and reextract at the destination.
- Check if the chosen function is a polymorphic method.

  If I'm in an object-oriented language, I have to take account of super- and subclass declarations.
- Copy the function to the target context. Adjust it to fit in its new home.

  If the body uses elements in the source context, I need to either pass those elements as parameters or pass a reference to that source context.

  Moving a function often means I need to come up with a different name that works better in the new context.
- Perform static analysis.
- Figure out how to reference the target function from the source context.
- Turn the source function into a delegating function.
- Test.
- Consider Inline Function on the source function.

  The source function can stay indefinitely as a delegating function. But if its callers can just as easily reach the target directly, then it's better to remove the middle man.

Example: Moving a Nested Function to Top Level

I'll begin with a function that calculates the total distance for a GPS track record.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
```

```
    const pace = totalTime / 60 /  totalDistance ;
    return {
      time: totalTime,
      distance: totalDistance,
      pace: pace
    };

    function calculateDistance() {
      let result = 0;
      for (let i = 1; i < points.length; i++) {
        result += distance(points[i-1], points[i]);
      }
      return result;
    }
    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
    function calculateTime() { ... }
  }
```

I'd like to move `calculateDistance` to the top level so I can calculate distances for tracks without all the other parts of the summary.

I begin by copying the function to the top level.

```
  function trackSummary(points) {
    const totalTime = calculateTime();
    const totalDistance = calculateDistance();
    const pace = totalTime / 60 /  totalDistance ;
    return {
      time: totalTime,
      distance: totalDistance,
      pace: pace
    };

    function calculateDistance() {
      let result = 0;
      for (let i = 1; i < points.length; i++) {
        result += distance(points[i-1], points[i]);
      }
      return result;
    }
    ...
    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
    function calculateTime() { ... }
  }

  function top_calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
```

When I copy a function like this, I like to change the name so I can distinguish them both in the code and in my head. I don't want to think about what the right name should be right now, so I create a temporary name.

The program still works, but my static analysis is rightly rather upset. The new function has two undefined symbols: `distance` and `points`. The natural way to deal with `points` is to pass it in as a parameter.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

I could do the same with `distance`, but perhaps it makes sense to move it together with `calculateDistance`. Here's the relevant code:

function trackSummary…

```
function distance(p1,p2) {
  // haversine formula see http://www.movable-type.co.uk/scripts/latlong.html
  const EARTH_RADIUS = 3959; // in miles
  const dLat = radians(p2.lat) - radians(p1.lat);
  const dLon = radians(p2.lon) - radians(p1.lon);
  const a = Math.pow(Math.sin(dLat / 2),2)
          + Math.cos(radians(p2.lat))
          * Math.cos(radians(p1.lat))
          * Math.pow(Math.sin(dLon / 2), 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
  return EARTH_RADIUS * c;
}
function radians(degrees) {
  return degrees * Math.PI / 180;
}
```

I can see that `distance` only uses `radians` and `radians` doesn't use anything inside its current context. So rather than pass the functions, I might as well move them too. I can make a small step in this direction by moving them from their current context to nest them inside the nested `calculateDistance`.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
```

```
      return result;

      function distance(p1,p2) { ... }
      function radians(degrees) { ... }
    }
```

By doing this, I can use both static analysis and testing to tell me if there are any complications. In this case all is well, so I can copy them over to `top_calculateDistance`.

```
  function top_calculateDistance(points) {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;

    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
  }
```

Again, the copy doesn't change how the program runs, but does give me an opportunity for more static analysis. Had I not spotted that `distance` calls `radians`, the linter would have caught it at this step.

Now that I have prepared the table, it's time for the major change—the body of the original `calculateDistance` will now call `top_calculateDistance`:

```
  function trackSummary(points) {
    const totalTime = calculateTime();
    const totalDistance = calculateDistance();
    const pace = totalTime / 60 /  totalDistance ;
    return {
      time: totalTime,
      distance: totalDistance,
      pace: pace
    };

    function calculateDistance() {
      return top_calculateDistance(points);
    }
```

This is the crucial time to run tests to fully test that the moved function has bedded down in its new home.

With that done, it's like unpacking the boxes after moving house. The first thing is to decide whether to keep the original function that's just delegating or not. In this case, there are few callers and, as usual with nested functions, they are highly localized. So I'm happy to get rid of it.

```
  function trackSummary(points) {
    const totalTime = calculateTime();
    const totalDistance = top_calculateDistance(points);
    const pace = totalTime / 60 /  totalDistance ;
    return {
      time: totalTime,
      distance: totalDistance,
      pace: pace
    };
```

Now is also a good time to think about what I want the name to be. Since the top-level function has the highest visibility, I'd like it to have the best name. `totalDistance` seems like a good choice. I can't use that immediately since it will be shadowed by the variable inside `trackSummary`—but I don't see any reason to keep that anyway, so I use Inline Variable on it.

```
  function trackSummary(points) {
    const totalTime = calculateTime();
    const pace = totalTime / 60 /  totalDistance(points) ;
    return {
      time: totalTime,
      distance: totalDistance(points),
      pace: pace
    };

  function totalDistance(points) {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
```

If I'd had the need to keep the variable, I'd have renamed it to something like `totalDistanceCache` or `distance`.

Since the functions for `distance` and `radians` don't depend on anything inside `totalDistance`, I prefer to move them to top level too, putting all four functions at the top level.

```
function trackSummary(points) { ... }
function totalDistance(points) { ... }
function distance(p1,p2) { ... }
function radians(degrees) { ... }
```

Some people would prefer to keep `distance` and `radians` inside `totalDistance` in order to restrict their visibility. In some languages that may be a consideration, but with ES 2015, JavaScript has an excellent module mechanism that's the best tool for controlling function visibility. In general, I'm wary of nested functions—they too easily set up hidden data interrelationships that can get hard to follow.

Example: Moving between Classes

To illustrate this variety of Move Function, I'll start here:

class Account...

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }

  get overdraftCharge() {
    if (this.type.isPremium) {
      const baseCharge = 10;
      if (this.daysOverdrawn <= 7)
        return baseCharge;
      else
        return baseCharge + (this.daysOverdrawn - 7) * 0.85;
    }
    else
      return this.daysOverdrawn * 1.75;
  }
```

Coming up are changes that lead to different types of account having different algorithms for determining the charge. Thus it seems natural to move `overdraftCharge` to the account type class.

The first step is to look at the features that the `overdraftCharge` method uses and consider whether it is worth moving a batch of methods together. In this case I need the `daysOverdrawn` method to remain on the account class, because that will vary with individual accounts.

Next, I copy the method body over to the account type and get it to fit.

class AccountType...

```
  overdraftCharge(daysOverdrawn) {
    if (this.isPremium) {
      const baseCharge = 10;
      if (daysOverdrawn <= 7)
        return baseCharge;
      else
        return baseCharge + (daysOverdrawn - 7) * 0.85;
    }
    else
      return daysOverdrawn * 1.75;
  }
```

In order to get the method to fit in its new location, I need to deal with two call targets that change their scope. `isPremium` is now a simple call on `this`. With `daysOverdrawn` I have to decide—do I pass the value or do I pass the account? For the moment, I just pass the simple value but I may well change this in the future if I require more than just the days overdrawn from the account—especially if what I want from the account varies with the account type.

Next, I replace the original method body with a delegating call.

class Account...

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }
  get overdraftCharge() {
    return this.type.overdraftCharge(this.daysOverdrawn);
  }
```

Then comes the decision of whether to leave the delegation in place or to inline `overdraftCharge`. Inlining results in:

class Account…

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0)
      result += this.type.overdraftCharge(this.daysOverdrawn);
    return result;
  }
```

In the earlier steps, I passed `daysOverdrawn` as a parameter—but if there's a lot of data from the account to pass, I might prefer to pass the account itself.

class Account…

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }

  get overdraftCharge() {
    return this.type.overdraftCharge(this);
  }
```

class AccountType…

```
  overdraftCharge(account) {
    if (this.isPremium) {
      const baseCharge = 10;
      if (account.daysOverdrawn <= 7)
        return baseCharge;
      else
        return baseCharge + (account.daysOverdrawn - 7) * 0.85;
    }
    else
      return account.daysOverdrawn * 1.75;
  }
```

previous:
Moving Features

next:
Move Field