# Static functions vs. instance methods

Before we get to Algorithm #3 (accelerate towards the mouse), we need to cover one more rather important aspect of working with vectors and the `PVector` object: the difference between using static functions and instance methods.

Forgetting about vectors for a moment, take a look at the following code:

```
var x = 0;
var y = 5;
x = x + y;
```

Pretty simple, right? `x` has the value of 0, we add `y` to it, and now `x` is equal to 5. We could write the corresponding code pretty easily based on what we've learned about `PVector`.

```
var v = new PVector(0,0);
var u = new PVector(4,5);
v.add(u);
```

The vector v has the value of (0,0), we add u to it, and now v is equal to (4,5). Easy, right?

Let's take a look at another example of some simple math:

```
var x = 0;
var y = 5;
var z = x + y;
```

`x` has the value of 0, we add `y` to it, and store the result in a new variable `z`. The value of `x` does not change in this example, and neither does y! This may seem like a trivial point, and one that is quite intuitive when it comes

to mathematical operations with numbers. However, it’s not so obvious with mathematical operations in `PVector`. Let’s try to write the code based on what we know so far.

```
var v = new PVector(0,0);

var u = new PVector(4,5);

var w = v.add(u); // Don’t be fooled; this is incorrect!!!
```

The above might seem like a good guess, but it’s just not the way the `PVector` object works. If we look at the definition of `add()`...

```
PVector.prototype.add = function(v) {

    this.x = this.x + v.x;

    this.y = this.y + v.y;

 };
```

...we see that this code does not accomplish our goal. First, it does not return a new `PVector` (there is no `return` statement) and second, it changes the value of the `PVector` upon which it is called. In order to add two `PVector` objects together and return the result as a new `PVector`, we must use the "static" `add()` function.

A "static" function is a function that is defined on an object, but it doesn't change properties of the object. So why even define it on the object? Typically, it has something to do with the object, so it is logical to attach it to it. It treats the object more like a namespace. For example, all the static functions on `PVector` perform some sort of manipulation on passed in `PVector` objects and always return back some value. We could define those functions globally as well, but this way, we avoid global functions and have better ways of grouping related functionality.

Let's contrast. Here's how we use the `add()` instance method:

```
v.add(u);
```

That line of code would modify `v`, so we wouldn't need to save a return value. Conversely, here's how we use the `add()` static function:

```
var w = PVector.add(v, u);
```

If we didn't save the result of that function into a variable, that line of code would be useless, because the static version doesn't change the objects themselves. `PVector`'s static functions allow us to perform generic mathematical operations on `PVector` objects without having to adjust the value of one of the input `PVectors`.

Here's how we would write the static version of `add()`:

```
PVector.add = function(v1, v2) {
  var v3 = new PVector(v1.x + v2.x, v1.y + v2.y);
  return v3;
};
```

There are several differences here:

- We define the function directly on the object, *not on its prototype*
- We never access the `this` keyword inside the function
- We return a value from the function
  The PVector object has static versions of `add()`, `sub()`, `mult()`, and `div()`. It also has additional static functions that don't exist as instance methods, like `angleBetween()`, `dot()`, and `cross()`. We'll find ourselves using these functions as we continue making programs with `PVector`.