Name Property

name property in ES5 and its limitations

The name of a function can be retrieved using the name property of a function.

```
let guessMyName = function fName() {};
let fName2 = function() {};
let guessMyProperty = {
    prop: 1,
    methodName() {},
    get myProperty() {
        return this.prop;
    },
    set myProperty( prop ) {
        this.prop = prop;
    }
};


console.log( guessMyName.name );
//> "fName"
console.log( fName2.name );
//> "fName2"
console.log( guessMyProperty.methodName.name  );
//> "methodName"
console.log( guessMyProperty.methodName.bind( this ).name  );
//> "bound methodName"
```

When it comes to getters and setters, retrieving method names is a bit more complicated:

```
let propertyDescriptor = Object.getOwnPropertyDescriptor(
        guessMyProperty, 'myProperty' );

console.log( propertyDescriptor.get.name );
//> "get myProperty"
console.log( propertyDescriptor.set.name );
//> "set myProperty"
```

Function names provide you with limited debugging capabilities. For instance, you can create an automated test that checks if a function name is bound, making sure that no-one will delete the function binding in the code.

In theory, another use case is to retrieve the class name of an object by querying `obj.constructor.name`. Before using this construct, read the warnings here.

If you would still like to use the `name` property to retrieve the class of an object, note that it is bad practice. You can use `new.target` inside the `constructor` function just as well as the `name` property. Set the internal state of your object such that you will never need to retrieve the actual constructor of your object. Encapsulate all the roles and responsibilities of your object inside it. Checking for the constructor types outside the object definition results in worse maintainability and tighter coupling.

I will talk about `new.target` in more detail in the next lesson.