Module 0341: Typos, argh!

Tak Auyeung, Ph.D.

November 11, 2021

Contents

1 About this module

- Prerequisites:
- Objectives: This module looks into some techniques to deal with typo mistakes.

2 A scripting language vs a compiled language

Generally speaking, a compiled language is "strongly typed" and a scripting/interpreted language is not strongly typed.

"Strongly typed" means the compiler tries to catch all the mistakes that can be caught by statically analyzing a program. This includes the rule that an object can only have members that are officially declared in a `class` definition. Any reference of a member that is not in a class definition is reported as an error.

While this feature of a strongly typed language is great for catching the occasional typo mistakes, it also means interdependent parts of an app have brittle connections. When a new memeber is necessary, a class definition is changed, and then all the code needs to be recompiled. Depending on the size of the project, this can take a bit of time.

On the flip side, an language that is not strongly typed, and most interpreted languages including ECMAScript are dyanmically typed. This means that not only is membership dynamic, the kind (type) of values that can be stored is dynamic, as well.

At one moment, the value of a variable or member of an object can be a string, at the next moment, it can be an object, then it can be an integer. This can be seen as a flexibility, but it can also cause major issues because no tool can actually analyze an entire project for consistency.

3 Typos *will* happen

It is only a matter of time. So, be prepared!

There are some techniques to help detect typo mistakes early on, or at least product useful messages when there are problems. This section enumerates some of these techniques.

3.1 'use strict';

Literally.

At the very beginning of an ECMAScript file, use a line to specify this:

```
'use strict';
```

This informs the interpreter to be strict about the use of *variables*. In other words, this does not do anything to members of objects. We will get to that later.

What 'use strict' *does* do is to see references to variables and parameters that have not been "declared" as errors.

This means the following code is okay.

Listing 1: Variable declared before use

```
001 'use strict';
002 let x
003 console.log(`still a bad idea, but here is the value of x: \${x}`)
```

A variable that is just declared has a value of `undefined`, and that is exactly what `console.log` will print as a the value of `x`.

The following, however, will end up with an error when the code is run.

Listing 2: Variable *not* declared before use

```
001 'use strict';
002 console.log(`still a bad idea, but here is the value of x: \${x}`)
```

The use of `'use strict';` is really a no-brainer. The reliance on "automatic declaration upon first reference" is really a bad idea!

3.2 The use of `const` and alternate way to refer to a member

`'use strict';` does not handle the following problem:

Listing 3: Mispelling the name of a member of an object

```
001 'use strict';
002 let obj = { m1: 'value of member m1' }
003 console.log(`The value of obj.m1 is \${obj.m2}`)
```

The code will *run*, but it will output `undefined` as the value of member `m1` of `obj`. For this type of typo, there is a way to let the interpreter detect the problem:

Listing 4: Mispelling the name of a member of an object, but have the interpreter catch it

```
001 'use strict';
002 const strM1 = 'm1' // just a const string def of the member name
003 let obj = { m1: 'value of member m1' }
004 console.log(`The value of obj.m1 is \${obj[strM2]}`)
```

This time, the program will end with an error because the reference of `strM2` (instead of `strM1`) triggers the interpreter to complain that `strM2` is undefined.

This approach uses the alternative method of accessing a member from an object. As it turns out, `obj.m1` is really an abbreviated form of `obj['m1']`. However, there is a significant difference between the two *forms* of member reference.

When `obj.m1` is used, `m1` is not considered a string, it is just an identifier. However, when `obj['m1']` is used, `m1` is enclosed in quotes, meaning that the `[]` notation expects the name of a member *as a string*.

This difference by itself does not help, but we can use `const` (constant) definitions to define the names of members, then the `[]` notation can refer to a constant instead of a literal string. References to a undefined constant/variable/parameter results in an error because of `'use strict';`, and that is why this method works.

Note that this aproach works particularly well in the context of web scripting. For example, let's say `reset` is a URL parameter. Then the use of `const strReset = 'reset'` can be used to generate HTML code such as `res.write(`<a href="?${strReset}` as well as code to check whether reset is defined, and if so, whether it is 1:

```
if (req.query[\${strReset}] !== undefined && req.query[\${strReset}] == 1)
```

In other words, this approach makes it easier to enforce the consistency HTML code and references to parameters from the HTML code. The argument also applies to references to names of tables and columns of a database. The only downside of this approach is that it is a little more verbose than without the indirect reference to constants.

3.3 The "get" method

While directly reference a member is handy, it is often considered bad programming practice from the perspectives of ADT (abstract data type) and OOP (object oriented programming) paradigms.

This subsection does get into a bit of OOP discussion, which is a bit advanced for beginning programmers.

Listing 5: Use get method to access a member

```
'use strict';
let obj =
  {
    m1: 'the value of m1',
    getM1: function ()
    {
      return this.m1
    },
    setM1: function (newValue)
    {
      this.m1 = newValue
    },
  }
console.log(`the value of obj.m1 is ${obj.getM1()}`)
```

You can download this code here

In this code, you can see how `getM1` and `setM1` are really just like member `m1` of `obj`. This is one of the key concepts of OOP: a function can be a member of an object. In the `getM1` function, you can also see how the word `this` is used to refer to the owner of the method *when the member function is called*.

While this approach seems like an overkill for a simple problem, it does have some additional advantages. This is because whatever the `get` and `set` methods (a *method* means the same as a "member function", this is OOP talk) are getting and setting does not need to be a member. The thing being get ("gotten") and set can be computed, or not even represented in the app itself.

In casual terms, accessing a member directly is like "give me your watch (so I can check the current time", whereas using a get method is more like "tell me the current time." To generalize, accessing a member directly is acquiring the resource to accomplish an objective, whereas calling a get method is stating an objective and let the object decide how to fulfill the objective.

But this approach may not work when we are *dynamically* adding a member, right? This is why an interpreted language is both awesome and scary at the same time. Observe the following program.

Listing 6: Defining a 'get' method on-the-fly to access a member

```
'use strict';
let obj = {}
obj.m1 = 'the value of m1'
obj.getM1 = function ()
{
  return this.m1
}
obj.setM1 = function (newValue)
```

```
{
  this.m1 = newValue
}
// from this point on, do not ask for m1 directly
// use getm1 and setM1 to access m1
console.log(`the value of obj.m1 is ${obj.getM1()}`)
```

You can download this code here

This approach is an overkill to refer to GET parameters or session values because of the sheer number of GET parameters or session values typically utilized in a web application. It'd be extremely tedious to manually define all the `get` and `set` methods.

There *are* ways to "autogenerate" `get` and `set` methods from a list of GET parameter names or session names. However, such a mechanism is quite beyond the scope of this module (intended for beginning or intermediate developers). Furthermore, such a mechanism is not efficient.

That said, I know someone is still reading. For those who are curious *all the way*, here is your reward:

Listing 7: Use a get-set-factory to manufactor methods, then use 'get' method to access a member

```
'use strict';
let obj = {}
const strM1 = 'm1'
function addMember(obj, memberName, initialValue)
{
  if (typeof(obj) != 'object')
  {
    throw new Error(`obj must be an object`)
  }
  if (typeof(memberName) != 'string')
  {
    throw new Error(`membername must be a string`)
  }
  obj[memberName] = initialValue
  addGetSetMethod(obj, memberName, false)
  // the following line "camelize" so the first letter
}
function addGetSetMethod(obj, memberName, safetyCheck = true)
{
  if (safetyCheck)
  {
    if (typeof(obj) != 'object')
    {
      throw new Error(`obj must be an object`)
    }
    if (typeof(memberName) != 'string')
    {
      throw new Error(`membername must be a string`)
    }
  }
  // turn first letter to uppercase to use camel method names
  let camelMemberName = memberName[0].toUpperCase() + memberName.slice(1)
  obj[`get${camelMemberName}`] = function ()
  {
    return this[memberName]
  }
  obj[`set${camelMemberName}`] = function (newValue)
  {
    this[memberName] = newValue
```

```
040     }
041 }
042 addMember(obj, strM1, 'Value of member m1')
043 // from this point on, do not ask for m1 directly
044 // use getm1 and setM1 to access m1
045 console.log(`the value of obj.m1 is ${obj.getM1()}`)
046 console.log(`the value of obj.m1 is ${obj.getM2()}`)
```

You can download this code here

Note that the function `addGetSetMethod` can be applied to the `query` and `session` members of the request (`req`) parameters of an end-point handler.

If there is a long list of GET parameters, the following code can reduce the tedium involved:

```
[ strP1, strP2, strP3, strP4 ].forEach((e) => addGetSetMethod(req.query, e))
```

4 `inspect` to the rescue

The `inspector` can also be helpful to find typo mistakes. However, this requires a little more effort on the part of the developer. When a GET parameter is set (as seen in the URL bar of the browser), but the script acts as if it is not, then it is time to use the inspector to examine the code.

A good place to put a break point is at the very beginning of an end-point handler. This is before your code has control. At that break point, use `repl` (REPL means Real-Eval-Print-Loop) in the inspector to evaluate `req.query` and `req.session`. You can also narraw down and try to evaluate an individual value such as `req.session.counter`.

It may also be helpful to show the `typeof` of a member. This is because it is easy to miss the distinction between a string of 1 (`'1'`) versus a numerical value of 1 (`1`).

If you can confirm the existence and value of the member, then the fault is probably due to the *reference* of the member, so you can focus on how the member is referred to in the end-point handler.