Chapter 50: Strict mode

Section 50.1: For entire scripts

Strict mode can be applied on entire scripts by placing the statement `"use strict";` before any other statements.

```
"use strict";
// strict mode now applies for the rest of the script
```

Strict mode is only enabled in scripts where you define `"use strict"`. You can combine scripts with and without strict mode, because the strict state is not shared among different scripts.

Version ≥ 6

Note: All code written inside ES2015+ modules and classes are strict by default.

Section 50.2: For functions

Strict mode can also be applied to single functions by prepending the `"use strict";` statement at the beginning of the function declaration.

```
function strict() {
   "use strict";
   // strict mode now applies to the rest of this function
   var innerFunction = function () {
     // strict mode also applies here
   };
}

function notStrict() {
  // but not here
}
```

Strict mode will also apply to any inner scoped functions.

Section 50.3: Changes to properties

Strict mode also prevents you from deleting undeletable properties.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

The above statement would simply be ignored if you don't use strict mode, however now you know why it does not execute as expected.

It also prevents you from extending a non-extensible property.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
    myObject.age = 25;   // No errors
}

function setAge() {
```

```
    "use strict";
    myObject.age = 25;  // TypeError: can't define property "age": Object is not extensible
}
```

Section 50.4: Changes to global properties

In a non-strict-mode scope, when a variable is assigned without being initialized with the **`var`**, **`const`** or the **`let`** keyword, it is automatically declared in the global scope:

```
a = 12;
console.log(a); // 12
```

In strict mode however, any access to an undeclared variable will throw a reference error:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

This is useful because JavaScript has a number of possible events that are sometimes unexpected. In non-strict-mode, these events often lead developers to believe they are bugs or unexpected behavior, thus by enabling strict-mode, any errors that are thrown enforces them to know exactly what is being done.

```
"use strict";
                        // Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the
                        // misspelling of variable
```

This code in strict mode displays one possible scenario: it throws a reference error which points to the assignment's line number, allowing the developer to immediately detect the mistype in the variable's name.

In non-strict-mode, besides the fact that no error is thrown and the assignment is successfully made, the `mistypedVaraible` will be automatically declared in the global scope as a global variable. This implies that the developer needs to look up manually this specific assignment in the code.

Furthermore, by forcing declaration of variables, the developer cannot accidentally declare global variables inside functions. In non-strict-mode:

```
function foo() {
   a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

In strict mode, it is necessary to explicitly declare the variable:

```
function strict_scope() {
   "use strict";
   var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

The variable can also be declared outside and after a function, allowing it to be used, for instance, in the global scope:

```
function strict_scope() {
   "use strict";
   a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Section 50.5: Duplicate Parameters

Strict mode does not allow you to use duplicate function parameter names.

```
function foo(bar, bar) {}  // No error. bar is set to the final argument when called

"use strict";
function foo(bar, bar) {}; // SyntaxError: duplicate formal argument bar
```

Section 50.6: Function scoping in strict mode

In Strict Mode, functions declared in a local block are inaccessible outside the block.

```
"use strict";
{
  f(); // 'hi'
  function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

Scope-wise, function declarations in Strict Mode have the same kind of binding as **`let`** or **`const`**.

Section 50.7: Behaviour of a function's arguments list

`arguments` object behave different in *strict* and *non strict* mode. In *non-strict* mode, the `argument` object will reflect the changes in the value of the parameters which are present, however in *strict* mode any changes to the value of the parameter will not be reflected in the `argument` object.

```
function add(a, b){
    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 5,10
}

add(1, 2);
```

For the above code, the `arguments` object is changed when we change the value of the parameters. However, for *strict* mode, the same will not be reflected.

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 1,2
```

```
}
```

It's worth noting that, if any one of the parameters is **`undefined`**, and we try to change the value of the parameter in both *strict-mode* or *non-strict* mode the `arguments` object remains unchanged.

Strict mode

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1,undefined
    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined

add(1)
// 1, undefined
// 1, undefined
```

Non-Strict Mode

```
function add(a,b) {

    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

Section 50.8: Non-Simple parameter lists

```
function a(x = 5) {
  "use strict";
}
```

is invalid JavaScript and will throw a `SyntaxError` because you cannot use the directive `"use strict"` in a function with Non-Simple Parameter list like the one above - default assignment `x = 5`

Non-Simple parameters include -

- Default assignment

```
function a(x = 1) {
  "use strict";
```

```
}
```

- Destructuring

```
function a({ x }) {
  "use strict";
}
```

- Rest params

```
function a(...args) {
  "use strict";
}
```