1. **Функции**

# ES5:

# [15.2.3.12](http://www.ecma-international.org/ecma-262/5.1/#sec-15.2.3.12) Object.isFrozen ( O )

When the **isFrozen** function is called with argument O, the following steps are taken:

1. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
   1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
   2. If [IsDataDescriptor](http://www.ecma-international.org/ecma-262/5.1/" \l "sec-8.10.2)(*desc*) is **true** then
      1. If *desc*.[[Writable]] is **true**, return **false**.
   3. If *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **false**, then return **true**.
4. Otherwise, return **false**.

# ES6:

#### Object.isFrozen ( O )

When the **isFrozen** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object, return **true**.
2. Return TestIntegrityLevel(*O*, "**frozen**").

Анализируя выше приведенные цитаты можно заметить, что в версии ES5, если аргумент object не является объектом, то будет вызываться исключение TypeError. Однако в новой версии ES6, в том случае, если object не является объектом, возвращается значение true, иначе возвращается абстрактная операция TestIntegrityLevel(O, "frozen"), которой передаются параметры (O, "frozen").

Изменения, произошедшие в ES6, на мой взгляд, повышают эффективность работы языка. Теперь больше не будет возникать ошибка TypeError, всякий раз, когда операнд или аргумент, переданный функции, несовместим с типом, который данная функция ожидает получить. Вместо этого просто будет возвращаться значение true.

# Если глубже посмотреть, что же происходит внутри TestIntegrityLevel (O, level) то можно увидеть следующее (согласно пункту 7.3.12 версии ES6):

1. [Assert](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-algorithm-conventions): [Type](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-ecmascript-data-types-and-values)(*O*) is Object.
2. [Assert](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-algorithm-conventions): *level* is either "**sealed**" or "**frozen**".
3. Let *status* be [IsExtensible](https://people.mozilla.org/~jorendorff/es6-draft.html" \l "sec-isextensible-o)(*O*).
4. [ReturnIfAbrupt](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-returnifabrupt)(*status*).
5. If *status* is **true**, then return **false**
6. NOTE If the object is extensible, none of its properties are examined.
7. Let *keysArray* be the result of calling the [[OwnPropertyKeys]] internal method of *O*.
8. Let *keys* be [CreateListFromArrayLike](https://people.mozilla.org/~jorendorff/es6-draft.html" \l "sec-createlistfromarraylike)(*keysArray*).
9. [ReturnIfAbrupt](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-returnifabrupt)(*keys*).
10. Let *pendingException* be **undefined**.
11. Let *configurable* be **false**.
12. Let *writable* be **false**.
13. Repeat for each element *k* of *keys*,
    1. Let *status* be the result of calling the [[GetOwnProperty]] internal method of *O* with *k*.
    2. If *status* is an [abrupt completion](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-completion-record-specification-type), then
       1. If *pendingException* is **undefined**, then set *pendingException* to *status*.
       2. Let *configurable* be **true**.
    3. Else,
       1. Let *currentDesc* be *status*.[[value]].
       2. If *currentDesc* is not **undefined**, then
          1. Set *configurable* to *configurable* logically ored with *currentDesc*.[[Configurable]].
          2. If [IsDataDescriptor](https://people.mozilla.org/~jorendorff/es6-draft.html" \l "sec-isdatadescriptor)(*currentDesc*) is **true**, then
             1. Set *writable* to *writable* logically ored with *currentDesc*.[[Writable]].
14. If *pendingException* is not **undefined**, then return *pendingException*.
15. If *level* is "**frozen**" and *writable* is **true**, then return **false**.
16. If *configurable* is **true**, then return **false**.
17. Return **true**.

# Анализируя дальше стандарты версий ES5 и ES6, можно заметить, что похожие изменения коснулись и Object.isExtensible ( O ) и Object.isSealed ( O ). Однако если object не является объектом Object.isSealed ( O ), то возвращается так же значение true. А в случае с Object.isExtensible ( O ) будет возвращаться значение false.

Что касается возникновения ошибки TypeError, то как в случае с Object.isExtensible ( O ), так и с Object.isSealed ( O ), она больше не будет возникать. Что, безусловно, поможет при написании кода.

# ES5:

# Object.isSealed ( O )

When the **isSealed** function is called with argument O, the following steps are taken:

1. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
   1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
   2. If *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **false**, then return **true**.
4. Otherwise, return **false**.

# ES6:

#### 19.1.2.13 Object.isSealed ( O ).

When the **isSealed** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object, return **true**.
2. Return TestIntegrityLevel(*O*, "**sealed**").

# ES5:

# Object.isExtensible ( O )

When the **isExtensible** function is called with argument O, the following steps are taken:

1. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*O*) is not Object throw a **TypeError** exception.
2. Return the Boolean value of the [[Extensible]] internal property of *O*.

# ES6:

#### Object.isExtensible ( O )

When the **isExtensible** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object, return **false**.
2. Return the result of IsExtensible(*O*).

### По сравнению с версией ES5, в ES6 появилась новая функция Object.is.

# ES6:

#### Object.is ( value1, value2 )

When the **is** function is called with arguments *value1* and *value2* the following steps are taken:

1. Return SameValue(*value1*, *value2*).

Благодаря вновь появившейся функции можно легко сравнить два значения, не волнуясь о том, что результат сравнения может быть неправильным (как в случае, например, с оператором идентичности «===»)

# ES5/ ES6:

# The Strict Equality Comparison Algorithm

The comparison x === y, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is different from [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*y*), return **false**.
2. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Undefined, return **true**.
3. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Null, return **true**.
4. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Number, then
   1. If *x* is **NaN**, return **false**.
   2. If *y* is **NaN**, return **false**.
   3. If *x* is the same Number value as *y*, return **true**.
   4. If *x* is **+0** and *y* is **−0**, return **true**.
   5. If *x* is **−0** and *y* is **+0**, return **true**.
   6. Return **false**.
5. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
6. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
7. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.

NOTET his algorithm differs from [the SameValue Algorithm (9.12)](http://www.ecma-international.org/ecma-262/5.1/#sec-9.12) in its treatment of signed zeroes and NaNs.

Т.е. если попробовать сравнить отрицательный ноль с положительным нулем, то мы получим true (хотя на самом деле это не так).

0 === -0; // true

Или же сравнить NaN с еще одним NaN , то результат так же будет неправильным.

NaN === NaN; // false

Конечно же, если обратить внимание на NOTET, то можно заметить, что The SameValue Algorithm позволяет избежать подобной ошибки:

# The SameValue Algorithm

The internal comparison abstract operation SameValue(x, y), where x and y are ECMAScript language values, produces **true** or **false**. Such a comparison is performed as follows:

1. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is different from [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*y*), return **false**.
2. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Undefined, return **true**.
3. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Null, return **true**.
4. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Number, then.
   1. If *x* is NaN and *y* is NaN, return **true**.
   2. If *x* is +0 and *y* is -0, return **false**.
   3. If *x* is -0 and *y* is +0, return **false**.
   4. If *x* is the same Number value as *y*, return **true**.
   5. Return **false**.
5. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
6. If [Type](http://www.ecma-international.org/ecma-262/5.1/#sec-8)(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
7. Return true if *x* and *y* refer to the same object. Otherwise, return **false**.

Однако вариант, предлагаемый в версии ES6, на мой взгляд, намного удобнее.

#### Object.is ( value1, value2 )

When the **is** function is called with arguments *value1* and *value2* the following steps are taken:

1. Return SameValue(*value1*, *value2*).

#### Очевидно, что результат выполнения функции Object.is ( value1, value2 ) будет правильным.

Object.is(0, -0); // false

Object.is(NaN, NaN); // true

В результате выполнения этой функции будет возвращаться результат операции SameValue(value1, value2), которая работает, без сомнения, правильно.

1. **Цикл**

По сравнению с версией ES5 (где присутствует функциональная область видимости), в ES6 появилась возможность создавать блочную область видимости. Чтобы определить переменную с ограниченной областью видимости необходимо написать ключевое слово let. Так же появилось ключевое слово const, с помощью которого можно объявить переменную и присвоить ей значение, которое потом невозможно будет изменить.

# ES5:

# 12.2 Variable Statement

## Syntax

*VariableStatement* **:**

**var** *VariableDeclarationList* **;**

*VariableDeclarationList* **:**

*VariableDeclaration*

*VariableDeclarationList* **,** *VariableDeclaration*

*VariableDeclarationListNoIn* **:**

*VariableDeclarationNoIn*

*VariableDeclarationListNoIn* **,** *VariableDeclarationNoIn*

*VariableDeclaration* **:**

*Identifier* *Initialiser*opt

*VariableDeclarationNoIn* **:**

*Identifier* *InitialiserNoIn*opt

*Initialiser* **:**

**=** *AssignmentExpression*

*InitialiserNoIn* **:**

**=** *AssignmentExpressionNoIn*

A variable statement declares variables that are created as defined in [10.5](http://www.ecma-international.org/ecma-262/5.1/#sec-10.5). Variables are initialised to **undefined** when created. A variable with an *Initialiser* is assigned the value of its *AssignmentExpression* when the *VariableStatement* is executed, not when the variable is created.

# ES6:

### Let and Const Declarations

NOTE A **let** and **const** declarations define variables that are scoped to the running execution context’s LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable’s *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initializer* is assigned the value of its *Initializer*’s *AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a **let** declarationdoes not have an *Initializer* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

Syntax

LexicalDeclaration[In, Yield] **:**

LetOrConst BindingList[?In, ?Yield] **;**

LetOrConst **:**

**let**  
**const**

BindingList[In, Yield] **:**

LexicalBinding[?In, ?Yield]  
BindingList[?In, ?Yield] **,** LexicalBinding[?In, ?Yield]

LexicalBinding[In, Yield] **:**

BindingIdentifier[?Yield] Initializer[?In, ?Yield]optBindingPattern[?Yield] Initializer[?In, ?Yield]

**Например,**

var a = 7;

var b = 3;

if (a === 7) {

let a = 8; // видна только внутри if

var b = 1; // видна везде внутри функции

console.log(a); // 8

console.log(b); // 1

}

console.log(a); // 7

console.log(b); // 1

**Или же (с const)**

const a = 9;

console.log(a); // 9

a = 5;

console.log(a); // 9

const b;

console.log(b); // undefined

b = 5;

console.log(b); // undefined

### Подводя итог, можно отметить, что переменные, объявленные c помощью ключевого слова var, будут видны в любом месте функции (даже если они объявлены внутри блока), в отличие от ключевого слова let, которое позволит видеть переменные только внутри блока, где они непосредственно и были объявлены.

### Появление let и const в новой версии будет являться, бесспорно, преимуществом по сравнению с предыдущей версией и поможет упростить процесс решение задач.