СТРОГИЙ РЕЖИМ

Переключает JS в некий "современный режим". В начале файла пишется 'use strict'. Строгий режим - это директива, призванная добавить и изменить некоторые "опасные" возможности старого JS.

Строгий режим запрещает:
1) Присваивать значение в неопределенную переменную
2) Нельзя использовать инструкцию with
3) В объекте нельзя определить два свойства с одним именем let obj = {a:0,a:1}
4) Нельзя определить два одинаковых аргумента у функции (a,a)=>{some code...}
5) Изменения внутри функции её объекта arguments не изменяют аргументы function f(x)=>{arguments[0]=5;return x}; f(10)// Вернет 10
6) this в функции не преобразуется в объект
7) Строгий режим резервирует больше слов (let, private, interface, public...)

Есть еще некоторые изменения, но, считаю, что перечислил основные.

ОТЛИЧИЕ var, let и const

1) Область видимости var - функция. Область видимости let - блок.

Что это значит:
var видна ВЕЗДЕ в функции. Даже если объявлена, как переменная цикла for ВНУТРИ функции.
let видна только по иерархии вниз. И не видна по иерархии вверх. Т.е. не видна вне цикла и вне функции.

2) Если использовать переменную var до объявления, то вернется undefined. Если же использовать let до объявления,
то упадет ошибка ReferenceError

Немного о const. Ведет себя так же, как let. Но ее нельзя ПЕРЕНАЗНАЧИТЬ.
Что это значит:
const obj = {name:"Henry"};
obj.name = "John"// Все нормально
obj = {} // Упадет ошибка

ОПЕРАТОРЫ СРАВНЕНИЯ

Всегда возвращают логический тип

1) '==' и '==='
'==' - оператор сравнения, который приводит операнды к числу (если сравниваются строки используется побуквенное
сравнение букв, в зависимости от номера букв в кодировке Unicode)

'===' - оператор строгого сравнения, который не приводит операнды к числу (если операнды разного типа, возвращает false)

2) '!==' и '!===' - операторы неравенства. Действуют так же, как и ребята выше. Но, являются их логической противоположностью

3) null == undefined // true Это специальное правило языка. Тут уж ничего не поделаешь
null === undefined //false т.к. их типы разные

4) null > 0 // false потому что null здесь 0
null == 0 //false здесь null ни к чему не приводится т.к. при сравнении он равен ТОЛЬКО undefined
null >= 0 //true потому что null здесь 0

Нестрогое равенство == и операторы сравнения '>' и '<' работают по разному.
Сравнение преобразуют null в число, а равенство
не преобразует null. Ибо null равен только undefined

5) undefined > 0 // false
undefined < 0 // false
undefined == 0 // false

В первых двух случаях с операторами сравнения undefined превращается в NaN - математическую ошибку.
В третьем случае false, ибо undefined равен ТОЛЬКО null

МОЖНО ЛИ СОХРАНИТЬ ФУНКЦИЮ В ПЕРЕМЕННУЮ?

Конечно можно. Причем, как стрелочную функцию, так и Expression, и Declaration

```
let sum = (a,b)=>{
        return a+b;
}
```

ФУНКЦИЯ КОЛЛБЭК В РЕПОЗИТОРИИ В ПАПКЕ public/callback.html

КЛЮЧЕВОЕ СЛОВО this

Это слово указывает на текущий контекст. Может быть использовано, как в функции, так и в объекте.

Пример применения в функции let say = function(){alert(this.name)} // В строгом режиме упадет ошибка
Пример применения в объекте let obj = {age:55, func(){alert(this.age)}}; obj.func() // Вернет 55

Потеря this : Как только метод передается в отрыве от объекта, this теряется. Его можно привязать методом bind(context)

Пример:
Ошибка: let user = {name:"Dmitry",say(){alert(`Hello ${this.name}`)}}; let func = user.say; func()
Правильно: let func = user.say.bind(user); func()

Решение проблемы потери контекста через стрелочные функции:
Стрелочные функции хороши, если в методе объекта есть вложенные методы. Тогда this внутри этих методов будет всегда ссылаться на текущий объект, т.к. у стрелочных функций нет своего контекста. Они берут родительский контекст.

Пример

```
let obj = {
        name:"John",
        doSome(){
                setTimeout(()=>{ //Если здесь не будет стрелочной функции, this будет потерян
                        alert(this.name)
                },1000)
        }
}
obj.doSome();
```

ТАЙМЕРЫ

let timer = SetTimeout(func,ms) - используется для вызова функции "func", спустя заданные промежуток времени "ms" ОДИН РАЗ
let interval = SetInterval(func,ms) - используется для вызова функции "func", спустя заданные промежуток времени "ms" ПОСТОЯННО

Отмена таймеров:
clearTimeout(timer)
clearInterval(interval)

Рекурсивный SetTimeout :

Более гибкий, чем SetInterval. Время каждого последующего вызова можно задавать, в зависимости от результатов предыдущего.
Реализация:
let timer = setTimeout(function some(){timer = setTimeout(tick,1000)},1000)