Интернет-технологии и распределённая обработка данных

Лекция 12

Инструкции и обработка исключений JavaScript

1. Инструкции – общие сведения
2. Инструкции-объявления
3. Условные переходы
4. Циклы
5. Инструкции безусловного перехода
6. Генерация и обработка исключений

Что такое «инструкция»?

Инструкция (statement) – автономная часть скрипта, которая может быть выполнена; **команда**.

Значит, любой скрипт – это набор инструкций.

Для разделения инструкций JavaScript использует **;** или переход на новую строку (обсуждали это ранее).

Инструкция block

Иногда синтаксис JavaScript **требует** в определённом месте наличия только одной инструкции.

А по смыслу программы инструкций надо несколько.

Инструкция block используется для группировки инструкций (несколько инструкций, рассматриваются как одна).

Блок ограничивается фигурными скобками:

{ statement_1; statement_2; ... statement_n; }

обычно используется с управляющими инструкциями (например, if, for, while).

Пустые инструкции

Иногда синтаксис JavaScript **требует** в определённом месте наличия инструкции (любой).

А по смыслу программы инструкция не нужна.

Выход: использовать *пустую инструкцию* (вроде и инструкция, но ничего не делает).

Пустая инструкция в JavaScript – это просто символ **;**

Инструкции-выражения

Простейший вид инструкции – это выражение с побочным эффектом (помним о точке с запятой)

```
greeting = "Hello " + name;   // присваивание

counter++;                    // инкремент (и декремент)

delete obj.prop;              // оператор delete

alert(greeting);              // вызов функции
```

Инструкции объявления

Инструкция `var` позволяет объявить одну или несколько переменных (опционально – с начальным значением):

`var` *имя_1* `[ =` *знач_1*`][,...,` *имя_n* `[=` *знач_n*`]]`

Если значение не указано, подразумевается `undefined`.

Инструкция `var` может размещаться внутри функции (*локальная переменная функции*), вне функции – на верхнем уровне скрипта (*глобальная переменная*).

```
var x = 10;                 // глобальная переменная
var y;                      // неинициализированная
переменная

function f() {
    var z = "Str"; // локальная переменная
}

// счётчик цикла
for(var i = 0; i < 10; i++) alert(i);
```

let Объявляет локальную переменную в области видимости блока, необязательно инициализирует её значением.

```
if (x > y) {
let gamma = 12.7 + y;
 i = gamma * x;
}
```

const Объявляет именованную константу только для чтения. const MY_FAV = 7;

```
const MY_OBJECT = {"key": "value"};
```

Инструкция var – ловушка!

```
var x = y = 10;
```

Вы думаете, что это

```
var y = 10;
var x = 10;
```

Нет! Это вот что:

```
y = 10;
var x = 10;
```

Скоро станет понятно, что это **очень разные вещи**.

В JavaScript при объявлении переменной не указывается тип, потому что переменные могут хранить значения *любых типов*:

```
var x = 10;              // сначала храним число
x = "Srt";               // затем строку
x = [2, 3, 5];           // затем массив (объект)
```

Инструкция function

Инструкция `function` служит для определения функции:

```
function имя_функции([арг_1 [,арг_2 [..., арг_n]]]){
  инструкции
}
```

Круглые и фигурные скобки здесь обязательны.

Инструкция `function` может располагаться на «верхнем уровне» скрипта или быть вложенной в функцию.

В любом случае, не допускается вложение этой инструкции в ветвления, циклы и тому подобное! (нужен «первый уровень вложенности»)

Условные переходы
Инструкция if

Две формы записи:

```
if (выражение)
    инструкция
```

```
if (выражение)
    инструкция1
else
    инструкция2
```

выражение вычисляется, результат приводится к boolean.

Приведение типов: напоминание

Ложными являются следующие значения

- `false`
- `null`
- `undefined`
- `""` (пустая строка)
- `0`
- `NaN`

Все остальные значения являются истинными.

```
if (0) { // 0 преобразуется к false
...
}

if (1) { // 1 преобразуется к true
...
}

var cond = (year != 2011); // true/false
if (cond) {
 ...
}
```

```
var  year = prompt('Введите год появления стандарта ECMA-262 5.1',
'');
if (year == 2011) {
    alert( 'Да вы знаток!' );
} else {
    alert( 'А вот и неправильно!' ); // любое значение, кроме 2011
}


var year = prompt('В каком году появилась спецификация ECMA-262 5.1?',
' ');
if (year < 2011) {
    alert( 'Это слишком рано..' ); }
else if (year > 2011) {
    alert( 'Это поздновато..' ); }
 else {
```

```
var userName = prompt('Кто пришёл?', '');
if (userName == 'Админ') {
  var pass = prompt('Пароль?', '');
  if (pass == 'Повелитель сайта') {
    alert( 'Добро пожаловать!' );}
  else if (pass == null) { // нажатие «Отмена»
    alert( 'Вход отменён' );
  } else {
  alert( 'Пароль неверен' );}
} else if (userName == null) {
  alert( 'Вход отменён' );
} else {
  alert( 'Я вас не знаю' ); }
```

Еще раз о тернарной операции

```
var access;
var age = prompt('Сколько вам лет?', ' ');


if (age > 14) {
access = true;
} else {
access = false;
}
 alert(access);
```

```
access = age > 14 ? true : false;
```

```
access = age > 14;
```

```
var age = prompt('возраст?', 18);
var message = (age < 3) ? 'Здравствуй, малыш!' :
(age < 18) ? 'Привет!' :
(age < 100) ? 'Здравствуйте!' :
'Какой необычный возраст!';
  alert( message );

if (age < 3) {
message = 'Здравствуй, малыш!';
} else if (age < 18) {
message = 'Привет!';
} else if (age < 100) {
message = 'Здравствуйте!';
} else {
message = 'Какой необычный возраст!';
}
```

Инструкция switch

```
switch (expression) {
    case valueExpression1: // if (expression === valueExpression1)
 // инструкции, соответствующие valueExpression1
        [break;]
    ...
    case valueExpressionN: // if (expression === valueExpressionN)
 // инструкции, соответствующие значению valueExpressionN
        [break;]
    [default:
 // инструкции, которые выполняются при отсутствии совпадений
        [break;]
    ]
}
```

На первый взгляд похожа на аналоги из других языков. Однако после `case` указываются **выражения**!

Сначала вычисляем *`expression`*.

Затем вычисляем выражение *`valueExpression1`*.

Если значения совпали (`===`), выполняем соответствующий набор инструкций.

Если не совпали, переходим к вычислению *`valueExpression2`*.

```
var  a = 2 + 2;
switch (a) {
case 3:
    alert( 'Маловато' );
    break;
case 4:
    alert( 'В точку!' );
    break;
case 5:
    alert( 'Перебор' );
    break;
default:
alert( 'Я таких значений не знаю' );
}
```

Инструкция switch

1. Часть `default` не является обязательной.

2. Части `case` можно группировать:

```
case 0:
case 1:
    // инструкции
```

3. Набор инструкций после `case` (и `default`) может завершаться переходом (`break`, `return`), но **не обязан**!

Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются

```
var a = 2 + 2;
switch (a) {
    case 3: alert( 'Маловато' );
    case 4: alert( 'В точку!' );
    case 5: alert( 'Перебор' );
    default: alert( 'Я таких значений не знаю' );
}
```

В case могут быть любые выражения, в том числе включающие в себя переменные и функции:

```
var x = 12;
switch(true) {
  case x < 0:
    alert("Negative");
    break;
  case x >= 0 && x <= 100:
    alert("Between 0 and 100");
    break;
  default:
    alert("More than 100");
}
```

Несколько значений case можно **группировать**:
case 3 и case 5 выполняют один и тот же код

```
var a = 2+2;
switch (a) {
    case 4: alert('Верно!');
            break;
    case 3:
    case 5:  alert('Неверно!');
            alert('Немного ошиблись, бывает.');
            break;
    default: alert('Странный результат, очень странный');
}
```

```
var arg = prompt("Введите arg?") ;
switch (arg) {
case '0':
case '1':
    alert( 'Один или ноль' );
case '2':
    alert( 'Два' ); break;
case 3:
    alert( 'Никогда не выполнится' );
default:
    alert('Неизвестное значение: ' + arg)
}
```

```
var a = +prompt('a?', ' ');
switch (a) {
    case 0:
       alert( 0 );
       break;
    case 1:
        alert( 1 );
        break;
    case 2:
    case 3:
        alert( '2,3' );
        break;
}
```

```
var a = +prompt('a?', ' ');
if (a == 0) {
    alert( 0 );
}
if (a == 1) {
    alert( 1 );
}
if (a == 2 || a == 3) {
    alert( '2,3' );
}
```

Циклы while и do-while

Две формы циклов:

```
while (выражение)
    инструкция
```

```
do
    инструкция
while (выражение);
```

Циклы работают, пока *выражение*, приведённое к boolean, равно `true`. Обратите внимание на обязательную `;` в цикле `do-while`.

```
var i = 0;
while (i < 3) {
alert( i );
i++;
}
```

```
var i = 0;
do {
alert( i );
i++;
} while (i < 3);
```

вместо `while (i!=0)` обычно пишут `while (i)`

Цикл for

```
for (инициализация; проверка; инкремент)
    инструкция
```

Это (почти) эквивалентно:

```
инициализация;
while (проверка) {
    инструкция;
    инкремент;
}
```

Цикл for

Инициализация – вычисляется один раз перед циклом. Обычно это присваивание. Допускается инструкция `var`.

Проверка – это выражение вычисляется перед каждой итерацией. Тело цикла выполняется, если `true`.

Инкремент – это выражение вычисляется в конце цикла. Обычно это присваивание, или ++, или --.

```
for (var count = 0; count < 10; count++)
    alert(count);
```

```
var i, j;
for (i = 0, j = 10; i < 10; i++, j--)
    sum += i * j;
```

```
function tail(o) {    // находит последний элемент
в списке
  for(; o.next; o = o.next) ;
  return o;
}
```

```
var i = 0;

for (; i < 3;) {
alert( i );
// цикл превратился в аналог while (i<3)
}
```

А можно и вообще убрать всё, получив бесконечный цикл:

```
for (;;) {
// будет выполняться вечно
}
```

Цикл for...in

```
for (переменная in объект)
    инструкция
```

Этот цикл выполнят перебор имён (строки!) свойств объекта .

объект – выражение, которое преобразуется в объект.

переменная – любое левостороннее выражение или инструкция `var`.

```
// вывод значений свойств объекта
for(var prop in obj)
  alert(obj[prop]);

// копируем имена свойств в массив
var a = [], i = 0;
for(a[i++] in obj) ; // пустое тело цикла

// индексы массива - это свойства объекта
for(var i in a)
  alert(a[i]);
```

Цикл for...in перебирает только *перечислимые свойства* объекта (как сделать такое свойство – особый вопрос).

Перебираются свойства, которые есть и в объекте, и в его прототипах.

Обычно порядок перебора соответствует порядку определения свойств (свойства прототипа – в конце), но спецификация не даёт указаний на этот счёт.

Цикл for...of

В то время как for...in обходит имена свойств,

for...of выполняет обход значений свойств:

```
let arr = [ 3, 5, 7 ];
arr.foo = "hello";

for (let i in arr) {
console.log(i); // выведет "0", "1", "2", "foo"
}
for (let i of arr) {
console.log(i); // выведет "3", "5", "7"
}
```

Метки инструкций

Любая инструкция может быть снабжена меткой:

идентификатор: инструкция

1. Метки используются для переходов при помощи `break` или `continue`.
2. Метки работают только внутри тех инструкций, к которым они применяются.

помечают `switch`, `while`, `do`, `for`.

Инструкция break

Первая форма инструкции `break`:

```
break;
```

Эта форма используется для выхода из `switch` или для выхода из цикла (самого внутреннего цикла, если циклы вложены).

```
var sum = 0;

while (true) {

var value = +prompt("Введите число", '');

if (!value) break;  //ничего не введено


sum += value;

}
alert( 'Сумма: ' + sum );
```

Вторая форма инструкции `break`:

```
break имя_метки;
```

Эта форма выполняет переход на следующую инструкцию за помеченной.

Break должен находиться внутри отмеченного блока, который соответствует метке.

Отмеченная инструкция может быть любой блочной инструкцией;

она не обязательно должна являться циклом.

```
loops:
    for (var i = 0; i < 10; i++) {
        for (var j = 0; j < 15; j++) {
            if (i == 5 && j == 5) break loops;
//выход из двух циклов сразу
        }
    }
alert(i + j);     // выведет 10
```

Инструкция continue

Существует в двух формах:

```
continue;
```

```
continue имя_метки;
```

прекращает выполнение *текущей итерации* цикла

В отличие от break: прерывает не весь цикл,

а только текущее выполнение его тела

```
for (i = 0; i < data.length; i++) {
    // не обрабатывать неопределенные данные
    if (!data[i]) continue;

    total += data[i];
}
```

Особенность continue в разных циклах:

1. В цикле while *выражение* в начале цикла проверяется снова, и если оно равно true, тело цикла выполняется с начала.

2. В цикле for вычисляется выражение *инкремента* и снова вычисляется выражение *проверки*, чтобы понять, следует ли выполнять следующую итерацию.

continue также может быть использована с меткой, в этом случае управление перепрыгнет на следующую итерацию цикла с меткой.

n>1 – простое, если при делении на любое число
от 2 до n-1 есть остаток.

выводит все простые числа из интервала от 2 до 10

```
nextPrime:
  for (var i = 2; i < 10; i++) {
    for (var j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert( i ); // простое
  }
```

Инструкция return

Инструкция `return` осуществляет немедленный выход из функции, возвращая указанное выражение или undefined, если не указано:

```
return выражение;
```

```
return;
```

Используется **только в функциях**.

```
  return;
  return true;
  return false;
  return x;
  return x + y / 3;


return
a + b;
// трансформируется ASI в
return;
a + b;
 // Консоль предупреждает "unreachable code after return
```

Генерация исключений

Для генерации исключения используется инструкция `throw`, а для его обработки инструкция `try...catch`.

```
throw выражение;
```

выражение содержит значение, которое будет выброшено и может иметь **любой тип результата**! , но рекомендуется объект, желательно – совместимый со стандартным, то есть чтобы у него были как минимум свойства name и message.

```
throw "Error2"; // string
throw 42; // number
throw true; // boolean
throw { toString: function() { return "I'm an object!"; } }; // object
```

В качестве конструктора ошибок можно использовать встроенный конструктор: new Error(message) или любой другой.

Конструкторы для стандартных ошибок:
SyntaxError, ReferenceError, RangeError

```
function factorial(x) {
  if (x < 0)
    throw new Error("x cannot be negative");

  for(var f = 1; x > 1; f *= x, x--) ;
  return f;
}
```

Случай 1. `throw` используется внутри функции:

Выполнение функции прекращается, управление передаётся на ближайший `catch` в стеке вызова.

Если `catch` отсутствует, прекращается выполнения всего скрипта (но не других скриптов на странице).

Случай 2. `throw` используется вне функции («глобально»):

Если есть обрамляющий блок обработки, управление передаётся на его `catch`.

Если обрамляющий блок обработки отсутствует, прекращается выполнения всего скрипта (но не других скриптов на странице).

Инструкция try...catch

```
try {
    // инструкции, которые могут сгенерировать исключение
}
catch (e) {
    // блок обработки исключений
}
finally {
    // инструкции, которые выполняются всегда,
    // независимо от того, что произошло в блоке try
}
```

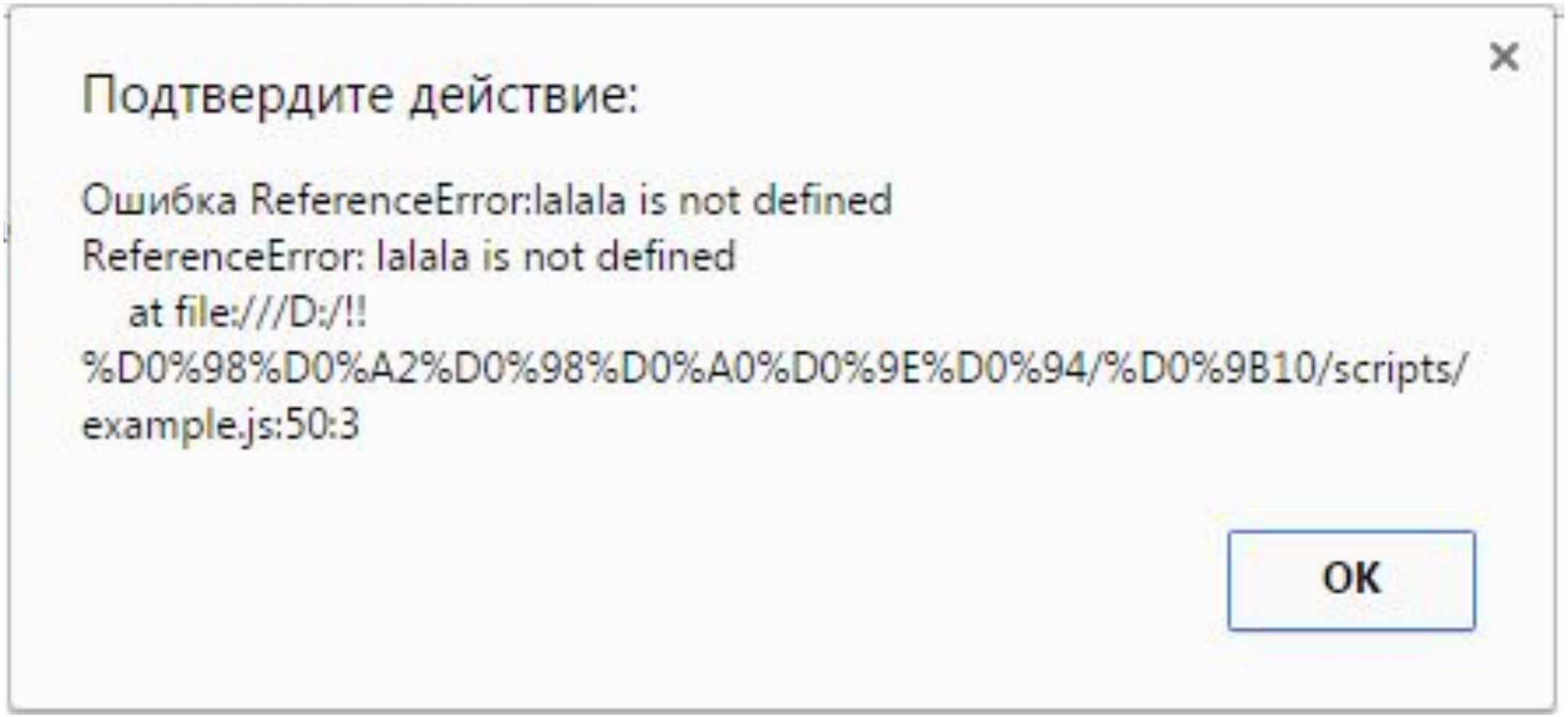
Обработка исключений

Все фигурные и круглые скобки обязательны.

Присутствует или блок `catch`, или блок `finally`, или оба.

Переменная **`e`** (имя может быть любым) видима только в `catch`-блоке, будет содержать объект ошибки с подробной информацией о произошедшем , имеет значение, которое указывали после `throw` при генерации исключения.

```
try {

  alert('Начало блока try');  // (1) <--

  lalala; // ошибка, переменная не определена!

  alert('Конец блока try');  // (2)

} catch(e) {

  alert('Ошибка ' + e.name + ":" + e.message + "\n" + e.stack); // (3) <--

}

alert("Потом код продолжит выполнение...");
```


Подтвердите действие:
Ошибка ReferenceError:lalala is not defined
ReferenceError: lalala is not defined
at file:///D:/!!
%D0%98%D0%A2%D0%98%D0%A0%D0%9E%D0%94/%D0%9B10/scripts/
example.js:50:3
OK

объект ошибки

`name` тип ошибки. Например, при обращении к несуществующей переменной: `"ReferenceError"`.

`message` текстовое сообщение о деталях ошибки.

`stack` содержит строку с информацией о последовательности вызовов, которая привела к ошибке (кроме IE8-)

В зависимости от браузера, могут быть и дополнительные свойства

В блоке catch можно обработать исключение или (и) сгенерировать исключение повторно.

Секцию finally используют, чтобы завершить начатые операции при любом варианте развития событий.

Блок finally выполняется, если блок try завершился:

- как обычно, достигнув конца блока
- из-за инструкции break, continue или return
- с исключением, обработанным в блоке catch
- с неперехваченным исключением, которое продолжает своё распространение на более высокие уровни

Блок finally – вопрос ?

```
function f() {
  try {
    return 5;
  }
  finally {
    return 10;
  }
}

// функция f() вернёт?  10
var x = f(); alert(x);
```

```
    openMyFile();
try {
    writeMyFile(theData);
} catch(e) {
    handleError(e);
} finally {
    closeMyFile();
}
```

window.onerror

Если ошибка произошла вне блока try..catch или выпала из try..catch наружу, во внешний код, то рекомендуется свойство window.onerror.

если в него записать функцию, то она выполнится и получит в аргументах сообщение ошибки, текущий URL и номер строки, откуда «выпала» ошибка.

Необходимо , чтобы функция была назначена заранее.

```
window.onerror = function(message, url, lineNumber) {
alert("Поймана ошибка, выпавшая в глобальную область!\n"
+ "Сообщение: " + message + "\n(" + url + ":" + lineNumber + ")");
};

function readData() {
error(); // ой, что-то не так
 }

 readData();
```

Как правило, роль window.onerror заключается в том, чтобы не оживить скрипт – скорее всего, это уже невозможно, а в том, чтобы отослать сообщение об ошибке на сервер, где разработчики о ней узнают.

Существуют даже специальные веб-сервисы, которые предоставляют скрипты для отлова и аналитики таких ошибок, например:

https://errorception.com/ или http://www.muscula.com/.