09 - Обработка ошибок. Объект Date.

1. Обработка ошибок window.onerror

Свойство **onerror объекта Window** – это обработчик событий, который вызывается во всех случаях, когда необработанное исключение достигло вершины стека вызовов и когда броузер готов отобразить сообщение об ошибке в консоли JavaScript. Если присвоить этому свойству функцию, функция будет вызываться всякий раз, когда в окне будет возникать ошибка выполнения программного кода JavaScript: присваиваемая функция станет обработчиком ошибок для окна. Исторически сложилось так, что обработчику события onerror объекта Window передается три строковых аргумента, а не единственный объект события, как в других обработчиках.

Первый аргумент обработчика window.onerror – это сообщение, описывающее произошедшую ошибку.

Второй аргумент – это строка, содержащая URL-адрес документа с JavaScript-кодом, приведшим к ошибке.

Третий аргумент – это номер строки в документе, где произошла ошибка.

Помимо этих трех аргументов важную роль играет значение, возвращаемое обработчиком onerror. Если обработчик onerror возвращает true, это говорит браузеру о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, браузер не должен выводить собственное сообщение об ошибке.

Обработчик onerror является пережитком первых лет развития JavaScript, когда в базовом языке отсутствовала инструкция try/catch обработки исключений. В современном программном коде этот обработчик используется редко. Однако на время разработки вы можете определить свой обработчик ошибок, как показано ниже, который будет уведомлять вас о всех происходящих ошибках:

```
// Вывести сообщение об ошибке в виде диалога, но не более 3 раз
window.onerror = function(msg, url, line) {
      if (onerror.num++ < onerror.max) {
            alert("ОШИБКА: " + msg + "\n" + url + ":" + line);
            return true;
      }
}
onerror.max = 3;
onerror.num = 0;
```

2. Инструкция throw

Исключение – это сигнал, указывающий на возникновение какой-либо исключительной ситуации или ошибки. Возбуждение исключения (throw) – это способ просигнализировать о такой ошибке или исключительной ситуации. Перехватить исключение (catch) – значит обработать его, т. е. предпринять действия, необходимые или подходящие для восстановления после исключения. В JavaScript исключения возбуждаются в тех случаях, когда возникает ошибка времени выполнения и когда

программа явно возбуждает его с помощью инструкции throw. Инструкция throw имеет следующий синтаксис:

```
throw выражение;
```

Результатом выражения может быть значение любого типа. Инструкции throw можно передать число, представляющее код ошибки, или строку, содержащую текст сообщения об ошибке. Интерпретатор JavaScript возбуждает исключения, используя экземпляр класса Error одного из его подклассов, и вы также можете использовать подобный подход. Объект Error имеет свойство name, определяющее тип ошибки, и свойство message, содержащее строку, переданную функции-конструктору. Ниже приводится пример функции, которая возбуждает объект Error при вызове с недопустимым аргументом:

```
function factorial(x) {
    // Если входной аргумент не является допустимым значением, возбуждается исключение!
    if (x < 0) throw new Error("x не может быть отрицательным");

    // В противном случае значение вычисляется и возвращается нормальным образом
    for(var f = 1; x > 1; f *= x, x--) /* пустое тело цикла */ ;
        return f;
    }
```

Когда возбуждается исключение, интерпретатор JavaScript немедленно прерывает нормальное выполнение программы и переходит к ближайшему обработчику исключений. В обработчиках исключений используется конструкция catch инструкции try/catch/finally, описание которой приведено в следующем разделе. Если блок программного кода, в котором возникло исключение, не имеет соответствующей конструкции catch, интерпретатор анализирует следующий внешний блок программного кода и проверяет, связан ли с ним обработчик исключений. Это продолжается до тех пор, пока обработчик не будет найден. Если исключение генерируется в функции, не содержащей инструкции try/catch/finally, предназначенной для его обработки, то исключение распространяется выше, в программный код, вызвавший функцию. Таким образом исключения распространяются по лексической структуре методов JavaScript вверх по стеку вызовов. Если обработчик исключения так и не будет найден, исключение рассматривается как ошибка и о ней сообщается пользователю.

3. Инструкция try/catch/finally

Инструкция try/catch/finally реализует механизм обработки исключений в JavaScript. Конструкция try в этой инструкции просто определяет блок кода, в котором обрабатываются исключения. За блоком try следует конструкция catch с блоком инструкций, вызываемых, если где-либо в блоке try возникает исключение. За конструкцией catch следует блок finally, содержащий программный код, выполняющий заключительные операции, который гарантированно выполняется независимо от того, что

происходит в блоке try. И блок catch, и блок finally не являются обязательными, однако после блока try должен обязательно присутствовать хотя бы один из них. Блоки try, catch и finally начинаются и заканчиваются фигурными скобками. Это обязательная часть синтаксиса, и она не может быть опущена, даже если между ними содержится только одна инструкция. Следующий фрагмент иллюстрирует синтаксис и назначение инструкции try/catch/finally:

```
try {
    // Обычно этот код без сбоев работает от
начала до конца.
    // Но в какой-то момент в нем может быть
сгенерировано исключение
    // либо непосредственно с помощью
инструкции throw, либо косвенно -
    // вызовом метода, генерирующего
исключение.
} catch (e) {
    // Инструкции в этом блоке выполняются
тогда и только тогда, когда в блоке try
    // возникает исключение. Эти инструкции
могут использовать локальную переменную e,
    // ссылающуюся на объект Error или на другое
значение, указанное в инструкции throw.
    // Этот блок может либо некоторым образом
обработать исключение, либо
    // проигнорировать его, делая что-то
другое, либо заново сгенерировать
    // исключение с помощью инструкции throw.
} finally {
    // Этот блок содержит инструкции, которые
выполняются всегда, независимо от того,
    // что произошло в блоке try. Они
выполняются, если блок try завершился:
    // 1) как обычно, достигнув конца блока
    // 2) из-за инструкции break, continue или return
    // 3) с исключением, обработанным
приведенным в блоке catch выше
    // 4) с неперехваченным исключением,
которое продолжает свое
    // распространение на более высокие уровни
}
```

Обратите внимание, что за ключевым словом catch следует идентификатор в скобках. Этот идентификатор похож на параметр функции. Когда будет перехвачено исключение, этому параметру будет присвоено исключение (например, объект Error). В отличие от обычной переменной идентификатор, ассоциированный с конструкцией catch, существует только в теле блока catch.

Далее приводится более реалистичный пример инструкции try/catch. В нем вызываются метод factorial(), определенный в предыдущем разделе, и методы prompt() и alert() клиентского JavaScript для организации ввода и вывода:

```
try {
// Запросить число у пользователя
var n = Number(prompt("Введите положительное
число", ""));

// Вычислить факториал числа, предполагая,
что входные данные корректны
var f = factorial(n);

// Вывести результат
alert(n + "! = " + f);
}
catch (ex) { // Если данные некорректны,
управление будет передано сюда
      alert(ex); // Сообщить пользователю об
ошибке
}
```

Это пример инструкции try/catch без конструкции finally. Хотя finally используется не так часто, как catch, тем не менее иногда эта конструкция оказывается полезной. Однако ее поведение требует дополнительных объяснений. Блок finally гарантированно исполняется, если исполнялась хотя бы какая-то часть блока try, независимо от того, каким образом завершилось выполнение программного кода в блоке try. Эта возможность обычно используется для выполнения заключительных операций после выполнения программного кода в предложении try.

В обычной ситуации управление доходит до конца блока try, а затем переходит к блоку finally, который выполняет необходимые заключительные операции. Если управление вышло из блока try как результат выполнения инструкций return, continue или break, перед передачей управления в другое место выполняется блок finally.

Если в блоке try возникает исключение и имеется соответствующий блок catch для его обработки, управление сначала передается в блок catch, а затем – в блок finally. Если отсутствует локальный блок catch, то управление сначала передается в блок finally, а затем переходит на ближайший внешний блок catch, который может обработать исключение.

Если сам блок finally передает управление с помощью инструкции return, continue, break или throw или путем вызова метода, генерирующего исключение, незаконченная команда на передачу управления отменяется и выполняется новая. Например, если блок finally сгенерирует исключение, это исключение заменит любое ранее сгенерированное исключение. Если в блоке finally имеется инструкция return, произойдет нормальный выход из метода, даже если генерировалось исключение, которое не было обработано.

Конструкции try и finally могут использоваться вместе без конструкции catch. В этом случае блок finally – это просто набор инструкций, выполняющих заключительные операции, который будет гарантированно выполнен независимо от наличия в блоке try

инструкции break, continue или return. Напомню, из-за различий в работе инструкции continue в разных циклах невозможно написать цикл while, полностью имитирующий работу цикла for. Однако если добавить инструкцию try/finally, можно написать цикл while, который будет действовать точно так же, как цикл for, и корректно обрабатывать инструкцию continue:

```
    // Имитация цикла for( инициализация ;
проверка ; инкремент ) тело цикла;
инициализация;

    while( проверка ) {
        try { тело цикла ; }
        finally { инкремент ; }
    }
```

Обратите однако внимание, что тело цикла while, содержащее инструкцию break, будет вести себя несколько иначе (из-за выполнения лишней операции инкремента перед выходом), чем тело цикла for, поэтому даже используя конструкцию finally, невозможно точно сымитировать цикл for с помощью цикла while.

4. Определение типа браузера с помощью свойства navigator.userAgent

```
    // Определяет свойства browser.name и browser.version,
позволяющие выяснить
    // тип клиента. За основу взят программный
код из библиотеки jQuery
    // Оба свойства, name и version, возвращают
строки, и в обоих случаях
    // значения могут отличаться от
фактических названий браузеров и версий.
    // Определяются следующие названия
броузеров:
    //
    // "webkit": Safari или Chrome; version содержит номер
сборки WebKit
    // "opera": Opera; version содержит фактический
номер версии броузера
    // "mozilla": Firefox или другие броузеры,
основанные на механизме gecko;
    // version содержит номер версии Gecko
    // "msie": IE; version содержит фактический номер
версии броузера

    var browser = (function() {
        var s = navigator.userAgent.toLowerCase();
        var match = /(webkit)[ \/]([\w.]+)/.exec(s) ||
        /(opera)(?:.*version)?[ \/]([\w.]+)/.exec(s) ||
        /(msie) ([\w.]+)/.exec(s) ||
        !/compatible/.test(s) && /(mozilla)(?:.*?
rv:([\w.]+))?/.exec(s) ||
        [];
```

```
        return { name: match[1] || "", version: match[2] || "0" };
    }());
```

5. Оператор in.

Оператор in требует, чтобы левый операнд был строкой или мог быть преобразован в строку. Правым операндом должен быть объект. Результатом оператора будет значение true, если левое значение представляет собой имя свойства объекта, указанного справа. Например:

```
    var point = { x:1, y:1 }; // Определить объект
    "x" in point // => true: объект имеет свойство с именем "x"
    "z" in point // => false: объект не имеет свойства с именем "z".
    "toString" in point // => true: объект наследует метод toString

    var data = [7,8,9]; // Массив с элементами 0, 1 и 2
    "0" in data // => true: массив содержит элемент "0"
    1 in data // => true: числа преобразуются в строки
    3 in data // => false: нет элемента 3
```

6. Оператор instanceof.

Оператор instanceof требует, чтобы левым операндом был объект, а правым – имя класса объектов. Результатом оператора будет значение true, если объект, указанный слева, является экземпляром класса, указанного справа. В противном случае результатом будет false. Например:

```
    var d = new Date(); // Создать новый объект с помощью конструктора Date()
    d instanceof Date; // Вернет true; объект d был создан с функцией Date()
    d instanceof Object; // Вернет true; все объекты являются экземплярами Object
    d instanceof Number; // Вернет false; d не является объектом Number
    var a = [1, 2, 3]; // Создать массив с помощью литерала массива
    a instanceof Array; // Вернет true; a - это массив
    a instanceof Object; // Вернет true; все массивы являются объектами
    a instanceof RegExp; // Вернет false; массивы не являются регулярными выражениями
```

Обратите внимание, что все объекты являются экземплярами класса Object. Определяя, является ли объект экземпляром класса, оператор instanceof принимает во внимание и «суперклассы». Если левый операнд instanceof не является объектом, instanceof возвращает false. Если правый операнд не является функцией, возбуждается исключение TypeError.

7. Оператор typeof

Унарный оператор typeof помещается перед единственным операндом, который может иметь любой тип. Его значением является строка, указывающая на тип данных операнда. Следующая таблица определяет значения оператора typeof для всех значений, возможных в языке JavaScript:

```
x                                   typeof x
undefined                           "undefined"
null                                "object"
true или false                           "boolean"
любое число или NaN                          "number"
любая строка                                  "string"
любая функция                          "function"
любой объект базового языка,
не являющийся функцией                        "object"
```

Оператор typeof может применяться, например, в таких выражениях:

```
(typeof value == "string") ? "'" + value + "'" : value
```

Оператор typeof можно также использовать в инструкции switch. Обратите внимание, что операнд оператора typeof можно заключить в скобки, что делает оператор typeof более похожим на имя функции, а не на ключевое слово или оператор:

```
typeof(i)
```

Обратите внимание, что для значения null оператор typeof возвращает строку «object». Если вам потребуется отличать null от других объектов, добавьте проверку для этого спец. случая. Для всех объектных типов и типов массивов результатом оператора typeof является строка «object», поэтому он может быть полезен только **для определения принадлежности значения к объектному или к простому типу.** Чтобы отличить один класс объектов от другого, следует использовать другие инструменты, такие как оператор instanceof.

Несмотря на то что функции в JavaScript также являются разновидностью объектов, оператор typeof отличает функции, потому что они имеют собственные возвращаемые значения.

8. Объект Date.

Для работы с датой и временем в JavaScript используются объекты Date. Объект Date – это тип данных, встроенный в язык JavaScript. Объекты Date создаются с помощью синтаксиса new Date(). После создания объекта Date можно воспользоваться его многочисленными методами. Многие из методов позволяют получать и устанавливать поля года, месяца, дня, часа, минуты, секунды и миллисекунды в соответствии либо с локальным временем, либо с временем UTC (универсальным, или GMT). Метод toString() и его варианты преобразуют даты в понятные для восприятия строки. getTime() и setTime() преобразуют количество миллисекунд, прошедших с полуночи (GMT) 1 января 1970 года, во внутреннее представление объекта Date и обратно. В этом стандартном миллисекундном формате дата и время представляются одним целым, что делает дату очень простой арифметически. Стандарт ECMAScript требует, чтобы объект Date мог

представить любые дату и время с миллисекундной точностью в пределах 100 миллионов дней до и после 01.01.1970. Этот диапазон равен ±273785 лет, поэтому JavaScript-часы будут правильно работать до 275755 года.

Конструктор Date() без аргументов создает объект Date со значением, равным текущим дате и времени. Если конструктору передается единственный числовой аргумент, он используется как внутреннее числовое представление даты в миллисекундах, аналогичное значению, возвращаемому методом getTime(). Когда передается один строковый аргумент, он рассматривается как строковое представление даты в формате, принимаемом методом Date.parse(). Кроме того, конструктору можно передать от двух до семи числовых аргументов, задающих индивидуальные поля даты и времени.

Все аргументы, кроме первых двух – полей года и месяца, – могут отсутствовать. Обратите внимание: эти поля даты и времени задаются на основе локального времени, а не времени UTC (Universal Coordinated Time – универсальное скоординированное время), аналогичного GMT (Greenwich Mean Time – среднее время по Гринвичу). В качестве альтернативы может использоваться статический метод Date.UTC().

Date() может также вызываться как функция (без оператора new). При таком вызове Date() игнорирует любые переданные аргументы и возвращает текущие дату и время.

```
new Date()
new Date(миллисекунды)
new Date(строка_даты)
new Date(год, месяц, день, часы, минуты, секунды, мс)
```

9. Методы объекта Date.

У объекта Date нет доступных для записи или чтения свойств; вместо этого доступ к значениям даты и времени выполняется через методы. Большинство методов объекта Date имеют две формы: одна для работы с локальным временем, другая – с универсальным временем (UTC или GMT). Если в имени метода присутствует строка «UTC», он работает с универсальным временем.

Известно множество методов, позволяющих работать с созданным объектом Date:

```
d = new Date(); // Получает текущую дату и время
document.write('Сегодня: "+d.toLocaleDateString() + '. ');// Показывает дату
document.write('Время: '+ d.toLocaleTimeString()); // Показывает время
var dayOfWeek = d.getDay(); // День недели
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6); // Сегодня выходной?
```

Еще одно типичное применение объекта Date – это вычитание миллисекундного представления текущего времени из другого времени для определения относительного местоположения двух временных меток. Следующий пример клиентского кода показывает два таких применения:

```
      today = new Date(); // Запомнить сегодняшнюю дату
      christmas = new Date(); // Получить дату из текущего
года
      christmas.setMonth(11); // Установить месяц
декабрь...
      christmas.setDate(25); // и 25-е число

      // Если Рождество еще не прошло, вычислить
количество миллисекунд между текущим
моментом
      // и Рождеством, преобразовать его в
количество дней и вывести сообщение
      Date.getDate() 785
      if (today.getTime() < christmas.getTime()) {
            difference = christmas.getTime() - today.getTime();
            difference = Math.floor(difference / (1000 * 60 * 60 * 24));
            document.write('Всего ' + difference + ' дней до
Рождества!<p>');
      }

      // Здесь мы используем объекты Date для
измерения времени
      // Делим на 1000 для преобразования
миллисекунд в секунды
      now = new Date();
      document.write('<p>Страница загружалась' +
      (now.getTime()-today.getTime())/1000 + 'секунд.');
```

Методы объекта Date могут вызываться только для объектов типа Date и генерируют исключение TypeError, если вызывать их для объектов другого типа.

get[UTC]Date() - возвращает день месяца из объекта Date в соответствии с локальным или универсальным временем.

get[UTC]Day() - возвращает день недели из объекта Date в соответствии с локальным или универсальным временем.

get[UTC]FullYear() - возвращает год даты в полном четырехзначном формате в локальном или универсальном времени.

get[UTC]Hours() - возвращает поле часов в объекте Date в локальном или универсальном времени.

get[UTC]Milliseconds() - возвращает поле миллисекунд в объекте Date в локальном или универсальном времени.

get[UTC]Minutes() - возвращает поле минут в объекте Date в локальном или универсальном времени.

get[UTC]Month() - возвращает поле месяца в объекте Date в локальном или универсальном времени.

get[UTC]Seconds() - возвращает поле секунд в объекте Date в локальном или универсальном времени.

getTime() - возвращает внутреннее представление (миллисекунды) объекта Date. Обратите внимание: это значение не зависит от часового пояса, следовательно, отдельный метод getUTCTime() не нужен.

getTimezoneOffset() - возвращает разницу в минутах между локальным и универсальным представлениями даты. Обратите внимание: возвращаемое значение зависит от того, действует ли для указанной даты летнее время.

set[UTC]Date() - устанавливает день месяца в Date в соответствии с локальным или универсальным временем.

set[UTC]FullYear() - устанавливает год (и, возможно, месяц и день) в Date в соответствии с локальным или универсальным временем.

set[UTC]Hours() - устанавливает час (и, возможно, поля минут, секунд и миллисекунд) в Date в соответствии с локальным или универсальным временем.

set[UTC]Milliseconds() - устанавливает поле миллисекунд в Date в соответствии с локальным или универсальным временем.

set[UTC]Minutes() - устанавливает поле минут (и, возможно, поля секунд и миллисекунд) в Date в соответствии с локальным или универсальным временем.

set[UTC]Month() - устанавливает поле месяца (и, возможно, дня месяца) в Date в соответствии с локальным или универсальным временем.

set[UTC]Seconds() - устанавливает поле секунд (и, возможно, поле миллисекунд) в Date в соответствии с локальным или универсальным временем.

setTime() - устанавливает поля объекта Date в соответствии с миллисекундным форматом.

toDateString() - возвращает строку, представляющую дату из Date для локального часового пояса.

toISOString() - преобразует Date в строку, используя стандарт ISO-8601, объединяющий формат представления даты/времени и UTC.

toJSON() - сериализует объект Date в формат JSON с помощью метода toISOString().

toLocaleDateString() - возвращает строку, представляющую дату из Date в локальном часовом поясе в соответствии с локальными соглашениями по форматированию дат.

toLocaleString() - преобразует Date в строку в соответствии с локальным часовым поясом и локальными соглашениями о форматировании дат.

toLocaleTimeString() - возвращает строку, представляющую время из Date в локальном часовом поясе на основе локальных соглашений о форматировании времени.

toString() - преобразует Date в строку в соответствии с локальным часовым поясом.

toTimeString() - возвращает строку, представляющую время из Date в локальном часовом поясе.

toUTCString() - преобразует Date в строку, используя универсальное время.

valueOf() - преобразует объект Date в его внутренний миллисекундный формат.

Задание на лекции:
3. Дополнить авторизацию регистрацией
4. Защитить страницу от прямого доступа.
3. Написать код для отображения часов.

Домашнее задание:
1. Напишите функцию getSecondsToTomorrow() которая возвращает, сколько секунд осталось до завтра. Например, если сейчас 23:00, то:

getSecondsToTomorrow() == 3600

P.S. Функция должна работать в любой день, т.е. в ней не должно быть конкретного значения сегодняшней даты.

2. Напишите функцию formatDate(date), которая выводит дату date в формате дд.мм.гг: Например:

```
var d = new Date(2014, 0, 30); // 30 января 2014
alert( formatDate(d) ); // '30.01.14'
```

P.S. Обратите внимание, ведущие нули должны присутствовать, то есть 1 января 2001 должно быть 01.01.01, а не 1.1.1.

3. Выведите на экран текущий месяц словом, по-русски.

4. Выведите на экран текущий день недели (словом, по-русски). Создайте для этого вспомогательную функцию, которая параметром принимает число, а возвращает день недели по-русски.

5. Выведите на экран количество минут, часов, дней с 1-го января 1970 года до настоящего момента времени.

6. Создайте инпут, в который пользователь вводит дату своего рождения в формате '2014-12-31' (с конкрентным годом). По нажатию на кнопку выведите под инпутом сколько дней осталось до его дня рождения.