2. Лекция: Типы данных и операторы

Рассматриваются основы синтаксиса языка JavaScript: литералы, переменные, массивы, условные операторы, операторы циклов.

Как и любой другой язык программирования, JavaScript поддерживает встроенные структуры и *типы данных*. Все их многообразие подразделяется на:

- литералы;
- переменные;
- массивы;
- функции;
- объекты.

При этом все они делятся на встроенные и определяемые программистом. Функции и объекты будут рассмотрены в следующей лекции.

Литералы

Литералом называют данные, которые используются в программе непосредственно. При этом под данными понимаются числа или строки текста. Все они рассматриваются в JavaScript как элементарные *типы данных*. Приведем примеры литералов:

```
числовой литерал: 10
числовой литерал: 2.310
числовой литерал: 2.3e+2
строковый литерал: 'Это строковый литерал'
строковый литерал: "Это строковый литерал"
```

Литералы используются в операциях присваивания значений переменным или в операциях сравнения:

```
var a=10;
var str = 'Строка';
if(x=='test') alert(x);
```

Оператор присваивания (`переменная = выражение`) возвращает результат вычисления `выражения`, поэтому ничто не мешает полученное значение присвоить еще и другой переменной. Таким образом, последовательность операторов присваивания выполняется справа налево:

```
result = x = 5+7;
```

Два варианта строковых литералов необходимы для того, чтобы использовать вложенные строковые литералы. Если в строковом литерале требуется использовать одинарную кавычку, то сам литерал можно заключить в двойные кавычки: `"It's cool!"`. Верно и обратное. Но если есть необходимость использовать в строковом литерале оба вида кавычек, то проще всего всех их "экранировать" символом обратной косой черты `\`, при этом саму строку можно заключить в любую пару кавычек. Например:

```
команда:
document.write("It\'s good to say \"Hello\" to someone!");
выдаст:
It's good to say "Hello" to someone!
```

Помимо строковых литералов (последовательностей символов, заключенных в кавычки) есть еще строковые *объекты*; они создаются конструктором: `var s = new String()`. У этого объекта существует много методов (об объектах и методах пойдет речь в следу-

ющей лекции). Следует понимать, что строковый литерал и строковый объект — далеко не одно и то же. Но зачастую мы этого не замечаем, т.к. при применении к строчным литералам методов строчных объектов происходит преобразование первых в последние.

Например, можно сначала присвоить `var s='abra-kadabra'`, а затем применить метод: `var m=s.split('b')`, который неявно преобразует строковый литерал `s` в строковый объект и затем разбивает строку в тех местах, где встречается подстрока `'b'`, возвращая массив строк-кусков: массив `m` будет состоять из строк `'a'`, `'ra-kada'` и `'ra'` (массивы рассматриваются ниже).

Переменные

Переменная — это область памяти, имеющая свое имя и хранящая некоторые данные. Переменные в JavaScript объявляются с помощью оператора `var`, при этом можно давать или не давать им начальные значения:

```
var k;
var h='Привет!';
```

Можно объявлять сразу несколько переменных в одном операторе `var` (тем самым уменьшая размер кода), но тогда их надо писать через **запятую**. При этом тоже можно давать или не давать начальные значения:

```
var k, h='Привет!';
var t=37, e=2.71828;
```

Тип переменной определяется по присвоенному ей значению. Язык JavaScript — слабо типизирован: в разных частях программы можно присваивать одной и той же переменной значения различных типов, и интерпретатор будет "на лету" менять тип переменной. Узнать тип переменной можно с помощью оператора `typeof()`:

```
var i=5;           alert(typeof(i));
i= new Array();    alert(typeof(i));
i= 3.14;           alert(typeof(i));
i= 'Привет!';      alert(typeof(i));
i= window.open(); alert(typeof(i));
```

Переменная, объявленная оператором `var` вне функций, является ***глобальной*** — она "видна" всюду в скрипте. Переменная, объявленная оператором `var` внутри какой-либо функции, является ***локальной*** — она "видна" только в пределах этой функции. Подробнее о *функциях* будет рассказано в следующем разделе этой лекции.

Например, в следующем фрагменте ничего не будет выведено на экран, несмотря на то, что мы обращаемся к переменной `k` после описания функции, и даже после ее вызова:

```
function f()
{ var k=5; }

f(); alert(k);
```

Причина в том, что переменная `k` является локальной, и она существует только в процессе работы функции `f()`, а по окончании ее работы уничтожается.

Если имеется *глобальная* переменная `k`, а внутри функции объявляется *локальная* переменная с тем же именем (оператором `var k`), то это будет другая переменная, и изменение ее значения внутри функции никак не повлияет на значение глобальной пере-

менной с тем же именем. Например, этот скрипт выдаст 7, поскольку вызов функции f() не затронет значения глобальной переменной k:

```
var k=7;

function f()
{ var k=5; }

f(); alert(k);
```

То же касается и аргументов при описании функций (с той лишь разницей, что перед ними не нужно ставить var): если имеется *глобальная* переменная k, и мы опишем функцию function f(k) {...}, то переменная k внутри {...} никак не связана с одноименной глобальной переменной. В этом плане JavaScript не отличается от других языков программирования.

Примечание. Объявлять переменные можно и без оператора var, просто присваивая переменной начальное значение. Так зачастую делают с переменными циклов. В следующем примере, даже если переменная i не была объявлена ранее, все будет работать корректно:

```
for(i=0; i<8; i++) { ... }
```

Однако опускать var не рекомендуется. Во-первых, это нарушает ясность кода: если написано i=5, то непонятно, вводится ли здесь новая переменная или меняется значение старой. Во-вторых, и это главное, нужно помнить следующий момент, часто приводящий к неправильной работе программы.

Вне функций объявление переменной без оператора var равносильно объявлению с оператором var — в обоих случаях переменная будет глобальной. Внутри же функции объявление переменной без оператора var делает переменную *глобальной* (а не локальной, как можно было бы предположить), и значит, ее значение могут "видеть" и менять другие функции или операторы вне этой функции. При этом такая переменная становится глобальной не после описания, а после вызова этой функции. Пример:

```
function f()
{ var i=5; k=7; }

f(); alert(k);
```

В приведённом примере после i=5 стоит точка с запятой (а не запятая). Значит, k=7 — это отдельное объявление переменной, уже без оператора var. Поэтому переменная k "видна" снаружи и ее значение (7) будет выведено оператором alert(k). Чтобы скрыть переменную k, нужно было после i=5 поставить запятую.

Рассмотрим другой пример, показывающий, к каким неожиданным последствиям может привести отсутствие var при описании переменной:

```
function f(i)
{ k=7;
  if(i==3) k=5;
  else { f(3); alert(k); }
}
f(0);
```

Мы вызываем f(0), переменной присваивается значение k=7, далее выполнение функции идет по ветке else — и оператор alert(k) выдает 5 вместо ожидавшегося 7. Причина в том, что вызов f(3) в качестве "побочного эффекта" изменил значение k. Чтобы

такого не произошло, нужно перед `k=7` поставить `var`. Тогда переменная `k` будет локальной и вызов `f(3)` не сможет ее изменить, так как при вызове функции создаются новые копии всех ее локальных переменных.

При написании больших программ подобные ошибки трудно отследить, поэтому настоятельно рекомендуется все переменные объявлять с оператором `var`, особенно внутри функций.

Массивы

Массивы делятся на встроенные (`document.links[]`, `document.images[]` и т.п. — их еще называют *коллекциями*) и определяемые пользователем (автором документа). Коллекции будут обсуждаться в следующей лекции. Здесь же мы подробно остановимся на массивах, определяемых пользователем. Для массивов определено несколько методов: `join()`, `reverse()`, `sort()` и другие, а также свойство `length`, которое позволяет получить число элементов массива.

Для определения массива пользователя существует специальный конструктор `Array`. Если ему передается единственный аргумент, причем целое неотрицательное число, то создается незаполненный массив соответствующей длины. Если же передается один аргумент, не являющийся числом, либо более одного аргумента, то создается массив, заполненный этими элементами:

```
a = new Array();
// пустой массив (длины 0)

b = new Array(10);
// массив длины 10

c = new Array(10,'Привет');
// массив из двух элементов: числа и строки

d = [5, 'Тест', 2.71828, 'Число e'];
// краткий способ создать массив из 4 элементов
```

Элементы массива нумеруются с нуля. Поэтому в последнем примере значение `d[0]` равно `5`, а значение `d[1]` равно `'Тест'`. Как видим, массив может состоять из разнородных элементов. Массивы не могут быть многомерными, однако ничто не мешает завести массив, элементами которого будут тоже массивы.

Метод join()

Метод `join()` позволяет объединить элементы массива в одну строку. Он является обратным к рассмотренному выше методу `split()`, который разрезает объект типа `String` на куски и составляет из них массив. Кстати, метод `split()` демонстрирует тот факт, что массив можно получить и без конструктора массива.

Рассмотрим пример преобразования локального URL в глобальный URL, где в качестве адреса сервера будет выступать `www.intuit.ru`. Пусть в переменной `localURL` хранится локальный URL некоторого файла:

```
localURL = "file:///C:/department/internet/js/2/2.html"
```

Разрежем строку в местах вхождения комбинации символов "`:/`", выполнив команду: `b = localURL.split(':/')`. Получим массив:

```
b[0] = "file";
b[1] = "//C";
b[2] = "department/internet/js/2/2.html";
```

Заменяем 0-й и 1-й элементы на требуемые:

```
b[0] = "http:";
b[1] = "/www.intuit.ru";
```

Наконец, склеиваем полученный массив, вставляя косую черту в местах склейки: `globalURL = b.join("/")`. В итоге мы получаем требуемый глобальный URL — значение `globalURL` будет равно: `http://www.intuit.ru/department/internet/js/2/2.html`.

Метод reverse()

Метод `reverse()` применяется для изменения порядка элементов массива на противоположный. Предположим, массив упорядочен по возрастанию:

```
a = new Array('мать','видит','дочь');
```

Упорядочим его обратном порядке, вызвав метод `a.reverse()`. Тогда новый массив `a` будет содержать:

```
a[0]='дочь';
a[1]='видит';
a[2]='мать';
```

Метод sort()

Метод `sort()` интерпретирует элементы массива как **строковые литералы** и сортирует массив в **алфавитном** (т.н. лексикографическом) порядке. Обратите внимание: метод `sort()` меняет массив. В предыдущем примере, применив `a.sort()`, мы получим на выходе:

```
a[0]='видит';
a[1]='дочь';
a[2]='мать';
```

Однако, это неудобно, если требуется отсортировать числа, поскольку согласно алфавитному порядку `40` идет раньше чем `5`. Для этих целей у метода `sort()` имеется необязательный аргумент, являющийся именем функции, согласно которой требуется отсортировать массив, т.е. в этом случае вызов метода имеет вид: `a.sort(myfunction)`. Эта функция должна удовлетворять определенным требованиям:

- у нее должно быть ровно два аргумента;
- функция должна возвращать число;
- если первый аргумент функции должен считаться меньшим (большим, равным) чем второй аргумент, то функция должна возвратить отрицательное (положительное, нуль) значение.

Например, если нам требуется сортировать числа, то мы можем описать следующую функцию:

```
function compar(a,b)
{
 if(a < b)  return -1;
 if(a > b)  return  1;
```

```
 if(a == b) return  0;
}
```

Теперь, если у нас есть массив `b = new Array(10,6,300,25,18);`, то можно сравнить результаты сортировки без аргумента и с функцией `compar` в качестве аргумента:

```
document.write("Алфавитный порядок:<BR>");
document.write(b.sort());
document.write("<BR>Числовой порядок:<BR>");
document.write(b.sort(compar));
```

В результате выполнения этого кода получим следующее:

```
Алфавитный порядок:
10,18,25,300,6
Числовой порядок:
6,10,18,25,300
```

Обратите внимание: метод `sort()` **интерпретирует** элементы массива как строки (и производит лексикографическую сортировку), но не **преобразует** их в строки. Если в массиве были числа, то они числами и останутся. В этом легко убедиться, если в конце последнего примера выполнить команду `document.write(b[3]+1)`: результат будет `26` (т.е. `25+1`), а не `251` (т.е. `"25"+1`).

Операторы языка

В этом разделе будут рассмотрены операторы JavaScript. Основное внимание при этом мы уделим операторам декларирования и управления потоком вычислений. Без них не может быть написана ни одна JavaScript-программа.

Общий перечень этих операторов выглядит следующим образом (сразу оговоримся, что этот список неполный):

- `{...}`
- `if ... else ...`
- `()?`
- `while`
- `for`
- `break`
- `continue`
- `return`

{...}

Фигурные скобки определяют составной оператор JavaScript — ***блок***. Основное назначение блока — определение тела цикла, тела условного оператора или функции.

if ... else ...

Условный оператор применяется для ветвления программы по некоторому логическому условию. Есть два варианта синтаксиса:

```
if (логическое_выражение) оператор_1;
if (логическое_выражение) оператор_1; else оператор_2;
```

Логическое выражение — это выражение, которое принимает значение `true` или `false`. В первом варианте синтаксиса: если `логическое_выражение` равно `true`, то выполняется указанный `оператор`. Во втором варианте синтаксиса: если `логическое_выражение` равно `true`, то выполняется `оператор_1`, если же оно равно `false` `оператор_2`. Пример использования (об объекте `navigator` читай лекцию "Программируем свойства окна браузера"):

```
if (navigator.javaEnabled())
 alert('Ваш браузер поддерживает Java');
else
 alert('Ваш браузер НЕ поддерживает Java');
```

()?

Этот оператор, называемый *условным выражением*, выдает одно из двух значений в зависимости от выполнения некоторого условия. Синтаксис его таков:

```
 (логическое_выражение)? значение_1 : значение_2
```

Если `логическое_выражение` равно `true`, то возвращается `значение_1`, в противном случае `значение_2`. Условное выражение легко имитируется оператором `if...else`, однако оно позволяет сделать более компактным и легко воспринимаемым код программы. Например, следующие два фрагмента равносильны:

```
 TheFinalMessage = (k>5)? 'Готово!' : 'Подождите...';

 if(k>5) TheFinalMessage = 'Готово!';
 else    TheFinalMessage = 'Подождите...';
```

while

Оператор `while` задает цикл. Определяется он в общем случае следующим образом:

```
while (условие_продолжения_цикла) тело_цикла;
```

Тело цикла может быть как простым, так и составным оператором. Составной оператор, как всегда, заключается в фигурные скобки. Рекомендуется и простой оператор заключать в них, чтобы программу можно было легко модифицировать. `Условие_продолжения_цикла` является логическим выражением. Тело исполняется до тех пор, пока верно логическое условие. Формально, цикл `while` работает следующим образом:

1. проверяется `условие_продолжения_цикла`:
    - если оно ложно (`false`), цикл закончен,
    - если же истинно (`true`), то продолжаем далее;
2. выполняется `тело_цикла`;
3. переходим к пункту 1.

Такой цикл используется, когда заранее неизвестно количество итераций, например, в ожидании некоторого события. Пример:

```
var s='';
while (s.length<6)
{
 s=prompt('Введите строку длины не менее 6:','');
}
alert('Ваша строка: ' + s + '. Спасибо!');
```

for

Оператор `for` — это еще один оператор цикла. В общем случае он имеет вид:

```
for (инициализация_переменных_цикла;
     условие_продолжения_цикла;
     модификация_переменных_цикла)  тело_цикла;
```

Тело цикла может быть как простым, так и составным оператором (составной необходимо заключать в фигурные скобки). Операторы `инициализация_переменных_цикла` и `модификация_переменных_цикла` могут состоять из нескольких простых операторов, в этом случае простые операторы должны быть разделены **запятой**. `Условие_продолжения_цикла` является логическим выражением. Цикл `for` работает следующим образом:

1. выполняется `инициализация_переменных_цикла`;
2. проверяется `условие_продолжения_цикла`:
    - если оно ложно (`false`), цикл закончен,
    - если же истинно (`true`), то продолжаем далее;
3. выполняется `тело_цикла`;
4. выполняется `модификация_переменных_цикла`;
5. переходим к пункту 2.

Рассмотрим типичный пример использования этого оператора:

```
document.write('Кубы чисел от 1 до 100:');

for (n=1; n<=100; n++)

 document.write('<BR>'+n+'<sup>3</sup> = '+ Math.pow(n,3));
```

Здесь `Math` — встроенный объект, предоставляющий многочисленные математические константы и функции, а `Math.pow(n,m)` вычисляет степенную функцию n^m. Результат работы скрипта получите самостоятельно.

break

Оператор `break` позволяет досрочно покинуть тело цикла. Возвращаясь к нашему примеру с кубами чисел, распечатаем только кубы, не превышающие `5000`.

```
document.write('Кубы чисел, меньшие 5000:');

for (n=1; n<=100; n++)
{
 s=Math.pow(n,3);
 if(s>5000) break;

 document.write('<BR>'+n+'<sup>3</sup> = '+s);
}
```

Несмотря на то, что переменную `n` мы заставили пробегать от `1` до `100`, т.е. заведомо с запасом, реально же цикл выполнится для значений `n` от `1` до ... получите сами!

continue

Оператор `continue` позволяет перейти к следующей итерации цикла, пропустив выполнение всех нижестоящих операторов в теле цикла. Если нам нужно вывести кубы чисел от `1` до `100`, превышающие `10 000`, то мы можем составить такой цикл:

```
document.write('Кубы чисел от 1 до 100, большие 10 000:');

for (n=1; n<=100; n++)
{
 s=Math.pow(n,3);
 if(s <= 10000) continue;

 document.write('<BR>'+n+'<sup>3</sup> = '+s);
}
```

Проверьте самостоятельно, кубы каких чисел будут выведены скриптом. Разумеется, для большей гибкости можно использовать в циклах оба оператора `break` и `continue`.

return

Оператор `return` используют для возврата значения из *функции* или обработчика события. Рассмотрим пример с функцией:

```
function sign(n)
{
 if (n>0) return 1;
 if (n<0) return -1;
 return 0;
}

alert( sign(-3) );
```

Обратите внимание: оператор `return` не только указывает, какое значение должна вернуть функция, но и прекращает выполнение дальнейших операторов в теле функции.

При использовании в обработчиках событий оператор `return` позволяет отменить или не отменять действие по умолчанию, которое совершает браузер при возникновении данного события. Отменить его, однако, можно не для всех событий. Рассмотрим пример:

```
<FORM ACTION="newpage.html" METHOD=post>
<INPUT TYPE=submit VALUE="Отправить?"
onClick="alert('Не отправим!');return false;">
</FORM>
```

В этом примере без оператора `return false` пользователь увидел бы окно предупреждения "Не отправим!" и далее был бы перенаправлен на страницу `newpage.html`. Оператор же `return false` позволяет отменить отправку формы, и пользователь лишь увидит окно предупреждения.

Аналогично, чтобы отменить действие по умолчанию для событий `onClick`, `onKeyDown`, `onKeyPress`, `onMouseDown`, `onMouseUp`, `onSubmit`, `onReset`, нужно использовать `return false`. Для события `onMouseOver` с этой же целью нужно использовать оператор `return true`. Для некоторых же событий, например `onMouseOut`, `onLoad`, `onUnload`, отменить действие по умолчанию невозможно.