Пользовательские объекты

Объекты реального мира окружают нас повсюду, и каждый из них имеет атрибуты и модель поведения, которые мы можем описать:

- Атрибуты — это особенности, характеризующие объект.
- Поведение — это действия, которые могут выполняться над объектом или которые может выполнять сам объект.

Например, объект автомобиль можно описать с помощью атрибутов «марка» и «модель», а также модель поведения «ускорение» и «торможение».

Эти элементы могут быть представлены в JavaScript с помощью пользовательского объекта **car**, содержащего свойства **make** и **model**, а также методы **accelerate( )** и **brake( )**.

Пробелы игнорируются в списке пар name: value (имя: значение), но не забывайте эти пары разделять запятыми.

Значения присваиваются объекту в виде разделенного запятыми списка пар **name: value** (имя: значение), заключенного в фигурные скобки **{ }**, например:

```
let car = { make: 'Jeep' , model: 'Wrangler' ,
   accelerate: function ( ) { return this.model + ' drives away' } ,
   brake: function ( ) {return this.make + ' pulls up' }
}
```

Доступ к свойству объекта можно получить двумя способами: с помощью точечной записи objectName.propertyName или objectName[‘propertyName’]

Методы объекта могут быть вызваны с помощью точечной нотации **objectName.methodName( )**.

Ключевое слово **this** относится к объекту, к которому принадлежит. В нашем примере оно относится к объекту **car**, поэтому, используя запись **this.model**, мы ссылаемся на свойство **car.model**, а **this.make** — на свойство **car.make**.

1. Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную, содержащую определение объекта.

object.html

```
let car = {
   // Здесь будет ваш код.
}
```

2. Для определения свойств объекта вставьте следующие операторы.

```
make: 'Jeep' ,
model: 'Wrangler' ,
```

3. Для определения методов объекта вставьте следующие операторы.

```
accelerate: function ( ) {
   return this.model + ' drives away' } ,
brake: function ( ) {
   return this.make + ' pulls up' }
```

4. Для вывода строки, содержащей значения свойств объекта, добавьте после символа закрывающей фигурной скобки } следующий оператор.

Для вызова метода необходимо добавить круглые скобки ( ). В противном случае метод вернет определение функции.

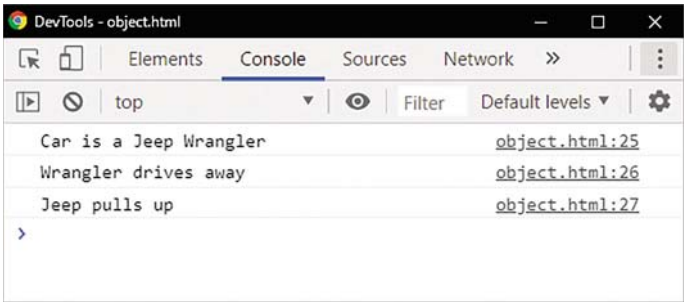
```
console.log( 'Car is a ' + car.make + ' ' + car[ 'model' ] )
```

5. Для вызова каждого метода объекта добавьте следующие операторы.

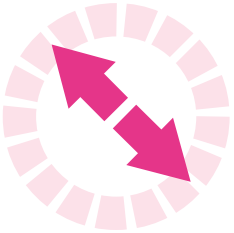
```
console.log( car.accelerate( ) )
console.log( car.brake( ) )
```

6. Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные

результаты — значения свойств объекта и возвращаемые методами строки.



Расширенные функции



Пользовательские объекты могут быть в любое время легко расширены и изменены. Для этого нужно всего лишь назначить новое значение с помощью точечной нотации, например: ***имяОбъекта.имяСвойства.***

Специальный цикл **for in** проходит через все перечисляемые свойства объекта, по каждому отдельному элементу и имеет следующий синтаксис:

for (*свойство* in *имяОбъекта*) { console.log(*свойство*) }

Чтобы на каждой итерации ссылаться на значение каждого свойства, его имя следует указать после имени объекта в квадратных скобках, например: ***имяОбъекта*[*свойства*].**

extend.html

1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную, которая точно воссоздает объект из предыдущего примера.

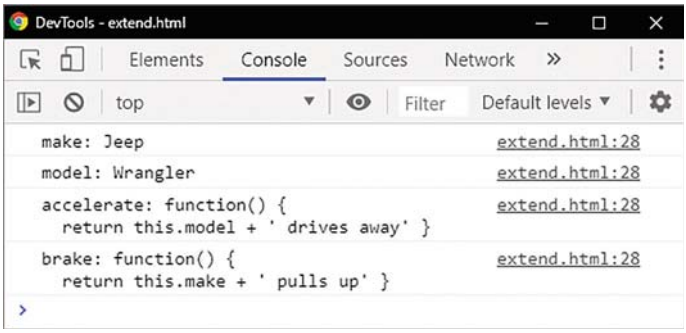
```
let property, car = {
   make: 'Jeep' ,
   model: 'Wrangler' ,
   accelerate: function ( ) {
      return this.model + ' drives away' },
   brake: function ( ) {
      return this.make + ' pulls up' }
}
```

2 Добавьте оператор цикла для перечисления имен и значений каждого свойства и метода.

```
for( property in car ) {
  console.log( property + ': ' + car[ property ] )
}
```

Фактически в этом примере свойству может быть присвоено любое допустимое имя.

3 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — ключи и значения объекта.



4 Добавьте операторы для присвоения новых значений двум существующим свойствам объекта.

```
car.make = 'Ford'
car.model = 'Bronco'
```

5 Для присвоения объекту дополнительных свойств и методов добавьте следующие операторы.

```
car.engine = 'V6'
car.start = function ( ) {
  return this.make + ' motor is running' }
```

Чтобы включить в вывод новую строку (разрыв строки), в этом примере используется escape-последовательность \n.

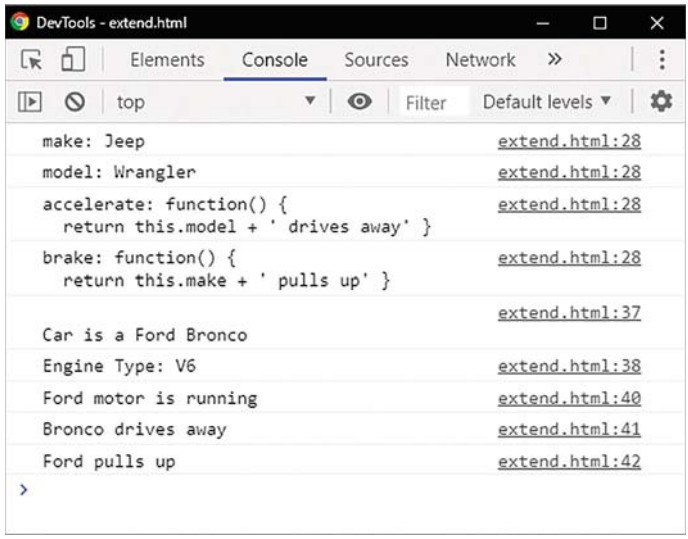
6 В соответствии с правилами добавьте к выходным строкам, содержащим значения свойств объекта, следующие операторы.

```
console.log( '\nCar is a ' + car.make + '
                                    ' + car[ 'model' ] )
console.log( 'Engine Type: ' + car.engine )
```

7 Добавьте операторы для вызова каждого метода объекта.

```
console.log( car.start( ) )
console.log( car.accelerate( ) )
console.log( car.brake( ) )
```

8 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — значения свойств расширенного объекта и возвращаемые его методами строки.



Встроенные объекты

В таблице ниже перечислены встроенные стандартные объекты JavaScript. Каждый из них, за исключением объекта **Math**, имеет одноименный конструктор, который можно использовать для создания объекта с помощью ключевого слова **new**. Например, создание объекта **Date** выглядит так: **let now=new Date( )**.

Однако не рекомендуется для всех типов объектов использовать ключевое слово **new** и конструктор, кроме объектов **Error** и **Date**. JavaScript распознает,

какой тип объекта может быть создан с помощью присвоенного значения, только если это значение не строка **string**, число **number** или логическое значение **boolean**. Такие типы данных называются примитивами. Они не имеют свойств или методов, поэтому оператор **typeof** возвращает для них значения **string**, **number** и **boolean**, а не **object**.

Все объекты в JavaScript наследуют свойства и методы из прототипа объекта высокого уровня **Object.prototype**. Например, с объектом более низкого уровня, таким как **Date.prototype**, это осуществляется методом **toLocaleString( )**, поэтому, чтобы получить строку даты в локальном формате, вы можете к объекту **Date** добавить вызов метода **toLocaleString( )**.

Также можно вызывать унаследованные методы для примитивных строковых (**string**), числовых (**number**) и логических (**boolean**) значений, так как JavaScript автоматически вызывает метод из эквивалентного объекта. Значит, чтобы получить числовую строку в локальном формате, вы можете к числовому литералу добавить вызов метода **toLocaleString( )**.

Если вы с самого начала работаете с этой книгой, то уже должны были изучить объекты String, Number, Boolean, Error и Object. Объект Array описан на стр. 81, объект Date — на стр. 90, объект RegExp — на стр. 99, а объект Math — на стр. 106.

Объект
String (создается с помощью ключевого слова new)
Number (создается с помощью ключевого слова new)
Boolean (создается с помощью ключевого слова new)
Object — определенный вами объект
Date — используется для работы с датой и временем
Array — используется для хранения упорядоченных коллекций данных
RegExp — создает объект регулярного выражения для сопоставления текста с шаблоном
Math — встроенный объект, хранящий в своих свойствах и методах различные математические константы и функции
Error — создает объект ошибки

extend.html

1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные и присвойте им примитивные литеральные значения.

```
let jsString = 'Text' // неверно — new String( 'Text' ).

let jsNumber = 125000       // неверно — new
                            Number( 125000 ).

let jsBoolean = true        // неверно — new
                            Boolean( true ).
```

Объект Math — единственный объект JavaScript, который не является конструктором.

2 Для создания объектов присвойте следующие значения.

```
let jsObject = {firstName: 'Mike' , lastName:  'McGrath' }
let jsDate = new Date( )
let jsArray = [ 1, 2, 3 ]
let jsRegExp = /ineasysteps/i
let jsMath = Math
let jsError = new Error( 'Error!' )
```

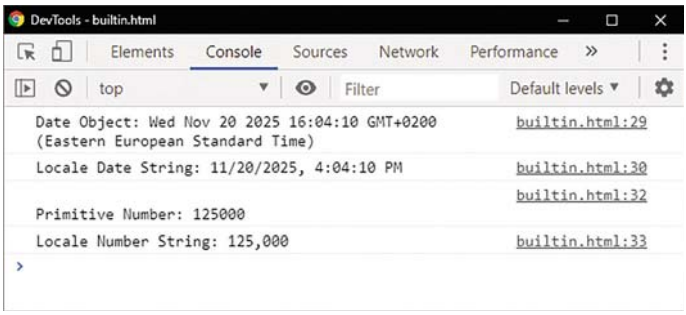
3 Добавьте операторы для вывода содержимого объекта **Date** и строки в локальном формате.

```
console.log( 'Date Object: ' + jsDate )
console.log( 'Locale Date String: ' +
                                jsDate.toLocaleString( ) )
```

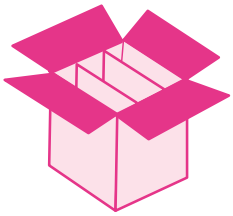
4 Добавьте операторы для вывода примитивного числового значения и числовой строки в локальном формате.

```
console.log( '\nPrimitive Number: ' + jsNumber )
console.log( 'Locale Number String: ' +
                                jsNumber.toLocaleString( ) )
```

5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Создание массивов



Объект **Array** — это встроенный объект JavaScript, который используется для хранения упорядоченных коллекций данных (с различными типами данных). Объекты массивов создаются путем присвоения переменным литеральных значений, указанных в квадратных скобках **[]** и разделенных запятыми. Синтаксис объекта **Array** выглядит следующим образом:

let *имя массива* = [*значение1* , *значение2* , *значение3*]

В отличие от пользовательских объектов, где каждое свойство содержит имя, элементы массива нумеруются автоматически, начиная с нуля. Первому элементу присваивается 0, второму — 1, третьему — 2 и т.д. Эту систему нумерации часто называют «индексом с отсчетом от нуля».

Доступ к элементам массива осуществляется с помощью оператора **[ ]**. Например, **colors[0]** ссылается на значение в первом элементе массива с именем «colors».

Можно создать пустой массив и позже присвоить значения его элементам, например:

```
let colors = [ ]
colors[ 0 ] = 'Red'
colors[ 1 ] = 'Green'
colors[ 2 ] = 'Blue'
```

Все имена встроенных объектов начинаются с символа верхнего регистра, поэтому следует различать «Array» и «array».

Можно создать массив, используя конструктор **Array( )**. Однако такой способ использовать нежелательно, так как это может привести к неожиданным результатам. Например, создание массива, инициализирующего первый элемент:

```
let jsArray = new Array( 10 )
```

Конечно, вы можете ожидать, что **jsArray[0]** будет ссылаться на целочисленное значение 10 первого элемента, но на самом деле он возвращает **неопределенное** значение. Почему? Это аномалия, которая возникает только тогда, когда вы указываете один целочисленный аргумент для конструктора, что заставляет JavaScript создавать массив из 10 пустых элементов! Создание массива с помощью **jsArray = [10]** не дает такого эффекта и создает массив с первым элементом, содержащим целочисленное значение 10, как и предполагалось.

array.html

1. Создайте HTML-документ с самовызывающейся функцией, которая начинается с создания массива — неверно.

```
let jsArray = new Array( 10 )
```

2. Для вывода значения первого элемента массива и перечисления массива добавьте следующие операторы.

```
console.log( jsArray[ 0 ] )
console.log( jsArray )
```

3. Для объявления переменной и переменной, инициализированной массивом, добавьте следующие операторы.

```
let month, summer = [ 'June' , 'July' , 'August' ]
```

4. Добавьте циклическую структуру для вывода номера индекса и значения каждого элемента.

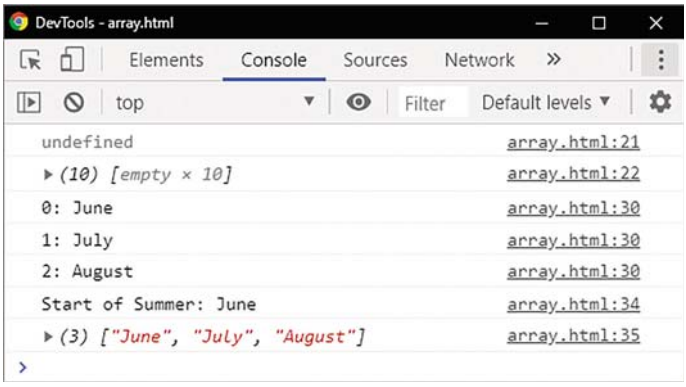
```
for ( month in summer )
{
  if ( month !== ' ' )
  {
    console.log( month + ': ' + summer[ month ] )
  }
}
```

Рекомендуется заключать тело цикла for in в оператор if. Это гарантирует, что элемент не пустой.

5 Наконец, для вывода значения первого элемента массива и перечисления массива добавьте следующие операторы.

```
console.log( 'Start of Summer: ' + summer[ 0 ] )
console.log( summer )
```

6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — вывод содержимого элементов массива.



Обход элементов в цикле

Как правило, при работе с массивами используются циклы. Для заполнения элементов массива значениями можно использовать любой из них. Таким образом можно заполнить элементы даже очень больших массивов, используя при этом удивительно простой код.

Также их используют для быстрого считывания значений каждого элемента массива и выполнения каких-либо действий с этим значением.

Чтобы узнать длину массива, необходимо обратиться к его свойству **length**. В результате индексирования с отсчетом от нуля длина массива всегда будет на единицу больше, чем номер индекса последнего элемента, поэтому его можно легко использовать в условии для завершения цикла.

elements.html

1. Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
let i, result, boolArray = [ ]
```

2. Выведите простой заголовок.

```
console.log( 'Fill Elements...')
```

3. Добавьте цикл для заполнения 10 элементов массива логическими значениями и вывода их порядковых номеров и каждого сохраненного значения.

```
for( i = 1; i < 11; i++ )
{
  boolArray[ i ] = ( i % 2 === 0 ) ? true : false
  console.log( 'Element ' + i + ': ' + boolArray[ i ] )
}
```

4. Выведите еще один заголовок и присвойте переменной пустое строковое значение.

```
console.log( 'Read Elements...' )
result = ''
```

5. Добавьте цикл для присвоения индексных номеров любых элементов, содержащих значение **true**.

```
for( i = 1 ; i < boolArray.length ; i++ )
{
 if( boolArray[ i ] ) { result += i + ' | ' }
}
```

6. Выведите строку, чтобы отобразить порядковые номера элементов, содержащих значение **true**.

```
console.log( 'True in Elements: ' + result )
```

Длина этого массива — 11, так как он состоит из 11 элементов, даже если нулевой элемент пустой.

7 Присвойте строковой переменной пустое значение.

```
result = ''
```

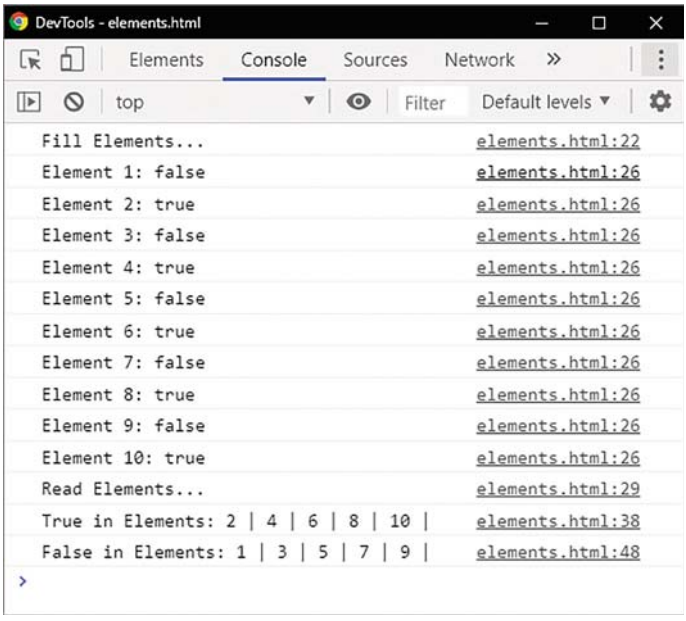
8 Добавьте цикл для присвоения индексных номеров любых элементов, содержащих значение **false**.

```
for( i = 1 ; i < boolArray.length ; i++ )
{
  if( !boolArray[ i ] ) { result += i + ' | ' }
}
```

9 Наконец, выведите строку, чтобы отобразить порядковые номера элементов, содержащих значение **false**.

```
console.log( 'False in Elements: ' + result )
```

10 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — запись и чтение элементов массива.



Условия для логического значения не должны включать инструкцию ===true, так как цикл выполняется автоматически.

Методы управления элементами в массиве

Объекты JavaScript имеют свойства и методы. В дополнение к свойству **length** у каждого объекта массива имеются методы, которые можно использовать для управления элементами в массиве. Ниже в таблице перечислены такие методы и их краткое описание.

Метод	Описание
join(separator)	Объединяет все элементы массива в одну строку и разделяет их запятыми.
pop( )	Удаляет последний элемент из массива и возвращает его значение.
push(*value* , *value*)	Добавляет один и более элементов в конец массива и возвращает новую длину массива.
reverse( )	Обращает порядок следования элементов массива. Первый элемент становится последним, а последний — первым.
shift( )	Удаляет первый элемент из массива и возвращает его значение.
slice(*begin* , *end*)	Возвращает новый массив, содержащий копию части исходного массива.
sort()	Сортирует элементы массива и возвращает отсортированный массив.
splice(*position* , *number* , *value* , *value*)	Изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые.
unshift(*value* , *value*)	Добавляет один и более элементов в начало массива и возвращает новую длину массива.

Метод join( ) объединяет множество элементов в одну строку. Оператор + объединяет несколько значений элементов.

Метод slice( ) возвращает новый массив, который содержит копии элементов, вырезанные из исходного массива.

Если значения не определены методами **push( )** или **unshift( )**, в массив добавляется один пустой элемент. Чтобы изменить несколько элементов, список значений, разделенных запятыми, может быть указан в методах **push( )**, **unshift( )** и **splice( ).**

1. Создайте HTML-документ с самовызывающейся функцией и объявите массив.

```
let seasons = [ 'Spring', 'Summer', 'Fall', 'Winter' ]
console.log( 'Elements: ' + seasons )
```

2. Выведите измененный список элементов.

```
console.log( 'Joined: ' + seasons.join(' & ') )
```

3. Извлеките последний элемент из массива.

```
console.log( 'Popped: ' + seasons.pop( ) )
console.log( 'Elements: ' + seasons )
```

4. Верните последний элемент обратно в массив.

```
console.log( 'Pushed: ' + seasons.push( 'Winter' ) )
console.log( 'Elements: ' + seasons )
```

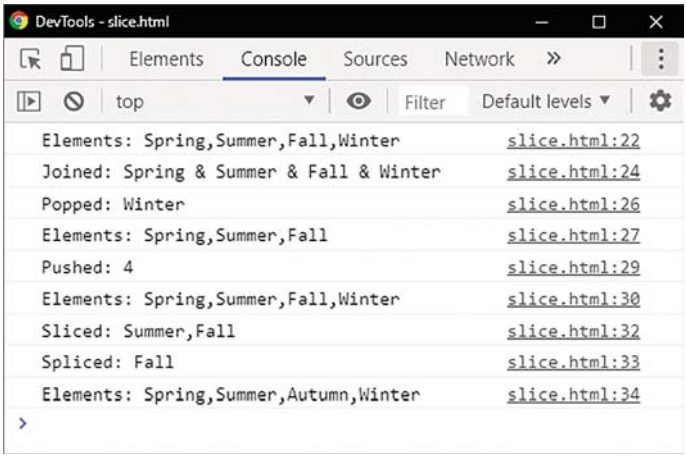
5. Затем выведите только два значения элемента.

```
console.log( 'Sliced: ' + seasons.slice( 1, 3 ) )
```

6. Замените значение в третьем элементе.

```
console.log( 'Spliced: ' + seasons.splice( 2, 1, 'Autumn' ) )
console.log( 'Elements: ' + seasons )
```

7. Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



slice.html

Чтобы удалить указанное количество элементов в указанной позиции, используйте метод slice( ) без значения замены. Нумерация всех оставшихся элементов, которые следуют в этом массиве, автоматически изменится.

Методы shift() и unshift() работают аналогично методам pop() и push(), но с первым элементом, а не с последним. Методы reverse() и sort() рассматриваются в следующем примере на стр. 88–89.

Сортировка элементов массива

Иногда возникает необходимость расположить значения элементов массива в определенном порядке. Метод **sort()** сортирует элементы массива и возвращает отсортированный массив. При необходимости можно указать аргумент функции, определяющий порядок сортировки.

Всегда помните, что при использовании метода sort( ) меняется порядок значений, хранящихся в элементах массива.

Если функция сравнения не указана, элементы сортируются путем преобразования их в строки и сравнения строк в порядке следования кодовых точек Unicode, сравнивая каждый первый символ, затем каждый второй символ и т. д. Если элементы содержат совпадающие строки, различающиеся только регистром символов, строка с наибольшим количеством символов верхнего регистра получает нижнюю позицию индекса, а перед ней появляется строка с меньшим количеством символов верхнего регистра.

Такой вид сортировки подходит для строковых значений, но совершенно не годится для числовых. Например, при сортировке трех значений 30, 100, 20 результат будет 100, 20, 30, так как первые символы разные, значения сортируются только при их сравнении. Как правило, необходимо, чтобы все числовые значения были отсортированы в порядке возрастания или убывания. Поэтому для определения порядка сортировки в методе **sort()** необходимо указать функцию сравнения.

Работа метода sort( ) по умолчанию эквивалентна функции сравнения, сравнивающей аргументы x и y со следующими операторами: if(x > y) return 1 else if(x < y) return -1 else return 0.

Если функция сравнения указана, элементы массива будут сортироваться в соответствии с ее возвращенным значением. Если первое значение больше второго, то вернется значение больше 0, и первому значению будет присвоена более высокая позиция индекса. И наоборот, функция сравнения возвращает значение меньше 0, а первому значению присваивается более низкая позиция индекса. Если оба значения равны, то в результате возвращается 0, и позиции элементов остаются неизменными. Когда

все сравнения будут выполнены, элементы расположатся в порядке возрастания значений. При необходимости сортировки элементов по убыванию можно использовать метод **reverse()**. Он обращает порядок следования элементов массива.

Если функция сравнения сравнивает числовые значения, то для получения необходимого результата следует вернуть результат вычитания второго и первого переданного значения.

slice.html

1. Создайте HTML-документ с самовызывающейся функцией и объявите два массива.

```
let hues = [ 'Red', 'RED', 'red', 'Green', 'Blue' ]
let nums = [ 1, 20, 3, 17, 14, 0.5 ]
```

2. Выведите значения каждого массива и значения их элементов, отсортированных по ключу.

```
console.log( 'Colors: ' + hues )
console.log( 'Dictionary Sort: ' + hues.sort( ) )
console.log( '\nNumbers: ' + nums )
console.log( 'Dictionary Sort: ' + nums.sort( ) )
```

3. Добавьте оператор для вывода числовых значений, отсортированных после вызова функции сравнения.

```
console.log( 'Numerical Sort: ' + nums.sort( sortNums ) )
```

4. Для вывода числовых значений в порядке убывания добавьте в функциональном блоке следующий оператор.

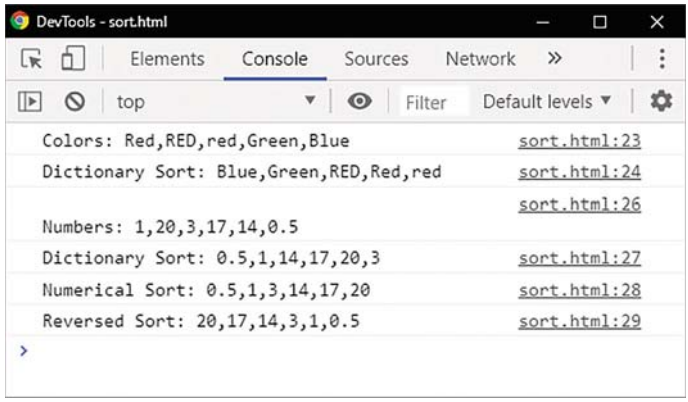
```
console.log( 'Reversed Sort: ' + nums.reverse( ) )
```

5. Добавьте функцию сравнения.

```
function sortNums( numOne, numTwo ) {
  return numOne - numTwo
}
```

В методе sort( ) функция сравнения указывается только по имени. Не используйте закрывающиеся скобки после имени функции сравнения в аргументе метода sort( ).

6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — отсортированные элементы.



Получение даты

Встроенный объект **Date** представляет конкретную дату, время и часовой пояс. Он создается с помощью оператора **new**, конструктора объекта **Date( )** и присвоения имени переменной. Конструктор **Date( )**, вызванный без аргументов, создаст объект **Date** со значением, которое будет соответствовать текущей дате и времени. Объект **Date** поддерживает ряд UTC (универсальных) методов, а также методов локального времени. UTC, также известный как среднее время по Гринвичу (GMT), относится к времени, установленному мировым стандартом времени.

Дата и время в JavaScript в основном определяются как количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 GMT, то есть прошедшего с начала эпохи UNIX. Это число получается из объекта **Date** с помощью метода **getTime( )** и может быть вычтено из числа другого объекта **Date** для вычисления прошедшего периода между двумя точками сценария. Например, для вычисления периода, необходимого для выполнения цикла.

Строковое представление объекта **Date** можно получить с помощью методов **toString( )** или **toUTCString( )**, преобразующего дату в строку с использованием часового пояса UTC.

С помощью метода **getTimezoneOffset( )** JavaScript может определить, в каком часовом поясе находится пользователь при условии, что в системе корректно настроен локальный часовой пояс. Этот метод возвращает смещение часового пояса относительно часового пояса UTC в минутах для текущей локали. Расчет выполняется в минутах, а не часах, так как некоторые часовые пояса смещены с интервалом, отличным от одного часа. Например, Ньюфаундленд, Канада — UTC –3:30 (UTC –2:30 в летнее время).

Значение смещения часового пояса можно использовать для локальных настроек часовых поясов США, но для перехода на летнее время их необходимо скорректировать посредством вычитания 60 (минут). Далее в примере демонстрируется вызов методов **getMonth( )** и **getDate( )** объекта **Date** для корректировки значения смещения часового пояса, если летнее время не действует в текущую дату.

На стр. 92–98 рассмотрены примеры, демонстрирующие, как использовать компоненты объекта **Date** и его распространенные методы.

1. Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
const now = new Date( )
let offset = now.getTimezoneOffset( )
let dst = 60
```

date.html

2. Добавьте операторы для отключения летнего времени с 3 ноября по 10 марта.

```
if( ( now.getMonth( ) < 3 ) && ( now.getDate( ) < 10 ) )
{ dst = 0 }
if( ( now.getMonth( ) > 9 ) && ( now.getDate( ) > 2 ) )
{ dst = 0 }
```

Метод **getMonth( )** возвращает месяц и дату по местному времени. Нумерация месяцев начинается с нуля. В нашем примере март находится на позиции 2. Подробнее об этом см. стр. 93.

3. Добавьте операторы, чтобы установить часовой пояс.

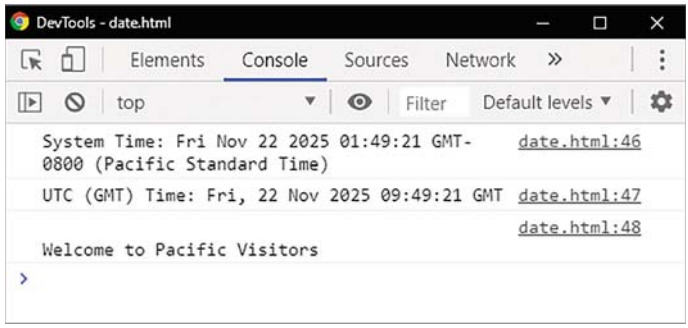
Локальный часовой пояс пользователя можно использовать для перенаправления браузера на страницу, относящуюся к этому часовому поясу. Например, пользователей в часовом поясе Тихого океана можно перенаправить на страницу, содержащую только дистрибьюторов из Калифорнии. Однако следует учитывать, что информация о системном времени может быть легко изменена пользователем на любое время, дату или часовой пояс, поэтому не обязательно сообщать фактическое местоположение.

```
switch( offset )
{
   case ( 300 - dst ) : offset = 'East Coast' ; break
   case ( 360 - dst ) : offset = 'Central' ; break
   case ( 420 - dst ) : offset = 'Mountain' ; break
   case ( 480 - dst ) : offset = 'Pacific' ; break
   default : offset = 'All'
}
```

4 Добавьте операторы для вывода информации о дате и времени и приветственное сообщение.

```
console.log( 'System Time: ' + now.toString( ) )
console.log( 'UTC (GMT) Time: ' + now.toUTCString( ) )
console.log( '\nWelcome to ' + offset + ' Visitors' )
```

5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Получение календаря

Объект **Date** предоставляет отдельные методы для получения следующих значений: год, месяц, день месяца и день недели.

Метод **getFullYear( )** объекта **Date** возвращает год в виде четырехзначного числа, например **2025**. Метод **getDate( )** объекта **Date** возвращает день месяца. В первый день месяца возвращается число **1**.

Метод	Описание
getFullYear( )	Возвращает год указанной даты по местному времени (уууу)
getMonth( )	Возвращает месяц указанной даты по местному времени (0–11)
getDate( )	Возвращает день указанной даты по местному времени (1–31)
getDay( )	Возвращает день недели указанной даты по местному времени (0–6).

В зависимости от региональных настроек методы **getMonth( )** и **getDay( )** возвращают числовые значения индекса, которые необходимо преобразовать в названия месяца и дня на местном языке. Такое преобразование легко выполнить для месяцев с помощью создания массива всех названий месяцев, начиная с января, а затем используя номер индекса, возвращаемого **getMonth( )**, для ссылки на соответствующее название месяца из элемента массива.

Аналогичным образом можно выполнить преобразование для дней недели с помощью создания массива названий всех дней недели, начиная с воскресенья, а затем с использованием номера индекса, возвращаемого методом **getDay( )**, для ссылки на соответствующее название дня из элемента массива.

Затем различные компоненты могут быть объединены в строку даты, упорядоченную в соответствии с необходимым форматом даты для любой локали.

1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

calendar.html

```
const days = [ 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat' ]
const months = [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec' ]
const now = new Date( )
```

2 Добавьте операторы для извлечения отдельных компонентов даты с помощью методов объекта **Date**.

```
let year = now.getFullYear( )
let month = now.getMonth( )
let dayNumber = now.getDate( )
let dayName = now.getDay( )
```

3 Добавьте операторы для преобразования извлеченных номеров индексов в значения названий месяца и дня недели.

```
month = months[ month ]
dayName = days[ dayName ]
```

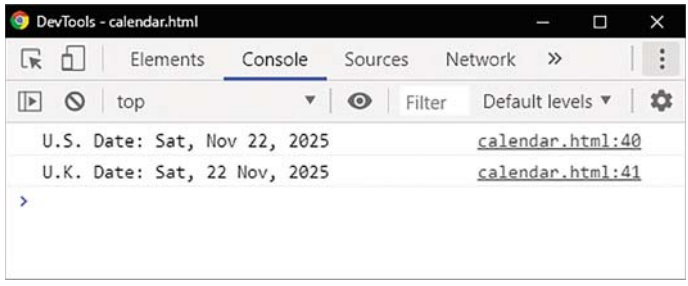
4 Объедините компоненты даты в строки даты — в американском и британском форматах.

```
let usDate = dayName + ', ' + month + ' ' +
                                    dayNumber + ', ' + year
let ukDate = dayName + ', ' +
                          dayNumber + ' ' + month + ', ' + year
```

5 Для вывода каждой строки даты добавьте следующие операторы.

```
console.log( 'U.S. Date: ' + usDate )
console.log( 'U.K. Date: ' + ukDate )
```

6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — форматы записи даты.



Нумерация месяцев начинается с нуля (0), а не с единицы (1) — так, например, месяцу март присвоен индекс [2], а не [3].

Получение времени

Объект JavaScript **Date** предоставляет отдельные методы для получения следующих значений: часы, минуты, секунды и миллисекунды.

Метод	Описание
getHours( )	Возвращает часы даты по местному времени (0–23)
getMinutes( )	Возвращает минуты даты по местному времени (0–59)
getSeconds( )	Возвращает секунды даты по местному времени (0–59)
getMilliseconds( )	Возвращает миллисекунды даты по местному времени (0–999)

Метод **getHours( )** объекта **Date** возвращает часы даты по местному времени в 24-часовом формате в виде значения в диапазоне 0–23. Методы **getMinutes( )** и **getSeconds( )** возвращают значение в диапазоне 0–59. Метод **getMilliseconds( )** возвращает значение в диапазоне 0–999.

Значения могут быть объединены в строку, однако для улучшения читабельности возникает необходимость добавлять ноль к значениям минут и секунд. Например, 10:05:02 предпочтительнее 10:5:2.

В зависимости от времени суток (утро, день или вечер) можно добавить аббревиатуры. В случае 12-часового формата времени используются сокращения AM или PM. Например, 13:00 можно преобразовать в 1:00 PM.

Значения времени основаны на системном времени пользователя, которое может быть неточным.

1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте пять переменных.

time.html

```
const now = new Date( )
let hour = now.getHours( )
let minute = now.getMinutes( )
let second = now.getSeconds( )
let millisecond = now.getMilliseconds( )
```

Объект Date также предоставляет методы для получения даты и времени по всемирному координированному времени (UTC) — например, методы getUTCMonth( ) и getUTCHours( ).

2 Для удобочитаемости с помощью следующих операторов добавьте 0 в значениях минут и секунд, значение которых меньше10.

```
if( minute < 10 ) { minute = '0' + minute }
if( second < 10 ) { second = '0' + second }
```

3 Объедините все составляющие времени в строку, затем выведите ее.

```
let time = 'It is now: ' + hour + ':' + minute + ':' +
           second + ' and ' + millisecond + ' milliseconds'
console.log( time )
```

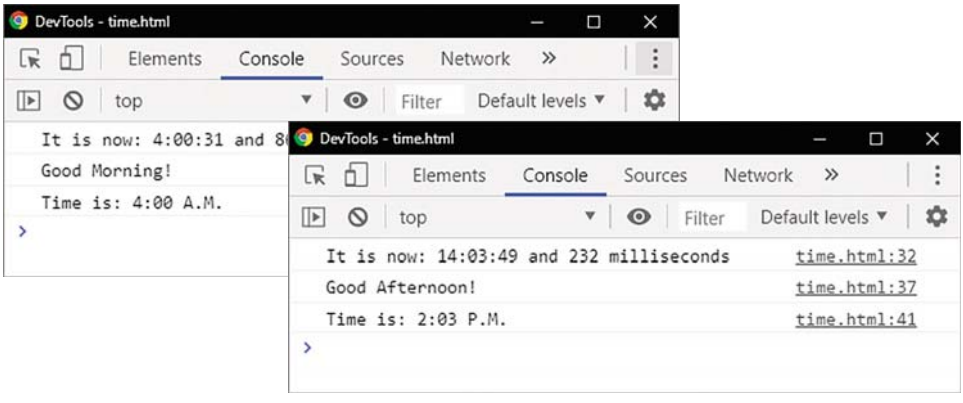
4 В зависимости от времени суток выведите приветствие.

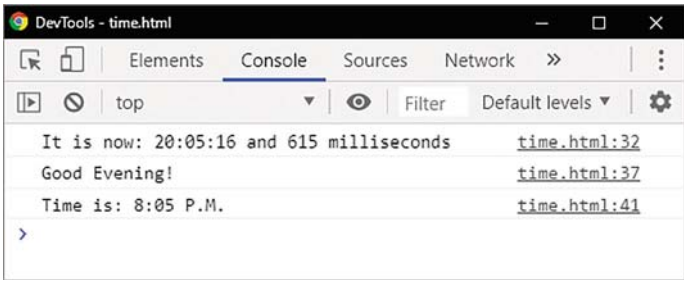
```
let greeting = 'Good Morning!'
if( hour > 11 ) { greeting = 'Good Afternoon!' }
if( hour > 17 ) { greeting = 'Good Evening!' }
console.log( greeting )
```

5 Выведите время в 12-часовом формате.

```
let suffix = ( hour > 11 )? ' P.M.' : ' A.M.'
if( hour > 12 ) { hour -= 12 }
console.log( 'Time is: ' + hour + ':' + minute + suffix )
```

6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — формат записи времени.





Установка даты и времени

Конструктор **Date( )** может дополнительно указывать от двух до семи аргументов для установки значений для каждого из его компонентов, например:

new Date(***год, месяц, день, часы, минуты, секунды, миллисекунды*** **)**

Если указаны только год и месяц, соответствующим компонентам устанавливается единица (1), а остальным — ноль (0).

Объект **Date** также предоставляет отдельные методы для установки значений даты и времени:

Метод	Описание
setDate( )	Устанавливает день (1–31)
setFullYear( )	Устанавливает год (yyyy)
setMonth( )	Устанавливает месяц (0–11)
setHours( )	Устанавливает часы (0–23)
setMinutes( )	Устанавливает минуты (0–59)
setSeconds( )	Устанавливает секунды (0–59)
setMilliseconds( )	Устанавливает миллисекунды (0–999)

Метод setMonth( ) устанавливает месяц в диапазоне, где 0 = январь — 11 = декабрь. При необходимости с помощью метода setFullYear( )можно установить месяц и день, используя следующий синтаксис:

date.setFullYear(***год , числоМесяца , числоДня*** **)**

Метод **toString( )** возвращает строковое значение любого объекта JavaScript.

Значения каждого компонента даты и времени могут быть установлены с помощью объекта **Date**. Кроме того, существуют методы для вывода различных строк, отображающих дату и время. Метод **toString( )** преобразует дату в строковое значение; метод **toUTCString( )** преобразует дату в ее эквивалент в формате UTC; метод **toLocaleString( )** отображает дату в зависимости от используемой локали. Методы **toDateString( )** и **toTimeString( )** могут отображать компоненты даты и времени.

setdate.html

1. Создайте HTML-документ с самовызывающейся функцией и создайте объект **Date** «4 июля».

```
const holiday = new Date( 2025, 6, 4 )
console.log( 'Object: ' + holiday )
```

2. Чтобы в полдень наступило Рождество, необходимо изменить отдельные компоненты даты. Для этого добавьте операторы.

```
holiday.setFullYear( 2028 )
holiday.setMonth( 11 )
holiday.setDate( 25 )
holiday.setHours( 12 )
holiday.setMinutes( 0 )
holiday.setSeconds( 0 )
holiday.setMilliseconds( 0 )
```

Также существует метод **setTime( )**, который устанавливает объект **Date** в значение времени, представленным количеством миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC. Каждый день — 86400 000 миллисекунд, поэтому **setTime(86400000)** устанавливает дату 1 января 1970 года.

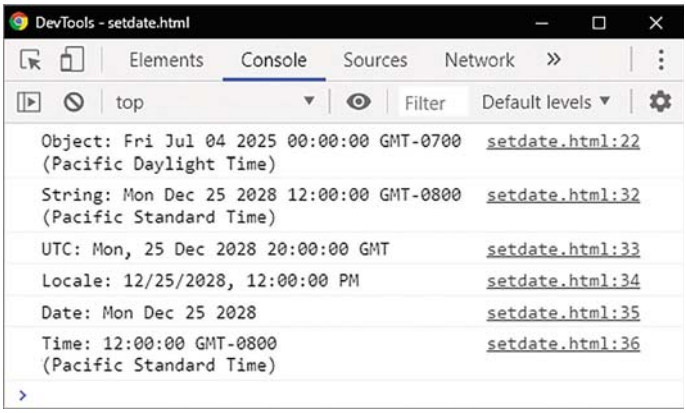
3. Для вывода измененной даты, времени и эквивалента в формате UTC (GMT) добавьте следующие операторы.

```
console.log( 'String: ' + holiday.toString( ) )
console.log( 'UTC: ' + holiday.toUTCString( ) )
```

4. Добавьте операторы для вывода измененной даты и времени в формате строк локали, даты и времени.

```
console.log( 'Locale: ' + holiday.toLocaleString( ) )
console.log( 'Date: ' + holiday.toDateString( ) )
console.log( 'Time: ' + holiday.toTimeString( ) )
```

5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — установленные дату и время.

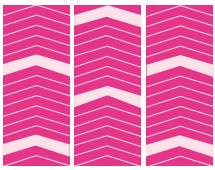


Сопоставление текста с шаблоном

Объект **RegExp** — это встроенный объект JavaScript, который создает объект регулярного выражения для сопоставления текста с шаблоном. Регулярные выражения используются как средство для поиска и замены текста при помощи шаблонов.

Шаблон регулярного выражения может состоять целиком из символьных литералов, заключенных между символами **//**. Например, регулярное выражение **/wind/** найдет совпадение в строке «windows» — шаблон строго совпадает со строкой в тексте. Как правило, шаблон регулярного выражения состоит из набора символьных литералов и специальных символов или метасимволов.

Для выполнения поиска без учета регистра шаблон может также включать модификатор **i** после последнего символа **/** или модификатор **g** для выполнения глобального поиска всех совпадений шаблона. Метод **test( )** выполняет поиск сопоставления регулярного выражения



Тема регулярных выражений обширна и выходит за рамки этой книги. Здесь на тему регулярных выражений приводятся лишь краткие сведения.

Диапазон символов [a-z] соответствует только строчным символам латинского алфавита, а диапазон [a-z0–9] включает еще и цифры.

указанной строке. При обнаружении совпадения метод **test( )** возвращает значение **true**, в противном случае — **false**. Метод **exec( )** выполняет поиск сопоставления регулярного выражения в указанной строке и возвращает массив с результатами или значение **null**. **Index** — индекс сопоставления в строке — начинается с нуля.

Специальный символ	Значение	Пример
.	Любой символ	ja..pt
^	Начало строки	^ja
$	Конец строки	... pt$
*	Повторение фрагмента 0 или более раз	ja*
+	Повторение фрагмента 1 или более раз	ja+
?	Фрагмент либо присутствует, либо отсутствует (0 или 1)	ja?
{}	Множественное повторение	ja{3}
[]	Класс символов	[a-z]
\	Специальная последовательность	\d
|	Фрагмент слева или фрагмент справа	a|b
( )	Группировка выражений	(...)

regexp.html

1. Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
const system = 'Windows', suite = 'Office' pattern = /ice/i
```

2. Добавьте операторы для вывода результатов поиска.

```
console.log( 'In ' + system +'? ' + pattern.test( system ) )
console.log(' In ' + suite + '? ' + pattern.test( suite ) )
```

3. Добавьте операторы для вывода совпадающего текста и позиции. В противном случае — сообщение о том, что совпадения не найдены.

```
let result = pattern.exec( suite )
if( result )
{
   console.log( 'Found ' +  result + ' at ' + result.index )
}
else { console.log( 'No Match Found' ) }
```

Используемое в этом примере регулярное выражение проверяет только самые основные требования к формату электронной почты.

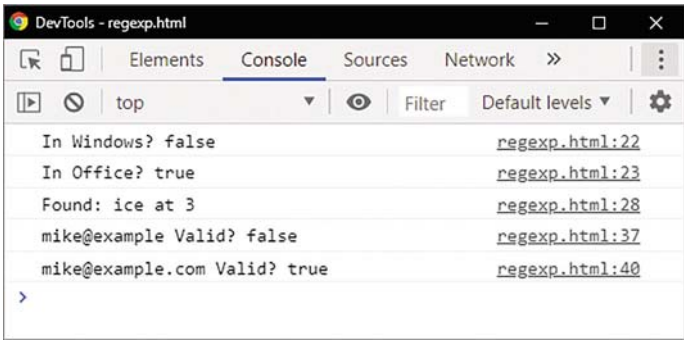
4 Добавьте операторы для вывода результата проверки некорректного адреса электронной почты.

```
let email = 'mike@example'
const format = /.+\@.+\..+/
console.log( email + ' Valid? ' + format.test( email ) )
```

5 Добавьте операторы для исправления адреса и вывода результата проверки.

```
email += '.com'
console.log( email + ' Valid? ' + format.test( email ) )
```

6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — регулярные выражения.



Нумерация индексов в строке начинается с нуля, поэтому четвертый символ находится в позиции индекса 3.

Заключение

- Значения присваиваются объекту в виде разделенного запятыми списка **name: value** (имя: значение), заключенного в фигурные скобки **{}**.
- На значения свойств объекта можно ссылаться с использованием точечной нотации, или указав их в квадратных скобках **[]**.
- Методы объекта вызываются с помощью добавления после имени круглых скобок **( )**.
- Ключевое слово **this** относится к объекту, к которому оно принадлежит.
- Пользовательские объекты расширяются путем присвоения нового значения с использованием точечной записи для ссылки на свойство.
- Пользовательские объекты могут быть в любое время легко расширены и изменены. Для этого нужно лишь назначить новое значение с помощью точечной нотации.
- Цикл **for in** используется для перечисления всех свойств и методов объекта.
- Все объекты в JavaScript наследуют свойства и методы из прототипа объекта **Object.prototype**.
- Объект **Array** — это встроенный объект JavaScript, который используется для хранения упорядоченных коллекций данных (с различными типами данных). Элементы массива нумеруются автоматически, начиная с нуля.
- Объекты массивов создаются путем присвоения переменным литеральных значений, указанных в квадратных скобках **[]** и разделенных запятыми.
- Доступ к элементам массива осуществляется с помощью оператора **[]**.

- Чтобы узнать длину массива, необходимо обратиться к его свойству **length**.

- Встроенный объект **Date** представляет конкретную дату, время и часовой пояс.

- Конструктор **Date( )** может дополнительно указывать от двух до семи аргументов для установки значений для каждого из его компонентов.

- Объект **Date** также предоставляет отдельные методы для установки значений даты и времени.

- Объект **RegExp** — это встроенный объект JavaScript, который создает объект регулярного выражения для сопоставления текста с шаблоном.

- Методы **test( )** и **exec( )** выполняют поиск сопоставления регулярного выражения в указанной строке.