内置核心对象

本章描述在核心 JavaScript 中预定义的一些对象: `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`.

Array Object

JavaScript其实并没有精确的数组数据类型，但你可以用 `Array` 对象及其方法在你的程序中进行数组编程。`Array` 对象拥有操作数组的各种方法，例如相加、反转、排序等。它有一个属性以确定数组的长度，还有一些用于正则表达式(?)。

数组是一个通过名称和索引来引用的有序的值的集合。例如，可以有一个以员工为索引的员工名称的数组 emp。`emp[1]` 将是第一号员工，`emp[2]` 就是第二号员工，依此类推。

创建数组

以下三种创建数组的方式是等价的:

```
var arr = new Array(element0, element1, ..., elementN);
var arr = Array(element0, element1, ..., elementN);
var arr = [element0, element1, ..., elementN];
```

`element0, element1, ..., elementN` 是一系列数组元素的值。在此形式下，数组通过给定的元素值初始化，且数组的 `length` 属性被设为参数的个数。

方括号语法被称作“数组字面量”或“数组初始化器”。该形式比较简洁因而通常受到推荐。参见 Array Literals 以获取数组字面量的更多信息。

若要创建一长度不为零但却不包含任何元素的数组，可以使用下列方式之一:

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);
var arr = [];
arr.length = arrayLength;
```

注意：在上面的代码中，`arrayLength` 必须是一个数值。否则将创建具有单一元素（为给定值）的数组。调用 `arr.length` 将返回 `arrayLength`, 但数组实际上包含空元素（`undefined`）。对数组使用 `for...in` 循环并不返回任何数组元素。

除了象上面那样将数组赋给新变量，它也可以被赋给新对象或现有对象的属性。

```
var obj = {};obj.prop = [element0, element1, ..., elementN];
var obj = {prop: [element0, element1, ...., elementN]}
```

如果你需要将数组初始化包含唯一元素，并且这相元素恰好是单个数值，你必须使用方括号语法。当单个数值被传给构造函数或函数（取决于是否使用了 new 关键字） Array() 时，它被解释为 arrayLength，而不是唯一元素。

```
var arr = [42];
var arr = Array(42); // Creates an array with no element, but with
arr.length set to 42

// The above code is equivalent to
var arr = [];
arr.length = 42;
```

对于 Array(N) 调用，若 N 是一个小数部分不为零的数值，则会得到 RangeError。下面的例子说明了这个行为:

```
var arr = Array(9.3);  // RangeError: Invalid array length
```

填充数组

你可以通过给元素赋值来填充数组。例如:

```
var emp = [];
emp[0] = "Casey Jones";
emp[1] = "Phil Lesh";
emp[2] = "August West";
```

注意: 如果你在上面的代码中为数组操作符传递了一个非整型的值，则将为代表数组的对象创建一个属性，而不是一个数组元素:

```
var arr = [];
arr[3.4] = "Oranges";
console.log(arr.length);                    // 0
console.log(arr.hasOwnProperty[3.4]);   // true
```

你可以在创建一个数组的同时填充它:

```
var myArray = new Array("Hello", myVar, 3.14159);
var myArray = ["Mango", "Apple", "Orange"]
```

引用数组元素

你可以通过元素的序号引用数组中的元素。举例而言，假如你定义了如下数组:

```
var myArray = ["Wind", "Rain", "Fire"];
```

接下来你可以通过 myArray[0] 引用第一个元素，通过 myArray[1] 引用第二个元素。元素的索引是从0开始的。

注意: 数组操作符（方括号）也用于访问数组的属性（在 JavaScript 中数组也是一种对象）。例如：

```
var arr = ["one", "two", "three"];
arr[2];  // three
arr["length"];  // 3
```

理解长度

从实现角度，JavaScript 的数组将其元素作为普通的对象数组存储，并使用数组索引作为属性名。length 属性有些特别，它总是返回最后一个元素的索引再加上1。记着，JavaScript 的数组索引是从0而不1开始的。这意味着length属性总是比数组中最大的索引还要大1:

```
var cats = [];
cats[30] = ['Dusty'];
print(cats.length);
```

你还可以给 length 属性赋值。写入一个比现有元素数小的值将截断数组，写入0将把数组全部清空：

```
var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length);
cats.length = 2;
console.log(cats);
cats.length = 0;
console.log(cats);
cats.length = 3;
console.log(cats);
```

遍历数组

一个常见的操作是遍历数组的元素，并对每个元素进行处理。做这事最简单的方式如下：

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

如果你已知数组中的任何元素在条件上下文中都不会被计算成 false——例如数组包含 DOM 元素——你可以使用一种更高效的惯用法：

```
var divs = document.getElementsByTagName('div');
```

```
for (var i = 0, div; div = divs[i]; i++) {


}
```

这避免了检查数组长度带来的开销，并且在每一次循环中更便捷地确保将当前项重新赋给变量 div 。

在 JavaScript 1.6 中引入的 `forEach()` 方法，提供了另一种遍历数组的方法：

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

传递给 `forEach` 的函数将针对每个元素执行一次，并且数组元素将作为参数传递给函数。未赋值的元素将不会在 `forEach` 循环中被遍历。

注意 forEach 的清单中不包括在数组定义时被省略的项目，但会包括被显式赋值为 undefined 的项目：

```
var array = ['first', 'second', , 'fourth'];
array.forEach(function(element) {
  console.log(element);
})

if(array[2] === undefined) { console.log('array[2] is undefined'); }
var array = ['first', 'second', undefined, 'fourth'];
array.forEach(function(element) {
  console.log(element);
})
```

因为 JavaScript 元素被作为标准的对象属性存储，所以不建议对数组使用 for...in 循环进行遍历，它导致普通的元素和所有可枚举的属性都会出现在清单中。

数组的方法

`Array` 对象有如下方法：

- `concat()` 将两个数组连接成一个新数组。

  ```
  var myArray = new Array("1", "2", "3");
  myArray = myArray.concat("a", "b", "c"); // myArray is now ["1", "2",
  "3", "a", "b", "c"]
  ```

- `join(deliminator = ",")` 将数组的所有元素连接成一个字符串。

```
var myArray = new Array("Wind", "Rain", "Fire");
var list = myArray.join(" - "); // list is "Wind - Rain - Fire"
```

- `push()` 在数组的最后增加一个元素并且返回数组的新长度。

```
var myArray = new Array("1", "2");
myArray.push("3"); // myArray is now ["1", "2", "3"]
```

- `pop()` 从数组中删除最后一个元素并且返回该元素。

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop(); // myArray is now ["1", "2"], last = "3"
```

- `shift()` 从数组中删除第一个元素并且返回该元素。

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift(); // myArray is now ["2", "3"], first is "1"
```

- `unshift()` 在数组开头增加一个或多个元素并且返回数组的新长度。

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5"); // myArray becomes ["4", "5", "1", "2", "3"]
```

- `slice(start_index, upto_index)` 抽取数组的一个片断并将其作为新数组返回。

```
var myArray = new Array ("a", "b", "c", "d", "e");
myArray = myArray.slice(1, 4); /* starts at index 1 and extracts all
elements
  until index 3, returning [ "b", "c", "d"] */
```

- `splice(index, count_to_remove, addelement1, addelement2, ...)` 从数组中删除元素并且（可选地）替换它们。

```
var myArray = new Array ("1", "2", "3", "4", "5");
myArray.splice(1, 3, "a", "b", "c", "d"); // myArray is now ["1", "a",
"b", "c", "d", "5"]
  // This code started at index one (or where the "2" was), removed 3
elements there,
  // and then inserted all consecutive elements in its place.
```

- `reverse()` 将数组元素进行倒以：第一个的数组元素将变为最后一个，而最后的元素将变为第一个。

```
var myArray = new Array ("1", "2", "3");
```

```
myArray.reverse(); // transposes the array so that myArray = [ "3",
"2", "1" ]
```

- `sort()` 对数组元素进行排序。

  ```
  var myArray = new Array("Wind", "Rain", "Fire");
  myArray.sort(); // sorts the array so that myArrray = [ "Fire", "Rain",
  "Wind" ]
  ```

  `sort()` 也可以接收一个函数用于判定元素的比较结果。该函数对两个值进行比较并且返回以下三个值之一:

  - 如果在排序方式中 a 小于 b，则返回 -1 (或任何负数)
  - 如果在排序方式中 a 大于 b，则返回 1 (或任意正数)
  - 如果 a 和 b 被认为相等，则返回 0。

  例如，下面的代码按数组中最后一个字符进行排序:

  ```
  var sortFn = function(a, b){
    if (a[a.length - 1] < b[b.length - 1]) return -1;
    if (a[a.length - 1] > b[b.length - 1]) return 1;
    if (a[a.length - 1] == b[b.length - 1]) return 0;
  }
  myArray.sort(sortFn); // sorts the array so that myArray =
  ["Wind","Fire","Rain"]
  ```

这些函数针对旧浏览器中的兼容代码可以在其单独的页面上找到。不同浏览器针对这些功能的原生支持列于 此处.

- `indexOf(searchElement[, fromIndex])` searches the array for `searchElement` and returns the index of the first match.

  ```
  var a = ['a', 'b', 'a', 'b', 'a'];
  alert(a.indexOf('b')); // Alerts 1
  // Now try again, starting from after the last match
  alert(a.indexOf('b', 2)); // Alerts 3
  alert(a.indexOf('z')); // Alerts -1, because 'z' was not found
  ```

- `lastIndexOf(searchElement[, fromIndex])` works like `indexOf`, but starts at the end and searches backwards.

  ```
  var a = ['a', 'b', 'c', 'd', 'a', 'b'];
  alert(a.lastIndexOf('b')); // Alerts 5
  ```

```
// Now try again, starting from before the last match
alert(a.lastIndexOf('b', 4)); // Alerts 1
alert(a.lastIndexOf('z')); // Alerts -1
```

- `forEach(callback[, thisObject])` execute `callback` on every array item.

```
var a = ['a', 'b', 'c'];
a.forEach(alert); // Alerts each item in turn
```

- `map(callback[, thisObject])` returns a new array of the return value from executing `callback` on every array item.

```
var a1 = ['a', 'b', 'c'];
var a2 = a1.map(function(item) { return item.toUpperCase(); });
alert(a2); // Alerts A,B,C
```

- `filter(callback[, thisObject])` returns a new array containing the items for which callback returned true.

```
var a1 = ['a', 10, 'b', 20, 'c', 30];
var a2 = a1.filter(function(item) { return typeof item == 'number'; });
alert(a2); // Alerts 10,20,30
```

- `every(callback[, thisObject])` returns true if `callback` returns true for every item in the array.

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
alert(a1.every(isNumber)); // Alerts true
var a2 = [1, '2', 3];
alert(a2.every(isNumber)); // Alerts false
```

- `some(callback[, thisObject])` returns true if `callback` returns true for at least one item in the array.

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
alert(a1.some(isNumber)); // Alerts true
```

```
var a2 = [1, '2', 3];
alert(a2.some(isNumber)); // Alerts true
var a3 = ['1', '2', '3'];
alert(a3.some(isNumber)); // Alerts false
```

The methods above that take a callback are known as *iterative methods*, because they iterate over the entire array in some fashion. Each one takes an optional second argument called `thisObject`. If provided, `thisObject` becomes the value of the `this` keyword inside the body of the callback function. If not provided, as with other cases where a function is invoked outside of an explicit object context, `this` will refer to the global object (`window`).

The callback function is actually called with three arguments. The first is the value of the current item, the second is its array index and the third is a reference to the array itself. JavaScript functions ignore any arguments that are not named in the parameter list so it is safe to provide a callback function that only takes a single argument, such as `alert`.

- `reduce(callback[, initialValue])` applies `callback(firstValue, secondValue)` to reduce the list of items down to a single value.

  ```
  var a = [10, 20, 30];
  var total = a.reduce(function(first, second) { return first + second;
  }, 0);
  alert(total) // Alerts 60
  ```

- `reduceRight(callback[, initialValue])` works like `reduce()`, but starts with the last element.

`reduce` and `reduceRight` are the least obvious of the iterative array methods. They should be used for algorithms that combine two values recursively in order to reduce a sequence down to a single value.

Multi-Dimensional Arrays

Arrays can be nested, meaning that an array can contain another array as an element. Using this characteristic of JavaScript arrays, multi-dimensional arrays can be created.

The following code creates a two-dimensional array.

```
var a = new Array(4);
for (i = 0; i < 4; i++) {
  a[i] = new Array(4);
  for (j = 0; j < 4; j++) {
```

```
    a[i][j] = "[" + i + "," + j + "]";
  }
}
```

This example creates an array with the following rows:

```
Row 0: [0,0] [0,1] [0,2] [0,3]
Row 1: [1,0] [1,1] [1,2] [1,3]
Row 2: [2,0] [2,1] [2,2] [2,3]
Row 3: [3,0] [3,1] [3,2] [3,3]
```

Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `RegExp.exec()`, `String.match()`, and `String.split()`. For information on using arrays with regular expressions, see Regular Expressions.

Working with Array-like objects

有一些 JavaScript 对象, 比如使用 `document.getElementByTagName` 返回的 `NodeList` 或者函数内的 `arguments`, 表现的很像数组但却不能共享所有的数组方法. 比如 `arguments` 对象有一个 `length` 属性但不能调用 `forEach` 方法.

数组泛化(Array generics, introduced in JavaScript 1.6), 提供了一种针对类数组对象运行 Array (实例)方法的途径. 每一个标准 Array (实例)方法在 Array 对象本身上有一个对应的泛化方法; 比如:

```
function alertArguments() {
   Array.forEach(arguments, function(item) {
     alert(item);
   });
 }
```

在旧版本的 JavaScript 中, 可以使用函数对象的 `call` 方法模拟这些泛化方法:

```
Array.prototype.forEach.call(arguments, function(item) {
   alert(item);
 });
```

数组泛化方法同样可以被使用在字符串上, 因为它们提供了顺序访问它们的字符的方式，而这和数组很类似:

```
Array.forEach("a string", function(chr) {
```

```
   alert(chr);
});
```

下面列举了更多将数组方法应用到字符串上的例子, also taking advantage of JavaScript 1.8 expression closures:

```
var str = 'abcdef';
var consonantsOnlyStr = Array.filter(str, function (c) !
(/[aeiou]/i).test(c)).join(''); // 'bcdf'
var vowelsPresent = Array.some(str, function (c) (/[aeiou]/i).test(c)); //
true
var allVowels = Array.every(str, function (c) (/[aeiou]/i).test(c)); //
false
var interpolatedZeros = Array.map(str, function (c) c+'0').join(''); //
'a0b0c0d0e0f0'
var numerologicalValue = Array.reduce(str, function (c, c2)
c+c2.toLowerCase().charCodeAt()-96, 0);
// 21 (reduce() since JS v1.8)
```

Note that `filter` and `map` do not automatically return the characters back into being members of a string in the return result; an array is returned, so we must use `join` to return back to a string.

Array comprehensions

Introduced in JavaScript 1.7, array comprehensions provide a useful shortcut for constructing a new array based on the contents of another. Comprehensions can often be used in place of calls to `map()` and `filter()`, or as a way of combining the two.

The following comprehension takes an array of numbers and creates a new array of the double of each of those numbers.

```
var numbers = [1, 2, 3, 4];
var doubled = [i * 2 for (i of numbers)];
alert(doubled); // Alerts 2,4,6,8
```

This is equivalent to the following `map()` operation:

```
var doubled = numbers.map(function(i){return i * 2;});
```

Comprehensions can also be used to select items that match a particular expression. Here is a comprehension which selects only even numbers:

```
var numbers = [1, 2, 3, 21, 22, 30];
```

```
var evens = [i for (i of numbers) if (i % 2 === 0)];
alert(evens); // Alerts 2,22,30
```

`filter()` can be used for the same purpose:

```
var evens = numbers.filter(function(i){return i % 2 === 0;});
```

`map()` and `filter()` style operations can be combined into a single array comprehension. Here is one that filters just the even numbers, then creates an array containing their doubles:

```
var numbers = [1, 2, 3, 21, 22, 30];
var doubledEvens = [i * 2 for (i of numbers) if (i % 2 === 0)];
alert(doubledEvens); // Alerts 4,44,60
```

The square brackets of an array comprehension introduce an implicit block for scoping purposes. New variables (such as i in the example) are treated as if they had been declared using `let`. This means that they will not be available outside of the comprehension.

The input to an array comprehension does not itself need to be an array; iterators and generators can also be used.

Even strings may be used as input; to achieve the filter and map actions (under Array-like objects) above:

```
var str = 'abcdef';
var consonantsOnlyStr = [c for (c of str) if (!(/[aeiouAEIOU]/).test(c))
].join(''); // 'bcdf'
var interpolatedZeros = [c+'0' for (c of str) ].join(''); // 'a0b0c0d0e0f0'
```

Again, the input form is not preserved, so we have to use `join()` to revert back to a string.

Boolean Object

The `Boolean` object is a wrapper around the primitive Boolean data type. Use the following syntax to create a `Boolean` object:

```
var booleanObjectName = new Boolean(value);
```

Do not confuse the primitive Boolean values `true` and `false` with the true and false values of the `Boolean` object. Any object whose value is not `undefined` , `null`, `0`, `NaN`, or the empty string , including a `Boolean` object whose value is false, evaluates to true when passed to a conditional statement. See if...else Statement for more information.

Date Object

JavaScript does not have a date data type. However, you can use the `Date` object and its methods to work with dates and times in your applications. The `Date` object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

The `Date` object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a `Date` object:

```
var dateObjectName = new Date([parameters]);
```

where `dateObjectName` is the name of the `Date` object being created; it can be a new object or a property of an existing object.

Calling Date without the new keyword simply converts the provided date to a string representation.

The `parameters` in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date();`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `var Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `var Xmas95 = new Date(1995, 11, 25)`.
- A set of integer values for year, month, day, hour, minute, and seconds. For example, `var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);`.

JavaScript 1.2 and earlier

The `Date` object behaves as follows:

- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the `Date` object varies from platform to platform.

Methods of the Date Object

The `Date` object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in `Date` objects.
- "get" methods, for getting date and time values from `Date` objects.
- "to" methods, for returning string values from `Date` objects.

- parse and UTC methods, for parsing `Date` strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a `getDay` method that returns the day of the week, but no corresponding `setDay` method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 (Sunday) to 6 (Saturday)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
var Xmas95 = new Date("December 25, 1995");
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getFullYear()` returns 1995.

The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
var today = new Date();
var endYear = new Date(1995, 11, 31, 23, 59, 59, 999); // Set day and month
endYear.setFullYear(today.getFullYear()); // Set year to this year
var msPerDay = 24 * 60 * 60 * 1000; // Number of milliseconds per day
var daysLeft = (endYear.getTime() - today.getTime()) / msPerDay;
var daysLeft = Math.round(daysLeft); //returns days left in the year
```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between `today` and `endYear`, using `getTime` and rounding to a whole number of days.

The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
var IPOdate = new Date();
IPOdate.setTime(Date.parse("Aug 9, 1995"));
```

Using the Date Object: an Example

In the following example, the function `JSClock()` returns the time in the format of a digital clock.

```
function JSClock() {
  var time = new Date();
  var hour = time.getHours();
  var minute = time.getMinutes();
  var second = time.getSeconds();
  var temp = "" + ((hour > 12) ? hour - 12 : hour);
  if (hour == 0)
    temp = "12";
  temp += ((minute < 10) ? ":0" : ":") + minute;
  temp += ((second < 10) ? ":0" : ":") + second;
  temp += (hour >= 12) ? " P.M." : " A.M.";
  return temp;
}
```

The `JSClock` function first creates a new `Date` object called `time`; since no arguments are given, time is created with the current date and time. Then calls to the `getHours`, `getMinutes`, and `getSeconds` methods assign the value of the current hour, minute and seconds to `hour`, `minute`, and `second`.

The next four statements build a string value based on the time. The first statement creates a variable `temp`, assigning it a value using a conditional expression; if `hour` is greater than 12, (`hour - 12`), otherwise simply hour, unless hour is 0, in which case it becomes 12.

The next statement appends a `minute` value to `temp`. If the value of `minute` is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to `temp` in the same way.

Finally, a conditional expression appends "PM" to `temp` if `hour` is 12 or greater; otherwise, it appends "AM" to `temp`.

Function Object

The predefined `Function` object specifies a string of JavaScript code to be compiled as a function.

To create a `Function` object:

```
var functionObjectName = new Function ([arg1, arg2, ... argn],
functionBody);
```

`functionObjectName` is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as `window.onerror`.

`arg1`, `arg2`, ... `argn` are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm".

`functionBody` is a string specifying the JavaScript code to be compiled as the function body.

`Function` objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the `function` statement and the function expression. See the JavaScript Reference for more information.

The following code assigns a function to the variable `setBGColor`. This function sets the current document's background color.

```
var setBGColor = new Function("document.bgColor = 'antiquewhite'");
```

To call the `Function` object, you can specify the variable name as if it were a function. The following code executes the function specified by the `setBGColor` variable:

```
var colorChoice="antiquewhite";
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

1. `document.form1.colorButton.onclick = setBGColor;`

2. ```
   <INPUT NAME="colorButton" TYPE="button"
       VALUE="Change background color"
       onClick="setBGColor()">
   ```

Creating the variable `setBGColor` shown above is similar to declaring the following function:

```
function setBGColor() {
  document.bgColor = 'antiquewhite';
}
```

Assigning a function to a variable is similar to declaring a function, but there are differences:

- When you assign a function to a variable using `var setBGColor = new Function("...")`, `setBGColor` is a variable for which the current value is a reference to the function created with `new Function()`.
```

- When you create a function using `function setBGColor() {...}`, `setBGColor` is not a variable, it is the name of a function.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

Math Object

The predefined `Math` object has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of `Math`. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note that all trigonometric methods of `Math` take arguments in radians.

The following table summarizes the `Math` object's methods.

Table 7.1 Methods of Math

Method	Description
`abs`	Absolute value
`sin`, `cos`, `tan`	Standard trigonometric functions; argument in radians
`acos`, `asin`, `atan`, `atan2`	Inverse trigonometric functions; return values in radians
`exp`, `log`	Exponential and natural logarithm, base `e`
`ceil`	Returns least integer greater than or equal to argument
`floor`	Returns greatest integer less than or equal to argument
`min`, `max`	Returns greater or lesser (respectively) of two arguments
`pow`	Exponential; first argument is base, second is exponent
`random`	Returns a random number between 0 and 1.
`round`	Rounds argument to nearest integer
`sqrt`	Square root

Unlike many other objects, you never create a `Math` object of your own. You always use the predefined `Math` object.

Number Object

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself.

The following table summarizes the `Number` object's properties.

Table 7.2 Properties of Number

Property	Description
`MAX_VALUE`	The largest representable number
`MIN_VALUE`	The smallest representable number
`NaN`	Special "not a number" value
`NEGATIVE_INFINITY`	Special negative infinite value; returned on overflow
`POSITIVE_INFINITY`	Special positive infinite value; returned on overflow

The Number prototype provides methods for retrieving information from Number objects in various formats. The following table summarizes the methods of `Number.prototype`.

Table 7.3 Methods of Number.prototype

Method	Description
`toExponential`	Returns a string representing the number in exponential notation.
`toFixed`	Returns a string representing the number in fixed-point notation.
`toPrecision`	Returns a string representing the number to a specified precision in fixed-point notation.
`toSource`	Returns an object literal representing the specified `Number` object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of the specified object. Overrides the `Object.valueOf` method.

RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in Regular Expressions.

String Object

The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
var s1 = "foo"; //creates a string literal value
var s2 = new String("foo"); //creates a String object
```

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal.

You should use string literals unless you specifically need to use a `String` object, because `String` objects can have counterintuitive behavior. For example:

```
var s1 = "2 + 2"; //creates a string literal value
var s2 = new String("2 + 2"); //creates a String object
eval(s1); //returns the number 4
eval(s2); //returns the string "2 + 2"
```

A `String` object has one property, `length`, that indicates the number of characters in the string. For example, the following code assigns `x` the value 13, because "Hello, World!" has 13 characters:

```
var mystring = "Hello, World!";
var x = mystring.length;
```

A `String` object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and `"hello, world!".toUpperCase()` return the string "HELLO, WORLD!"

The `substring` method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, `mystring.substring(4, 9)` returns the string "o, Wo". See the `substring` method of the `String` object in the JavaScript Reference for more information.

The `String` object also has a number of methods for automatic HTML formatting, such as `bold` to create boldface text and `link` to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the `link` method as follows:

```
mystring.link("http://www.helloworld.com")
```

The following table summarizes the methods of `String` objects.

Table 7.4 Methods of String Instances

Method	Description
`anchor`	Creates HTML named anchor.
`big`, `blink`, `bold`, `fixed`, `italics`, `small`, `strike`, `sub`, `sup`	Create HTML formatted string.
`charAt`, `charCodeAt`	Return the character or character code at the specified position in string.
`indexOf`, `lastIndexOf`	Return the position of specified substring in the string or last position of specified substring, respectively.
`link`	Creates HTML hyperlink.
`concat`	Combines the text of two strings and returns a new string.
`fromCharCode`	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
`split`	Splits a `String` object into an array of strings by separating the string into substrings.
`slice`	Extracts a section of an string and returns a new string.
`substring`, `substr`	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
`match`, `replace`, `search`	Work with regular expressions.
`toLowerCase`, `toUpperCase`	Return the string in all lowercase or all uppercase, respectively.