Interactive Web Development

Grammar

Dr Russ Ross

Dixie State University—Computer and Information Technologies

Spring 2016
Grammar

Reading: *JavaScript: The Good Parts*, Chapter 2
Whitespace is usually ignored, but sometimes matters:

```javascript
var that = this;
```

Here whitespace is necessary between `var` and `that`, but nowhere else.

You should use whitespace to make your code easy to read and follow. Proper indentation is not required by the language, but is essential for human reasons.
Comments

Comments are treated like whitespace. JavaScript understands two kinds of comments:

- /* block style, possibly over multiple lines */
- // line style, everything to the end of the current line

The block style is useful for commenting out blocks of code, but it can be confused by regular expressions:

```javascript
/*
  var rm_a = /a*/.match(s);
*/
```

Best to avoid using the block style and stick to the single line form.
A name is a letter optionally followed by one or more letters, digits, or underscores. A name cannot be one of the reserved words (many of which are not actually used by the language, but are still reserved):

```
abstract
boolean break byte
case catch char class const continue
destructor default delete do double
default else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```
Numbers

JavaScript has a single number type.

- Internally, it is stored as a 64-bit float (just like double in C or Java)
- There is no separate integer type, so 1 and 1.0 are the same value
- Number literals can include exponents, e.g. 1.5e9, and negative numbers are formed by putting – in front, e.g., –15
- The special value NaN is the result of an operation that cannot produce a normal number. NaN is not equal to anything, including itself. You can detect it using isNaN(n)
- The value Infinity represents values greater than 1.79769313486231570e+308 (and can be negative)
- Numbers have methods. In addition, the Math object has methods that act on numbers, e.g. Math.floor(n).
Strings

- literals can be wrapped in (matching) single quotes or double quotes
- can contain zero or more characters
- backslash (\) is an escape character
- all characters are 16 bits
- there is no character type; instead use a string with one character in it
- unicode characters can be specified using the \u escape sequence, e.g., "A" === "\u0041"

I recommend using single quoted strings by default, because HTML calls for double quotes around attribute values. Use double quotes for strings containing single quotes if it is not confusing.
Strings (continued):

- Strings have a length property, e.g., "seven".length === 5
- Strings are immutable. Once created, a string can never be changed. You can easily create a new string, however, and store it where the old string was, for example with the + operator.
- Two strings containing exactly the same characters in the same order are considered to be the same string, e.g., 'c' + 'a' + 't' === 'cat'
- Strings have methods, e.g., 'cat'.toUpperCase() === 'CAT'
Compilation units

Most programming languages are organized around *compilation units*, typically one per file. In JavaScript, each `<script>` tag introduces a new compilation unit, either included inline or sourced from another file.

When a new compilation unit is encountered, it is immediately compiled and executed.

All code is executed in a single, global namespace.

The `var` statement—when used inside a function—creates a local variable, private to that function.
Statements

Statements are normally executed in order from top to bottom. This flow is broken by:

- **Conditional statements**: `if` and `switch`
- **Looping statements**: `while`, `for`, and `do`
- **Disruptive statements**: `break`, `return`, and `throw`
- **Function invocation**

A *block* is a set of statements wrapped in curly braces. In JavaScript, blocks do **not create a new scope**. To avoid confusion, variables should be defined at the top of the function, not in blocks.
The **if** statement has this form:

```java
if (<expression>) {
  // then block
} else {
  // else block
}
```

The **else** block is optional. For both **then** and **else**, a single statement without curly braces can take the place of the block. This statement can be another **if** statement:

```java
if (<expression>) {
  // then block
} else if (<expression>) {
  // then block
} else {
  // else block
}
```
Truthiness

The if statement evaluates an expression and selects the then block if the expression is truthy. Otherwise, the optional else branch is taken.

A value is truthy if it is not falsy. The falsy values are:

- false
- null
- undefined
- The empty string ''
- The number 0
- The number NaN

Everything else is truthy, including true, 'false', and all objects.
The **switch** statement performs a multiway branch. It looks like this:

```java
switch (<expression>) {
    case 'String':
        // do something
        break;
    case 45:
        // do something else
        break;
    case 12:
        // fallthrough
    case 'Twe' + 'lve':
        // for either 12 or 'Twelve'
        break;
    default:
        // optional default clause
}
```

Note that the **case** expressions need not be constants. Without **break** (or another disruptive statement), execution falls through to the next **case** clause. **default** is optional.
while

while statements look like this:

```java
while (<expression>) {
    // block
}
```

As with `if`, the block can be replaced with a single statement and no curly braces.

`while` performs a simple loop. If the expression is false, the loop breaks. While it is truthy, the block is executed.
The `for` statement has two forms. First the conventional form:

```java
for (<expr statement>; <expression>; <expr statement>) {
    // block
}
```

The first expression statement is executed before the loop begins. The second is executed after each iteration. The expression is evaluated before each iteration, similar to a `while` loop. If the expression is omitted, `true` is assumed.

An `expression statement` can either assign values to one or more variables or members, invoke a method, or delete a property from an object:

```java
for (i = 0; i < 15; i += 3) {
    ... 
}
```

The `=` operator is used for assignment. `+=` can add or concatenate.
The second form enumerates the property names (or keys) of an object. It is called **for in**:

```javascript
for (myvar in obj) {
    // block
}
```

It is usually wise to test if the property name is actually a part of the object, or if it was found through the prototype chain (part of JavaScript’s object system):

```javascript
for (myvar in obj) {
    if (obj.hasOwnProperty(myvar)) {
        // ...
    }
}
```
The **do** statement is similar to **while**:

```java
do {
    // block
} while (<expression>);
```

In a **do** statement, the expression is evaluated **after** each iteration. As a result, the block will always be executed at least once.
The `try` statement executes a block of code and catches any exceptions that were thrown by the block:

```java
try {
    // block
} catch (myvar) {
    // block
}
```

The `catch` clause defines a new variable that will receive the exception object.
The `throw` statement raises an exception:

```java
if (somethingexceptional) {
    throw <expression>;
}
```

If the `throw` statement is inside a `try` block, control passes to the `catch` clause. The expression is evaluated, and its value is assigned to the variable in the `catch` clause.

If the `throw` statement is not inside a `try` block, the function execution is abandoned and control is passed to the `catch` clause of the `try` in the calling function.
The `return` statement causes the early return from a function. It can optionally specify the value to be returned. If it is not specified, the return value is `undefined`:

```java
if (<expression>) {
    return <expression>;
}

return;
```
The `break` statement causes the exit from a loop or `switch` statement:

```python
if (timetoquitloop)
    break;
```

`switch`, `while`, `for`, and `do` statements are allowed to have an optional `label` prefix. The `break` statement can have an optional label that specifies the loop or `switch` that it applies to:

```python
OUTERLOOP: while (<expression>) {
    while (<expression>) {
        if (timetoquitouterloop)
            break OUTERLOOP;
    }
}
```

A line break is **not** permitted between the `break` and the label.
Expressions

An expression is a chunk of code that produces a value:

- A literal value (string, number, etc.)
- A variable or built-in value (true, false, null, undefined, NaN, or Infinity)
- new followed by an invocation expressions
- delete followed by a refinement expression (one that selects something, such as a field in an object)
- Another expression wrapped in parentheses
- Another expression preceded by a prefix operator
Expressions are often composed of an expression followed by:

- An infix operator (+, −, *, etc.) followed by another expression
- The `?` ternary operator followed by `<expr> : <expr>`
- An invocation (calling a function)
- A refinement (selecting an object field, etc.)
Operator precedence

In an expression like $3 + 7 * 9$, operator precedence dictates which operators are applied first:

- `[]` (`()`
- `delete new typeof + - !`
- `* / %`
- `+ -`
- `>= <= > <`
- `=== !==`
- `&&` and
- `||`
- `?:`

Refinement and invocation
Unary operators
Multiplication, division, modulo
Addition/concatenation, subtraction
Inequality
Equality
Logical and
Logical or
Ternary

Operators in the list have higher precedence than those below them.
Operator notes:

- `typeof` produces the values 'number', 'string', 'boolean', 'undefined', 'function', and 'object'. If the operand is an array or `null`, the result is 'object', which is wrong.
- `!` produces `false` if its operand is truthy, otherwise `true`.
- The `+` operator adds or concatenates. If you want it to add, make sure both operators are numbers.
- The `/` operator can produce a noninteger result even if both operands are integers.
- The `&&` operator produces the value of its first operand if the first operand is falsy. Otherwise, it produces the value of the second operand.
- The `||` operator produces the value of its first operand if the first operand is truthy. Otherwise, it produces the value of the second operand.
Object literals are a convenient notation for creating new objects. The names of the properties can be specified as names or strings. The values of the properties are expressions:

The names are treated as literal names, not variable names, so the property names must be known at compile time:

```javascript
var name = 'zebra';
var newobj = {
    name: 'value', // property is 'name', not 'zebra'
    'age': 24,
    in: 'trouble', // ILLEGAL: 'in' is reserved
};
```

Array literals allow the convenient creation of new arrays:

```javascript
var newarray = [ 1, 3, 4, 'strings', 'okay', 'too' ];
```

We will discuss both array and object literals in more detail later.
Functions

A function literal defines a function value:

```javascript
var f = function (a, b, c) { return a + b + c; }
```

It can have an optional name that it can use for recursion:

```javascript
var f = function fact(n) {
    if (n <= 1) return 1;
    else return n * fact(n - 1);
};
```

A function literal can be used immediately:

```javascript
var sum = (function (a, b) {
    return a + b;
})(5, 6);
```

Functions can be stored in variables, objects, and arrays, passed as arguments to other functions, and returned as values. We will have much more to say about functions later.